Κεφάλαιο 2

2.1 Μοντέλο της διεργασίας

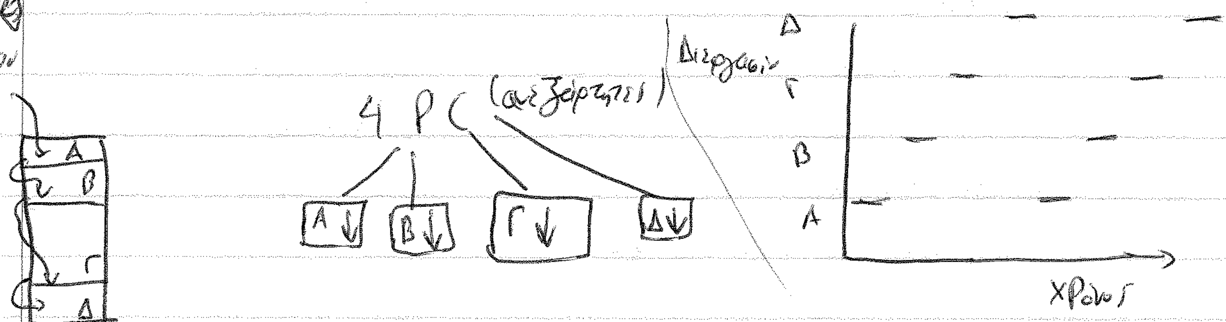
Ψευδοπαραλληλία: Σε συστήματα πολυπρογραμματισμού, η CPU εναλλάσσεται διαδοχικά και γρήγορα σε διαφορετικές διεργασίες, εκτελώντας καθε μια από αυτές για μερικές δεκάδες ή εκατοντάδες milliseconds. Η CPU εκτελεί κάθε στιγμή 1 διεργασία, δίνοντας την ψευδαίσθηση στο χρήστη την παραλληλία.

Ο όρος λέγεται ψευδοπαραλληλία για να τονιστεί η αντίθεση με την πραγματική παραλληλία υλικού (hardware parallelism) των συστημάτων πολυεπεξεργασίας (2 ή περισσότερες CPU που μοιράζονται την ίδια μνήμη)

Ακολουθιακές διεργασίες: Όλο το λογισμικό του υπολογιστή (μαζί με το OS) οργανώνεται σε ένα πλήθος από ακολουθιακές διεργασίες (sequential processes) οι οποίες λέγονται διεργασίες

- instance προγράμματος που έχει τα values του PC, Reg & var
- έχει δική της virtual CPU

Εναλλαγή διεργασιών



Μόνο 1 πρόγραμμα ενεργό

Σε διεργασίες που περιέχουν κρίσιμες απαιτήσεις real-time (events που πρέπει να γίνουν σε συγκεκριμένο αριθμό χιλιοστών του δευτερολέπτου), πρέπει να λαμβάνονται μέτρα για να υλοποιούνται.

Διεργασία/πρόγραμμα - Διεργασία: εκτέλεση συνταγής για παρασκευή του φαγητού
Πρόγραμμα: συνταγή μαγειρικής

Διεργασία: Δραστηριότητα κάποιου είδους, Περιλαμβάνει πρόγραμμα, input, output και είναι σε ένα state

Scheduling Algorithm: Η CPU μοιράζεται σε διάφορες διεργασίες μέσω ενός αλγορίθμου χρονοπρογραμματισμού ο οποίος καθορίζει πότε σταματά η εκτέλεση μιας διεργασίας και πότε αρχίζει η εκτέλεση μιας άλλης.

2.1α Δημιουργία Διεργασιών

ιντά
αι Γενικά
Συστήματα

Στα απλά συστήματα είναι δυνατόν όλες οι διεργασίες να δημιουργούνται μετά το boot
Στα συστήματα γενικής χρήσης όμως χρησιμοποιούνται, χρησιμοποιούνται, ειδικές διαδικασίες για τη δημιουργία / τερματισμό των διεργασιών

γοι Δημιουργίας

1) Ανάθεση αρχικών τιμών (initialization) στο σύστημα
2) Εκτέλεση syscall από διεργασία που ήδη εκτελείται
3) Αίτηση χρήστη για τη δημιουργία μιας νέας διεργασίας
4) Εκκίνηση διεργασίας δέσμης (batch job)

O.S boot → New Processes

↙ Background ↓

δε σχετίζονται με χρήστες
έχουν όμως αρμοδιότητες
(mails, print pages)
λέγονται daemons
(ps στο UNIX για να τις δούμε)

↘ Foreground ↓

interaction με
users και
εξυπηρέτησή τους

: fork()

Δημιουργία νέων διεργασιών στο UNIX (syscall fork)
- Δημιουργείται ακριβές αντίγραφο της μητρικής διεργασίας με
① ίδιο memory image, ② ίδια environment strings ③ open files

- Κάποια θυγατρική εκτελεί κάποια εντολή της οικογένειας exec για να αλλάξει memory image και να εκτελέσει άλλο πρόγραμμα
- Ο λόγος που ακολουθείται αυτή η διαδικασία σε 2 βήματα είναι ότι έτσι η θυγατρική αποκτά τη δυνατότητα να χειρίζεται τους δικούς της file descriptors (μετά από fork, πριν από exec) προκειμένου να κάνει redirect του stdin, του stdout, και του stderr

CreateProcess

Στα Windows η CreateProcess δημιουργεί νέες διεργασίες
- Χειρίζεται τόσο τη δημιουργία όσο και τη φόρτωση του σωστού προγράμματος σε αυτή.
- 10 arguments
  - → προδιαγραφές για το παράθυρο που θα δημιουργηθεί
  - → info για priority
  - → attributes ασφαλείας
  - → bit που δείχνει αν τα open files μεταβιβάζονται κληρονομικά
  - → το πρόγραμμα που θα εκτελεστεί
  - → παράμετροι γραμμής διαταγών
  - → δείκτη στη δομή που θα αποθηκεύονται οι πληροφορίες για την πορεία της νέας διεργασίας και επιστρέφονται στον καλούντα πρόγραμμα

Τόσο στο UNIX όσο και στα Windows, μετά από δημιουργία διεργασίας η γονική έχει διαφορετικό χώρο διευθ. από την θυγατρική. Αν αλλάξει μία λέξη στη μία, δε φαίνεται στην άλλη
- Στο UNIX το address space της θυγατρικής είναι copy του parent
- Στα Windows, τα address space είναι διαφορετικά εξ αρχής

2.1.3 Τερματισμός Διεργασίας

Λόγοι
1) Normal exit (ολοκλήρωση του έργου)
2) Exit from error (εκούσιο) (σφάλμα σε αρχείο για files όπου files δεν υπάρχει)
3) Fatal error (ακούσια) (σφάλμα στον κώδικα)
4) Kill from other process (ακούσια) (kill ή TerminateProcess, ο δολοφόνος πρέπει να τα δικαιώματα για να το κάνει αυτό)

Όταν μια διεργασία τερματίζεται με οποιονδήποτε τρόπο τα παιδιά της δε πεθαίνουν (Win & Unix)

2.14 Ιεραρχίες διεργασιών

- Parent δημιουργεί Process_a } ⇒ Σχηματίζεται ιεραρχία
- Process_a δημιουργεί Process_b

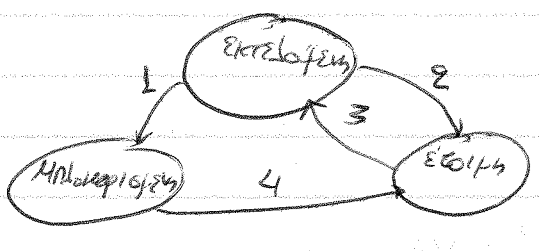
Process group (unix)

(δεν υπάρχει ιεραρχία, όλες οι διεργασίες ίδιο επίπεδο αλλά ο parent αποκτά ένα handle για το παιδί (windows))

2.15 Καταστάσεις Διεργασιών

Καταστάσεις Διεργασιών

1) Εκτελούμενη (running), πράγμα που σημαίνει ότι χρησιμοποιεί πραγματικά τη CPU εκείνη τη στιγμή
2) Έτοιμη (ready) ή εκτελέσιμη (runnable). έχει διακοπεί προσωρινά για να εκτελεστεί κάποια άλλη
3) Μπλοκαρισμένη (blocked), δεν μπορεί να συνεχίσει την εκτέλεση μέχρι να γίνει κάποιο εξωτερικό συμβάν



1) Η διεργασία μπλοκάρεται καθώς περιμένει input
2) ο scheduller επιλέγει άλλο process
3) ο " επιλέγει τη συγκεκριμένη διεργασία
4) το input είναι διαθέσιμο

Η (1) μετάβαση συμβαίνει όταν μια διαδικασία διαπιστώνει ότι δε μπορεί να συνεχίσει να εκτελείται.

Η (2) (3) προκαλούνται από τον scheduller (core του OS) και το process δεν το αντιλαμβάνεται

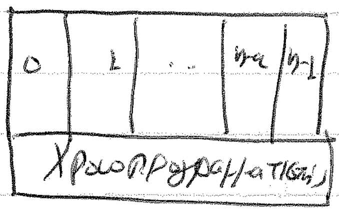
→(2) ο scheduller αποφασίζει ότι το process τρέχει πολύ ώρα

→(3) ο scheduller βλέπει ότι όλα τα processes εξάντλησαν το χρόνο της CPU, άρα επιστρέφει στο αρχικό process για να συνεχίσει

(4) Εξωτερικό event, το οποίο ανέμενε η διεργασία

Ο scheduler βρίσκεται στο χαμηλότερο level του OS ενώ από πάνω του μια ποικιλία από διεργασίες. Ο χειρισμός σημάτων διακοπής καθώς και οι λεπτομέρειες που αφορούν το boot και τη διακοπή του exec των διεργασιών βρίσκονται στο scheduler, που στην πραγματικότητα αποτελείται από ελάχιστο κώδικα. Το υπόλοιπο O.S είναι καλά δομημένο με τη μορφή — διεργασιών.

Διεργασίες



2.1.6 Υλοποίηση διεργασιών

Το OS διατηρεί ένα πίνακα διεργασιών (process table) και περιλαμβάνει μια καταχώρηση για κάθε διεργασία (ή μπλοκ ελέγχου διεργασιών). Περιέχει info σχετικά με κατάσταση διεργασίας, PC, Stack Pointer, κατανομή μνήμης, κατάσταση αρχείων που έχει ανοίξει η διεργασία, πληροφορίες διαχειριστικού χαρακτήρα και scheduling και ό,τι άλλο χρειάζεται για να αλλάξει state προκειμένου να συνεχιστεί. *(Καταχώρηση)*

(Διάνυσμα Διακοπών) Σε κάθε κατηγορία Ε/Ε αντιστοιχεί μια θέση η οποία λέγεται διάνυσμα διακοπών. Η θέση αυτή περιλαμβάνει τη διεύθυνση της διαδικασίας εξυπηρέτησης διακοπής.

~~Όταν φτάσει ένα σήμα διακοπής, ο PSW (Program status word) ή και άλλοι regs τοποθετούνται στη στοίβα (τρέχουσα) από το υλικό διακοπής. Στη συνέχεια το control μεταφέρεται στη διεύθυνση που ορίζεται από το διάνυσμα διακοπών. Αυτά~~ αναλαμβάνει το hardware

Διακοπή

1) Το υλικό τοποθετεί στη στοίβα το PC, PSW, κτλ
2) Το υλικό φορτώνει το νέο PC από το διάνυσμα διακοπής
3) Η διαδικασία που είναι γραμμένη σε συμβολική γλώσσα αποθηκεύει τους regs
4) " " " " " δημιουργεί νέο stack
5) Η εξυπηρέτηση διακοπής (σε) εκτελείται και συνήθως διαβάζει και αποθηκεύει προσωρινά το input
6) Ο scheduller αποφασίζει ποια διεργασία θα εκτελεστεί αμέσως μετά
7) Η διαδικασία σε C επιστρέφει τον έλεγχο στην αντίστοιχη διαδικασία συμβολικής γλώσσας
8) Η διαδικασία συμβολικής γλώσσας ξεκινάει την εκτέλεση της νέας διεργασίας

2.1.7 Μοντελοποίηση πολυπρογραμματισμού

- n διεργασίες
- p κλάσμα χρόνου του process

} Αξιοποίηση CPU $= 1 - p^n$

2.2 Νήματα

- Κάθε διεργασία έχει το δικό της address space
- Εμφανίζονται καταστάσεις όπου είναι επιθυμητή η ύπαρξη πολλών νημάτων τα οποία εκτελούνται ψευδοπαράλληλα στο ίδιο address space σαν να ήταν ξεχωριστές (στο ίδιο address space).
- Το μοντέλο αυτό βασίζεται σε: α) ομαδοποίηση των πόρων β) εκτέλεση

Διαχωρισμός → νήματα

2.2.1 Χρήση των νημάτων

Λόγοι χρήσης νημάτων: Δραστηριότητες που εκτελούνται παράλληλα (Μερικές μπλοκάρονται περιστασιακά)

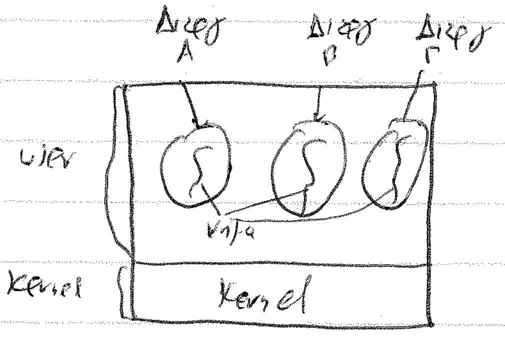
Νήματα
- Ίδιο address space και data
- Δημιουργία - καταστροφή νημάτων αρκετά εύκολη (αφού δε διαθέτουν πόρους)
- Εξοικονομούν χρόνο α υπάρχει πολύ μεγάλος υπολογιστικός φόρτος μαζί με εκτεταμένη χρήση των συσκευών Ε/Ε (κάλυψη αυτών των εργασιών αυτών → ταχύτερη εκτέλεση)
- Πραγματική παραλληλία

FSM

Η κατάσταση των υπολογισμών πρέπει [illegible] ρητά να αποθηκεύεται και να επαναφέρεται κάθε φορά που ο διακομιστής εναλλάσσει στις αιτήσεις

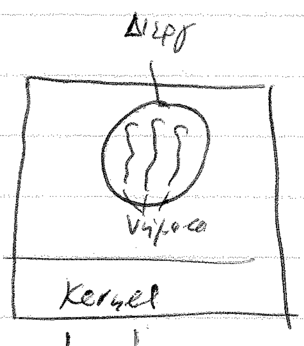
2.2.2 Κλασικό μοντέλο των νημάτων

Χρήση νημάτων
- Ταυτόχρονη εκτέλεση διαφόρων ροών ελέγχου στο κοινό περιβάλλον μιας διεργασίας (Ανεξάρτητες ροές)
- Ελαφριές διεργασίες
- Πολυνημάτωση (εναλλαγή διεργασιών σε χρόνο [illegible])



3 Διεργασίες με 1 νήμα η καθεμία

- Κάθε διεργασία δικό της space και 1 νήμα ελέγχου
- Κάθε νήμα δικό του χώρου



1 διεργασία με 3 νήματα

- ίδιο space (ίδια νου)
- no protection
- Κατάλληλο για νήματα τα οποία συνεργάζονται μαζί

Στοιχεία ανά διεργασία	Στοιχεία ανά νήμα
Address space	PC
Global var	Regs
open files	Stack
Θυγατρικές διεργασίες	State
Εκκρεμή σήματα συναγερμού	
Σήματα και χειριστές σημάτων	
Διαχειριστικές πληροφορίες	

2.2.3 νήματα στο POSIX

Pthread_create	new thread
Pthread_exit	termination of thread
Pthread_join	Αναμονή για έξοδο
Pthread_yield	free CPU για να ξεκινήσει το next thr
Pthread_attr_init	instantiation and init of a thread
Pthread_attr_destroy	deletion of thread's structure

2.2.4 Υλοποίηση νημάτων στο user space

POSITIVE

1) Τοποθέτηση του πακέτου νημάτων εξολοκλήρου στο user space (Ο kernel δε το γνωρίζει απολύτως και διαχειρίζεται τις διεργασίες). Έτσι μπορούμε να έχουμε ένα OS με thread ακόμη και αν δε το υποστηρίζει
2) Κάθε thread έχει scheduling algorithm
3) Δε υπάρχει ανησυχία για διακοπή
4) Ευκολότερη προσαρμογή των εφαρμογών

Jacket-wrapper & κώδικες βιβλιοθήκες για να εκτελέσουν έλεγχο αν θα υπάρχει blockάρισμα ή όχι.

NEGATIVE

1) Ο τρόπος Υλοποίησης των syscall μπλοκάρουν τα νήματα που τα καλούν

2) Page fault: Αν το πρόγραμμα καλεί μεταφέρεται σε εντολή που δεν υπάρχει στη μνήμη, το O.S πρέπει να προσκομίσει τη σελίδα από το δίσκο. Γίνεται block μέχρι να βρεθεί το Page και να διαβαστεί η εντολή. Αν thread προκαλέσει Page-fault ο kernel blockάρει τη διεργασία αυτή και αν τα υπόλοιπα threads μπορούν να συνεχίσουν

3) Όταν εκτελείται ένα νήμα, δε μπορεί να διακοπεί για κάποιο άλλο μέχρις ότου να παραδώσει τον έλεγχο της CPU εκούσια

4) οι scheduller χρειάζονται τα threads στις περιπτώσεις που αυτά μπλοκάρονται πιο συχνά

Κάθε Process έχει δικό του thread table

2.2.5 Υλοποίηση threads στον kernel

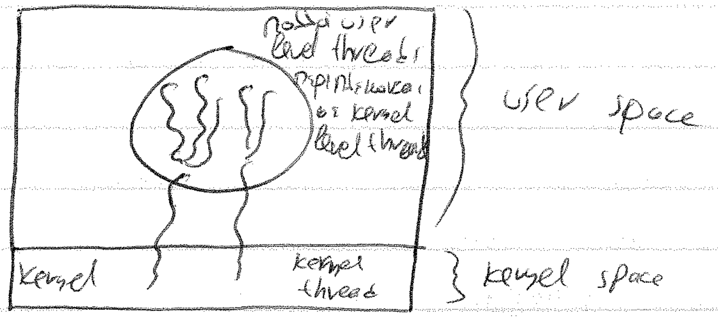
- Ο kernel είναι ενήμερος και διαχειρίζεται τα threads
- Δεν υπάρχει thread table για το κάθε νήμα
- Δε χρειάζεται scheduller κάθε thread
- Όλες οι blocking call υλοποιούνται ως syscalls με σημαντικά μεγαλύτερο κόστος χρόνου εκτέλεσης
- Ανακύκλωση νημάτων (Υψηλό κόστος δημιουργίας καταστροφής): Όταν ένα νήμα εξαλείφεται, σημαδεύεται ως μη εκτελέσιμο αλλά οι δομές του δεν αλλάζουν. Αργότερα όταν χρειάζεται ένα νέο νήμα ενεργοποιείται το παλιό.
-

Τι γίνεται αν νήμα κάνει fork

Τι γίνεται με τα σήματα που πηγαίνουν μόνο σε διεργασίες

2.2.6 Υβριδικές Υλοποιήσεις

- Χρήση νημάτων πυρήνα και στη συνέχεια πολύπλεξη νημάτων σε επίπεδο χρήστη σε μερικά ή όλα τα νήματα του kernel. Έτσι ο προγραμματιστής μπορεί να αποφασίσει πόσα νήματα του kernel θα χρησιμοποιήσει και πόσα του user level θα πολυπλεχθούν σε καθένα από τα πρώτα.
- Ο kernel διαχειρίζεται - αντιλαμβάνεται μόνο τα threads του πυρήνα. Σε μερικά μπορεί να έχουν πολυπλεχθεί πολλά νήματα από user level (τα οποία δημιουργούνται καταστρέφονται και γίνονται scheduling όπως στο user level threads)
- Κάθε νήμα στο kernel level εξυπηρετεί εκ περιτροπής ένα σύνολο από νήματα χρήστη



2.2.7 Ενεργοποιήσεις scheduler

heduler ctivation: Μιμείται τη λειτουργικότητα των νημάτων του kernel προσφέροντας a) καλύτερη απόδοση b) μεγαλύτερη ευελιξία

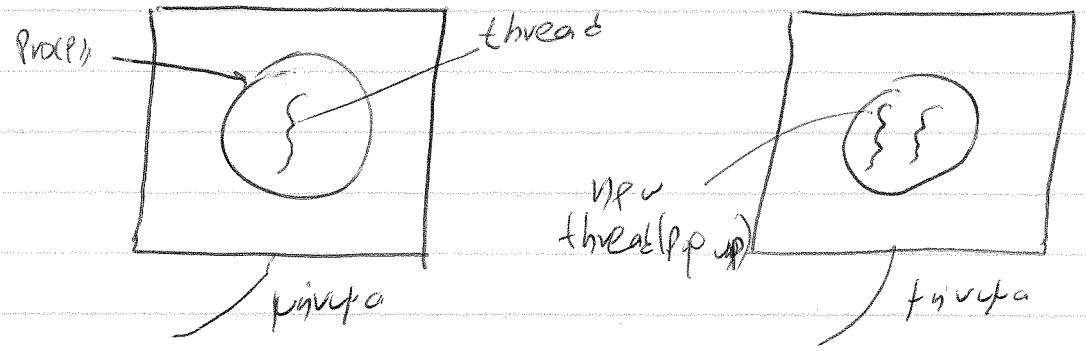
Τα user threads δε χρειάζεται να ενεργοποιούν ειδικές μη ανασταλτικές syscall ή να ελέγχουν εκ των προτέρων αν είναι ασφαλείς ή όχι (syscalls)

Δηλαδή, αν υπάρξει block από syscall ή page fault, θα πρέπει να είναι δυνατή η εκτέλεση άλλων νημάτων του ίδιου process, αν είναι ready

upcall: Αν ο kernel αντιληφθεί πως έχει γίνει block ενός thread ενημερώνει το σύστημα εκτέλεσης χρόνου της διεργασίας σχετικά με αυτό, βάζοντας στο stack τις πληροφορίες (params) του και τη περιγραφή του event

2.2.8 Pop up threads

Threads χρήσιμα στα distributed systems.
Αντί να έχω process ή thread το οποίο είναι blocked και περιμένει το μήνυμα (παραδοσιακός τρόπος), μπαίνει το σύστημα να δημιουργήσει νέο thread (pop up thread) για να χειριστεί αυτό το μήνυμα και έπειτα να τερματίσει

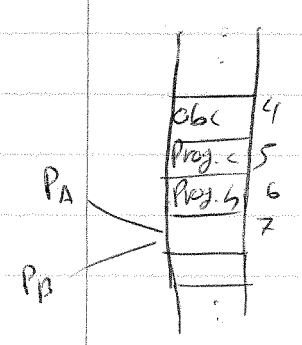


2.3 Διαδιεργασιακή επικοινωνία (Inter Process Communication - IPC)

1) Με ποιο τρόπο μπορεί μια διεργασία να μεταβιβάσει πληροφορία σε μια άλλη
2) Εξασφάλιση ότι 2 ή περισσότερες διεργασίες δεν εμποδίζουν η μία την άλλη όταν εκτελούν κρίσιμες ενέργειες
3) Κατάλληλη αλληλουχία ενεργειών που πρέπει να ακολουθηθεί όταν υπάρχουν εξαρτήσεις

2.3.1 Συνθήκες συναγωνισμού

Συνεργαζόμενες διεργασίες — κοινή περιοχή αποθήκευσης / καθε διεργασία μπορεί να γράψει και να διαβάσει



PA, PB δίνουν access στο out[4]

2.3.2 Κρίσιμες περιοχές

αμοιβαίος αποκλεισμός — Αν: μια διεργασία χρησιμοποιεί μια κοινή μεταβλητή ή αρχείο οι άλλες διεργασίες αποκλείονται από την εκτέλεση της ίδιας ενέργ

κρίσιμη περιοχή / κρίσιμο τμήμα — Τμήμα του προγράμματος στο οποίο γίνεται προσπέλαση κοινόχρηστης μνήμης

Συνθήκες να εξασφαλίζουν αμοιβαίο αποκλεισμό

1) Δύο συναρτήσεις δε βρίσκονται ποτέ ταυτόχρονα στις κρίσιμες περιοχές
2) Δεν επιτρέπονται παραδοχές σε ότι αφορά την ταχύτητα ή το πλήθος των CPU
3) Διεργασία που δε βρίσκεται σε κρίσιμο σημείο, δε μπορεί να μπλοκάρει άλλες διεργασίες
4) Δεν επιτρέπεται μια διεργασία να αναμένει επ' αόριστον να μπει στην κρίσιμη περιοχή της

2.3.3 Αμοιβαίος αποκλεισμός μέσω αναμονής με απασχόληση

Απενεργοποίηση διακοπών

1) Κάθε διεργασία που μπαίνει στην κρίσιμη περιοχή πρέπει να απενεργοποιεί όλες τις διακοπές και να τις επανεργοποιεί ~~αφού~~ βγ
2) Απενεργοποίηση → δε γίνεται διακοπή ρολογιού → Η CPU πριν δεν αλλάζει διεργασίες

~~3)~~ Αρνητικά: a) Οι διεργασίες χρήστη μπορούν να απενεργοποιούν διακοπές

b) Δεν εφαρμόζεται σε συστήματα με πολλές CPU

Θετικά: • Βολικό για το kernel να κάνει disable, όταν περάζει λίστες ή μεταβλητές

→ Χρήσιμο για το ίδιο το ΛΣ αλλά όχι για το USER

- Μεταβλητές κλειδώματος
  2 Processes στο ίδιο κρίσιμο σημείο (Κίνδυνος)

Αυστηρή εναλλαγή

```
while (TRUE) {                      while (TRUE) {
  while (turn != 0)                   while (turn != 1)
  critical_region();                  critical_region();
  turn = 1;                           turn = 0
  noncritical_region();               noncritical_region();
}                                   }
      P1                                  P2
```

- busy waiting: Συνεχής έλεγχος ενός var μέχρι να πάρει συγκεκριμένη τιμή (① σπατάλη CPU ② πρόβλημα συγχρονισμού αν μια process πιο αργή από την άλλη)

Peterson

Εξασφάλιση κάθε διεργασίας ότι είναι η σειρά της για να μπει. Αν μπουν και οι 2, η τελευταία που έγραψε είναι στην turn

```
#define FALSE 0
#define TRUE 1
#define N 2              /* Processes count */
int turn;                /* ποια διεργασία */
int interested[N];       /* όλες οι τιμές 0 */
void enter_region(int process) {
    int other;
    other = 1 - process;
    interested[process] = TRUE;
    turn = process;
  while (turn == process && interested[other] == TRUE);
}

void leave_region(int process)
{
   interested[process] = F
}
/* έξοδος από κρίσι
```

TSL

Test and set lock

Εντολή:

TSL REGISTER, LOCK

- Διαβάζει το περιεχόμενο μιας λέξης lock
- Το αποθηκεύει στον καταχ. Register
- αποθηκεύει στη μνήμη μια μη μηδενική τιμή στο lock
- Κανείς άλλος CPU δεν έχει πρόσβαση (αναγκή CPU lock bus)

```
enter_region:
    TSL REGISTER, LOCK      | register=lock; lock=1
    CMP REGISTER, #0        | ήταν το lock 0;
    JNE enter_region        | lock!=0, goto enter_region
    RET                     | return caller function
```

```
leave_region:
    MOVE LOCK, #0           | lock=0
    RET                     | return caller function
```

για XCHG λίγα (x86 CPU)

```
enter_region:
    MOVE REGISTER, #1       | τιμή 1 στο Reg
    XCHG REGISTER, LOCK     | swap values
    ίδιο μετά
```

2.3.4 Λήθαργος και αφύπνιση

Priority inversion Problem

Υ, Χ διεργασίες (Υ high priority, Χ low)
Scheduler επιβάλλει Υ αν είναι ready
Έστω ότι Χ κρίσιμο τμήμα, Υ ready.
Υ busy waiting, αλλά αφού scheduler δεν ασχολείται με τη Χ δε φεύγει από το κρίσιμο σημείο και η Υ κάνει rolling επ'άπειρο.

Παραγωγός - Καταναλωτής (περιορισμένη προσωρινή μνήμη)

- Δύο διεργασίες μοιράζονται κοινή κάποια θέση προσωρινής μνήμης
- Ο Παραγωγός (μία από τις δύο) τοποθετεί πληροφορίες στην περιοχή, ενώ ο καταναλωτής τις αφαιρεί

Πρόβλημα Παραγωγός θέλει να βάλει νέες πληροφορίες αλλά η μνήμη είναι γεμάτη

Λύση sleep παραγωγό μέχρι ο καταναλωτής να αφαιρέσει από τη μνήμη (αντίστοιχα για τον καταναλωτή)

```
#define N 100          /* θέσεις στη μνήμη */
int count;             /* στοιχεία στη μνήμη */

void producer(void) {
    int item;
    while (1) {                    /* loop */
        item = produce_item();     /* make item */
        if (count == N) sleep();   /* full mem */
        insert_item(item);         /* put item */
        count++;                   /* increase count */
        if (count == 1) wakeup(consumer);
    }
}

void consumer(void) {
    int item;
    while (1) {
        if (count == 0) sleep();
        item = remove_item();
        count--;
        if (count == N-1) wakeup(producer);
        consume_item(item);
    }
}
```

Πρόβλημα
Χάνεται το bit αφύπνισης, όταν απενεργοποιείται η διεργασία που δεν είχε αποκοιμηθεί

πρέπει empty;

2.35 Σηματοφόροι

Σηματοφόροι

Ακέραιο μεταβλητή με τιμή 0 (κανένα σήμα αφύπνισης) ή θετικός ανάλογα με το πλήθος σημάτων

down = sleep , up = wakeup

↓

Checks if (ακέραιος>0) { [illegible]; ακέραιο--; go_on(); }
if (ακέραιος==0) { sleep(); }

Ο έλεγχος, αλλαγή τιμής, πιθανή απενεργοποίηση πραγματοποιείται σε __αδιαίρετη ενέργεια (atomic action)__

```
# define N 100                /* αριθμός θέσεων στη μνήμη */
typedef int semaphore         /* ειδική κατηγορία ακεραίων */
semaphore mutex = 1;          /* access κρίσιμη περιοχή */
semaphore empty = N           /* πλήθος κενών θέσεων */
semaphore full = 0;           /* πλήθος γεμάτων θέσεων */

void producer(void) {
    int item;
    while (1) {
      item = produce_item();
      down(&empty);     /* μείωση θέσεων κατά 1 */
      down(&mutex);     /* είσοδος κρίσιμη περιοχή */
      insert_item(item);
      up(&mutex);       /* έξοδος από κρίσιμη */
      up(&full);        /* αύξηση θέσεων κατά 1 */
    }
}
```

```
void consumer(void) {
  int item;
  while(1) {
    down(&full);
    down(&mutex);
    item = item.remove();
    up(&mutex);
    up(&empty);
    consume_item(item);
  }
}
```

Σημαφόροι → Συγχρονισμός
full και empty εξασφαλίζουν ότι θα εμφανίζονται ή όχι συγκεκριμένες ακολουθίες συμβάντων
Ο παραγωγός παύει αν γεμίσει full
Ο καταναλωτής παύει αν γεμίσει ~~e~~ empty

2.3.6 mutex

2 states: locked, unlocked, άρα 1 bit μόνο

mutex_lock → access
Αν locked, τότε το καλούν νήμα blockάρεται
mutex_unlock → ~~free~~ έξοδος από κρίσιμη

```
mutex_lock:
  TSL REGISTER, MUTEX
  CMP REGISTER, #0
  JZE ok                | αν mutex=0, άρα επιστροφή
  CALL thread_yield     | busy schedule άλλο thread
  JMP mutex_lock        | try again
ok: RET                 | critical region entered
```

```
mutex_unlock:
  MOVE MUTEX, #0
  RET
```

23.7 Ελεγκτές

Συλλογή από διαδικασίες, μεταβλητές, δομές δεδομένων που ομαδοποιούνται σε μία ειδική κατηγορία υποπρογράμματος (module ή πακέτου). Οι διεργασίες μπορούν να καλέσουν τις διαδικασίες ενός ελεγκτή όποτε θέλουν, αλλά δε μπορούν να προσπελάσουν απευθείας τις εσωτερικές δομές του ελεγκτή χρησιμοποιώντας διαδικασίες που γράφτηκαν έξω από αυτό

- Επίτευξη αμοιβαίου αποκλεισμού

Ισοβιότητα συνθήκης

- Αν μια διαδικασία ελεγκτή δε μπορεί να συνεχίσει, εφαρμόζει τη λειτουργία wait() σε κάποια μεταβλητή (full πχ) η οποία blockαρει την καλούσα διεργασία. Επιτρέπει επίσης σε μία άλλη διεργασία, στην οποία πριν είχε απαγορευθεί η είσοδος στον ελεγκτή, να το χρησιμοποιήσει
- Η άλλη διεργασία (καταναλωτής), μπορεί να αφυπνίσει την αποκλεισμένη διεργασία (παραγωγό) εφαρμόζοντας μια λειτουργία σήμα σε μεταβλητή συνθήκης (την οποία αναμένει αφού είναι σε λήθαργο)

γίνεται με το signal;

Ο Brinch Hansen πρότεινε: η διεργασία που καλεί το signal να αποχωρεί και να βγει από τον ελεγκτή
Δηλαδή, αν το signal εφαρμόστηκε σε μεταβλητή συνθήκης για την οποία αναμένουν πολλές διεργασίες, ο scheduler θα αποφασίσει ποια θα είναι η μοναδική που θα αφυπνιστεί

23.8 Μεταβίβαση μηνυμάτων

- Διαδιεργασιακή επικοινωνία
- Κλήσεις send, receive (χαρακτηρίζονται syscalls)
  - στέλνει data → λαμβάνει data από
  - σε destination → πηγή
  - send(destination, &message) → receive(source, &message)

Αν δεν υπάρχει διαθέσιμο μήνυμα, ο receiver blockαρεται μέχρι να φτάσει ένα μήνυμα

Ένα μήνυμα ενδέχεται να χαθεί στο δίκτυο.

Λύση Ο αποστολέας και ο παραλήπτης μπορούν να συμφωνήσουν ότι όταν ο δεύτερος παραλάβει ένα μήνυμα (ack) στέλνει το μήνυμα πίσω. Αν ο πρώτος δε το λάβει μέσα σε ένα καθορισμένο διάστημα, ξαναστέλνεται όλο το μήνυμα.

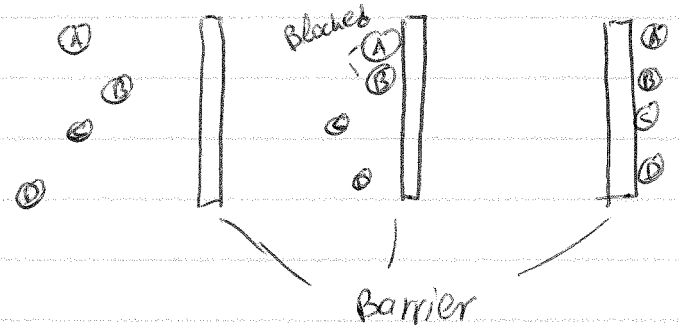
```
#define N 100
void producer(void) {
  int item;
  message m;                          /* προσωρινή μνήμη μηνύματος */
  while (1) {
    item = produce_item();            /* νέο στοιχείο στη προς μνήμη */
    receive(consumer, &m);            /* αναμονή νέα μηνύματος */
    build_message(&m, item);          /* κατασκευή μηνύματος */
    send(consumer, &m);               /* αποστολή μηνύματος */
  }
}

void consumer(void) {
  int item, i;
  message m;

  for (i=0; i<N; i++) send(producer, &m);
  while (1) {
    receive(producer, &m);            /* receive msg */
    item = extract_item(&m);          /* extract msg */
    send(producer, &m);               /* send empty msg */
    consume_item(item);               /* use msg */
  }
}
```

23.9 Φράγματα

Μερικές εφαρμογές διαιρούνται σε φάσεις (διαστήματα) και χρήσιμα μερικές διεργασίες δε μπορεί να προχωρήσουν στην επόμενη φάση μέχρι να είναι όλες έτοιμες.



2.4 Χρονοπρογραμματιστής

Scheduler τμήμα του O.S που ελέγχει κάθε φορά ποια διεργασία θα πάρει τον έλεγχο της CPU

Scheduling Algorithm ο αλγόριθμος που χρησιμοποιείται

Ρόλοι scheduler

1) Σωστή επιλογή διεργασίας
2) Αποδοτική χρήση της CPU
3) Εναλλαγή από user ... σε kernel level
   - Αποθήκευση των reg σε process table ώστε να μπορεί να ξαναφορτωθεί μετά
   - Εύρεση νέας διεργασίας μέσω του αλγορίθμου
   - Load στον Memory Management Unit το χάρτη της νέας διεργασίας
   - Start της process
   - Αδειάσμα cache μνήμης

→ Τα switches αυτά οδηγούν σε σπατάλη CPU

compute-bound Μεγάλα διαστήματα χρήσης της CPU
I/O-bound Μικρά " " " " , αλλά μεγάλη αναμονή λόγω I/O

Πότε: γίνεται Χρονοπρογραμματισμός

1) Δημιουργία νέου Process
2) Τερματισμός διεργασίας
3) block process από Ε/Ε / semaphore / κάτι άλλο
4) Διακοπή από Ε/Ε

Είδη αλγορίθμων Χρονοπρογραμματισμού

Μη προεκχωρητικοί	Προεκχωρητικοί
Η διεργασία τρέχει μέχρι να βγει block	Η διεργασία τρέχει με συγκεκριμένο χρόνο. Αν τελειώσει, ανασταλεί, η εφαρμογή κλπ, επιλέγεται άλλη

Κατηγορίες Αλγορίθμων Χρονοπρογραμματισμού

Δέσμη:
- Δεν έχει users
- Προεκχωρητικοί ή όχι αλγόριθμοι αποδεκτοί
- Μειωμένες εναλλαγές → Καλύτερη απόδοση

Αλληλεπιδραστικά
- Προεκχώρηση απαραίτητη (για να μην παίρνει ένα process όλη τη CPU)

Πραγματικού χρόνου
- Προεκχώρηση μη απαραίτητη
- Οι διεργασίες γνωρίζουν πότε πρέπει να τελειώσουν και μοιράζονται σύντομα
→ Διαφορά με Αλληλεπιδραστικά
- Τα Real-time έχουν κατά συνθήκη προγράμματα που έχουν σκοπό να υποβοηθήσουν οι υπάρχουσες εφαρμογές
- Τα αλληλεπιδραστικά εξυπηρετούν γενικές ανάγκες

Σενάριοι Συστημάτων

Για όλα τα συστήματα	Συστήματα δέσμης	Αλληλεπιδραστικά	Real time
Δικαιοσύνη	Διαπερατικά ικανότητα	Χρόνος απόκρισης	Τήρηση προθεσμιών
Επιβολή πολιτικής	Χρόνος διαπερατικότητας	Τήρηση αναλογιών	Προβλεψιμότητα
Ισορροπία	Αξιοποίηση CPU		

2.4.9 Χρονοπρογραμματισμός Συστήματα δέσμης

1) first come first served :- Ουρά με διεργασίες
- 1η στην ουρά εκτελείται
- αν μπλοκαριστεί, συνεχίζει η επόμενη
- αν τερματιστεί, πάει στο τέλος

2) shortest job first

Βάζουμε στην αρχή αυτές με μικρότερο χρόνο ολοκλήρωσης

3) shortest remaining time next

Όποια διεργασία ~~εκτ~~ έχει το μικρότερο χρόνο εκτέλεσης, ξεκινάει πρώτη

2.4.3 Χρονοπρογραμματισμός σε αλληλεπιδραστικά συστήματα

1) Εκ περιτροπής (round bin)
- Κάθε process έχει κάποιο κβάντο χρόνου (ένα κομμάτι)
- Αν τελειώσει ο χρόνος και όχι η διεργασία, τότε η CPU παραχωρείται σε άλλη διεργασία
- Αν ολοκληρώθηκε ή blockαρίστηκε, τότε κάποια άλλη τρέχει

2) Βάσει προτεραιότητας
- Κάθε process έχει κάποια προτεραιότητα
- Πιο υψηλή προτεραιότητα → τρέχει πρώτο
- Priority -- σε κάθε tick ρολογιού
- Αν priority < next_priority_process → swap

3) Πολλαπλές ουρές

Ένα process 100 κβάντων
Αρχικά τρέχει 1 κβάντο → χάνει CPU
Μετά τρέχει 2, 4, 8, ... 64
7 switches αντί 100 του round bin

Αν ένα process χρησιμοποιεί όλα τα κβάντα του, έπεφτε σε επίπεδο priority

4) Μικρότερη διάρκεια
Δεχόμαστε inputs, estimated time per input exec / T_0
την επόμενη φορά T_1, νέα εκτίμηση $= aT_0 + (1-a)T_1$

T_0, $T_0/2 + T_1/2$, $T_0/4 + T_1/4 + T_2/2$, $T_0/8 + T_1/8 + T_2/4 + T_3/2$

aging technique

5) Εξισορρόπηση Προγραμματισμού
Υποχρέωση στους users που τρέχουν
η users, 1/η CPU each

6) Χρονοπρογραμματισμός με λοταρία
λαχνοί στις processes, ανά πόσο πάρουν την CPU

- Καλή απόδοση ως προς απόκριση

7) Δίκαιη διανομή
Κάθε user παίρνει ένα % της CPU και ο scheduler επιλέγει με τέτοιο τρόπο τις διεργασίες ώστε αυτό να εξασφαλίζεται

2.4.4 Χρονοπρογραμματισμός στα Real Time Systems

Χρόνος → πρωταρχικός ρόλος

Είδη

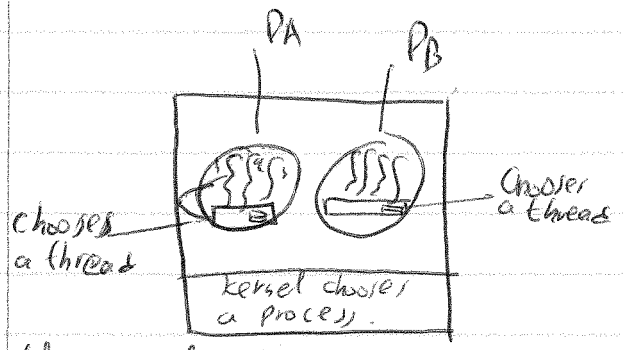
Αυστηρά	Χαλαρά
Τήρηση προθεσμιών των	Υπάρχει αποδεκτή στη μη τήρηση

Απόκριση σε
α) περιοδικά (κάποια χρονικά διαστήματα)
β) απεριοδικά (τυχαία)

m periodic events
event i occurs every P_i
and requires C_i time

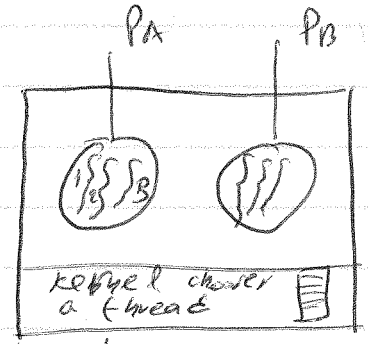
$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$$

2.4.6 Χρονοπρογραμματισμός νημάτων



User only threads

Πιθανή ακολουθία : $A_1 A_2 A_3 A_1 A_2 A_3$
αδύνατη : $A_1 B_1 A_2 B_2 A_3 B_3$



Kernel threads

Πιθανή ακολουθία: $A_1 A_2 A_3 A_1 A_2 A_3$
Και : $A_1 B_1 A_2 B_2 A_3 B_3$

Πρόβλημα φιλοσόφων

```
#define N 5                          /* 5 φιλόσοφοι */
#define LEFT (i+N-1)%N               /* αριστερά */
#define RIGHT (i+1)%N                /* δεξιά */
#define THINKING 0
#define HUNGRY 1
#define EATING 2
typedef int semaphore;
int state[N];
semaphore mutex = 1;
semaphore s[N];                      /* semaphore per philosopher */

void philosopher(int i) {
    while (1) {
        think();
        take_forks(i);
        eat();
        put_forks(i);
    }
}

void take_forks(int i) {
    down(&mutex);                    /* είσοδος κρίσιμης */
    state[i] = HUNGRY;               /* πεινάει */
    test(i);                         /* check left, right available */
    up(&mutex);                      /* exit κρίσιμη */
    down(&s[i]);                     /* block αν απαιτήθηκαν */
}

void put_forks(i) {
    down(&mutex);                    /* είσοδος κρίσιμη */
    state[i] = THINKING;
    test(LEFT);                      /* check left */
    test(RIGHT);                     /* check right */
    up(&mutex);                      /* exit κρίσιμη */
}

void test(int i) {
    if (state[i] == HUNGRY && state[LEFT] != EATING &&
        state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```

Αναχωρώντες γραφείς

```
typedef int semaphore;
semaphore mutex = 1;
semaphore db = 1;
int rc = 0;                    /* πλήθος διεργασιών που θέλουν να διαβάσουν */
void reader(void) {
  while (1) {
    down(&mutex);              /* αποκλειστικό access στο rc */
    rc = rc + 1;
    if (rc == 1) down(&db);    /* 1ος */
    up(&mutex);
    read_data_base();
    down(&mutex);
    rc = rc - 1;
    if (rc == 0) up(&db);
    up(&mutex);
    use_data_read();
  }
}

void writer(void) {
  while (1) {
    think_up_data();
    down(&db);
    write_data_base();
    up(&db);
  }
}
```

Πρόβλημα: μπορεί να έρχονται αναχωρητές συνέχεια και ο γραφείς να είναι forma-blocked.

Λύση: αν γραφείς περιμένει, οι επόμενοι readers θα μπουν μετά το γραφείο.

Μειονεκτήματα: concurrency και χαμηλή απόδοση