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 πληροφορίες 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.

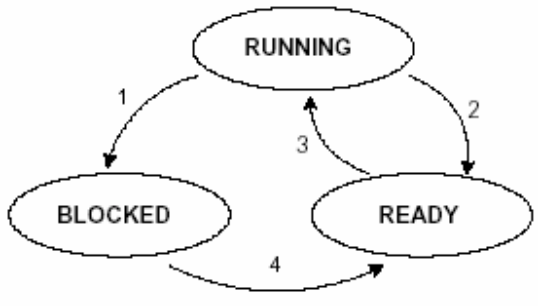
Ποιά η διαφορά μεταξύ "process" & "πρόγραμμα";

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.



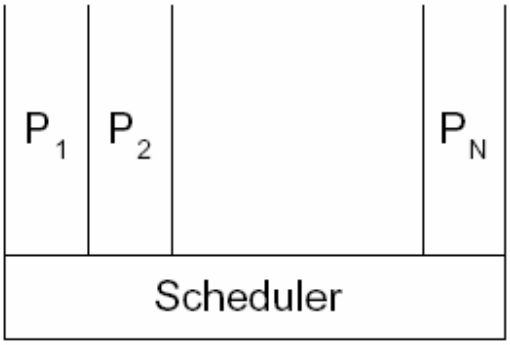
1. περιμένει I/O
2. CPU δίνεται σ' άλλο process
3. CPU δίνεται σ' αυτό το process
4. I/O έγινε

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 εκτελεί 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] ≠ 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.

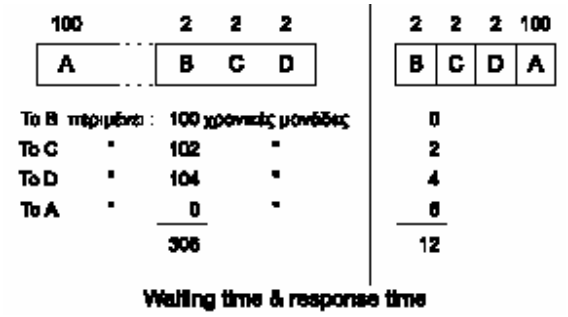
Είναι επίσης δυνατόν οι προτεραιότητες των processes να απορρέουν από τον τύπο του processes.
π.χ.
processes που περιμένουν για terminal I/O <-> 1
processes που περιμένουν για disk I/O <-> 2
processes των οποίων το quantum τελείωσε <-> *3*
processes που έχουν χρησιμοποιήσει πολλά quanta <-> 4.

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 το 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 = αT0 + (1-α)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 = α T0 + (1-α) T2

Αν α=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

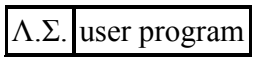
- Ο **Διαχειριστής Μνήμης (ΔΜ) (memory manager)** πρέπει να:

  - 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.



- 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

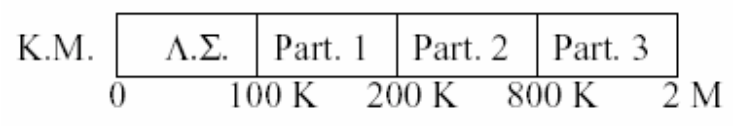
➔ με P = 80% το CPU utilization είναι:

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)**

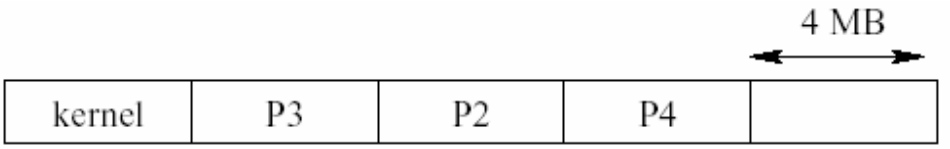
- Η έννοια των fixed partitions δεν βοηθά πολύ: Συχνά, σπαταλείται πολύ χώρος ΚΜ γιατί δεν γίνεται καλό "ταίριασμα" μεταξύ μεγεθών partition & process.

  ➔ **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**

K.M.	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.

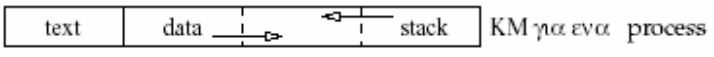


Σπάνια όμως χρησιμοποιείται γαιτί απαιτεί χρονοβόρο memory-to-memory copying (CPU time) (1MB/sec).

- **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**]

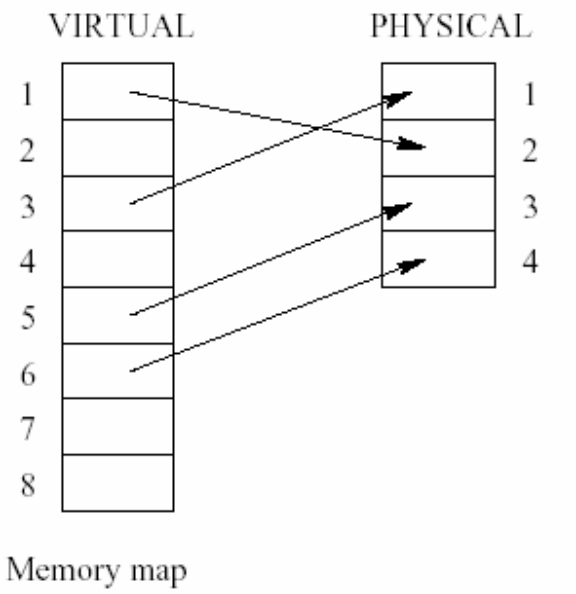
Το σύνολο των memory addresses που παράγονται κατά την εκτέλεση ενός προγράμματος ονομάζεται **address space**. Επειδή, στη γενική περίπτωση, αυτό το address space είναι μεγαλύτερο της ΚΜ, δεν μπορεί να υπάρξει μονοσήμαντη αντιστοιχία μεταξύ ενός address που παράγει ένα πρόγραμμα και ενός **physical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

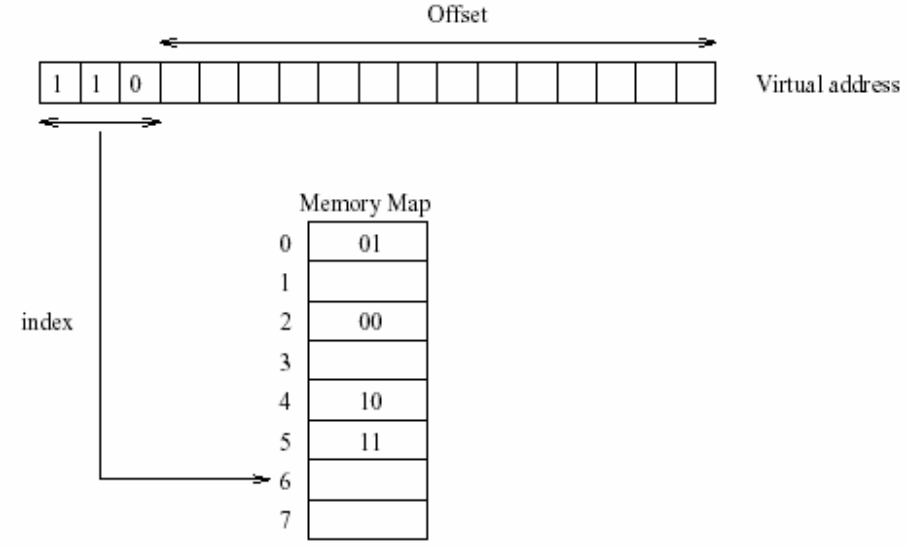
Το Λ.Σ. εξυπηρετεί ένα page fault ως εξής:

- 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.

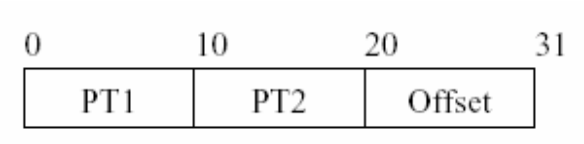
3.3.2 Πίνακες Σελίδων (Page Tables)

- 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 K.M. (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 K.M. 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:

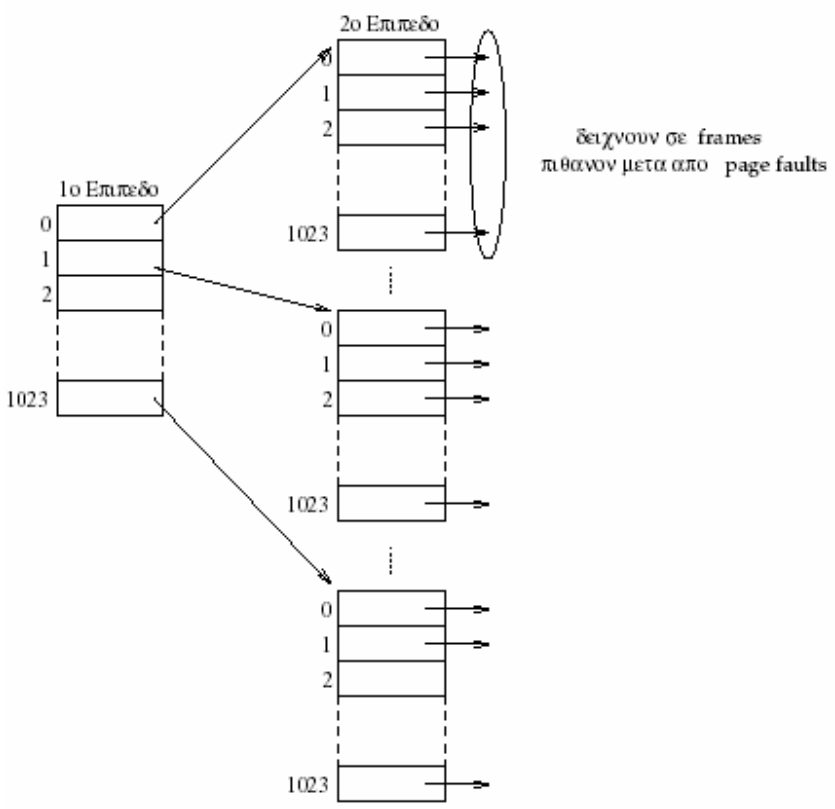
- 0-4MB ➔ text
- 4MB - 8MB ➔ data
- τα τελευταία 4MB ➔ stack (μεγαλώνει "προς τα κάτω"))

➔ 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)

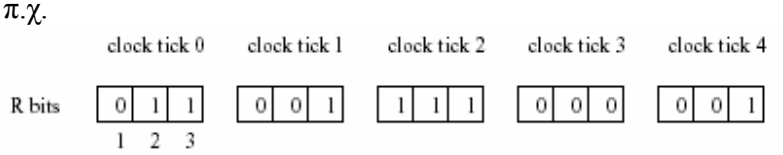
Βασική ιδέα: σελίδες που προσπελάστηκαν στο πρόσφατο παρελθόν θα επανα-προσπελασθούν και στο εγγύς μέλλον (και αντιστρόφως). Ετσι, το θύμα είναι η σελίδα που δεν έχει χρησιμοποιηθεί για το μεγαλύτερο χρονικό διάστημα.

- 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.



Counters	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 είναι 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 το 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)

Αλγόριθμος

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 ➔

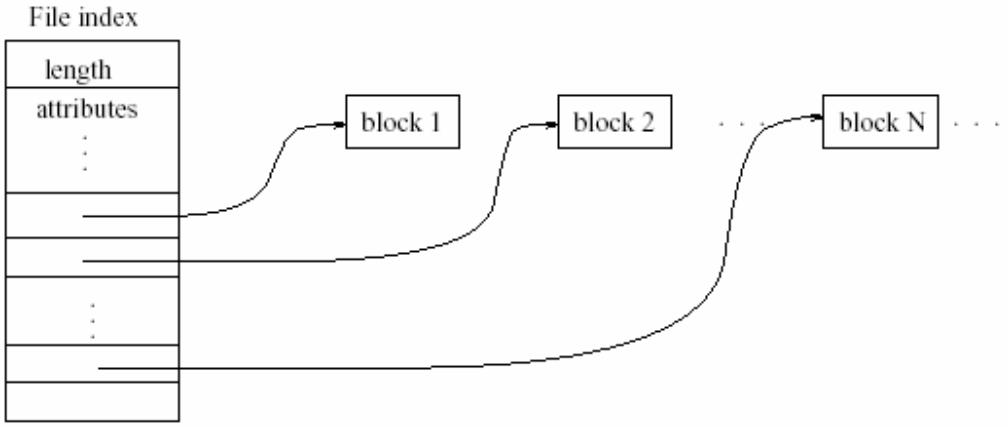
(a) UFID -> block pointer of file index
(b) offset -> block pointer to required page
(a)+(b) : Two step translation

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

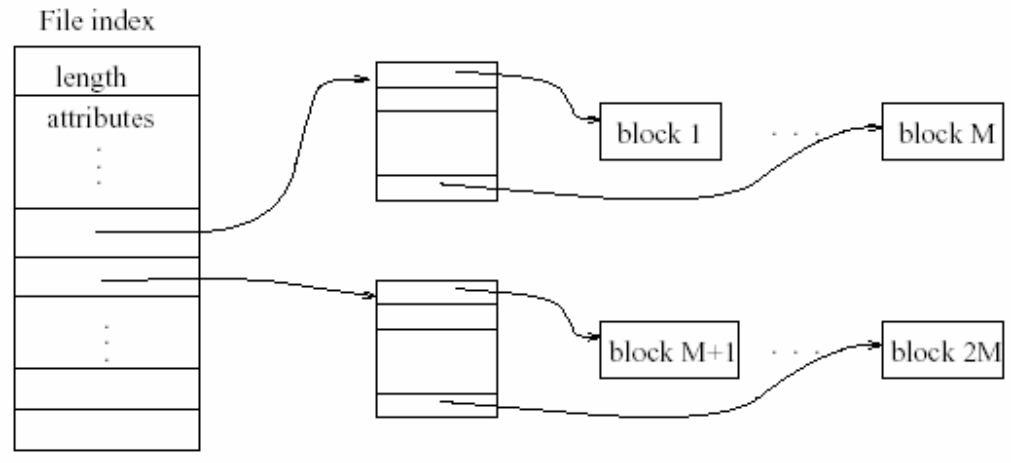
One step translation: UFID + offset -> block ptr για referenced block

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

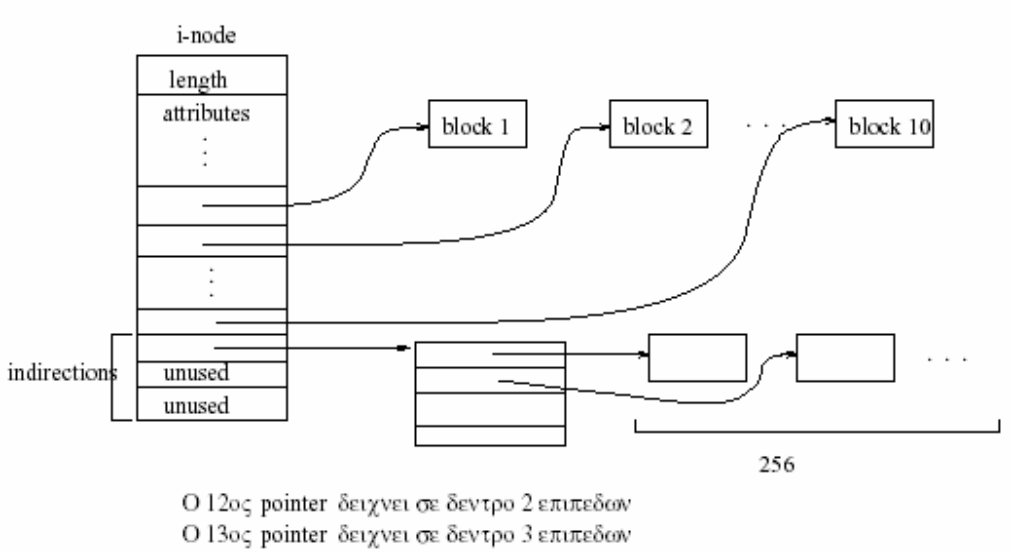
Παράδειγμα File index



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).



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 μπορούν 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 ωε "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 MONO 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).

Η δομή των files ποικίλει:

- 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.

Τύποι Files:

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.

Τύποι προσπέλασης: 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 K.M. 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 K.M. 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:

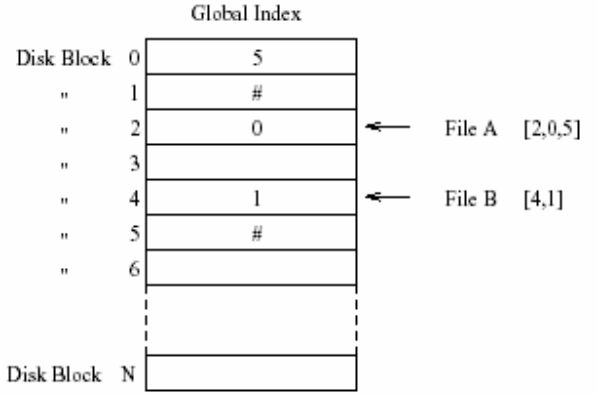
Η πρώτη επιλογή αφορά στο αν υπάρχει "per file" index ή ένα index για όλα τα αρχεία στο σύστημα. Το UNIX FS χρησιμοποιεί per-file index ενώ το MS-DOS χρησιμοποιεί "global" index.

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 υπεύθυνος 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 K.M. 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).

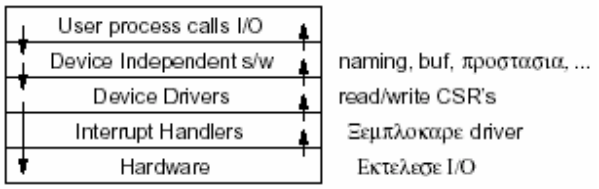
Ο spooler είναι ένα ειδικό process (δαίμονας) που διαχειρίζεται ένα spooling directory.

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).

Παράδειγμα:

	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.