Πανεπιστημιακές Παραδόσεις

«Εργαστήριο Λειτουργικών Συστημάτων»

Καθ. Π. Τριανταφύλλου

2003

Μέρος 1ο

Παραδοσιακά Λειτουργικά Συστήματα

Κεφάλαιο 1. Εισαγωγή σε Λειτουργικά Συστήματα

1.1 Γενικά

Συστήματα Υπολογιστών: CPUs, RAM, Δίσκοι, Τερματικά, ρολόϊ,... (Αυτό είναι το hardware).

Χωρίς το λογισμικό, αυτοί οι πόροι είναι σχεδόν αχρησιμοποίητοι.

Συστήματα Λογισμικού: **SW Συστήματος** (Λ.Σ., Compilers, Editors) και Εφαρμογές. Το system s/w αποτελεί τη βάση πάνω στην οποία στηρίζονται οι εφαρμογές.

Παράδειγμα:
Μία εφαρμογή χρησιμοποιεί τον editor για να φτιάξει ένα αρχείο που περιέχει ένα πρόγραμμα. Μετά χρησιμοποιεί τον compiler για να δημιουργηθεί το object code, δηλαδή ο κώδικας που μπορεί να τρέξει στη μηχανή. Οταν ο κώδικας τρέχει, το Λ.Σ. καλείται να χρησιμοποιήσει τους απαραίτητους πόρους για να προσφερθούν οι κατάλληλες υπηρεσίες.

Περίληψη Εκτέλεσης Προγράμματος

1. Το πρόγραμμα δημιουργείται, γίνεται compiled, παράγεται ο object code ο οποίος αποθηκεύεται σε ένα αρχείο (στη δευτεροβάθμια μνήμη).
2. Οταν το πρόγραμμα ζητείται να τρέξει, (μέρος του) πρέπει να αποθηκευτεί στη κύρια μνήμη.
3. Κάθε εντολή του προγράμματος «στέλνεται» (φορτώνεται) στο κεντρικό επεξεργαστή μαζί με τα δεδομένα (που μπορεί να αποθηκευτούν στους registers του CPU) και εκτελείται.

➔ απαιτείται διαχείριση κύριας μνήμης, δευτεροβάθμιας μνήμης, και του CPU. Τέτοιου είδους διαχειρήσεις αναλαμβάνει το Λ.Σ.

Συμπέρασμα:
Από τα παραπάνω προκύπτει ότι το Λ.Σ. είναι το κεντρικό κομμάτι λογισμικού ενός συστήματος computer το οποίο διαχειρίζεται τους υλικούς πόρους του.

Αυτή η θεώρηση όμως δεν αρκεί!

Παράδειγμα: I/O με δίσκους

Θεωρείστε ένα πρόγραμμα που θέλει να δημιουργήσει ένα αρχείο με δεδομένα. Τα

αρχεία (files) αποθηκεύονται σε μαγνητικούς δίσκους, οι οποίοι, εν ολίγοις, αποτελούνται από ένα ηλεκτρονικό τμήμα (ένα controller - επεξεργαστής) και ένα μαγνητικό τμήμα (επιφάνειες δίσκων πάνω στις οποίες γράφουν κεφαλές).

Ο επεξεργαστής δέχεται εντολές για να γράψει (write) ή να ανακτήσει δεδομένα (read) σε (από) συγκεκριμμένες διευθύνσεις, να μετακινήσει τις κεφαλές σε καινούριες διευθύνσεις, κ.λπ. Η επικοινωνία με τον controller απαιτεί το γράψιμο ειδικών εντολών καθώς και το γράψιμο των παραμέτρων σε συγκεκριμμένες διευθύνσεις στη RAM. Η δομή και η μορφή αυτών των εντολών διαφέρει από controller σε controller. Επιπλέον, όταν ο controller εκπληρώσει μία εντολή επιστρέφει ειδικούς κώδικες σε ειδική μορφή οι οποίοι πρέπει να αναλυθούν για να επαληθευθεί ότι όλα πήγαν καλά, κ.λπ. Αυτοί οι κώδικες επικοινωνίας είναι πολύπλοκοι, και τα προγράμματα επικοινωνίας με τέτοιες συσκευές περιφέρειας είναι επίσης πολύπλοκα (και μεγάλα)!

Το Λ.Σ. αναλαμβάνει να απαλλάξει τον προγραμματιστή από όλες αυτές τις δυσκολίες, επιτρέποντας του να ασχοληθεί με τις δυσκολίες του προγράμματος που γράφει.

Από αυτή τη δεύτερη θεώρηση, προκύπτει ότι το Λ.Σ. ενεργεί σαν μία μηχανή αφαίρεσης (abstraction machine) η οποία αφαιρεί όλες τις λεπτομέρειες που καθιστούν την χρήση των πόρων ενός υπολογιστή πολύπλοκη.

Αρα, το Λ.Σ. παρέχει δύο βασικές αφηρημένες έννοιες:

1. Διεργασίες (Processes)
2. Αρχεία (Files)

Με την έννοια της «διεργασίας» ασχολούνται εκείνες οι διαδικασίες του Λ.Σ. που καλούνται να διαχειριστούν τον CPU, RAM, Δίσκους, Τερματικά, κ.λπ. Αυτή η διαχείριση εστιάζει στην ίση κατανομή των υλικών πόρων του υπολογιστή σε όλα τα προγράμματα που τρέχουν σε μία δεδομένη στιγμή.

Με την έννοια του «αρχείου» ασχολούνται οι διαδικασίες του Λ.Σ. που παρέχουν την δυνατότητα δημιουργίας μη-προσωρινών δεδομένων (που αποθηκεύονται στη δευτεροβάθμια μνήμη). Αυτές οι διαδικασίες ασχολούνται επίσης και με την επικοινωνία με τις περιφερειακές συσκευές.

1.2 Ο πυρήνας του Λ.Σ. (Kernel)

Ο Kernel του Λ.Σ. αναφέρεται στο κύριο τμήμα του Λ.Σ. το οποίο υλοποιεί τις δύο βασικές οντότητες (processes και files) του Λ.Σ. Αυτός ο κώδικας είναι προστατευμένος, με την έννοια ότι δεν ανήκει σε κανένα χρήστη (user process). Διαφορετικά, ο κάθε χρήστης θα μπορούσε να αλλάξει αυτό τον κώδικα και έτσι να μονοπωλήσει τους πόρους του συστήματος, CPU, RAM, δίσκοι, τερματικά, κ.λπ.

Αυτή η προστασία επιτυγχάνεται χρησιμοποιώντας δύο τρόπους λειτουργίας: user mode και kernel mode. Το σύστημα βρίσκεται υπό προστασία όταν βρίσκεται σε kernel mode.

Δηλαδή, ειδικές εντολές που διαχειρίζονται τους πόρους του συστήματος μπορούν να εκτελεσθούν μόνο όταν το σύστημα λειτουργεί σε kernel mode.

Το Λ.Σ. είναι το μόνο system s/w το οποίο εκτελείτε σε kernel mode.

System Calls.

Για να μπορέσει ένα user process να χρησιμοποιήσει τους πόρους πρέπει να καλέσει τις κατάλληλες ρουτίνες του kernel. Αυτό επιτυγχάνεται μέσω system calls. Ετσι, υπάρχουν system calls όπως fork(), exec(), malloc(), read(), printf() τα οποία αφορούν στη δημιουργία ενός process, στην εκτέλεση από κάποιο process ενός προγράμματος, στην παροχή μνήμης, στην ανάκτηση τμήματος αρχείου, στην εκτύπωση στην οθόνη, κ.λπ. Το κάθε system call υλοποιείται μέσω μίας ρουτίνας η οποία βρίσκεται σε μία βιβλιοθήκη που γίνεται linked με το κώδικα της user process. Ολες οι ρουτίνες αυτής της βιβλιοθήκης εκτελούν μία ειδική εντολή, που ονομάζεται TRAP. Η εντολή TRAP είναι αυτή η οποία αλλάζει το σύστημα από user mode σε kernel mode, αλλάζοντας ένα bit σε ένα CPU register το οποίο καταδεικνύει τον τρόπο λειτουργίας του συστήματος.

Επιπλέον, η ρουτίνα που καλεί TRAP είναι υπεύθηνη να τοποθετήσει τις παραμέτρους του system call σε μία προσυμφωνημένη διεύθυνση, (συνήθως, CPU registers ή ακόμα και στο stack) όπου ο kernel θα τις βρει.

Οταν ο kernel τελειώσει, τότε τοποθετεί επιστρεφόμενες πληροφορίες σε registers και εκτελεί ένα RETURN FROM TRAP ενεργοποιώντας πάλι τη ρουτίνα της βιβλιοθήκης. Αυτή η ρουτίνα, επιστρέφει την πληροφορία από τους registers στη user process.

1.3 Βασικές Οντότητες

Οι βασικές οντότητες είναι: Processes και Files.

Διεργασίες (Processes).

Η κάθε διεργασία αντιπροσωπεύει ένα εκτελούμενο πρόγραμμα. Με αυτή την έννοια το Λ.Σ. προσπαθεί να οργανώσει την λειτουργία του με βάση το τι έχει κάνει και το τι πρέπει να κάνει γιά το κάθε process. Οι βασικές πληροφορίες που αφορούν μία διεργασία είναι: το object code, τα δεδομένα του προγράμματος, το stack, το program counter, το stack pointer, και οι τιμές των άλλων βασικών registers. Ο kernel κρατάει αυτές τις πληροφορίες σε κατάλληλες δομές.

Μία από αυτές τις δομές λέγεται **process table**. Σε αυτόν τον πίνακα, υπάρχει μία εγγραφή γιά κάθε process.

Οι διεργασίες μπορούν να έχουν και μία ιεραρχική σχέχη (πατέρας, παιδί, πρόγονος, κ.λπ.). Η σχέση αυτή δημιουργείται μέσω του system call fork().

Τυπικά system calls αφορούν την δημιουργία και τερματισμό διεργασιών (fork() και kill()). Επίσης, υπάρχουν και τα wait(), malloc(), exec(), κλπ.

Τέλος, συχνά χρειάζεται να κοινοποιήσει ο kernel σε ένα process πληροφορία. Αυτό επιτυγχάνεται με την χρήση των **signals**. Για παράδειγμα, όταν μία διεργασία διαιρεί διά του 0, τότε ένα signal στέλνεται με αυτή τη πληροφορία.

Αρχεία (Files).

Η δεύτερη βασική οντότητα είναι τα αρχεία. Συνδέονται με system calls που δημιουργούν, διαγράφουν, διαβάζουν, ενημερώνουν, (create(), rm(), read(), write()), κ.λπ. αρχεία. Συνήθως, ένα αρχείο πρέπει πρώτα να «ανοιχθεί» (open()) πριν διαβασθεί ή ενημερωθεί και μετά να «κλεισθεί» (close()).

Τα αρχεία είναι οργανωμένα σε ομάδες που ονομάζονται directories. Ενα directory μπορεί να περιέχει ένα αλλο directory, και έτσι δημιουργείται ένα ιεραρχικό file system.

Με κάθε αρχείο συνδέεται επίσης και μία λίστα ελέγχου πρόσβασης (**access control list**) η οποία καθορίζει ποιός μπορεί να έχει πρόσβαση στο αρχείο. Οταν κάποιος επιχειρεί να ανοίξει ένα αρχείο, εξετάζεται η access control list και αν όλα είναι εντάξει, τότε το σύστημα επιστρέφει ένα **capability** που ονομάζεται επίσης file descriptor ή file handle. Οι εντολές read(), write(), και close() που θα ακολουθήσουν δέχονται σαν παράμετρο αυτό το descriptor και έτσι αποφεύγετε η χρονοβόρα διαδικασία επανεξέτασης των δικαιωμάτων πρόσβασης του χρήστη γιά κάθε system call στο αρχείο.

Πέρα από την χρησιμοποίηση των files από τους χρήστες για μη-προσωρινά δεδομένα, τα files προσφέρουν και ένα τρόπο υψηλού-επιπέδου πρόσβασης σε περιφερειακές συσκευές (π.χ. δίσκους, τερματικά, κ.λπ). Οι συσκευές αυτές παρουσιάζονται σαν ειδικά αρχεία (device special files). Η διασύνδεση (interface) με τα device special files γίνεται με τον ίδιο τρόπο όπως και με τα παραδοσιακά αρχεία. Ετσι δημιουργείται ένα abstraction για αυτές τις συσκευές που κρύβει ολες τις πολύπλοκες λεπτομέρειες που αφορούν στη χρήση τους.

Pipes.

Pipes είναι ένας μηχανισμός ο οποίος σχετίζεται με files και με processes. Χρησιμοποιείται ώστε δύο διεργασίες να ανταλλάξουν πληροφορίες. Ο τρόπος ανταλλαγής πληροφοριών γίνεται μέσω ενός file interface. Δηλαδή, το pipe παρουσιάζεται σαν ένα ειδικό αρχείο, το οποίο μπορεί να ανοιχθεί, ενημερωθεί/διαβασθεί, και να κλεισθεί. Μία διεργασία Δ1 για να στείλει πληροροφορίες σε μία άλλη Δ2, ανοίγει ένα pipe και το ενημερώνει. Η Δ2 επίσης το ανοίγει και το διαβάζει. Ετσι βλέπουμε ότι το file αποτελεί ένα abstraction για πολλά είδη I/O: με περιφερειακές συσκευές (π.χ., δίσκους, τερματικά) και με δια-διεργαστική επικοινωνία (interprocess communication).

1.4 Βασικές Εννοιες

Batching.
Μιά έννοια που αναπτύχθηκε περίπου στις αρχές της δεκαετίας του ‘60. Αφορά στην δημιουργία ομάδων προγραμμάτων τις οποίες ένας χειριστής «φόρτωνε» μαζί στον υπολογιστή των ημερών. Αυτό ήταν μία βελτίωση σε σχέση με το προηγούμενο τρόπο λειτουργίας, όπου ο χειριστής «φόρτωνε» κάθε job (το πρόγραμμα του χρήστη, τον compiler, κλπ) ξεχωριστά, ένα-ένα - μιά χρονοβόρα διαδικασία.

Multiprogramming.
Πολλά είδη προγράμματων περιέχουν αρκετό I/O (ονομάζονται I/O bound). Οταν καλούν κάποια I/O διαδικασία (π.χ. πρόσβαση σε δίσκο) δεν χρεάζονται τον κεντρικό επεξεργαστή. Αν υπάρχει μόνο ένα πρόγραμμα που τρέχει στο σύστημα, τότε σπαταλείται χωρίς λόγο ένας σημαντικός πόρος (CPU cycles). Η έννοια του multiprogramming (που αναπτύχθηκε στα τέλη του ‘60) επιτρέπει την συμβίωση πολλών προγραμμάτων που τρέχουν. Οταν το ένα πρόγραμμα κάνει I/O, τότε το CPU «δίνεται» σε κάποιο άλλο πρόγραμμα που το χρειάζεται, κ.ο.κ. Ετσι αυξάνεται σημαντικά η απόδοση του συστήματος.

Η αρχική λύση που δόθηκε εστίαζε στη δημιουργία τμημάτων της μνήμης, όπου σε κάθε τμήμα υπήρχε ένα διαφορετικό πρόγραμμα. Αργότερα αυτό βελτιώθηκε, όπως θα δούμε.

Η τεχνική του multiprogramming συνδυάστηκε επιτυχώς με την τεχνική του **spooling**. Με αυτή τη τεχνική, προγράμματα διαβάζονταν κατ’ευθείαν στο δίσκο του συστήματος, παράλληλα με το τρέξιμο των προγραμμάτων που βρίσκονταν στη μνήμη του υπολογιστή. Οταν κάποιο απ’αυτά τα προγράμματα τελείωνε, τότε κάποιο άλλο διαβαζόταν από τον δίσκο στο τμήμα της μνήμης που έμεινε κενό, και άρχιζε να τρέχει.

Το πρόβλημα με τα παραπάνω έγκειται στο μεγάλο χρόνο απόκρισης (από τη στιγμή που ο χρήστης έδινε το προγραμμά του στον χειριστή, μέχρι να πάρει τα αποτελέσματα). Ετσι προέκυψε η έννοια του timesharing.

Timesharing.
Πρόκειται γιά μία μορφή multiprogramming, όπου ο κάθε χρήστης έχει στη διάθεσή του ένα τερματικό. Παρατηρήθηκε ότι πολλοί χρήστες συνήθως σκεφτόντουσαν τις επόμενες κινήσεις τους και έτσι δεν χρεάζονταν τον CPU. Το CPU μοιραζόταν στους χρήστες περιοδικά. Σε κάθε περίοδο ένας χρήστης είχε το CPU μέχρι να τελειώσει η περίοδος, εκτός και αν περίμενε για I/O, η «σκεφτόταν».

Μοντέρνα Συστήματα.

Αυτά τα συστήματα εκμεταλλεύονται κυρίως τις δυνατότητες ενός δικτύου υπολογιστών. Υπάρχουν δύο βασικές έννοιες: network operating systems και distributed operating systems.

Network operating systems.

Σε αυτά τα συστήματα το Λ.Σ. δίνει την δυνατότητα επικοινωνίας με άλλες μηχανές συνδεδεμένες με το ίδιο δίκτυο. Για παράδειγμα, διεργασίες σε ένα υπολογιστή μπορούν να ζητήσουν αρχεία να μεταφερθούν από ένα άλλο υπολογιστή. Ακόμα μπορούν να κάνουν remote login σε άλλους υπολογιστές και να χρησιμοποιήσουν τους πόρους τους.

Distributed operating systems.

Τα κατανεμημένα Λ.Σ. παρέχουν επίσης τις παραπάνω δυνατότητες. Αλλά με τρόπο **διαφανή**. Δηλαδή, ο χρήστης δεν χρειάζεται να γνωρίζει ποιό αρχείο έχει αποθηκευτεί σε ποιό υπολογιστή κ.λπ. ή γενικά να γνωρίζει τίποτα περί κατανομής. Ετσι φαίνεται ότι το Λ.Σ. είναι ένα κεντρικό και όχι κατανεμημένο Λ.Σ. Ο τρόπος πρόσβασης σε όλους τους υλικούς και λογισμικούς πόρους είναι ανεξάρτητος της κατανομής.

1.5 Δομή του Λ.Σ.

Το Λ.Σ. αποτελείται από ένα αριθμό ρουτινών (procedures). Η θεμελιώδης ερώτηση εδώ είναι αν υπάρχει καμμία δομή/οργάνωση αυτών των ρουτινών, με βάση τις υπηρεσίες που προσφέρουν. Για παράδειγμα, όλες οι ρουτίνες που σχετίζονται με την παροχή πρόσβασης σε αρχεία μπορούν να αποτελέσουν ένα module (λειτουργική μονάδα) του Λ.Σ. Σε αυτή τη περίπτωση, η μονάδα παρέχει μόνο ένα interface στα άλλα modules του Λ.Σ. - δηλαδή, για παράδειγμα, μία ρουτίνα του module «διαχείρισης μνήμης» δεν μπορεί να καλέσει άμεσα μία ρουτίνα του module «σύστημα αρχείου», αλλά πρέπει να χρησιμοποιήσει το interface που παρέχει το module.

Μονολιθικά Λ.Σ.

Σε αυτά τα συστήματα δεν υπάρχει καμμία δομή όσον αφορά την οργάνωση των ρουτινών του Λ.Σ. Οποιαδήποτε ρουτίνα μπορεί να κληθεί από οποιαδήποτε άλλη. Για να χτιστεί το object code του Λ.Σ. κάθε ρουτίνα γίνεται compiled ξεχωριστά και μετά linked σε ένα εννιαίο executable image.

Η διαδικασία κλήσης των ρουτινών του πυρήνα του Λ.Σ. που προαναφέραμε (μέσω system calls που αλλάζουν τον τρόπο λειτουργίας του συστήματος από user mode σε kernel mode) ισχύει εδώ.

Παραδείγματα μονολιθικών συστημάτων είναι το UNIX (ATT και BSD).

Μικρο-Πυρήνες.

Αυτή η προσέγγιση βασίζεται στη φιλοσοφία ο πυρήνας του Λ.Σ. να είναι όσο τον δυνατόν μικρότερος. Ρουτίνες οι οποίες παραδοσιακά βρίσκονταν στο kernel τώρα βρίσκονται έξω απ'αυτόν. Με αυτή τη φιλοσοφία, δημιουργούνται ξεχωριστές διεργασίες, αποτελούμενες από λογικά σχετιζόμενες ρουτίνες. Οι διεργασίες αυτές τρέχουν σε user mode και παρέχουν υπηρεσίες (και γι'αυτό ονομάζονται **server**

processes - εξυπηρετητές) στις διεργασίες των χρηστών, οι οποίες ονομάζονται **client processes - πελάτες**.

Οι client processes επικοινωνούν με τις server processes μέσω ρουτινών που υλοποιούν ένα επικοινωνιακό πακέτο λογισμικού (interprocess communication - IPC). Ο kernel έτσι υλοποιεί μόνο τις ρουτίνες του IPC πακέτου π.χ. send_msg() και recv_msg() και προσφέρει πρόσβαση στο υλικό.

Σε αυτό το μοντέλο (που ονομάζεται client-server model) τα παραδοσιακά system calls αντικαθιστούνται από τις IPC ρουτίνες.

Τα πλεονεκτήματα αυτής της προσέγγισης απορρέουν πρώτα από το **modular programming**. Κάθε υπηρεσία του Λ.Σ. απομονώνεται και έτσι κατανοείται καλύτερα. Επιπλέον, δεν υπάρχει ένα και μοναδικό object code του Λ.Σ., πράγμα που οδηγεί σε μεγαλύτερη **αξιοπιστία (reliability)**. Ετσι, άν υπάρχει σε κάποια ρουτίνα ένα bug, τότε μόνο ο server που περιέχει την ρουτίνα θα πέσει και όχι όλο το Λ.Σ. Επίσης, έτσι προσφέρεται **διαφάνεια κατανομής - distribution transparency**. Δηλάδή, ο file server, για παράδειγμα, μπορεί να μετακινηθεί σε άλλο Σ.Ε. που συνδέεται με ένα δίκτυο υπολογιστών, χωρίς να χρειαστεί να αλλαχθεί κανένα πρόγραμμα επειδή το interface παραμένει το ίδιο (στείλε μήνυμα, λάβε μήνυμα, κ.λπ).

Προσέξτε, ότι εφ'όσον λειτουργίες του Λ.Σ. παρέχονται τώρα από user-mode processes οι χρήστες μπορούν να γράψουν τους δικούς τους servers, αν δεν είναι ικανοποιημένοι με τους servers που παρέχει το Λ.Σ. Επιπλέον, παραπάνω του ενός server μπορούν να συμβιώνουν στο σύστημα. Για παράδειγμα, μπορεί να προκύψουν συστήματα με δύο διαφορετικούς file servers.

Κεφάλαιο 2. Διεργασίες (PROCESSES)

2.1 Εισαγωγή

Μία από τις δύο κεντρικές έννοιες και abstractions ενός Λ.Σ. Αποτελεί ένα **μοντέλο για ένα πρόγραμμα που εκτελείται**

Οπως είπαμε, τα process εκτελούνται "παράλληλα": δηλ. όχι σειριακά. Υπάρχουν δύο τρόποι:

- Οσο μια περιφερειακή συσκευή (π.χ. δίσκος) εκτελεί μια εντολή, το Λ.Σ. "δίνει" (dispatches) το CPU σ' ένα άλλο "έτοιμο" (ready) πρόγραμμα.
- Οταν το CPU "δίνεται" σ' ένα process, το process συνήθως δεν κρατάει το CPU μέχρι να τελειώσει, ακόμα και όταν το process δεν κάνει I/O. Κάθε λίγο (π.χ. 10ms) το CPU δίνεται σ' άλλο process.

Ετσι προκύπτει ότι σε κάθε στιγμή σ' έναν υπολογιστή εκτελούνται πολλές παράλληλες ενέργειες. Για ν' αντιμετωπίσουν τις δυσκολίες που προκύπτουν, τα Λ.Σ. χρησιμοποιούν την έννοια του process. -> όλα τα προγράμματα που τρέχουν αντιπροσωπεύονται από ένα process.

multiprogramming: αφορά στο γεγονός ότι πολλά processes τρέχουν ανά πάσα στιγμή.

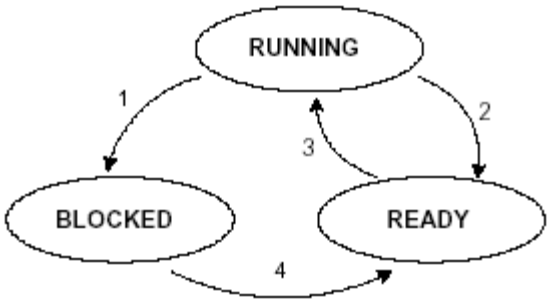
Ποιά η διαφορά μεταξύ "process" & "πρόγραμμα";

Ενα process αντιπροσωπεύει μια διαδικασία η οποία εκτελείται με βάση ένα πρόγραμμα, input/output και έχει και μια κατάσταση (state - δηλ. οι τιμές των μεταβλητών του προγράμματος, CPU registers, PC, SP, stack, ...) (Κοιτάξτε το παράδειγμα συνταγής και διαδικασίας μαγειρέματος που δίνει το βιβλίο).

Τα process σχηματίζουν μια ιεραρχία πατέρα-παιδιού (ή πρόγονου-απόγονου) μέσω του **fork** system call.

Κατάσταση ενός process (ως προς την εκτελεσιμότητά του)

- τρέχει (**running**): έχει το CPU
- έτοιμο (**ready**): μπορεί να τρέξει, αλλά το CPU δόθηκε αλλού.
- μπλοκαρισμένο (**blocked**): περιμένει εξωτερικό γεγονός.



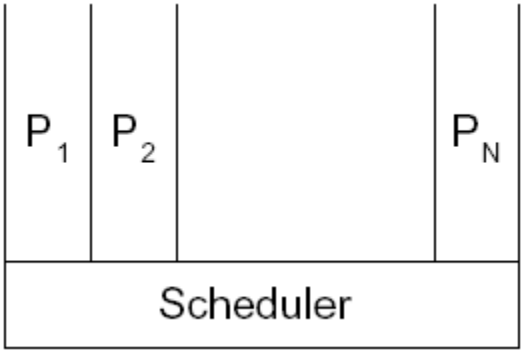
1. περιμένει I/O
2. CPU δίνεται σ' άλλο process
3. CPU δίνεται σ' αυτό το process
4. I/O έγινε

Για την μετάβαση (1) υπάρχει συνήθως ένα **block** system call. Οι μεταβάσεις (2) και (3) προκύπτουν από παρεμβάσεις του scheduler (δρομολογητή) του Λ.Σ.

Πώς και πότε τρέχει ο scheduler; (ποιός τον κάνει schedule;)

- clock interrupts: εαν το **time slice** του process έχει παρέλθει.
- block: όταν το τωρινό process μπλοκάρει.

Χρησιμοποιώντας το μοντέλο των processes μπορούμε ευκολότερα ν' αναπαριστούμε τις ενέργειες που εκτελούνται κάθε στιγμή σ' ένα σύστημα.



Σ' αυτό το μοντέλο ο scheduler είναι το κατώτερο επίπεδο του Λ.Σ. και κρύβει λεπτομέρειες που αφορούν το σταμάτημα και τρέξιμο processes και διαχείριση interrupts.

2.2 Υλοποίηση του PROCESS

Περίληψη πληροφορίας για κάθε process:

- PC, SP, CPU registers, PSW, signals
- Page Tables και άλλες πληροφορίες που αφορούν μνήμη (π.χ. swap space), pointers σε text, data, bss ...
- pid, parent pid, uid, gid, κατάσταση εκτελεσιμότητας
- root dir, file descriptors για open files.

Υπάρχει ένας πίνακας (**process table**) που συνήθως (στα συστήματα UNIX) υλοποιείται σαν ένα linked list of structs. Το struct για το κάθε process έχει τις παραπάνω πληροφορίες.

Πώς επιτυγχάνεται ο παραλληλισμός (**concurrency** - δηλ. ότι υπάρχουν πολλά processes που τρέχουν) όταν υπάρχει ένα και μοναδικό CPU ;

1. Λόγω του διαμοιρασμού CPU μέσω των time slices (δηλ. το timesharing) αν εξετάσουμε το σύστημα για μια χρονική στιγμή μεγαλύτερη ενός time slice θα δούμε ότι σ' αυτή τη χρονική στιγμή "τρέχουν" > 1 processes.
2. Πολλές συσκευές περιφέρειας (peripheral devices) έχουν δικούς τους επεξεργαστές. Ετσι όσο I/O (π.χ. σε δίσκο) εκτελείται εκ μέρους ενός process, ένα άλλο process εκτελείται από το CPU.

Διαχείριση interrupts:

Interrupt Vector: Καλύπτει τις "χαμηλότερες" διευθύνσεις της μνήμης. Αποτελείται από μια σειρά διευθύνσεων, μια για κάθε συσκευή, που "δείχνουν" στις ρουτίνες που πρέπει να κληθούν για να εξυπηρετηθεί το interrupt της κάθε συσκευής (-> δηλ. μια ρουτίνα για δίσκο, άλλη για τερματικό, άλλη για το ρολόϊ, κ.λπ). Αυτές οι ρουτίνες λέγονται **interrupt service routines** (ISR).

Αλγόριθμος για μεταχείριση interrupt:

1. Hardware:

- τοποθέτησε (push) στο stack τις τιμές των PC, SP, PSW, και κάποιων CPU registers
- βρες την κατάλληλη διεύθυνση στο interupt vector και "φόρτωσε" την στο PC. Ετσι, εκτελείται η ISR.

2. Software:

- Η ISR αποθηκεύει τα CPU registers στο κατάλληλο struct του Process Table (assembly lang).
- Μετά, το SP ενημερώνεται να "δείχνει" σ' ένα άλλο προσωρινό stack (assembly lang)
- Μετά, καλείται μια C ρουτίνα που βρίσκει ποιό process αφορά αυτό το interrupt. Αυτό το process τώρα γίνεται "έτοιμο" (δηλ. το process struct φεύγει από την Sleep λίστα και πηγαίνει στην Ready λιστα).

- Καλείται ο scheduler να διαλέξει ένα Process
- Η C ρουτίνα επιστρέφει στην assembly lang. ρουτίνα που φορτώνει τα CPU Register και Page Tables του process που διάλεξε ο scheduler.

2.3 Επικοινωνία Διεργασιών (Interprocess Communication)

Race Conditions: περιγράφει καταστάσεις όπου > *1* processes διαβάζουν ή γράφουν σε κοινά δεδομένα και το τελικό αποτέλεσμα εξαρτάται από το πότε τρέχει το κάθε process. Το πρόβλημα είναι: έλεγχος πρόσβασης σε κοινή μνήμη.

Παράδειγμα:

Δύο processes κάνουν και οι δύο αναλήψεις από ένα τραπεζικό λογαριασμό (κοινό) - η μιά π.χ. στον σύζυγο και η άλλη αντίστοιχα στη σύζυγο. Η ρουτίνα *ΑΝΑΛΗΨΗ* καλείται και από τις δύο processes.

ΑΝΑΛΗΨΗ: read (acc_bal); write (acc_bal = acc_bal - amount): Το P1 εκτελεί το read() και μετά το CPU δίνεται στο P2 το οποίο εκτελέί read() και write(). Μετά, το CPU δίνεται στο P1 που εκτελεί το write(). Το αποτέλεσμα είναι ότι μόνο η μία ανάληψη φαίνεται - η άλλη "χάθηκε". Αυτό είναι ένα "τυπικό" race condition.

Critical Sections:

Πώς μπορούμε ν' αποφύγουμε race conditions ; Πρέπει να βρούμε ένα τρόπο ούτως ώστε μόνο ένα process να έχει πρόσβαση στα κοινά δεδομένα σε κάθε στιγμή. Αυτή η ιδιότητα λέγεται και mutual exclusion (αλληλοεξαίρεση). Το πρόβλημα του mutual exclusion είναι κομβικό για ένα Λ.Σ.

Ενας άλλος τρόπος θεώρησης του προβλήματος είναι να εστιάσει στην εκτέλεση ενός process και ν' απομονώσουμε το τμήμα του κώδικα το οποίο γράφει/διαβάζει κοινή μνήμη. Αυτό το τμήμα ονομάζεται critical section. Η λύση του προβλήματος έγκειται στο να σιγουρέψουμε ότι μόνο ένα process μπορεί να μπεί στο critical section του σε κάθε στιγμή.

2.3.1 Απενεργοποίηση (Disabling) Διακοπών (Interrupts)

Ας υποθέσουμε ότι ένα process δεν μπορεί να μπλοκάρει μέσα στο critical section. Τότε ο μόνος τρόπος για να δοθεί το CPU σ' ένα άλλο process είναι μέσω ενός clock interrupt που θα σημάνει το τέλος του time slice του process.
Αν κάνουμε **interrupt disabling** μόλις μπούμε στο critical section και **enabling** μόλις πριν βγούμε τότε κανένα άλλο process δεν πρόκειται να βρίσκεται ταυτόχρονα με μας στο critical section του.

Αλλά: δίνοντας τη δυνατότητα σε user processes να κάνει interrupt disabling είναι επικίνδυνο (π.χ. bugs => interrupts μένουν disabled ...) => αυτή η λύση στο mutual exclusion δεν είναι αποδεκτή.

2.3.2 Μεταβλητές κλειδώματος (Locks)

Μια άλλη λύση μπορεί να βασισθεί σε lock vars. Πριν ένα process μπεί στο critical section, εξετάζει ένα lock var.

- Αν η τιμή του είναι 1, τότε περιμένει [μπλοκάρει;] μέχρι να γίνει 0.
- Αν είναι 0, τότε το κάνει 1 και μετά εισβάλλει στο critical section του.

Το πρόβλημα όμως δεν λύθηκε, απλώς "μετατέθηκε". Τι γίνεται αν 2 processes διαβάσουν την τιμή του lock var την ίδια στιγμή ;

2.3.3 Η λύση του Peterson.

Αποτελείται από 2 ρουτίνες τις οποίες οι processes καλούν δίνοντας το process id τους για να μπούν και να βγούν σε/από critical section: *enter_region (process)* και *leave_region(process)*.
Υπάρχει επίσης ένα flag, "turn", το οποίο χρησιμοποιείται για να δώσει την σειρά σε ένα μόνο ενδιαφερόμενο process, και ένας πίνακας από flags, "interested[]" που δείχνει την επιθυμία ενός process να μπεί στο critical section του.

```
enter_region (process)
{
other = 1 - process;
interested[process] = TRUE;
turn = process;
while (turn == process &&
     interested[other] == TRUE) ;
}
```

```
leave_region (process)
{
        interested[process] = FALSE ;
}
```

Αν η P1 εκτελέσει enter_region() πρώτη, μιας και το interested [other] $\neq$ TRUE δεν θα περιμένει (με busy waiting) και θα συνεχίσει. Οταν μετά ενδιαφερθεί η P2 το interested[P1] = TRUE και το P2 θα συνεχίζει να εκτελεί το while loop.

Αν και οι δύο processes καλέσουν enter_region() (σχεδόν) ταυτόχρονα, τότε, μιας και έχουμε ένα CPU, κάποια από τις 2 processes θ' εκτελέσει τελευταία την εντολή turn=process. Αυτή η process δεν θα μπορέσει να μπεί στο critical section και θα κάνει busy wait στο while loop. Ετσι, μόνο η άλλη process θα μπεί στο critical section της.

2.3.4 Λύση με Test και Set Lock (TSL)

Πολλά computers παρέχουν TSL instructions οι οποίες εξετάζουν (read) ένα register το οποίο φορτώνεται μ' ένα memory word, και μετά αποθηκεύουν μια $\neq 0$ τιμή στο memory word αυτό (write). Οι πράξεις read & write σ' αυτό το register είναι **atomic** (δηλ. κανείς άλλος δεν μπορεί να προσπελάσει το register μέχρι το TSL (read & write) να έχει εκτελεστεί). Η TSL μπορεί, λοιπόν να εκτελεστεί σ' ένα flag, το οποίο είναι ένα κοινό variable. Οταν το flag = 0, τότε ένα process μπορεί να εκτλέσει TSL στο flag και να το κάνει 1. Επειδή το TSL είναι atomic, έτσι διασφαλίζεται το mutual exclusion.

```
enter_region:

TSL reg, flag          /* reg <-- flag && flag = 1 */
cmp reg, 0             /* flag  = 0 ; */
jnz enter_region       /* try again */
ret

leave_region:
mov flag, 0            /*  flag = 0 */
ret
```

Ενα process που καλεί enter_region θα μπεί μόνο αν το flag ήταν 0. Αλλοιώς θα κάνει συνέχεια "jump" μέχρι το flag = 0. Προσέξτε ότι η λύση είναι η ίδια με την lock variables. Το πρόβλημα της τελευταίας δεν υφίσταται εδώ επειδή το hardware δεν επιτρέπει δύο process να διαβάσουν την τιμή του flag πριν κανένα από τα δύο την κάνει 1.

Προσέξτε ότι:

1. οι δύο τελευταίες λύσεις απαιτούν **busy waiting**.
2. Αν ένα process προσπαθήσει να προσπελάσει το flag χωρίς να χρησιμοποιήσει το enter_region ή δεν χρησιμοποιήσει το leave_region τότε η λύση καταρρέει. [π.χ. ένα process μπορεί να εκτελέσει "mov flag, #0" πριν καλέσει enter_region...]

Το busy waiting έχει δύο αρνητικές επιπτώσεις:

- Σπατάλη CPU cycles.
- Priority inversion.
- Π.Χ. Ο scheduler ενός Λ.Σ. προσπαθεί να τρέχει πάντα processes υψηλής προτεραιότητας, ας πούμε P1. Αν υπάρχει κάποιο P2, χαμηλής προτεραιότητας, και μπεί στο critical section μπορεί να προκύψει πρόβλημα αν και το P1 θέλει να μπει στο critical section. Το P1 θα κάνει busy wait και δεν θα δώσει ποτέ την ευκαιρία στο P2 να τρέξει ώστε να βγεί από το critical section => το P1 θα κάνει busy wait για πολύ.

Λύσεις χωρίς BUSY WAIT

Βασική Ιδέα: Processes μπλοκάρουν αντί για busy wait. Οταν μπλοκάρει ένα process, ο scheduler του Λ.Σ. δρομολογεί άλλα processes και έτσι αποφεύγονται τα προβλήματα του busy wait.

Υπάρχουν 2 systems calls: SLEEP & WAKEUP (καμμιά φορά αποκαλούνται και WAIT & SIGNAL, αντίστοιχα). Οταν ένα process καλεί SLEEP τότε από την κατάσταση "RUNNING" περνάει στη κατάσταση "BLOCKED". Οταν κάποιο άλλο process καλέσει WAKEUP(P) τότε το process P, θα περάσει στη κατάσταση "READY".

2.3.5 Το Πρόβλημα Παραγωγού-Καταναλωτή

Κλασσικό παράδειγμα προβλήματος IPC, mutual exclusion, race conditions, κ.λπ. Υπάρχουν 2 processes ο παραγωγός, P, και ο καταναλωτής, C. Ο P παράγει πληροφορία και την αποθηκεύει σ' ένα buffer B. Ο C προσπελάυνει τον B καταναλώνοντας την πληροφορία.

Προβλήματα προς λύση:

- Τι γίνεται όταν το B είναι γεμάτο: πού θα βάλει τα δεδομένα ο P ;
- Τι γίνεται όταν το B έιναι άδειο και ο C επιχειρεί να καταναλώσει πληροφορία ;

Σημειωση:
Πέρα της διαχείρισης/ελέγχου της κοινής μνήμης, εδώ απαιτείται και συγχρονισμός (synchronization).

Οι λύσεις βασίζονται στα εξής:

- Ο P "κοιμάται" όταν ο B είναι γεμάτος. Τον ξυπνάει ο C.
- Ο C "κοιμάται" όταν ο B είναι άδειος. Τον ξυπνάει ο P.
- Υπάρχει μια μεταβλητή N που δείχνει το μέγεθος του B.
- Υπάρχει μια μεταβλητή count που δείχνει τον αριθμό πληροφοριακών μονάδων στον B.

Producer

```
while (1) {
    prod_item();
    if (count==N)
       sleep();
    put_item();
    count ++ ;
    if (count == 1)
     wakeup(consumer)
}
```

Consumer

```
while (1) {
            if (count ==0)
                        sleep();
            remove_item();
            count = count - 1
            if (count == N - 1)
                        wakeup (producer);
            cons_item()
}
```

Ετσι ο P κοιμάται όταν ο B είναι γεμάτος και ξυπνάει τον C μόλις ο B παύει να είναι άδειος. Ενώ, ο C κοιμάται όταν ο B είναι άδειος και ξυπνάει τον P όταν ο B παύει να είναι γεμάτος.

Μπορεί να προκύψει πρόβλημα, όμως, λόγω του ότι P & C προσπελαύνουν το count χωρίς περιορισμό.

Ειδικότερα: είναι δυνατόν ο C να δει ότι count = 0, και πριν προλάβει να καλέσει sleep() το CPU να δοθεί στο P ο οποίος παράγει, κάνει count = 1 και καλεί wakeup. Ομως ο C δεν κοιμάται ακόμα => το wakeup "χάνεται". Αργότερα, το CPU θα δοθεί στον C ο οποίος θα καλέσει sleep() και θα κοιμηθεί. Τώρα, όταν ο P γεμίσει τον B, θα καλέσει sleep() και αυτός => P & C κοιμούνται τον αιώνιο ύπνο...

Το πρόβλημα προκύπτει διότι το "wakeup χάθηκε". Θα μπορούσαμε να χρησιμοποιήσουμε ένα wakeup bit. Οταν λοιπόν, επιχειρείται ένα wakeup για κάποιο process που δεν κοιμάται, τότε το bit γίνεται 1 => δεν χάνεται η πληροφορία του wakeup.
Οταν αργότερα ένα process πρόκειται να καλέσει sleep() αν το bit είναι ένα τότε το process δεν καλεί sleep() και ξανακάνει το bit 0.

Είναι σωστή αυτή η λύση ; (με το wakeup bit)

2.3.6 Σημαφόροι (Semaphores)

Ενας τύπος μεταβλητών. Στην ουσία "μετράει" όλα τα wakeup για ένα process. Δύο πράξεις: UP & DOWN (γενίκευση των WAKEUP & SLEEP).

```
DOWN(s) : 1 atomic action
{
  if s = 0 then sleep()
  else s = s - 1
}

UP(s):
  s: = s + 1
```

Αν μετά την εκτέλεση του UP(s), s = 1 και υπάρχει κάποιο process που κοιμάται γιατί το s ήταν 0, τότε το σύστημα διαλέγει κάποιο απ' τα κοιμώμενα processes και το ξυπνάει. Ολα αυτά που εκτελούνται στο UP() είναι επίσης atomic.

Πώς λοιπόν, μπορούμε να λύσουμε το Prod-Cons πρόβλημα; Τα UP & DOWN υλοποιούνται σαν system calls. Το atomicity υλοποιείται ως εξής: Με 1 CPU : interrupt disabling. Με N CPUs: interrupt disabling + TSL

Θα χρησιμοποιήσουμε 3 semaphores:

- #full: μετράει πόσα items έχει ο B
- #empty: μετράει πόσες θέσεις του B είναι άδειες

- mutex: διασφαλίζει ότι μόνο ο P ή μόνο ο C προσπελαύνουν το B. Ο mutex είναι binary semaphore: παίρνει τιμές = 0 και 1 μόνο.

Οι αρχικές τιμές τους είναι: 0, N, και 1, αντίστοιχα.

Προσέξτε ότι τα semaphores προσφέρουν ένα τρόπο επίλυσης προβλημάτων συγχρονισμού (#full, #empty) και ένα τρόπο επίλυσης προβλημάτων αλληλο-εξαίρεσης (mutual exclusion):

mutex: για mutual exclusion

#full, #empty: για να σταματά ο P (C) όταν ο B είναι γεμάτος (άδειος).

```
semaphore mutex = 1
semaphore #full = 0
semaphore #empty = N

producer ()
  while (1) {
     prod_item () ;
     down (#empty);
     down (mutex) ;
     put_item () ;
     up (mutex) ;
     up (#full) ;
}

consumer ()
 while (1) {
    down (#full) ;
    down (mutex) ;
    cons_item () ;
    up (mutex)  ;
    up (#empty)
}
```

Τι θα συνέβαινε αν ο P άλλαζε τη σειρά των 2 DOWN ;
Τι θα συνέβαινε αν ο C άλλαζε τη σειρά των 2 DOWN ;

2.3.7 Monitors

Αν και φαινομενικά εύκολη, η χρήση των semaphores συνήθως κρύβει δυσκολίες (π.χ. η σειρά με την οποία κλήθηκαν τα DOWN στο Prod_Cons πρόβλημα).

Τα monitors δημιουργήθηκαν για συγχρονισμό σε υψηλότερο επίπεδο - οι προγραμματιστές δεν ασχολούνται πλέον με τέτοια low-level θέματα όπως η σειρά των DOWN, κ.λπ.

Ενα monitor είναι ένα ειδικό module: διαδικασίες, μεταβλητές, δομές δεδομένων. Οι διαδικασίες του monitor είναι οι μόνες που επηρεάζουν την κατάσταση των μεταβλητών

και των δεδομένων => ένα process για να προσπελάσει τα δεδομένα ενός monitor πρέπει να καλέσει τις διαδικασίες του.

Η πιο σημαντική ιδιότητα του monitor είναι ότι **μόνο ένα process μπορεί να βρίσκεται μέσα στο monitor σε κάθε στιγμή**.

Το monitor είναι ένας μηχανισμός που προσφέρεται από μια γλώσσα προγραμματισμού. Ο compiler αναγνωρίζει calls σε monitor διαδικασίες και τα χειρίζεται διαφορετικά. Οι πρώτες εντολές σε μια monitor διαδικασία, ελέγχον αν βρίσκεται κάποιο άλλο process ενεργά μέσα στο monitor. Αν ναι, το process που καλεί το monitor θα γίνει "suspended". Αλλλοιώς το process θα μπεί στο monitor.

Ο compiler, λοιπόν, υλοποιεί το mutual exclusion (συνήθως χρησιμοποιώντας a binary semaphore). Ομως, όπως είδαμε πιο πάνω, mutual exclusion είναι μόνο μέρος του προβλήματος (απαιτείται και συγχρονισμός).

Τα προβλήματα συγχρονισμού λύνονται από τον monitor μέσω condition variables. Cond. vars δέχονται εντολές WAIT & SIGNAL

Οταν ένα monitor procedure δεν μπορεί να συνεχίσει (π.χ. ο P όταν ο B είναι γεμάτος) τότε καλεί WAIT(c), όπου c είναι ένα cond. var. Ετσι η process που καλεί WAIT μπλοκάρει. Επίσης, επιτρέπει σε κάποιο άλλο process, που είχε επιχειρήσει να μπεί στο monitor και απέτυχε, να μπεί τώρα στο monitor. Προσέξτε ότι μόνο ένα ενεργό process είναι στο monitor.

Αυτό το άλλο process μπορεί να "ξυπνήσει" το process που έκανε WAIT μέσω του SIGNAL(c). Τώρα όμως πρέπει ν' αποφύγουμε την πιθανότητα να είναι και τα 2, πλέον ενεργά, processes στο monitor. => συνήθως το SIGNAL() επιτρέπεται μόνο σαν η τελευταία εντολή σ' ένα monitor procedure.

Αν πολλά processes έχουν εκτελέσει WAIT(c), τότε η εντολή SIGNAL(c) θα ξυπνήσει μόνο ένα εξ' αυτών. Το SIGNAL(c) "χάνεται" αν κανείς δεν έχει WAIT(c). Τα WAIT & SIGNAL είναι όμοια με τα SLEEP & WAKEUP που είδαμε πριν. Πώς λοιπόν, αποφεύγεται το πρόβλημα που εξετάσαμε με το SLEEP & WAKEUP ;

Τα monitors απλοποιούν πολύ το έργο των προγραμματιστών ! Το μόνο που κάνουν οι προγραμματιστές είναι να περικλείσουν σ' ενα monitor procedure τα critical section τους.

```
monitor Prod_Cons
    condition full, empty;
    integer count ;
    procedure enter {
      if count = N then wait(full) ;
      enter_item ;
      count ++
      if count = 1 then signal(empty)
    }
```

```
    procedure remove {
      if count = 0 then wait(empty) ;
      remove_item ;
      count -- ;
      if count = N - 1 then signal(full)
    }

count = 0 ;
enter monitor ;


procedure producer          procedure consumer
   while (1) do {            while (1) do {
     prod_item() ;                Prod_Cons.remove ;
     Prod_Cons.enter              consume_item() ;
   }                          }
```

2.3.8 Ανταλλαγή Μηνυμάτων (Message Passing)

Processes επικοινωνούν μ' ενταλλαγή μηνυμάτων: system calls SEND και RECEIVE: send(dest, &msg); recv(source, &msg).

Γενικά για msg-passing συστήματα:

Μηνυματα χάνονται: Γι αυτό συνήθως ο αποδέκτης μετά την λήψη, στέλνει ένα ειδικό μήνυμα που λέγεται acknowledgement. Οταν ο αποστολέας λαμβάνει το ACK τότε ξέρει ότι όλα πήγαν καλά. Αλλοιώς, ξαναστέλνει το μήνυμα.

Αν χαθει το ack τότε το ίδιο μήνυμα θα σταλεί > 1 φορές => πρέπει ο αποδέκτης να μπορεί να ξεχωρίζει τ' αντίγραφα. [αυτό γίνεται έχοντας τον αποστολέα να χρησιμοποιεί έναν counter στο msg και αν έχει λάβει ένα μήνυμα από τον ίδιο αποστολέα με την ίδια τιμή στον counter τότε ξέρει ότι είναι αντίγραφο].

Ονοματα των processes: πρέπει να είναι μοναδικά, αλλοιώς το μήνυμά μας μπορεί να σταλεί σ' άλλο process. Αυτό έιναι δύσκολο πρόβλημα. Συνήθως: process@machine.domain

Authentication: Πώς ξέρω ότι αυτός που επικοινωνώ είναι πράγματι αυτός που ισχυρίζεται ότι είναι ; Encryption και κλειδιά λύνουν το πρόβλημα αυτό: το κλειδί για το μήνυμα τόχει μόνο ο σωστός αποδέκτης - κανείς άλλος δεν μπορεί να κάνει decrypt.

Πώς στέλνω μηνύματα στην ίδια μηχανή ; (δηλ. αποστολέας και αποδέκτης τρέχουν στην ίδια μηχανή). Παραδοσιακές λύσεις βασίζονται στην αντιγραφή του μηνύματος από το address space του αποστολέα στο address space του αποδέκτη. Αυτό κοστίζει πολύ (ιδιαίτερα για μεγάλα μηνύματα).

2.3.9 Παραγωγός-Καταναλωτής -- Producer-Consumer και msg passing

Τώρα δεν έχουμε κοινούς buffer. Τα N buffers αντικαθίστανται από N msgs. Στην αρχή, ο cons στέλνει N "άδεια" μηνύματα στον prod. Κάθε φορά που ο producer παράγει ένα item, παίρνει ένα "άδειο" μήνυμα, το γεμίζει, και το στέλνει πίσω στον consumer. Ο prod μπλοκάρει μέχρι να λάβει άδεια μηνύματα από τον cons. Ο cons μπλοκάρει μέχρι να λάβει γεμάτα μηνύματα απ' τον prod.

producer

```
  while (1) {
    prod_item (& item);
    recv (consumer, &msg);
    make\_msg (&msg, &item);
    send (consumer, &msg)
  }
```

consumer

```
For i:= 1 to N send(producer,&m);
while (1)  {
            recv (producer, &msg);
            get_item (&item);
            send (producer, msg);
            consume_item (item)
}
```

Προσέξτε, ότι τα send()/recv() είναι blocking system calls. Δηλαδή μπορεί να προκαλέσουν το μπλοκάρισμα του process. Πότε ακριβώς μπλοκάρουν εξαρτάται από την υλοποίηση του msg passing.

Υλοποίηση με Mailboxes. Αντί τα μηνύματα να στέλνονται σε process names, μπορούμε να δημιουργήσουμε ένα mailbox για κάθε process, όπου στέλνονται τα μηνύματα. Το κάθε process τότε εκτελεί recv από το δικό του mailbox. Το mailbox ενός process περιέχει όλα τα μηνύματα που στάλθηκαν στο process και δεν έχουν γίνει received από το process. Processes μπλοκάρουν όταν προσπαθούν να διαβάσουν/λάβουν από άδειο mbox ή να στείλουν σε γεμάτο mbox.

Η λύση με mbox απαιτεί φυσικά buffering (χώρο στη Κ.Μ.). Μία εναλλακτική λύση βασίζεται στην έννοια του ραντεβού (rendezvous).

Υλοποίηση με Ραντεβού (Rendezvous)

Το send() μπλοκάρει μέχρι να γίνει το αντίστοιχο recv() από το άλλο process. Επίσης, το recv() μπλοκάρει μέχρι να γίνει το αντίστοιχο send(). Αυτή η λύση δεν απαιτεί καθόλου buffering: το μήνυμα μεταφέρεται κατ' ευθείαν από τον αποστολέα στον αποδέκτη. Η λύση με rendezvous είναι πιο απλή, αλλά όχι ευέλικτη. π.χ. τι γίνεται αν ο producer είναι πολύ πιο γρήγορος από τον consumer, ή αντίστροφα ;

2.3.10 Οι φιλόσοφοι που γευματίζουν - Dining Philosophers

Το πρόβλημα:

5 φιλόσοφοι κάθονται σ' ένα τραπέζι με 5 πιάτα φαγητό (σούσι). Για να φάει ένας φιλόσοφος χρειάζεται 2 ξυλάκια (chopsticks) - δεξιά και αριστερά κάθε πιάτου βρίσκεται από ένα ξυλάκι. Ο κάθε φιλόσοφος περνάει την ζωή του σκεπτόμενος και τρώγοντας,

εναλλάξ. Το πρόβλημα είναι να γράψεις έναν αλγόριθμο για την ζωή των φιλοσόφων που δεν "κολλάει" ποτέ. π.χ.

```
philosopher (i)
  while (1) {
    think;
    take_fork(i);
    take_fork(i+1 mod 5);
    eat;
    put_fork(i);
    put_fork(i+1 mod 5);
  }
```

Ο παραπάνω αλγόριθμος δεν ικανοποιεί! Αν όλοι οι φιλόσοφοι πεινάσουν την ίδια στιγμή και εκτελέσουν τον αλγόριθμο τότε όλοι θα πάρουν το ξυλάκι στα αριστερά τους. Ομως κανείς δεν θα καταφέρει να πάρει το δεξί ξυλάκι -> όλοι θα πεθάνουν της πείνας.

Αυτό είναι ένα γενικότερο και πολύ σημαντικό πρόβλημα για την δημιουργία συστημάτων λογισμικού και λέγεται **αδιέξοδο (deadlock)**. Συμβαίνει όταν >1 processes δημιουργούν μια κυκλική αλυσίδα: και όπου το κάθε process για να συνεχίσει χρειάζεται έναν πόρο που τον κατέχει το επόμενο process κ.ο.κ.

Μια λύση είναι ν' αναγκάσουμε τον κάθε φιλόσοφο, μόλις πάρει το αριστερό ξυλάκι να εξετάσει αν το δεξί είναι διαθέσιμο. Αν ναι, εντάξει. Αλλοιώς, αφήνει το αριστερό ξυλάκι και δοκιμάζει πάλι μετά από κάποιο χρονικό διάστημα.

Το πρόβλημα μ' αυτή τη λύση είναι λίγο διαφορετικό. Αν όλοι οι φιλόσοφοι αρχίσουν την ίδια στιγμή, θα πάρουν το αριστερό την ίδια στιγμή, θα τ' αφήσουν την ίδια στιγμή κ.ο.κ. Αυτό το φαινόμενο ονομάζεται **επ' αόριστον αναβολή (indefinite postponement ή starvation)**: τα processes συνεχίζουν να τρέχουν αλλά δεν σημειώνουν καμμιά πρόοδο...

Μια άλλη λύση είναι να δημιουργήσουμε ένα critical section στον αλγόριθμο των φιλοσόφων (μετά το "think") ο οποίος να προστατεύεται από ένα semaphore (mutex = 1). Πριν μπεί στο critical section καλεί DOWN(mutex) και μόλις βγεί καλεί UP(mutex). Η λύση αυτή είναι σωστή αλλά θεωρείται μη αποδοτική! Γιατί ;

Μια καλύτερη λύση:

Χρησιμοποιείται ένα semaphore (mutex = 1) για προστασία του critical section. Χρησιμοποιείται ένα semaphore για κάθε φιλόσοφο s[1..5], αρχικά s[i] = 0. Τέλος υπάρχει και ένα var που αντιπροσωπεύει την κατάσταση ενός φιλόσοφου (με τιμές από: ΣΚΕΦΤΕΤΑΙ, ΠΕΙΝΑΕΙ, ΤΡΩΕΙ).

Για να φάει ένας φιλόσοφος, κανείς από τους 2 γείτονές του δεν μπορεί να βρίσκεται στη κατάσταση ΤΡΩΕΙ.

```
philosopher (i)
  while (1) {
      think;
      take_forks(i);
      eat;
      put_forks(i);
  }
take_forks(i)                 put_forks(i)
    down (&mutex);                down (&mutex);
    state[i]: = hungry;           state[i]: = thinking;
    test (i);                      test[i-1mod 5];
    up (\&mutex);                  test[i+1 mod 5];
    down (& s[i]);                up (&mutex);

test (i)
   if (state[i] == hungry && state[i-1 mod 5] != eating
                    && state[i+1 mod 5] != eating)
   {
      state[i] = eating;
      up (& s[i])
   }
```

2.3.11 Το Πρόβλημα Αναγνωστών/Εγγραφών (Readers & Writers)

Χρησιμοποιείται κυρίως για την μοντελοποίηση προβλημάτων ταυτόχρονης πρόσβασης σε βάσεις δεδομένων.

Πολλοί αναγνώστες (readers) μπορούν ταυτόχρονα να προσπελαύνουν ένα αντικείμενο. Αν όμως αυτό προσπελαύνεται από έναν εγγραφέα, τότε κανείς άλλος (reader ή writer) δεν μπορεί να προσπελάσει το αντικείμενο.

π.χ.
τραπεζικοί λογαριασμοί = data items -- ερωτήσεις για ύψος υπολοίπου = readers -- αναλήψεις, καταθέσεις = writers

```
reader
   while (1) {
       down (&mutex);
       read_count ++;
       if (read_count==1)
          down (&item);
       up (&mutex) ;
       read_item();
       down (&mutex);
       read_count --;
       if (read_count == 0)
          up (&item);
       up (&mutex);
   }
```

```
writer
      while (1) {
              create_data();
              down (& item) ;
              write_data_item();
              up (&item)
      }
```

Αρχικά, mutex = 1 και item = 1. Υπάρχει κάποιο πρόβλημα μ' αυτή την λύση;

2.4 Χρονοπρογραμματισμός CPU - SCHEDULING

Οταν πολλά processes βρίσκονται στην κατάσταση RUNNABLE / READY, ποιό process θα πάρει τον CPU; Αυτό το αποφασίζει εκείνο το τμήμα του Λ.Σ. που λέγεται scheduler (scheduling algorithm).

Ποιός χρονοπρογραμματίζει τον χρονοπρογραμματιστή; Ο scheduler καλείται:
1. από την ρουτίνα εξυπηρέτησης διακοπής ρολογιού (clock interrupt service routine) και
2. από την ρουτίνα block() του kernel (που καλείται όταν το τρέχον process θα μπλοκάρει).

Πότε ένας αλγόριθμος δρομολόγησης είναι "καλός";

- Δίκαιος: ισοκατανομή CPU κύκλων στα processes
- Αποδοτικός:
    - CPU utilization (ποσοστό χρήσης του CPU)
    - response time (χρόνος απόκρισης - interactive)
    - turnaround time (χρόνος απόκρισης - batch)
    - throughput (# processes/μονάδα χρόνου)

Δεν μπορούμε να ικανοποιήσουμε όλα αυτά ταυτόχρονα !
(batch εναντίον interactive).

Process Τύπος: I/0 - bound ; ή CPU bound ;

Σε κάθε "τικ" το clock interrupt service routine καλεί τον scheduler ο οποίος εξετάζει αν το τρέχον process έχει χρησιμοποιήσει τον CPU αρκετά. Αν ναί, τότε κάποιο άλλο process (αυτό που θα διαλέξει ο αλγόριθμος δρομολόγησης θα τρέξει). Αν όμως το τρέχον process είναι I/O-bound τότε συνήθως θα μπλοκάρει συχνά, και έτσι θα χρησιμοποιεί λιγότερο το CPU απ' ότι ένα CPU-bound process.

Preemptive Scheduling:

Οταν ο αλγόριθμος επιτρέπει το CPU να δοθεί σ' άλλο process πριν το τρέχον process να έχει τελειώσει.

Non-preemptive αλγόριθμοι δίνουν το CPU σ' ένα process μέχρι να τελειώσει, μετά διαλέγουν άλλο process ... κ.ο.κ.

Το preemptive scheduling δημιουργεί τα προβλήματα που είδαμε σχετικά με ταυτόχρονη πρόσβαση σε κοινή μνήμη. Απ' την άλλη μεριά, non-preemptive scheduling φαίνεται μη εφαρμόσιμο για τις πιο πολλές εφαρμογές (που έχουν πολλά processes στο σύστημα ταυτόχρονα).

Round - Robin Scheduling:

Απλός, δίκαιος, και πιο πολύ-χρησιμοποιημένος αλγόριθμος. Το κάθε process έχει ένα time quantum (ή time slice). Ετσι θα τρέξει για ένα διάστημα ίσο με το time slice, μετά το CPU θα δοθεί σε κάποιο άλλο process, κ.ο.κ. Αν το process μπλοκάρει πριν το time slice παρέλθει, τότε πάλι το CPU θα δοθεί σ' άλλο process. Απλή υλοποίηση!

Η διαδικασία κατά την οποία δίνεται το CPU σ' άλλο process αποκαλείται CPU switch, process switch, & **context switch**. Ο χρόνος που απαιτείται για το context switch είναι σημαντικός: αλλαγή "περιβάλλοντος" : σώσιμο regs, PC, SP, ... στη δομή για το τρέχον process στο Process Table, φόρτωμα των regs, PC, SP από τη δομή για το επόμενο process στο Process Table, αλλαγή Page Tables, ...κ.ο.κ.

Η αποδοτικότητα του Round-Robin αλγόριθμου εξαρτάται από την διάρκεια του quantum (σε σχέση και με τον χρόνο που απαιτείται για context switch). Μεγάλο quantum ➔ μεγάλος χρόνος απόκρισης για interactive χρήστες. Μικρό quantum ➔ για μεγάλα ποσοστά του χρόνου το CPU χρησιμοποιείται για context switch.

Χρονοπρογραμματισμός Με Προτεραιοτητες

Ο αλγόριθμος Round-Robin είναι χρήσιμος όταν όλα τα processes έχουν την ίδια σπουδαιότητα. Αυτό όμως δεν ισχύει συχνά!

Σε μερικές εφαρμογές, processes κάποιων χρηστών έχουν μεγαλύτερη προτεραιότητα (π.χ. στρατηγών, καλών πελατών). Γενικά, ο αλγόριθμος διαλέγει το process με την μεγαλύτερη προτεραιότητα.

Συχνά, η προτεραιότητα ενός process μεταβάλλεται με το χρόνο και εξαρτάται από το πόσα CPU cycles έχει ήδη χρησιμοποιήσει.

Σε πολλά συστήματα υπάρχουν αρκετά **I/O-bound jobs**. Αυτά πρέπει νάχουν μεγαλύτερη προτεραιότητα! Ετσι, θα πάρουν το CPU και πολύ σύντομα θα ζητήσουν I/O. Τότε το I/O process και τ' άλλα CPU-bound process τρέχουν παράλληλα. Αυτός ο παραλληλισμός μειώνει τον μέσο χρόνο απόκρισης και αυξάνει το throughput του συστήματος.

Ενας απλός τρόπος υπολογισμού προτεραιότητας: 1/p, όπου p είναι το ποσοστό του time slice που χρησιμοποίησε το process την τελευταία φορά που έτρεξε. I/O-bound processes έχουν χαμηλό p => υψηλή προτεραιότητα.

Υβριδικός αλγοριθμος: Round-Robin και προτεραιοτητες

Το σύστημα παρέχει N διαφορετικές προτεραιότητες οι οποίες σχηματίζουν M ομάδες, καθεμιά με N/M προτεραιότητες.

Ο αλγόριθμος διαλέγει πάντα ένα process από την ομάδα με τις μεγαλύτερες προτεραιότητες.

Σε κάθε ομάδα χρησιμοποιείται Round-Robin scheduling π.χ. N=M=10: Αν υπάρχει κάποιο process με προτεραιότητα 1 τότε αυτό τρέχει, αλλοιώς κάποιο process με προτεραιότητα 2, κ.λπ. Ολα τα processes με προτεραιότητα 1 τρέχουν με Round-Robin κ.λπ. Με το πέρασμα του χρόνου οι προτεραιότητες πρέπει να αλλάζουν. Αλλοιώς μερικά processes δεν θα τρέξουν ποτέ.

Ο υβριδικός αλγόριθμος υλοποιείται συνήθως μ' ένα multi-level queue. Το 1ο queue έχει processes της 1ης ομάδας, κ.λπ. Ο αλγόριθμος βρίσκει το πρώτο μη άδειο queue, διαλέγει το process στην κορυφή του queue και όταν τελειώσει το quantum του process το τοποθετεί στο τέλος του ίδιου queue. Το Λ.Σ. UNIX χρησιμοποιεί αυτό τον αλγόριθμο (με N=128, M=32, quantum=100 Msec).

Είναι δυνατόν κάθε ομάδα να σχετίζεται με διαφορετικό quantum. Δηλ. η κατώτερη ομάδα νάχει 1 quantum, η αμέσως επόμενη νάχει 2 quanta, κ.λπ. Ετσι ένα process με προτεραιότητα Π τρέχει για Π x quanta χρόνο.

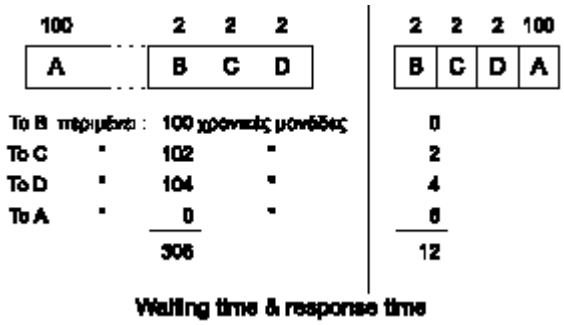
Είναι επίσης δυνατόν οι προτεραιότητες των processes να απορρέουν από τον τύπο του processes.
π.χ.
processes που περιμένουν για terminal I/O <-> 1
processes που περιμένουν για disk I/O <-> 2
processes των οποίων το quantum τελείωσε <-> *3*
processes που έχουν χρησιμοποιήσει πολλά quanta <-> 4.

Σ' αυτά τα συστήματα υπάρχουν 4 queues ένα για κάθε προτεραιότητα. Ο αλγόριθμος διαλέγει πάντα το process στην κορυφή του queue με την μεγαλύτερη προτεραιότητα. Ετσι, interactive χρήστες/processes και disk I/O processes προτιμούνται.

Ολοι οι παραπάνω αλγόριθμοι σχεδιάστηκαν για συστήματα με πολλά interactive processes. Αν το σύστημά μας έχει batch jobs, τότε κάτι άλλο χρειάζεται.

Η Μικρότερη Δουλειά Πρώτη (SHORTEST JOB FIRST) (SJF)

SJF είναι χρήσιμος όταν ο χρόνος CPU που χρειάζεται το κάθε process είναι γνωστός από πριν (π.χ. εφαρμογές όπως ασφαλιστικές, αεροπορικές εταιρείες, κ.λπ). SJF διαλέγει πάντα το process που χρειάζεται λιγότερο το CPU.



Το παράδειγμα δείχει ότι το SJF είναι βέλτιστο (μπορεί ν' αποδειχθεί εύκολα).

Μιας και to SJF ελαχιστοποιεί τον χρόνο απόκρισης είναι επιθυμητό να τον χρησιμοποιήσουμε και για interactive συστήματα. Για να γίνει όμως αυτό πρέπει να μπορούμε να υπολογίσουμε πόσο CPU χρόνο απαιτεί κάθε process (ή μάλλον κάθε εντολή που ένας χρήστης πληκτρολόγησε στο τερματικό του).

Εστω ότι εκτιμήθηκε ότι ένα job χρειάζεται χρόνο T0.

Οταν τρέχει σημειώνεται ότι χρησιμοποίησε το CPU για χρόνο T1. Τότε σημειώνουμε την εκτιμησή μας με βάση T0 & T1. π.χ. $T0 = \alpha T0 + (1-\alpha)T1$ αποτελεί την καινούργια εκτίμηση. Την επόμενη φορά που τρέχει το ίδιο job σημειώνουμε ότι πήρε χρόνο T2. Τότε ενημερώνουμε την εκτίμησή μας με βάση T0 & T2. π.χ. $T0 = \alpha\, T0 + (1-\alpha)\, T2$

Αν $\alpha=1/2$ ➔ $T0 = (T0 / 2) + (T1 / 2)$, $T0 = (T0 / 4) + (T1 / 4) + (T2 / 2)$, ...

Οσο πιο μικρή η τιμή του α, τόσο πιο γρήγορα ξεχνιούνται οι παλιές εκτιμήσεις....

Δυστυχώς το SJF δεν είναι βέλτιστο όταν όλα τα jobs που θα δρομολογηθούν δεν είναι γνωστά εκ προϊμίου. Φτιάξε ένα δικό σας παράδειγμα που να δείχνει αυτό.

Χρονοπρογραμματισμός με εγγυήσεις - Guaranteed Sched.

Βασική ιδέα: Δώσε εγγυήσεις σε κάθε χρήστη αναφορικά με το πόσα CPU cycles θα δοθούν στο process του. π.χ. Αν υπάρχουν N χρήστες, τότε όλοι θα πάρουν 1/N των CPU cycles.

Η υλοποίηση:

1. Το Λ.Σ. υπολογίζει πόσο CPU χρόνο έχει πάρει το κάθε process.
2. Επίσης, υπολογίζει πόσο χρόνο έπρεπε να είχε πάρει (login time/N).
3. Μετά υπολογίζει τον λόγο των (1) και (2).
4. Ο αλγόριθμος διαλέγει το process με τον μικρότερο λόγο, το οποίο τρέχει μέχρι ο λόγος του να γίνει μεγαλύτερος κάποιου άλλου process.

Διαχωρισμός Πολιτικής & Μηχανισμού Χρονοπρογραμματισμού

Ο διαχωρισμός αυτός είναι καλή ιδέα γενικά και όχι μόνο για scheduling. Η πολιτική αναφέρεται στην "φιλοσοφία" του αλγόριθμου. Ο μηχανισμός αναφέρεται κυρίως στην "υλοποίηση" του αλγορίθμου.

Παράδειγμα: Ενας χρήστης μπορεί ν' αποφασίζει για τις σχετικές προτεραιότητες των διεργασιών του. Ο kernel τις δρομολογεί ανάλογα μ' αυτές τις προτεραιότητες. Παρομοίως, ένα parent process δίνει προτεραιότητες στα παιδιά του.

➔ ο αλγόριθμος που εκτελείται από το Λ.Σ. έχει παραμέτρους που τις δίνουν user processes.

Πολυ-Επίπεδος Χρονοπρογραμματισμός (2-Level Scheduling)

Μοντέρνα συστήματα συνήθως έχουν πολλά (πολλές φορές) μεγάλα processes που θέλουν να τρέξουν. Επειδή Κ.Μ. κοστίζει πολύ, είναι συνήθως μικρή => δεν χωράνε όλα τα processes στην κύρια μνήμη.

Η δρομολόγηση των processes σ' αυτά τα συστήματα γίνεται ως εξής:

1. short-term scheduling: Αφορά την δρομολόγηση που έχουμε δει μέχρι τώρα. Δηλ. διαλέγει ποιό από τα processes που βρίσκονται στην Κ.Μ. θα τρέξει
2. Long-term scheduling: Υπεύθυνο για να καθορίζει ποιά από όλα τα processes που θέλουν να τρέξουν θα εισέλθουν στην κύρια μνήμη.

Ετσι, αντί να έχουμε έναν μονολιθικό αλγόριθμο που λαμβάνει υπ' όψιν του το κόστος για context switch ενός process που βρίσκεται στο δίσκο, υπάρχει ο παραπάνω διαχωρισμός ευθυνών.

Συνήθως ο LTS χρησιμοποιεί (μερικά από) τα εξής κριτήρια:
- χρόνος παραμονής στο δίσκο ή Κ.Μ.
- χρόνος CPU που δόθηκε στο process
- προτεραιότητα του process
- μέγεθος του process (συνήθως τα μεγάλα processes στέλνονται στο δίσκο για νάρθουν πολλά μικρά ...).

Κεφάλαιο 3. Διαχείριση Κύριας Μνήμης

3.1 Απλή Διαχείριση: χωρίς Swapping και Paging

- Ο **Διαχειριστής Μνήμης (ΔΜ) (memory manager)** πρέπει να:

  - ξέρει ποιά τμήματα της ΚΜ χρησιμοποιούνται
  - μπορεί ν' αναθέτει μνήμη σε processes όταν την χρειάζονται
  - παίρνει πίσω μνήμη που "ελευθερώνεται" από processes
  - διαχειρίζεται το swapping processes μεταξύ ΚΜ & δίσκου.

- Οι ΔΜ χωρίζονται σε κατηγορίες ανάλογα με το αν επιτρέπουν, paging, swapping (δηλ. μεταφορά τμημάτων ή ολόκληρων processes μεταξύ ΚΜ και δίσκου).
- Χωρίς swapping και paging ο ΔΜ είναι απλός.

Μονοπρογραμματισμός (Monoprogramming)

- Μόνο ένα process βρίσκεται στη ΚΜ κάθε φορά
- Η ΚΜ χωρίζεται σε 2 μέρη: ένα για το Λ.Σ. και ένα για το user process που τρέχει Κ.Μ.

Κ.Μ.	Λ.Σ.	user program

- Μόλις ο χρήστης ζητήσει μέσω terminal να τρέξει ένα πρόγραμμα, το Λ.Σ. το "φορτώνει" στη ΚΜ και το τρέχει
- Το monoprogramming δεν αρκεί στις πιο πολλές περιπτώσεις:

  - I/O-bound jobs: ➔ το CPU υποαπασχολείται ➔ κακή απόδοση.
  - Πολλά interactive processes: ➔ χρόνος απόκρισης πολύ μεγάλος.
  - Πολλές εφαρμογές θέλουν την δυνατότητα δημιουργίας πολλών processes που τρέχουν "ταυτόχρονα".

➔ χρειαζόμαστε multiprogramming ➔ η δουλειά του Δ.Μ. γίνεται πολύ πιο δύσκολη

Μοντέλο για Πολυπρογραμματισμό (multiprogramming)

- Εστω P η πιθανότητα ότι σε μια χρονική στιγμή ένα process βρίσκεται στη κατάσταση I/O-wait
- Εστω επίσης ότι υπάρχουν N τέτοια processes
- Το CPU utilization δίνεται από την φόρμουλα:

$CPU\ utilization = 1 - P^N$

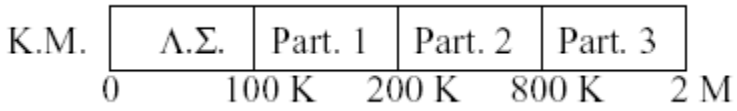
➔ με P = 80% το CPU utilization είναι:

N	1	2	3	4
CPU util.	20%	36%	49%	59%

➔ το multiprogramming έχει σοβαρές ευεργετικές συνέπειες. Σημείωση: Το παραπάνω μοντέλο είναι υπεραπλουστευμένο. Γιατί;

Διαχείριση Μνήμης με Μη-μεταβαλλόμενα Τμήματα (Partitions)

- Το partitions είναι ένα τμήμα, σταθερού μεγέθους, μνήμης. Διαφορετικά partitions μπορεί νάχουν διαφορετικό μέγεθος.
- Το κάθε partition μπορεί να "φιλοξενεί" processes (το μέγεθος των οποίων πρέπει να είναι < του μεγέθους του partition).



- Ο Δ.Μ. έχει μια ουρά από processes που περιμένουν να μπούν σ' ένα partition. Μπορεί να υπάρχει μια ουρά για όλα τα partitions ή μια ουρά για κάθε partition.

  - Στη τελευταία περίπτωση μπορεί να υπάρχει διαθέσιμη μνήμη και να μη τρέχει κάποιο process στο διαθέσιμο partition.
  - Στη 1η περίπωση ο Δ.Μ. πρέπει να προσέξει να μην σπαταλεί μνήμη.

  π.χ. Αν ένα μικρό process μπεί στο Part.3 τότε αν και υπάρχει πολύ διαθέσιμος χώρος στο Part.3 δεν μπορεί να χρησιμοποιηθεί. Αυτό ονομάζεται **εσωτερικός κατακερματισμός (internal fragmentation** (αναφέρεται στον αχρησιμοποίητο χώρο του partition).

  - Η λύση σ' αυτό το πρόβλημα δεν είναι προφανής. Για παράδειγμα αν κάθε φορά ο Δ.Μ. ψάχνει για το process στην ουρά που ελαχιστοποιεί το internal fragmentation, τότε μικρά processes θ' αδικούνται (θυμηθείτε τα πλεονεκτήματα του SJF αλγόριθμου και ότι μικρά, διαδραστικά (interactive) processes πρέπει να έχουν υψηλή προτεραιότητα).
  - Μια λύση είναι το κάθε μικρό process να σχετίζεται με μια μεταβλητή, , που αντιπροσωπεύει πόσες φορές ο scheduler δεν έδωσε το partition στο process. Μετά από φορές, το partition δίνεται στο process.

Θεμελιώδη Προβλήματα των PARTITIONS

- Ακόμα και με monoprogramming υπάρχουν 2 partitions.
- Ο linker και ο loader συνεργάζονται και παράγουν τις διευθύνσεις των εντολών και των δεδομένων του προγράμματος. Ο linker παράγει **"λογικές" διευθύνσεις και όχι "φυσικές"**, - δηλ. η πρώτη διεύθυνση είναι 0, κ.λπ. Ομως, στη φυσική διεύθυνση 0 βρίσκονται εντολές ή δεδομένα του Λ.Σ. **Δηλαδή τίθεται θέμα προστασίας του Λ.Σ.**
- **Η λύση εστιάζει στη "μετάφραση" των λογικών addresses σε "φυσικές" addresses.**
  Για παράδειγμα αν το process έχει μπει στο Part1 τότε σ' όλες τις διευθύνσεις που παράγονται κατά την εκτέλεση του process, το hardware προσθέτει 100 K.
- Συνήθως η παραπάνω στρατηγική υλοποιείται με την χρησιμοποίηση ενός ειδικού **καταχωρητή**: **base register**. Στο παραπάνω παράδειγμα, όταν το process φορτωθεί στο Part. 1 και αμέσως πριν του δοθεί το CPU το base register παίρνει την τιμή 100 K. Αυτό γίνεται από τον kernel του Λ.Σ. (δηλ. οι user processes δεν έχουν πρόσβαση σ' αυτόν τον register).
- Ετσι επιτυγχάνεται το επιθυμητό αποτέλεσμα.

Ο καταχωρητής βάσης (base register) όμως δεν αρκεί!

Στην περίπτωση του multiprogramming πρέπει το σύστημα να εγγυάται ότι καθένα πρόγραμμα θα προστατεύεται από τ' άλλα που είναι στη Κ.Μ.

➔ απαιτείται έλεγχος σχετικά με τα addresses που μπορεί να παράγει ένα process. π.χ. το process στο Part1 δεν μπορεί να παράγει διευθύνσεις > 200K γιατί τότε θα κάνει ζημιά στο process που τρέχει στο Part2.

Αυτό το πρόβλημα λύνεται μ' ένα άλλο ειδικό **καταχωρητή ορίου (limit register)**. Ο limit register περιέχει την μέγιστη διεύθυνση του partition στο οποίο τρέχει το process. Κάθε διεύθυνση που παράγεται, λοιπόν, προστίθεται στο base register και το αποτέλεσμα συγκρίνεται με το limit register: αν είναι μεγαλύτερο, το process πεθαίνει.

Σημείωση: Με base και limit registers τα προγράμματα μπορούν να μετακινηθούν στην Κ.Μ. (δηλαδή ν' ανατεθούν σε διαφορετικά partitions κατά την διάρκεια της εκτέλεσής τους).
Αυτό ονομάζεται **relocation.**

3.2 Swapping

- Τί γίνεται όταν υπάρχουν πιο πολλά processes απ' όσα μπορεί να υποστηρίξει η Κ.Μ.;
  Με batch systems αρκεί κάτι σαν multiprogramming με fixed partitions. Με interactive systems, όμως, δεν αρκεί.
- Η μεταφορά processes μεταξύ ΚΜ και swap space στο δίσκο ονομάζεται **swapping**.

Διαχείριση Μνήμης με Μεταβαλλόμενα Τμήματα (Multiprogramming with variable partitions)

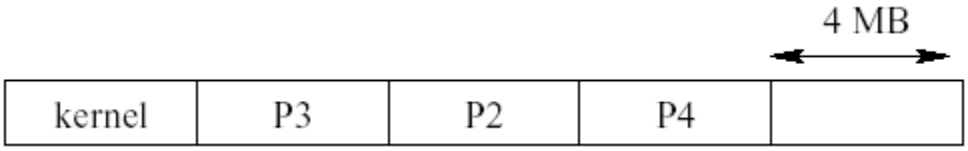
- Η έννοια των fixed partitions δεν βοηθά πολύ: Συχνά, σπαταλείται πολύ χώρος ΚΜ γιατί δεν γίνεται καλό "ταίριασμα" μεταξύ μεγεθών partition & process.

  ➔ **variable partitions** (μεταβαλλόμενου μεγέθους τμήματα).

- Βασική ιδέα: οι διευθύνσεις, ο αριθμός, και τα μεγέθη των partitions μεταβάλλονται ανάλογα με τις ανάγκες.
- Το σημαντικό αποτέλεσμα είναι ότι τα μεταβαλλόμενα partitions ελαχιστοποιούν την σπατάλη χώρου στη ΚΜ. Από την άλλη πλευρά, όμως, η ανάθεση ΚΜ σε processes και γενικά το έργο διαχείρισης ΚΜ δυσκολεύει.
- **Παράδειγμα: Variable Partitions**

Κ.Μ.	Kernel	P1				Μόλις ήρθε το process P1
>>	Kernel	P1		P2		Μόλις ήρθε και το process P2
>>	Kernel			P2		Έφυγε το process P1
>>	Kernel	P3		P2		Ήρθε το process P3
>>	Kernel	P3		P2	P4	Ήρθε το process P4

- Βλέπουμε ότι υπάρχουν 2 "άδεια" τμήματα της ΚΜ, έστω με μέγεθος 3MB και 1MB αντίστοιχα. Αν τώρα έρθει ένα process P5 που χρειάζεται 4MB, δεν μπορεί να γίνει δεκτό, παρότι υπάρχει 4MB ελεύθερος χώρος στη ΚΜ. Αυτό το φαινόμενο ονομάζεται **εξωτερικός κατακερματισμός (external fragmentation)**.
- Μια λύση στο πρόβλημα του external fragmentation είναι το **storage compaction** που συνδέει όλους τους άδειους χώρους της ΚΜ σ' ένα συνεχόμενο τμήμα της ΚΜ. π.χ.

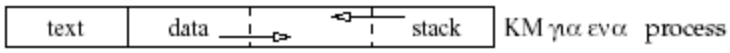


Σπάνια όμως χρησιμοποιείται γαιτί απαιτεί χρονοβόρο memory-to-memory copying (CPU time) (1MB/sec).

- **Πόση μνήμη ανατίθεται σ' ένα process;**

Η δυσκολία έγκειται στο ότι συνήθως το μέγεθος ενός process μεταβάλλεται κατά την εκτέλεσή του. Ενα process έχει τρία τμήματα:

- **text**: το object code - δεν μεταβάλλεται
- **data**: τα δεδομένα -μεταβάλλεται [βλ. malloc()]
- **stack**: το stack επίσης μεταβάλλεται



Το data τμήμα μεγαλώνει προς το stack και αντίστροφα.

- Προσέξτε: αν δίναμε ακριβώς όσο χώρο χρειάζεται ένα process όταν κάναμε swap in τότε θ' απαιτείται mem-to-mem copying για να μεταφέρουμε ένα process που μεγαλώνει σε κάποιο άλλο partition.

Διαχείριση Μνήμης

- Θεμελιώδες θέμα είναι:

  - ποιά είναι τα άδεια τμήματα της μνήμης;
  - ποιό process έχει ποιό partition της μνήμης

- Οι απαντήσεις σ' αυτά τα ερωτήματα δίνονται χρησιμοποιώντας 3 εναλλακτικούς τρόπους:

  - Bit Maps: πίνακες (vectors) που περιέχουν ένα bit ανά partition
  - Linked Lists: λίστες των οποίων οι κόμβοι αντιστοιχούν σε partitions
  - Buddy Systems: βασίζεται σε διαφορετικές λίστες, οι κόμβοι των οποίων αντιστοιχούν σε partitions των οποίων το μέγεθος είναι διαφορετικές δυνάμεις του 2.

Bit Maps

- Η κύρια μνήμη χωρίζεται σε "μονάδες ανάθεσης" (allocation units) των οποίων το μέγεθος είναι λίγα bytes ή λίγα Kbytes.
- Το μέγεθος των μονάδων ανάθεσης είναι σταθερό.
- Σε κάθε μονάδα ανάθεσης αντιστοιχεί ένα bit στον Bit Map. Αν η *I* θέση είναι 1, τότε η μονάδα ανάθεσης *I* δεν χρησιμοποιείται από κανένα process.
- Οσο πιο μεγάλο είναι το μεγέθος των μονάδων ανάθεσης τόσο πιο μικρό είναι το Bit Map. Από την άλλη μεριά όμως, μεγάλες μονάδες ανάθεσης έχουν σαν αποτέλεσμα σπατάλη χώρου, λόγω internal fragmentation (η τελευταία μονάδα ανάθεσης δεν θα χρησιμοποιείται όλη).
- Το βασικότερο πλεονέκτημα των Bit Maps είναι η απλότητα τους (ευκολία υλοποίησης).
- Το βασικότερο μειονέκτημα έγκειται στο κόστος ανάθεσης. Οταν ένα process γίνεται swapped in, πρέπει να βρεθούν στο Bit Map αρκετά συνεχόμενα 0 - μια χρονοβόρα διαδικασία.

Linked Lists

- Στην πιο απλή μορφή τους, κάθε κόμβος αντιστοιχεί είτε σε τμήμα που χρησιμοποιείται από κάποιο process, είτε σ' ένα "ελεύθερο" τμήμα.

- Οι κόμβοι της λίστας είναι ταξινομημένοι με βάση τη διεύθυνση του 1ου byte του τμήματος που αντιπροσωπεύουν.

Η ταξινόμηση βοηθά στην διαχείριση:

- όταν ένα process τελειώσει ή γίνει **swapped out** η ενημέρωση της λίστας είναι γρήγορη: πρέπει να εξεταστούν ο επόμενος και ο προηγούμενος κόμβος ούτως ώστε να επιτευχθεί η συγχώνευση των άδειων τμημάτων (**holes)** - δηλ. >=2 συνεχόμενα holes (άδεια τμήματα) να συγχωνευθούν σ' ένα.

Τα πεδία ενός κόμβου είναι: 1 bit (process ή hole), 1 αριθμός (αρχική διεύθυνση του τμήματος), 1 αριθμός (μέγεθος του τμήματος), 2 pointers (σε προηγούμενο και επόμενο κόμβο).

Αλγόριθμοι Ανάθεσης Μνήμης σε Processes

Εκτελούνται κατά την διάρκεια δημιουργίας ενός process ή κατά το swap in ενός process.

Μέθοδος First-Fit: Βρίσκει τον 1ο hole που χωράει το process

Μέθοδος Next-Fit: Σαν τον first-fit, μόνο που κάθε φορά ξεκινά το ψάξιμο από το προηγούμενο hole που βρήκε

Μέθοδος Best-Fit: Βρίσκει το hole με το μικρότερο internal fragmentation

Μέθοδος Worst-Fit: Βρίσκει το hole με το μεγαλύτερο fragmentation

Η Best-Fit είναι πιο αργή, προφανώς, από το first-fit γιατί απαιτεί πιο πολύ ψάξιμο.

Η βασική ιδέα του worst fit είναι ν' αφήνει όσο το δυνατόν μεγαλύτερα τμήματα προς μελλοντική χρησιμοποίηση. Πειραματικά, όμως, έχει αποδειχθεί ότι το worst fit δεν είναι αποδοτικό.

Το best fit έχει χαμηλότερο memory utilization σε σχέση με το first fit. Αυτό εξηγείται διότι το best fit έχει την τάση ν' αφήνει πολύ μικρά "holes" που δεν μπορούν να χωρέσουν processes.

Ενας τρόπος να αυξηθεί η επίδοση των παραπάνω αλγόρίθμων είναι να χρησιμοποιούνται 2 διαφορετικές λίστες με processes και με holes (αυτό όμως αυξάνει το κόστος όταν ένα process τελειώνει ή γίνει swapped out).

Αν η λίστα των holes είναι ταξινομημένη με βάση το μέγεθος τότε best fit & first fit είναι εξίσου γρήγορα.

Ο αλγόριθμος **quick fit**:

- κρατά πολλές λίστες holes η κάθε μια αντιπροσωπεύει holes με διαφορετικά μεγέθη.
- έτσι βρίσκεται πολύ γρήγορα ένα hole ενός συγκεκριμένου μεγέθους.
- το μειονέκτημά του έγκειται στο οτι όταν ένα process τελειώσει ή γίνει swapped out, τότε είναι πιο χρονοβόρο ν' εξετασθούν οι περιπτώσεις για συγχώνευση holes.

Buddy Systems

- Αντιμετωπίζει το μειονέκτημα του quick fit. Διαχειρίζεται holes των οποίων το μέγεθος είναι 1,2,4,8,16,32... bytes (δηλ. δυνάμεις του 2). Ετσι χρειάζονται λογαριθμικός αριθμός από λίστες.
- Αρχικά υπάρχει μόνο μια λίστα, μ' ένα hole ίσο με το μέγεθος της μνήμης.
- Αιτήσεις για μνήμη γίνονται πάντα για τμήματα με μέγεθος που είναι δύναμη του 2. (δηλ. ένα process 70K θα ζητήσει 128K μνήμης).
- Αν δεν υπάρχει hole με το κατάλληλο μέγεθος τότε ένα μεγαλύτερο hole διασπάται ως εξής:

Ενα hole π.χ. 512K διασπάται σε 2 holes **buddies** 256K έκαστο.
Το ένα εκ των δύο holes των 256K διασπάται σε 2 buddies 128 K έκαστο, το ένα εκ των οποίων θα δοθεί στο process.
Δηλαδή, κάθε φορά βρίσκεται το μικρότερο hole, h, που μπορεί να ικανοποιήσει μια αίτηση a. Αν χρειάζεται, τότε το h διασπάται σε δύο ίσα τμήματα και επαναλαμβάνεται η διαδικασία.

Οταν τμήματα ελευθερώνονται απο processes τότε η συγχώνευση γίνεται μόνο σε 2 buddies => δηλ. μόνο η λίστα με τα καταλλήλου μεγέθους holes προσπελαύνεται για την συγχώνευση.

Τα μειονεκτήματα έγκειται στο **internal fragmentation** (αφού τα holes είναι δυνάμεις του 2) και στο **external fragmentation.**

3.3 Εικονική Μνήμη (Virtual Memory)

Τί γίνεται όταν προγράμματα είναι μεγαλύτερα της ΚΜ; Δηλαδή το σύνολο των μεγεθών των text, data και stack υπερβαίνει το όριο/μέγεθος της ΚΜ;

Η λύση είναι η τεχνική **virtual memory: Το Λ.Σ. κρατά μερικά τμήματα ενός process στη ΚΜ ενώ άλλα τμήματα βρίσκονται στο swap space.**

- Συνήθως λίγα κομμάτια του text, data & stack είναι στη Κ.Μ. Αυτά τα κομμάτια αφορούν τις εντολές, και δεδομένα που χρειάζονται άμεσα καθώς και αυτά που αναμένεται να χρησιμοποιηθούν στο εγγύς μέλλον.
- Ετσι αυτό που γίνεται swap in & out είναι κομμάτια των text, data, stack τμημάτων ενός process, και όχι ολόκληρα processes.

3.3.1 Σελιδοποίηση (PAGING)

Μια τεχνική υλοποίησης του virtual memory.

Η έννοια του **Page** αναφέρεται στο κομμάτι εκείνο ενός process που γίνεται swapped in & out - δηλαδή είναι η μονάδα swap. [Χρησιμοποιούνται επίσης και οι έννοιες **pagein & pageout**]

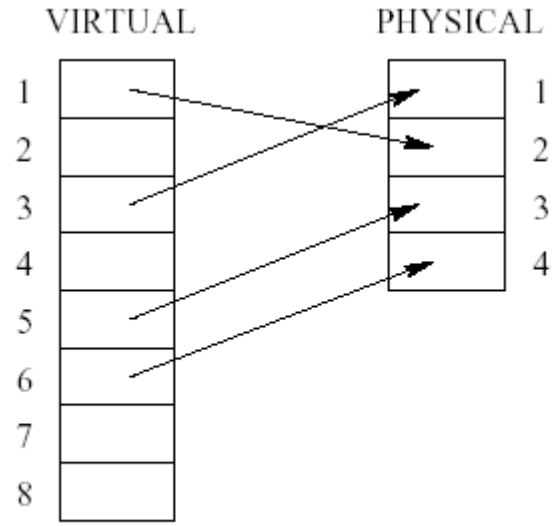
Το σύνολο των memory addresses που παράγονται κατά την εκτέλεση ενός προγράμματος ονομάζεται **address space**. Επειδή, στη γενική περίπτωση, αυτό το address space είναι μεγαλύτερο της ΚΜ, δεν μπορεί να υπάρξει μονοσήμαντη αντιστοιχία μεταξύ ενός address που παράγει ένα πρόγραμμα και ενός **physical address. Ετσι τα addresses που παράγονται από ένα πρόγραμμα λέγονται virtual addresses ➔ virtual address space**.

Χωρίς virtual memory, συνεπώς, ότι διεύθυνση παράγεται δίνεται στο memory bus το οποίο προσπελαύνει την διεύθυνση. Οταν όμως υπάρχει virtual memory, απαιτείται η παραγόμενη διεύθυνση να "μεταφραστεί" πριν δοθεί στο memory bus. Αυτή η μετάφραση γίνεται από ειδικό hardware που λέγεται **memory management unit (MMU)**.

Το virtual address space χωρίζεται/διαιρείται σε μονάδες μνήμης που λέγονται **pages.** Το κάθε page (σελίδα) όταν χρειαστεί αποθηκεύεται σε μια σελίδα (**page frame**) της ΚΜ. Pages & frames, συνεπώς, έχουν το ίδιο μέγεθος [συνήθως από 512 bytes μέχρι 8 K bytes).

Η μονάδα μνήμης που γίνεται swap είναι το page.

Παράδειγμα: Θεωρείστε ένα κομπιούτερ με 16-bit addresses (δηλ. virtual address space = 64K), με 32 KB μνήμη και με page size = 8K



Memory map

Το MMU μεταφράζει τις παραγόμενες διευθύνσεις ως εξής:
0-8K ➔ 8K-16K
8K-16K ➔ ;
16K-24K ➔ 0K-8K
24K-32K ➔ ; κ.ο.κ

Τα ερωτηματικά σημαίνουν ότι αυτές οι σελίδες πρέπει να γίνουν page in αντικαθιστώντας μία άλλη σελίδα.

Οταν το MMU βρίσκει ότι η σελίδα που περιέχει μια διεύθυνση δεν υπάρχει στη Κ.Μ., τότε κάνει ένα **trap στο Λ.Σ.** Αυτό το trap ονομάζεται **page fault**.

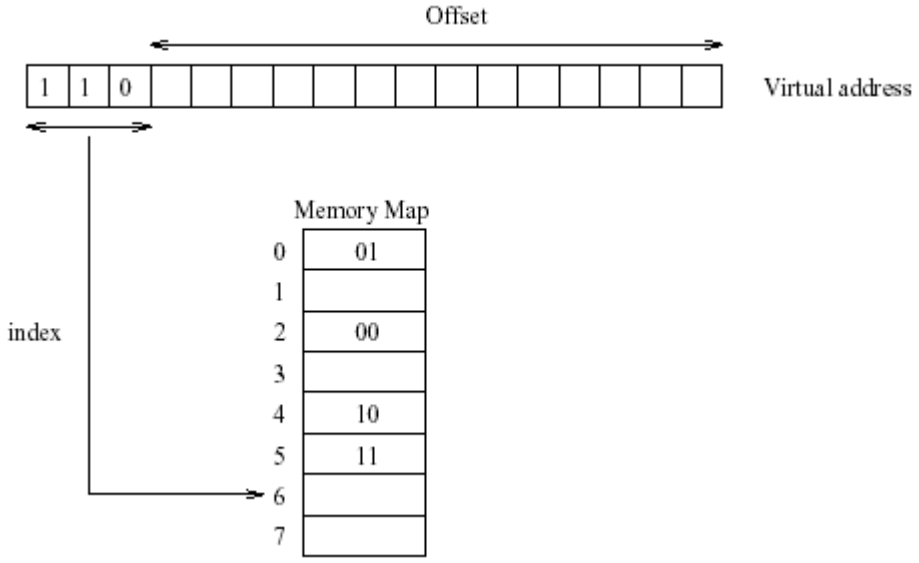
Το Λ.Σ. εξυπηρετεί ένα page fault ως εξής:

- από κάποιες δομές που διαχειρίζεται για το process που τρέχει βρίσκει που υπάρχει αυτή η σελίδα στο swap space του δίσκου.
- βρίσκει μια σελίδα που βρίσκεται ήδη στη Κ.Μ. (συνήθως την λιγότερο χρησιμοποιημένη) -"**θύμα**". Αν έχει ενημερωθεί, τότε την γράφει στη θέση της στο swap space.
- Κατόπιν, φέρνει την ζητούμενη σελίδα στο frame που ελευθερώθηκε.
- Τέλος ενημερώνει το memory map:
  - την εγγραφή που δείχνει στα frame που χρησιμοποιήθηκε - δηλ. την εγγραφή της σελίδας "θύμα".
  - την εγγραφή της ζητούμενης σελίδας.
- Τώρα, ξανακαλείται η εντολή που προκάλεσε το trap.

Η μετάφραση virtual -> physical:

Στο παράδειγμα μας, κάθε virtual address έχει 16 bits.
Αφού η σελίδα είναι 8K ➔ απαιτούνται 13 bits για τις διευθύνσεις των bytes μέσα στη σελίδα **(offset).** Με 64K virtual address space και 8K σελίδες μπορούμε να έχουμε 8 σελίδες virtual address ➔ απαιτούνται 3 bits.

➔ Τα 3 πρώτα bits ενός virtual address ορίζουν την virtual σελίδα, και τα υπόλοιπα 13 bits ενός virtual address ορίζουν ένα byte μέσα στη virtual σελίδα.



Αν η 6η virtual σελίδα τοποθετηθεί στο frame 3 τότε η μεταφρασμένη διεύθυνση **physical** είναι: 0110...0 + 000offset. Το αποτέλεσμα είναι 15 bits.

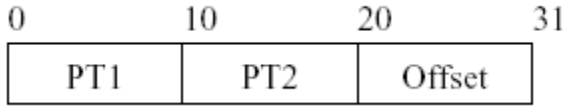
3.3.2 Πίνακες Σελίδων (Page Tables)

- Στην ουσία, τα Page Tables είναι αυτό που έχουμε ονομάσει έως τώρα Memory Maps.
- Η λειτουργία τους δηλ. είναι η ίδια:
  - Το virtual address χωρίζεται σε virtual page number (high-order bits) και offset (low-order bits).
  - Το virtual page number χρησιμοποιείται σαν index στο PT.
  - (ίσως μετά από page fault) το frame number στην εγγραφή του PT αποτελεί τα high-order bits που αντικαθιστούν τα high-order bits για το virtual page number. Ετσι παράγεται το σωστό physical address.
- Προκύπτουν όμως 2 σημαντικά προβλήματα:
  - Τα PTs μπορεί να είναι πολύ μεγάλα. Σημειώστε ότι κάθε process έχει δικό του PT!
  - Η λειτουργία πρέπει να είναι πολύ γρήγορη γιατί χρησιμοποιείται για κάθε διεύθυνση που παράγεται!

- Αναλογιστείτε ότι σχεδόν όλα τα μοντέρνα συστήματα έχουν >32-bit address ➔ >4GB virtual addr. space. Αν page size = 4K ➔το κάθε PT έχει ένα εκατομμύριο εγγραφές ...
- Επίσης, λόγω του μεγέθους των, PTs αποθηκεύονται στη Κ.Μ. (και όχι σε registers) έτσι τουλάχιστον διπλασιάζονται οι προσβάσεις στη ΚΜ που απαιτούνται για να εκτελεστεί μια εντολή ...! Συνήθως επιχειρούνται λύσεις που άλλοτε βασίζονται σε h/w (CPU registers) για το mapping ή σε τεχνικές που "μικραίνουν" το μέγεθος των PTs.

Πολυ-επιπεδοι Πίνακες (Multi-level Page Tables)

- Μια τεχνική που αποσκοπεί στο να τμηματοποιήσει το PT (που είναι πολύ μεγάλο) έτσι ώστε να απαιτούνται λίγα (και μικρά) τμήματά του στη ΚΜ σε κάθε στιγμή.
- Παράδειγμα: 32-bit address



Τα 10 bits του PT1 ορίζουν ένα PT με 1024 εγγραφές. Η κάθε μια απ' αυτές τις εγγραφές δείχνουν σ' ένα άλλο Page Table (στο 2ο επίπεδο) στη Κ.Μ. Ετσι η τιμή του PT1 χρησιμοποιείται ως index στο high-level PT η εγγραφή του οποίου οδηγεί σ' ένα PT στο 2ο επίπεδο.

Η τιμή του PT2 χρησιμοποιείται ως index στο PT που βρέθηκε στο προηγούμενο βήμα. Η εγγραφή στο 2ο PT δείχνει στα frame της ΚΜ για το ζητούμενο page. Το 12-bit offset χρησιμοποιείται για να βρεθεί το κατάλληλο byte στο frame του προηγούμενου βήματος.

Στο παράδειγμα:

- Κάθε page είναι 4KB (αφού offset είναι 12 bits)
- Κάθε PT στο 2ο επίπεδο κάνει map 4MB
- Το PT στο 1ο επίπεδο κάνει map 4GB (32 bits -> 4 GB).

Ετσι αν τα text, data, και stack τμήματα του process είναι το καθένα <=4MB τότε χρειαζόμαστε να έχουμε στη κύρια μνήμη μόνο 3 PTs του 2ου επιπέδου. Αφού το text τμήμα δεν μεγαλώνει, τότε η ανάθεση virtual διευθύνσεων στο process μπορεί να γίνει ως εξής:

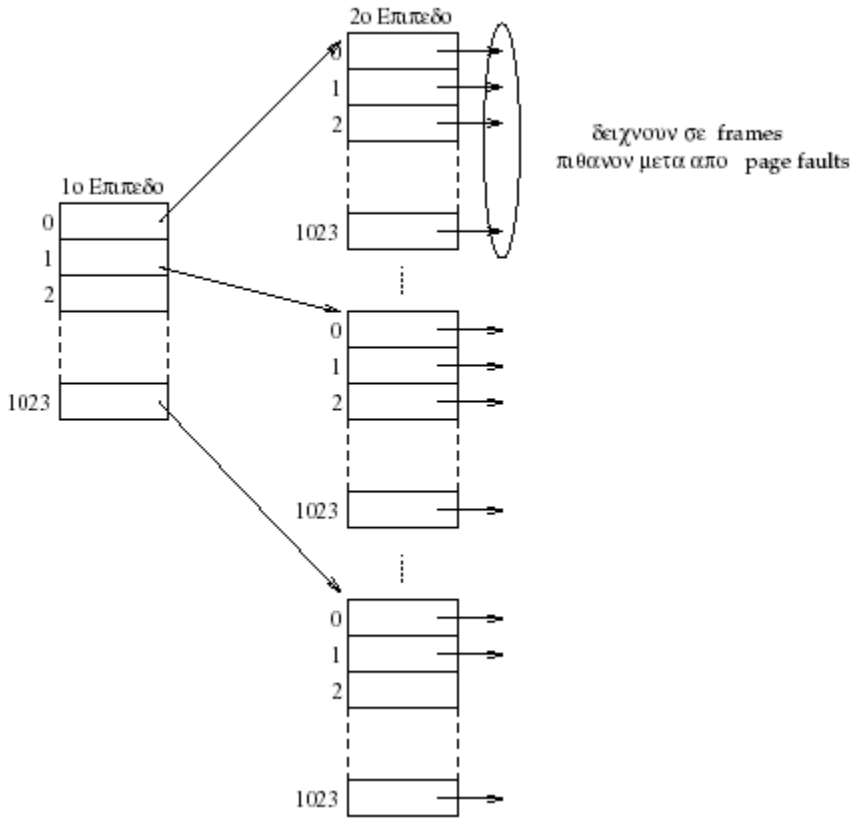
- 0-4MB ➔ text
- 4MB - 8MB ➔ data
- τα τελευταία 4MB ➔ stack (μεγαλώνει "προς τα κάτω"))

➔ θα χρειαστούν περισσότερα PTs στη ΚΜ μόνο αν το data ή το stack τμήμα μεγαλώσει πάνω από 4MB το καθένα.

Η Δομη των Εγγραφών των Page Tables

Συνήθως η ακριβής δομή είναι machine-dependent. Αλλά, τα πιο πολλά PT entries (εγγραφές) έχουν τις εξής πληροφορίες:

Protection bits: ορίζουν αν η σελίδα αυτή μπορεί να γίνει read/write/execute



Modify bit: δηλώνει αν έχει ενημερωθεί η σελίδα από τότε που έγινε page in.
Reference bits: δηλώνουν πόσες φορές το τελευταίο χρονικό διάστημα έγινε αναφορά σ' αυτή τη σελίδα.
Present bit: δηλώνει αν η σελίδα βρίσκεται στη ΚΜ.
Page Frame bits: ορίζουν σε ποιό frame της ΚΜ βρίσκεται αυτή η σελίδα.

Προσέξτε ότι η διεύθυνση της σελίδας στο swap space (δίσκο) δεν βρίσκεται στο PT entry.

Modify bit: χρησιμοποιείται όταν αποφασίζετε να εκδιωχθεί μια σελίδα απ' την ΚΜ για να ελευθερωθεί χώρος για μια άλλη σελίδα. Αν το bit αυτό είναι 1 τότε η σελίδα γράφεται στο swap space - αλλοιώς δεν χρειάζεται αυτό το disk write.

Reference bit: Χρησιμοποιείται για ν' αποφασισθεί ποιά σελίδα ("θύμα") θ' αντικατασταθεί.

Κρυφή Μνήμη γιά τους Πίνακες (PAGE TABLE CACHES)

Αφού τα PTs είναι πολύ μεγάλα για να χωρέσουν σε CPU registers, πώς μπορούμε να μειώσουμε το κόστος πρόσβασης σ' αυτά (στη ΚΜ) ;
Η απάντηση βασίζεται στη χρήση **associative memory** που λειτουργεί σαν **cache**. Η θεμελιώδης παρατήρηση είναι ότι η συμπεριφορά των περισσότερων προγραμμάτων είναι τέτοια ώστε να παρατηρούνται μεγάλοι αριθμοί αναφορών σε λίγες σελίδες (που αλλάζουν αργά με το πέρασμα του χρόνου) π.χ. - όλες οι εντολές σε μια σελίδα εντολών (βλέπε loops).

Ετσι χρησιμοποιείται ειδικό hardware που αποτελείται από registers το σύνολο των οποίων καλούνται **translation lookaside buffer.** Ο αριθμός αυτών των registers είναι μικρός (συνήθως <= 64).
Ο κάθε register έχει την ίδια πληροφορία που υπάρχει σ' ένα PT entry και επιπλέον έχει και το virtual page number.
Το MMU εξετάζει πρώτα αν το ζητούμενο virtual page number βρίσκεται στην associative memory (όλοι οι registers εξετάζονται παράλληλα).

- Αν ναί, τότε αποφεύγεται η πρόσβαση του PT στη ΚΜ.
- Αν όχι, τότε προσπελαύνεται το PT και το κατάλληλο PT entry αντικαθιστά κάποιο άλλο στην associative μνήμη.

Προσέξτε ότι αν γίνει process/context switch, τότε θα πρέπει να "μηδενισθεί" η associative memory!

3.3.3 Αντικατάσταση Σελίδων (Page Replacement)

Οταν σημειώνεται ένα **page fault** τότε το Λ.Σ. πρέπει (μερικές φορές - αν δεν υπάρχει ελεύθερο frame) να διαλέξει μια σελίδα **(victim)** "θύμα" που θα αντικατασταθεί από την ζητούμενη σελίδα.

Αν το "θύμα" είναι **dirty** (δηλ. έχει ενημερωθεί όσο ήταν στη Κ.Μ.) τότε το Λ.Σ. πρέπει να την ξαναγράψει πάλι στο swap space. Μια σελίδα είναι dirty αν το PT entry που δείχνει σ' αυτήν έχει το M bit = 1.

Προφανώς η απόδοση του συστήματος είναι καλύτερη αν το "θύμα" δεν είναι μια πολυ-χρησιμοποιημένη σελίδα. ➔ Το μεγάλο πρόβλημα κόστους πηγάζει από την ανάγκη πρόσβασης στο δίσκο για αντικατάσταση σελίδων.

Η βέλτιστη στρατηγική: Αν ξέραμε μετά από πόσο χρόνο (π.χ. αριθμό εντολών) η κάθε σελίδα στη ΚΜ θα επαναπροσπελασθεί, τότε μπορούμε να διαλέξουμε σαν θύμα την σελίδα που δεν θα χρειαστεί για το μεγαλύτερο χρονικό διάστημα. Ομως, σε πραγματικά συστήματα, τέτοιου είδους πληροφορία δεν υπάρχει, ούτε μπορεί να παραχθεί.

[Σε μερικές περιπτώσεις είναι δυνατόν να υποβληθεί ένα πρόγραμμα σε κάποιο είδους tracer το οποίο καταχωρεί την συμπεριφορά του προγράμματος αναφορικά με τις προσπελάσεις σε σελίδες. Αυτή η πληροφορία χρησιμοποιείται όταν το πρόγραμμα τρέχει πραγματικά και έτσι μπορούμε να υλοποιήσουμε τη βέλτιστη στρατηγική...]

Ο Αλγόριθμος Not Recently Used

Συνοπτικά, αυτή η μέθοδος διαλέγει σαν θύμα μια σελίδα που δεν έχει χρησιμοποιηθεί πρόσφατα. Πρέπει λοιπόν να υπάρχουν πληροφορίες σχετικά με το πόσο πρόσφατα χρησιμοποιήθηκε η κάθε σελίδα. Γι' αυτό το σκοπό χρησιμοποιούνται τα **R και M bits** που συνήθως υπάρχουν στις εγγραφές των Page Tables. (Το R γίνεται ένα όταν η αντίστοιχη σέλίδα προσπελαύνεται. Το M γίνεται 1 όταν η αντίστοιχη σελίδα ενημερώνεται.) Τις ενημερώσεις των R, M bits τις κάνει το hardware.

Αλγόριθμος

1. Οταν αρχίζει ένα process τότε όλα τα R, M bits των εγγραφών του PT παίρνουν την τιμή 0.
2. Σε κάθε **clock interrupt** το R bit γίνεται πάλι 0. Ετσι αν εξετάσουμε μια PT εγγραφή και δούμε το R = 1 ➔ ότι υπήρξε αναφορά σ' αυτή τη σελίδα μετά το τελευταίο clock tick.
3. Οταν σημειώνεται ένα **page fault:** Το Λ.Σ. δημιουργεί 4 κατηγορίες σελίδων με βάση τις τιμές των R & M bits.

K1 R = 0 M = 0

K2 R = 0 M = 1 **(υπάρχει αυτή η κατηγορία; ΝΑΙ)**

K3 R = 1 M = 0

K4 R = 1 M = 1

Το Λ.Σ. διαλέγει σαν θύμα μια από τις σελίδες στην κατώτερη, μη-άδεια, κατηγορία.
Προσέξτε ότι προτιμάει θύματα από K1 και όχι από K2, λόγω του κόστους της αντικατάστασης μιας σελίδας στη κατηγορία K2.

Ο NRU είναι απλός, ευκολο-υλοποιήσιμος και έχει καλή απόδοση.

<u>Αλγόριθμοι βασισμένοι σε FIFO</u>

Προφανώς η "καθαρόαιμη" FIFO στρατηγική έχει προβλήματα. Γιατί ;

Ο Αλγόριθμος Δεύτερης Ευκαιρίας (Second - Chance)

- Βασίζεται στη στρατηγική FIFO

- Αλλά, αντί να αντικαθιστά την πιο παλιά σελίδα, εξετάζει το R bit. Αν R=0 τότε την αντικαθιστά.
- Αν R = 1 τότε
    - R = 0
    - η σελίδα τοποθετείται στο τέλος της λίστας (δηλ. "βαφτίζεται" σαν η πιο πρόσφατη σελίδα στη ΚΜ)
    - η αναζήτηση θύματος συνεχίζεται στην επόμενη πιο παλιά σελίδα.
- Αν R=1 για όλες τις σελίδες, τότε second-chance είναι στην ουσία FIFO (Γιατί ;)

Ο Αλγόριθμος Ρολόι (Clock)

- Το σημαντικότερο μειονέκτημα του second-chance είναι ότι συνεχώς ενημερώνει την λίστα - χρονοβόρα διαδικασία.
- Στον αλγόριθμο clock, οι σελίδες οργανώνονται σ' ένα circular list που προσομειώνει ένα ρολόϊ, δηλαδή υπάρχει ένας δείκτης που δείχνει στην πιο παλιά σελίδα.
- Οταν σημειώνεται ένα page fault:
    - εξετάζεται η σελίδα του δείκτη:
    - Αν R=0, τότε τότε αυτή είναι το θύμα & ο δείκτης δείχνει στην επόμενη σελίδα.
    - Αν R=1, τότε R = 0 & ο δείκτης δείχνει στην επόμενη σελίδα και η αναζήτηση συνεχίζεται.

Ο Αλγόριθμος Least-Recently Used (LRU)

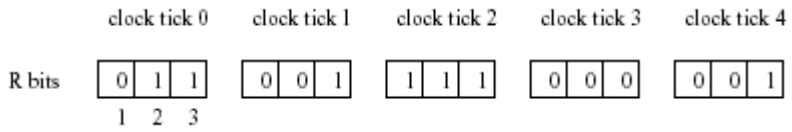
Βασική ιδέα: σελίδες που προσπελάστηκαν στο πρόσφατο παρελθόν θα επανα-προσπελασθούν και στο εγγύς μέλλον (και αντιστρόφως). Ετσι, το θύμα είναι η σελίδα που δεν έχει χρησιμοποιηθεί για το μεγαλύτερο χρονικό διάστημα.

- Μια προσέγγιση υλοποίησης είναι με μια λίστα όλων των σελίδων στη ΚΜ με την πιο-πρόσφατα-χρησιμοποιημένη σελίδα στην αρχή της λίστας και την LRU σελίδα στο τέλος.
- Απλή μέθοδος, αλλά η λίστα χρειάζεται ενημέρωση σε κάθε αναφορά στη μνήμη (που γίνεται > 1 φορές για κάθε εντολή) ➔ το κόστος είναι απαγορευτικό!
- Μια προσέγγιση αποφυγής του παραπάνω κόστους είναι να χρησιμοποιηθεί **ειδικό hardware**. Μια μέθοδος χρησιμοποιεί έναν 64-bit counter η τιμή του οποίου αυξάνεται σε κάθε εντολή. Επίσης, κάθε εγγραφή του PT έχει 64 bits παραπάνω, έτσι ώστε μετά από κάθε αναφορά σε μια σελίδα της ΚΜ η τιμή του counter να αποθηκεύεται στην αντίστοιχη εγγραφή του PT. **Ετσι, η μέθοδος διαλέγει σαν θύμα την σελίδα της οποίας η εγγραφή στο PT έχει την μικρότερη τιμή του counter.**
- Το παραπάνω ειδικό h/w κοστίζει, φυσικά, και μπορεί να μην υπάρχει/προσφέρεται από κάποιο σύστημα. Ετσι πολλοί σχεδιαστές Λ.Σ. καταφεύγουν σε λύσεις software με τις οποίες προσπαθούν να προσομοιώσουν την στρατηγική LRU.

Μια προσομοίωση του LRU είναι **Not-Frequently-Used (NFU)**.

- Με κάθε σελίδα συνδέεται και ένας software counter (αρχικά έχει την τιμή 0).
- Με κάθε clock interrupt το Λ.Σ. προσπελαύνει όλες τις σελίδες στην Κ.Μ. Για κάθε σελίδα:
    - ο counter γίνεται **shift right**
    - η τιμή του counter τότε προσαυξάνεται με την τιμή του R (0 ή 1) της ίδιας σελίδας.
    - Το R προστίθεται στο πιο σημαντικό bit του counter.
- Οταν σημειώνεται ένα page fault, η σελίδα με τη μικρότερη τιμή του counter επιλέγεται σαν θύμα.
- Ο παραπάνω αλγόριθμος είναι μια συγκεκριμένη μορφή NFU που ονομάζεται **aging.**

π.χ.



<u>Counters</u>	000000	000000	100000	010000	001000
	100000	010000	101000	010100	001010
	100000	110000	111000	011100	101110

Αν μια σελίδα δεν έχει χρησιμοποιηθεί για N ticks ➔ τα πρώτα N bits = 0. Το θύμα θα είναι η σελίδα 1.

Αυτός ο αλγόριθμος διαφέρει κατά 2 τρόπους από τον LRU:

1. Αν και η σελίδα 1 είναι το θύμα, είναι πιθανό να είχε χρησιμοποιηθεί μετά την σελίδα 2 στο clock tick 2.
2. Υπάρχει ένας πεπερασμένος αριθμός (για τον counter) bits (π.χ. αν όλα τα 6 bits 2 σελίδων είναι 0 τότε διαλέγουμε στην τύχη μεταξύ τους). Αυτό όμως δεν δημιουργεί ιδιαίτερα προβλήματα!

3.4 Σχεδιασμός Συστημάτων Σελιδοποίησης

Οταν αρχίζει ένα process το Λ.Σ., μπορεί να επιλέξει:

- να μην φέρει καμμιά σελίδα του στη Κ.Μ. (δηλαδή να αφήσει το Paging σύστημα να φέρει την κάθε σελίδα απ' τον δίσκο όταν υπάρξει η πρώτη αναφορά σ' αυτήν). Αυτή η στρατηγική ονομάζεται **demand paging.**
- να φορτώσει κάποιες σελίδες του process απ' τον δίσκο πριν το process αρχίσει να τρέχει. Αυτή η στρατηγική ονομάζεται **prepaging.**

Οι έννοιες Working Set και Locality of Reference.

Συνήθως η συμπεριφορά των προγραμμάτων που τρέχουν χαρακτηρίζεται από διάφορες φάσεις. Σε κάθε φάση οι αναφορές μνήμης που γίνονται απ' το πρόγραμμα εντοπίζονται σ' ένα μικρό σύνολο σελίδων του (locality of reference) που είναι ένα πολύ μικρό υποσύνολο του address space του προγράμματος.
Αυτό το σύνολο των σελίδων που χρησιμοποιεί ένα πρόγραμμα στη τωρινή φάση του ονομάζεται working set.
Αν όλο το working set ενός process βρίσκεται στη ΚΜ τότε το process αυτό θα δημιουργεί page faults μόνο κατά τη μετάβαση στην επόμενη φάση. Αν το working set είναι πολύ μεγάλο και δεν χωράει στη διαθέσιμη "ελεύθερη" μνήμη τότε το πρόγραμμα θα τρέξει πολύ αργά.

Το φαινόμενο thrashing: Οταν ένα process σπαταλεί ένα μεγάλο μέρος του συνολικού χρόνου σε page faults (π.χ. ένα process έτρεξε για 15 secs, τα 14 από τα οποία ήταν για page faults). Προσέξτε:

- κάθε εντολή παίρνει < 1 μς
- κάθε πρόσβαση στο δίσκο παίρνει > 10 ms

Το Μοντέλο Working Set (WS)

Λ.Σ. που εφαρμόζουν αυτό το μοντέλο πρέπει να:

- ξέρουν ποιό έιναι το working set κάθε process, και
- πριν τρέξουν το process (είτε στην αρχή, είτε μετά από swap in) να έχουν φέρει το working set στη Κ.Μ., (δηλ. εφαρμόζουν **prepaging**).

Πώς μπορεί το Λ.Σ. να ξέρει ποιό είναι το WS; Μία μέθοδος βασίζεται στον αλγόριθμο aging που είδαμε παραπάνω. Το WS ενός process αποτελείται απ' όλες τις σελίδες των οποίων τα πρώτα N bits των counter δεν είναι 0. Δηλ. αν στα τελευταία N clock ticks έχει σημειωθεί αναφορά στη σελίδα, τότε η σελίδα ανήκει στο WS του process.

Το μοντέλο WS μπορεί να συνδυαστεί με τον αλγόριθμο **clock**: Αντί να επιλέγεται ως θύμα η σελίδα στην οποία δείχνει ο δείκτης αν το R = 0:

πρώτα εξετάζεται αν η σελίδα αυτή ανήκει στο WS.

- Αν ναι, τότε δεν γίνεται θύμα, και συνεχίζεται η αναζήτηση.
- Αλλιώς, επιλέγεται ως θύμα.

Αυτός ο αλγόριθμος ονομάζεται WS_Clock

Τοπική και Καθολική Αντικατάσταση (Global και Local Page Replacement)

Θεμελιώδης ερώτηση: Οταν σημειώνεται ένα page fault, και αναζητούμε ένα θύμα, θα περιορίσουμε την έρευνα μας μόνο στις σελίδες του process που προκάλεσε to page fault, ή θα εξετάσουμε όλες τις σελίδες της Κ.Μ. ανεξάρτητα από το σε ποιό process ανήκουν; Στην πρώτη περίπτωση η στρατηγική ονομάζεται **local** ενώ στη δεύτερη περίπτωση ονομάζεται **global.** Με global αλγόριθμους, ο αριθμός των frames που έχουν δοθεί σε κάθε process διαφέρει.

- Συνήθως προτιμούνται οι global αλγόριθμοι διότι συνήθως το μέγεθος του WS αλλάζει με την πάροδο του χρόνου. Με local αλγόριθμους: Αν το WS μικρύνει τότε σπαταλάται (στην ουσία) χώρος της Κ.Μ. έλλειψη του οποίου δημιουργεί page faults σ' άλλα processes. Αντίθετα, αν το WS μεγαλώσει, μπορεί να συμβεί **thrashing.**
- Οι global αλγόριθμοι δεν έχουν αυτά τα προβλήματα. Ομως πρέπει ν' αποφασίσουν πόσα frames να δώσουν σε κάθε process. Συνήθως, ανάλογα με το μέγεθος του process, αναθέτεται και ένας αριθμός frames. Αλλά και αυτό μπορεί να δημιουργήσει **thrashing.** Το πρόβλημα μπορεί να λυθεί μετρώντας τη **page fault** συχνότητα (frequency) κάθε process και χρησιμοποιώντας 2 κατώφλια: **High PFF** και **Low PFF.** Οταν μετριέται το PFF ενός process p:
    - Αν PFFp > HPFF τότε το Λ.Σ. δίνει πιο πολλά frames
    - Αν PFFp < LPFF τότε το Λ.Σ. μπορεί να πάρει frames από το process p.

Μέγεθος Σελίδας

- Κατά μέσο όρο, η τελευταία σελίδα κάθε segment (text, data, ή stack) είναι άδεια ➔ **μικρές σελίδες είναι καλύτερα.**
- Επίσης όταν ένα page είναι μεγάλο τότε μπορεί το process να χρειάζεται μόνο ένα υποσύνολο της πληροφορίας στο page και έτσι σπαταλείται Κ.Μ. ➔ μικρές σελίδες είναι καλύτερα.
- Από την άλλη μεριά:
    - μικρές σελίδες ➔ μεγάλα PTs ➔ σπαταλείται Κ.Μ.
    - μικρές σελίδες ➔ μη αποδοτικό I/O κατά τη διάρκεια page faults. Αυτό συμβαίνει γιατί σελίδες όταν γίνονται swap in/out ο μεγαλύτερος χρόνος που απαιτείται οφείλεται στη χρησιμοποίηση του δίσκου για **seek** (δηλ. να τοποθετηθεί η κεφαλή του δίσκου στη σωστή σελίδα) ➔ συμφέρει να μεταφέρουμε μεγάλες σελίδες !!!

3.5 Ζητήματα Υλοποίησης

Πισωγύρισμα Εντολής (Instruction Back-Up)

- Θεωρείστε την εντολή που προκαλεί page fault. Το page fault μπορεί να προκλήθηκε όταν έγινε αναφορά στη σελίδα της εντολής, ή σε μια από τις σελίδες που περιέχουν τις παραμέτρους. Μόλις το Λ.Σ. φέρει μέσα την ζητούμενη σελίδα πρέπει να επαναληφθεί η εντολή από την αρχή ➔ το Λ.Σ. πρέπει να βρεί που

είναι το 1ο byte της εντολής. Συνήθως οι εντολές είναι πολλά bytes (π.χ. 2 bytes για opcode και 2 για κάθε παράμετρο) ➔ δεν είναι εύκολη υπόθεση.

- Μερικά συστήματα είτε κρατούν αυτή τη πληροφορία σε ειδικούς registers, είτε το microcode της εντολής την κάνει push στο stack, όπου το Λ.Σ. την βρίσκει!

Κλείδωμα Σελίδων (Page Locking)

Υπάρχει αλληλεπίδραση μεταξύ I/O και virtual Memory συστημάτων. Θεωρείστε ένα process, P, που εκτελεί read (fd, &buf, ...). Το Λ.Σ. δίνει την εντολή στον disk controller να φέρει το ζητούμενο disk block στην σελίδα (ες) που βρίκεται ο buf. Το P μπλοκάρει και το CPU δίνεται στο P1. Το P1 προκαλεί page faults και σαν θύμα το paging system επιλέγει μια από τις σελίδες όπου βρίσκεται ο buf της P και την φορτώνει με την επιθυμητή σελίδα της P1. Αργότερα ο disk controller φέρνει την απάντηση από τον δίσκο για την P και χρησιμοποιώντας DMA την αποθηκεύει στη διεύθυνση όπου τώρα υπάρχει η σελίδα της P1 ... καταστροφή.

Η λύση στο πρόβλημα είναι να χρησιμοποιηθεί ένα **lock bit** για κάθε frame. Οταν το lock bit είναι 1 τότε αυτή η σελίδα δεν μπορεί να επιλεχθεί σα θύμα από τον pager. Ετσι, αν όλες οι σελίδες που εμπλέκονται σε I/O έχουν το lock bit = 1 τότε το παραπάνω πρόβλημα αποφεύγεται.

Διαμοιρασμός Σελιδων (Page Sharing)

Συνήθως >1 process τρέχουν το ίδιο πρόγραμμα (π.χ. vi) ➔ συμφέρει οι σελίδες να διαμοιράζονται, αντί να υπάρχει ένα αντίγραφο του κοινού προγράμματος για κάθε process. ➔ όταν ένα process γίνεται swap out (ή τελειώνει) πρέπει οι σελίδες (text) που χρησιμοποιούνται από άλλα processes να μην γίνουν swap out (και το swap space να μην ελευθερωθεί)

Swap Space

- Συνήθως είναι ένα ειδικό partition στο δίσκο.
- Υπάρχει ένα **free list** έτσι ώστε καινούργια processes να βρίσκουν swap space.
- Οταν ένα process αρχίζει, το Λ.Σ. του αναθέτει τόσο swap space όσο είναι και το μέγεθός του.
- Στην εγγραφή του Process Table του Λ.Σ. για το process P υπάρχει ένα πεδίο που δείχνει στην διεύθυνση στο δίσκο για το process P (swap space). Ετσι αν προσθέσουμε το virtual page number σ' αυτή την διεύθυνση βρίσκουμε που είναι η κατάλληλη σελίδα στο swap space.
- Συνήθως τα segments όπως το stack και το data ενός process έχουν διαφορετικά swap spaces, λόγω κυμαινόμενου μεγέθους.

Οι Δαίμονες PAGER

- Σε πολλά συστήματα paging deamons (ειδικά processes) διασφαλίζουν ότι υπάρχουν "αρκετά" ελεύθερα page frames.

- Περιοδικά, ο paging deamon ξυπνάει, ελέγχει αν υπάρχουν αρκετά ελεύθερα frames.
  Αν ναι, ξανακοιμάται. Αν όχι, τρέχει τον αλγόριθμο page replacement μέχρι ώτου αρκετά frames γίνουν ελεύθερα.
- Ετσι το σύστημα έχει καλύτερη απόδοση. Γιατί ;

Χειρισμός Λάθους (Page Fault Handling)

<u>Αλγόριθμος</u>

1. Εχουμε kernel trap. Ο kernel σώζει PC και διεύθυνση εντολής (1ο byte)
2. kernel σώζει general-purpose regs.
3. Βρίσκει ποιά virtual page προκάλεσε page fault (μπορεί να χρειαστεί να κάνει parse την εντολή).
4. Αν οι έλεγχοι base & limit registers και των protection bits περάσουν, το Λ.Σ. προσπαθεί να βρεί ελεύθερο frame (τρέχοντας ίσως τον page replacement αλγόριθμο).
5. Αν το θύμα έχει M=1, τότε ζητάει απ' τον disk controller το γράψιμο του page και δρομολογεί άλλο process (αφού κλειδώσει την σελίδα).
6. Αφού γραφτεί η σελίδα, ο kernel βρίσκει που στο swap space είναι η ζητούμενη σελίδα και ζητά απ' τον disk controller να την φέρει.
7. Οταν έρθει η σελίδα τα PTs ενημερώνονται και η σελίδα ξεκλειδώνεται.
8. Ο kernel βρίκει το 1ο byte της εντολ;ης που προκάλεσε το page fault.
9. Το process που προκάλεσε το page fault ξανατρέχει (ο έλεγχος επιστρέφει στην assembly-code ρουτίνα του βήματος 2).
10. Η assembly ρουτίνα επανατοποθετεί τις κατάλληλες τιμές των general-purpose CPU regs και επιστρέφει. Το process τώρα τρέχει σαν το page fault να μην είχε συμβεί.

Κεφάλαιο 4. Συστήματα Διαχείρισης Αρχείων

4.1 Περίληψη

Αρχείο (File): Ενα abstraction που αφορά ένα σύνολο λογικά συσχετιζόμενων δεδομένων, αποθηκευμένο σε μαγνητικούς δίσκους.

Συστήματα Αρχείων (File Systems): Μέρος ενός Λ.Σ. Ασχολείται με την οργάνωση, αποθήκευση, προσπέλαση, προστασία, διαμοιρασμό και ονομασία (naming) των αρχείων. Προσφέρει το **file abstraction**: κρύβει όλες τις λεπτομέρειες διαχείρισης και προσπέλασης μαγνητικών δίσκων, οργάνωσης δεδομένων ενός αρχείου, κλπ.

Κατάλογος (Directory): Επιτρέπει ονόματα υψηλού επιπέδου (π.χ. ASCII strings) ν' ανατεθούν σ' αυτό το storage abstraction π.χ. " peter/foo" -> σε κάποιο εσωτερικό file id (συνήθως ένας μεγάλος αριθμός που ορίζει το αρχείο peter/foo στο σύστημα).

Ονομασία (File naming): Η διαδικασία μετάφρασης ενός υψηλού-επιπέδου (user-name) ονόματος ενός αρχείου στο αντίστοιχο file ID.

Έλεγχος πρόσβασης (Access Control): περιορίζει πρόσβαση σε αρχεία. Μόνο εξουσιοδοτημένοι χρήστες μπορούν να προσπελάσουν συγκεκριμένα αρχεία.

<u>Οργάνωση ενός File System</u>

Υπηρεσία Καταλόγου (Directory Service):

- Directory Module: Υψηλού επιπέδου ονόματα -> File Ids
- Access Control Module: έλεγχος άδειας πρόσβασης

Υπηρεσία Αρχείων (File Service):

- File Module: File IDs -> "files"
- File Access Module: Προσπέλαση αρχείου

Υπηρεσία Αποθήκευσης (Block Service):

- Block Module: Διαχείριση χώρου στο δίσκο
- Device Module: disk I/O & buffering

File Service: - Υλοποιεί έναν «επίπεδο χώρο ονομάτων» (**flat file namespace)**. Πρέπει να μπορεί να προσδιορίζει την ταυτότητα του ζητούμενου αρχείου (μονοσήμαντα) ➔

unique file ids (UFIDs). Δεδομένου του UFID η πληροφορία του αντίστοιχου αρχείου μπορεί να εντοπιστεί.

Files = Data + attributes. **Attributes = Metadata** (δηλ. πληροφορία για την πληροφορία). Συνήθως τα attributes ενός αρχείου περιγράφουν το μέγεθος, ιδιοκτήτη, λίστα access control, κ.λπ. Μερικά attributes είναι προσπελάσιμα από τους χρήστες.

Block Service: Διαχειρίζεται disk blocks: Αναθέτει/ελευθερώνει blocks σε/από files και προσπελαύνει/ενημερώνει blocks.

Directory Service: Διαχειρίζεται directories (που συνήθως υλοποιούνται σαν ειδικά αρχεία). ➔ ο **directory server** είναι πελάτης του file server.Αναθέτει UFIDs, όταν δημιουργούνται αρχεία, που επιστρέφονται στους χρήστες. Επειτα, οι χρήστες αναφέρονται στα αρχεία χρησιμοποιώντας τα UFIDs.
UFIDs μπορεί να είναι **capabilities**: δηλ. χρησιμοποιούνται από τους κατέχοντας σαν απόδειξη προς τον File Server δικαιώματος προσπέλασης αρχείων.
➔ ο directory server πριν δώσει UFIDs σε χρήστες πρέπει πρώτα να ελέγξει ότι οι χρήστες διαθέτουν τα ανάλογα δικαιώματα πρόσβασης.

Ο διαχωρισμός του Directory Service από το File Service μπορεί να δημιουργήσει προβλήματα! π.χ. όταν διαγράφουμε ένα αρχείο, πρέπει να: 1) διαγραφεί directory info και 2) διαγραφούν file blocks Αν το πρόγραμμα, λόγω βλάβης, δεν εκτελέσει το βήμα (2) τότε έχουμε ένα μη-προσπελάσιμο αρχείο το οποίο δεν μπορούμε να διαγράψουμε.

4.2 Υπηρεσία Καταλόγου (Directory Service)

ACCESS CONTROL: Ο dir. Server αποθηκεύει ζεύγη (file name, UFID). Πριν επιστρέψει ένα UFID, πρέπει να ελέγξει αν ο χρήστης έχει δικαίωμα πρόσβασης ➔ access control.

User Ids: προσδιορίζουν την ταυτότητα χρηστών που προσπελαύνουν αρχεία. Κάθε χρήστης σχετίζεται με ένα user id μετά από μια διαδικασία authentication (εξακρίβωση στοιχείων) (π.χ. passwords). Μπορεί να υπάρχει ξεχωριστός authentication server.

Access Controll Lists (ACLs) π.χ.

Operation	Users
Read	Λίστα με χρήστες
Write	Λίστα με χρήστες

Επίσης ομάδες χρηστών (με ομαδικά δικαιώματα πρόσβασης) μπορούν να δημιουργηθούν

Σύνοψη: Δεδομένου ενός user id και ενός file name ο dir. server προσπελαύνει το ACL του file και, αν όλοι οι έλεγχοι εκτελεστούν ομαλά, επιστρέφει ένα UFID στον χρήστη.

Σε όλες τις μετέπειτα αιτήσεις του ο χρήστης προσκομίζει μόνο το UFID για ν' αποδείξει το δικαίωμα πρόσβασης του στο αρχείο.

Βασική Υπηρεσία Καταλόγων:

- Ονομασία και έλεγχος πρόσβασης (access control) Διαχειρίζεται directories (ειδικά αρχεία) Επιστρέφει ένα UFID με καθορισμένα δικαιώματα προσπέλασης ή λάθος! Τα directories επίσης ονομάζονται με τα UFIDs.

Λειτουργίες:

- **LookUp(DirUFID, Name, Access Mode, Userld, &UFID)** - Σε επιτυχία ελέγχου πρόσβασης επιστρέφει το UFID του named file
- **AddName(DirUFID, Name, UFID, UserId)**
  Προσθέτει το ζεύγος (Name, UFID) στο directory
- **UnName (DirUFID, Name)**
  Το ζεύγος που περιέχει το "Name" διαγράφεται από το DirUFID
- **ReName (DirUFID, Old Name, New Name)**
  Αλλάζει το ζεύγος (OldName, UFID) -> (NewName, UFID)
- **GetName (DirUFID, pattern)**
  Επιστρέφει όλα τ' ονόματα το DirUFID που ταιριάζουν στο "pattern".

Το Directory service πρέπει επίσης να μπορεί να αλλάζει attributes ενός file - π.χ. το ACL για να υποστηρίζει λειτουργίες chmod. Επίσης πρέπει να παρέχει λειτουργίες για να εξετάζονται τα attributes.

Το directory service είναι ο ιδιοκτήτης όλων των directories. Για κοινόχρηστα (shared) directories πρέπει να δηλώσουμε ένα χρήστη σαν owner και μια στρατηγική access-permission παρόμοια με αυτή για τα κοινόχρηστα files.

Χαμενα Αρχεια: Για λειτουργία create-file: καλούμε το create system call (λειτουργία File Service) και κατόπιν AddName (λειτουργία Directory Service).

4.3 Υπηρεσία Αρχείων (File Service)

Λειτουργίες (στα δεδομένα)

- **Create (&UFID)**: Δημιουργεί ένα νέο (άδειο) file και επιστρέφει ένα UFID
- **Delete (UFID)** : Διαγράφει το file
- **Length (UFID)** : Επιστρέφει το μήκος του file
- **Read (UFID, position, number, &buffer)**: Διαβάζει "number" bytes αρχίζοντας από τη θέση "position" του αρχείου UFID και τα τοποθετεί στο "buffer".
- **Write (UFID, position, buffer)**: Γράφει τα δεδομένα του "buffer" στο file UFID στη θέση "position".

Αυτές οι λειτουργίες, όπως ορίζονται είναι επαναλήψιμες (repeatable). Θυμηθείτε τη συζήτηση (του RPC) και προσέξτε ότι clients του file service μπορούν να προσπελάσουν files χρησιμοποιώντας το RPC! Οι επαναλήψιμες λειτουργίες επιτρέπουν stateless file servers. Το UNIX διατηρεί έναν r/w pointer για κάθε ανοιχτό file ανα process - αυτό είναι state. File servercrash ➔ αρχίζει από την αρχή, ο pointer του server είναι λάθος.

Υπηρέτες με Μνήμη (Stateful servers): Τους χρειάζονται πολλές εφαρμογές
Παραδείγματα:
1) Κλειδώματα για συγχρονισμό σε ταυτόχρονη πρόσβαση.
2) Ο server χρειάζεται να ξέρει ποιός έχει αντίγραφο (στην cache) ποιών file blocks.

Λειτουργίες στα atributes:
- SetAttributes (UFID, attributes)
- GetAttributes (UFID, &attributes)

UFIDs: Πρέπει να είναι:

- μοναδικά (μεταξύ όλων των files και όλων των hosts που υλοποιούν το F.S.)
- δύσκολο να "μαντευθεί"

48 bits	32 bits	32 bits
server host id	file number	random number

server host id -> Internet address, (μπορεί να είναι μικρότερο για ένα ethernet) random number -> για να ειναι δύσκολο να μαντευθεί.

Για shared files πολλοί διαφορετικοί τύποι πρόσβασης (για διαφορετικούς clients) είναι δυνατοί ➔Τα UFIDs πρέπει να είναι περισσότερο ευέλικτα. Τα UFIDs μπορούν να περιέχουν ένα permission field (π.χ. r, w, ...) ➔ διαφορετικά UFIDs για διαφορετικούς τύπους πρόσβασης για το ίδιο file. Παράδειγμα: 2 bits (για r-o, w-o, & r+w) για permission field.

➔ Χρειάζεται πολύ προσοχή με αυτό το σχήμα: Το permission field πρέπει να συνδυάζεται με το τυχαίο κομμάτι. Αλλιώς κάποιος εύκολα θ' άλλαζε permissions για πιο αυστηρή πρόσβαση.

➔ Encrypt permission field + random number χρησιμοποιώντας ένα κρυφό κλειδί! (γνωστό στο server).

Υλοποίηση - Απαιτείται δυναμική ανάθεση [(de) allocation] των blocks του δίσκου ώστε να επιτρέπονται δυναμικές αλλαγές στα αρχεία.

Το file γενικά καταλαμβάνει μη συνεχόμενα disk blocks (Γιατί;) Το file service διατηρεί μια δομή δεδομένων, το **file index**. Το file index περιέχει τα attributes του file και μια ακολουθία από pointers στα block που περιέχουν τα δεδομένα του file. Το file index πρέπει να υποστηρίζει sequential + random access ενός file.

Γιατί τα attributes δεν αποθηκεύονται σε blocks μαζί με τα δεδομένα;
(Εχουν διαφορετικές απαιτήσεις ελέγχου πρόσβασης).

Εντοπίζοντας δεδομένα αρχείων
Input: UFID
Output: Τοποθεσία του file index
Το κάνει το **File Location Map!**

Στην πραγματικότητα κάθε request που έρχεται στο file service περιέχει ένα UFID και ένα offset ➔

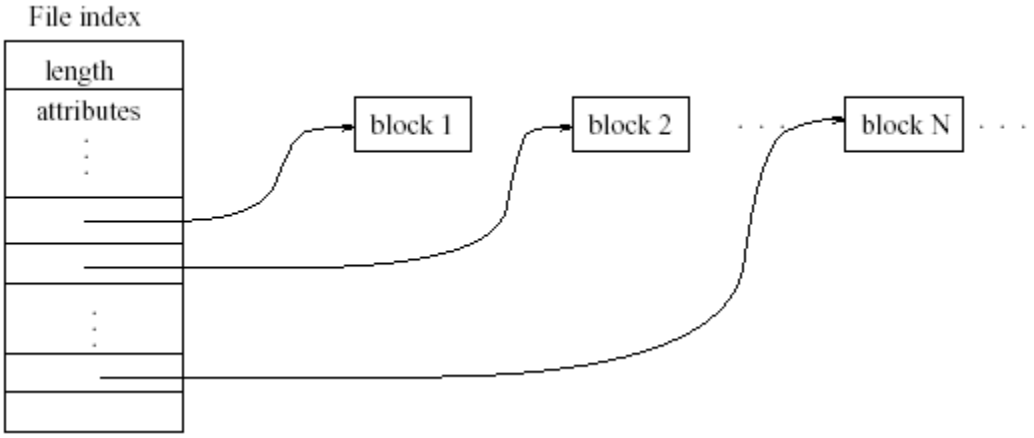
(a) UFID -> block pointer of file index
(b) offset -> block pointer to required page
(a)+(b) : Two step translation

Τα File Location Maps κάνουν το (a). Εξ αιτίας της ύπαρξης τους τα files μπορούν να μεταφερθούν. Το (b) γίνεται χρησιμοποιώντας το file index.

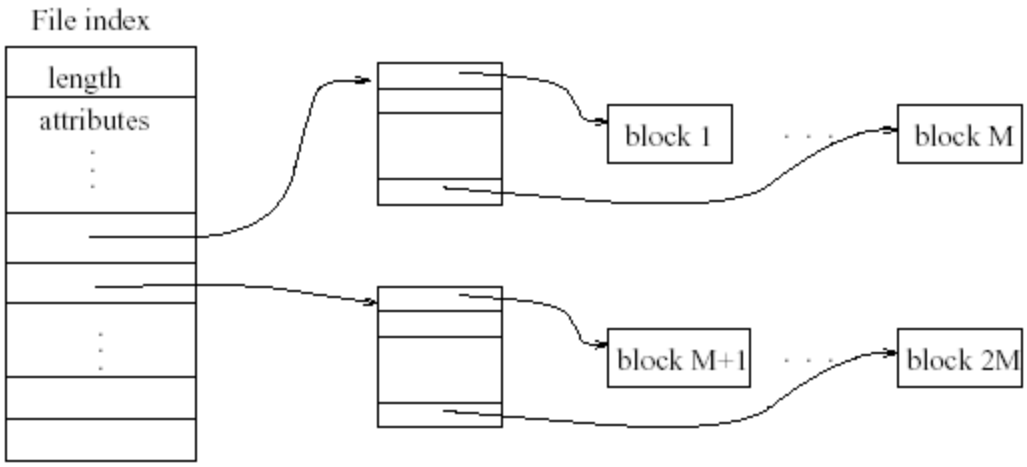
One step translation: UFID + offset -> block ptr για referenced block

Αυτό συνδυάζει το File Location Map και το file index σε μια δομή.

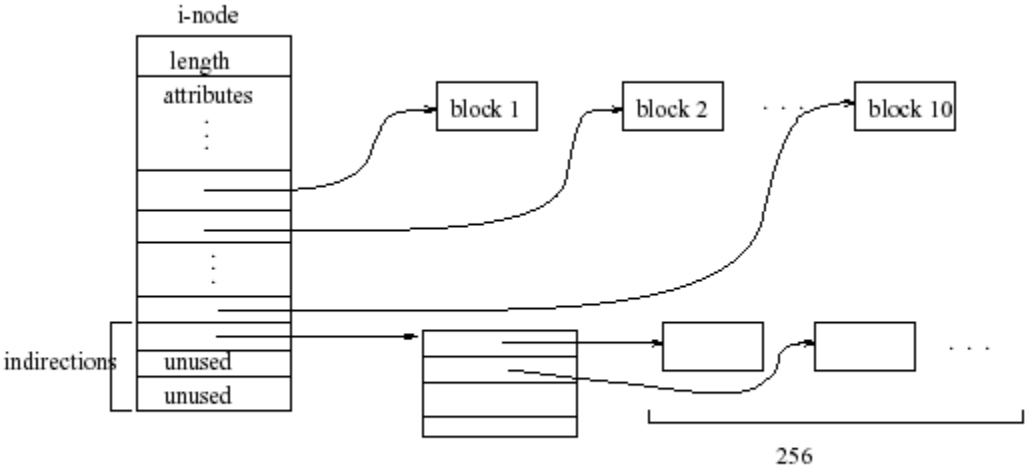
Παράδειγμα File index



Το file index απασχολεί επίσης blocks
Μπορούμε να έχουμε file index πολλαπλών επιπέδων π.χ. file index 2 επιπέδων :



Το UNIX έχει μια δομή που ονομάζεται i-node (file index).



Ο 12ος pointer δειχνει σε δεντρο 2 επιπεδων
Ο 13ος pointer δειχνει σε δεντρο 3 επιπεδων

4.4 Υπηρεσία Αποθήκευσης (Block Service)

- δέσμευση και αποδέσμευση blocks του δίσκου
- μεταφορά δεδομένων από και προς τα blocks του δίσκου
- δείκτες σε blocks: disk_id + διεύθυνση στο δίσκο

Λειτουργίες

- **Allocate Block (&blockptr)** -- δεσμεύει ένα νέο block στο δίσκο και επιστρέφει ένα δείκτη σε αυτό το block
- **Free Block (blockptr)** -- ελευθερώνει το block που δείνεται από το blockptr.

- **GetBlock (blockptr, &buffer) --** διαβάζει τα περιεχόμενα του block που δείχνεται από το δείκτη blockptr και τ' αποθηκεύει στο buffer.
- **Put Block (blockptr, &buffer)** -- γράφει τα περιεχόμενα του buffer στο disk block που δείχνεται από το blockptr

Οι παραπάνω λειτουργίες υλοποιούνται χρησιμοποιώντας περισσότερο στοιχειώδεις λειτουργίες του δίσκου (**primitive disk ops**), οι οποίες παρέχονται από το device module.

Μέγεθος Μονάδας Πληροφορίας (Block Size)

Tradeoff: μειωμένη χρήση αποθηκευτικού χώρου στο δίσκο (μεγάλα blocks) **έναντι** καλύτερης χρήσης αποθηκευτικού χώρου στο δίσκο (μικρότερα blocks: θυμηθείτε την ύπαρξη πολλών μικρότερων αρχείων). (Η μεταφορά των δεδομένων από το δίσκο στην κύρια μνήμη είναι περίπου 1000 φορές πιο γρήγορη από το κόστος αναζήτησης + κόστος περιστροφής => όσο λιγότερα blocks αναζητούμε τόσο το καλύτερο.

Κάποια πιο σύγχρονα συστήματα χρησιμοποιούν πολλαπλά block sizes (UNIX 4.2. BSD): Ενα block χωρίζεται σε τμήματα (fragments) 2,4, και 8. Ενα αρχείο των N bytes αποτελείται από (N / b) blocks και από κανένα ή ένα τμήμα από κάθε μέγεθος (b/2, b/4, b/8) (b είναι ο αριθμός των bytes/block) Εάν b είναι 8K, πολλά μικρά αρχεία μποορύν να αποθηκευτούν σ' ένα απλό block (χρησιμοποιώντας τα τμήματά του). Αυτός ήταν ο κύριος λόγος της επιτάχυσνσης του συστήματος αρχείων του UNIX BSD από 2-5% σε 30-47% του raw disk bandwidth.

4.4.1 Λειτουγίες του File Service και Επικοινωνία με το Block Service

Create: παράγει UFID + δεσμεύει χώρο για το file index + αρχικοποιεί διοχαρακτηριστικά (attributes) + προσθέτει μια νέα είσοδο (entry) στο File Location Map επιστρέφει το UFID.

Delete: σβήνει την είσοδο από το FLM + ελευθερώνει όλα τα blocks του αρχείου καθώς και τ' αντίστοιχα blocks του file index.

Read: χρησιμοποιείται η GetBlock

Write: (Allocate Block(στο τέλος του αρχείου) + PutBlock) ή (GetBlock(overwrite) + PutBlock)

4.5 Ανεκτικότητα σε λάθη

Γενικά, έτσι προφυλασσόμαστε από αποτυχία (crashing) του συστήματος, s/w bugs του λειτουργικού συστήματος, προσωρινά λάθη h/w κατά τη διάρκεια ανάγνωσης/εγγραφής στο δίσκο.

Προσεκτική Μνήμη (Careful Storage: CarefulPut and CarefulGet)

1. Αποθήκευση ενός checksum μαζί με κάθε block κάθε φορά που μεταβάλλονται τα περιεχόμενα του block.
2. Read: επαναπροσδιορισμός του checksum των δεδομένων που διαβάζουμε και σύγκριση μ' εκείνο που είναι αποθηκευμένο μαζί με το block. Εάν διαφέρουν τότε: επανέλαβε - εγκατέλειψε και επέστρεψε μήνυμα λάθους
3. Write: Μετά την εγγραφή, ξαναδιάβασε το block από το δίσκο και σύγκρινε το με το πρωτότυπο. Επανέλαβε, εάν χρειάζεται. Εάν το πρόβλημα εμμένει, σημείωσε το block ως "bad" ζήτησε ένα νέο block από το block service, και ξαναδοκίμασε.

Παρατήρηση:
Εάν το σύστημα αποτύχει κατά τη διάρκεια κάποιας εγγραφής, τότε τα δεδομένα χάνονται, π.χ. ένα block ίσως εγγραφεί κατά ένα μέρος του. Εάν όμως το block είναι τμήμα του file index, τότε το αρχείο χάνεται (το ίδιο με το FLM)

Σταθερή Μνήμη (Stable Storage)

Είναι ένα abstraction που χρησιμοποιείται για ν' αποφευχθεί το παραπάνω πρόβλημα (βασίζεται σε προσεκτικά (careful) put & (careful) get). Προκειμένου το σύστημα ν' αναρρώσει από αποτυχίες εγγραφών, χρειάζεται κάποιος πλεονασμός. Λογικά, το σύστημα θ' αποτύχει καθώς εγγράφεται ένα από τα δύο αντίγραφα => το άλλο θα παραμείνει ανέπαφο => μπορούμε να χρησιμοποιήσουμε το ανέπαφο αντίγραφο για να επαναφέρουμε το block στην κατάσταση που ήταν πριν το write failure.

Ενας σταθερός αποθηκευτικός χώρος (stable storage) αποτελείται από σταθερά blocks (stable blocks). Κάθε σταθερό block αντιπροσωπεύεται από δύο διαφορετικά blocks του δίσκου και οι λειτουργίες εγγραφής διεξάγονται και στα δύο blocks (είναι προτιμότερο να τοποθετούνται τα δύο αντίγραφα σε διαφορετικούς δίσκους).

StableGet: read one block with CarefulGet; εάν συμβεί λάθος τότε διάβασε το άλλο block με την CarefulGet.

StablePut: γράψε σε κάθε block με αυστηρή σειρά χρησιμοποιώντας (careful) PutBlock. Το δεύτερο block εγγράφεται ΜΟΝΟ εάν το πρώτο (careful) PutBlock ήταν επιτυχές.

Για παράλληλες λειτουργίες StablePut, χρησιμοποιείται monitor ή mutex.

4.6 Αρχεία "στο μικροσκόπιο"

Γιατί Αρχεία; 3 κύριοι λόγοι:

1. Για να μπορούν προγράμματα να χρησιμοποιήσουν πληροφορία μεγαλύτερη από το μέγεθος του virtual address space.
2. Για να υπάρχουν πληροφορίες/δεδομένα ανεξάρτητα από την ζωή των προγραμμάτων που τα χρησιμοποιούν.

3. Για να μπορούν δεδομένα να διαμοιράζονται μεταξύ προγραμμάτων.

Το F.S. είναι υπεύθυνο για την δόμηση/οργάνωση, προσπέλαση, προστασία, ονομασία, και υλοποίηση στο δίσκο.

Η "ονομασία" αναφέρεται στη διαδικασία μετάφρασης ενός υψηλού-επιπέδου ASCII ονόματος σε εσωτερικά file ids με βάση τα οποία το σύστημα μπορεί να προσπελάσει το αρχείο. Αυτή η διαδικασία υλοποιεί/προσφέρει το abstraction του file (δηλ. κρύβει λεπτομέρειες υλοποίησης του file).

<u>Η δομή των files ποικίλει:</u>

- Συνήθως ένα file είναι απλώς μια σειρά από bytes χωρίς καμμιά δομή, τουλάχιστον σε ότι αφορά το Λ.Σ.
- Σε μερικά F.S. ένα file είναι μια σειρά από records (που αποτελούνται από ένα συγκεκριμένο αριθμό bytes). Read & write λειτουργίες τώρα επιδρούν σε records και όχι σε τυχαία bytes.
- Σ' άλλα F.Ss ένα file έχει μια δενδρική οργάνωση, όπου οι εσωτερικοί ή όλοι οι κόμβοι είναι records. Κάθε record έχει και ένα κλειδί/κλειδολέξη με βάση την οποία το F.S. μπορεί να προσπελάσει τα συγκεκριμένα records που έχουν την ζητούμενη κλειδολέξη.

<u>Τύποι Files:</u>

Data Files: περιέχουν user-data
Directory files: περιέχουν κυρίως ζεύγη όπως (ASCII file name, εσωτερικό file id).
Block Special Files: χρησιμοποιούνται για να μοντελοποιήσουν δίσκους σαν αρχεία ➔ απ' ευθείας πρόσβαση σε ένα δίσκο γίνεται με open(), read(), write(), close()...
Character Special Files: Μοντελοποιούν σειριακές συσκευές όπως terminal, printers, h/ws, κ.λπ.

Data Files επίσης, μπορεί να είναι ή ASCII ή binary files. ASCII files περιέχουν text (κείμενο από ASCII chars, <*CR*>, κ.λπ). Binary Files συνήθως είναι executable images (δηλ. προγράμματα μετά το compilation & (linking) ή compiled ρουτίνες μιας βιβλιοθήκης που περιμένουν να γίνουν linked με προγράμματα.

<u>Τύποι προσπέλασης</u>: Sequential & Random Access. Ολα τα bytes ενός file προσπελαύνονται κατά σειρά (από το 1ο μέχρι το τελευταίο) όταν το file προσπελαύνεται με sequential mode. Οταν τα bytes/records ενός file προσπελαύνονται κατά μια τυχαία σειρά τότε έχουμε random access.

Η τυχαία προσπέλαση υλοποιείται με 2 τρόπους:

- Κάθε read() & write() προσδιορίζει που ακριβώς στο file θα επιδράσει η λειτουργία - δηλ. ποιό θα είναι το 1ο byte read/written.

- Το F.S. κρατάει έναν read/write pointer (για κάθε process που προσπελαύνει ένα file). Το 1ο byte read/written είναι αυτό στο οποίο δείχνει ο read/write pointer. Υπάρχει και ένα system call seek() με το οποίο ο χρήστης μπορεί να ενημερώσει τον δικό του r/w pointer.

Λειτουργίες που παρέχονται συνήθως από ένα F.S.: open(), close(), read(), write(), append(), seek(), create(), delete(), get/set-attributes)(, rename().

Οι μόνες λειτουργίες που χρειάζονται δικαιολόγηση ύπαρξης είναι οι open() & close(). Ο κύριος λόγος ύπαρξης είναι performance. Ο χρήστης δηλώνει ότι πρόκειται να προσπελάσει ένα file με το open(). Το F.S, ελέγχει τα δικαιώματα πρόσβασης του χρήστη στο αρχείο και αποταμιεύει στη Κ.Μ. του FS χρήσιμες πληροφορίες, όπως σε ποιά disk blocks βρίσκονται αποθηκευμένα τα δεδομένα του αρχείου και τα attributes. Αυτές οι πληροφορίες έτσι θα βρίσκονται στη Κ.Μ. και έτσι κάθε read() & write() δεν θα χρειαστεί διπλό κόστος I/O, ούτε το κόστος ελέγχου πρόσβασης.

Memory mapped files

Πολλοί υποστηρίζουν ότι το παραπάνω interface του F.S. είναι... "άβολο". Μια εναλλακτική λύση είναι memory-mapped files. Ο χρήστης μπορεί να καλέσει το Map (filename, virtual address), όπου το virtual address αντιπροσωπεύεται από μια μεταβλητή του προγράμματος του (π.χ. char *). Το Λ.Σ., εκτελώντας αυτή την εντολή, ενημερώνει τους πίνακες που κρατούν τις διευθύνσεις των disk blocks που αποτελούν το "swap space" για το process. Ετσι το swap space για το virtual address space στο οποίο γίνεται map το file είναι τα disk blocks του file ➔ τα page faults εξυπηρετούνται προσπελαύνοντας τα κατάλληλα disk blocks ➔ το process προσπελαύνει ένα file όπως οποιαδήποτε data structure του προγράμματός του (π.χ. με ένα for loop...).

Επιτρέποντας memory-mapped files δημιουργεί προβλήματα διαμοιρασμού. π.χ. ένα process έχει ενημερώσει ένα αρχείο αλλά τα σχετικά pages του αρχείου δεν έχουν αποθηκευτεί στο δίσκο ακόμα. Ενα άλλο process τότε προσπελάυνει το αρχείο χρησιμοποιώντας για παράδειγμα το κανονικό File System interface ➔ δεν θα δει την πιο πρόσφατη πληροφορία).

4.7 Κατάλογοι "στο μικροσκόπιο"

Εχουμε πει ότι directories συνήθως υλοποιούνται σαν files - directory files.

Γενικά οι εγγραφές ενός directory (directory entries) είναι ζεύγη: (filename, UFID). Ειδικότερα αυτό το UFID προσδιορίζει τις διευθύνσεις των disk blocks που αποτελούν το file καθώς και τα attributes.

Συνήθως, είτε το dir. entry περιέχει κατ' ευθείαν την πληροφορία για disk block addresses & attributes, είτε το dir. entry περιέχει ένα pointer σε κάποιο kernel data structure που περιέχει την παραπάνω πληροφορία.

Οταν το F.S. εξυπηρετεί την εντολή open(fname, ...) ψάχνει το directory μέχρι να βρεί την εγγραφή για το fname. Μετά ενημερώνει έναν kernel πίνακα με τα attributes και τα disk block addresses του file fname. Ολα τα read()/write() του ίδιου χρήστη γι αυτό το αρχείο χρησιμοποιούν αυτόν τον kernel πίνακα για να εκτελεστούν.

Συχνά, οι χρήστες θέλουν να οργανώσουν λογικά τα αρχεία τους. Αυτό έχει συνήθως ως αποτέλεσμα την δημιουργία ιεραρχικών F.S. Δηλ. υπάρχουν directories τα οποία περιέχουν άλλα directories κ.ο.κ. Αυτό με τη σειρά του έχει ως αποτέλεσμα τα files να ονομάζονται με pathnames όπως /usr/peter/foo. Το πρώτο "/" είναι το root directory του F.S. Κάποιο dir. του root έχει όλους τους χρήστες, "usr". Ενας χρήστης είναι ο "peter" ο οποίος έχει το δικό του dir το οποίο με τη σειρά του έχει ένα αρχείο με το όνομα "foo".

Σε τέτοια συστήματα υπάρχουν 2 τρόποι ονομασίας ενός αρχείου: absolute & relative path name. Το absolute όνομα είναι μοναδικό για κάθε αρχείο και πάντα αρχίζει με το root directory. Το relative όνομα σχετίζεται με την έννοια του current (ή working) directory. Αν το τωρινό dir είναι το /usr/peter, τότε τα ονόματα "foo" & "/usr/peter/foo" αναφέρονται στο ίδιο αρχείο. Κάθε process φυσικά έχει το δικό της current directory (δηλ. είναι μια per-process μεταβλητή). Ετσι αν για τον χρήστη "dora" το current dir. είναι /usr/dora και για τον χρήστη "peter" το current dir είναι /usr/peter τότε το όνομα "foo" αναφέρεται σε διαφορετικά αρχεία, ανάλογα με τον χρήστη.

Οι πιο συνήθης λειτουργίες που παρέχονται από directory services είναι: opendir(), closedir(), create(), delete(), read-dir(), rename(), link(), unlink().

Το read_dir() μπορεί υλοποιηθεί χρησιμοποιώντας το read() για το σχετικό dir. file. Αλλά συνήθως υλοποιείται διαφορετικά, έτσι ώστε να επιστρέφει την επόμενη εγγραφή του dir. Ετσι απαλλάσεται ο προγραμματιστής από την ανάγκη να ξέρει ακριβώς την ασωτερική δομή των dir.entries (π.χ. πόσα bytes και που είναι το filename, τα attributes, disk blocks, κ.λπ).

Το link() επιτρέπει σε πολλά directories να "περιέχουν" το ίδιο αρχείο - έτσι επιτρέπεται ο διαμοιρασμός του αρχείου από διαφορετικούς χρήστες που το τοποθετούν στο δικό τους directory.

Το unlink() διαγράφει ένα directory entry. Αν αυτό ήταν το μοναδικό dir. entry που αναφέρεται στο αρχείο τότε το αρχείο διαγράφεται επίσης.

4.8 Σημαντικά Θέματα Υλοποίησης

Διαχείριση των disk blocks ενός αρχείου

Συνήθως ο τρόπος διαχείρισης των disk blocks εξαρτάται από τον τρόπο ανάθεσης blocks σ' ένα αρχείο.

Υπάρχουν 3 γενικές στρατηγικές: contiguous, linked-list, & indexed allocation

Συνεχιζόμενη Ανάθεση (Contiguous): Ο πιο απλός τρόπος.

Οταν ένα file χρειάζεται N disk blocks, τότε N συνεχόμενα blocks ανατίθενται στο αρχείο αυτό. Πέραν της απλότητας, αυτή η στρατηγική έχει το πλεονέκτημα ότι για να διαβαστεί ολόκληρο το αρχείο απαιτείται να πληρωθεί το κόστος για το disk seek και το disk rotational delay μόνο μια φορά. Αυτό είναι πολύ σημαντικό!!!

Τα μειονεκτήματα έγκειται στο ότι συχνά δεν είναι γνωστό από πριν το μέγιστο μέγεθος του αρχείου και άρα το FS δεν μπορεί να ξέρει πόσα disk blocks ν' αναθέσει.

Επίσης (κυρίως όταν τα αρχεία διαγράφονται) δημιουργούνται προβλήματα fragmentation. Δηλ. ένας δίσκος μπορεί να έχει συνολικά N free blocks και ένα αρχείο που χρειάζεται N blocks να μην μπορεί να δημιουργηθεί λόγω του ότι τα N free blocks του δίσκου δεν είναι συνεχόμενα.

Αυτό το πρόβλημα λύνεται μόνο μέσω compaction αλλά αυτό στοιχίζει πάρα πολύ (disk_to_disk copying...). Μπορεί όμως να γίνει "off-line" δηλ. την νύχτα όταν το σύστημα δεν χρησιμοποιείται.

Linked List Allocation:

Μ' αυτή τη στρατηγική τα αρχεία είναι μια λίστα από blocks, με μερικά bytes του κάθε block να δείχνουν στο επόμενο block της λίστας. Ετσι αποφεύγεται το πρόβλημα fragmentation. Επίσης το dir. entry αποθηκεύει μόνο το address του 1ου block του αρχείου.

Το βασικό πρόβλημα αυτής της μεθόδου είναι ότι όταν επιθυμείται random access, η απόδοση του συστήματος είναι άθλια! π.χ. Για να προσπελαστεί το N-οστό block πρέπει να γίνουν N-1 παραπανίσια disk block I/Os.

Το πρόβλημα αυτό μπορεί να λυθεί χρησιμοποιώντας ένα index στη Κ.Μ. που περιέχει τους pointers για τα disk blocks (αφήνοντας τα disk blocks να περιέχουν μόνο data).

Indexed Allocation:

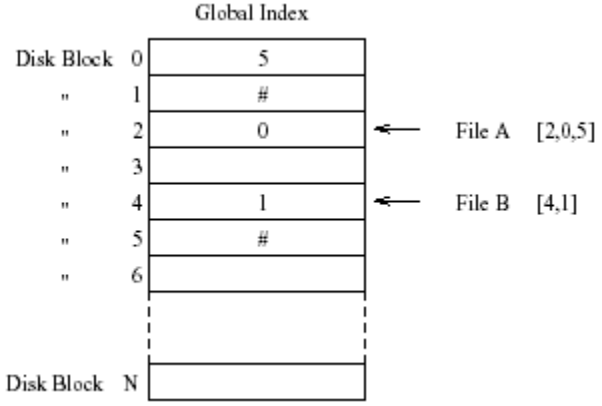
Η πρώτη επιλογή αφορά στο αν υπάρχει "per file" index ή ένα index για όλα τα αρχεία στο σύστημα. Το UNIX FS χρησιμοποιεί per-file index ενώ το MS-DOS χρησιμοποιεί "global" index.

Με global index, το index βρίσκεται πάντα στη ΚΜ ➔ παραπανίσια disk I/Os ελαχιστοποιούνται. Το πρόβλημα όμως είναι ότι το global index καταναλώνει πολύ Κ.Μ. γιατί διαθέτει ένα entry για κάθε block του δίσκου.

Ενα παράδειγμα per-file index είναι τα inodes που χρησιμοποιεί το UNIX. Το inode πρώτα περιέχει τα attributes. Κατόπιν 10 ptrs σε disk block για τα πρώτα 10 blocks του file. Κατόπιν ένα ptr που δείχνει σ' ένα disk block (single indirect block) που με τη σειρά

του δείχνει στα επόμενα 256 disk blocks του file. Κατόπιν έναν ptr σε double indirect block Κατόπιν έναν ptr σε triple indirect block.

Υλοποίηση Καταλόγων



Η δομή των dir. entries διαφέρει από FS σε FS όπως είπαμε. Στο UNIX έχει τη μορφή (fname, inode#). Ολα τα inodes (για όλα τα FS) είναι αποθηκευμένα στην αρχή του δίσκου που αποθηκεύει το FS, και ονομάζονται με inode1, 2, 3, ...

Μετάφραση Ονόματος (Pathname translation): Η διαδικασία που δέχεται σαν είσοδο ένα pathname (π.χ. "/d1/d2/d3/foo") και παράγει σαν έξοδο το inode# του αρχείου.

Η διαδικασία αυτή έχει ως εξής:

1)
   a) Το inode για το root directory "/" ανακτάται από τον δίσκο. (Αυτό το inode βρίσκεται σ' ένα disk address γνωστό και μη ενημερώσιμο).
   b) Απ' το inode του root βρίσκονται τα disk blocks (που περιέχουν τα dir.entries) ανακτώνται και ψάχνονται για το entry "d1". Οταν βρίσκεται το dir. entry για το "d1" τότε βρίκεται και το inode# του d1.
2)
   a) Το inode για το "d1" ανακτάται απ' τον δίσκο.
   b) Τα disk blocks του dir file "d1" ανακτώνται και ψάχνονται για το entry "d2"….
   c) Τέλος το inode & disk blocks του "dir3" ανακτώνται και ψάχνονται για το entry "foo". Ετσι βρίσκεται το inode# για το file foo το οποίο ανακτάται και τοποθετείται στη μνήμη (σ' ένα cache ειδικό για inodes που λέγεται inode table).
3) Αφού τοποθετηθεί το inode στο inode table του kernel ο file descriptor πίνακας του process ενημερώνεται. Μια εγγραφή του χρησιμοποιείται για έναν ptr ακολουθώντας τον οποίο μπορεί ο kernel να βρεί το inode στο inode table. Τέλος επιστρέφεται ο δείκτης ...

Διαμοιραζόμενα Αρχεία [SHARED FILES]

Για να διευκολυνθεί ο διαμοιρασμός αρχείων συχνά ένα file εμφανίζεται σε πολλά διαφορετικά dir. (π.χ. σ' ένα dir για κάθε χρήστη που το διαμοιράζεται). Αυτό επιτυγχάνεται με links που συνδέουν ένα dir. entry μ' ένα αρχείο σε κάποιο άλλο directory.

Αν τα dir. entries ενπεριέχουν τα disk block addresses τότε στο dir. που ορίζεται το link σ' ένα αρχείο F1 θα πρέπει να γίνει ένα αντίγραφο των disk block addresses του F1. Αν αυτά αλλάξουν (π.χ. append()) τότε προκύπτει πρόβλημα διαχείρισης αντιγράφων.

Στο UNIX το πρόβλημα λύνεται ως εξής: Dir. entries δεν περιέχουν disk block addresses αλλά pointers σε kernel data structures που τις περιέχουν (δηλ. τα inodes).

Μια άλλη λύση βασίζεται στην έννοια symbolic link. Οταν δημιουργείται ένα link L1 στο file F1, το L1 είναι ένα ειδικό αρχείο τύπου link. Τα περιεχόμενα αρχείων τύπου link είναι μόνο το pathname του αρχείου στο οποίο είναι links. Οπως είναι προφανές, symbolic links προσθέτουν παραπάνω overhead.

Πρέπει επίσης να προσεχθεί να μην δημιουργούνται infinite loops σε προγράμματα τα οποία προσπελαύνουν όλα τα αρχεία ενός directory tree (ή ακόμα, να μην προσπελασθούν μερικά αρχεία >1 φορές: μια από το κανονικό dir. και μια μέσω του link).

Ανάθεση Μονάδων Δίσκου σε Αρχεία

Οπως προείπαμε, αρχεία συνήθως δεν αποθηκεύονται σε συνεχή sectors του δίσκου. Αυτό συμβαίνει για να αποφευχθούν τα προβλήματα fragmentation και disk_to_disk copying.

➔ αρχεία χωρίζονται σ' ένα αριθμό blocks και το κάθε file block αποθηκεύεται σε συνεχόμενα disk sectors.

Μέγεθος BLOCK

Πόσο μεγάλα πρέπει να είναι τα file blocks; Οπως προείπαμε, σε πολλά περιβάλλοντα τα αρχεία είναι μικρά.

➔ Αν το file block είναι μεγάλο τότε θα σπαταλείται disk space. Αν, αντιθέτως, τα file blocks είναι μικρά, τότε για να ανακτηθεί ένα ολόκληρο αρχείο θα χρειστούν πολλά disk seeks (ένα για κάθε block).

➔ Η απόδοση (throughput) του συστήματος θα είναι κακή γιατί για μεγάλα χρονικά διαστήματα ο δίσκος κάνει seek (και όχι "χρήσιμες" λειτουργίες).

➔ βλέπουμε ένα trade-off μεταξύ space efficiency & throughput.

Διαχείριση Ελευθέρων Μονάδων στο Δίσκο

Συνήθως, χρησιμοποιούνται 2 μεθόδοι:

Linked List:
Κάθε κόμβος της λίστας είναι ένα disk block.
Κάθε κόμβος περιέχει:

- την διεύθυνση του επόμενου κόμβου της λίστας
- και N-1 διευθύνσεις ελευθέρων (free) blocks, όπου N είναι ο αριθμός των disk block διευθύνσεων που χωρούν σ' ένα disk block. (π.χ. με μέγεθος disk blocks = 2K και με 3-byte disk block addresses, N = 2048/3).

Bit Map:
Ενας πίνακας/διάνυσμα (bit map) με N εγγραφές απαιτείται για ένα δίσκο με N blocks. Τα ελεύθερα blocks αντιπροσωπεύονται με την τιμή 1 στο bit map.
➔ για κάθε block κρατείται πληροφορία μεγέθους 1 bit και όχι μερικά bytes όπως στη μεθοδο linked list
➔ (εκτός μερικών περιπτώσεων) η μεθοδος bit map είναι προτιμότερη, από πλευράς χώρου. Επίσης, επειδή το bit map είναι μικρότερο ➔ υπάρχει μεγαλύτερη πιθανότητα να χωράει στη Κ.Μ.

- Αν ναι, τότε η μεθοδος bit map προτιμάται
- Αν όχι, τότε κάθε block/κόμβος της λίστας (linked list) περιέχει συνήθως πιο πολλά ελεύθερα blocks απ' ότι ένα block του bit map ➔ ίσως να είναι προτιμότερη η μέθοδος linked list.

4.9 Βελτιώνοντας την Απόδοση του Συστήματος

Οπως έχουμε πει, κάθε πρόσβαση στη Κ.Μ. κοστίζει < 1 μs, ενώ κάθε πρόσβαση στο δίσκο κοστίζει μερικές δεκάδες ms.

➔ ο FS θα έχει τη ν καλύτερη δυνατή απόδοση όσο πιο λίγες φορές χρειάζεται disk I/O!!!

Η επίτευξη του στόχου αυτού βασίζεται στην έννοια του **buffer cache**. Το buffer cache είναι ένα τμήμα της Κ.Μ. (στο kernel space) το οποίο αποθηκεύει προσωρινά μερικά disk blocks.

Οταν κάποιο user process ζητεί πρόσβαση σε κάποιο block, ο kernel φορτώνει πρώτα αυτό το block στη buffer cache και μετά το δίνει στο user process address space.

➔ οι μεταγενέστερες αιτήσεις (από το ίδιο ή άλλα user processes) για το ίδιο block θα εξυπηρετηθούν από τη Buffer Cache
➔ το disk I/O αποφεύγεται!

Ετσι, κάθε φορά που ζητείται κάποιο disk block, ο kernel πρώτα ψάχνει τη Buffer Cache και μόνο αν δεν το βρει απευθύνεται στο δίσκο.

Φυσικά, απαιτείται κάποιος αλγόριθμος replacement για όταν γεμίζει η Buffer Cache. Αυτός ο αλγόριθμος είναι συνήθως LRU (με μερικές αλλαγές).

- Πριν ένα block της Buffer Cache "φιλοξενήσει" ένα καινούργιο disk block, θα πρέπει το αρχικό block να εγγραφεί στο δίσκο αν έχει ενημερωθεί όσο ήταν στη Buffer Cache (δηλ. αν είναι dirty).

  [Αυτή είναι η ανάλογη ενέργεια που συμβαίνει από τον paging system, που γράφει ένα page στο swap space πριν το ίδιο frame φιλοξενήσει κάποιο καινούργιο page].

- Οι αλλαγές στον αλγόριθμο LRU χρειάζονται:
  1. για να αποφευχθούν όσο το δυνατόν περισσότερα disk I/Os.
  2. για να αποφευχθούν προβλήματα ασυνέπειας του F.S. ή χάσιμο δεδομένων όταν συμβούν βλάβες.
- Οταν κάποιο block στη Buffer Cache περιέχει ευαίσθητα δεδομένα, όσον αφορά την συνέπεια του συστήματος, τότε όταν αυτό το block ενημερωθεί εγγράφεται αμέσως στο δίσκο.

  Ευαίσθητα δεδομένα είναι: inode blocks, directory blocks, και γενικά blocks που δεν έχουν user data.

  π.χ. ¯Σκεφθείτε τι θα συμβεί αν το inode block ενός αρχείου ενημερωθεί και πριν γραφτεί στο δίσκο, πέσει το σύστημα.

- Οταν κάποιο block στη Buffer Cache είναι μόνο μερικώς ενημερωμένο τότε τοποθετείται στο τέλος της LRU λίστας ➔ αφού δεν αναπληρώνεται αμέσως υπάρχει πιθανότητα όταν, μετά από χρόνο, αναπληρωθεί, να έχει ενημερωθεί ολόκληρο και έτσι θα συμβεί ένα disk I/O για όλες τις ενημερώσεις ενός block.
- Επίσης συνήθως υπάρχει κάποιο άνω όριο, όσον αφορά τον χρόνο κατά τον οποίο ένα ενημερωμένο block παραμένει στη Buffer Cache χωρίς να εγγραφεί στο δίσκο.
- Για να μην χαθούν δεδομένα όταν συμβούν βλάβες, πολλά συστήματα παρέχουν ειδικά system calls (π.χ. flush() ή SYNC) τα οποία γράφουν όλα ενημερωμένα blocks στη Buffer Cache στο δίσκο.

Στο UNIX F.S. υπάρχει ένα background program το οποίο εκτελείται κάθε 30 secs το οποίο εκτελεί SYNC ➔ τα χαμένα data μπορεί να είναι αυτά που γράφτηκαν μόνο τα τελευταία 30 secs.

Ομαδοποίηση (Clustering)

Μια άλλη τεχνική για να βελτιωθεί η απόδοση ενός F.S. βασίζεται στην έννοια του clustering. Μ' αυτή τη τεχνική, blocks που ανακτώνται/ ζητούνται συνήθως το ένα μετά το άλλο, τοποθετούνται κοντά στο δίσκο (π.χ. στο ίδιο ή κάποιο κοντινό κύλινδρο).

➔ το seek που απαιτείται για ν' ανακτηθεί το επόμενο block προσθέτει ελάχιστο overhead (αφού το κοστος του seek είναι συνάρτηση του διαστήματος που διανύουν οι κεφαλές του δίσκου).

Προσέξτε ότι όταν η διαχείριση των free blocks γίνεται με bit maps διευκολύνεται η ανάθεση συνεχόμενων blocks σε κοντινά disk blocks. Πώς μπορεί να επιτευχθεί κάτι ανάλογο με free lists ;

Τοποθέτηση λαμβάνοντας υπ'όψιν την περιστροφή δίσκου (Rotationally-optimal Placement)

Στρατηγικές ανάθεσης free blocks σε files προσπαθούν επίσης να μειώσουν τον χρόνο περιστροφής (rotational delay). Ετσι, όταν συνεχόμενα file blocks τοποθετούνται στον ίδιο track (ή cylinder του δίσκου) υπάρχει ένα "κενό" το μέγεθος του οποίου εξαρτάται από τον χρόνο που απαιτείται για την ικανοποίηση ενός get-disk-block-request.

π.χ. αν μια ολόκληρη περιστροφή του δίσκου γίνεται σε 10 ms και από την στιγμή που ανακτάται ένα disk block περνούν 3 ms πριν ο δίσκος λάβει την αίτηση για το επόμενο block, τότε το μέγεθος του "κενού" είναι 3 ms.

Σχετική Τοποθέτηση inode και Data

Σε παλιά UNIX F.S. τα blocks των inodes τοποθετούνται στα αρχικά blocks του δίσκου.
➔ απαιτείται μεγάλο seek μεταξύ του inode block ενός αρχείου και των data blocks του αρχείου.

Μια λύση στο πρόβλημα αυτό βασίζεται στην έννοια των cylinder groups. Κάθε cylinder group αποτελείται από N συνεχόμενα cylinders και περιέχει inodes, blocks για data, και την δική του free list.

Οταν δημιουργείται ένα αρχείο επιλέγεται οποιοδήποτε inode από κάποιο cylinder group. Καθώς το μέγεθος του αρχείου μεγαλώνει, το σύστημα καταβάλλει προσπάθειες ούτως ώστε τα data blocks του αρχείου να βρίσκονται στο ίδιο cylinder group με το inode block.

Κεφάλαιο 5. Συστήματα Εισόδου - Εξόδου (INPUT/OUTPUT)

5.1 I/O Υλικό

Το I/O σύστημα αποτελεί ένα πολύ μεγάλο τμήμα ενός Λ.Σ.

Συνήθως "διευθύνει" όλες τις συσκευές I/O (π.χ. δίσκους, τερματικά, εκτυπωτές, δίκτυα, ...). Κυρίως πρέπει ένα I/O σύστημα να εκδίδει εντολές στις συσκευές, ν' αναγνωρίζει και να εξυπηρετεί τα interrupts των συσκευών, και να χειρίζεται βλάβες και λάθη σχετικά με I/O.

Πριν εξετάσουμε ένα I/O (software) σύστημα, ας δούμε πρώτα το I/O hardware από κοντά. Θα εστιάσουμε την προσοχή μας σε γενικές πληροφορίες για I/O συσκευές, για controllers συσκευών (device controllers) και για Direct Memory Access.

5.1.1 I/O Συσκευές (Devices)

Όπως δίσκοι, τερματικά, εκτυπωτές, δίκτυα, ποντίκια, ταινίες, ... Γενικά, χωρίζονται σε 2 μεγάλες κατηγορίες block devices και character devices.

- Block devices αποθηκεύουν πληροφορία σε μονάδες σταθερού μεγέθους (# bytes) που λέγονται blocks. Κάθε block πληροφορίας έχει δική του διεύθυνση και μπορεί να εγγραφεί ή ν' ανακτηθεί ανεξάρτητα από τ' άλλα blocks.
  Επίσης, το block αποτελεί την μικρότερη μονάδα πληροφορίας που μεταφέρεται από (προς) την συσκευή.
- Character devices διαχειρίζονται αδόμητη πληροφορία - δηλ. απλώς μια σειρά από chars. Δεν υπάρχει η δυνατότητα διευθυνσιοποίησης πληροφορίας και ανεξάρτητης πρόσβασης. Πρέπει να σημειωθεί πρώτον ότι υπάρχουν διάφορες ερμηνείες για τα χαρακτηριστικά που πρέπει να έχουν blocks και char. devices. Δεύτερον, ότι ακόμα και block devices έχουν ένα character interface (π.χ. disks σε UNIX).

Ολοι συμφωνούν ότι οι δίσκοι είναι block devices και ότι τερματικά, εκτυπωτές, δίκτυα, ποντίκια είναι character devices. Πολλοί επίσης θεωρούν τις μαγνητικές ταινίες ως block devices.

5.1.2 Ελεγκτές Συσκευών (CONTROLLERS)

- Πολλές συστκευές I/O έχουν δύο διαφορετικά τμήματα: ένα ηλεκτρονικό και ένα μηχανικό. Το ηλεκτρονικό τμήμα, που είναι στην ουσία ένας επεξεργαστής, ονομάζεται controller (ή adapter).
- Ο controller είναι μια κάρτα που μπαίνει στον υπολογιστή. Αυτή η κάρτα έχει εισδοχές για συνδέσεις με I/O συσκευές.
- Η επικοινωνία του controller με τις συσκευές γίνεται μέσω ενός I/O bus. Το I/O bus είναι διαφορετικό από το system bus που συνδέει τον CPU, memory και controllers.

- Η πρώτη ευθύνη ενός disk controller είναι να κάνει error checking. Πρώτα οργανώνει τα bits που έρχονται από την συσκευή σε bytes και μετά πιστοποιεί το checksum που περιέχει η πληροφορία. Κατόπιν, η πληροφορία μπορεί να μεταφερθεί στη Κ.Μ.
- Σαν άλλο παράδειγμα ένας terminal controller είναι υπέυθυνος να διαβάζει τα bytes (chars) από την Κ.Μ. και να καθοδηγεί την ακτίνα του CRT (οθόνης) για να τυπωθούν οι αντίστοιχοι ASCII χαρακτήρες. Επίσης, ανάλογα χειρίζεται λειτουργίες όπως scrolling, κ.λπ.

Επικοινωνία με Ελεγκτές

- Ο πιο διαδεδομένος τρόπος βασίζεται στην έννοια του memory-mapped I/O. Ενα μέρος του address space του υπολογιστή είναι αφοσιωμένο (dedicated) να παρέχει ειδικούς registers (device registers - Control and Status Registers (CSR)).
- Το Λ.Σ. επικοινωνεί με τον Controller χρησιμοποιώντας αυτούς τους registers. π.χ. μπορεί να "γράψει" εντολές (δίνοντας ειδικές τιμές) και να ορίσει τιμές παραμέτρων όπως πιο disk block πρέπει να προσπελαστεί, διεύθυνση στη Κ.Μ. όπου πρέπει ν' αποθηκευτεί, κ.λπ.
- Αφού δεχθεί μια εντολή ο controller θα την εκτελέσει και θα γράψει πληροφορίες σχετικά με την εντολή σε ειδικούς registers που θ' εξεταστούν από το Λ.Σ. (π.χ. για να διαπιστωθεί αν η εντολή εκτελέστηκε επιτυχώς).
- Κατόπιν, ο controller θα προκαλέσει ένα interrupt ώστε να εκτελεστεί το κατάλληλο τμήμα του Λ.Σ. και να συνεχίσει την περαιτέρω επεξεργασία του I/O call.

5.1.3 Απ' Ευθείας Πρόσβαση στη Μνήμη (Direct Memory Access -- DMA)

DMA αναφέρεται στην ικανότητα των controllers να εκτελέσουν πλήρως μια I/O εντολή χωρίς την ανάμειξη του CPU.

π.χ. ένας disk controller μπορεί ν' ανακτήσει το ζητούμενο block από το δίσκο και μετά να το μεταφέρει στην κατάλληλη διεύθυνση στη Κ.Μ. Χωρίς την ανάγκη να επέμβει ο CPU να κάνει την αντιγραφή από τον buffer του controller στη Κ.Μ.

Αυτή η ικανότητα DMA ενός controller κρίνεται πολύ σημαντική γιατί εφ' όσον γίνεται η παραπάνω αντιγραφή, το CPU εκτελεί εντολές κάποιου άλλου προγράμματος ➔ η απόδοση του συστήματος είναι κατά πολύ καλύτερη.

Είδαμε πως οι disk controllers έχουν δικούς τους buffers. Γιατί αυτοί χρειάζονται;

Ο κύριος λόγος απορρέει από την ανάγκη χρησιμοποίησης του system bus για να μεταφερθεί πληροφορία από τον δίσκο στη Κ.Μ. Το system bus χρησιμοποιείται από το CPU για πρόσβαση στη μνήμη και για επικοινωνία CPU. Ετσι, την στιγμή που τα bits ενός disk block καταφθάνουν στον controller, το system bus μπορεί να μην είναι διαθέσιμο (δηλ. να μεταφέρει κάποια άλλη πληροφορία) ➔ Οι buffers ενός controller

αποφεύγουν τέτοια προβλήματα. Ενα άλλο πρόβλημα προκύπτει από την ανικανότητα μερικών controllers να κάνουν input και output ταυτόχρονα.

π.χ. όταν αρχίζουν DMA για ένα block στον buffer τους, το επόμενο ζητούμενο block του δίσκου περνά κάτω από την κεφαλή και δν μπορεί να αποθηκευτεί σ' έναν άλλο buffer. Μόλις τελειώσει το 1ο DMA, τότε θα πρέπει να περιμένει ο controller μέχρι το επόμενο ζητούμενο block να ξαναπεράσει κάτω από την κεφαλή - μια σημαντική καθυστέρηση.

Μια λύση στο πρόβλημα αυτό βασίζεται στην έννοια του block interleaving: Αντί τα blocks ενός track του δίσκου ν' αποθηκεύονται με την σειρά 1, 2, 3, ... αποθηκεύονται με την σειρά 1, 5, 2, 6, 3, 7, 4, 8

➔ αντί για καθυστέρηση μιας σχεδόν-ολόκληρης περιστροφής για ν' ανακτηθεί το block#2, (όσο το block#1 γίνεται DMA) υπάρχει μόνο η καθυστέρηση να περάσει το block#5 κάτω από την κεφαλή.

5.2 Λογισμικό I/O (Software)

Το λογισμικό ενός I/O συστήματος είναι πολύ πολύπλοκο. Ως εκ τούτου, συνήθως, οργανώνεται με την βοήθεια πολλών επιπέδων. Κάθε επίπεδο είναι αποκλειστικά υπεύθυνο για τμήμα της συνολικής λειτουργικότητας και παρέχει interface και συγκεκριμένες υπηρεσίες στο αμέσως πιο πάνω επίπεδο. Συνήθως το λογισμικό ενός I/O συστήματος οργανώνεται ως εξής:

user level s/w	4
device independent s/w	3
device drivers	2
interrupt handlers	1

Ισως η πιο σημαντική έννοια σχετικά με I/O s/w να είναι device independence. Είναι ξεκάθαρο ότι μέρος του I/O s/w εξαρτάται από το είδος και τα χαρακτηριστικά της I/O συσκευής. Είναι όμως εξ ίσου επιθυμητό τα προγράμματα που προσπελαύνουν αρχεία σ' ένα floppy να μπορούν να τρέξουν αν αυτά τα αρχεία μεταφερθούν στο σκληρό δίσκο.

Μια άλλη επιθυμητή ιδιότητα είναι uniform naming: Δηλ. όλες οι συσκευές κατανομάζονται ομοιόμορφα (π.χ. στο UNIX τ' ονόματα συσκευών είναι ένα pathname).

Τέλος, το I/O s/w πρέπει να κρύβει την ασύγχρονη συμπεριφορά/ιδιότητα του I/O (δηλ. βασίζονται σε interrupts, και όταν αυτά συμβούν, το CPU τρέχει κάποιο άλλο πρόγραμμα έχοντας μπλοκάρει το πρόγραμμα που ζήτησε το I/O ➔ το process που κάνει I/O νομίζει ότι το I/O είναι blocking και όχι asynchronous.

5.2.1 Χειρισμός Διακοπών (Interrupts)

Βρίσκονται στο κατώτερο επίπεδο του I/O s/w ώστε τα ανώτερα επίπεδα να μην χρειάζεται να ξέρουν όλες τις "δυσάρεστες" λεπτομέρειες. Αυτό επιτυγχάνεται με το να μπλοκάρουν όποιο process επιχειρεί I/O (για το οποίο χρειάζεται επικοινωνία με τον controller).

Οταν ο controller τελειώσει, θα προκαλέσει interrupt και ο handler θα ξεμπλοκάρει το process (π.χ. βγάζοντας το από ένα WAIT LIST και βάζοντάς το στο READY LIST που εξετάζεται από τον CPU scheduler - στο UNIX).

5.2.2 Οδηγοί Συσκευών (Device Drivers)

Περιέχει τον κώδικα που είναι εξαρτώμενος από την συσκευή (device-dependent). Ο οδηγός έχει την βασική ευθύνη να επικοινωνεί με τον controller (μέσω των CSR του controller σ' ένα memory-mapped I/O system, όπως είδαμε πριν).

Ο οδηγός συνήθως έχει ένα work queue όπου εναποτίθενται αιτήσεις για I/O. Η σειρά με την οποία οι διάφορες αιτήσεις αποθηκεύονται στο work queue ενός driver είναι πολύ σημαντική για την απόδοση του συστήματος γιατί καθορίζει π.χ. την σειρά με την οποία τα ζητούμενα disk blocks θ' ανακτηθούν (➔ εδώ έχουμε ένα άλλο είδος scheduling - disk scheduling).

Ο driver επίσης κρατάει αρκετές πληροφορίες για την συσκευή που διαχειρίζεται.

π.χ. για ένα δίσκο πρέπει να ξέρει την "γεωμετρία του" π.χ. πόσα disk platters, πόσοι κύλινδροι, πόσα tracks σε κάθε κύλινδρο, πόσα sectors σε κάθε track, κ.λπ. Επίσης πρέπει να ξέρει σε ποιό κύλινδρο βρίσκεται τώρα η κεφαλή (π.χ. για να εκτιμήσει αν χρειάζεται να ζητήσει από τον controller να κάνει Seek.

Αφ' ής στιγμής εκδώσει την εντολή στον controller, μπλοκάρει. Θα τον ξεμπλοκάρει ο interrupt handler του interrupt που θα προκαλέσει ο controller όταν διεκπεραιώσει την εντολή για I/O. Ευθύς αμέσως θα εξετάσει τον κατάλληλο CSR register για τυχόν λάθη. Αν έχει υπάρξει λάθος, θα προσπαθήσει να το αντιμετωπίσει. Αλλιώς, θα επικοινωνήσει με το αμέσως ανώτερο επίπεδο (δηλ. το device indep. s/w) συνήθως για να του "περάσει" πληροφορία, όπως π.χ. ένα disk block που ανακτήθηκε, error message, κ.λπ.

Κατόπιν, αν το work queue δεν είναι άδειο, θα αφαιρέσει την επόμενη I/O αίτηση και θα στείλει την αίτηση στον controller. Αν είναι άδειο, τότε θα περιμένει την επόμενη αίτηση (π.χ. θα μπλοκάρει αν ο driver είναι process ή άλλως θα "επιστρέψει" (δηλ. καλεί return).

5.2.3 Λογισμικό Ανεξάρτητο Συσκευών (Device-Independent I/O S/W)

Οι κύριες λειτουργίες σ' αυτό το επίπεδο είναι: ονομασία (naming), προστασία, buffering, ανάθεση blocks, διαιτησία πρόσβασης σε "αφοσιωμένες" συσκευές και αναφορά λαθών, και buffering.

Οπως είπαμε πρέπει να υπάρχει uniform naming. Στο UNIX κάθε device έχει ένα file name (στο οποίο αντιστοιχεί ένα ειδικό inode) - device special file.

Το ειδικό inode περιέχει δύο αριθμούς: major device number, minor device number. Ο major device number ταυτοποιεί τον driver (π.χ. disk driver, tape driver, ...). Ο minor device number ταυτοποιεί την συσκευή που εμπλέκεται στο I/O (π.χ. ποιός δίσκος).

Φυσικά, αφού πολλές συσκευές χρησιμοποιούνται από πολλούς χρήστες, οι συσκευές χρειάζονται προστασία π.χ. ο κάθε χρήστης δεν μπορεί νάχει απ' ευθείας πρόσβαση σε όποιο δίσκο θέλει (γιατί π.χ. δεν πρέπει να μπορεί να προσπελάσει ξένη πληροφορία).

Στο UNIX αυτό επιτυγχάνεται με την χρήση των rwx bits.

Συνήθως το I/O σύστημα s/w χρειάζεται buffers στη Κ.Μ. Για block devices, είπαμε ότι η μονάδα προσπέλασης πληροφορίας είναι το block - το h/w δεν επιτρέπει τίποτα άλλο. Χρησιμοποιώντας buffers το Λ.Σ. επιτρέπει σε χρήστες (user processes) να προσπελάσουν οποιοδήποτε τμήμα πληροφορίας θέλουν ➔ κρύβεται αυτή η h/w απαίτηση/περιορισμός από χρήστες.

Σε block devices, όπως δίσκοι, αυτό το σύστημα buffering έχει ευεργετικές συνέπειες για την απόδοση του συστήματος. Γιατί, αν το ζητούμενο block είναι σ' ένα buffer τότε αποφεύγεται το disk I/O.

Σε character devices, οι buffers χρειάζονται γιατί π.χ. input από πληκτρολόγιο μπορεί να φθάσει πριν να μπορεί να γίνει output.

Ο αλγόριθμος και οι δομές δεδομένων, για να αναθέτουν ελεύθερα (free) disk blocks, είναι ένα άλλο μέρος του device indep. I/O s/w.

Μερικές συσκευές (π.χ. εκτυπωτές) μπορούν να χρησιμοποιηθούν μόνο από ένα process κάθε φορά. Ετσι, αιτήσεις για μη-διαθέσιμες συσκευές είτε απορρίπτονται είτε μπλοκάρουν.

Οταν ο driver δεν μπόρεσε ν' αντιμετωπίσει κάποιο λάθος (συνήθως, προσπαθώντας/επαναλαμβάνοντας το I/O call μερικές φορές) τότε ειδοποιεί το device indep. I/O s/w για το λάθος. Αυτό, ανάλογα με το είδος του λάθους, είτε το αναφέρει στο User process, είτε τερματίζει το Λ.Σ. (για πιό σοβαρά λάθη).

5.2.4 Λογισμικό στο Επίπεδο του Χρήστη (User-level I/O S/W)

Ενα μεγάλο μέρος αυτού του τμήματος περιέχει τις library routines (π.χ. read(), write(), open(), close(), seek(), printf, getc, ...).

Ενα άλλο μεγάλο μέρος είναι spooling s/w. Αυτό το λογισμικό έχει την ευθύνη να διαχειρίζεται αφοσιωμένες συσκευές σε multiprogramed συστήματα (π.χ. εκτυπωτές: μόνο ένας χρήστης πρέπει να μπορεί να εκτυπώνει κάθε φορά).

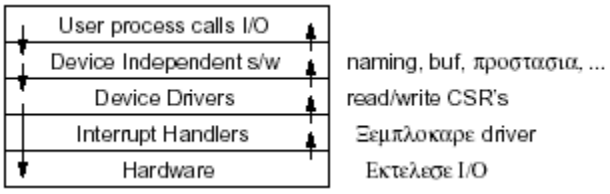
Ο spooler είναι ένα ειδικό process (δαίμονας) που διαχειρίζεται ένα spooling directory.

Για να εκτυπωθεί ένα αρχείο, το process το βάζει στο spooling directory.

Ο spooler είναι το μόνο process του printer. Ο spooler μόλις τελειώσει μια αίτηση, παίρνει ένα άλλο αρχείο από το spooling directory και το εκτυπώνει.

Το spooling χρησιμοποιείται επίσης και για άλλες λειτουργίες: π.χ. network transfer.

5.2.5 Ροή Πληροφορίας στο I/O s/w



5.3 Μαγνητικοί Δίσκοι

Από τις πιο σημαντικές συσκευές ενός σύγχρονου υπολογιστή. Το πλεονέκτημά τους:

- πολύ μεγαλύτερη χωρητικότητα από Κ.Μ.
- πιό φθηνή μνήμη
- "σταθερή" μνήμη (non-volatile): όταν καταρεύσει η μηχανή, τα περιεχόμενα δεν χάνονται.

Γεωμετρία:

Κύλινδροι περιέχουν tracks που περιέχουν sectors. Κάθε δίσκος του drive έχει 2 επιφάνειες. Για κάθε επιφάνεια αντιστοιχεί μία κεφαλή (head). Ολοι οι sectors περιέχουν τον ίδιο αριθμό bytes. Αν και τα "εσωτερικά" tracks είναι μικρότερα από τα εξωτερικά, όλα τα tracks έχουν τον ίδιο αριθμό sectors (zoned disks).

(Πρόσφατα παρουσιάστηκαν δίσκοι, όπου αυτό δεν ισχύει πλέον ➔ εξωτερικά tracks έχουν πιο πολλά sector).

Συνήθως, όταν ένας controller ελέγχει > *1* δίσκου, τότε υπάρχει η δυνατότητα για overlapped seeks (δηλ. ο controller ζητεί απ' ένα δίσκο ένα seek και πριν τελειώσει ζητεί και 2ο seek απ' άλλον δίσκο). Επίσης συχνά μπορεί να ζητηθεί ένα read/write από ένα δίσκο καθ' όσον άλλος δίσκος εκτελεί μια εντολή seek. Δυστυχώς, όμως, παράλληλα read/write σε δύο δίσκους ενός controller, δεν είναι δυνατό).

Φυσικά, τα παραπάνω έχουν ευεργετικές συνέπειες για την απόδοση του συστήματος.

5.3.1 Αλγόριθμοι Χρονοπρογραμματισμού Πρόσβασης σε Δίσκους (Disk Scheduling)

Software: ο αλγόριθμος με βάση τον οποίο ο disk driver επιλέγει την επόμενη αίτηση για να εξυπηρετήσει ο δίσκος.

Στόχος των αλγορίθμων αυτών είναι η βελτιστοποίηση της απόδοσης.

Βασικά συστατικά κόστους προσπέλασης στον δίσκο:

- seek time: κόστος μετακίνησης του βραχίωνα (με τις κεφαλές) στο σωστό κύλινδρο. Εξαρτάται από την απόσταση που διανύεται ➔ συμφέρουν "μικρά" seeks.
- rotational delay: ο χρόνος που απαιτείται (μετά το seek) ώστε το επιθυμητό sector να έρθει κάτω από την κεφαλή.
- transfer time: ο χρόνος που απαιτείται για την ανάκτηση/μεταφορά δεδομένων (μετά από seek και rotational delay) από/προς τον δίσκο.

Με τωρινή τεχνολογία, το seek time συνεισφέρει το μεγαλύτερο κόστος ➔ αυτό είναι το κόστος που οι disk scheduling αλγόριθμοι προσπαθούν να ελαχιστοποιήσουν.

Ο Αλγόριθμος FIFO

Ο driver διαχειρίζεται μια ουρά. Καινούργιες αιτήσεις τοποθετούνται στο τέλος της ουράς.
Μετά από κάθε interrupt απ' τον δίσκο, που δηλώνει το πέρας της προηγούμενης αίτησης, ο driver δίνει στον δίσκο την αίτηση στη κορυφή της ουράς. Φυσικά, με FIFO δεν προσφέρονται δυνατότητες ελαχιστοποίησης του seek time. Για να επιτύχουμε αυτό το στόχο πρέπει να αξιοποιήσουμε τις εξής γνώσεις των driver/controller:

- τωρινή θέση (δηλ. αριθμός κυλίνδρου) των κεφαλών
- επιθυμητός κύλινδρος για κάθε αίτηση στην ουρά.

Ο Αλγόριθμος SHORTEST SEEK FIRST

"Επόμενη" αίτηση ορίζεται ως αυτή που αναφέρεται στον κοντινότερο κύλινδρο (σε σχέση με τον κύλινδρο που βρίσκονται οι κεφαλές).

Ο αλγόριθμος σίγουρα ελαχιστοποιεί το seek time και είναι εύκολα υλοποιήσιμος.

Το πρόβλημα του SSF έγκειται στο ότι δεν είναι "δίκαιος" και έτσι μπορεί να προκύψει starvation. π.χ. η κεφαλή βρίσκεται στη μέση του δίσκου και μια καινούργια αίτηση καταφθάνει για τον 1ο κύλινδρο. Μέχρι να εξυπηρετηθεί αυτή η αίτηση είναι πιθανό πάντα να καταφθάνει καινούργια αίτηση πιο κοντά στη μέση του δίσκου (έχει παρατηρηθεί ότι οι κεφαλές έχουν την τάση να παραμένουν πιο πολύ στους μεσαίους κυλίνδρους παρά στους ακραίους).

Ο Αλγόριθμος SCAN

Λέγεται και (ανελκυστήρας) elevator. Οι κεφαλές "σκουπίζουν" τους δίσκους:

- Αρχίζουν εξυπηρετώντας την αίτηση για τον πιο μικρό κύλινδρο
- συνεχίζουν να κινούνται στην ίδια κατεύθυνση εξυπηρετώντας την αίτηση για τον κύλινδρο πιο κοντά στη τωρινή θέση.
- Οταν εξυπηρετήσουν την τελευταία αίτηση, αλλάζουν κατεύθυνση, ακολουθώντας την ίδια μεθοδο.

Στην ουσία SCAN είναι ίδιος με τον SSF, μόνο που υπάρχει η έννοια της κατεύθυνσης.
➔ περιμένουμε (και έτσι είναι) ότι ο SCAN είναι λίγο χειρότερος από τον SSF.

Με μια μικρή τροποποίηση ο αλγόριθμος γίνεται circular SCAN (ή C-SCAN): Αντί ν' αλλάξει η κατεύθυνση, μετά το τέλος εξυπηρέτησης της τελευταίας αίτησης, αρχίζει το "σκούπισμα" πάλι από την αρχή (δηλ. οι κεφαλές επανατοποθετούνται στο μικρότερο κύλινδρο).

Το πλεονέκτημα του C-SCAN είναι ότι μειώνεται περαιτέρω το "variance" (η απόκλιση από τον μέσο όρο). Γιατί;

Περαιτέρω πιθανότητες βελτίωσης υπάρχουν όταν εκκρεμούν αιτήσεις για διαφορετικά sectors του ίδιου κυλίνδρου. Οταν ο controller εξάγει την πληροφορία για το ποιό sector περνά κάτω από την κεφαλή τότε ο driver εξυπηρετεί πρώτα την αίτηση για το sector που θα περάσει κάτω από την κεφαλή πρώτα.

Οταν υπάρχουν πολλά drives, όσον γίνεται κάποιο transfer, ο driver εκδίδει εντολές seek στ' άλλα drives ώστε να γίνουν όσο το δυνατόν περισσότερες λειτουργίες παράλληλα.

Δρομολόγηση με πολλές συσκευές

Οταν μια εντολή εγγραφής ή ανάκτησης τελειώσει, ο driver μπορεί να ελέγξει αν για κάποιο drive ισχύει ότι οι κεφαλές του είναι τοποθετημένες σε κύλινδρο για τον οποίο υπάρχει αίτηση στο Work Queue του drive και έτσι δρομολογείται αυτή η αίτηση αφού δεν χρειάζεται να γίνει seek [κατά πάσα πιθανότητα το seek έγινε παράλληλα με την προηγούμενη ανάκτηση/εγγραφή].

Πρέπει να σημειωθεί ότι οι πιο πολυ-υλοποιημένοι αλγόριθμοι βασίζονται σε παραλλαγές του SCAN. Δηλαδή γίνεται προσπάθεια ελαχιστοποίησης του seek time.

Αν η τεχνολογία δίσκων αλλάξει και το seek time παύσει να είναι το κυρίαρχο κόστος, τότε θα προκύψει η ανάγκη για καινούργιους αλγόριθμους για disk scheduling.

Κεφάλαιο 6. Αδιέξοδα (DEADLOCKS)

6.1 Μοντέλα Αδιεξόδων

Αδιέξοδο = ένα σύνολο από διεργασίες που δημιουργούν μια κυκλική αλυσίδα όπου κάθε process στην αλυσίδα δεν μπορεί να προχωρήσει και περιμένει για κάποιο γεγονός που μπορεί να προκληθεί μόνο από κάποιο άλλο μέλος της αλυσίδας.

Τα γεγονότα για τα οποία περιμένουν οι διεργασίες είναι η απελευθέρωση κάποιου resource.

Για να χρησιμοποιήσουν κάποιο πόρο οι διεργασίες πρέπει πρώτα να:

1. ζητήσουν τους πόρους: Αν δεν είναι διαθέσιμο (δηλ. κάποια άλλη διαδικασία το χρησιμοποιεί) τότε η διεργασία που το ζητάει μπλοκάρει.
2. Χρησιμοποιούν τον πόρο, αν είναι ελεύθερος
3. Απελευθερώνουν τον πόρο.

Αναγκαίες Συνθήκες για Αδιέξοδο

Οι επόμενες 4 συνθήκες πρέπει να ισχύουν για να δημιουργηθεί ένα deadlock:

- mutual exclusion (αμοιβαίος αποκλεισμός): Μόνο μια διεργασία μπορεί να χρησιμοποιεί ένα resource.
- hold & wait: Οι διεργασίες που συμμετέχουν στο αδιέξοδο πρέπει και να κατέχουν κάποιο resource αλλά και να περιμένουν για κάποιο resource.
- No preemption: Μόνο η κατέχουσα διεργασία μπορεί ν' απελευθερώσει το resource - δηλ. το resource δεν μπορεί ν' αφαιρεθεί από τη διεργασία.
- circular wait: κυκλική αλυσίδα >= *2* διεργασιών, όπου κάθε διεργασία περιμένει για ένα resource που το κατέχει η επόμενη διεργασία στην αλυσίδα.

Να σημειωθεί, ότι τα resources μπορεί να είναι λογισμικό και υλικό (π.χ. mutex, κλειδιά για αρχεία, ή δίσκοι, εκτυπωτές, κ.λπ).

Το πρόβλημα deadlock μοντελοποιείται ως εξής:

- Δημιουργείται ένας γράφος κατευθυνόμενος.
- Κόμβοι του γράφου είναι οι διεργασίες και οι πόροι.
- Η ακμή P ➔ R σημαίνει ότι το process P περιμένει για το resource R.
- Η ακμή R ➔ P σημαίνει ότι το process P κατέχει το resource R.
- Στο σύστημα υπάρχει deadlock εάν και μόνο εάν ο κατευθυνόμενος γράφος περιέχει ένα κύκλο!

Ετσι, το σύστημα μπορεί να χρησιμοποιεί ένα τέτοιο γράφο για να ανιχνεύει deadlock.

Υπάρχουν 4 γενικές στρατηγικές για την αντιμετώπιση του προβλήματος:

- Στρουθοκαμηλισμός: Κάνε τίποτα.
- Ανίχνευση και ανάνηψη (χρήση του γράφου).
- Αποφυγή deadlock (προσέχεις πότε δίνονται τα resources στις διεργασίες)
- Πρόληψη (σιγουρεύει ότι μια από τις 4 αναγκαίες συνθήκες δεν μπορεί να ισχύει).

Στρουθοκαμηλισμός:

Το πρόβλημα είναι ότι η αντιμετώπιση του προβλήματος κοστίζει ακριβά. Γι αυτό, πολλά Λ.Σ. επιλέγουν να μην αντιμετωπίζουν καθόλου το πρόβλημα - ούτε καν ανίχνευση!!!
Το UNIX ανήκει σ' αυτή την κατηγορία.

6.2 Ανίχνευση και Ανάνηψη

Η στρατηγική αυτή βασίζεται στη χρήση του κατευθυνόμενου γράφου που παρουσιάσαμε πριν, αφ' ένός. Αφ' ετέρου, βασίζεται στην ύπαρξη ενός αλγόριθμου που δεδομένου ενός κατευθυνόμενου γράφου, βρίσκει αν υπάρχει κύκλος.

Εφ' όσον ανιχνευθεί ένας κύκλος-deadlock, η ανάνηψη μπορεί να βασισθεί (και συνήθως βασίζεται) στον τερματισμό (killing) μερικών διεργασιών.

Η επιλογή της διεργασίας-θύματος: μπορεί να γίνει ανάμεσα στις διεργασίες που συμμετέχουν στο deadlock. (Τα κριτήρια που χρησιμοποιούνται συνήθως είναι, η ηλικία ή ο αριθμός resources που κατέχει, κ.λπ).

ή

μπορεί να γίνει ανάμεσα και σε άλλες διεργασίες π.χ. μπορεί να τερματισθεί μια "μεγάλη" διεργασία που κατέχει πολλά resources για τα οποία περιμένει κάποια διεργασία που είναι σε αδιέξοδο.

6.3 Αποφυγή Αδιεξόδων

Ας υποθέσουμε ότι για κάθε διεργασία, όταν αυτή αρχίζει, το σύστημα γνωρίζει τον μέγιστο αριθμό και τον τύπο κάθε resource που θα χρειαστεί η διεργασία.

Ο γνωστός Banker's algorithm μπορεί να χρησιμοποιηθεί τότε

- βασίζεται στην έννοια των safe states (ασφαλείς καταστάσεις). Ενα σύστημα βρίσκεται σ' ένα safe state αν:
    - δεν υπάρχει deadlock, και
    - υπάρχει τρόπος ικανοποίησης των αιτημάτων για resources με το να δρομολογήσεις τις διεργασίες (με κάποια σειρά).

<u>Παράδειγμα</u>:

	Έχει	Max
P1	3	9
P2	2	4
P3	2	7

Ελεύθερα resources: 3

Αυτό είναι ένα safe state γιατί:

- μπορούμε να δώσουμε 2 (απ' τα 3) ελεύθερα resources στο P2.
- όταν τελειώσει θάχουμε 5 ελεύθερα resources που τα δίνουμε στο P3 και όταν τελειώσει δίνουμε 6 resources στο P1 και έτσι όλα τελειώνουν χωρίς deadlock.

Αν τώρα το P1 είχε 4 (αντί για 3) resources το σύστημα δεν θα ήταν σε safe state. Γιατί;

Ετσι ο αλγόριθμος (Banker's algorithm) είναι:

- Για κάθε αίτηση για ένα resource:
    - Εξέτασε, αν δινόταν το resource, αν το σύστημα θα είναι σ' ένα safe state:
        - Αν ναι, τότε δώσε το resource
        - Αν όχι, τότε το resource δεν δίνεται
    - Για να ελεγχθεί αν το σύστημα είναι σ' ένα safe state:
        - εξετάζεται το: αν δοθούν τα ελεύθερα resources σε κάποιο process, τότε καλύπτονται οι ανάγκες του process;
        - Αν υπάρχει κάποιο τέτοιο process, τότε αυτό το process θεωρείται ότι τελείωσε (ελευθερώνοντας resources). Μετά, επαναλαμβάνεται η παραπάνω διαδικασία και αν όλα τα process θεωρηθούν τελειωμένα τότε το σύστημα είναι σε safe state. Αλλοιώς, το σύστημα είναι σε unsafe state.

Ο παραπάνω αλγόριθμος χρησιμοποιείται για συστήματα μ' ένα μόνο τύπο resources.
Ο αλγόριθμος μπορεί εύκολα να τροποποιηθεί για πολλούς τύπους resources. Αντί για μια ακέραια μεταβλητή για τα resources, υπάρχουν πίνακες:

- Has[N] όπου N είναι ο αριθμός τύπων resources δηλ. Has[i] για το process P δείχνει πόσα resources τύπου i το P κατέχει.

- Wants[N] που ορίζεται όμοια και εκφράζει πόσα resources κάθε τύπου χρειάζεται κάθε process.
- Free[N]: δηλώνει πόσα resources κάθε τύπου είναι ελεύθερα.

Το μεγαλύτερο πρόβλημα του Banker's algorithm είναι η υπόθεσή του: Είναι πολύ σπάνιο να ξέρουμε από πριν τις αιτήσεις κάθε process για τα resources που πρόκειται να εκδοθούν από τα process αργότερα!

6.4 Πρόληψη Αδιεξόδων

Εγγυώνται, αυτές οι στρατηγικές, ότι μια από τις 4 αναγκείες συνθήκες δεν μπορεί να συμβεί/ισχύσει.

- Mutual Exclusion: πολλά resources απαιτούν mutual exclusion (π.χ. ταινίες, εκτυπωτές) ➔ δεν μπορούμε σε γενικές γραμμές ν' αποφύγουμε mutual exclusion.
- Hold & Wait: αποφεύγεται αν στη αρχή των process γίνουν όλες οι αιτήσεις για όλα τα resources του process.

  Αλλά, όπως προείπαμε, αυτή η πληροφορία δεν υπάρχει. Επίσης preemption σε πολλά resources (π.χ. εκτυπωτές) preemption είναι αδύνατο.

- Circular Wait: Μπορεί ν' αποφευχθεί με τον εξής τρόπο:
  - σ' όλα τα resources δίνεται και ένας αύξοντας αριθμός.
  - όλα τα process αναγκάζονται να ζητούν τα resources με την σειρά που αντιστοιχεί στους αριθμούς των resources ➔ έτσι δεν μπορεί να προκύψει κυκλική αλυσίδα.

Μέρος 2ο

Κατανεμημένα Λειτουργικά Συστήματα

Κεφάλαιο 7. Εισαγωγή σε Κατανεμημένα Συστήματα

1. Γιατί ;
2. Τι ;
3. Δυσκολίες
4. Χρήσιμες Τεχνικές
5. Στόχοι
6. Αρχιτεκτονική
7. Προχωρημένες Εννοιες

7.1 Γιατί; (Κίνητρα)

Οι χρήστες βρίσκονται διεσπαρμένοι. Υπάρχει η επιθυμία να επικοινωνήσουν και να ανταλλάξουν πληροφορίες.

Πτωτική πορεία του κόστους επεξεργαστών και μνήμης. Κάθε χρόνο μπορούμε να αγοράσουμε όλο και πιό ισχυρούς σταθμούς εργασίας (Σ.Ε.) με τα ίδια χρήματα.

Καινούριες εφαρμογές καθιστούν τα παραδοσιακά συστήματα "απηρχειωμένα".

Επεκτασιμότητα: Για να αυξήσουμε την ισχύ ενός κατανεμημένου συστήματος, μπορούμε απλώς να προσθέσουμε ένα Σ.Ε. (π.χ. σαν File ή CPU Server) κάθε φορά.

Availability (***Διαθεσιμότητα***): Τι γίνεται όταν συμβουν βλάβες (π.χ. πέφτουν μηχανήματα); Η λύση είναι replication: δηλαδή να υπάρχουν πολλοί ίδιοι πόροι για υλικό και αντίγραφα για λογισμικό και δεδομένα. Όταν παρουσιάζεται βλάβη και κάποιος πόρος δεν είναι διαθέσιμος, χρησιμοποιείται το αντίγραφό του.

*Scalability (**Κλιμακωσιμότητα**)*: Η ικανότητα του συστήματος για υψηλή απόδοση ακόμα και όταν κλιμακωθούν οι απαιτήσεις (π.χ. από αύξηση του αριθμού χρηστών, δεδομένων, συστημάτων μελών, κ.λπ.)

Reliability (***αξιοπιστία***): Η ικανότητα του συστήματος να συμπεριφέρεται σωστά ακόμα και όταν υπάρχουν βλάβες.

7.2 Ορισμός (Τι είναι;)

Αναγνώριση: Πως αποφασίζουμε αν ένα σύστημα είναι πράγματι ένα "κατανεμημένο πληροφοριακό σύστημα"; Ποιά είναι τα κριτήρια; Ποιές οι προυποθέσεις; Αυτά τα ζητήματα είναι αμφιλεγόμενα στην ευρύτερη ερευνητική κοινότητα!

Ο Lamport είπε: "Το Κ.Σ. είναι ένα σύστημα που σε εμποδίζει να κάνεις δουλειά όταν κάποια μηχανή, που δεν έχεις καν ξανακούσει, παθαίνει βλάβη".

Οι Tanenbaum kai Van Renesse υποστηρίζουν ότι "Ενα Κ.Σ. φαίνεται στους χρήστες του σαν ένα κοινό, παραδοσιακό σύστημα με την διαφορά ότι τρέχει σε πολλά ανεξάρτητα CPUs".

Δύο έννοιες κλειδιά προκύπτουν από τις παραπάνω δηλώσεις:
Separation (***Διαχωρισμός***): Πολλά, διαφορετικά, CPUs, μνήμες, κ.λπ.

Transparency (***Διαφάνεια***): Παροχή της ψευδαίσθησης ότι υπάρχει μόνο μία μηχανή - αυτή που χρησιμοποιείται από τον χρήστη. Δηλαδή έχουμε ένα μηχανισμό αφαίρεσης (abstraction).

Η έννοια του διαχωρισμού αναφέρεται στο γεγονός ότι υπάρχουν πολλά διαφορετικά προγράμματα, στην ανάγκη απομόνωσης βλαβών στο εκάστοτε πρόγραμμα (ή έστω μηχάνημα) και στην δυνατότητα βαθμιαίας ανάπτυξης.
Η έννοια της διαφάνειας είναι σύνθετη και πιό περιεκτική:

- Access transparancy: Διαφάνεια πρόσβασης. Η επικοινωνία χρήστη-συστήματος πρέπει να είναι όμοια με αυτή του χρήστη με ένα συγκεντρωτικό σύστημα.
- Location transparancy: Διαφάνεια θέσης. Η συμπεριφορά του συστήματος πρέπει να είναι ανεξάρτητη της θέσης του χρήστη, των δεδομένων που προσπελαύνονται, των προγραμμάτων που χρησιμοποιούνται κ.λπ.

Υπάρχουν επίσης πολλές άλλες έννοιες διαφάνειας.

Οι έννοιες του διαχωρισμού και της διαφάνειας είναι απαραίτητες αλλά όχι ικανές συνθήκες για να καθορίσουν ένα σύστημα σαν Κ.Σ. Για παράδειγμα, με τις παραπάνω έννοιες δεν γίνεται κανένας διαχωρισμός μεταξύ ενός πολυ-επεξεργαστκού (παράλληλου) συστήματος και ενός γεωγραφικά κατανεμημένου συστήματος.

Είναι δύσκολο ίσως να επιχειρήσουμε έναν ακριβή ορισμό Κ.Σ. Μπορούμε, π.χ., να επιμείνουμε στην αρχή του διαχωρισμού και στην ανάγκη απομόνωσης βλαβών, δηλαδή αν μία βλάβη λάβει χώρα, τότε μόνο ένα μέρος του συστήματος υποκύπτει. Αλλά, αυτό θα απέκλειε Κ.Σ. σε LANs (σκεφτείτε τι θα συμβεί αν ένας κεραυνός διακόψει την παροχή ρεύματος σε ένα κτίριο στο οποίο βρίσκεται το LAN).

Ας ορίσουμε λοιπόν ένα σύνολο χαρακτηριστικών τα οποία θεωρούνται γενικώς σαν αναγκαία και ικανά για να ορίσουν ένα Κ.Σ.

1. Πολλοί ανεξάρτητοι κόμβοι (CPUs, μνήμες, ...

2. Σύστημα επικοινωνίας (comp. network) (λογισμικό και υλικό)

3. Ανεξάρτητες βλάβες κόμβων (σε μερικά πολυ-επεξεργαστικά συστήματα, όταν πέφτει ένας κόμβος, πέφτει όλο το σύστημα).

4. Shared State: Κοινή κατάσταση. Οι κόμβοι του συστήματος πρέπει να μοιράζονται την κατάσταση του συστήματος. Για παράδειγμα, ένα σύστημα αποτελούμενο από ένα κεντρικό File Server και ένα σύνολο από Σ.Ε. χωρίς δίσκους (diskless) δεν θεωρείται σαν ένα κατανεμημένο σύστημα. Αν ο file server πέσει, τότε όλο το σύστημα υποκύπτει σ' αυτή τη βλάβη. Αν όμως, οι Σ.Ε. είχαν δίσκους, τότε θα μπορούσαν, έστω θεωρητικά, να έχουν αντίγραφα των αρχείων και έτσι να συνεχιστεί η λειτουργεία του συστήματος. Αυτή την άποψη, πάντως, δεν την αποδέχονται όλοι.

5. Διαφάνεια: Access και Location Transparency.

7.3 Δυσκολίες στη Κατασκευή Κ.Σ.

1. Ενοποίηση Πόρων (Υλικό και λογισμικό).

   Παράδειγμα.
   Το πρωτόκολλο token ring χρησιμοποιείται σε πολλά δίκτυα υπολογιστών για να ισοκατανείμει το δίκτυο σ'όλους τους χρήστες. Λειτουργεί ως εξής: Ολοι οι κόμβοι είναι οργανωμένοι σε ένα δακτύλιο. Ενας κόμβος για να στείλει ένα πακέτο (μήνυμα) πρέπει πρώτα να παραλάβει ένα token (κουπόνι). Το κουπόνι στριφογυρίζει στο δακτύλιο, μέχρις να το χρειαστεί ο επόμενος κόμβος. Μόλις ένας κόμβος στείλει ένα πακέτο, αναγκάζεται να αφήσει το token ώστε να απαγορεύεται η μονοπώλησή του.

   Θεωρείστε τώρα δύο κόμβους Α και Β που θέλουν να στείλουν πακέτα στο κόμβο Γ. Ο Α στέλνει το πακέτο του, και μετά ο Β στέλνει το πακέτο του αμέσως μετά τον Α (δηλ. το τελευταίο bit του Α ακολουθείται από το πρώτο bit του Β). Σ'αυτή τη περίπτωση προέκυψε το εξής πρόβλημα: το κύκλωμα στο Γ που είναι υπεύθυνο να λαβαίνει πακέτα, δεν μπορούσε να αναγνωρίσει τα πακέτα του Β. Προσέξτε ότι και τα δύο συστήματα που ενοποιήθηκαν (το κύκλωμα παραλαβής πακέτων και το token ring πρωτόκολλο) είχαν τεσταριστεί με επιτυχία.

2. Κατανομή λαθών.

   Παράδειγμα.
   Στο (θρυλικό) ARPANET υπήρχε αρχικά ένα bug στον κώδικα που όριζε την διαδρομή που θα ακολουθήσει το κάθε πακέτο για να φτάσει στον προορισμό του. Αυτος ο κώδικας απαιτούσε από όλους τους κόμβους να ενημερώνουν τους γείτονές τους για τις αποστάσεις τους μέχρι τους άλλους κόμβους. Εξ αιτίας κάποιου bug κάποιος κόμβος "νόμισε" ότι η απόστασή του από όλους τους άλλους κόμβους είναι 0. Ετσι, όλοι οι άλλοι κόμβοι άρχισαν να στέλνουν τα πακέτα τους μέσω του λανθασμένου κόμβου, προκαλώντας το πέσιμο του συστήματος.

3. Scale (Κλίμακα). Ενα σύστημα με καλή απόδοση με 10 χρήστες, μπορεί να έχει άθλια απόδοση αν το σύστημα "μεγαλώσει" σε 100 χρήστες.

4. Μερική Βλάβη. Οι αλγόριθμοι που υλοποιούνται στο σύστημα πρέπει να μπορούν να ανεχθούν "μη ολοκληρωτικές" βλάβες. Για παράδειγμα, το σύστημα μπορεί να πέσει αφού έχουν εκτελεσθεί οι μισές εντολές ενός προγράμματος. (Σκεφτείτε τι μπορεί να συμβεί, αν αυτό το πρόγραμμα ήταν ένα πρόγραμμα που μεταφέρει χρήματα από ένα λογαριασμό σε ένα άλλον).

7.4 Χρήσιμες τεχνικές για την κατασκευή Κ.Σ.

1. Δημιούργησε αντίγραφα για υψηλή διαθεσιμότητα.

2. Χρησιμοποίησε προσωρινή μνήμη (cache) και υποδείξεις (hints). Αποθήκευσε σημαντικά data (π.χ. ενός server) τοπικά (στον δικό σου Σ.Ε.) εάν υπάρχει περίπτωση να ξαναχρησιμοποιηθούν. Αλλά, αντιμετώπισε τα cache data σαν υποδείξεις μόνο: δηλ. μπορεί οι τοπικές πληροφορίες να είναι παλιές-ανημέρωτες.

3. Stashing.
   Σαν το caching. Αλλά, οι τοπικά-αποθηκευμένες πληροφορίες χρησιμοποιούνται μόνο όταν ο server δεν είναι προσβάσιμος (π.χ. λόγω βλάβης).

4. Χρησιμοποίησε timeouts. Κυρίως για την ανίχνευση βλαβών (π.χ. όταν περιμένουμε μία απάντηση μόνο για εύλογο χρονικό διάστημα). Επίσης για να μην επιτρέπεται σε ένα κόμβο να μονοπωλεί πόρους.

5. Απόδειξη ορθότητας αλγορίθμων. Αλλά, αν δεν υλοποιήσουμε το σύστημά μας, δεν μπορούμε να είμαστε σίγουροι...

6. Στάνταρντ μηχανισμοί κλήσης "μακρινών" διεργασιών (π.χ. RPC). Να μην εξαναγκάζεται ο χρήστης να μαθαίνει πολλά διαφορετικά εργαλεία για την ίδια δουλειά.

7. Χρησιμοποίησε encryption πριν οι πληροφορίες μπουν στο δίκτυο.

8. Προστάτευσε τις πληροφορίες των χρηστών. Περιλαμβάνει έλεγχο δικαιώματος πρόσβασης σε δεδομένα (access control lists and capabilities.)

7.5 Στόχοι Σχεδίασης Κ.Σ.

- Διαφάνεια. (Access και Location Transparency).

- Συνέπεια (Consistency). Αναφέρεται στη προστασία (που παρέχεται από το σύστημα) από 1) βλάβες και 2) ταυτόχρονη πρόσβαση στα ίδια δεδομένα από διαφορετικούς χρήστες. Ακριβώς τι σημαίνει "συνέπεια" εξαρτάται από την εφαρμογή.

  <u>Παραδείγματα</u>.

  - Μερική βλάβη: Ενα πρόγραμμα μεταφοράς 50 εκατ. δρχ. από λογαριασμό Λ1 στο λογαριασμό Λ2 τρέχει και διακόπτεται (επειδή το σύστημα πέφτει) αφού έχει ήδη αφαιρέσει τα 50 εκατ. και πριν προλάβει να τα καταθέσει στον Λ2.

  - Ταυτόχρονη πρόσβαση: 2 προγράμματα Π1 και Π2 κάνουν ανάληψη 1 εκατ. από Λ1. Η διαδικασία που εκτελείται είναι: read(Y = υπόλοιπο του

Λ1); write(νέο υπόλοιπο, Y = Y - 1 εκατ.). Αν το Π1 και το Π2 πρώτα εκτελέσουν το read(), τότε, αν και 2 εκατ. θα έχουν αναληφθεί, στον υπολογιστή θα φαίνεται ότι μόνο 1 εκατ. αναλήφθηκε.

- Αποτελεσματικότητα.
    - Γρήγορη απόδοση.
    - Επεκτασιμότητα.
    - Καινούργιες υπηρεσίες, - openess.
    - Αξιοπιστία.

7.6 Αρχιτεκτονικά Μοντέλα

- Το WS Model. Ο κάθε χρήστης έχει το δικό του Σ.Ε. Αυτοί οι Σ.Ε. είναι διαφόρων ποιοτήτων και ισχύων. Υπάρχουν ισχυροί servers οι οποίοι προσφέρουν εξειδικευμένες υπηρεσίες (π.χ. file servers, printer servers, authentication servers, κ.λπ.). Τα προγράμματα των χρηστών τρέχουν στους δικούς τους Σ.Ε.
- Processor Pool Model (Δεξαμενή Κοινόχρηστων Επεξεργαστών). Οι χρήστες "ζητούν" (μέσω ενός τερματικού) από το σύστημα να τρέξει ένα πρόγραμμά τους. Ενας ειδικός server αναθέτει προγράμματα στους διάφορους επεξεργαστές. Η ανάθεση γίνεται με βάση τις απαιτήσεις του προγράμματος και την ισχύ των διαφόρων CPUs. Ενας άλλος server φορτώνει το πρόγραμμα στον συγκεκριμμένο CPU.
- Υβριδικό Μοντέλο. Κοινόχρηστοι επεξεργαστές μέσα σε ένα WS Μοντέλο. Για παράδειγμα, μεγάλα προγράμματα μπορούν να τρέχουν σε πολύ ισχυρούς CPU servers.

Τα πλεονεκτήματα των 2 παραπάνω μοντέλων είναι σημαντικά: Ενας χρήστης μπορεί να τρέχει πολλά προγράμματα παράλληλα σε διαφορετικές μηχανές. Επίσης, χρησιμοποιείται το κατάλληλο CPU για την κατάλληλη δουλειά.

Κεφάλαιο 8. Μοντέλα Διεργασιών και Επεξεργαστών

1. Νήματα Ελέγχου (Threads)
    - Εισαγωγή
    - Δόμηση πολύ-νηματικών (multi-threaded) διεργασιών
    - Ελεγχος σε ταυτόχρονη πρόσβαση
    - Πρόσβαση σε global variables
    - Υλοποίηση threads
    - Το Πακέτο (OSF-DCE Threads Package)

2. Μοντέλα Επεξεργαστών
    - Το Μοντέλο WS
    - Το Μοντέλο Processor Pool
    - Ανάθεση Processes σε Επεξεργαστές

8.1 Νήματα και Πολυνηματικές Διεργασίες

8.1.1 Εισαγωγή

Η παραδοσιακή έννοια του process: ένα address space και ένα thread of control (νήμα ελέγχου) που εκτελεί τις εντολές του προγράμματος.

Χρειαζόμαστε: πολλά threads of control, που μοιράζονται ένα address space, και που τρέχουν παράλληλα.

Στην ουσία αυτά τα πολλά threads όλα δικαιούνται πρόσβαση στον κώδικα και στα δεδομένα. Δηλαδή μπορούν να προσφέρουν ίδιες υπηρεσίες και να μοιράζονται πληροφορία (δεδομένα).

Παράδειγμα – File Server (FS): read (fd, &buf, count)

Οταν ο FS λαβαίνει την αίτηση read() εξετάζει πρώτα ένα αποταμιευτή (buffer cache -- BC) στην κύρια μνήμη. Αν η ζητούμενη πληροφορία είναι εκεί, την ανακτά και την επιστρέφει. Αλλοιώς, εκδίδει μια αίτηση στον δίσκο (disk controller) να φέρει την πληροφορία στο BC. Ο FS μπλοκάρει μέχρι ν'ανακτηθεί η πληροφορία από τον δίσκο και να τοποθετηθεί στο BC.

Οσο ο FS περιμένει για το disk I/O καμμιά άλλη αίτηση για file service δεν εξυπηρετείται➔ σπατάλη CPU cycles και κακή/αργή απόκριση στους πελάτες.

Η εστία του προβλήματος είναι: **ανασταλτικές κλήσεις (blocking system calls)** (για I/O, IPC, κ.λπ.). Blocking system calls είναι επιθυμητά !!! Απλοποιούν σημαντικά τον προγραμματισμό. Σκεφθείτε πως θα γράφατε έναν F.S. χωρίς blocking system calls.

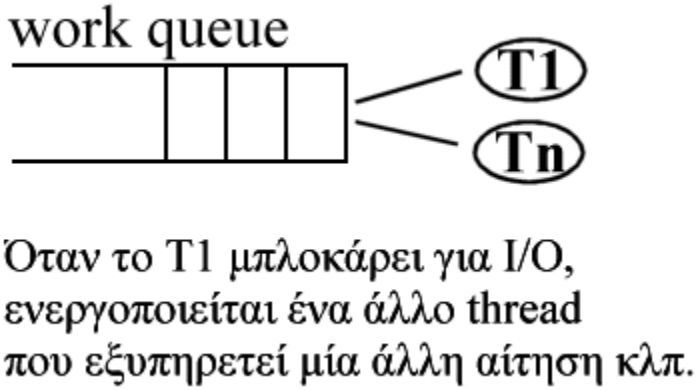
Στόχος: Συνέχιση παροχής υπηρεσιών ακόμα και με blocking system calls.

Μια άλλη εναλλακτική λύση θα μπορούσε να είναι η χρήση πολλών processors που να προσφέρουν το file service. Αυτά τα processes θα τρέχουν τον ίδιο κώδικα (τον F.S.). Το πρόβλημα έγκειται ακριβώς στον ορισμό του process, το κάθε process έχει το δικό του ξεχωριστό address space και έτσι αυτά τα process δεν μπορούν να έχουν ένα κοινό ΒC.

Αλλά, ακόμα και αν ξεπεράσουμε το παραπάνω πρόβλημα, επειδή το κάθε process έχει δικό του address space, context switching & process creation/termination θα είναι πολύ χρονοβόρες διαδικασίες.

Η σωστή λύση λοιπόν είναι multi-threaded processes!

Για το παραπάνω παράδειγμα, ο F.S. θα μπορούσε να δομηθεί ως εξής:

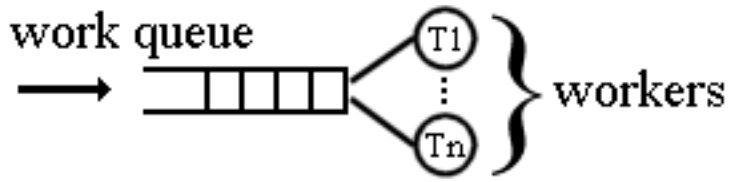


Γεγονός 1: Τα threads μοιράζονται address space => ένα thread μπορεί να "πειράξει" πληροφορία που χρειάζεται ένα άλλο thread. Επίσης, χρειάζεται έλεγχος ταυτόχρονης πρόσβασης (αλληλοεξαίρεση -- multual exclusion).

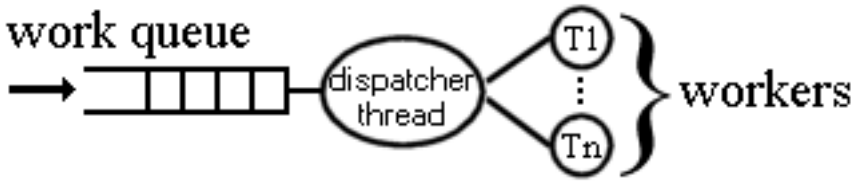
Γεγονός 2: Χρειαζόμαστε ένα scheduler (δρομολογητή) ο οποίος καλείται κάθε φορά που ένα thread μπλοκάρει και διαλέγει ένα "έτοιμο" thread να τρέξει. Αυτός ο scheduler δεν είναι ο ίδιος που δρομολογεί τα processes (όπως τον μάθατε στα ΛΣ) ;

8.1.2 Δομή Πολυνηματικών Διεργασιών

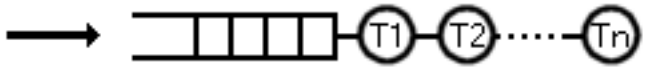
Το team μοντέλο: Ένας αριθμός ισόβαθμων threads που συνεργάζονται, λαβαίνοντας αιτήσεις από ένα work Q και εξυπηρετώντας αυτές.



Το dispatcher μοντέλο: Ένα thread διαχειρίζεται το work Q και διανέμει αιτήσεις στις κατάλληλες worker threads.



Το pipe μοντέλο: το κάθε thread επεξεργάζεται πληροφορίες και δίνει το output στο επόμενο thread για επεξεργασία κ.λπ.



Πότε χρησιμοποιούμε κάποιο μοντέλο;

Πόσους workers; [μπορεί να μεταβληθεί ο αριθμός τους δυναμικά].

Με ένα μεταβλητό αριθμό threads, μπορούμε να δημιουργούμε threads με κάθε αίτηση, για παράδειγμα. Αυτό το σχήμα είναι πιο ευέλικτο, αλλά λιγότερο αποδοτικό (κόστος create-thread (), terminate-thread ()).

8.1.3 Ταυτόχρονη Πρόσβαση σε Διαμοιραζόμενη Μνήμη

Τα εργαλεία που χρησιμοποιούνται είναι όμοια με αυτά για παραδοσιακά processes (semaphores, monitors, critical sections).

Ενα εργαλείο που χρησιμοποιείται σε threads packages είναι τα mutexes. Το mutex, στην ουσία του, είναι ένα binary semaphore.

Εντολές: lock (mutex), unlock (mutex), try lock (mutex).

Χρήση: Κάθε critical section συνδέεται με ένα mutex

- Πριν ένα thread μπεί σε ένα critical section, καλεί lock (mutex). Αν το lock του mutex τόχει πάρει κάποιο άλλο thread, τότε το πρώτο thread μπλοκάρει.

- Οταν ένα thread βγεί από το critical section τότε καλεί unlock (mutex). Έτσι αν κάποιο άλλο thread περιμένει, θα του δοθεί το lock και θα μπεί στο δικό του critical section.

Δηλαδή, μόνο ένα thread μπορεί να έχει το lock ενός mutex. Τι γίνεται όμως αν υπάρχουν πολλά threads που περιμένουν; Σε μερικές εφαρμογές θέλουμε να ξυπνήσουμε όλα αυτά τα threads. Σε άλλες μόνο ένα thread πρέπει να ξυπνήσει. Τα πιο πολλά threads packages προσφέρουν και τις δύο συμπεριφορές και ο προγραμματιστής της εφαρμογής επιλέγει.

Η εντολή trylock (mutex) είναι παρόμοια με την lock (mutex). Η διαφορά τους έγκειται στο ότι αν το lock δεν είναι διαθέσιμο (δηλ. κάποιο άλλο thread το έχει πάρει) τότε το thread δεν μπλοκάρει, αλλά συνεχίζει.

Ένα δεύτερο χρήσιμο εργαλείο είναι τα condition variables.
Χρησιμοποιούνται κυρίως για συγχρονισμό (δηλ. κάποιο thread περιμένει να συμβεί κάποιο γεγονός, για το οποίο ευθύνεται κάποιο άλλο thread). Για παράδειγμα, σκεφτείτε το πρόβλημα producer-consumer: Ενα "consumer" thread περιμένει για ένα buffer από ένα producer" thread.

Κάθε condition variable συνδέεται με ένα mutex.

Εντολές: wait(cond-var) και wakeup (cond-var).
Χρήση: Σαν έννοιες cond. vars και mutexes είναι όμοιες. Γενικά, το mutex χρησιμοποιείται για critical sections (δηλ. βραχυπρόθεσμο locking). Condition variables χρησιμοποιούνται για πρόσβαση σε πόρους resources - μακρυπρόθεσμο locking/waiting.

Κάθε condition variable συσχετίζεται με ένα lock. Όταν καλείται το wait(C) τότε το thread περιμένει αν κάποιο άλλο thread έχει το resource μέχρι αυτό το thread να καλέσει wakeup(C).

(***) Επίσης, αυτόματα όταν καλείται το wait(C), το mutex που συνδέεται με το C γίνεται unlocked.

Παράδειγμα:
Ένα thread T1 θέλει πρόσβαση σε ένα resource R το οποίο είναι "κλειδωμένο" από ένα άλλο thread.

T1	**T2**
lock(mutex);	lock(mutex);
while R is busy;	mark R free;
wait (c);	unlock(mutex);
mark R as busy;	wakeup (c);
unlock(mutex);	

Βλέπουμε ότι το R γίνεται "busy" και "free" μόνο στο critical section των threads το οποίο φυλάγεται με ένα mutex.

Τώρα δες (***) παραπάνω: χωρίς αυτή τη συμπεριφορά, θα είχαμε deadlock (αδιέξοδο). Προσοχή: το wakeup μπορεί να ξυπνήσει πολλά threads.

8.1.4 Πρόσβαση σε Global Μεταβλητές

Πρόβλημα: Υπάρχουν μερικές global vars που αφορούν στην διαχείριση διεργασιών συνολικά. Όταν υπάρχουν πολλά threads που διαβάζουν και αλλάζουν την τιμή ενός global var τότε προβλήματα μπορούν να προκύψουν.

π.χ. το *errno* στο Λειτουργικό Σύστημα UNIX: χρησιμοποιείται για την επιστροφή κωδικοποιημένης πληροφορίας που αφορά την επιτυχή ή μη εκτέλεση ενός system call.
Ας υποθέσουμε ότι το T1 καλεί το *access*() για να δεί αν δικαιούται πρόσβαση σε ένα αρχείο. Ο kernel επιστρέφει μια τιμή που αποθηκεύεται στην μεταβλητή errno. Πριν όμως το T1 αρχίσει, ο scheduler δίνει το CPU σε ένα άλλο thread T2 που καλεί ένα άλλο system call που αποτυγχάνει και ο kernel γράφει τον κωδικό αποτυχίας στο errno. Όταν το T1 αρχίσει πάλι θα συμπεριφερθεί λανθασμένα...

Η λύση που προτιμάται είναι η δημιουργία ενός καινούργιου scope: δηλαδή: per-thread global variables. Έτσι κάθε thread έχει το δικό του errno. Αλλά πως χρησιμοποιούμε/υλοποιούμε ένα τέτοιο καινούργιο scope; Οι πιο πολλές γλώσσες προγραμματισμού δεν υποστηρίζουν per-thread vars.

Μια λύση είναι να δημιουργηθούν library routines όπως: create-global(v), read-global (v), write-global(v), όπου v είναι ένα per-thread global var (ή μάλλον η διεύθυνσή του).
Για κάθε thread, διαφορετικές διευθύνσεις στο heap χρησιμοποιούνται για τα global var τους.

8.1.5 Υλοποίηση των Threads

Η υποστήριξη για threads μπορεί να υλοποιηθεί είτε στο kernel είτε στο user level.

Η θεμελιώδης ερώτηση εδώ είναι: Πρέπει ο kernel να έχει γνώση (και συνεπώς να διαχειρίζεται) threads ή πρέπει να ξέρει μόνο για παραδοσιακά processes (και να μην μπορεί να διακρίνει αν ένα process είναι multi-threaded);

Στην τελευταία περίπτωση τα threads υλοποιούνται στο user-level (δηλ. το λογισμικό που εκτελεί create(), delete(), και το scheduling για threads δεν είναι τμήμα του kernel).

Νήματα σε επίπεδο χρήστη (User-level Threads)

Το πρώτο βασικό πλεονέκτημα είναι ότι μπορούν να χρησιμοποιηθούν ακόμα και σε παραδοσιακά Λ.Σ. που δεν υποστηρίζουν threads.
Η λειτουργία τους συνοψίζεται ως εξής:

- Οταν ένα thread πρόκειται να κάνει κάτι που μπορεί να το μπλοκάρει (π.χ. recv_msg(), read(fd, ...), write(...)) τότε καλεί ένα ειδικό system call με το οποίο καθορίζεται αν το thread πρόκειται πράγματι να μπλοκάρει. Αν ΟΧΙ, τότε το thread συνεχίζει.
- Αν η απάντηση είναι ΝΑΙ, τότε το blocking system call του thread δεν καλείται. Το threads package καλεί τον Scheduler" ο οποίος βρίσκει ένα άλλο "έτοιμο" thread του ίδιου process και το τρέχει. Αυτός ο Scheduler" μιμείται τον scheduler του Λ.Σ. κρατώντας πληροφορίες για το stack, PC, SP, κ.λπ. register του κάθε thread.
- Το νέο thread τρέχει μόλις το PC, SP, και τα registers φορτωθούν.

Οσο υπάρχει κάποιο "έτοιμο" thread στο process και το time slice του process δεν έχει παρέλθει το threads πακέτο συνεχίζει να δρομολογεί threads απ' αυτό το process.

ΣΗΜΕΙΩΣΗ: οι πιο πολλές μηχανές έχουν instructions για "φόρτωμα" "ξεφόρτωμα" πολλών registers μαζί => thread switching είναι πολύ γρήγορο!!! (1 τάξη μεγέθους γρηγορότερο).

Αυτό είναι και το βασικότερο πλεονέκτημα των user-level threads (αποφέυγουν το κόστος των kernel traps).
Ενα άλλο σημαντικό πλεονέκτημα είναι ότι τώρα application processes (εφαρμογές) μπορούν να χρησιμοποιήσουν τον δικό τους scheduling αλγόριθμο.

Υλοποίηση σε επίπεδο πυρήνα (Kernel-level Threads)

Ο kernel του Λ.Σ. κρατάει πληροφορίες για κάθε thread ενός process, παρόμοιες μ' αυτές που κρατούσε παραδοσιακά για processes (δηλ. έχει δομές για κάθε thread όπου σημειώνονται οι τιμές των registers, PC, SP, stack, κ.λπ). Για κάθε multi-threaded process συνεχίζει να κρατάει πληροφορίες παρόμοιες με single-threaded processes, όπως pointers στα Page Tables, swap space αναθέσεις, κ.λπ.

Οταν ένα thread μπλοκάρει, ο kernel μπορεί να επιλέξει ένα άλλο threrad από το ίδιο ή άλλο process (σε αντίθεση με το Scheduler" για user-level threads).
Οι λόγοι για τους οποίους user-level threads συγκριτικά μειονεκτούν είναι:

- Υλοποίηση blocking system calls
- Ισοκατανομή χρόνου scheduling στα threads ενός process.

Blocking system Calls

Με kernel-level threads επιτρέπεται να κληθουν blocking system calls. Μετά το kernel trap το Λ.Σ. αρχίζει ένα καινούργιο thread. Στα user-level threads αυτό δεν μπορεί να γίνει. Διότι τότε ο kernel θα μπλόκαρε όλο το (multi-threaded) process και έτσι θα έτρεχε κάποιο άλλο thread από κάποιο άλλο process ➔ κανένα όφελος από multi-threading!!!

Μια λύση είναι ν' αλλάξουν όλα τα blocking system calls σε non-blocking. Αλλά αυτό απαιτεί αλλαγή του Λ.Σ.

Μια άλλη λύση εστιάζει στην χρησιμοποίηση «jacket» κώδικα γύρω από system calls που μπορούν να μπλοκάρουν. Για π.χ. στο UNIX υπάρχει το SELECT system call το οποίο σε πολλές περιπτώσεις προειδοποιεί αν ένα system call (e.g. read από pipe) θα μπλοκάρει. Οπότε ο jacket κώδικας καλεί το SELECT και αν δεν πρόκειται να μπλοκάρει τότε καλεί read(), αλλιώς ο scheduler" καλείται να τρέξει ένα άλλο thread. Κάθε φορά που ξανακαλείται ο scheduler" εξετάζει αν κάποιο από τα SELECT έχει επιστρέψει την ένδειξη ότι πλέον δεν θα μπλοκάρει, και έτσι εκτελείται τότε το read().

Αυτή η λύση απαιτεί αλλαγή στο system call library και δεν είναι υλοποιήσιμη απ' όλα τα Λ.Σ. (π.χ. μπορεί να μην παρέχεται το SELECT ή αντίστοιχο).

Δίκαιο scheduling

Ο scheduler" των user-level threads τρέχει μόνο όταν κάποιο thread τον καλέσει. Έτσι, είναι δυνατόν να μονοπωληθεί το CPU από ένα thread. Με kernel threads ο scheduler καλείται είτε από το clock interrupt handler είτε από το block(). Έτσι, το πρόβλημα δεν υφίσταται με kernel threads.

Συμπέρασμα:

Τα threads χρειάζονται ακριβώς σε εφαρμογές που καλούν blocking system calls (π.χ. FSs). Όταν όλα τα threads ενός process συνέχεια καλούν system calls τότε όλα τα threads θα προξενήσουν kernel traps. Αφού γίνει το kernel trap, το επιπλέον κόστος για thread switching είναι αμελητέο ➔ δεν ισχύει το μεγαλύτερο πλεονέκτημα για user-level threads.

Εφαρμογές που είναι CPU-bound & non-blocking δεν χρειάζονται threads. (δηλ. user-level threads είναι προτιμότερο για εφαρμογές που δεν χρειάζονται threads καθόλου...)!!!

8.1.6 Το πακέτο νημάτων της OSF/DCE

Πάνω από 50 system calls για threads, χωρισμένα σε 7 κατηγορίες:

1. Thread management: create, exit, join, detach.
   Το join αφορά στη συμπεριφορά του πατέρα που περιμένει ή όχι για το παιδί-thread (όμοιο με το wait() του UNIX). Το detach το χρησιμοποιεί ο πατέρας-thread για να δηλώσει ότι δεν τον ενδιαφέρει το παιδί-thread. [Τα resources του παιδιού χρησιμοποιούνται αμέσως μόλις καλέσει exit()].

2. Template management: για τη διαχείριση threads, mutexes και cond.vars. Χρησιμοποιώντας κυρίως για να γίνει ευκολότερη η χρήση των threads, cond.vars, mutexes. Τα templates δίνουν αρχικές τιμές (όταν δημιουργούνται threads, cond.vars, mutexes) στις περισσότερους παραμέτρους. Έτσι ο χρήστης αλλάζει μόνο λίγες παραμέτρους.

3. mutexes: create, destroy, lock, unlock, trylock.
   Fast mutex & friendly mutex: μπορείς να κάνεις lock ένα mutex που ήδη έχεις κάνει lock;

4. Condition Variables: create, destroy, wait, wakeup, broadcast. Broadcast ξυπνάει όλα τα threads που περιμένουν.

5. Per-Thread Global Vars:
    a. create a global var γι αυτό το thread (αναθέτει μνήμη)
    b. getspecific: read: διάβασε ένα global var.
    c. setspecific: write: γράψε ένα global var.

6. Ακύρωση (Cancel) threads: "σκότωσε" cancel ένα άλλο thread. SetCancel: ένα thread ορίζει ότι επιτρέπει ή όχι να σκοτωθεί από ένα άλλο thread.

7. Δρομολόγηση (Scheduling): Το πακέτο επιτρέπει να ορισθεί ο αλγόριθμος δρομολόγησης για τα threads ενός process.

8.2 Μοντέλα Επεξεργαστών

Πώς οργανώνεται το σύστημα (με τα CPU του);

Τρία βασικά μοντέλα: WS model, Processor Pool, Hybrid

8.2.1 Το μοντέλο σταθμών εργασίας (WS Model)

Το Κ.Σ. είναι μια σειρά από Σ.Ε. συνδεδεμένοι με ένα δίκτυο (π.χ. LAN). Ο κάθε Σ.Ε. "ανήκει" σε κάποιον χρήστη και μπορεί να είναι με ή χωρίς δίσκο (diskless).

Αν οι ΣΕ είναι χωρίς δίσκους, τότε δημιουργούνται File Servers που κρατούν τα αρχεία των χρηστών. Οι αιτήσεις κατευθύνονται μέσω του δικτύου στους FSs.

Πλεονεκτήματα των Σ.Ε. χωρίς δίσκους

- (Diskless WSs) είναι πιο φθηνά.
- Η συντήρηση του συστήματος (admin) είναι πιο εύκολο (εγκατάσταση καινούργιου SW) μόνο στους FSs, backup, διαχείριση h/w, κ.λπ.).
- "συμμετρία" πρόσβασης: χρήστες μπορούν να κάνουν login από οποιοδήποτε WS.

WSs με δίσκους χρησιμοποιούνται ποικιλοτρόπως:

- paging/swapping και temp files (π.χ. compilers, editors, email, ...)

- Caching : πολυ-χρησιμοποιημένα αρχεία τοποθετούνται στον τοπικό δίσκο ➔ πιο γρήγορη πρόσβαση + λίγότερο φορτίο στο δίκτυο. Όμως η συνέπεια των αρχείων που έχουν γίνει cache σε διαφορετικά WSs είναι σημαντικό & δύσκολο πρόβλημα.

- τοπικό F.S.: αυτονομία, λιγότερο φορτίο στο δίκτυο. Ο διαμοιρασμός αρχείων όμως γίνεται πιο δύσκολος.

Πλεονεκτήματα του WS Model: απλό μοντέλο, σταθερός χρόνος απόκρισης, αυτονομία,

Μειονεκτήματα του WS Model: Τί γίνεται όταν #CPUs > #χρηστών; ➔ Ανενεργοί Σ.Ε. (Idle WSs) => σπατάλη CPU.

Ανενεργοί Σ.Ε.

Το πρόβλημα αντιμετωπίζεται με ένα μηχανισμό ο οποίος ενημερώνεται για ανενεργούς Σ.Ε. και "στέλνει" processes να εκτελεστούν σ' αυτά.
Πως μπορούμε να δούμε αν ένα WS είναι idle; #χρηστών logged-in ; ΟΧΙ!

... γιατί οι χρήστες "μπορεί νάχουν πάει ... για καφέ..".
Επίσης, ακόμα και χωρίς κανένα χρήστη ένα CPU μπορεί νάχει πολύ δουλειά.
Συνήθως: αν κάποιος έχει "αγγίξει" το ποντίκι ή το πληκτρολόγιο του τα τελευταία λεπτά, τότε δεν είναι ανενεργό.

Πως βρίσκουμε ανενεργούς Σ.Ε.;

<u>Ενας αλγόριθμος:</u>

- Υπάρχει ένα DB το οποίο σημειώνει τους ανενεργούς Σ.Ε.

- Όταν ένα ΣΕ γίνει ανενεργό, καλεί ένα ειδικό πρόγραμμα, το οποίο τοποθετεί αυτόν τον ΣΕ στο DB των ανενεργών ΣΕ.

- Όταν ένας χρήστης σ' ένα άλλο ΣΕ χρειαστεί CPUs, τότε καλεί ένα άλλο πρόγραμμα το οποίο τον πληροφορεί για τους ανενεργούς ΣΕ.

Πως μπορούμε να τρέξουμε ένα process σε ένα άλλο ΣΕ;

- το εύκολο: να στείλουμε τον κώδικα.
- το δύσκολο: να δημιουργήσουμε το ίδιο "περιβάλλον"

- ίδια "θέα" του FS, ίδιο working directory, shell vars, ...
- που θα τρέξουν τα system calls;

  π.χ.
  - read (fd, ...) πρέπει να τρέξει στο αρχικό WS ("σπίτι") αν αυτό έχει το FS του χρήστη, αλλιώς στο FS.
  - sys. calls για την οθόνη πρέπει να σταλούν "σπίτι".
  - sys. calls που αφορούν χρόνο είναι πρόβλημα διότι τα ρολόγια διαφορετικών WSs δεν μπορούν να είναι πλήρως συγχρονισμένα (βλεπε προβλήματα που προκύπτουν σε εφαρμογές όπως το make)
  - NICE πρέπει να εκτελεστεί στο καινούργιο CPU

➔ πολλές δυσκολίες ...

8.2.2 Το Μοντέλο Δεξαμενής Επεξεργαστών (Processor Pool)

Η βασική ιδέα είναι κοινοκτημοσύνη των υπολογιστικών πόρων του Κ.Σ. Υπάρχει ένα ειδικό μέρος μ' όλα τα CPUs. Οι χρήστες μέσω xterminals (ή άλλων τερματικών) ζητούν από το Κ.Σ. ένα CPU για το πρόγραμμά τους.
Από τη φύση της αυτή η λύση αποφεύγει τα προβλήματα των ανενεργών ΣΕ.

Μια θεωρητική προσέγγιση: Όταν ένα σύστημα έχει μέσο ρυθμό αφίξεως λ και μέσο ρυθμό εξυπηρέτησης μ τότε ο μέσος χρόνος απόκρισης Τ είναι:

$T = [1/(\mu\lambda)]$ [1]

π.χ. λ=40 αιτήσεις/sec, μ= 50 αιτήσεις/sec ➔ T= 1/10 sec.

Αν έχουμε Ν συστήματα το καθένα με λ,μ τότε για κάθε αίτηση το Τ δίνεται από την [1].
Αν όμως, δημιουργήσουμε ένα "υπέρ-σύστημα" με ρυθμό αφίξεως $\lambda' = N \times \lambda$ και ρυθμό εξυπηρέτησης $\mu' = N \times \mu$ τότε ο μέσος χρόνος απόκρισης σ' αυτό το σύστημα Τ' είναι:

$T' = [1/(\mu' \lambda')] = [1/(N\mu \ N\lambda)] = [1/N] \times [1/(\mu\lambda)] = [T/N]$

➔ αντικαθιστώντας Ν "μικρούς" υπολογιστικούς πόρους με 1 "μεγάλο" που είναι Ν φορές ισχυρότερος από τους μικρούς, ο μέσος χρόνος απόκρισης γίνεται 1/Ν του χρόνου απόκρισης των Ν μικρών συστημάτων.

Όμως υπάρχουν και άλλοι σημαντικοί παράγοντες (ο χρόνος απόκρισης ίσως δεν είναι ο πιο σημαντικός). Οπως: αξιοπιστία και υψηλή διαθεσιμότητα, price/performance.

Επίσης, μία δεξαμενή από Ν CPUs δεν ισοδυναμεί με έναν CPU που είναι Ν φορές ισχυρότερος (διότι ίσως να μην μπορεί να χρησιμοποιηθούν όλοι οι CPUs ταυτόχρονα -- αν π.χ. μία δουλειά δεν μπορεί να "σπάσει" σε Ν κομμάτια).

Το μοντέλο "processor pool" χρησιμοποιείται για περιβάλλοντα S/W development.

8.3 Ανάθεση CPU σε Processes

Το πρόβλημα προκύπτει και στα 2 προηγούμενα μοντέλα.

«Non-migratory» λύσεις που δεν επιτρέπουν την μεταφορά διεργασιών, κάνουν μόνο μια ανάθεση για 1 process. Μόλις αρχίσει η εκτέλεσή του, το process δεν ξαναμεταφέρεται.

Οι «migratory» λύσεις επιτρέπουν την μεταφορά ενός process ακόμα και μετά το ξεκίνημα εκτέλεσής του.

Πρέπει να καθοριστούν οι στόχοι:

- μεγιστοποίηση της χρησιμοποίησης του CPU (utilization)
- ελαχιστοποίηση χρόνου απόκρισης (response time), κ.λπ.

Πρέπει να ληφθούν υπ' όψη: η ισχύς του κάθε CPU, ο φόρτος των CPU, ο αλγόριθμος δρομολόγησης σε κάθε σύστημα, κ.λπ.

π.χ.	CPU1: 100 MIPS	CPU2: 1 K MIPS
	ουρά: άδεια	ουρά: 10 secs αναμονή

process A: 10M instructions, process B: 100M instructions
Η ανάθεση: A→CPU1, B→CPU2: 1/10 sec + 10.1 sec = 10.2 sec
Η ανάθεση: A→CPU2, B→CPU1: 1 sec + 10.01 = 11.01 sec

8.3.1 Θέματα σχεδίασης και υλοποίησης

Κεντρικοποίηση ή κατανομή (Centralized vs Distributed)

Μαζεύοντας όλες τις απαραίτητες πληροφορίες σ' ένα κεντρικό κόμβο, οδηγεί σε καλύτερες αποφάσεις.

Αλλά ο κεντρικός κόμβος μπορεί να γίνει σημείο συμφόρησης (bottleneck) και υπάρχουν προβλήματα διαθεσιμότητας.

Πολιτική Μεταφοράς (Transfer Policy)

Πότε αποφασίζουμε να "μετακομίσουμε" ένα process; Βασιζόμαστε μόνο σε τοπικές πληροφορίες; (ευκολία αντί απόδοσης).

Πολιτική Χωροθέτησης (Location Policy)

Πού το στέλνουμε το process;
Χρειαζόμαστε πληροφορίες για την κατάσταση όλων των υποψήφιων κόμβων-δεκτών.

Μέτρηση Φόρτου Εργασίας

- ΟΧΙ: #χρηστών logged-in.
- ΟΧΙ: μέτρηση Running & Runnable processes
- Ισως η καλύτερη λύση είναι η μέτρηση του "λόγου απασχόλησης" του CPU. Δηλαδή, περιοδικά εξέτασε αν ο CPU είναι ανενεργός. Μετά από πολλές μετρήσεις έχεις μια ένδειξη αν ο CPU υποαπασχολείται.

Το πρόβλημα με την τελευταία λύση είναι ότι πολλές φορές το Λ.Σ. απενεργοποιεί interrupts (disable). Έτσι η παραπάνω ένδειξη είναι υπο-εκτίμηση.

Λάβε υπ' όψη το κόστος της μεταφοράς

Αν το αναμενόμενο όφελος από μια μετακίνηση είναι σχετικά μικρό (π.χ. < 20%) ίσως νάναι καλύτερα να μην μετακινήσουμε το process.

Κεφάλαιο 9. Επικοινωνία σε Κατανεμημένα Συστήματα

9.1 Εισαγωγή

9.1.1 Η φύση της επικοινωνίας/διάδρασης μεταξύ διεργασιών

Η προσέγγιση I/O: το επικοινωνιακό σύστημα με το δίκτυο είναι σαν οποιοδήποτε άλλο επικοινωνιακό σχήμα για I/O ➔ "file model" για input/output: δηλαδή άνοιξε (open) μια σύνδεση, read & write από/και στη σύνδεση, και close() ...

Η διαδικαστική προσέγγιση (procedural approach): Επίσης αναφέρεται σαν το **request/reply μοντέλο**. Ο client στέλνει ένα request μήνυμα και ο server απαντά στλενοντας ένα reply μήνυμα.

Οριζόντια και κάθετη διάδραση:

Το s/w ενός συστήματος έχει μια πολυεπίπεδη δομή (π.χ. εφαρμογές σε ανώτερα επίπεδα, Λ.Σ. στα κατώτερα ...)

Οριζόντια: μεταξύ 2 επιπέδων σε 2 διαφορετικές μηχανές
Κάθετη: μεταξύ 2 επιπέδων στην ίδια μηχανή

Το μοντέλο request/reply είναι κατάλληλο και για τα δύο αυτά είδη διάδρασης ➔ ο ρόλος του πελάτη ή του εξυπηρετητή αναλαμβάνεται δυναμικά (δηλ. ένα process πρέπει να μπορεί να παίζει και τους δύο ρόλους)Είναι όμως πανάκεια; ΟΧΙ!

Παράδειγμα 1: Συστήματα πραγματικού χρόνου (real-time). Υπάρχουν διεργασίες sensors & διεργασίες reactors. Ενα sensor λαμβάνει μετρήσεις και τις στέλνει σε έναν reactor.
Δεν απαιτείται ούτε ACK ούτε απάντηση από τον reactor. [Αν το μήνυμα χαθεί, αντί να σταλεί πάλι, στέλνεται ένα καινούργιο μήνυμα με τις καινούργιες μετρήσεις].

Παράδειγμα 2: Οταν τα μηνύματα μεταφέρουν "φωνή", όπως σε μια τηλεφωνική συνδιάλεξη, ένα χαμένο πακέτο δεν έχει σοβαρή αρνητική επίδραση στην ποιότητα της επικοινωνίας = > δεν χρειάζεται ούτε ACK, ούτε reply.

9.1.2 Μέγεθος μεταφερομένων πληροφοριών

Στέλνονται μεγάλα ή μικρά μηνύματα;

Γιατί είναι σημαντική αυτή η ερώτηση; Συνήθως μπορούμε να βελτιστοποιήσουμε το επικοινωνιακό λογισμικό αν ξέρουμε την απάντηση. Κοίτα το μέγεθος των αρχείων στο File System (εκεί καταλήγουν τα μηνύματα και από εκεί απορρέουν). ➔ πρωτόκολλα πρέπει να εστιάσουν σε μικρά μηνύματα;

Το ποσοστό των αρχείων που είναι < 2K είναι < 5%

➔ Πρωτόκολλα πρέπει νάχουν καλή απόδοση με μεγάλα και μικρά μηνύματα. Μερικοί πιστεύουν ότι χρειάζονται διαφορετικά πρωτόκολλα για μικρά και για μεγάλα μηνύματα.

9.1.3 Μοντέλα επικοινωνίας

Η θεμελιώδης ερώτηση εδώ αφορά στο πως ένα μήνυμα φθάνει στο προορισμό του. Τρία μοντέλα: **packet, message, & circuit switching**. Packet: μονάδα πληροφορίας: το κάθε μήνυμα διαιρείται σε packets (στο ethernet ένα packet = 1.5 Kbytes).

<u>Packet switching:</u> Ο αποστολέας "δίνει" στο δίκτυο μια σειρά από packets (με σταθερό μεγιστο μέγεθος). Το κάθε πακέτο στέλνεται ανεξάρτητα των άλλων πακέτων του ίδιου μηνύματος ➔ ακολυθεί διαφορετικά "μονοπάτια" μπορεί να χαθεί, μπορεί να φθάσει "εκτός σειράς", ... κ.λπ.

<u>Message switching:</u> Ο αποστολέας "δίνει" στο δίκτυο ένα ολόκληρο μήνυμα. Το μήνυμα στέλνεται σαν μια μονάδα (αν και μπορεί να διαιρεθεί σε πακέτα από το δίκτυο πριν σταλεί). Κάθε ενδιάμεσος κόμβος συλλέγει όλα τα πακέτα του μηνύματος πριν στείλει στον επόμενο κόμβο κάποιο πακέτο αυτού του μηνύματος.

<u>Circuit switching:</u> πρώτα επιτυγχάνεται μια σύνδεση (μεταξύ του κόμβου-αποστολέα και του κόμβου-αποδέκτη) πριν σταλεί το πρωτο πακέτο. Επιτυγχάνεται συνήθως μέσω ενός ειδικού μηνύματος το οποίο κάνει κρατήσεις των απαραίτητων πόρων σε κάθε ενδιάμεσο κόμβο.
Κατόπιν, όλα τα πακέτα ακολουθούν την ίδια διαδρομή και καταφθάνουν με τη σωστή σειρά.

Το πιο σημαντικό κριτήριο: **end-to-end delay** (απόκριση/καθυστέρηση)

9.1.4 Σύνοδοι (Sessions)

Οι οντότητες που επικοινωνούν πρέπει να συνδέονται με έναν τρόπο που να διασφαλίζει την κατάλληλη επικοινωνία:

naming: πως βρίσκεις το όνομα του εξυπηρετητή;

authentication: βεβαιώσου ότι αυτός με τον οποίο επικοινωνείς είναι όντως αυτός που ισχυρίζεται.

Συχνά, κάθε μακρινή εντολή ("remote operation") αποτελεί ένα session. Η κατάσταση ενός session συνήθως αποθηκεύεται προσωρινά (**cached)** ούτως ώστε δεν πληρώνουμε κάθε φορά το κόστος για naming & authentication.

Συνήθως το πρόβλημα της ονομασίας λύνεται δίνοντας ονόματα στις διάφορες υπηρεσιες (π.χ. file service, disk service, ...). Ο κάθε χρήστης, έτσι δεν χρειάζεται να γνωρίζει πως έχει υλοποιηθεί μια υπηρεσία (πόσοι servers, πού, διευθύνσεις, κ.λπ.).

Το πρόβλημα της αυθεντικοποίησης συνήθως λύνεται με ένα **κλειδί** για κάθε σύνοδο. Το κλειδί είναι γνωστό μόνο στους πραγματικούς χρήστες/processes και χρησιμοποιείται για την **κρυπτογράφηση /αποκρυπτογράφηση.**

9.1.5 Ανασταλόμενα και μη (Blocking & non-blocking) Πρωτόκολλα

Αναφέρεται στην αποστολή και στην λήψη μηνυμάτων. Υπάρχουν ποικίλες ερμηνείες για το τι σημαίνουν αυτές οι έννοιες.
Το βασικότερο πλεονέκτημα των non-blocking (μη ανασταλτικών) πρωτοκόλλων είναι ο **παραλληλισμός**: ο αποστολέας για παράδειγμα συνεχίζει τη δουλειά του, παράλληλα με την αποστολή του μηνύματός του από το δίκτυο.

Το βασικότερο πλεονέκτημα των ανασταλτικών πρωτόκολλων είναι ότι προκύπτουν ευκολότερα προγράμματα.

9.1.6 Το ISO/OSI πρότυπο

Ενα πρότυπο-μοντέλο αναφοράς για την δημιουργία συστημάτων επικοινωνιών. Αποτελείται από 7 επίπεδα. Το κάθε επίπεδο προσφέρει εξειδικευμένες υπηρεσίες στο αμέσως επόμενο επίπεδο, μέσω ενός interface, και των διαδικασιών που το υποστηρίζουν. Συστήματα με πολυ-επίπεδη οργάνωση έχουν πολλά πλεονεκτήματα - απορρέουν κυρίως από την έννοια του **modularization** και του **abstraction**. Το Open Systems Interconnect Μοντέλο έχει τα εξής 7 επίπεδα:

Physical Level: Οι υπηρεσίες του αφορούν στην αποστολή των bits στις επικοινωνιακές γραμμές. Ιδιαίτερα: πόσα volts είναι το 1 και το 0, πόσα bits/sec μπορούν να σταλούν, διπλής ή μονής κατεύθυνσης επικοινωνία...

Data Link Level: Το 1ο επίπεδο αρκεί αν δεν υπάρχει η πιθανότητα λάθους. (π.χ. στέλνω "1" και λαμβάνει "0"). Ο μηχανισμός που ασχολείται με την ανίχνευση και αντιμετώπιση λαθών είναι το κύριο μέρος του 2ου επιπέδου. Τα bits ομαδοποιούνται σε frames και το data link layer διασφαλίζει ότι το κάθε frame καταφθάνει "σώο και αβλαβές".

Συνήθως, χρησιμοποιείται ένα checksum (ένας τύπος αθροίσματος των bits στο frame)

το οποίο στέλνεται μαζί με το msg. Οταν ο αποδέκτης data link layer λάβει το μήνυμα ξαναυπολογίζει το checksun και το συγκρίνει με αυτό στο μήνυμα. Αν είναι διαφορετικά, τότε κάτι δεν πήγε καλά. Επίσης, ACK msgs χρησιμοποιούνται για αντιμετώπιση "χαμένων" μηνυμάτων/frames. Αν το ACK για ένα frame δεν ελήφθη, τότε το frame ξαναστέλνεται.

Network Level: Ασχολείται κυρίως με τα προβλήματα **routing & congestion control**. Δηλαδή ασχολείται κυρίως με το να διαλέγει την καλύτερη διαδρομή που θα ακολουθήσει ένα msg για να φθάσεει στον προορισμό του. Congestion control είναι ένας έλεγχος ούτως ώστε να μην πολυ-χρησιμοποιούνται μερικές γραμμές, δημιουργώντας bottlenecks, ενώ θα μπορούσαν να χρησιμοποιηθούν άλλες.

Transport Level: Ασχολείται κυρίως με την αξιόπιστη μεταφορά των πακέτων από τον αποστολέα στον αποδέκτη. Δημιουργεί δηλαδή αξιόπιστες συνδέσεις (**reliable connection**). Τα προβλήματα κυρίως είναι: χαμένα πακέτα, αλλοιωμένα πακέτα, και πακέτα "εκτός σειράς". Δημιουργεί τη σύνδεση **end-to-end**. Μπορεί επίσης να διαλέξει και την κατάλληλη υπηρεσία (π.χ. **point-to-point, broadcasting, reliability, ...**). Από το transport layer και πάνω, τα πρωτόκολλα εμπλέκουν το κόμβο-αποστολέα με το κόμβο-αποδέκτη. Τα επίπεδα 1-3 εμπλέκουν έναν κόμβο με τον αμέσως επόμενο στη διαδρομή που ακολουθεί το πακέτο.

Session Level: Συμπληρώνει το transport layer. π.χ. διορθώνει σπασμένες συνδέσεις - μπορεί να χρησιμοποιεί ckeckpoints για ανάνηψη από crash. Διαλέγει επίσης το κατάλληλο transport layer πρωτόκολλο. Δεν χρησιμοποιείται συχνά!

Presentation Level: Τι σημαίνουν τα bits (π.χ. data encoding (ASCII, EBCDIC για chars) (1's ή 2' s complement για ints). Επίσης, **data compression & data encryption/decryption**.

Application Level: Μια συλλογή από πρωτόκολλα που προσφέρουν πολυ-ζητημένες υπηρεσίες, π.χ. e-mail, ftp, κ.λπ.

Το OSI μοντέλο είναι πολύ heavy-weight - δηλ. κοστίζει πολύ για να χρησιμοποιηθεί για πολλά Κ.Σ. Το μεγάλο κόστος προκύπτει από τα πολλά επίπεδα: δύο αντίστοιχα επίπεδα (ένα στον αποστολέα κια έναν στον αποδέκτη) επικοινωνούν μέσω ενός header.

Οπως το msg περνάει από το ένα επίπεδο στο αμέσως κατώτερο επίπεδο στον αποστολέα, το κάθε επίπεδο βάζει το δικό του header στο μήνυμα που στέλνεται. Αντίστοιχα, όπως το μήνυμα "ανεβαίνει" επίπεδα στον αποδέκτη, το κάθε επίπεδο βγάζει το κατάλληλο header από το μήνυμα πριν το περάσει στο ανώτερο επίπεδο. Η "επικοινωνία" των πρωτοκόλλων στο ίδιο επίπεδο γίνεται μέσω του header (π.χ. το transport πρωτόκολλο βάζει ένα header με τον αριθμό του μηνύματος, τον αριθμό του πακέτου. Αυτοί οι αριθμοί χρησιμοποιούνται από τον αποδέκτη-transport πρωτόκολλο για αντιμετώπιση λαθών).

Αυτή η επεξεργασία κοστίζει ... (ιδιαίτερα σε μικρά, LAN-based Κ.Σ.).

Πρόσφατα, πολλοί αναφέρονται σε μια υποδιαίρεση του data link level, ονόματι **Medium Access Control Level**: Ασχολείται με **mutual exclusion**: μόνο ένας κόμβος μπορεί να χρησιμοποιεί το "καλώδιο" σε πολλά δίκτυα, π.χ. ethernet, token bus/ring, ... Σημαντικό επίπεδο γιατί υπάρχει ένα και μοναδικό, "κοινό" "καλώδιο".

9.2 Το μοντέλο πελάτη-εξυπηρετή (Client-Server)

Το Κ.Σ. δομείται σαν μια σειρά διεργασιών (πελατών & εξυπηρετητών). Η επικοινωνία τους βασίζεται στο απλό μοντέλο request-reply. Ετσι αποφεύγονται τα πολλά επίπεδα! (απαιτούνται μόνο το Physical layer, Data Link Layer, και το Session layer που υλοποιεί το request-reply πρωτόκολλο.

Μιας και το υλικό υλοποιεί τα πρώτα 2 επίπεδα, από πλευράς λογισμικού μένει μόνο η υλοποίηση του request-reply πρωτόκολλου.
Δύο systems calls αρκούν: send (dest& msg) & receive (addr &).

- το dest ορίζει τον αποδέκτη-process
- &msg είναι η διεύθυνση του μηνύματος
- addr είναι η διεύθυνση στην οποία "ακούει" ο αποδέκτης για μηνύματα.
- &buf είναι η διεύθυνση του buffer όπου θέλει ο αποδέκτης να "ληφθεί" το μήνυμα.

9.2.1 Διευθύνσεις Οντοτήτων (Addressing entities)

Πως ορίζουμε τον αποδέκτη του μηνύματος.

Α. Μπορούμε να δώσουμε την διεύθυνση στο δίκτυο της μηχανής όπου τρέχει ο αποδέκτης. Πως, όμως, ξέρουμε αυτή τη διεύθυνση;
Τι γίνεται αν αλλάξει; Τι γίνεται αν τρέχουν πολλά processes στη μηχανή;

Β. Μπορούμε να στείλουμε στη διεύθυνση process@machine (ή machine.service_name, όπου ο κάθε server με ένα system call λέει στον kernel το δικό του service_name).

Το βασικότερο πρόβλημα είναι η έλλειψη διαφάνειας. Ο χρήστης πρέπει να ξέρει σε ποιά μηχανή τρέχει ο αποδέκτης.

Γ. Μπορούμε να αφήσουμε το κάθε process να διαλέξει την διευθυνσή του από ένα πολύ μεγάλο και αραιό address space (large, sparse) π.χ.64-bit ακέραιοι. Η πιθανότητα σύγκρουσης είναι αμελητέα.

Χρειάζεται όμως πριν από κάθε επικοινωνία να σταλεί σε όλους (broadcast) ένα ειδικό packet για να μάθει ο αποστολέας την διεύθυνση της μηχανής που τρέχει το process.

Δ. Μια καλή λύση είναι να χρησιμοποιηθεί ένας name server ο οποίος συνδέει ASCII, υψηλού-επιπέδου ονόματα, με χαμηλού-επιπέδου διευθύνσεις (π.χ. "file service" → machine: panoramix; process: 17). Ετσι πρώτα ρωτάται ο name server και μετά επικοινωνεί με το κατάλληλο process.

Ολες αυτές οι λύσεις έχουν προβλήματα: έλλειψη διαφάνειας, κόστος broadcasting, bottlenecks στο name server).

9.2.2 Ανασταλτική και μη επικοινωνία (Blocking vs non-blocking)

Η θεμελιώδης ερώτηση είναι: όταν εκτελείται η εντολή send() ή receive(), πότε θα εκτελεστεί η επόμενη εντολή του process; (ή πότε θα επιστραφεί το control στο user process). Οι απόψεις για το πότε ένα comunication interface είναι blocking ή non-blocking διΐστανται.

send() : **Τρία σενάρια**

Α. αποστολέας-process μπλοκάρει έως ότου το msg γραφθεί σε έναν kernel buffer και τότε ο kernel επιστρέφει control στον αποστολέα ο οποίος συνεχίζει να εκτελεί εντολές παράλληλα με την αποστολή του msg.

Β. Το process-αποστολέας μπλοκάρει μόνο μέχρις ώτου ο kernel κάνει λίγο book-keeping (π.χ. σημειώσεις που είναι το msg, που πάει, κ.λπ.)

Έτσι επιτυγχάνει ακόμα μεγαλύτερο παραλληλισμό.

Γ. Ο αποστολέας μπλοκάρει έως ότου ο αποδέκτης-kernel να λάβει το μήνυμα και να στείλει το σχετικό ACK.

Ολοι συμφωνούν ότι το Β είναι non-blocking και ότι το Γ είναι blocking. Οι διαφωνίες αφορούν στο Α. Ο Tanenbaum θεωρεί το Α "blocking", στην ουσία του. Ο λόγος είναι ότι ο αποστολέας του μπορεί με ασφάλεια να ξαναχρησιμοποιήσει τον buffer μιας και έχει αντιγραφθεί στο kernel. Αντιθέτως, το Β θεωρείται non-blocking γιατί ο αποστολέας δεν μπορεί να ξέρει πότε μπορεί με ασφάλεια να ξαναχρησιμοποιήσει το buffer του.

Ενα παράδειγμα χρήσης blocking επικοινωνίας :

Producer – Consumer -- 3 processes:

producer

```
while (not done) {
   produce data;
   send(buffer_mngr, &msg)
}
```

consumer

```
while (not done) {
    send(buffer_mngr, request);
    receive(buffer_mngr, &buf
    consume data;
}
```

Buffer_mngr

```
while (not done)  {
        if (buffer is empty) then {
                receive (producer, &data);
                put data in buffer
        }
        else if (buffer is full) then {
                receive(consumer, request);
                get data from buffer into &msg;
                send (consumer, &msg)
        }
        else {
                receive_any (& msg);
                if (sender = producer)
                        put data in buffer
                else {
                        get data from buffer into &msg;
                        send (consumer, &msg) ;
                }
        }
}
```

Πάντως, πρέπει ν' αναφερθεί ότι η επιλογή Α είναι επικίνδυνη: σκεφτείτε τι θα γίνει αν πολλά processes στέλνουν μηνύματα συνέχεια και πριν σταλούν τα προηγούμενά τους. Πολλές φορές προκύπτουν deadlocks που είναι πολύ δύσκολο να λύσεις.

receive()

Blocking: το process που καλεί receive() αναστέλεται μέχρι να φθάσει ένα μήνυμα (δηλ. να τοποθετηθεί στο buffer που όρισε).

Non-blocking: Αν δεν υπάρχει κάποιο κάποιο μήνυμα για το process που καλεί receive(), τότε το process δεν αναστέλεται, αλλά συνεχίζει την εκτέλεση του προγράμματός του και "ειδοποιείται" ασύγχρονα (π.χ. με signal) όταν ένα μήνυμα καταφθάσει.

Διελκυνστίδες (Trade-offs):

- Παραλληλισμός: μπορώ να στέλνω/λαμβάνω μηνύματα και παράλληλα να κάνω άλλα πράγματα.
- Ευκολία προγραμματισμού: Ασύγχρονη συμπεριφορά είναι δύσκολο να προγραμματιστεί σωστά.
- Απαιτήσεις για buffer χώρους

Non-blocking: δύσκολα προγράμματα, αλλά με παραλληλισμό.

Blocking: εύκολα προγράμματα.

Non-blocking με kernel buffer: αρκετός παραλληλισμός αλλά προβλήματα με buffer χώρους. (deadlock & κόστος αντιγραφής).

Συνήθως, blocking system calls προτιμούνται !!!

9.2.3 Αποθηκευόμενη και μη Επικοινωνία (buffered & non-buffered)

Θεωρείστε το system call: receive (addr, &msg). Δίνει την εντολή στον kernel να τοποθετήσει ένα (επόμενο μήνυμα), που απευθύνεται στο addr, στον buffer που ορίζει η δεύτερη παράμετρος.

Τι κάνει ο kernel στο μηχάνημα του αποδέκτη όταν λάβει ένα μήνυμα με αποδέκτη-προορισμό addr, πριν ο αποδέκτης-process έχει εκτελέσει την εντολή receive(addr, &msg) ;

Τι θα γίνει αν πολλά processes στέλνουν μηνύματα-αιτήσεις που απαιτούν πολύ χρόνο για εξυπηρέτηση; Αυτά τα system calls: receive(addr, &msg) και send(addr, &msg) λέγονται **unbufered**.

Buffered System Calls χρησιμοποιούνται για να λύσουν αυτά τα προβλήματα. Η θεμελιώδης έννοια εδώ είναι το **mailbox**. Τα processes που πρόκειται να λάβουν msgs ζητούν από τον kernel να δημιουργήσει ένα mailbox το οποίο σχετίζουν με ένα address. Ετσι όλα τα μηνύματα που λαμβάνονται από τον kernel με αυτό το addr τοποθετούνται στο κατάλληλο mailbox.

Κάθε receive(addr, &msg) αφαιρεί ένα μήνυμα από το mailbox και το βάζει στο &msg. (Αν το mailbox είναι άδειο, τότε μπορεί να μπλοκάρει).

Πρόβλημα: Τι γίνεται αν γεμίσουν τα mailboxes;
Ετσι στην ουσία τα προβλήματα με τα unbuffered system calls παραμένουν και στα buffered system calls.

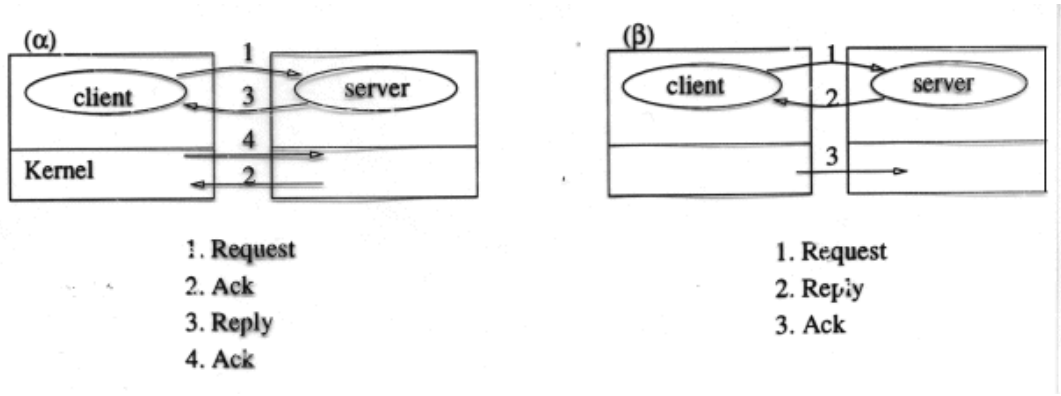
Μία Λύση: Ο αποστολέας μπλοκάρει μέχρι να ληφθεί το msg. Αν το mailbox είναι γεμάτο, ο αποστολέας μπλοκάρει...

9.2.4 Αξιόπιστη και μη επικοινωνία (Reliable και Non-Reliable)

Θεμελιώδες θέμα: πως μπορεί ο αποστολέας να είναι σίγουρος ότι το μήνυμα του έχει ληφθεί;

Σε μερικές εφαρμογές αναξιόπιστα system calls είναι αποδεκτά. Αλλως, οι λύσεις χρησιμοποιούν επιβεβαιώσεις (**ACKs)** που είναι ειδικά μηνύματα που ειδοποιούν ότι ένα

request ή ένα reply έχει ληφθεί. Τα ACKs στέλνονται από τους kernel, όχι από τις διεργασίες των χρηστών



Στο (β) αν ο client θέλει να στείλει πολλές αιτήσεις, τότε η κάθε επόμενη αίτηση είναι το ACK για το προηγούμενο reply.

Επίσης σε μερικές εφαρμογές δεν χρειάζεται το ACK στο (β). π.χ. File Server: αν ο client δεν λάβει την απάντηση θα ξαναστείλει την αίτηση.

Στο (α) ο client είναι μπλοκαρισμένος μέχρι ο kernel του λάβει το ACK (2.)

Στο (β) ο client μπλοκάρει μέχρι να ληφθεί το reply

Σκεφθείτε τις εναλλακτικές λύσεις blocking & non-blocking σε σχέση με reliability.

Ίσως να χρειαστεί να αποθηκευτούν (buffered) τα περιεχόμενα του reply στο (β). Πότε ωφελεί αυτό;

9.3 Παράδειγμα: Mach IPC

9.3.1 Εισαγωγή

Tasks: αντιπροσωπεύουν multi-threaded processes. Οταν να process δημιουργείται, υπάρχει μόνο 1 thread. Άλλα threads μπορούν να δημιουργηθούν στο ίδιο task (το κάθε thread μπορεί να δημιουργήσει κάποιο άλλο thread του task).

Η επικοινωνία IPC στο Mach είναι έμεση -δηλαδή msgs στέλνονται σε ειδικά mbox που λέγονται ports. Msg passing είναι ο μοναδικός τρόπος επικοινωνίας στο Mach. Msgs στέλνονται σε και λαμβάνονται από ports.

Ολα τα threads ενός task έχουν τα ίδια δικαιώματα πρόσβασης σε ports. Τα δικαιώματα μπορεί να είναι SEND, RECEIVE, ή SEND_ONCE.

Ports

Τα ports είναι kernel data structures (αντιπροσωπεύουν προστατευμένα mailboxes).

Ένα port υποστηρίζει:

- επικοινωνία μονής κατεύθυνσης (π.χ. το A στέλνει msg στο port του B, αλλά για να στείλει το B msg στον A, άλλο port χρησιμοποιείται). Δηλ. μόνο ένα task έχει RECEIVE RIGHTS σε ένα port.
- αξιόπιστη επικοινωνία: αν δεν κρασάρει ο κόμβος κανένα μήνυμα δεν θα χαθεί και κανένα μήνυμα δεν φθάνει "εκτός σειράς".
- "σεβεται" τα όρια των msgs. Αν 2 msgs 1K το καθένα σταλούν και μετά ο αποδέκτης καλέσει receive() τότε θα λάβει 1 msg των 1K bytes και όχι και τα 2 msgs (2K bytes) μαζί.

Δικαιώματα (capabilities)

Ετσι, τα ονόματα των ports είναι δείκτες στον πίνακα που διαχειρίζεται ο kernel για τα ports του task. Αυτά τα ονόματα είναι capabilities ονόματα. Και οι εγγραφές του πίνακα είναι capabilities. Οταν ένα thread ζητάει την δημιουργία ενός port, ο kernel πρώτα εξετάζει αν υπάρχει χώρος στον πίνακα. Αν ναι, η πρώτη άδεια/αχρησιμοποίητη εγγραφή του πίνακα θα αποτελέσει το capability για το καινούργιο port. Το port data structure δημιουργείται, η εγγραφή ενημερώνεται να δείχνει στο port και να δηλώνει τα access rights, και τέλος επιστρέφεται ο αύξων αριθμός της εγγραφής στον πίνακα του task.

RECEIVE capability σημαίνει ότι τα threads του task μπορούν να λάβουν (και να στείλουν) msgs στο port send capability σημαίνει ότι τα threads του task μπορούν μόνο να στείλουν msgs στο port.

SEND_ONCE: μπορούν να στείλουν μόνο μια φορά. (μετά την 1η φορά, ο kernel καταστρέφει το capability). Το SEND_ONCE χρησιμοποιείται συχνά σε request-reply πρωτόκολλα: Οταν ένας client στέλνει μια αίτηση σ' έναν server, του στέλνει και ένα capability για το port του client στο οποίο ο server πρέπει να στείλει την απάντηση. Μόλις, ο server στείλει την απάντηση, το task του server χάνει πλέον τα δικαιώματά του να ξαναστείλει στο port του client.

Οταν ένα task τελειώσει (exit) ή σκοτωθεί (kill) τότε ο πίνακας-capability list δεν υφίσταται πλέον. Οσα ports ανήκουν στο task (δηλ. έχει RECEIVE rights) καταστρέφονται.

Το port είναι ένα kernel data structure. Αποτελείται από 64 bytes. Οταν το kernel λάβει ένα system call να δημιουργήσει ένα port, τότε βρίσκει 64B μνήμης που την σχετίζει με το task που τρέχει.

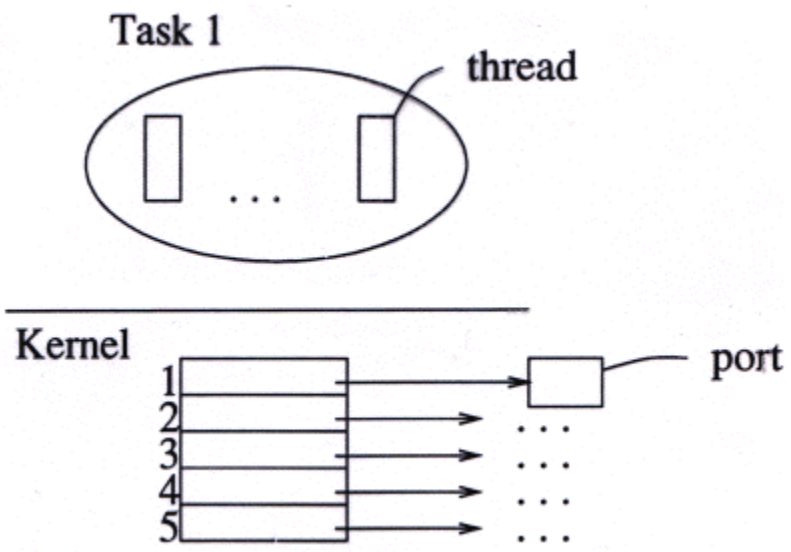
Το port (data structure), ειδικότερα, περιέχει:

- ένα δείκτη στην ουρά μηνυμάτων msg-Q που έχουν σταλεί σ' αυτό το port.
- αριθμό που δηλώνει πόσα msgs υπάρχουν στο msg-Q
- μέγιστο δυνατό αριθμό msgs στο msg_Q.
- δείκτη σε ουρά όλων των threads που έχουν μπλοκάρει εκτελώντας receice() σ' αυτό το port
- δείκτη στο kernel data structure για το task που κατέχει τα RECEIVE δικαιώματα στο port.

9.3.2 Ονομασία των ports

Πως τα threads ορίζουν/αναφέρονται στα ports; Παρόμοια με το πως UNIX processes αναφέρονται σε files. Το open(f_name,...) επιστρέφει έναν file descriptor, fd. Το port_allocate (..., &port_name) επιστρέφει ένα όνομα. Οπως και τα fd, τα port names είναι ακέραιοι αριθμοί.

Για κάθε port ενός task υπάρχει ένα πεδίο ενός πίνακα Τα περιεχόμενα του πεδίου ορίζουν τα δικαιώματα πρόσβασης του task στο port (π.χ. SEND, RECEIVE) και περιέχουν ένα δείκτη στο port (δηλ. στο kernel data structure).



Ενα task με 5 ports με ονόματα 1,2,3,4,5.

Προσέξτε: διαφορετικά tasks έχουν διαφορετικά ονόματα για ίδια ports!

Κάθε εγγραφή του capability list μπορεί να αντιπροσωπεύει:

1. ένα capability για ένα port
2. ένα capability για ένα port set

3. αχρησιμοποίητη (capability)
4. ένα port το οποίο έχει καταστραφεί (dead capability)

Η τελευταία περίπτωση συμβαίνει όταν ένα task έχει SEND capability για ένα port το οποίο καταστρέφεται από το task ιδιοκτήτη. Τότε το SEND capability γίνεται “dead capability”.

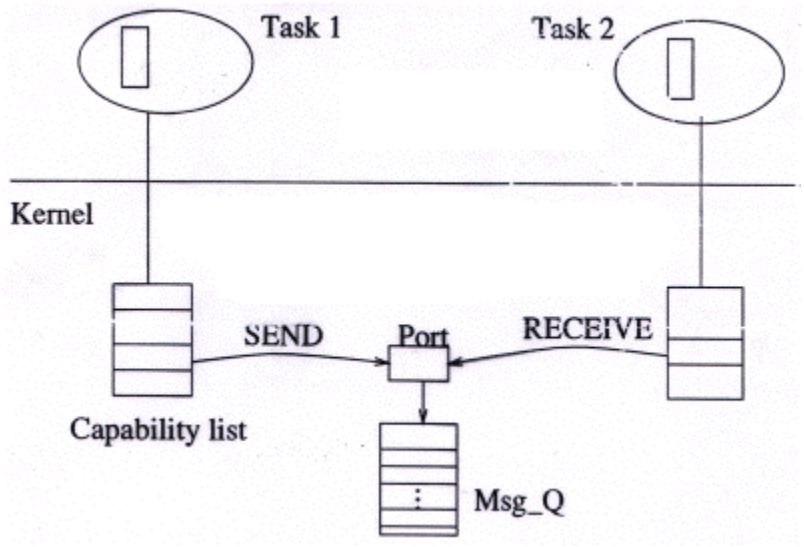
PORT Sets: Μια ομάδα από ports. Threads μπορούν να διαβάσουν από ένα port set (Δεν μπορούν όμως να στείλουν msgs σε ένα port set). Οταν ένα thread ζητάει receive(port_set...) τότε ο kernel επιστρέφει ένα msg από κάποιο port του port set.

Τα port sets συνήθως χρησιμοποιούνται από servers που προσφέρουν διαφορετικές υπηρεσίες σε διαφορετικούς clients - ένα port του port set για καθε υπηρεσία. (π.χ. ο Buf_mngr έχει ένα port για consumer και έναν για τον producer). Υλοποιούνται μέσω δημιουργίας μιας ουράς με δείκτες στις ουρές (msgs_Qs) των μελών-ports του port set.

Προσέξτε: Capability rights για ένα port μπορούν να "περαστούν" από ένα task σε ένα άλλο (χρησιμοποιώντας message passing). Πως γίνεται αυτό;

Πρόβλημα: το όνομα του capability ενός port σε ένα task, δεν έχει καμμιά έννοια για κάποιο άλλο task (αφού χρησιμοποιεί διαφορετικό capability list - πίνακα).

9.3.3 Msg – Passing με Ports



Τα δύο βασικά system calls είναι:

- msg_send (&header, option, timeout)
- msg_receive (&header, option, timeout)
- Υπάρχει επίσης και ένα higher-level interface, mach-msg (&header, options, send_size, rcv_port, timeout, notify_port);

Το mach_msg μπορεί να χρησιμοποιηθεί για να στείλει ένα msg, να λάβει ένα msg, ακόμα και να στείλει ένα msg και να λάβει αμέσως μετά ένα msg (αυτό λέγεται RPC).

- Τα bits στο options καθορίζουν τι απ' όλα θα συμβεί. Το timeout χρησιμοποιείται για να ξεμπλοκάρει το thread που καλεί το mach_msg().
- Send_size & rcv_size ορίζουν το μέγεθος του msg & του buffer που θα κρατήσει το λαμβανόμενο msg. αντίστοιχα.
- rcv_port: που θα τοποθετηθεί το λαμβανόμενο msg
- header είναι δείκτης για το που θα τοποθετηθεί το μήνυμα (π.χ. receive()) ή για το που βρίσκεται το μήνυμα που θα σταλεί (π.χ. send()).
- το mach_msg καλεί msg_send() & msg_receive() ανάλογα.

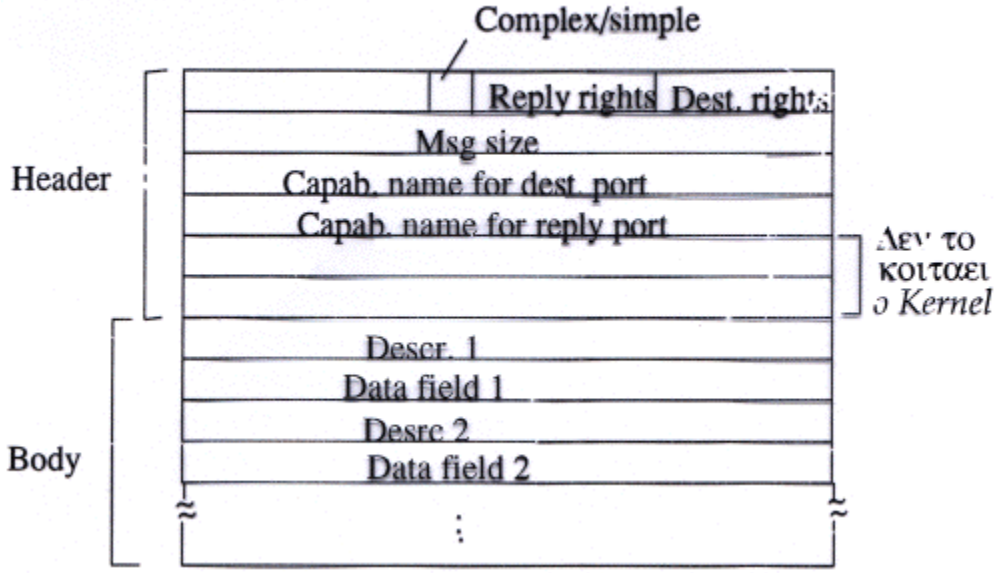
Οταν χρησιμοποιείται το interface: msg_send() msg_receive() :

- Τα option bits ορίζουν: πόσο θα μπλοκάρει ο αποστολέας ή ο αποδέκτης. Η αρχική συμπεριφορά είναι
    - msg_send(): μπλοκάρει μέχρι να τοποθετηθεί το msg στο port του receiver.
    - msg_receive(): μπλοκάρει μέχρι ένα msg να ληφθεί (δηλ. από το port στο address space του αποδέκτη).

Άλλες επιλογές είναι:

- Αν το port του αποδέκτη είναι γεμάτο, τότε να μην μπλοκάρει ο sender.
- όταν παρέλθει ο χρόνος timeout τότε ξεμπλοκάρισε.

Η ΔΟΜΗ ΤΩΝ MSGS



Απλά (simple) msgs δεν μεταφέρουν capabilities ή pointers.

msg size: το μέγεθος του msg: header + body

Capability names: για τον server port και για το port του client όπου θα τοποθετηθεί η απάντηση.
Το msg-send() τοποθετεί το msg στο port του αποδέκτη. Το msg_receive() τοποθετεί το msg στο address space (δηλ. σε κάποιο buffer) του αποδέκτη.
Το κύριο μέρος ενός msg (body) αποτελείται από μια σειρά ζευγών (description, data fields).

Ο κάθε descriptor ορίζει τα περιεχόμενα του data field Συνήθως: τον τύπο (int, char, boolean, - strings, float, capability, bit, byte, ...)
Αυτές οι πληροφορίες χρειάζονται για πιθανή "μετάφραση" από μηχανή σε μηχανή (ASCII ←→EBCDIC, byte orders..)

Μεταφορά capability

Είναι δυνατόν να δοθεί σε ένα άλλο task SEND capability (από τον ιδιοκτήτη) ή ακόμα και να μεταφερθεί το RECEIVE capability σε "νέο" ιδιοκτήτη. Ο descriptor ορίζει ότι το capability πρέπει να τοποθετηθεί/δοθεί στο capability list του αποδέκτη. Ο descriptor επίσης ορίζει αν κατά τη μεταφορά το capability θα υποστεί "μετάλλαξη". Δηλαδή ο αποστολέας έχει RECEIVE και δίνει μόνο SEND ή SEND_ONCE.

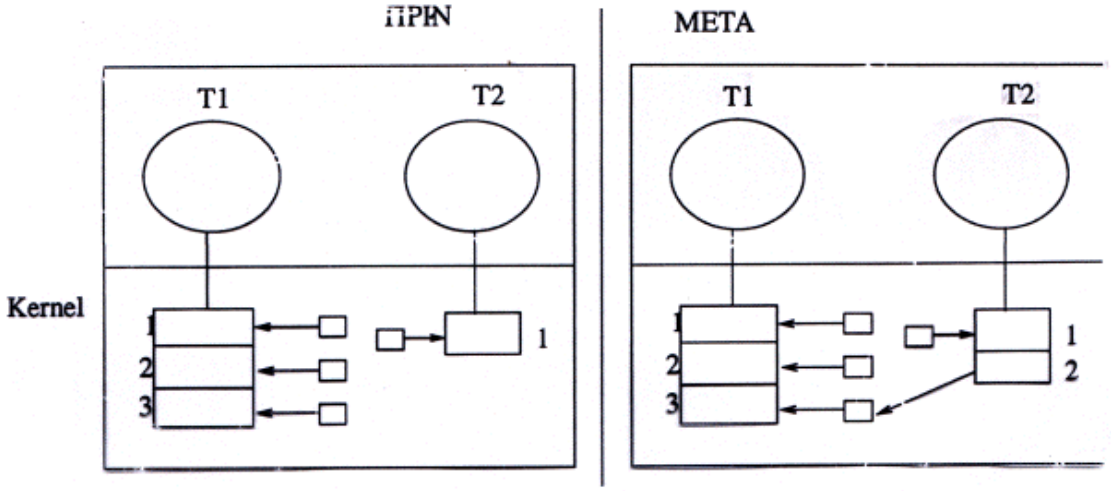


Figure 1: π.χ. T1 στέλνει το capability 3 στο T2

Οταν φτάσει το μήνυμα στο port του T2, ο Kernel το εξετάζει και βλέπει από τον header ότι είναι complex. Ο descriptor στο msg_body "φανερώνει" ότι μεταφέρεται ένα capability (με οδηγίες ότι δίνεται SEND capability sto Πορτ 3 toy T1). Ο kernel βρίσκει μία άδεια εγγραφή στο capability list του T2 (δηλ. η εγγραφή 2) και αλλάζει το όνομά της από 3 σε 2. Οταν ο receiver εκτελέσει receive() θα δει ότι έχει ένα καινούργιο capability με όνομα 2.

9.3.4 Υλοποίηση για υψηλή απόδοση

<u>Επικοινωνία IPC στον ίδιο κόμβο</u>

Πώς μπορούμε να "στείλουμε" ένα μήνυμα;

<u>Πρόβλημα</u>: οι διεργασίες αποδέκτης και αποστολέας έχουν διαφορετικά address spaces.

- Για να σταλεί ένα μήνυμα πρέπει πρώτα να τρέξει ο αποστολέας να καλέσει msg_send (&hdr, ...) και μετά ο αποδέκτης να καλέσει msg_receive(&hdr, ...)
- Το &hdr και το &hdr των sender & receiveer αντίστοιχα είναι διευθύνσεις σε ξεχωριστά address spaces που "μεταφράζονται" χρησιμοποιώντας διαφορετικά memory maps/Page Tables.
- Η παραδοσιακή UNIX λύση στο πρόβλημα βασίζεται στο εξής: Οταν ο kernel λαμβάνει την εντολή msg_send (&hdr, ...) χρησιμοποιεί το physical address του μηνύματος και αντιγράφει το msg σε ένα kernel buffer.

Οταν, κατόπιν, τρέξει ο receiver και εκτελέσει msg_send (&hdr, ...) τότε αντιγράφει το msg από τον kernel buffer στο physical address που αντιστοιχεί στο buffer του receiver (που βρίσκεται από το memory map / Page Table του receiver). Αυτή η λύση επιτυγχάνει το επιθυμητό αποτέλεσμα χωρίς να θυσιάσει την προστασία που παρέχεται από τα διαφορετικά address spaces των processes.

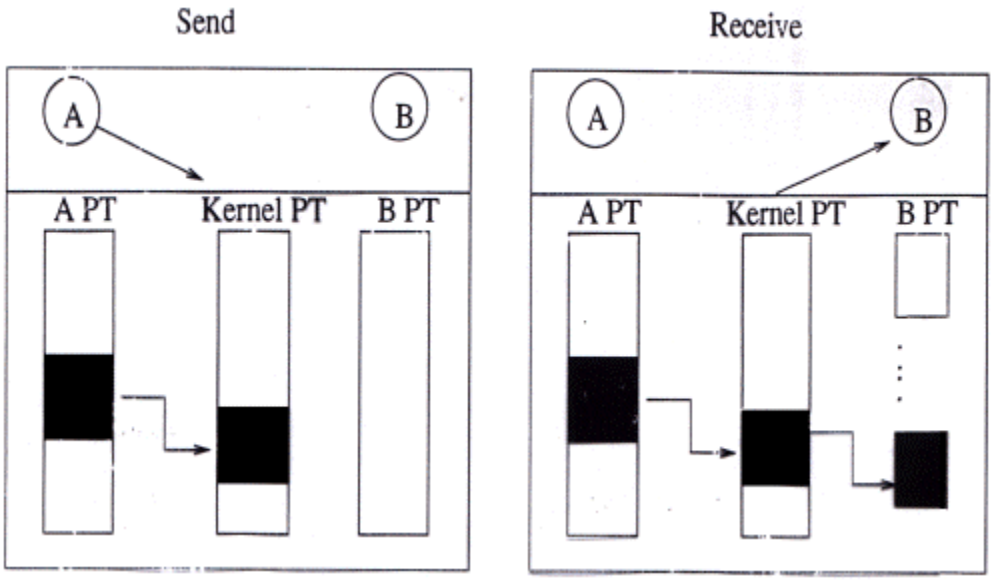
Το πρόβλημα με αυτή τη λύση είναι το κόστος της. Απαιτούνται 2 αντίγραφα: sender → kernel & kernel → receiver.

Επίσης όταν το msg έχει pointers, αυτοί έχουν νόημα μόνο στο "περιβάλλον"/context του sender και όχι στο context του receiver.

Το mach αποφεύγει το κόστος της παραδοσιακής τεχνικής (copy) ως εξής:

- Τα msgs στέλνονται χρησιμοποιώντας memory management. Προσέξτε ότι το κάθε task έχει ένα Pape Table του οποίου οι εγγραφές "μεταφράζουν" τις vistual addresses ενός msg στις physical addresses της μηχανής.
- Οταν ο sender καλεί msg_send (&hdr, ...) ο kernel αντιγράφει το τμήμα του Page Table του sender που αναφέρεται στη διεύθυνση (virtual)&hdr -δηλ. στο msg - σ' ένα δικό του προσωρινό Page Table (kernel Map).
- Οι εγγραφές του PT του sender που αναφέρονται στο msg χαρακτηρίζονται σαν copy-on-write. Αυτό σημαίνει ότι αν ο sender μετά την επιστροφή του msg_send() θελήσει να ενημερώσει το msg (δηλ. &hdr) τότε οι σελίδες της μνήμης που περιέχουν το msg θ' αντιγραφθούν, το PT του sender θα ενημερωθεί, και μετά θα επιτραπεί η ενημέρωση των δομών του msg (πάνω στο καινούργιο αντίγραφο).
- το παλιό αρχικό αντίγραφο του msg συνεχίζουν να "δείχνουν" οι εγγραφές του PT του kernel.

- Οταν ο reciever εκτελέσει msg_receive(&hdr, ...), ο kernel βρίσκει ένα άδειο τμήμα του Page Table του receiver στο οποίο αντιγράφει το δικό του προσωρινό page Table που αναφέρεται στο msg. Ετσι το msg λαμβάνεται από τον receiver χωρίς copy αλλά μέσω αντιγραφών των σχετικών τμημάτων των Page Tables.



Προσέξτε ότι με το copy-on-write, ο sender μπορεί να συνεχίσει να χρησιμοποιεί το &hdr μόλις το system call msg_send(&hdr, ..) επιστρέψει.

Οι "αντίπαλοι" αυτής της τεχνικής υποστηρίζουν ότι:

- copy-on-write δεν παρέχεται από όλα τα hardware και
- συνήθως, ο sender θα ξαναχρησιμοποιήσει τις σελίδες του msg, οπότε το copy θα γίνει έτσι και αλλοιώς.

<u>IPC πάνω από το δίκτυο</u>

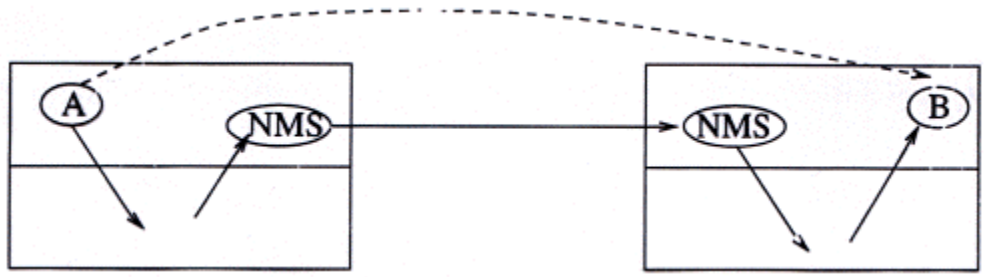
Οταν τα μηνύματα δεν στέλνονται στον ίδιο κόμβο, τότε η επικοινωνία βασίζεται στους Network Message Servers ένα αντίγραφο του NMS τρέχει σε κάθε κόμβο σε ένα Mach Distributed System.

Το NMS είναι ένα multi threaded Process που υποστηρίζει:

- επικοινωνία με local threads
- αποστολή msgs στο δίκτυο
- "μεταφράσεις" μεταξύ διαφορετικών μηχανών
- name service
- authentication

Οταν ένα task A στέλνει ένα msg στο task B σε άλλη μηχανή, τότε:

- ο kernel εξετάζοντας το msg βλέπει ότι στέλνεται σε remote μηχανή. Οπότε το μεταβιβάζει στο NMS.
- Ο NMS έχει μια database με τη βοήθεια της οποίας βρίσκει την μηχανή στην οποία υπάρχει το port που ζητείται.
- Σε μια άλλη δομή ο NMS στην μηχανή-αποστολέα βρίσκει το "network port" στο οποίο τρέχει/ακούει ο NMS της μηχανής-αποδέκτη και στέλνει το μήνυμα εκεί.
- Ο NMS στον αποδέκτη λαμβάνει το μήνυμα, και το δίνει στο κατάλληλο port (local).
- Οταν το task B (κάποιο thread του, δηλαδή) εκτελέσει msg_rceive() από αυτό το port το msg θα ληφθεί.



<u>Η υπηρεσία ονομασίας (Name Service)</u>

- Μόλις ένα thread αποκτήσει κάποιο port μπορεί να το καταχωρήσει σε ένα ειδικό Name DB, και να το συνδέσει με ένα όνομα (υψηλού επιπέδου). Το ειδικό DB μπορεί να εξυπηρετήσει queries για τέτοια ονόματα, επιστρέφοντας το port.

netname_check_in(nameserver_port, ßervice_name", ...,port) Αυτό καταχωρεί το port στο DB και το συνδέει με το ßervice_name"

netname_look_up(name_server_port, "host_name", ßervice_name",*port_id) επιστρέφει το port_id που συνδέεται με το ßervice_name" Το host_name ορίζει σε ποιό host να ψάξει ο Name Server. Αν host_name = "*" τότε όλο το h/w ψάχνεται.

Ο name server είναι ένα κομμάτι του Network Message Server.

9.4 Κλήση μακρινών διαδικασιών -- Remote Procedure Call (RPC)

Μηχανισμός (software) που διευκολύνει την κατασκευή κατανεμημένων προγραμμάτων. "Κρύβει" λεπτομέρειες για

- επικοινωνία (IPC)

- βλάβες (κόμβων και επικοινωνιακών γραμμών)
- ετερογενή hardware κόμβοι

"Remote" = Ενεργοποιούμε μια διαδικασία σε διαφορετικό κόμβο ή διαφορετικό process (δηλ. address space) με τρόπο διαφανή!

<u>Θεμελιώδη Ζητήματα :</u>

- Binding (σύνδεση client και server)
- Heterogeneity (ετερογενή συστήματα)
- Transparency (διαφάνεια)
- Concurrency (παράλληλη εκτέλεση)

Σύνδεση

Αναφέρεται στο πως "συνδέεται" ο client με τον server (που εκτελεί την κληθείσα διαδικασία). Με ποιο τρόπο, μεταφράζει την κλήση μιας διαδικασίας από τον client στην εκτέλεση μιας διαδικασίας σε διαφορετικό process και διαφορετικό κόμβο.

Ενπεριέχει την διαδικασία εύρεσης του "remote" κόμβου και της διεργασίας του server ➔ addressing & naming.

Είναι θεμιτό το binding νσ είναι δυναμικό ➔ ο server μπορεί να μετακομίσει ή να αντιγραφεί χωρίς να χρειάζεται να ξανα-μεταγλωτιστούν (compiled) όλα τα προγράμματα που αναφέρονται στις διαδικασίες που εκτελούνται από τον server.

Συνήθως αυτό επιτυγχάνεται μέσω ενός name server. Ενας server εγγράφεται (register) (εξάγει-exports) σ' έναν name server. Τα στοιχεία της εγγραφής αποτελούνται από: όνομα, διευύθυνση (handle), έκδοση (version), οι ρουτίνες του server, ...

Οταν ένας client καλέσει τον server για πρώτη φορά, τότε ερωτάται ο name server για την διεύθυνση του server, τις ρουτίνες που υποστηρίζει κ.λπ.

Ετερογένεια

Ο client και ο server μπορούν να τρέχουν σε ετερογενείς μηχανές (h/w & s/w) και να είναι γραμμένοι σε διαφορετικές γλώσσες προγραμματισμού.

➔ διαφορετική αναπαράσταση data types (strings in ASCII ή EBCDIC, integers σε 1's ή 2's complement, float point, κ.λπ) και διαφορετικά byte orders.

Δηλαδή όταν στέλνουμε σαν παράμετρους ή σαν αποτέλεσμα δεδομένα αυτά μπορεί ν' "αλλοιωθούν" λόγω προβλημάτων διαφορετικής αναπαράστασης των δεδομρενων στις επικοινωνούντες μηχανές.

STUBS στον client και στον Server

Η θεμελιώδης αρχή του RPC είναι η διαφάνεια: Remote διαδικασίες καλούνται από προγράμματα με τον ίδιο τρόπο που καλούνται οι παραδοσιακές διαδικασίες στο address space του client. Πως μπορεί να επιτευχθεί αυτός ο στόχος όμως, υπό το φως των προβλημάτων binding & heterogeneity.

Η απάντηση βασίζεται στην έννοια των stubs

- stub: μια συλλογή διαδικασιών
- Υπάρχουν client & server stubs
- Οι ρουτίνες των stubs γίνονται linked με το πρόγραμμα του (client) και, αντίστοιχα του server.
- Οταν ένας client καλεί ένα (remote procedure ενός) server για πρώτη φορά ενεργοποιείται ο stub ο οποίος αναλαμβάνει να επικοινωνήσει με τον name server για να βρεί την διεύθυνση του server.
- Ολα τα stubs ξέρουν πως μπορούν να μεταφράσουν δεδομένα μεταξύ όλων των τρόπων αναπαράστασης δεδομένων και byte ordering. Αν χρειαστεί λοιπόν κάνουν αυτή τη μετάφραση.
- Κατόπιν, κατασκευάζουν τα κατάλληλα msgs με στοιχεία, την διεύθυνση του server, το όνομα της διαδικασίας του που καλείται, τις παραμέτρους, κ.λπ. και στέλνουν το μήνυμα αίτησης ή απάντησης.

Ποιός όμως δημιουργεί τα stubs ; Ευτυχώς στα πιο πολλά συστήματα δημιουργούνται αυτόματα (από ειδικούς compilers).

Το μόνο που απαιτείται από έναν compiler για την δημιουργία των stubs είναι οι προδιαγραφές του server, οι οποίες γίνονται export από τον server όταν αυτός δημιουργείται.

Από τις προδιαγραφές (specifications) του server και από τους κανόνες "μετάφρασης" από και σε διαφορετικούς τρόπους αναπαράστασης πληροφορίας ο compiler μπορεί να δημιουργήσει client stubs το οποίο δημιουργεί το κατάλληλο msg (με την σωστή σειρά και σωστό μέγεθος κάθε παραμέτρου και του ονόματος της διαδικασίας).

Παρόμοια μπορεί να κατασκευασθεί ένα server stub το οποίο κάνει unpack το όνομα και τις παραμέτρους της διαδικασίας και καλε;ι την σωστή διαδικασία του server.

<u>Παράδειγμα</u>: Προδιαγραφές ενός File Server.

Specification: File Server: Version: 1

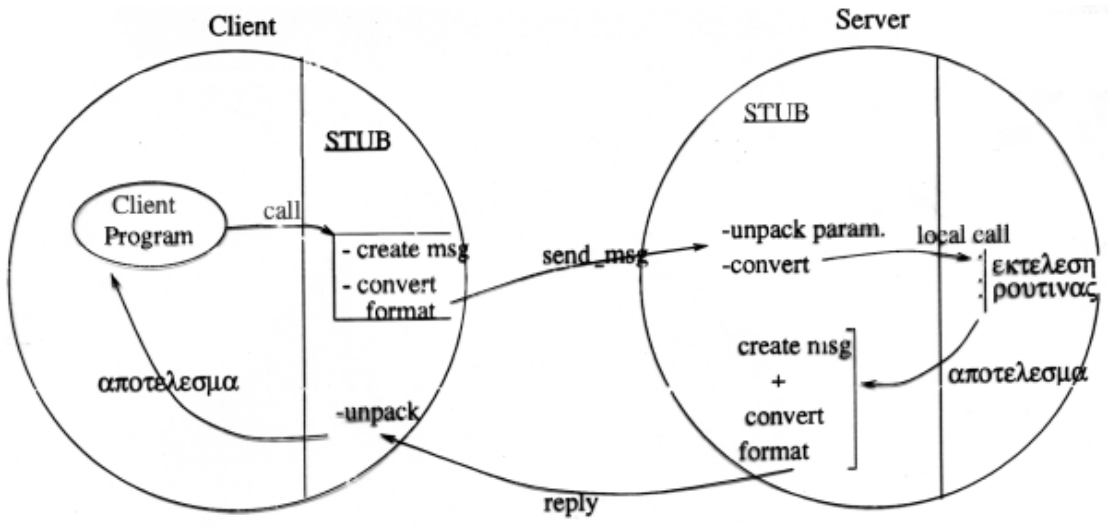
```
long read (in char name[max], out char buf[size],
            in long bytes, in long position) ;

long write ( ....);
```

end specification: Fil Server

Παράδειγμα μηνύματος για File Service

read (f_name, buf, bytes, position)



Διαφάνεια

Εχουμε δει ότι όσο αφορά τα ζητήματα binding & heterogeneity τα client & server stubs μπορούν να επιτύχουν την διαφάνεια.

Το πρόβλημα περιπλέκεται όταν υπάρχει η πιθανότητα βλάβης (failure) Βλάβες μπορούν να παρουσιαστούν στον server, στον client, και στο επικοινωνιακό σύστημα.

Α. Βλάβη -- Ο Server δεν απαντάει

Σ' αυτή την περίπτωση μπορεί ο server να "έχει πέσει" ή να έχει αλλάξει (π.χ. νέα έκδοση) Μία λύση είναι exception handling Σε μερικές γλώσσες είναι δυνάτόν ο προγραμματιστής να γράψει ειδικές ρουτίνες για εξαιρετικά φαινόμενα (excpetions) π.χ. Ada. Σε μερικά συστήματα, π.χ. Unix, ο προγραμματιστής μπορεί να γράψει singnal handlers, δηλαδή το καθε exception δημιουργεί ένα signal το οποίο εξυπηρετείται από τον handler του προγραμματιστή. Το πρόβλημα μ' αυτή τη λύση είναι ότι δεν είναι γενική! Επίσης, όταν ο προγραμματιστής γράφει exception handlers τότε καταστρέφεται η διαφάνεια...

➔ η λύση στην περιπτωση Α είναι δύσκολη

Β. Βλάβη -- Χάθηκε η αίτηση στον Server

Συνήθως, ο kernel-αποστολέας κάνει time-out και ξαναστέλνει την αίτηση. Ετσι λύνεται το πρόβλημα, εκτός από παθολογικές καταστάσεις όπου το δίκτυο έχει σοβαρά προβλήματα.

Γ. Βλάβη -- Χάθηκε η απάντηση του Server

Το θεμελιώδες πρόβλημα είναι ότι αν παρέλεθει ο χρόνος του time-out ο kernel (στον client) δεν μπορεί να ξέρει τι ακριβώς έγινε: είναι πεσμένος ο server, χάθηκε η αίτηση ή χάθηκε η απάντηση; Αυτό είναι η πηγή πολλών δυσκολιών στην κατασκευή κατανεμημένων λογισμικών συστημάτων.

Μια πρώτη (αδαής) λύση βασίζεται στην χρήση του timeout: αν η απάντηση δεν έχει έλθει όταν παρέλθει ο χρόνος του timeout, ξαναστείλε την αίτηση.

Το πρόβλημα μ' αυτήν την λύση είναι ότι σε μερικές περιπτώσεις αρκεί, σε άλλες όμως, όχι. π.χ. - αιτήσεις του τύπου: x:=15 καλύπτονται από την παραπάνω λύση. - αιτήσεις του τύπου: transfer (ac_A, ac_B, 1M) δεν καλύπτονται μιας και δεν είναι επαναλήψιμες (repeatable). Αν δηλ. το πρόβλημα ήταν ότι χάθηκε η απάντηση του server (και όχι ότι χάθηκε η αίτηση) τότε η αίτηση transfer δεν πρέπει να επαναληφθεί/ επανεκτελεστεί.

Μια λύση είναι ο client (kernel) να χρησιμοποιεί sequence numbers -κάθε καινούργια αίτηση έχει καινούργιο αύξοντα αριθμό. Ο sever (kernel) διαχειρίζεται έναν πίνακα που έχει μια εγγραφή για κάθε client που δηλώνει το μεγαλύτερο sequence number από τον client. Αν ο client μετά από timeout χρησαιμοποιεί το ίδιο sequence number όταν ξαναστέλνει την αίτηση, τότε ο server μπορεί να ξεχωρίσει/διακρίνει εάν είναι καινούργια αίτηση.

Δ. Βλάβες (Crashes) στον Server.

Το ίδιο πρόβλημα υφίσταται και εδώ - μόνο που η λύση με το sequence numbers δεν αρκεί. Αν ο server πέσει πριν εκτελέσει την αίτηση τότε η αίτηση πρέπει να επαναξυπηρετηθεί. Αν ο server πέσει αφού έχει εκτελέσει την αίτηση, τότε δεν πρέπει να την επαναξυπηρετήσει. Ομως ο client δεν μπορεί να ξέρει ποιά από τις δύο περιπτώσεις ισχύει, και έτσι όταν παρέλθει ο χρόνος στο timeout δεν ξέρει τι πρέπει να κάνει. Να ξαναστείλει ή όχι την αίτηση ;

Η λύση εξαρτάται από το τι θεωρείται ως κατάλληλη συμπεριφορά του συστήματος υπό την παρουσία βλαβών (semantics (RPC semantics in the presence of failures).

Υπάρχουν 3 λύσεις:

- *at-least-once semantics*: (τουλάχιστον μία φορά) -- ο πελάτης συνεχώς ξαναστέλνει την αίτηση μέχρι να λάβει απάντηση πριν το timeout. Δηλ. το RPC εκτελέστηκε τουλάχιστον μια φορά.

- *at-most-once semantics*: (τομπολύ μία φορά) -- μόλις το timeout παρέλθει, ο kernel δεν ξαναστέλνει την αίτηση - απλώς ειδοποιεί τον πελάτη. Ο πελάτης ξέρει ότι η αίτησή του εκτελέστηκε το πολύ μια φορά.
- Καμμιά εγγύηση: η αίτηση μπορεί να εκτελέστηκε από τον server από 0 μέχρι N φορές, N > 0.

Φυσικά, το επιθυμητό είναι *exactly-once semantics*. Αλλά αυτό δεν μπορεί ν' επιτευχθεί. Γιατί ;

➔ η πιθανότητα βλάβης στον server καταστρέφει ελπίδες για διαφάνεια.

9.5 Επικοινωνία σε ομάδες διεργασιών (Group Communication)

Οι μηχανισμοί IPC που είδαμε μέχρι τώρα αφορούν «one-to-one» επικοινωνία. Συχνά είναι επιθυμητό να δημιουργήσουμε ομάδες διεργασιών (process groups) όπου τα μέλη- διεργασίες συνεργάζονται για κάποια υπηρεσία.

π.χ. Servers υψηλής διαθεσιμότητας: αν καταρεύσει ένας εξυπηρέτης τότε υπάρχουν άλλοι (backups) που συνεχίζουν να προσφέρουν τις υπηρεσίες. Ενα από τα μεγαλύτερα προβλήματα του RPC είναι ότι δεν είναι κατάλληλο για group communication. (Φυσικά μπορούν να γίνουν πολλά ξεχωριστά RPCs σε κάθε μελος της ομάδας, αλλά αυτό έχει μεγάλο κόστος και δεν επιτρέπει διαφάνεια).

Process Groups λοιπόν είναι ένας μηχανισμός αφαίρεσης (abstraction). Επιθυμούμε διάδραση με τέτοιες ομάδες χωρίς να χρειάζεται να ξέρουμε λεπτομέρειες: πόσα μέλη, που είναι, ποιά είναι, ...

Συχνά το υλικό παρέχει σχετική υποστήριξη:

<u>multicast</u>: σε μερικά συστήματα υπάρχουν ειδικές διευθύνσεις στις οποίες "δέχονται" μηνύματα ομάδες από κόμβους. Ετσι στέλνεται 1 μόνο πακέτο και λαμβάνεται απ' όλους τους εμπλεκόμενους κόμβους.

<u>broadcast</u>: Ακόμα και αν δεν παρέχουν multicast h/w παρέχουν μια ειδική διεύθυνση στην οποία δέχονται msgs όλοι οι κόμβοι του δικτύου.

Αν δεν παρέχονται ούτε multicasting ούτε broadcasting, τότε διαφορετικά msgs πρέπει να σταλθούν σε κάθε μέλος της ομάδας.

Οταν μιλούμε για group communication εννοούμε:

1. ένα process, εκτός ομάδας, να μπορεί να επικοινωνήσει με τα μέλη- processes της ομάδας.
2. το κάθε μέλος να μπορεί να επικοινωνήσει με τα άλλα μέλη. Οταν το σύστημα επιτρέπει μόνο το (2) τότε το group ονομάζεται closed. Αλλως, ονομάζεται open.

Οι διεργασίες χρηστών πρέπει νάχουν πρόσβαση σε system calls σαν send (id, &msg), όπου το id να μπορεί ν' αναφέρεται σε ένα process ή και σε ομάδα.

Χρειαζόμαστε:

send(gid, &msg); *recv*(gid,&msg); *join_group*(gid, pid); *leave-group*(pid, gid)

Οι ομάδες μπορεί νάχουν εσωτερική δομή

- ιεραρχικές ομάδες: π.χ. υπάρχουν συντονιστές (coordinators) και εργάτες (workers).
- peer-groups: όλα τα μέλη είναι "ισόβαθμα"

Το πρόβλημα με ιεραρχίες είναι ότι έχουν μοναδικά σημεία βλάβης (single points of failure). δηλ. αν ο συντονιστής καταρεύσει, τότε τίποτα δεν γίνεται. Το πρόβλημα με peer groups είναι το κόστος λήψης αποφάσεων: πρέπει να διεξαχθεί ψηφοφορία, όπου κάθε μέλος στέλνει ένα μήνυμα με την ψήφο του στους άλλους....

Επίσης θα πρέπει να τονισθεί ότι όταν ένα process-μέλος αποχωρεί από την ομάδα, τότε θα πρέπει να ειδοποιήσει και τ'άλλα μέλη. Αν όμως ένα μέλος καταρεύσει, τότε στην ουσία φεύγει από την ομάδα, χωρίς ειδοποίηση. Τ' άλλα μέλη πρέπει ν' ανακαλύψουν αυτό το γεγονός μόνα τους ... Δύσκολο.

Θεμελιώδη προβλήματα

Βλάβες: πως επηρεάζουν το send(gid, &msg); Είναι αποδεκτό να ληφθεί ένα msg μόνο από ένα υποσύνολο; Η απάντηση εξαρτάται από την εφαρμογή π.χ. Name service.

Ταυτοχρονισμός (Concurrency): Θεωρείστε 2 μηνύματα που στέλνονται στην ίδια ομάδα. Ποιοί περιορισμοί υπάρχουν για τη σειρά παραλαβής των από τα μέλη της; Δηλ. πρέπει όλα τα μέλη να λάβουν τα μηνύματα με την ίδια σειρά; Και πάλι η απάντηση εξαρτάται από την εφαρμογή π.χ. x : = x+1

Παραδείγματα:

1. Replicated Name/Location Servers

Στέλνεται ένα μήνυμα που ειδοποιεί ότι το x άλλαξε θέση από L1 στη L2. Λόγω βλαβών, το μήνυμα δεν παραλαμβάνεται απ' όλα τα μέλη. Σ' αυτή την εφαρμογή οι βλάβες αυτές είναι αντιμετωπίσιμες, διότι εφαρμογές μπορούν να προσπελάσουν το x. Αν δοκιμάσουν να βρούν το x στη θέση L1 θα λάβουν ένα μήνυμα λάθους. Τότε θα στείλουν ένα bcast msg ρωτώντας "που είναι ο x"; Ετσι οι πιο ενημερωμένοι servers θ' απαντήσουν.

Σ' αυτή την εφαρμογή είναι αποδεκτό να μην ληφθεί ένα μήνυμα απ' όλα τα μελη της ομάδας. [Πρέπει όμως να υπάρχει ένα exception handler που όταν λαμβάνει ένα error msg, αναλαμβάνει να bcast στους άλλους servers].

2. Τράπεζες αποφασίζουν να έχουν πολλά αντίγραφα των λογαριασμών-αρχείων ώστε να είναι σε θέση να προσφέρουν υπηρεσίες στους πελάτες τους όταν ένας server καταρέει.

Ας υποθέσουμε ότι ενημερώσεις λογαριασμών πρέπει να γίνονται σ' όλα τα αντίγραφα του λογαρισμού-αρχείου, ενώ reads εξυπηρετούνται από οποιοδήποτε αντίγραφο. Αν λόγω βλάβης, κάποια αντίγραφα δεν ενημερωθούν, μεταγενέστερα reads μπορούν να επιστρέψουν λανθασμένες πληροφορίες. ➔ Σ' αυτή την εφαρμογή είναι απαράδεκτο να ληφθεί ένα μήνυμα μόνο από μια υπο-ομάδα.

Ατομικότητα (Atomicity): Ενα send(gid, &msg) system call είναι ατομικό (atomic) εάν και μόνον εάν είτε όλα τα μέλη λάβουν το μήνυμα, είτε κανένα μέλος δεν το λάβει. → atomic bcast (ή atomic mcast).

3. Θεωρείστε ένα αντικείμενο x με αντίγραφα. Εάν οι ενημερώσεις του x διέπονται από την μεταφορική ιδιότητα (commutative) τότε δύο διαφορετικές ενημερώσεις που στέλνονται από διαφορετικά μέλη στην ομάδα μπορούν να ληφθούν με διαφορετική σειρά σε διαφορετικά μέλη:

π.χ. x = x + 1, βάλε στοιχείο A στο σύνολο Σ, κ.λπ. Αλλά αν οι πράξεις δεν έχουν αυτή την ιδιότητα, τότε θα πρέπει όλα τα μέλη να λάβουν τα μηνύματα με την ίδια σειρά, αλλοιώς το σύστημά μας θα είναι ασυνεπές.

➔ οι ιδιότητες διάταξης (ordering properties) ενός συστήματος για group communication είναι πολύ σημαντικές.

9.5.1 Επικοινωνία με ομάδες στο σύστημα ISIS

Το ABCAST πρωτόκολλο (atomic broadcast).

<u>Στόχος</u>: Οποιαδήποτε συνεπής διάταξη (consistent ordering) είναι αποδεκτή. π.χ. για κάθε 2 μηνύματα M1, M2 που στέλνονται στο G, το συστημά μας είναι συνεπές εφόσον είτε όλα τα μέλη του G τα λάβουν με την σειρά M1, M2 είτε με την σειρά M2 M1. (M1 M2 σ' ένα μέλος και M2 M1 σ' άλλο μέλος δεν είαι αποδεκτό).

Το πρωτόκολλο έχει 2 φάσεις (2 phases) και η κάθε φάση έχει 2 γύρους (2 rounds). Ενα μέλος είναι ο συντονιστής.

1. Ο αποστολέας στέλνει το μήνυμα στα μέλη της ομάδας.

2. Κάθε παραλήπτης, **i,** τοποθετεί το μήνυμα σ' ένα queue και το σχετίζει μ' ένα βαθμό προτεραιότητας **Pi** - ένας ακέραιος που είναι μεγαλύτερος όλων των άλλων βαθμών προτεραιότητας που έχει αναθέσει ο **i**. Επίσης συσχετίζει το μήνυμα με μία boolean μεταβλητή "undeliverable». Τέλος, επιστρέφει το Pi στον αποστολέα.

3. Ο αποστολέας συλλέγει τα Pi, i=1,2,3… και υπολογίζει MAX = το μέγιστο Pi και στέλνει το MAX στα άλλα μέλη.

4. Κάθε μελος, **i**, ενημερώνει Pi = MAX και σημειώνει το μήνυμα σαν παραδώσιμο ("deliverable"). Ταξινομεί το queue με τα μηνύματα με βάση τους βαθμούς προτεραιότητας.

Ολα τα msgs "deliverable" μπορούν να παραδοθούν στις εφαρμογές με τη σειρά των προτεραιοτήτων τους τώρα.

<u>Παράδειγμα</u>:

S1 & S2 στέλνουν μηνύματα M1 & M2 αντίστοιχα. Οι X και Y είναι μέλη του group. Ας υποθέσουμε ότι πριν την λήψη των msgs αυτών Py = 8 & Px = 12.

Το Y δίνει: Py = 9 στο M1 Το X δίνει: Px = 13 στο M2

Py = 10 στο M2 Px = 14 στο M1

Το S2 λαμβάνει: <10, 13> και στέλνει MAX = 13 (για το M2)
Το Y αλλάζει "U" -> "D" & Py = 13
Το X αλλάζει "U" -> "D" μόνο.
Στο μέλος X M2 μπορεί να παραδοθεί
Στο μέλος Y M2 δεν μπορεί αφού υπάρχει msg στο Queue
με μεγαλύτερη προτεραιότητα που είναι "U".

S1 λαμβάνει: <9, 14> και στέλνει 14 = max (για το M1)
Το X αλλάζει "U" -> "D" μόνο
Το Y αλλάζει "U" -> "D" και Py = 14
Στο X το M1 τώρα μπορεί να παραδοεθεί
Στο Y M1 & M2 τώρα μπορεί να παραδοθούν
(με σειρά προτεραιότητας)

Στο προηγούμενο παράδειγμα είδαμε πως επιτυγχάνεται η επιθυμητή σειρά παράδοσης μηνυμάτων σε όλα τα μέλη. Αλλά τι γίνεται με την ατομικότητα;

Αν ο συντονιστής δεν λάβει απαντήσεις από όλα τα μέλη (στο τέλος της 1ης φάσης) τότε κάνει abort δηλ. ζητά και από εκείνους που απαντήσανε να πετάξουν το msg. ➔ αν όλα τα μέλη απήντησαν τότε όλοι τόχουνε βάλει στο msg queue.

Αν, λόγω βλαβών, κάποιο μέλος δεν λάβει το μήνυμα στη 2η φάση από τον συντονιστή (αυτό που καθορίζει την maximum προτεραιότητα) τότε μπορεί:

- να ελέγξει αν ο coordinator έχει καταρεύσει (ή μάλλον αν μπορεί να επικοινωνήσει μαζί του)
- Αν ναι, τότε ζητά το msg που δεν έλαβε
- Αν όχι, τότε μπορεί αυτό το μέλος να παίξει τον ρόλο του συντονιστή (δηλ. να ξανατρέξει το πρωτόκολλο).

➔ έτσι εγγυάται και η ατομικότητα.

Πρέπει ο νέος συντονιστής να αρχίζει πάντα από την αρχή;

Μήπως δεν καλύψαμε κάποιο είδος βλάβης που μπορεί να προκαλέσει μη ατομικότητα;

Το πρωτόκολλο CBCAST (causal bcast)

Η σειρά παράδοσης του ABCAST δεν είναι πάντοτε κατάλληλη π.χ. Ας υποθέσουμε ότι ο S1 θέλει να ενημερώσει πολλές φορές ένα αντικείμενο χ που υλοποιείται με replicated servers: x=10; ...; x=15 (όπου 15 είναι η τελευταία τιμή που δίνει στο χ). Εδώ το ABCAST δεν είναι κατάλληλο γιατί θα μπορούσε να επιβάλλει (x: =15; x:=10) ως την κατάλληλη σειρά σε όλα τα μέλη.

Σε γενικές γραμμές η έννοια του causality είναι η εξής: όταν ένας κόμβος S λάβει ένα μήνυμα m1 και μετά στείλει ένα m2 (δηλ. το m2 να εξαρτάται κατά κάποιο τρόπο από το m1). Σ' αυτή την περίπτωση το m1 πρέπει να παραληφθεί πριν από το m2 (σε όλα τα μέλη).

Αρα κατ' αυτήν την έννοια το CBCAST είναι πιο αυστηρό από το ABCAST. Από την άλλη μεριά, όμως, το CBCAST επιδρά μόνο σε μηνύματα που είναι causally related και η σειρά παράδοσης του CBCAST αφορά μόνο αυτά τα μηνύματα. Ετσι, κατ' αυτήν την έννοια το ABCAST είναι πιο αυστηρό από το CBCAST.

Το CBCAST είναι πολύ πιο αποδοτικό από ότι το ABCAST (λόγω της τελευταίας παρατήρησης).

CBCAST

Κάθε μέλος, **j**, της ομάδας έχει έναν πίνακα (vector) με μέγεθος ίσο με τον αριθμό του τελευταίου μηνύματος που ο **j** έλαβε από κάποιο μέλος.

Οταν το μέλος **i** στέλνει ένα μήνυμα, αυξάνει την εγγραφή **i** του δικού του πίνακα, και στέλνει τον δικό του πίνακα μαζί με το μήνυμα.

Οταν το μέλος **j** λαμβάνει το μήνυμα του **I**, τότε αν L = ο πίνακας του **j** και V = ο πίνακας του **i** που περιέχεται στο μήνυμα, ο **j** "δέχεται" το μήνυμα αν:

- V[i] = L[i] +1 [δηλ. ο **j** δεν έχει χάσει κανένα μήνυμα από τον **i**]

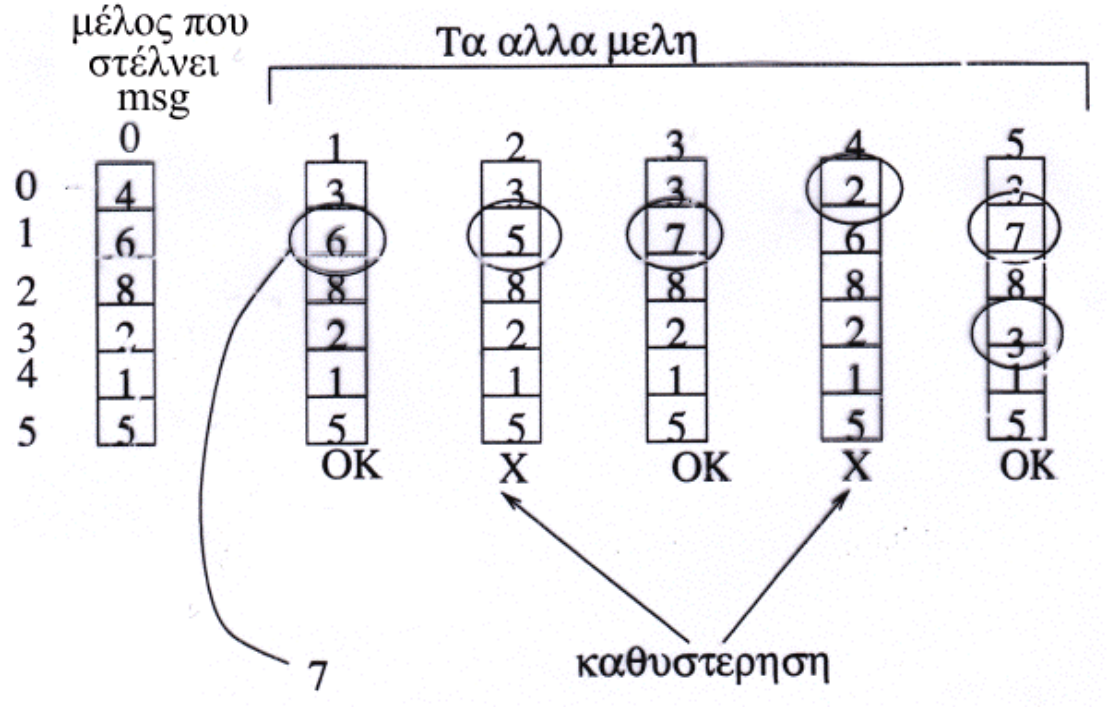
και αν:

- V[k] > L[k], για κάποιο μέλος k.

Η τελευταία συνθήκη διασφαλίζει ότι το μέλος **j** δεν έχει χάσει κάποιο μήνυμα το οποίο το έχει δει το μέλος **i**.

Οταν το μήνυμα γίνεται δεκτό ο **j** υπολογίζει το MAX(V, L) και ενημερώνει τον πίνακα L, L=MAX (V,L)

Αν το μήνυμα δεν γίνεται δεκτό τότε διατηρείται σε κάποιο buffer (καθυστερεί)



Κεφάλαιο 10. Κατανεμημένα Συστήματα Διαχείρισης Αρχείων

1. Ονομασία (Naming)
2. Διαμοιρασμός (File Sharing)
3. Προσωρινή αποταμίευση (Caching)
4. Δομή Συστήματος
5. Αντίγραφα αρχείων (Replication)
6. Παράδειγμα: Andrew File System

10.1 Ονομασία

Οπως προείπαμε η διαδικασία "ονομασίας" είναι μια μετάφραση: όνομα υψηλού επιπέδου (ASCII) → εσωτερικό UFID (symbolic name → binary name)

<u>Επιθυμητές ιδιότητες:</u>

- *Διαφάνεια θέσης* (Location transparency): τ' όνομα ενός αρχείου δεν αποκαλύπτει που έχει αποθηκευτεί το αρχείο (δηλ. το "physical location").
- *Ανεξαρτησία θέσης (*Location Independence): τ' όνομα ενός αρχείου δεν αλλάζει όταν το αρχείο "μετακομίζει".

Η ανεξαρτησία θέσης είναι πιο "δυνατή" ιδιότητα. Με την διαφάνεια θέσης αν και η θέση είναι κρυμένη δεν μπορεί ν' αλλαχθεί: π.χ. /server1/peter/foo. Αν το ~peter/foo μετακομίσει στον server2/peter/foo τότε έχουμε προβλήματα. Με την ανεξαρτησία θέσης και η θέση είναι κρυμένη, αλλά και το όνομα συνεχίζει να ισχύει όταν το αρχείο μετακομίσει.

<u>Σχήματα ονομάτων</u> (Naming Schemes)

- host name: local_name (UNIX path). Δεν υφίσταται διαφάνεια και ανεξαρτησία θέσης.
- Mounting μακρινά directories σε τοπικά directories (SUN NFS) π.χ. /fserver2/user/peter/ → /fserver1/peter. Εχει Loc. Transp. αλλά όχι Loc. Indep.
- Καθολικά ονόματα (Global Naming): όλα τ' αρχεία είναι γνωστά σε όλους με τα ίδια ονόματα.

Τα καθολικά ονόματα είναι τα πιο ακριβά (αλλά και το πιο επιθυμητό). Μπορεί να υλοποιηθεί χρησιμοποιώντας δύο πίνακες. Πρώτα: ένα symbolic name μεταφράζεται μέσω του 1ου πίνακα σε ένα locat. indep. file id. Κατόπιν: Μέσω του 2ου πίνακα, το loc. indep. file id μεταφράζεται στο τωρινό location.

Εσωτερικά ονόματα (Binary names) μπορεί να είναι δικαιώματα (capabilities) ή της μορφής *host: local name*. Ακόμη, ένα συμβολικό, υψηλού επιπέδου όνομα (symbolic name) μπορεί να σχετίζεται με μια λίστα από εσωτερικά ονόματα (π.χ. στην περίπτωση που υπάρχουν αντίγραφα).

10.2 Διαμοιρασμός & Σημασία (Semantics)

Με την φράση "semantics" εδώ εννοούμε το πως ορίζεται η σωστή συμπεριφορά του συστήματος όταν το ίδιο αρχείο προσπελαύνεται ταυτόχρονα από διαφορετικούς clients.

Αν όλοι οι πελάτες μόνο διαβάζουν, τότε δεν υπάρχει πρόβλημα. Προβλήματα προκύπτουν όταν υπάρχει ένας writer.
Η μια διάσταση του προβλήματος αφορά το πόσο πρόσφατη είναι η πληροφορία που επιστρέφεται από μία αίτηση read. Η άλλη διάσταση αφορά στην σειρά με την οποία εφαρμόζονται ταυτόχρονες αιτήσεις write ().

UNIX semantics:

- Η πληροφορία που επιστρέφει το READ είναι η πιο πρόσφατη. (Σε Κ.Σ.) τα UNIX semantics είναι δύσκολο να εγγυηθούν με υψηλή απόδοση). (Sprite File System, Berkeley).
- Αν υπάρχει μόνο ένας F.S. και δεν επιτρέπεται η προσωρινή αποθήκευση αρχείων σε πελάτες (caching), τότε τα UNIX semantics μπορούν εύκολα να υλοποιηθούν.

<u>Σημείωση</u>: η έννοια της πιο πρόσφατης πληροφορίας δεν αφορά στο πραγματικό χρόνο, αλλά στη σειρά με την οποία ο FS εφήρμοσε τα WRITE.

Από την στιγμή που επιτραπούν αντίγραφα ή προσωρινή αποθήκευση σε πελάτες (replication ή caching) τότε πρέπει να αναπτυχθούν ειδικά πρωτόκολλα που εγγυούνται UNIX semantics.

Session Semantics:

Ένα session ορίζεται ως το σύνολο των open(), read(), ..., write() & close() που εκτελεί ένας πελάτης για ένα αρχείο. Session semantics υπαγορεύουν ότι αλλαγές σ' ένα αρχείο φαίνονται αμέσως στον client (ή στην μηχανή του) που τις ζήτησε. Σ' άλλους clients (ή μηχανές) θα φανούν όταν ο client θα κλείσει το session με close(). (Andrew File System, CMU).

Η υλοποίηση βασίζεται συνήθως στο ότι το αρχείο γίνεται cached στον πελάτη κατά την διάρκεια του open(). Οταν ο πελάτης ζητήσει close(), τότε οι αλλαγές στο τοπικό αντίγραφο στέλνονται στον F.S.

Αν δύο sessions δημιουργηθούν παράλληλα (δηλ. πριν κλείσει το 1ο session, ανοίξει το 2ο) τότε είτε το τελικό αρχείο ανήκει στον client που έκλεισε τελευταίος είτε είναι το αρχείο όπως ενημερώθηκε από ένα εκ των δύο sessions.

Immutable Semantics: εντελώς διαφορετικό μοντέλο: Σ' αυτό το μοντέλο δεν επιτρέπονται WRITE εντολές. Στην ουσία όταν χρειάζεται να ενημερωθεί ένα αρχείο, δημιουργείται ένα καινούργιο αρχείο (με διαφορετικό όνομα - π.χ. με με τον αύξοντα αριθμό έκδοσης (version number suffix).
Ετσι μονο CREATE() & READ() επιτρέπονται.

Είναι δυνατόν να δημιουργθεί ένα καινούργιο αρχείο (με παλιό όνομα) αντικαθιστώντας το παλιό αρχείο με το ίδιο όνομα.

Transactional Semantics:

Ο πελάτης περικλείει μέσα σε μία συνδιαλλαγή (transaction) ένα ή περισότερα sessions. Το σύστημα εγγυάται ότι ταυτόχρονες συνδιαλλαγές που προσπελαύνουν το ίδιο αρχείο θα δούν την ίδια κατάσταση που θα έβλεπαν αν έτρεχαν σειριακά (δηλ. το ένα μετά το άλλο και όχι παράλληλα).

Transactional semantics θεωρούνται σαν πολύ ακριβά και έτσι ακατάλληλα για F.S.s.

Immutable Semantics λύνουν πολλά προβλήματα εξ ορισμού (αφού δεν επιτρέπονται WRITEs). Αλλά το κόστος μνήμης για όλες τις εκδόσεις μπορεί να είναι απαγορευτικό.

10.3 Προσωρινή αποταμίευση (Caching)

Σε ένα Κ.Σ. η πρόσβαση σε ένα αρχείο κοστίζει συνήθως λόγω του κόστους πρόσβασης στο δίκτυο και λόγω κόστους πρόσβασης στο δίσκο. Η προσωρινή αποταμίευση αναφέρεται σε τεχνικές που αποσκοπούν στην αποφυγή ενός ή και των δύο παραπάνω πηγών κόστους:

- Α. Χωρίς caching: + n/w + δίσκο.
- Β. Caching: server MM: + n/w – δίσκο.
- Γ. Caching: client δίσκο: - n/w + δίσκο.
- Δ. Caching: client MM : - n/w - δίσκο

Η εναλλακτική λύση Β θεωρείται καλύτερη από την Γ.

Μονάδα αποταμίευσης (Caching unit): disk block, file, ...

Αν η μονάδα αποταμίευσης είναι το block δίσκου, τότε η διαχείριση της cache και του χώρου του δίσκου μπορεί να είναι πιο αποδοτική. Αν η μονάδα είναι ολόκληρο το αρχείο, τότε το κόστος μεταφοράς disk → MM είναι μικρότερο και επίσης απαιτούνται λιγότερα μηνύματα για να γίνει η μεταφορά από εξυπηρέτη → πελάτη.

Απαιτείται ένας αλγόριθμος αντικατάστασης για το όταν ο αποταμιευτής (cache) είναι γεμάτος. (Συνήθως LRU).

Για μεγάλα Κ.Σ., ιδιαίτερα, το κόστος πρόσβασης στο δίκτυο είναι σημαντικό → η χρήση αποταμιευτών στους πελάτες επιβάλλεται. Τα πιο πολλά συστήματα χρησιμοποιούν αποταμιευτές στη κύρια μνήμη του πελάτη (είτε στο virtual address space του client process, είτε στον client kernel, είτε σ' ένα ξεχωριστό process cache manager στο user-level). Σκεφτείτε ποιό συμφέρει.

Σημειωτέον ότι όλοι οι εξυπηρέτες έχουν ένα MM cache γιατί αλλοιώς το κόστος προσπέλασης αρχείων θα ήταν πολύ μεγάλο (θα περιελάμβανε το κόστος πρόσβασης στο δίσκο, πάντα!).

Το βασικό πλεονέκτημα του αποταμιευτή-πελάτη στο δίσκο είναι αξιοπιστία (non-volatile memory).

Τα βασικότερα πλεονεκτήματα του αποταμιευτή-πελάτη στη κύρια μνήμη (MM client cache) είναι :

- πολύ γρήγορη πρόσβαση
- ομοιόμορφη στρατηγική caching για πελάτες & εξυπηρέτες,
- πελάτες χωρίς δίσκους (diskless clients) μπορούν να αποταμιεύσουν.

Συνέπεια αποταμιευμένης πληροφορίας

Το πρόβλημα της ασυνέπειας προκύπτει τώρα γιατί υπάρχουν πολλά αντίγραφα ενός αρχείου: στον server, σε διάφορα client caches, κ.λπ.

Σε μεγάλο βαθμό η συνέπεια του συστήματος εξαρτάται από την πολιτική μετάδοσης των ενημερώσεων (update policy). Το θεμελιώδες ερώτημα εδώ είναι πότε οι εγγραφές (WRITE) σε έναν αποταμιευτή διαδίδονται στον εξυπηρέτη (ή και στους άλλους αποταμιευτές) (propagation policy).

<u>Write through</u>: Μόλις ενημερώνεται ένα αρχείο σε ένα cache, στέλνεται αμέσως στον server (και συνεχίζει να παραμένει στη client cache). Αυτό όμως δεν εγγυάται ότι μεταγενέστερα READ του ίδιου client θα επιστρέψουν την πιο πρόσφατη πληροφορία (γιατί κάποιος άλλος πελάτης μπορεί νάχει το ίδιο αρχείο αποταμιευμένο).

Μια λύση είναι ο cache manager να "ρωτάει" τον server πριν χρησιμοποιήσει τα αποταμιευμένα αρχεία για να εξυπηρετήσει ένα τοπικό READ. Το σημαντικότερο πρόβλημα της πολιτικής write-through έγκειται στο ότι η τοπική cache δεν μειώνει πολύ

το κόστος του δικτύου (όλα τα write() απαιτούν το δίκτυο και όλα τα read() επίσης αν απαιτείται αυστηρή συνέπεια (strong file sharing semantics).

Μια λύση είναι "batch writes" -- περιοδικά όλα τα WRITE του τελευταίου διαστήματος, στέλνονται όλα μαζί στον server με ένα WRITE. Ενα επιμέρους πλεονέκτημα αυτής της πολιτικής είναι ότι αν το αρχείο διαγραφεί πριν το batch write και μετά την ενημέρωση τότε δεν στέλνεται.

Η παραπάνω πολιτική ονομάζεται 'delayed write'. Το βασικότερο μειονέκτημα του delayed write είναι ότι δημιουργεί προβλήματα ορισμού της συνέπειας. Αν ένα READ εκτελεστεί πριν το batch write τότε "βλέπει" παλιά πληροφορία ενώ αν εκτελεστεί μετά, τότε "βλέπει" την πρόσφατη πληροφορία → "non- determinitic semantics".

Μια άλλη πολιτική είναι η «write-on-close» η οποία ταιριάζει όταν το σύστημα εφαρμόζει session semantics. Το ενημερωμένο αρχείο στέλνεται στον server μόλις τελειώσει το session (ή και αργότερα ακόμα). Οπως είπαμε πριν παράλληλα sessions που ενημερώνουν ένα αρχείο μπορεί να δημιουργήσουν το πρόβλημα της χαμένης ενημέρωσης (lost updates), αφού το τελευταίο close σβήνει στον server το αρχείο που γράφτηκε από τ' άλλο παράλληλο session. (Μπορεί αυτό το πρόβλημα να συμβεί σε centralized F.Ss ;)

Συχνά αυστηρή συνέπεια (strong consistency semantics) υλοποιείται με την βοήθεια "κεντρικού ελέγχου" από τον F.S.

- Ο F.S. διαχειρίζεται μια λίστα που αποτελείται από τους πελάτες έχουν ανοίξει το αρχείο, σε τι mode (read, write, ...) κ.λπ.
- Μια αίτηση για open σε οποιοδήποτε mode επιτρέπεται μόνο αν δεν υπάρχει κάποιος πελάτης που άνοιξε το αρχείο σε συγκρουόμενο mode (2 αιτήσεις open έχουν συγκρουόμενα modes εάν τουλάχιστον η μια είναι σε write mode). Ετσι συγκρουόμενες αιτήσεις αναγκάζονται να περιμένουν σε ουρές (wait Qs) (ή και να απορριφθούν).

Υπό την παρουσία συγκρουόμενων αιτήσεων μια άλλη λύση βασίζεται στην απενεργοποίηση του caching. (*cache invalidation/disabling*). Ο server στέλνει ένα μήνυμα σ' όλους τους συγκρουόμενους clients "διατάζοντας" τους να διαγράψουν το αρχείο από την cache τους. Αυτή η λύση ξεφεύγει από το μοντέλο client-server. Επίσης μπορεί να παρουσιάσει προβλήματα υλοποίησης: ο client δεν περιμένει τέτοιο μήνυμα, → δεν ξέρει πότε να καλέσει msg-receive(). Αν υπάρχουν threads ή non-blocking msg-receive() τότε το πρόβλημα λύνεται πιο εύκολα.

Γενικά, οι λύσεις στο πρόβλημα της συνέπειας της αποταμιευμένης πληροφορίας (cache consistency) χωρίζονται σε 2 κατηγορίες:

- Προκαλούμενες από τον πελάτη (client-initiated): Ο πελάτης ρωτάει τον εξυπηρέτη αν η cache είναι συνεπής.

- Προκαλούμενες από τον εξυπηρέτη (server-initiated): Ο εξυπηρέτης ειδοποιεί τους πελάτες όταν η cache τους είναι ασυνεπής.

Προσέξτε ότι με immutable files τα περισσότερα προβλήματα εξαφανίζονται.

10.4 Δομή του Συστήματος

Υπάρχουν τρία θεμελιώδη θέματα:

- Ποιοί μπορούν να είναι servers ;
- Ο directory server & ο file server είναι διαφορετικά processes, ή συνδυάζονται σε ένα process;
- Ο F.S. είναι stateful ή stateless;

Στο SUN NFS οποιοσδήποτε client κόμβος μπορεί να γίνει server. Αυτό προκύπτει διότι δεν υπάρχει διαφοροποίηση μεταξύ client και server software. ➔ clients και servers τρέχουν το ίδιο software.

Αλλοι (π.χ. Andrew FS) ισχυρίζονται ότι πρέπει να υπάρχει ισχυρός διαχωρισμός, συνήθως για λόγους προστασίας (ο server θεωρείται εμπιστεύσιμος -- trusted s/w).

Αν directory servers kai F.Ss. είναι διαφορετικά processes τότε σίγουρα υπάρχει το έξτρα IPC κόστος (και όχι μόνο: yp;arxoyn kai ta k;osth gia context switching, kernel traps). Αλλοι υποστηρίζουν ότι πρέπει να είναι διαφορετικά processes αφού παρέχουν εντελώς διαφορετικές υπηρεσίες.

Αν υπάρχουν διαφορετικοί εξυπηρέτες καταλόγου, τότε είναι πιθανό (ιδιαίτερα σε μεγάλα συστήματα) το file name space να χωριστεί σε πολλούς D.Ss. Ετσι το path name translation (δηλ. η μετάφραση του /J1/J2/J3/.../foo στο σωστό αρχείο) μπορεί ν' απαιτήσει την πρόσβαση σε πολλούς D.Ss. π.χ. ο DS1 έχει το /J1/J2, αλλά το κατάλληλο dir. entry δείχνει στον DS2, κ.λπ. Σε αυτήν την περίπτωση είτε ο DS1 ενημερώνει τον client να επικοινωνήσει με τον DS2, ή το κάνει ο ίδιος. Αν ο DS1 επικοινωνήσει απ' ευθείας με τον DS2 τότε αυτό πρέπει να γίνει χωρίς το κλασσικό RPC (αφού αυτό επιμένει ότι ο DS1 πρέπει ν' απαντήσει στον client και όχι ο DS2).

Το βασικό συμπέρασμα είναι ότι το pathname translation μπορεί να είναι πού χρονοβόρα διαδικασία. Μερικά συστήματα χρησιμοποιούν <u>cached names</u>. Στο προηγούμενο παραδειγμα ο DS1 μπορεί να έχει κάνει cache το binary name του /J1/J2/.../foo και έτσι να μην χρειστεί να επικοινωνήσει με τον DS2 (ο οποίος μπορεί να χρειαστεί να επικοινωνήσει με τον DS3 ... κ.ο.κ).

Συνήθως η αποταμίευση μεταφράσεων ονομάτων (name caching) υλοποιείται με την τεχνική των **hints**. Δηλαδή το αποταμιευμένο όνομα χρησιμοποιείται σαν υπόδειξη και μόνο: Αρα:

1. Μπορεί να είναι λάθος (π.χ. "μετακόμιση" αρχείου). Συνεπώς το σύστημα πρέπει να παρέχει την δυνατότητα διεξόδου - δηλ. ακολουθείται το κανονικό pathname translation χωρίς ν' εξετασθεί το αποταμιευμένο.

2. Το πρόβλημα της συνέπειας των αποταμιευμένων ονομάτων λύνεται με τον πιο αποδοτικό τρόπο (δηλ. δεν εφαρμόζεται καμμιά έννοια consistency-παρακάμπτεται το πρόβλημα).

Για εφαρμογές όπως το name caching, η τεχνική των hints είναι πολύ σημαντική μιας και αποφεύγει το μεγάλο κόστος για εγγύηση συνέπειας.

Πρέπει οι servers να είναι stateful/stateless ; Υπάρχει διχογνωμία. Η ερώτηση είναι αν ο server κρατά πληροφορίες για το session του με τον client πέραν αυτής που χρειάζεται για να εκτελέσει την τρέχουσα αίτηση.

- Με stateless servers κάθε αίτηση πρέπει να είναι αυτοδύναμη (δηλ. νάχει όλες τις πληροφορίες, π.χ. offset στο UNIX F.S.). Δηλαδή να μην χρειάζονται οι πίνακες που χρησιμοποιεί το UNIX FS για να "μεταφράσει" κάθε file descriptor στο κατάλληλο file.
- Θυμηθείτε επίσης τα προβλήματα που προκύπτουν όταν ο server είναι stateful και συμβαίνουν βλάβες.
- Με stateful servers :
  - Μικρότερα msgs (λείπει όνομα, κ.λπ.)
  - Καλύτερη απόδοση (π.χ. caching των inodes στο UNIX FS)
  - Ανίχνευση αιτήσεων-αντιγράφων πιο εύκολα
  - Διαμοιρασμός και locking μπορεί να γίνει από τον server.
- Με stateless servers :
  - Μεγαλύτερη αντοχή βλαβών (servers & clients)
  - Δεν χρειάζονται OPEN & CLOSE
  - Μικρότερες απαιτήσεις για server MM
  - Χωρίς όρια στον αριθμό open αρχείων από έναν client (δεν υπάρχει κάποιος πίνακας που να γεμίσει ...)

10.5 Αντιγραφή Αρχείων (Replication)

- Κίνητρα
  - Αξιοπιστία (reliability) : Βλάβες των servers δεν προκαλούν ασυνέπεια ή χάσιμο πληροφορίας.
  - Διαθεσιμότητα (availability): Συνέχιση παροχής υπηρεσιών στους clients ακόμα και υπό την παρουσία βλαβών.
  - Απόδοση (Performance): Εξυπηρέτηση αίτησης από κάποιον λιγότερο φορτωμένο server ή κάποιον "κοντινότερο" server.

- Διαφάνεια αντιγράφων (Replication Transparency:)
  - Θεωρείται θεμιτή ιδιότητα ενός συστήματος που υποστηρίζει replication.
  - Η διαφάνεια αυτή εννοεί ότι οι πελάτες προσπελαύνουν αρχεία χωρίς να (χρειάζεται να) ξέρουν ότι τ' αρχεία είναι αντεγραμμένα.

- Δυο Τρόποι διαχείρισης
    - Lazy Replication: Οι clients επικοινωνούν μόνο με ένα server. Οταν πρόκειται για αίτηση WRITE αυτός ο server επικοινωνεί με τους άλλους off-line (δηλ. στο background - όταν έχει ελύθερα CPU cycles π.χ.).

      Η αίτηση WRITE μπορεί ν' αφορά ενημέρωση ενός αρχείου ή ακόμα την δημιουργία/ διαγραφή ενός αρχείου.

    - File Server Group: Υπάρχει ένα group από file servers και η επικοινωνία των clients με το group γίνεται μέσω group communication (π.χ. ABCAST, CBCAST, κ.λ.π.).

- Πρωτόκολλα ενημέρωσης (Update)

  Συνήθως τα WRITE calls πρέπει να σταλούν σε >1 server. Πώς γίνεται αυτό; Μια αδαής λύση:

    - Υπάρχει ένα stub το οποίο πρώτα αποφασίζει αν το αρχείο είναι replicated.
    - Αν ναι, βρίσκει τους servers που έχουν αντίγραφα και εκδίδει ένα WRITE για το αρχείο σε κάθε server.

  Το πρόβλημα είναι ότι αν συμβεί μια βλάβη στον πελάτη πριν το stub έχει αποστείλει το WRITE σ' όλους τους servers, τότε μόνο μερικοί servers θάχουν την "σωστή" πληροφορία. Αυτό το φαινόμενο ονομάζεται αμοιβαία ασυνέπεια (mutual inconsistency).

  Μια πολύ χρησμοποιημένη λύση βασίζεται στην έννοια του πρωτεύοντος αντιγράφου (**primary copy**).

    - Κάθε WRITE στέλνεται σ' ένα διακεκριμένο αντίγραφο, το primary copy.
    - Αυτο το αντίγραφο αποστέλλει το WRITE στους άλλους εξυπηρέτες, που λέγονται δευτερεύοντες εξυπηρέτες (secondary servers).
    - Το όποιο READ μπορεί να εξυπηρετηθεί είτε από τον primary ή από κάποιο secondary.

  Και εδώ πρέπει να εξασφαλιστεί ότι βλάβες δεν θα καταλήξουν σε ασυνέπεια. Ετσι πριν ενημερωθεί το primary copy, το WRITE γράφεται σε σταθερή μνήμη (stable storage). Ετσι, κατά την διάρκεια ανάνηψης ενός server μπορεί να ελεγχθεί κατά πόσον είναι ενημερωμένος και αν όχι να ζητήσει να ενημερωθεί από το primary copy.
  ➔ όταν τελειώσει η διαδικασία ανάνηψης, όλοι οι servers είναι ενημερωμένοι.

Ενα σημαντικό μειονέκτημα του primary copy πρωτόκολλου είναι ότι αν το primary δεν είναι προσπελάσιμο (λόγω βλαβών) τότε κανένα WRITE δεν μπορεί ν' εξυπηρετηθεί. Διάφορες λύσεις σ' ευτό έγκειται στην εκλογή ενός καινούργιου (ενημερωμένου) primary μεταξύ των "επιζόντων" server.

Μια άλλη πολύ γνωστή λύση βασίζεται στις έννοιες της ψήφου και της συγκατάθεσης πλειοψηφίας (voting και read quorum/ write quorum).

- Για κάθε read/write αίτηση απαιτείται η συμμετοχή ενός προκαθορισμένου αριθμού servers. Για READ χρειάζονται read quorum, Qr, servers. Για WRITE χρειάζονται write quorum, Qw, servers.
- Qr και Qw συνδέονται με τις εξής σχέσεις:
  - Qr + Qw > N και
  - 2 x Qw > N

Προσέξτε ότι κάθε READ εξυπηρετείται από τον πιο ενημερωμένο server.

Με κάθε copy συνδέεται ένα version number το οποίο αυξάνεται (κατά 1) με κάθε WRITE.

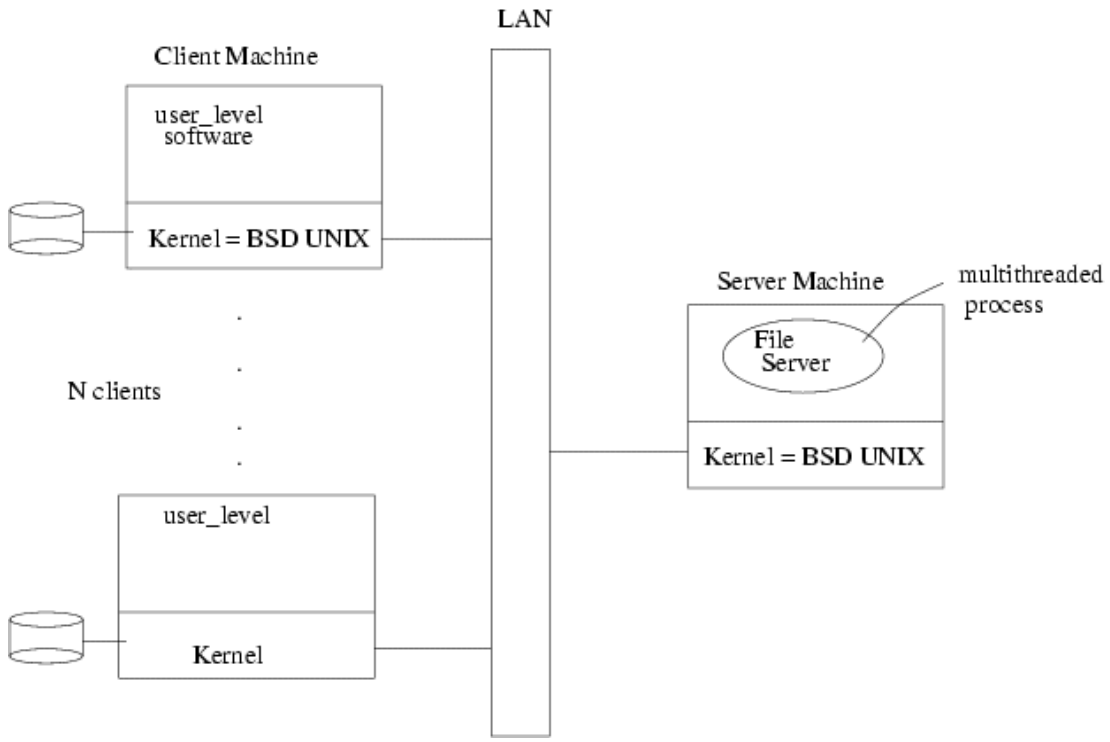
Αρχεία για τα οποία οι προσπελάσεις είναι μόνο READ (ή η πλειοψηφία τους είναι READ) μπορούν νάχουν μικρό Qr και έτσι υψηλή απόδοση. Το ανάλογο συμβαίνει και για αρχεία "mostly WRITE".

10.6 Το Andrew File System (AFS)

Κύρια απαίτηση σχεδιασμού: Δυνατότητα κλιμάκωσης (Scalability) >1000 WSs

10.6.1 Αρχιτεκτονική του AFS

- Το σχήμα δείχνει έναν cluster του AFS
- Τα clusters (μέσω gateways) μπορούν να οργανωθούν σε cells → (υπερ) εθνικά F.S.
- Ο λόγος ύπαρξης των clusters είναι για τοπικη πρόσβαση ("localized access") - δηλ. τα αρχεία των πελατών βρίσκονται συνήθως στον server που είναι στον ίδιο cluster.



Venus: S/W στον client που παρεμβαίνει σ' όλες τις αιτήσεις για File Service και τις εξυπηρετεί επικοινωνώντας με τον File Server.

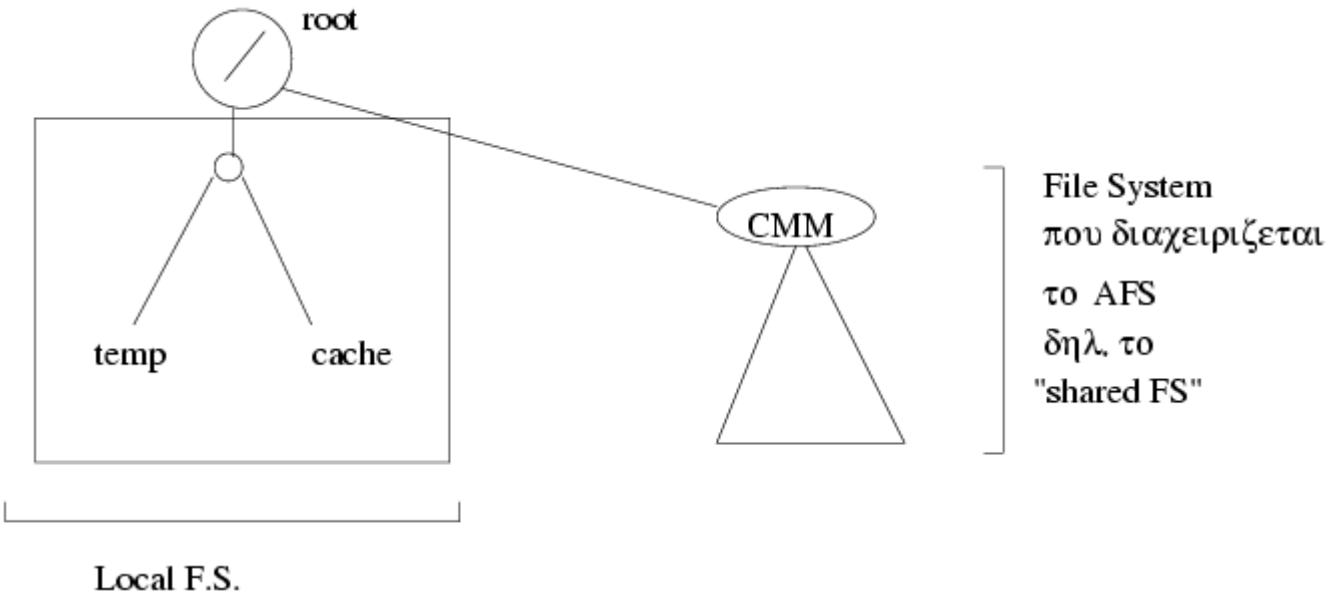
Vice: Ο File Server (S/W). Επικοινωνεί με τα Venus μέσω RPC.

Κεντρική Ιδέα: Για λόγους scalability, το μεγαλύτερο μέρος της επεξεργασίας πρέπει να γίνεται στον client ➔ το open() εξυπηρετείται ως ακολούθως:

- Ολόκληρο το αρχείο "φορτώνεται" από τον Vice.
- γίνεται cached στον πελάτη (το cache το διαχειρίζεται ο Venus).
- Το open () επιστρέφει ένα fd στον χρήστη, το οποίο "οδηγεί" στο cache αντίγραφο
- όλες οι μεταγενέστερες προσπέλάσεις εξυπηρετούνται από το cache (παρακάμπτοντας Venus & Vice).
- Το caching είναι transparent (διαφανές).

Ασφάλεια -- Trust: Μόνο οι Vice είναι εμπιστεύσιμοι

- Μόνο οι Vice αποθηκεύουν πληροφορίες σχετικές με την προστασία των files.
- Επικοινωνία μεταξύ Venus & Vice είναι κρυπτογραφημένη (encrypted).



Θεώρηση του File Name Space.

Σε κάθε client:
Volume:

- ένα σύνολο files (π.χ. ενός χρήστη ή sys.bin.)
- μπορεί να είναι "r" ή "rw"

10.6.2 AFS Semantics

- Τα processes που τρέχουν στην ίδια μηχανή client "βλέπουν" UNIX semantics (λόγω του ότι χρησιμοποιούν την ίδια cache).
- Οταν το P1 ανοίγει ένα αρχείο από τη μηχανή M1, φορτώνει το cache με το αντίγραφο του Server. Αυτό το αντίγραφο περιέχει τις αλλαγές που έχουν γίνει μόνο από closed sessions: Αν το αρχείο είναι ανοικτό και ενημερωμένο από το P2 που τρέχει στο M2, το P1 δεν θα δει τις αλλαγές του P2, αν το P2 δεν έχει κλείσει το αρχείο.

  ➔ Τα AFS Semantics προσεγγίζουν session semantics.

- Callbacks: Υλοποιηστικά θέματα

  Πρόβλημα: Ο Venus κρατά ένα αρχείο στο cache ακόμα και μετά του κλεισίματός του. Αν λάβει μια καινούργια αίτηση open() για το αρχείο, πως μπορεί να ξέρει αν το αντίγραφο στη cache είναι το πιο ενημερωμένο (πράγμα που απαιτείται για την εφαρμογή των AFS semantics).
  Λυση: Ο Venus ζητάει από το Vice ένα **callback promise**. Ο vice κρατάει

πληροφορίες για το ποιοί venus έχουν αποταμιεύσει το αρχείο. Αν κάποιο άλλο process ανοίξει το αρχείο για "write" σε άλλον πελάτη, τότε ο vice ειδοποιεί τους venus που έχουν αντίγραφο. ➔ ο Venus θα συνεχίσει να εξυπηρετεί read/write αιτήσεις χρησιμοποιώντας τη cache. Οταν όμως το session κλείσει το αρχείο διαγράφεται από τη local cache !!! ➔ μελλοντικά open() επικοινωνούν με τον Vice.

10.6.3 Ονομασία και προσπέλαση (Naming & Access)

AFS αρχεία (δηλ. αυτά στο /cmu dir.) χρησιμοποιούν ένα σχήμα ονομασίας διαφορετικό του UNIX.

Venus & Vice χρησιμοποιούν **fids** για να προσδιορίσουν την ταυτότητα αρχείων (αντί για inode numbers).

fid : < volume#, vnode#, uniquifier>
32 bits 32 bits 32 bits

Το volume# ορίζει ένα volume. Το vnode# ορίζει ένα file μέσα στο volume

- Naming: μεταφράζει ονόματα (ASCII strings) σε fids
  - Εκτελείται από το Venus όταν καλείται το open()
  - Αφού βρεθεί το fid του file ο Venus το στέλνει στο Vice.
  - Το Vice διατηρεί ένα πίνακα που συσχετίζει volumes με File Servers ➔ ο F.S. που έχει το volume που περιέχει το ζητούμενο αρχείο εντοπίζεται ➔ η αίτηση μεταβιβάζεται στον κατάλληλο vice, ο οποίος θα χρησιμοποιήσει τα volume# & vnode# για να βρεί το ζητούμενο αρχείο. ➔απαιτείται:

    1. μετάφραση ASCII string → fid
    2. μετάφραση (volume#, unode#) → UNIX inode.

Η παραπάνω διαδικασία εκτελείται για κάθε συστατικό μέρος ενός pathname ➔ είναι πολύ χρονοβόρα. Τελικά, το fid για το τελευταίο συστατικό μέρος του pathname επιστρέφεται στο Venus.

- Προσπέλαση: Εχοντας το fid, ο Venus:
  - εξετάζει τη τοπική cache
  - Αν η cache έχει ένα "έγκυρο" αντίγραφο, τότε επιστρέφει στο process έναν fd που "οδηγεί" στο cache αντίγραφο (τοπικό inode στο /cache).
  - Αν το αντίγραφο στη cache είναι "άκυρο" (invalid), ο Venus επικοινωνεί με το Vice που στέλνει το νεώτερο αρχείο (αν υπάρχει).
  - Αν η cache δεν έχει αντίγραφο, το Venus ζητά από το Vice να στείλει ένα ➔ Οταν επιστρέφει το open() υπάρχει πάντα ένα έγκυρο αντίγραφο στο /cache.
  - Read() + write() λειτουργούν όπως σε παραδοσιακά centralized UNIX συστήματα (δηλ. χωρίς την παρέμβαση των Venus, Vice).

- Close(): στέλνει ενημερωμένα αρχεία στον Vice.

- Κόστος του Naming:
  Για να αποφευχθεί η χρονοβόρα διαδικασία ο Venus έχει μία cache με μεταφράσεις ASCII pathnames σε fids. π.χ. (path1/foo1, fid1).

  Πρόβλημα: Αν το path1/foo1 διαγραφεί, το fid του μπορεί να "δοθεί" σε κάποιο άλλο καινούργιο αρχείο π.χ. path2/foo2. Ετσι ένα παλιό cache μπορεί να προκαλέσει το open(path1/foo1) αντί να επιστρέψει λάθος να ανοίξει το path2/foo2... Το πρόβλημα λύνεται μέσω του uniquifier. Οταν ένα <volume#, unode#> ανακυκλώνεται, το ανακυκλωμένο fid έχει διαφορετικό uniquifier.

Κεφάλαιο 11. Συντονισμός Γεγονότων σε Κατανεμημένα Συστήματα

11.1 Συγχρονισμός και Διάταξη Γεγονότων

[Lamport, Communication of ACM, July 1978 p558], είναι το κλασσικό άρθρο στη περιοχή.

Κάθε κόμβος του Κ.Σ. έχει και δικό του ρολόϊ (clock or timer) που συνήθως βασίζεται σε quartz κρυστάλλους. Η συχνότητα με την οποία ταλαντεύονται τα quartz κρύσταλλα είναι σταθερή. Το πρόβλημα προκύπτει διότι διαφορετικά quartz κρύσταλλα έχουν λίγο διαφορετικές συχνότητες

➔ ρολόγια σε διαφορετικούς κόμβους δείχνουν διαφορετικές τιμές (clock drift ή skew).

Δύο θεμελιώδεις ερωτήσεις:

1. Μπορούν να συγχρονιστούν όλα τα ρολόγια ενός Κ.Σ.; Δηλαδή μπορεί να υπάρξει καθολικός χρόνος (global time) ;
2. Πρέπει ο καθολικός χρόνος να αντιστοιχεί ακριβώς στον πραγματικό χρόνο ; Δηλαδή, αν όλοι συμφωνούν ότι η ώρα είναι 13:00 πειράζει αν είναι πράγματικά 13:01 ;

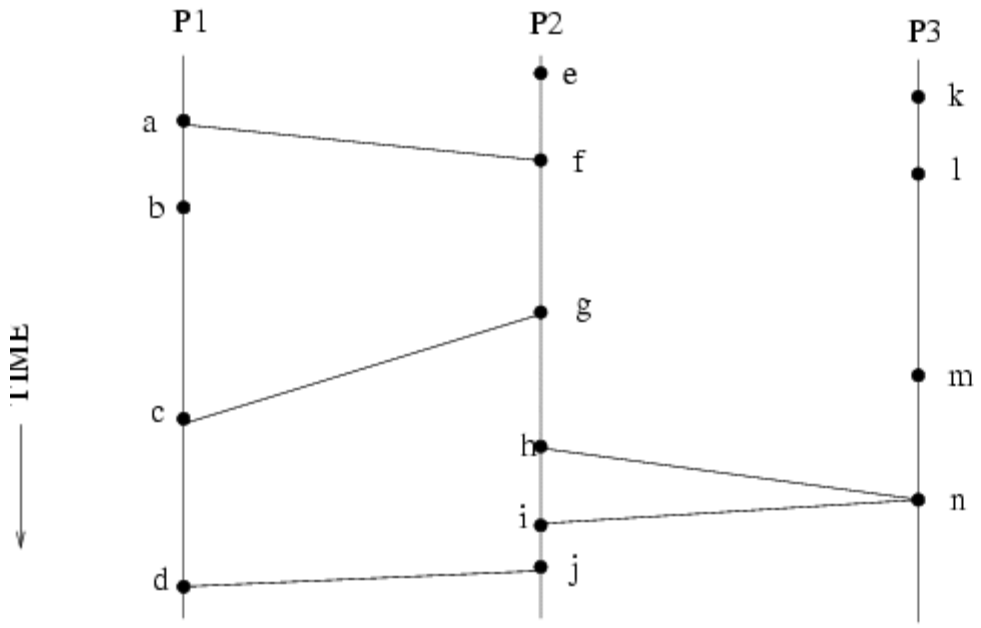
Πολλές εφαρμογές χρειάζονται μόνο την ύπαρξη καθολικού χρόνου (ας μην αντιστοιχεί στον πραγματικό χρόνο).

<u>Παράδειγμα</u> το πρόγραμμα make σε UNIX συστήματα. Οταν ο editor και ο compiler τρέχουν σε διαφορετικούς κόμβους, αυτό που ενδιαφέρει είναι αν το object file δημιουργήθηκε πριν ή μετά το source file (και όχι πότε ακριβώς δημιουργήθηκε). Ετσι, το make μπορεί ν' αποφασίσει αν χρειάζεται να κάνει compile το source file.

Ρολόγια που σχεδιάσηκαν ν' απαντούν στην πρώτη ερώτηση λέγονται «λογικά ρολόγια» (logical clocks). Τα λογικά ρολόγια χρησιμοποιούνται για να ορίσουν ή να εξακριβώσουν την σειρά με την οποία διάφορα γεγονότα συνέβησαν.

Τα «φυσικά ρολόγια» έχουν τον επιπλέον περιορισμό ελέγχου της απόκλισης του καθολικού χρόνου από τον πραγματικό.

Ο Lamport πρώτος έδειξε πως μπορεί να υπάρξει καθολικός χρόνος σε Κ.Σ. Η βασική του παρατήρηση είναι ότι αν 2 process δεν αλληλοεπηρεάζονται (interact) τότε δεν πειράζει αν τα ρολόγια τους δεν είναι συγχρονισμένα (γιατί η έλλειψη συγχρονισμού δεν "φαίνεται"). Ετσι, συγχρονισμός απαιτείται μόνο αν οι ενέργειες ενός process μπορούν να επηρεάσουν τις ενέργειες ενός άλλου process. Αυτό το φαινόμενο ονομάζεται causal effect/relation.



Ο συγχρονισμός του Lamport βασίζεται στην έννοια του happens-before relation (→).

- <u>a happens before b αν:</u>
    1. a & b είναι γεγονότα (events) στο ίδιο process και το a έλαβε χώρα πριν το b
    2. Αν το event a = msg_send() και το b είναι το αντίστοιχο msg_receive() τότε a → b
    3. Αν υπάρχει event c ώστε a → c & c → b τότε α → β (μεταβατική).
- <u>Δύο γεγονότα a & b λέγονται ταυτόχρονα (concurrent) αν a not → b & b not → a</u>

<u>Δημιουργία καθολικού χρόνου</u>

Βασίζεται στην επόμενη συνθήκη που ονομάζεται clock condition.

if a → b τότε c(a) < c(b) όπου c(e) συμβολίζει την ώρα, σύμφωνα με το global clock που το event e συνέβηκε.

Για να ικανοποιήσουμε το clock condition:

1. Ci(a) < Ci(b), αν το a συνέβη πριν το b στο process i.
2. Ci(a) < Cj(b), αν a=msg_send(i, j, msg) και b = msg_receive(i, j, msg), όπου Ci(e) συμβολίζει τον χρόνο που συνέβη το e στον κόμβο i σύμφωνα με το ρολόϊ του i.

Για να εγγυηθούν τα παραπάνω πρέπει:

1. Κάθε τοπικό (local) clock να "προχωρά" ενδιάμεσα σε δύο τοπικά γεγονότα.
2. Για κάθε msg από S στο R η ώρα της αποστολής στο ρολόϊ του S να είναι μικρότερη της ώρας λήψης του msg στο ρολόϊ του R.

Οι παρακάτω κανόνες διασφαλίζουν την ισχύ του clock condition

Κανόνες υλοποίησης

- IR1) Αύξηση την τιμή του Ci μετά από κάθε γεγονός στο process i
- IR2)
  - Κάθε msg εμπεριέχει την ώρα αποστολής, tCs, σύμφωνα με το ρολόϊ του S.
  - Με την λήψη του msg το process R ενημερώνει Cr ως εξής: Cr = max (Cr , tCs) + 1

Οι παραπάνω κανόνες υλοποίησης επιτρέπουν μια διάταξη των γεγονότων, μέσω του global timer C. Ομως πρόκειται για μερική διάταξη (partial order) (συσχετίζει κυρίως γεγονότα που συνδέονται με το happens-before relation).

Πώς μποορύμε ν' αποκτήσουμε μια ολική διάταξη (total ordering) ;

<u>Προσέξτε:</u> Αυτό το global time βασίζεται στο ότι:

αν a → b, ➔ C(a) < C (b)

Πώς μπορούμε να εγγυηθούμε το αντίστροφο; Υπόδειξη: Επανεξετάστε το CBCAST πρωτόκολλο.

11.2 Αμοιβαία Εξαίρεση σε Κ.Σ.

<u>Θεμελιώδης ερώτηση:</u> Πώς μπορούμε να συγχρονίσουμε την πρόσβαση σε διαμοιραζόμενα δεδομένα μεταξύ διεργασιών που εκτελούνται σε διαφορετικούς κόμβους σε Κ.Σ. ;

- Σε συστήματα με μια CPU η λύση περιλαμβάνει: monitors, semaphores, mutexes, κ.λπ. (critical region).
- Σε περιβάλλοντα με πολλές CPU που δεν έχουν διαμοιραζόμενη μνήμη, η υλοποίηση semaphores για παράδειγμα, χρειάζεται επανεξέταση.

11.2.1 Ενας κεντρικοποιημένος αλγόριθμος

<u>Βασικά Σημεία:</u>

- Υπάρχει μια διεργασία C που λειτουργεί σαν "συντονιστής πρόσβασης" (access coordinator).

- Κάθε διεργασία P που θέλει να μπεί στο critical section, στέλνει ένα μήνυμα "αίτηση πρόσβασης στην C".
- Αν η C δεν έχει ήδη επιτρέψει την πρόσβαση σε άλλη διεργασία, τότε επιστρέφει στην P ένα μήνυμα "αποδεκτή αίτηση".
- Αλλως η C είτε στέλνει ένα μήνυμα "απόρριψης" στην P, είτε την καθυστερεί μέχρι η διεργασία που βρίσκεται ήδη στο critical section, τελειώσει με αυτό. Τότε στέλνει ένα μήνυμα "αποδεκτή αίτηση" στην P.
- Η P μόλις λάβει το μήνυμα "αποδεκτή αίτηση", μπαίνει στο critical section, εκτελεί υπολογισμούς και όταν τελειώσει στέλνει ένα μήνυμα "τέλος πρόσβασης" στην C.

Κυρίως πρόβλημα: Αν η C καταρρεύσει, καμμιά διεργασία δεν μπορεί να μπει στο critical section.

11.2.2 Ενας αλγόριθμος βασισμένος σε token ring

Βασικά Σημεία:

- Ολες οι διεργασίες είναι "λογικά" τοποθετημένες σε μορφή δακτυλίου (ring).
- η διεργασία P0 αρχικά παίρνει το εισητήριο (token).
- Μια διεργασία Pi περιμένει μέχρι να της δοθεί το εισητήριο. Επειτα, εκτελεί υπολογισμούς στο critical section και μετά δίνει το εισητήριο στην P(i+1).
- Αν η P(i+1) θέλει να μπει στο critical section το κάνει και μετά στέλνει το token στην P(i+2), κ.λπ.

Προβλήματα:

- (α). Το token μπορεί να χαθεί.
- (β). Οι διεργασίες καταρρέουν (με ή χωρίς το token).

Στο (α) πρέπει να επαναδημιουργήσουμε το token. ΑΛΛΑ πώς γνωρίζουμε ότι χάθηκε το token ; Δεν το γνωρίζουμε !

11.2.3 Συγκατάθεση Πλειοψηφίας (Quorum Concencus)

Κεντρική ιδέα: Παίρνουμε άδεια από ένα "συμβούλιο" άλλων διεργασιών.

Το "συμβούλιο" μπορεί να είναι οποιοδήποτε σύνολο διεργασιών που περιέχει την πλειοψηφία των διεργασιών που μπορεί να χρειαστεί να μπούν στο critical section.

Ο αλγόριθμος είναι:

1. Στείλε αίτηση σε N διεργασίες.
2. Περίμενε μέχρι (N+1/2) διεργασίες σου δώσουν άδεια.

Θετικό στοιχείο: Απαιτείται πλειοψηφία προσβάσιμων διεγρασιών - κατάρρευση μιας και μόνο διεργασίας δεν επηρεάζει την πρόσβαση στο critical section από άλλες διεργασίες (τις περισσότερες φορές).

Ορθότητα: Δύο διεργασίες δεν πρέπει ποτέ να βρίσκονται στο critical section ταυτόχρονα.

Η ορθότητα εξασφαλίζεται ως εξής:

- Οταν μια διεργασία δίνει άδεια πρόσβασης σε κάποιο πόρο (resource) σε μια άλλη διεργασία, θέτει ένα lock σχετικό με το resource.
- Οταν μια διεργασία βγαίνει από το critical section, στέλνει ένα μήνυμα στις διεργασίες που του είχαν δώσει άδεια να ελευθερώσουν τα lock τους.
- Μια διεργασία δίνει άδεια μόνο αν δεν έχει θέσει το αντίστοιχο lock.
- Τα παραπάνω λέγονται "συγκατάθεση πλειοψηφίας"

11.2.4 Coteries (γενίκευση του quorum consensus)

Cotery είναι ένα σύνολο από quorums, καθένα από τα οποία αντιπροσωπεύει μια λίστα διεργασιών από τις οποίες χρειαζόμαστε άδεια.

Π.χ. Αν όλες οι διεργασίες είναι οι: a, b, c, d, e τότε το cotery για την συγκατάθεση πλειοψηφίας είναι: { {a, b, c}, {a, b, d}, {a, b, e}, {a, c, d}, {a, c, e}, {b, c, d}, {b, c, e} }.

Γενικά ένα cotery C αποτελείται από σύνολα Q1, Q2, ..., QN με τις εξής ιδιότητες:

- $\forall$ i,j: $Q_i \cap Q_j \neq \emptyset$ (χρειάζεται για ορθότητα)
- $\forall$ i,j: $Q_i \not\subset Q_j$ (λόγοι απόδοσης)

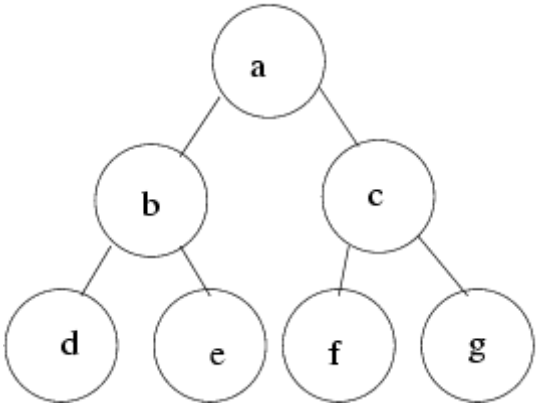
Ενα cotery δυαδικού δένδρου

Βασική Ιδέα:

Κατασκευάζουμε ένα πλήρες δυαδικό δένδρο από τις διεργασίες. Κατασκευάζουμε quorums ως εξής:

- Κατασκευάζουμε ένα quorum αποτελούμενο από όλους τους κόμβους σ' ένα μονοπάτι από τη ρίζα σε κάποιο φύλλο, για κάθε μονοπάτι.
- Για κάθε μονοπάτι που ξεκινάει στο επίπεδο i (i>1) και τελειώνει σε κάποιο φύλλο, κατασκευάζουμε ένα quorum από την ένωση των κόμβων στο μονοπάτι και το σύνολο των κόμβων του ίδιου επιπέδου.
- Κατασκευάζουμε να quorum από όλα τα φύλλα.

Παραδειγμα: Ας υποθέσουμε ότι έχουμε τους εξης 7 κόμβους, a, b, c, d, e, f, g.



C = { {a, b, d}, {a, b, e}, {a, c, f}, {a, c, g}, {b, d, c}, {b, e, c}, {c, f, b}, {c, g, b}, {d, e, f, g} }
Αντί να ζητάμε άδεια από 4 διεργασίες (όπως με τη προηγούμενη μέθοδο), ζητάμε μόνο από 3.

11.3 Αδιέξοδα σε Κ.Σ.

11.3.1 Γενικά

Υπάρχουν 4 τρόποι αντιμετώπισης των αδιεξόδων

- να μην κάνεις τίποτα (! δημοφιλής) -- (αφήνει την ευθύνη στην εφαρμογή)
- ανίχνευση και ανάρρωση (δημοφιλής: άσε να συμβεί το αδιέξοδο, ανίχνευσέ το και λύστο)
- πρόληψη (δημοφιλής: στατιστικά κάνε τα αδιέξοδα απίθανα)
- αποφυγή: (μη πρακτικό: πριν δώσεις ένα πόρο σε μια διεργασία χρειάζεσαι πληροφορία για το πώς οι άλλες διεργασίες σκοπεύουν να χρησιμοποιήσσουν τον ίδιο πόρο. Αν αυτή η πληροφορία είναι διαθέσιμη τότε μπορεί να προσδιοριστεί το εάν δίνοντας το resource σε μια διεργασία οδηγηθούμε σε σταθερή/ασταθή κατάσταση. Το πρόβλημα είναι ότι αυτή η πληροφορία είναι σπανίως διαθέσιμη.

11.3.2 Ανίχνευση και ανάρρωση

Α. Κεντρικοποιημένη προσέγγιση.

Βασικές Αρχές:

1. Ασε κάθε κόμβο στο Κ.Σ. να διατηρεί το δικό του γράφο resource-process για τις τοπικές διεργασίες και resources. Αυτός ο γράφος δείχνει ποιά διεργασία κατέχει ποιό resource (πόρο) και ποιές διεργασίες περιμένουν για ποιά resources.
2. Ενας κεντρικός κόμβος διατηρεί την ένωση όλων των τοπικών γράφων.

Τα προβλήματα μ' αυτήν την προσέγγιση είναι:

- ο κεντρικός κόμβος μπορεί να γίνει σημείο συμφόρησης (bottleneck)

- είναι πιθανό να καταλήξει σε λάθος συμπεράσματα (ανύπαρκτα αδιέξοδα, false deadlocks) εξ αιτίας χαμένων μηνυμάτων ή αναδιαταγμένων (reorded) μηνυμάτων.

Β. Κατανεμημένες προσεγγίσεις -- έχουν προταθεί αλλά κοστίζουν (μη πρακτικές ;)

11.3.3 Πρόληψη

Γνωστές χρήσιμες τεχνικές περιλαμβάνουν:

- προαπόκτηση αποκλειστικής πρόσβασης σ' όλους τους πόρους,
- διάταξη όλων των πόρων και υποχρέωση κάθε διεργασίας να ζητά τους πόρους με αύξουσα (ή φθίνουσα) σειρά.

Τα παραπάνω δεν είναι και πολύ πρακτικά.

Οι δοσοληψίες (transactions) και ο καθολικός χρόνος (global time) απλοποιούν την πρόληψη. Βασικές αρχές:

1. κάθε διεργασία συνδέεται με μία μοναδική χρονοσφραγίδα (timestamp) κατά τη στιγμή της δημιουργίας.
    a. Αν η διεργασία Π1 προσπαθήσει να πάρει ένα πόρο που κατέχει η Π2, τότε η Π1 θα μπλοκάρει εάν είναι παλαιότερη της Π2 ή
    b. η P1 αναμένει ένα είναι νεότερη από την P2.
2. μια διεργασία που δεν αναμένει, πεθαίνει.

Κατ' αυτόν τον τρόπο, κάθε αλυσίδα μπλοκαρισμένων διεργασιών οδηγεί σε νεότερες (ή γηραιότερες) διεργασίες και γι αυτό ένας κύκλος (αδιέξοδο) δεν δύναται να σχηματισθεί.!

Είναι προτιμότερο να δίνουμε προτεραιότητα στις παλιότερες διεργασίες.

Ο αλγόριθμος wound-wait

- Εάν P1 < P2 τότε σκότωσε την P2 (εάν είναι δυνατό) (η παλιότερη σκοτώνει την νεότερη),
- αλλιώς η P1 περιμένει την P2 (η νεότερη περιμένει)

Εάν η P2 είναι δοσοληψία που βρίσκεται στη διαδικασία commit τότε δεν μπορείς να σκοτώσεις την P2 - γι αυτό λέμε "wound" αντί "kill".

Ο αλγόριθμος "wait-die"

- Αν P1 < P2 τότε η P1 περιμένει (η παλαιότερη περιμένει) αλλιώς η P1 πεθαίνει (η νεότερη πεθαίνει)

η P1 θα πρέπει να ξαναρχίσει με το ίδιο timestamp - διαφορετικά δεν θα μπορέσει ποτέ να γίνει αρκετά παλαιά.

Στο "wait-die" η P1 είναι δυνατό να ξαναρχίσει αρκετές φορές (εώς ότου γίνει αρκετά παλαιά). Θα μπορούσε κάτι τέτοιο να γίνει στο "wound-wait" ; (Απάντηση: μόνο αν υπήρχαν αρκετές παλιές διεργασίες που θα έκαναν wound μια διεργασία).

Στο "wait-die" ωστόσο, μια διεργασία θα επιτραπεί να τρέξει μέχρι το τέλος εφόσον ζητήσει όλους τους πόρους - (κάτι που δεν μπορεί να εγγυηθεί ο αλγόριθμος "wound-wait")

Ωστόσο, στο "wound-wait" μπορεί να τροποποιηθεί ώστε να σκοτώνεται μια διεργασία μόνο κάτω από συγκεκριμένες συνθήκες.

Κεφάλαιο 12. Συνέπεια σε Κατανεμημένα Συστήματα

12.1 Εισαγωγή σε Συνδιαλλαγές (Transactions)

Δυνητικά, δύο γεγονότα μπορούν να προκαλέσουν ασυνέπεια (inconsistency) σ' ένα σύστημα:

- Βλάβες (failures)
- Ταυτοχρονισμός (concurrency)

<u>Παράδειγμα.</u>
Θεωρείστε ένα σύστημα για μια τράπεζα, όπου data = δεδομένα για λογαριασμούς πελατών. Εστω X και Y 2 λογαριασμοί. Θεωρείστε 2 προγράμματα P1 και P2 όπου το P1 μεταφέρει χρήματα από τον X στον Y, ενώ το P2 απλώς πληροφορείται για το υπόλοιπό των.

P1	P2
X ← X - amount;	read (X);
Y ← Y + amount;	read (Y);

1) X' = read(X)
2) X = X' - amount
3) Y' = read (Y)
4) Y = Y' + amount

Αν ο κόμβος όπου τρέχει το Π1 καταρεύσει αμέσως μετά την εκτέλεση της εντολής (2), τότε το σύστημα είναι φυσικά ασυνεπές.
Επίσης αν τα Π1 και Π2 τρέχουν ταυτόχρονα, οι εντολές τους μπορεί να εκτελεστούν με την εξής σειρά:
X' = read(X); X = X' - ποσό; read(X); read(Y); Y' = read(Y); Y = Y' + ποσό.
Με αυτή τη σειρά εκτέλεσης (schedule) ασυνεπής πληροφορία δίνεται στον χρήστη!

Για να ελεγχθούν οι παραπάνω "πηγές" ασυνέπειας δημιουργήθηκαν οι συνδιαλλαγές (transactions).

Θεμελιώδη χαρακτηριστικά:

- Μια ομάδα εντολών read/write περικλεισμένες από Begin Trans., Commit Trans. (ή Abort Trans.)
- Κάθε transaction έχει 4 ιδιότητες:
  - Atomicity (Ατομικότητα): Είτε όλες οι εντολές ενός transaction είτε καμμιά εντολή του επηρεάζουν την κατάσταση της βάσης δεδομένων.
  - Consistency (Συνέπεια): Κάθε transaction φέρνει το σύστημα από μια συνεπή κατάσταση σε μια άλλη συνεπή κατάσταση.

- Isolation (Απομόνωση): Τα ενδιάμεσα αποτελέσματα ενός transaction (δηλ. τ' αποτελέσματα των εντολών του πριν γίνει commit) δεν είναι ορατά από τ' άλλα transactions που τρέχουν ταυτόχρονα.
- Durability (Διάρκεια ή Permanence): Αφ' ής στιγμής ένα transaction γίνει commit, όλες οι καινούργιες τιμές που εγκατέστησε στη βάση δεδομένων (δηλ. όλες οι ενημερώσεις του) δεν δύναται να χαθούν.

Αυτές είναι οι λεγόμενες ACID ιδιότητες των transactions.

Οι ιδιότητες των Atomicity & Durability διαχειρίζονται failures. Consistency & Isolation διαχειρίζονται ταυτοχρονισμό μέσω αλγορίθμων concurrency control.

12.2 Ατομικότητα Συνδιαλλαγών σε Κ.Σ.

Transactions που προσπελαύνουν αντικείμενα σε ένα Κ.Σ. λέγονται distributed transactions ➔ προσπελαύνονται διάφορα αντικείμενα που είναι αποθηκευμένα σε διάφορους κόμβους.

Θα περιγράψουμε την εκτέλεση ενός distributed transaction υποθέτοντας ότι:

- στον κόμβο όπου δημιουργείται το transaction (ή όπου καλείται) υπάρχει ένα coordinator process που επιβλέπει συνολικά την εκτέλεση
- σε κάθε κόμβο του Κ.Σ. που προσπελαύνεται από το transaction υπάρχει ένα βοηθητικό (cohort) process που επιβλέπει την τοπική εκτέλεση του transaction.
- Ο coordinator στέλνει την κάθε εντολή του transaction στον κατάλληλο cohort ο οποίος επιστρέφει τ' αποτελέσματα (αν υπάρχουν).
- Για να διασφαλιστεί η ατομικότητα ενός distributed transaction πρέπει: είτε να εκτελεστούν όλες οι εντολές του (από όλους τους συμμετέχοντες κόμβους) είτε να μην εκτελεστεί καμμιά εντολή του transaction (σε κανέναν κόμβο).

  ➔ το βασικό πρόβλημα είναι πρόβλημα συμφωνίας (agreement): πρέπει όλοι οι cohorts να πάρουν την ίδια απόφαση commit/abort για το transaction. Αυτά διασφαλίζεται με ειδικά πρωτόκολλα που είναι γνωστά σαν atomic commit protocols.

12.2.1 Κεντρικοποιημένο Πρωτόκολλο Ατομικότητας 2 Φάσεων (Centralized Two-Phase Commit Protocol).

Στόχος είναι όλοι οι cohort να πάρουν την ίδια απόφαση commit/abort. Πρέπει αυτό να επιτευχθεί ακόμα και υπό την παρουσία βλαβών (κόμβων, επικοινωνιακού συστήματος).

Αρκεί μια φάση μόνο ; Γιατί δύο φάσεις ;

Γενική περιγραφή πρωτοκόλλου

- 1η φάση:
  - Coordinator στέλνει ένα "prepare-to-commit" msg σ' όλους τους cohort.
  - Κάθε cohort απαντά αν είναι διατεθειμένος να κάνει commit το transaction.
- 2η φάση:
  - Ο coordinator συλλέγει τις απαντήσεις: Αν όλοι οι cohort απάντησαν, και απάντησαν θετικά, ο coordinator αποφασίζει commit και στέλνει ένα commit msg στους cohort.
  - Αν κάποιος cohort απάντησε αρνητικά, ή δεν απάντησε, ο coordinator στέλνει ένα abort msg στους cohort.
  - Ολοι οι cohort εφαρμόζουν το commit/abort, όπως τους υπέδειξε ο coordinator με το μήνυμά του.

Πρέπει όλα τα εμπλεκόμενα process να φτάσουν στην ίδια απόφαση ακόμα και όταν συμβαίνουν βλάβες ➔ Χρειαζόμαστε:

- stable storage: κρατάει πληροφορία σχετικά με την κατάσταση ενός transaction (π.χ. executing, commiting, commited, aborted, κ.λπ) και τις αλλαγές στην κατάστασή της Β.Δ. που επιφέρει η εκτέλεση του transaction
- ένα πρωτόκολλο που πάντα οδηγεί όλα τα processes στην (ίδια) απόφαση ακόμα και αν το επικοινωνιακό σύστημα πάθει βλάβη κατά την εκτέλεση του πρωτοκόλλου.

<u>Ο αλγόριθμος του Συντονιστή (Coordinator):</u>

- στείλε "prepare" msg στους cohort;
- περίμενε για απαντήσεις (Ναι/Οχι) από όλους: άρχισε το timeout.
- Αν (ο χρόνος του timeout παρέλθει) ή (τουλάχιστον ένα cohort απήντησε Οχι) τότε
  - γράψε abort record σ' ένα stable storage log
  - στείλε το "αbort" msg στους cohort
  - return
- Αλλιώς (<u>όλοι</u> απήντησαν <u>θετικά</u>):
  [CP]: γράψε commit record στο stable storage log
  - στείλε το "commit" msg στους cohort
  - return

Η εντολή [CP] είναι το commit point: μετά την εκτέλεσή της, ότι κι αν προκύψει το distributed transaction θα γίνει δεκτό από όλους!!!

Ο Αλγόριθμος των Πρακτόρων (Cohort)

- περίμενε για το "prepare" msg από coordinator; Μετά την λήψη του "prepare" msg:
    - Αν ο cohort "ψηφίζει" Ναι:
        - γράψε ένα "prepared" record στο τοπικό stable storage log.
        - στείλε ένα "Ναι" msg στον coordinator
        - περίμενε για την απάντηση του coordinator. Αν η απάντηση είναι commit, γράψε commit record στο τοπικό log.
        - Αλλιώς γράψε abort record στο τοπικό log
    - Αν ο cohort "ψηφίζει" ΟΧΙ:
        - γράψε abort record στο log
        - στείλε "ΟΧΙ" msg στον coordinator
- return.

Γιατί κάποιος cohort να ψηφίσει "ΟΧΙ" ; (integrity constraints, deadlocks, ...)

Κατά τη διάρκεια της ανάνηψης (recovery) ένα ειδικό recovery process εξετάζει το log (στο stable storage) για να δει αν υπάρχουν τα commit/ abort/prepare records. Ετσι μπορεί ν' αποφασισθεί αν το transaction έχει γίνει commit ή abort.

12.2.2 Απαιτήσεις για Atomic Commit Protocols

AC1: Όλα τα process που αποφασίζουν, λαμβάνουν την ίδια απόφαση.

AC2: Κανένα process δεν μπορεί ν' αντιστρέψει την αρχική του απόφαση.

AC3: Η απόφαση commit μπορεί να ληφθεό ΜΟΝΟ αν ΟΛΑ τα process ψήφισαν ΝΑΙ.

AC4: Υπό την απουσία βλαβών: Αν όλα τα process ψήφισαν ΝΑΙ ➔ η απόφαση θα είναι "commit".

AC5: Αν όλες οι βλάβες έχουν διορθωθεί και δεν συμβούν καινούργιες βλάβες, τότε όλα τα process τελικά θα καταλήξουν σε μια απόφαση.

AC1 ➔ μπορεί (θεωρητικά) κάποιο process να μην καταλήξει ποτέ σε απόφαση (μόνιμη βλάβη).

Προσέξτε το AC4, σε συνδυασμό με το αντίστροφο του AC3 (Δηλ. είναι δυνατόν όλοι να ψηφίσουν ΝΑΙ και η απόφαση να είναι äbort"
AC2 + AC3 ➔ ότι ένα process μπορεί από μόνο του να προκαλέσει abortion του transaction (αρκεί να μην έχει ψηφίσει ΝΑΙ).

Μερικοί σημαντικοί όροι:

- Περίοδος αβεβαιότητας (Uncertain Period): ένας cohort είναι σε αβεβαιότητα αφού ψηφίσει NAI και πριν λάβει την απόφαση.
- Ανεξάρτητη ανάνηψη (Independent Recovery): Η δυνατότητα ενός κόμβου ν' ανανήψει χωρίς ανάγκη επικοινωνίας μ' άλλα process (Αν ενας cohort είναι uncertain είναι αδύνατη η ανεξάρτητη ανάνηψη).
- Blocking: Αν βλάβες εμποδίζουν ένα certain process P να ... βεβαιωθεί, τότε λέμε ότι το P είναι blocked.

Γεγονός 1: Αν το επικοινωνιακό σύστημα μπορεί να πάθει βλάβη (ή όλοι οι κόμβοι να καταρρεύσουν ταυτόχρονα) τότε δεν υπάρχει ACP πρωτόκολλο που να αποφεύγει το blocking.

Γεγονός 2: Κανένα πρωτόκολλο ACP δεν μπορεί να εγγυηθεί την ανεξάρτητη ανάνηψη.

Το 2PC πρωτόκολλο που παρουσιάσαμε σαφώς ικανοποιεί τους όρους AC1 - AC4. Για να ικανοποιηθεί ο AC5 χρειαζόμαστε ένα Termination protocol (TP). Ενας cohort εκτελεί το TP εφού ψηφίσει NAI και ο χρόνος του timeout παρέλθει χωρίς να λάβει την απόφαση του coordinator.

Termination Protocol (Πρωτόκολλο Τερματισμού)

Ο αβέβαιος cohort είναι ο ïnitiator". Τα άλλα process είναι "responder"

Αλγόριθμος Initiator

- στείλε ένα "DECISION_REQ" msg σ' όλα τα process. if timeout repeat
    - Αν η απόφαση = commit, γράψε commit record στο log
    - Αλλιώς, γράψε abort record στο log

Αλγόριθμος Responder

- Αν δεν έχει ψηφίσει NAI ή αποφάσισε äbort" στέλνει äbort" απάντηση στον initiator
- Αλλιώς, αν αποφάσισε "commit" στέλνει "commit" απάντηση στον initiator
- Αλλιώς δεν μπορεί να βοηθήσει: στέλνει «sorry..."

12.3 Ανάνηψη (Recovery)

Ο coordinator πρέπει να στείλει την λίστα με τους συμμετέχοντες (processes) σε κάθε cohort, έτσι ώστε όλοι οι cohort να μπορούν ν' εκτελέσουν το TP.

Αλγόριθμος Ανάνηψης του Coordinator

- εξέτασε το log
- Αν υπάρχει commit record για το transaction (re)multicast το commit msg στους cohort
- Αλλιώς, (re)multicast το abort msg στους cohort.

Αλγόριθμος Ανάνηψης Cohort

- εξέτασε το log
- Αν το log δείχνει ένα commit/abort record: εφάρμοσε/εκτέλεσε την εντολή commit/abort.
- Αν στο log υπάρχει "prepared" record αλλά ούτε "commit" ούτε äbort" record, τότε: κάλεσε το TP
- Αλλιώς (το log δεν έχει καν "prepared" record: προκάλεσε το abortion του transaction.

WriteAhead Logging (WAL)

- Κάθε φορά που ένα Tid εκτελεί ένα write(x), πριν ενημερωθεί η τιμή του X στη Β.Δ. προσθέτεται ένα record στο log της μορφής < Tid, x, παλιά_τιμή_του_ x, καινούργια_τιμή_του_x>
- Έτσι, αν η απόφαση είναι abort, μπορεί να ελεγχθεί αν η τιμή του x στη Β.Δ. είναι ενημερωμένη και αν ναι, ν' αντικατασταθεί με τη παλιά τιμή.
- Ομοίως, αν η απόφαση είναι commit η καινούργια τιμή του x υπάρχει στο log και μπορεί ν' αντικαταστήσει την τιμή στη Β.Δ.

12.4 Ελεγχος Ταυτοχρονισμού [Concurrency Control]

Οπως είδαμε οι ιδιότητες consistency + isolation διασφαλίζουν την συνέπεια του συστήματος υπό την παρουσία ταυτοχρόνων συνδιαλλαγών.

> ➔ το transaction είναι υψηλότερου-επιπέδου μηχανισμός ελέγχου ταυτοχρονισμού (απ' ότι semaphores, critical sections, monitors, ...)

Υπάρχουν μυριάδες αλγόριθμοι για conc. control (cc). Ιδιαίτερα για Σ.Β.Δ. όλοι οι αλγόριθμοι εγγυώνται την ιδιότητα *serializability*: Ενα schedule το οποίο επιτρέπεται από κάποιον σωστό αλγόριθμο C.C. έχει ένα συνολικό αποτέλεσμα (δηλ. φέρνει την Β.Δ. σε μια κατάσταση) που μπορεί να παραχθεί αν τα ίδια transactions εκτελούνταν σειριακά: δηλ. πρώτα όλες οι εντολές ενός trans. , μετά όλες οι εντολές ενός άλλου, κ.ο.κ.

Οι πιο δημοφιλείς /πολύ διαδεδομένοι αλγόριθμοι C.C. βασίζονται στις έννοιες read lock/write lock.

Επίσης:

- πριν ένα trans. read (ή write) κάποιο αντικείμενο x τότε ζητά από το σύστημα ένα read (ή write) lock για το x.
- Τα read locks είναι διαμοιραζόμενα (shared.
- Τα write locks είναι αποκλειστικά (exclusive).

Συμβατότητα locks:

O = owner. R = requester

O \ R	R	W
R	√	X
W	X	X

Ενα σημαντικό θέμα είναι πότε μπορούν να "ελευθερωθούν" κατεχόμενα locks από ένα transaction.

<u>Ο C.C. αλγόριθμος 2-Phase Locking (2PL)</u>

- Η εκτέλεση ενός transaction χωρίζεται σε 2 φάσεις:

- Στη μια φάση (growing) το transaction μπορεί να αποκτά τα απαραίτητα locks.
- Στην άλλη φάση (shrinking) το transaction μπορεί μόνο να ελευθερώσει locks.
- Ενα transaction εισέρχεται στο shrinking phase μετά την ελευθέρωση του πρώτου lock. Σ' αυτή την φάση δεν μπορεί πλέον να αποκτήσει κάποιο καινούργιο lock.

Εχει αποδειχθεί ότι αν όλα τα transactions ακολουθούν το 2PL τότε όλα τα schedules που παράγονται από το σύστημα είναι serializable (δηλ. αντιστοιχούν σε κάποιο σειριακό schedule ➔ είναι σωστά).

Στο Κ.Σ. ένα distributed trans. αποκτά locks σε >1 κόμβους.

Πρακτικά: 2PL σημαίνει ότι για να ελευθερωθούν locks ενός trans. το σύστημα πρέπει να ξέρει ότι το trans. δεν θα ζητήσει πρόσβαση σε κάποιο άλλο αντικείμενο ➔ το trans. έχει τελειώσει μ' όλες τις εντολές του. Πώς το ξέρει αυτό το σύστημα: Οταν αρχίσει π.χ. το ACP.

Κεφάλαιο 13. Μοντέρνα Συστήματα I/O ή Συστήματα για Παράλληλη Είσοδο/Εξοδο

13.1 Εισαγωγή

Πολλές μοντέρνες εφαρμογές χρειάζονται μεγάλους αποθηκευτικούς χώρους και μεγάλο I/O bandwidth. (π.χ. ψηφιακές βιβλιοθήκες, πολυμέσων, ...)

Είναι γενικά αποδεκτό ότι το μεγαλύτερο bottleneck μοντέρνων υπολογιστικών συστημάτων είναι το I/O. Αυτό οφείλεται στο ότι:

- Η απόδοση των CPU chips αυξάνει κατα > 40 % κάθε χρόνο για τα τελευταία 20 περίπου χρόνια.
- Την ίδια στιγμή, η χωρητικότητα των memory chips αυξάνεται με μεγαλύτερους ή περίπου τους ίδιους ρυθμούς (η χωρητικότητα τετραπλασιάζεται κάθε 2-3 χρόνια).
- Ανάλογα περίπου αυξάνεται και η χωρητικότητα μαγνητικών δίσκων (διπλασιάζεται κάθε 3 χρόνια περίπου).
- Αλλά μόνο η αύξηση της χωρητικότητας δεν αρκεί !!! Η απόδοση των memories έχει βελτιωθεί (δηλ. ο χρόνος που απαιτείται για να φορτωθεί το CPU με data και εντολές) κυρίως λόγω cache memories και της βελτιωμένης τεχνολογίας SRAM που αποτελούν τις cache memories (> 40 % βελτίωση κάθε χρόνο).
- <u>ΟΜΩΣ</u> : η απόδοση μαγνητικών δίσκων δεν βελτιώνεται με τους ίδιους ρυθμούς, π.χ. το seek time βελτιώνεται με ρυθμούς ~ 7% το χρόνο. Ενώ το rotational delay υποδιπλασιάζεται κάθε 10 χρόνια

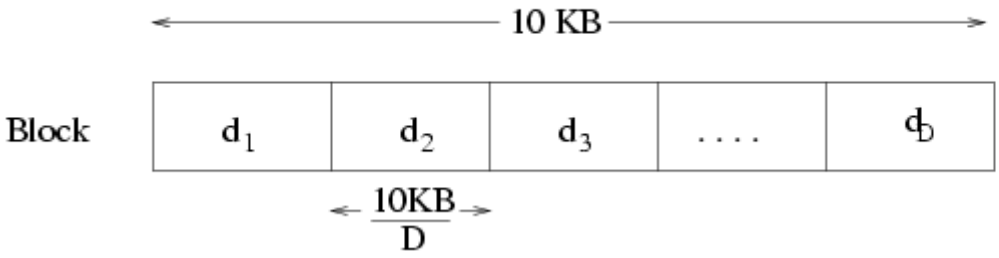
  ➔ I/O performance είναι το bottleneck.

<u>Σημείωση</u> : Εφαρμογές που διέπονται από locality of reference και για τις οποίες η αστάθεια της RAM δεν είναι πρόβλημα, μπορούν να χρησιμοποιήσουν μεγάλα RAM cache buffers και έτσι να αποφεύγουν πολλά I/O.

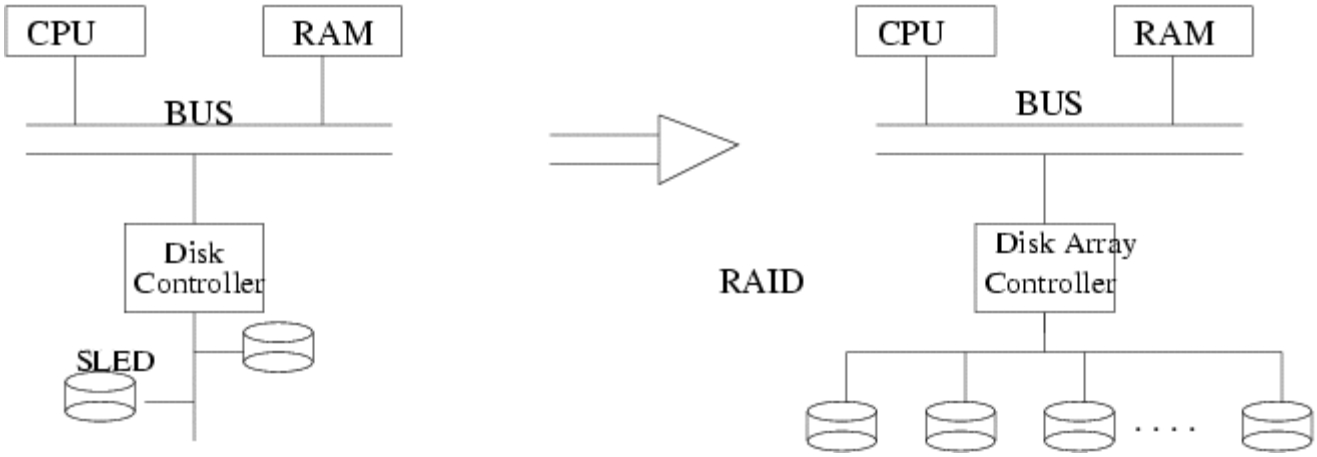
Μια δημοφιλής λύση : Disk Arrays & Striping.

<u>Βασική Ιδέα:</u>

- ομαδοποίησε ανεξάρτητα disk drives σε (λογικές) ομάδες δημιουργώντας disk arrays.
- κάθε block πληροφορίας κατανέμεται στους δίσκους ενός disk array π.χ. έστω disk array με D δίσκους και block size = 10 Kbytes. Η πληροφορία κατανέμεται ως εξής :



- Αυτή η διαδικασία ονομάζεται striping. Το καθένα από τα D κομμάτια του block ονομάζεται stripe unit (ή απλώς stripe).
- Το ith stripe γράφεται στον (ith mod D) δίσκο
- Ετσι



Τα πλεονεκτήματα των disk arrays προέρχονται κυρίως από την δυνατότητα παράλληλης πρόσβασης στους D δίσκους. Ετσι, είναι σαν να πολλαπλασιάζεται το transfer rate του I/O συστήματος κατα D και ανάλογα μειώνεται το κόστος πρόσβασης (δηλ. για ένα seek time και ένα rot. delay το σύστημα τώρα μπορεί να δώσει D φορές μεγαλύτερη πληροφορία απ' ότι πρίν!)

Rotational Synchronization : Ιδιότητα με την οποία όλοι οι δίσκοι ενός array περιστρέφονται συγχρονισμένα (δηλ. οι κεφαλές βρίσκονται πάνω από την ίδια θέση την ίδια στιγμή).

Fine - grained Striping (FGS): Η τεχνική striping κατά την οποία το κάθε block εμπλέκει όλους τους δίσκους του array (π.χ. το μέγεθος του stripe unit είναι μικρό) ➔ σε κάθε I/O συμμετέχουν όλοι οι δίσκοι του array.

Coarse - grained Striping (CGS): Ένα υποσύνολο των δίσκων του array εμπλέκονται σε κάθε I/O

➔ Μεγάλα I/O requests με μεγάλα blocks χρησιμοποιούν D δίσκους
Μικρότερα I/O requests με μικρά blocks χρησιμοποιούν < D δίσκους

➔ Με FGS όλο το I/O σύστημα μπορεί να εξυπηρετεί μόνο ένα I/O request. Αλλά, υπάρχει load balancing στους δίσκους

Stripe width, W: Πόσοι από τους D δίσκους χρησιμοποιούνται για κάθε block;

Μεγαλο W ➔ + load balancing
+ υψηλό disk utilization
Μικρό W ➔ + το array εξυπηρετεί πιο πολλές αιτήσεις ταυτόχρονα
+ μικρότερος αριθμός δίσκων εμπλέκεται
στο overhead ενός I/O

- Στην εργασία [Triantafillou, Faloutsos] υπολογίζεται το βέλτιστο stripe width W.
- Ο μεγάλος εχθρός disk arrays : Reliability (Αξιοπιστία). $MTTF_1$=(Mean time to failure) για ένα μοντέρνο δίσκο είναι = 200K - 1000 ώρες (= 20 - 100 χρόνια).

  MTTF για ένα array D δίσκων = MTTF = [($MTTF_1$)/D]

  Προσέξτε ότι με την τεχνική striping μία βλάβη σε οποιοδήποτε απ' τους D δίσκους προκαλεί ζημιά σε ΟΛΑ τα αρχεία.
  ➔ απαιτούνται πλεονάζοντα αντίγραφα για κρίσιμες πληροφορίες (redundancy, replication).

- RAID Levels
  - RAID Level 1 : Mirrored disks (σαν αυτούς που είδαμε στην υλοποίηση του stable storage)
    - κάθε δίσκος απαιτεί και έναν δίσκο "αντίγραφο"
  - RAID Level 2 : Hamming Code.
    - Το κάθε block γίνεται bit-interleaved σε G δίσκους ενός array (δηλ. το stripe unit size = 1 bit).
    - Μετά προστίθενται και όσοι check disks χρειάζονται για να είναι δυνατόν να ανιχνευθεί και να διορθωθεί ένα λάθος. Αν G=10 τότε χρειάζονται 4 check disks Αν G=25 τότε χρειάζονται 5 check disks

    αίτηση ανάκτησης πληροφορίας με μέγεθος μικρότερο από G sectors απαιτεί την ανάκτηση παραπανίσιας πληροφορίας αφού ο κάθε δίσκος ανακτά τουλάχιστον ένα sector.

    Αίτηση εγγραφής πληροφορίας μικρότερου μεγέθους από G sectors απαιτεί :

- ανάκτηση G sectors
- merge νέας και παλιάς πληροφορίας
- ενημέρωση check bits

- <u>RAID Level 3 :</u> Ενας check δίσκος / parity disk.
  - Μοντέρνοι controllers μπορούν να μας πληροφορήσουν ποιός δίσκος έπεσε ➔ δεν χρειάζονται πολλοί check disks.
  - Η πληροφορία σε ένα failed disk επισκευάζεται με το να
    1. υπολογίσουμε το parity των "καλών" δίσκων
    2. και να το συγκρίνουμε με το αντίστοιχο parity στο parity disk.

    Αν τα parity είναι ίδια ➔ failed bit = 0

    Αλλιώς ➔ failed bit = 1.

  - Writes πρέπει να προσπελάσουν και τον parity disk.
  - Reads πρέπει επίσης να προσπελάσουν τον parity disk.

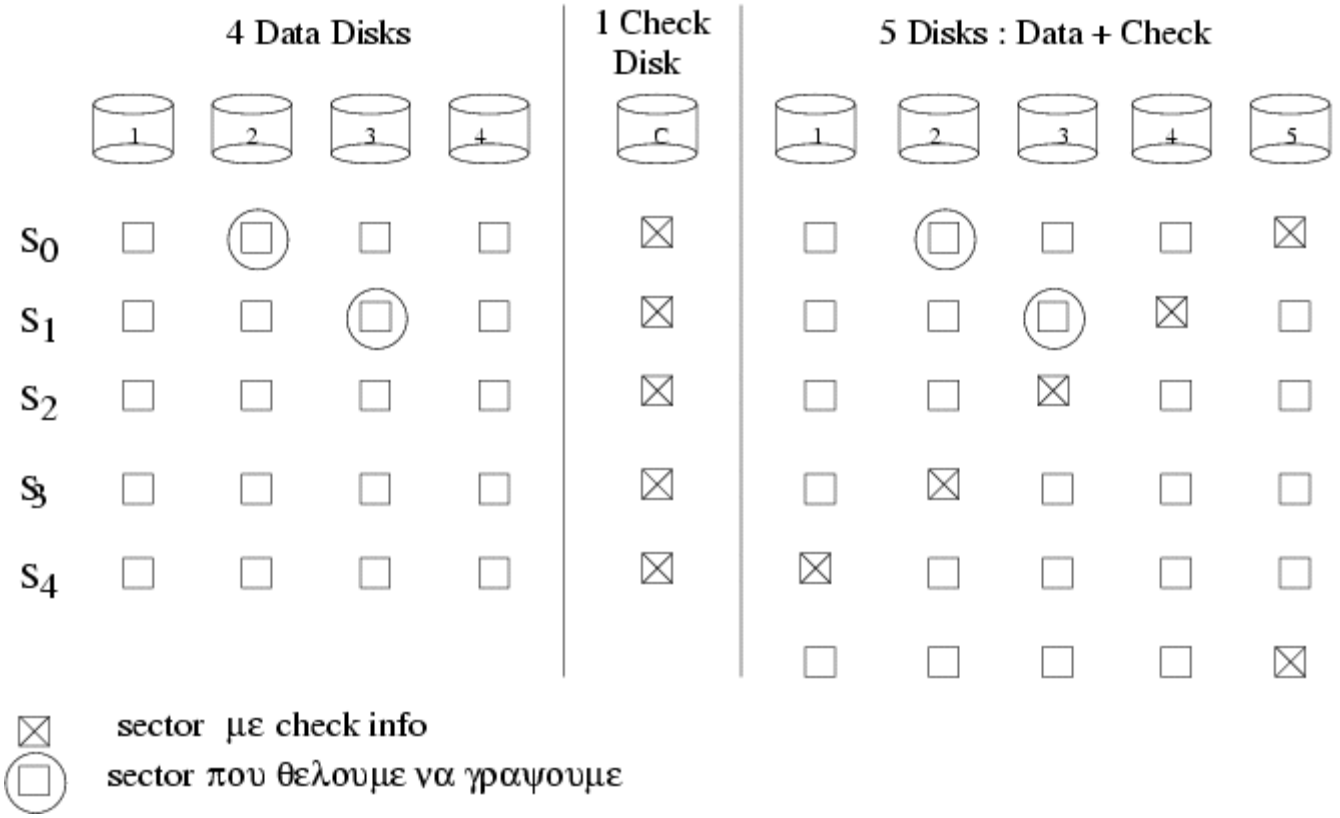
- <u>RAID Level 4 :</u> Ανεξάρτητα Reads/Writes & Sector Interleaving

  Η κύρια διαφορά είναι ότι αντί για bit-interleaving έχουμε sector-interl. (δηλ. το stripe είναι ένας sector).

  new parity = (new sector XOR old sector) XOR old parity ➔ για να γράψουμε έναν sector χρειαζόμαστε 2 disk reads και 2 disk writes. Για ένα read ενός sector χρειαζόμαστε μόνο ένα disk read (μιας και ο disk controller μπορεί να μας πεί αν υπάρχει κάποια βλάβη/λάθος/ζημιά στον sector).

  Προσέξτε ότι κάθε write πρέπει να γράψει και τον parity disk ➔ κάθε group G δίσκων (στο οποίο αντιστοιχεί και ένας parity disk) μπορεί να εξυπηρετεί μόνο 2 write κάθε φορά.

- <u>RAID Level 5 :</u> Χωρίς Parity Disk. Δεν υπάρχει ξεχωριστός parity disk.



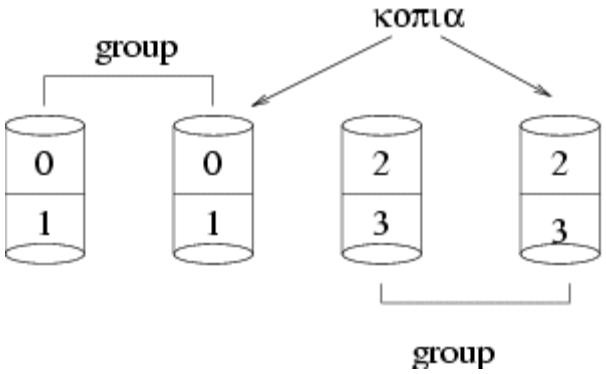
- Στο παράδειγμα τα 2 writes δεν μπορούν να συμβούν ταυτόχρονα στο disk array Level 4 ενώ μπορούν στο Level 5.

13.2 Κατανομή Πληροφορίας με Replication σε Disk Arrays - Αλγόριθμοι

Απαιτείται replication διότι αλλοιώς το disk array θα ήταν αναξιόπιστο (unreliable) και ως εκ τούτου ... άχρηστο.

13.2.1 Αλγόριθμος Mirroring

- Τον έχουμε ήδη παρουσιάσει (stable storage)
- 2 κόπιες ➔ χρειάζεται διπλάσιος αριθμός δίσκων



Κάθε δίσκος (χωρίζεται σε δύο κομμάτια στο σχήμα) έχει ένα αντίγραφο. RAID1, mirroring, duplexing, disk shadowing ...

Μειονεκτήματα: - Διπλάσιο κόστος αποθήκευσης
- κόστος WRITE (γιατί;)

Πλεονεκτήματα: + Κόστος READ (γιατί;)
+ Κάθε group αντέχει μέχρι D-1 failures
όταν έχει D δίσκους (D=2 στο σχήμα)

Συνήθως ένα disk array έχει πολλά groups. Κάθε disk failure σε ένα group μειώνει την απόδοση του group σε σχέση με τα άλλα groups ...! (αν το αργό group γίνει το bottleneck ...)

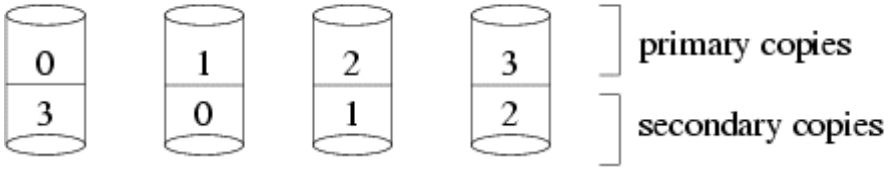
13.2.2 Αλγόριθμος Chained Declustering

Κίνητρο είναι να αποφευχθεί ένα group που έχει υποστεί βλάβες να γίνει το bottleneck.

<u>Βασική ιδέα</u>: όταν σημειώνεται βλάβη να κατανέμεται ο φόρτος ισόποσα στους "επιζώντες" δίσκους.

Το chained declustering

- χωρίζει κάθε δίσκο σε D τμήματα
- το πρώτο τμήμα κάθε δίσκου έχει πρωτότυπη πληροφορία (primary copy)
- κάθε πρώτο τμήμα ενός δίσκου είναι διαφορετικό από τα άλλα πρώτα τμήματα
- Για κάθε primary υπάρχουν D-1 κόπιες (secondary). Το primary copy του δίσκου i έχει το secondary copy του στο 2ο τμήμα του δίσκου (i+1 mod D).

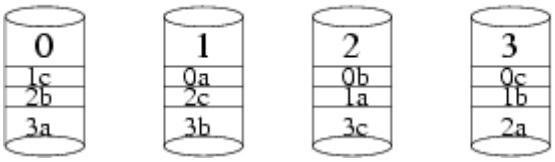


Απ'το σχήμα προκύπτει ότι αν "πέσει" ο δίσκος i τότε οι δίσκοι i-1 και i+1 θα μοιραστούν το παραπανίσιο workload για το secondary και primary section, αντίστοιχα ➔ δεν έχουμε το φαινόμενο ένας δίσκος να έχει διπλάσιο workload όπως στο mirroring. Προσέξτε όμως ότι αυξήθηκαν οι πιθανότητες χασίματος πληροφορίας (γιατί;).

13.2.3 Αλγόριθμος Interleaved Declustering

Το κίνητρο είναι το ίδιο και εδώ.

- Ο κάθε δίσκος χωρίζεται σε D τμήματα.
- Η βασική διαφορά από τον chained declustering είναι ότι το secondary copy (αντί να τοποθετηθεί στον επόμενο δίσκο) γίνεται striped στους υπόλοιπους δίσκους.



- Αυτή η μέθοδος επιτυγχάνει μεγαλύτερο load balancing απ' 'οτι η προηγούμενη.
- Το μειονέκτημα είναι ότι αυξάνεται περισσότερο η πιθανότητα χασίματος πληροφορίας (οποιοσδήποτε συνδυασμός D βλαβών είναι καταστροφικός).