

Περιεχόμενα

Πρόλογος	11
----------------	----

ΚΕΦΑΛΑΙΟ 1

Εισαγωγή: Βασικές έννοιες και ιστορικό

<i>Σκοπός, Προσδοκώμενα Αποτελέσματα, Έννοιες κλειδιά, Δομή κεφαλαίου</i>	13
1.1 Βασικές έννοιες	15
1.2 Ταξινόμηση λειτουργικών συστημάτων	16
1.3 Ιστορική αναδρομή της εξέλιξης των λειτουργικών συστημάτων	18
1.3.1 Η μηδενική γενιά (1940)	18
1.3.2 Η πρώτη γενιά (1950 – 1960)	18
1.3.3 Η δεύτερη γενιά (1959 – 1965)	20
1.3.4 Η τρίτη γενιά (1965 – 1980)	20
1.3.5 Η τέταρτη γενιά (1980 – 1990)	21
1.3.6 Η πέμπτη γενιά (1990 – σήμερα)	22
<i>Σύνοψη</i>	23
<i>Βιβλιογραφία κεφαλαίου</i>	23
<i>Γλωσσάρι κεφαλαίου</i>	24

ΚΕΦΑΛΑΙΟ 2

Διαδικασίες

<i>Σκοπός, Προσδοκώμενα Αποτελέσματα, Έννοιες κλειδιά, Δομή κεφαλαίου</i>	25
2.1 Εισαγωγή	27
2.2 Η έννοια της διαδικασίας	27
2.3 Η αναπαράσταση μιας διαδικασίας	28
2.3.1 Καταστάσεις διαδικασίας	28
2.3.2 Το μπλοκ ελέγχου διαδικασιών 20	29
2.4 Λειτουργίες επί διαδικασιών	31

2.5	Η έννοια της διακοπής	33
2.5.1	Τύποι διακοπών	34
2.5.2	Χειρισμός διακοπών	35
2.6	Ο πυρήνας του λειτουργικού συστήματος	37
	<i>Σύνοψη</i>	38
	<i>Βιβλιογραφία κεφαλαίου</i>	39
	<i>Γλωσσάρι κεφαλαίου</i>	40

ΚΕΦΑΛΑΙΟ 3

Συντονισμός διαδικασιών

	<i>Σκοπός, Προσδοκώμενα Αποτελέσματα, Έννοιες κλειδιά, Δομή κεφαλαίου</i>	41
3.1	Ορισμοί	44
3.2	Εντολές παραλληλισμού	44
3.3	Η Ανάγκη συντονισμού	45
3.4	Κρίσιμες περιοχές	47
3.5	Η φύση του προβλήματος του αμοιβαίου αποκλεισμού	48
3.6	Γενικές παρατηρήσεις	56
3.7	Συνεργασία διαδικασιών (Επικοινωνία και Σημαφόροι)	57
3.7.1	Επικοινωνία διαδικασιών	57
3.7.2	Σημαφόροι (Semaphores)	58
3.7.3	Υλοποίηση κρίσιμων περιοχών με σημαφόρους	60
3.7.4	Κρίσιμες περιοχές υπό συνθήκη	65
3.7.5	Υλοποίηση γενικής σημαφόρου με δυαδική σημαφόρο	75
3.7.6	Υλοποίηση P και V μέσω hardware	76
3.7.7	Άλλες εντολές συντονισμού	79
3.8	Συντονισμός διαδικασιών που δεν συνεργάζονται	79
3.9	Κατανεμημένοι αλγόριθμοι για αμοιβαίο αποκλεισμό	80
3.9.1	Ο Αλγόριθμος του ζαχαροπλαστείου (The Bakery Algorithm)	81

Σύνοψη	83
Βιβλιογραφία κεφαλαίου	85
Γλωσσάρι κεφαλαίου	85

ΚΕΦΑΛΑΙΟ 4

Διαχείριση μνήμης (Memory Management)

Σκοπός, Προσδοκώμενα Αποτελέσματα, Έννοιες κλειδιά, Δομή κεφαλαίου	87
4.1 Εισαγωγικά (Preliminaries)	92
4.2 Σκέτη μηχανή (Bare machine)	96
4.3 Παραμένων επόπτης (Resident Monitor)	98
4.3.1 Υλικό προστασίας (Protection Hardware)	98
4.3.2 Μετατόπιση (Relocation)	100
4.4 Εναλλαγή (Swapping)	103
4.4.1 Επικουρική μνήμη (Backing Store)	103
4.4.2 Χρόνος εναλλαγής (Swap Time)	104
4.4.3 Επικαλυπτόμενη εναλλαγή (Overlapped Swapping)	105
4.5 Πολλαπλές υποδιαιρέσεις μνήμης (Multiple Partitions)	108
4.5.1 Υλικό προστασίας	109
4.5.2 Σταθερές περιοχές (Fixed Regions)	111
4.5.3 Τεμαχισμός μνήμης (Memory Fragmentation)	118
4.5.4 Μεταβλητές διαιρέσεις (Variable Partitions)	119
4.5.5 Συμπύεση (Compaction)	125
4.6 Σελιδοποίηση (Paging)	128
4.6.1 Υλικό (Hardware)	129
4.6.2 Χρονοδρομολόγηση διαδικασιών	132
4.6.3 Υλοποίηση του πίνακα σελίδων	135
4.6.4 Κοινές σελίδες	138
4.6.5 Προστασία	138

4.6.6 Δύο θεωρήσεις της μνήμης (Two Views of Memory)	141
4.7 Τμηματοποίηση (Segmentation)	143
4.7.1 Η θεώρηση της μνήμης από το χρήστη	143
4.7.2 Υλικό	144
4.7.3 Υλοποίηση των πινάκων τμημάτων (Implementation of Segment Tables)	147
4.7.4 Προστασία και κοινή χρήση	148
4.7.5 Κλασματοποίηση (χρονοδρομολόγηση διαδικασιών)	151
4.8 Συνδυασμένα συστήματα (Combined Systems)	152
4.8.1 Τμηματοποιημένη σελιδοποίηση (Segmented Paging)	152
4.8.2 Σελιδοποιημένη τμηματοποίηση (Paged Segmentation)	153
Σύνοψη	156
Βιβλιογραφία κεφαλαίου	158
Γλωσσάρι κεφαλαίου	159

ΚΕΦΑΛΑΙΟ 5

Ιδεατή μνήμη (Virtual Memory)

Σκοπός, Προσδοκώμενα Αποτελέσματα, Έννοιες κλειδιά, Δομή κεφαλαίου	161
5.1 Επικαλύψεις	165
5.2 Σελιδοποίηση ζήτησης	166
5.3 Απόδοση της σελιδοποίησης ζήτησης	173
5.4 Αντικατάσταση σελίδας	177
5.5 Έννοιες της ιδεατής μνήμης	180
5.6 Αλγόριθμοι αντικατάστασης σελίδας	182
5.6.1 FIFO	183
5.6.2 Βέλτιστη αντικατάσταση (Optimal Replacement)	186
5.6.3 LRU (Least Recently Used)	187
5.6.4 Προσέγγιση του LRU	189

5.6.5 Πρόσθετα bits αναφοράς	190
5.6.6 Αντικατάσταση δεύτερης ευκαιρίας	191
5.6.7 Ο Αλγόριθμος της λιγότερο συχνά χρησιμοποιούμενης σελίδας	191
5.6.8 Ο Αλγόριθμος της πιο συχνά χρησιμοποιούμενης σελίδας	192
5.6.9 Κλάσεις σελίδων	196
5.6.10 Ad Hoc αλγόριθμοι	197
5.7 Αλγόριθμοι κατανομής	198
5.7.1 Ελάχιστος αριθμός πλαισίων	199
5.7.2 Γενική έναντι τοπικής κατανομής	200
5.7.3 Αλγόριθμοι κατανομής	201
5.8 Λυγισμός (Thrashing)	203
5.8.1 Τοπικότητα	204
5.8.2 Μοντέλο συνόλου εργασίας (Working Set Model)	205
5.8.3 Συχνότητα σφαλμάτων σελίδας	208
5.9 Άλλοι παράγοντες	209
5.9.1 Πρόωρη σελιδοποίηση	209
5.9.2 Κλείδωμα σελίδων για I/O (I/O Interlock)	210
5.9.3 Μέγεθος σελίδας	212
5.9.4 Δομή διαδικασίας (προγράμματος)	214
5.9.5 Ιεραρχία αποθήκευσης	216
Σύνοψη	218
Βιβλιογραφία κεφαλαίου	220
Γλωσσάρι κεφαλαίου	221

Πρόλογος

Το βιβλίο αυτό εκπονήθηκε το ακαδημαϊκό έτος 1999 – 2000, σύμφωνα με τις οδηγίες και τα πρότυπα του Ελληνικού Ανοικτού Πανεπιστημίου, για τη Θεματική Ενότητα «Τεχνολογία Λογισμικού». Αποτελεί τον *πρώτο τόμο* του αντικειμένου των Λειτουργικών Συστημάτων και η φυσική συνέχειά του είναι το βιβλίο των προχωρημένων θεμάτων στα Λειτουργικά Συστήματα.

Το βιβλίο έχει γραφτεί με απλό τρόπο και η ύλη του έχει διδαχθεί κατ' επανάληψη στο Πανεπιστήμιο Πατρών, και ειδικότερα στο Τμήμα Μηχανικών Η/Υ & Πληροφορικής. Οι διδακτικές σημειώσεις του αντίστοιχου μαθήματος στο Πανεπιστήμιο Πατρών είναι και η πρώτη ύλη για το παρόν βιβλίο. Το βιβλίο απευθύνεται σε προπτυχιακούς σπουδαστές με ωριμότητα στη θετική σκέψη.

Θα ήθελα να ευχαριστήσω θερμά το συνάδελφο Καθηγητή Βασίλη Ταμπακά για τη συνεισφορά του στις σημειώσεις. Επίσης, την υποψήφια διδακτόρισσα Μαρία Ανδρέου για την επιμέλεια των ασκήσεων του βιβλίου. Τους/τις συναδέλφους (και διδάκτορες/ισσές μου) Χρήστο Μπούρα, Μαρίνα Παπατριανταφύλλου και Παναγιώτα Φατούρου για ουσιαστική συμβολή σε σχόλια και ύλη. Τον Αναπληρωτή Καθηγητή του Ελληνικού Ανοικτού Πανεπιστημίου Θανάση Χατζηλάκο για τα εκτενή σχόλια και τις παρατηρήσεις του.

Τέλος, θερμά ευχαριστήρια οφείλω στις κυρίες Ρ. Κουτσουβέλη και Ρ. Αργυρακοπούλου, που βοήθησαν τα μέγιστα στην (βασανιστική) εκτύπωση και διόρθωση του κειμένου.

Πάτρα, Ιούνιος 2000

Πάυλος Σπυράκης

*Καθηγητής Τμήματος Μηχανικών Η/Υ & Πληροφορικής
Πανεπιστημίου Πατρών*

Εισαγωγή: Βασικές έννοιες και ιστορικό

Σκοπός

Ο σκοπός αυτού του κεφαλαίου είναι η δημιουργία κοινής αφετηρίας γνώσεων στους φοιτητές του Ελληνικού Ανοικτού Πανεπιστημίου, μέσω της εισαγωγής στις βασικές έννοιες των λειτουργικών συστημάτων και μιας ιστορικής αναδρομής όσον αφορά την εξέλιξή τους.

Προσδοκώμενα αποτελέσματα

- Με τη μελέτη του κεφαλαίου αυτού, θα είστε σε θέση να:
- Κατανοήσετε, καταρχήν, τι είναι ένα λειτουργικό σύστημα.
- Περιγράψετε την ιστορική πορεία των λειτουργικών συστημάτων.
- Αναφέρετε τις βασικές αρχές λειτουργίας των λειτουργικών συστημάτων.

Έννοιες κλειδιά

- Λειτουργικό σύστημα
- Μικροκώδικας
- Ομαδική επεξεργασία
- Διαμοίραση χρόνου
- Συστήματα πραγματικού χρόνου
- Κεντρική μονάδα επεξεργασίας
- Κύρια και βοηθητική μνήμη
- Πολυπρογραμματισμός
- Εναλλαγή
- Γενές λειτουργικών συστημάτων

Δομή Κεφαλαίου

Το κεφάλαιο αυτό περιέχει τις παρακάτω ενότητες:

1.1 Βασικές Έννοιες

1.2 Ταξινόμηση Λειτουργικών Συστημάτων

1.3 Ιστορική Αναδρομή της Εξέλιξης των Λειτουργικών Συστημάτων

1.1 Βασικές έννοιες

Το λειτουργικό σύστημα είναι μια συλλογή από προγράμματα τα οποία ενεργούν ως «ενδιάμεσο» μεταξύ των χρηστών (π.χ. προγράμματα, εφαρμογές, συσκευές, άνθρωποι) και του Η/Υ. Αυτά τα προγράμματα μπορεί να είναι γραμμένα σε γλώσσα μηχανής ή assembly, ή ακόμα και σε μια γλώσσα όπως η C. Τα τελευταία χρόνια παρουσιάζεται η τάση ορισμένα από αυτά τα προγράμματα να είναι γραμμένα σε μικροκώδικα (κώδικα στοιχειωδών εντολών, ενσωματωμένο στο hardware).

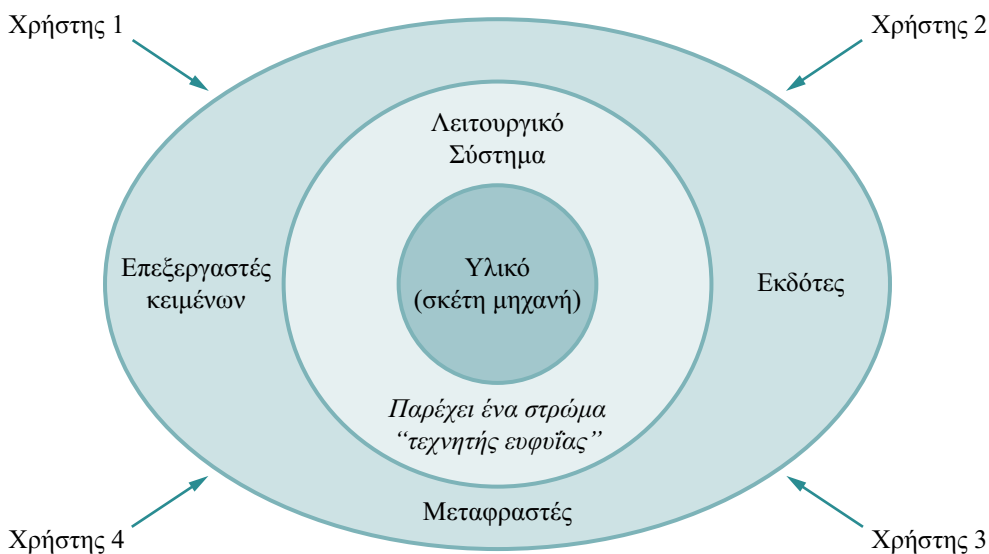
Χωρίς αμφιβολία, δύο είναι οι βασικοί στόχοι που εξυπηρετεί ένα λειτουργικό σύστημα:

- Να διαχειρίζεται τα δομικά στοιχεία (π.χ. μνήμη, hardware) του υπολογιστή
- Να εξασφαλίζει τη λειτουργία του Η/Υ με αποδοτικό τρόπο (ως προς το ποσοστό χρόνου χρήσης πόρων, τους χρόνους απόκρισης σε εντολές, τη διευκόλυνση του χρήστη κτλ.).

Ας δούμε τώρα τι σημαίνει «το λειτουργικό σύστημα είναι ενδιάμεσο μεταξύ των χρηστών και του Η/Υ».

1. Παρέχει διευκολύνσεις στο χρήστη για να μπορεί να χρησιμοποιεί τους πόρους του Η/Υ. Με τον όρο «πόροι ενός Η/Υ» εννοούμε:

- Τα εργαλεία Εισόδου/Εξόδου
- Τη μνήμη
- Την κεντρική μονάδα επεξεργασίας
- Δεδομένα με κάποιας μορφής οργάνωση (π.χ. αρχεία, μηνύματα)



Σχήμα 1.1

Σχέση του λειτουργικού συστήματος με τον Η/Υ και τους χρήστες

2. Ελέγχει την κατανομή των πόρων ενός Η/Υ στους χρήστες του με κάποια πολιτική κατανομής πόρων κατά τρόπο αποδοτικό, φιλικό και δίκαιο, χωρίς να δημιουργούνται προβλήματα λειτουργίας.
3. Εξασφαλίζει την προστασία των πόρων και των προγραμμάτων των χρηστών από ανεπιθύμητες καταστάσεις (σφάλμα εκτέλεσης προγράμματος, προσπέλαση σε μνήμη που ανήκει σε άλλο χρήστη ή στο σύστημα κτλ.)

Στο Σχήμα 1.1 δίνεται μια εικόνα της σχέσης του λειτουργικού συστήματος με τον Η/Υ και τους χρήστες.

1.2 Ταξινόμηση Λειτουργικών Συστημάτων

Μια πρώτη ταξινόμηση των λειτουργικών συστημάτων μπορεί να γίνει σε σχέση με το είδος της αλληλεπίδρασης που επιτρέπεται ανάμεσα σε ένα χρήστη και στο πρόγραμμά του, καθώς και στους περιορισμούς που μπαίνουν σε σχέση με την απόκριση του συστήματος. Υπάρχουν τρεις ιδανικές κατηγορίες λειτουργικών συστημάτων με βάση την πιο πάνω ταξινόμηση:

1. **Λειτουργικό σύστημα ομαδικής επεξεργασίας.** Είναι ένα λειτουργικό σύστημα στο οποίο οι εργασίες (τα προγράμματα) των χρηστών υποβάλλονται στον Η/Υ σε ομάδες. Σε αυτού του είδους τα λειτουργικά συστήματα, δεν είναι δυνατή καμιά αλληλεπίδραση ανάμεσα σε ένα χρήστη και στο πρόγραμμά του κατά τη διάρκεια της επεξεργασίας του προγράμματος. Ο χρόνος απόκρισης του Η/Υ για κάθε χρήστη είναι το χρονικό διάστημα από την υποβολή ως την παραλαβή της εργασίας του χρήστη. Το λειτουργικό σύστημα έχει τη δυνατότητα να κάνει καλή διανομή πόρων και να εφαρμόσει καλή πολιτική ως προς το χρόνο και τη σειρά εκτέλεσης των εργασιών.
2. **Λειτουργικό σύστημα με μοίρασμα χρόνου.** Εδώ το λειτουργικό σύστημα παρέχει τις υπολογιστικές του υπηρεσίες σε πολλούς χρήστες παράλληλα, επιτρέποντας στον καθέναν από αυτούς να αλληλεπιδρά με το πρόγραμμά του. Το φαινόμενο της «ταυτόχρονης» πρόσβασης στον Η/Υ είναι φυσικά απατηλό. Μοιράζοντας όμως το χρήσιμο χρόνο της ΚΜΕ (Κεντρική Μονάδα Επεξεργασίας) και άλλους πόρους του Η/Υ ανάμεσα σε πολλούς χρήστες, το λειτουργικό σύστημα δημιουργεί την ψευδαίσθηση στους χρήστες ότι ο καθένας από αυτούς έχει το σύνολο των πόρων του Η/Υ στη διάθεσή του. Το λειτουργικό σύστημα κατορθώνει να δίνει πάντοτε κάποια απόκριση σε οποιαδήποτε απαίτηση του χρήστη μέσα σε λίγο χρόνο. Στην πραγματικότητα, ο Η/Υ στρέφει την προσοχή του στην κάθε εργασία για ένα μικρό κομμάτι του χρόνου. Αν η εργασία δεν τελειώσει στο τέλος αυτού του χρόνου, τότε διακόπτεται προσωρινά και «αποθηκεύεται» σε μια ουρά

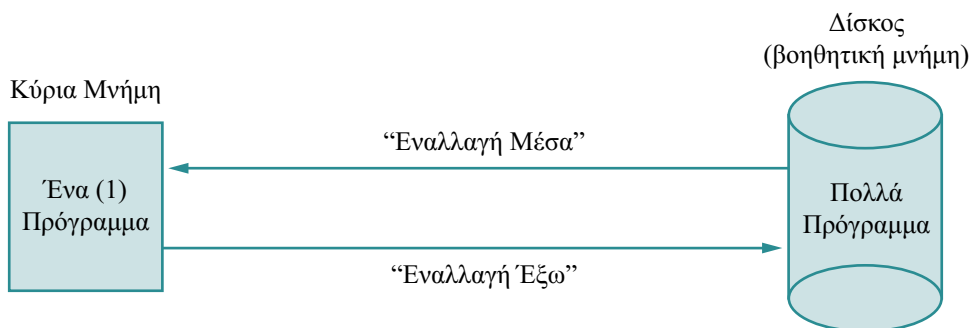
αναμονής. Αυτό επιτρέπει σε κάποια άλλη εργασία να πάρει χρόνο στον Η/Υ κ.ο.κ.

3. **Λειτουργικό σύστημα πραγματικού χρόνου.** Το λειτουργικό αυτό σύστημα εξυπηρετεί τις εργασίες που του υποβάλλονται μέσα σε αυστηρούς χρονικούς περιορισμούς. Δηλαδή ο χρόνος απόκρισης του Η/Υ είναι αυστηρά ορισμένος. Τέτοια λειτουργικά σύστημα συναντάμε, για παράδειγμα, σε αεροδρόμια για τον έλεγχο της εναέριας κυκλοφορίας. Σήματα διακοπής από τις εργασίες καλούν τον Η/Υ να διαθέσει υπολογιστικούς πόρους σε αυτές. Αν αυτά τα σήματα δεν πάρουν απάντηση σύντομα ή αν η επιθυμητή λειτουργία του Η/Υ αργήσει, τότε η εργασία ίσως αποτύχει ως προς το σκοπό της ή δεχτεί λάθος απαντήσεις.

Ένα συγκεκριμένο λειτουργικό σύστημα μπορεί να ανήκει σε περισσότερες από μία από τις πιο πάνω εξιδανικευμένες κατηγορίες. Είναι συνηθισμένο φαινόμενο τα λειτουργικά συστήματα με μοίρασμα χρόνου και αυτά του πραγματικού χρόνου να έχουν τη δυνατότητα να συμπεριφέρονται και ως λειτουργικά συστήματα ομαδικής επεξεργασίας. Αυτό το κατορθώνουν με διαφανή προς τους χρήστες τρόπο, όταν δεν υπάρχει στο προσκήνιο καμιά εργασία.

Ας συζητήσουμε τώρα τις μεθόδους με τις οποίες μπορούμε να κατανοήσουμε βαθύτερα τις εναλλακτικές μορφές λειτουργίας ενός λειτουργικού συστήματος. Η πιο κοινή μέθοδος είναι του *πολυπρογραμματισμού*. Ένα πολυπρογραμματιζόμενο λειτουργικό σύστημα είναι ένα λειτουργικό σύστημα το οποίο έχει τη δυνατότητα να διατηρεί περισσότερα από ένα προγράμματα χρηστών στην κύρια μνήμη του Η/Υ. Ακόμα, το λειτουργικό σύστημα έχει τη δυνατότητα να μοιράζει το χρόνο της ΚΜΕ, το χώρο της μνήμης και όλους τους άλλους πόρους του Η/Υ ανάμεσα στις εργασίες των χρηστών. Αυτό το μοίρασμα των πόρων επεκτείνεται και στο ίδιο το λειτουργικό σύστημα. Δηλαδή τα προγράμματα που αποτελούν το λειτουργικό σύστημα είναι στη διάθεση όλων των εργασιών των χρηστών του Η/Υ.

Μια άλλη τεχνική με την οποία μπορεί ένα λειτουργικό σύστημα να επεξεργάζεται πολλές εργασίες ταυτόχρονα είναι η τεχνική της εναλλαγής. Στο Σχήμα 1.2 δίνεται μια εικόνα για το πώς λειτουργεί αυτή η τεχνική.



Σχήμα 1.2
Λειτουργικό
σύστημα
εναλλαγής

Η βασική ιδέα είναι ότι στην κύρια μνήμη βρίσκεται κάθε φορά μια μόνο εργασία (ή λίγες μόνο εργασίες), ενώ στο δίσκο (βοηθητική μνήμη) βρίσκονται πολλές εργασίες. Το λειτουργικό σύστημα εναλλάσσει την εργασία που βρίσκεται στην κύρια μνήμη με μια άλλη από αυτές που βρίσκονται στο δίσκο. Αν η εργασία που μετακινήθηκε προς το δίσκο δεν έχει τελειώσει, θα μεταφερθεί αργότερα ξανά στην κύρια μνήμη. Τέτοια λειτουργικά συστήματα συναντάμε κυρίως σε μικρά υπολογιστικά συστήματα με μοίρασμα χρόνου.

Αξίζει να αναφέρουμε σε αυτό το σημείο μια έννοια που συχνά συγχέεται με τον πολυπρογραμματισμό, την έννοια της πολυεπεξεργασίας. Η «πολυεπεξεργασία» περιγράφει την αρχιτεκτονική του υλικού του υπολογιστικού συστήματος, ενώ ο «πολυπρογραμματισμός» έχει σχέση με λειτουργικά συστήματα, δηλαδή με το λογικό. Για να γίνουμε πιο σαφείς, θα δώσουμε και τον ορισμό της πολυεπεξεργασίας. Ένα υπολογιστικό σύστημα με πολυεπεξεργασία είναι ένα συγκρότημα υπολογιστικού υλικού με περισσότερους από έναν επεξεργαστές, οι οποίοι έχουν τη δυνατότητα να λειτουργούν ανεξάρτητα και παράλληλα. Ένα τέτοιο σύστημα περιλαμβάνει κεντρικές μονάδες επεξεργασίας, επεξεργαστές εισόδου – εξόδου, κανάλια δεδομένων, καθώς και επεξεργαστές ειδικής χρήσης. Το βιβλίο αυτό ασχολείται κύρια με λειτουργικά συστήματα πολυπρογραμματισμού, δίνοντας όμως και προεκτάσεις στα άλλα είδη, όπου αυτό κρίνεται σκόπιμο.

1.3 Ιστορική αναδρομή της εξέλιξης των Λειτουργικών Συστημάτων

Τα λειτουργικά συστήματα, όπως και οι Η/Υ, είχαν μια εξέλιξη που χαρακτηρίστηκε από σημαντικές αλλαγές στον τρόπο υλοποίησής τους, στους υπολογιστικούς πόρους στους οποίους δίνουν έμφαση και στην αντιμετώπιση του τελικού χρήστη. Αυτές οι αλλαγές δημιούργησαν τις εξής γενιές λειτουργικών συστημάτων:

1.3.1 Η μηδενική γενιά (1940)

Τα πρώτα υπολογιστικά συστήματα δεν είχαν λειτουργικό σύστημα. Οι χρήστες είχαν άμεση προσπέλαση στη γλώσσα μηχανής και προγραμματίζαν τα πάντα κυριολεκτικά «με το χέρι».

1.3.2 Η πρώτη γενιά (1950 – 1960)

Από το 1949 μέχρι το 1956 η βασική οργάνωση και ο τρόπος λειτουργίας του Η/Υ έμεινε σχετικά στο ίδιο σημείο. Η κλασική αρχιτεκτονική von Neumann των τότε Η/Υ συμπληρωνόταν με μια αυστηρά ακολουθιακή εκτέλεση εντολών, συμπεριλαμβάνοντας εντολές εισόδου – εξόδου. Ακόμα και στη διάρκεια του φορτώματος

(loading) και του τρεξίματος προγραμμάτων, οι χρήστες δούλευαν πάνω στην «κονσόλα», μεταβάλλοντας το περιεχόμενο των καταχωρητών, εκτελώντας εντολές βήμα βήμα, εξετάζοντας θέσεις μνήμης και, γενικά, αλληλεπιδρώντας με τον Η/Υ στο χαμηλότερο δυνατό επίπεδο (μηχανής). Τα προγράμματα γράφονταν σε απόλυτη γλώσσα μηχανής (π.χ. δεκαδικό ή οκταδικό σύστημα) και για τη φόρτωσή τους χρησιμοποιούνταν ένας «απόλυτος» φορτωτής προγραμμάτων (loader), που ήταν σε καθορισμένη θέση της μνήμης και που φόρτωνε το πρόγραμμα σε κάποιο καθορισμένο σύνολο διαδοχικών θέσεων μνήμης. Σε εκείνα τα «άγρια χρόνια», το λεγόμενο «υποβοηθητικό λογικό» (programming aids) βασικά δεν υπήρχε. Σιγά σιγά, όμως, καθώς γινόταν αντιληπτή η σημασία του συμβολικού προγραμματισμού και καθώς οι μεταφραστές σε «γλώσσες μηχανής» (assemblers) άρχισαν να εμφανίζονται, άρχισε να δημιουργείται μια «προκαθορισμένη» σειρά λειτουργιών: Ένας φορτωτής φόρτωνε έναν assembler στο σύστημα. Ο assembler μετάφραζε (σε απόλυτο κώδικα μηχανής) μερικούς σωρούς από κάρτες με προγράμματα χρηστών και «λογικό βιβλιοθήκης» (library routines). Ο απόλυτος κώδικας που έβγαινε από αυτή τη φράση γραφόταν σε ταινία ή κάρτες και μετά ένας φορτωτής ξαναχρησιμοποιούνταν για να φορτώσει τον κώδικα (ως ένα ενιαίο σύνολο πια) στην κύρια μνήμη του (απόλυτου) προγράμματος.

Οι παραπάνω αδυναμίες, μαζί με μια σειρά άλλους λόγους (το κόστος της συνεχούς ανθρώπινης παρέμβασης στη διαδικασία, τη διαθεσιμότητα της FORTRAN και άλλων γλωσσών, την ανάπτυξη κώδικα βιβλιοθήκης και κώδικα λειτουργιών εισόδου – εξόδου), έγιναν πειστικοί λόγοι για την εξέλιξη των ΛΣ σε ΛΣ «πρώτης γενιάς». Τα πρώτα συστήματα batch αυτοματοποίησαν την ακολουθία: φόρτωμα – μετάφραση – φόρτωμα – εκτέλεση, χρησιμοποιώντας ένα κεντρικό πρόγραμμα ελέγχου που ανακαλούσε και φόρτωνε «προγράμματα συστήματος» (system programs) (π.χ. assembler, compiler, φορτωτή ή κώδικες βιβλιοθήκης) και που αναλάμβανε τη μετάβαση από εργασία σε εργασία. Οι μεταφραστές γλωσσών ξαναγράφτηκαν, ώστε να παράγουν «μετατοπιζόμενο» (relocatable) κώδικα αντί για απόλυτο κώδικα. Οι «συνδετικοί φορτωτές» (linking loaders) άρχισαν να εμφανίζονται. Χάρη σε αυτούς, επιτράπηκε η ανάμειξη τμημάτων (ομάδων από κάρτες) «κώδικα πηγής» (source code) και τμημάτων μετατοπιζόμενου «κώδικα μηχανής αντικειμένου» (object code). Επίσης, οι κώδικες βιβλιοθήκης μπορούσαν πια να αποθηκεύονται σε μορφή μετατοπιζόμενου κώδικα.

Σε εκείνα τα ΛΣ, τα θέματα προστασίας του συστήματος ήταν τα πιο δύσκολα προβλήματα. Ήταν σχετικά εύκολο για το σύστημα να καταστρέψει τον εαυτό του ή να καταστραφεί από κάποιο χρήστη ή ήταν πιθανό ένας χρήστης να «διαβάσει» πέρα

από τη δική του εργασία, επεμβαίνοντας (ίσως άθελά του) στην επόμενη εργασία. Η κατανομή των πόρων (κύριας μνήμης και συσκευών Εισόδου/Εξόδου) ήταν δουλειά του χρήστη και όχι του ΛΣ.

1.3.3 Η δεύτερη γενιά (1959 – 1965)

Γύρω στα 1959 – 63 μερικές σημαντικές ανακαλύψεις στα συστήματα υλικού έδωσαν ώθηση στην παραπέρα εξέλιξη των ΛΣ. Ίσως η πιο σημαντική ανακάλυψη hardware, εκείνη την εποχή, ήταν το λεγόμενο «κανάλι δεδομένων» (data channel), δηλαδή ένας «πρωτόγονος» H/Y και τα εργαλεία εισόδου/εξόδου. Μόλις το κανάλι πάρει μια «εντολή αίτησης για I/O» από τον επεξεργαστή (ΚΜΕ), το κανάλι αρχίζει να εκτελεί και να ελέγχει την I/O εργασία ασύγχρονα και παράλληλα με την (συνεχιζόμενη εν τω μεταξύ) εκτέλεση εντολών από την ΚΜΕ. Δηλαδή η επικάλυψη των εργασιών της ΚΜΕ και των συστημάτων I/O (χρονικά) είναι πια γεγονός. Το κανάλι και η ΚΜΕ μοιράζονται την κύρια μνήμη, που περιέχει προγράμματα και δεδομένα και για τους δύο. Αρχικά, μόνο η ΚΜΕ μπορούσε να ρωτήσει ποια είναι η κατάσταση του καναλιού ανά πάσα στιγμή. Αργότερα όμως έγινε φανερό ότι το όλο σύστημα θα εργαζόταν πιο αποδοτικά εάν το κανάλι μπορούσε να διακόψει την εργασία της ΚΜΕ για να παραδώσει ένα μήνυμα, που συνήθως ήταν η λήξη μιας I/O εργασίας.

Αμέσως άρχισαν να γράφονται πολύπλοκα συστήματα λογικού που μπορούσαν να εκμεταλλευτούν τις πιθανές χρήσεις της νέας αρχιτεκτονικής. Τα συστήματα αυτά περιλάμβαναν διαδικασίες «λογικής μόνωσης» (software buffering), που επέτρεπαν, για παράδειγμα, «στοίβαγμα» (queuing) αποτελεσμάτων λόγω καθυστέρησης γραψίματος της εισόδου κτλ. Ακόμα, περιλάμβαναν ρουτίνες «χειρισμού των σημάτων διακοπής» (interrupt handling), για να γίνεται δυνατή η διαδικασία της απάντησης (από την ΚΜΕ) σε περίπτωση I/O interrupt και η διαδικασία επιστροφής του ελέγχου στη διαδικασία που διακόπηκε, μετά την εξυπηρέτηση του σήματος διακοπής.

1.3.4 Η τρίτη γενιά (1965 – 1980)

Η (φαινομενική) ασυμβατότητα ανάμεσα σε υπολογιστές για αριθμητικούς υπολογισμούς μεγάλης κλίμακας και σε εμπορικής χρήσης υπολογιστές αντιμετωπίστηκε πρώτα από την IBM με το σύστημα 360. Η γραμμή παραγωγής 360 ήταν μια σειρά από συμβατούς, όσον αφορά το λογισμικό, υπολογιστές. Το λειτουργικό τους σύστημα (OS 360), παρ' ότι μεγάλο και δύσχρηστο, εντούτοις εισάγει σημαντικότερες έννοιες, που δεν υπάρχουν στη δεύτερη γενιά.

Μεταξύ αυτών, τα πλέον σημαντικά είναι αυτά του «πολυπρογραμματισμού» (multi programming). Η ιδέα χρησιμοποιεί διαχωρισμούς της μνήμης σε διάφορα «μέρη»

(partitions), έτσι ώστε διάφοροι υπολογισμοί (εργασίες) να εξυπηρετούνται «ταυτόχρονα». Με τον τρόπο αυτό κατέστη δυνατή η συνύπαρξη και η συνεκτέλεση αριθμητικών υπολογισμών μεγάλης κλίμακας και εμπορικών (εντατικών σε I/O) υπολογισμών για πρώτη φορά.

Άλλη σημαντική εξέλιξη των συστημάτων της τρίτης γενεάς υπήρξε η ικανότητα φόρτωσης πολλαπλών εργασιών (από κάρτες) στο σύστημα, ώστε να μην καθυστερεί η εκτέλεσή τους. Η ιδιότητα αυτή ονομάστηκε Spooling (από το Simultaneous Peripheral Operations on Line). Με τη χρήση του Spooling περιορίστηκε σοβαρά η χρήση μαγνητικών ταινιών.

Η ανάγκη για γρήγορο χρόνο απόκρισης οδήγησε στην έννοια της «διαμοίρασης χρόνου» (time sharing), μια παραλλαγή του πολυπρογραμματισμού, όπου ο κάθε χρήστης έχει το δικό του τερματικό και το κεντρικό σύστημα μπορεί να δώσει «αλληλεπιδραστική» (interactive) εξυπηρέτηση σε όλους τους χρήστες χάρη στο μοίρασμα του χρόνου της CPU.

Το πλέον επιτυχημένο time-sharing λειτουργικό σύστημα της εποχής αυτής υπήρξε το MULTICS (διάδοχος του συστήματος CTSS, από το MIT, τα Bell Labs και τη General Electric και τα Unix), ικανό να εξυπηρετήσει εκατοντάδες χρήστες ταυτόχρονα.

Οι ιδέες του MULTICS οδήγησαν το σχεδιαστή λειτουργικών συστημάτων Ken Thomson (των Bell Labs) στη δημιουργία του πρώτου UNIX συστήματος (μιας μικρής κλίμακας ειδικής των MULTICS για τη σειρά PDP). Το αρχικό του όνομα ήταν UNICS (Uniplexed Information and Computing Service), αλλά ο συν-σχεδιαστής του, ο B. Kernigham, το μετέτρεψε σε UNIX.

Στην ομάδα προσετέθη και άλλος επιστήμονας των Bell Labs, ο D. Ritchie, και το UNIX ξαναγράφτηκε στη νέα (τότε) γλώσσα C (που σχεδιάστηκε και υλοποιήθηκε από τον Ritchie). Τα εργαστήρια Bell έδωσαν άδεια χρήσης και ανάπτυξης του UNIX στα πανεπιστήμια δωρεάν και το σύστημα γρήγορα μετακόμισε σε πλειάδα αρχιτεκτονικών (π.χ. VAX, Motorola κτλ.). Η χρήση του ακόμα και σήμερα είναι ταχύτατα ανοδική.

1.3.5 Η τέταρτη γενιά (1980 – 1990)

Η τέταρτη γενιά λειτουργικών συστημάτων χαρακτηρίστηκε από την ταυτόχρονη εμφάνιση και κυριαρχία των προσωπικών υπολογιστών (PC's). Αυτή η πραγματικότητα οδήγησε στην εμφάνιση λειτουργικών συστημάτων «φιλικών προς το χρήστη» (user friendly) και ευέλικτων (π.χ. XENIX, DOS κτλ.) και αργότερα, λόγω της εμφάνισης τοπικών δικτύων, σε ευέλικτα λειτουργικά συστήματα με αντιλήψεις *κατανεμημένου υπολογισμού* και *πολλαπλών καθηκόντων* (multi – tasking OS).

1.3.6 Η πέμπτη γενιά (1990 – σήμερα)

Η σημερινή πραγματικότητα χαρακτηρίζεται από την κυριαρχία του Διαδικτύου και του Παγκόσμιου Ιστού (Internet και Web) και την ύπαρξη γρήγορων τοπικών και μη τοπικών δικτύων. Τα σύγχρονα λειτουργικά συστήματα τροποποίησαν τις παλαιότερες αρχές σχεδιασμού, ώστε να ικανοποιούν τη διαχείριση των δικτυακών πόρων. Έτσι, έχουμε την κυρίαρχη σχεδιαστική αντίληψη των «processes» (διεργασιών ή διαδικασιών), καθώς και τη σχεδιαστική αρχή της επικοινωνίας των διεργασιών με τη μέθοδο «Client – Server» (Πελάτη – Εξυπηρετητή) και άλλες μεθόδους επικοινωνίας (π.χ. sockets), που συμβαδίζουν με τις τεχνολογικές εξελίξεις στα δίκτυα IP.

Τα λειτουργικά συστήματα που κυριαρχούν χαρακτηρίζονται από την **ανεξαρτησία τους** από δεσμεύσεις του hardware, καθώς και από τη φιλικότητα προς το χρήστη και τις επικοινωνιακές τους ικανότητες. Στο επίπεδο των PC's έχουμε την κυριαρχία των Windows, ενώ στο υψηλότερο επίπεδο των multi – tasking συστημάτων εμφανίζονται οι μετεξελίξεις και οι οικογένειες των UNIX (π.χ. Solaris, POSIX κτλ.), καθώς και τα Windows NT για μικρότερα multi – tasking συστήματα. Σήμερα, οι περισσότεροι προσωπικοί υπολογιστές έχουν το λειτουργικό σύστημα ενός χρήστη Windows (π.χ. των ετών 1998 ή 2000). Αυτό το λειτουργικό σύστημα είναι ιδιαίτερα φιλικό προς τον τελικό χρήστη (άνθρωπο), ενώ είναι εφοδιασμένο με πολλά προγράμματα / διευκολύνσεις για σύνδεση με το Διαδίκτυο, με περιφερειακές συσκευές και άλλες διευκολύνσεις για τη χρήση πολυμέσων [ψηφιακών δίσκων (CDs), ψηφιακού video (DVD), ήχου και εικόνας]. Μολονότι το λειτουργικό αυτό σύστημα δεν είναι σύστημα πολλών διεργασιών από την πλευρά του χρήστη, εντούτοις διαχειρίζεται με περίπλοκο τρόπο τη μνήμη και την ποικιλία των τερματικών ή επικοινωνιακών διατάξεων που χρειάζονται σήμερα τα συστήματα εφαρμογών πολυμέσων.

Στις μηχανές «υψηλής απόδοσης» (υπερυπολογισμού – high end) υπάρχει αρκετή αβεβαιότητα για το πλέον κατάλληλο λειτουργικό σύστημα, ενώ το UNIX απλώνεται και προς την κατεύθυνση αυτή.

Είναι επίσης χαρακτηριστική η πρώτη εμφάνιση στοιχείων τεχνητής νοημοσύνης και βάσεων δεδομένων / γνώσεων στο σχεδιασμό τέτοιων συστημάτων, ενώ η αλληλεπίδρασή τους (interfaces) με το Web φαίνεται καθοριστικό στοιχείο για την αποδοχή τους. Οι καταναεμημένες εφαρμογές είναι πλέον πραγματικότητα. Αντίθετα, υπάρχουν ιδιαίτερες δυσκολίες στην αποδοχή λειτουργικών συστημάτων για ευρυζωνικά δίκτυα (WAN's), παρά τις αξιόλογες εξελίξεις στο θέμα της διαχείρισης πόρων των δικτύων (NMS).

Σύνοψη

Στο πρώτο αυτό κεφάλαιο ορίσαμε τι είναι ένα λειτουργικό σύστημα. Ακολούθως ορίσαμε ως βασικούς στόχους ενός λειτουργικού συστήματος να μπορεί να κάνει έναν *H/Y* να δουλεύει φιλικά ως προς το χρήστη και να εξασφαλίζει τη λειτουργία του *H/Y* με αποδοτικό τρόπο. Επίσης, εξηγήσαμε πώς το λειτουργικό σύστημα είναι ενδιάμεσο μεταξύ χρηστών (τελικών χρηστών ή εφαρμογών) και *H/Y*.

Στη συνέχεια παρουσιάσαμε μια πρώτη ταξινόμηση των λειτουργικών συστημάτων σε σχέση με το είδος της αλληλεπίδρασης που επιτρέπεται ανάμεσα σε ένα χρήστη και στο πρόγραμμα του, καθώς και στους περιορισμούς που μπαίνουν σε σχέση με την απόκριση του συστήματος. Με βάση αυτή την ταξινόμηση, τα λειτουργικά συστήματα χωρίζονται στις ακόλουθες τρεις ιδανικές κατηγορίες: (α) λειτουργικά συστήματα ομαδικής επεξεργασίας, (β) λειτουργικά συστήματα με μοίρασμα χρόνου και (γ) λειτουργικά συστήματα πραγματικού χρόνου.

Η λειτουργία ενός λειτουργικού συστήματος πολλών χρηστών βασίζεται συνήθως στη μέθοδο του πολυπρογραμματισμού. Τα συστήματα αυτά έχουν τη δυνατότητα, ανάμεσα σε άλλα, να μοιράζουν το χρόνο της *KME*, το χώρο της κύριας μνήμης και όλους τους άλλους πόρους του *H/Y* ανάμεσα στις εργασίες των χρηστών.

Αναφερθήκαμε επίσης στην τεχνική διαχείρισης της μνήμης, που ονομάζεται «εναλλαγή» (*swapping*) και στον τρόπο με τον οποίο αυτή η τεχνική υποβοηθά τον πολυπρογραμματισμό.

Τέλος, στην τελευταία ενότητα αυτού του κεφαλαίου παρουσιάσαμε την εξέλιξη των λειτουργικών συστημάτων μέσα στο χρόνο, από το 1940 ως σήμερα, μέσα από πέντε γενιές λειτουργικών συστημάτων. Καθεμιά από τις γενιές αυτές έχει τα δικά της χαρακτηριστικά και τις δικές της ανάγκες, οι οποίες οδήγησαν τους ερευνητές να κάνουν ανακαλύψεις τόσο σε επίπεδο υλικού όσο και λογικού για να τις καλύψουν και να οδηγήσουν ιστορικά τα λειτουργικά συστήματα από τη μια γενιά στην άλλη μέσα στο χρόνο.

Βιβλιογραφία κεφαλαίου

A) ΠΡΟΑΙΡΕΤΙΚΗΣ ΑΝΑΓΝΩΣΗΣ

Brooks, *The mythical man-month: Essays on software engineering*, reading MA: Addison-Wesley, 1975.

Corbato, *On building systems that will fail*, commun. CACM, vol. 34, June 1991, pp. 72 – 81.

Deitel, *Operating Systems*, 2nd edition, Reading MA: Addison-Wesley, 1990.

Finkel, *An operating systems vade mecum*, 2nd edition, Englewood Cliffs, NJ: Prentice Hall, 1988.

Lampson, *Hints for computer systems design*, IEEE software, vol. 1, Jan. 1984, pp. 11 – 28.

Β) ΣΥΜΠΛΗΡΩΜΑΤΙΚΗ

Silberschatz et al., *Operating System Concepts*, 3rd edition, reading, MA: Addison–Wesley, 1991.

Tanenbaum, *Σύγχρονα λειτουργικά συστήματα*, Prentice Hall and Παπασωτηρίου, 1993.

Γλωσσάρι κεφαλαίου

Assembler:	μεταφραστής γλώσσας μηχανής
Batch system:	συστήματα ομαδικής επεξεργασίας
CPU:	κεντρική μονάδα επεξεργασίας
Data channel:	κανάλι δεδομένων
I/O (input/output):	είσοδος/έξοδος
Interrupt handling:	χειρισμός των συστημάτων διακοπής
Library routines:	λογικό βιβλιοθήκης
Linking loaders:	συνδετικοί φορτωτές
Load:	φορτώνω
Multiprogramming:	πολυπρογραμματισμός
Object code:	κώδικας μηχανής, κώδικας αντικειμένου
Programming aids:	επιβοηθητικό λογικό
Queuing:	στοίβαγμα
Real time system:	σύστημα πραγματικού χρόνου
Relocatable code:	μετατοπιζόμενος κώδικας
Software buffering:	λογική μόνωσης
Source code:	κώδικας πηγής
Swapping:	εναλλαγή
System program:	πρόγραμμα συστήματος
Timesharing:	σύστημα με μοίρασμα χρόνου



Διαδικασίες

Σκοπός

Στο κεφάλαιο αυτό παρουσιάζονται δύο κεντρικές έννοιες των λειτουργικών συστημάτων: η «διαδικασία» και η «διακοπή». Σκοπός του κεφαλαίου είναι η κατανόηση για μεν τις διαδικασίες θεμάτων αναπαράστασης, διαχείρισης και καταστάσεών τους, για δε τη διακοπή, η προέλευση, η αντιμετώπιση και η κατηγοριοποίησή τους.

Προσδοκώμενα αποτελέσματα

Με τη μελέτη του κεφαλαίου αυτού, θα είστε σε θέση να:

- Ορίσετε τι είναι μια διαδικασία
- Καθορίσετε ποιες πληροφορίες πρέπει να είναι γνωστές στο λειτουργικό σύστημα για κάθε διαδικασία που βρίσκεται στο σύστημα. Τι είναι δηλαδή ο PCB και ποιες πληροφορίες διατηρεί.
- Εξηγήσετε ποιες είναι οι καταστάσεις μιας διαδικασίας και πώς και πότε (μετά από ποιο συμβάν) μια διαδικασία μεταπίπτει από μια κατάσταση σε μια άλλη.
- Εξηγήσετε ποιες λειτουργίες μπορούν να γίνουν σε μια διαδικασία.
- Αναφέρετε τις αιτίες που επιφέρουν τη νάρκωση μιας ή περισσότερων διαδικασιών.
- Ορίσετε τι είναι η διακοπή.
- Περιγράψετε τις ενέργειες που γίνονται από το λειτουργικό σύστημα όταν συμβεί μια διακοπή.
- Αναφέρετε τις κατηγορίες διακοπών και τα χαρακτηριστικά τους.
- Ορίσετε τι είναι «ρυθμός χρήστη» και τι «ρυθμός επιβλέποντος».
- Περιγράψετε τις τρεις βασικές ενέργειες που πρέπει να γίνουν για την εξυπηρέτηση μιας διακοπής.
- Ορίσετε τι είναι ο ΧΕΔΠΕ (Χειριστής Διακοπών Πρώτου Επιπέδου).
- Ορίσετε τι είναι η αλυσίδα υπερπήδησης.
- Αναφέρετε τις διαφορές επανεισαγωγίμης και μη επανεισαγωγίμης διαδικασίας.
- Περιγράψετε τι είναι ο πυρήνας ενός λειτουργικού συστήματος και ποιες λειτουργίες χειρίζεται.

Έννοιες κλειδιά

- Διαδικασία
- Πυρήνας ΛΣ
- Κατάσταση διαδικασίας
- Τρέχουσα διαδικασία
- Έτοιμη διαδικασία (να «τρέξει»)
- Μπλοκαρισμένη διαδικασία
- Τμήμα ελέγχου της διαδικασίας
- Προτεραιότητα
- Κβάντο
- Μπλοκάρισμα και ξεμπλοκάρισμα διαδικασίας
- Δρομολόγηση διαδικασίας
- Νάρκωση και αφύπνιση διαδικασίας
- Διακοπή
- Διαχειριστής διακοπής
- Ρυθμός χρήστη και ρυθμός επιβλέποντος
- Αλυσίδα υπερπήδησης
- Επανεισαγώγιμη και μη επανεισαγώγιμη διαδικασία

Δομή κεφαλαίου

Το κεφάλαιο αυτό περιέχει τις παρακάτω ενότητες:

- 2.1 Εισαγωγή
- 2.2 Η Έννοια της Διαδικασίας
- 2.3 Η Αναπαράσταση της Διαδικασίας
- 2.4 Λειτουργίες επί Διαδικασιών
- 2.5 Η Έννοια της Διακοπής
- 2.6 Ο Πυρήνας του Λειτουργικού Συστήματος

2.1 Εισαγωγή

Σε αυτό το κεφάλαιο κυρίαρχο θέμα είναι η έννοια της διαδικασίας και ο τρόπος με τον οποίο μπορούμε να αναπαριστούμε τις διαδικασίες σε ένα λειτουργικό σύστημα. Αναπαράσταση μιας διαδικασίας στην ουσία σημαίνει η μέθοδος απόδοσης στη διαδικασία ορισμένων διακεκριμένων καταστάσεων και ενός δελτίου ταυτότητας το οποίο θα περιέχει τα χαρακτηριστικά της διαδικασίας που είναι απαραίτητα ώστε το λειτουργικό σύστημα να τη διαχειρίζεται σταθερά και δυναμικά.

Επίσης, γίνεται αναφορά στις διακοπές, ως το πιο χαμηλό μέσο επικοινωνίας μεταξύ των διαδικασιών, αλλά και μεταξύ του software και του hardware μέρους του H/Y.

Οι διακοπές είναι ίσως το πιο σημαντικό μέρος της «καρδιάς» ενός λειτουργικού συστήματος, που ονομάζουμε «πυρήνα» (kernel). Ο πυρήνας είναι το μέρος του λειτουργικού συστήματος που βρίσκεται πιο κοντά στο hardware.

2.2 Η έννοια της Διαδικασίας

Δεν υπάρχει αυστηρός ορισμός που να περιγράφει πλήρως την έννοια της διαδικασίας. Από την άλλη πλευρά, είναι σίγουρο πως κάποιος μπορεί να κατανοήσει διαισθητικά την έννοια αυτή αν αφιερώσει μερικές ώρες στο τερματικό ενός H/Y. Δίνουμε μερικούς ορισμούς της έννοιας αυτής.

Διαδικασία είναι:

- Ένα μέρος του κώδικα που εκτελείται.
- Το «ζωντάνεμα» ενός κώδικα.
- Η θέση του ελέγχου σ' ένα πρόγραμμα που εκτελείται.
- Αυτό που υποδηλώνει η ύπαρξη ενός τμήματος ελέγχου διαδικασιών (PCB) στο λειτουργικό σύστημα.
- Μια οντότητα στην οποία προσφέρουν εξυπηρέτηση το λειτουργικό σύστημα και οι επεξεργαστές.
- Ένα ξεχωριστό κομμάτι δραστηριότητας που μπορεί να εναλλαχθεί με κάποιο άλλο παρόμοιο κτλ.

Ο καθένας από τους παραπάνω ορισμούς εστιάζει σε μια πλευρά της έννοιας της διαδικασίας. Ο αναγνώστης καλείται να προσπαθήσει να τους διερμηνεύσει διαισθητικά και όχι τυπικά. Για παράδειγμα, στην πράξη μια διεργασία μπορεί να συνεπάγεται την εκτέλεση περισσότερων του ενός προγραμμάτων. Αντίστροφα, ένα συγκεκριμένο πρόγραμμα μπορεί να αποτελεί μέρος περισσότερων της μιας διαδικασιών (ένα τέτοιο παράδειγμα είναι ένα πρόγραμμα που χρησιμοποιείται για να προσθέτει

ένα αντικείμενο σε μια λίστα και που μπορεί να χρησιμοποιείται από κάθε διαδικασία που συμπεριλαμβάνει στις δραστηριότητες της χειρισμούς ουρών).

2.3 Η αναπαράσταση μιας Διαδικασίας

Μια διαδικασία εξελίσσεται και αλλάζει τα χαρακτηριστικά της μέσα στο χρόνο. Είναι πολύ σημαντικό για το λειτουργικό σύστημα να μπορεί να παρακολουθεί και να καταγράφει τις δυναμικές μεταβολές της διαδικασίας, για να μπορούν να εκτελούνται πολλές διαδικασίες ταυτόχρονα στο σύστημα. Η κατάσταση και όλα τα άλλα χαρακτηριστικά της διαδικασίας αναγράφονται και διατηρούνται στο μπλοκ ελέγχου διαδικασίας.

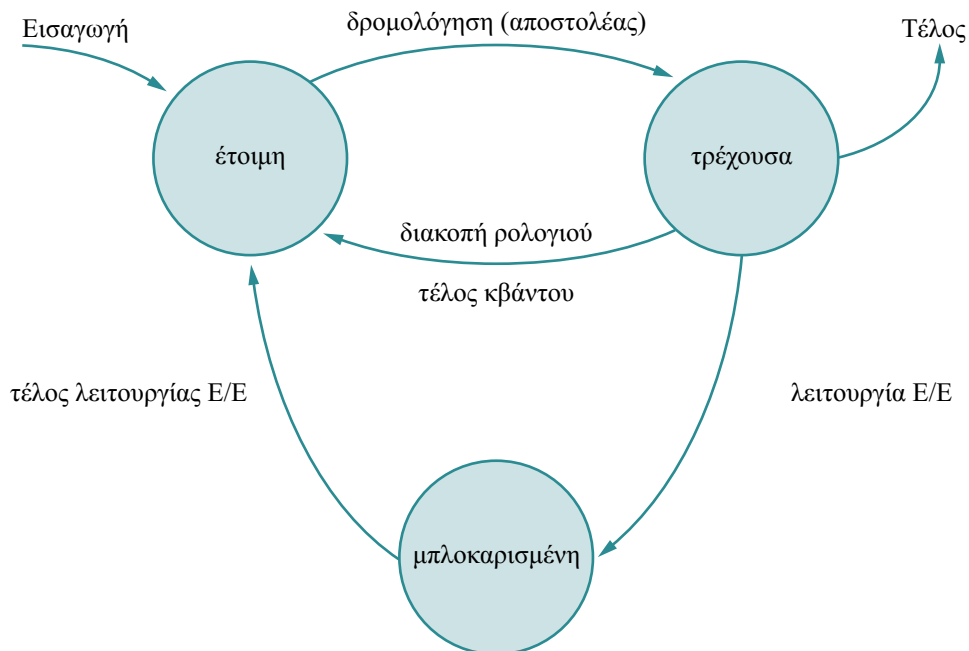
2.3.1 Καταστάσεις Διαδικασίας

Παρ' όλο που μακροσκοπικά πολλές διαδικασίες τρέχουν παράλληλα σε έναν υπολογιστή (αυτός, εξάλλου, είναι και ένας βασικός λόγος που χρειαζόμαστε το ΛΣ), στο «μικροχρόνο» κάθε CPU του υπολογιστή (και στο βιβλίο αυτό μελετάμε ΛΣ για υπολογιστές με μία μόνο CPU) εξυπηρετεί *μία μόνο* διαδικασία κάθε στιγμή: αυτή τη διαδικασία, και για το διάστημα αυτό, τη λέμε *τρέχουσα*. Οι υπόλοιπες διαδικασίες που περιμένουν τη σειρά τους για να εξυπηρετηθούν από την CPU λέμε ότι είναι *έτοιμες*, ενώ οι διαδικασίες που περιμένουν κάτι άλλο (π.χ. να απαντήσει το modem ή να τους διατεθεί αρκετή μνήμη) είναι οι *μπλοκαρισμένες*.

Έτσι, έχουμε τρεις βασικές καταστάσεις στις οποίες μπορεί να βρίσκεται μια διαδικασία: *τρέχουσα*, *έτοιμη* και *μπλοκαρισμένη*. Αν το δούμε από την πλευρά του λειτουργικού συστήματος, έχουμε (σε κανονική λειτουργία) μια *τρέχουσα* διαδικασία, μια *ουρά από έτοιμες* (εκ των οποίων μία είναι η επόμενη, αυτή που είναι στην κεφαλή της ουράς), και μια ομάδα από *μπλοκαρισμένες*.

Μια διαδικασία ξεκινάει ως «έτοιμη» και, όταν έρθει η σειρά της, γίνεται «τρέχουσα» για λίγο, μέχρι ή να περάσει ο χρόνος της (το quantum που ορίζει το ρολόι του υπολογιστή), οπότε επιστρέφει στην ετοιμότητα, ή να ζητήσει κάτι εκτός CPU (συνήθως I/O), οπότε μπλοκάρει έως ότου ικανοποιηθεί αυτό που ζήτησε (και τότε επιστρέφει στην ετοιμότητα).

Από την πλευρά του ΛΣ τώρα: ο δρομολογητής (ή αποστολέας), βασικό κομμάτι κάθε ΛΣ, επιλέγει ποια από τις έτοιμες διαδικασίες θα γίνει *τρέχουσα* (βάσει ενός αρκετά σύνθετου συστήματος προτεραιοτήτων, όπως θα δούμε παρακάτω), φροντίζει η *τρέχουσα* που μόλις τελείωσε να επιστρέψει στην ουρά των *ετοιμών*, η *τρέχουσα* που ζήτησε εξωτερικούς πόρους να πάει στις *μπλοκαρισμένες* και η *μπλοκαρισμένη* να πάει στις *έτοιμες* μόλις της διατεθεί ο πόρος που προκάλεσε το μπλοκάρισμα.

**Σχήμα 2.1**

Γράφημα
καταστάσεων
διαδικασίας

2.3.2 Το Μπλοκ Ελέγχου Διαδικασιών

Η αναπαράσταση σε φυσικό επίπεδο μιας διαδικασίας είναι μια δομή δεδομένων του λειτουργικού συστήματος που ονομάζεται «μπλοκ ελέγχου διαδικασίας» (Process Control Block – PCB). Το μπλοκ ελέγχου διαδικασίας είναι γνωστό και ως «περιγραφέας διαδικασίας» (process descriptor) ή «διάνυσμα κατάστασης» (state vector). Για συντομία, στο εξής θα το αναφέρουμε ως PCB.

Δείκτες	Κατάσταση διαδικασίας
	ταυτότητα διαδικασίας
	μετρητής προγράμματος
	καταχωρητές
	όρια μνήμης
	λίστα ανοιχτών αρχείων
	χρησιμοποιούμενοι πόροι
	⋮

Σχήμα 2.2

Το μπλοκ ελέγχου διαδικασίας

Στην ουσία, ένα PCB είναι ένα τμήμα της μνήμης που περιέχει τις πληροφορίες που είναι σχετικές με τη διαδικασία. Σε κάθε διαδικασία αντιστοιχεί ξεχωριστό PCB.

Οι πληροφορίες που είναι δυνατόν να περιέχονται στο PCB μιας διαδικασίας είναι:

- Η τρέχουσα κατάσταση της διαδικασίας.
- Η ταυτότητα της διαδικασίας. Η ταυτότητα κάθε διαδικασίας είναι ένας ακεραίος αριθμός, διαφορετικός για κάθε διαδικασία.
- Η ταυτότητα της διαδικασίας που δημιούργησε την εν λόγω διαδικασία, καθώς και η ταυτότητα του χρήστη στον οποίο ανήκει αυτή (η ταυτότητα του χρήστη είναι επίσης ένας ακεραίος αριθμός, διαφορετικός για τον καθένα).
- Η προτεραιότητα της διαδικασίας, που, εκτός των άλλων, ρυθμίζει με ποια σειρά η διαδικασία θα γίνει τρέχουσα.
- Ορισμένοι δείκτες που καταγράφουν τις διευθύνσεις της μνήμης στις οποίες βρίσκεται ο κώδικας και τα δεδομένα της συγκεκριμένης διαδικασίας.
- Δείκτες που καταγράφουν τους «πόρους» (resources) που έχει στην κατοχή της η διαδικασία.
- Μια περιοχή που περιέχει το «ευάλωτο περιβάλλον» (volatile environment) της διαδικασίας.

Όταν μια διαδικασία από τρέχουσα γίνεται έτοιμη ή μπλοκαρισμένη, τότε πρέπει να αποθηκευτούν όλες οι πληροφορίες που είναι απαραίτητες για να μπορέσει να συνεχίσει την εκτέλεσή της από το σημείο που σταμάτησε, όταν θα γίνει ξανά τρέχουσα στο ευάλωτο περιβάλλον του PCB. Μερικές από αυτές τις πληροφορίες είναι οι τιμές των καταχωρητών της μηχανής, όπως ο απαριθμητής προγράμματος και ο συσσωρευτής, ορισμένες διευθύνσεις της μνήμης που είναι απαραίτητες για τη συνέχιση της εκτέλεσης της διαδικασίας κτλ. Όταν η διαδικασία γίνει πάλι τρέχουσα, τότε το ευάλωτο περιβάλλον του PCB της θα φορτωθεί στους καταχωρητές της μηχανής και η εκτέλεση της διαδικασίας θα συνεχιστεί. Ο αποστολέας του λειτουργικού συστήματος υλοποιεί την παραπάνω διαδικασία.

Το λειτουργικό σύστημα πρέπει να μπορεί να διαχειρίζεται γρήγορα τα PCB's. Έτσι, ο σχεδιαστής του συστήματος δεν πρέπει να «φορτώσει» το PCB με πληροφορίες που δεν είναι άμεσα χρησιμοποιήσιμες. Πολλά υπολογιστικά συστήματα για να βελτιώσουν την ταχύτητα προσπέλασης στο PCB χρησιμοποιούν ειδικό καταχωρητή μνήμης, που πάντα δείχνει στο PCB της τρέχουσας διαδικασίας. Επίσης, συχνά χρησιμοποιούνται εντολές μηχανής για την επαναποθήκευση και φόρτωση του άστατου περιβάλλοντος της διαδικασίας από και προς το PCB.

2.4 Λειτουργίες επί Διαδικασιών

Το λειτουργικό σύστημα πρέπει να είναι σε θέση να εκτελέσει μερικές λειτουργίες επί διαδικασιών. Μερικές από αυτές είναι:

Δημιουργία μιας διαδικασίας

Το λειτουργικό σύστημα ονομάζει κάθε καινούρια διαδικασία, την τοποθετεί στη λίστα των γνωστών διαδικασιών του συστήματος, δίνει στη διαδικασία μια αρχική προτεραιότητα, δημιουργεί το μπλοκ ελέγχου διαδικασίας και, τέλος, αποδίδει στη διαδικασία τους αρχικούς πόρους που είναι απαραίτητοι σε αυτή. Μια διαδικασία μπορεί να δημιουργήσει καινούριες διαδικασίες. Τότε, η πρώτη διαδικασία ονομάζεται «γονέας–διαδικασία», ενώ οι δημιουργούμενες «παιδιά–διαδικασίες». Έτσι, έχουμε τη δημιουργία μιας ιεραρχικής δενδροειδούς δομής διαδικασιών.

Οριστική διακοπή μιας διαδικασίας

Σε αρκετές περιπτώσεις, το λειτουργικό σύστημα πρέπει να απομακρύνει μια ορισμένη διαδικασία. Αυτό σημαίνει πως πρέπει να ελευθερωθούν οι πόροι που κατέχει αυτή η διαδικασία, το μπλοκ ελέγχου διαδικασίας να διαγραφεί και η ίδια η διαδικασία να απομακρυνθεί από τις λίστες και τους πίνακες του λειτουργικού συστήματος που είναι συνδεδεμένη. Η οριστική διακοπή μιας διαδικασίας είναι αρκετά πιο πολύπλοκη όταν η διαδικασία έχει ένα ή περισσότερα παιδιά. Σε ορισμένα λειτουργικά συστήματα τα «παιδιά–διαδικασίες» διακόπτονται οριστικά μαζί με το γονέα (π.χ. UNIX), ενώ σε ορισμένα άλλα τα παιδιά συνεχίζουν ανεξάρτητα την εκτέλεσή τους.

Αλλαγή προτεραιότητας

Είναι πολλές οι αιτίες που αναγκάζουν ένα λειτουργικό σύστημα να αλλάξει την προτεραιότητα μιας διαδικασίας (π.χ. χρονοδρομολόγηση της διαδικασίας, εκτέλεση λειτουργίας I/O κτλ.). Αυτό σημαίνει μια απλή αλλαγή στο αντίστοιχο πεδίο του PCB της διαδικασίας.

Μπλοκάρισμα / Ξεμπλοκάρισμα διαδικασίας

Ήδη έχουμε αναφερθεί στις παραπάνω λειτουργίες. Εδώ απλώς συμπληρώνουμε πως το μπλοκάρισμα μιας διαδικασίας προκαλείται από την ίδια τη διαδικασία, ενώ το ξεμπλοκάρισμα από οντότητες και αιτίες εξωτερικές προς τη διαδικασία.

Δρομολόγηση διαδικασίας

Επίσης, έχουμε ήδη αναφερθεί στην έννοια της δρομολόγησης μιας διαδικασίας. Η

ταχύτητα δρομολόγησης είναι ένα «κρίσιμο» σημείο για το λειτουργικό σύστημα, και γι' αυτό πρέπει να διατηρείται υψηλή.

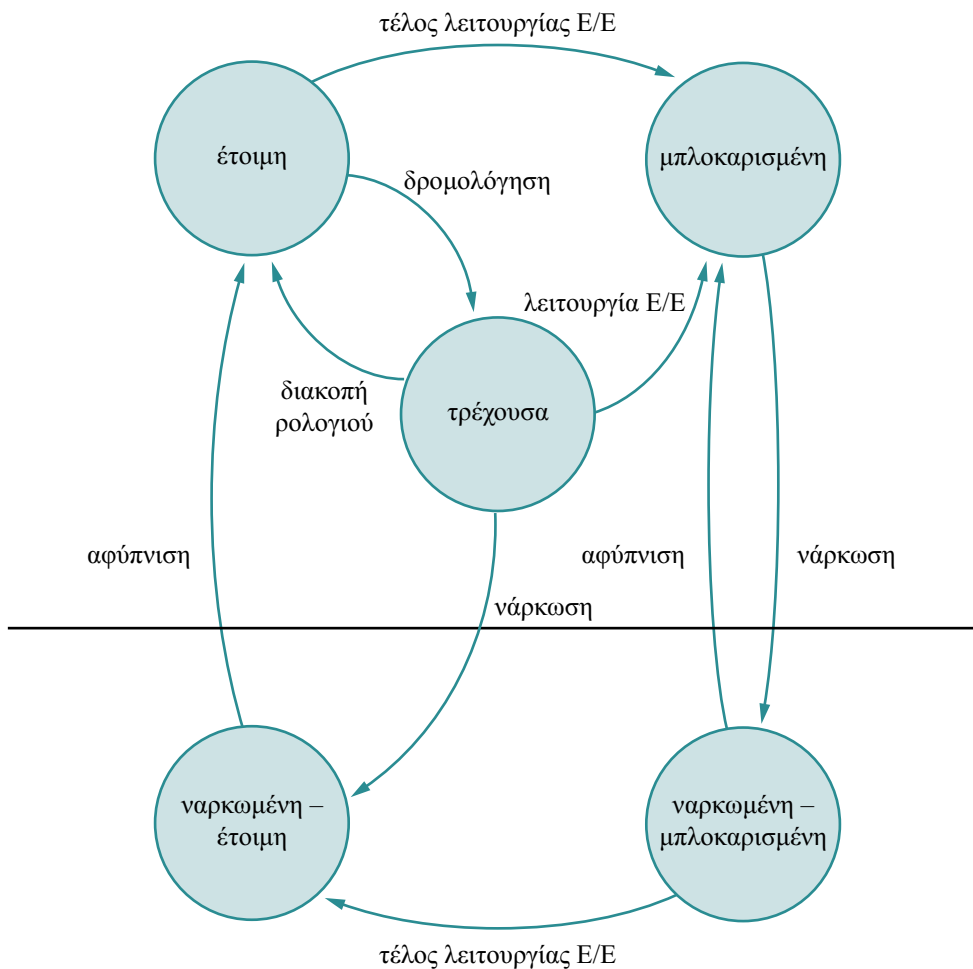
Νάρκωση / Αφύπνιση διαδικασίας (suspend / resume)

Σε αρκετά λειτουργικά συστήματα υπάρχουν οι παραπάνω λειτουργίες. Συνήθως η νάρκωση μιας διαδικασίας διαρκεί σύντομο χρονικό διάστημα. Εναπόκειται στο λειτουργικό σύστημα η απόφαση εάν κατά τη διάρκεια της νάρκωσης οι πόροι που κατέχει η διαδικασία θα επιστραφούν σε κοινή χρήση. Επίσης, το τελευταίο εξαρτάται από τη φύση των πόρων. Για παράδειγμα, η μνήμη είναι από τους πρώτους πόρους που θα ελευθερωθούν όταν η διαδικασία ναρκωθεί. Η αφύπνιση (ή ενεργοποίηση) μιας διαδικασίας σημαίνει τη συνέχιση της διαδικασίας από το σημείο που βρισκόταν όταν ναρκώθηκε.

Υπάρχουν αρκετές αιτίες που προκαλούν τη νάρκωση μιας ή περισσότερων διαδικασιών:

- Εάν το σύστημα υπολειτουργεί λόγω μερικής βλάβης (π.χ. όταν ο εκτυπωτής δεν έχει χαρτί ή η τηλεφωνική γραμμή δεν απαντά), τότε οι διαδικασίες που εκτελούνται είναι δυνατόν να ναρκωθούν και να αφυπνιστούν αργότερα, όταν ξεπεραστεί το πρόβλημα.
- Όταν ένας χρήστης θέλει να ελέγξει τα αποτελέσματα μιας διαδικασίας πριν αυτή τελειώσει, έχει τη δυνατότητα να τη ναρκώσει και να την αφυπνίσει, αφού βεβαιωθεί πως η εκτέλεση της διαδικασίας είναι σωστή.
- Στην περίπτωση που το σύστημα είναι πολύ φορτωμένο, τότε μερικές διαδικασίες μπορούν να ναρκωθούν και να αφυπνιστούν αργότερα, όταν το φορτίο επανέλθει σε κανονικά επίπεδα.

Η χρήση των λειτουργιών νάρκωσης / αφύπνισης σημαίνει πως πρέπει να προστεθούν καινούριες καταστάσεις στο γράφημα καταστάσεων μιας διαδικασίας (Σχήμα 2.3). Έτσι, μια διαδικασία μπορεί να βρεθεί στην κατάσταση «ναρκωμένη-έτοιμη» και «ναρκωμένη-μπλοκαρισμένη». Όταν αφυπνιστεί μια «ναρκωμένη-έτοιμη» διαδικασία, γίνεται «έτοιμη». Όταν μια διαδικασία είναι «τρέχουσα» και ναρκωθεί, γίνεται «ναρκωμένη-έτοιμη», ενώ, όταν είναι «μπλοκαρισμένη», γίνεται «ναρκωμένη-μπλοκαρισμένη». Η «ναρκωμένη-μπλοκαρισμένη» διαδικασία για την οποία έχει περατωθεί η αιτία για την οποία μπλοκαρίστηκε θα γίνει «ναρκωμένη-έτοιμη». Οι λόγοι που οδηγούν στη νάρκωση μια «μπλοκαρισμένη» διαδικασία είναι το γεγονός ότι η αιτία του μπλοκαρίσματος είναι δυνατόν να αργήσει απεριόριστα (π.χ. εκτυπωτής ή modem).

**Σχήμα 2.3**

Πλήρες γράφημα
καταστάσεων
διαδικασίας

2.5 Η έννοια της Διακοπής

Γενικά, μπορούμε να πούμε πως η διακοπή είναι ένα γεγονός που αλλάζει τη σειρά με την οποία ο επεξεργαστής εκτελεί εντολές. Αν και σε μερικά λειτουργικά συστήματα υπάρχουν ορισμένα είδη διακοπών που προκαλούνται από το λογικό του υπολογιστικού συστήματος, οι διακοπές γενικά παράγονται από το υλικό. Όταν συμβεί μια διακοπή, το λειτουργικό σύστημα κάνει τις εξής ενέργειες:

- Αποκτά τον έλεγχο του επεξεργαστή.
- Φυλάσσει την κατάσταση της διακοπόμενης διαδικασίας. Όπως αναφέρθηκε, οι πληροφορίες που αφορούν την κατάσταση της διαδικασίας είναι δυνατόν να σωθούν στο μπλοκ ελέγχου διαδικασίας.
- Διερευνά την αιτία και την προέλευση της διακοπής και μεταβιβάζει τον έλεγχο στην αντίστοιχη ρουτίνα που θα εξυπηρετήσει τη διακοπή.

Μια διακοπή μπορεί να προκληθεί από την τρέχουσα διαδικασία, αλλά μπορεί να είναι άσχετη με αυτή. Γενικά, σε όλα τα λειτουργικά συστήματα διακρίνουμε αρκετούς τύπους διακοπών.

2.5.1 Τύποι Διακοπών

Δεν είναι εύκολο να μιλήσει κανείς για συγκεκριμένους τύπους διακοπών, γιατί σχεδόν κάθε είδος λειτουργικού συστήματος χρησιμοποιεί τους δικούς του τύπους και ονόματα διακοπών. Πάντως, μπορούμε να διακρίνουμε δύο μεγάλες κατηγορίες: τις διακοπές που προέρχονται από το άμεσο περιβάλλον του επεξεργαστή (εσωτερικές) και τις διακοπές που είναι εξωτερικές του επεξεργαστή.

Στην πρώτη κατηγορία ανήκουν οι διακοπές:

- Διακοπές **κλήσης επιβλέποντος** (supervisor call). Προκαλούνται από την τρέχουσα διαδικασία, η οποία απαιτεί τη χρήση μιας προνομιακής εντολής. Ονομάζουμε προνομιακές εντολές ορισμένες λειτουργίες του λειτουργικού συστήματος, όπως την εκτέλεση εισόδου/εξόδου, την απόκτηση περισσότερης μνήμης, την προσπέλαση καταχωρητών κτλ., οι οποίες, για λόγους ασφάλειας, δεν μπορούν να αφεθούν σε ελεύθερη χρήση από τις διαδικασίες των χρηστών. Αντί αυτού, η συγκεκριμένη διαδικασία που θέλει να εκτελέσει μια προνομιακή λειτουργία μέσω μιας διακοπής κλήσης επιβλέποντος απαιτεί από το λειτουργικό σύστημα να εκτελέσει, αντί αυτής, την προνομιακή λειτουργία. Με αυτό τον τρόπο το λειτουργικό σύστημα είναι εν γνώσει των προσπαθειών των διαδικασιών των χρηστών να παραβιάσουν τα όριά του και είναι δυνατόν να αρνηθεί ορισμένες απαιτήσεις από αυτές όταν δε διαθέτουν τα κατάλληλα προνόμια.
- Διακοπές **ελέγχου προγράμματος**. Προκαλούνται από διάφορους τύπους λαθών των προγραμμάτων των χρηστών, όπως στην περίπτωση διαίρεσης διά μηδενός, στην προσπάθεια εκτέλεσης μιας προνομιακής εντολής ή στην προσπάθεια εκτέλεσης ενός άκυρου κώδικα λειτουργίας κτλ.
- Διακοπές που προκαλούνται από **το ρολόι πραγματικού χρόνου** του συστήματος, διακοπές **ελέγχου του υλικού** του συστήματος κτλ.

Μερικές από τις εξωτερικές διακοπές είναι:

- Διακοπές **εισόδου/εξόδου**. Προκαλούνται από το υλικό εισόδου/εξόδου. Σηματοδοτούν στον επεξεργαστή πως η κατάσταση ενός καναλιού ή μιας περιφερειακής συσκευής έχει αλλάξει. Διακοπές εισόδου/εξόδου έχουμε όταν μια λειτουργία περατωθεί, όταν συμβεί ένα λάθος ή όταν ένα περιφερειακό είναι έτοιμο για λειτουργία.

- Διακοπές που προκαλούνται από **την πίεση του πλήκτρου restart και interrupt** του πληκτρολογίου ή όταν **ληφθεί ένα σήμα από διαφορετικό επεξεργαστή** κτλ.

Γενικά, μπορούμε να πούμε πως οι περισσότεροι υπολογιστές λειτουργούν σε έναν από δύο «ρυθμούς» (modes), συνήθως γνωστούς ως «ρυθμός χρήστη» (user mode) και «ρυθμός επιβλέποντος» (supervisor mode). Κατά τη λειτουργία σε «ρυθμό χρήστη» εκτελούνται τα προγράμματα των χρηστών. Στο «ρυθμό επιβλέποντος» το λειτουργικό σύστημα έχει τον έλεγχο του επεξεργαστή, και τότε μπορούν να εκτελεστούν οι προνομιακές εντολές. Η εμφάνιση μιας εκ των διακοπών που αναφέραμε προκαλεί τη μετάπτωση από το «ρυθμό χρήστη» στο «ρυθμό επιβλέποντος».

2.5.2 Χειρισμός Διακοπών

Το λειτουργικό σύστημα συμπεριλαμβάνει ρουτίνες γνωστές ως «χειριστές διακοπών», που είναι υπεύθυνες για την επεξεργασία κάθε διαφορετικού τύπου διακοπής. Τρεις είναι οι βασικές ενέργειες που πρέπει να γίνουν για την εξυπηρέτηση μιας διακοπής:

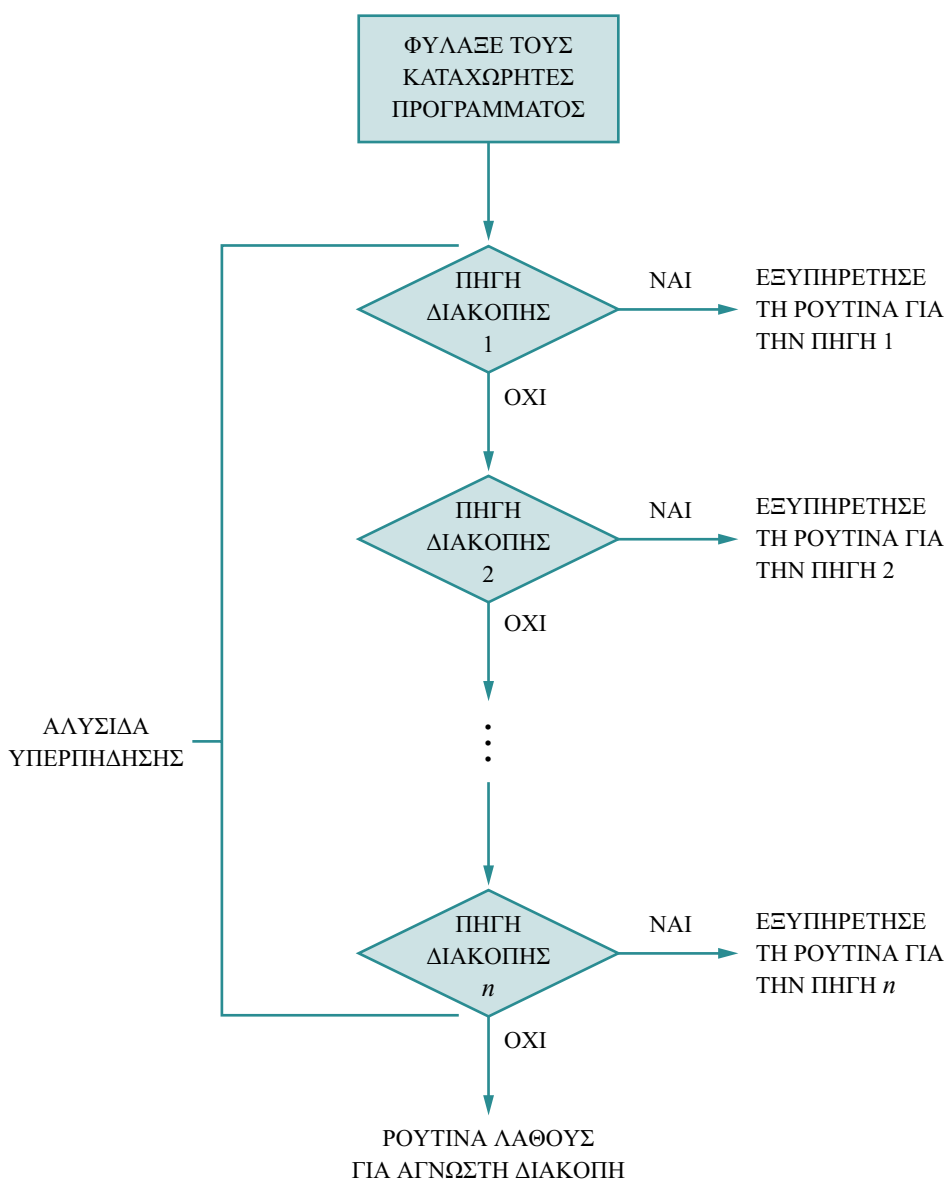
1. Η διάσωση της κατάστασης της διακοπτόμενης διαδικασίας.
2. Ο καθορισμός της πηγής της διακοπής.
3. Η μεταβίβαση του ελέγχου στον αντίστοιχο χειριστή της διακοπής για την εξυπηρέτηση της διακοπής.

Οι παραπάνω τρεις λειτουργίες υλοποιούνται από το μέρος του λειτουργικού συστήματος που ονομάζεται «χειριστής διακοπών πρώτου επιπέδου» (ΧΕΔΠΕ). Ο «χειριστής διακοπών πρώτου επιπέδου» είναι συνήθως ένα σχετικά απλό πρόγραμμα που λειτουργεί σε ένα συγκεκριμένο μέρος της μνήμης, το δε σύνολο των καταχωρητών που απαιτείται για τη λειτουργία του δεν είναι μεγάλο. Σίγουρα, πάντως, είναι μικρότερο σε μέγεθος από το άστατο περιβάλλον της διακοπτόμενης διαδικασίας, το οποίο δε χρειάζεται να φυλαχτεί στο σύνολό του. Υπάρχουν αρκετές τεχνικές για το φύλαγμα των τιμών των καταχωρητών που χρησιμοποιεί η διακοπτόμενη διαδικασία. Για παράδειγμα, είναι δυνατή η παροχή ενός επιπλέον συνόλου καταχωρητών για χρήση μόνο σε «ρυθμό επιβλέποντος». Ο «χειριστής διακοπών πρώτου επιπέδου» μπορεί να χρησιμοποιήσει αυτούς τους καταχωρητές και να αφήσει εκείνους της διακοπτόμενης διαδικασίας άθικτους.

Όταν δεν υπάρχουν αρκετοί καταχωρητές στο σύστημα, τότε το ευάλωτο περιβάλλον της διαδικασίας μπορεί να διασωθεί στο αντίστοιχο σημείο του μπλοκ ελέγχου διαδικασίας.

Επίσης, ο καθορισμός της πηγής της διακοπής μπορεί να γίνει με διάφορους τρό-

πους, ανάλογα με το υλικό που παρέχεται. Έτσι, μια περίπτωση είναι όταν όλες οι διακοπές μεταφέρουν τον έλεγχο στην ίδια θέση και η αναγνώριση πραγματοποιείται από μια ακολουθία από ελέγχους των «σημαιών κατάστασης» (status flags) όλων των πιθανών πηγών διακοπής. Αυτή η ακολουθία ονομάζεται «αλυσίδα υπερπήδησης» (Σχήμα 2.4) και είναι φανερό πως πρέπει να την κωδικοποιήσουμε, έτσι ώστε οι περισσότερες συχνές διακοπές να εμφανίζονται στην κεφαλή της.



Σχήμα 2.4

Αλυσίδα υπερπήδησης για εξυπηρέτηση διακοπής

Είναι δυνατή η χρησιμοποίηση πιο τελειοποιημένου υλικού το οποίο να ξεχωρίζει ανάμεσα σε πηγές διακοπών και να μεταφέρει τον έλεγχο σε μια διαφορετική θέση

για κάθε πηγή. Αυτό ελαττώνει το χρόνο που χρειάζεται για να αναγνωριστεί μια διακοπή, με κόστος τις επιπλέον θέσεις διακοπής που χρησιμοποιούνται και, φυσικά, το περισσότερο υλικό. Μια ενδιάμεση λύση είναι να παρέχεται ένας μικρός αριθμός θέσεων διακοπής, ο καθένας από τους οποίους διαμοιράζεται από ένα σύνολο πηγών διακοπής. Το πρώτο βήμα της αναγνώρισης διακοπής επιτυγχάνεται από το υλικό, ενώ μια μικρή αλυσίδα υπερπήδησης, που ξεκινάει από κάθε θέση, είναι ικανή να ολοκληρώσει τη διαδικασία.

Όταν η εξυπηρέτηση της διακοπής ολοκληρωθεί, τότε η CPU παραχωρείται είτε στη διαδικασία που έτρεχε κατά τη στιγμή της διακοπής είτε στην έτοιμη διαδικασία με τη μεγαλύτερη προτεραιότητα. Αυτό εξαρτάται από το αν η δρομολόγηση των διαδικασιών είναι «επανεισαγώγιμη» (preemptive) ή «μη επανεισαγώγιμη» (nonpreemptive). Στη δεύτερη περίπτωση κάθε διαδικασία που καταλαμβάνει την CPU παραμένει εκεί μέχρι να περατωθεί η εκτέλεσή της. Σε αυτή την περίπτωση η διακοπτόμενη διαδικασία καταλαμβάνει ξανά την CPU. Αν η δρομολόγηση είναι επανεισαγώγιμη, τότε καταλαμβάνει την CPU η πρώτη διαδικασία που βρίσκεται στην κεφαλή της ουράς των έτοιμων διαδικασιών. Οι διαδικασίες δεν εκτελούνται μέχρι τέλους σε αυτή την περίπτωση, αλλά μέχρι να διακοπούν για λειτουργίες I/O ή λόγω του ότι τελείωσε το quantum τους. Όταν η δρομολόγηση δεν είναι επανεισαγώγιμη, η διακοπτόμενη διαδικασία προστίθεται στην ουρά των έτοιμων διαδικασιών.

2.6 Ο Πυρήνας του Λειτουργικού Συστήματος

Όλες οι λειτουργίες που αφορούν τις διαδικασίες υλοποιούνται από το μέρος του λειτουργικού συστήματος που ονομάζεται «πυρήνας» (kernel). Ο πυρήνας αποτελεί μόνο ένα μικρό κλάσμα ολόκληρου του λειτουργικού συστήματος, όμως είναι από τα περισσότερο χρησιμοποιούμενα μέρη του.^[1]

Επίσης, επειδή ο πυρήνας χτίζεται κατευθείαν πάνω στο υλικό του H/Y, είναι το περισσότερο εξαρτώμενο από τη μηχανή μέρος του λειτουργικού συστήματος και είναι απαραίτητο να γραφεί σε «γλώσσα μηχανής» (assembly), με λίγες μικρές εξαιρέσεις. Ο περιορισμός στη χρησιμοποίηση της γλώσσας assembly σε ένα μικρό μέρος του λειτουργικού συστήματος (π.χ. στο UNIX όχι παραπάνω από 1000 εντολές) έγινε δυνατός χάρη στην ιεραρχική δομή των λειτουργικών συστημάτων και προσέφερε αρκετά στην υλοποίηση ενός προϊόντος απαλλαγμένου από λάθη, εύκολα κατανοήσιμου και «επιδεκτικού συντήρησης» (maintainable).

[1] Γι' αυτό το λόγο ο πυρήνας παραμένει κανονικά στην κύρια μνήμη, ενώ τα άλλα μέρη του λειτουργικού συστήματος μεταφέρονται από και προς τη δευτερεύουσα μνήμη, όταν είναι απαραίτητο.

Ο πυρήνας ενός λειτουργικού συστήματος συνήθως περιέχει κώδικα που υλοποιεί τις παρακάτω λειτουργίες:

- Χειρισμός διακοπών (βλ. 2.5.2)
- Δημιουργία και καταστροφή διαδικασιών (βλ. 2.4)
- Δρομολόγηση (dispatching) (βλ. 2.4)
- Νάρκωση και αφύπνιση διαδικασίας (βλ. 2.4)
- Συντονισμός διαδικασιών (βλ. 3.3)
- Επικοινωνία μεταξύ διαδικασιών (βλ. 2.1)
- Υποστήριξη δραστηριοτήτων E/E (βλ. 2.4)
- Υποστήριξη δέσμευσης και αποδέσμευσης μνήμης κτλ.

Σύνοψη

Βασικό θέμα της μελέτης μας σε αυτό το κεφάλαιο ήταν η διαδικασία. Η διαδικασία είναι η οντότητα εκείνη στην οποία προσφέρουν εξυπηρέτηση το λειτουργικό σύστημα και οι επεξεργαστές. Μπορεί να βρίσκεται σε μια από τρεις καταστάσεις: τρέχουσα, έτοιμη και μπλοκαρισμένη. Όταν μια διαδικασία φτάσει στην κεφαλή της ουράς έτοιμων διαδικασιών (έχει τη μεγαλύτερη προτεραιότητα), τότε είναι και η επόμενη διαδικασία η οποία θα εισέλθει στην CPU (μόλις είναι διαθέσιμη) για να εκτελεστεί. Η τρέχουσα διαδικασία σταματάει να εκτελείται (βγαίνει από την CPU) όταν περάσει το κβάντο της ή όταν ζητήσει μια λειτουργία εισόδου/εξόδου. Η τρέχουσα διαδικασία θα μεταφερθεί στην ουρά των έτοιμων διαδικασιών στην πρώτη περίπτωση, από αυτές που περιγράψαμε πιο πάνω, και στην ουρά των μπλοκαρισμένων διαδικασιών στη δεύτερη περίπτωση. Στο μπλοκ ελέγχου διαδικασιών διατηρούνται όλες οι απαραίτητες πληροφορίες για μια διαδικασία, έτσι ώστε να μπορεί να συνεχίζει την εκτέλεσή της κάθε φορά που βρίσκεται στην CPU από το σημείο ακριβώς που σταμάτησε την προηγούμενη φορά. Σε μια διαδικασία μπορούν να γίνουν οι ακόλουθες λειτουργίες: δημιουργία, καταστροφή, αλλαγή προτεραιότητας, μπλοκάρισμα και ξεμπλοκάρισμα, δρομολόγηση, νάρκωση και αφύπνιση.

Στην Ενότητα 2.5 μελετάμε την έννοια της διακοπής και ορίζουμε ως διακοπή ένα γεγονός που αλλάζει τη σειρά με την οποία ο επεξεργαστής εκτελεί εντολές. Οι διακοπές μπορούν να χωριστούν στις εξής πέντε κατηγορίες: διακοπές κλήσης επιβλέποντος, διακοπές ελέγχου προγράμματος, διακοπές που προκαλούνται από το ρολόι

πραγματικού χρόνου του συστήματος, διακοπές εισόδου/εξόδου και διακοπές που προκαλούνται από την πίεση των πλήκτρων *restart* και *interrupt*. Ο χειρισμός διακοπών γίνεται από ρουτίνες του λειτουργικού συστήματος, που είναι γνωστές ως «χειριστές διακοπών». Οι βασικές ενέργειες που πρέπει να γίνονται για την εξυπηρέτηση των διακοπών είναι: α) να διασωθεί η κατάσταση της τρέχουσας διαδικασίας η οποία διακόπτεται, (β) ο καθορισμός της πηγής της διακοπής και (γ) η μεταβίβαση του ελέγχου στον αντίστοιχο χειριστή διακοπής για την εξυπηρέτηση της διακοπής. Αυτές οι λειτουργίες είναι το μέρος του λειτουργικού συστήματος που ονομάζεται «χειριστής διακοπών πρώτου επιπέδου». Όταν η εξυπηρέτηση της διακοπής ολοκληρωθεί, τότε η CPU παραχωρείται στη διαδικασία που διακόπηκε λόγω της διακοπής, αν η δρομολόγηση που χρησιμοποιεί το λειτουργικό σύστημα είναι μη επανεισαγωγή, ή στη διαδικασία που βρίσκεται πρώτη στην ουρά των έτοιμων διαδικασιών, αν η δρομολόγηση είναι επανεισαγωγή.

Τέλος, στην Ενότητα 2.6 μελετάμε τον πυρήνα ενός λειτουργικού συστήματος, που είναι ουσιαστικά το μέρος του λειτουργικού συστήματος που υλοποιεί όλες τις λειτουργίες που αφορούν διαδικασίες. Οι κυριότερες από αυτές είναι: ο χειρισμός διακοπών, η δημιουργία και η καταστροφή διαδικασιών, η δρομολόγηση, η νάρκωση και η αφύπνιση διαδικασιών, ο συγχρονισμός (συντονισμός) και η επικοινωνία διαδικασιών, η υποστήριξη δραστηριοτήτων εισόδου/εξόδου, καθώς και η υποστήριξη δέσμευσης και αποδέσμευσης μνήμης και άλλες.

Βιβλιογραφία κεφαλαίου

A) ΠΡΟΑΙΡΕΤΙΚΗΣ ΑΝΑΓΝΩΣΗΣ

Andrews and Schneider, *Concepts and Notations for Concurrent Programming*, Computing Surveys, vol. 15, March 1983, pp. 3 – 43.

Ben–Ari, *Principles of Concurrent Programming*, Englewood Cliffs, NJ: Prentice Hall International, 1982.

Dubois et al., *Synchronization, Coherence, and Event Ordering in Multiprocessors*, IEEE Computers, vol. 21, Feb. 1988, pp. 9 – 21.

B) ΣΥΜΠΛΗΡΩΜΑΤΙΚΗ

Silberschatz et al., *Operating System Concepts*, 3rd edition, reading, MA: Addison–Wesley, 1991.

Γλωσσάρι κεφαλαίου

Blocked process:	μπλοκαρισμένη διαδικασία
Dispatching:	δρομολόγηση
Dispatcher:	αποστολέας
Hardware:	υλικό
Interrupt:	διακοπή
Kernel:	πυρήνας
Maintainable:	επιδεκτικό συντήρησης
Mode:	ρυθμός
nonpreemptive:	μη επανεισαγώγιμη
Preemptive:	επανεισαγώγιμη
Process Control Block (PCB):	μπλοκ ελέγχου διαδικασίας
Process descriptor:	περιγραφέας διαδικασίας
Queue:	ουρά
Quantum:	κβάντο ή διάστημα χρόνου
Ready or runnable process:	έτοιμη ή τρέξιμη διαδικασία
Resource:	πόρος
Resume:	αφύπνιση
Running process:	τρέχουσα διαδικασία
Software:	λογικό
State vector:	διάνυσμα κατάστασης
Status flag:	σημείο κατάστασης
Supervisor call:	κλήση επιβλέποντος
Supervisor mode:	ρυθμός επιβλέποντος
Suspend:	νάρκωση
User mode:	ρυθμός χρήστη
Volatile environment:	ευάλωτο περιβάλλον



Συντονισμός Διαδικασιών

Σκοπός

Σκοπός αυτού του κεφαλαίου είναι η κατανόηση του τρόπου εκτέλεση των διαδικασιών σε ένα σύστημα πολυπρογραμματισμού, των προβλημάτων που παρουσιάζονται σε αυτή την περίπτωση, καθώς και των λύσεων που έχουν δοθεί μέσω διαφόρων τεχνικών. Οι διαδικασίες σε ένα τέτοιο σύστημα θα θεωρούμε ότι εκτελούνται παράλληλα, με αποτέλεσμα να παρουσιάζονται διάφορα προβλήματα κατά την εκτέλεσή τους. Το βασικότερο πρόβλημα είναι ότι δεν είναι ντετερμινιστικά τα αποτελέσματά τους, κι αυτό συμβαίνει γιατί χρησιμοποιούν κοινούς πόρους ταυτόχρονα. Η χρήση των πόρων αυτών γίνεται έτσι ώστε οι διαδικασίες που βρίσκονται στο σύστημα να έχουν τη δυνατότητα να επικοινωνούν μεταξύ τους.

Ένας από τους κυριότερους στόχους αυτού του μαθήματος είναι να κατανοήσετε το πρόβλημα του αμοιβαίου αποκλεισμού (διαδικασιών από την ταυτόχρονη χρήση πόρων) και τις λύσεις που δόθηκαν γι' αυτό. Με την παρουσίαση διαφόρων εργαλείων τα οποία χρησιμοποιούνται για να υπάρχει η δυνατότητα στις διαδικασίες ενός συστήματος να επικοινωνούν και να συντονίζονται μεταξύ τους, πιστεύουμε ότι θα επιτύχουμε το στόχο μας.

Η επικοινωνία αυτή και ο συντονισμός των διαδικασιών γίνεται κυρίως μέσω της χρήσης κοινών πόρων / μεταβλητών. Η πρόσβαση κάθε διαδικασίας στους πόρους αυτούς πρέπει να προστατεύεται. Με τεχνικές όπως η μεταβίβαση μηνυμάτων και οι σημαφόροι, το ΛΣ εξασφαλίζει ότι ανά πάσα στιγμή το πολύ μία διαδικασία έχει πρόσβαση στους κοινούς πόρους. Η έννοια του «συντονισμού» (ή «συγχρονισμού»: *synchronization*) είναι κεντρική όχι μόνο στα λειτουργικά συστήματα αλλά σε όλη την επιστήμη των υπολογιστών, και οι έννοιες και οι τεχνικές που θα αναπτυχθούν στο κεφάλαιο αυτό έχουν τα αντίστοιχά τους σε περιοχές όπως οι βάσεις δεδομένων και η αρχιτεκτονική υπολογιστών.

Προσδοκώμενα αποτελέσματα

Με τη μελέτη του κεφαλαίου αυτού θα είστε σε θέση να:

- Περιγράψετε ποια είναι η λύση του Dekker στο πρόβλημα του αμοιβαίου αποκλεισμού και γιατί πετυχαίνει το σκοπό της.
- Ορίσετε τι είναι οι αναμείξεις των ακολουθιών των εντολών διαδικασιών που εκτε-

λούνται ταυτόχρονα και τότε προκαλούν προβλήματα.

- Εξηγήστε τι είναι ο συντονισμός διαδικασιών και γιατί χρειάζεται.
- Εξηγήστε τι είναι ο κανόνας αμοιβαίου αποκλεισμού.
- Αναφέρετε τις τρεις συνθήκες που πρέπει να ικανοποιούνται από μια ρεαλιστική λύση του προβλήματος του αμοιβαίου αποκλεισμού. Επίσης, πρέπει να είστε σε θέση να καταλαβαίνεται γιατί αυτές οι συνθήκες είναι ικανές και απαραίτητες σε αυτή την περίπτωση.
- Περιγράψτε πώς λειτουργούν οι καταναμημένοι αλγόριθμοι αμοιβαίου αποκλεισμού.
- Ορίστε τι είναι ένα μήνυμα και ποια η έννοια των εντολών *send* και *receive*.
- Ορίστε τι είναι μια σημαφόρος, ποιο σκοπό εξυπηρετεί και ποια η λειτουργία των εντολών $P(u)$ [*wait(u)*] και $V(u)$ [*signal(u)*]; Ποιες είναι οι διαφορές των γενικών σημαφόρων από τους δυαδικούς σημαφόρους;
- Εξηγήστε τι είναι η συνθήκη συντονισμού και πώς ελέγχεται η είσοδος διαδικασιών στις κρίσιμες περιοχές χρησιμοποιώντας αυτό το εργαλείο, καθώς και ποιες λειτουργίες μπορούν να οριστούν σε συνθήκες συντονισμού.
- Εξηγήστε ποια η έννοια της ακόλουθης γλωσσικής έκφρασης «*region b do*» και γιατί μπορούμε να γράψουμε πολύ απλό κώδικα συντονισμού χρησιμοποιώντας την. Τι υποθέτουμε για να έχουμε το πιο πάνω πλεονέκτημα με τη χρήση της;
- Εξηγήστε τι είναι τα εργαλεία συντονισμού: *monitor* και ουρές γεγονότων, πώς χρησιμοποιούνται και ποια τα ιδιαίτερα χαρακτηριστικά τους.
- Εξηγήστε ποια η έννοια των διακοπών *block* και *wakeup* και τότε χρησιμοποιούνται.
- Ορίστε πότε δύο διαδικασίες λέγονται παράλληλες και σε ποια περίπτωση οι εντολές γλώσσας μηχανής που τους αντιστοιχούν μπορούν πράγματι να επικαλύπτονται χρονικά.
- Ορίστε τι είναι η πολυάσχολη αναμονή.

Έννοιες κλειδιά

- Συνεργασία διαδικασιών
- Συντονισμός διαδικασιών
- Ταυτόχρονη εκτέλεση διαδικασιών

- *Ανάμειξη των εντολών διαδικασιών που εκτελούνται ταυτόχρονα*
- *Cobegin, coend*
- *Αμοιβαίος αποκλεισμός*
- *Πολυάσχολη αναμονή*
- *Απεριόριστη αναβολή*
- *Αδιέξοδο*
- *Κρίσιμη περιοχή*
- *Μήνυμα*
- *Send, receive*
- *Σημαφόρος*
- *Wait, signal*
- *Συνθήκη συντονισμού*
- *Await*
- *Test and set*
- *Monitor*
- *Ουρές γεγονότων*
- *Block, wakeup*

Δομή κεφαλαίου

Το κεφάλαιο αυτό περιέχει τις παρακάτω ενότητες:

- 3.1 Ορισμοί
- 3.2 Εντολές Παραλληλισμού
- 3.3 Η Ανάγκη Συντονισμού
- 3.4 Κρίσιμες Περιοχές
- 3.5 Η Φύση του Προβλήματος του Αμοιβαίου Αποκλεισμού
- 3.6 Γενικές Παρατηρήσεις
- 3.7 Συνεργασία Διαδικασιών
- 3.8 Συντονισμός Διαδικασιών που Δε Συνεργάζονται
- 3.9 Κατανεμημένοι Αλγόριθμοι για Αμοιβαίο Αποκλεισμό

3.1 Ορισμοί

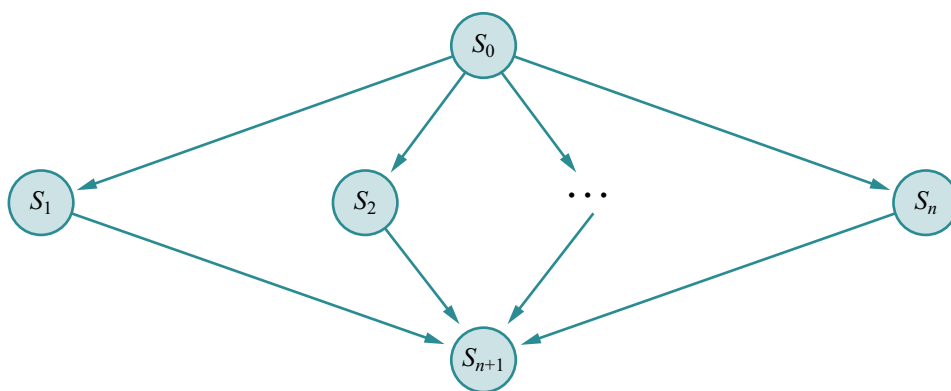
Στη συνέχεια υποθέτουμε ότι, όταν ξεκινήσει η εκτέλεση μιας εντολής, θα ολοκληρωθεί σε πεπερασμένο χρόνο και θα παραγάγει αποτέλεσμα το οποίο είναι μια συνάρτηση της εισόδου και μόνο (δηλ. των παραμέτρων της εντολής) και δεν εξαρτάται από την πιθανή παρουσία άλλων εντολών στο σύστημα.

Δύο διαδικασίες λέγονται «παράλληλες» (concurrent) όταν οι εκτελέσεις τους επικαλύπτονται χρονικά. Σε περιβάλλον πολυεπεξεργαστών, οι εντολές γλώσσας μηχανής δύο παραλλήλων διαδικασιών μπορεί να επικαλύπτονται πράγματι χρονικά (true parallelism). Αλλά σε περιβάλλον ενός μόνο επεξεργαστή, μια εντολή γλώσσας μηχανής κάποιας διαδικασίας μπορεί μόνο να «παρεμβληθεί» ανάμεσα σε δύο εντολές άλλων διαδικασιών. Εδώ γενικά υποθέτουμε ότι όλες οι δυνατές αναμειξίες των ακολουθιών των εντολών είναι δυνατόν να συμβούν (indeterminism). Έχει παρατηρηθεί ότι σε όλες τις πρακτικές περιπτώσεις μπορούμε να χρησιμοποιούμε την ιδέα του indeterminism ως υπόθεση εργασίας. Τα λογικά αποτελέσματα ή προβλήματα φαίνονται να είναι ίδια και στις δύο περιπτώσεις (είτε κάνουμε την παραπάνω υπόθεση εργασίας είτε όχι). Τα προβλήματα αυτά προκαλούνται από την άγνοιά μας για τις σχετικές ταχύτητες των παράλληλων διαδικασιών. Στην περίπτωση όπου υπάρχει μόνο ένας επεξεργαστής στο σύστημα, θα χρησιμοποιούμε και τον όρο «ταυτόχρονα», για να δηλώσουμε ότι κάποιες διαδικασίες εκτελούνται «παράλληλα», με βάση τον πιο πάνω ορισμό των παραλλήλων διαδικασιών.

3.2 Εντολές Παραλληλισμού

Η εντολή (γλωσσικό σχήμα)

cobegin $s_1 ; s_2 ; \dots ; s_n$ coend



Σχήμα 3.1

Παράλληλες
διαδικασίες

ή

parbegin $s_1 ; s_2 ; \dots ; s_n$ parend

υποδηλώνει ότι οι κώδικες s_1, s_2, \dots, s_n εκτελούνται «παράλληλα» (γενικά, όχι αναγκαστικά με την ίδια ταχύτητα). Για παράδειγμα, το πρόγραμμα s_0 ; *cobegin* s_1 ; s_2 ; ... ; s_n *coend* ; s_{n+1} έχει το γράφημα προτεραιότητας του Σχήματος 3.1. Επομένως, η σειρά με την οποία θα εκτελεστούν οι επιμέρους εντολές είναι άγνωστη ή, αλλιώς, μπορεί να εκτελεστούν σε οποιαδήποτε σειρά, αν δεν πάρει περιοριστικά μέτρα το ΛΣ. Στην επόμενη ενότητα θα δούμε γιατί είναι αναγκαία κάποια μέτρα περιορισμού που δε θα επιτρέπουν όλες τις δυνατές σειρές εκτέλεσης.

3.3 Η ανάγκη συντονισμού

Έστω ένα σύστημα στο οποίο εκτελούνται ταυτόχρονα δύο διαδικασίες κι ας υποθέσουμε ότι οι διαδικασίες αυτές έχουν κοινές μεταβλητές. Όταν μια από τις διαδικασίες αυτές αλλάζει μια ή περισσότερες από τις κοινές μεταβλητές, τότε το αποτέλεσμα που παράγεται από τη δεύτερη διαδικασία επηρεάζεται από τη σχετική ταχύτητα των δύο διαδικασιών.

Για παράδειγμα, θεωρήστε ότι οι διαδικασίες P_1, P_2 καταγράφουν τη θέση του cursor όταν αυτή αλλάζει ως αποτέλεσμα της μετακίνησής του με το ποντίκι (P_1) ή με χρήση του πληκτρολογίου (δηλαδή μετά από γράψιμο, σβήσιμο ή μετακίνηση με τα βελάκια) (P_2). Τέτοιες διαδικασίες χρησιμοποιούνται σε «επεξεργασία κειμένου με αλληλεπίδραση» (interactive text processing). Η καθεμία από τις διαδικασίες P_i ($i = 1, 2$) εκτελεί τους ακόλουθους κώδικες για τις διαδικασίες P_1 και P_2 [θεωρούμε ότι διαδικασίες P_i ($i = 1, 2$) αναφέρονται στο ίδιο τερματικό, έστω T]:

<u>BEGIN P_1</u>	<u>BEGIN P_2</u>
Αν αλλάξει η θέση του cursor στο T , τότε	Αν αλλάξει η θέση του cursor στο T , τότε
1. διάβασε τη θέση του cursor	1. διάβασε τη θέση του cursor
στη μεταβλητή χ_1	στη μεταβλητή χ_2
2. γράψε την τιμή της χ_1 στην	2. γράψε την τιμή της χ_2 στην
CURSORPOSITION	CURSORPOSITION
<u>END</u>	<u>END</u>

Εδώ CURSORPOSITION είναι μια μεταβλητή (κοινή για τις δύο διαδικασίες) και αντιστοιχεί στη θέση του cursor στο τερματικό T . Το χ_i είναι (τοπική) μεταβλητή της διαδικασίας P_i .

Έστω ότι μια τρέχουσα τιμή της μεταβλητής CURSORPOSITION είναι, π.χ., 653. Για

να προσδιορίσουμε τη θέση του cursor στην οθόνη, πρέπει να ξέρουμε σε πόσες θέσεις, έστω X , μπορεί να μετακινηθεί σε μια γραμμή. Τότε, με το ακέραιο πηλίκο της τιμής της μεταβλητής CURSORPOSITION και του X θα βρούμε τη γραμμή στην οποία βρίσκεται και με το $\text{CURSORPOSITION} \bmod X$ τη θέση του στη γραμμή. Έστω ότι ο cursor μετακινείται με το ποντίκι στη θέση 342, άρα η διαδικασία P_1 θα αλλάξει την τιμή της μεταβλητής CURSORPOSITION σε 342. Παράλληλα, ένα καινούριο γράμμα γράφεται σε ένα κείμενο, άρα η διαδικασία P_1 θα αλλάξει την τιμή της μεταβλητής CURSORPOSITION σε 654. Ας θεωρήσουμε ότι γράφεται το τελευταίο γράμμα σε ένα κείμενο και έπειτα μετακινείται ο cursor στην αρχή της τελευταίας σελίδας (π.χ. για να γίνει μια διόρθωση στη συνέχεια). Με βάση αυτή την περιγραφή, θα περιμέναμε ότι η τελική τιμή της μεταβλητής CURSORPOSITION θα ήταν 342. Έστω η ακόλουθη χρονική σειρά εκτέλεσης των αναμειγμένων εντολών των διαδικασιών P_1 και P_2 :

- P_2 : διαβάζει τη θέση του cursor στη χ_2 ($\chi_2 = 654$)
- Εδώ η P_2 χάνει την CPU, γιατί τελειώνει ο στοιχειώδης χρόνος δρομολόγησής της. Η P_1 αρχίζει να τρέχει.
- P_1 : διαβάζει τη θέση του cursor στη χ_1 ($\chi_1 = 342$)
- P_1 : γράφει την τιμή της χ_1 στην CURSORPOSITION (CURSORPOSITION = 342)
- Εδώ η P_1 χάνει την CPU, γιατί τελειώνει ο στοιχειώδης χρόνος δρομολόγησής της. Η P_2 αρχίζει να τρέχει.
- P_2 : γράφει την τιμή της χ_2 στην CURSORPOSITION (CURSORPOSITION = 654)

Το αποτέλεσμα της παραπάνω σειράς γεγονότων είναι να μην καταγραφεί η σωστή τιμή στη μεταβλητή CURSORPOSITION! (Το CURSORPOSITION γίνεται τελικά 654 αντί για 342).

Άσκηση Αυτοαξιολόγησης 3.1

Η διαδικασία total αυξάνει στην καθολική κοινή μεταβλητή tally κατά μία μονάδα, όπως παρουσιάζεται στη συνέχεια:

<pre> Const n = 50; var tally : shared integer; procedure total var count : integer; begin for count := 1 to n do tally := tally + 1; </pre>	<pre> begin (* main program *) tally := 0; cobegin total; total; </pre>
--	---

end

coend;
writeIn(tally)

end

Προσδιορίστε την ελάχιστη και τη μέγιστη τιμή που μπορεί να έχει η μεταβλητή tally μετά το τέλος της εκτέλεσης του παραπάνω προγράμματος, όπου καλείται δύο φορές ταυτόχρονα η διαδικασία total.

Λάθη σαν τα παραπάνω έχουν το εξής χαρακτηριστικό: Επειδή εξαρτώνται από τις σχετικές ταχύτητες των διαδικασιών, γενικά δεν επαναλαμβάνονται / αναπαράγονται (δηλαδή δεν είναι το ίδιο αποτέλεσμα κάθε φορά που τρέχουν παράλληλα οι ίδιες διαδικασίες, άρα είναι μη ντετερμινιστικό το αποτέλεσμα). Αυτό κάνει πιο δύσκολα τα πράγματα, με αποτέλεσμα να μην μπορεί ο προγραμματιστής να τα ανιχνεύσει εύκολα και να τα διορθώσει! Ουσιαστικά, η διόρθωση τέτοιων προβλημάτων είναι σχεδόν αδύνατη.

Βλέπουμε, λοιπόν, ότι, όταν παράλληλες διαδικασίες αλληλεπιδρούν (μέσω κοινών μεταβλητών), πρέπει να ασκηθεί κάποιος έλεγχος στην αλληλεπίδρασή τους. Θα καλούμε «συντονισμό» (συγχρονισμό) γενικά κάθε περιορισμό στη σχετική διάταξη των εντολών (των διαδικασιών που αλληλεπιδρούν) στο χρόνο. Οι κοινές μεταβλητές γενικά παριστάνουν κοινούς «πόρους», όπως, π.χ., φυσικούς πόρους (μνήμη, I/O εργαλεία) ή μηνύματα.

Στο προηγούμενο παράδειγμα η CURSORPOSITION ήταν ο «πόρος». Το λάθος έγινε επειδή οι P_1 και P_2 χρησιμοποίησαν το CURSORPOSITION «ταυτόχρονα» (ενώ η P_1 άρχισε να ασχολείται με την κοινή μεταβλητή, η P_2 παρεμβλήθηκε και άλλαξε την τιμή της). Το λάθος δε θα γινόταν αν υπήρχε ο περιορισμός του τύπου «εφόσον μια διαδικασία έχει πρόσβαση σε μία μεταβλητή, οι άλλες διαδικασίες δεν μπορούν να έχουν προσπέλαση στη μεταβλητή αυτή». Αυτός ο περιορισμός (κανόνας συντονισμού) ονομάζεται «κανόνας αμοιβαίου αποκλεισμού» (mutual exclusion rule).

3.4 Κρίσιμες περιοχές

Αν η μεταβλητή u παριστάνει κάποιον κοινό πόρο, δηλαδή χρησιμοποιείται από πολλές διαδικασίες, και επομένως υπάρχει ανάγκη συντονισμού τους, στις προηγούμενες ενότητες υποθέσαμε ότι όλες οι εντολές που αφορούν τη u βρίσκονται σε ένα μέρος του κώδικα που το λέγαμε «χρήση του u » και στο οποίο εξασφαλίσαμε τον αμοιβαίο αποκλεισμό των διαδικασιών, δηλαδή την εκτέλεση του κώδικα χωρίς διακοπή / παρεμβολή από άλλη διαδικασία (ατομική εκτέλεση). Τώρα θα γενικεύσου-

με την έννοια αυτή: Δεδομένου ενός κοινού πόρου u τύπου T , ονομάζουμε *κρίσιμη περιοχή* (ως προς u) ένα μέρος κώδικα για το οποίο εξασφαλίζεται η ατομική του εκτέλεση από τις διάφορες διαδικασίες που το χρειάζονται. Η έννοια της «κρίσιμης περιοχής» (critical region) κωδικοποιεί τις απαιτήσεις του αμοιβαίου αποκλεισμού:

Ας θεωρήσουμε τη γλωσσική δήλωση

`var u : shared T ;`

που σημαίνει ότι η μεταβλητή u είναι κοινή (μπορεί να προσπελαστεί από περισσότερες της μιας διαδικασίες). T είναι ο τύπος της u (π.χ. INTEGER, QUEUE, LIST κτλ.)

και

`region u do S ;`

που ορίζει ότι ο κώδικας S είναι κρίσιμη περιοχή της u .

Για τις κρίσιμες περιοχές ισχύουν τα εξής:

- Το πολύ μια διαδικασία μπορεί να βρίσκεται σε κρίσιμη περιοχή (που αναφέρεται στη u) κάθε στιγμή.
- Όταν μια διαδικασία περιμένει για να εισέλθει στην κρίσιμη περιοχή, τότε αυτό θα συμβεί σε πεπερασμένο χρόνο.
- Μια διαδικασία δεν μπορεί να μένει στην κρίσιμη περιοχή απεριόριστα.

3.5 Η Φύση του προβλήματος του Αμοιβαίου Αποκλεισμού

Ας θεωρήσουμε δύο κυκλικές διαδικασίες (οι διαδικασίες αυτές επαναλαμβάνουν συνέχεια ένα κομμάτι κώδικα) P και Q , που συχνά χρησιμοποιούν τον κοινό πόρο (μεταβλητή) R . Η καθεμία από τις διαδικασίες P και Q χρησιμοποιούν τον R μόνο για περιορισμένο χρόνο (κάθε φορά που έχουν προσπέλαση στον R). Το πρόβλημα που εισάγει ο αμοιβαίος αποκλεισμός είναι ο τρόπος με τον οποίο θα εξασφαλιστεί ο ακόλουθος περιορισμός. Οι διαδικασίες P και Q δεν πρέπει να χρησιμοποιήσουν τον κοινό πόρο R «ταυτόχρονα». Αυτό ορίζει τον κανόνα του αμοιβαίου αποκλεισμού, όπως αναφέραμε σε προηγούμενη ενότητα.

Πώς θα λύσουμε το πρόβλημα; Ας δοκιμάσουμε πρώτα να συντονίσουμε τις διαδικασίες P και Q μέσω μιας κοινής μεταβλητής «free» που δείχνει αν ο πόρος είναι διαθέσιμος. Αν δεν είναι, (free = False), η διαδικασία περιμένει. Αν είναι διαθέσιμος (free = True), η διαδικασία τον μαρκάρει ως μη διαθέσιμο και τον χρησιμοποιεί. Όταν τελειώσει, πάλι μέσω της μεταβλητής free, δηλώνει ότι ο πόρος είναι ελεύθερος.

Λύση 1

```

var free : shared boolean ;
  begin
    free := true ;
    cobegin
      “P” repeat
repeat until free;
  free := false ;
  [χρήση του R]
  free := true ;
  forever
      “Q” repeat
repeat until free;
  free := false ;
  [χρήση του R]
  free := true ;
  forever
    coend
  end

```

Αρχικά η *free* είναι αληθής. Συνεπώς είναι δυνατόν η P και η Q να αναφερθούν «ταυτόχρονα» στη *free* και να την βρουν αληθή. (Αυτό, π.χ., θα γίνει εάν οι P και Q εκτελέσουν το «repeat until *free*» η μία μετά την άλλη). Αν γίνει αυτό, και οι δύο διαδικασίες θα κάνουν τη *free* ψευδή (η μία μετά την άλλη) και θα χρησιμοποιήσουν την R ταυτόχρονα! (Δηλαδή η «λύση» έχει λάθος.)

Ας δοκιμάσουμε μια δεύτερη λύση: κάθε διαδικασία θα παίρνει τον πόρο εναλλάξ, με τη σειρά της. Αντί της *free*, θα χρησιμοποιήσουμε τη μεταβλητή P_{turn} , που δείχνει αν είναι η σειρά της P να χρησιμοποιήσει τον R. Αυτό συμβαίνει όταν $P_{\text{turn}} = \text{True}$ (και είναι False όταν είναι η σειρά της Q).

Λύση 2

```

var  $P_{\text{turn}}$  : shared boolean ;
  begin
     $P_{\text{turn}}$  := true;
    cobegin
      “P” repeat
repeat until  $P_{\text{turn}}$  ;
  [χρήση του R]
   $P_{\text{turn}}$  := false ;
  forever
      “Q” repeat
repeat until (not  $P_{\text{turn}}$ ) ;
  [χρήση του R]
   $P_{\text{turn}}$  := true ;
  forever
    coend
  end

```

coend

end

Η Λύση 2 πετυχαίνει αμοιβαίο αποκλεισμό, επειδή η P_{turn} μπορεί να είναι μόνο true (ή μόνο false) σε μια δεδομένη χρονική στιγμή. Αλλά η λύση αυτή αναγκάζει το ΛΣ να παρέχει τον πόρο R στις P και Q εναλλάξ, ανεξάρτητα από τις ανάγκες τους, και γι' αυτό δεν είναι αποδεκτή. Αν, π.χ., είναι η σειρά της P να χρησιμοποιήσει τον R, αλλά δεν τον χρειάζεται ακόμη επειδή έχει να κάνει ένα μεγάλο υπολογισμό, τότε η Q περιμένει χωρίς λόγο, ενώ ο R μένει σε αχρηστία. Ένα άλλο ανεπιθύμητο χαρακτηριστικό της λύσης αυτής είναι η σπατάλη χρόνου που γίνεται στα μη παραγωγικά loops (στα «repeat until...»), κατάσταση που ονομάζεται «πολύασχολη αναμονή» (busy waiting).

Άσκηση Αυτοαξιολόγησης 3.2

Μπορείτε να σκεφτείτε κάποιον τρόπο έτσι ώστε να αποφεύγεται η πολύασχολη αναμονή;

Ας δοκιμάσουμε τώρα να αποφύγουμε την αυστηρή εναλλαγή, χρησιμοποιώντας δύο boolean μεταβλητές οι οποίες δείχνουν ποιες διαδικασίες έχουν πρόθεση να χρησιμοποιήσουν τον κοινό πόρο. Η λύση αυτή λέει: «Πριν χρησιμοποιήσεις τον κοινό πόρο (α), δήλωσε την πρόθεση σου (β): αν κανείς άλλος δεν έχει δηλώσει όμοια πρόθεση, τότε χρησιμοποίησε τον πόρο και, όταν τελειώσεις, ανακάλεσε την πρόθεση χρήσης».

Λύση 3

```

var  $P_{turn}$ ,  $Q_{turn}$  : shared boolean ;

begin
   $P_{turn}$  : false ;  $Q_{turn}$  := false ;
cobegin

  “P” repeat
     $P_{turn}$  := true ;
  repeat until (not  $Q_{turn}$ ) ;
  [χρήση της R] ;
   $P_{turn}$  := false ;
  ...

  “Q” repeat
     $Q_{turn}$  := true ;
  repeat until (not  $P_{turn}$ ) ;
  [χρήση της R] ;
   $Q_{turn}$  := false ;
  ...

```

foreverforever

coend

end

Άσκηση Αυτοαξιολόγησης 3.3

Η πρώτη «λύση» σε ελεύθερο λόγο έλεγε: «Χρησιμοποιούμε τη μεταβλητή “κράτηση του πόρου R”. Αν ο πόρος δεν είναι κρατημένος (σύμφωνα με τη μεταβλητή), οποιαδήποτε διαδικασία τον χρειάζεται μπορεί να τον “καπαρώσει”, τον χρησιμοποιεί όσο θέλει και μετά τον ελευθερώνει δηλώνοντας τη μεταβλητή “ελεύθερος”». Ο λόγος για τον οποίο η λύση αυτή είναι λάθος, πάλι σε ελεύθερο λόγο είναι ο εξής: «Μεταξύ της ερώτησης “Ελεύθερος;” της απάντησης “Ναι” και της δήλωσης “Σε καπαρώνω” από τη διαδικασία 1 μπορεί να παρεμβληθούν οι ίδιες ερωτήσεις από τη διαδικασία 2 ως εξής:

Ελεύθερος;

Ναι.

_____ Ελεύθερος;

_____ Ναι.

Σε καπαρώνω.

_____ Σε καπαρώνω.

Αυτό έχει ως αποτέλεσμα την παράλληλη χρήση του πόρου από τις δύο διαδικασίες».

Περιγράψτε με ανάλογο τρόπο (ελεύθερο λόγο) τη Λύση 2, με τις δύο μεταβλητές P_{turn} , Q_{turn} , καθώς και τα προβλήματα που θα προκαλούσε η υιοθέτησή της.

Άσκηση Αυτοαξιολόγησης 3.4

Περίγραψε με ελεύθερο λόγο τη Λύση 3. Είναι πραγματική λύση ως προς το πρόβλημα του αμοιβαίου αποκλεισμού; Μήπως δημιουργεί άλλες παρενέργειες;

Το πρόβλημα του αμοιβαίου αποκλεισμού λύθηκε για πρώτη φορά από τον Dekker το 1965.

Λύση 4 (Dekker)

```

var outside 1, outside 2 : shared boolean ;
    turn : shared 1...2 ;
begin
    outside 1 : true ; outside 2 := true ; turn := 1 ;
cobegin
    “P” repeat
begin
    repeat
        outside 1:= false ;
    repeat
        if outside 2 then go to enter ;
    until turn = 2 ;
    outside 1:= true ;
    repeat until turn = 1 ;
    forever ;
end
enter P inside ;
[χρήση του R] ;
turn:= 2 ;
outside 1:= true ;
P outside ;
forever
coend
end
    “Q” repeat
begin
    repeat
        outside 2:= false ;
    repeat
        if outside 1 then go to enter ;
    until turn = 1 ;
    outside 2:= true ;
    repeat until turn = 2 ;
    forever ;
end
enter Q inside ;
[χρήση του R] ;
turn:= 1 ;
outside 2:= true ;
Q outside ;
forever
end
end

```

Η λύση του Dekker πετυχαίνει αμοιβαίο αποκλεισμό στη χρήση της R. Ο Dijkstra γενίκευσε τη λύση για n διαδικασίες.

Οδηγία προς τον αναγνώστη:

Η λύση του Dekker είναι το πολυπλοκότερο θέμα που έχετε αντιμετωπίσει μέχρι σήμερα στα πλαίσια του βιβλίου αυτού και –ίσως– και σε όλη τη Θεματική Ενότητα. Πρέπει να δώσετε αρκετό χρόνο για να την καταλάβετε. Μην προχωρήσετε χωρίς να τη μελετήσετε και να λύσετε την άσκηση αυτοαξιολόγησης που ακο-

λουθεί. Είναι σκόπιμο να επανέλθετε στον αλγόριθμο αυτό αργότερα, όταν, για παράδειγμα, θα έχετε ολοκληρώσει τη μελέτη του κεφαλαίου και να προσπαθήσετε να τον αναπαραγάγετε όχι αυτολεξεί αλλά με το δικό σας τρόπο.

Άσκηση Αυτοαξιολόγησης 3.5

Προσθέστε σε κάθε γραμμή ψευδοκώδικα σχόλια τύπου:

outside 1 := false /* Η διαδικασία 1 δηλώνει την πρόθεσή της να χρησιμοποιήσει τον πόρο R */.

If outside 2 then goto enter /* Η διαδικασία 1 ελέγχει αν η διαδικασία 2 δε χρησιμοποιεί ούτε έχει δηλώσει την πρόθεσή της να χρησιμοποιήσει τον πόρο R. Στην περίπτωση αυτή παίρνει τη χρήση του πόρου.

Ποιες τιμές μπορεί να έχουν οι μεταβλητές outside1, outside2 και turn όταν τον πόρο τον χρησιμοποιεί η διαδικασία 1; Ποιες τιμές όταν τον χρησιμοποιεί η διαδικασία 2; Ποιες όταν δεν τον χρησιμοποιεί καμιά; Αν η μεταβλητή turn δείχνει «τίνος σειρά είναι», τότε πώς μπορεί να χρησιμοποιήσει τον πόρο για πρώτη φορά η διαδικασία 2;

Γράψτε τον αλγόριθμο με ψευδοκώδικα σε πρώτο πρόσωπο από τη μεριά της διαδικασίας. Για παράδειγμα, το εσωτερικό loop θα μπορούσε να διατυπωθεί:

ΠΑΙΡΝΩ ΤΟΝ ΠΟΡΟ (enter)

ΤΟΝ ΧΡΗΣΙΜΟΠΟΙΩ. ΤΕΛΕΙΩΣΑ.

ΔΙΝΩ ΤΗ ΣΕΙΡΑ ΜΟΥ ΣΤΗΝ ΑΛΛΗ ΔΙΕΡΓΑΣΙΑ (turn = 2)

ΔΗΛΩΝΩ ΟΤΙ ΔΕ ΘΑ ΤΟΝ ΧΡΗΣΙΜΟΠΟΙΗΣΩ ΑΛΛΟ (out 1 = True)

ΑΦΗΝΩ ΤΟΝ ΠΟΡΟ.

Πώς τελικά εξασφαλίζεται ότι οι δύο διεργασίες δε χρησιμοποιούν «παράλληλα» τον πόρο; Τι ακριβώς σημαίνει ότι δεν τον χρησιμοποιούν παράλληλα;

Άσκηση Αυτοαξιολόγησης 3.6

Πρώτος ο Dijkstra πρότεινε την έννοια της κρίσιμης περιοχής και ο Hansen (1972) πρότεινε τη γλωσσική έκφραση «region u do S».

A. Πώς μπορεί να χρησιμοποιήσει ο compiler μια τέτοια γλωσσική έκφραση;

B. Αντικαταστήστε τη λύση Dekker με κώδικα που χρησιμοποιεί κρίσιμη περιο-

χή. Παρατηρήστε πόσο πιο απλό είναι. Βέβαια, η εντολή `region u do S` προϋποθέτει τη λύση Dekker (ή κάτι ισοδύναμο) για την υλοποίησή της.

Γ. Σκεφτείτε και διατυπώστε μια παράγραφο για το αν εντέλει η έννοια της κρίσιμης περιοχής συνιστά απλοποίηση της λύσης Dekker και με ποιον τρόπο.

Απάντηση:

A. (Έτσι, ο compiler μπορεί να ελέγξει την ορθή προσπέλαση των κοινών μεταβλητών.)

B. Π.χ. ο παρακάτω κώδικας μπορεί να αντικαταστήσει τη λύση Dekker:

```
var R : shared T ;
```

<p>“P”</p> <p><u>repeat</u></p> <p>region R do [χρήση του R];</p> <p><u>forever</u></p>	<p><u>cobegin</u></p> <p>“Q”</p> <p><u>repeat</u></p> <p>region R do [χρήση του R];</p> <p><u>forever</u></p> <p><u>coend</u></p>
---	---

Γ. Η έννοια της κρίσιμης περιοχής δε συνιστά απλοποίηση της λύσης Dekker. Είναι όμως μια αφαίρεση, η οποία μας επιτρέπει να διατυπώσουμε και να λύσουμε συνθετότερα προβλήματα, όπως αυτά που θα δούμε στα επόμενα κεφάλαια. Τέτοιου είδους αφαιρέσεις είναι χαρακτηριστικές της προόδου της πληροφορικής, αλλά και άλλων επιστημών.

Άσκηση Αυτοαξιολόγησης 3.7

Σχολιάστε την ορθότητα της εξής λύσης του προβλήματος του αμοιβαίου αποκλεισμού. (`noop` = εντολή που δεν κάνει τίποτα)

```
var flag: shared array[0..1] of boolean; /* αρχικές τιμές είναι FALSE */
```

```
turn : shared 0..1;
```

Κώδικας για `pi` ($i = 0$ ή 1 . Όταν $i = 0$, $j = 1$, και αντίστροφα).

```
repeat
```

```

    flag[i] = TRUE;
    while (turn != i) do begin
        while (flag[j]) do noop;
        turn := i;
    end
    Κρίσιμη περιοχή
    flag[i] = FLASE;
until FALSE;

```

Το πιο πάνω πρόγραμμα δεν είναι σωστή λύση για το πρόβλημα του αμοιβαίου αποκλεισμού, γιατί υπάρχει τουλάχιστον μια ανάμειξη των εντολών των διαδικασιών p0 και p1, η οποία επιτυγχάνει να βρίσκονται και οι δύο διαδικασίες στην κρίσιμη περιοχή. Η ανάμειξη αυτή έχει ως ακολούθως:

Έστω turn := 1,

```

P0:  flag[0] := true;
P0:  while (turn != 0)
P0:  while (flag[1])
P0:  turn := 0;
P0:  κρίσιμη περιοχή
P0:  flag[0] = false;
P1:  flag[1] := true;
P1:  while (turn != 1)
P1:  while (flag[0])
P0:  flag[0] = true;
P0:  while (turn != 0)
P0:  κρίσιμη περιοχή
P1:  turn = 1;
P1:  κρίσιμη περιοχή ].

```

3.6 Γενικές παρατηρήσεις

Άσκηση Αυτοαξιολόγησης 3.8

Στη Λύση 3 το πρόβλημα του αδιεξόδου δημιουργείται επειδή κάθε διαδικασία δηλώνει την πρόθεσή της να χρησιμοποιήσει τον πόρο ($P_{\text{turn}} = \text{true}$) και περιμένει την άλλη διαδικασία να ανακαλέσει την αντίστοιχη δήλωσή της. Αυτό μοιάζει με δύο ανθρώπους που επιχειρούν να καλέσουν ο ένας τον άλλον ταυτόχρονα και, επειδή βρίσκουν τη γραμμή κατειλημμένη, ξανακαλούν συνεχώς. Αν περίμεναν «λιγάκι» (κι αν αυτό το «λιγάκι» ήταν τυχαίο και διαφορετικό για τον καθένα), τότε, κατά πάσα πιθανότητα, το πρόβλημα θα λυνόταν. Μπορείτε να «διορθώσετε» τον κώδικα της Λύσης 3 με αντίστοιχο τρόπο;

Απάντηση:

Αν αντικαταστήσουμε τον κώδικα «repeat until (not Q_{turn})» της P με τον κώδικα

```
repeat
begin
     $P_{\text{turn}} := \text{false};$ 
    περίμενε για ένα τυχαίο αριθμό βημάτων ;
     $P_{\text{turn}} := \text{true};$ 
end ;
until (not  $Q_{\text{turn}}$ ) ;
```

(και κάνουμε το ίδιο και στον αντίστοιχο κώδικα της Q), τότε έχουμε μια λύση που ικανοποιεί τον αμοιβαίο αποκλεισμό και που δημιουργεί αδιέξοδο μόνο με πολύ μικρή πιθανότητα [πρέπει, βέβαια, η δημιουργία τυχαίων αριθμών στην P (Q) να είναι ανεξάρτητη από τη δημιουργία τυχαίων αριθμών στην Q (P)].

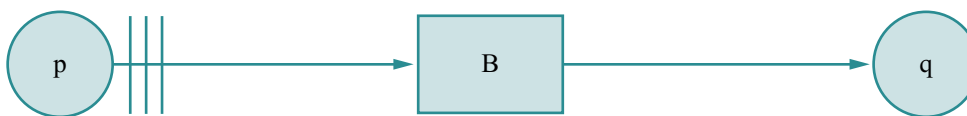
2. Το φαινόμενο στο οποίο μια διαδικασία P εκτελεί μεν χρήσιμο κώδικα, αλλά δεν μπορεί ποτέ να χρησιμοποιήσει τον πόρο R (γιατί άλλες, ταχύτερες, διαδικασίες, προλαβαίνουν κάθε φορά και εμποδίζουν την P) ονομάζεται «απεριόριστη αναβολή» (indefinite postponement). Όταν μια λύση του προβλήματος του αμοιβαίου αποκλεισμού αποφεύγει το indefinite postponement, τότε η λύση λέγεται «δίκαια» (fair).
3. Σε ένα ΛΣ η λύση στο πρόβλημα του αμοιβαίου αποκλεισμού θα πρέπει να ικανοποιεί τα εξής:

- α) Όταν πολλές διαδικασίες ζητούν ταυτόχρονα έναν πόρο, δεν μπορεί όλες να περιμένουν απερίοριστα: κάποια πρέπει να αποκτήσει προσπέλαση σε πεπερασμένο χρόνο.
- β) Όταν μια διαδικασία αποκτήσει τον πόρο, δεν μπορεί να τον κρατήσει απερίοριστα: πρέπει να τον αφήσει κάποτε.
- γ) Καλό είναι να αποφεύγεται η «πολυάσχολη αναμονή».

3.7 Συνεργασία Διαδικασιών (Επικοινωνία και Σημαφόροι)

3.7.1 Επικοινωνία Διαδικασιών

Ας θεωρήσουμε εδώ την περίπτωση της ύπαρξης δύο διαδικασιών P και C , όπου η P παράγει και στέλνει μια ακολουθία δεδομένων στην C , και η C τα παραλαμβάνει και τα χρησιμοποιεί (Producer, Consumer). Τα δεδομένα μεταδίδονται σε διακριτά τμήματα, που ονομάζονται «μηνύματα» (messages). Επειδή οι P και C μπορεί να εργάζονται με διαφορετική ταχύτητα, είναι δυνατόν ο αποστολέας (P) να παράγει ένα μήνυμα σε κάποια στιγμή κατά την οποία ο παραλήπτης (C) δεν μπορεί να παραλάβει (π.χ. επειδή η C επεξεργάζεται ένα παλιότερο μήνυμα εκείνη τη στιγμή). Για να μην καθυστερήσουμε τον αποστολέα, θα εισαγάγουμε στο υπολογιστικό περιβάλλον ένα χώρο αποθήκευσης B , όπου ο αποστολέας θα μπορεί να τοποθετήσει ένα ή περισσότερα μηνύματα. Θα καλούμε τον B «απομονωτή» (buffer) (βλ. Σχήμα 3.2). Ο ρόλος του απομονωτή είναι να εξομαλύνει διαφορές ταχυτήτων ανάμεσα σε δύο διαδικασίες.



Σχήμα 3.2

Η ιδέα του απομονωτή

Βέβαια, ο παραπάνω τρόπος επικοινωνίας πρέπει να ικανοποιεί δύο περιορισμούς:

1. Ο αποστολέας δεν μπορεί να βάλει στο B περισσότερα μηνύματα από την πεπερασμένη χωρητικότητα του B .
2. Ο παραλήπτης δεν μπορεί να παραλαμβάνει μηνύματα με ρυθμό μεγαλύτερο από αυτόν που παράγονται.

Οι δύο αυτοί κανόνες ικανοποιούνται εάν το σύστημα τηρήσει την εξής αρχή συντονισμού:

- Ο αποστολέας που επιχειρεί να βάλει μήνυμα σε *πλήρη* (γεμάτο) απομονωτή θα καθυστερήσει έως ότου ο παραλήπτης παραλάβει κάποιο μήνυμα από τον απομονωτή και
- Ο παραλήπτης που επιχειρεί να παραλάβει μήνυμα από έναν *άδειο* απομονωτή θα καθυστερήσει μέχρις ότου ο αποστολέας βάλει κάποιο μήνυμα στον απομονωτή.

Υποθέτουμε επίσης πως όλα τα μηνύματα παραλαμβάνονται με τη σειρά που στάλθηκαν και με το αρχικό τους περιεχόμενο. Δηλαδή τα μηνύματα δε χάνονται ούτε αλλοιώνονται ούτε χάνουν τη σειρά τους μέσα στο B. Θα θέλαμε, λοιπόν, να έχουμε μια γλωσσική έκφραση του τύπου

var B: buffer max of T;

που ορίζει έναν απομονωτή B, χωρητικότητας max μηνυμάτων, τύπου T.

Οι διαδικασίες που επικοινωνούν μέσω του B θα χρησιμοποιούν τις εντολές

send (M,B);

που στέλνει μήνυμα M στο B, και

receive (M,B);

που παίρνει ένα μήνυμα από το B και το τοποθετεί στο χώρο της τοπικής μεταβλητής M.

Επειδή οι send και receive αναφέρονται σε μια κοινή μεταβλητή (τη B), πρέπει να αποκλειστούν αμοιβαία, δηλαδή τα send και receive είναι κρίσιμες περιοχές όσον αφορά το B.

Παρατήρηση: Τα send, receive μπορούν να μετακινούν μηνύματα είτε «με αντιγραφή» (by value) είτε «με αναφορά» (by reference)

Άσκηση Αυτοαξιολόγησης 3.9

Τα ανωτέρω μπορούν να επεκταθούν κατάλληλα για την περίπτωση πολλών αποστολέων και πολλών παραληπτών. Ποια θα πρέπει να είναι τα ορίσματα των εντολών send και receive;

3.7.2 Σημαφόροι (Semaphores)

Σε πολλές περιπτώσεις συντονισμού αρκεί μόνο ένα «σήμα συντονισμού» από μια διαδικασία προς κάποια άλλη όταν κάποιο συγκεκριμένο γεγονός λάβει χώρα. Ένα

τέτοιο σήμα είναι «μήνυμα με κενό περιεχόμενο». Τέτοιου είδους μηνύματα δεν ξεχωρίζουν το ένα από το άλλο, δε χρειάζεται λοιπόν να τα αποθηκεύει το σύστημα, αρκεί να τα μετράει. Ο απομονωτής, σε αυτή την περίπτωση, καταλήγει να είναι ένας μη αρνητικός ακέραιος που ορίζει τον αριθμό των μηνυμάτων που έχουν σταλεί αλλά δεν έχουν ακόμη παραληφθεί. Μια τέτοια μεταβλητή συντονισμού καλείται «σημαφόρος» (semaphore). Μπορεί να δηλωθεί με τη γλωσσική δήλωση:

```
var v: semaphore;
```

Τα αντίστοιχα των send και receive είναι τώρα οι εντολές συντονισμού

signal (u) [ή V(u)] και wait (u) [ή P(u)].

Τα P και V (πάνω στην ίδια σημαφόρο) αποκλείονται αμοιβαία, δηλαδή ορίζουν κρίσιμες περιοχές όσον αφορά τη συγκεκριμένη σημαφόρο. Το καθένα από τα P και V εκτελείται «ατομικά», δηλαδή χωρίς διακοπή ή παρεμβολή κώδικα άλλης διαδικασίας.

Άσκηση Αυτοαξιολόγησης 3.10

Ορίστε τα V(u) και P(u) χρησιμοποιώντας τις εντολές sent και receive.

Μπορούμε να αναφερθούμε στη σημαφόρο u με 3 ακέραιους:

1. s(u): ο αριθμός των σημάτων που έχουν σταλεί
2. r(u): ο αριθμός των σημάτων που έχουν ληφθεί
3. c(u): ο αριθμός των «αρχικών σημάτων»

στον απομονωτή.

Η «ανισότητα της σημαφόρου» εκφράζει ότι κάθε στιγμή:

$$0 \leq r(u) \leq s(u) + c(u) \leq r(u) + \text{maxint}$$

όπου maxint είναι ο μεγαλύτερος ακέραιος που δέχεται η υπολογιστική μηχανή.

Φυσικά, μπορούμε να χαρακτηρίσουμε τη σημαφόρο και με ένα μόνο ακέραιο, u, τον αριθμό των αποσταλθέντων μηνυμάτων που δεν έχουν παραληφθεί ακόμη

$$u = s(u) + c(u) - r(u)$$

[Αρχικά $u = c(u)$].

Οι κανόνες συντονισμού των σημαφόρων έχουν ως εξής:

1. P(u) [ή wait(u)]: Αν το P(u) εκτελεστεί όταν $u > 0$, τότε το $u := u - 1$ και η διαδικασία που εκτέλεσε το P(u) θα συνεχίσει.

Αν όμως το $P(u)$ επιχειρήσει να εκτελεστεί όταν το $u = 0$, τότε η διαδικασία μπαίνει στην ουρά αναμονής της σημαφόρου u , την q_u , όπου θα περιμένει μέχρις ότου γίνει δυνατόν να προχωρήσει.

2. $V(u)$ [ή $\text{signal}(u)$]: Το $V(u)$ αυξάνει το u κατά 1. Αν η q_u δεν είναι άδεια, τότε το $\Lambda\Sigma$ επιλέγει μία από τις διαδικασίες της ουράς, τη βγάζει από την ουρά και της επιτρέπει να συνεχίσει (δηλαδή το pcb της μεταφέρεται από την q_u στην ready queue), ολοκληρώνοντας βέβαια και το $P(u)$ που την είχε στείλει στην ουρά, δηλαδή αφαιρώντας 1 από το u . Η χρονοδρομολόγηση των διαδικασιών που αναμένουν στην ουρά q_u εφαρμόζει την «πολιτική» του $\Lambda\Sigma$ (για παράδειγμα, μπορεί να αντιμετωπίζει όλους τους χρήστες «ισότιμα» ή να δίνει προτεραιότητα ανάλογα με τη σημασία της διαδικασίας, όπως ένα σύστημα που χειρίζεται εργοστάσια πρέπει να δίνει προτεραιότητα σε διαδικασίες που αφορούν την ασφάλεια). Σε κάθε περίπτωση όμως, πρέπει το $\Lambda\Sigma$ να εξασφαλίσει ότι δεν μπορεί μια διαδικασία να μείνει στην ουρά q_u απεριόριστα.

3.7.3 Υλοποίηση Κρίσιμων Περιοχών με Σημαφόρους

Ο κώδικας

```
var R : shared T ;
```

```
cobegin
```

<u>process P1</u>	<u>process P2</u>	...	<u>process Pn</u>
region R do S1	region R do S2	...	region R do Sn

```
coend
```

μπορεί να υλοποιηθεί χρησιμοποιώντας σημαφόρους P και V ως εξής:

```
var R : record content : T;
```

```
mutex : semaphore;
```

```
begin
```

```
mutex := 1;
```

```
cobegin
```

```
“P1”
```

```
“Pn”
```

```
begin ... begin
```

```
wait (mutex); ... wait (mutex);
```



```

S1; ... Sn ;
signal (mutex); ... signal (mutex);
end ... end
coend
end

```

Απόδειξη:

Επειδή οι διαδικασίες εκτελούν το wait (P) πριν από το signal (V), έχουμε

$$0 \leq s(\text{mutex}) \leq r(\text{mutex})$$

Αφετέρου, η ανισότητα της σημαφόρου δίνει

$$0 \leq r(\text{mutex}) \leq s(\text{mutex}) + 1$$

Συνδυάζοντας έχουμε

$$0 \leq r(\text{mutex}) - s(\text{mutex}) \leq 1$$

Αλλά $r(\text{mutex}) - s(\text{mutex})$ είναι ο αριθμός των διαδικασιών που έχουν εκτελέσει την πράξη P αλλά όχι την V, δηλαδή ο αριθμός των διαδικασιών μέσα στις κρίσιμες περιοχές τους. Άρα, το πολύ μια (1) κρίσιμη διαδικασία μπαίνει μέσα στην κρίσιμη περιοχή.

Παρατηρήσεις:**Άσκηση Αυτοαξιολόγησης 3.11**

- Είδαμε ότι, για να υλοποιήσουμε κρίσιμες περιοχές, χρησιμοποιούμε σημαφόρους (που είναι πιο απλές κρίσιμες περιοχές). Όμως και αυτές δεν έρχονται από τον ουρανό: είναι σύνθετες λειτουργίες, η ατομική εκτέλεση των οποίων πρέπει να υλοποιηθεί με κάποιον τρόπο. Δείξτε πώς μπορούμε να υλοποιήσουμε σημαφόρους χρησιμοποιώντας τον αλγόριθμο του Dekker. Αν αναλύσετε ακόμη περισσότερο σε τι βασίζεται και ο αλγόριθμος αυτός, θα δείτε ότι στηρίζεται σε ατομική λειτουργία βασικών εντολών του hardware, και συγκεκριμένα στο memory interlock, που είναι ακόμα πιο απλή κρίσιμη περιοχή. Δηλαδή, για να υλοποιήσουμε μια κρίσιμη περιοχή, απαιτείται μια πιο απλή κρίσιμη περιοχή και ξανά μέχρι το ατομικό επίπεδο που οι ατομικές καταστάσεις είναι διακριτές και αμοιβαία αποκλειόμενες λόγω κατασκευής του hardware.

2. Η γλωσσική έκφραση «region v do S» υπερέχει έναντι της υλοποίησης με P και V, γιατί δίνει στον Compiler τη δυνατότητα συντακτικής αναγνώρισης της κρίσιμης περιοχής.

Άσκηση Αυτοαξιολόγησης 3.12

Δύο κοινά λάθη

Ας θεωρήσουμε τον κώδικα

mutex :=1	
και	
wait(mutex);	signal(mutex);
S;	S;
wait(mutex);	wait(mutex);
(A)	(B)

Τι πρόβλημα παρουσιάζει η περίπτωση (A) και τι η (B) αν υποθέσουμε ότι έχουν προταθεί για να εξασφαλίσουν αμοιβαίο αποκλεισμό διαδικασιών;

Απάντηση:

Η περίπτωση (A) οδηγεί σε αδιέξοδο. Η (B) επιτρέπει σε περισσότερες από μία διαδικασίες να εισέλθουν ταυτόχρονα στην κρίσιμη περιοχή.

Άσκηση Αυτοαξιολόγησης 3.13

Ένα σχοινί είναι δεμένο από τη μια άκρη μιας χαράδρας ως την άλλη, έτσι ώστε όποιος θέλει να διασχίσει τη χαράδρα να το κάνει κρεμασμένος από το σχοινί. Αν χρησιμοποιήσουν το σχοινί άτομα που κινούνται και προς τις δύο κατευθύνσεις, θα δημιουργηθεί πρόβλημα (αδιέξοδο) στο σημείο συνάντησής τους πάνω από τη χαράδρα. Όταν κάποιος θελήσει να διασχίσει τη χαράδρα, πρέπει να ελέγξει αν έρχεται άλλος από την αντίθετη κατεύθυνση. Γράψτε ένα πρόγραμμα για την αποφυγή αδιεξόδου με τη χρήση σημαφόρων. Δεν υπάρχει πρόβλημα αν άτομα που κινούνται προς τη μια κατεύθυνση καθυστερούν για αόριστο χρονικό διάστημα αυτούς που κινούνται προς την άλλη κατεύθυνση.

Απάντηση:

```

#define TRUE 1

#define FALSE 0

typedef int semaphore;

semaphore mutex =1; /* επιβάλλει μετακίνηση στο σκοινί προς τη μια
κατεύθυνση */

semaphore west = 1; /* κίνηση προς τη δύση */

semaphore east = 1; /* κίνηση προς την ανατολή */

share int west_count = 0, east_count = 0; /* Μετρητές όσων κινούνται προς
δύση και ανατολή αντίστοιχα */

void east(void) {
    while (TRUE) {
        wait(&east);
        east_count++;
        if(east_count ==1)
            wait(&mutex);
        signal(&east);
        move_east();
        wait(&east);
        east_count--;
        if(east_count == 0)
            signal(&mutex);
        signal(&east);
    }
}

void west(void) {
    while (TRUE) {
        wait(&west);
        west_count++;
        if(west_count ==1)
            wait(&mutex);
        signal(&west);
        move_west();
        wait(&west);
        west_count--;
        if(west_count == 0)
            signal(&mutex);
        signal(&west);
    }
}

```

Άσκηση Αυτοαξιολόγησης 3.14

Επαναλάβετε το προηγούμενο πρόβλημα, αλλά αποφεύγοντας τώρα την παρατεταμένη στέρηση πόρων. Αν κάποιος θελήσει να διασχίσει τη χαράδρα προς μια κατεύθυνση (π.χ. ανατολικά) και όταν φτάσει δει πως κάποιος άλλος τη διασχίζει προς τα δυτικά, περιμένει μέχρι να αδειάσει το σχοινί, αλλά δεν επιτρέπεται σε άλλους να μετακινηθούν προς τα δυτικά αν δεν περάσει τουλάχιστον ένας προς τα ανατολικά.

Απάντηση:

```
#define TRUE 1
#define FALSE 0

typedef int semaphore;

semaphore mutex = 1; /* επιτυχάνει αμοιβαίο αποκλεισμό στη χρήση των
κοινών μεταβλητών east_count και west_count*/

semaphore turn = 1; /*καθορίζει ποια κατεύθυνση έχει προτεραιότητα για
κίνηση*/

share int west_count = 0, east_count = 0; /* Μετρητές όσων κινούνται προς
δύση και ανατολή αντίστοιχα */

share int flag = 0; /* το χρησιμοποιεί η καθεμιά από τις διαδικασίες που ακο-
λουθούν για να δει αν είναι η σειρά της να χρησιμοποιήσει το σχοινί */

void east(void) {
    while (TRUE) {
        wait(&mutex);
        east_count++;
        signal(&mutex);
        if(flag==0){
            move_east();
            wait(&mutex);
            east_count--;
        }
    }
}

void west(void) {
    while (TRUE) {
        wait(&mutex);
        west_count++;
        signal(&mutex);
        if(flag==1) {
            move_west();
            wait(&mutex);
            west_count--;
        }
    }
}
```

```

        if(west_count != 0)                if(east_count != 0)
            flag = 1;                        flag = 0;
            signal(&mutex);                  signal(&mutex);
        }                                    }
    }                                        }
}                                           }

```

3.7.4 Κρίσιμες Περιοχές υπό Συνθήκη

Μέχρι τώρα χρησιμοποιήσαμε τα ακόλουθα εργαλεία για να καθυστερήσουμε μια διαδικασία έως ότου ισχύσει κάποια συνθήκη.

<u>Εργαλείο</u>	<u>Συνθήκη</u>
• κρίσιμη περιοχή	αμοιβαίος αποκλεισμός
• απομονωτής	μήνυμα διαθέσιμο
• σημαφόρος	σήμα διαθέσιμο

Τώρα θα εισαγάγουμε ένα πιο γενικό εργαλείο συντονισμού, που δίνει τη δυνατότητα σε μια διαδικασία να περιμένει έως ότου μια αυθαίρετη συνθήκη ισχύσει.

Ας θεωρήσουμε τη γλωσσική έκφραση

```

var u : shared T
region u do
    begin
        So ;
    await B;
        S1
    end

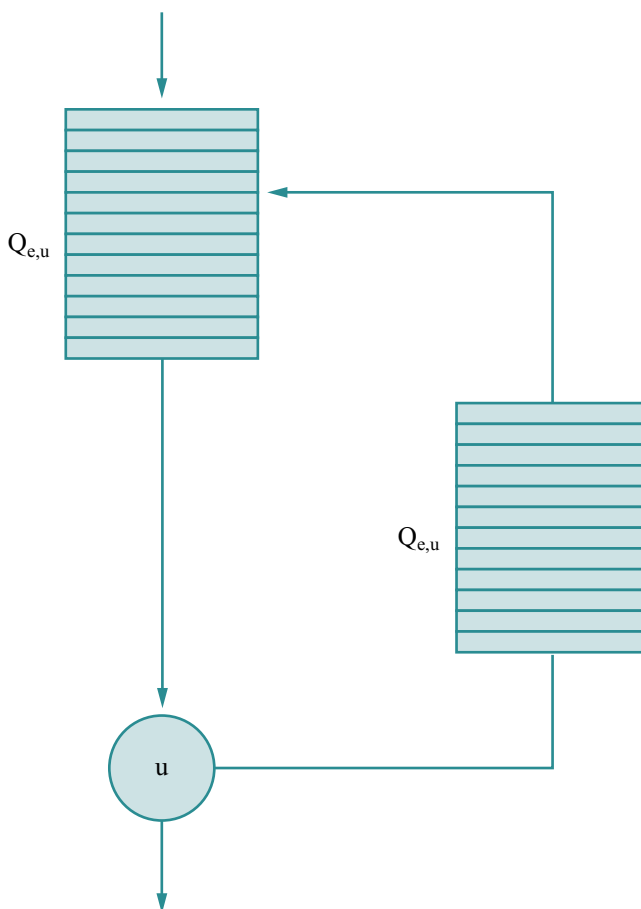
```

Η «συνθήκη συντονισμού» B είναι μια οποιαδήποτε boolean συνάρτηση, του u.

Μόλις μια διαδικασία είναι έτοιμη να εισέλθει σε μια κρίσιμη περιοχή της u, τότε εισέρχεται πρώτα σε μια ουρά αναμονής Q_u (ειδική για τη μεταβλητή u). Από αυτή την ουρά οι διαδικασίες εισέρχονται στις κρίσιμες περιοχές τους, μία κάθε φορά, πετυχαίνοντας αμοιβαίο αποκλεισμό.

Αφού εισέλθει στην κρίσιμη περιοχή, η διαδικασία ελέγχει εάν η $B(u)$ είναι αληθής (αρχή του wait). Αν ναι, τότε η διαδικασία εκτελεί τον κώδικα $S1$ και εξέρχεται από την κρίσιμη περιοχή. Αν όχι, η διαδικασία αφήνει την κρίσιμη περιοχή προς στιγμή και μπαίνει σε μια δεύτερη ουρά, την $Q_{e,u}$ [event queue $Q(e,u)$].

Άλλες διαδικασίες μπορούν τώρα να εισέλθουν στις κρίσιμες περιοχές τους μέσω της Q_u . Αυτές μπορούν να εκτελέσουν await σε κάποια –πιθανόν– διαφορετική boolean συνθήκη B . Αν η συνθήκη δεν ικανοποιείται, τότε η αντίστοιχη διαδικασία πάει στην ίδια ουρά Q_e . Όταν μια διαδικασία ολοκληρώσει επιτυχώς την κρίσιμη περιοχή, όλες οι διαδικασίες της ουράς $Q(e,u)$ μεταφέρονται στην ουρά Q_u , απ' όπου θα ξαναμπούν στην κρίσιμη περιοχή τους και θα ελέγξουν τις συνθήκες τους ξανά.



Σχήμα 3.3

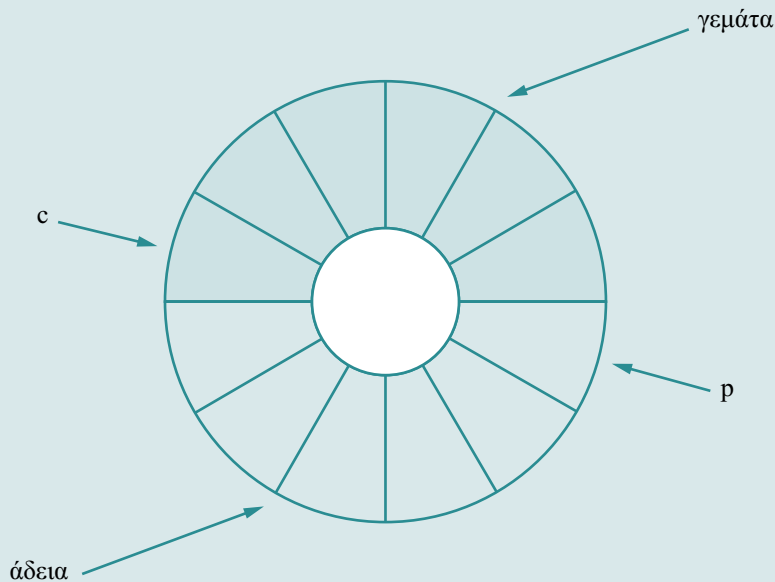
*Η σημασία της
κρίσιμης περιοχής
υπό συνθήκη*

Άσκηση Αυτοαξιολόγησης 3.15

1. Τι νόημα έχει να ξαναπροσπαθήσει μια διαδικασία; Δε θα είναι πάλι η τιμή του B ψευδής;
2. Γιατί δεν επιστρέφει η διαδικασία που απέτυχε στο τέλος της ουράς Q_u και δημιουργούμε μια ξεχωριστή ουρά;

Απάντηση:

1. Αφού η διαδικασία που τελείωσε επιτυχώς την κρίσιμη περιοχή της μπορεί να άλλαξε την τιμή του u , έτσι ώστε μερικές από τις συνθήκες να έχουν πάρει αληθή τιμή.
2. Μήπως διότι η ουρά Q_u δεν είναι αναγκαστικά FIFO;

**Σχήμα 3.4**Απομονωτής
μηνυμάτων**Άσκηση Αυτοαξιολόγησης 3.16**

Υλοποιήστε τον απομονωτή σαν κυκλική ουρά. Μπορείτε να χρησιμοποιήσετε ένα δείκτη c για την αρχή των θέσεων που είναι γεμάτες με μηνύματα και ένα δείκτη p για την αρχή των άδειων θέσεων. Μια μεταβλητή `full` μπορεί να δείχνει πόσα μηνύματα έχει ο απομονωτής και μια άλλη, `empty`, πόσα κενά.

1. Υλοποιήστε τον απομονωτή χρησιμοποιώντας κρίσιμες περιοχές υπό συνθήκη.
2. Υλοποιήστε τον απομονωτή χρησιμοποιώντας σηματοφόρους.

Απάντηση:

Ας υποθέσουμε ότι η ακέραια μεταβλητή *full* δείχνει πόσα μηνύματα (που έχουν σταλεί αλλά όχι παραληφθεί) έχει ο απομονωτής. Αρχικά *full* = 0. Έστω ότι η ακέραια μεταβλητή *empty* δείχνει πόσα κενά έχει ο απομονωτής. Αρχικά *empty* = *max*.

Υλοποίηση με κρίσιμες περιοχές υπό συνθήκη

type B = shared record /* δηλαδή ορίζουμε σαν B ένα διαμοιρασμένο record */

buffer : shared array 0...*max* - 1 of T ;

p, *c* : shared 0...*max* - 1;

full : shared 0...*max* - 1;

p = *c* = *full* = 0;

procedure send (*m* : T ; var *b* : B) ;

region *b* do

begin

await *full* < *max* ;

buffer (*p*) := *m* ;

p := (*p* + 1) mod *max*;

full := *full* + 1;

end

procedure receive(var *m* : T; *b* : B);

region *b* do

begin

await *full* > 0;

m := buffer(*c*);

c := (*c* + 1) mod *max*;

full := *full* - 1;

end

Υλοποίηση με σηματοφόρους

```
type B = record
v : shared record
buffer : shared array 0...max - 1 of T;
p,c : shared 0...max - 1;
full, empty : semaphore;

p = c = full = 0;
empty = max ;

procedure send (m : T; var b : B);
begin
    with b do
        begin
            wait (empty);
            region u do
                begin
                    buffer (p) := m ;
                    p:= (p + 1) mod max;
                end
            signal(full);
        end
    end
end

procedure receive (var m : T; b : B) ;
begin
    with b do
```

```
begin
    wait (full) ;
    region u do
        begin
            m: = buffer(c);
            c: = (c + 1) mod max ;
        end
        signal (empty);
    end
end
```

Παράδειγμα 3.1: Αναγνώστες και γραφείς

Το πρόβλημα:

Δύο ομάδες παράλληλων διαδικασιών (οι αναγνώστες και οι γραφείς) μοιράζονται έναν κοινό πόρο (π.χ. ένα multimedia web document). Οι αναγνώστες μπορούν να διαβάσουν ταυτόχρονα, αλλά κάθε γραφέας πρέπει να έχει αποκλειστική προσπέλαση στο βιβλίο. Όταν ένας γραφέας είναι έτοιμος να γράψει, θα πρέπει να του δίνεται η δυνατότητα όσο το δυνατόν πιο γρήγορα.

Η ομάδα των αναγνωστών και των γραφέων αποκλείουν η μία την άλλη (δηλαδή, όταν ένας αναγνώστης διαβάζει το βιβλίο, κανένας γραφέας δε γράφει, και, όταν ένας γραφέας γράφει, κανένας αναγνώστης δε διαβάζει).

Ορολογία:

Η κάθε διαδικασία εκτελεί τον ακόλουθο κύκλο:

1. Ζήτη το multimedia web document
2. Αφού το πάρεις, διάβασε (ή γράψε)
3. Άφησε το document

Μια διαδικασία καλείται ενεργός από τη στιγμή που ζήτησε το βιβλίο μέχρι τη στιγμή που το έδωσε πίσω. Μια διαδικασία καλείται εκτελούσα από τη στιγμή που της δόθηκε άδεια να χρησιμοποιήσει το document (αφού το πάρεις) μέχρι τη στιγμή που

το αφήνει.

Έστω ότι

ar = αριθμός ενεργών αναγνωστών

rr = αριθμός εκτελούντων αναγνωστών

aw = αριθμός ενεργών γραφέων

rw = αριθμός εκτελούντων γραφέων

Αρχικά $ar = rr = aw = rw = 0$

Κριτήρια Ορθότητας:

1. $0 \leq rr \leq ar$ και $0 \leq rw \leq aw$ και $rr \leq ar$ και $rw \leq aw$
2. $\text{not } (rr > 0 \text{ και } rw > 0)$ (Αμοιβαίος αποκλεισμός) $rr * rw = 0$
3. $(rr = 0 \text{ και } rw = 0)$ και
4. $(ar > 0 \text{ ή } aw > 0)$ αν $(rr > 0 \text{ ή } rw > 0)$ (πεπερασμένος χρόνος – αποφυγή αδιεξόδου)
5. Προτεραιότητα γραφέων. Το βιβλίο μπορεί να δοθεί σε ενεργό αναγνώστη μόνο όταν δεν υπάρχουν ενεργοί γραφείς (δηλαδή $aw = 0$).

Άσκηση Αυτοαξιολόγησης 3.17

Υλοποιήστε τις διαδικασίες «Αναγνώστης» και «Γραφέας» με σημαφόρους. Μπορείτε να γράψετε και να χρησιμοποιήσετε την υπορουτίνα «grant reading», που εξετάζει αν το βιβλίο μπορεί να δοθεί για διάβασμα αμέσως, και την υπορουτίνα «grant writing», που εξετάζει αν το βιβλίο μπορεί να δοθεί αμέσως στους γραφείς. (Η άσκηση είναι δύσκολη, αλλά μην κοιτάξετε κατευθείαν την απάντηση! Αν δεν την καταφέρετε, κοιτάξτε την απάντηση και προσθέστε αναλυτικά σχόλια για το τι κάνει κάθε βήμα. Επανέλθετε σε μια βδομάδα και επιχειρήστε ξανά.

Υλοποιήστε τις διαδικασίες χρησιμοποιώντας κρίσιμες περιοχές υπό συνθήκη.

Απάντηση:

Υλοποίηση με σημαφόρους (δύσκολη)

type T = record

ar, rr, aw : shared integer ;

```
var u : shared T ;  
reading, writing, w : semaphore ;
```

Αρχικοποίηση $ar = rr = aw = rw = \text{reading} = \text{writing} = 0, w = 1$.

```
cobegin  
begin “reader”  
    region u do  
        begin  
            ar := ar + 1 ;  
            grant reading (u, reading);  
        end  
        wait (reading);  
        read ;  
        region u do  
            begin  
                rr := rr + 1;  
                ar := ar - 1;  
                grant writing (u, writing) ;  
            end  
            ...  
        end  
    end  
begin “writer”  
    region u do  
        begin  
            aw := aw + 1;  
            grant writing (u, writing);  
        end  
    end
```

```
...
end
begin "writer"
  region u do
    begin
      aw := aw + 1;
      grant writing (u, writing);
    end
    wait (writing);
    P(w);
    write;
    V(w);
  Region u do
    begin
      rw := rw + 1;
      aw := aw - 1;
      grant reading (u, reading);
    end
    ...
  end
...
coend

procedure grant reading (var u : T ; reading : semaphore) ;
begin
  with v do
    if rr = 0 then
      while rw < aw do
```

```
begin
    rw := rw + 1;
    signal (writing);
end
```

```
end
```

Υλοποίηση με κρίσιμες περιοχές υπό συνθήκη

```
var u : shared record
rr, aw : shared integer
```

Αρχικά $rr = aw = 0$

```
cobegin
    begin "reader"
        region u do
            begin await aw = 0; rr := rr + 1 end
            read;
            region u do rr := rr - 1
            ...
        end
    begin "writer"
        region u do
            begin aw = aw + 1 ; await rr := 0 end
            write;
            region u do aw := aw - 1;
            ...
        end
    ...
coend
```

3.7.5 Υλοποίηση Γενικής Σημαφόρου με Δυαδική Σημαφόρο

Μια γενική σημαφόρος S μπορεί να αντικατασταθεί από μια ακέραια μεταβλητή N_s και δύο δυαδικές σημαφόρους (δηλαδή σημαφόρους που παίρνουν μόνο τιμές true και false [ή 0 και 1]) έστω mutex_s , delay_s , ως εξής (υποθέτουμε ότι τα P και V είναι δοσμένα και ατομικά για δυαδικές σημαφόρους).

$\text{mutex}_s, \text{delay}_s$: binary semaphore;

S : semaphore;

N_s : shared integer;

Αρχικοποίηση $\text{mutex}_s = 1$, $\text{delay}_s = 0$, $N_s = S$;

cobegin

Begin 'P_{γενικό} (S)'

$P(\text{mutex}_s)$;

$N_s := N_s - 1$;

 if $N_s \leq -1$; then

 begin

$V(\text{mutex}_s)$;

$P(\text{delay}_s)$;

 end

 else $V(\text{mutex}_s)$;

 End

Begin 'V_{γενικό} (S)'

$P(\text{mutex}_s)$;

$N_s := N_s + 1$;

 if $N_s < 0$ then $V(\text{delay}_s)$;

$V(\text{mutex}_s)$;

 End

Άσκηση Αυτοαξιολόγησης 3.18

Εξηγήστε γιατί τα γενικευμένα P και V όπως ορίζονται παραπάνω δουλεύουν σωστά.

Τι ρόλο παίζει καθεμία από τις δυαδικές σημαφόρους; Πότε έχουν την τιμή 0 και πότε 1;

Μπορούμε να αντιμεταθέσουμε τη σειρά των εντολών

V(mutex_s);

P (delay_s);

στον κώδικα του P_{γενικό};

Εξηγήστε πώς εξασφαλίζεται ο αμοιβαίος αποκλεισμός όταν τα πιο κάτω ζευγάρια διαδικασιών εκτελούνται παράλληλα:

$$P_{\text{γενικό}} - P_{\text{γενικό}}, V_{\text{γενικό}} - V_{\text{γενικό}} \text{ και } P_{\text{γενικό}} - V_{\text{γενικό}}.$$

Γράψτε τον ορισμό της γενικής σημαφόρου (χωρίς να τον κοιτάξετε!). Συμπληρώστε τον ορισμό που γράψατε βάσει του ορισμού της Ενότητας 3.7.2. Τι άλλο πρέπει να εξηγήσετε για να ολοκληρώσετε την «απόδειξη» της ορθότητας του κώδικα των P_{γενικό} και V_{γενικό}; Γράψτε το.

3.7.6 Υλοποίηση P και V μέσω hardware

Ας κοιτάξουμε προσεκτικά τον κώδικα της Λύσης 1 για να δούμε το λόγο που δε δουλεύει σωστά. Η «τρύπα» βρίσκεται μεταξύ του ελέγχου «Είναι ο πόρος ελεύθερος» της θετικής απάντησης και της δήλωσης «Τον καταλαμβάνω». Αν ήταν δυνατόν ο έλεγχος της τιμής της μεταβλητής και η αλλαγή της στην περίπτωση που είναι ελεύθερη να γίνει μεμιάς, ατομικά, χωρίς δυνατότητα παρεμβολής, τότε η Λύση 1 θα ήταν σωστή. Το hardware πολλών H/Y παρέχει μια τέτοια ακριβώς δυνατότητα: την εντολή συντονισμού test-and-set, που γίνεται ατομικά (χωρίς καμιά διακοπή ή παρεμβολή)

var X : shared boolean

Αρχικά X : true;

Για να υλοποιήσουμε δυαδικές σημαφόρους:

Begin “P(S)”

L: if TS(S) then go to L ;

End

Begin “V(S)”

S: = false;

End

Άσκηση Αυτοαξιολόγησης 3.19

Η εντολή FA(m,r) ατομικά προσθέτει το περιεχόμενο της μεταβλητής m και με την τιμή r αποθηκεύει το άθροισμα στη m και επιστρέφει ως αποτέλεσμα την αρχική τιμή της m. Υλοποιήστε αμοιβαίο αποκλεισμό για αυθαίρετο αριθμό διεργασιών χρησιμοποιώντας μόνο τη FA ως εργαλείο συντονισμού.

Απάντηση:

```

procedure FA (var m: integer, r: integer)
var temp: integer;
begin
    temp := m;
    m := m + r;
    return (temp);
end

```

Ένας τρόπος με τον οποίο υλοποιούμε αμοιβαίο αποκλεισμό χρησιμοποιώντας τη διαδικασία FA είναι ο ακόλουθος:

```

έστω var lock, key: shared integer;
και noop εντολή που δεν κάνει τίποτα.
procedure pi
begin
    lock := 0;
    key := 1;

```

```

repeat
    while (FA (lock, key) > 0) noop;
    Κρίσιμο τμήμα;
    lock := 0;
    Υπόλοιπο της διαδικασίας
until false;
end

```

Ερώτηση:

Είναι αυτή η λύση ρεαλιστική για το πρόβλημα του αμοιβαίου αποκλεισμού (ικανοποιούνται και οι τρεις συνθήκες που έχουμε ορίσει στην Ενότητα 3.5);

Απάντηση:

Παραβιάζεται η τελευταία συνθήκη που αφορά την «πολύασχολη αναμονή». Παρατηρούμε ότι, αν μια διαδικασία είναι πολύ πιο γρήγορη από μια άλλη και κατορθώνει να τελειώνει και το κομμάτι του κώδικά της που ακολουθεί το κρίσιμο τμήμα πολύ γρήγορα, πριν τελειώσει το κβάντο του χρόνου της, τότε αυτή η διαδικασία εκτελείται πολύ περισσότερες φορές από μια άλλη, πιο αργή διαδικασία, η οποία περιμένει σε «πολύασχολη αναμονή» για μεγάλα χρονικά διαστήματα.

Θα παρουσιάσουμε στη συνέχεια μια νέα λύση στο πρόβλημα του αμοιβαίου αποκλεισμού χρησιμοποιώντας τη FA η οποία ικανοποιεί όλες τις συνθήκες που χρειάζονται για να είναι η λύση ρεαλιστική.

Έστω οι εξής κοινές μεταβλητές:

```

var lock: shared integer;

και var waiting: shared array[0..n-1] of boolean; με αρχικές τιμές false.

και noop εντολή που δεν κάνει τίποτα.

procedure pi
var j: integer; (j: 0 .. n-1)
    key: integer;
begin
    key := 1;
repeat

```

```

    waiting[i] := true;
    while (waiting[i] and (FA (lock, key) > 0)) do noop;
    waiting[i] := false;
    Κρίσιμο τμήμα;
    j := (i + 1) mod n;
    while ((j != i) and (!waiting[j])) do
        j := (j + 1) mod n;
    if (j == i) then lock := 0;
    else waiting[j] := false;
    Υπόλοιπο της διαδικασίας
until false;
end

```

3.7.7 Άλλες εντολές συντονισμού

- Monitor: Είναι μια κοινή δομή δεδομένων με ένα σύνολο ατομικών, αμοιβαία αποκλειόμενων «πράξεων», ορισμένων πάνω στη δομή δεδομένων.
- Ουρές Γεγονότων (Event Queues)

Δηλώνονται ως:

```
var e : event u
```

(συσχετίζει την ουρά e με την κοινή μεταβλητή u).

Μια διαδικασία μπορεί να φύγει από μια κρίσιμη περιοχή (της u) και να πάει στην e με την εντολή: await (e). Μια διαδικασία μπορεί να μεταφέρει όλες τις διαδικασίες (που περιμένουν στην (e) ξανά στην ουρά Q_u της u, με την εντολή: cause (e). Οι ουρές γεγονότων είναι απλοί monitors και επιτρέπουν λεπτομερή έλεγχο στη δρομολόγηση των διαδικασιών.

3.8 Συντονισμός διαδικασιών που δεν συνεργάζονται

Όλες οι εντολές που εξετάστηκαν ως τώρα αφορούν διαδικασίες που *συνεργάζονται* για να συντονιστούν. Ένα πιο βασικό (πρωτογενές) σύνολο εντολών που έχει προταθεί για συγχρονισμό διαδικασιών μη συνεργάσιμων είναι το δίδυμο Block – Wakeup.

Άσκηση Αυτοαξιολόγησης 3.20

Τι σημαίνει ότι οι διαδικασίες «συνεργάζονται»; Τι σημαίνει «μη συνεργάσιμες» διαδικασίες;

Απάντηση:

Λέμε ότι οι διαδικασίες συνεργάζονται αν είναι γραμμένες από την αρχή έτσι ώστε να λαμβάνουν υπόψη την πιθανή ύπαρξη άλλων διαδικασιών που μπορεί να χρειαστούν τους ίδιους πόρους. Τέτοιες είναι, για παράδειγμα, οι διαδικασίες του ίδιου του ΛΣ. Αντίστροφα, για τις διαδικασίες που προκύπτουν από προγράμματα χρηστών δεν μπορεί το ΛΣ να υποθέσει ότι έχουν λάβει τέτοια πρόνοια, αλλά ότι θεωρούν ότι «είναι μόνες στον κόσμο». Επομένως, το συντονισμό θα τον κάνει το ίδιο το ΛΣ, χωρίς να υποθέσει ότι οι διαδικασίες είναι γραμμένες «σωστά» από άποψη συντονισμού.

Κάθε PCB μιας διαδικασίας i έχει έναν boolean wakeup – waiting «διακόπτη» $wws(i)$. Η σημασιολογία των block και wakeup έχει ως εξής:

Block (i):

```
if (not ww(i) then Block process(i)
    else wws(i) := false;
```

Wakeup(i):

```
If ready (i) then wws(i) := true
    else Activate process (i);
```

Η «Block process(i)» βάζει την i σε μια ουρά αναμονής. Η «Activate» βάζει την i στην ready queue. Το τεστ ready(i) επιστρέφει «true» αν η i είναι στην κατάσταση running ή ready. Επιστρέφει «false» σε κάθε άλλη περίπτωση.

Βλέπουμε ότι οι Block – Wakeup ενεργούν άμεσα σε διαδικασίες, αντί να στέλνουν σήματα σε διαδικασίες που πιθανόν να είναι ανέτοιμες να τα παραλάβουν.

3.9 Κατανεμημένοι αλγόριθμοι για αμοιβαίο αποκλεισμό

Σε ένα κατανεμημένο σύστημα δεν υπάρχουν «κεντρικές» οντότητες προσβάσιμες σε όλες τις διαδικασίες. Για παράδειγμα, δεν μπορούμε να υποθέσουμε ότι υπάρχει ένα κοινό ρολόι, ούτε καν κοινές μεταβλητές. Οι διαδικασίες επικοινωνούν ανταλλάσσοντας μηνύματα.

Για λόγους σαφήνειας, αναπτύχθηκε μια μεθοδολογία γραφής κατανεμημένων πρωτοκόλλων που κρύβει την ύπαρξη μηνυμάτων με το να χρησιμοποιεί ψευδο-κοινές μεταβλητές για επικοινωνία διαδικασιών. Σε τέτοιου είδους περιγραφές, οι διαδικασίες έχουν δύο ειδών μεταβλητές:

1. τις «τυπικές» (local) μεταβλητές, που ανήκουν μόνο σε μια διαδικασία η καθεμία και που είναι απροσπέλαστες από άλλες διαδικασίες,
2. τις «ειδικές» (specific ή flags), η καθεμία εκ των οποίων μπορεί να διαβαστεί / τροποποιηθεί από μία μόνο διαδικασία, ενώ οι άλλες διαδικασίες μπορούν μόνο να τη διαβάσουν.

Βέβαια, στην πραγματικότητα το διάβασμα μιας μεταβλητής ειδικής για τη διαδικασία P_i σημαίνει δύο γεγονότα: (α) την αποστολή ενός μηνύματος που ζητάει την τρέχουσα τιμή της μεταβλητής και (β) την παραλαβή ενός μηνύματος που περιέχει αυτή την τιμή. Έτσι, χάρη στις flags, η περιγραφή των αλγόριθμων αποφεύγει αρχιτεκτονικές λεπτομέρειες του δικτύου διασύνδεσης των επεξεργαστών στους οποίους τρέχουν οι διαδικασίες.

Το μειονέκτημα τέτοιων περιγραφών είναι ότι συνεπάγονται μεγάλο αριθμό μηνυμάτων. Αυτό ισχύει διότι (με το διάβασμα των flags) τιμές ζητούνται ανεξάρτητα από το αν οι τιμές αυτές έχουν αλλάξει από το προηγούμενο διάβασμα («τυφλό» testing), ενώ θα ήταν σωστότερο να στέλνονται μηνύματα μόνο όταν οι τιμές των μεταβλητών αυτών αλλάζουν.

Ο κάθε αλγόριθμος βασισμένος στη χρήση τυπικών και ειδικών μεταβλητών μπορεί να υλοποιηθεί σε περιβάλλον μιας ΚΜΕ. Έχει, μάλιστα, αρκετά πλεονεκτήματα έναντι των κλασικών αλγόριθμων που χρησιμοποιούν κοινές μεταβλητές: είναι πιο ανθεκτικός σε λάθη, π.χ. το αναπάντεχο σταμάτημα (breakdown) μιας διαδικασίας δεν επηρεάζει τη συνολική συμπεριφορά του συστήματος.

3.9.1 Ο Αλγόριθμος του Ζαχαροπλαστέιου (The Bakery Algorithm)

Προτάθηκε από τον Lamport το 1974. Βασίζεται στην ιδέα (με την οποία λειτουργούν ορισμένα καταστήματα) σύμφωνα με την οποία κάθε πελάτης παίρνει ένα αριθμημένο εισιτήριο κατά την άφιξή του στο σύστημα, που ορίζει και τη διαδοχή στην εξυπηρέτηση.

Εδώ καθεμία από τις διαδικασίες P_0, P_1, \dots, P_{n-1} διαλέγει μόνη της τον αριθμό εισιτηρίου της, βασισμένη στους αριθμούς που έχουν ήδη επιλεγεί από άλλες διαδικασίες. Αν δύο διαδικασίες επιλέξουν τον ίδιο αριθμό, τότε αυθαίρετα δεχόμαστε ότι η διαδικασία με το μικρότερο δείκτη (ταυτότητα της διαδικασίας) θα εξυπηρετηθεί πρώτη.

Οι αναγκαίες μεταβλητές είναι:

var choice : array [0...(n-1)] of boolean;

number array [0...(n-1)] of integer ;

οι οποίες αρχικοποιούνται σε “false” και 0 αντίστοιχα.

Ας σημειωθεί ότι το ζεύγος

choice [i], number [i]

είναι ζεύγος flags που ανήκουν στην P_i . Μόνο η P_i μπορεί να τροποποιήσει αυτές τις δύο μεταβλητές. Κάθε P_j ($j \neq i$) μπορεί μόνο να τις διαβάσει.

Ορισμός

$$[(a,b) < (c,d)] \text{ é } [a < c \text{ ή } (a = c \text{ και } b < d)]$$

Το πρωτόκολλο που τρέχει η P_i είναι το εξής (όπου η j είναι τοπική μεταβλητή της P_i).

choice [i] = true ;

number [i] = 1 + max(number [0]+...number[n-1]) ;

choice [i] = false ;

for j = 0 to n - 1, i ≠ j

begin

wait until not choice [j] ;

wait until number [j] = 0 or

(number [j], i) < (number [j], j)

end ;

ΚΡΙΣΙΜΗ ΠΕΡΙΟΧΗ;

Number [i] = 0 ;

Το πρωτόκολλο αυτό αποφεύγει το αδιέξοδο και εγγυάται κάποια δίκαιη προσπέλαση της κρίσιμης περιοχής (μια διαδικασία θα περιμένει το πολύ $n - 1$ βήματα πριν εισέλθει). Ένα ασθενές σημείο του αλγόριθμου είναι ότι η τιμή του value [i] μπορεί να αυξάνει χωρίς άνω φράγμα. Αυτό το πρόβλημα μπορεί να αντιμετωπιστεί σχετικά εύκολα (πώς;)

Σύνοψη

Μια από τις βασικότερες έννοιες στα λειτουργικά συστήματα είναι η έννοια του πολυ-προγραμματισμού, και έχει τις ρίζες της από την εποχή της τρίτης γενιάς λειτουργικών συστημάτων. Η ταυτόχρονη εκτέλεση όμως πολλών διαδικασιών είναι συνδεδεμένη και με ένα πολύ σημαντικό πρόβλημα, το πρόβλημα του συντονισμού διαδικασιών, το οποίο και μελετήσαμε εκτενώς στο παρόν κεφάλαιο.

Θεωρούμε καταρχήν ότι είναι δυνατές όλες οι πιθανές αναμειγξεις των ακολουθιών των εντολών των διαδικασιών που τρέχουν ταυτόχρονα σε ένα σύστημα. Αποτέλεσμα αυτής της αποδοχής είναι να υπάρχει περίπτωση να γίνονται λάθος ή μη ντετερμινιστικοί υπολογισμοί κατά τη διάρκεια της παράλληλης εκτέλεσης των διαδικασιών αυτών, λόγω του ότι η πρόσβαση από όλες αυτές τις διαδικασίες στις κοινές μεταβλητές δεν προφυλάσσεται με κανέναν τρόπο. Για να λυθούν αυτής της φύσης τα προβλήματα, ορίστηκαν οι έννοιες της **κρίσιμης περιοχής** και του **αμοιβαίου αποκλεισμού**. Ορίσαμε ως κανόνα του αμοιβαίου αποκλεισμού έναν περιορισμό που θα έλεγε «όσο μια διαδικασία χρησιμοποιεί μια κοινή μεταβλητή, οι άλλες διαδικασίες ΔΕΝ μπορούν να έχουν πρόσβαση σε αυτή τη μεταβλητή» και ως κρίσιμη περιοχή ένα μέρος του κώδικα μιας διαδικασίας που αναφέρεται σε μια κοινή μεταβλητή και για το οποίο: (α) όταν μια διαδικασία πρόκειται να εισέλθει στην κρίσιμη περιοχή, τότε αυτό θα συμβεί σε πεπερασμένο χρόνο, (β) κάθε χρονική στιγμή το πολύ μια διαδικασία μπορεί να βρίσκεται στην κρίσιμη περιοχή για κάθε κοινή μεταβλητή και (γ) μια διαδικασία δεν μπορεί να μείνει απεριορίστα στην κρίσιμη περιοχή.

Στην Ενότητα 3.4 παρουσιάσαμε διάφορες προβληματικές λύσεις για το πρόβλημα του αμοιβαίου αποκλεισμού και καταλήξαμε στη λύση του Dekker, η οποία όντως επιτυγχάνει αμοιβαίο αποκλεισμό. Στη συνέχεια παρουσιάσαμε διάφορα εργαλεία τόσο σε επίπεδο λογικού όσο και σε επίπεδο υλικού για την υλοποίηση της συνεργασίας διαδικασιών. Το πρώτο εργαλείο που χρησιμοποιήθηκε είναι ένας απομονωτής (έστω B), ο οποίος χρησιμοποιείται για προσωρινή αποθήκευση των μηνυμάτων που στέλνει μια διαδικασία «παραγωγού» σε μια διαδικασία «καταναλωτή». Σε αυτό τον τρόπο επικοινωνίας υπάρχουν δύο περιορισμοί: ο αποστολέας δεν μπορεί να βάλει στον B περισσότερα μηνύματα απ' όσα μπορεί να χωρέσει και ότι ο παραλήπτης δεν μπορεί να καταναλώσει μηνύματα ταχύτερα από το ρυθμό παραγωγής τους. Οι δύο αυτοί περιορισμοί ικανοποιούνται αν το σύστημα τηρήσει την εξής αρχή συντονισμού: «αν ο αποστολέας προσπαθήσει να βάλει ένα μήνυμα μέσα σε έναν πλήρη απομονωτή, τότε ο αποστολέας θα αναγκαστεί να καθυστερήσει έως ότου ο παραλήπτης παραλάβει τουλάχιστον ένα μήνυμα από τον απομονωτή και, αν ο παραλήπτης προσπαθήσει να παραλάβει ένα μήνυμα από έναν άδειο απομονωτή, τότε ο παραλήπτης

θα καθυστερήσει μέχρις ότου ο αποστολέας βάλει ένα τουλάχιστον μήνυμα στον απομονωτή». Η επικοινωνία επιτυγχάνεται με τις εντολές $send(M, B)$ και $receive(M, B)$, όπου M το μήνυμα και B ο απομονωτής.

Το δεύτερο εργαλείο επικοινωνίας διαδικασιών είναι η σημαφόρος. Η «σημαφόρος» ($var u: semaphore$) είναι ουσιαστικά ένας μετρητής «αποσταλέντων και μη εισέτι παραληφθέντων» σημάτων, ένας μη αρνητικός ακέραιος. Οι διαδικασίες μπορούν να τον αυξήσουν [κατά ένα κάθε φορά: $signal(u)$ ή $V(u)$] ή να τον μειώσουν [κατά ένα πάλι: $wait(u)$ ή $P(u)$] μέχρι το μηδέν. Όποια διαδικασία επιχειρήσει να μειώσει μια σημαφόρο που είναι ήδη στο μηδέν (δηλαδή να παραλάβει σήμα που δεν υπάρχει) πάει σε ουρά αναμονής, ειδική για τη συγκεκριμένη σημαφόρο. Αντίστοιχα, όταν προστεθεί ένα σήμα και αν υπάρχουν διαδικασίες στην ουρά αναμονής, τότε μια από αυτές μπορεί να συνεχίσει την εκτέλεσή της – είναι θέμα πολιτικής του ΛΣ ποια θα επιλεγεί.

Ακολούθησαν και μερικά άλλα εργαλεία συντονισμού, όπως: ένα γενικό εργαλείο συγχρονισμού, το οποίο καλείται «συνθήκη συντονισμού», η εντολή $monitor$ και ουρές γεγονότων, καθώς επίσης και μια υλοποίηση των εντολών σε σημαφόρους με χρήση της εντολής $test\ and\ set$ του υλικού. Τέλος, παρουσιάστηκαν και λύσεις για το συντονισμό διαδικασιών που δε συνεργάζονται και καταναμεμημένοι αλγόριθμοι για αμοιβαίο αποκλεισμό.

Βιβλιογραφία κεφαλαίου

ΠΡΟΑΙΡΕΤΙΚΗΣ ΑΝΑΓΝΩΣΗΣ

Andrews and Schneider, *Concepts and Notations for Concurrent Programming*, Computing Surveys, vol. 15, March 1983, pp. 3–43.

Ben–Ari, *Principles of Concurrent Programming*, Englewood Cliffs, NJ: Prentice Hall International, 1982.

Dubois et al., *Synchronization, Coherence, and Event Ordering in Multiprocessors*, IEEE Computers, vol. 21, Feb. 1988, pp. 9–21.

Silberschatz et al., *Operating System Concepts*, 3rd edition, reading, MA: Addison–Wesley, 1991.

Γλωσσάρι κεφαλαίου

Breakdown:	αναπάντεχο σταμάτημα
Buffer:	απομονωτής
Busy waiting:	πολύασχολη αναμονή
By reference:	με αναφορά
By value:	με αντιγραφή
Concurrent:	ταυτόχρονα
Consumer:	παραλήπτης, καταναλωτής
Critical region/section:	κρίσιμη περιοχή
Deadlock:	αδιέξοδο
Fair:	δίκαια
False:	ψευδής
Indeterminism:	απροσδιοριστία (είναι δυνατόν να συμβούν όλες οι δυνατές αναμείξεις των ακολουθιών των εντολών δύο διαδικασιών που τρέχουν παράλληλα)
Infinite postponement:	απεριόριστη αναβολή
Memory interlock, storage arbiter:	χαρακτηριστικό του υλικού με βάση το οποίο οι

	εντολές store αποκλείουν η μια την άλλη στο χρόνο, δηλαδή δεν μπορούν να γράψουν στην ίδια μεταβλητή ταυτόχρονα
Message:	μήνυμα
Monitor:	παρακολουθητής
Mutual exclusion:	αμοιβαίος αποκλεισμός
New line:	καινούρια γραμμή
Producer:	αποστολέας, παραγωγός
Reader:	αναγνώστης
Resource:	πόρος
Semaphore:	σημαφόρος
True:	αληθής
True parallelism:	οι εντολές γλώσσας μηχανής δύο παράλληλων διαδικασιών επικαλύπτονται χρονικά
Writer:	γραφέας

Διαχείριση Μνήμης (Memory Management)

Τα προγράμματα κατοικοεδρεύουν στην περιφερειακή μνήμη, στο δίσκο συνήθως, όπου:

- Παραμένουν επ' αόριστον και δεν επηρεάζονται από τη λειτουργία ή μη του Η/Υ (non-volatility).
- Η παρουσία τους έχει μικρό κόστος (σχετικά με το αντίστοιχο στην κεντρική μνήμη).

Είναι δυνατόν να μεταφερθούν, με ταχύτητα λογική αλλά κατά πολύ μικρότερη των δυνατοτήτων της CPU, στην κεντρική μνήμη για εκτέλεση.

Οι διαδικασίες, αποτέλεσμα της εκτέλεσης των προγραμμάτων, πρέπει να βρίσκονται στην κεντρική μνήμη όσο εκτελούνται, για λόγους ταχύτητας και σχεδιασμού του υπολογιστή. Όμως, όπως είδαμε στα προηγούμενα κεφάλαια, κατά την εκτέλεσή της μια διαδικασία πολλές φορές περιμένει άλλους πόρους και δεν αξιοποιεί την CPU. Επομένως, η CPU εξυπηρετεί πολλές διαδικασίες «παράλληλα». Αν οι διαδικασίες αυτές δε βρίσκονται συνεχώς στην κεντρική μνήμη, τότε χάνεται χρόνος για να μεταφερθούν από και προς το δίσκο. Αν βρίσκονται συνεχώς στη μνήμη¹, τότε σπαταλιέται μνήμη. Τέτοιου είδους προβλήματα time-space trade-offs (πάρε-δώσε μεταξύ χώρου και χρόνου) είναι κεντρικά και χαρακτηριστικά για το σχεδιασμό ΛΣ (και αρχιτεκτονικής υπολογιστών). Στο κεφάλαιο αυτό θα παρουσιάσουμε διάφορες μεθόδους αντιμετώπισης του προβλήματος αυτού, του προβλήματος της διαχείρισης της μνήμης. Δεν υπάρχει λύση-πανάκεια, βέλτιστη για όλες τις περιπτώσεις, ούτε μια γραμμική κλίμακα όπου μπορούμε να κοιτάζουμε τις λύσεις με σειρά αποτελεσματικότητας. Το ζητούμενο εκπαιδευτικό αποτέλεσμα είναι να καταλάβει ο φοιτητής ποια είναι τα trade-offs, ώστε να μπορεί να κατανοήσει και να αξιολογήσει συγκεκριμένα, πραγματικά ΛΣ.

Η κεντρική μνήμη δεν είναι ποτέ αρκετή. Αυτό δεν έχει σχέση με την τιμή ή με την ταχύτητά της, αλλά με τη *σχετική* τιμή και ταχύτητα κεντρικής και περιφερειακής μνήμης. Επομένως, το ΛΣ δε διαθέτει σε κάθε διαδικασία όση μνήμη «θα ήθελε» ούτε μπορεί να αφήσει στην κεντρική μνήμη όλες τις ενεργές διαδικασίες για όλη τη διάρκεια εκτέλεσής τους.

[1] Θα χρησιμοποιούμε τον όρο «μνήμη» όταν αναφερόμαστε στην «κεντρική μνήμη».

Σκοπός

Ο σκοπός αυτού του κεφαλαίου είναι η παρουσίαση και η σύγκριση των διαφόρων συστημάτων διαχείρισης μνήμης, με τα οποία διάφορα λειτουργικά συστήματα διαχειρίζονταν/όνται τη μνήμη. Τα περισσότερα από αυτά τα συστήματα παρουσιάζονται για διδακτικούς σκοπούς, λόγω του ότι δε χρησιμοποιούνται πλέον. Ουσιαστικά, τα συστήματα αυτά εξελίχθηκαν/εξελίσσονται στο χρόνο αντίστοιχα με την εξέλιξη και την πρόοδο της τεχνολογίας και της αρχιτεκτονικής των ΗΥ. Το πρώτο σύστημα διαχείρισης μνήμης αντιστοιχούσε στη σκέτη μηχανή. Στη συνέχεια ήταν το σύστημα του παραμένοντα επόπτη. Ακολούθησαν τα συστήματα με πολλαπλές υποδιαιρέσεις της μνήμης. Οι υποδιαιρέσεις αυτές ήταν με σταθερές ή μεταβλητές περιοχές. Τα συστήματα που αναφέραμε πιο πάνω δε χρησιμοποιούνται στους σύγχρονους υπολογιστές. Σήμερα, χρησιμοποιούνται συστήματα σελιδοποίησης και τμηματοποίησης, καθώς και συστήματα σελιδοποιημένης τμηματοποίησης και της τμηματοποιημένης σελιδοποίησης. Ανάμεσα στους σκοπούς του κεφαλαίου αυτού συμπεριλαμβάνονται και η παρουσίαση / σύγκριση αλγόριθμων οι οποίοι χρησιμοποιούνται σε καθένα από τα πιο πάνω συστήματα, καθώς και μια εισαγωγή στις βασικές έννοιες που σχετίζονται με τη μεταφορά διαδικασιών από τη μνήμη στο δίσκο, και αντίστροφα. Επίσης, θέματα χρονοδρομολόγησης διαδικασιών και θέματα που αφορούν τη διακοπή διαδικασιών συμπεριλαμβάνονται στο κεφάλαιο αυτό.

Προσδοκώμενα αποτελέσματα

Με τη μελέτη του κεφαλαίου αυτού, θα είστε σε θέση να:

- Καθορίσετε από ποια στάδια διέρχεται μια διαδικασία για να εκτελεστεί. Επίσης, πώς αναπαρίστανται οι διευθύνσεις της μνήμης στις οποίες αναφέρονται εντολές τις διαδικασίας κατά τη διάρκεια αυτών των σταδίων.
- Περιγράψετε τις λειτουργίες που γίνονται σε κάθε φάση ενός τυπικού κύκλου εκτέλεσης μιας εντολής.
- Περιγράψετε τα εξής συστήματα διαχείρισης μνήμης και εξηγήσετε τα πλεονεκτήματα και τα μειονεκτήματα καθενός από αυτά: σκέτη μηχανή, παραμένων επόπτης, με πολλαπλές υποδιαιρέσεις της μνήμης, σελιδοποίηση, τμηματοποίηση, καθώς και συστήματα σελιδοποιημένης τμηματοποίησης και τμηματοποιημένης σελιδοποίησης.
- Κατανοήσετε γιατί δεν υπάρχει σύστημα διαχείρισης μνήμης που να είναι βέλτιστο για όλες τις περιπτώσεις και να είστε σε θέση για κάθε δύο συστήματα να δώσετε ένα παράδειγμα που να παρουσιάζει κάθε σύστημα καλύτερο από το άλλο.

- Ορίσετε τι είναι και να εξηγήσετε πώς λειτουργεί το υλικό προστασίας.
- Περιγράψτε τον τρόπο που καθορίζεται μια σωστή διεύθυνση φραγής.
- Ορίσετε τι είναι μετατόπιση και από ποιους παράγοντες επηρεάζεται.
- Περιγράψτε τι συμβαίνει όταν το δέσιμο των εντολών και των δεδομένων ενός προγράμματος γίνεται κατά το χρόνο μετάφρασης και τι όταν γίνεται κατά το χρόνο φόρτωσης.
- Ορίσετε τι είναι ο μεταβατικός κώδικας επόπτη και πότε χρησιμοποιείται.
- Περιγράψτε πώς προστατεύονται οι διαδικασίες και το πρόγραμμα του επόπτη από κακόβουλη προσπάθεια πρόσβασης (από άλλες διαδικασίες) σε διευθύνσεις μνήμης που δεν έχουν το δικαίωμα.
- Ορίσετε τι είναι η εναλλαγή διαδικασιών.
- Σχολιάστε τι είναι και πώς επηρεάζει την απόδοση του συστήματος ο χρόνος εναλλαγής.
- Σχολιάστε τι είναι και πώς επηρεάζει το σύστημα η επικαλυπτόμενη εναλλαγή.
- Περιγράψτε τι είναι και πώς λειτουργεί ο αλγόριθμος πολυπρογραμματισμού σε σταθερό αριθμό διαδικασιών και τι σε μεταβλητό. Πώς επιλέγονται τα μεγέθη κάθε περιοχής; Τι είναι το σύστημα με πολλές ουρές (μια για κάθε περιοχή) και το σύστημα με μια ουρά; Πώς καθορίζεται ποια θα είναι η επόμενη διαδικασία που θα καταλάβει ένα άδειο τμήμα;
- Εξηγήστε τι θα συμβεί αν μια διαδικασία κατά τη διάρκεια της εκτέλεσής της χρειαστεί περισσότερη μνήμη. Ποιες είναι οι εναλλακτικές λύσεις;
- Εξηγήστε γιατί είναι δύσκολο να βρεθεί μια βέλτιστη στρατηγική συμπίεσης.
- Αναφέρετε τι είναι η σελιδοποίηση. Επίσης, να είστε σε θέση να πείτε ποιοι ήταν οι λόγοι που οδήγησαν σε αυτή τη λύση. Πώς υλοποιείται σε επίπεδο υλικού, τι είναι, τι εξυπηρετεί και πώς χρησιμοποιείται ο πίνακας διαδικασιών, καθώς επίσης και η συσχετιστική μνήμη;
- Προσδιορίστε πληροφορίες οι οποίες πρέπει να είναι αποθηκευμένες στον πίνακα σελίδων.
- Αναλύστε τα πλεονεκτήματα και τα μειονεκτήματα της χρήσης πίνακα σελίδων και πίνακα πλαισίων.
- Εξηγήστε πώς επιτυγχάνεται η προστασία των σελίδων που ανήκουν σε μια διαδικασία και να σχολιάσετε τι συμβαίνει όταν υπάρχει ανάγκη για διαμοιρασμό σελίδων.

- Εξηγήστε γιατί η καλύτερη επιλογή για το μέγεθος της σελίδας είναι να είναι δύναμη του 2, και να γιατί δεν υπάρχει βέλτιστο μέγεθος σελίδας.
- Εξηγήστε γιατί χρειάστηκε να γίνεται διαχείριση της μνήμης μέσω τμηματοποίησης και πώς λειτουργεί ένα σύστημα το οποίο υποστηρίζει τμηματοποίηση, καθώς και πώς παρέχεται προστασία στις διαδικασίες και τι συμβαίνει όταν υπάρχει ανάγκη διαμοιρασμού διαφόρων τμημάτων διαδικασιών.
- Καθορίστε τι εξυπηρετεί η τμηματοποιημένη σελιδοποίηση και τι η σελιδοποιημένη τμηματοποίηση και να τις συγκρίνετε.
- Ορίστε τι είναι η εσωτερική και εξωτερική κλασματοποίηση. Πώς εμφανίζονται και γιατί στις περιπτώσεις της σελιδοποίησης και της τμηματοποίησης;

Έννοιες κλειδιά

- Διαχείριση μνήμης
- Συμβολικές και μετατοπιζόμενες διευθύνσεις
- Σκέτη μηχανή
- Παραμένων επόπτης
- Διεύθυνση φραγής
- Καταχωρητής βάσης, καταχωρητής ορίου
- Εναλλαγή, χρόνος εναλλαγής, επικαλυπτόμενη εναλλαγή
- Επικουρική μνήμη
- Διαίρεση μνήμης σε στατικές και δυναμικές περιοχές
- Πολυπρογραμματισμός με σταθερό αριθμό διαδικασιών (ΠΣΔ)
- Πολυπρογραμματισμός με μεταβλητό αριθμό διαδικασιών (ΠΜΔ)
- Προτεραιότητα διαδικασίας
- Πολιτικές και αλγόριθμοι διαχείρισης μνήμης: καλύτερου ταιριάσματος μόνο, καλύτερου διαθέσιμου ταιριάσματος
- Χρονοδρομολόγηση – Αλγόριθμοι χρονοδρομολόγησης Round – robin, FCFS
- Ενάλλαξε μέσα – Ενάλλαξε έξω
- Εσωτερική και εξωτερική κλασματοποίηση

- *Οπή*
- *Συμπίεση*
- *Σελίδα, σελιδοποίηση, πλαίσιο σελίδας, πίνακας σελίδων και τμημάτων*
- *Συσχετιστικοί καταχωρητές (κρυφή μνήμη)*
- *Προστασία*
- *Μετατόπιση*
- *Τμηματοποίηση*
- *Τμηματοποιημένη σελιδοποίηση*
- *Σελιδοποιημένη τμηματοποίηση*

Δομή κεφαλαίου

Το κεφάλαιο αυτό περιέχει τις παρακάτω ενότητες:

4.1 Εισαγωγικά

4.2 Γυμνή Μηχανή

4.3 Παραμένων Επόπτης

4.4 Εναλλαγή

4.5 Πολλαπλές Υποδιαιρέσεις Μνήμης

4.6 Σελιδοποίηση

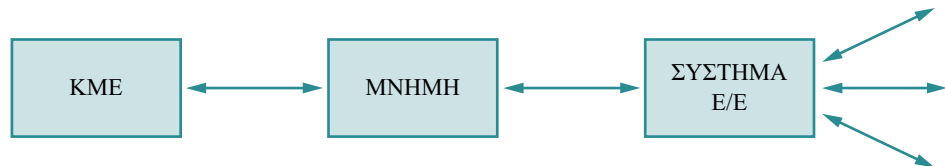
4.7 Τμηματοποίηση

4.8 Συνδυασμένα Συστήματα

4.1 Εισαγωγικά (Preliminaries)

Η μνήμη έχει καθοριστικό ρόλο στη λειτουργία ενός σύγχρονου ΛΣ. Όπως φαίνεται στο Σχήμα 4.1, η κεντρική μονάδα επεξεργασίας και το σύστημα εισόδου/εξόδου (I/O) αλληλεπιδρούν με τη μνήμη. Τη μνήμη μπορούμε να τη θεωρήσουμε ως ένα μεγάλο διάνυσμα από bytes, όπου το καθένα από αυτά τα bytes έχει τη δική του διεύθυνση. Οι διευθύνσεις αυτές είναι ένας αύξων αριθμός, ο οποίος αρχίζει από το μηδέν και φτάνει μέχρι το μέγιστο αριθμό bytes στη μνήμη, δηλαδή το μέγεθος της μνήμης. Οι αριθμοί που αντιστοιχούν στις διευθύνσεις αυτές είναι στο δυαδικό σύστημα αρίθμησης, κι αυτό γιατί το ΛΣ και το υλικό είναι έτσι σχεδιασμένα. Αν η μνήμη είναι μεγέθους M και η διεύθυνση κάθε byte της μνήμης χρειάζεται m bits, τότε το m επιλέγεται έτσι ώστε $2^m = M$. Η αλληλεπίδραση της μνήμης με την CPU και το σύστημα I/O επιτυγχάνεται μέσω ακολουθιών από αναγνώσεις / εγγραφές από/σε συγκεκριμένες διευθύνσεις μνήμης. Για παράδειγμα, η CPU φέρνει (fetches) από τη μνήμη δεδομένα και αποθηκεύει (stores) δεδομένα στη μνήμη.

Σχήμα 4.1
Ο κεντρικός ρόλος
της μνήμης σε ένα
υπολογιστικό
σύστημα

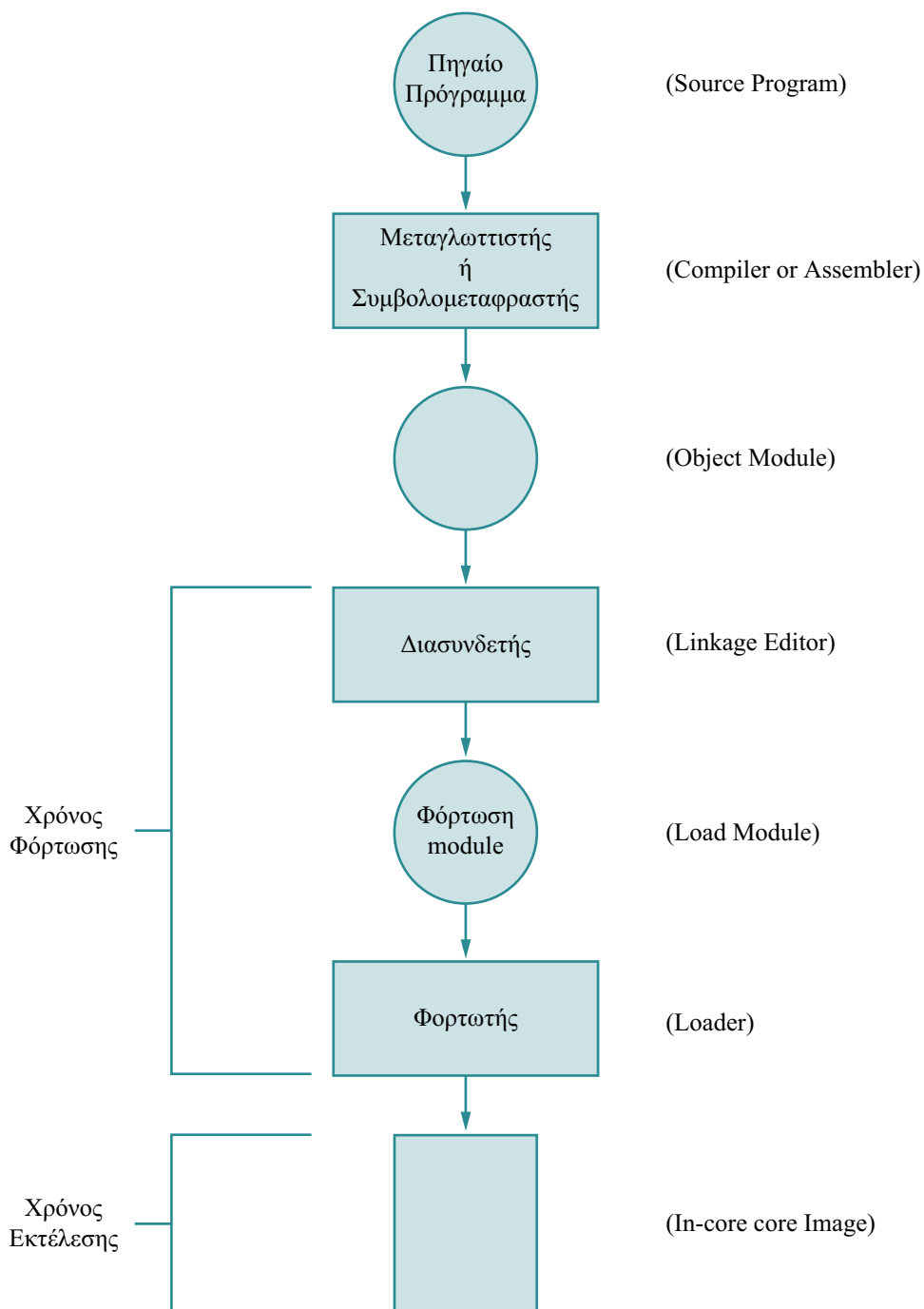


Έστω μια διαδικασία η οποία αντιστοιχεί σε ένα πρόγραμμα χρήστη (μια εφαρμογή γραμμένη από ένα χρήστη). Στις πιο πολλές περιπτώσεις, η διαδικασία αυτή χρειάζεται να διέλθει από διάφορα στάδια πριν εκτελεστεί. Το Σχήμα 4.2 παρουσιάζει τα στάδια αυτά και τη χρονολογική σειρά με την οποία εκτελούνται. Στο κεφάλαιο αυτό μάς ενδιαφέρει πώς αναπαρίστανται οι διευθύνσεις της μνήμης στις οποίες αναφέρονται οι εντολές της διαδικασίας, κατά τη διάρκεια αυτών των σταδίων. Αρχικά, στο «πηγαίο πρόγραμμα» (source program) οι διευθύνσεις είναι γενικά συμβολικές. Αυτό σημαίνει ότι στο πηγαίο πρόγραμμα ο χρήστης χρησιμοποιεί σύμβολα για να αναφερθεί στην τιμή μιας μεταβλητής. Για παράδειγμα, έστω X μια μεταβλητή τύπου ακεραίου. Όταν ο χρήστης θέλει να αλλάξει την τιμή της μεταβλητής X , δεν τον ενδιαφέρει σε ποια διεύθυνση της μνήμης είναι αποθηκευμένη η τιμή αυτής της μεταβλητής, έτσι ώστε να έχει πρόσβαση στη συγκεκριμένη θέση στη μνήμη και να την αλλάξει. Ο χρήστης χρησιμοποιεί απλώς ένα σύμβολο, το X , για να αναφερθεί και να τροποποιήσει τη μεταβλητή αυτή. Το επόμενο στάδιο αφορά τη μεταγλώττιση του πηγαίου προγράμματος. Ένας μεταγλωττιστής / μεταφραστής θα «συνδέσει» [δεσμεύσει] (bind) αυτές τις συμβολικές διευθύνσεις σε «μετατοπιζόμενες διευθύνσεις»

(relocatable addresses). Όταν το πρόγραμμα βρίσκεται σε «μορφή αντικειμένου» (object module), θεωρεί ότι θα του δοθεί ένα συνεχόμενο κομμάτι μνήμης. Οι διευθύνσεις, λοιπόν, σε αυτό το στάδιο είναι μετατοπιζόμενες, αυτό σημαίνει ότι κάθε μεταβλητή του πηγαίου προγράμματος έχει αντιστοιχιστεί σε μια διεύθυνση, π.χ. η μεταβλητή *X* έχει διεύθυνση 01110 = το 14ο byte από την αρχή του κομματιού της μνήμης που έχει δοθεί στο πρόγραμμα. Ο «διασυνδετής» (linkage editor) ή ο «φορτωτής» (loader), με τη σειρά του, θα συνδέσει τις μετατοπιζόμενες διευθύνσεις με «απόλυτες διευθύνσεις» (absolute addresses) (π.χ. 000010000100000 = 10560 byte της μνήμης – το μέγεθος αυτής της μνήμης είναι 32768 bytes). Κάθε σύνδεση είναι μια αντιστοίχιση από τον ένα «χώρο διευθύνσεων» (address space) σε έναν άλλο.

Αφού οι διευθύνσεις μιας διαδικασίας / προγράμματος έχουν αντιστοιχιστεί σε απόλυτες διευθύνσεις μνήμης, τότε, για να μπορέσει να εκτελεστεί, πρέπει να φορτωθεί στη μνήμη. Κατά τη διάρκεια εκτέλεσης μιας διαδικασίας ουσιαστικά εκτελούνται σειριακά οι εντολές που αντιστοιχούν σε αυτή. Οι εντολές αυτές και τα δεδομένα που τυχόν χρειάζονται (οι εντολές) βρίσκονται στη μνήμη. Στην επόμενη παράγραφο θα περιγράψουμε τον τυπικό «κύκλο εκτέλεσης μιας εντολής» (instruction cycle). Όταν μια διαδικασία τελειώσει την εκτέλεσή της, τότε προφανώς ελευθερώνει το χώρο που καταλάμβανε στη μνήμη. Ο χώρος αυτός δηλώνεται διαθέσιμος και η επόμενη διαδικασία η οποία είναι έτοιμη για εκτέλεση μπορεί να φορτωθεί και να εκτελεστεί. Η επιλογή της επόμενης διαδικασίας που θα φορτωθεί στη μνήμη γίνεται από ένα μέρος του ΛΣ το οποίο ονομάζεται «χρονοδρομολογητής / χρονοπρογραμματιστής» με βάση κάποιο αλγόριθμο χρονοδρομολόγησης (στα πρώτα τρία συστήματα δεν υπήρχε πολυπρογραμματισμός και η μεταφορά των διαδικασιών από τη μνήμη στο δίσκο γινόταν από το χρήστη, τον παραμένοντα επόπτη και το διεκπεραιωτή αντίστοιχα). Σε κάθε σύστημα διαχείρισης μνήμης αντιστοιχεί ένα σύνολο τέτοιων αλγόριθμων. Το ΛΣ μπορεί να εκτελεί περισσότερους από έναν (από αυτούς τους αλγόριθμους), π.χ. σε περιπτώσεις όπου οι διαδικασίες οι οποίες είναι έτοιμες για εκτέλεση μπαίνουν σε διαφορετικές ουρές (δες Ενότητα 4.5). Οι αλγόριθμοι αυτοί επιλέγονται από το σχεδιαστή του ΛΣ.

Στο σημείο αυτό θα περιγράψουμε σε υψηλό επίπεδο έναν τυπικό κύκλο εκτέλεσης μιας εντολής από την CPU. Πιο ακριβή και λεπτομερή περιγραφή θα βρείτε σε βιβλία αρχιτεκτονικής υπολογιστών.



Σχήμα 4.2
Πολλαπλών βημά-
των επεξεργασία
ενός προγράμμα-
τος χρήστη

1. Πρώτα από όλα η μονάδα ελέγχου^[2] φέρνει / διαβάζει εντολή από τη μνήμη στον καταχωρητή εντολών IR.^[3]
2. Αλλάζει τον καταχωρητή PC^[4] να δείχνει τη διεύθυνση της επόμενης εντολής.
3. Αποφασίζει το είδος της εντολής.
4. Αν η εντολή χρησιμοποιεί δεδομένα από τη μνήμη, αποφασίζει σε ποια διεύθυνση βρίσκονται.
5. Φέρνει τα δεδομένα, όταν χρειάζονται, από τη μνήμη σε καταχωρητές της CPU.
6. Εκτελεί την εντολή.
7. Αποθηκεύει τα αποτελέσματα στον κατάλληλο χώρο (καταχωρητή ή μνήμη).
8. Πηγαίνει στο 1 και ξεκινάει την εκτέλεση της επόμενης εντολής.

Η κάθε εντολή είναι ουσιαστικά ένας δυαδικός αριθμός ο οποίος χρησιμοποιείται για να καθοριστεί μια λειτουργία και οι «τελεστέοι» (operands) (ή δεδομένα) που θα χρησιμοποιηθούν από αυτή τη λειτουργία. Στα βήματα 3 και 4 στην περιγραφή του πιο πάνω κύκλου εκτέλεσης μιας εντολής ουσιαστικά γίνεται η αποκωδικοποίηση της εντολής. Η λειτουργία και οι τελεστέοι της περιγράφονται από συγκεκριμένα πεδία της εντολής. Το πεδίο της λειτουργίας καλείται «κωδικός λειτουργίας»^[5] (opcode) και αντιπροσωπεύεται από τα πρώτα x bits (π.χ. $x = 8$) του αριθμού δυαδικού αριθμού που αντιστοιχεί στην εντολή. Αφού είναι πλέον γνωστό για ποια λειτουργία πρόκειται, τα υπόλοιπα bits του αριθμού που αντιστοιχεί στην εντολή θα προσδιορίσουν τους τελεστέους της. Τα bits που αντιστοιχούν σε καθέναν από τους τελεστέους αντιπροσωπεύουν συνήθως διευθύνσεις στην κεντρική μνήμη (όπου, για παράδειγμα, είναι αποθηκευμένη η τιμή μιας μεταβλητής) ή δηλώνουν κάποιον καταχωρητή ειδικού σκοπού της CPU. Οι εντολές διαιρούνται σε τρεις κατηγορίες: (1) καταχωρητή – καταχωρητή, (2) καταχωρητή – μνήμης, (3) μνήμης – μνήμης. Ανάλογα με την κατηγορία όπου ανήκει μια εντολή προσδιορίζονται οι τελεστέοι της.

[2] Η «μονάδα ελέγχου» (control unit) φέρνει εντολές από τη μνήμη και αποφασίζει το είδος τους.

[3] Ο «καταχωρητής εντολών IR» (instruction register) περιέχει κάθε στιγμή την εντολή που εκτελείται από την CPU εκείνη τη στιγμή.

[4] Ο «μετρητής προγράμματος PC» (program counter register) περιέχει κάθε στιγμή τη διεύθυνση της εντολής που εκτελείται από την CPU εκείνη τη στιγμή, έτσι ώστε μετά την εκτέλεσή της να δείχνει τη διεύθυνση της επόμενης εντολής, προκειμένου αυτή να ανακληθεί και να εκτελεστεί.

[5] Ο «κωδικός λειτουργίας» (opcode) είναι ένας δυαδικός αριθμός. Το μέγεθος του αριθμού αυτού είναι ανάλογο με το πόσο μεγάλο σύνολο λειτουργιών έχει κάθε υπολογιστής (instruction set). Ο καθένας από αυτούς τους αριθμούς αντιστοιχεί σε μια ξεχωριστή λειτουργία, όπως πρόσθεση δύο μεταβλητών, πρόσθεση μιας μεταβλητής και μιας σταθεράς, αποθήκευση δεδομένων στη μνήμη, μετακίνηση μιας μεταβλητής σε μια άλλη και άλλες.

Το πλήθος, το είδος και η θέση των τελεστών αυτών προσδιορίζεται επακριβώς από τον κωδικό λειτουργίας και την αρχιτεκτονική του συστήματος. Στην πιο πάνω περιγραφή παραλείψαμε σκόπιμα λεπτομέρειες, γιατί, όπως είπαμε και προηγουμένως, αυτές θα τις μελετήσετε σε ένα μάθημα αρχιτεκτονικής.

Σε αυτό το κεφάλαιο εξετάζουμε πολλά διαφορετικά συστήματα διαχείρισης μνήμης. Σε καθένα από αυτά τα συστήματα θα αναφερθούμε σε ένα σύνολο αλγόριθμων χρονοδρομολόγησης, οι οποίοι χρησιμοποιούνται για να καθορίσουν ποιες διαδικασίες θα βρίσκονται στη μνήμη κάθε χρονική στιγμή, και σε ένα σύνολο αλγόριθμων / πολιτικών, οι οποίοι θα καθορίζουν σε ποια/ες περιοχή/-ές της μνήμης θα βρίσκεται η κάθε διαδικασία (ή μέρος της διαδικασίας). Η αποδοτικότητα των αλγόριθμων αυτών ορίζεται ξεχωριστά σε κάθε σύστημα, λόγω του ότι σε καθένα από αυτά τα συστήματα υπάρχουν επιμέρους διαφορετικοί στόχοι. Για παράδειγμα, σε ένα σύστημα όπου η μνήμη υποδιαιρείται σε μεταβλητά τμήματα το κριτήριο στην απόδοση των αλγόριθμων σε αυτό το σύστημα θα ήταν ο όσο το δυνατόν μικρότερος εξωτερικός κατακερματισμός (δες Ενότητα 4.5.4), ενώ σε ένα σύστημα μνήμης με σελιδοποίηση ο καλύτερος αλγόριθμος θα ήταν αυτός που θα έκανε τα λιγότερα λάθη σελίδων (δες Ενότητα 4.6). Η επιλογή κάποιου συστήματος διαχείρισης μνήμης για έναν καθορισμένο Η/Υ εξαρτάται από πολλούς παράγοντες, αλλά κύρια από την αρχιτεκτονική του υπολογιστή αυτού. Σε μερικές περιπτώσεις, οι αλγόριθμοι που αντιστοιχούν στα συστήματα μνήμης απαιτούν τη δική τους υποστήριξη από το υλικό. Ξεκινάμε με την πιο απλή αρχιτεκτονική και υλικό και συνεχίζουμε προς τα πιο πολύπλοκα και εξελιγμένα συστήματα.

Στο σημείο αυτό θέλουμε να κάνουμε την εξής επισήμανση. Σε κάθε σύστημα χρησιμοποιούμε παραδείγματα τα οποία αφορούν / αντικατοπτρίζουν τις συνθήκες, τα δεδομένα και τις δυνατότητες των εκάστοτε υπολογιστικών συστημάτων. Θεωρούμε σκόπιμο κάτι τέτοιο, κι ας φαίνονται εκτός πραγματικότητας τα παραδείγματά μας, γιατί, αν τα δεδομένα που θα χρησιμοποιούσαμε θα ήταν με βάση τη σημερινή τεχνολογία, τότε ίσως και τα συμπεράσματα, τα μειονεκτήματα και τα πλεονεκτήματα κάθε συστήματος να ήταν διαφορετικά. Φυσικά, παραθέτουμε σε αρκετά σημεία αντίστοιχα παραδείγματα ή ενδείξεις για το τι συμβαίνει σε ένα σύγχρονο υπολογιστικό σύστημα.

4.2 Σκέτη μηχανή (Bare Machine)

Σε αυτή την ενότητα, καθώς και στις Ενότητες 4.3, 4.4 και 4.5, παρουσιάζουμε τρία συστήματα διαχείρισης μνήμης τα οποία δε χρησιμοποιούνται πλέον. Σας τα παρουσιάζουμε όμως για διδακτικούς λόγους και για να γνωρίσετε βήμα βήμα τις εξελίξεις και τους λόγους που οδήγησαν από το ένα σύστημα στο άλλο.

Φυσικά, ο πιο απλός τρόπος διαχείρισης μνήμης είναι να μην υπάρχει διαχείριση

μνήμης. Ο χρήστης εφοδιάζεται με τη μηχανή και έχει απόλυτο έλεγχο πάνω σε ολόκληρο το χώρο της μνήμης (Σχήμα 4.3).



Σχήμα 4.3

Σκέτη μηχανή

Αυτή η προσέγγιση έχει ορισμένα αναμφισβήτητα πλεονεκτήματα. Δίνει στο χρήστη μέγιστη ευελιξία, ο οποίος μπορεί να ελέγχει τη χρήση της μνήμης με όποιον τρόπο θέλει. Έχει μέγιστη απλότητα και ελάχιστο κόστος. Δεν υπάρχει ανάγκη για ειδικό υλικό ούτε για λογισμικό λειτουργικού συστήματος.

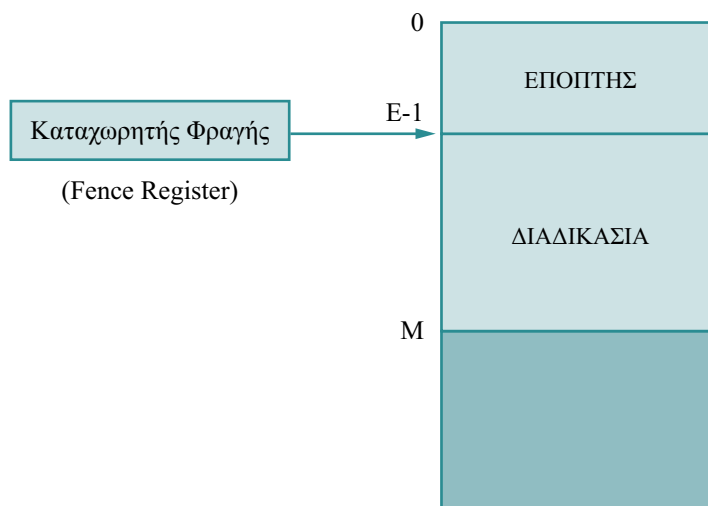
Αυτό το σύστημα έχει όμως και περιορισμούς: Δεν προσφέρει καμιά υπηρεσία. Ο χρήστης έχει απόλυτο έλεγχο πάνω στον υπολογιστή, αλλά το ΛΣ δεν έχει τον έλεγχο των διακοπών^[6] ούτε «παραμένοντα επόπτη» (resident monitor) (δες Ενότητα 4.3) για την επεξεργασία των «κλήσεων συστήματος»^[7] (system calls) ή των λαθών και άλλα. Γενικά, χρησιμοποιείται μόνο σε «εξειδικευμένα συστήματα» (dedicated systems), όπου οι χρήστες απαιτούν ευελιξία και απλότητα και είναι διατεθειμένοι να γράψουν τις δικές τους ρουτίνες υποστήριξης.

[6] Οι «διακοπές» (interrupts) είναι σήματα τα οποία φτάνουν στο μικροεπεξεργαστή για να τον ενημερώσουν ότι κάτι συμβαίνει στο σύστημα. Για παράδειγμα, διακοπές μπορεί να είναι γραμμές εισόδου στο μικροεπεξεργαστή από μονάδες I/O. Ο μικροεπεξεργαστής λείει σε μια μονάδα I/O να ξεκινήσει μια λειτουργία και ακολούθως αυτό ασχολείται με κάτι άλλο. Όταν η μονάδα I/O συμπληρώσει την εργασία της, τότε ένα chip ελέγχου διακόπτει το μικροεπεξεργαστή για να ενημερωθεί για τον τερματισμό της εργασίας I/O. Υπάρχουν και άλλοι λόγοι για τους οποίους συμβαίνουν διακοπές, όπως η προσπάθεια μιας διαδικασίας να έχει πρόσβαση στο χώρο διευθύνσεων μιας άλλης διαδικασίας και λοιπά.

[7] Οι «κλήσεις συστήματος» (system calls) είναι οι εντολές οι οποίες χρησιμοποιούνται για να επιτευχθεί επικοινωνία μεταξύ του ΛΣ και των προγραμμάτων του χρήστη (διαδικασίες). Οι κλήσεις αυτές συνήθως δημιουργούν και καταστρέφουν διάφορες οντότητες τις οποίες διαχειρίζεται το ΛΣ. Οι πιο σημαντικές από αυτές τις οντότητες είναι οι διεργασίες και τα αρχεία.

4.3 Παραμένων Επόπτης (Resident Monitor)

Το επόμενο σε πολυπλοκότητα σύστημα είναι να διαιρεθεί η μνήμη σε δύο περιοχές, μία για μια μόνο διαδικασία (πρόγραμμα χρήστη) η οποία βρίσκεται στη μνήμη και μία για τον «παραμένοντα επόπτη» του ΛΣ (Σχήμα 4.4). Το ίδιο το ΛΣ είναι ένα πρόγραμμα, και φυσικά, όταν εκτελείται, πρέπει να βρίσκεται στην κεντρική μνήμη. Ακόμη και όταν εκτελείται μια διαδικασία, ένα μικρό μέρος του ΛΣ πρέπει να βρίσκεται στη μνήμη για να «εποπτεύει» την εκτέλεση της διαδικασίας (π.χ. για να το σταματήσει, αν χρειαστεί, για να ζητήσει για λογαριασμό της υπηρεσίες, π.χ. κάποια λειτουργία εισόδου/εξόδου, ή πόρους, π.χ. έναν εκτυπωτή κτλ.). Αυτό το ελάχιστο μέρος του ΛΣ που πρέπει πάντοτε να βρίσκεται στην κεντρική μνήμη ονομάζεται «παραμένων επόπτης» και συνήθως τοποθετείται στην αρχή ή στο τέλος της μνήμης. Αυτό θα πει ότι, αν ο «παραμένων επόπτης» χρειάζεται E bytes για να αποθηκευτεί στη μνήμη, τότε τα πρώτα E bytes δίνονται σε αυτόν (δηλαδή αυτά που έχουν διευθύνσεις από 0_{10} έως $E-1_{10}$ – χρησιμοποιούμε ακέραιους σε αυτό το σημείο, για να είναι πιο εύκολα κατανοητές) αν τοποθετείται στην αρχή, ή στα E τελευταία bytes αν τοποθετείται στο τέλος. Εδώ θεωρούμε ότι είναι στην αρχή.



Σχήμα 4.4

*Παραμένων
επόπτης*

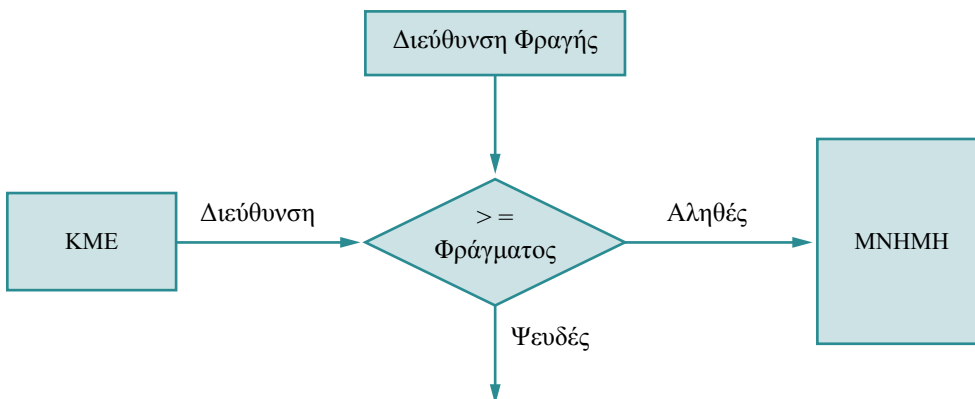
Αυτή η προσέγγιση χρησιμοποιήθηκε στο Fortran Monitoring System για τον 7094, ένα από τα πρώτα ΛΣ. Είναι η κύρια προσέγγιση για πολλά σύγχρονα συστήματα μικροϋπολογιστών, όπως το CP/M.

4.3.1 Υλικό προστασίας (Protection Hardware)

Έστω ότι η μνήμη του συστήματος έχει M θέσεις και ότι ο επόπτης βρίσκεται στις αρχικές θέσεις μνήμης ($0 - E-1$). Άρα η διαδικασία βρίσκεται στις θέσεις E έως $M-1$.

Χρειάζεται να προστατέψουμε τον κώδικα και τα δεδομένα του επόπτη από αλλαγές (εκούσιες ή ακούσιες) που μπορεί να προκαλέσει η διαδικασία, δηλαδή να εμποδίσουμε την προσπέλαση στις πρώτες E θέσεις της μνήμης από τη διαδικασία η οποία θα βρίσκεται στη μνήμη για να εκτελεστεί.

Αυτή η προστασία πρέπει να προσφερθεί από το υλικό και μπορεί να υλοποιηθεί με αρκετούς τρόπους. Η γενική προσέγγιση δίνεται στο Σχήμα 4.5. Κάθε διεύθυνση (εντολής ή δεδομένων) που παράγεται από τη διαδικασία συγκρίνεται με μια «διεύθυνση φραγής» (fence address). Αν η δημιουργημένη διεύθυνση είναι μεγαλύτερη ή ίση με τη διεύθυνση φραγής, τότε είναι μια επιτρεπτή αναφορά σε μνήμη χρήστη και στέλνεται στη μονάδα μνήμης, ως συνήθως. Αν η διεύθυνση που παράχθηκε είναι μικρότερη αυτής της φραγής, τότε είναι μια απαγορευμένη αναφορά σε μνήμη επόπτη. Η αναφορά διακόπτεται και μια «παγίδα» (trap or interrupt) προς το ΛΣ δημιουργείται (λάθος διευθυνσιοδότησης). Το ΛΣ τότε θα κάνει την κατάλληλη ενέργεια (συνήθως τερματισμός του προγράμματος με κατάλληλο μήνυμα λάθους (π.χ. «This program has performed an illegal operation and will be shut down») και αντιγραφή του «περιεχομένου της μνήμης» (memory dump) στο δίσκο για πιθανό εκ των υστέρων έλεγχο.



Σχήμα 4.5

Προστασία διευθύνσεων από το υλικό για έναν παραμένοντα επόπτη

Παρατηρήστε ότι κάθε αναφορά στη μνήμη από τη διαδικασία πρέπει να ελεγχθεί. Γενικά, αυτός ο έλεγχος θα καθυστερήσει κάθε προσπέλαση μνήμης κατά το χρόνο σύγκρισης, έτσι μια αναφορά στη μνήμη μπορεί να χρειαστεί 995ns αντί για 980ns. Προσεκτική σχεδίαση κυκλωμάτων θα μπορούσε να επικαλύψει τη σύγκριση με άλλες δραστηριότητες, για να μειώσει τον «ισοδύναμο χρόνο προσπέλασης» (effective access time).

Το ΛΣ, όταν εκτελεί σε «καθεστώς επόπτη» (monitor mode), έχει απεριόριστη προσπέλαση στη μνήμη και του επόπτη και της διαδικασίας. Αυτό επιτρέπει στο ΛΣ να φορτώσει τη διαδικασία στη μνήμη της διαδικασίας, να τη διακόψει σε περίπτωση

λαθών, να προσπελάσει και να αλλάξει τις παραμέτρους των κλήσεων συστήματος κτλ.

Μια κύρια διαφορά των υπολογιστικών συστημάτων είναι ο τρόπος με τον οποίο ορίζεται η διεύθυνση φραγής. Μια προσέγγιση είναι να σχεδιαστεί στο υλικό ως «συγκεκριμένη σταθερά» (fixed constant). Για παράδειγμα, στον HP2116B υπολογιστή ο «βασικός δυαδικός φορτωτής» (basic binary loader) αποθηκευόταν στις θέσεις 77700₈ έως 77777₈ (τελευταίες θέσεις μνήμης μιας μηχανής 32K bytes). Κατά τη διάρκεια εκτέλεσης προγραμμάτων του χρήστη δεν ήταν δυνατόν να προσπελαστεί οποιαδήποτε θέση πάνω από τη θέση 77700₈, το φράγμα ήταν χτισμένο στο υλικό.

Το ελάττωμα μιας αμετάβλητης, προσδιορισμένης από το υλικό, διεύθυνσης φραγής είναι ότι καμιά επιλογή δεν είναι βέλτιστη για όλες τις περιπτώσεις, με δεδομένο μάλιστα ότι τα ΛΣ αλλάζουν στο χρόνο, ενώ το υλικό παραμένει σταθερό. Αν η διεύθυνση φραγής είναι σχεδόν ίση με το μέγεθος του επόπτη, τότε δεν υπάρχει χώρος για επέκτασή του. Υπάρχουν όμως περιπτώσεις που χρειάζεται να αλλάξει το μέγεθος του επόπτη (συνεπώς και η θέση του φράγματος) κατά την εκτέλεση του προγράμματος. Π.χ. ο επόπτης περιέχει κώδικα και «μνήμη απομονωτών» (buffer space) για «οδηγούς μονάδων» (device drivers)^[8]. Εάν ένας οδηγός μονάδας (ή άλλη υπηρεσία του ΛΣ) δε χρησιμοποιείται, είναι ανεπιθύμητη η αποθήκευση του κώδικα και των δεδομένων του στη μνήμη, αφού μπορεί να χρησιμοποιήσουμε αυτό το χώρο για άλλους σκοπούς. Τέτοιου είδους κώδικας μερικές φορές ονομάζεται «μεταβατικός κώδικας επόπτη» (transient monitor code), επειδή έρχεται και φεύγει ανάλογα με την ανάγκη που υπάρχει. Άρα η χρήση τέτοιου κώδικα αλλάζει το μέγεθος του επόπτη κατά την εκτέλεση του προγράμματος (του επόπτη).

Αν η διεύθυνση φραγής είναι πολύ μεγαλύτερη από το μέγεθος του επόπτη, τότε σπαταλιέται κάποια μνήμη. Για να λυθεί αυτό το πρόβλημα χρησιμοποιείται συνήθως ένας καταχωρητής φραγής που περιέχει τη διεύθυνση φραγής και χρησιμοποιείται στον έλεγχο ορθότητας όλων των αναφορών της μνήμης της διαδικασίας. Μπορεί να φορτωθεί από το ΛΣ με τη χρήση μιας ειδικής «προνομιακής χρήσης εντολής» (special privilege instruction). Ο καταχωρητής φραγής μπορεί να φορτωθεί μόνο από το πρόγραμμα που εκτελείται σε καθεστώς επόπτη. Αυτό το σύστημα επιτρέπει στον επόπτη να αλλάζει την τιμή του καταχωρητή φραγής όποτε το μέγεθός του αλλάζει.

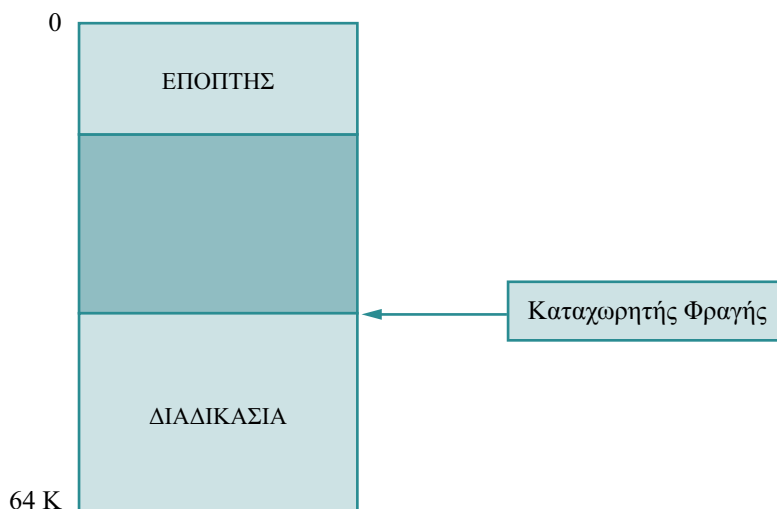
4.3.2 Μετατόπιση (Relocation)

Ένα άλλο πρόβλημα είναι το φόρτωμα των διαδικασιών. Παρ' όλο που ο χώρος διευ-

[8] Οι «οδηγοί συσκευών» (device drivers) είναι χαμηλού επιπέδου λογισμικό το οποίο διαχειρίζεται τις συσκευές εισόδου/εξόδου. Κάθε κατηγορία τέτοιων συσκευών χρειάζεται το δικό της λογισμικό.

θύνσεων αρχίζει στο 0, η πρώτη διεύθυνση της διαδικασίας δεν είναι αυτή, αλλά η πρώτη διεύθυνση μετά το φράγμα. Αυτή η διάταξη μπορεί να επηρεάσει τις διευθύνσεις που χρησιμοποιεί η διαδικασία. Η σύνδεση των εντολών και των δεδομένων με διευθύνσεις μνήμης μπορεί να γίνει είτε κατά το χρόνο μετάφρασης είτε κατά το χρόνο φόρτωσης. Αν η διεύθυνση φραγής είναι γνωστή κατά το χρόνο μετάφρασης, μπορεί να δημιουργηθεί «κώδικας απόλυτων διευθύνσεων» (absolute code). Ο κώδικας αυτός θα ξεκινάει από το φράγμα και πέρα. Αν, όμως, αλλάξει η διεύθυνση φραγής, ο κώδικας πρέπει να μεταφραστεί ξανά. Εναλλακτικά, ο μεταφραστής μπορεί να παράγει «μετατοπίσιμο κώδικα» (relocatable code). Σε αυτή την περίπτωση, η σύνδεση γίνεται κατά το χρόνο φόρτωσης. Αν αλλάξει η διεύθυνση φραγής, αρκεί η επαναφόρτωση του κώδικα για να συμπεριλάβει την αλλαγή. Και στις δύο περιπτώσεις, πάντως, το φράγμα πρέπει να είναι σταθερό κατά την εκτέλεση της διαδικασίας. Είναι φανερό ότι, αν οι διευθύνσεις της διαδικασίας έχουν αντιστοιχιστεί σε φυσικές (απόλυτες) διευθύνσεις βάσει της θέσης του φράγματος, τότε αυτές οι διευθύνσεις δεν ισχύουν αν αλλάξει θέση το φράγμα. Συνεπώς, το φράγμα μπορεί να μετατοπιστεί μόνο όταν καμιά διαδικασία δεν εκτελείται.

Έχουν χρησιμοποιηθεί δύο τρόποι για να μεταβληθεί το βασικό σχήμα που παρουσιάσαμε ώστε να επιτραπεί να αλλάζει το μέγεθος του επόπτη δυναμικά. Ένα αρχικό ΛΣ για τον PDP – 11 χρησιμοποιούσε την προσέγγιση του Σχήματος 4.6. Η διαδικασία φορτωνόταν από την υψηλή μνήμη και κάτω, προς το φράγμα, αντί να ξεκινάει από το φράγμα και να εκτείνεται προς την υψηλή μνήμη. Το πλεονέκτημα είναι ότι όλος ο ελεύθερος χώρος είναι στη μέση και είτε η διαδικασία είτε ο επόπτης μπορούν να επεκταθούν σε αυτή τη μνήμη ανάλογα με τις ανάγκες τους.

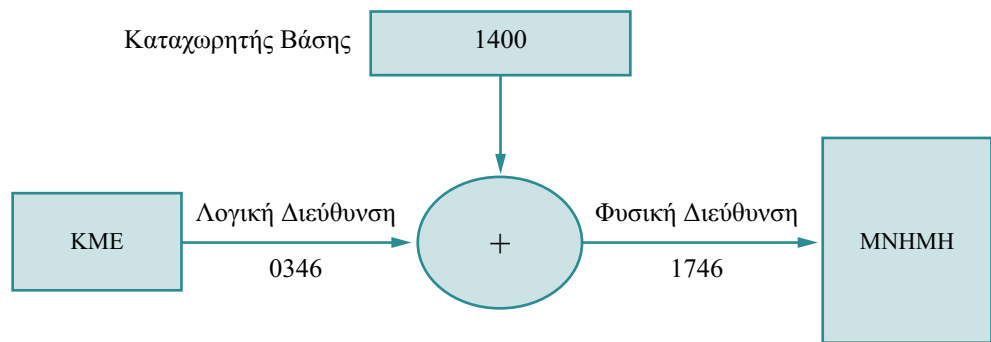


Σχήμα 4.6

Φόρτωση
του χρήστη σε
υψηλή μνήμη

Μια πιο γενική προσέγγιση που χρησιμοποιήθηκε στους υπολογιστές CDC6600 είναι να καθυστερεί το «δέσιμο» των διευθύνσεων έως το χρόνο εκτέλεσης. Αυτό το σχήμα «δυναμικής μετατόπισης» (dynamic relocation) απαιτεί ελαφρά διαφορετική υποστήριξη υλικού, όπως φαίνεται στο Σχήμα 4.7. Ο καταχωρητής φραγής ονομάζεται τώρα «καταχωρητής μετατόπισης ή βάσης» (relocation or base register). Η τιμή του καταχωρητή προστίθεται σε κάθε διεύθυνση παραγόμενη από μια διαδικασία, τη στιγμή που αυτή στέλνεται στη μνήμη. Για παράδειγμα, αν το φράγμα είναι στο 1400, τότε μια προσπάθεια αναφοράς από τη διαδικασία στη θέση 0 μετατοπίζεται δυναμικά στη θέση 1400, μια προσπέλαση στη θέση 346 μετατοπίζεται στη θέση 1746.

Σχήμα 4.7
Δυναμική μετατόπιση με τη χρήση καταχωρητή μετατόπισης



Παρατηρήστε ότι η διαδικασία δε βλέπει ποτέ την πραγματική φυσική διεύθυνση. Η διαδικασία μπορεί να δημιουργήσει ένα δείκτη στη θέση 346, να τον αποθηκεύσει στη μνήμη, να τον χειριστεί, να τον συγκρίνει με άλλες διευθύνσεις, πάντα ως τον αριθμό 346. Μόνο όταν χρησιμοποιείται ως διεύθυνση μνήμης (ίσως σε μια έμμεση αποθήκευση ή φόρτωση) μετατοπίζεται σε σχέση με τον καταχωρητή βάσης. Οι διευθύνσεις με τις οποίες ασχολείται η διαδικασία ονομάζονται «λογικές διευθύνσεις». Το «υλικό αντιστοίχισης μνήμης» (memory mapping hardware) μετατρέπει τις λογικές σε φυσικές διευθύνσεις.

Γι' αυτό το υλικό, μια αλλαγή στο φράγμα απαιτεί μόνο την αλλαγή του καταχωρητή βάσης και τη «μεταφορά» (moving) της μνήμης της διαδικασίας στις σωστές θέσεις σε σχέση με τη νέα τιμή του φράγματος. Αυτός ο τρόπος μπορεί να απαιτεί την αντιγραφή ενός σημαντικού όγκου μνήμης, αλλά επιτρέπει την αλλαγή του φράγματος οποιαδήποτε στιγμή.

Παρατηρήστε, επίσης, ότι τώρα έχουμε δύο τύπους διευθύνσεων: λογικές διευθύνσεις (από 0 έως max) και φυσικές διευθύνσεις (από R+0 έως R+max για τιμή φράγματος R). Η διαδικασία παράγει μόνο λογικές διευθύνσεις και θεωρεί ότι τρέχει στις θέσεις 0 έως max. Το ΛΣ μπορεί να προσπελάσει όλη τη φυσική μνήμη κατευθείαν,

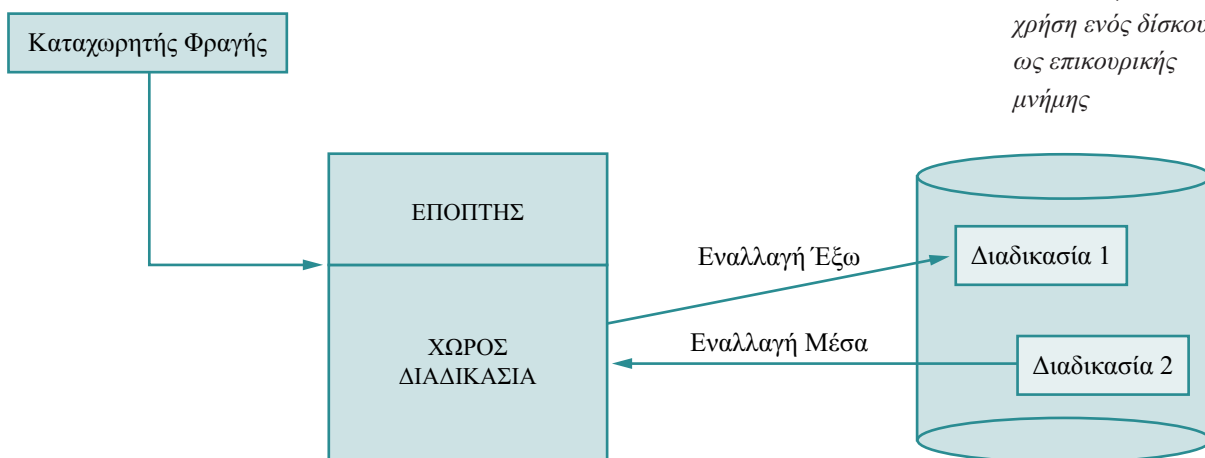
σε καθεστώς επόπτη. Όλη η πληροφορία που μεταβιβάζεται από τη διαδικασία στο ΛΣ (όπως οι διευθύνσεις των απομονωτών για τις κλήσεις συστήματος) πρέπει να μετατοπιστεί από λογισμικό ΛΣ πριν να χρησιμοποιηθεί. Αυτή η ανάγκη είναι ιδιαίτερα επιτακτική για διευθύνσεις που δίνονται σε μονάδες I/O. Η διαδικασία δίνει τις λογικές διευθύνσεις που πρέπει να αντιστοιχιστούν με φυσικές διευθύνσεις πριν χρησιμοποιηθούν. Η έννοια του «χώρου λογικών διευθύνσεων» (logical address space) είναι βασική στη διαχείριση μνήμης.

4.4 Εναλλαγή (Swapping)

Ο τρόπος διαχείρισης μνήμης με παραμένοντα επόπτη μπορεί να δείχνει ότι είναι μικρής χρησιμότητας, αφού φαίνεται να έχει σχεδιαστεί για μια διαδικασία. Ήταν όμως το βασικό σχήμα σε δύο αρχικά συστήματα με καταμερισμό χρόνου: το CTSS και το Q-32. Αυτά τα συστήματα χρησιμοποιούσαν έναν παραμένοντα επόπτη και η υπόλοιπη μνήμη ήταν διαθέσιμη στη διαδικασία που βρισκόταν στη μνήμη εκείνη τη στιγμή. Όταν άλλαζαν στην επόμενη διαδικασία, τα τρέχοντα περιεχόμενα της μνήμης της αρχικής διαδικασίας γράφονταν σε επικουρική μνήμη (δίσκος ή τύμπανο) και η μνήμη της επόμενης διαδικασίας φορτωνόταν. Αυτό το σχήμα ονομάζεται «εναλλαγή» (Σχήμα 4.8).

Σχήμα 4.8

Εναλλαγή δύο διαδικασιών με χρήση ενός δίσκου ως επικουρικής μνήμης



4.4.1 Επικουρική μνήμη (Backing Store)

Η εναλλαγή απαιτεί επικουρική μνήμη. Αυτή είναι συνήθως ένας γρήγορος δίσκος. Πρέπει να είναι αρκετά μεγάλη, ώστε να εξυπηρετεί αντίγραφα όλων των εικόνων της μνήμης όλων των διαδικασιών, και πρέπει να προσφέρει απευθείας προσπέλαση σε αυτά. Η «ουρά ετοιμότητας» (ready queue) αποτελείται από όλες τις διαδικα-

σίες που είναι έτοιμες να εκτελεστούν και των οποίων οι εικόνες μνήμης βρίσκονται στην επικουρική μνήμη. Μια ξεχωριστή μεταβλητή συστήματος δείχνει ποια διαδικασία είναι στη μνήμη αυτή τη στιγμή. Όταν έρθει η σειρά να εκτελέσει μια διαδικασία, καλείται ο «διεκπαιωτής» (dispatcher). Ο διεκπαιωτής ελέγχει αν αυτή η διαδικασία βρίσκεται στη μνήμη, αν όχι, «βγάζει έξω» (swaps out) την τρέχουσα στη μνήμη διαδικασία και «φέρει μέσα» (swaps in) την επιθυμητή διαδικασία. Η επιλεγμένη διαδικασία παίρνει τον έλεγχο και μπορεί να εκτελεστεί μόνο όταν δοθούν κατάλληλες τιμές σε κάποιους καταχωρητές, όπως στον καταχωρητή προγράμματος, στο μετρητή προγράμματος και λοιπά. Οι τιμές των καταχωρητών αυτών αποτελούν το περιβάλλον της διαδικασίας. Η μελέτη της χρησιμότητας και του τρόπου λειτουργίας των καταχωρητών αυτών είναι θέματα αρχιτεκτονικής ΗΥ.

4.4.2 Χρόνος Εναλλαγής (Swap Time)

Στα προηγούμενα κεφάλαια είδαμε τους λόγους για τους οποίους το ΛΣ εναλλάσσει τις διαδικασίες όσον αφορά την εκτέλεσή τους.

Άσκηση Αυτοαξιολόγησης 4.1

Γράψτε σε 3–4 γραμμές δύο τέτοιους λόγους. Συγκρίνετε αυτό που γράψατε με το Κεφάλαιο 2.

Από την προηγούμενη ενότητα όμως ίσως γίνεται φανερό ότι αυτή η εναλλαγή κοστίζει σε χρόνο: όσο παίρνει για να γραφούν τα περιεχόμενα της μνήμης στο δίσκο και να διαβαστούν τα αντίστοιχα της νέας διαδικασίας από το δίσκο στη μνήμη. Αν, μάλιστα, δεν μπορούμε να εξασφαλίσουμε κάποιον παραλληλισμό με άλλες διαδικασίες, τότε ο χρόνος αυτός πάει τελείως χαμένος: ούτε η CPU ούτε οι μονάδες I/O θα κάνουν «χρήσιμη» δουλειά. Για να πάρουμε μια ιδέα αυτού του χρόνου, ας υποθέσουμε ότι η διαδικασία είναι 20K bytes και η επικουρική μνήμη είναι ένα «τύμπανο σταθερής κεφαλής» (fixed-head drum) με «μέση καθυστέρηση» (average latency) 8ms και ρυθμό μεταφοράς δεδομένων (transfer rate) 250.000 bytes το δευτερόλεπτο. Τότε, η μεταφορά της διαδικασίας των 20K bytes προς ή από τη μνήμη διαρκεί:

$$\begin{aligned} 8\text{ms} + (20\text{K bytes} / 250.000 \text{ bytes/sec}) &= 8\text{ms} + (2/25) \text{ sec} \\ &= 8 \text{ ms} + (2000/25)\text{ms} \\ &= 88\text{ms} \end{aligned}$$

Επειδή πρέπει να «εναλλάξουμε έξω» και να «εναλλάξουμε μέσα», ο συνολικός χρόνος εναλλαγής είναι 176ms.

Στο σύστημα Q-32 χρησιμοποιούσαν ένα τύμπανο με μέση καθυστέρηση 10ms και ρυθμό μεταφοράς 363.000 bytes/sec, με αποτέλεσμα έναν αναμενόμενο χρόνο εναλλαγής 130ms για 20K bytes.

Για αποδοτική χρήση της CPU, θέλουμε ο χρόνος εκτέλεσης για κάθε διαδικασία να είναι μεγάλος σε σχέση με το χρόνο εναλλαγής. Έτσι, π.χ. στο round – robin αλγόριθμο χρονοδρομολόγησης (δες Ενότητα 4.5) της CPU το «χρονικό κβάντο» (time quantum) θα έπρεπε να είναι αρκετά μεγαλύτερο από 0,176 δευτερόλεπτα. Παρατηρήστε ότι το μεγαλύτερο μέρος του χρόνου εναλλαγής είναι ο «χρόνος μεταφοράς» (transfer time). Ο συνολικός χρόνος μεταφοράς είναι ευθέως ανάλογος της μνήμης που εναλλάσσεται. Αν έχουμε ένα υπολογιστικό σύστημα των 32K, με παραμένοντα επόπτη των 12K, η μέγιστη διαδικασία είναι 20K. Πάντως, πολλές διαδικασίες μπορεί να είναι έστω 4K. Διαδικασία των 4K θα μπορούσε να «εναλλαχθεί έξω» μόνο σε 24ms σε σύγκριση με τα 88ms που χρειάζονται τα 20K. Συνεπώς, είναι ωφέλιμο να ξέρουμε ακριβώς πόση μνήμη χρησιμοποιεί μια διαδικασία και όχι απλώς πόση θα μπορούσε να χρησιμοποιήσει. Για να είναι αποδοτικό αυτό το σύστημα, η διαδικασία πρέπει να ενημερώνει τον επόπτη για όποιες αλλαγές στις απαιτήσεις μνήμης. Έτσι, μια διαδικασία με «δυναμικές απαιτήσεις μνήμης» (dynamic memory requirements) πρέπει να εκδίδει κλήσεις συστήματος (π.χ. request memory / release memory), για να πληροφορεί τον επόπτη για τις μεταβαλλόμενες ανάγκες του σε μνήμη.

Η αποδοτικότητα της εναλλαγής μπορεί επίσης να βελτιωθεί με την αύξηση της απόδοσης της επικουρικής μνήμης. Για παράδειγμα, αναλογιστείτε το Large Storage (LCS) της IBM και το Extended Core Storage της CDC. Το LCS έχει χρόνο προσπέλασης 8ms και ρυθμό μεταφοράς 400.000 bytes/sec. Χρειάζεται δε 100ms για να εναλλάξει 20K. Το ECS έχει χρόνο προσπέλασης 3ms και ρυθμό μεταφοράς 10.000.000 bytes/sec. Έτσι, ο συνολικός χρόνος εναλλαγής 20K λέξεων είναι μόνο 4ms. Μνήμες νέου τύπου (π.χ. mass core, mass semiconductor ή bubble μνήμες) επιτρέπουν παρόμοιες ταχείες εναλλαγές από την κύρια προς την επικουρική μνήμη. Η Άσκηση αυτοαξιολόγησης 27 δίνει ένα παράδειγμα το οποίο αφορά σύγχρονα συστήματα.

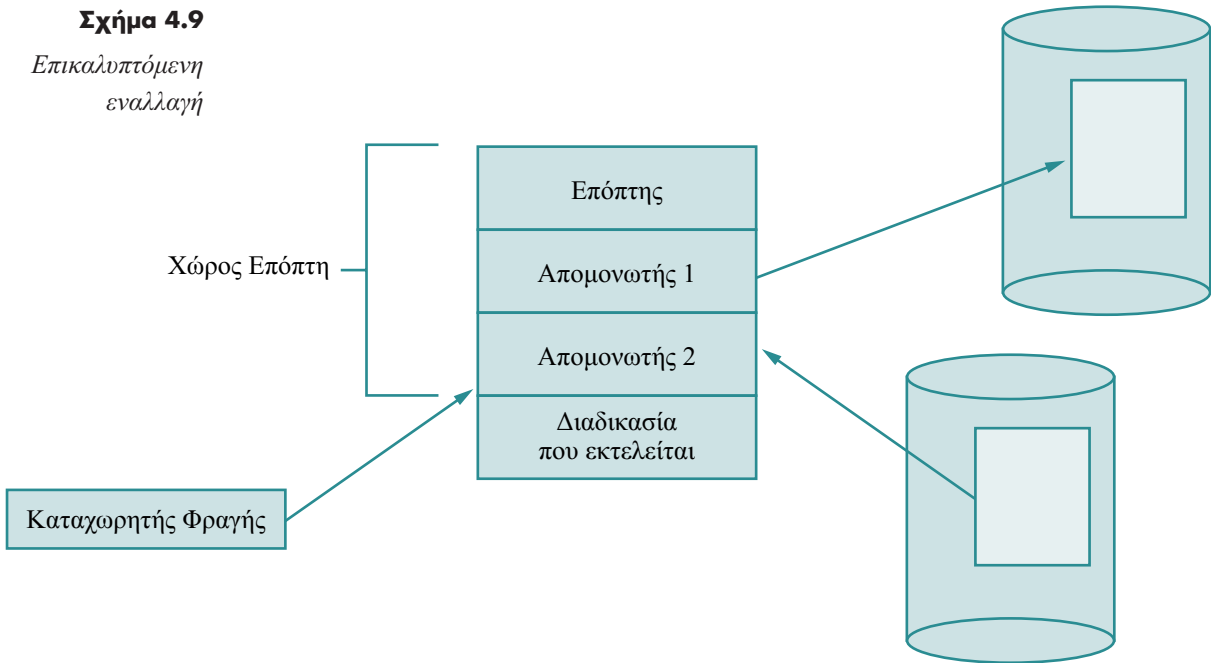
4.4.3 Επικαλυπτόμενη εναλλαγή (Overlapped Swapping)

Η επίδραση της εναλλαγής στο χρόνο αλλαγής περιεχομένου μπορεί να μειωθεί περαιτέρω με παραλληλισμό της εναλλαγής με την εκτέλεση της διαδικασίας. Κοιτάξτε το Σχήμα 4.9. Το ζητούμενο είναι η επικάλυψη της εναλλαγής μιας διαδικασίας με την εκτέλεση μιας άλλης. Έτσι, η CPU δε θα είναι «άπραγη» (idle) όσο διε-

νεργείται η εναλλαγή. Ενώ μια διαδικασία εκτελείται, η προηγούμενη διαδικασία «βγαίνει έξω» από τον απομονωτή 1 και η επόμενη προς εκτέλεση διαδικασία «μπαίνει μέσα» στον απομονωτή 2.

Σχήμα 4.9

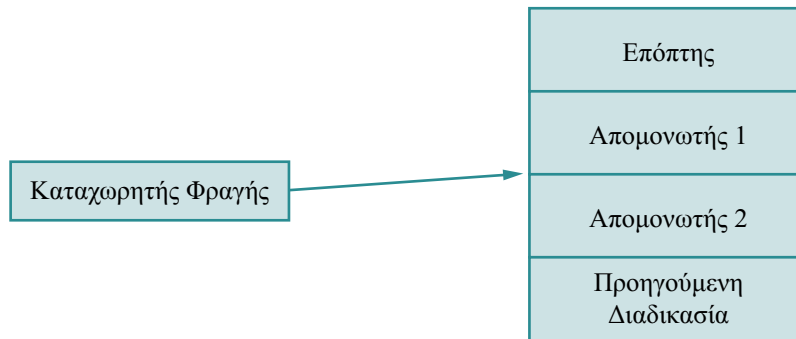
*Επικαλυπτόμενη
εναλλαγή*



Άσκηση Αυτοαξιολόγησης 4.2

Στο προηγούμενο κεφάλαιο χρησιμοποιήσαμε την έννοια του απομονωτή για το συντονισμό διαδικασιών. Εδώ για ποιο σκοπό χρησιμοποιούνται; Με ποια έννοια οι δύο αυτές διαφορετικές χρήσεις είναι όμοιες, ώστε να δίνουμε την ίδια ονομασία: «απομονωτής» (buffer).

Προσέξτε πάντως ότι, αφού η τρέχουσα διαδικασία ελευθερώσει την CPU, πρέπει να μεταφέρουμε την επόμενη διαδικασία από τον απομονωτή 2 στην περιοχή των διαδικασιών για να γίνει δυνατή η εκτέλεσή της. Επίσης, η διαδικασία που βρίσκεται εκείνη τη στιγμή στην περιοχή των διαδικασιών πρέπει να μεταφερθεί σε έναν από τους απομονωτές για να εναλλαχθεί. Αν δε γινόταν αυτό, τότε θα μπορούσαμε να εκτελέσουμε τη διαδικασία στον απομονωτή 2 μόνο μεταφέροντας το φράγμα, όπως στο Σχήμα 4.10. Η προηγούμενη διαδικασία (στην περιοχή των διαδικασιών) θα ήταν τότε ευάλωτη σε ακατάλληλη μετατροπή από τη διαδικασία στον απομονωτή 2. Συνεπώς, είναι απαραίτητη η «εναλλαγή από μνήμη σε μνήμη» (memory-to-memory swap).

**Σχήμα 4.10**

Προσπάθεια εκτέλεσης διαδικασίας με μετακίνηση φράγματος

Άσκηση Αυτοαξιολόγησης 4.3

Ποια υπόθεση δυνατότητας πραγματικού παραλληλισμού υποβόσκει στα προηγούμενα;

Απάντηση:

Υποθέτουμε ότι την ώρα που τρέχει η διαδικασία Δ_1 (τρέχουσα) φορτώνεται η επόμενη Δ_2 . Για να έχει νόημα η μετακίνηση της Δ_2 από το δίσκο στον απομονωτή (στη μνήμη) ενόσω εκτελεί η τρέχουσα Δ_1 , πρέπει να υπάρχει δυνατότητα πραγματικού παραλληλισμού μεταξύ χρήσης CPU και μεταφοράς δεδομένων από/προς δίσκο σε/από μνήμη.

Αυτή η προσέγγιση έχει περιορισμούς αν χρησιμοποιούνται μονάδες εναλλαγής με ρυθμό μεταφοράς περίπου ίσο με την ταχύτητα της κύριας μνήμης. Όταν το ECS μεταφέρει, δεν απομένουν «κύκλοι μνήμης» (memory cycles) για να τους χρησιμοποιήσει η CPU. Συνεπώς, η λειτουργία της CPU σταματάει εντελώς όταν χρησιμοποιείται το ECS, και η επικάλυψη της CPU με την εναλλαγή δεν είναι δυνατή. Το τελευταίο αυτό σημείο δεν είναι απόλυτο, γιατί υπάρχει δυνατότητα η CPU να χρησιμοποιεί άλλες θέσεις της μνήμης, αν η αρχιτεκτονική του υλικού το επέτρεπε. Είναι όμως προφανές ότι ελαττώνεται πολύ η χρήση της CPU.

Υπάρχουν επίσης και άλλοι περιορισμοί στην εναλλαγή. Για να εναλλάξει το ΛΣ μια διαδικασία, πρέπει να εξασφαλίσει ότι η διαδικασία είναι εντελώς αδρανής. Ιδιαίτερου ενδιαφέροντος είναι οι όποιες εκκρεμείς λειτουργίες I/O υπάρχουν. Αν μια διαδικασία περιμένει κάποια λειτουργία I/O, μπορεί το ΛΣ να θέλει να την εναλλάξει για να ελευθερώσει τη μνήμη που καταλαμβάνει. Εάν, όμως, αυτή η λειτουργία I/O, ασύγχρονα, προσπελάζει τη μνήμη των διαδικασιών για απομονωτές I/O, τότε η διαδικασία δεν πρέπει να εναλλαχθεί. Υποθέστε ότι η λειτουργία I/O έχει μπει σε ουρά επειδή η μονάδα I/O είναι απασχολημένη. Εάν «βγάλει έξω» τη διαδικασία 1 και

«εναλλάξει μέσα» τη διαδικασία 2, η λειτουργία I/O μπορεί να προσπαθήσει να χρησιμοποιήσει μνήμη που τώρα ανήκει στη διαδικασία. Οι δύο κύριες λύσεις αυτού του προβλήματος είναι: (α) ποτέ να μην εναλλάσσεται διαδικασία με «εκκρεμή I/O» (pending I/O) ή (β) η εκτέλεση των λειτουργιών I/O να γίνεται μόνο μέσα σε απομονωτές του ΛΣ. Μεταφορές μεταξύ του ΛΣ και της μνήμης των διαδικασιών γίνονται μόνο όταν η διαδικασία «εναλλαχθεί μέσα».

4.5 Πολλαπλές υποδιαιρέσεις μνήμης (Multiple Partitions)

Η επικαλυπτόμενη εναλλαγή είναι η κεντρική ιδέα για την «οργάνωση» (configuration) της μνήμης στον πολυπρογραμματισμό: το ζητούμενο είναι η γρήγορη εναλλαγή της CPU μεταξύ πολλών διαδικασιών που βρίσκονται στη μνήμη ταυτόχρονα. Πώς όμως κατανέμεται η μνήμη μεταξύ των διαφόρων εκτελέσιμων διαδικασιών; Η μνήμη χωρίζεται σε «περιοχές» (regions) ή «υποδιαιρέσεις» (partitions), που περιέχουν μια διαδικασία προς εκτέλεση η καθεμία. Κάθε τόσο το ΛΣ μετακινεί διαδικασίες από την κεντρική προς την περιφερειακή μνήμη (π.χ. δίσκο). Μερικοί από τους λόγους για τους οποίους γίνεται αυτή η μετακίνηση είναι όταν οι διαδικασίες ολοκληρώσουν την εκτέλεσή τους, όταν πρόκειται να περιμένουν άεργες για πολύ, όταν έχουν καταναλώσει πολλούς πόρους και πρέπει να δώσουν τη σειρά τους σε άλλες κτλ. Αντίστροφα, από το δίσκο στη μνήμη η μετακίνηση γίνεται όταν έρθει η σειρά, ή πρόκειται σύντομα να έρθει η σειρά, μιας διαδικασίας που είναι στην ουρά να εκτελέσει. Οι περιοχές (υποδιαιρέσεις) της μνήμης μπορεί να είναι στατικές (ορισμένες εκ των προτέρων) ή δυναμικές (με μέγεθος που αλλάζει κατά την ώρα της εκτέλεσης, βάσει των πραγματικών απαιτήσεων σε μνήμη των διαδικασιών). Αυτό, αντίστοιχα, προκαλεί σταθερό ή μεταβλητό (μέγιστο) αριθμό διαδικασιών πολυπρογραμματισμού στο σύστημα.

Στην ενότητα αυτή θεωρούμε ότι ένα μέρος του ΛΣ, το οποίο ονομάζεται «χρονοδρομολογητής» (scheduler) είναι υπεύθυνος για τη μεταφορά των διαδικασιών από το δίσκο στη μνήμη, και αντίστροφα. Ο χρονοδρομολογητής είναι επίσης υπεύθυνος να αποφασίσει ποια από τις διαδικασίες που είναι έτοιμη για εκτέλεση θα είναι η επόμενη που θα εκτελεστεί. Υπάρχουν διάφοροι αλγόριθμοι χρονοδρομολόγησης, τους οποίους θα μελετήσουμε στο μάθημα Λειτουργικά Συστήματα II. Στο σημείο αυτό θα αναφέρουμε μερικούς αλγόριθμους χρονοδρομολόγησης, οι οποίοι αναφέρονται και στη συνέχεια του κεφαλαίου αυτού.

ΧΡΟΝΟΔΡΟΜΟΛΟΓΗΣΗ ΕΞΥΓΗΡΕΤΗΣΗΣ ΕΚ ΠΕΡΙΤΡΟΠΗΣ (ROUND ROBIN)

Ο «αλγόριθμος χρονοδρομολόγησης εκ περιτροπής» είναι ένας από τους παλαιότερους, δικαιότερους και πλέον χρησιμοποιούμενους αλγόριθμους χρονοδρομολόγησης. Σε μια διαδικασία εκχωρείται ένα χρονικό διάστημα, το οποίο καλείται «κβάντο

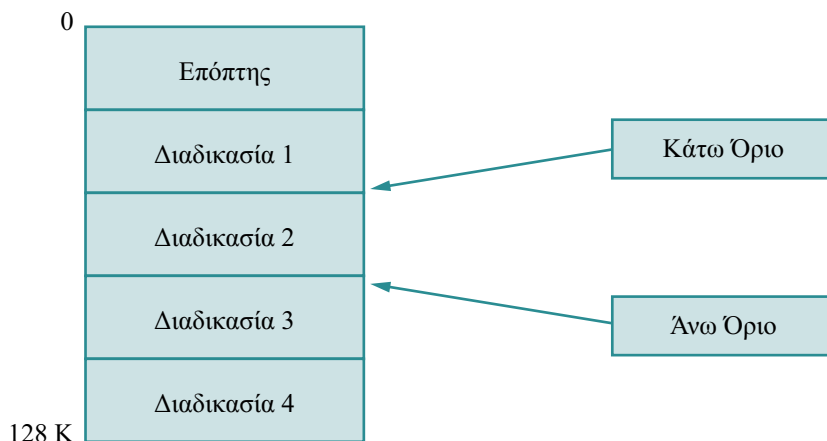
χρόνου» (quantum), μέσα στα όρια του οποίου επιτρέπεται η εκτέλεσή της. Αν η διαδικασία εκτελείται και μετά το τέλος του κβάντο χρόνου της, τότε η CPU προεκχωρείται σε άλλη διαδικασία. Αν η διαδικασία ανεστάλη ή τελείωσε πριν από το τέλος του κβάντο χρόνου της, τότε η CPU παρεμβαίνει στην αντίστοιχη χρονική στιγμή.

ΧΡΟΝΟΔΡΟΜΟΛΟΓΗΣΗ ΠΡΩΤΗ ΕΙΣΕΡΧΟΜΕΝΗ – ΠΡΩΤΗ ΕΞΕΡΧΟΜΕΝΗ (FIRST COME FIRST SERVE – FCFS)

Ο αλγόριθμος αυτός χρονοδρομολογεί / δρομολογεί τις διαδικασίες με τη σειρά που αυτές φτάνουν στο σύστημα. Δηλαδή, η πρώτη διαδικασία που φτάνει στο σύστημα θα εκτελεστεί πρώτη, η δεύτερη διαδικασία θα εκτελεστεί δεύτερη και ούτω καθεξής. Η βασική διαφορά του αλγόριθμου αυτού από τον αλγόριθμο χρονοδρομολόγησης, που παρουσιάσαμε προηγουμένως, είναι ότι σε αυτό τον αλγόριθμο μια διαδικασία παραμένει στην CPU έως ότου τελειώσει την εκτέλεσή της ή μέχρι να διακοπεί λόγω λειτουργιών E/E.

4.5.1 Υλικό προστασίας

Αυτή η «συγκατοίκηση» διαδικασιών στη μνήμη δημιουργεί κινδύνους: κατά λάθος ή επίτηδες, μια διαδικασία μπορεί να επιχειρήσει πρόσβαση στο χώρο μνήμης κάποιας άλλης. Χρειάζεται λοιπόν προστασία ή περιορισμός (δες Υποενότητα 4.3.2). Δύο καταχωρητές (Σχήματα 4.10 και 4.11) ορίζουν το άνω και κάτω όριο των διευθύνσεων που επιτρέπεται να παραχθούν από μια διαδικασία. Μπορούν να οριστούν απόλυτα ή σχετικά.



Σχήμα 4.11

Δύο καταχωρητές ορίων ορίζουν ένα χώρο φυσικών διευθύνσεων

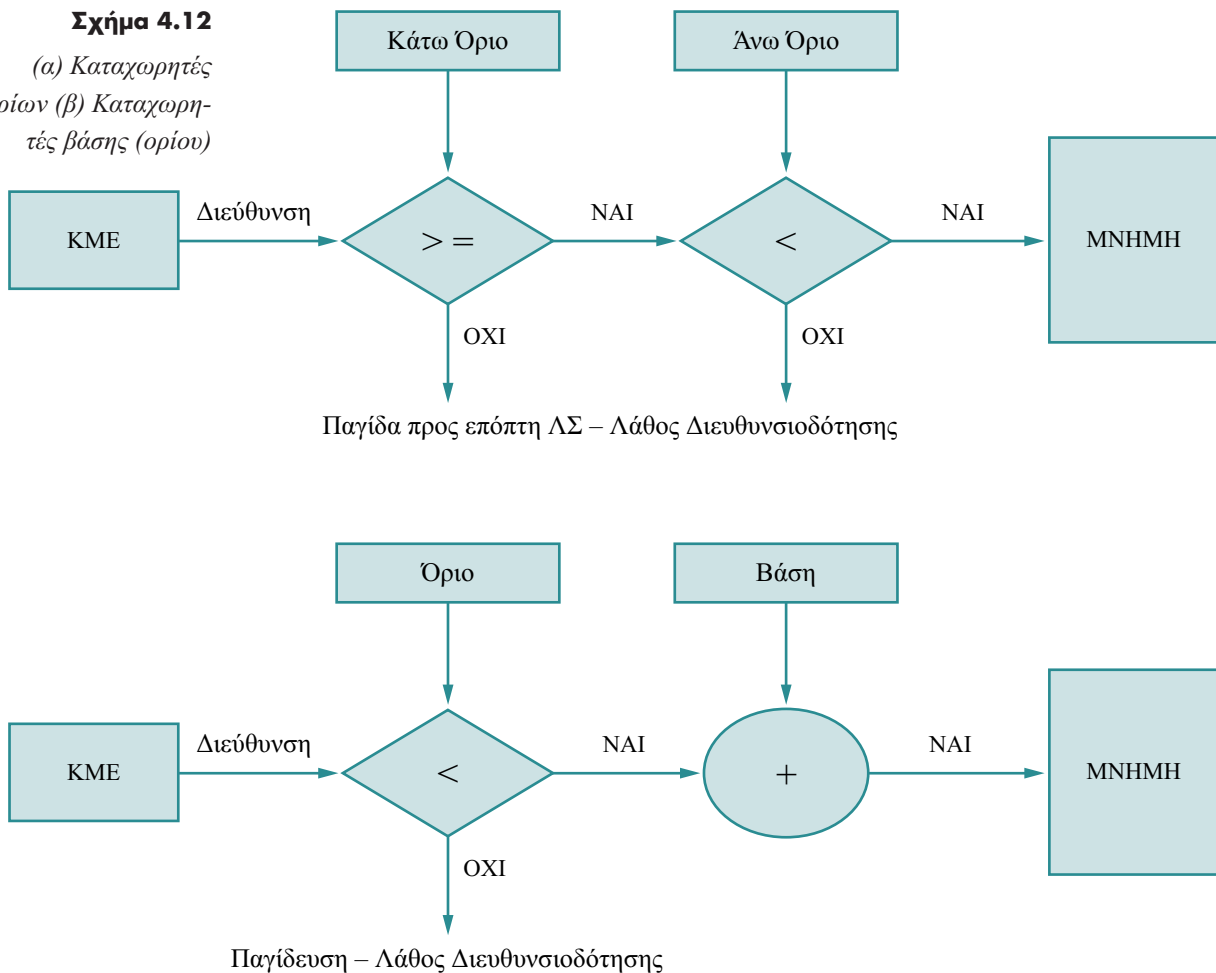
Καταχωρητές ορίων (bound registers). Οι τιμές της μικρότερης και μεγαλύτερης φυσικής διεύθυνσης (π.χ κάτω όριο = 174640). Οι επιτρεπτές διευθύνσεις της διαδικασίας είναι απόλυτες και εκτείνονται από το κάτω έως το άνω όριο.

Καταχωρητές βάσης και ορίου (base and limit registers). Οι τιμές της μικρότερης φυσικής διεύθυνσης και του «διαστήματος» (range) των λογικών διευθύνσεων (π.χ. βάση = 100040 και όριο = 74600). Οι επιτρεπτές διευθύνσεις της διαδικασίας είναι σχετικές, εκτείνονται μεταξύ 0 και ορίου και μετατοπίζονται δυναμικά σε απόλυτες φυσικές διευθύνσεις που εκτείνονται από τη βάση έως τη (βάση + όριο).

Όπως φαίνεται και στο Σχήμα 4.12, το υλικό για τη χρήση αυτών των δύο καταχωρητών διαφέρει ελαφρά. Οι καταχωρητές ορίου απαιτούν στατική μετατόπιση κατά το χρόνο φορτώματος ή συμβολομετάφρασης (assembly time). Κάθε φυσική διεύθυνση που αντιστοιχίζεται σε κάποια λογική διεύθυνση πρέπει να είναι μεταξύ των ορίων. Με τους καταχωρητές βάσης και ορίου κάθε φυσική διεύθυνση που αντιστοιχίζεται σε κάποια λογική διεύθυνση πρέπει να είναι μικρότερη του ορίου, και μετά μετατοπίζεται δυναμικά, κατά την τιμή του καταχωρητή βάσης. Αυτή η μετατοπισμένη διεύθυνση στέλνεται στη μνήμη. Ο υπολογιστής CDC 6600 και οι απόγονοί του χρησιμοποιούν αυτό τον τρόπο.

Σχήμα 4.12

(α) Καταχωρητές ορίων (β) Καταχωρητές βάσης (ορίου)



4.5.2 Σταθερές περιοχές (Fixed Regions)

Στο σύστημα με σταθερές περιοχές, οι περιοχές είναι σταθερές και δεν αλλάζουν καθώς το σύστημα τρέχει. Για παράδειγμα, μια μνήμη των 32K μπορεί να διαιρεθεί σε περιοχές με τα ακόλουθα μεγέθη:

Παραμένων επόπτης 10K

Μικρού μεγέθους διαδικασίες 4K

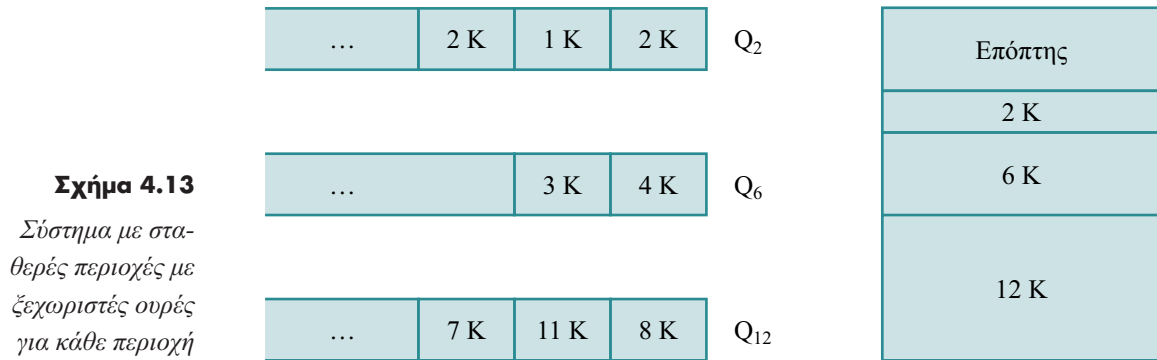
Μεσαίου μεγέθους διαδικασίες 6K

Μεγάλου μεγέθους διαδικασίες 12K

Πολιτικές και αλγόριθμοι διαχείρισης μνήμης σε σύστημα με σταθερές περιοχές

Καθώς οι διαδικασίες εισέρχονται στο σύστημα, εισάγονται στην ουρά διαδικασιών. Ο χρονοδρομολογητής διαδικασιών συνυπολογίζει τις απαιτήσεις μνήμης κάθε διαδικασίας και τις διαθέσιμες περιοχές, για να αποφασίσει, με βάση μια πολιτική διαχείρισης μνήμης, σε ποιες διαδικασίες θα κατανεμηθεί μνήμη. Όταν δοθεί χώρος σε μια διαδικασία, αυτή φορτώνεται σε μια περιοχή. Μετά μπορεί να ανταγωνιστεί τις άλλες διαδικασίες για την απόκτηση της CPU. Όταν μια διαδικασία τερματίζει, απελευθερώνει την περιοχή μνήμης της, την οποία ο χρονοδρομολογητής μπορεί να γεμίσει με μια άλλη διαδικασία από την ουρά διαδικασιών.

Υπάρχουν διάφορες πολιτικές διαχείρισης μνήμης στην κατανομή μνήμης στις διαδικασίες. Μια στρατηγική είναι να χωρίζονται σε κλάσεις οι διαδικασίες με την είσοδό τους στο σύστημα, ανάλογα με τις απαιτήσεις τους σε μνήμη. Αυτός ο διαχωρισμός μπορεί να γίνει ζητώντας σχετική δήλωση από τη διαδικασία (μέγιστη ποσότητα μνήμης που μπορεί να απαιτηθεί). Εναλλακτικά, το σύστημα μπορούσε να προσπαθήσει να προσδιορίσει τις απαιτήσεις μνήμης αυτόματα, π.χ. ένα προκαταρκτικό πέρασμα (prepass) των καρτών ελέγχου μπορούσε να βοηθήσει στον καθορισμό της μέγιστης απαίτησης μνήμης (στο σημείο αυτό χρησιμοποιήσαμε παρατατικό χρόνο, γιατί σήμερα δε χρησιμοποιούνται κάρτες ελέγχου. Όπως αναφέραμε και προηγουμένως, τα συστήματα διαχείρισης μνήμης αυτής της ενότητας δε χρησιμοποιούνται πλέον, τα παρουσιάζουμε για διδακτικό σκοπό). Κάθε περιοχή μνήμης έχει τη δικιά της ουρά (Σχήμα 4.13). Ο «διαχωρισμός των διαδικασιών» (job classification) χρησιμοποιείται για την επιλογή της κατάλληλης ουράς για τη διαδικασία, η οποία είναι αυτή η οποία έχει το ίδιο ή το αμέσως μεγαλύτερο μέγεθος από το μέγεθος της διαδικασίας.



Άσκηση Αυτοαξιολόγησης 4.4

Έστω μια μνήμη με τρεις περιοχές για διαδικασίες. Αν οι περιοχές αυτές έχουν μεγέθη 2K, 6K και 12K, τότε χρειάζονται τρεις ουρές, έστω Q₂, Q₆, Q₁₂, αντίστοιχα, με τα πιο πάνω μεγέθη μνήμης. Έστω τρεις εισερχόμενες διαδικασίες που απαιτούν μνήμη η πρώτη 5K, η δεύτερη 10K και η τρίτη 2K. Σε ποια ουρά θα πήγαινε η κάθε διαδικασία;

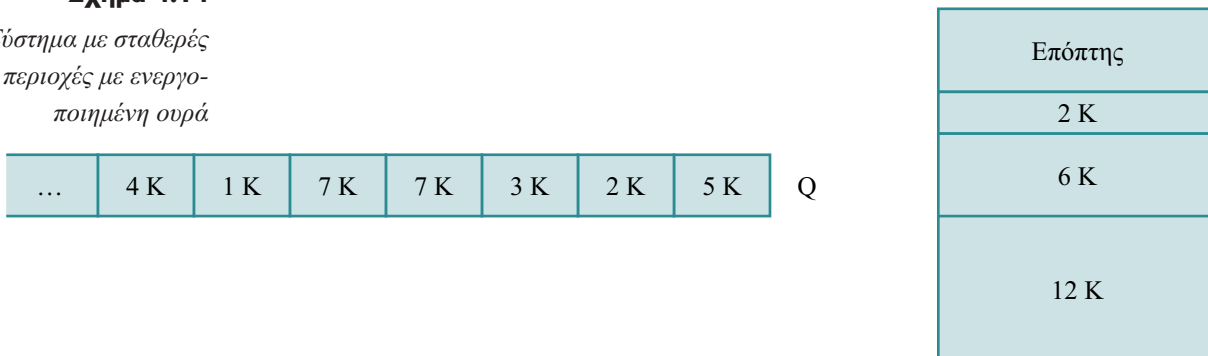
Απάντηση:

Η πρώτη διαδικασία θα πήγαινε στη ουρά Q₆, η δεύτερη θα πήγαινε στην ουρά Q₁₂ και, τέλος, η τελευταία διαδικασία στην ουρά Q₂.

Παρατηρούμε εδώ ότι είναι δυνατόν κάθε ουρά να χρονοδρομολογείται ξεχωριστά. Αυτό είναι δυνατόν, εφόσον κάθε ουρά έχει τη δική της περιοχή μνήμης, δεν υπάρχει ανταγωνισμός μεταξύ των ουρών.

Μια άλλη προσέγγιση είναι να μουν όλες οι διαδικασίες στην ίδια ουρά (Σχήμα 4.14). Ο χρονοδρομολογητής επιλέγει την επόμενη προς εκτέλεση διαδικασία και περιμένει μέχρι μια περιοχή μνήμης αυτού του μεγέθους να γίνει διαθέσιμη. Υποθέστε ότι είχαμε έναν FCFS χρονοδρομολογητή, την ουρά του Σχήματος 4.14 και περιοχές των 2K, 6K, 12K.

Σχήμα 4.14
Σύστημα με σταθερές περιοχές με ενεργοποιημένη ουρά



Άσκηση Αυτοαξιολόγησης 4.5

Σε ποια περιοχή μνήμης θα ανατεθεί η κάθε διαδικασία;

Απάντηση:

Αρχικά η διαδικασία 1 θα ανατίθεται στην περιοχή των 6K και η διαδικασία 2 στην περιοχή των 2K. Η επόμενη διαδικασία απαιτεί 3K, και γι' αυτό θα ανατεθεί στην περιοχή των 6K, αφού αυτή η περιοχή χρησιμοποιείται από τη διαδικασία 1 (η διαδικασία 3 πρέπει να περιμένει τη διαδικασία 1). Η διαδικασία 4 πρέπει να περιμένει τη δρομολόγηση της διαδικασίας 3 (αφού ο χρονοδρομολογητής είναι FCFS), αν και η περιοχή της διεργασίας 4 είναι ελεύθερη (είναι η περιοχή των 12K). Οι υπόλοιπες διαδικασίες θα ανατεθούν με ανάλογο τρόπο.

Μια προφανής παραλλαγή αυτής της πολιτικής προσπαθεί να μην αφήνει τις περιοχές μνήμης άδειες. Έτσι, όταν μια περιοχή είναι διαθέσιμη, ο χρονοδρομολογητής διατρέχει την ουρά ψάχνοντας για την πρώτη διαδικασία που μπορεί να χωρέσει σε αυτή την περιοχή και δρομολογεί αυτή τη διαδικασία (έστω Δ), έστω κι αν υπάρχουν άλλες διαδικασίες που περιμένουν πριν από αυτή στην ουρά.

Άσκηση Αυτοαξιολόγησης 4.6

Ποιο είναι το χαρακτηριστικό των διαδικασιών αυτών; Εμποδίζεται η εκτέλεση των διαδικασιών αυτών;

Απάντηση:

Οι διαδικασίες που περιμένουν πριν από τη διαδικασία Δ στην ουρά δεν μπορούν να κάνουν χρήση της διαθέσιμης περιοχής, αφού είναι πολύ μεγάλες. Άρα δεν τις εμποδίζει από το να προχωρήσουν και να εκτελεστεί μια μικρότερη διαδικασία (Δ) η οποία βρίσκεται μετά από αυτές στην ουρά και καταλαμβάνει την άδεια περιοχή μνήμης.

Μια άλλη παραλλαγή απαντάει την εξής ερώτηση: Γιατί κρατιέται σε αναμονή η διαδικασία 3 όταν υπάρχει μια άδεια περιοχή που είναι αρκετά μεγάλη για να εκτελεστεί η διαδικασία 3; Προφανώς μπορεί να δρομολογηθεί η διαδικασία 3 στην περιοχή των 12K. Είναι γεγονός ότι η διαδικασία 3 θα μπορούσε να τρέξει στην περιοχή των 6K, αλλά αυτή είναι κατειλημμένη. Θα έπρεπε να δρομολογηθεί η διαδικασία 3

στην περιοχή των 12 K (σπαταλώντας χώρο) ή θα έπρεπε να περιμένει, όσο η διαδικασία 4, η οποία βρίσκεται αργότερα στην ουρά (και μπορεί να εκτελεστεί μόνο στην περιοχή των 12K) για να δρομολογηθεί; Αν αποφασιζόταν να εκτελεστεί η διαδικασία 4, αναγκάζοντας την 3 να περιμένει, τι θα έκανε ο αλγόριθμος χρονοδρομολόγησης όταν η διαδικασία 4 τερμάτιζε; Θα άφηνε άδεια την περιοχή των 12K (δεσμευμένη για όποια μεγάλη διαδικασία που ίσως έρθει) ή θα τη χρησιμοποιούσε για κάποια διαδικασία που θα μπορούσε να εκτελεστεί σε μικρότερη περιοχή;

Αυτές οι αποφάσεις αντανακλούν την επιλογή μεταξύ της πολιτικής διαχείρισης μνήμης του «καλύτερου ταιριάσματος μόνο» (best-fit-only) και της πολιτικής του «καλύτερου διαθέσιμου ταιριάσματος» (best-available-fit). Για τον προσδιορισμό (εύρεση των περιοχών αυτών) εκτελούνται από το χρονοδρομολογητή αντίστοιχοι αλγόριθμοι, οι οποίοι έχουν και τα ίδια ονόματα με αυτές τις πολιτικές.

Άσκηση Αυτοαξιολόγησης 4.7

Διαλέξτε δύο οποιεσδήποτε πολιτικές διαχείρισης μνήμης από τις πιο πάνω. Ποια νομίζεται ότι είναι καλύτερη και γιατί; Τώρα φτιάξτε μια ουρά αναμονής διαδικασιών (με μέγεθος και διάρκεια εκτέλεσης των διαδικασιών) που να δείχνει ότι η πολιτική που θεωρήσατε χειρότερη είναι καλύτερη στη συγκεκριμένη περίπτωση.

Μια ακόμα παραλλαγή του συστήματος με σταθερές περιοχές μπορεί να δημιουργηθεί με την προσθήκη της εναλλαγής. Αν υπάρχουν αρκετές διαδικασίες που καθεμία χωράει σε μια συγκεκριμένη περιοχή, μπορούν να εναλλάσσονται μέσα και έξω από αυτή την περιοχή. Για παράδειγμα, υποθέστε ότι έχουμε 4 περιοχές, καθεμία με round-robin αλγόριθμο χρονοδρομολόγησης της CPU. Όταν το κβάντο τελειώσει, ο διαχειριστής μνήμης θα «έβγαζε έξω» τη διαδικασία που μόλις τελείωσε και θα «έβαζε μέσα» μια άλλη διαδικασία γι' αυτή την περιοχή. Στον ενδιάμεσο χρόνο, ο χρονοδρομολογητής θα έδινε τον έλεγχο της CPU σε κάποια διαδικασία σε άλλη περιοχή. Κάθε διαδικασία, όταν τελειώνει το κβάντο της, εναλλάσσεται με μια άλλη διαδικασία γι' αυτή την περιοχή. Ο διαχειριστής μνήμης θα εναλλάσσει διαδικασίες στη μνήμη έτοιμες προς εκτέλεση όταν ο χρονοδρομολογητής της CPU θέλει να εναδρομολογήσει την CPU.

Μια παραλλαγή αυτής της πολιτικής εναλλαγής χρησιμοποιείται σε «αλγόριθμους χρονοδρομολόγησης που βασίζονται σε προτεραιότητες» (priority base scheduling algorithms). Εάν μια υψηλότερης προτεραιότητας διαδικασία αφιχθεί και ζητάει εξυ-

πηρέτηση, ο διαχειριστής μνήμης μπορεί να «βγάλει έξω» τη χαμηλότερης προτεραιότητας διαδικασία, ώστε να φορτώσει και να εκτελέσει την υψηλότερης προτεραιότητας. Όταν η υψηλότερης προτεραιότητας διαδικασία τερματίσει, η χαμηλότερης προτεραιότητας μπορεί να «μπει μέσα» και να συνεχιστεί. Μια διαδικασία που «εναλλάσσεται έξω» σε σύστημα με σταθερές περιοχές θα «εναλλαχθεί μέσα» στην ίδια περιοχή. Αυτός ο περιορισμός υπαγορεύεται και από την πολιτική κατανομής των περιοχών και από τη μέθοδο μετατόπισης. Αν η μετατόπιση γίνεται κατά το χρόνο φορτώματος ή συμβολομετάφρασης (στατική μετατόπιση), τότε η διαδικασία δεν μπορεί να μεταφερθεί σε διαφορετική περιοχή. Με τη δυναμική μετατόπιση, όπως με καταχωρητές βάσης / ορίου, είναι δυνατόν να εναλλαχθεί μια διαδικασία σε διαφορετική περιοχή.

Είδαμε στο Κεφάλαιο 2 ότι μια διαδικασία που εκτελείται μπορεί να ζητήσει περισσότερη μνήμη. Αυτό επιτρέπει σε μια διαδικασία να καθορίσει δυναμικά τις απαιτήσεις της για μνήμη, βασιζόμενη στα δεδομένα εισόδου της. Αυτό μπορεί να δημιουργήσει προβλήματα σε μια στατική πολιτική διαχείρισης μνήμης.

Άσκηση Αυτοαξιολόγησης 4.8

Δώστε ένα παράδειγμα.

Απάντηση:

Υποθέστε ότι έχουμε μια διαδικασία των 4K τοποθετημένη σε περιοχή των 6K. Η διαδικασία μπορεί να απαιτήσει και να ελευθερώσει μνήμη χωρίς περιορισμούς, εφόσον δεν απαιτεί πιο πολύ από 6K (το μέγεθος της περιοχής) την κάθε φορά. Παρατηρήστε ότι όλες αυτές οι αιτήσεις δεν έχουν πραγματική επίδραση στο ποσό της μνήμης που έχει δοθεί στη διαδικασία. Η διαδικασία θα έχει πάντοτε 6K μνήμης, έστω κι αν χρησιμοποιεί μόνο 4K. Τι μπορεί να κάνει το σύστημα αν η διαδικασία απαιτήσει ακόμα περισσότερη μνήμη; Υπάρχουν τρεις κύριες εναλλακτικές:

- Το ΛΣ τερματίζει τη διαδικασία. Εάν απαιτούμε από τη διαδικασία να δηλώσει το μέγιστο ποσό μνήμης που θα χρειαστεί και χρησιμοποιείτε αυτή την τιμή για να επιλεγεί περιοχή, τότε μια αίτηση για χώρο μνήμης που υπερβαίνει αυτή την περιοχή είναι ένα λάθος εκτέλεσης (run-time error).
- Μπορεί απλώς να επιστραφεί ο έλεγχος (return control) στη διαδικασία με μια ένδειξη καταστάσεως (status indicator) ότι δεν υπάρχει άλλη μνήμη. Η διαδικασία μπορεί να εγκαταλείψει ή να τροποποιήσει τη λειτουργία του, ώστε να εκτελεστεί στο διαθέσιμο χώρο. Πολλοί αλγόριθμοι παρουσιάζουν ένα συμβι-

βασμό χώρου – χρόνου: μεγαλύτερος χώρος επιτρέπει στη διαδικασία να εκτελεστεί γρηγορότερα, αλλά μπορεί να εκτελεστεί (με επιβράδυνση) σε μικρότερο χώρο μνήμης.

- Μπορεί το ΛΣ: (1) να βγάλει έξω την διαδικασία, (2) να περιμένει να γίνει διαθέσιμη μια μεγαλύτερη περιοχή, (3) να τη βάλει μέσα σε μεγαλύτερη περιοχή και (4) να συνεχίσει την εκτέλεση. Αυτή η λύση είναι δυνατή μόνο όταν το υλικό υποστηρίζει δυναμική μετατόπιση, και είναι αρκετά ακριβή λύση (ερώτηση: Ως προς τι;), αλλά προσφέρει μέγιστη ευελιξία στις διαδικασίες για τροποποίηση των αναγκών τους σε μνήμη, όπως χρειάζεται.

Σημειώστε επίσης ότι, όταν το ΛΣ είναι ενημερωμένο από τη διαδικασία για το πραγματικό μέγεθος όγκου μνήμης που αυτή χρειάζεται, ο χρόνος εναλλαγής μπορεί να μειωθεί. Εάν σε μια διαδικασία δοθεί μια περιοχή των 12K αλλά χρησιμοποιεί μόνο 8K, τότε μόνο 8K χρειάζεται να εναλλαχθούν.

Επιλογή του Μεγέθους της Περιοχής (Regions Size Selection)

Άλλο ένα σχεδιαστικό πρόβλημα για το σύστημα με σταθερές περιοχές είναι ο καθορισμός των μεγεθών των περιοχών. Αν έχουμε 32K συνολική μνήμη και ο παραμένων επόπτης είναι 10K, μας μένουν 22K να διαμοιράσουμε τους χρήστες. Επαναλαμβάνουμε σε αυτό το σημείο ότι η ενότητα αυτή είναι για διδακτικό σκοπό. Οι σημερινές είναι μερικές χιλιάδες φορές μεγαλύτερες (256M) και, αντίστοιχα, οι διαδικασίες έχουν αυξήσει ανάλογα το μέγεθός τους. Αυτός είναι ουσιαστικά και ο λόγος για τον οποίο η αύξηση της μνήμης δεν είναι αρκετή για να μπορούν να εφαρμοστούν στις μέρες μας τέτοια συστήματα διαχείρισης μνήμης, τα οποία είναι πολύ πιο απλά από τα συστήματα τμηματοποίησης που χρησιμοποιούνται και θα τα μελετήσουμε στη συνέχεια του κεφαλαίου αυτού. Οι σχεδιαστές των συστημάτων αυτών έπρεπε να αποφασίσουν πόσες και τι μεγέθους περιοχές θα δημιουργούσαν. Η αρχική απόφαση βασιζόταν σε μια εκτίμηση των απαιτήσεων μνήμης των διαδικασιών, η οποία θα ήταν όσο πιο ακριβής γινόταν. Ιστορικά, αναφέρουμε ότι για ένα batch σύστημα που έτρεχε μικρές εφαρμογές I/O (από 1K έως 3K), έναν 8K μεταφραστή FORTRAN και ένα 4K συμβολομεταφραστή θα μπορούσαν να δημιουργηθούν: μια περιοχή των 10K για να επιτραπεί η μετάφραση και άλλων μεγάλων διαδικασιών και δύο περιοχές των 4K για διαδικασίες I/O και το συμβολομεταφραστή. Επειδή αυτή η διαίρεση αφήνει 4K ακατανέμητα, θα μπορούσε να αυξηθεί η περιοχή των 10K σε 14K ή μια περιοχή των 4K στα 8K ή και οι δύο περιοχές των 4K σε 6K.

Όταν το σύστημα άρχιζε να λειτουργεί, μπορούσαν να συγκεντρωθούν πληροφορίες γύρω από τους αριθμούς και τα μεγέθη των διαδικασιών που εκτελούνται στην πραγματικότητα στο σύστημα αυτό. Αυτά τα στοιχεία μπορεί να έδειχναν ότι ένα διαφορετικό σύστημα θα ήταν καλύτερο.

Το κύριο πρόβλημα με το σύστημα σταθερών περιοχών είναι το πώς θα ταίριαζαν σωστά τα μεγέθη των περιοχών με τις πραγματικές απαιτήσεις μνήμης των διαδικασιών. Η συνολική απόδοση του υπολογιστικού συστήματος είναι, γενικά, ανάλογη του «επίπεδου πολυπρογραμματισμού» (multiprogramming level). Το επίπεδο πολυπρογραμματισμού είναι ο αριθμός των διαδικασιών που βρίσκονται στη μνήμη. Οι διαδικασίες αυτές είναι όλες έτοιμες για εκτέλεση, και έτσι η χρήση της CPU να είναι όσο μεγαλύτερη γίνεται (να τείνει δηλαδή στο 100%). Είναι προφανές ότι το επίπεδο αυτό επηρεάζεται απευθείας από το πόσο καλά διαχειρίστηκε τη μνήμη. Αν η μισή μνήμη δε χρησιμοποιείται, τότε θα μπορούσε να εκτελεστεί διπλάσιος αριθμός διαδικασιών, αν υπήρχε τρόπος να γίνει εκμετάλλευση αυτού του χώρου.

Άσκηση Αυτοαξιολόγησης 4.9

Υποθέστε ότι ένας υπολογιστής έχει 2M Bytes μνήμη, από τα οποία το λειτουργικό σύστημα χρησιμοποιεί τα 512K. Αν όλα τα προγράμματα έχουν αναμονή λόγω εισόδου εξόδου το 60% του χρόνου τους, τότε σε τι ποσοστό θα ανέβει η απόδοση του συστήματος με την αύξηση της μνήμης κατά 1Mbyte;

Απάντηση:

Αρχικά η μνήμη αποτελούνταν από 2Mbyte. Αφού το λειτουργικό σύστημα χρησιμοποιεί τα 512K της, σημαίνει ότι στη μνήμη ταυτόχρονα μπορούν να βρίσκονται 3 προγράμματα (κάθε πρόγραμμα χρειάζεται 512K). Η αξιοποίηση της CPU κάτω από αυτές τις συνθήκες είναι: $1 - (0,6)^3 = 78,4\%$.

Όταν προστεθεί στη μνήμη ακόμα 1Mbyte, τότε ο αριθμός των προγραμμάτων σε αυτή είναι 5 και η αξιοποίηση της CPU είναι $1 - (0,6)^5 = 92,224\%$.

Το ποσοστό αύξησης της απόδοσης της CPU υπολογίζεται με απλή εφαρμογή της μεθόδου των τριών, ως εξής:

Στα 78,4 υπάρχει αύξηση 13,824

Στα 100 πόση αύξηση υπάρχει X

$$X = 13,824 * 100 / 78,4 = 17,632\%$$

4.5.3 Τεμαχισμός μνήμης (Memory Fragmentation)

Μια διαδικασία που χρειάζεται m θέσεις μνήμης μπορεί να τρέξει σε μια περιοχή n bytes, όπου $n \geq m$. Η διαφορά των δύο αυτών αριθμών ($n - m$) είναι η «εσωτερική φύρα κλασματοποίησης» (internal fragmentation), δηλαδή μνήμη που εμπεριέχεται σε μια περιοχή αλλά δε χρησιμοποιείται. «Εξωτερική κλασματοποίηση» (external fragmentation) έχουμε όταν μια περιοχή είναι αχρησιμοποίητη και διαθέσιμη, αλλά πολύ μικρή για οποιαδήποτε εν αναμονή διαδικασία. Και οι δύο τύποι τεμαχισμού σπαταλούν μνήμη στο σύστημα σταθερών τμημάτων.

Άσκηση Αυτοαξιολόγησης 4.10

Υποθέστε ότι διαιρούμε την περιοχή διαδικασιών, η οποία έχει μέγεθος 22K, σε μια περιοχή των 10K και σε τρεις περιοχές των 4K. Αν η ουρά περιέχει διαδικασίες που απαιτούν 7K, 3K, 6K και 6K, αναθέστε τις διαδικασίες σε περιοχές μνήμης με δύο διαφορετικούς τρόπους και υπολογίστε την εσωτερική και την εξωτερική κλασματοποίηση.

Απάντηση:

Μπορούμε να αναθέσουμε στην 7K διαδικασία τη 10K περιοχή (δημιουργώντας 3K εσωτερικής κλασματοποίησης) και την 3K διαδικασία σε μία από τις 4K περιοχές (δημιουργώντας 1K εσωτερικής κλασματοποίησης). Επειδή οι δύο εναπομείναντες διαδικασίες είναι πολύ μεγάλες για τις δύο διαθέσιμες περιοχές, μας μένουν δύο αχρησιμοποίητες περιοχές των 4K η καθεμία, σύνολο 8K εξωτερικής κλασματοποίησης. Η συνολική κλασματοποίηση, εσωτερική ή εξωτερική, είναι 12K, δηλαδή μεγαλύτερη από τη μισή μας μνήμη. Αναθέστε και εσείς τις διαδικασίες σε διαφορετικές περιοχές μνήμης και υπολογίστε, αντίστοιχα, την κλασματοποίηση.

Άσκηση Αυτοαξιολόγησης 4.11

Ας διαιρέσουμε τη μνήμη των διαδικασιών της πιο πάνω άσκησης σε περιοχές των 10K, 8K και 4K. Αν η ουρά περιέχει ξανά τις διαδικασίες που απαιτούν 7K, 3K, 6K και 6K μνήμη, αναθέστε τις διαδικασίες σε περιοχές μνήμης με διαφορετικούς τρόπους και υπολογίστε την εσωτερική και την εξωτερική κλασματοποίηση.

Απάντηση:

Θα μπορούσαμε να αναθέσουμε την 7K διαδικασία στην περιοχή των 8K και την 3K διαδικασία στην 4K περιοχή, παράγοντας 2K εσωτερικής κλασματοποίησης.

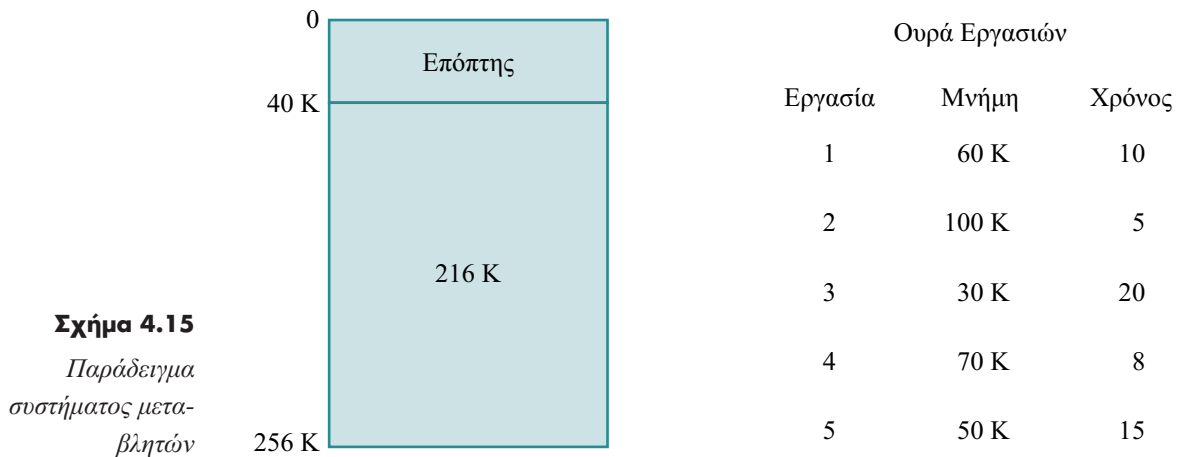
Ανάλογα με τον αλγόριθμο χρονοδρομολόγησης διαδικασιών που τρέχει στο σύστημα, θα μπορούσαμε να βάλουμε τις δύο διαδικασίες των 6K σε αναμονή (δημιουργώντας 10K εξωτερικής κλασματοποίησης) ή να τρέξουμε μία από αυτές στην περιοχή των 10K. Αν αναθέσουμε έτσι τη μια διαδικασία, θα δημιουργήσουμε πρόσθετα 4K εσωτερικής κλασματοποίησης, αλλά θα εξαλείψουμε την εξωτερική κλασματοποίηση. Εάν οι περιοχές ταίριαζαν ακριβώς με τα μεγέθη των διαδικασιών, θα μπορούσαμε να τρέξουμε και τις 4 διαδικασίες, χωρίς ούτε εξωτερική ούτε εσωτερική κλασματοποίηση.

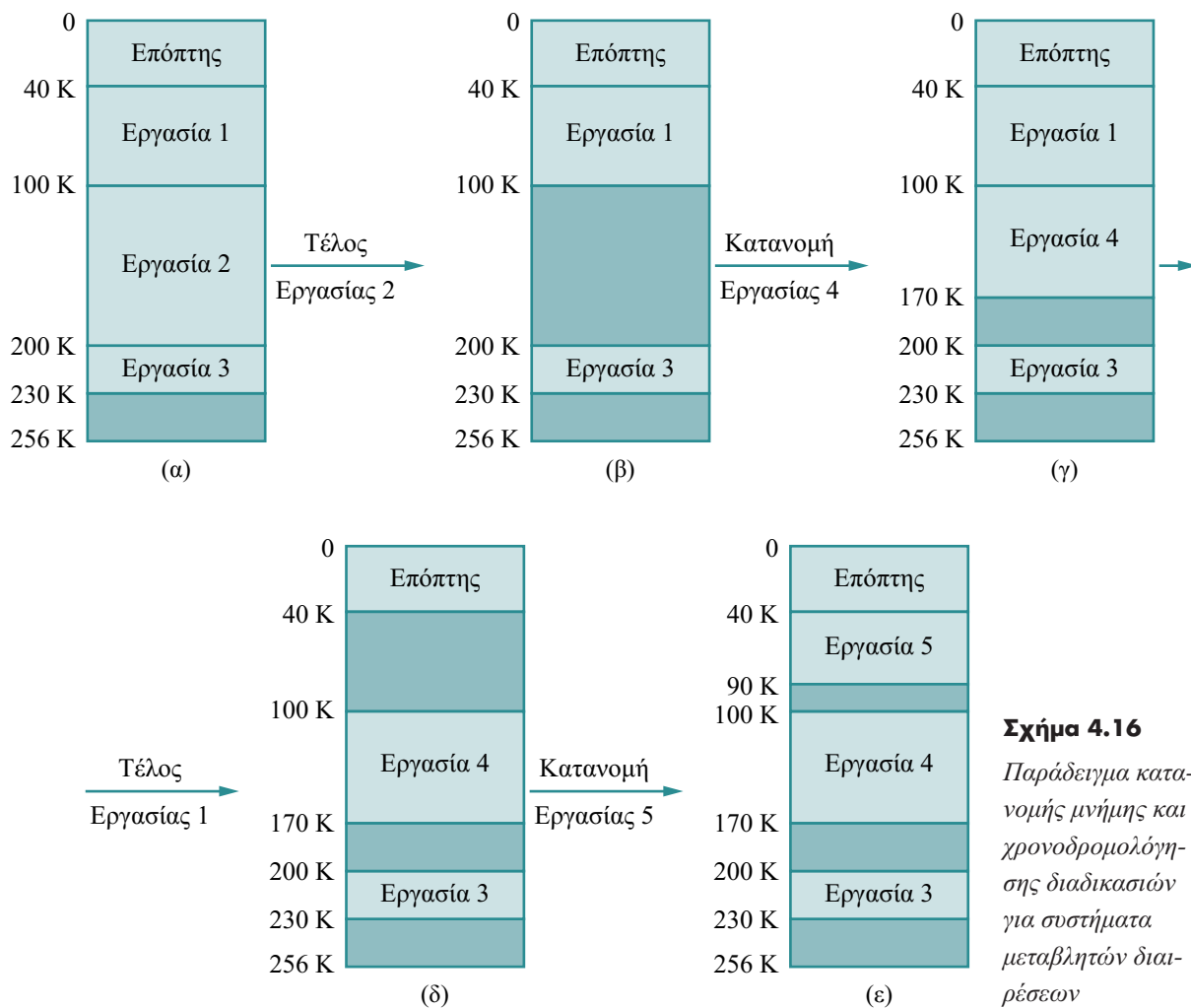
4.5.4 Μεταβλητές διαιρέσεις (Variable Partitions)

Το κύριο πρόβλημα με το σύστημα σταθερών περιοχών είναι ο καθορισμός των βέλτιστων μεγεθών των περιοχών ώστε να ελαχιστοποιηθεί η εσωτερική και εξωτερική κλασματοποίηση. Δυστυχώς, με ένα δυναμικό σύνολο διαδικασιών προς εκτέλεση, κατά πάσα πιθανότητα, δεν υπάρχει καμία βέλτιστη διαίρεση της μνήμης. Για παράδειγμα, έστω ότι έχουμε 120K διαθέσιμης μνήμης για διαδικασίες και ότι όλες οι διαδικασίες είναι 20K, εκτός από μια μεγάλη διαδικασία των 80K που τρέχει μια φορά την ημέρα. Πρέπει να διαθέσουμε μια περιοχή των 80K για να επιτρέψουμε σε αυτή τη διαδικασία να εκτελεστεί, αλλά, επειδή όλες οι διαδικασίες είναι 20K, αντιμετωπίζουμε 60K (ο μισός χώρος μνήμης των διαδικασιών) εσωτερικής κλασματοποίησης, εκτός από όταν τρέχει αυτή η μόνη μεγάλη διαδικασία μια φορά κάθε μέρα.

Μια λύση σε αυτό το πρόβλημα είναι να επιτρέψουμε στα μεγέθη των περιοχών να αλλάζουν δυναμικά. Το σύστημα μεταβλητών διαιρέσεων διαχείρισης μνήμης είναι αρκετά απλό: Το ΛΣ κρατάει έναν πίνακα που δείχνει ποια τμήματα της μνήμης είναι διαθέσιμα και ποια κατειλημμένα. Αρχικά, όλη η μνήμη για διαδικασίες είναι διαθέσιμη και θεωρείται ως μια μεγάλη περιοχή διαθέσιμης μνήμης, μια οπή (hole). Όταν μια διαδικασία έρχεται και χρειάζεται μνήμη, ο διαχειριστής μνήμης ψάχνει για μια οπή όπου να τη χωράει, και της διαθέτει μόνο τη μνήμη που χρειάζεται. Για παράδειγμα, υποθέστε ότι υπάρχουν 256K διαθέσιμης μνήμης και παραμένων επόπτης των 40K, άρα 216K για διαδικασίες (Σχήμα 4.15). Με δεδομένα την ουρά του σχήματος και FCFS αλγόριθμο χρονοδρομολόγησης διαδικασιών, μπορεί να ανατεθεί μνήμη στις διαδικασίες 1, 2 και 3, δημιουργώντας το «χάρτη μνήμης» (memory map) του Σχήματος 4.16 (α). Σε αυτή την περίπτωση υπάρχει εξωτερική κλασματοποίηση 26K. Αν το ΛΣ χρησιμοποιούσε τον round – robin αλγόριθμο χρονοδρομολόγησης της CPU με κβάντο μιας χρονικής μονάδας, η διαδικασία 2 θα τερμάτιζε τη στιγμή 14, ελευθερώνοντας τη μνήμη της [Σχήμα 4.16 (β)]. Ο διαχειριστής της

μνήμης θα φόρτωνε στη μνήμη την επόμενη διαδικασία, νούμερο 4, από την ουρά. Ο χάρτης μνήμης σε αυτή την περίπτωση φαίνεται στο Σχήμα 4.16 (γ). Η διαδικασία 1 θα τερμάτιζε τη στιγμή 28 [Σχήμα 4.16 (δ)] και μετά δρομολογείται η διαδικασία 5 [Σχήμα 4.16 (ε)]. Αυτό το παράδειγμα σκιαγραφεί αρκετά σημεία του συστήματος μεταβλητών διαιρέσεων. Γενικά, ανά πάσα στιγμή υπάρχει ένα σύνολο από οπές διαφόρων μεγεθών, σκορπισμένων μέσα στη μνήμη. Όταν μια διαδικασία αφιχθεί και χρειάζεται μνήμη, ο διαχειριστής μνήμης ψάχνει αυτό το σύνολο για μια οπή που να τη χωράει και ό,τι περισσέψει επιστρέφεται στο σύνολο των οπών. Όταν μια διαδικασία τερματίζει, απελευθερώνει την περιοχή της μνήμης της, που εντάσσεται στο σύνολο των οπών. Αν αυτή η νέα οπή είναι «γειτονική» (adjacent) με άλλες, τότε οι γειτονικές αυτές οπές ενώνονται για να σχηματίσουν μια μεγαλύτερη, η οποία πιθανόν να μπορεί να χωρέσει μια από τις διαδικασίες που περιμένουν στην ουρά ελλείψει ικανής συνεχούς μνήμης.



**Σχήμα 4.16**

Παράδειγμα κατανομής μνήμης και χρονοδρομολόγησης διαδικασιών για συστήματα μεταβλητών διαιρέσεων

Αυτή η πολιτική διαχείρισης μνήμης είναι μια ιδιαίτερη εφαρμογή του γενικότερου προβλήματος δυναμικής κατανομής χώρου αποθήκευσης, που συζητήθηκε στην Υποενότητα 4.3.2. Οι πιο κοινοί αλγόριθμοι / πολιτικές κατανομής μνήμης είναι του «πρώτου ταιριάσματος» (first-fit) και του «καλύτερου ταιριάσματος» (best-fit).

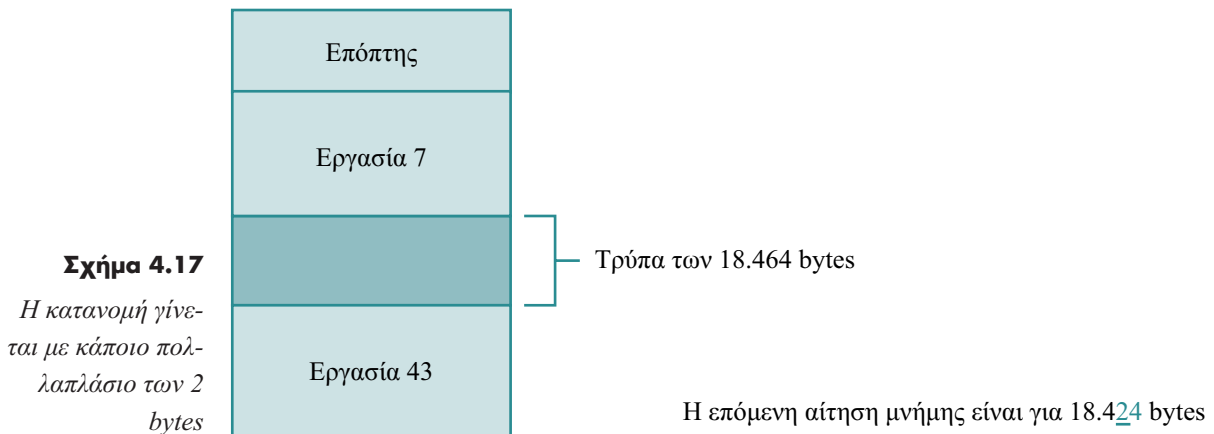
Άσκηση Αυτοαξιολόγησης 3.20

Μπορείτε να φανταστείτε και να περιγράψετε σε μια παράγραφο τι μπορεί να ορίζουν οι δύο αυτοί αλγόριθμοι;

Το ΛΣ ορίζει την περιοχή μνήμης κάθε διαδικασίας με δύο καταχωρητές (ανά διαδικασία), που περιέχουν το άνω και κάτω όριο της περιοχής και ενημερώνονται τη στιγμή που φορτώνεται η διαδικασία στη μνήμη. Έτσι, κάθε διεύθυνση που παράγεται από την CPU ελέγχεται και το ΛΣ επιτρέπει σε κάθε διαδικασία την πρόσβαση μόνο στη δική της περιοχή μνήμης, προστατεύοντας κάθε διαδικασία από τις υπόλοιπες.

Παρατηρήστε ότι το λογισμικό καθορίζει τη διαφορά μεταξύ του συστήματος σταθερών περιοχών και του συστήματος μεταβλητών διαιρέσεων. Το υλικό είναι το ίδιο σε αυτά τα δύο συστήματα.

Ένα άλλο πρόβλημα που παρουσιάζεται στο σύστημα μεταβλητών διαιρέσεων παρουσιάζεται στο Σχήμα 4.17. Θεωρήστε την οπή των 18.464 bytes. Εάν η επόμενη διαδικασία χρειάζεται 18.462 bytes, τι κάνουμε; Αν ανατεθεί ακριβώς η προηγούμενη περιοχή, παραμένει μία οπή των 2 bytes. Το «ποσό μνήμης» (overhead) που χρειάζεται για να κρατάμε πληροφορίες γι' αυτή την οπή θα είναι πολύ μεγαλύτερο από την ίδια την οπή. Η γενική προσέγγιση είναι να αναθέτονται οι πολύ μικρές οπές ως μέρος της μεγαλύτερης αίτησης για μνήμη. Συνεπώς, η μνήμη που ανατίθεται μπορεί να είναι ελαφρά μεγαλύτερη της απαιτούμενης, επανεισάγοντας ένα μικρό ποσό εσωτερικής κλασματοποίησης. Για παράδειγμα, ο CDC 6600 ανάθετε μνήμη μόνο σε ποσότητες των 64 bytes, ο IBM 360 σε ποσότητες των 2K και ο PDP-11 χρησιμοποιούσε μια ελάχιστη περιοχή των 8 bytes.



ΠΟΛΙΤΙΚΕΣ ΔΙΑΧΕΙΡΙΣΗΣ ΜΝΗΜΗΣ ΣΕ ΣΥΣΤΗΜΑΤΑ ΜΕΤΑΒΛΗΤΩΝ ΔΙΑΙΡΕΣΕΩΝ

Όπως και στο σύστημα σταθερών περιοχών, το σύστημα μεταβλητών διαιρέσεων αλληλεπιδρά ισχυρά με τη χρονοδρομολόγηση διαδικασιών. Σε οποιαδήποτε δεδομένη στιγμή, υπάρχει ένας κατάλογος όπου διατηρούνται οι διαθέσιμες περιοχές μνή-

μης και μια ουρά διαδικασιών που απαιτούν μνήμη. Ο χρονοδρομολογητής διαδικασιών διατάσσει την ουρά σύμφωνα με κάποιον αλγόριθμο χρονοδρομολόγησης. Η μνήμη ανατίθεται στις διαδικασίες με βάση κάποια πολιτική διαχείρισης μνήμης, μέχρις ότου να μην υπάρχει διαθέσιμη περιοχή μνήμης (οπή) αρκετά μεγάλη για καμιά διαδικασία στην ουρά. Ο χρονοδρομολογητής διαδικασιών μπορεί τότε να περιμένει έως ότου μια αρκετά μεγάλη περιοχή γίνει διαθέσιμη ή μπορεί να διατρέξει την ουρά διαδικασιών για να δει αν οι μικρότερες απαιτήσεις μνήμης κάποιας χαμηλότερης προτεραιότητας διαδικασίας μπορούν να ικανοποιηθούν. Αυτή η απόφαση παράγει μια επιλογή μεταξύ χρονοδρομολόγησης της CPU, με ή χωρίς υπερπήδηση.

Η «χρήση της μνήμης» (memory utilization) είναι γενικά καλύτερη στο σύστημα μεταβλητών διαιρέσεων παρά στο σύστημα σταθερών περιοχών. Υπάρχει μικρή ή καθόλου εσωτερική κλασματοποίηση.

Άσκηση Αυτοαξιολόγησης 4.13

Πότε υπάρχει μικρή και πότε καθόλου εσωτερική κλασματοποίηση;

Μπορεί όμως να υπάρχει εξωτερική κλασματοποίηση. Κοιτάζοντας το Σχήμα 4.16 βλέπουμε δύο τέτοιες περιπτώσεις. Στο Σχήμα 4.16 (α) υπάρχει συνολική εξωτερική κλασματοποίηση των 26K, χώρος που είναι πολύ μικρός για να ικανοποιήσει οποιαδήποτε απαίτηση διαδικασίας. Στο Σχήμα 4.16 (γ) έχουμε συνολική εξωτερική κλασματοποίηση 56K (= 30K + 26K). Αυτός ο χώρος είναι αρκετά μεγάλος για να εκτελεστεί η διαδικασία 5 (που χρειάζεται 50K), αλλά αυτή η ελεύθερη μνήμη δεν είναι συνεχής.

Αυτό το πρόβλημα κλασματοποίησης μπορεί να είναι αρκετά σοβαρό. Στη μέση περίπτωση, ο αριθμός των άδειων περιοχών είναι ο μισός του αριθμού των κατειλημμένων περιοχών. Το φαινόμενο αυτό αποδείχτηκε από τον Knuth και είναι γνωστό ως ο «κανόνας του πενήντα τοις εκατό».

Άσκηση Αυτοαξιολόγησης 4.14

Αποδείξτε τον κανόνα του πενήντα τοις εκατό.

Αν όλη αυτή η μνήμη υπήρχε σε μια μεγάλη ελεύθερη περιοχή, ίσως να μπορούσαν να εκτελεστούν αρκετές ακόμα διαδικασίες. Η επιλογή του αλγόριθμου του «πρώ-

του ταιριάσματος» ή του «καλύτερου ταιριάσματος» μπορεί να επηρεάσει το ποσοστό της κλασματοποίησης.

Άσκηση Αυτοαξιολόγησης 4.15

Κάτω από ποιες συνθήκες είναι δυνατό το «πρώτο ταίριασμα» να έχει καλύτερο αποτέλεσμα από το «καλύτερο ταίριασμα»;

Ανεξάρτητα όμως από το ποιοι αλγόριθμοι χρησιμοποιούνται, η εξωτερική κλασματοποίηση θα είναι πάντα ένα πρόβλημα.

Άσκηση Αυτοαξιολόγησης 4.16

Κατασκευάστε ένα παράδειγμα όπου η στατική ανάθεση μνήμης είναι καλύτερη από τη δυναμική.

Άσκηση Αυτοαξιολόγησης 4.17

Θεωρούμε ένα σύστημα εναλλαγής στο οποίο η μνήμη περιέχει τα ακόλουθα μεγέθη κενών κατά σειρά: 10K, 4K, 20K, 18K, 7K, 9K, 12K και 15K. Ποιο κενό χρησιμοποιείται για καθεμιά από τις ακόλουθες τρεις αιτήσεις, όταν αυτές φτάνουν με την εξής σειρά: 12K, 10K, 8K, με τον αλγόριθμο πρώτης τοποθέτησης, τον αλγόριθμο καλύτερης τοποθέτησης, τον αλγόριθμο χειρότερης τοποθέτησης και τον αλγόριθμο επόμενης τοποθέτησης;

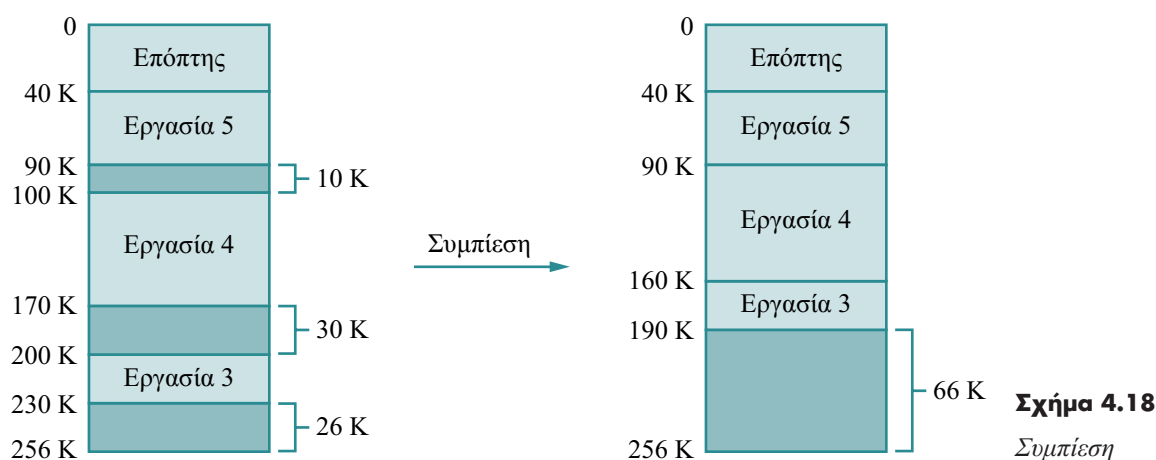
Απάντηση:

Κενά: Α 10K, Β 4K, Γ 20K, Δ 18K, Ε 7K, Ζ 9K, Η 12K, Θ 15K

Αίτηση \ Αλγόριθμος	Αλγόριθμος πρώτης τοποθ.	Αλγόριθμος καλύτερης τοπ.	Αλγ. χειρότερης τοποθέτησης	Αλγόριθμος επόμενης τοπ.
12K	Γ	Η	Γ	Γ
10K	Α	Α	Δ	Δ
8K	Γ	Γ	Θ	Ζ

4.5.5 Συμπίεση (Compaction)

Μια λύση στο πρόβλημα του τεμαχισμού είναι η συμπίεση, δηλαδή η αναδιάταξη της μνήμης, ώστε να βρεθεί όλη η ελεύθερη μνήμη σε μια μεγάλη περιοχή. Για παράδειγμα, ο χάρτης μνήμης του Σχήματος 4.16 (ε) μπορεί να μετασχηματιστεί όπως φαίνεται στο Σχήμα 4.18. Οι τρεις οπές των 10K, 30K και 26K θα συμπιεστούν σε μία των 66K.



Η συμπίεση δεν είναι πάντοτε δυνατή. Παρατηρήστε ότι στο Σχήμα 4.18 οι διαδικασίες 4 και 3 μεταφέρονται σε διαφορετικές περιοχές της μνήμης. Για να μπορέσουν αυτές οι διαδικασίες να εκτελεστούν στις νέες τους θέσεις, όλες οι εσωτερικές διευθύνσεις πρέπει να μετατοπιστούν.

Άσκηση Αυτοαξιολόγησης 4.18

Σε ποια περίπτωση δεν είναι δυνατόν να γίνει συμπίεση λόγω του ότι οι διευθύνσεις μιας διαδικασίας δε θα μπορούσαν να μετατοπιστούν;

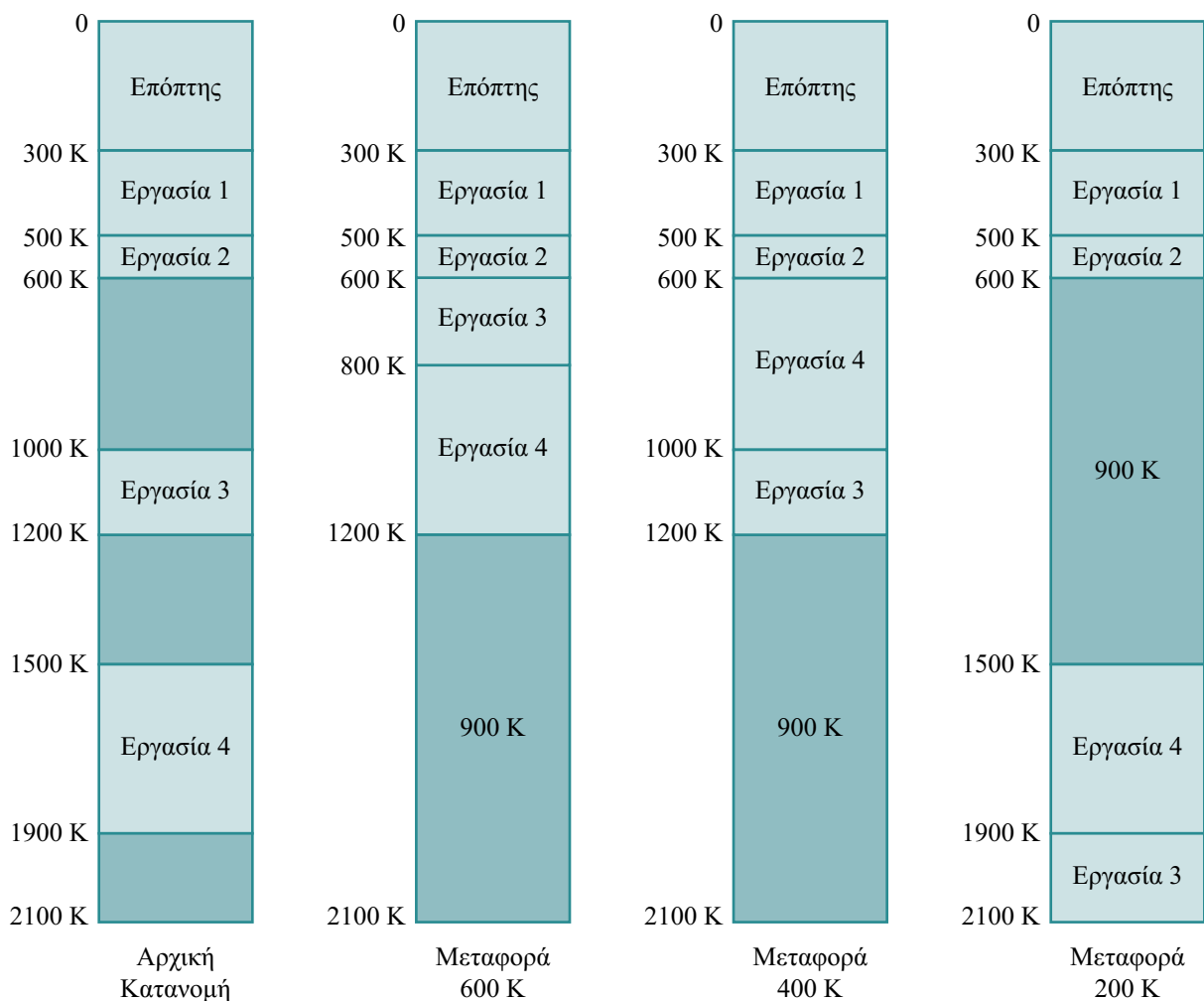
Απάντηση:

Όταν η μετατόπιση είναι στατική και γίνεται κατά το χρόνο συμβολομετάφρασης ή φορτώματος.

Όταν οι διευθύνσεις μετατοπίζονται δυναμικά (όπως στον CDC 6600), η μετατόπιση απαιτεί απλώς τη μεταφορά του προγράμματος και των δεδομένων και μετά την αλλαγή του καταχωρητή βάσης, ώστε να αντικατοπτρίζει τη νέα διεύθυνση βάσης.

Όταν η συμπίεση είναι δυνατή, πρέπει να προσδιορίσουμε το κόστος της. Ο πιο απλός αλγόριθμος συμπίεσης είναι να μεταφέρουμε όλες τις διαδικασίες προς τη μία άκρη της μνήμης: όλες οι οπές μετακινούνται προς την αντίθετη κατεύθυνση, παράγοντας μία μεγάλη οπή διαθέσιμης μνήμης. Αυτός ο τρόπος μπορεί να είναι αρκετά ακριβός.

Θεωρήστε την κατανομή μνήμης του Σχήματος 4.19. Αν χρησιμοποιήσουμε αυτό τον απλό αλγόριθμο, πρέπει να μετακινήσουμε τις διαδικασίες 3 και 4, μεταφέροντας συνολικά 600K.



Σχήμα 4.19

Σύγκριση μερικών διαφορετικών τρόπων συμπίεσης μνήμης

Η επιλογή μιας βέλτιστης στρατηγικής συμπίεσης είναι αρκετά δύσκολη.

Ιστορικά, το ΛΣ SCOPE του CDC 6600 χρησιμοποιούσε σύστημα μεταβλητών διαιρέσεων με συμπίεση. Μέχρι οκτώ διαδικασίες επιτρεπόταν να υπάρχουν στην κύρια

μνήμη κάθε φορά. Όταν η διαδικασία τερματίζει, η μνήμη συμπιέζοταν για να κρατιέται όλος ο ελεύθερος χώρος σε μια οπή στο τέλος της μνήμης.

Όπως και με άλλα συστήματα, η εναλλαγή μπορεί να συνδυαστεί με το σύστημα μεταβλητών διαιρέσεων.

Άσκηση Αυτοαξιολόγησης 4.19

Πώς θα μπορούσαν να μεταφερθούν οι διαδικασίες με διαφορετικό τρόπο, έτσι ώστε ο συνολικός αριθμός από bytes που θα μεταφερόταν να είναι μικρότερος; Κάντε παρατηρήσεις για το πού θα είναι η περιοχή της διαθέσιμης μνήμης και τι περιορισμούς μπορεί να επιφέρει.

Απάντηση:

Σε αυτή την περίπτωση, θα μπορούσε απλώς να μεταφερθεί η διαδικασία 4 «πάνω από» τη διαδικασία 3, μεταφέροντας έτσι μόνο 400K, ή να μεταφερθεί η διαδικασία 3 «κάτω από» τη διαδικασία 4, μεταφέροντας μόνο 200K. Παρατηρήστε ότι στην τελευταία περίπτωση η μία μεγάλη οπή διαθέσιμης μνήμης δεν είναι στο τέλος της μνήμης αλλά στη μέση. Επίσης, παρατηρήστε ότι, αν η ουρά περιείχε μόνο μια διαδικασία που ήθελε 450K, θα μπορούσαμε να ικανοποιήσουμε αυτή τη συγκεκριμένη απαίτηση μετακινώντας τη διαδικασία 2 κάπου αλλού (π.χ. κάτω από τη διαδικασία 4). Παρ' όλο που αυτή η λύση δε δημιουργεί τη μεγαλύτερη δυνατή οπή, δημιουργεί μια αρκετά μεγάλη για να ικανοποιήσει την άμεση απαίτηση.

Μια άλλη προσέγγιση που χρησιμοποιούσαν υπολογιστές, όπως ο PDP-1, είναι το σπάσιμο της μνήμης που χρειάζεται μια διαδικασία σε δύο μέρη. Ο PDP-1 είχε δύο ζευγάρια καταχωρητών βάσης / ορίου. Η μνήμη χωριζόταν στη μέση με τη χρήση του bit υψηλότερης τάξης της διεύθυνσης.^[9] Η χαμηλή μνήμη μετατοπίζεται / περιορίζεται από το ζευγάρι καταχωρητών βάσης / ορίου 0, η υψηλή μνήμη μετατοπίζεται / περιορίζεται από το ζευγάρι καταχωρητών βάσης / ορίου 1. Από σύμβαση, οι μεταφραστές και συμβολομεταφραστές τοποθετούν τις τιμές που μόνο διαβάζονται (όπως σταθερές και εντολές) στην υψηλή μνήμη και τις μεταβλητές στη χαμηλή μνήμη. Bits προστασίας συσχετίζονται με κάθε ζευγάρι καταχωρητών και μπορούν να επιβάλλουν τον περιορισμό (μόνο διαβάσματος) της υψηλής μνήμης. Αυτή η διάταξη επιτρέπει στις διαδικασίες (που είναι αποθηκευμένες στην υψηλή μνήμη) να

[9] Το bit υψηλότερης τάξης της διεύθυνσης είναι το πρώτο bit από αριστερά.

χρησιμοποιούνται από πολλές διαδικασίες παράλληλα. Η καθεμιά από αυτές τις διαδικασίες έχει το δικό της τμήμα χαμηλής μνήμης.

Ο Univac 1108 είχε μια παρόμοια διάταξη, που βασίζεται σε ένα διαφορετικό τρόπο διαχωρισμού. Σε κάθε αλληλεπίδραση της CPU και της μνήμης, η CPU διαβάζει μια εντολή ή διαβάζει / αποθηκεύει δεδομένα από/προς τη μνήμη. Είναι γνωστό πότε η αλληλεπίδραση CPU και μνήμης γίνεται για να διαβαστεί μια εντολή ή για να διαβαστούν / αποθηκευτούν δεδομένα. Έτσι, ο Univac 1108 έχει δύο ζευγάρια καταχωρητών βάσης / ορίου: ένα για τις εντολές και ένα για τα δεδομένα. Το ζευγάρι καταχωρητών βάσης / ορίου για τις εντολές είναι «αυτόματου διαβάσματος μόνο» (automatically read-only), ώστε οι διαδικασίες να μπορούν να χρησιμοποιούνται από πολλές άλλες διαδικασίες.

Και στις δύο περιπτώσεις, με το διαχωρισμό των εντολών και των δεδομένων και με τη μετατόπιση του καθενός ξεχωριστά, είναι δυνατόν πολλές διαδικασίες να χρησιμοποιούν την ίδια διαδικασία. Έτσι, γίνεται καλύτερη χρήση της μνήμης, μειώνοντας και την κλασματοποίηση και τα πολλαπλά αντίγραφα του ίδιου κώδικα, και ειδικότερα του συχνά χρησιμοποιούμενου κώδικα, όπως οι μεταφραστές, οι επεξεργαστές κειμένου (editors) κτλ.

4.6 Σελιδοποίηση (Paging)

Ο εξωτερικός κατακερματισμός μειώνει σοβαρά την απόδοση της διαχείρισης της μνήμης με μεταβλητές υποδιαιρέσεις. Αυτό είναι γενικότερο πρόβλημα κατακερματισμού της μνήμης. Επειδή η μνήμη κάθε διαδικασίας πρέπει να είναι συνεχής,

Άσκηση Αυτοαξιολόγησης 4.20

Γιατί;

αυτή η διασκορπισμένη, ασυνεχής μνήμη δεν μπορεί να χρησιμοποιηθεί. Αυτό το πρόβλημα έχει δύο γενικές λύσεις: (1) Η συμπίεση αλλάζει την κατανομή της μνήμης για να κάνει τον ελεύθερο χώρο συνεχή και, επομένως, αξιοποιήσιμο. (2) Η σελιδοποίηση επιτρέπει σε μια διαδικασία να αξιοποιεί μη συνεχή μνήμη. Οι διαδικασίες απαιτούν συνεχή χώρο μνήμης. Το ΛΣ έχει κατακερματισμένη μνήμη στη διάθεσή του. Αν υπήρχε τρόπος έτσι ώστε η κατακερματισμένη στην πραγματικότητα μνήμη να παρουσιάζεται στην κάθε διαδικασία ως ενιαία, τότε θα λυνόταν το πρόβλημα του κατακερματισμού. Η σελιδοποίηση είναι ένας τέτοιος τρόπος.

4.6.1 Υλικό (Hardware)

Έστω ότι ο χώρος μνήμης μιας διαδικασίας είναι 512K bytes, δηλαδή οι λογικές διευθύνσεις της παίρνουν τιμές από 0 έως 512K. Η διαδικασία εκτελείται σαν να είχε στη διάθεσή της τις διευθύνσεις 0 έως 1777777₈. Έστω ότι 1K bytes είναι μια «φυσική» μονάδα μνήμης στο συγκεκριμένο σύστημα (π.χ. επειδή το υλικό είναι φτιαγμένο έτσι ώστε η μεταφορά από το δίσκο στη μνήμη να γίνεται σε φουρνιές των 1K bytes). Ας ονομάσουμε αυτή τη μονάδα μνήμης ως «σελίδα». Για παράδειγμα, η διαδικασία αποτελείται από 512 σελίδες μνήμης. Κάθε διεύθυνση που παράγεται κατά την εκτέλεση της διαδικασίας είναι ένας αριθμός μεταξύ 0 και 1777777₈ (ή ένας 19ψήφιος δυαδικός αριθμός). Την κάθε διεύθυνση θα μπορούσαμε να την προσδιορίσουμε και διαφορετικά, με βάση τις σελίδες οι οποίες αποτελούν το χώρο διευθύνσεων της διαδικασίας. Στο πιο πάνω παράδειγμα, η διαδικασία αποτελείται από 512 σελίδες. Άρα με 9 bits μπορούμε να έχουμε διαφορετική διεύθυνση για καθεμία από αυτές τις σελίδες. Αφού κάθε σελίδα έχει μέγεθος 1K, τότε με 10 bits μπορούμε να προσδιορίσουμε κάθε byte μέσα στην κάθε σελίδα.

Άσκηση Αυτοαξιολόγησης 3.21

Γιατί με 9 bits υπάρχει διαφορετική διεύθυνση για καθεμία από 512 σελίδες και γιατί με 10 bits διαφορετική διεύθυνση για κάθε byte μέσα σε μια σελίδα 1K;

Απάντηση:

$2^9 = 512$ και $2^{10} = 1024 = 1\text{K bytes}$.

Ουσιαστικά χωρίσαμε το 19 bits της διεύθυνσης κάθε byte της διαδικασίας σε 2 ομάδες. Τα 9 bits υψηλότερης τάξης προσδιορίζουν τη σελίδα και τα υπόλοιπα 10 τη θέση μέσα στη σελίδα.

Άσκηση Αυτοαξιολόγησης 4.22

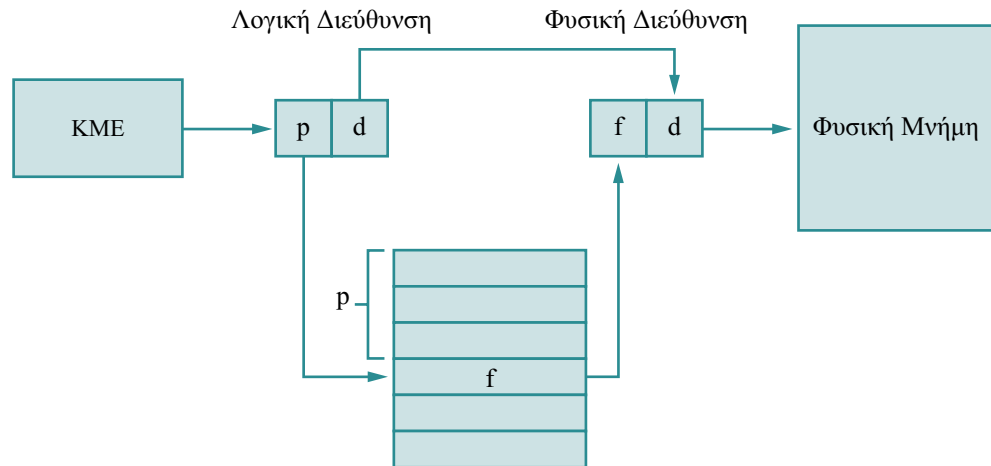
Ποια σελίδα και ποια θέση ορίζει η διεύθυνση 1357246₈;

Απάντηση: $1357246_8 = 1\ 011\ 101\ 111\ 010\ 100\ 110_2$

$= \boxed{101110111} \boxed{1010100110}$

$= \boxed{567} \boxed{1246}$, δηλαδή τη σελίδα 567₈ στη θέση 1246₈.

Στην πραγματικότητα, η διαδικασία δεν έχει στη διάθεσή της τις θέσεις μνήμης 0 – 1777777, αλλά, όπως είδαμε μέχρι τώρα, τις θέσεις X έως $X + 1777777$, δηλαδή τις 512 σελίδες, ξεκινώντας από τη σελίδα X/K , όπου X είναι η θέση που άρχισε η φόρτωση της διαδικασίας. Αν τώρα, αντί των 512 διαδοχικών σελίδων, το ΛΣ διαθέσει στη διαδικασία *οποιοσδήποτε* 512 σελίδες, κρατώντας έναν πίνακα αντιστοιχίας έτσι ώστε να ξέρει σε ποια πραγματική (φυσική) σελίδα μνήμης αντιστοιχεί η καθεμία από τις 512 λογικά διαδοχικές σελίδες της διαδικασίας, τότε δε χρειάζεται μισό MB συμπαγές στη μνήμη, αλλά μπορεί να εκμεταλλευτεί όλες τις σκόρπιες σελίδες που υπάρχουν ελεύθερες. Αυτή είναι η μέθοδος διαχείρισης μνήμης με σελιδοποίηση. Φυσικά, η αντιστοίχιση των λογικών σε φυσικές σελίδες πρέπει να γίνεται πολύ γρήγορα και στην πράξη να υποστηρίζεται από ειδικό υλικό. Το υλικό υποστήριξης της σελιδοποίησης φαίνεται στο Σχήμα 4.20. Κάθε διεύθυνση που παράγεται από την CPU χωρίζεται σε δύο μέρη: τον «αριθμό σελίδας» (p) (page number) και τη «μετατόπιση σελίδας» (d) (page offset). Ο αριθμός σελίδας χρησιμοποιείται ως δείκτης στον «πίνακα σελίδων» (page table). Ο πίνακας σελίδων περιέχει τη διεύθυνση βάσης της κάθε σελίδας στη φυσική μνήμη.

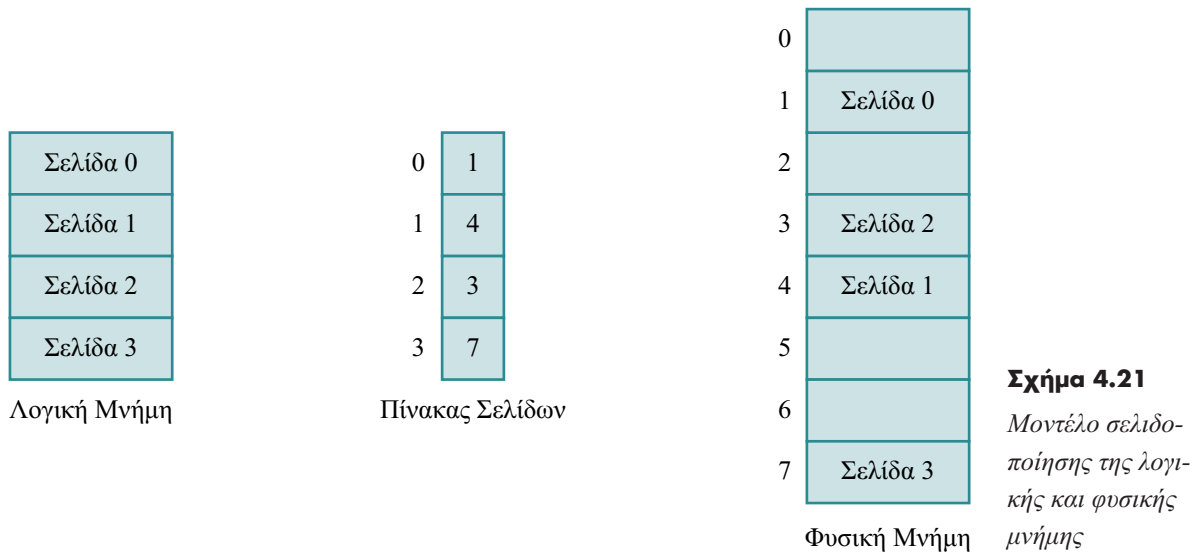


Σχήμα 4.20

Υλικό της σελιδοποίησης

Αυτή η διεύθυνση βάσης συνδυάζεται με τη μετατόπιση σελίδας για τον καθορισμό της διεύθυνσης φυσικής μνήμης που στέλνεται στη μονάδα μνήμης.

Το μοντέλο σελιδοποίησης της μνήμης φαίνεται στο Σχήμα 4.21. Η φυσική μνήμη σπάει σε σταθερού μήκους τμήματα, που ονομάζονται «πλαίσια σελίδας» (frames). Η λογική μνήμη επίσης σπάει σε τμήματα του ίδιου μεγέθους, που ονομάζονται «σελίδες» (pages). Όταν μια διαδικασία πρόκειται να εκτελεστεί, οι σελίδες της φορτώνονται σε όποια διαθέσιμα πλαίσια υπάρχουν και ο πίνακας σελίδων χρησιμοποιείται βασικά για να μεταφράζει τις σελίδες της διαδικασίας σε πλαίσια μνήμης.



Υπάρχουν και άλλες πληροφορίες σε κάθε εγγραφή του πίνακα σελίδων. Μερικές από τις πληροφορίες αυτές αφορούν το αν έγινε αναφορά σε μια σελίδα η οποία βρίσκεται στη μνήμη, αν έχει το δικαίωμα μια διαδικασία η οποία προσπαθεί, για παράδειγμα, να γράψει σε μια σελίδα να το κάνει και σε άλλες. Στη συνέχεια γίνεται εκτενής αναφορά στο πώς αναπαρίστανται τέτοιες πληροφορίες στον πίνακα σελίδων και σε τι χρησιμεύουν.

Το μέγεθος της σελίδας (και του πλαισίου σελίδας) ορίζεται από το υλικό. Το μέγεθος της σελίδας είναι τυπικά δύναμη του 2. Για παράδειγμα, ο IBM 370 χρησιμοποιεί 2048 ή 4096 bytes/σελίδα, ο XDS-940 2048 bytes/σελίδα. Ο Atlas και ο Sigma 7, επίσης, χρησιμοποιούσαν σελίδες των 512 bytes (Σχήμα 4.22). Γενικά, αν το μέγεθος της σελίδας είναι P , τότε η λογική διεύθυνση, U , παράγει έναν αριθμό σελίδας, p , και μία μετατόπιση (offset), d , ως εξής:

$$p = U \text{ div } P$$

$$d = U \bmod P$$

όπου div είναι η ακέραια διαίρεση και \bmod το υπόλοιπό της. Η επιλογή μιας δύναμης του 2 ως μεγέθους σελίδας καθιστά ιδιαίτερα εύκολη τη μετάφραση μιας λογικής διεύθυνσης σε έναν αριθμό και μία μετατόπιση σελίδας. Αν η σελίδα έχει μήκος 2^n μονάδες διευθυνσιοδότησης (bytes), τότε τα n «χαμηλής τάξης» bits της λογικής διεύθυνσης προσδιορίζουν τη μετατόπιση σελίδας και τα υπόλοιπα «υψηλής τάξης» bits τον αριθμό σελίδας. Έτσι, αν το μέγεθος της σελίδας είναι δύναμη του 2, μπορούμε να αποφύγουμε τη διαίρεση.

Σχήμα 4.22
Αριθμός των bits
για διάφορους
υπολογιστές με
σελιδοποίηση

Μηχανή	Διεύθυνση	Bits Σελίδας	Μετατόπιση
Atlas	20	11	9
DEC-10	18	9	9
Sigma 7	17	8	9
Nova 3/D	15	5	10
XDS-940	14	3	11
IBM 370	24	13 ή 12	11 ή 12
Vax	32	21	9
SPARC	32	20	12
68030	32	21	11

Για ένα πολύ απλό συγκεκριμένο παράδειγμα (χρησιμοποιείται μόνο για να κατανοήσετε τις έννοιες αυτές, οι μνήμες σήμερα έχουν μέγεθος 256M), ας δούμε τη μνήμη του Σχήματος 4.23. Χρησιμοποιώντας ένα μέγεθος σελίδας των 4 bytes και μια φυσική μνήμη των 32 bytes (8 σελίδες), δίνουμε ένα παράδειγμα του πώς μπορεί να αντιστοιχιστεί ο τρόπος αντίληψης της μνήμης από τη διαδικασία στη φυσική μνήμη. Η λογική διεύθυνση 0 είναι σελίδα 0, μετατόπιση 0. Στον πίνακα σελίδων βλέπουμε ότι η σελίδα 0 βρίσκεται στο πλαίσιο 5. Έτσι, η λογική διεύθυνση 0 αντιστοιχεί στη φυσική διεύθυνση 20 ($= 5 \times 4 + 0$). Η λογική διεύθυνση 3 (σελίδα 0, μετατόπιση 3) αντιστοιχεί στη φυσική διεύθυνση 23 ($= 5 \times 4 + 3$). Η λογική διεύθυνση 4 είναι σελίδα 1, μετατόπιση 0: σύμφωνα με τον πίνακα σελίδων, η σελίδα 1 αντιστοιχεί στο πλαίσιο 6. Έτσι, η λογική διεύθυνση 4 αντιστοιχεί στη φυσική διεύθυνση $(6 \times 4 + 0) = 24$. Η λογική διεύθυνση 13 αντιστοιχεί στη φυσική διεύθυνση 9.

Είναι σημαντικό να παρατηρήσετε ότι η σελιδοποίηση είναι μια μορφή δυναμικής μετατόπισης (relacation). Κάθε λογική διεύθυνση αντιστοιχίζεται μέσω του υλικού σελιδοποίησης σε κάποια φυσική διεύθυνση.

4.6.2 Χρονοδρομολόγηση διαδικασιών

Όπως και προηγουμένως, ο τρόπος διαχείρισης της μνήμης επηρεάζει το χρονοδρομολογητή διαδικασιών. Όταν μια διαδικασία φτάνει για εκτέλεση, ο χρονοδρομολογητής διαδικασιών εξετάζει το μέγεθός της, έστω n σελίδες. Αν υπάρχουν n πλαίσια σελίδων διαθέσιμα, ο δρομολογητής τα διαθέτει στη διαδικασία. Κάθε σελίδα της διαδικασίας αντιστοιχεί σε ένα πλαίσιο σελίδας (μια σελίδα στη φυσική μνήμη) και το ΛΣ διατηρεί το σχετικό πίνακα (Σχήμα 4.24).

0	a
1	b
2	c
3	d
4	e
5	f
6	g
7	h
8	i
9	j
10	k
11	l
12	m
13	n
14	o
15	p

Λογική Μνήμη

0	5
1	6
2	1
3	2

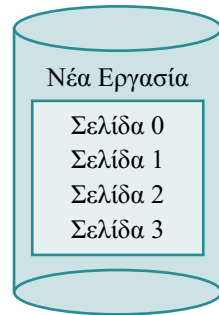
Πίνακας Σελίδων

0	
4	i j k l
8	m n o p
12	
16	
20	a b c d
24	e f g h
28	

Φυσική Μνήμη

Σχήμα 4.23

Παράδειγμα σε-
λι-
δοποίησης για
μνήμη 32 bytes με
σελίδες των 4
bytes



Λίστα Ελεύθερων Πλαισίων

14

13

18

20

15

13

14

15

16

17

18

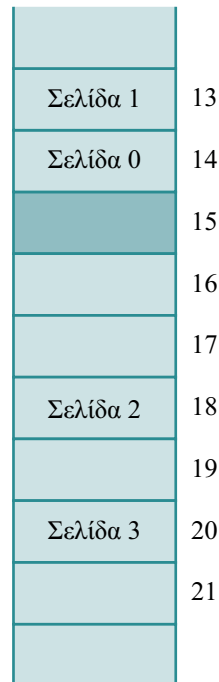
19

20

21

Φυσική Μνήμη

(α)



Λίστα Ελεύθερων Πλαισίων

15

0

14

1

13

2

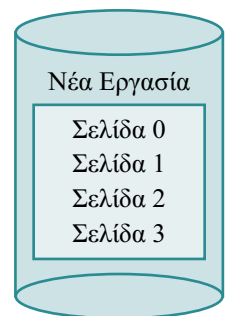
18

3

20

Πίνακας Σελίδων Νέας Εργασίας

(β)

**Σχήμα 4.24**

Κατανομή ελεύθερων πλαισίων: (α) πριν (β) μετά

Άσκηση Αυτοαξιολόγησης 4.23

Υπάρχει ή όχι εξωτερική κλασματοποίηση και εσωτερική κλασματοποίηση στο σύστημα διαχείρισης μνήμης με σελιδοποίηση και γιατί;

Άσκηση Αυτοαξιολόγησης 4.24

1. Έστω μια διαδικασία 8629 λέξεων των 32 bits, δηλαδή μια λέξη αντιστοιχεί σε 4 bytes. Τι εσωτερικό κατακερματισμό θα δημιουργούσε σελιδοποίηση όπου η κάθε σελίδα θα είχε μέγεθος 512B, 1KB, 2KB, 4KB, 8KB, 10KB; Ποια από τα μεγέθη αυτά είναι πιθανά μεγέθη σελίδας πραγματικών συστημάτων και γιατί;
2. Έστω μια διαδικασία N λέξεων των β bits και σελιδοποίηση ανά P bytes. Ποια N προκαλούν μέγιστο εσωτερικό κατακερματισμό;
3. Έστω ότι στη μνήμη του συστήματος βρίσκονται k διαδικασίες και η σελιδοποίηση είναι ανά P bytes. (α) Πόση μνήμη σπαταλιέται από εσωτερικό κατακερματισμό; (β) Τι υποθέσεις κάνατε για να δώσετε την απάντηση στο (α); (γ) Επομένως, πώς θα ελαχιστοποιούσαμε αυτή τη σπατάλη, δηλαδή πώς θα μικραίναμε το συνολικό εσωτερικό κατακερματισμό; (δ) Μπορείτε να σκεφτείτε γιατί ΔΕΝ κάνουμε την επιλογή αυτή, δηλαδή την ελαχιστοποίηση του εσωτερικού κατακερματισμού; [Η απάντηση στο (δ) είναι το αντικείμενο της επόμενης ενότητας. Δώστε όμως λίγο χρόνο σκεπτόμενοι πιθανούς λόγους πριν προχωρήσετε στη μελέτη της.]

4.6.3 Υλοποίηση του Πίνακα Σελίδων

Στην απλούστερη περίπτωση, ο πίνακας σελίδων υλοποιείται ως ένα σύνολο εξειδικευμένων καταχωρητών. Ο διεκπεραιωτής της CPU επαναφορτώνει αυτούς τους καταχωρητές, όπως επαναφορτώνει και τους άλλους καταχωρητές προγράμματος. Οι εντολές για τη φόρτωση ή την τροποποίηση των καταχωρητών του πίνακα σελίδων είναι φυσικά με προνόμια, έτσι ώστε μόνο το ΛΣ να μπορεί να αλλάξει το χάρτη μνήμης. Αυτή η προσέγγιση χρησιμοποιήθηκε στον XDS-940, που είχε 8 σελίδες των 2048 bytes η καθεμία και 8 καταχωρητές του πίνακα σελίδων. Ο Nova 3/D είχε 32 σελίδες των 1024 bytes και 32 καταχωρητές του πίνακα σελίδων. Ο Sigma 7 είχε αριθμό σελίδας των 8 bits, απαιτώντας 256 καταχωρητές για τον πίνακα σελίδων του. Έτσι, μπορεί να χρειαστούν από 8 έως 256 καταχωρητές για τον πίνακα σελίδων του. Αυτοί οι καταχωρητές μπορεί να είναι φτιαγμένοι από υλικό υψηλής ταχύ-

τητας, ώστε η μετάφραση διευθύνσεων σελιδοποίησης να έχει μεγάλη αποδοτικότητα. Κάθε προσπέλαση στη μνήμη πρέπει να περάσει από τον πίνακα σελίδων, και γι' αυτό το λόγο η ταχύτητα με την οποία παίρνουμε απάντηση από τον πίνακα επηρεάζει σε μεγάλο βαθμό την αποδοτικότητα του συστήματος.

Η χρήση καταχωρητών για τον πίνακα σελίδων είναι ικανοποιητική λύση αν ο πίνακας είναι σχετικά μικρός. Όμως, ο DEC-10 έχει 512 σελίδες, ο IBM 370 έχει μέχρι 4096 σελίδες, ενώ το σύστημα Multics έχει δυνατότητα για 16.777.216 σελίδες. Γι' αυτούς τους υπολογιστές η χρήση γρήγορων καταχωρητών για την υλοποίηση του πίνακα σελίδων δεν είναι εφικτή λύση (οι καταχωρητές κοστίζουν ακριβά). Γι' αυτό ο πίνακας σελίδων κρατιέται στην κύρια μνήμη και ένας «Καταχωρητής Βάσης Πίνακα Σελίδων» (Page Table Base Register – PTBR) δείχνει τον πίνακα σελίδων. Η αλλαγή πίνακα σελίδων απαιτεί την αλλαγή αυτού του καταχωρητή, μειώνοντας ουσιαστικά το χρόνο αλλαγής περιεχομένου των καταχωρητών.

Το πρόβλημα σε αυτό το σύστημα διαχείρισης μνήμης είναι ο χρόνος που απαιτείται για την προσπέλαση μιας θέσης μνήμης της διαδικασίας. Αν θέλουμε να προσπελάσουμε τη θέση i , πρέπει πρώτα να ανατρέξουμε στον πίνακα σελίδων με δείκτη την τιμή του PTBR μετατοπισμένη κατά τον αριθμό σελίδας για την i . Αυτή η διαδικασία απαιτεί μια προσπέλαση μνήμης. Μας δίνει τον αριθμό πλαισίου, ο οποίος, συνδυαζόμενος με τη μετατόπιση σελίδας παράγει την πραγματική διεύθυνση. Τότε, μπορούμε να προσπελάσουμε την επιθυμητή θέση στη μνήμη. Με αυτό τον τρόπο δύο προσπελάσεις μνήμης χρειάζονται για την προσπέλαση μιας λέξης^[10] (μία για τον πίνακα σελίδων και μία για τη λέξη). Έτσι, διπλασιάζεται ο χρόνος προσπέλασης στη μνήμη, πράγμα απαράδεκτο (Γιατί;).

Για να αντιμετωπιστεί αυτό το πρόβλημα, χρησιμοποιήθηκε μια ειδική, μικρή μνήμη, που έχει διάφορες ονομασίες, όπως συσχετιστικοί καταχωρητές (associative registers) ή κρυφή μνήμη (cache) ή look-aside μνήμη ή μνήμη διευθυνσιοδοτούμενη από τα περιεχόμενα (contact addressable memory). Ένα σύνολο συσχετιστικών καταχωρητών είναι φτιαγμένο από ιδιαίτερα υψηλής ταχύτητας μνήμη. Κάθε καταχωρητής αποτελείται από δύο μέρη: ένα κλειδί και μια τιμή. Όταν στους συσχετιστικούς καταχωρητές δίνεται ένα στοιχείο, αυτό συγκρίνεται με όλα τα κλειδιά ταυτόχρονα. Αν βρεθεί, το αντίστοιχο πεδίο τιμής εξάγεται. Το ψάξιμο είναι ταχύτατο, όμως το απαιτούμενο υλικό είναι ακριβό.

[10] Μια λέξη αντιστοιχεί σε έναν αριθμό από bytes. Στις μέρες μας, μια σελίδα είναι ίση με 4 ή 8 bytes.

Οι συσχετιστικοί καταχωρητές χρησιμοποιούνται με τους πίνακες σελίδων ως εξής: οι συσχετιστικοί καταχωρητές περιέχουν μερικά μόνο από τα δεδομένα του πίνακα σελίδων. Όταν μια λογική διεύθυνση δημιουργείται από την CPU, ο αριθμός σελίδας της δίνεται σε ένα σύνολο συσχετιστικών καταχωρητών που περιέχουν αριθμούς σελίδων και τους αντίστοιχους αριθμούς πλαισίων. Αν ο αριθμός σελίδας βρεθεί στους συσχετιστικούς καταχωρητές, ο αριθμός πλαισίου είναι άμεσα διαθέσιμος και χρησιμοποιείται για να προσπελαστεί η μνήμη.

Αν ο αριθμός σελίδας δεν υπάρχει στους συσχετιστικούς καταχωρητές, τότε μια αναφορά μνήμης στον πίνακα σελίδων είναι απαραίτητη. Όταν βρεθεί ο αριθμός πλαισίου, μπορούμε να τον χρησιμοποιήσουμε για να προσπελάσουμε τη μνήμη. Επίσης, προσθέτουμε τον αριθμό σελίδας και τον αριθμό πλαισίου στους συσχετιστικούς καταχωρητές, ώστε στην επόμενη αναφορά αυτοί να βρεθούν πολύ γρήγορα.

Η συχνότητα με την οποία ένας αριθμός σελίδας βρίσκεται στους συσχετιστικούς καταχωρητές («λόγος επιτυχίας») συνδέεται προφανώς με τον αριθμό των συσχετιστικών καταχωρητών. Με 8 ή 16 καταχωρητές μπορούμε να έχουμε λόγο επιτυχίας της τάξης του 85%, δηλαδή με πιθανότητα 0,85 βρίσκουμε τον επιθυμητό αριθμό σελίδας στους συσχετιστικούς καταχωρητές. Για παράδειγμα, αν χρειάζονται 50ns για να ψάξουμε στους συσχετιστικούς καταχωρητές και 750ns για να προσπελάσουμε τη μνήμη, τότε μια χαρτογραφημένη αναφορά μνήμης (χρήση συσχετιστικών καταχωρητών) χρειάζεται 800ns όταν ο αριθμός σελίδας είναι στους συσχετιστικούς καταχωρητές. Αν αποτύχουμε να βρούμε τον αριθμό σελίδας (50ns), τότε πρέπει πρώτα να προσπελάσουμε τη μνήμη για τον πίνακα σελίδων / αριθμό πλαισίου (750ns) και μετά να προσπελάσουμε την επιθυμητή λέξη στη μνήμη (750ns), σύνολο 1550 ns.

Άσκηση Αυτοαξιολόγησης 4.25

Ποιος είναι ο μέσος χρόνος προσπέλασης μνήμης; Ποιος θα ήταν αν είχαμε 90% επιτυχία (η σελίδα βρίσκεται στη συσχετιστική μνήμη);

Απάντηση:

Για να βρούμε το «μέσο χρόνο προσπέλασης μνήμης» (effective memory access time), πρέπει να ζυγίσουμε κάθε περίπτωση με την πιθανότητά της:

$$\text{Effective access time} = 0,80 \times 800 + 0,20 \times 1550 = 950 \text{ ns}$$

Σε αυτό το παράδειγμα υπάρχει μια καθυστέρηση 26,6% στο χρόνο προσπέλασης μνήμης (από τα 750 στα 950ns). Για λόγο επιτυχίας ίσο με 90% έχουμε

$$\text{Effective access time} = 0,90 \times 800 + 0,10 \times 1550 = 875 \text{ ns}$$

Αυτός ο αυξημένος λόγος επιτυχίας παράγει μια καθυστέρηση μόνο 16,6 %.

4.6.4 Κοινές σελίδες

Άλλο ένα πλεονέκτημα της σελιδοποίησης είναι η δυνατότητα χρήσης του ίδιου κώδικα από πολλές διαδικασίες. Υποθέστε ένα σύστημα που υποστηρίζει 40 διαδικασίες, που η καθεμία εκτελεί έναν επεξεργαστή κειμένου. Αν ο επεξεργαστής αποτελείται από 30K κώδικα και 5K χώρο δεδομένων, θα χρειαζόμαστε 1400K για την υποστήριξη των 40 διαδικασιών. Αν, όμως, ο κώδικας είναι «επανεισαγόμενος» (reentrant, που επίσης ονομάζεται «καθαρός κώδικας» – pure code), το οποίο θα πει ότι ο κώδικας αυτός είναι μη αυτο-μεταβαλλόμενος κώδικας, δεν αλλάζει κατά τη διάρκεια της εκτέλεσής του, επομένως πολλές διαδικασίες μπορούν να τον εκτελούν παράλληλα. Κάθε διαδικασία έχει το δικό της αντίγραφο των καταχωρητών και χώρο αποθήκευσης δεδομένων για να κρατάει τα δεδομένα που χρειάζονται στην εκτέλεση (Γιατί συμβαίνει αυτό;).

Επανεισαγόμενος κώδικας θα μπορούσε να διαμοιραστεί όπως φαίνεται Σχήμα 4.25. Εδώ βλέπουμε έναν επεξεργαστή 3 σελίδων (3-page editor) που χρησιμοποιείται από τρεις διαδικασίες. Κάθε διαδικασία έχει τη δική της σελίδα δεδομένων.

Άσκηση Αυτοαξιολόγησης 4.25

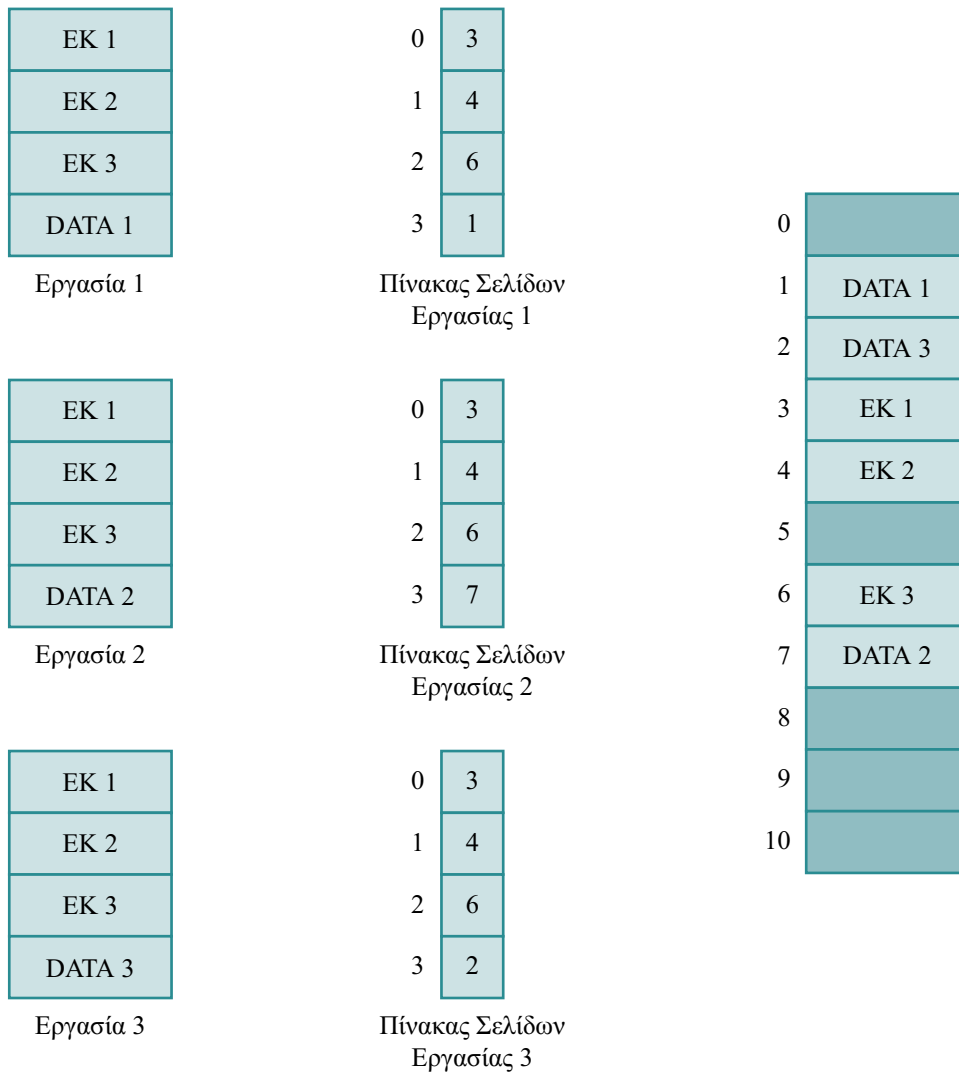
Πόση μνήμη χρειάζονται οι 40 διαδικασίες του επεξεργαστή κειμένου αν μπορούν να χρησιμοποιούν το ίδιο αντίγραφο κώδικα;

Απάντηση:

Μόνο ένα αντίγραφο του επεξεργαστή χρειάζεται να κρατηθεί στη φυσική μνήμη. Ο πίνακας σελίδων κάθε χρήστη αντιστοιχεί στο ίδιο φυσικό αντίγραφο του επεξεργαστή, αλλά οι σελίδες δεδομένων αντιστοιχούν σε διαφορετικά πλαίσια. Έτσι, για την υποστήριξη 40 διαδικασιών χρειαζόμαστε ένα μόνο αντίγραφο του επεξεργαστή (30K), συν 40 αντίγραφα του μεγέθους 5K χώρου δεδομένων. Ο συνολικός χώρος που χρειάζεται γίνεται τώρα 230K, αντί για 1400K – μια σημαντική βελτίωση.

4.6.5 Προστασία

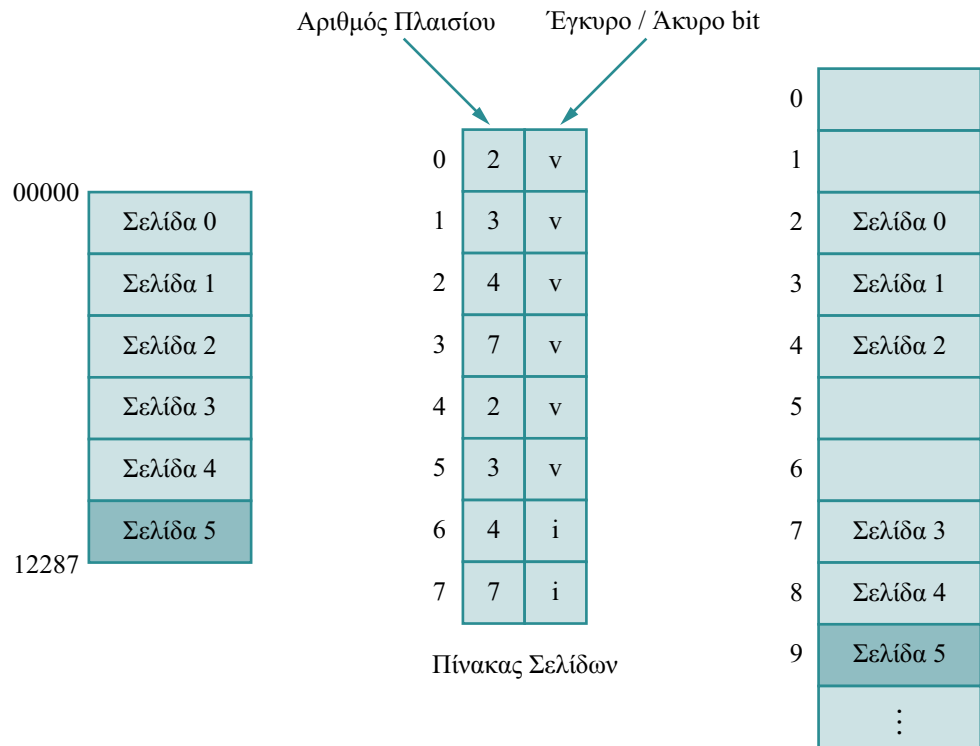
Το ΛΣ πρέπει να εξασφαλίσει ότι κάθε διαδικασία, αφενός, δε θα επιχειρήσει προσπέλαση εκτός του χώρου της και, αφετέρου, ότι δε θα επιχειρήσει να αλλάξει τον κοινόχρηστο κώδικα που εκτελεί μαζί με άλλες διαδικασίες. Αν, λοιπόν, ο μέγιστος χώρος μνήμης διαδικασίας είναι k σελίδες, ο πίνακας σελίδων έχει k εγγραφές, και

**Σχήμα 4.25**

*Μοίρασμα κώδικα
σε ένα περιβάλλον
με σελιδοποίηση*

εκεί, εκτός από την αντιστοίχιση λογικού με φυσικό αριθμό σελίδας, υπάρχει και η πληροφορία τού τι είδους προσπέλαση επιτρέπεται: πλήρης, μόνο για διάβασμα ή καθόλου. Αυτό γίνεται με 2–3 ειδικά bits για κάθε σελίδα, χωρίς επιπλέον χρόνο επεξεργασίας. Οι «παραβάτες» αποστέλλονται στο ΛΣ για τα περαιτέρω (J hardware trap – memory protection violation). Οι παράνομες διευθύνσεις παγιδεύονται με τη χρήση έγκυρου / άκυρου bit. Το ΛΣ θέτει αυτό το bit για κάθε σελίδα, για να επιτρέψει ή να αποτρέψει προσπελάσεις σε αυτή τη σελίδα. Για παράδειγμα, σε ένα σύστημα με χώρο διευθύνσεων των 14 bits (0 έως 16.383) μπορεί να έχουμε μια διαδικασία που πρέπει να χρησιμοποιεί μόνο τις διευθύνσεις 0 έως 10.468. Με δεδομένο μέγεθος σελίδας τα 2K, έχουμε την κατάσταση που φαίνεται στο Σχήμα 4.26.

Οι διευθύνσεις στις σελίδες 0, 1, 2, 3, 4 και 5 αντιστοιχίζονται κανονικά μέσω του πίνακα σελίδων. Όποια όμως προσπάθεια δημιουργίας μιας διεύθυνσης που βρίσκεται στις σελίδες 6 ή 7, βρίσκει ότι το έγκυρο / άκυρο bit έχει τεθεί στην τιμή άκυρο και ο υπολογιστής θα την παγιδέψει προς το ΛΣ (άκυρη αναφορά σελίδας).



Σχήμα 4.26

Έγκυρο (v) ή
άκυρο (i) bit στον
πίνακα σελίδων

Παρατηρήστε ότι, εφόσον η διαδικασία εκτείνεται μόνο μέχρι τη διεύθυνση 10.468, κάθε αναφορά πέρα από αυτή τη διεύθυνση είναι παράνομη. Όμως, αναφορές στη σελίδα 5 θεωρούνται έγκυρες, άρα προσπελάσεις στις διευθύνσεις έως 12.287 είναι έγκυρες. Μόνο οι διευθύνσεις από 12.288 έως 16.383 είναι άκυρες. Αυτό το πρόβλημα είναι αποτέλεσμα του μεγέθους σελίδας (2K) και αντανακλά την εσωτερική κλασματοποίηση της σελιδοποίησης.

Ιστορικό σημείωμα:

Ο XDS-940 πρόσθετε ένα bit σε κάθε σελίδα για να ελέγχει την «ανάγνωση μόνο» ή «ανάγνωση – εγγραφή – προσπέλαση». Ένας αριθμός πλαισίου ίσος με 0 μεταφραζόταν σαν μια άκυρη είσοδος του πίνακα σελίδων. Οι σχεδιαστές σκέφτηκαν ότι το ΛΣ θα κατείχε το πλαίσιο 0 της μνήμης, και συνεπώς κανενός χρήστη ο πίνακας σελίδων δε θα έπρεπε να αντιστοιχίζεται στο πλαίσιο 0.

4.6.6 Δύο θεωρήσεις της μνήμης (Two Views of Memory)

Ένα πολύ σημαντικό στοιχείο της σελιδοποίησης είναι ο καθαρός διαχωρισμός μεταξύ της θεώρησης της μνήμης από τη διαδικασία και της πραγματικής φυσικής μνήμης. Η διαδικασία βλέπει τη μνήμη ως συνεχή χώρο που περιέχει μόνο αυτή τη διαδικασία. Στην πραγματικότητα, η διαδικασία είναι διασκορπισμένη στη φυσική μνήμη, που επίσης περιέχει και άλλες διαδικασίες. Η διαφορά μεταξύ της θεώρησης της μνήμης από τη διαδικασία και της πραγματικής φυσικής μνήμης αντιμετωπίζεται από το υλικό μετάφρασης διευθύνσεων ή αντιστοίχισης. Το υλικό αντιστοίχισης μεταφράζει τις λογικές διευθύνσεις σε φυσικές διευθύνσεις. Αυτή η αντιστοίχιση είναι κρυμμένη από το χρήστη και ελέγχεται από το ΛΣ.

Ένα αποτέλεσμα αυτού του διαχωρισμού είναι ότι τα σύνολα των λογικών και φυσικών διευθύνσεων μπορεί να διαφέρουν. Υπάρχουν περιπτώσεις που η φυσική μνήμη είναι μεγαλύτερη από τη λογική, και αντίστροφα. Η λογική μνήμη σε ένα σύστημα των n bits είναι συνήθως 2^n , 2^{n-1} ή 2^{n-2} . Η φυσική μνήμη έχει ένα μέγιστο 2^n , αλλά συνήθως είναι πολύ μικρότερη. Αν δούμε την ιστορία των υπολογιστικών συστημάτων, όταν μια αρχιτεκτονική πρωτοεμφανίζεται, έχει πολύ περισσότερες δυνατότητες απ' όσες υλοποιεί (άρα λογική μνήμη μεγαλύτερη της φυσικής), ενώ προς το τέλος του κύκλου ζωής της μπορεί η φυσική μνήμη να γίνεται μεγαλύτερη της λογικής.^[11]

Ιστορικό σημείωμα:

Στον XDS – 940, για παράδειγμα, μια λογική διεύθυνση έχει 14 bits, ενώ η φυσική διεύθυνση έχει 16 bits. Ένας αριθμός σελίδας των 3 bits δεικτοδοτείται στον πίνακα σελίδων για να επιλέξει έναν αριθμό πλαισίου των 5 bits. Έτσι, μπορεί να υπάρχει έως και 4 φορές περισσότερη φυσική μνήμη απ' όση οποιαδήποτε μοναδική διαδικασία μπορεί να διευθυνσιοδοτήσει. Ο πολυπρογραμματισμός είναι εύκολος, τουλάχιστον 4 διαδικασίες μπορούν να υπάρχουν στη μνήμη ταυτόχρονα.

Αυτή η τεχνική υιοθετήθηκε ιδιαίτερα από κατασκευαστές μίνι υπολογιστών. Πολλοί mini υπολογιστές σχεδιάστηκαν στο τέλος της δεκαετίας 1960–70, όταν η μνήμη ήταν ακριβή και τα προγράμματα έπρεπε να είναι μικρά. Έτσι, οι περισσότερες διευθύνσεις ήταν περιορισμένες στα 15 ή 16 bits. Με τη διάθεση φτηνότερης μνήμης ημιαγωγών έγινε προσοδοφόρα η πρόσθεση περισσότερης φυσικής μνήμης σε αυτούς τους mini υπολογιστές. Αλλά η αύξηση του μεγέθους της διεύθυνσης, έτσι ώστε να επιτραπούν οι μεγαλύτερες των 17 ή 18 bits διευθύνσεις που χρειαζόνταν για την αυξημένη φυσική μνήμη, σήμαινε ότι έπρεπε να ξανασχεδια-

[11] Αυτό συμβαίνει γιατί οι τεχνικοοικονομικές εξελίξεις στην αρχιτεκτονική των υπολογιστών προχωρούν με μεγαλύτερα κβάντα απ' ό,τι οι τεχνικοοικονομικές εξελίξεις στον τομέα της μνήμης, οι οποίες έχουν συνήθως κβάντο 2.

στεί το σύνολο των εντολών ή να επεκταθεί το μέγεθος της λέξης για να εξυπηρετηθούν τα παραπάνω bits. Και οι δύο λύσεις απαιτούσαν μια μεγάλη αλλαγή, που θα ακύρωνε όλα τα υπάρχοντα προγράμματα και τεχνικά εγχειρίδια. Η λύση που οι περισσότεροι κατασκευαστές ακολούθησαν ήταν η αντιστοίχιση μνήμης. Οι λογικές διευθύνσεις (15 ή 16 bits) αντιστοιχίζονται σε μεγαλύτερες (17 ή 18 bits) φυσικές διευθύνσεις. Με τον πολυπρογραμματισμό του συστήματος όλη η μνήμη μπορεί να χρησιμοποιηθεί. Οι χρήστες όμως δεν μπορούν να χρησιμοποιήσουν πιο πολύ μνήμη από πριν, εφόσον ο χώρος λογικών διευθύνσεων δεν έχει αυξηθεί.

Για παράδειγμα, ο Nova 3/D αντιστοιχίζει έναν αριθμό σελίδας των 5 bits σε έναν αριθμό πλαισίου των 7 bits. Ο HP 2100 κάνει το ίδιο. Ο DEC-10 αντιστοιχίζει έναν αριθμό σελίδας των 9 bits σε έναν αριθμό πλαισίου των 13 bits, αλλάζοντας μια λογική διεύθυνση των 18 bits σε μια φυσική διεύθυνση των 22 bits.

Το ΛΣ ελέγχει αυτή την αντιστοίχιση και μπορεί να την ενεργοποιήσει για το χρήστη και να την απενεργοποιήσει για το ΛΣ. Επειδή το ΛΣ διαχειρίζεται φυσική μνήμη, πρέπει να είναι ενήμερο της φύσης της φυσικής μνήμης: ποια πλαίσια κατανέμονται, ποια πλαίσια είναι διαθέσιμα, πόσα συνολικά πλαίσια υπάρχουν κτλ. Αυτή η πληροφορία, γενικά, κρατιέται σε μια δομή δεδομένων, που λέγεται «πίνακας πλαισίων». Ο πίνακας πλαισίων έχει μια εγγραφή για κάθε φυσικό πλαίσιο σελίδας, που δείχνει αν το πλαίσιο είναι ελεύθερο ή έχει διανεμηθεί και, αν έχει δοθεί, σε ποια σελίδα ποιας διαδικασίας.

Επιπρόσθετα, το ΛΣ πρέπει να ξέρει ότι οι διαδικασίες λειτουργούν στο χώρο των διαδικασιών και ότι όλες οι λογικές διευθύνσεις πρέπει να αντιστοιχιστούν για να παραγάγουν φυσικές διευθύνσεις. Το ΛΣ διατηρεί ένα αντίγραφο του πίνακα σελίδων για κάθε διαδικασία, καθώς και ένα αντίγραφο του μετρητή προγράμματος και του περιεχομένου των καταχωρητών. Αυτό το αντίγραφο (του πίνακα σελίδων) χρησιμοποιείται για τη μετάφραση των λογικών διευθύνσεων σε φυσικές, οπότε το ΛΣ πρέπει να αντιστοιχίσει τη λογική διεύθυνση σε φυσική. Επίσης, χρησιμοποιείται από το διεκπεραιωτή για τον ορισμό του υλικού πίνακα σελίδων όταν πρόκειται να δοθεί η CPU σε μια διεύθυνση.

Άσκηση Αυτοαξιολόγησης 4.27

Μια μηχανή έχει διάστημα διευθύνσεων των 32 bits και σελίδες μεγέθους 8K. Ο πίνακας σελίδων βρίσκεται εξολοκλήρου στο υλικό, με μια λέξη των 32 bits για κάθε καταχώρισή του. Όταν ξεκινάει μια διαδικασία, ο πίνακας σελίδων αντιγράφεται από τη μνήμη στο υλικό, με ρυθμό μια λέξη κάθε 100 nsec. Αν κάθε διαδικασία εκτελείται για 100 msec (συμπεριλαμβανομένου και του χρόνου για τη φόρτωση του πίνακα σελίδων), ποιο είναι το ποσοστό του χρόνου της CPU που αφιε-

ρώνεται στη φόρτωση του πίνακα σελίδων;

Απάντηση:

Για να μπορέσουμε να προσδιορίσουμε το μέγεθος του πίνακα σελίδων, πρέπει πρώτα να υπολογίσουμε πόσα bits χρησιμοποιούνται για μετατόπιση (αφού ξέρουμε το μέγεθος της σελίδας). Άρα, αφού η σελίδα έχει μέγεθος 8K, αυτό σημαίνει ότι $8K = 2^3 * 2^{10}$ και ότι χρειαζόμαστε 13 bits για τη μετατόπιση. Ξέρουμε επίσης από τα δεδομένα της άσκησης ότι το διάστημα των διευθύνσεων είναι 32 bits. Το μέγεθος του πίνακα σελίδων είναι $32 - 13 = 19$ bits. Άρα ο πίνακας σελίδων έχει 524.288 εγγραφές (λέξεις).

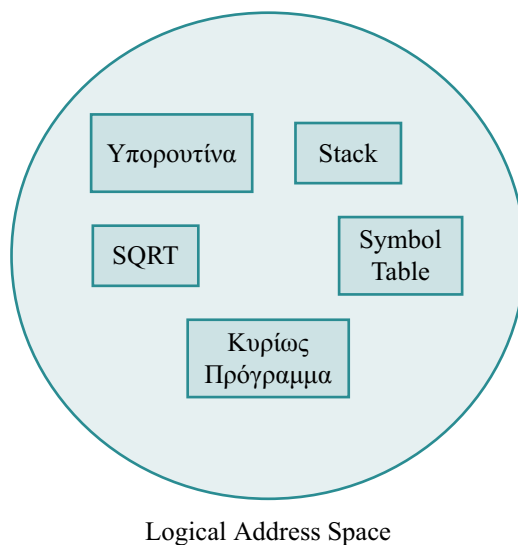
Για την αντιγραφή του πίνακα σελίδων από τη μνήμη στο υλικό χρειάζεται χρόνος ίσος με $(524288 * 100 \text{ nsec}) = 52428800 \text{ nsec} = 52.428800 \text{ msec}$.

Στα 100 msec εκτέλεσης μιας εντολής τα 52.4288 msec χρησιμοποιούνται για την φόρτωση του πίνακα. Δηλαδή το ποσοστό του χρόνου της CPU που αφιερώνεται στην φόρτωση του πίνακα σελίδων είναι 52%.

4.7 Τμηματοποίηση (Segmentation)

4.7.1 Η Θεώρηση της μνήμης από το χρήστη

Ο προγραμματιστής ενός συστήματος θεωρεί τη μνήμη σαν μια συλλογή τμημάτων μεταβλητού μεγέθους, όχι αναγκαστικά σε κάποια σειρά (Σχήμα 4.27).



Σχήμα 4.27

Θεώρηση ενός προγράμματος από το χρήστη-προγραμματιστή

Σκεφτείτε πώς βλέπετε ένα πρόγραμμα όταν το γράφετε. Το θεωρείτε σαν ένα κυρίως πρόγραμμα με ένα σύνολο από υπορουτίνες, διαδικασίες, συναρτήσεις ή ενότητες. Μπορεί, επίσης, να υπάρχουν διάφορες δομές δεδομένων: πίνακες, διανύσματα, στοίβες (stacks), μεταβλητές κτλ. Καθεμία από αυτές τις ενότητες ή από αυτά τα στοιχεία δεδομένων αναφέρονται με κάποιο όνομα. Μιλάτε για τον «πίνακα συμβόλων» (symbol table), για τη «συνάρτηση Sqrt», για το «κυρίως πρόγραμμα», χωρίς να νοιάζεστε ποιες διευθύνσεις στη μνήμη κατέχουν αυτά τα στοιχεία. Δε σας ενδιαφέρει αν ο πίνακας συμβόλων είναι αποθηκευμένος πριν ή μετά τη συνάρτηση Sqrt. Καθένα από αυτά τα τμήματα είναι μεταβλητού μεγέθους, το μέγεθος ορίζεται από το σκοπό του τμήματος στο πρόγραμμα. Στοιχεία μέσα σε ένα τμήμα αναγνωρίζονται από τη μετατόπισή τους (offset) από την αρχή του τμήματος: η πρώτη εντολή του προγράμματος, η 17η εγγραφή ή στον πίνακα συμβόλων, η πέμπτη εντολή της συνάρτησης Sqrt κτλ.

Η τμηματοποίηση είναι ένα σχήμα διαχείρισης της μνήμης που υποστηρίζει αυτή τη θεώρηση της μνήμης από το χρήστη. Ο χώρος λογικών διευθύνσεων είναι μια συλλογή από τμήματα. Κάθε τμήμα έχει όνομα και μήκος. Οι διευθύνσεις καθορίζουν και το όνομα του τμήματος και τη μετατόπιση μέσα στο τμήμα. Συνεπώς, η διαδικασία ορίζει κάθε διεύθυνση με δύο ποσότητες: ένα όνομα τμήματος και μια μετατόπιση. (Αντιπαραθέστε αυτό τον τρόπο με τη σελιδοποίηση, όπου η διαδικασία καθόριζε μόνο μία διεύθυνση, που διαχωριζόταν από το υλικό, σε αριθμό σελίδας και μετατόπιση, και όλη η λειτουργία ήταν αόρατη για τον προγραμματιστή.)

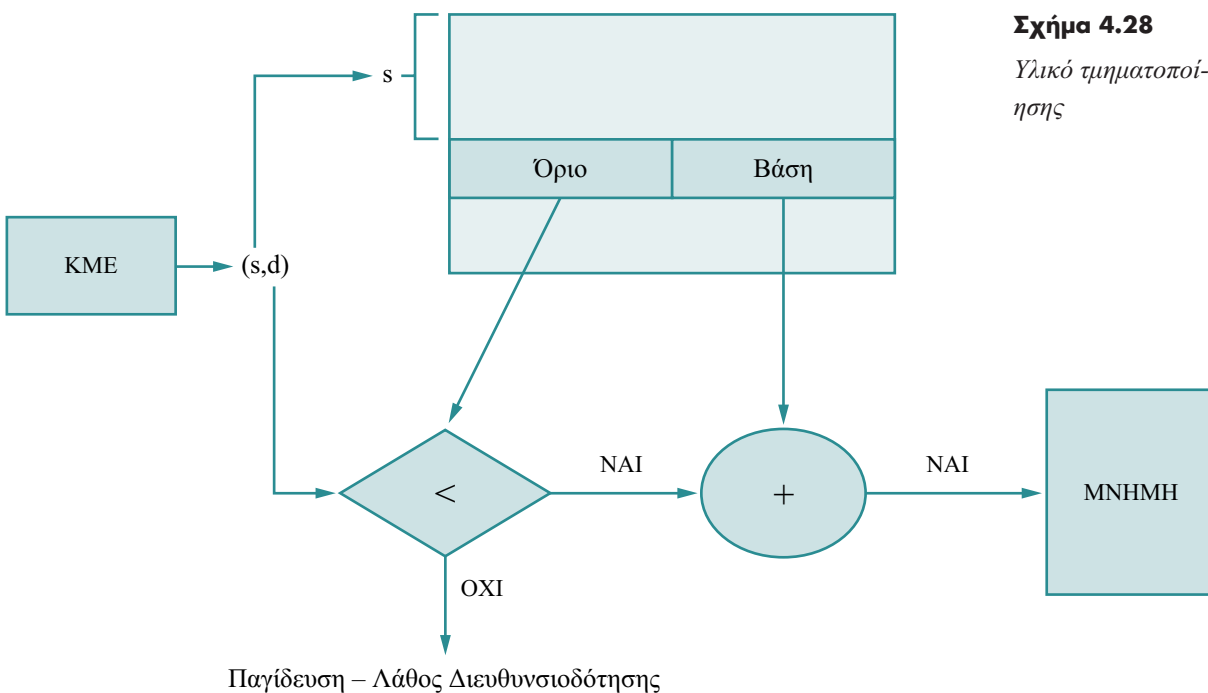
Για λόγους απλοποίησης της υλοποίησης, τα τμήματα είναι αριθμημένα και οι αναφορές σε αυτά γίνονται με τον αριθμό τμήματος αντί με το όνομα τμήματος. Κανονικά, η διαδικασία (αρχικά πηγαίο πρόγραμμα) μεταφράζεται (ή συμβολομεταφράζεται) και ο μεταφραστής (ή συμβολομεταφραστής) κατασκευάζει, αυτόματα, τμήματα που αντι-κατοπτρίζουν το πρόγραμμα εισόδου. Ένας συμβολομεταφραστής της Java, C++, visual basic, Pascal μπορεί να δημιουργήσει ξεχωριστά τμήματα για: (1) τις καθολικές (global) μεταβλητές, (2) τη στοίβα κλήσεων των διαδικασιών (procedure call stack), για την αποθήκευση παραμέτρων και των διευθύνσεων επιστροφής, (3) το κομμάτι κώδικα της κάθε διαδικασίας ή συνάρτησης και (4) τις τοπικές μεταβλητές κάθε διαδικασίας και συνάρτησης. Ένας συμβολομεταφραστής της Fortran μπορεί να δημιουργήσει ένα ξεχωριστό τμήμα για κάθε «κοινή ενότητα» (common block). Στα «διανύσματα» (arrays) μπορεί να δοθούν ξεχωριστά τμήματα. Ο «φορτωτής» (loader) θα πάρει όλα αυτά τα τμήματα και θα τους δώσει αριθμούς τμημάτων.

4.7.2 Υλικό

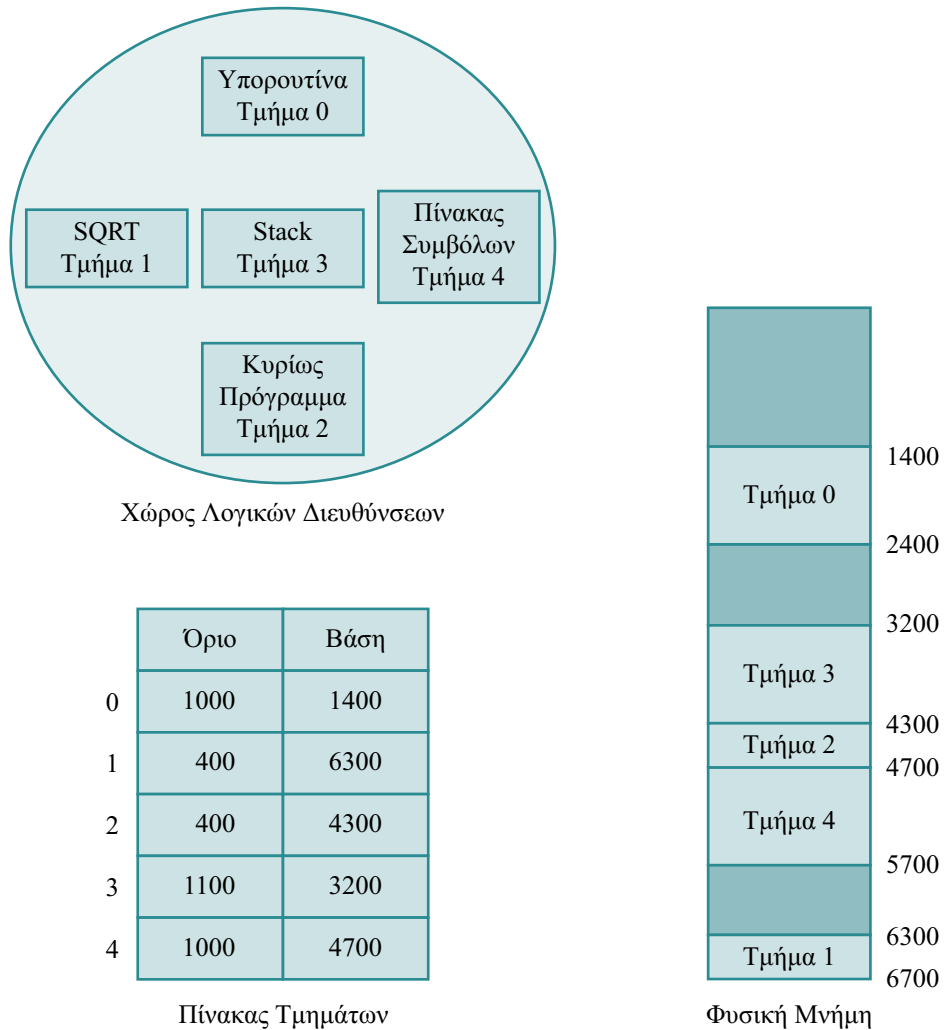
Παρ' όλο που ο χρήστης μπορεί τώρα να αναφερθεί σε αντικείμενα μέσα στη διαδικασία με μία δύο διαστάσεων διεύθυνση, η πραγματική φυσική μνήμη παραμένει,

φυσικά, μία, μονοδιάστατη ακολουθία λέξεων. Έτσι, πρέπει να ορίσουμε μια υλοποίηση που να αντιστοιχίζει τις δισδιάστατες, ορισμένες από το χρήστη, διευθύνσεις σε μονοδιάστατες φυσικές διευθύνσεις. Αυτή η αντιστοίχιση επιτυγχάνεται από τον «πίνακα τμημάτων» (segment table).

Η χρήση του πίνακα τμημάτων απεικονίζεται στο Σχήμα 4.28. Μια λογική διεύθυνση αποτελείται από δύο μέρη: έναν αριθμό τμήματος, s , και μια μετατόπιση μέσα σε αυτό το τμήμα, d . Ο αριθμός τμήματος χρησιμοποιείται ως δείκτης σε πίνακα τμημάτων. Κάθε εγγραφή του πίνακα σελίδων περιέχει τη «βάση τμήματος» (segment base) και το «όριο τμήματος» (segment limit). Η μετατόπιση, d , της λογικής διεύθυνσης πρέπει να είναι μεταξύ 0 και του ορίου τμήματος. Αν δεν είναι, προκαλείται κλήση προς το ΛΣ (προσπάθεια λογικής διευθυνσιοδότησης πέρα από το τέλος του τμήματος). Αν η μετατόπιση είναι νόμιμη, προστίθεται στη βάση τμήματος για να παραγάγει τη διεύθυνση φυσικής μνήμης που περιέχει την επιθυμητή λέξη. Ο πίνακας τμημάτων είναι, συνεπώς, ένα διάγραμμα από ζευγάρια καταχωρητών βάσης / ορίου.



Ως παράδειγμα πάρτε την κατάσταση που φαίνεται στο Σχήμα 4.29. Έχουμε 5 τμήματα αριθμημένα από το 0 έως το 4, αποθηκευμένα στη φυσική μνήμη. Ο πίνακας τμημάτων έχει μια ξεχωριστή εγγραφή για κάθε τμήμα, η οποία δίνει την αρχική διεύθυνση του τμήματος στη φυσική μνήμη (η βάση) και το μήκος του τμήματος (το όριο).

**Σχήμα 4.29**

Παράδειγμα τμη-
ματοποίησης

Άσκηση Αυτοαξιολόγησης 4.28

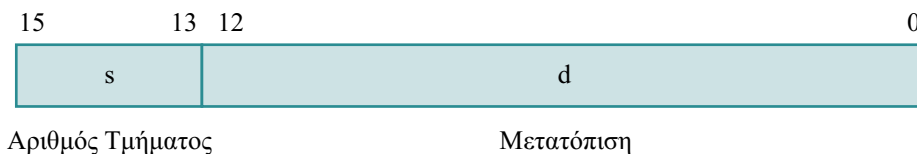
Σε ποια φυσική διεύθυνση αντιστοιχούν οι λογικές διευθύνσεις (τμήμα 2, λέξη 53), (τμήμα 3, λέξη 852), (τμήμα 0, λέξη 1222) του Σχήματος 4.29;

Απάντηση:

Το τμήμα 2 έχει μήκος 400 λέξεις, ξεκινώντας από τη θέση 4300. Έτσι, μια αναφορά στη λέξη 53 του τμήματος 2 αντιστοιχεί στη θέση $4300 + 53 = 4353$. Μια αναφορά στο τμήμα 3, λέξη 852 αντιστοιχεί στη θέση 3200 (η βάση του τμήματος 3) $+ 852 = 4052$. Μια αναφορά στη λέξη 1222 του τμήματος 0 θα είχε ως αποτέλεσμα παγίδευση προς το ΛΣ, αφού αυτό το τμήμα είναι μόνο 1000 λέξεις μακρύ.

4.7.3 Υλοποίηση των πινάκων τμημάτων (Implementation of Segment Tables)

Όπως ο πίνακας σελίδων, έτσι και ο πίνακας τμημάτων μπορεί να τοποθετηθεί είτε σε γρήγορους καταχωρητές είτε στη μνήμη. Ένας πίνακας τμημάτων που κρατιέται σε καταχωρητές μπορεί να προσπελαστεί πολύ γρήγορα, η πρόσθεση στη βάση και η σύγκριση με το όριο μπορούν να γίνουν ταυτόχρονα, κερδίζοντας έτσι χρόνο. Ιστορικά, ο PDP-11/45 χρησιμοποιούσε αυτή την προσέγγιση έχοντας οκτώ καταχωρητές τμημάτων. Μια διεύθυνση των 16 bits σχηματίζεται από έναν αριθμό τμήματος των 3 bits και μια μετατόπιση τμήματος των 13 bits. Αυτός ο σχεδιασμός επιτρέπει μέχρι οκτώ τμήματα, και κάθε τμήμα μπορεί να είναι μέχρι 8 bytes (Σχήμα 4.30). Κάθε εγγραφή του πίνακα τμημάτων περιέχει τη διεύθυνση βάσης, το μήκος και ένα σύνολο από bits ελέγχου προσπέλασης, που καθορίζουν τα εξής: «μη προσπέλαση» (no-access), «ανάγνωση μόνο», «ανάγνωση – εγγραφή».



Σχήμα 4.30

*Τμηματοποιημένη
διευθυνσιοδότηση
για το PDP-11/45*

Ο Burroughs B5500 επέτρεπε 32 τμήματα, μέχρι 1024 λέξεις το καθένα. Αυτός ο σχεδιασμός όριζε έναν αριθμό τμήματος των 5 bits και μια μετατόπιση των 10 bits. Όμως, η χρήση αυτού του συστήματος έδειξε ότι υπήρχαν πολύ λίγα τμήματα και ότι το όριο μεγέθους του τμήματος ήταν πολύ μικρό (διανύσματα πάνω από 1K έπρεπε να διασπαστούν σε αρκετά τμήματα). Έτσι, ο GE 645, που χρησιμοποιήθηκε για το Multics, επιτρέπει μέχρι 256K τμήματα, που είναι έως 64K λέξεις.

Με τόσα πολλά τμήματα, δεν είναι αποδοτικό να κρατιέται ο πίνακας τμημάτων σε καταχωρητές, συνεπώς πρέπει να κρατηθεί στη μνήμη. Ένας «Καταχωρητής Βάσης Πίνακα Τμημάτων» (STBR – Segment Table Base Register) δείχνει τον πίνακα τμημάτων. Επίσης, επειδή ο αριθμός των τμημάτων που χρησιμοποιούνται από μια διαδικασία μπορεί να μεταβάλλεται με μεγάλο εύρος, χρησιμοποιείται ένας «Καταχωρητής Μήκους Πίνακα Τμημάτων» (STLR – Segment Table Length Register). Σε μια λογική διεύθυνση (s, d) πρώτα ελέγχουμε ότι ο αριθμός τμήματος, s, είναι νόμιμος ($s < \text{STLR}$). Μετά προσθέτουμε τον αριθμό τμήματος στον STBR, με αποτέλεσμα τη διεύθυνση ($\text{STBR} + s$) στη μνήμη εγγραφής του πίνακα τμημάτων. Αυτή η εγγραφή διαβάζεται από τη μνήμη και προχωράμε όπως και πριν: ελέγχεται η μετατόπιση έναντι του μήκους τμήματος και υπολογίζεται η φυσική διεύθυνση της επι-

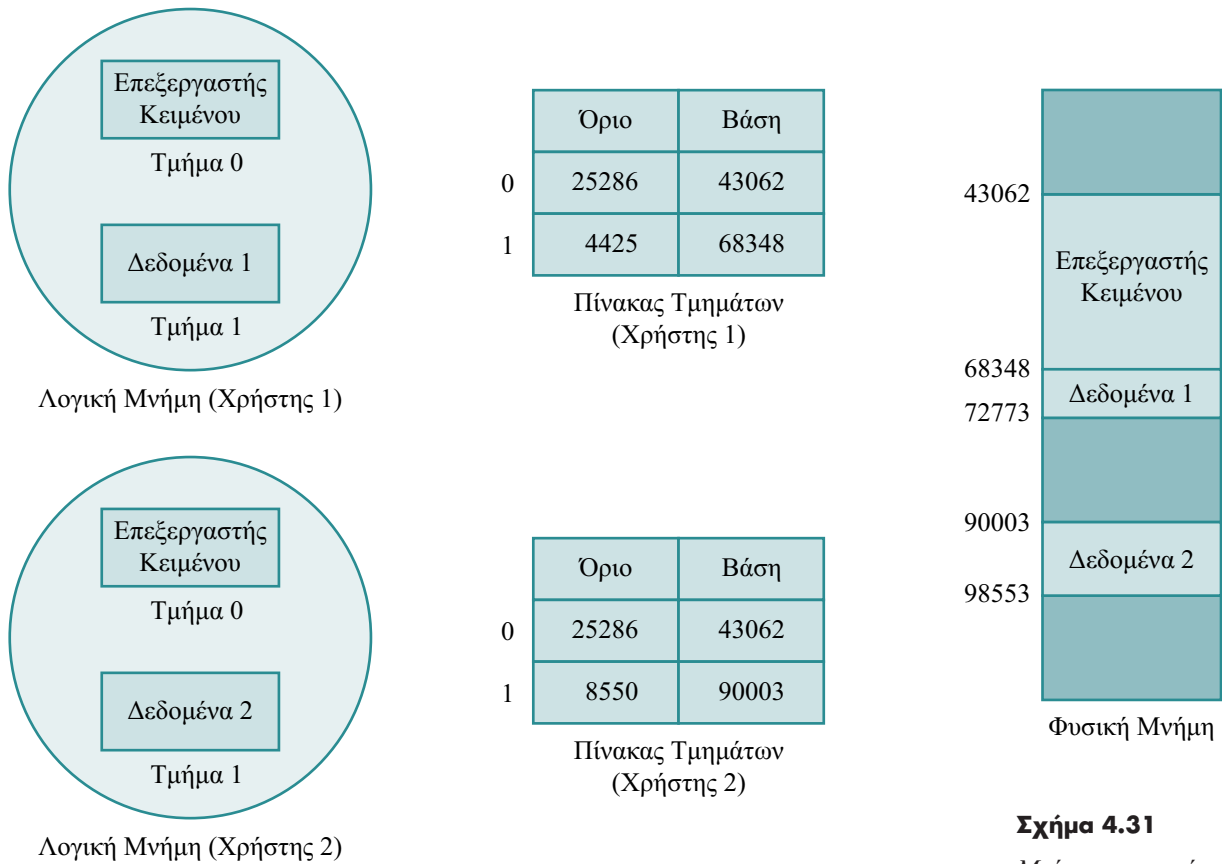
θυμητής λέξης ως το άθροισμα της βάσης τμήματος και της μετατόπισης.

Όπως και με τη σελιδοποίηση, αυτή η αντιστοίχιση απαιτεί δύο αναφορές μνήμης για κάθε λογική διεύθυνση, υποδιπλασιάζοντας – μειώνοντας την ταχύτητα του υπολογιστικού συστήματος, εκτός κι αν παρθούν μέτρα. Η κανονική λύση είναι να χρησιμοποιηθεί ένα σύνολο συσχετιστικών καταχωρητών για να κρατούν τις πιο πρόσφατα χρησιμοποιημένες εγγραφές του πίνακα τμημάτων. Ένα σχετικά μικρό σύνολο συσχετιστικών καταχωρητών (8 ή 16) μπορούν συνήθως να μειώσουν την καθυστέρηση των προσπελάσεων μνήμης σε μεγάλο βαθμό.

4.7.4 Προστασία και κοινή χρήση

Ένα ιδιαίτερο πλεονέκτημα της τμηματοποίησης είναι ο συσχετισμός της προστασίας με τα τμήματα. Εφόσον τα τμήματα αντιπροσωπεύουν ένα εννοιολογικά ορισμένο κομμάτι της διαδικασίας, είναι πιθανό ότι όλες οι εγγραφές στο τμήμα χρησιμοποιούνται με τον ίδιο τρόπο. Έτσι, έχουμε ορισμένα τμήματα που περιέχουν μόνο εντολές, ενώ άλλα έχουν μόνο δεδομένα. Σε μια σύγχρονη αρχιτεκτονική, οι εντολές είναι μη αυτο-μεταβαλλόμενες, έτσι τα τμήματα που τις περιέχουν μπορούν να οριστούν ως «ανάγνωσης μόνο» ή «εκτέλεσης μόνο». Το «υλικό αντιστοίχισης μνήμης» (memory-mapping hardware) θα ελέγξει τα bits προστασίας που σχετίζονται με κάθε εγγραφή του πίνακα τμημάτων, για να αποτρέψει παράνομες προσπελάσεις στη μνήμη, όπως προσπάθειες για την εγγραφή σε τμήμα «ανάγνωσης μόνο» ή χρήσης ενός τμήματος «εκτέλεσης μόνο» ως δεδομένα. Με την τοποθέτηση ενός πίνακα στο δικό του τμήμα, το υλικό διαχείρισης μνήμης, αυτόματα, θα ελέγξει ότι οι δείκτες του πίνακα είναι έγκυροι και δεν εκτείνονται πέρα από τα όριά του. Έτσι, πολλά κοινά προγραμματιστικά λάθη θα ανιχνευτούν από το υλικό πριν επιφέρουν σοβαρή ζημιά.

Άλλο ένα πλεονέκτημα της τμηματοποίησης είναι ο κοινόχρηστος κώδικας και δεδομένα. Κάθε διαδικασία έχει έναν πίνακα τμημάτων συσχετισμένο με το τμήμα ελέγχου διαδικασίας, τον οποίο ο διεκπεραιωτής χρησιμοποιεί για να ορίσει τον «πίνακα τμημάτων του υλικού» (hardware segment table) όταν η CPU δοθεί σε αυτή τη διαδικασία. Τα τμήματα διαμοιράζονται όταν οι εγγραφές, στον πίνακα τμημάτων, δύο διαφορετικών διαδικασιών δείχνουν τις ίδιες θέσεις (Σχήμα 4.31).

**Σχήμα 4.31**

Μοίρασμα τμημάτων σε ένα σύστημα τμηματοποιημένης μνήμης

Το μοίρασμα συμβαίνει στο επίπεδο τμήματος. Έτσι, οποιαδήποτε πληροφορία μπορεί να χρησιμοποιηθεί από πολλούς αν οριστεί ως τμήμα. Αρκετά τμήματα μπορούν να είναι κοινά, έτσι μια διαδικασία που συντίθεται από διάφορα τμήματα μπορεί να είναι κοινόχρηστη.

Άσκηση Αυτοαξιολόγησης 4.29

Σκεφτείτε τη χρήση ενός επεξεργαστή κειμένου σε ένα σύστημα διαμοιρασμένου χρόνου. Ένας πλήρης επεξεργαστής μπορεί να είναι πολύ μεγάλος και να αποτελείται από πολλά τμήματα. Αυτά τα τμήματα μπορούν να διαμοιραστούν μεταξύ όλων των διαδικασιών, περιορίζοντας τη φυσική μνήμη που χρειάζεται για να υποστηριχτούν διαδικασίες επεξεργασίας κειμένου. Σχολιάστε πως θα μπορούσε να αντιμετωπιστεί αυτό το πρόβλημα.

Είναι επίσης δυνατόν να διαμοιράζονται μόνο μέρη διαδικασιών. Για παράδειγμα,

πακέτα κοινών υπορουτίνων μπορούν να μοιράζονται μεταξύ πολλών διαδικασιών, με το να τα ορίζουμε ως διαμοιρασμένα, «ανάγνωσης μόνο», τμήματα. Δύο προγράμματα σε Fortran, για παράδειγμα, μπορεί να χρησιμοποιούν την ίδια υπορουτίνα Sqrt, αλλά μόνο ένα φυσικό αντίγραφο της Sqrt θα χρειαζόταν.

Αν και αυτό το μοίρασμα φαίνεται απλό, υπάρχουν ορισμένες λεπτές παρατηρήσεις. Τα τμήματα κώδικα, τυπικά, περιέχουν αναφορές στους εαυτούς τους. Για παράδειγμα, μία «διακλάδωση υπό συνθήκη» (conditional jump) έχει μία «διεύθυνση μεταφοράς» (transfer address). Αυτή η διεύθυνση μεταφοράς είναι ένας αριθμός τμήματος και μία μετατόπιση. Ο αριθμός τμήματος της διεύθυνσης μεταφοράς θα είναι ο αριθμός τμήματος του τμήματος κώδικα. Αν προσπαθήσουμε να διαμοιράσουμε αυτό το τμήμα, όλες οι διαδικασίες που το μοιράζονται πρέπει να ορίσουν το διαμοιρασμένο τμήμα κώδικα να έχει τον ίδιο αριθμό τμήματος.

Για παράδειγμα, αν θέλουμε να διαμοιράσουμε τη ρουτίνα Sqrt και μια διαδικασία θέλει να την κάνει τμήμα 4 και μια άλλη θέλει να την κάνει τμήμα 17, πώς θα έπρεπε να αναφέρεται στον εαυτό της η ρουτίνα Sqrt; Εφόσον υπάρχει μόνο ένα φυσικό αντίγραφο της Sqrt, αυτή πρέπει να αναφέρεται στον εαυτό της με τον ίδιο τρόπο και για τις δύο διαδικασίες: πρέπει να έχει ένα μοναδικό αριθμό τμήματος. Καθώς ο αριθμός των διαδικασιών που διαμοιράζονται το τμήμα αυξάνεται, αυξάνει και η δυσκολία εύρεσης ενός αποδεκτού αριθμού τμήματος.

Άσκηση Αυτοαξιολόγησης 4.30

Πώς θα μπορούσε να λυθεί αυτό το πρόβλημα με έμμεση αναφορά ενός κώδικα στον εαυτό του;

Τμήματα δεδομένων «ανάγνωσης μόνο» (χωρίς δείκτες) μπορούν να διαμοιραστούν ως διαφορετικοί αριθμοί τμημάτων, όπως και τμήματα κώδικα που δεν αναφέρονται στον εαυτό τους άμεσα αλλά έμμεσα. Για παράδειγμα, διακλαδώσεις υπό συνθήκη που ορίζουν τη διεύθυνση διακλάδωσης ως μια μετατόπιση από την παρούσα τιμή του μετρητή προγράμματος, ή σχετική με κάποιο καταχωρητή που περιέχει τον τρέχοντα αριθμό τμήματος, θα επέτρεπαν στον κώδικα να αποφύγει «άμεση αναφορά» (direct reference) στον τρέχοντα αριθμό τμήματος.

Ο υπολογιστής GE 645, που χρησιμοποιήθηκε για το Multics, έχει 4 καταχωρητές, που περιέχουν τους αριθμούς τμήματος για το τρέχον τμήμα, το τμήμα στοίβας (stack

segment), το τμήμα σύνδεσης (linkage segment) και ένα τμήμα δεδομένων. Οι διαδικασίες σπάνια αναφέρονται άμεσα σε έναν αριθμό τμήματος, αλλά πάντοτε έμμεσα, μέσω αυτών των τεσσάρων καταχωρητών. Αυτό επιτρέπει στον κώδικα να διαμοιράζεται ελεύθερα.

4.7.5 Κλασματοποίηση (Χρονοδρομολόγηση διαδικασιών)

Ο χρονοδρομολογητής διαδικασιών πρέπει να βρει και να κατανείμει μνήμη για όλα τα τμήματα μιας διαδικασίας. Αυτή η κατάσταση είναι παρόμοια με τη σελιδοποίηση, εκτός του ότι τα τμήματα είναι μεταβλητού μεγέθους, ενώ οι σελίδες είναι όλες του ίδιου μεγέθους. Έτσι, όπως και με το σύστημα μεταβλητών διαιρέσεων, η κατανομή μνήμης είναι ένα πρόβλημα «δυναμικής κατανομής χώρου αποθήκευσης» (dynamic storage allocation), που βρίσκει λύση με τη χρήση ενός αλγόριθμου «καλύτερου ταιριάσματος» ή «πρώτου ταιριάσματος».

Η τμηματοποίηση μπορεί τότε να δημιουργήσει εξωτερική κλασματοποίηση, όταν δεν υπάρχει κομμάτι ελεύθερης μνήμης αρκετά μεγάλο για να χωρέσει ένα τμήμα. Σε αυτή την περίπτωση, η διαδικασία μπορεί να πρέπει να περιμένει να ελευθερωθεί περισσότερη μνήμη (μεγαλύτερες οπές), ή μπορεί να χρησιμοποιηθεί η συμπίεση για να δημιουργήσει μεγαλύτερες οπές. Επειδή η τμηματοποίηση είναι από τη φύση της ένας αλγόριθμος δυναμικής μετατόπισης, μπορούμε να συμπιέσουμε τη μνήμη όποτε θέλουμε. Αν ο χρονοδρομολογητής της CPU πρέπει να περιμένει για μια διαδικασία, λόγω προβλήματος στην κατανομή μνήμης, μπορεί να (ή μπορεί να μη) διατρέξει την ουρά της CPU ψάχνοντας για μια μικρότερη, χαμηλότερης προτεραιότητας, διαδικασία για να την εκτελέσει.

Πόσο άσχημη είναι η εξωτερική κλασματοποίηση για ένα σχήμα τμηματοποίησης; Θα βοηθούσε χρονοδρομολόγηση διαδικασιών με συμπίεση ή υπερπήδηση; Οι απαντήσεις σε αυτές τις ερωτήσεις εξαρτώνται κυρίως από το μέσο μέγεθος τμήματος και τον τρόπο με τον οποίο έχει κατανεμηθεί η μνήμη. Στο ένα άκρο θα μπορούσαμε να ορίσουμε κάθε διαδικασία να είναι ένα τμήμα.

Άσκηση Αυτοαξιολόγησης 4.31

Τι σχέση έχει το σύστημα μεταβλητών διαιρέσεων με αυτό το σχήμα;

Απάντηση:

Είναι το σύστημα μεταβλητών διαιρέσεων.

Στο άλλο άκρο, κάθε λέξη θα μπορούσε να μπει στο δικό της τμήμα και να μετατο-

πιστεί ξεχωριστά. Αυτός ο σχεδιασμός εξαφανίζει την εξωτερική κλασματοποίηση. Αν το μέσο μέγεθος τμήματος είναι μικρό, η εξωτερική κλασματοποίηση θα είναι μικρή. (Ως αναλογία, σκεφτείτε να βάλετε βαλίτσες στο πορτμπαγκάζ ενός αυτοκινήτου, τότε δε χωράνε όλες. Αν, όμως, ανοίξετε τις βαλίτσες και βάλετε κάθε πράγμα μόνο του, τότε όλα χωράνε.) Επειδή τα ξεχωριστά τμήματα είναι μικρότερα της συνολικής διαδικασίας, είναι πιο πιθανό να χωρούν στα διαθέσιμα κομμάτια μνήμης.

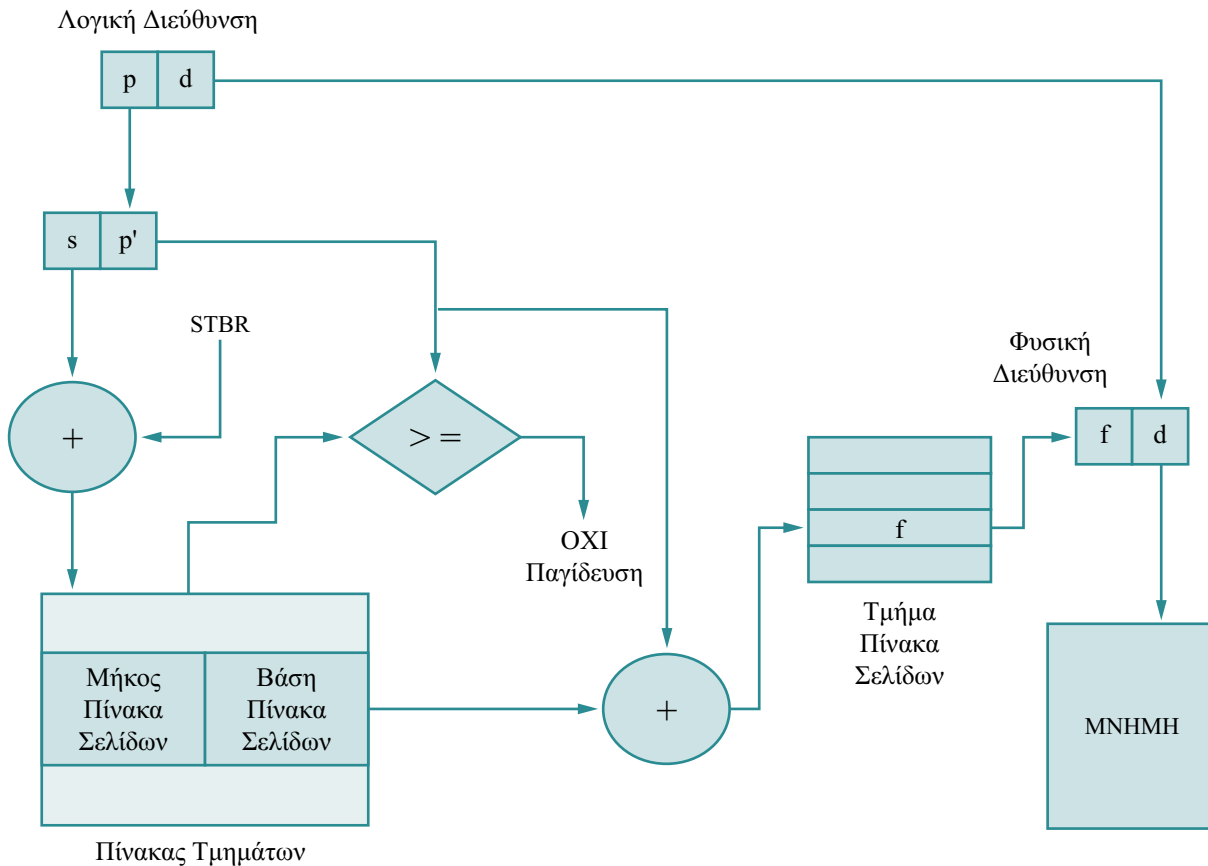
4.8 Συνδυασμένα συστήματα (Combined Systems)

Η σελιδοποίηση και η τμηματοποίηση έχουν και οι δύο πλεονεκτήματα και μειονεκτήματα. Είναι όμως δυνατόν να συνδυαστούν επωφελώς. Αυτοί οι συνδυασμοί απεικονίζονται καλύτερα από δύο συγκεκριμένα συστήματα: το IBM 360/67 (και αργότερα το IBM 370) και το GE 645 για το Multics.

4.8.1 Τμηματοποιημένη σελιδοποίηση (Segmented Paging)

Το IBM 370/67 ήταν ένα από τα πρώτα σελιδοποιημένα συστήματα. Η βασική των 24 bits (1 byte) διεύθυνση διαιρούνταν σε ένα 12 bits αριθμό σελίδας και μια απόσταση (offset) των 12 bits, και ένα έγκυρο / άκυρο bit. Το πρόβλημα ήταν ότι ένας αριθμός σελίδας των 12 bits επέτρεπε 4096 σελίδες και απαιτούσε 8K για κάθε πίνακα σελίδων. Πρόσθετα, ήταν επιθυμητό να επεκταθεί η λογική διεύθυνση στα 32 bits με έναν αριθμό σελίδας των 20 bits. Αυτό θα απαιτούσε έναν πίνακα σελίδων με 1.048.576 εγγραφές ή 2.097.152 bytes.

Ιδιαίτερα με το μεγαλύτερο χώρο διευθύνσεων, το πιο πολύ μέρος του πίνακα σελίδων θα ήταν άδειο, αφού οι περισσότερες διαδικασίες χρησιμοποιούν μόνο ένα μικρό κλάσμα του συνολικού δυνατού χώρου διευθύνσεων. Συνεπώς, ο πίνακας σελίδων τμηματοποιήθηκε. Τα άνω 4 bits του αριθμού σελίδας θεωρούνται ως αριθμός τμήματος το οποίο χρησιμεύει για την επιλογή μιας από τις δεκαέξι εγγραφές του πίνακα τμημάτων. Κάθε τμήμα μπορεί να είναι μέχρι 268.435.456 bytes σε μήκος, παρ' όλο που το μέγεθος του πρέπει να είναι ένα πολλαπλάσιο των 4096 bytes. Η εγγραφή του πίνακα τμημάτων δείχνει τη βάση ενός πίνακα σελίδων γι' αυτό το τμήμα, και επίσης δείχνει το μήκος του πίνακα σελίδων (Σχήμα 4.32). Με αυτό τον τρόπο μεγάλες περιοχές του πίνακα σελίδων που ήταν μηδέν θα μπορούσαν να εκφυλιστούν θέτοντας τη διεύθυνση του πίνακα σελίδων στο 0.

**Σχήμα 4.32**

Τμηματοποιημένη
σελιδοποίηση του
IBM 360/67

Φυσικά, αυτός ο τρόπος απαιτεί, στη χειρότερη περίπτωση, τρεις προσπελάσεις μνήμης για κάθε επιθυμητή αναφορά μνήμης. Ένα σύνολο από 8 συσχετιστικούς καταχωρητές μειώνει το επιπλέον κόστος σε μόνο 150ns πιο αργά απ' ό,τι μια «μη αντιστοιχούμενη» (unmapped) αναφορά (750ns) όταν σημειώνεται επιτυχία.

Το σημαντικό είναι ότι ο χρήστης συνεχίζει να βλέπει ένα γραμμικό σελιδοποιημένο χώρο διευθύνσεων. Ένα αποτέλεσμα αυτού του σχεδιασμού είναι ότι είναι δυνατόν να αυξηθεί ένας δείκτης και να παραγάγει πρώτα την τελευταία θέση του τμήματος i και κατόπιν του τμήματος $i + 1$. Η μνήμη παραμένει στη βάση της γραμμικής.

4.8.2 Σελιδοποιημένη τμηματοποίηση (Paged Segmentation)

Το σύστημα Multics αντιμετώπιζε ένα διαφορετικό πρόβλημα. Οι λογικές διευθύνσεις σχηματίζονται από έναν αριθμό τμήματος των 18 bits και μια μετατόπιση (offset) των 16 bits. Παρ' όλο που αυτό το σχήμα δημιούργησε ένα χώρο με διευθύνσεις των 34 bits, το επιπλέον κόστος για τον πίνακα είναι ανεκτό, αφού ο μεταβλητός αριθ-

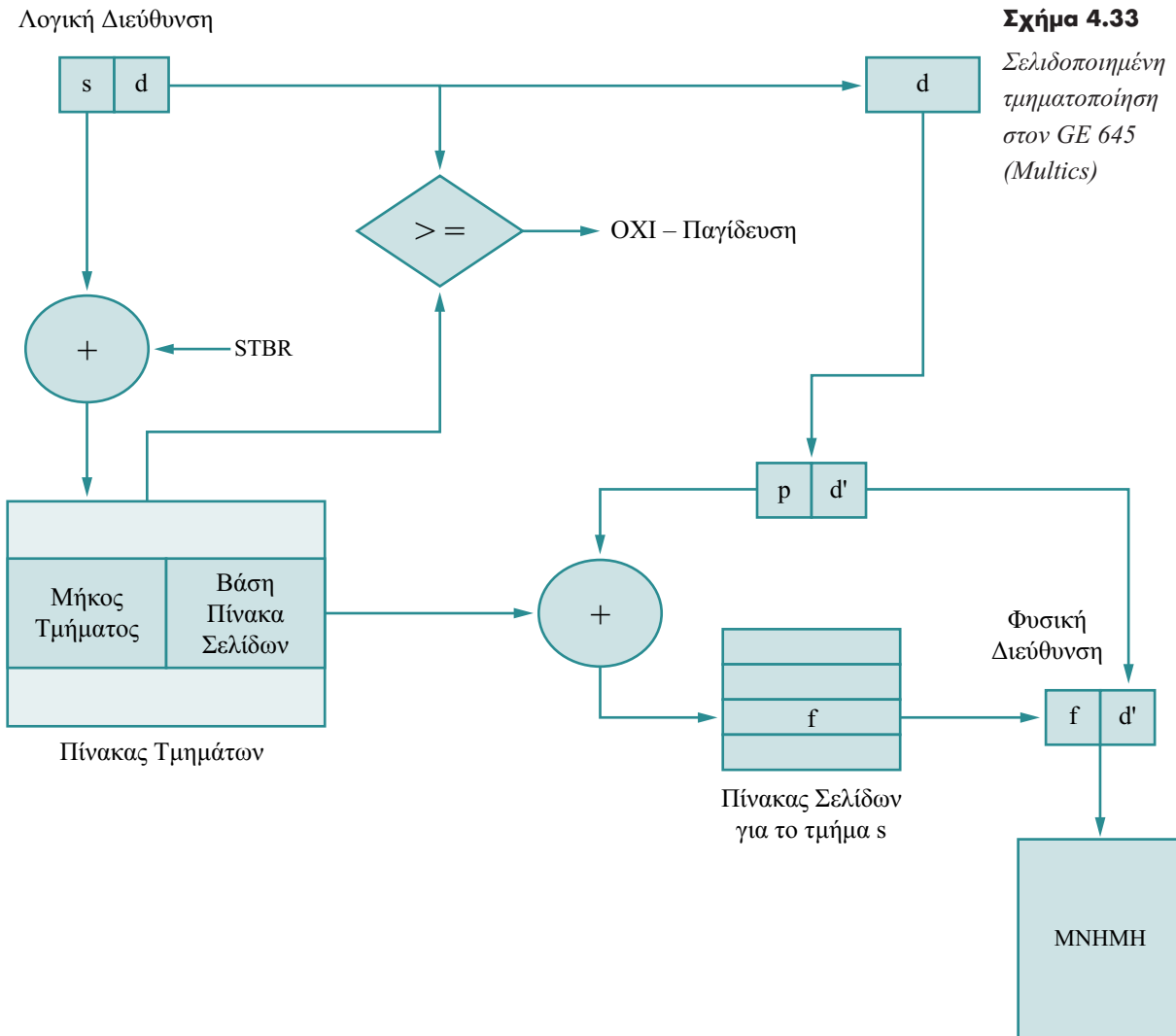
μός των τμημάτων υπονοεί έναν «Καταχωρητή Μήκους Πίνακα Τμημάτων» (STLR – Segment Table Length Register). Χρειαζόμαστε μόνο τόσες (πόσες;) εγγραφές στον πίνακα τμημάτων.

Όμως, με τμήματα των 64K λέξεων, το μέσο μέγεθος τμήματος μπορεί να είναι αρκετά μεγάλο και η εξωτερική κλασματοποίηση μπορεί να δημιουργεί προβλήματα. Ακόμα κι αν η εξωτερική κλασματοποίηση δεν είναι πρόβλημα, ο χρόνος ανίχνευσης για την κατανομή ενός τμήματος με τη χρήση του «πρώτου ταιριάσματος» ή του «καλύτερου ταιριάσματος» θα μπορούσαμε να είναι μεγάλος. Έτσι, μπορεί να σπαταλάμε μνήμη εξαιτίας της εξωτερικής κλασματοποίησης ή να σπαταλάμε χρόνο εξαιτίας μακροσκελών ανιχνεύσεων, ή και τα δύο.

Η λύση που υιοθετήθηκε ήταν η σελιδοποίηση των τμημάτων. Η σελιδοποίηση εξαφανίζει την εξωτερική κλασματοποίηση και μηδενίζει το πρόβλημα κατανομής: οποιoδήποτε άδειο πλαίσιο μπορεί να χρησιμοποιηθεί για την επιθυμητή σελίδα. Το αποτέλεσμα φαίνεται στο Σχήμα 4.33. Παρατηρήστε ότι η διαφορά μεταξύ αυτής της λύσης και της καθαρής τμηματοποίησης είναι ότι η εγγραφή του πίνακα τμημάτων δεν περιέχει τη διεύθυνση βάσης του τμήματος, αλλά τη «διεύθυνση βάσης» (base address) του πίνακα σελίδων γι' αυτό το τμήμα. Η «μετατόπιση του τμήματος» (segment offset) σπάει σε έναν αριθμό σελίδας των 6 bits και σε μια «μετατόπιση σελίδας» (page offset) των 10 bits. Ο αριθμός σελίδας δεικτοδοτεί στον πίνακα σελίδων για να δώσει τον αριθμό πλαισίου. Τελικά, ο αριθμός πλαισίου συνδυάζεται με τη μετατόπιση σελίδας για να σχηματίσει τη φυσική διεύθυνση.

Τώρα πρέπει να έχουμε έναν ξεχωριστό πίνακα σελίδων για κάθε τμήμα. Πάντως, εφόσον κάθε τμήμα περιορίζεται σε μήκος από την εγγραφή του στον πίνακα τμημάτων, ο πίνακας σελίδων δε χρειάζεται να έχει πλήρες μέγεθος. Απαιτεί μόνο τόσες εγγραφές όσες χρειάζονται πραγματικά. Πρόσθετα, η τελευταία σελίδα κάθε τμήματος γενικά δε θα είναι πλήρης. Έτσι, θα έχουμε, κατά μέσο όρο, μισή σελίδα εσωτερικής κλασματοποίησης ανά τμήμα. Συνεπώς, παρ' όλο που εξαφανίσαμε την εξωτερική κλασματοποίηση, εισαγάγαμε εσωτερική κλασματοποίηση και αυξήσαμε το επιπλέον κόστος του χώρου πίνακα.

Άλλα συστήματα επίσης χρησιμοποιούν τη σελιδοποιημένη τμηματοποίηση. Το σύστημα Prime 500 χρησιμοποιεί έναν αριθμό σελίδας των 12 bits και μια μετατόπιση λέξης (word offset) χωρισμένη σε έναν αριθμό σελίδας των 6 bits και μια μετατόπιση των 10 bits. Το RCA Spectra 70/46 είχε έναν αριθμό τμήματος των 5 bits και μια μετατόπιση των 18 bits, αποτελούμενη από έναν αριθμό σελίδας των 6 bits και μια 12 bits μετατόπιση σελίδας.



Η σελιδοποιημένη τμηματοποίηση του Multics, που παρουσιάσαμε, είναι υπεραπλουστευμένη. Εφόσον ο αριθμός τμήματος είναι μια ποσότητα των 18 bits, θα μπορούσαμε να έχουμε έως 262.144 τμήματα, που θα απαιτούσαν ένα πολύ μεγάλο πίνακα τμημάτων. Για να ελαφρυνθεί αυτό το πρόβλημα, το Multics σελιδοποιεί τον πίνακα τμημάτων. Έτσι, γενικά, μια διεύθυνση στο Multics χρησιμοποιεί τον αριθμό τμήματος για να ορίσει ένα δείκτη σελίδας σε έναν πίνακα σελίδων για τον πίνακα τμημάτων. Από αυτή την εγγραφή εντοπίζει το μέρος του πίνακα τμημάτων που έχει την εγγραφή γι' αυτό το τμήμα. Η εγγραφή του πίνακα τμημάτων δείχνει σε έναν πίνακα σελίδων γι' αυτό το τμήμα, που καθορίζει το πλαίσιο που περιέχει την επιθυμητή λέξη.

Σύνοψη

Οι αλγόριθμοι / πολιτικές διαχείρισης μνήμης για πολυπρογραμματισμένα ΛΣ εκτείνονται από την απλή προσέγγιση του παραμένοντα επόπτη έως τη σελιδοποιημένη τμηματοποίηση. Ο προσδιορισμός της πολιτικής που χρησιμοποιείται σε κάποιο συγκεκριμένο σύστημα εξαρτάται άμεσα από την αρχιτεκτονική του συστήματος και τις δυνατότητες του υλικού που διατίθεται. Κάθε διεύθυνση μνήμης που παράγεται από την CPU, αφού ελεγχθεί και βρεθεί επιτρεπτή, θα αντιστοιχιστεί σε μια φυσική διεύθυνση. Ο έλεγχος μπορεί να υλοποιηθεί αποδοτικά μόνο από το υλικό.

Οι αλγόριθμοι διαχείρισης μνήμης που συζητήθηκαν (γυμνή μηχανή, παραμένων επόπτη, σύστημα σταθερών περιοχών, σύστημα μεταβλητών διαιρέσεων, σελιδοποίηση, τμηματοποίηση, και συνδυασμός αυτών των δύο) διαφέρουν σε πολλά σημεία. Ο ακόλουθος κατάλογος παρουσιάζει μερικά από τα πιο σημαντικά σημεία για τη σύγκριση των διαφορετικών πολιτικών διαχείρισης μνήμης:

- **Υποστήριξη υλικού.** Ένας απλός καταχωρητής φραγής ή ένα ζευγάρι καταχωρητών ορίων είναι αρκετά για τον παραμένοντα επόπτη, το σύστημα σταθερών περιοχών και το σύστημα μεταβλητών διαιρέσεων, ενώ η σελιδοποίηση και η τμηματοποίηση χρειάζονται «πίνακες αντιστοίχισης» (*mapping tables*) για να ορίσουν την αντιστοίχιση διεύθυνσης.
- **Απόδοση (Performance).** Όσο ο αλγόριθμος γίνεται πιο πολύπλοκος, ο χρόνος που χρειάζεται για την αντιστοίχιση μιας λογικής διεύθυνσης σε μια φυσική αυξάνεται. Για τα απλά συστήματα χρειάζεται μόνο να συγκρίνουμε ή να προσθέσουμε (τι;) στη λογική διεύθυνση λειτουργίες που είναι αρκετά γρήγορες. Η σελιδοποίηση και η τμηματοποίηση μπορούν να είναι το ίδιο γρήγορες αν ο πίνακας υλοποιηθεί με γρήγορους καταχωρητές. Αν, όμως, ο πίνακας βρίσκεται στη μνήμη, οι προσπελάσεις στη μνήμη του χρήστη μπορεί να πέσουν αρκετά σε απόδοση. Ένα σύνολο από συσχετιστικούς καταχωρητές μπορεί να μειώσει την πτώση της απόδοσης σε ένα αποδεκτό επίπεδο.
- **Κλασματοποίηση (Fragmentation).** Για ένα δεδομένο σύνολο διαδικασιών, το επίπεδο πολυπρογραμματισμού μπορεί να αυξηθεί μόνο με την τοποθέτηση περισσότερων διαδικασιών στη μνήμη. Για την επίτευξη αυτού του σκοπού πρέπει να μειώσουμε τη σπατάλη ή κλασματοποίηση της μνήμης. Συστήματα με σταθερού μεγέθους «περιοχές κατανομής» (*allocation units*), όπως το σύστημα σταθερών περιοχών ή η σελιδοποίηση, υποφέρουν από εσωτερική κλασματοποίηση. Σύστημα με μεταβλητού μεγέθους περιοχές, όπως το σύστημα μεταβλητών διαιρέσεων και η τμηματοποίηση, πάσχουν από εξωτερική κλασματοποίηση.

- **Μετατόπιση (Relocation).** Μια λύση στο πρόβλημα της εξωτερικής κλασματοποίησης είναι η συμπίεση. Η συμπίεση περιέχει τη μεταφορά διαδικασιών μέσα στη μνήμη χωρίς να υπάρχουν επιπτώσεις στην εκτέλεση των διαδικασιών αυτών. Για να μην επηρεάζεται η εκτέλεση των διαδικασιών, απαιτείται οι λογικές διευθύνσεις να μετατοπίζονται δυναμικά κατά το χρόνο εκτέλεσης. Αν οι διευθύνσεις μετατοπίζονται μόνο κατά το χρόνο φόρτωσης, δεν μπορούμε να συμπίεσουμε το χώρο αποθήκευσης.
- **Εναλλαγή (Swapping).** Σε κάθε αλγόριθμο μπορεί να προστεθεί η εναλλαγή. Κατά διαστήματα, που καθορίζονται από το ΛΣ και συνήθως υπαγορεύονται από τους αλγόριθμους χρονοδρομολόγησης της CPU, διαδικασίες αντιγράφονται από την κύρια μνήμη προς μια επικουρική μνήμη και αργότερα αντιγράφονται ξανά στην κύρια μνήμη. Αυτό το σχήμα επιτρέπει να εκτελεστούν περισσότερες δουλειές απ' όσες μπορούν εφάπαξ να χωρέσουν στη μνήμη.
- **Μοίρασμα (Sharing).** Ένας άλλος τρόπος αύξησης του επιπέδου του πολυπρογραμματισμού είναι το μοίρασμα κώδικα και δεδομένων μεταξύ των διαφόρων διαδικασιών. Το μοίρασμα, γενικά, απαιτεί τη χρήση είτε της σελιδοποίησης είτε της τμηματοποίησης, ώστε να παρέχονται μικρά πακέτα πληροφορίας (σελίδες ή τμήματα) που μπορούν να διαμοιραστούν. Το μοίρασμα είναι ένας τρόπος που επιτρέπει σε πολλές διαδικασίες να εκτελεστούν με περιορισμένο ποσό μνήμης, αλλά οι διαμοιρασμένες διαδικασίες και δεδομένα πρέπει να σχεδιαστούν προσεκτικά.
- **Προστασία (Protection).** Αν χρησιμοποιείται σελιδοποίηση ή τμηματοποίηση, διαφορετικά κομμάτια μιας διαδικασίας μπορούν να κληρυχθούν ως «εκτέλεση μόνο», «ανάγνωση μόνο», ή «ανάγνωσης – εγγραφής». Αυτός ο περιορισμός είναι απαραίτητος όταν υπάρχει διαμοιρασμένος κώδικας ή δεδομένα, και είναι γενικά χρήσιμο σε κάθε περίπτωση να παρέχονται απλοί έλεγχοι, κατά το χρόνο τρεξίματος, για συνήθη προγραμματιστικά λάθη.

Βιβλιογραφία κεφαλαίου

Η εξωτερική και εσωτερική κλασματοποίηση συζητούνται στον Randell [1969]. Το σχήμα κατανομής με «σταθερές διαιρέσεις» (fixed partitions) χρησιμοποιείται στο IBM OS/360, το σύστημα σταθερών περιοχών. Το σχήμα κατανομής με «μεταβλητές διαιρέσεις» (variable partitions) χρησιμοποιείται στο IBM OS/360, το σύστημα μεταβλητών διαιρέσεων. Συζητήσεις και των δύο συστημάτων υπάρχουν στους Knight [1968], Madnick and Donovan [1974, Section (3. 3)], Hoare and McKeage [1972]. Η ιδέα της σελιδοποίησης μπορεί να αποδοθεί στους σχεδιαστές του συστήματος Atlas [Kilburn et al. 1961, 1962]. Ένα από τα πρώτα σελιδοποιημένα συστήματα είναι το XDS – 940, που περιγράφεται από τους Lichtenberger and Pirtle [1965] και Lampson et al. [1966].

ΠΡΟΑΙΡΕΤΙΚΗΣ ΑΝΑΓΝΩΣΗΣ

Knuth, *The art of computer programming vol. 1: fundamental algorithms*, 2nd edition reading, MA: Addison–Wesley, 1973.

Silberschatz et al., *Operating system concepts*, 3rd edition reading, MA: Addison–Wesley, 1991.

Γλωσσάρι κεφαλαίου

Absolute addresses:	απόλυτες διευθύνσεις
Address space:	χώρος διευθύνσεων
Affective access time:	χρόνος προσπέλασης
Bare machine:	γυμνή μηχανή
Base register:	καταχωρητής βάσης
Basic binary loader:	βασικός δυαδικός φορτωτής
Bind:	δένω
Buffer space:	μνήμη απομονωτών
Control card:	κάρτα ελέγχου
CPU scheduling:	χρονοδρομολόγηση CPU
CPU utilization:	χρήση της CPU
Dedicated system:	εξειδικευμένο σύστημα
Device driver:	οδηγός μονάδας
Dispatcher:	διεκπεραιωτής
Dynamic relocation:	δυναμική μετατόπιση
Fence address:	διεύθυνση φραγής
Fetch:	φέρνω
Indexing:	χρήση δεικτών
Interrupt vector:	διάνυσμα διακοπών
Job sequencing:	σειριοποίηση διαδικασιών
Linkage editor:	διασυνδετής
Literal addresses:	σταθερές διευθύνσεις
Loader:	φορτωτής
Logical address space:	χώρος λογικών διευθύνσεων
Memory dump:	περιεχόμενα μνήμης
Memory mapping hardware:	υλικό αντιστοίχισης μνήμης

Module:	τμήμα
Monitor mode:	καθεστώς επόπτη
Moving:	μεταφορά
Operand:	τελεστής
Program counter:	μετρητής προγράμματος
Protection hardware:	υλικό προστασίας
Ready queue:	ουρά ετοιμότητας
Relocatable addresses:	μετατοπιζόμενες διευθύνσεις
Relocatable code:	μετατοπιζόμενος κώδικας
Relocation register:	καταχώρητης μετατόπισης
Resident monitor:	παραμένων επόπτης
Source program:	πηγαίο πρόγραμμα
Special privilege instruction:	ειδικό περιεχόμενο εντολής
Store:	αποθηκεύω
Stream:	σειρά, ακολουθία
Swap out:	εναλλάσσω έξω
Swap in:	εναλλάσσω μέσα
Swap time:	χρόνος εναλλαγής
System call:	κλήση συστήματος
Time quantum:	κβάντο χρόνου
Transfer rate:	ρυθμός μεταφοράς
Transient monitor code:	μεταβατικός κώδικας επόπτη
Trap:	παγίδα

Ιδεατή μνήμη (Virtual Memory)

Στο προηγούμενο κεφάλαιο συζητήσαμε διάφορες στρατηγικές διαχείρισης μνήμης που έχουν χρησιμοποιηθεί σε υπολογιστικά συστήματα. Όλες αυτές οι στρατηγικές έχουν τον ίδιο στόχο: να κρατούν πολλές διεργασίες στη μνήμη ταυτόχρονα, ώστε να επιτρέπουν πολυπρογραμματισμό. Όμως, μέχρι τώρα υποθέσαμε ότι, για να μπορεί να εκτελεστεί μια διεργασία, απαιτείται να βρίσκεται εξολοκλήρου στη μνήμη.

Η «ιδεατή μνήμη» είναι μια τεχνική που επιτρέπει την εκτέλεση διεργασιών που μπορεί να μην είναι ολόκληρες στη μνήμη. Το κύριο ορατό πλεονέκτημα αυτού του σχήματος είναι ότι τα προγράμματα του χρήστη μπορούν να είναι μεγαλύτερα από τη διαθέσιμη φυσική μνήμη. Όμως, η ιδεατή μνήμη δεν είναι εύκολο να υλοποιηθεί και μπορεί να μειώσει σημαντικά την απόδοση αν χρησιμοποιηθεί απρόσεκτα. Σε αυτό το κεφάλαιο θα συζητήσουμε την ιδεατή μνήμη στη μορφή της «σελιδοποίησης ζήτησης» (demand paging) (δες Ενότητα 5.2) και θα εξετάσουμε την πολυπλοκότητά της και το κόστος της.

Σκοπός

Ο σκοπός αυτού του κεφαλαίου είναι η μελέτη μιας ιδιαίτερα σημαντικής τεχνικής, η οποία ονομάζεται «ιδεατή μνήμη», η οποία επιτρέπει προγράμματα μεγαλύτερου μεγέθους από τη διαθέσιμη φυσική μνήμη. Θα ορίσουμε τις βασικές έννοιες που προκύπτουν από τη μελέτη της τεχνικής αυτής, καθώς και διάφορους αλγόριθμους αντικατάστασης σελίδας. Τέλος, θα προσδιορίσουμε τις σχέσεις που έχουν οι παράμετροι ενός συστήματος που χρησιμοποιεί ιδεατή μνήμη και τις βέλτιστες τιμές τους.

Προσδοκώμενα αποτελέσματα

Με τη μελέτη του κεφαλαίου αυτού, θα είστε σε θέση να:

- Εξηγήσετε πώς μπορεί να εκτελεστεί μια διαδικασία χωρίς να βρίσκεται εξολοκλήρου στην κύρια μνήμη.
- Εξηγήσετε τι είναι και πώς λειτουργεί το σύστημα σελιδοποίησης ζήτησης.
- Ορίσετε τι είναι το σφάλμα σελίδας και πώς αντιμετωπίζεται. Ποια βήματα ακολουθεί το ΛΣ όταν προκύψει ένα σφάλμα σελίδας;
- Αναφέρετε τι υλικό χρειάζεται για να υλοποιηθεί το σύστημα σελιδοποίησης ζήτησης και γιατί.

- Εξηγήστε τι θα συμβεί αν προκύψει σφάλμα σελίδας στο μέσο μιας εντολής. Πώς αποφεύγονται τα σφάλματα σελίδων σε αυτές τις περιπτώσεις;
- Εξηγήστε πώς επηρεάζεται η απόδοση του συστήματος από το ρυθμό σφαλμάτων σελίδας. Γιατί ο ρυθμός σφαλμάτων σελίδας πρέπει να είναι πολύ μικρός; Πώς μπορεί να επιτευχθεί κάτι τέτοιο; Από τι επηρεάζεται ο ρυθμός σφαλμάτων σελίδας;
- Αιτιολογήστε πότε χρειάζεται να αντικατασταθεί μια σελίδα στη φυσική μνήμη και πώς επηρεάζουν το βαθμό πολυπρογραμματισμού ένας γενικός και ένας τοπικός αλγόριθμος αντικατάστασης σελίδας. Ποια τα πλεονεκτήματα και ποια τα μειονεκτήματα των αλγόριθμων που ανήκουν στις πιο πάνω κατηγορίες;
- Περιγράψτε πώς δουλεύουν οι διάφοροι αλγόριθμοι αντικατάστασης σελίδας και να μπορείτε να τους συγκρίνετε τόσο από πλευράς χρόνου όσο και σε σχέση με το βέλτιστο αλγόριθμο.
- Εξηγήστε γιατί ο βέλτιστος αλγόριθμος αντικατάστασης σελίδας δεν είναι υλοποιήσιμος και να εξηγήσετε ποιες παραλλαγές του υιοθετούνται, πώς υλοποιούνται και τι στήριξη χρειάζεται η καθεμιά από το υλικό.
- Εξηγήστε πώς επηρεάζει την απόδοση του συστήματος ο αριθμός των πλαισίων σελίδων. Τι συμβαίνει αν αυξηθούν τα διαθέσιμα πλαίσια σελίδων και τι αν μειωθούν; Τι είναι η ανωμαλία του Belady; Ποιοι αλγόριθμοι υποφέρουν από αυτή και γιατί;
- Εξηγήστε γιατί εξοικονομούνται λειτουργίες I/O με τη χρήση της δεξαμενής ελεύθερων πλαισίων.
- Εξηγήστε γιατί χρειάζεται να κατανέμονται σε κάθε διαδικασία τουλάχιστον ένας ελάχιστος αριθμός πλαισίων. Πώς επηρεάζεται η απόδοση του συστήματος και ο χρόνος εκτέλεσης της διαδικασίας από αυτό τον αριθμό;
- Εξηγήστε γιατί δεν είναι ντετερμινιστικά καθορισμένος (πάντα ο ίδιος) ο χρόνος εκτέλεσης μιας διαδικασίας όταν ο αλγόριθμος αντικατάστασης σελίδας είναι γενικής αντικατάστασης. Δηλαδή πώς επηρεάζουν το χρόνο εκτέλεσης μιας διαδικασίας οι υπόλοιπες διαδικασίες στο σύστημα και η ακολουθία αναφορών.
- Ορίστε τι είναι ο λυγισμός. Πότε η χρήση της CPU μειώνεται ενόσω αυξάνεται ο βαθμός πολυπρογραμματισμού;
- Εξηγήστε πώς επηρεάζει η δομή ενός προγράμματος (διαδικασίας) το ρυθμό λαθών σελίδας.
- Εξηγήστε πώς η στρατηγική του συνόλου εργασίας εμποδίζει το λυγισμό και πώς διατηρεί το βαθμό πολυπρογραμματισμού υψηλό.

- Ορίσετε, να εξηγήσετε και να καθορίσετε το σκοπό που εξυπηρετούν και τις παραμέτρους του συστήματος που επηρεάζουν (και επηρεάζονται) οι εξής όροι: λυγισμός, τοπικότητα, μοντέλο συνόλου εργασιών, ίχνος αναφοράς σελίδας, συχνότητα λαθών σελίδας, πρόωρη σελιδοποίηση, κλείδωμα σελίδων, μέγεθος σελίδας, δομή προγράμματος και ιεραρχία αποθήκευσης.

Έννοιες κλειδιά

- Ιδεατή μνήμη
- Σελιδοποίηση ζήτησης
- Ακολουθία αναφορών σελίδας
- Σφάλμα σελίδας
- Αντικατάσταση σελίδας
- Πλαίσιο-θύμα
- FIFO (First In First Out)
- Βέλτιστη αντικατάσταση
- LRU (Least Recently Used)
- Αντικατάσταση δεύτερης ευκαιρίας
- Λιγότερο συχνά χρησιμοποιούμενη σελίδα
- Πιο συχνά χρησιμοποιούμενη σελίδα
- Κλάσεις σελίδων
- Ad-hoc αλγόριθμοι
- Ανωμαλία του Belady
- Αλγόριθμοι γενικής και τοπικής κατανομής
- Λυγισμός
- Τοπικότητα
- Μοντέλο συνόλου εργασίας
- Ίχνος αναφοράς σελίδας
- Πρόωρη σελιδοποίηση

- Κλείδωμα σελίδων
- Ιεραρχία αποθήκευσης

Δομή κεφαλαίου

Το κεφάλαιο αυτό περιέχει τις παρακάτω ενότητες:

5.1 Επικαλύψεις

5.2 Σελιδοποίηση Ζήτησης

5.3 Απόδοση της Σελιδοποίησης Ζήτησης

5.4 Αντικατάσταση Σελίδας

5.5 Έννοιες της Ιδεατής Μνήμης

5.6 Αλγόριθμοι Αντικατάστασης Σελίδας

5.7 Αλγόριθμοι Κατανομής

5.8 Λυγισμός

5.9 Άλλοι Παράγοντες

5.1 Επικαλύψεις

Η βασική μας υπόθεση ότι κάθε διεργασία είναι γραμμένη σαν να ήταν μόνη της στο σύστημα είναι και σ' αυτό το κεφάλαιο η αφετηρία μας. Στο Κεφάλαιο 4 είδαμε με ποιον τρόπο το ΛΣ μπορεί να τοποθετεί τον κώδικα της διεργασίας σε διάφορα, όχι κατ' ανάγκη συνεχή, μέρη της μνήμης και η διαδικασία να εκτελείται σωστά ενώ είναι γραμμένη σαν να ήταν τοποθετημένη στη μνήμη χωρίς διακοπές, και μάλιστα ξεκινώντας από τη θέση 0. Στο κεφάλαιο αυτό θα δούμε με ποιον τρόπο το ΛΣ μπορεί να έχει στη μνήμη μέρος μόνο της διεργασίας: η υπόλοιπη βρίσκεται στο δίσκο («ιδεατή μνήμη») και φορτώνεται όταν (κι αν) χρειαστεί. Το πλεονέκτημα είναι φανερό: δεσμεύεται λιγότερη μνήμη απ' όση θα χρειαζόταν αλλιώς, και μάλιστα είναι δυνατόν να έχουμε διαδικασίες (προγράμματα) μεγέθους μεγαλύτερου της συνολικής φυσικής μνήμης του συστήματος. Τα μειονεκτήματα είναι δύο: ξοδεύεται χρόνος για να μπαίνει και να βγαίνει η διαδικασία από το δίσκο στη μνήμη και απαιτείται ένα πολυπλοκότερο ΛΣ για να το φροντίζει.

Άσκηση Αυτοαξιολόγησης 5.1

Σκεφτείτε και γράψτε διάφορους λόγους για τους οποίους το φόρτωμα ολόκληρης της διαδικασίας στη μνήμη (όπως υποθέσαμε στο Κεφάλαιο 4) μπορεί να αποτελέσει «σπατάλη» και, γενικότερα, τα μειονεκτήματα που θα προκαλούσε η απαίτηση αυτή.

Απάντηση:

Οι διαδικασίες συχνά έχουν κώδικα που χειρίζεται σπάνιες καταστάσεις λαθών. Επειδή αυτά τα λάθη σπάνια (αν ποτέ) συμβαίνουν στη χρήση, αυτός ο κώδικας σχεδόν ποτέ δεν εκτελείται.

Διάνυσμα (array), λίστες και πίνακες (tables) παίρνουν συνήθως περισσότερη μνήμη απ' ό,τι χρειάζονται πραγματικά. Ένα διάνυσμα μπορεί να δηλωθεί ότι έχει μέγεθος 100×100, παρ' όλο που σπάνια είναι μεγαλύτερο από 10×10. Ένας πίνακας συμβόλων ενός συμβολομεταφραστή μπορεί να έχει χώρο για 3000 σύμβολα, μόλο που το μέσο πρόγραμμα έχει λιγότερα από 200 σύμβολα.

Μερικές λειτουργίες και ικανότητες μιας διαδικασίας μπορεί να χρησιμοποιούνται σπάνια, όπως, για παράδειγμα, η εντολή ενός επεξεργαστή κειμένου που αλλάζει τους χαρακτήρες μιας ομάδας γραμμών σε κεφαλαία. Ακόμα και στις περιπτώσεις που ολόκληρη η διαδικασία χρειάζεται, μπορεί να μη χρειάζεται ολόκληρη την ίδια στιγμή. Η ικανότητα εκτέλεσης μιας διαδικασίας που βρίσκεται μερικώς μόνο στη

μνήμη έχει πολλά οφέλη.

Μια διαδικασία δε θα περιοριζόταν από το μέγεθος της φυσικής μνήμης που είναι διαθέσιμη. Οι χρήστες θα μπορούσαν να γράψουν προγράμματα (διαδικασίες) για έναν πολύ μεγάλο «ιδεατό χώρο διευθύνσεων» (virtual address space), απλοποιώντας έτσι την προγραμματιστική δουλειά.

Εφόσον ο κάθε χρήστης θα μπορούσε να καταλάβει λιγότερη φυσική μνήμη, περισσότεροι χρήστες θα μπορούσαν να τρέχουν στον ίδιο χρόνο, με αντίστοιχη αύξηση στη «χρήση» (utilization) και στο «ρυθμό» (throughput) της CPU, αλλά χωρίς αύξηση στο «χρόνο απόκρισης» (response time) ή στο «χρόνο ανακύκλωσης» (turnaround time).

Λιγότεροι I/O θα χρειάζονταν για το φόρτωμα ή την εναλλαγή κάθε χρήστη στη μνήμη, έτσι κάθε χρήστης θα έτρεχε πιο γρήγορα.

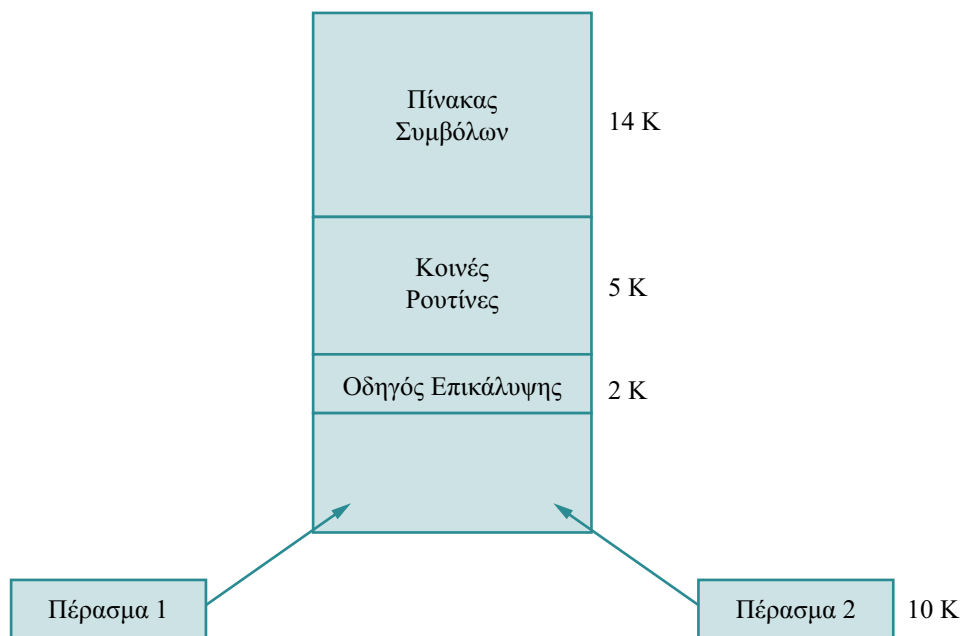
Έτσι, η εκτέλεση μιας διαδικασίας που δεν είναι ολόκληρη στη μνήμη θα ωφελούσε και το σύστημα και το χρήστη.

Ιστορικό σημείωμα:

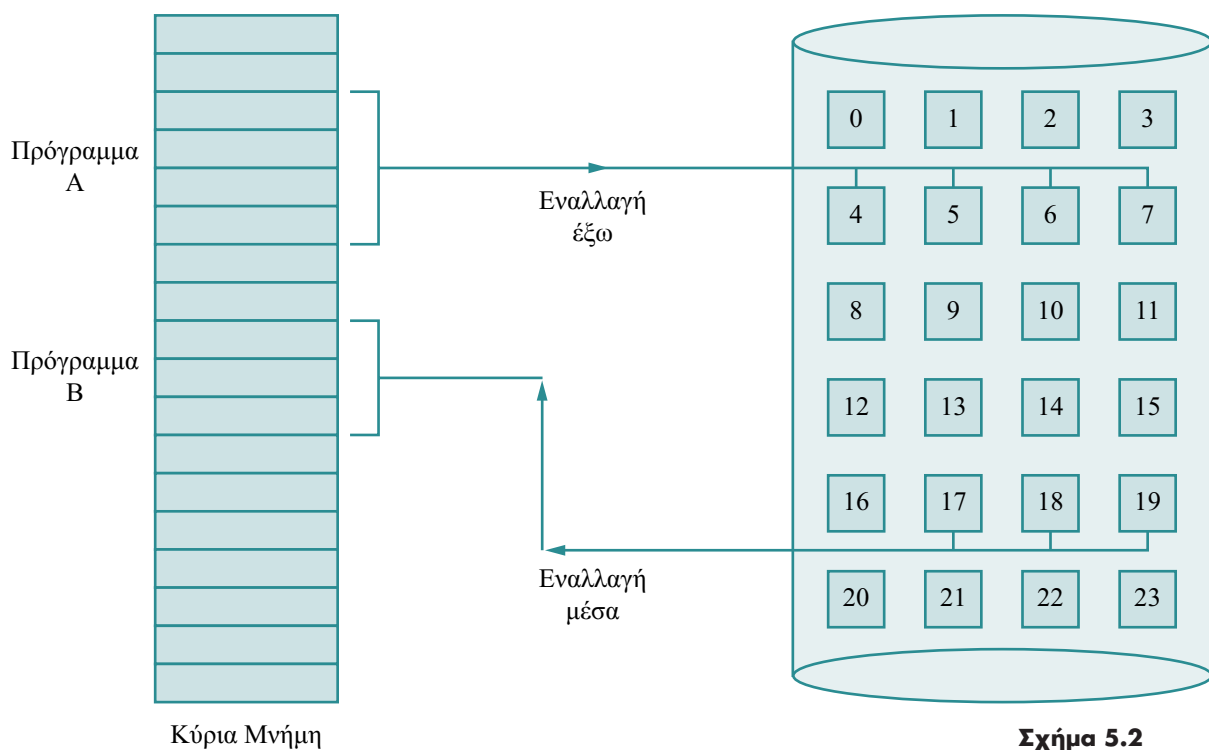
Στα πρώτα υπολογιστικά συστήματα χρησιμοποιήθηκαν «τεχνικές αυτοεπικάλυψης» (overlaying) των διαδικασιών. Έτσι, μια διαδικασία η οποία μπορεί να χωριστεί σε δύο μέρη με κοινά δεδομένα (π.χ. ένας συμβολομεταφραστής) θα μπορούσε να κερδίσει μνήμη φορτώνοντας στην πρώτη φάση το πρώτο μέρος με τα δεδομένα και στη συνέχεια το δεύτερο μέρος (στη θέση ακριβώς του πρώτου, επικαλύπτοντάς το). Κάτι τέτοιο ονομάζεται «δυναμικό» φόρτωμα ή φόρτωμα «κατ' απαίτηση»: στη μνήμη φορτώνεται ό,τι χρειάζεται τη στιγμή που θα χρειαστεί, μπορεί να γίνει από προγράμματα συστήματος (συμβολομεταφραστές, βάσεις δεδομένων κτλ.) και δεν εμπλέκει ιδιαίτερα το ΛΣ.

5.2 Σελιδοποίηση ζήτησης

Η «σελιδοποίηση ζήτησης» (demand paging) είναι παρόμοια με ένα σελιδοποιημένο σύστημα με εναλλαγή (Σχήμα 5.2). Οι διαδικασίες βρίσκονται στο δίσκο. Όταν έρθει ο χρόνος εκτέλεσης μιας διαδικασίας, αυτή φορτώνεται στη μνήμη, αλλά όχι ολόκληρη: κάθε σελίδα φορτώνεται μόνο όταν (κι αν) χρειάζεται. Έτσι, και ο χρόνος εναλλαγής μειώνεται, αλλά και το μέγεθος της απαιτούμενης μνήμης, αυξάνοντας έτσι το «βαθμό πολυπρογραμματισμού».

**Σχήμα 5.1**

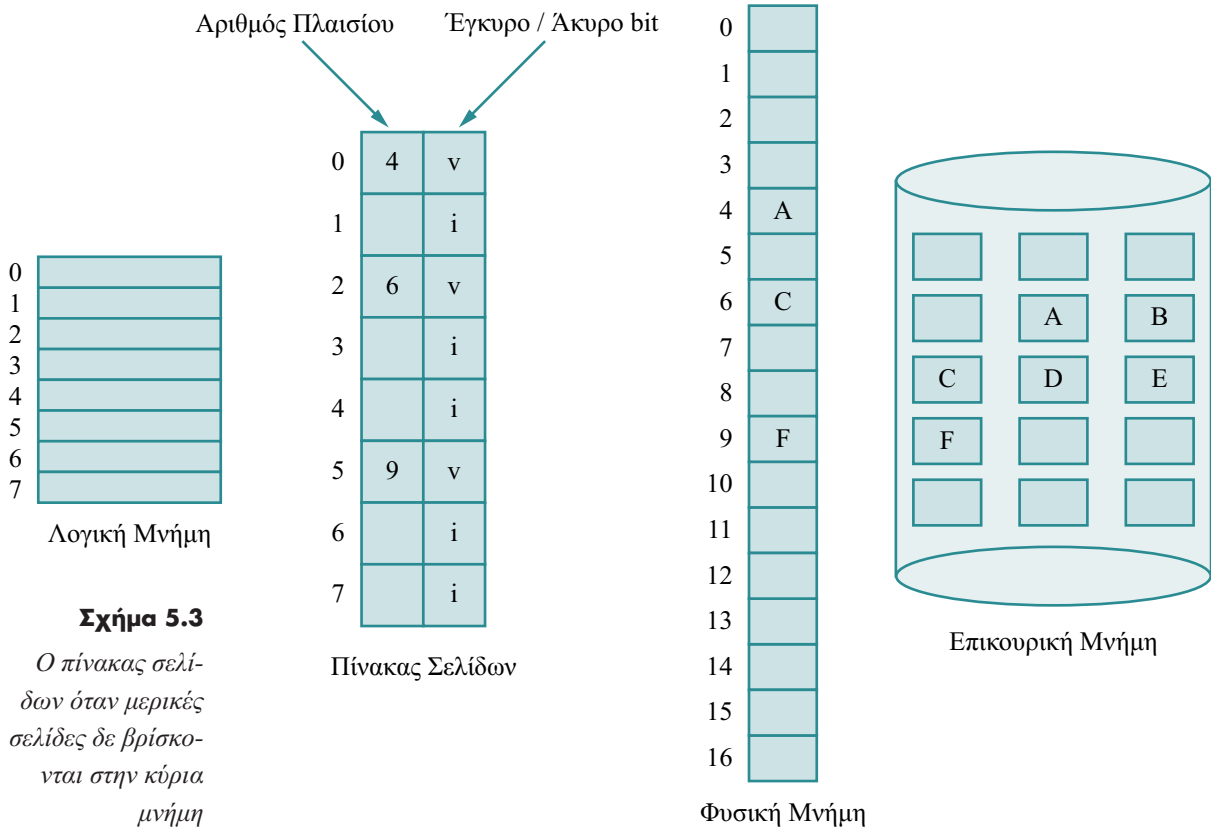
Επικαλύψεις για ένα συμβολομεταφραστή 2 περασμάτων

**Σχήμα 5.2**

Εναλλάσσοντας μια σελιδοποιημένη μνήμη σε συνεχή χώρο δίσκου

Αλλά τι συμβαίνει όταν η διαδικασία επιχειρήσει προσπέλαση σε σελίδα που δεν έχει μεταφερθεί στη μνήμη; Το ΛΣ θα επιχειρήσει να μεταφράσει τη διεύθυνση της

διαδικασίας μέσω του πίνακα σελίδων και δε θα βρει αντίστοιχη φυσική διεύθυνση, άρα, καταρχήν, η αναφορά αυτή θα θεωρηθεί «άκυρη» (σχετικό bit στον πίνακα σελίδων: «άκυρο» [Σχήμα 5.3]).



Σχήμα 5.3

Ο πίνακας σελίδων όταν μερικές σελίδες δε βρίσκονται στην κύρια μνήμη

Άσκηση Αυτοαξιολόγησης 5.2

Τι θα γίνει αν από την αρχή το ΛΣ είχε φορτώσει όλες και μόνο τις σελίδες που η εκτελούμενη διαδικασία χρησιμοποίησε;

Απάντηση:

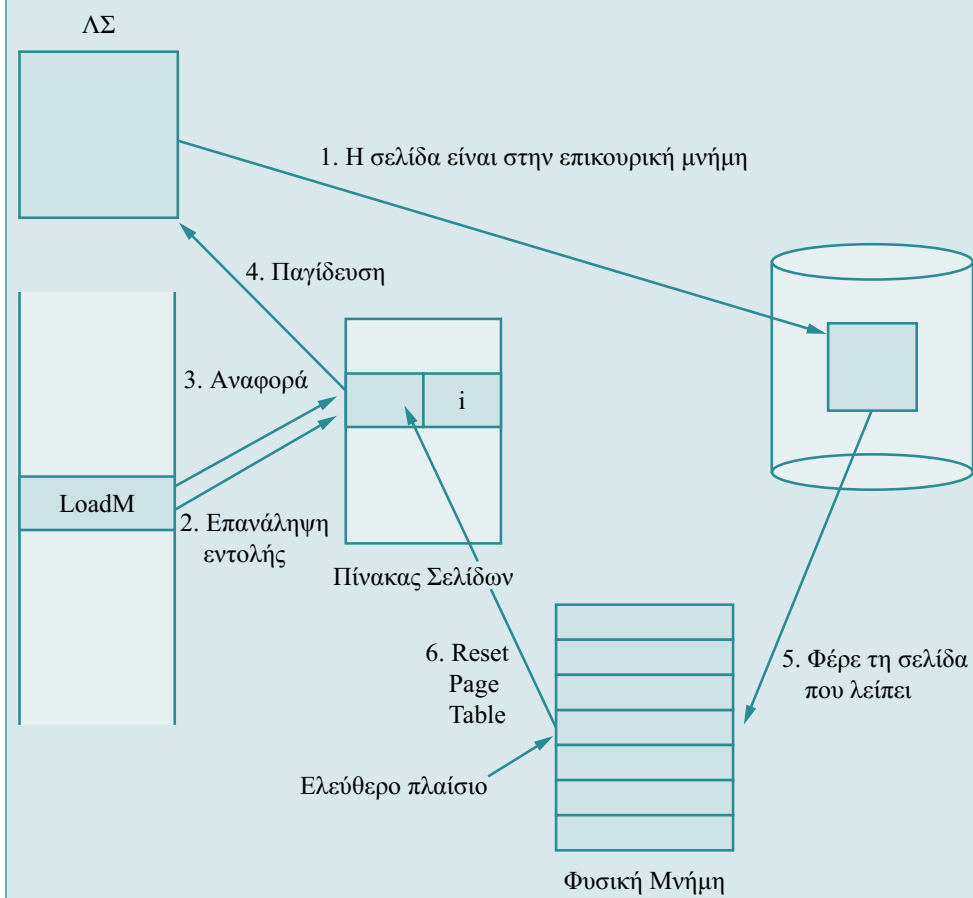
Παρατηρήστε ότι η ακύρωση, που αναφέραμε πιο πάνω, δεν έχει κανένα αποτέλεσμα αν η διαδικασία δεν προσπαθήσει ποτέ να προσπελάσει τη σελίδα που δε βρίσκεται στη μνήμη. Συνεπώς, αν το ΛΣ μαντέψει σωστά και «εναλλάξει-μέσα» όλες και μόνο εκείνες τις σελίδες που θα χρειαστούν πραγματικά, η διαδικασία θα τρέξει σαν να είχαν μεταφερθεί στη μνήμη όλες της οι σελίδες.

Αν όμως το ΛΣ μαντέψει λάθος και η διαδικασία προσπαθήσει να προσπελάσει μια

σελίδα που δε βρίσκεται στη μνήμη, τότε θα συμβεί «σφάλμα σελίδας» (page fault). Σε αντίθεση με τα συστήματα διαχείρισης μνήμης, για τα οποία μιλήσαμε στο Κεφάλαιο 4, μια αναφορά σε άκυρη διεύθυνση τώρα δε σημαίνει αναγκαστικά προσπάθεια ανεπίτρεπτης προσπέλασης: μπορεί να σημαίνει απλώς ότι η σελίδα αυτή δε βρίσκεται στην κύρια μνήμη. Επομένως, το ΛΣ (προς το οποίο θα στραφεί ο έλεγχος μετά την άκυρη αναφορά) θα πρέπει να ελέγξει μήπως η προσπέλαση ήταν επιτρεπτή, περίπτωση στην οποία θα πρέπει να φροντίσει για τη μεταφορά της σχετικής σελίδας από το δίσκο. Όσο, βέβαια, γίνεται η μεταφορά, κάποια άλλη διεργασία θα χρησιμοποιεί την CPU.

Άσκηση Αυτοαξιολόγησης 5.3

Δείτε το Σχήμα 5.4 και περιγράψτε τα βήματα που πρέπει να κάνει το ΛΣ στην περίπτωση που συναντήσει άκυρο bit σελίδας και προκληθεί η αντίστοιχη παγίδευση προς το ΛΣ.



Σχήμα 5.4

Βήματα χειρισμού ενός σφάλματος σελίδας

Απάντηση:

- (α) Το ΛΣ αρχικά ελέγχει έναν εσωτερικό πίνακα που κρατιέται στο τμήμα ελέγχου διαδικασίας (PCB) γι' αυτή τη διαδικασία, για να καθορίσει αν η αναφορά ήταν έγκυρη ή άκυρη προσπέλαση μνήμης. Αν ήταν άκυρη, τερματίζει η διαδικασία.
- (β) Αν ήταν έγκυρη αλλά δεν αφορά σελίδα που βρίσκεται στη μνήμη, τότε η σελίδα αυτή πρέπει να μεταφερθεί στη μνήμη.
- (γ) Στη συνέχεια, το ΛΣ βρίσκει ένα ελεύθερο πλαίσιο (π.χ. παίρνοντας ένα από τη λίστα των ελεύθερων πλαισίων).
- (δ) Προγραμματίζει μια λειτουργία δίσκου για διάβασμα της επιθυμητής σελίδας, στο πλαίσιο που επιλέχθηκε στο βήμα (γ).
- (ε) Όταν τελειώσει η ανάγνωση, τροποποιεί τον εσωτερικό πίνακα που κρατιέται στη διαδικασία και τον πίνακα σελίδων ώστε να δείχνει ότι η σελίδα βρίσκεται τώρα στη μνήμη.
- (ζ) Επαναλαμβάνει την εντολή που διακόπηκε από την «παγίδα παράνομης διεύθυνσης» (illegal address trap). Η διαδικασία μπορεί τώρα να προσπελάσει τη σελίδα σαν να ήταν πάντοτε στη μνήμη.

Είναι σημαντικό να αναγνωρίσουμε ότι με το να σωθεί η κατάσταση (καταχωρητές, μετρητής προγράμματος κτλ.) της διακοπέισας διαδικασίας, όταν προκληθεί η παγίδα παράνομης διεύθυνσης (σφάλμα σελίδας), τότε μπορεί να αρχίσει ξανά η διαδικασία στον ίδιο χώρο και στην ίδια κατάσταση, μόνο που τώρα η επιθυμητή σελίδα είναι στη μνήμη και είναι προσπελάσιμη. Με αυτό τον τρόπο μπορεί να εκτελείται μια διαδικασία παρ' όλο που κομμάτια της δε βρίσκονται (ακόμα) στη μνήμη. Το υλικό που χρειάζεται για την υποστήριξη της σελιδοποίησης ζήτησης είναι το ίδιο με αυτό της σελιδοποίησης και της εναλλαγής:

- Ένας πίνακας σελίδων με την ικανότητα σημείωσης μιας εγγραφής ως άκυρης, μέσω της χρήσης ενός έγκυρου / άκυρου bit ή μιας ειδικής τιμής των bits προστασίας.
- Επικουρική μνήμη, όπου θα κρατάει τις σελίδες που δε βρίσκονται στη μνήμη. Η μνήμη αυτή είναι συνήθως μια συσκευή με ταχύτητα ενδιάμεση του δίσκου και της μνήμης (συνήθως είτε ένας ειδικός υψηλής ταχύτητας δίσκος είτε cache memory). Πρέπει, επίσης, να επιβληθούν μερικοί ακόμη αρχιτεκτονικοί περιορι-

σμοί. Ένα κρίσιμο θέμα είναι η επανάληψη οποιασδήποτε εντολής μετά από ένα σφάλμα σελίδας. Στις περισσότερες περιπτώσεις αυτό είναι εύκολο να επιτευχθεί. Ένα σφάλμα σελίδας θα μπορούσε να συμβεί σε οποιαδήποτε αναφορά μνήμης. Αν το σφάλμα συμβεί κατά την «προσκόμιση εντολής» (instruction fetch), μπορούμε να αρχίσουμε φέρνοντας ξανά την εντολή. Αν συμβεί κατά την προσκόμιση ενός τελεστή, πρέπει να ξαναφέρουμε την εντολή, να την αποκωδικοποιήσουμε ξανά και μετά να φέρουμε τον τελεστή.

Ως χειρότερη περίπτωση θεωρήστε μια εντολή τριών διευθύνσεων, όπως η ADD A TO B, που τοποθετεί το αποτέλεσμα στο C. Τα βήματα εκτέλεσης αυτής της εντολής θα ήταν:

- (α) Προσκόμισε και αποκωδικοποίησε την εντολή (ADD).
- (β) Φέρε το A.
- (γ) Φέρε το B
- (δ) Πρόσθεσε.
- (ε) Αποθήκευσε το άθροισμα στο C.

Αν το σφάλμα συνέβαινε όταν προσπαθούσαμε να αποθηκεύσουμε στο C (επειδή το C βρίσκεται σε σελίδα που δεν υπάρχει στη μνήμη), θα έπρεπε να φέρουμε την επιθυμητή σελίδα στη μνήμη, να διορθώσουμε τον πίνακα σελίδων και να επαναλάβουμε την εντολή. Η επανάληψη θα απαιτούσε την επαναπροσκόμιση της εντολής, την αποκωδικοποίησή της, την προσκόμιση των τελεστών και μετά την πρόσθεσή τους ξανά. Πάντως, δεν επαναλαμβάνεται πολλή δουλειά (λιγότερη από μια ολόκληρη εντολή) και είναι η επανάληψη αναγκαία μόνο όταν συμβαίνει ένα σφάλμα σελίδας.

Μεγάλη δυσκολία υπάρχει όταν μια εντολή μπορεί να τροποποιήσει πολλές διαφορετικές θέσεις. Για παράδειγμα, σκεφτείτε την εντολή MVC (μετακίνηση χαρακτήρα) του IBM 370, που μπορεί να μετακινήσει μέχρι 256 bytes από μια θέση σε μια άλλη (πιθανόν επικαλυπτόμενη) θέση. Αν οποιοδήποτε από τα δύο τμήματα μνήμης (πηγής ή προορισμού) διασκελίζει ένα όριο σελίδας, ένα σφάλμα σελίδας μπορεί να συμβεί, αφού η μετακίνηση έχει εκτελεστεί μερικώς. Επιπρόσθετα, αν τα τμήματα μνήμης πηγής ή προορισμού επικαλύπτονται, το πηγαίο τμήμα μπορεί να έχει τροποποιηθεί, οπότε δεν μπορούμε απλώς να επαναλάβουμε την εντολή.

Αυτό το πρόβλημα λύνεται με δύο τρόπους, που εξαρτώνται από το μοντέλο που χρησιμοποιείται. Στη μια λύση, ο μικροκώδικας υπολογίζει και προσπαθεί να προσπελάσει και τις δύο άκρες και των δύο τμημάτων μνήμης. Αν πρόκειται να συμβεί ένα σφάλμα σελίδας, θα γίνει σε αυτό το βήμα, πριν να τροποποιηθεί οτιδήποτε.

Έτσι, η μετακίνηση μπορεί τώρα να εκτελεστεί με τη γνώση ότι δεν πρόκειται να συμβεί σφάλμα σελίδας, αφού όλες οι σχετιζόμενες με αυτή σελίδες υπάρχουν στη μνήμη. Η άλλη λύση χρησιμοποιεί προσωρινούς καταχωρητές για την κράτηση των τιμών των θέσεων που επανεγγράφονται. Αν συμβεί σφάλμα σελίδας, όλες οι παλιές τιμές ξαναγράφονται στη μνήμη πριν να συμβεί η παγίδευση. Αυτή η πράξη επαναφέρει τη μνήμη στην κατάσταση που είχε πριν να αρχίσει η εκτέλεση της εντολής, έτσι ώστε να είναι δυνατή η επανάληψή της.

Ένα παρόμοιο αρχιτεκτονικό πρόβλημα εμφανίζεται στο PDP-11, που χρησιμοποιεί ειδικούς τρόπους διευθυνσιοδότησης, που περιλαμβάνουν τους τρόπους «αυτόματης μείωσης» (auto-decrement) και «αυτόματης αύξησης» (auto-increment). Αυτοί οι τρόποι διευθυνσιοδότησης χρησιμοποιούν έναν καταχωρητή ως δείκτη και αυτόματα τον μειώνουν ή τον αυξάνουν, όπως είναι αναγκαίο. Η αυτόματη μείωση μειώνει τον καταχωρητή πριν να χρησιμοποιήσει το περιεχόμενό του ως την «τελευταία διεύθυνση» (operand address), ενώ η αυτόματη αύξηση αυξάνει τον καταχωρητή μετά τη χρησιμοποίηση του περιεχομένου του. Έτσι, η εντολή: `MOV ((R2) +, -(R3))` αντιγράφει τα περιεχόμενα της θέσης που δείχνεται από τον καταχωρητή 2 στη θέση που δείχνεται από τον καταχωρητή 3. Ο καταχωρητής 2 αυξάνεται (κατά δύο για μία λέξη, αφού ο PDP-11 είναι ένας byte-διευθυνσιοδοτούμενος υπολογιστής) αφού χρησιμοποιηθεί ως δείκτης, και ο καταχωρητής 3 μειώνεται (κατά δύο) πριν να χρησιμοποιηθεί ως δείκτης. Σκεφτείτε τώρα τι θα συμβεί αν έχουμε σφάλμα σελίδας όταν προσπαθήσουμε να αποθηκεύσουμε στη θέση που δείχνεται από τον καταχωρητή 3. Για να επαναρχίσουμε την εντολή, πρέπει να επαναφέρουμε τους δύο καταχωρητές στις τιμές που είχαν πριν ν' αρχίσουμε την εκτέλεση της εντολής. Στον PDP-11/45 και 11/70 ένας ειδικός καταχωρητής (SR1 – Status Register 1) δημιουργήθηκε για την παρακολούθηση του αριθμού του καταχωρητή και του ποσού τροποποίησης οποιουδήποτε καταχωρητή που μεταβάλλεται κατά τη διάρκεια της εκτέλεσης μιας εντολής. Αυτός ο καταχωρητής επιτρέπει στο ΛΣ την κατάργηση των αποτελεσμάτων μιας μερικώς εκτελεσμένης εντολής που προκάλεσε σφάλμα σελίδας.

Αυτά, φυσικά, δεν είναι τα μόνα αρχιτεκτονικά προβλήματα που δημιουργούνται όταν προσθέτουμε τη σελιδοποίηση σε μια υπάρχουσα αρχιτεκτονική για να επιτρέψουμε τη σελιδοποίηση ζήτησης. Μερικές από τις δυσκολίες που προκύπτουν είναι: Σε ένα υπολογιστικό σύστημα η σελιδοποίηση προστίθεται ανάμεσα στην CPU και στη μνήμη. Έτσι, συχνά πιστεύεται ότι η σελιδοποίηση μπορεί να προστεθεί σε οποιοδήποτε σύστημα. Ενώ, όμως, η υπόθεση αυτή είναι σωστή όταν η σελιδοποίηση χρησιμοποιείται για μετατόπιση όπου ένα σφάλμα σελίδας αντιπροσωπεύει ένα πολύ «σοβαρό λάθος» (fatal error), δεν ισχύει το ίδιο όταν το σφάλμα σελίδας σημαί-

νει ότι μια πρόσθετη σελίδα πρέπει να μεταφερθεί στη μνήμη και η διαδικασία πρέπει να επανεκτελεστεί.

5.3 Απόδοση της Σελιδοποίησης Ζήτησης

Η σελιδοποίηση ζήτησης μπορεί να επηρεάσει σημαντικά την απόδοση ενός υπολογιστικού συστήματος. Για να δούμε το γιατί, ας υπολογίσουμε τον «τελικό χρόνο προσπέλασης» (effective access time) για μια μνήμη σελιδοποίησης ζήτησης. Ο «χρόνος προσπέλασης μνήμης» (memory access time), ma , για τα περισσότερα υπολογιστικά συστήματα είναι από 100ns έως τα 2μs. Όταν δεν έχουμε σφάλματα σελίδας, ο τελικός χρόνος προσπέλασης είναι ίσος με τον ma . Αν, όμως, συμβεί ένα σφάλμα σελίδας, πρέπει πρώτα να διαβάσουμε τη σχετική σελίδα από την επικουρική μνήμη και μετά να προσπελάσουμε την επιθυμητή λέξη.

Έστω ότι p είναι η πιθανότητα να συμβεί ένα σφάλμα σελίδας ($0 \leq p \leq 1$). Θα περιμέναμε ότι το p είναι πολύ κοντά στο μηδέν, δηλαδή θα υπάρχουν λίγα σφάλματα σελίδας. Ο τελικός χρόνος προσπέλασης είναι:

$$\text{Τελικός χρόνος προσπέλασης} = (1-p) \times ma + p \times \text{«χρόνος σφάλματος»}$$

Για να υπολογίσουμε τον τελικό χρόνο προσπέλασης, πρέπει να ξέρουμε πόσος χρόνος χρειάζεται για την εξυπηρέτηση ενός σφάλματος σελίδας. Ένα σφάλμα σελίδας προκαλεί την παρακάτω ακολουθία ενεργειών:

1. Παγίδευση προς το ΛΣ.
2. Σώσιμο των καταχωρητών χρήστη και της κατάστασης της διαδικασίας (προγράμματος).
3. Έλεγχος ότι η διακοπή προκλήθηκε από σφάλμα σελίδας.
4. Έλεγχος της νομιμότητας της αναφοράς σελίδας και καθορισμός της θέσης της σελίδας στην επικουρική μνήμη.
5. Διάβασμα από την επικουρική μνήμη σ' ένα ελεύθερο πλαίσιο:
 - Αναμονή σε ουρά αυτής της μονάδας έως ότου εξυπηρετηθεί η αίτηση διαβάσματος.
 - Αναμονή έως ότου ολοκληρωθεί η εύρεση των δεδομένων στη μονάδα (device seek και/ή latency time)¹.

[1] Οι χρόνοι latency και seek time αφορούν το χρόνο που χρειάζεται το ΛΣ για να προσδιορίσει πού βρίσκεται μια σελίδα στην επικουρική μνήμη. Ο χρόνος μεταφοράς είναι ο χρόνος που χρειάζεται ώστε να μεταφερθεί μια σελίδα από την επικουρική μνήμη στην κύρια μνήμη.

- Μεταφορά της σελίδας σε ένα ελεύθερο πλαίσιο.
- 6. Ανάθεση της CPU σε κάποια άλλη διαδικασία (cpu scheduling).
- 7. Διακοπή από την επικουρική μνήμη (περάτωση I/O).
- 8. Σώσιμο των καταχωρητών και της κατάστασης της εκτελούμενης διαδικασίας.
- 9. Έλεγχος ότι η διακοπή προερχόταν από την επικουρική μνήμη.
- 10. Διόρθωση του πίνακα σελίδων και των άλλων πινάκων, ώστε να φαίνεται ότι η επιθυμητή σελίδα βρίσκεται τώρα στη μνήμη.
- 11. Αναμονή έως ότου η CPU δοθεί ξανά σε αυτή τη διαδικασία.
- 12. Επαναφορά των καταχωρητών χρήστη, κατάσταση προγράμματος και νέου πίνακα σελίδων στις σωστές τιμές και μετά συνέχιση της εντολής που είχε διακοπεί.

Μπορεί όλα αυτά τα βήματα να μην είναι αναγκαία σε κάθε περίπτωση. Για παράδειγμα, υποθέτουμε ότι στο βήμα 5 η CPU δίνεται σε μια άλλη διαδικασία όσο εκτελείται η λειτουργία I/O. Αυτό το σχήμα επιτρέπει τη χρήση της CPU, αλλά απαιτεί επιπλέον χρόνο για τη συνέχιση της ρουτίνας εξυπηρέτησης σφάλματος σελίδας όταν η μεταφορά I/O τελειώσει.

Σε κάθε περίπτωση βρισκόμαστε αντιμέτωποι με τρεις κύριες παραμέτρους του χρόνου εξυπηρέτησης σφάλματος σελίδας:

- Εξυπηρέτηση της διακοπής σφάλματος σελίδας.
- «Εναλλαγή μέσα» της σελίδας.
- Επανεκκίνηση της διαδικασίας.

Η πρώτη και η τρίτη παράμετρος μπορούν να μειωθούν, με προσεκτική κωδικοποίηση, σε αρκετές εκατοντάδες εντολές. Αυτές οι παράμετροι μπορεί να χρειαστούν από 100 έως 1000ns. Ο χρόνος εναλλαγής σελίδας είναι γύρω στα 9ms. Για παράδειγμα, ένα τύμπανο έχει τυπικούς χρόνους 8ms για *latency* και 1ms «χρόνο μεταφοράς» (transfer time). Αν χρησιμοποιηθεί δίσκος κινητής κεφαλής (μια όλο και πιο συνήθης τακτική), τότε πρέπει να περιλάβουμε και το «χρόνο ψαξίματος» (seek time).

Μια μονάδα κινητής κεφαλής θα μπορούσε να αυξήσει το συνολικό χρόνο εναλλαγής πάνω από τα 30ms. Επίσης, θυμηθείτε ότι ενδιαφερόμαστε μόνο για το «χρόνο εξυπηρέτησης μονάδος» (device service time). Αν υπάρχει ουρά διαδικασιών που περιμένουν για τη μονάδα (άλλες διαδικασίες που έχουν προκαλέσει σφάλματα σελίδας), πρέπει να προσθέσουμε και το «χρόνο ουράς της μονάδας» (device queueing time), καθώς πρέπει να περιμένουμε να ελευθερωθεί η μονάδα για να εξυπηρετήσει

την αίτησή μας. Έτσι, λοιπόν, ο χρόνος εναλλαγής αυξάνεται ακόμα πιο πολύ.

Ο χρόνος προσπέλασης στη μνήμη όπως και ο χρόνος εξυπηρέτησης ενός σφάλματος σελίδας εξαρτώνται από την τεχνολογία του υλικού. Το 1970 οι χρόνοι αυτοί ήταν 1msec και 10msec, αντίστοιχα. Το 2000 είναι 100nsec και 1000nsec αντίστοιχα. Παρ' ότι οι τεχνολογίες κύριας και επικουρικής μνήμης δεν εξελίσσονται με τον ίδιο ακριβώς τρόπο, χοντρικά εξακολουθούν να διαφέρουν κατά μία τάξη μεγέθους: μια σελίδα από το δίσκο χρειάζεται δεκαπλάσιο χρόνο από μια προσπέλαση στη μνήμη. Επομένως, ο τελικός χρόνος προσπέλασης θα είναι ανάλογος του ρυθμού σφαλμάτων σελίδας, ενώ ο χρόνος προσπέλασης, αν δε χρησιμοποιήσουμε ιδεατή μνήμη, θα είναι ανάλογος του συνολικού μεγέθους της διαδικασίας.

Άσκηση Αυτοαξιολόγησης 5.4

1. Βρείτε τα εξής τεχνικά χαρακτηριστικά του υπολογιστή σας: μέγεθος σελίδας, χρόνος μεταφοράς μιας τυχαίας σελίδας από το δίσκο στη μνήμη, χρόνος μαζικής μεταφοράς από το δίσκο στη μνήμη ανά σελίδα (γιατί αυτό διαφέρει από το προηγούμενο;), χρόνος προσπέλασης στη μνήμη.
2. Έστω μια διαδικασία N σελίδων και μια εκτέλεσή της, στη διάρκεια της οποίας χρησιμοποιούνται $p \cdot N$ διαφορετικές σελίδες και γίνονται A προσπελάσεις στη μνήμη. Υπολογίστε το μέσο χρόνο προσπέλασης στη μνήμη χωρίς ιδεατή μνήμη ως εξής: χρόνος φόρτωσης από δίσκο σε μνήμη των N σελίδων, συν το χρόνο των A προσπελάσεων, διά A . Τώρα υπολογίστε το μέσο χρόνο προσπέλασης στη μνήμη με ιδεατή μνήμη ως εξής: χρόνος φόρτωσης των $p \cdot N$ σελίδων (ξεχωριστά η καθεμία), συν χρόνο των A προσπελάσεων, διά A . Δώστε μερικές τιμές στα N , p και A , συγκρίνετε και σχολιάστε.

Αν θεωρήσουμε ότι ο μέσος χρόνος εξυπηρέτησης σφάλματος σελίδας είναι 10ms και ο χρόνος προσπέλασης μνήμης είναι 1μs τότε:

$$\begin{aligned} \text{Τελικός χρόνος προσπέλασης} &= (1 - p) \times (1 \mu\text{s}) + p \times (10 \text{ ms}) \\ &= ((1 - p) + 10000 \times p) \mu\text{s} \\ &= (1 + 9999 \times p) \mu\text{s} \end{aligned}$$

Βλέπουμε ότι ο τελικός χρόνος προσπέλασης είναι ευθέως ανάλογος του «ρυθμού σφαλμάτων σελίδας» (page fault rate). Αν μια προσπέλαση στις χίλιες προκαλεί ένα σφάλμα σελίδας, τότε ο τελικός χρόνος προσπέλασης είναι 11μs. Ο υπολογιστής θα είχε καθυστέρηση 10 φορές εξαιτίας της σελιδοποίησης ζήτησης.

Άσκηση Αυτοαξιολόγησης 5.5

Πόσος πρέπει να είναι ο ρυθμός σφαλμάτων σελίδας αν θέλουμε η καθυστέρηση λόγω σελιδοποίησης ζήτησης να είναι λιγότερη του 10%;

Απάντηση:

$$1,10 > 1 + 9999 \times p$$

$$0,10 > 9999 \times p$$

$$p < 0,00001$$

Δηλαδή, για να κρατήσουμε την καθυστέρηση που οφείλεται στη σελιδοποίηση σε ένα λογικό επίπεδο, τότε λιγότερες από μία προσπέλαση στη μνήμη ανά 100.000 προσπελάσεις θα πρέπει να μπορούν να δημιουργούν σφάλματα σελίδας.

Είναι πολύ σημαντικό να κρατιέται χαμηλός ο ρυθμός σφαλμάτων σελίδας σε ένα σύστημα με σελιδοποίηση ζήτησης. Στην αντίθετη περίπτωση, ο τελικός χρόνος προσπέλασης αυξάνεται, καθυστερώντας δραματικά την εκτέλεση των διαδικασιών.

Άσκηση Αυτοαξιολόγησης 5.6

Ένας Η/Υ εκτελεί μια εντολή σε 1μsec. Καθεμιά από αυτές τις εντολές κάνει 2 προσπελάσεις στη μνήμη. Το 90% των προσπελάσεων γίνονται μέσω συσχετιστικής μνήμης, και στην περίπτωση αυτή δεν έχουμε πρόσθετες καθυστερήσεις. Όμως, όταν μια προσπέλαση γίνει μέσω πίνακα σελίδων, τότε υπάρχει πρόσθετη καθυστέρηση 0,5μsec. Το κάθε λάθος σελίδας συνεπάγεται πρόσθετη καθυστέρηση 20msec. Προσδιορίστε το ρυθμό λαθών σελίδας έτσι ώστε ο μέσος χρόνος εκτέλεσης μιας εντολής να είναι μικρότερος από 1,2μsec.

Απάντηση:

Στην περίπτωση που δε συμβαίνουν λάθη σελίδων, έχουμε μέσο χρόνο εκτέλεσης κάθε εντολής ίσο με:

$$0,9 * 1\mu\text{sec} + 0,1 * (1 + 0,5) \mu\text{sec}$$

$$= (0,9 + 0,15) \mu\text{sec}$$

$$= (1,05) \mu\text{sec}$$

Έστω f ο ρυθμός με τον οποίο συμβαίνουν λάθη σελίδων. Τότε, ο μέσος χρόνος

εκτέλεσης εντολών είναι:

$$\begin{aligned} & 1,05 \text{ } \mu\text{sec} * (1-f) + (1,05 \text{ } \mu\text{sec} + 20 \text{ msec})f \\ & = 1,05 \text{ } \mu\text{sec} - 1,05f \text{ } \mu\text{sec} + 1,05f \text{ } \mu\text{sec} + 20000f \text{ } \mu\text{sec} \\ & = 20000f \text{ } \mu\text{sec} + 1,05 \text{ } \mu\text{sec} \end{aligned}$$

Θέλουμε ο μέσος χρόνος εκτέλεσης εντολών να είναι μικρότερος από 1,2μsec. Άρα

$$20000f \text{ } \mu\text{sec} + 1,05 \text{ } \mu\text{sec} < 1,2 \text{ } \mu\text{sec}$$

$$f < (1,2 - 1,05) / 20000 \text{ } \mu\text{sec}$$

$$f < 0,15 / 20000 \text{ } \mu\text{sec}].$$

5.4 Αντικατάσταση Σελίδας

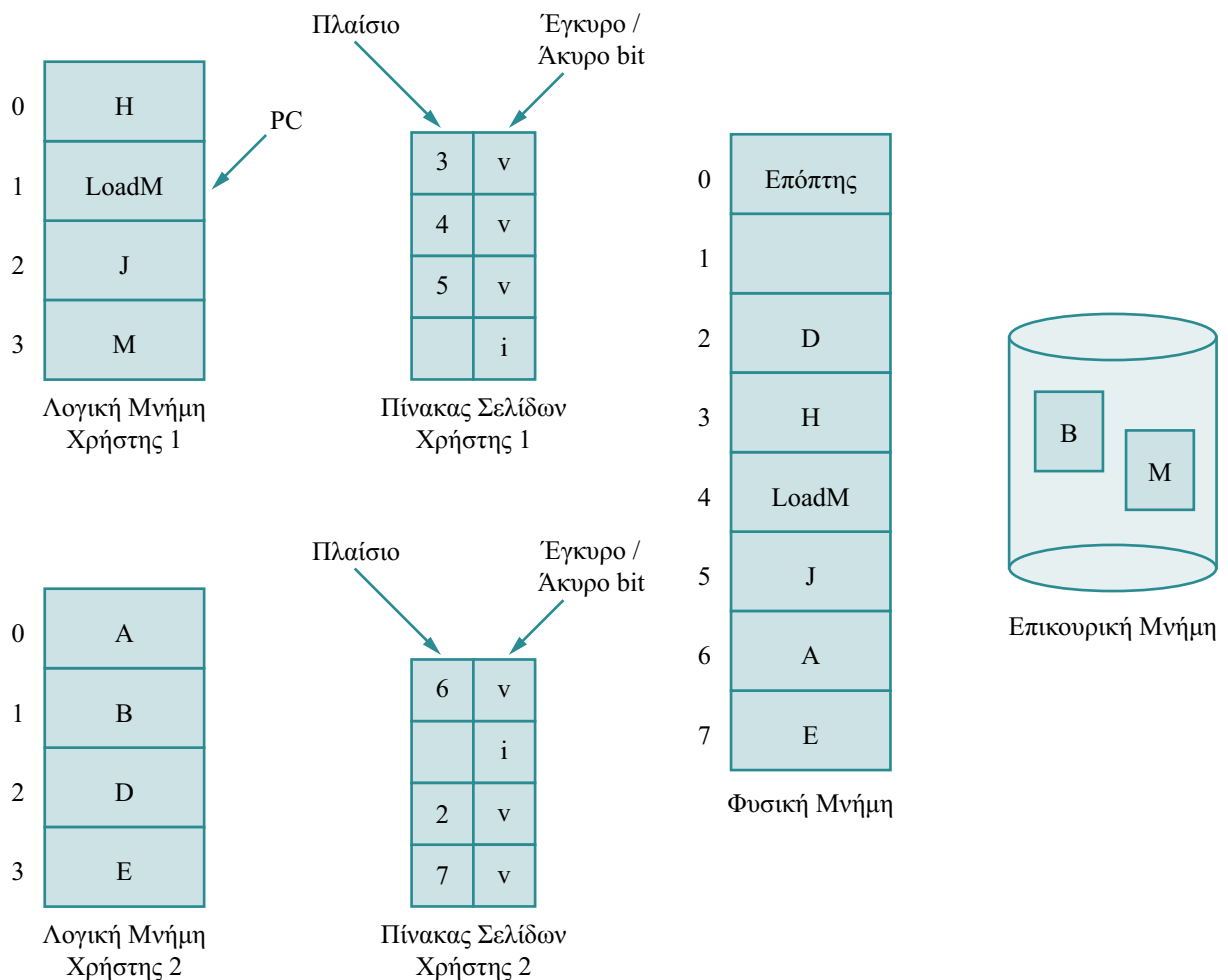
Στην παρουσίαση που κάναμε μέχρι τώρα, ο ρυθμός σφάλματος σελίδας δεν αποτελεί, πραγματικά, πρόβλημα, αφού υποθέσαμε ότι σε κάθε σελίδα αντιστοιχεί το πολύ ένα σφάλμα σελίδας, που συμβαίνει όταν γίνεται αναφορά σε αυτή για πρώτη φορά. Αυτή όμως η παρουσίαση δεν είναι ακριβής. Σκεφτείτε ότι, αν μια διαδικασία δέκα σελίδων σε μια πραγματική εκτέλεση χρησιμοποιεί μόνο τις μισές, τότε η σελιδοποίηση ζήτησης αποφεύγει την I/O που χρειάζεται για να φορτωθούν οι πέντε σελίδες που δε χρησιμοποιούνται ποτέ. Έτσι, θα μπορούσαμε να αυξήσουμε το βαθμό πολυπρογραμματισμού τρέχοντας διπλάσιο αριθμό διαδικασιών.

Για παράδειγμα, αν είχαμε σαράντα πλαίσια, θα μπορούσαμε να τρέξουμε οκτώ διαδικασίες αντί για τέσσερις, που θα μπορούσαν να τρέξουν αν η καθεμιά απαιτούσε δέκα πλαίσια (πέντε από τα οποία δε χρησιμοποιούνται ποτέ) και χρειαζόνταν αν βρίσκονταν εξολοκλήρου στη μνήμη.

Αν αυξήσουμε το βαθμό πολυπρογραμματισμού, τότε υπερκατανέμουμε τη μνήμη (ορίζεται στην επόμενη παράγραφο). Αν τρέξουμε έξι διαδικασίες, που η καθεμιά έχει δέκα σελίδες μέγεθος, αλλά στην πραγματικότητα χρησιμοποιεί μόνο τις πέντε, έχουμε υψηλότερη χρήση της CPU και «απόδοση» (throughput) και μας μένουν άδεια δέκα πλαίσια. Είναι όμως πιθανό καθεμιά από αυτές τις διαδικασίες, για ένα συγκεκριμένο σύνολο δεδομένων, να προσπαθήσει ξαφνικά να χρησιμοποιήσει και τις δέκα σελίδες του, με αποτέλεσμα να χρειαστούμε εξήντα πλαίσια, ενώ υπάρχουν μόνο σαράντα. Παρ' όλο που αυτή η κατάσταση φαίνεται απίθανη, γίνεται όλο και πιο πιθανή καθώς αυξάνουμε το επίπεδο πολυπρογραμματισμού, έτσι ώστε η μέση

χρήση μνήμης να είναι κοντά στη διαθέσιμη φυσική μνήμη. (Στο παράδειγμά μας, γιατί να σταματήσουμε σε ένα επίπεδο πολυπρογραμματισμού των έξι διαδικασιών, όταν μπορούμε να προχωρήσουμε σε ένα επίπεδο των επτά ή οκτώ;)

Η υπερκατανομή μνήμης θα εμφανιστεί με τον ακόλουθο τρόπο: Ενώ εκτελείται μια διαδικασία, συμβαίνει ένα σφάλμα σελίδας. Το υλικό προκαλεί μια παγίδευση του ΛΣ, το οποίο ελέγχει τους εσωτερικούς πίνακές του για να βεβαιωθεί ότι είναι σφάλμα σελίδας και όχι μια παράνομη προσπέλαση μνήμης. Το ΛΣ βρίσκει πού είναι η επιθυμητή σελίδα στην επικουρική μνήμη, αλλά μετά ανακαλύπτει ότι δεν υπάρχουν ελεύθερα πλαίσια στη λίστα των ελεύθερων πλαισίων (Σχήμα 5.5).



Σχήμα 5.5

Ανάγκη αντικατάστασης σελίδας

Το ΛΣ έχει αρκετές εναλλακτικές λύσεις σε αυτή την περίπτωση:

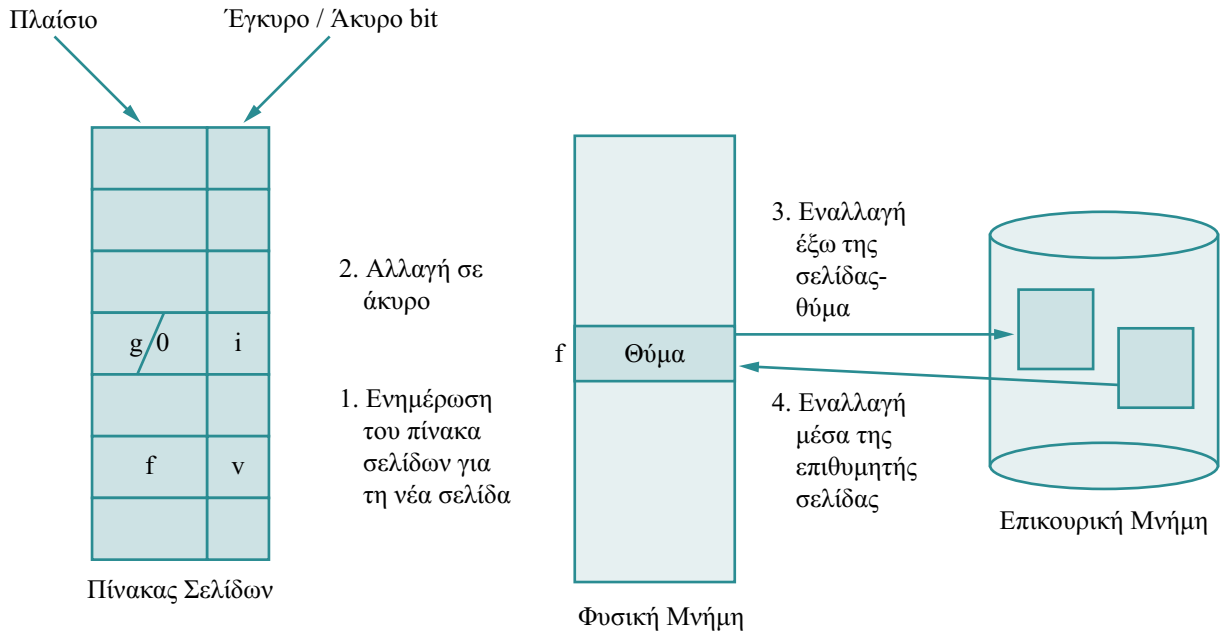
1) Θα μπορούσε να τερματίσει τη διαδικασία. Όμως, η σελιδοποίηση ζήτησης εφαρ-

μόζεται για να βελτιωθεί η χρήση και η απόδοση του υπολογιστικού συστήματος, χωρίς να γίνονται όλα αυτά αντιληπτά από το χρήστη.

2) Θα μπορούσε να «εναλλάξει έξω» μια διαδικασία, ελευθερώνοντας όλα τα πλαίσιά της και μειώνοντας το επίπεδο πολυπρογραμματισμού. Αυτή η λύση έχει πλεονεκτήματα και θα τη διερευνήσουμε στην Ενότητα 5.8.1, αλλά καταρχήν ας μελετήσουμε την «αντικατάσταση σελίδας» (page replacement).

3) Η αντικατάσταση σελίδας χρησιμοποιεί την ακόλουθη προσέγγιση: Αν δεν υπάρχει ελεύθερο πλαίσιο, βρίσκει ένα που δε χρησιμοποιείται και το ελευθερώνει. Ένα πλαίσιο ελευθερώνεται με την εγγραφή των περιεχομένων του στην επικουρική μνήμη και την αλλαγή του πίνακα σελίδων ώστε να δείχνει ότι η σελίδα δεν είναι πια στη μνήμη (Σχήμα 5.6). Το ελεύθερο πλαίσιο μπορεί τώρα να χρησιμοποιηθεί για να κρατήσει τη σελίδα για την οποία προκλήθηκε σφάλμα σελίδας από τη διαδικασία. Η ρουτίνα εξυπηρέτησης σφάλματος σελίδας, για να συμπεριλάβει την αντικατάσταση σελίδας, αλλάζει ως ακολούθως:

1. Βρες τη θέση τη ζητούμενης σελίδας στην επικουρική μνήμη.
2. Ψάξε για ένα ελεύθερο πλαίσιο.
 - (α) Αν υπάρχει ελεύθερο πλαίσιο, χρησιμοποίησέ το.
 - (β) Αλλιώς, χρησιμοποίησε έναν αλγόριθμο αντικατάστασης σελίδας για την επιλογή ενός «πλαισίου-θύματος» (victim frame).
 - (γ) Γράψε τη σελίδα-θύμα στην επικουρική μνήμη, άλλαξε τους πίνακες σελίδων και πλαισίων αντίστοιχα.
3. Διάβασε την επιθυμητή σελίδα και τοποθέτησέ την στο ελεύθερο πλαίσιο, άλλαξε τους πίνακες σελίδων και πλαισίων.
4. Επανεκκίνησε τη διαδικασία του χρήστη.

**Σχήμα 5.6**

Αντικατάσταση
σελίδας

Παρατηρήστε ότι, αν δεν υπάρχουν ελεύθερα πλαίσια, χρειάζονται δύο μεταφορές σελίδας (μία προς τα έξω και μία προς τα μέσα). Αυτή η κατάσταση διπλασιάζει το χρόνο εξυπηρέτησης σφάλματος σελίδας και θα αυξήσει, αντίστοιχα, τον τελικό χρόνο προσπέλασης.

Αυτό το κόστος μπορεί να μειωθεί με τη χρήση ενός bit ανά σελίδα που δείχνει αν η σελίδα έχει αλλάξει από τότε που ήρθε στη μνήμη. Αν δεν έχει αλλάξει και το πλαίσιο όπου βρίσκεται ζητηθεί από το ΛΣ, τότε η σελίδα δε χρειάζεται να γραφτεί στην επικουρική μνήμη, και ο χρόνος εξυπηρέτησης του σφάλματος σελίδας μειώνεται σημαντικά.

Άσκηση Αυτοαξιολόγησης 5.7

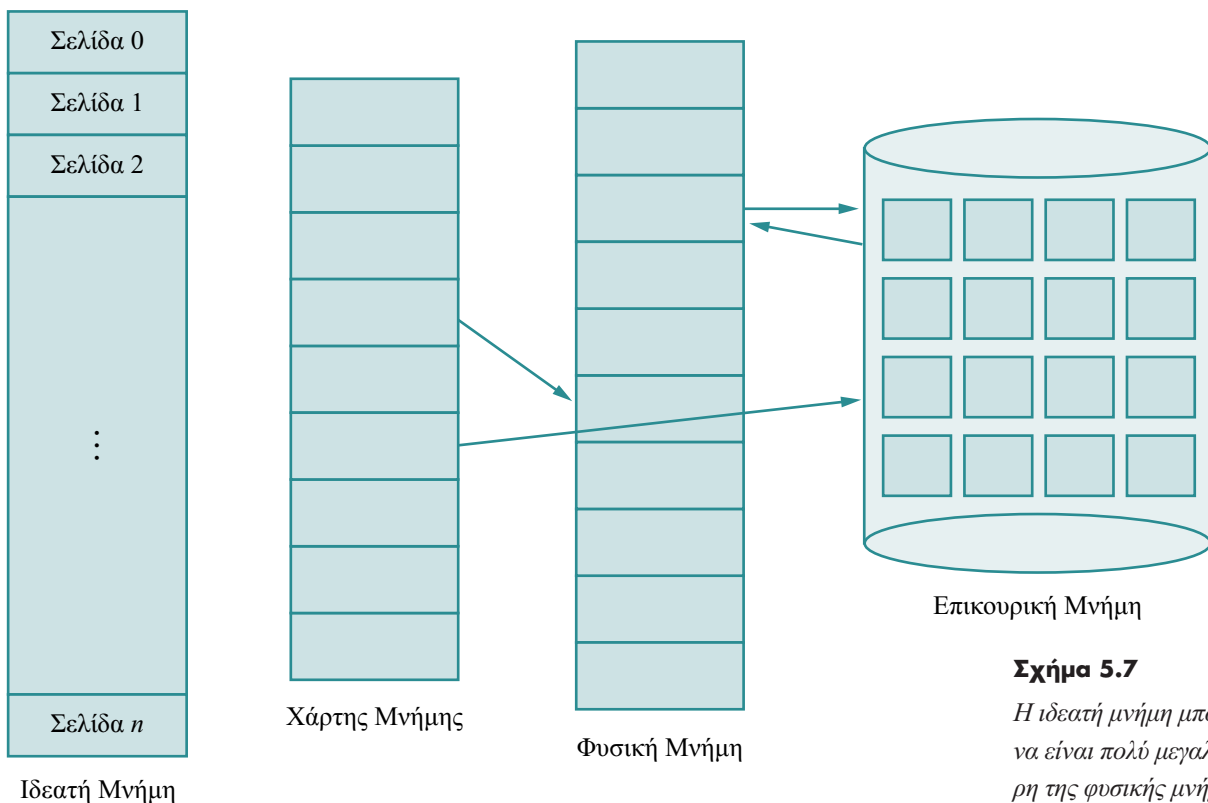
Γιατί το bit αυτό είναι γνωστό στα αγγλικά ως «dirty bit».

5.5 Έννοιες της Ιδεατής Μνήμης

Η αντικατάσταση σελίδων είναι βασικό στοιχείο της σελιδοποίησης ζήτησης. Ολοκληρώνει το διαχωρισμό μεταξύ λογικής και φυσικής μνήμης. Με απλή σελιδοποίηση οι διευθύνσεις του χρήστη απεικονίζονται σε ένα διαφορετικό σύνολο φυσικών διευθύνσεων. Με τη σελιδοποίηση ζήτησης το μέγεθος του χώρου λογικών διευ-

θύνσεων δεν περιορίζεται από τη φυσική μνήμη. Μια διαδικασία 20 σελίδων μπορεί να εκτελεστεί σε 10 πλαίσια, απλώς χρησιμοποιώντας σελιδοποίηση ζήτησης και έναν αλγόριθμο αντικατάστασης για να βρίσκει / δημιουργεί ένα ελεύθερο πλαίσιο οπότε χρειαστεί. Αν μια σελίδα πρόκειται να αντικατασταθεί, τα περιεχόμενά της γράφονται στην επικουρική μνήμη. Μια μετέπειτα αναφορά σε αυτή τη σελίδα θα προκαλέσει σφάλμα σελίδας. Τότε, η σελίδα θα επαναφερθεί στη μνήμη, ίσως αντικαθιστώντας κάποια άλλη σελίδα.

Η «ιδεατή μνήμη» είναι ο διαχωρισμός της λογικής μνήμης του χρήστη από τη φυσική μνήμη και συνήθως υλοποιείται με σελιδοποίηση ζήτησης. Με αυτό τον τρόπο μπορεί να προσφερθεί στους προγραμματιστές μια πολύ μεγάλη ιδεατή μνήμη που στηρίζεται σε μια μικρότερη φυσική μνήμη (Σχήμα 5.7). Η ιδεατή μνήμη κάνει το έργο του προγραμματισμού πολύ πιο εύκολο, αφού ο προγραμματιστής δε χρειάζεται πια να ανησυχεί για το ποσό της διαθέσιμης μνήμης, αλλά μπορεί να αφοσιωθεί στο πρόβλημα που έχει να προγραμματίσει. Η ιδεατή μνήμη αχρηστεύει τις επικαλύψεις.



Η ιδεατή μνήμη μπορεί να υλοποιηθεί και σε ένα *σύστημα τμηματοποίησης*. Αρκετά συστήματα, όπως το Multics, προσφέρουν ένα σύστημα σελιδοποίησης τμηματοποίησης, όπου τα «τμήματα» (segments) είναι χωρισμένα σε σελίδες. Με αυτό τον τρόπο, ο χρήστης νομίζει ότι έχει τμηματοποίηση, αλλά το ΛΣ μπορεί να την υλοποιεί μέσω σελιδοποίησης ζήτησης. Η «τμηματοποίηση ζήτησης» (demand segmentation) μπορεί επίσης να χρησιμοποιηθεί για την παροχή ιδεατής μνήμης – τα υπολογιστικά συστήματα Burroughs έχουν χρησιμοποιήσει αυτό τον τρόπο. Όμως οι αλγόριθμοι αντικατάστασης τμήματος είναι πιο πολύπλοκοι από τους αλγόριθμους αντικατάστασης σελίδας.

Δύο κύρια προβλήματα πρέπει να λυθούν για να υλοποιηθεί η σελιδοποίηση ζήτησης: ο αλγόριθμος κατανομής πλαισίων στις διάφορες διαδικασίες και ο αλγόριθμος αντικατάστασης σελίδας.

5.6 Αλγόριθμοι Αντικατάστασης Σελίδας

Υπάρχουν πολλοί αλγόριθμοι αντικατάστασης, και βασικό κριτήριο επιλογής είναι ο (χαμηλός) ρυθμός σφαλμάτων σελίδας που ο καθένας εξασφαλίζει.

Η αποτίμηση κάθε αλγόριθμου βασίζεται στον αριθμό των σφαλμάτων σελίδας που προκαλούν διάφορες ακολουθίες αναφορών μνήμης, που ονομάζονται «ακολουθίες αναφορών» (reference string). Οι ακολουθίες αναφορών μπορούν να παραχθούν τεχνητά (π.χ. από μια γεννήτρια τυχαίων αριθμών) ή παρακολουθώντας ένα σύστημα και αποθηκεύοντας τη διεύθυνση κάθε αναφοράς μνήμης. Η τελευταία αυτή επιλογή παράγει ένα μεγάλο όγκο δεδομένων (της τάξης του ενός εκατομμυρίου διευθύνσεων ανά sec). Για να μειώσουμε τον όγκο των δεδομένων παρατηρούμε δύο πράγματα:

Πρώτον, για δεδομένο μέγεθος σελίδας (ορισμένο από το υλικό ή το ΛΣ) χρειάζεται να λάβουμε υπόψη μας μόνο τον αριθμό σελίδας και όχι ολόκληρη τη διεύθυνση.

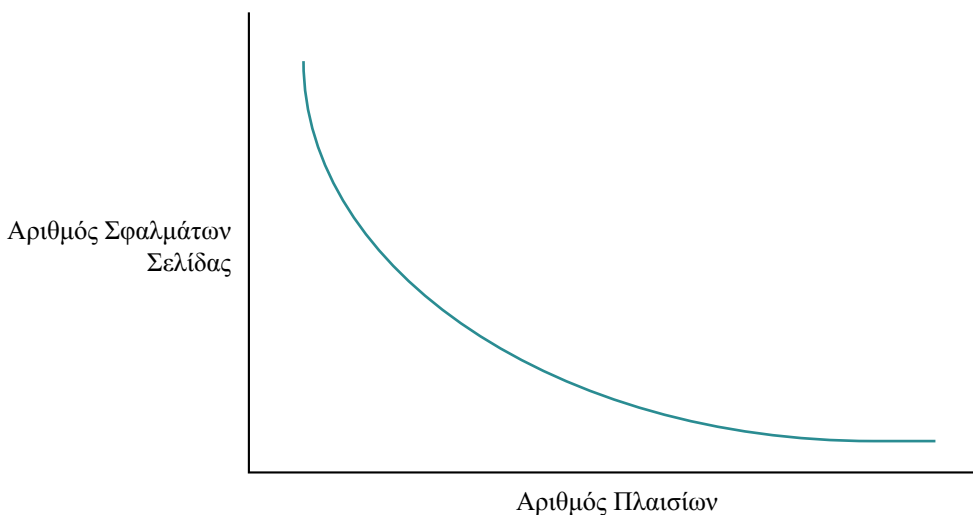
Δεύτερον, αν έχουμε μια αναφορά στη σελίδα p , τότε οποιαδήποτε αμέσως ακολουθούσα αναφορά (ή αναφορές) στη σελίδα p δε θα προκαλέσουν ποτέ σφάλμα σελίδας. Η σελίδα p μετά την πρώτη αναφορά θα βρίσκεται στη μνήμη, οι αμέσως ακόλουθες αναφορές δε θα προκαλέσουν σφάλμα σελίδας. Για παράδειγμα, αν παρακολουθήσουμε κάποια διαδικασία, μπορεί να έχουμε την ακόλουθη σειρά διευθύνσεων:

0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101, 0611, 0102, 0103,
0104, 0101, 0610, 0102, 0103, 0104, 0101, 0609, 0102, 0105

η οποία με 100 λέξεις ανά σελίδα μειώνεται στην παρακάτω ακολουθία αναφορών:

1,4,1,6,1,6,1,6,1,6,1

Για να καθορίσουμε τον αριθμό των σφαλμάτων σελίδας για μια συγκεκριμένη ακολουθία αναφορών και για έναν αλγόριθμο αντικατάστασης σελίδων, χρειαζόμαστε να ξέρουμε και τον αριθμό των διαθέσιμων πλαισίων σελίδων. Προφανώς, καθώς αυξάνεται ο αριθμός των διαθέσιμων πλαισίων, ο αριθμός των σφαλμάτων σελίδας θα μειώνεται. Για παράδειγμα, για την πιο πάνω ακολουθία αναφορών, αν είχαμε τρία ή περισσότερα πλαίσια, θα είχαμε μόνο τρία σφάλματα, ένα σφάλμα για την πρώτη αναφορά στην κάθε σελίδα. Από την άλλη μεριά, με διαθέσιμο μόνο ένα πλαίσιο, θα είχαμε μια αντικατάσταση για κάθε αναφορά, με αποτέλεσμα να δημιουργηθούν 11 σφάλματα σελίδας. Γενικά, περιμένουμε μια καμπύλη όπως αυτή του Σχήματος 5.8.



Σχήμα 5.8

Γραφική παράσταση των σφαλμάτων σελίδας έναντι του αριθμού πλαισίων

Για να προσδιορίσουμε τον αριθμό σφαλμάτων στα παραδείγματα που ακολουθούν, θεωρούμε ότι υπάρχει μια μνήμη με τρία πλαίσια και χρησιμοποιούμε την εξής ακολουθία αναφορών:

7,0,1,1,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1.

5.6.1 FIFO

Ο πιο απλός αλγόριθμος αντικατάστασης σελίδων είναι ο «First-In-First-Out» (FIFO), ο οποίος αντικαθιστά κάθε φορά που χρειάζεται την «αρχαιότερη» σελίδα στη μνήμη. Για την υλοποίησή του αρκεί μια ουρά με όλες τις σελίδες στη μνήμη στη σειρά που έρχονται

Για την ακολουθία αναφορών που χρησιμοποιούμε ως παράδειγμα, τα τρία πλαίσια είναι αρχικά άδεια (βλ. Σχήμα 5.9). Οι πρώτες τρεις αναφορές (7,0,1) δημιουργούν σφάλματα σελίδας και θα εισαχθούν σε αυτά τα άδεια πλαίσια. Η επόμενη αναφο-

ρά (2) αντικαθιστά τη σελίδα 7, επειδή αυτή είχε έρθει πρώτη. Εφόσον η 0 είναι η επόμενη αναφορά και η 0 είναι ήδη στη μνήμη, δεν υπάρχει σφάλμα σελίδας. Η πρώτη αναφορά στη σελίδα 3 έχει ως αποτέλεσμα την αντικατάσταση της σελίδας 0, επειδή ήταν η πρώτη από τις τρεις σελίδες στη μνήμη (0,1 και 2) που εισήχθη. Αυτή η αντικατάσταση σημαίνει ότι η επόμενη αναφορά στην 0 θα προκαλέσει σφάλμα σελίδας. Τότε, η σελίδα 1 θα αντικατασταθεί από τη σελίδα 0. Αυτή η διαδικασία συνεχίζεται όπως φαίνεται στο Σχήμα 5.9. Κάθε φορά που συμβαίνει σφάλμα σελίδας δείχνουμε ποιες σελίδες βρίσκονται στα τρία πλαίσια. Υπάρχουν συνολικά 15 σφάλματα σελίδας.

Ακολουθία Αναφορών

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

Σχήμα 5.9
Αντικατάσταση
σελίδων FIFO

7	7	7	2	2	2	4	4	4	0	0	0	7	7	7
	0	0	0	3	3	3	2	2	2	1	1	1	0	0
		1	1	1	0	0	0	3	3	3	2	2	2	1

Ο FIFO αλγόριθμος αντικατάστασης σελίδων είναι εύκολο να κατανοηθεί και να προγραμματιστεί. Αλλά η απόδοσή του δεν είναι πάντοτε καλή. Η σελίδα που αντικαταστάθηκε μπορεί να είναι κώδικας αρχικοποίησης που χρησιμοποιήθηκε πριν από πολύ καιρό και δε χρειάζεται πια. Μπορεί όμως να περιέχει μια συχνά χρησιμοποιούμενη μεταβλητή που αρχικοποιήθηκε νωρίς και είναι σε συνεχή χρήση.

Άσκηση Αυτοαξιολόγησης 5.8

Έστω ότι η σελίδα-θύμα είναι ενεργός. (α) Τι θα πει αυτό; (β) Τι επιπτώσεις έχει στην απόδοση του αλγόριθμου; (γ) Τι επιπτώσεις έχει στην ορθότητα του αλγόριθμου;

Απάντηση:

Παρατηρήστε ότι, ακόμα και εάν διαλέξουμε για αντικατάσταση μια σελίδα που είναι ενεργός (δηλαδή εξακολουθούν να γίνονται αναφορές σε διευθύνσεις που ανήκουν σε αυτή τη σελίδα), όλα δουλεύουν σωστά. Αφού βγάζουμε μια ενεργό σελίδα για να εισαγάγουμε μια καινούρια, σχεδόν αμέσως σφάλουμε για την ενεργό σελίδα. Κάποια άλλη σελίδα θα χρειαστεί να αντικατασταθεί, έτσι ώστε να έρθει η ενεργός σελίδα ξανά στη μνήμη. Έτσι, μια άσχημη επιλογή αντικατάστασης

αυξάνει το ρυθμό σφαλμάτων σελίδας και καθυστερεί την εκτέλεση της διαδικασίας, αλλά δεν προκαλεί εσφαλμένη εκτέλεση.

Άσκηση Αυτοαξιολόγησης 5.9

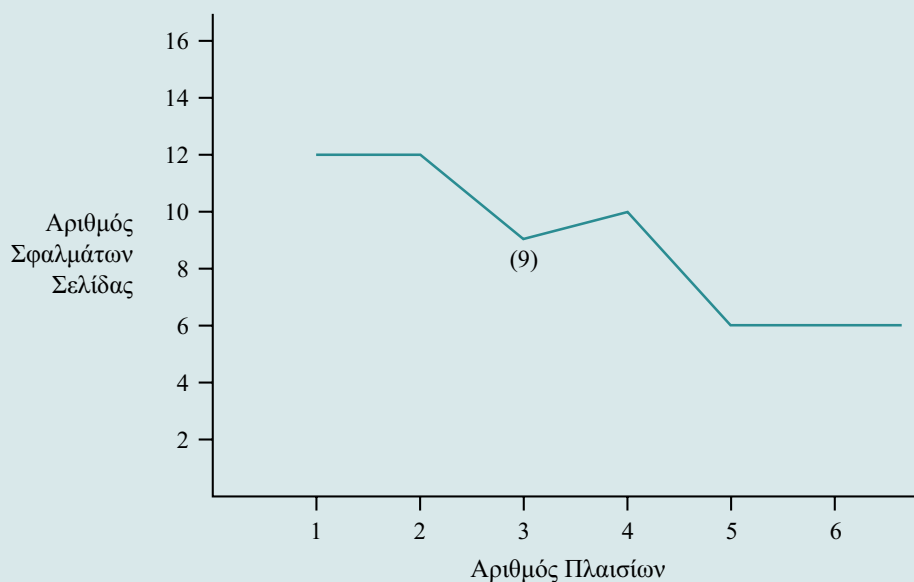
Για να σκιαγραφήσουμε τα πιθανά προβλήματα ενός FIFO αλγόριθμου, θεωρήστε την ακολουθία αναφορών:

1,2,3,4,1,2,51,2,3,4,5

Σχεδιάστε την καμπύλη των σφαλμάτων σελίδας έναντι του αριθμού των διαθέσιμων πλαισίων. Τι γίνεται αν έχουμε 3 πλαίσια και τα αυξήσουμε κατά 1;

Απάντηση:

Το Σχήμα 5.10 δείχνει την καμπύλη των σφαλμάτων σελίδας έναντι του αριθμού των διαθέσιμων πλαισίων. Παρατηρούμε ότι ο αριθμός των λαθών για τέσσερα πλαίσια (10) είναι μεγαλύτερος του αριθμού των λαθών για τρία πλαίσια (9). Αυτό το αναπάντεχο αποτέλεσμα είναι γνωστό ως «ανωμαλία του Belady» (Belady's anomaly). Αντικατοπτρίζει δε το γεγονός ότι για μερικούς αλγόριθμους αντικατάστασης σελίδων ο ρυθμός σφαλμάτων σελίδας μπορεί να αυξηθεί παρ' ότι αυξάνεται ο αριθμός των πλαισίων. Ενώ κάποιος θα περίμενε ότι η απόδοση μιας διαδικασίας θα βελτιωνόταν δίνοντάς του πιο πολύ μνήμη, παρατηρήθηκε σε κάποια αρχική έρευνα ότι αυτή η υπόθεση δεν ήταν πάντοτε αληθής. Ως αποτέλεσμα ανακαλύφθηκε η ανωμαλία του Belady.



Σχήμα 5.10

Καμπύλη σφαλμάτων σελίδας για FIFO αντικατάσταση σε μια ακολουθία αναφορών

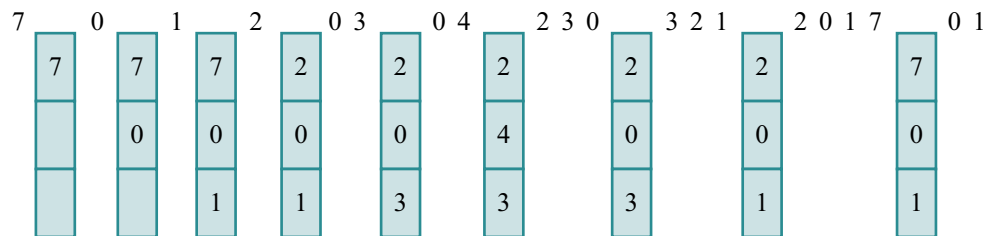
5.6.2 Βέλτιστη Αντικατάσταση (Optimal Replacement)

Ποιος είναι ο (θεωρητικά) βέλτιστος αλγόριθμος αντικατάστασης σελίδας; Θα πρέπει, αφενός, να έχει το χαμηλότερο ρυθμό σφαλμάτων σελίδας από όλους τους αλγόριθμους και ποτέ να μην παρουσιάζει την ανωμαλία του Belady. Υπάρχει ένας τέτοιος «βέλτιστος αλγόριθμος» (optimal algorithm) και ονομάζεται OPTIMAL ή MIN. Λέει δε απλά ότι: «Αντικατάστησε αυτή τη σελίδα που δε θα χρησιμοποιηθεί για τη μεγαλύτερη χρονική περίοδο». Αυτό, βέβαια, προϋποθέτει «γνώση του μέλλοντος».

Η χρήση αυτού του αλγόριθμου εγγυάται τον ελάχιστο δυνατό ρυθμό σφαλμάτων σελίδας για ένα σταθερό αριθμό πλαισίων.

Για παράδειγμα, για τη συμβολοσειρά αναφορών που χρησιμοποιήσαμε προηγουμένως ο βέλτιστος αλγόριθμος θα έδινε εννέα σφάλματα σελίδας, όπως φαίνεται στο Σχήμα 5.11. Οι πρώτες τρεις αναφορές παράγουν σφάλματα τα οποία γεμίζουν τα τρία άδεια πλαίσια. Η αναφορά στη σελίδα 2 αντικαθιστά τη σελίδα 7, επειδή η 7 δε θα χρησιμοποιηθεί μέχρι την αναφορά 18, ενώ η σελίδα 0 θα χρησιμοποιηθεί στην 5 και η σελίδα 1 στη 14. Η αναφορά στη σελίδα 3 αντικαθιστά τη σελίδα 1, αφού η 1 είναι η τελευταία από τις τρεις σελίδες στη μνήμη που θα αναφερθεί ξανά. Με μόνο εννέα σφάλματα σελίδας, η βέλτιστη αντικατάσταση είναι πολύ καλύτερη από το FIFO, το οποίο είχε δεκαπέντε σφάλματα. (Αν αγνοήσουμε τα πρώτα τρία που κάνει κάθε αλγόριθμος, τότε ο FIFO είναι δύο φορές χειρότερος από τον βέλτιστο. Γενικά, είναι k φορές χειρότερος, όπου k ο αριθμός των πλαισίων σελίδας). Καθένας αλγόριθμος αντικατάστασης δεν μπορεί να επεξεργαστεί αυτή τη συμβολοσειρά αναφορών σε τρία πλαίσια με λιγότερα από εννέα σφάλματα. (Η απόδειξη αυτού είναι θέμα που υπερβαίνει τα πλαίσια αυτού του βιβλίου.)

Σχήμα 5.11
Βέλτιστη αντικατάσταση σελίδας



Ο βέλτιστος αλγόριθμος δεν είναι εφαρμόσιμος, αφού προϋποθέτει μελλοντική γνώση της συμβολοσειράς αναφορών (θα συναντήσουμε μια παρόμοια περίπτωση με τον αλγόριθμο χρονοδρομολόγησης της CPU στη «συντομότερη εργασία πρώτη» (Shortest–Job–First), τον οποίο θα διδαχθείτε στα Λειτουργικά Συστήματα II). Ο βέλτιστος αλγόριθμος χρησιμοποιείται κυρίως για συγκριτικές μελέτες.

5.6.3 LRU (Least Recently Used)

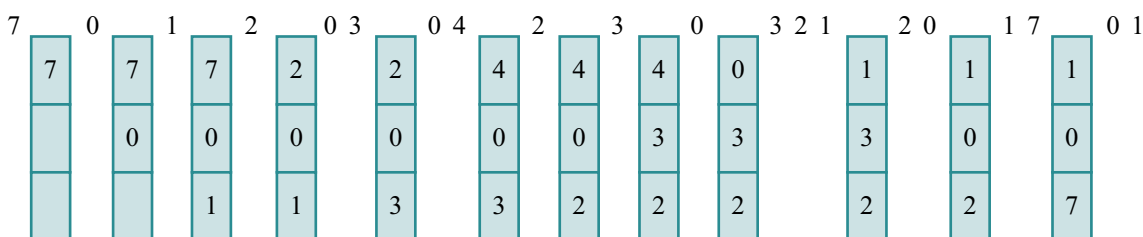
Αν ο βέλτιστος αλγόριθμος δεν είναι υλοποιήσιμος, πιθανότατα μπορούμε να πετύχουμε μια προσέγγισή του. Η βασική διαφορά μεταξύ του FIFO και του OPTIMAL (πέραν του να κοιτάζουν μπρος ή πίσω στο χρόνο) είναι ότι ο FIFO χρησιμοποιεί το χρόνο που μια σελίδα ήρθε στη μνήμη, ενώ ο OPTIMAL χρησιμοποιεί το χρόνο που μια σελίδα πρόκειται να χρησιμοποιηθεί. Αν χρησιμοποιήσουμε το παρελθόν ως μια προσέγγιση του κοντινού μέλλοντος, τότε θα αντικαθιστούσαμε τη σελίδα που δεν έχει χρησιμοποιηθεί για τη μεγαλύτερη χρονική περίοδο (Σχήμα 5.12). Αυτός είναι ο αλγόριθμος της «λιγότερο πρόσφατα χρησιμοποιούμενης σελίδας» (Least Recently Used – LRU).

Η LRU αντικατάσταση συσχετίζει με κάθε σελίδα το χρόνο της τελευταίας της χρήσης. Όταν πρέπει να αντικατασταθεί μια σελίδα, ο LRU διαλέγει εκείνη τη σελίδα η οποία δεν έχει χρησιμοποιηθεί για τη μεγαλύτερη χρονική περίοδο. Αυτή είναι η περίπτωση του βέλτιστου αλγόριθμου που κοιτάζει πίσω στο χρόνο αντί για μπροστά.

Άσκηση Αυτοαξιολόγησης 5.10

Έστω μια οποιαδήποτε συμβολοσειρά. Πρώτον, παρατηρήστε και, δεύτερον, εξηγήστε γιατί ο αριθμός σφαλμάτων που παράγει ο LRU στην αντίστροφη συμβολοσειρά είναι ο ίδιος με τον αριθμό σφαλμάτων που παράγει ο OPTIMAL στη συμβολοσειρά.

Το αποτέλεσμα της εφαρμογής του LRU πάνω στη συμβολοσειρά αναφορών που χρησιμοποιούμε για παράδειγμα φαίνεται στο Σχήμα 5.12. Ο LRU παράγει 12 λάθη.



Σχήμα 5.12

Least Recently Used αντικατάστασης σελίδας

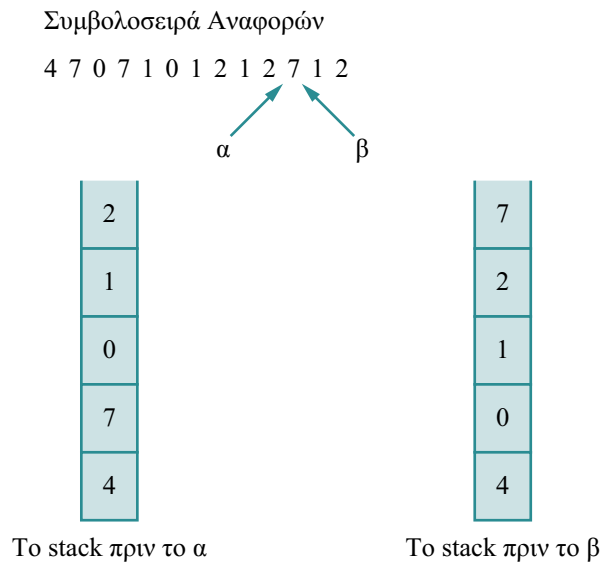
Παρατηρήστε ότι τα πέντε πρώτα είναι ίδια με αυτά του βέλτιστου αλγόριθμου. Όταν, όμως, γίνεται η αναφορά στη σελίδα 4, ο LRU βλέπει ότι, από τα τρία πλαίσια στη μνήμη, η σελίδα 2 είναι αυτή που δεν έχει χρησιμοποιηθεί για τη μεγαλύτερη χρονική περίοδο. Η πιο πρόσφατα χρησιμοποιημένη σελίδα είναι η 0 και αμέσως

πιο πριν από αυτή χρησιμοποιήθηκε η σελίδα 3. Έτσι, ο LRU αντικαθιστά τη σελίδα 2 χωρίς να ξέρει ότι πρόκειται να ξαναχρησιμοποιηθεί. Όταν σφάλει για τη σελίδα 2, ο LRU αντικαθιστά τη σελίδα 3, αφού από τις τρεις σελίδες στη μνήμη 0,3,4 η σελίδα 3 είναι η λιγότερο πρόσφατα χρησιμοποιημένη. Παρ' όλα αυτά τα προβλήματα, ο LRU με 12 σφάλματα είναι ακόμα πολύ καλύτερος από τον FIFO που έχει 15 σφάλματα.

Ο LRU χρησιμοποιείται συχνά ως αλγόριθμος αντικατάστασης σελίδας και μπορεί να χρειάζεται σημαντική υποστήριξη από το υλικό. Το πρόβλημα έγκειται στον καθορισμό μιας σειράς για τα πλαίσια, που ορίζεται με βάση το χρόνο τελευταίας χρήσης. Δύο υλοποιήσεις είναι αποδοτικές.

Μετρητές (Counters). Στην απλούστερη περίπτωση, συσχετίζουμε με κάθε εγγραφή στον πίνακα σελίδων έναν καταχωρητή που κρατάει το χρόνο χρήσης και προσθέτουμε στην CPU ένα λογικό ρολόι ή μετρητή. Οποτεδήποτε γίνεται αναφορά σε μια σελίδα, τα περιεχόμενα του «καταχωρητή-ρολογιού» (clock register) αντιγράφονται στον καταχωρητή του χρόνου χρήσης του πίνακα σελίδων, γι' αυτή τη σελίδα. Με αυτό τον τρόπο πάντοτε έχουμε το «χρόνο» της τελευταίας αναφοράς για κάθε σελίδα. Αντικαθιστούμε δε τη σελίδα με τη μικρότερη χρονική τιμή. Αυτό το σχήμα απαιτεί ένα ψάξιμο του πίνακα σελίδων για να βρεθεί η λιγότερο πρόσφατα χρησιμοποιημένη σελίδα. Επίσης, οι χρόνοι πρέπει να διατηρούνται όταν αλλάζουν οι πίνακες σελίδων (εξαιτίας της χρονοδρομολόγησης της CPU). Επίσης, πρέπει να ληφθεί υπόψη και η υπερχειλίση του ρολογιού.

Στοιβά (Stack). Μια άλλη προσέγγιση στην υλοποίηση του LRU είναι η διατήρηση ενός Stack από αριθμούς σελίδων. Οποτεδήποτε γίνεται αναφορά σε μια σελίδα, αυτή αφαιρείται από το stack και τοποθετείται στην κορυφή. Με αυτό τον τρόπο, στην κορυφή του stack είναι πάντοτε η πιο πρόσφατα χρησιμοποιημένη σελίδα και στο κάτω μέρος του stack είναι η λιγότερο πρόσφατα χρησιμοποιημένη σελίδα (Σχήμα 5.13). Εφόσον πρέπει να αφαιρούμε εγγραφές από τη μέση του stack, αυτό υλοποιείται καλύτερα ως μια διπλά διασυνδεδεμένη λίστα με ένα δείκτη αρχής (head) και έναν τέλους (tail). Η αφαίρεση μιας σελίδας και η τοποθέτησή της στην κορυφή του stack απαιτεί, στη χειρότερη περίπτωση, την αλλαγή έξι δεικτών. Κάθε ενημέρωση είναι λίγο πιο ακριβή, αλλά δε χρειάζεται ψάξιμο κατά την αντικατάσταση, ο δείκτης τέλους δείχνει στο κάτω μέρος του stack, όπου βρίσκεται η λιγότερο πρόσφατα χρησιμοποιημένη σελίδα. Αυτή η προσέγγιση είναι ιδιαίτερη κατάλληλη για υλοποίηση του LRU με λογισμικό ή «μικροκώδικα» (microcode).

**Σχήμα 5.13**

Χρήση ενός stack για την καταγραφή των πιο πρόσφατων αναφορών σελίδας

Ούτε η βέλτιστη αντικατάσταση ούτε η LRU αντικατάσταση υποφέρουν από την ανωμαλία του Belady. Υπάρχει μια τάξη αλγόριθμων αντικατάστασης σελίδας, που ονομάζονται «αλγόριθμοι στοίβας» (stack algorithms), που δεν εμφανίζουν ποτέ την ανωμαλία Belady. Ένας αλγόριθμος στοίβας είναι αυτός για τον οποίο μπορεί να αποδειχθεί ότι το σύνολο των σελίδων στη μνήμη για n πλαίσια είναι πάντοτε ένα υποσύνολο του συνόλου των σελίδων που θα ήταν στη μνήμη για $n+1$ πλαίσια. Για τον LRU, το σύνολο των σελίδων στη μνήμη θα ήταν οι n πιο πρόσφατα αναφερθείσες σελίδες. Αν ο αριθμός των πλαισίων αυξηθεί, αυτές οι n σελίδες συνεχίζουν να είναι οι πιο πρόσφατα αναφερθείσες, και έτσι θα είναι ακόμα στη μνήμη.

Παρατηρήστε ότι καμία υλοποίηση του LRU δε θα ήταν δυνατή χωρίς βοήθεια από το υλικό. Η ενημέρωση των καταχωρητών ρολογιού ή της στοίβας πρέπει να γίνεται για κάθε αναφορά μνήμης. Αν χρησιμοποιούσαμε μια διακοπή ώστε να επιτρέπουμε στο λογισμικό να ενημερώνει τέτοιες δομές δεδομένων, θα υπήρχε καθυστέρηση της κάθε αναφοράς μνήμης κατά έναν παράγοντα τουλάχιστον ίσο με 10, έτσι κάθε διαδικασία θα είχε ίση καθυστέρηση. Ελάχιστα συστήματα θα μπορούσαν να ανεχθούν αυτό το επίπεδο κόστους για τη διαχείριση της μνήμης.

5.6.4 Προσέγγιση του LRU

Λίγα συστήματα παρέχουν ικανοποιητική υποστήριξη από το υλικό για πραγματική LRU αντικατάσταση σελίδων. Μερικά συστήματα δεν προσφέρουν τέτοια υποστήριξη, και τότε πρέπει να γίνει χρήση άλλων αλγόριθμων αντικατάστασης σελίδας (όπως ο FIFO). Πολλά όμως συστήματα προσφέρουν κάποια βοήθεια με τη μορφή

του «bit αναφοράς» (reference bit). Το bit αναφοράς για μια σελίδα τίθεται από το υλικό όταν γίνει αναφορά στη σελίδα (είτε εγγραφή είτε ανάγνωση ενός byte ή λέξης στη σελίδα). Τα bits αναφοράς συσχετίζονται με κάθε εγγραφή του πίνακα σελίδων ή ως ξεχωριστός καταχωρητής με ένα bit ανά πλαίσιο. Ειδικές εντολές είναι διαθέσιμες για την ανάγνωση και την επαναφορά (clearing) αυτών των bits.

Αρχικά όλα τα bits μηδενίζονται από το λειτουργικό σύστημα. Καθώς εκτελείται μια διαδικασία, το bit που συσχετίζεται με κάθε αναφερόμενη σελίδα τίθεται (παίρνει την τιμή 1) από το υλικό. Μετά από κάποιο χρόνο μπορούμε να προσδιορίσουμε ποιες σελίδες χρησιμοποιήθηκαν και ποιες όχι, εξετάζοντας τα bits αναφοράς. Δεν ξέρουμε τη σειρά χρήσης, αλλά ξέρουμε ποιες χρησιμοποιήθηκαν και ποιες όχι. Αυτή η πληροφορία μερικής διάταξης οδηγεί σε αλγόριθμους αντικατάστασης σελίδας που προσεγγίζουν τη LRU αντικατάσταση.

5.6.5 Πρόσθετα Bits Αναφοράς

Πρόσθετη πληροφορία διάταξης μπορεί να αποκτηθεί με την καταγραφή των bits αναφοράς κατά τακτά διαστήματα. Μπορούμε να κρατάμε ένα byte των 8 bits για κάθε σελίδα σ' έναν πίνακα στη μνήμη. Κατά τακτά διαστήματα (π.χ. κάθε 100ms), μια «διακοπή χρονομετρητή» (timer interrupt) μεταφέρει τον έλεγχο στο ΛΣ. Το ΛΣ «ολισθαίνει» (shifts) το bit αναφοράς για κάθε σελίδα μέσα στο υψηλής τάξης bit του byte των 8 bits, ολισθαίνει δεξιά τα άλλα bits κατά ένα bit και απορρίπτει τα χαμηλής τάξης bits. Αυτοί οι καταχωρητές ολίσθησης των 8 bits περιέχουν την ιστορία της χρήσης σελίδων για τις τελευταίες οκτώ χρονικές περιόδους. Αν ο καταχωρητής ολίσθησης περιέχει 00000000, τότε η σελίδα δεν έχει χρησιμοποιηθεί για οκτώ χρονικές περιόδους – μια σελίδα που έχει χρησιμοποιηθεί τουλάχιστον μια φορά σε κάθε περίοδο θα έχει την τιμή 11111111 στον καταχωρητή ολίσθησης.

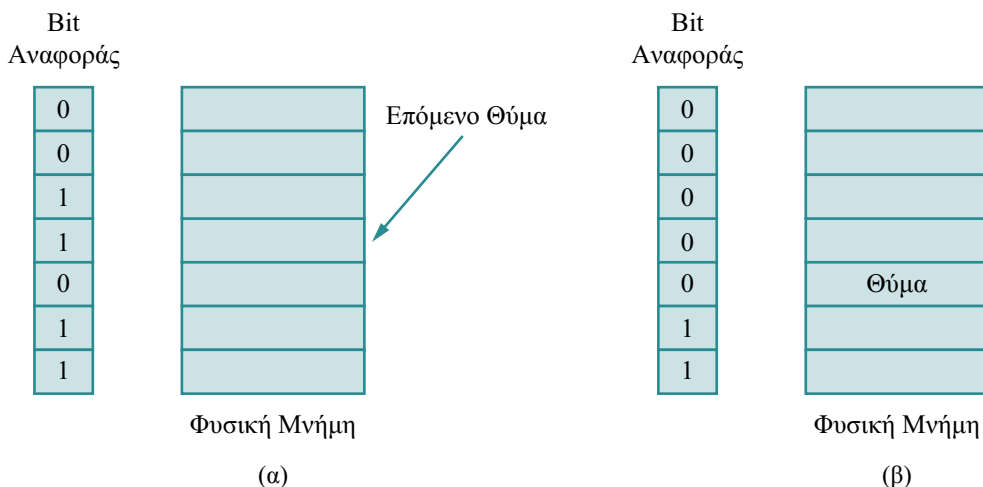
Μια σελίδα με τιμή καταχωρητή 11000100 έχει χρησιμοποιηθεί πιο πρόσφατα από μια τιμή 01110111. Αν μεταφράσουμε αυτά τα bytes των 8 bits ως ακραίους χωρίς πρόσημο, η σελίδα με το μικρότερο αριθμό είναι η λιγότερο πρόσφατα χρησιμοποιημένη και μπορεί να αντικατασταθεί. Παρατηρήστε ότι οι αριθμοί δεν είναι εγγυημένο ότι θα είναι μοναδικοί. Μπορούμε είτε να αντικαταστήσουμε («εναλλάξουμε έξω») όλες τις σελίδες με τη μικρότερη τιμή ή να χρησιμοποιήσουμε διαλογή FIFO αναμεταξύ τους.

Ο αριθμός των «bits ιστορίας» (history bits) μπορεί να παίρνει διάφορες τιμές, πρέπει όμως να διαλεχτεί έτσι ώστε να κάνει την ενημέρωση όσο πιο γρήγορη γίνεται. Στην πιο ακραία περίπτωση, μπορεί να μειωθεί στο μηδέν, αφήνοντας μόνο το bit αναφοράς. Αυτός ο αλγόριθμος ονομάζεται «αλγόριθμος της δεύτερης ευκαιρίας».

5.6.6 Αντικατάσταση Δεύτερης Ευκαιρίας

Ο βασικός «αλγόριθμος Δεύτερης Ευκαιρίας» (second chance replacement) είναι ένας FIFO αλγόριθμος αντικατάστασης. Όταν μια σελίδα αναφερθεί, ελέγχουμε το bit αναφοράς της. Αν είναι 0, αντικαθιστούμε αυτή τη σελίδα. Αν το bit αναφοράς είναι 1, δίνουμε σε αυτή τη σελίδα μια δεύτερη ευκαιρία και προχωρούμε για να επιλέξουμε την επόμενη FIFO σελίδα. Όταν μια σελίδα παίρνει μια δεύτερη ευκαιρία, το bit αναφοράς της μηδενίζεται και ο χρόνος άφιξης της ανανεώνεται στην τρέχουσα ώρα. Γι' αυτό μια σελίδα στην οποία δίνεται μια δεύτερη ευκαιρία δε θα αντικατασταθεί έως ότου όλες οι άλλες σελίδες αντικατασταθούν (ή δοθούν δεύτερες ευκαιρίες). Επιπλέον, αν μια σελίδα χρησιμοποιείται αρκετά συχνά ώστε να διατηρήσει το bit αναφοράς 1, δε θα αντικατασταθεί ποτέ.

Ένας τρόπος να φανταστεί κανείς τον αλγόριθμο δεύτερης ευκαιρίας είναι ως μια κυκλική ουρά. Ένας δείκτης δείχνει ποια σελίδα πρόκειται να αντικατασταθεί. Όταν χρειάζεται ένα πλαίσιο, ο δείκτης προχωρά μέχρι να βρει μια σελίδα που έχει το bit αναφοράς ίσο με το μηδέν. Καθώς προχωρά, επαναφέρει όλα τα bits αναφοράς (Σχήμα 5.14). Στη χειρότερη περίπτωση, όταν όλα τα bits αναφοράς είναι 1, ο δείκτης κυκλικά διατρέχει όλη την ουρά δίνοντας σε κάθε σελίδα μια δεύτερη ευκαιρία. Επαναφέρει δε όλα τα bits αναφοράς πριν να διαλέξει την επόμενη προς αντικατάσταση σελίδα. Αν όλα τα bits είναι 1, ο αλγόριθμος δεύτερης ευκαιρίας λειτουργεί σαν τον FIFO.



Σχήμα 5.14

Δεύτερης ευκαιρίας αντικατάσταση σελίδας

5.6.7 Ο Αλγόριθμος της Λιγότερο Συχνά Χρησιμοποιούμενης Σελίδας

Ο αλγόριθμος της «λιγότερο συχνά χρησιμοποιούμενης σελίδας» (Least Frequently Used – LFU) κρατάει ένα μετρητή του αριθμού των αναφορών που έχουν γίνει σε

κάθε σελίδα. Η σελίδα με το μικρότερο μετρητή αντικαθίσταται. Το κίνητρο αυτής της επιλογής είναι η υπόθεση ότι μια ενεργός χρησιμοποιούμενη σελίδα θα πρέπει να έχει μεγάλο αριθμό αναφορών. Ο αλγόριθμος αυτός έχει πρόβλημα όταν μια σελίδα χρησιμοποιήθηκε στο παρελθόν αλλά δε χρησιμοποιείται πια. Αν αναφέρθηκε πολλές φορές, έχει μεγάλη τιμή στο μετρητή και παραμένει στην μνήμη, παρ' όλο που δε χρειάζεται πια. Μια λύση είναι η ολίσθηση δεξιά των μετρητών κατά ένα bit σε τακτά διαστήματα, δημιουργώντας έτσι μια εκθετική μείωση κατά μέση τιμή χρήσης μετρητή.

5.6.8 Ο Αλγόριθμος της Πιο Συχνά Χρησιμοποιούμενης Σελίδας

Ένας άλλος αλγόριθμος αντικατάστασης σελίδων είναι ο αλγόριθμος της «πιο συχνά χρησιμοποιούμενης σελίδας» (Most Frequently Used – MFU), ο οποίος βασίζεται στην υπόθεση ότι η σελίδα με το μικρότερο μετρητή –πολύ πιθανόν– έχει μόλις μεταφερθεί και δεν έχει ακόμα χρησιμοποιηθεί. Ούτε ο MFU ούτε ο LRU είναι πολύ συνηθισμένοι. Η υλοποίηση αυτών των αλγόριθμων είναι αρκετά ακριβή.

Άσκηση Αυτοαξιολόγησης 5.11

Κατασκευάστε ακολουθίες αναφορών σελίδων που να έχουν τις ακόλουθες συμπεριφορές:

- α) Ο FIFO (First In First Out) αλγόριθμος αντικατάστασης σελίδας να είναι καλύτερος από τον LRU (Least Recently Used).
- β) Ο Random αλγόριθμος αντικατάστασης σελίδας (ο οποίος επιλέγει τυχαία μια σελίδα) να είναι καλύτερος από τον LRU.
- γ) Ο LRU αλγόριθμος αντικατάστασης σελίδας να είναι καλύτερος από τον LIFO (Last In First Out).

Απάντηση:

Έστω ότι έχουμε στη διάθεσή μας μια μνήμη με 4 πλαίσια σελίδας.

α) Έστω η ακολουθία αναφορών σελίδων σ_1 :

$$\sigma_1 = [0, 1, 7, 2, 3, 2, 7, 1, 0, 3]$$

Ο FIFO αλγόριθμος αντικατάστασης σελίδων κάνει 6 λάθη σελίδας σε αυτή την ακολουθία. Βλέπουμε στον ακόλουθο πίνακα πότε συμβαίνουν αυτά τα λάθη σελίδων.

σ1 :	0	1	7	2	3	2	7	1	0	3
πλαίσιο 0	0	0	0	0	3	3	3	3	3	3
πλαίσιο 1	1	1	1	1	1	1	1	0	0	
πλαίσιο 2		7	7	7	7	7	7	7	7	
πλαίσιο 3			2	2	2	2	2	2	2	

P = λάθος σελίδας

P	P	P	P	P					P
---	---	---	---	---	--	--	--	--	---

Ο LRU αλγόριθμος αντικατάστασης σελίδων κάνει 7 λάθη σελίδας σε αυτή την ακολουθία. Βλέπουμε στον ακόλουθο πίνακα πότε συμβαίνουν αυτά τα λάθη σελίδων.

σ1:	0	1	7	2	3	2	7	1	0	3
πλαίσιο 0	0	0	0	0	3	3	3	3	0	0
πλαίσιο 1		1	1	1	1	1	1	1	1	1
πλαίσιο 2			7	7	7	7	7	7	7	7
πλαίσιο 3				2	2	2	2	2	2	2

P λάθος σελίδας

P	P	P	P	P					P	P
---	---	---	---	---	--	--	--	--	---	---

β) Έστω η ακολουθία αναφορών σελίδων σ2.

$$\sigma_2 = [0, 1, 7, 2, 3, 2, 7, 1, 0, 3]$$

Ο LRU αλγόριθμος αντικατάστασης σελίδων κάνει 7 λάθη σελίδας σε αυτή την ακολουθία (όπως προηγουμένως, αφού $\sigma_1 = \sigma_2$)

Ο Random αλγόριθμος αντικατάστασης σελίδων κάνει 6 λάθη σελίδας σε αυτή την ακολουθία. Ο αλγόριθμος αυτός αντικαθιστά ισοπίθανα και ομοιόμορφα μια σελίδα από αυτές που βρίσκονται στη μνήμη. Βλέπουμε στον ακόλουθο πίνακα μια εκτέλεση αυτού του αλγόριθμου.

σ2 :	0	1	7	2	3	2	7	1	0	3
πλαίσιο 0	0	0	0	0	0	0	0	0	0	0
πλαίσιο 1		1	1	1	3	3	3	3	3	3
πλαίσιο 2			7	7	7	7	7	7	7	7
πλαίσιο 3				2	2	2	2	1	1	1

P λάθος σελίδας

P	P	P	P	P					P
---	---	---	---	---	--	--	--	--	---

γ) Έστω η ακολουθία αναφορών σελίδων σ_3 .

$$\sigma_3 = [0, 1, 2, 3, 2, 3, 2, 3, 2, 3, \dots]$$

Παρατηρούμε ότι ο LRU κάνει μόνο 5 λάθη σελίδων, ανεξάρτητα από το πόσο θα μεγαλώσει η ακολουθία σ_3 , φτάνει να συνεχίσει να εναλλάσσει στην αναφορά της τις σελίδες 2 και 3. Ο LIFO αλγόριθμος αντικατάστασης σελίδας θα κάνει σε κάθε βήμα λάθος σελίδας.

Άσκηση Αυτοαξιολόγησης 5.12

Δίνεται ακολουθία αναφορών σελίδων για μια διαδικασία με m πλαίσια (αρχικά όλα άδεια). Η ακολουθία έχει μήκος p , με n διαφορετικούς αριθμούς σελίδων σε αυτήν. Δώστε άνω (\max) και κάτω (\min) όριο για τον αριθμό λαθών αναφοράς που μπορούν να γίνουν από οποιονδήποτε αλγόριθμο αντικατάστασης σελίδας.

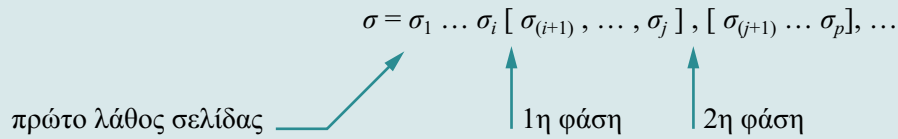
Απάντηση:

Διακρίνουμε δύο περιπτώσεις:

- α) Έστω $m \geq n$, τότε $\min = \max = n$. Αυτό συμβαίνει γιατί ο αριθμός των διαθέσιμων πλαισίων σελίδας είναι μεγαλύτερος από τον αριθμό των διακριτών σελίδων στις οποίες θα γίνει αναφορά, με αποτέλεσμα να γίνουν μόνο n λάθη σελίδων, ένα κάθε φορά που πρωτοεμφανίζεται καθεμιά από αυτές τις διακριτές σελίδες.
- β) $m < n$, τότε $\min = n$, γιατί θα γίνει τουλάχιστον ένα λάθος σελίδας την πρώτη φορά που θα αναφερθεί καθεμιά από τις διακριτές σελίδες. $\max = p$, γιατί η ακολουθία αναφορών μπορεί να φτιαχτεί έτσι ώστε να γίνεται αναφορά κάθε φορά στην τελευταία σελίδα που έχει απομακρυνθεί από τη μνήμη.

Άσκηση Αυτοαξιολόγησης 5.13

Δίνεται μνήμη με k πλαίσια. Έστω οι αλγόριθμοι αντικατάστασης σελίδων LRU (Least Recently Used) και OPTIMAL. Έστω ότι σ είναι μια αυθαίρετη σταθερή ακολουθία αναφορών σελίδων. Χωρίστε την ακολουθία σ σε φάσεις, όπου σε κάθε φάση ο LRU επιτελεί ακριβώς k «λάθη σελίδας» (page faults), ενώ η τελευταία αναφορά σελίδας κάθε φάσης είναι λάθος σελίδας για τον LRU:

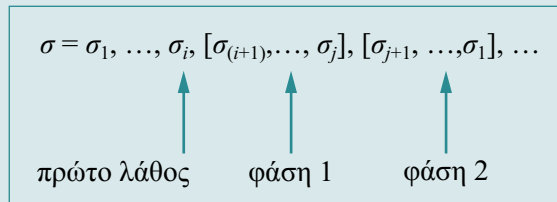


Για παράδειγμα, η πρώτη φάση τελειώνει με σ_j , όπου $j = \min \{ t : \text{Ο LRU κάνει } k \text{ λάθη σελίδας στο διάστημα } \sigma_{(i+1)} \dots \sigma_t \}$.

Ο LRU κάνει ακριβώς k λάθη σελίδας σε κάθε φάση εξ ορισμού.

Αποδείξτε ότι ο OPTIMAL κάνει τουλάχιστον ένα λάθος σελίδας σε κάθε φάση.

Απάντηση:

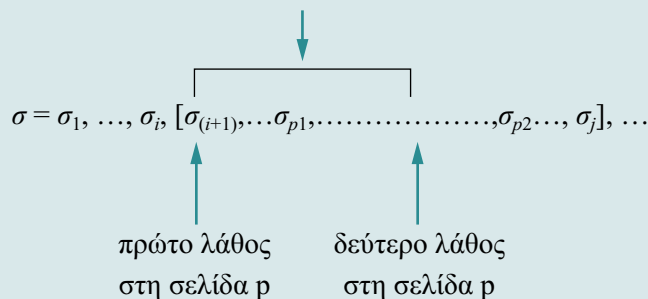


Θα εξετάσουμε δύο περιπτώσεις:

Περίπτωση 1η:

Τι θα συμβεί αν κατά τη διάρκεια μιας φάσης (έστω i) ο LRU κάνει δύο λάθη σελίδας πάνω στην ίδια σελίδα, έστω p ; Το τι συμβαίνει στη φάση αυτή φαίνεται στη συνέχεια:

Όλες οι k σελίδες στη μνήμη θα έχουν αναγκαστεί να απομακρυνθούν από τη μνήμη στο διάστημα μεταξύ του πρώτου και του δεύτερου λάθους σελίδας στη σελίδα p .



Από τη στιγμή που η σελίδα p εισέρχεται στη μνήμη μετά το πρώτο λάθος σελίδας, θα παραμείνει εκεί έως ότου αφαιρεθούν από τη μνήμη όλες οι άλλες σελίδες. Αυτό είναι συμπέρασμα από τον τρόπο λειτουργίας του αλγόριθμου αντικατάστα-

σης LRU. Έτσι, όταν υπάρξει και δεύτερο λάθος σελίδας στη σελίδα p , σημαίνει ότι όλες οι άλλες k σελίδες που θα βρίσκονται στη μνήμη έχουν εισαχθεί μετά το πρώτο λάθος στην p και πριν από το δεύτερο λάθος σε αυτή. Άρα, σε αυτή τη φάση έγινε αναφορά σε $k+1$ σελίδες. Αυτό σημαίνει ότι και ο Optimal αλγόριθμος θα κάνει τουλάχιστον ένα λάθος σε αυτή τη φάση.

Περίπτωση 2η:

Ο LRU κάνει λάθη σε k διαφορετικές σελίδες. Θα εξετάσουμε δύο υποπεριπτώσεις, με βάση το τελευταίο λάθος σελίδας, στη φάση ακριβώς πριν από την τρέχουσα φάση.

Περίπτωση 2α

$$\sigma_i = p, [\sigma_{i+1}, \dots, p, \dots, \sigma_j]$$

Αυτή η περίπτωση είναι ανάλογη με την περίπτωση 1. Πριν από το δεύτερο λάθος σελίδας στη σελίδα p , θα πρέπει να έχουν γίνει αναφορές σε k διαφορετικές σελίδες. Άρα, κατά τη διάρκεια αυτής της φάσης έγιναν αναφορές σε $k+1$ διαφορετικές σελίδες. Αφού έγινε αναφορά σε $k+1$ διαφορετικές σελίδες, τότε και ο OPTIMAL αλγόριθμος θα κάνει τουλάχιστον ένα λάθος σελίδας.

Περίπτωση 2β

$$\sigma_i = p, [\sigma_{i+1}, \dots, \sigma_j]$$

Δεν υπάρχει λάθος σελίδας στην παρούσα φάση στη σελίδα p . Έστω η p να βρίσκεται και στη μνήμη του OPTIMAL αλγόριθμου. Τότε, κατά την παρούσα φάση θα γίνουν αναφορές σε k διαφορετικές σελίδες εκτός της σελίδας p . Τότε, ο OPTIMAL αλγόριθμος θα κάνει ένα τουλάχιστον λάθος, γιατί η σελίδα p βρίσκεται στη μνήμη του.

5.6.9 Κλάσεις Σελίδων

Υπάρχουν πολλοί αλγόριθμοι που χρησιμοποιούνται για αντικατάσταση σελίδων. Για παράδειγμα, αν λάβουμε υπόψη μας και το bit αναφοράς και το dirty bit (Ενότητα 5.4) ως ένα διατεταγμένο ζεύγος έχουμε τις ακόλουθες 4 κλάσεις:

(0,0) μη χρησιμοποιημένη και μη αλλαγμένη

(0,1) μη χρησιμοποιημένη (πρόσφατα) αλλά αλλαγμένη

(1,0) χρησιμοποιημένη και μη αλλαγμένη

(1,1) χρησιμοποιημένη και αλλαγμένη

Όταν είναι απαραίτητη η αντικατάσταση σελίδων, κάθε σελίδα είναι σε μια από αυτές τις 4 κλάσεις. Αντικαθιστούμε οποιαδήποτε σελίδα στη χαμηλότερη μη κενή κλάση. Αν υπάρχουν πολλαπλές σελίδες στη χαμηλότερη κλάση, μπορούμε να χρησιμοποιήσουμε FIFO ή να διαλέξουμε τυχαία μεταξύ τους.

5.6.10 Ad Hoc Αλγόριθμοι

Συχνά, επιπρόσθετα με κάποιον αλγόριθμο αντικατάστασης σελίδων χρησιμοποιούνται και άλλες ρουτίνες. Για παράδειγμα, τα συστήματα συνήθως κρατούν μια «δεξαμενή» (pool) ελεύθερων πλαισίων. Όταν γίνεται ένα σφάλμα σελίδας, ένα πλαίσιο-θύμα διαλέγεται, όπως και προηγουμένως. Όμως, η επιθυμητή σελίδα μεταφέρεται σε ένα ελεύθερο πλαίσιο από τη δεξαμενή πριν το θύμα μεταφερθεί. Αυτός ο τρόπος επιτρέπει στη διαδικασία να ξαναρχίζει όσο πιο γρήγορα γίνεται, χωρίς να περιμένει να μεταφερθεί η σελίδα-θύμα. Όταν αργότερα γίνει αυτή η μεταφορά, το πλαίσιο προστίθεται στη δεξαμενή των ελεύθερων πλαισίων.

Μια επέκταση αυτής της ιδέας διατηρεί μια λίστα «αλλαγμένων» (dirty) σελίδων. Οποτεδήποτε η μονάδα σελιδοποίησης είναι ανενεργός, μια αλλαγμένη σελίδα επιλέγεται και γράφεται στην επικουρική μνήμη. Μετά επαναφέρεται το dirty bit. Αυτός ο τρόπος αυξάνει την πιθανότητα ότι η σελίδα που θα επιλεγεί για αντικατάσταση θα είναι καθαρή και δε θα χρειαστεί να μεταφερθεί.

Μια άλλη τροποποίηση είναι να κρατάμε μια δεξαμενή από ελεύθερα πλαίσια, αλλά να θυμόμαστε ποια σελίδα ήταν σε κάθε πλαίσιο. Επειδή τα περιεχόμενα του πλαισίου δεν τροποποιούνται με το γράψιμο του πλαισίου στην επικουρική μνήμη, η παλιά σελίδα μπορεί να επαναχρησιμοποιηθεί. Σε αυτή την περίπτωση δε χρειάζεται I/O. Όταν συμβεί ένα σφάλμα σελίδας, πρώτα ελέγχουμε αν η επιθυμητή σελίδα βρίσκεται στη δεξαμενή των ελεύθερων πλαισίων. Αν όχι, πρέπει να διαλέξουμε ένα ελεύθερο πλαίσιο και να μεταφέρουμε σε αυτό τη σελίδα.

Αυτή η τεχνική χρησιμοποιείται στο VAX/VMS σύστημα μαζί με έναν FIFO αλγόριθμο αντικατάστασης. Όταν ο FIFO αλγόριθμος αντικαθιστά λανθασμένα μια σελίδα που είναι ακόμα σε ενεργό χρήση, αυτή ανακτάται γρήγορα από τον «απομονωτή ελεύθερων πλαισίων» (Free – Frame Buffer), και έτσι δεν υπάρχει ανάγκη για I/O. Αυτός ο απομονωτής ελεύθερων πλαισίων προσφέρει προστασία έναντι του σχετικά φτωχού, αλλά απλού, αλγόριθμου FIFO.

5.7 Αλγόριθμοι Κατανομής

Αφού έχουμε επιλέξει έναν αλγόριθμο αντικατάστασης, έχουμε αρκετή ευελιξία στη διαχείριση της μνήμης. Η ιδεατή μνήμη του χρήστη μπορεί να είναι πολύ πιο μεγάλη από τη φυσική μνήμη. Η σελιδοποίηση ζήτησης και η αντικατάσταση σελίδας μάς επιτρέπουν να εκτελούμε μεγάλες διαδικασίες ακόμα και σε μικρή φυσική μνήμη.

Η απλούστερη περίπτωση ιδεατής μνήμης είναι ένα «σύστημα ενός χρήστη» (single-user system). Θεωρήστε ένα μικροϋπολογιστικό σύστημα ενός χρήστη με 128K bytes μνήμης που αποτελείται από σελίδες του 1K. Το ΛΣ χρειάζεται 35K bytes, αφήνοντας 93 πλαίσια για το πρόγραμμα του χρήστη. Κάτω από «καθαρή σελιδοποίηση ζήτησης» (pure demand paging), όλα τα 93 πλαίσια αρχικά θα έμπαιναν στη λίστα των ελεύθερων πλαισίων. Όταν ένα πρόγραμμα χρήστη αρχίσει να εκτελείται, θα δημιουργήσει μια ακολουθία σφαλμάτων σελίδας. Τα πρώτα 93 σφάλματα θα πάρουν ελεύθερα πλαίσια από τη λίστα των ελεύθερων πλαισίων. Όταν αδειάσει αυτή η λίστα, θα χρησιμοποιηθεί ένας αλγόριθμος αντικατάστασης σελίδας που θα επιλέξει μία από τις 93 χρησιμοποιούμενες (in-core) σελίδες για να αντικατασταθεί από την 94η και ούτω καθεξής. Όταν το πρόγραμμα θα τελειώσει, τα 93 πλαίσια θα τοποθετηθούν ξανά στη λίστα των ελεύθερων πλαισίων.

Υπάρχουν πολλές παραλλαγές αυτής της απλής στρατηγικής. Μπορούμε να σχεδιάσουμε το ΛΣ ώστε να παίρνει όλο το χώρο που χρειάζεται για απομονωτές και πίνακες από τη λίστα των ελεύθερων πλαισίων. Όταν ο χώρος αυτός δε χρησιμοποιείται από το ΛΣ, μπορεί να υποστηρίξει τη σελιδοποίηση του χρήστη. Θα μπορούσαμε να κρατάμε τρία ελεύθερα πλαίσια στη λίστα των ελεύθερων πλαισίων σε όλες τις στιγμές. Έτσι, όταν συμβεί ένα σφάλμα σελίδας, υπάρχει ελεύθερο πλαίσιο για την εξυπηρέτησή του. Ενόσω γίνεται η εναλλαγή σελίδας, θα μπορούσε να επιλεγεί ένας αντικαταστάτης, που μετά γράφεται στην επικουρική μνήμη, καθώς το πρόγραμμα του χρήστη συνεχίζει την εκτέλεσή του.

Πιθανές είναι επίσης και άλλες παραλλαγές, αλλά η βασική στρατηγική είναι εμφανής: το πρόγραμμα του χρήστη παίρνει οποιοδήποτε ελεύθερο πλαίσιο. Ένα διαφορετικό πρόβλημα εμφανίζεται όταν συνδυαστεί η σελιδοποίηση ζήτησης με τον πολυπρογραμματισμό. Ο πολυπρογραμματισμός βάζει δύο διαδικασίες (ή πιο πολλές) στη μνήμη την ίδια στιγμή (σε αυτές τις περιπτώσεις, η μνήμη είναι πολύ μεγαλύτερη, π.χ. 256MB). Πώς κατανέμουμε το σταθερό ποσό ελεύθερης μνήμης στις διάφορες διαδικασίες. Αν έχουμε 93 ελεύθερα πλαίσια και 2 διαδικασίες, πόσα πλαίσια παίρνει κάθε διαδικασία;

5.7.1 Ελάχιστος Αριθμός Πλαισίων

Υπάρχουν, φυσικά, περιορισμοί στην κατανομή μας. Δεν μπορούμε να κατανεύουμε περισσότερα πλαίσια από το συνολικό αριθμό διαθέσιμων πλαισίων (εκτός αν υπάρχει μοίρασμα σελίδας). Σε κάθε διαδικασία αντιστοιχεί ένας *ελάχιστος αριθμός πλαισίων*, τα οποία πρέπει να της δοθούν για να μπορεί να εκτελεστεί χωρίς υπερβολική καθυστέρηση. Αν της δοθούν λιγότερα πλαίσια, τότε αυξάνεται ο ρυθμός σφαλμάτων σελίδας (και ο χρόνος προσπέλασης), καθυστερώντας έτσι την εκτέλεση της διαδικασίας.

Εκτός από τις ανεπιθύμητες επιπτώσεις στην απόδοση που προκαλεί η κατανομή λίγων μόνο πλαισίων, υπάρχει ένας ελάχιστος αριθμός πλαισίων που πρέπει να κατανεμηθούν. Αυτός ο ελάχιστος αριθμός ορίζεται από την αρχιτεκτονική του συνόλου των εντολών. Θυμηθείτε ότι, όταν συμβαίνει ένα σφάλμα σελίδας πριν μια εκτελούμενη εντολή ολοκληρωθεί, πρέπει να επανεκτελέσουμε την εντολή. Συνεπώς, πρέπει να έχουμε αρκετά πλαίσια για να κρατάμε όλες τις διαφορετικές σελίδες που μπορεί να αναφερθούν από μια εντολή.

Για παράδειγμα, θεωρήστε το PDP-8. Όλες του οι εντολές αναφοράς μνήμης έχουν μόνο μία διεύθυνση μνήμης. Έτσι, χρειαζόμαστε τουλάχιστον ένα πλαίσιο για την εντολή και ένα πλαίσιο για την αναφορά μνήμης. Επιπλέον, η διεύθυνση που ορίζεται στην εντολή μπορεί να είναι μια έμμεση αναφορά. Έτσι, μια εντολή φόρτωσης στη σελίδα 16 μπορεί να αναφέρεται σε μια διεύθυνση στη σελίδα 0, που είναι μια έμμεση αναφορά στη σελίδα 23. Άρα, η σελιδοποίηση στο PDP-8 απαιτεί τουλάχιστον τρία πλαίσια ανά διαδικασία. Σκεφτείτε τι είναι πιθανό να γίνει αν μια διαδικασία είχε μόνο δύο πλαίσια.

Ο ελάχιστος αριθμός πλαισίων ορίζεται από την αρχιτεκτονική του υπολογιστή. Ενώ το PDP-8 χρειάζεται τρία πλαίσια, το PDP-11 απαιτεί τουλάχιστον έξι. Η εντολή μετακίνησης για μερικούς τύπους διευθυνσιοδότησης είναι πιο πολύ από μια λέξη, και έτσι η ίδια η εντολή μπορεί να διασκελίζει δύο σελίδες. Επιπρόσθετα, καθένας από τους τελεστές της μπορεί να είναι μια έμμεση αναφορά, συνολικά έξι πλαίσια. Η χειρότερη περίπτωση για τον IBM 370 είναι η εντολή μετακίνησης χαρακτήρα. Εφόσον η εντολή είναι από μνήμη σε μνήμη, χρειάζεται 6 bytes και μπορεί να διασκελίζει δύο σελίδες. Το σύνολο των χαρακτήρων προς μεταφορά και η περιοχή όπου θα μεταφερθούν αυτοί μπορούν επίσης να διασκελίζουν δύο σελίδες. Αυτή η περίπτωση θα απαιτούσε 6 πλαίσια. (Στην πραγματικότητα, η χειρότερη περίπτωση είναι αν η εντολή αυτή είναι ο τελεστής μιας εντολής Execute η οποία διασκελίζει το όριο μιας σελίδας – σε αυτή την περίπτωση χρειαζόμαστε 8 πλαίσια.)

Η αρχιτεκτονική του Data General Nova 3 επέτρεπε πολλαπλά επίπεδα έμμεσης αναφοράς: κάθε λέξη των 16 bits. Θεωρητικά, μια απλή εντολή φόρτωσης θα μπορούσε να αναφερθεί σε μια έμμεση διεύθυνση (σε άλλη σελίδα), που θα μπορούσε, επίσης, να αναφερθεί σε μια έμμεση διεύθυνση (σε μια ακόμα άλλη σελίδα) και ούτω καθεξής, έως ότου κάθε σελίδα στην ιδεατή μνήμη θα έπρεπε να είναι στη φυσική μνήμη. Παρατηρώντας ότι καμία διαδικασία δεν έκανε ποτέ μεγάλη χρήση αυτής της «δυνατότητας», οι μηχανικοί τροποποίησαν, ώστε να περιορίζεται μια εντολή σε 16 το πολύ επίπεδα έμμεσης αναφοράς. Όταν γίνεται η πρώτη έμμεση αναφορά, ένας μετρητής τίθεται στο 16 και μειώνεται για κάθε έμμεση αναφορά που κάνει αυτή η εντολή. Όταν ο μετρητής φτάσει το μηδέν, συμβαίνει μια παγίδευση (excession indirection). Αυτό το όριο μειώνει το μέγιστο αριθμό αναφορών μνήμης ανά εντολή στο 17, απαιτώντας τόσα πλαίσια.

Ο ελάχιστος αριθμός πλαισίων ανά διαδικασία ορίζεται από την αρχιτεκτονική, ενώ ο μέγιστος αριθμός ορίζεται από το ποσό της διαθέσιμης φυσικής μνήμης. Στο μεταξύ τους διάστημα έχουμε σημαντικές επιλογές στην κατανομή των πλαισίων.

5.7.2 Γενική έναντι Τοπικής Κατανομής

Σε ένα σύστημα υπάρχουν πολλαπλές διαδικασίες, οι οποίες ανταγωνίζονται για να καταλάβουν (η καθεμιά για λογαριασμό της) όσο το δυνατόν περισσότερα πλαίσια (γιατί;). Το ΛΣ δεν είναι αναγκαίο να αποφασίζει εκ των προτέρων, λεπτομερώς, πόσα πλαίσια πρέπει να κατανεμηθούν σε κάθε διαδικασία. Ο αριθμός των πλαισίων που έχουν κατανεμηθεί σε κάθε διαδικασία κάθε χρονική στιγμή καθορίζεται από το αν ο αλγόριθμος αντικατάστασης σελίδας είναι «γενικής αντικατάστασης» (global replacement) ή «τοπικής αντικατάστασης» (local replacement) και την ακολουθία αναφορών. Η *global αντικατάσταση* επιτρέπει σε μια διαδικασία να επιλέξει ένα πλαίσιο αντικατάστασης από το σύνολο όλων των πλαισίων, ακόμα κι αν το πλαίσιο αυτό είναι κατανεμημένο σε κάποια άλλη διαδικασία: μια διαδικασία μπορεί να πάρει ένα πλαίσιο από μια άλλη. Η *τοπική αντικατάσταση* απαιτεί ότι κάθε διαδικασία μπορεί να επιλέξει μόνο από το δικό της σύνολο κατανεμημένων πλαισίων.

Με μια στρατηγική τοπικής αντικατάστασης ο αριθμός των πλαισίων που έχουν κατανεμηθεί σε μια διαδικασία δεν αλλάζει. Με τη γενική αντικατάσταση μια διαδικασία μπορεί να διαλέξει πλαίσια που είναι κατανεμημένα και σε άλλες διαδικασίες.

Άσκηση Αυτοαξιολόγησης 5.14

Σε ποια περίπτωση αυξάνεται ο αριθμός των πλαισίων που έχουν κατανεμηθεί σε μια διαδικασία όταν ο αλγόριθμος αντικατάστασης σελίδων είναι γενικής αντικατάστασης;

Απάντηση:

Όταν οι άλλες διαδικασίες (οι οποίες εκτελούνται ταυτόχρονα με την εν λόγω διαδικασία) δε διαλέγουν τα δικά της πλαίσια για αντικατάσταση και αυτή διαλέγει πλαίσια άλλων διαδικασιών.

Ένα πρόβλημα ενός αλγόριθμου γενικής αντικατάστασης είναι ότι η κάθε διαδικασία δεν έχει τον έλεγχο του ρυθμού σφαλμάτων σελίδων. Το σύνολο των σελίδων, στη μνήμη, για μια διαδικασία εξαρτάται όχι μόνο από τη συμπεριφορά σελιδοποίησης γι' αυτή τη διαδικασία, αλλά και από τη συμπεριφορά σελιδοποίησης των άλλων διαδικασιών. Έτσι, η ίδια διαδικασία μπορεί να αποδώσει τελείως διαφορετικά (κάνοντας 0,5 sec για μια εκτέλεση και 10,3 sec για την επόμενη εκτέλεση), εξαιτίας εντελώς εξωτερικών καταστάσεων. Αυτό δε συμβαίνει σε έναν αλγόριθμο τοπικής αντικατάστασης. Κάτω από καθεστώς τοπικής αντικατάστασης, το σύνολο των σελίδων στη μνήμη για μια διαδικασία επηρεάζεται μόνο από τη συμπεριφορά σελιδοποίησης αυτής της διαδικασίας.

5.7.3 Αλγόριθμοι Κατανομής

Ο ευκολότερος τρόπος για να χωριστούν τα πλαίσια μεταξύ των διαδικασιών είναι να δοθεί στην καθεμιά ένα ίσο μερίδιο m/n πλαισίων. Για παράδειγμα, αν υπάρχουν 93 πλαίσια και 5 διαδικασίες, κάθε διαδικασία θα έπαιρνε 18 πλαίσια. Τα εναπομείναντα 3 πλαίσια θα μπορούσαν να χρησιμοποιηθούν ως μια δεξαμενή ελεύθερων πλαισίων. Αυτός ο τρόπος ονομάζεται «ίση κατανομή».

Μια παραλλαγή είναι η αναγνώριση του γεγονότος ότι διαφορετικές διαδικασίες θα χρειάζονται διαφορετικά μεγέθη μνήμης. Αν ένα μικρό φοιτητικό πρόγραμμα των 10K και μια interactive βάση δεδομένων των 127K είναι οι μόνες δύο διαδικασίες που τρέχουν σε ένα σύστημα με 62 πλαίσια, δεν είναι λογικό να δώσουμε σε καθεμιά 31 πλαίσια. Το φοιτητικό πρόγραμμα δε θέλει πάνω από 10 πλαίσια, και έτσι τα υπόλοιπα 21 πλαίσια σπαταλιούνται.

Για να λύσουμε αυτό το πρόβλημα, χρησιμοποιούμε «αναλογική κατανομή» (proportional allocation). Κατανέμουμε τη διαθέσιμη μνήμη σε κάθε διαδικασία ανά-

λογα με το μέγεθός της. Έστω ότι το μέγεθος της ιδεατής μνήμης για τη διαδικασία p_i είναι s_i , ορίζουμε

$$S = \sum s_i$$

Τότε, αν ο συνολικός αριθμός των διαθέσιμων πλαισίων είναι m , κατανέμουμε a_i πλαίσια στη διαδικασία p_i , όπου a_i είναι

$$a_i = \frac{s_i}{S} \times m$$

Φυσικά, πρέπει να κάνουμε τα a_i να είναι ακέραιοι μεγαλύτεροι από τον ελάχιστο αριθμό πλαισίων που απαιτείται από το σύνολο των εντολών, με άθροισμα όχι μεγαλύτερο του m .

Με αναλογική κατανομή θα χωρίζουμε 62 πλαίσια μεταξύ δύο διαδικασιών, μια των 10 σελίδων και μια των 127 σελίδων, δίνοντάς τους 4 πλαίσια και 57 πλαίσια αντίστοιχα, αφού $10/137 \times 62 \approx 4$ και $127/137 \times 62 \approx 57$.

Με αυτό τον τρόπο οι δύο διαδικασίες μοιράζονται τα διαθέσιμα πλαίσια σύμφωνα με τις ανάγκες τους και όχι ισομερώς.

Και στις δύο περιπτώσεις, η κατανομή προς κάθε διαδικασία μπορεί να μεταβάλλεται ανάλογα με το επίπεδο του πολυπρογραμματισμού. Αν το επίπεδο αυτό αυξηθεί, τότε η διαδικασία θα χάσει μερικά πλαίσια, για να παρασχεθεί στη νέα διαδικασία η αναγκαία μνήμη. Από την άλλη μεριά, αν το επίπεδο πολυπρογραμματισμού μειωθεί, τα πλαίσια που είχαν κατανεμηθεί στην περατωμένη διαδικασία μπορούν τώρα να μοιραστούν μεταξύ των εναπομεινανσών διαδικασιών.

Μέχρι τώρα δε λάβαμε υπόψη την *προτεραιότητα* των διαδικασιών.

Μια αντιμετώπιση είναι η χρησιμοποίηση ενός σχήματος αναλογικής κατανομής όπου η αναλογία των πλαισίων δεν εξαρτάται από το σχετικό μέγεθος των διαδικασιών, αλλά από τις προτεραιότητές τους ή από ένα συνδυασμό μεγέθους και προτεραιότητας.

Μια άλλη αντιμετώπιση είναι να επιτραπεί στις διαδικασίες υψηλής προτεραιότητας να διαλέγουν πλαίσια προς αντικατάσταση από διαδικασίες χαμηλής προτεραιότητας.

Μια διαδικασία μπορεί να διαλέξει ένα πλαίσιο από τα δικά της πλαίσια ή από τα πλαίσια οποιασδήποτε διαδικασίας χαμηλότερης προτεραιότητας. Αυτή η προσέγγιση επιτρέπει σε μια διαδικασία υψηλής προτεραιότητας να αυξήσει τα πλαίσιά της σε βάρος μιας διαδικασίας χαμηλής προτεραιότητας.

5.8 Λυγισμός (Thrashing)

Αν ο αριθμός των πλαισίων που είναι κατανεμημένα σε μια διαδικασία (συνήθως χαμηλής προτεραιότητας) πέσει κάτω από τον ελάχιστο αριθμό που απαιτείται από την αρχιτεκτονική του υπολογιστή, τότε πρέπει να διακοπεί η εκτέλεσή της. Αυτό ουσιαστικά σημαίνει ότι πρέπει να «εναλλαχθούν έξω» όλες οι σελίδες της, ελευθερώνοντας έτσι τα κατειλημμένα από αυτή πλαίσια. Αυτή η πράξη εισάγει ένα επίπεδο «εναλλαγής μέσα / έξω» στη «μεσαίου επιπέδου χρονοδρομολόγηση της CPU» (intermediate cpu scheduling) (δες Λειτουργικά Συστήματα II).

Παρατηρήστε οποιαδήποτε διαδικασία η οποία δεν έχει «αρκετά» πλαίσια. Παρ' όλο που είναι τεχνικά δυνατό να μειωθεί ο αριθμός των κατανεμημένων πλαισίων στο ελάχιστο, υπάρχει κάποιος (μεγάλος) αριθμός σελίδων που βρίσκονται σε χρήση. Αν η διαδικασία δεν έχει αυτό τον αριθμό πλαισίων, πολύ γρήγορα θα προκαλέσει σφάλμα σελίδας. Σε αυτό το σημείο θα πρέπει να αντικαταστήσει κάποια σελίδα που θα τη χρειαστεί αμέσως. Συνεπώς, πολύ γρήγορα γίνεται σφάλμα σελίδας και ξανά και ξανά. Η διαδικασία συνεχίζει να κάνει σφάλματα, αντικαθιστώντας σελίδες για τις οποίες θα δημιουργήσει αμέσως σφάλματα και θα τις επαναφέρει.

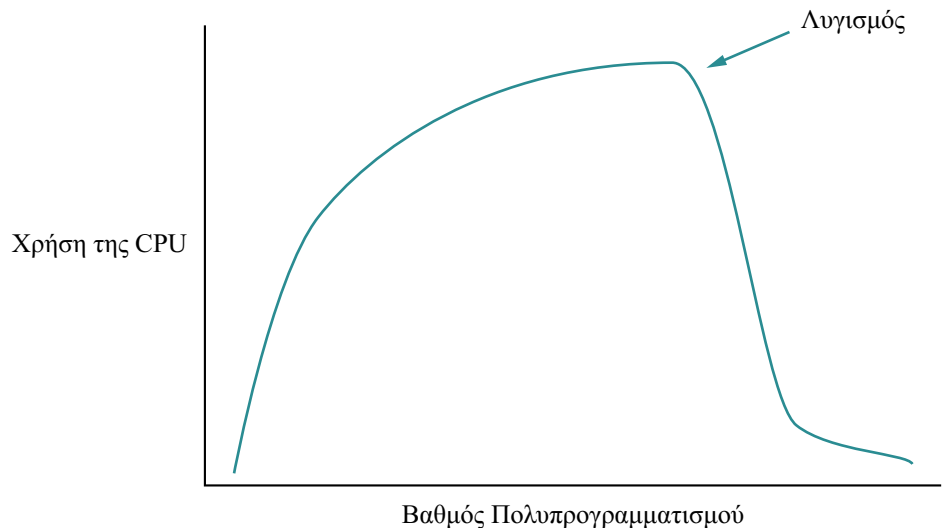
Αυτή η πολύ υψηλή δραστηριότητα σε σελιδοποίηση ονομάζεται «λυγισμός». Μια διαδικασία που υποφέρει από λυγισμό ξοδεύει περισσότερο χρόνο στη σελιδοποίηση παρά στην εκτέλεση. Ο λυγισμός μπορεί να προκαλέσει σοβαρά προβλήματα απόδοσης. Σκεφτείτε το ακόλουθο σενάριο, που είναι βασισμένο στην πραγματική συμπεριφορά των πρώιμων σελιδοποιημένων συστημάτων.

Το ΛΣ παρακολουθεί τη χρήση της CPU. Αν η χρήση της CPU είναι πολύ χαμηλή, ο βαθμός πολυπρογραμματισμού αυξάνεται εισάγοντας μια νέα διαδικασία στο σύστημα. Ένας αλγόριθμος γενικής αντικατάστασης σελίδων χρησιμοποιείται. Τώρα υποθέστε ότι μια διαδικασία εισέρχεται σε μια νέα φάση της εκτέλεσής της και χρειάζεται περισσότερα πλαίσια. Αρχίζει να δημιουργεί σφάλματα σελίδας και παίρνει σελίδες από τις άλλες διαδικασίες. Όλες οι διαδικασίες που κάνουν σφάλματα σελίδας πρέπει να χρησιμοποιήσουν τη μονάδα σελιδοποίησης για να εναλλάξουν σελίδες μέσα και έξω. Καθώς μπαίνουν στην ουρά για τη μονάδα σελιδοποίησης, η ουρά των διαδικασιών οι οποίες είναι έτοιμες για εκτέλεση αδειάζει. Δηλαδή πέφτει η χρήση της CPU.

Ο χρονοδρομολογητής της CPU βλέπει αυτή την πτώση και αυξάνει το βαθμό του πολυπρογραμματισμού. Η νέα διαδικασία προσπαθεί να ξεκινήσει παίρνοντας σελίδες από εκτελούμενες διαδικασίες, δημιουργώντας περισσότερα σφάλματα σελίδας, και η ουρά για τη μονάδα σελιδοποίησης μακραίνει. Ως αποτέλεσμα, η χρήση της CPU μειώνεται ακόμα περισσότερο και ο χρονοδρομολογητής προσπαθεί να αυξή-

σει ακόμα περισσότερο το βαθμό του πολυπρογραμματισμού. Έτσι, εμφανίζεται ο λυγισμός και η απόδοση του συστήματος πέφτει πολύ χαμηλά. Ο ρυθμός σφαλμάτων σελίδας αυξάνεται τρομερά. Ως αποτέλεσμα, ο μέσος χρόνος προσπέλασης μνήμης αυξάνεται. Καμιά δουλειά δεν τελειώνει, αφού οι διαδικασίες σπαταλούν το χρόνο τους στη σελιδοποίηση.

Αυτό το φαινόμενο απεικονίζεται στο Σχήμα 5.15. Καθώς ο βαθμός πολυπρογραμματισμού αυξάνεται, η χρήση της CPU αυξάνεται, αν και πιο αργά, έως ένα μέγιστο. Αν ο βαθμός πολυπρογραμματισμού αυξηθεί ακόμα περισσότερο, τότε έχουμε λυγισμό και τη χρήση της CPU και για να σταματήσει ο λυγισμός πρέπει να μειώσουμε το βαθμό του πολυπρογραμματισμού.



Σχήμα 5.15

Λυγισμός

Οι συνέπειες του λυγισμού μπορούν να περιοριστούν χρησιμοποιώντας έναν αλγόριθμο τοπικής ή με προτεραιότητα αντικατάστασης. Με την τοπική αντικατάσταση, αν μια διαδικασία φτάσει στο λυγισμό, δεν μπορεί να κλέψει πλαίσια από μια άλλη διαδικασία εισάγοντας και αυτή στο λυγισμό. Όμως, αν υπάρχουν διαδικασίες σε λυγισμό, θα βρίσκονται στην ουρά της μονάδας σελιδοποίησης τον περισσότερο χρόνο. Ο μέσος χρόνος εξυπηρέτησης ενός σφάλματος σελίδας θα αυξηθεί, λόγω της μεγαλύτερης μέσης ουράς για τη μονάδα σελιδοποίησης. Έτσι, ο πραγματικός χρόνος προσπέλασης θα αυξηθεί ακόμα και για μια διαδικασία που δεν είναι σε λυγισμό.

5.8.1 Τοπικότητα

Για να εμποδίσουμε το λυγισμό, πρέπει να δώσουμε σε μια διαδικασία όσα πλαίσια χρειάζεται. Αλλά πώς ξέρουμε πόσα πλαίσια «χρειάζεται»; Υπάρχουν αρκετές τεχνι-

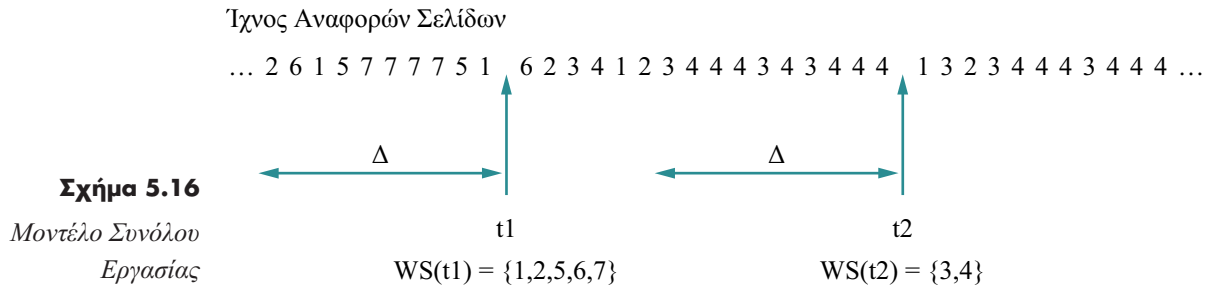
κές. Η στρατηγική του συνόλου εργασίας (Ενότητα 5.8.2) αρχίζει βλέποντας τι χρησιμοποιεί πραγματικά μια διαδικασία. Αυτή η προσέγγιση ορίζει το «μοντέλο τοπικότητας» (locality model) της εκτέλεσης μιας διαδικασίας.

Το μοντέλο τοπικότητας ορίζει ότι, καθώς μια διαδικασία εκτελείται, μετακινείται από τοπικότητα σε τοπικότητα. *Τοπικότητα* είναι το σύνολο των σελίδων που είναι όλες ενεργές. Μια διαδικασία, γενικά, αποτελείται από αρκετές διαφορετικές τοπικότητες που μπορεί να επικαλύπτονται. Για παράδειγμα, όταν καλείται μια υπορουτίνα, ορίζει μια νέα τοπικότητα. Σε αυτή οι αναφορές μνήμης γίνονται προς τις εντολές της υπορουτίνας, τις τοπικές της μεταβλητές και ένα υποσύνολο των γενικών μεταβλητών. Όταν τελειώνει η υπορουτίνα, η διαδικασία φεύγει από την τοπικότητα, αφού οι τοπικές μεταβλητές και οι εντολές της υπορουτίνας δε βρίσκονται πια σε ενεργό χρήση. Μπορεί να επιστρέψει σε αυτήν αργότερα. Έτσι, παρατηρούμε ότι οι τοπικότητες ορίζονται από τη δομή της διαδικασίας και από τις δομές δεδομένων τους. Το μοντέλο τοπικότητας ορίζει ότι όλες οι διαδικασίες θα εμφανίζουν αυτή τη βασική δομή αναφοράς μνήμης.

Υποθέστε ότι κατανέμουμε αρκετά πλαίσια σε μια διαδικασία για να ικανοποιήσουμε την τρέχουσα τοπικότητά της. Θα προκαλέσει σφάλματα σελίδας για τις σελίδες που ανήκουν στην τοπικότητά της έως ότου όλες αυτές οι σελίδες βρίσκονται στη μνήμη, και μετά δεν πρόκειται να κάνει σφάλματα σελίδας μέχρι να αλλάξει τοπικότητα. Αν κατανείμουμε λιγότερα πλαίσια από το μέγεθος της τρέχουσας τοπικότητας, η διαδικασία θα εισέλθει σε λυγισμό, αφού δεν μπορεί να κρατήσει στη μνήμη όλες τις σελίδες που χρειάζεται.

5.8.2 Μοντέλο Συνόλου Εργασίας (Working Set Model)

Αυτό το μοντέλο είναι βασισμένο στην υπόθεση της τοπικότητας. Χρησιμοποιεί μια παράμετρο Δ για να ορίσει το παράθυρο του συνόλου εργασίας. Η ιδέα είναι η εξέταση των Δ πιο πρόσφατων αναφορών σελίδας. Το σύνολο των σελίδων που είναι μέσα στις Δ πιο πρόσφατες αναφορές σελίδας είναι το σύνολο εργασίας (Σχήμα 5.16). Αν μια σελίδα βρίσκεται σε ενεργό χρήση, θα βρίσκεται στο σύνολο εργασίας. Αν δε χρησιμοποιείται άλλο, τότε θα βγει από το σύνολο εργασίας Δ χρονικές μονάδες μετά την τελευταία αναφορά της. Έτσι, το σύνολο εργασίας είναι μια προσέγγιση της τοπικότητας μιας διαδικασίας.



Για παράδειγμα, παίρνοντας των ακολουθία αναφορών μνήμης του Σχήματος 5.16, αν το $\Delta = 10$ αναφορές μνήμης, τότε το σύνολο εργασίας κατά το χρόνο t1 είναι 1,2,5,6,7. Κατά το χρόνο t2 το σύνολο εργασίας έχει αλλάξει σε 3,4.

Η ακρίβεια του συνόλου εργασίας εξαρτάται από την επιλογή του Δ . Αν το Δ είναι πολύ μικρό, τότε δε θα περιλαμβάνει ολόκληρο το σύνολο εργασίας. Αν το Δ είναι πολύ μεγάλο, μπορεί να επικαλύπτει αρκετές τοπικότητες. Στην ακραία περίπτωση, αν το Δ είναι άπειρο, τότε το σύνολο εργασίας είναι ολόκληρη η διαδικασία. Οι Madnick και Donovan [1974] προτείνουν το Δ να είναι περίπου 10.000 αναφορές.

Η πιο σημαντική ιδιότητα του συνόλου εργασίας είναι το μέγεθός του. Αν υπολογίσουμε το μέγεθος του συνόλου εργασίας, WSS_i , για κάθε διαδικασία στο σύστημα, τότε μπορούμε να πούμε:

$$D = \sum WSS_i$$

όπου D είναι η συνολική ζήτηση για πλαίσια. Κάθε διαδικασία ενεργά χρησιμοποιεί τις σελίδες που ανήκουν στο σύνολο εργασίας της. Έτσι, η διαδικασία i χρειάζεται WSS_i πλαίσια. Αν η συνολική ζήτηση είναι μεγαλύτερη από το συνολικό αριθμό διαθέσιμων πλαισίων, τότε θα εμφανιστεί λυγισμός, αφού κάποιες διαδικασίες δε θα έχουν αρκετά πλαίσια.

Η χρήση του συνόλου εργασίας είναι τότε αρκετά απλή. Το ΛΣ παρακολουθεί το σύνολο εργασίας κάθε διαδικασίας και κατανέμει σε αυτό αρκετά πλαίσια, ώστε να της παράσχει το μέγεθος του συνόλου εργασίας της. Αν υπάρχουν αρκετά ελεύθερα πλαίσια, άλλη μια διαδικασία μπορεί να εκτελεστεί. Αν το άθροισμα των μεγεθών των συνόλων εργασίας αυξηθεί, υπερβαίνοντας το συνολικό αριθμό των διαθέσιμων πλαισίων, το ΛΣ διαλέγει μια διαδικασία και τη σταματάει. Οι σελίδες γράφονται στην επικουρική μνήμη και τα πλαίσιά της επανακατανέμονται στις υπόλοιπες διαδικασίες. Η σταματημένη διαδικασία μπορεί να αρχίσει να εκτελείται αργότερα.

Αυτή η στρατηγική του συνόλου εργασίας εμποδίζει το λυγισμό και κρατάει το βαθμό του πολυπρογραμματισμού όσο πιο ψηλά γίνεται. Έτσι, προσπαθεί να βελτιστοποιήσει τη χρήση της CPU.

Η δυσκολία του μοντέλου του συνόλου εργασίας είναι η ιχνηλάτηση του συνόλου εργασίας. Το παράθυρο του συνόλου εργασίας είναι ένα κινητό παράθυρο. Σε κάθε αναφορά μνήμης, μια νέα αναφορά εμφανίζεται στο ένα άκρο και η παλαιότερη αναφορά φεύγει από το άλλο άκρο. Μια σελίδα ανήκει στο σύνολο εργασίας αν αναφέρεται οπουδήποτε μέσα στο παράθυρο του συνόλου εργασίας. Μπορούμε να προσεγγίσουμε το μοντέλο του συνόλου εργασίας με μια «χρονική διακοπή σταθερού διαστήματος» (fixed interval timer interrupt) και ένα bit αναφοράς.

Για παράδειγμα, υποθέστε ότι το Δ είναι 10.000 αναφορές και μπορούμε να δημιουργήσουμε μια χρονική διακοπή κάθε 5.000 αναφορές. Όταν δεχτούμε τη χρονική διακοπή, αντιγράφουμε και «καθαρίζουμε» τις τιμές των bits αναφοράς για κάθε σελίδα. Έτσι, αν γίνει ένα σφάλμα σελίδας, μπορούμε να εξετάσουμε το τρέχον bit αναφοράς και τα δύο bits στη μνήμη για να καθορίσουμε αν μια σελίδα χρησιμοποιήθηκε κατά τις τελευταίες 10.000 έως 15.000 αναφορές. Αν έχει χρησιμοποιηθεί, τουλάχιστον ένα από αυτά τα bit θα έχει τεθεί. Αυτές οι σελίδες που έχουν τουλάχιστον ένα bit στην τιμή 1 θα θεωρούνται ότι ανήκουν στο σύνολο εργασίας. Παρατηρήστε ότι αυτός ο τρόπος δεν είναι εντελώς ακριβής, αφού δεν μπορούμε να πούμε πότε ότι μέσα σε ένα διάστημα των 5.000 αναφορών συνέβη μια αναφορά. Μπορούμε να μειώσουμε την αβεβαιότητα αυξάνοντας τον αριθμό των bits ιστορίας και τον αριθμό των διακοπών (π.χ. 10 bits και διακοπές κάθε 1.000 αναφορές). Όμως, το κόστος εξυπηρέτησης αυτών των συχνών διακοπών θα είναι, αντίστοιχα, υψηλότερο.

Άσκηση Αυτοαξιολόγησης 5.15

Έστω $w = r_1, r_2, \dots, r_T$ μια ακολουθία αναφορών σελίδων και ας υποθέσουμε μια πολιτική συνόλου εργασίας. Έστω:

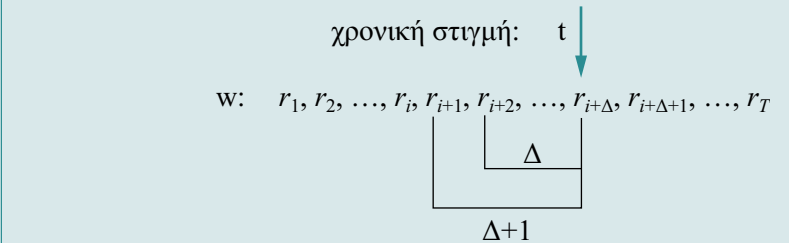
Δ = μέγεθος παραθύρου

$w(\Delta)$ = μέσο μέγεθος συνόλου εργασίας για το w

$F(\Delta)$ = μέσος ρυθμός λαθών αναφοράς

$w(t, \Delta)$ = μέγεθος συνόλου εργασίας τη χρονική στιγμή t

Δείξτε ότι για μεγάλα T το $F(\Delta)$ είναι περίπου $w(\Delta+1) - w(\Delta)$, δηλαδή το F είναι η παράγωγος του μέσου μεγέθους συνόλου εργασίας.

Απάντηση:

$$w(t, \Delta+1) = \begin{cases} w(t-1, \Delta), & \text{αν δεν έγινε λάθος σελίδας τη στιγμή } t \\ w(t-1, \Delta) + 1, & \text{διαφορετικά} \end{cases}$$

$$w(t, \Delta+1) = (1 - F(\Delta)) * (w(t-1, \Delta)) + F(\Delta) * (w(t-1, \Delta) + 1)$$

$$w(t, \Delta+1) = w(t-1, \Delta) + F(\Delta)$$

$$\Delta F(\Delta) = w(t, \Delta+1) - w(t-1, \Delta)$$

Για να προσδιορίσουμε τις μέσες τιμές $w(\Delta+1)$ και $w(\Delta)$ δουλεύουμε ως εξής:

$$1/T \sum (w(t, \Delta+1)) = 1/T \sum (w(t-1, \Delta) + F(\Delta))$$

Τα αθροίσματα είναι από $t = 0$ ως $t = T$.

$$\Rightarrow w(\Delta+1) = w(\Delta) - w(T, \Delta)/T + F(\Delta)$$

Αφού το T είναι μεγάλο, $w(T, \Delta)/T$ τείνει στο μηδέν

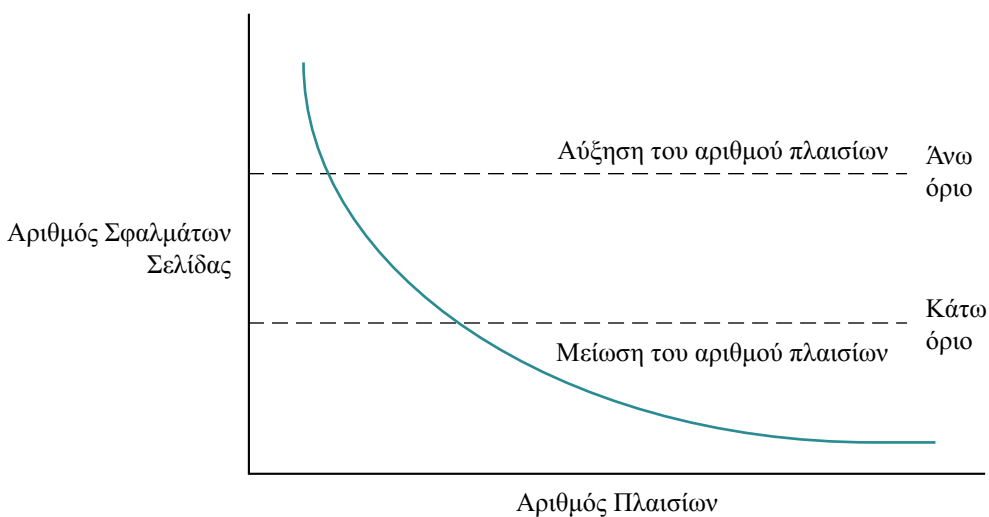
$$\Rightarrow F(\Delta) \cong w(\Delta+1) - w(\Delta)$$

5.8.3 Συχνότητα Σφαλμάτων Σελίδας

Το μοντέλο του συνόλου εργασίας είναι αρκετά επιτυχημένο και η γνώση του συνόλου εργασίας μπορεί να είναι χρήσιμη κατά την πρόωρη σελιδοποίηση (Ενότητα 5.9.1), αλλά μοιάζει με αδέξιο τρόπο για τον έλεγχο του λυγισμού. Η στρατηγική της «Συχνότητας των Σφαλμάτων Σελίδας» (Page Fault Frequency – PFF) χρησιμοποιεί μια πιο ευθεία προσέγγιση.

Το συγκεκριμένο πρόβλημα είναι η εμπόδιση του λυγισμού. Ο λυγισμός είναι υψηλός ρυθμός σφαλμάτων σελίδας. Συνεπώς, θέλουμε να ελέγξουμε το ρυθμό σφαλ-

μάτων σελίδας. Όταν είναι πολύ υψηλός, τότε ξέρουμε ότι η διαδικασία χρειάζεται περισσότερα πλαίσια. Όμοια, αν ο ρυθμός των σφαλμάτων σελίδας είναι πολύ χαμηλός, τότε η διαδικασία μπορεί να έχει πολλά πλαίσια. Μπορούμε να καθορίσουμε άνω και κάτω όρια για τον επιθυμητό ρυθμό σφαλμάτων σελίδας (Σχήμα 5.17). Αν ο πραγματικός αριθμός σφαλμάτων σελίδας ξεπερνάει το άνω όριο, τότε δίνουμε σε αυτή τη διαδικασία άλλο ένα πλαίσιο, αν ο ρυθμός των σφαλμάτων σελίδας πέσει κάτω από το κάτω όριο, αφαιρούμε ένα πλαίσιο από αυτή τη διαδικασία. Έτσι, μπορούμε άμεσα να μετράμε και να ελέγχουμε το ρυθμό των σφαλμάτων σελίδας ώστε να εμποδίσουμε το λυγισμό.



Σχήμα 5.17
Συχνότητα σφαλμάτων σελίδας

Όπως και με τη στρατηγική του συνόλου εργασίας, μπορεί να χρειαστεί να σταματήσουμε κάποια διαδικασία. Αν αυξηθεί ο ρυθμός των σφαλμάτων σελίδας και δεν υπάρχουν ελεύθερα πλαίσια, πρέπει να διαλέξουμε κάποια διαδικασία και να τη σταματήσουμε. Τα ελευθερωμένα πλαίσια διαμοιράζονται σε διαδικασίες με υψηλό ρυθμό σφαλμάτων.

5.9 Άλλοι Παράγοντες

Η επιλογή ενός αλγόριθμου αντικατάστασης και μιας πολιτικής κατανομής είναι οι κύριες αποφάσεις που παίρνονται για ένα σύστημα σελιδοποίησης, αλλά υπάρχουν και πολλοί άλλοι παράγοντες.

5.9.1 Πρόωρη Σελιδοποίηση

Μια εμφανής ιδιότητα ενός συστήματος σελιδοποίησης καθαρής ζήτησης είναι ο μεγάλος αριθμός σφαλμάτων σελίδας που γίνονται όταν ξεκινάει η διαδικασία. Αυτή

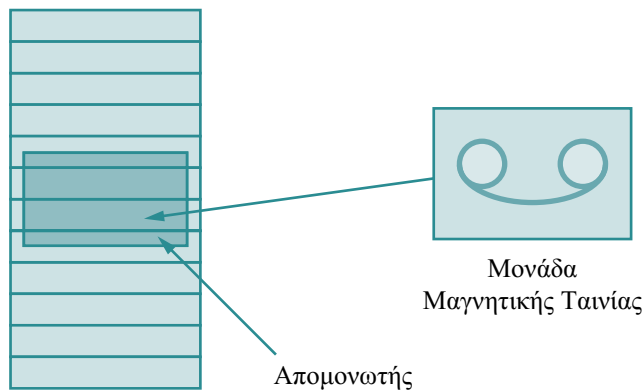
η κατάσταση είναι αποτέλεσμα της προσπάθειας να έρθει η αρχική τοπικότητα στη μνήμη. Το ίδιο μπορεί να συμβεί και άλλες στιγμές. Για παράδειγμα, όταν μια διαδικασία που έχει «εναλλαχθεί έξω» επαναρχίζει, όλες της οι σελίδες είναι στην επικουρική μνήμη και καθεμία πρέπει να μεταφερθεί μέσω του δικού της σφάλματος σελίδας. Η «πρόωρη σελιδοποίηση» (prepaging) είναι μια προσπάθεια να εμποδιστεί αυτή η υψηλού επιπέδου αρχική σελιδοποίηση. Η στρατηγική είναι να μεταφερθούν στη μνήμη μεμιάς όλες οι σελίδες που θα χρειαστούν.

Σε ένα σύστημα που χρησιμοποιεί το μοντέλο του συνόλου εργασίας, κρατάμε με κάθε διαδικασία μια λίστα των σελίδων που ανήκουν στο σύνολο εργασίας της. Αν πρέπει να σταματήσουμε μια διαδικασία (εξαιτίας μιας αναμονής για I/O ή λόγω έλλειψης ελεύθερων πλαισίων), θυμόμαστε το σύνολο εργασίας αυτής της διαδικασίας. Όταν πρόκειται να επαναρχίσει αυτή η διαδικασία (περάτωση της I/O ή αρκετά ελεύθερα πλαίσια), αυτόματα φέρνουμε στη μνήμη όλο το σύνολο εργασίας πριν επαναρχίσουμε τη διαδικασία.

Η πρόωρη σελιδοποίηση μπορεί, σε μερικές περιπτώσεις, να αποτελέσει πλεονέκτημα. Η ερώτηση είναι απλώς αν το κόστος της πρόωρης σελιδοποίησης είναι μικρότερο από το κόστος εξυπηρέτησης των αντίστοιχων σφαλμάτων σελίδας. Υπάρχει η περίπτωση πολλές από τις σελίδες που μεταφέρθηκαν στη μνήμη μέσω πρόωρης σελιδοποίησης να μη χρησιμοποιηθούν. Υποθέστε ότι s σελίδες σελιδοποιούνται πρόωρα και ότι μόνο ένα κλάσμα α αυτών των s σελίδων χρησιμοποιούνται στην πραγματικότητα ($0 \leq \alpha \leq 1$). Το ερώτημα τώρα είναι αν το κόστος των αs κερδισμένων σφαλμάτων σελίδας είναι μικρότερο ή μεγαλύτερο από το κόστος πρόωρης σελιδοποίησης $(1 - \alpha)s$ μη αναγκαίων σελίδων. Αν το α είναι κοντά στο μηδέν, τότε χάνει η πρόωρη σελιδοποίηση, αν το α είναι κοντά στο 1, τότε κερδίζει.

5.9.2 Κλείδωμα Σελίδων για I/O (I/O Interlock)

Όταν χρησιμοποιείται σελιδοποίηση ζήτησης, είναι μερικές φορές αναγκαίο να επιτρέψουμε το κλείδωμα μερικών σελίδων στη μνήμη. Μια τέτοια κατάσταση παρουσιάζεται όταν γίνεται I/O από ή προς την (ιδεατή) μνήμη χρήστη. Η I/O εργασία συνήθως γίνεται από έναν ξεχωριστό I/O επεξεργαστή. Για παράδειγμα, ένας «ελεγκτής μαγνητικής ταινίας» (magnetic tape controller) δέχεται τον αριθμό των λέξεων (ή bytes) προς μεταφορά και τη διεύθυνση μνήμης του απομονωτή (Σχήμα 5.18). Όταν τελειώσει η μεταφορά, η CPU διακόπτεται.

**Σχήμα 5.18**

Τα πλαίσια που χρησιμοποιούνται για I/O πρέπει να κρατούνται στη μνήμη

Πρέπει να αποτρέψουμε το εξής: Μια διαδικασία εκδίδει μια αίτηση για I/O και μπαίνει στην ουρά για την I/O μονάδα. Εν τω μεταξύ, η CPU δίνεται σε άλλες διαδικασίες. Αυτές οι διαδικασίες δημιουργούν σφάλματα σελίδας και χρησιμοποιώντας έναν αλγόριθμο γενικής αντικατάστασης μια από αυτές αντικαθιστά τη σελίδα που περιέχει τον απομονωτή για τη διαδικασία που βρίσκεται εν αναμονή. Μετά οι σελίδες εναλλάσσονται έξω. Αργότερα, όταν η αίτηση για I/O προωθηθεί στην κεφαλή της ουράς για τη μονάδα, γίνεται η I/O στην καθορισμένη διεύθυνση. Όμως, αυτό το πλαίσιο χρησιμοποιείται τώρα από μια διαφορετική σελίδα που ανήκε σε μια άλλη διαδικασία.

Υπάρχουν δύο κοινές λύσεις γι' αυτό το πρόβλημα. Μια είναι να μην εκτελείται ποτέ η I/O σε μνήμη του χρήστη. Αντίθετα, τα δεδομένα αντιγράφονται πάντα μεταξύ της μνήμης συστήματος και της μονάδας I/O. Για να γραφτεί ένα κομμάτι μνήμης στην ταινία, αυτό πρώτα αντιγράφεται στη μνήμη συστήματος και μετά στην ταινία.

Αυτή η πλεονάζουσα αντιγραφή μπορεί να καταλήξει σε απαράδεκτα υψηλό κόστος. Μια άλλη λύση είναι να επιτραπεί το κλειδωμά σελίδων στη μνήμη. Ένα *bit* κλειδώματος συσχετίζεται με κάθε πλαίσιο. Αν το πλαίσιο είναι κλειδωμένο, δε διαλέγεται για αντικατάσταση. Κάτω από αυτή την προσέγγιση, για να γραφτεί ένα κομμάτι μνήμης στην ταινία, οι σελίδες που περιέχουν τα δεδομένα κλειδώνονται στη μνήμη. Έτσι, το σύστημα συνεχίζει κανονικά. Οι κλειδωμένες σελίδες δεν μπορούν να αντικατασταθούν. Όταν τελειώσει η I/O, οι σελίδες ξεκλειδώνονται.

Μια άλλη χρήση του *bit* κλειδώματος υπάρχει στην κανονική αντικατάσταση σελίδας. Θεωρήστε την εξής ακολουθία γεγονότων: Μια χαμηλής προτεραιότητας διαδικασία δημιουργεί σφάλμα σελίδας. Διαλέγοντας ένα πλαίσιο αντικατάστασης, το σύστημα σελιδοποίησης εναλλάσσει την απαραίτητη σελίδα μέσα στη μνήμη. Έτοιμη να συνεχίσει, η χαμηλής προτεραιότητας διαδικασία εισέρχεται στην ουρά έτοι-

μων διαδικασιών και περιμένει για την CPU. Επειδή είναι χαμηλής προτεραιότητας, μπορεί ο χρονοδρομολογητής της CPU να μην τη διαλέξει για αρκετό χρονικό διάστημα. Ενόσω περιμένει αυτή, μια υψηλής προτεραιότητας διαδικασία δημιουργεί ένα σφάλμα σελίδας. Ψάχνοντας για αντικατάσταση, το σύστημα σελιδοποίησης βλέπει μια σελίδα στη μνήμη που δεν έχει αναφερθεί ή τροποποιηθεί, είναι η σελίδα που η χαμηλής προτεραιότητας διαδικασία έχει φέρει. Μοιάζει να είναι ο τέλειος αντικαταστάτης, είναι «καθαρή» και δε θα χρειαστεί να αντιγραφεί, και φαινομενικά δεν έχει χρησιμοποιηθεί για πολύ καιρό.

Η απόφαση για το αν η υψηλής προτεραιότητας διαδικασία θα πρέπει να αντικαταστήσει τη χαμηλής προτεραιότητας διαδικασία είναι απόφαση πολιτικής. Τελικά, το μόνο που κάνουμε είναι να καθυστερούμε τη χαμηλής προτεραιότητας διαδικασία προς όφελος της διαδικασίας υψηλής προτεραιότητας. Από την άλλη μεριά όμως, σπαταλούμε την προσπάθεια που ξοδεύτηκε για να φέρουμε τη σελίδα της διαδικασίας χαμηλής προτεραιότητας. Αν αποφασίσουμε να εμποδίσουμε την αντικατάσταση μιας σελίδας που μόλις «εναλλάχθηκε μέσα» έως ότου αυτή χρησιμοποιηθεί τουλάχιστον μια φορά, τότε μπορούμε να χρησιμοποιήσουμε το bit κλειδώματος για να υλοποιήσουμε αυτή την πολιτική. Όταν μια σελίδα επιλεγεί προς αντικατάσταση, το bit κλειδώματός της τίθεται και παραμένει έτσι έως ότου η διαδικασία χρονοδρομολογηθεί ξανά.

Η χρήση του bit κλειδώματος μπορεί να γίνει επικίνδυνη αν αυτό τίθεται και δεν επαναφέρεται ποτέ. Αν συμβεί αυτό (εξαιτίας κάποιου λάθους στο ΛΣ), το κλειδωμένο πλαίσιο αχρηστεύεται.

5.9.3 Μέγεθος Σελίδας

Οι σχεδιαστές ενός ΛΣ για μια ήδη υπάρχουσα μηχανή σπάνια έχουν επιλογή για το μέγεθος της σελίδας. Όμως, όταν σχεδιάζονται νέες μηχανές, πρέπει να παρθεί μια απόφαση για το καλύτερο μέγεθος της σελίδας – δεν υπάρχει ένα καλύτερο μέγεθος σελίδας. Μάλλον υπάρχει ένα σύνολο από παράγοντες που υποστηρίζουν διάφορα μεγέθη. Τα μεγέθη σελίδων είναι πάντοτε δυνάμεις του δύο, και γενικά ποικίλλουν από 256 (2⁸) έως 4096 (2¹²) bytes ή λέξεις.

Πώς διαλέγουμε ένα μέγεθος σελίδας; Ένας παράγοντας είναι το μέγεθος του πίνακα σελίδων. Για ένα δεδομένο χώρο ιδεατής μνήμης, η ελάττωση του μεγέθους της σελίδας αυξάνει τον αριθμό των σελίδων και, φυσικά, το μέγεθος του πίνακα σελίδων.

Για μια ιδεατή μνήμη των 4 μεγα-λέξεων (2²²) θα υπήρχαν 16.384 σελίδες των 256 λέξεων, αλλά μόνο 1024 σελίδες των 4096 λέξεων. Εφόσον κάθε ενεργός διαδικα-

σία πρέπει να έχει το δικό της αντίγραφο του πίνακα σελίδων, βλέπουμε ότι είναι επιθυμητό ένα μεγάλο μέγεθος σελίδας.

Από την άλλη μεριά, η μνήμη χρησιμοποιείται αποδοτικότερα με μικρές σελίδες. Αν μια διαδικασία παίρνει μνήμη από τη θέση 00000 έως ότου έχει πάρει όση χρειάζεται, είναι απίθανο ότι θα τελειώσει ακριβώς πάνω σε ένα «όριο σελίδας» (page boundary). Έτσι, πρέπει να του κατανεμηθεί ένα μέρος της τελευταίας σελίδας (αφού οι σελίδες είναι η μονάδα κατανομής), αλλά θα είναι αχρησιμοποίητο (εσωτερική κλασματοποίηση). Υποθέτοντας ανεξαρτησία μεταξύ μεγέθους διαδικασίας και μεγέθους σελίδας, θα περιμέναμε ότι, κατά μέσο όρο, θα πήγαινε χαμένη η μισή τελευταία σελίδα κάθε διαδικασίας. Αυτό το χάσιμο θα ήταν μόνο 128 λέξεις για μια σελίδα των 256 λέξεων, αλλά 2048 λέξεις για μια σελίδα των 4096 λέξεων. Για να ελαχιστοποιήσουμε την εσωτερική κλασματοποίηση, χρειαζόμαστε μικρό μέγεθος σελίδας.

Ένα άλλο πρόβλημα είναι ο χρόνος που χρειάζεται για να διαβαστεί ή γραφτεί μια σελίδα. Ο χρόνος της I/O συντίθεται (για μια μονάδα σταθερών κεφαλαίων) από το χρόνο latency και το χρόνο μεταφοράς. Ο τελευταίος είναι ανάλογος του όγκου που μεταφέρεται (δηλαδή του μεγέθους σελίδας), ένας παράγοντας που φαίνεται να υποστηρίζει μικρό μέγεθος σελίδας. Θυμηθείτε όμως ότι ο χρόνος latency συνήθως επισκιάζει το χρόνο μεταφοράς. Με ένα ρυθμό μεταφοράς 256.000 λέξεων/sec, η μεταφορά 512 λέξεων παίρνει μόνο 2msec. Ο χρόνος latency όμως μπορεί να είναι και 8msec. Η μείωση του μεγέθους σελίδας σε 128 λέξεις μειώνει το συνολικό χρόνο για I/O από 10 σε 9msec. Ο διπλασιασμός του μεγέθους σελίδας αυξάνει τον I/O χρόνο σε μόνο 12msec. Χρειάζονται, δηλαδή, 12msec για να διαβαστεί μια μόνο σελίδα των 1024 λέξεων, αλλά χρειάζονται 36msec για να διαβαστεί το ίδιο μέγεθος σαν 4 σελίδες των 256 λέξεων η καθεμία. Έτσι, η επιθυμία μείωσης του χρόνου I/O υποστηρίζει ένα μεγάλο μέγεθος σελίδας.

Όμως, με μικρό μέγεθος σελίδας η συνολική I/O θα πρέπει να μειωθεί, αφού βελτιώνεται η τοπικότητα. Ένα μικρό μέγεθος σελίδας επιτρέπει σε κάθε σελίδα να ταιριάζει καλύτερα την τοπικότητα της διαδικασίας. Για παράδειγμα, θεωρήστε μια διαδικασία των 20K λέξεων, από τις οποίες μόνο οι μισές (10K) χρησιμοποιούνται πραγματικά σε μια εκτέλεσή του. Αν έχουμε μόνο μια μεγάλη σελίδα, πρέπει να τη μεταφέρουμε ολόκληρη. Αν είχαμε σελίδες μιας μόνο λέξης, τότε θα μπορούσαμε να μεταφέρουμε μόνο 10.000 λέξεις που πραγματικά χρησιμοποιούνται, με αποτέλεσμα μόνο αυτές οι 10.000 λέξεις να μεταφερθούν και να κατανεμηθούν. Με μικρό μέγεθος σελίδας έχουμε καλύτερη ευελιξία, η οποία μας επιτρέπει να απομονώσουμε μόνο τη μνήμη που χρειαζόμαστε. Με μεγάλο μέγεθος σελίδας πρέπει να δεσμεύσουμε και να μεταφέρουμε όχι μόνο ό,τι χρειάζεται, αλλά και ό,τι άλλο τυχαίνει να

είναι στη σελίδα, είτε είναι αναγκαίο είτε όχι. Έτσι, το μικρό μέγεθος σελίδας θα είχε ως αποτέλεσμα λιγότερη I/O και λιγότερη συνολικά κατανεμημένη μνήμη.

Μήπως όμως παρατηρήσατε ότι με το μέγεθος σελίδας ίσο με 1 λέξη θα είχαμε ένα σφάλμα σελίδας για κάθε λέξη; Μια διαδικασία των 20K που χρησιμοποιεί μόνο τα μισά από αυτά θα δημιουργούσε μόνο ένα σφάλμα σελίδας αν η σελίδα είχε μέγεθος 20K, αλλά 10.000 σφάλματα σελίδας αν η σελίδα είχε μέγεθος 1 λέξη. Κάθε σφάλμα σελίδας δημιουργεί το μεγάλο κόστος που χρειάζεται για να σωθούν οι καταχωρητές, να αντικατασταθεί η σελίδα, να μπει στην ουρά της μονάδας σελιδοποίησης η αίτηση αντικατάστασης και να ενημερωθούν οι πίνακες. Για να μειώσουμε τον αριθμό των σφαλμάτων σελίδας, πρέπει να έχουμε μεγάλο μέγεθος σελίδας.

Υπάρχουν και άλλοι παράγοντες που πρέπει να παρθούν υπόψη [όπως η σχέση μεταξύ μεγέθους σελίδας και μεγέθους «τομέα» (sector) στην μονάδα σελιδοποίησης], αλλά το πρόβλημα δεν έχει βέλτιστη απάντηση. Μερικοί παράγοντες (εσωτερική κλασματοποίηση, τοπικότητα) υποστηρίζουν μικρό μέγεθος σελίδας, ενώ άλλοι (μέγεθος πίνακα, χρόνος I/O) υποστηρίζουν μεγάλο μέγεθος σελίδας. Για να απεικονίσουμε το πρόβλημα, υπάρχουν δύο συστήματα που επιτρέπουν δύο διαφορετικά μεγέθη σελίδων. Το υλικό του Multics (GE 645) επιτρέπει σελίδες των 64 ή των 1024 λέξεων. Το IBM/370 επιτρέπει σελίδες των 2K ή των 4K bytes. Η δυσκολία επιλογής μεγέθους σελίδας σκιαγραφείται από το γεγονός ότι το MVS στον IBM/370 διάλεξε 4K σελίδες, ενώ το VS/1 διάλεξε 2K σελίδες.

5.9.4 Δομή Διαδικασίας (Προγράμματος)

Η σελιδοποίηση ζήτησης έχει σχεδιαστεί έτσι ώστε να είναι διαφανής προς το πρόγραμμα του χρήστη. Σε πολλές περιπτώσεις, ο χρήστης είναι εντελώς ανύποπτος ως προς τη σελιδοποιημένη φύση της μνήμης. Σε άλλες περιπτώσεις, όμως, η απόδοση του συστήματος μπορεί να βελτιωθεί αν ληφθεί υπόψη η σελιδοποίηση ζήτησης.

Για παράδειγμα, υποθέστε ότι το μέγεθος σελίδας είναι 128 λέξεις. Επίσης, υποθέστε ότι έχουμε ένα πρόγραμμα σε Pascal που αρχικοποιεί έναν πίνακα 128×128 θέσεων στην τιμή. Ο ακόλουθος κώδικας είναι ένα τυπικό παράδειγμα.

```
var A: array [1...128] of array [1...128] of integer;
```

```
for j := 1 to 128
```

```
do for i :=1 to 128
```

```
do A[i] [j] := 0;
```

Αυτός ο κώδικας φαίνεται αρκετά αθώος, αλλά σκεφτείτε ότι ο πίνακας αποθηκεύεται σειρά με σειρά. Δηλαδή ο πίνακας αποθηκεύεται: $A[1][1]$, $A[1][2]$, ..., $A[1][128]$, $A[2][1]$, $A[2][2]$, ..., $A[125][128]$. Για σελίδες των 128 λέξεων, κάθε σειρά καταλαμβάνει μια σελίδα. Έτσι, ο παραπάνω κώδικας μηδενίζει μια λέξη σε κάθε σελίδα, μετά άλλη μια σε κάθε σελίδα και ούτω καθεξής, με αποτέλεσμα $18 \times 128 = 16.384$ σφάλματα σελίδας.

Αλλάζοντας τον κώδικα σε:

```
var A : array [1...128] of array [1...128] of integer;
```

```
for i := 1 to 128
```

```
  do for j := 1 to 128
```

```
    do A[i][j] := 0;
```

μηδενίζονται όλες οι λέξεις σε μια σελίδα πριν να ξεκινήσουμε την επόμενη, μειώνοντας έτσι τον αριθμό των σφαλμάτων σελίδας σε 128.

Προσεκτική επιλογή των δομών δεδομένων και των προγραμματιστικών δομών μπορεί να αυξήσει την τοπικότητα, και έτσι να μειώσει το ρυθμό σφαλμάτων σελίδας και τον αριθμό των σελίδων που ανήκουν στο σύνολο εργασίας. Το stack έχει καλή τοπικότητα, αφού η προσπέλασή του γίνεται πάντα στην κορυφή του. Αντίθετα, ένας πίνακας hash είναι σχεδιασμένος να διασκορπίζει τις αναφορές, παράγοντας κακή τοπικότητα.

Ο «μεταφραστής» (compiler) και ο «φορτωτής» (loader) μπορούν να επηρεάσουν σημαντικά τη σελιδοποίηση. Ο διαχωρισμός του κώδικα και των δεδομένων και η παραγωγή «επανεισαγόμενου» (reentrant) κώδικα σημαίνει ότι οι σελίδες κώδικα μπορούν να είναι μόνο ανάγνωσης, και έτσι δε θα τροποποιηθούν ποτέ. Οι «καθαρές» σελίδες δε χρειάζεται να «εναλλαχθούν έξω» πριν αντικατασταθούν. Ο φορτωτής μπορεί να αποφύγει να τοποθετήσει ρουτίνες πάνω σε όρια σελίδων, κρατώντας κάθε ρουτίνα σε μια σελίδα.

Ρουτίνες που καλούν η μια την άλλη πολλές φορές μπορούν να μουν στην ίδια σελίδα. Αυτό είναι μια παραλλαγή του bin – packing προβλήματος: προσπαθήστε να βάλετε τα μεταβλητού μεγέθους «τμήματα φόρτωσης» (load segments) σε σταθερού μεγέθους σελίδες έτσι ώστε να ελαχιστοποιηθούν οι μεταξύ σελίδων αναφορές. Αυτή η προσέγγιση είναι ιδιαίτερα χρήσιμη για σελίδες μεγάλου μεγέθους.

5.9.5 Ιεραρχία Αποθήκευσης

Πρέπει να υπογραμμίσουμε ότι η συζήτησή μας επικεντρώθηκε σε ένα μόνο από τα επίπεδα αποθήκευσης που υπάρχουν στα πιο πολλά συστήματα. Πολλοί υπολογιστές έχουν τώρα ως κύρια μνήμη μια υψηλής ταχύτητας cache μνήμη. Όταν γίνεται μια προσπέλαση στη μνήμη, τα περιεχόμενα της προσπελάσιμης θέσης, συν τους γείτονές της, αντιγράφονται ως cache. Αν γίνει και άλλη αναφορά σε αυτές τις θέσεις, αυτές μπορούν να μεταφερθούν κατευθείαν από την cache, χωρίς να χρειαστεί να πάμε στην πιο αργή κύρια μνήμη. Για μια λογικού μεγέθους cache, ένας ρυθμός επιτυχίας της τάξης του 80% είναι αρκετά συνηθισμένος.

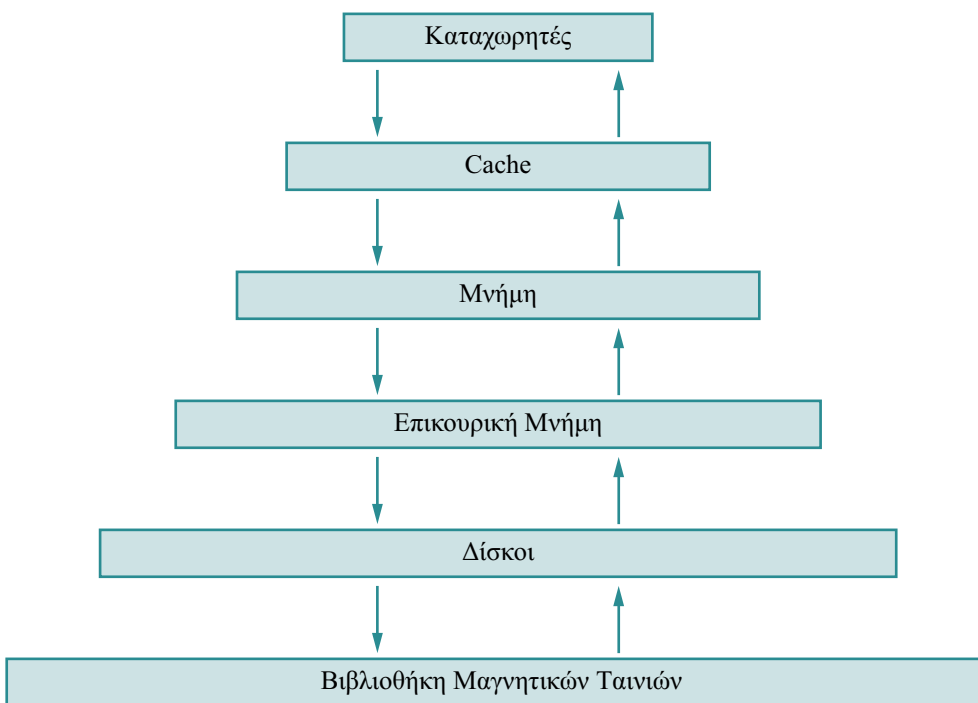
Ως ιστορική παρατήρηση, ο πρώτος υπολογιστής με σελιδοποίηση ήταν ο Atlas. Για κύρια μνήμη είχε ένα τύμπανο των 96K. Οι μνήμες τυμπάνων ήταν αρκετά συνηθισμένες στα τέλη του 1950 και χρησιμοποιούνταν επίσης και στον IBM 630. Ο Atlas είχε και 16K λέξεις μιας νέας, πιο γρήγορης, μνήμης, που ονομαζόταν «μνήμη μαγνητικού πυρήνα» (magnetic core memory). Αυτή ήταν ακόμα πειραματική και πολύ ακριβή. Χρησιμοποιούνταν δε ως cache για το τύμπανο. Οι τεχνικές σελιδοποίησης αναπτύχθηκαν για την υλοποίηση του διαχειρισμού της cache.

Επεκτείνοντας αυτή την άποψη, οι εσωτερικοί προγραμματιζόμενοι καταχωρητές, όπως οι συσσωρευτές, αποτελούν μια υψηλής ταχύτητας cache για την κύρια μνήμη. Ο προγραμματιστής (ή ο μεταφραστής) υλοποιεί τους αλγόριθμους κατανομής και αντικατάστασης των καταχωρητών (σελίδων) για να αποφασίσει τι πληροφορία θα κρατήσει στην πρωταρχική μνήμη (καταχωρητές) και τι θα κρατήσει στην επικουρική μνήμη (κύρια μνήμη). Ο βέλτιστος αλγόριθμος αντικατάστασης σελίδας χρησιμοποιείται συχνά, αφού ο προγραμματιστής ή ο μεταφραστής μπορούν να δουν τι θα χρειαστεί στο μέλλον.

Στην άλλη κατεύθυνση, το σύστημα αρχείων μπορεί να ιδωθεί ως η επικουρική μνήμη για το σύστημα σελιδοποίησης. Τα αρχεία μεταφέρονται από το σύστημα αρχείων στη μονάδα σελιδοποίησης όταν αυτά αναφερθούν («μεταφορά ζήτησης» – demand transfer). Το ίδιο σύστημα αρχείων μπορεί να εμπεριέχει πολλά επίπεδα αποθήκευσης. Ο πιο γρήγορος (αλλά περιορισμένος) δίσκος μπορεί να υποβοηθείται από την πιο μεγάλη (αλλά αργή) αποθήκευση σε ταινία. Οι μεταφορές μεταξύ αυτών των δύο επιπέδων αποθήκευσης συνήθως ζητούνται ιδιαίτερα, αλλά πολλά συστήματα, τώρα πια, αυτόματα αποθηκεύουν ένα αρχείο το οποίο δεν έχει χρησιμοποιηθεί για αρκετό καιρό (ένα μήνα) και το επαναφέρουν αυτόματα στο δίσκο μόλις αυτό αναφερθεί.

Μπορούμε να δούμε μια ποικιλία αποθήκευσης σ' ένα υπολογιστικό σύστημα που

μπορεί να οργανωθεί σε μια ιεραρχία (Σχήμα 5.19). Τα υψηλότερα επίπεδα είναι ακριβά, αλλά πολύ γρήγορα. Καθώς προχωρούμε προς τα κάτω στην ιεραρχία, το κόστος ανά bit μειώνεται, ενώ αυξάνεται ο χρόνος προσπέλασης και ο όγκος αποθήκευσης σε κάθε επίπεδο. Η κίνηση πληροφορίας μεταξύ επιπέδων μπορεί να είναι άμεση ή έμμεση. Οι αλγόριθμοι σελιδοποίησης είναι ικανοποιητικοί για πολλές από αυτές τις μεταφορές.



Σχήμα 5.19
Ιεραρχίες μνήμης

Αυτή η όψη της αποθήκευσης έφτασε στη φυσική της κατάληξη στο σύστημα Multics. Αυτό προσφέρει έναν πολύ μεγάλο τμηματοποιημένο χώρο διευθύνσεων. Τα τμήματα έχουν ονόματα και όλα τα αρχεία είναι τμήματα. Έτσι, άμεσες εντολές I/O δεν είναι αναγκαίες. Ένα αρχείο μπορεί να προσπελαστεί με τον καθορισμό του ονόματός του ως ονόματος τμήματος και τη φόρτωση ή αποθήκευση κατευθείαν στο τμήμα. Η αναφορά θα προκαλέσει ένα σφάλμα τμήματος (και μετά ένα σφάλμα σελίδας), που θα μεταφέρει την επιθυμητή πληροφορία από το σύστημα αρχείων στη μνήμη.

Σύνοψη

Το ζητούμενο είναι η εκτέλεση μιας διαδικασίας της οποίας ο χώρος λογικών διευθύνσεων είναι μεγαλύτερος από το διαθέσιμο χώρο φυσικών διευθύνσεων. Θα μπορούσε ο προγραμματιστής να καταστήσει μια τέτοια διαδικασία εκτελέσιμη, επαναδομώντας την με επικαλύψεις, αλλά αυτό γενικά είναι δύσκολο. Η ιδεατή μνήμη είναι η τεχνική που επιτρέπει σ' ένα μεγάλο χώρο λογικών διευθύνσεων να χαρτογραφηθεί πάνω σε μια μικρότερη φυσική μνήμη. Επίσης, επιτρέπει σε πολύ μεγάλες διαδικασίες να τρέξουν και επιτρέπει την αύξηση του βαθμού του πολυπρογραμματισμού, αυξάνοντας έτσι τη χρήση της CPU.

Η καθαρή σελιδοποίηση ζήτησης δε μεταφέρει μια σελίδα πριν αυτή ζητηθεί. Η πρώτη αναφορά προκαλεί ένα σφάλμα σελίδας προς τον παραμένοντα επόπτη ΛΣ. Το ΛΣ συμβουλευέται έναν εσωτερικό πίνακα για να καθορίσει τη θέση της σελίδας στην επικουρική μνήμη. Μετά βρίσκει ένα ελεύθερο πλαίσιο και μεταφέρει σ' αυτό τη σελίδα από την επικουρική μνήμη. Ο πίνακας σελίδων ενημερώνεται ώστε να αντικατοπτρίζει αυτή την αλλαγή και η εντολή που προκάλεσε το σφάλμα σελίδας επανεκτελείται. Αυτή η προσέγγιση επιτρέπει σε μια διαδικασία να τρέχει παρ' όλο που δε βρίσκεται στην κύρια μνήμη ολόκληρο το «αντίγραφο μνήμης» της (memory image). Όσο ο ρυθμός των σφαλμάτων σελίδας είναι μικρός, η απόδοση είναι ανεκτή.

Η σελιδοποίηση ζήτησης μπορεί να χρησιμοποιηθεί για να μειώσει τον αριθμό των πλαισίων που κατανέμονται σε μια διαδικασία. Αυτή η χρήση μπορεί να αυξήσει το βαθμό πολυπρογραμματισμού και τη χρήση της CPU. Επίσης, επιτρέπει σε διαδικασίες να τρέχουν μόλο που οι απαιτήσεις τους για μνήμη ξεπερνούν τη συνολική διαθέσιμη φυσική μνήμη. Τέτοιες διαδικασίες τρέχουν στην ιδεατή μνήμη.

Αν οι συνολικές απαιτήσεις μνήμης ξεπερνούν τη φυσική μνήμη, τότε ίσως να είναι αναγκαίο να αντικατασταθούν σελίδες, έτσι ώστε να ελευθερωθούν πλαίσια για τις νέες σελίδες. Χρησιμοποιούνται διάφοροι αλγόριθμοι αντικατάστασης σελίδας. Ο FIFO είναι εύκολο να προγραμματιστεί, αλλά υποφέρει από την ανωμαλία του Belady. Ο βέλτιστος αλγόριθμος χρειάζεται γνώση του μέλλοντος. Ο LRU είναι μια προσέγγιση του βέλτιστου, αλλά μπορεί να είναι δύσκολο να υλοποιηθεί. Οι περισσότεροι αλγόριθμοι, όπως αυτός της δεύτερης ευκαιρίας, είναι προσεγγίσεις του LRU.

Επιπρόσθετα, είναι απαραίτητη μια πολιτική κατανομής πλαισίων. Η κατανομή μπορεί να είναι σταθερή, προτείνοντας τοπική αντικατάσταση σελίδας, ή δυναμική, προτείνοντας σφαιρική αντικατάσταση. Το μοντέλο του συνόλου εργασίας υποθέτει ότι οι διαδικασίες εκτελούνται σε τοπικές. Το σύνολο εργασίας είναι το σύνολο των σελίδων στην ισχύουσα τώρα τοπικότητα. Αντίστοιχα, κάθε διαδικασία πρέπει να έχει

αρκετά πλαίσια, ώστε να εξυπηρετεί το σύνολο εργασίας της.

Αν μια διαδικασία δε διαθέτει αρκετή μνήμη για το σύνολο εργασίας της, θα περιέλθει σε λυγισμό. Η διάθεση αρκετών πλαισίων σε κάθε διαδικασία ώστε να αποφευχθεί ο λογισμός μπορεί να απαιτεί εναλλαγή διαδικασιών και χρονοδρομολόγηση.

Πέρα από τα μεγάλα προβλήματα της αντικατάστασης σελίδας και κατανομής πλαισίων, η σωστή σχεδίαση ενός συστήματος σελιδοποίησης πρέπει να απαντά και στα θέματα του μεγέθους σελίδας, της I/O, του κλειδώματος της πρόωρης σελιδοποίησης και άλλων. Η ιδεατή μνήμη μπορεί να θεωρηθεί και ως ένα επίπεδο στην ιεραρχία αποθήκευσης μέσα σ' ένα υπολογιστικό σύστημα. Κάθε επίπεδο έχει το δικό του χρόνο προσπέλασης, μέγεθος και τις δικές του παραμέτρους κόστους.

Βιβλιογραφία κεφαλαίου

ΠΡΟΑΙΡΕΤΙΚΗΣ ΑΝΑΓΝΩΣΗΣ

Denning, *Virtual Memory*, *Computing surveys*, vol. 2, Sept. 1970, pp. 153–189.

Denning, *Working sets past and present*, IEEE trans. On software engineering, vol. SE-6, Jan 1980, pp. 64–84.

Knuth, *The art of computer programming, vol. 1: Fundamental algorithms*, 2nd edition, reading, MA: Addison–Wesley, 1973.

Silberschatz et al., *Operating System Concepts*, 3rd edition, reading, MA: Addison–Wesley, 1991.

Γλωσσάρι κεφαλαίου

Absolute memory image:	απόλυτες εικόνες μνήμης
Array:	διάνυσμα
Auto-decrement:	αυτόματη μείωση
Auto-increment:	αυτόματη αύξηση
Belady anomaly:	ανωμαλία του Belady
Cache memory:	μνήμη υψηλής ταχύτητας
Clearing:	επαναφορά
Clock register:	καταχωρητής ρολογιού
Compiler:	μεταφραστής
Counter:	μετρητής
Cpu scheduling:	χρονοδρομολόγηση CPU
Demand paging:	σελιδοποίηση ζήτησης
Demand segmentation:	τμηματοποίηση ζήτησης
Demand transfer:	μεταφορά ζήτησης
Device seek:	αναμονή έως ότου ολοκληρωθεί η εύρεση των δεδομένων στη μονάδα
Device queuing time:	χρόνος ουράς της μονάδας
Device service time:	χρόνος εξυπηρέτησης μονάδας
Dirty bit:	βρόμικο bit
Dynamic loading:	δυναμικό φόρτωμα
Effective access time:	τελικός χρόνος προσπέλασης
Fatal error:	σοβαρό λάθος
FIFO (First In First Out):	πρώτος μέσα, πρώτος έξω
Fixed interval timer interrupt:	διακοπή σταθερού διαστήματος
Free frame buffer:	απομονωτής ελεύθερων πλαισίων
Global replacement:	γενική αντικατάσταση
Head:	αρχή
Illegal address trap:	παγίδα παράνομης διεύθυνσης
In-core:	σε χρήση
Instruction fetch:	προσκόμιση εντολών

Intermediate cpu scheduling:	μεσαίου επιπέδου χρονοδρομολόγηση της CPU
Invalid address error:	λάθος άκυρης διεύθυνσης
I/O interlock:	κλείδωμα σελίδων για I/O
Latency time:	αναμονή έως ότου ολοκληρωθεί η εύρεση των δεδομένων στη μονάδα
LFU (Least Frequently Used):	λιγότερο συχνά χρησιμοποιούμενη
Loader:	φορτωτής
Load segment:	φόρτωμα τμήματος
Locality model:	μοντέλο τοπικότητας
Local replacement:	τοπική αντικατάσταση
LRU (Least Recently Used):	λιγότερο πρόσφατα χρησιμοποιούμενη
Magnetic core memory:	μνήμη μαγνητικού πυρήνα
Magnetic tape controller:	ελεγκτής μαγνητικής ταινίας
Memory access time:	χρόνος προσπέλασης μνήμης
Memory image:	αντίγραφο μνήμης, εικόνα μνήμης
MFU (Most Frequently Used):	πιο συχνά χρησιμοποιούμενη
Microcode:	μικροκώδικας
Operand address:	τελευταία διεύθυνση
OPTIMAL:	βέλτιστο
Overlays:	επικαλύψεις
Overlay driver:	οδηγός επικάλυψης
Page boundary:	όριο σελίδας
Page fault:	λάθος σελίδας
PFF (Page Fault Frequency):	συχνότητα σφαλμάτων σελίδας
Page fault rate:	ρυθμός σφαλμάτων σελίδας
Page replacement:	αντικατάσταση σελίδων
Pool:	δεξαμενή
Prepaging:	πρόωρη σελιδοποίηση
Pure:	γνήσιο, καθαρό
Reentrant:	επανεισαγόμενο
Reference bit:	bit αναφοράς
Reference string:	ακολουθία αναφορών

Relocatable linking loader:	φορτωτής μετατοπιζόμενης σύνδεσης
Response time:	χρόνος απόκρισης
Second chance replacement:	αντικατάσταση δεύτερης ευκαιρίας
Sector:	τομέας
Seek time:	χρόνος ψαξίματος
Segment:	τμήμα
Shift:	ολισθαίνω
Single user system:	σύστημα ενός χρήστη
Stack:	στοίβα
Stack algorithm:	αλγόριθμος στοίβας
Status register:	ειδικός καταχωρητής
Swapping device:	μονάδα εναλλαγής
Swapping overhead:	κόστος εναλλαγής
Table:	πίνακας
Tail:	τέλος
Thrashing:	λυγισμός
Throughput:	ρυθμός, απόδοση
Timer interrupt:	χρονομετρητής διακοπών
Trap:	παγίδα
Turnaround time:	χρόνος ανακύκλωσης
Utilization:	χρήση
Virtual address space:	ιδεατός χώρος διευθύνσεων
Working set model:	μοντέλο συνόλου εργασιών