ΣΥΓΧΡΟΝΑ ΛΕΙΤΟΥΡΓΙΚΑ ΣΥΣΤΗΜΑΤΑ
3η έκδοση
ANDREW S. TANENBAUM

ΚΕΦΑΛΑΙΟ 2ο
Διεργασίες και Νήματα

Μετάφραση βασισμένη στις διαφάνειες της Αγγλικής έκδοσης

Κεφάλαιο 2
Διεργασίες και Νήματα

Περιεχόμενα

2.1 ΔΙΕΡΓΑΣΙΕΣ (PROCESSES)

Διεργασίες

- Διεργασία = Η αφαίρεση (abstraction) ενός προγράμματος που εκτελείται. Αποτελείται από:
  - Το **χώρο διευθύνσεων (address space)** της διεργασίας στη μνήμη που περιλαμβάνει:
    - τον **κώδικα** του προγράμματος που εκτελείται,
    - τα **δεδομένα** του προγράμματος και
    - τη **στοίβα**.
  - **Καταχωρητές**: όπως ο **μετρητής προγράμματος,** και ο **δείκτης στοίβας.**
  - Ανοικτά αρχεία, προειδοποιήσεις, λίστα σχετιζόμενων διεργασιών κτλ.

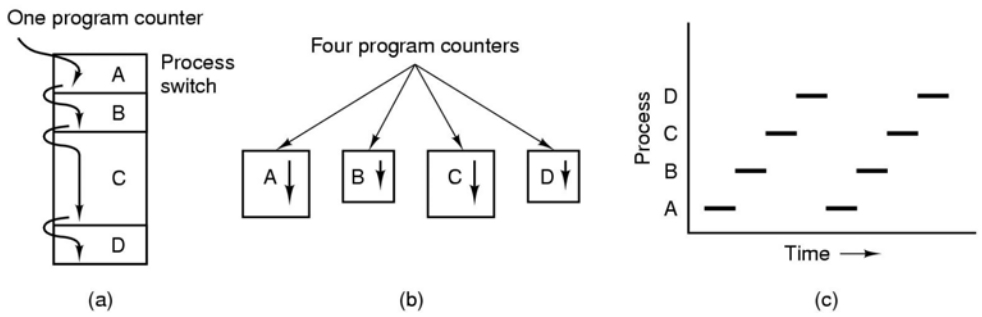
- Ένας Η/Υ εκτελεί (ψευδο)ταυτόχρονα πολλές διεργασίες.
  1. Παράδειγμα διεργασιών που εκτελούνται σε έναν εξυπηρετητή ιστού (web server):
     - Ο εξυπηρετητής δέχεται πολλές αιτήσεις από πελάτες για την προβολή ιστοσελίδων.
     - Κάθε αίτηση απαιτεί την αναζήτηση στο δίσκο.
     - Εφόσον η αναμονή για Ε/Ε είναι μεγάλη, ο εξυπηρετητής μεταβαίνει σε μία δεύτερη αίτηση, μέχρις ότου να είναι έτοιμα τα δεδομένα της προηγούμενης αίτησης.
  2. Παράδειγμα προσωπικού υπολογιστή:
     - Κατά την εκκίνηση του συστήματος πραγματοποιείται η εκκίνηση πολλών κρυφών και φανερών διεργασιών.

Το μοντέλο των διεργασιών

- Όλο το εκτελέσιμο λογισμικό οργανώνεται σε ένα πλήθος σειριακών διαδικασιών (sequential processes).
- Εννοιολογικά, κάθε διεργασία έχει τη δική της CPU.
  - Στην πραγματικότητα υπάρχει μόνο 1 CPU.
- Η γρήγορη εναλλαγή των διεργασιών στη CPU λέγεται πολυπρογραμματισμός (βλ. Κεφ.1).
  - (Ψευδο)παράλληλη εκτέλεση προγραμμάτων με 1 CPU.
- Η κοινή χρήση της CPU απαιτεί τη χρήση αλγορίθμων **χρονοπρογραμματισμού (scheduling algorithm).**

- Μία CPU εκτελεί κάθε στιγμή **1 μόνο διεργασία.**
  - Η εκτέλεση κάθε διεργασίας γίνεται για λίγα μόνο msec. Μετά το σύστημα μεταβαίνει σε μία άλλη διεργασία (και πάλι για λίγα msec).
  - Το αποτέλεσμα είναι η **(ψευδο) παραλληλία.**

Το μοντέλο των διεργασιών



Εικόνα 2-1. (a) Πολυπρογραμματισμός με τέσσερα προγράμματα. (b) Εννοιολογικό μοντέλο τεσσάρων ανεξάρτητων σειριακών διεργασιών. (c) Ενεργές διεργασίες στο χρόνο. Ένα πρόγραμμα είναι ενεργό κάθε στιγμή.

(a) Ένας **πραγματικός** μετρητής προγράμματος μεταφέρεται από το ένα πρόγραμμα στο άλλο.

(b) Το κάθε πρόγραμμα έχει το δικό του **λογικό** μετρητή προγράμματος.

(c) Σε κάποιο «μεγάλο» χρονικό διάστημα, π.χ. μερικά sec, όλα τα προγράμματα θα έχουν λάβει χρόνο στη CPU.

- Διαφορά μεταξύ διεργασίας και προγράμματος – παράδειγμα: ένας προγραμματιστής ετοιμάζει ένα γλυκό.
  - Συνταγή μαγειρικής: το πρόγραμμα.
  - Προγραμματιστής που μαγειρεύει: η CPU.
  - Συστατικά γλυκού: δεδομένα εισόδου.
  - Διεργασία = η δραστηριότητα που περιλαμβάνει την ανάγνωση της συνταγής από το μάγειρα, την ανάμιξη των υλικών και τη δημιουργία του γλυκού.
  - Σε περίπτωση που συμβεί κάτι σημαντικότερο, διακόπτει την παρασκευή του γλυκού. Εφόσον ολοκληρώσει την «επεξεργασία» της σημαντικότερης εργασίας επανέρχεται στη συνέχιση της παρασκευής του γλυκού (μεταγωγή επεξεργαστή σε διεργασία υψηλότερης προτεραιότητας).

Το μοντέλο των διεργασιών

- Η διεργασία είναι ένα **στιγμιότυπο (instance)** ενός εκτελέσιμου προγράμματος.
  - Εάν ένα πρόγραμμα εκτελεστεί δύο φορές, οδηγεί στη δημιουργία δύο διαφορετικών διεργασιών.
- Δεν πρέπει να γίνονται υποθέσεις για το χρόνο εκτέλεσης μίας διεργασίας.
  - Ένα πρόγραμμα που εκτελείται δύο φορές, ενδέχεται να έχει μεγάλες αποκλίσεις στο χρόνο εκτέλεσης!

Δημιουργία διεργασίας

Γεγονότα που προκαλούν τη δημιουργία μίας διεργασίας:

- (α) Εκκίνηση συστήματος.
- (β) Εκτέλεση μίας κλήσης συστήματος για τη δημιουργία διεργασίας (από μία άλλη διεργασία που εκτελείται).
- (γ) Αίτηση χρήστη για τη δημιουργία νέας διεργασίας.
- (δ) Εκκίνηση εργασίας δέσμης (batch job).

- (α) Κατά την εκκίνηση: Διεργασίες προσκηνίου και παρασκηνίου (foreground, background processes)
- (β) Κλήση συστήματος από μία εκτελούμενη διεργασία: Η διεργασία μπορεί να εκτελέσει την κλήση συστήματος fork ώστε να δημιουργήσει μία άλλη διεργασία.
- (γ) Η αίτηση χρήστη μπορεί να γίνει από το κέλυφος με την εκτέλεση μίας εντολής, ή μέσω μίας ενέργειας από το περιβάλλον GUI.
- (δ) Εργασία δέσμης: Το σύστημα δέσμης συγκεντρώνει εργασίες. Όταν το ΛΣ αποφασίσει ότι έχει αρκετούς διαθέσιμους πόρους, δημιουργεί μία διεργασία για την εκτέλεση της επόμενης στη σειρά εργασίας.

Τερματισμός διεργασίας

Γεγονότα που προκαλούν τον τερματισμό μίας διεργασίας είναι:

- (α) Κανονική έξοδος (εθελοντική).
- (β) Έξοδος σφάλματος (error exit) (εθελοντική).
- (γ) Έξοδος μοιραίου σφάλματος (fatal error) (μη εθελοντική).
- (δ) Τερματισμός από άλλη διεργασία (μη εθελοντική).

- (α) Τέλος διεργασίας, καλεί την κλήση exit().
- (β) π.χ. πληκτρολόγηση διαταγής που δεν γνωρίζει το σύστημα.
- (γ) π.χ. σφάλμα στον κώδικα του προγράμματος, διαίρεση με το 0 κτλ.
- (δ) Χρήση της κλήσης συστήματος kill.

Ιεραρχίες διεργασιών (UNIX)

- Μία (γονική) διεργασία μπορεί να δημιουργήσει μία ή περισσότερες θυγατρικές διεργασίες με την κλήση fork( ).
- Με αυτό τον τρόπο δημιουργούνται ιεραρχίες διεργασιών ή ομάδες διεργασιών (process group).
- Ένα σήμα από το πληκτρολόγιο προς μία διεργασία, θα διανεμηθεί σε όλη την ομάδα διεργασιών.
- Κάθε διεργασία της ομάδας επεξεργάζεται το σήμα.
  - Το δέχεται και εκτελεί κάποια προεπιλεγμένη ενέργεια.
  - Το αγνοεί.
  - Τερματίζεται.

- Κατά την εκκίνηση ενός συστήματος UNIX, εμφανίζεται μία ειδική διεργασία που λέγεται init.
- Η διεργασία αυτή δημιουργεί μία θυγατρική διεργασία για κάθε terminal που υπάρχει.
- Κάθε θυγατρική διεργασία περιμένει την επιτυχή σύνδεση χρήστη και δημιουργεί ένα κέλυφος (shell).
- Όλες οι διεργασίες του συστήματος δημιουργούν ένα δένδρο διεργασιών με ρίζα την init.

Ιεραρχίες διεργασιών (Windows)

- Στα Windows δεν υπάρχει ιεραρχία διεργασιών .
- Όλες οι διεργασίες είναι ισοδύναμες.
- Όταν δημιουργείται μία διεργασία, η «γονική» της αποκτά ένα ειδικό χειριστή (handle), μέσω του οποίου μπορεί να διαχειρίζεται τη θυγατρική διεργασία.
- Μπορεί να μεταβιβάσει το χειριστή σε κάποια άλλη διεργασία, παραβιάζοντας την ιεραρχία.
  - Κάτι τέτοιο δεν είναι δυνατό στο UNIX.

Καταστάσεις διεργασιών

cat file1 file2 | grep tree

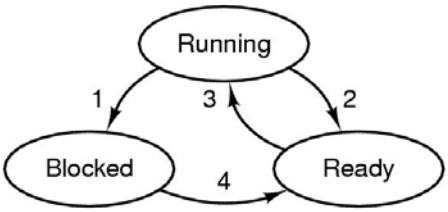
- Συνένωση δύο αρχείων και αποστολή στην έξοδο.
- Η σωλήνωση δίνει το αποτέλεσμα της διεργασίας ως είσοδο στην εντολή **grep** ώστε να αναζητηθεί μία λέξη.
- Εάν η **cat** δεν έχει ολοκληρωθεί, η **grep** παραμένει μπολκαρισμένη.

Το μπλοκάρισμα μίας διεργασίας μπορεί γενικά να συμβεί για δύο λόγους:

1.Η διεργασία μπλοκάρεται ενδογενώς, γιατί π.χ. περιμένει κάποια είσοδο.

2.Η διεργασία μπλοκάρεται εξωγενώς, γιατί το ΛΣ αποφασίζει να παραχωρήσει τη CPU σε κάποια άλλη διεργασία, δηλαδή λόγω κάποιας διακοπής (interrupt).

Καταστάσεις διεργασιών

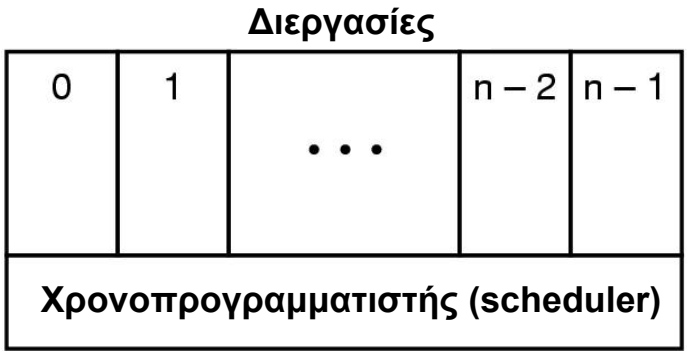


1. Η διεργασία μπλοκάρεται (περιμένει δεδομένα εισόδου).
2. Η διεργασία διακόπτεται από τον scheduler (χρονοπρογραμματιστή). Η CPU δώθηκε σε άλλη διεργασία.
3. Ο scheduler επιλέγει την διεργασία για να τρέξει στη CPU
4. Τα δεδομένα εισόδου που αναμένει η διεργασία είναι διαθέσιμα.

Εικόνα 2-2. Μία διεργασία μπορεί να εκτελείται, να είναι μπλοκαρισμένη ή να είναι έτοιμη για εκτέλεση.

1. Εκτελούμενη (running): Χρησιμοποιεί τη CPU.
2. Έτοιμη (ready): Είναι έτοιμη για εκτέλεση αλλά έχει διακοπεί προσωρινά.
3. Μπλοκαρισμένη (blocked): Δεν μπορεί να συνεχίσει την εκτέλεσή της, εάν δεν συμβεί κάποιο εξωτερικό γεγονός.
    - Στην περίπτωση αυτή, η διεργασία δεν μπορεί να εκτελεστεί, ακόμα και εάν η CPU είναι διαθέσιμη.

Υλοποίηση των διεργασιών (1)



Εικόνα 2-3. Το κατώτερο στρώμα του ΛΣ χειρίζεται τις διακοπές και το χρονοπρογραμματισμό.
Πάνω από αυτό το επίπεδο βρίσκονται οι σειριακές διεργασίες.

- Ο χειρισμός των σημάτων (signals) και των διακοπών (interrupts) γίνεται από το επίπεδο του χρονοπρογραμματιστή.
  - Ο scheduler υλοποιείται από ελάχιστο κώδικα.
- Το υπόλοιπο του ΛΣ υλοποιείται μέσα από τις διεργασίες.
  - Στην πραγματικότητα λίγα ΛΣ ακολουθούν ακριβώς αυτή τη δομή.

Υλοποίηση των διεργασιών (2)

Διαχείριση διεργασιών	Διαχείριση μνήμης	Διαχείριση αρχείων
Καταχωρητές (registers) Μετρητής (Program counter) Program Status Word (PSW) Δείκτης στοίβας (stack pointer) Κατάσταση διεργασίας Προτεραιότητα Παράμετροι scheduling Ταυτότητα διεργασίας (PID) Ταυτότητα γονικής (PPID) Ομάδα διεργασίας Σήματα Χρόνος εκκίνησης διεργασίας Χρόνος χρήσης CPU Χρόνος χρήσης CPU από θυγατρικές διεργασίες Χρονική στιγμή επόμενης ειδοποίησης	Δείκτης τμήματος κώδικα Δείκτης τμήματος δεδομένων Δείκτης τμήματος στοίβας	Βασικός κατάλογος Κατάλογος εργασίας Περιγραφείς αρχείων Ταυτότητα χρήστη (user ID) Ταυτότητα ομάδας (group ID)

Εικόνα 2-4. Μερικά πεδία μιας τυπικής καταχώρησης του **πίνακα διεργασιών (process table)**.

- Το ΛΣ διατηρεί μία δομή δεδομένων που ονομάζεται **πίνακας διεργασιών (process table)** με τα παραπάνω στοιχεία.

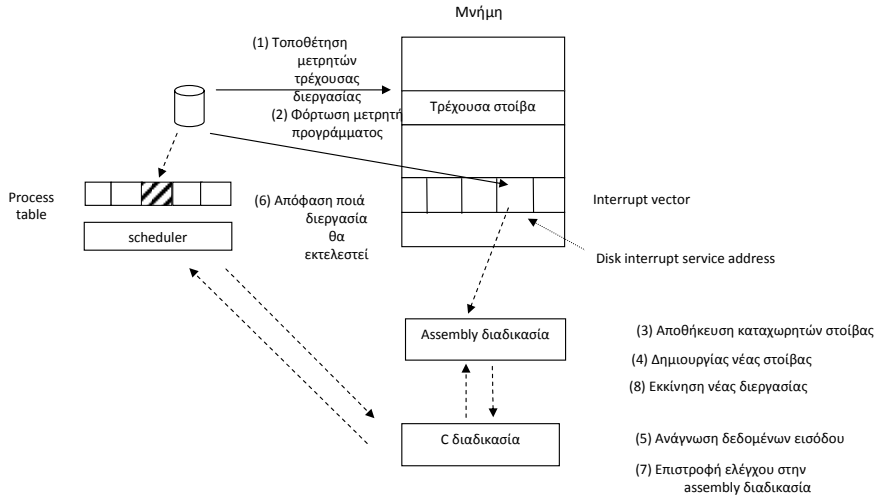
Υλοποίηση των διεργασιών (3)

1. Το υλικό τοποθετεί στη στοίβα τους μετρητές της διεργασίας που τρέχει.
2. Το υλικό φορτώνει το μετρητή του προγράμματος που δείχνει το διάνυσμα διακοπών (interrupt vector).
3. Η assembly διαδικασία αποθηκεύει τους καταχωρητές της τρέχουσας διεργασίας.
4. Η assembly διαδικασία δημιουργεί μία νέα προσωρινή στοίβα για το χειριστή διεργασιών.
5. Η C διαδικασία εξυπηρέτησης διακοπής (interrupt service) εκτελείται. Διαβάζει την είσοδο και την αποθηκεύει σε ένα buffer. Φέρνει σε ετοιμότητα μία διεργασία και καλεί τον χρονοπρογραμματιστή.
6. Ο χρονοπρογραμματιστής (scheduler) αποφασίζει ποια διεργασία θα εκτελεστεί.
7. Η C διαδικασία επιστρέφει τον έλεγχο στην αντίστοιχη assembly διαδικασία της διεργασίας που θα εκτελεστεί.
8. Η διαδικασία assembly εκκινεί τη νέα διεργασία.

Εικόνα 2-5. Οι ενέργειες που εκτελεί το χαμηλότερο επίπεδο του ΛΣ όταν συμβεί κάποια διακοπή.

- Σε κάθε συσκευή Ε/Ε αντιστοιχεί μία θέση στη μνήμη που λέγεται **διάνυσμα διακοπών (interrupt vector).**
- Περιλαμβάνει τη διεύθυνση μίας **διαδικασίας εξυπηρέτησης διακοπών (interrupt service vector).**

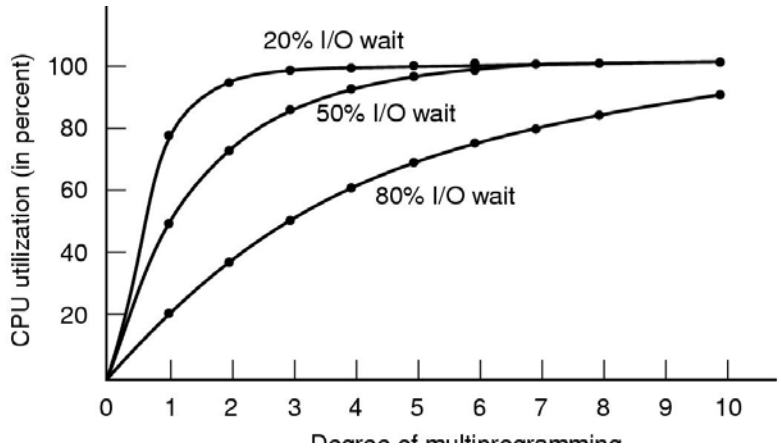
Υλοποίηση των διεργασιών (4)



Μοντελοποίηση πολυπρογραμματισμού (1)

- Εάν 1 διεργασία κάνει κατά Μ.Ο. χρήση της CPU στο 20% του χρόνου που βρίσκεται στη μνήμη, χρειάζονται 5 διεργασίες στη μνήμη για να έχουμε 100% *αξιοποίηση (utilization)* της CPU.
  - Εφόσον δεν περιμένουν Ε/Ε και οι 5 ταυτόχρονα.
- Θεωρία πιθανοτήτων:
  - Μ.Ο. αναμονής Ε/Ε: ***p%*** του χρόνου.
  - Με ***n*** διεργασίες στη μνήμη, η πιθανότητα για αδράνεια της CPU είναι p^n.
  - **Αξιοποίηση της CPU = 1 – p^n**

Μοντελοποίηση πολυπρογραμματισμού (2)



Εικόνα 2-6. Αξιοποίηση της CPU ως συνάρτηση του αριθμού των διεργασιών στη μνήμη.

- Στην πράξη χρόνοι αναμονής >80% είναι συνηθισμένοι.

Μοντελοποίηση πολυπρογραμματισμού (3)

- Το πιθανοθεωρητικό μοντέλο είναι απλοϊκό.
  - Υποθέτει ότι μπορούν να εκτελούνται περισσότερες διεργασίες ταυτόχρονα.
  - Στην πράξη χρησιμοποιείται θεωρία ουρών αναμονής.
- Χρήσιμο για προσεγγιστικούς υπολογισμούς.
  - 512 MB μνήμης, 128 MB για το ΛΣ, και κατά Μ.Ο. 128 MB ανά πρόγραμμα χρήστη.
  - 3 προγράμματα ταυτόχρονα στη μνήμη.
  - Με 80% αναμονή, αξιοποίηση = $1 - (0.8)^3 = 0.49$ ή 49%
  - Τι θα γίνει με άλλα 512 MB ή 1 GB μνήμης;

- Με 512 MB πρόσθετη μνήμη, αύξηση από 3πλο προγραμματισμό σε 7πλο και αξιοποίηση από 49% σε 79%.
- Με 1 GB πρόσθετη μνήμη, αύξηση από 3πλο προγραμματισμό σε 9πλο και αξιοποίηση από 49% σε 91%.

2.2 NHMATA (THREADS)

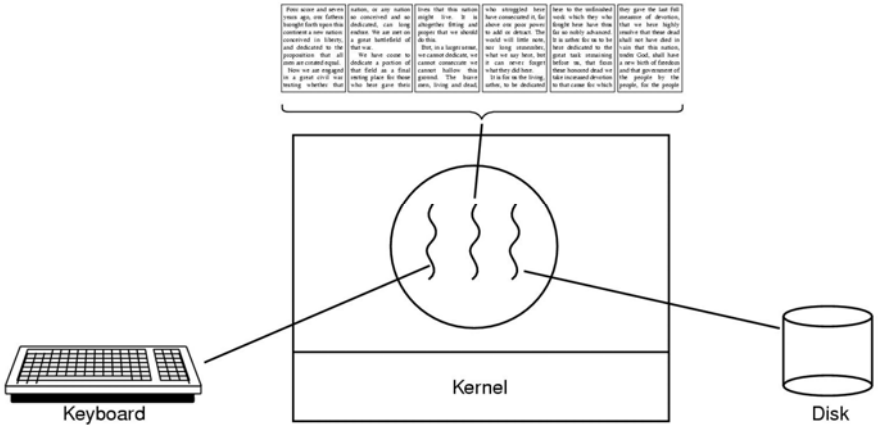
Τι είναι τα νήματα

- Διεργασία: ένα νήμα ελέγχου σε ένα χώρο διευθύνσεων.
- Σε πολλές περιπτώσεις υπάρχει η ανάγκη χρήσης περισσότερων νημάτων ελέγχου στον ίδιο χώρο διευθύνσεων
- Νήμα: ένα πρόγραμμα που εκτελείται, σε κοινό χώρο διευθύνσεων με άλλα νήματα.
- Επιτρέπει την (ψευδο)παράλληλη εκτέλεση εργασιών στον ίδιο χώρο διευθύνσεων.

Πλεονεκτήματα των νημάτων

1. Απαραίτητη η κοινή χρήση χώρου διευθύνσεων σε ορισμένες εφαρμογές.
2. Ελαφρύτερα από τις διεργασίες (10-100 φορές η δημιουργία, καταστροφή τους).
3. Καλύτερη επικάλυψη εργασιών σε περίπτωση μεγάλου υπολογιστικού φόρτου και παράλληλα εκτεταμένης χρήσης Ε/Ε.

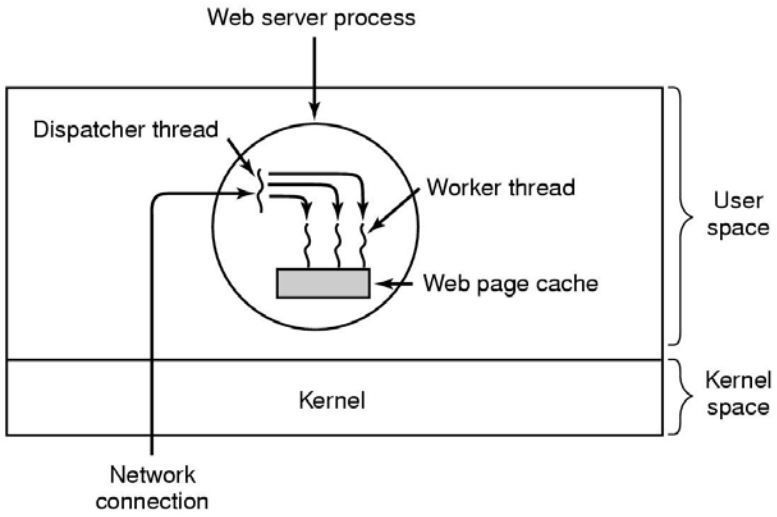
Χρήση των νημάτων (1)



Εικόνα 2-7. Ένας επεξεργαστής κειμένου με τρία νήματα.

- Εάν το πρόγραμμα ήταν υλοποιημένο με ένα νήμα, τότε την ώρα που γίνεται αυτόματη αποθήκευση, θα ήταν αδύνατη η λήψη εισόδου από πληκτρολόγιο.
- Η χρήση 3 ανεξάρτητων διεργασιών δεν θα έλυνε το πρόβλημα, εφόσον κάθε διεργασία θα είχε ανεξάρτητο χώρο διευθύνσεων.

Χρήση των νημάτων (2)



Εικόνα 2-8. Εξυπηρετητής ιστού με τη χρήση πολυνημάτωσης (multithreaded Web server).

- Κάθε αίτημα πελάτη, αντιστοιχίζεται από το νήμα διεκπεραίωσης (dispatcher thread) σε ένα νήμα εργασίας (worker thread).
- Όλα τα νήματα εργασίας έχουν πρόσβαση στην ίδια cache μνήμη.

Χρήση των νημάτων (3)

```
while (TRUE) {
    get_next_request(&buf);
    handoff_work(&buf);
}
```

(a)

```
while (TRUE) {
    wait_for_work(&buf)
    look_for_page_in_cache(&buf, &page);
    if (page_not_in_cache(&page))
        read_page_from_disk(&buf, &page);
    return_page(&page);
}
```

(b)

Εικόνα 2-9. Ένας γενικός σκελετός του κώδικα ενός multithreaded web server (Σχήμα 2-8).
(a) Το νήμα διεκπεραίωσης (Dispatcher thread).
(b) Το νήμα εργασίας (Worker thread).

- Εάν δεν υπήρχαν πολλά νήματα, ο εξυπηρετητής θα εξυπηρετούσε μόνο ένα αίτημα κάθε φορά (μέχρι να τελειώσει το αίτημα θα ήταν σε αναμονή).

Χρήση των νημάτων (4)

Model	Characteristics
Threads	Parallelism, blocking system calls
Single-threaded process	No parallelism, blocking system calls
Finite-state machine	Parallelism, nonblocking system calls, interrupts

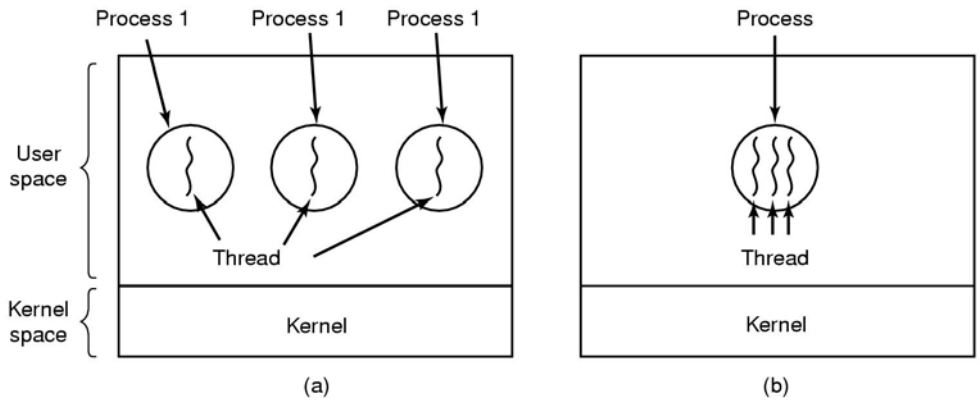
Εικόνα 2-10. Τρία μοντέλα κατασκευής ενός server.

- Στην 3η περίπτωση, θεωρούμε ότι υπάρχει μία έκδοση της κλήσης συστήματος read η οποία δεν οδηγεί σε μπλοκάρισμα (non-blocking system call).
  - Ο εξυπηρετητής γράφει την κατάσταση της τρέχουσας αίτησης σε έναν πίνακα και λαμβάνει την επόμενη αίτηση (ή την απάντηση από το δίσκο).
  - Προσομοίωση της λειτουργίας των νημάτων (με δύσκολο τρόπο).
  - Οδηγεί σε μία **μηχανή πεπερασμένων καταστάσεων (finate state machine)**, δηλαδή κάθε υπολογισμός διαθέτει μία αποθηκευμένη κατάσταση και κάθε συμβάν μπορεί να αλλάξει αυτή την κατάσταση.
- Η ύπαρξη των νημάτων οδηγεί στη διατήρηση ων κλήσεων συστήματος που προκαλούν μπλοκάρισμα (άρα ευκολότερο προγραμματισμό) και ταυτόχρονα επιτυγχάνουν παραλληλία.

Το κλασικό μοντέλο των νημάτων (1)

- Κλασικό μοντέλο διεργασίας: ομαδοποίηση συναφών πόρων (π.χ. ανοικτά αρχεία, θυγατρικές διεργασίες, εκκρεμή σήματα, κτλ)
- Κάθε διεργασία με ένα νήμα εκτέλεσης διαθέτει:
  1. ένα μετρητή προγράμματος που δείχνει την επόμενη εντολή που θα εκτελεστεί,
  2. καταχωρητές για τις τρέχουσες μεταβλητές,
  3. μία στοίβα με το ιστορικό εκτέλεσης.
- Θα μπορούσαμε να έχουμε **πολλές ροές εκτέλεσης** στον ίδιο χώρο διευθύνσεων.
  - Άρα πολλά νήματα σε μία διεργασία.

Το κλασικό μοντέλο των νημάτων (2)



Εικόνα 2-11. (a) Τρεις διεργασίες με ένα νήμα η κάθε μία. (b) Μια διεργασία με τρία νήματα.

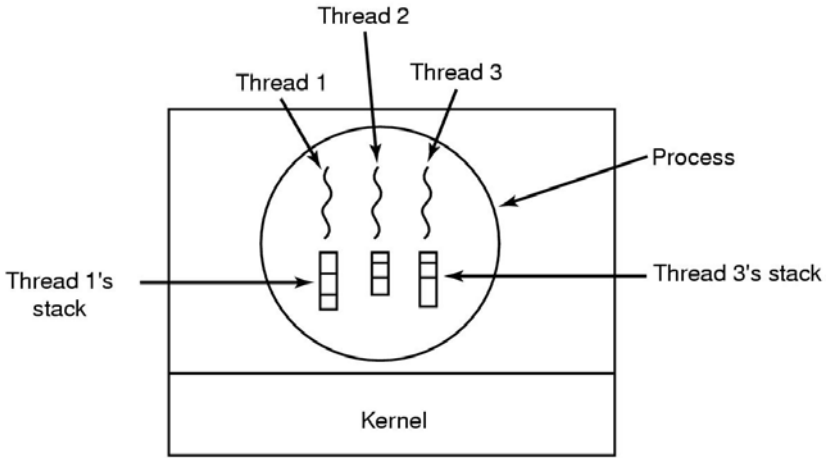
Το κλασικό μοντέλο των νημάτων (3)

Per process items	Per thread items
Address space (Χώρος διευθύνσεων)	Program counter (Μετρητής προγράμματος)
Global variables (Καθολικές μεταβλητές)	Registers (Καταχωρητές)
Open files (Ανοικτά αρχεία)	Stack (Στοίβα)
Child processes (Θυγατρικές διεργασίες)	State (Κατάσταση)
Pending alarms (Εκκρεμή σήματα)	
Signals and signal handlers	
Accounting information (Πληροφ. διαχείρισης)	

Εικόνα 2-12.
1η στήλη: τι μοιράζονται μεταξύ τους τα νήματα μίας διεργασίας.
2η στήλη: ατομικά στοιχεία κάθε νήματος.

- Στην περίπτωση πολλών νημάτων μέσα σε μία διεργασία, κάθε νήμα μπορεί να διαβάσει, γράψει ή διαγράψει τη στοίβα κάθε άλλου νήματος!
- Τα νήματα μοιράζονται επίσης (εκτός από το χώρο διευθύνσεων) τα ανοικτά αρχεία, θυγατρικές διεργασίες, σήματα κτλ.

Το κλασικό μοντέλο των νημάτων (4)



Εικόνα 2-13. Κάθε νήμα έχει τη δική του στοίβα.

- Η στοίβα κάθε νήματος περιέχει ένα πλαίσιο (frame) για κάθε διαδικασία που κλήθηκε από το νήμα και δεν έχει ολοκληρωθεί.
- Κάθε νήμα, καλεί συνήθως **διαφορετικές διαδικασίες** (εφόσον εκτελεί διαφορετική εργασία). Είναι προφανές ότι δημιουργεί διαφορετικό ιστορικό εκτέλεσης από τα άλλα νήματα
    - Γι' αυτό κάθε νήμα χρειάζεται τη δική του στοίβα.

Τα νήματα στο POSIX (1)

Thread call	Description
Pthread_create	Create a new thread
Pthread_exit	Terminate the calling thread
Pthread_join	Wait for a specific thread to exit
Pthread_yield	Release the CPU to let another thread run
Pthread_attr_init	Create and initialize a thread's attribute structure
Pthread_attr_destroy	Remove a thread's attribute structure

Εικόνα 2-14. Μερικές βασικές κλήσεις συστήματος της ομάδας Pthreads.

- Όταν αρχίζει μία διεργασία με πολυνημάτωση, ξεκκινά συνήθως με ένα νήμα εκτέλεσης.
- Μετά χρησιμοποιεί την Pthread_create για να δημιουργήσει έο νήμα (με όρισμα το όνομα της διαδικασίας που καλεί).
- Για να είναι δυνατή η γραφή φορητών νημάτων, το πρότυπο POSIX ορίζει ένα πρότυπο για τα νήματα, το Pthreads.
    - Ορίζει περίπου 60 κλήσεις συναρτήσεων.

Τα νήματα στο POSIX (2)

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define NUMBER_OF_THREADS    10

void *print_hello_world(void *tid)
{
      /* This function prints the thread's identifier and then exits. */
      printf("Hello World. Greetings from thread %d0, tid);
      pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
      /* The main program creates 10 threads and then exits. */
      pthread_t threads[NUMBER_OF_THREADS];
      int status, i;

      for(i=0; i < NUMBER_OF_THREADS; i++) {
            printf("Main here. Creating thread %d0, i);
            status = pthread_create(&threads[i], NULL, print_hello_world, (void *)i);

            if (status != 0) {
                  printf("Oops. pthread_create returned error code %d0, status);
                  exit(-1);
            }
      }
      exit(NULL);
}
```

Εικόνα 2-15. Παράδειγμα χρήσης νημάτων.

Με βάση αυτό το πρόγραμμα δίνεται η 1η Bonus εργασία:

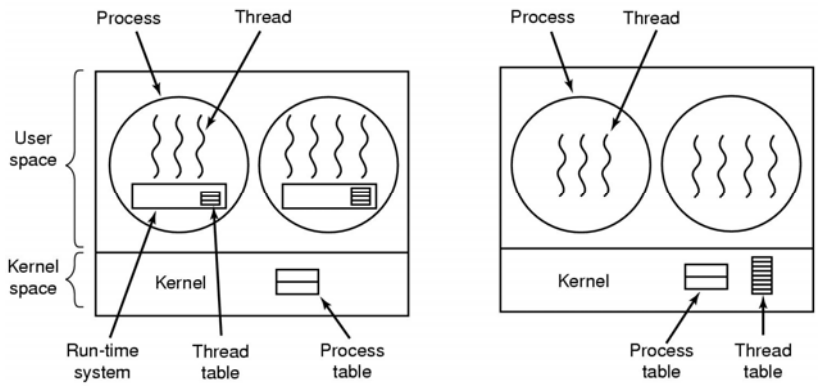
Δημιουργήστε ένα πρόγραμμα σε C, σε περιβάλλον POSIX (π.χ. Linux) το οποίο:

(α) Εμφανίζει τα attributes κάθε νήματος, αμέσως μετά τη δημιουργία κάθε νήματος.

(β) Δημιουργήστε ένα νήμα «καταστροφέα» το οποίο διαγράφει τα χαρακτηριστικά ενός άλλου νήματος και μετά το τερματίζει (να εμφανίζονται τα αντίστοιχα μηνύματα στην οθόνη).

(γ) Δημιουργήστε ένα νήμα το οποίο είναι «ευγενικό» και παραχωρεί την εκτέλεση στα υπόλοιπα. Πότε θα ολοκληρωθεί η εκτέλεσή του;

Υλοποίηση νημάτων στο χώρο του χρήστη (user space)



Εικόνα 2-16. (a) Πακέτο νημάτων επιπέδου χρήστη. (b) Πακέτο νημάτων που διαχειρίζεται από τον πυρήνα.

Δύο κύριοι τρόποι για την υλοποίηση ενός πακέτου νημάτων:

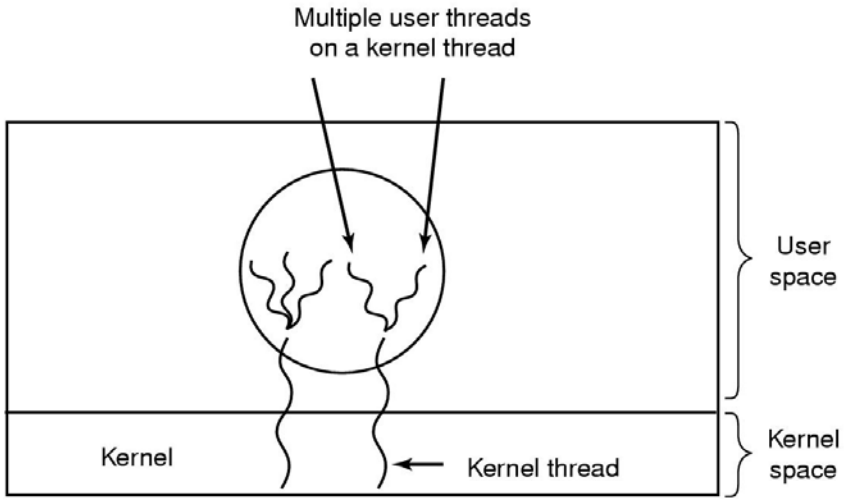
(A) στο χώρο του χρήστη (user space)

- Ο πυρήνας θεωρεί ότι εκτελεί συνηθισμένες διεργασίες με ένα νήμα η κάθε μία.
- Δεν χρειάζεται το ΛΣ να υποστηρίζει νήματα!
- Ο πίνακας διεργασιών (process table) βρίσκεται στον πυρήνα (πάντοτε). Όμως σε αυτή την περίπτωση ο πίνακας νημάτων (thread table) βρίσκεται στο χώρο χρήστη, εντός κάθε διεργασίας.
- Το πρόγραμμα χρήστη αποφασίζει πως θα μοιράσει το χρόνο εκτέλεσης που έχει όλη η διεργασία μεταξύ των νημάτων της.
- Γρηγορότερη η εναλλαγή μεταξύ των νημάτων (εφόσον δεν χρειάζεται παγίδευση στον πυρήνα).
- Όμως τα νήματα πρέπει να παραχωρούν αυτόβουλα τη CPU στα άλλα νήματα της διεργασίας.

(B) στο χώρο του πυρήνα (kernel space)

- Σε αυτή την περίπτωση, ο πίνακας νημάτων βρίσκεται στον πυρήνα.
- Μεγαλύτερο κόστος από την προηγούμενη λύση (εφόσον η δημιουργία / καταστροφή των νημάτων χρειάζεται μία κλήση συστήματος στον πυρήνα).
- Πολλές φορές γίνεται για λόγους βελτίωσης της απόδοσης «ανακύκλωση» νημάτων. Δηλαδή, εάν ένα νήμα δεν χρειάζεται δεν καταστρέφεται, αλλά σημειώνεται ως μη-χρήσιμο (ώστε να το χρησιμοποιήσει αργότερα μία άλλη λειτουργία).
- Το πλεονέκτημα είναι ότι η εναλλαγή μεταξύ των νημάτων γίνεται πλέον από τον πυρήνα (άρα δεν χρειάζεται αυτόβουλη παραχώρηση της CPU).

Υβριδικές υλοποιήσεις



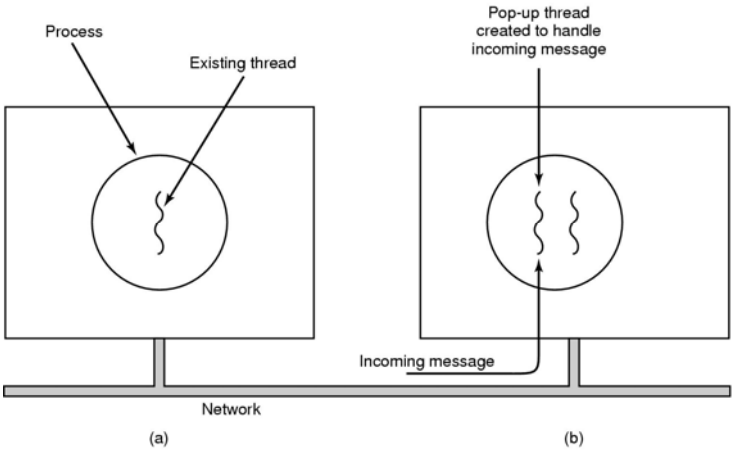
Εικόνα 2-17. Πολύπλεξη νημάτων επιπέδου χρήστη σε νήματα επιπέδου πυρήνα.

- Συνδυαστική λύση: ένα νήμα επιπέδου πυρήνα, μπορεί να χειριστεί πολλά νήματα επιπέδου χρήστη.

Ενεργοποιήσεις χρονοπρογραμματιστή (Scheduler Activations)

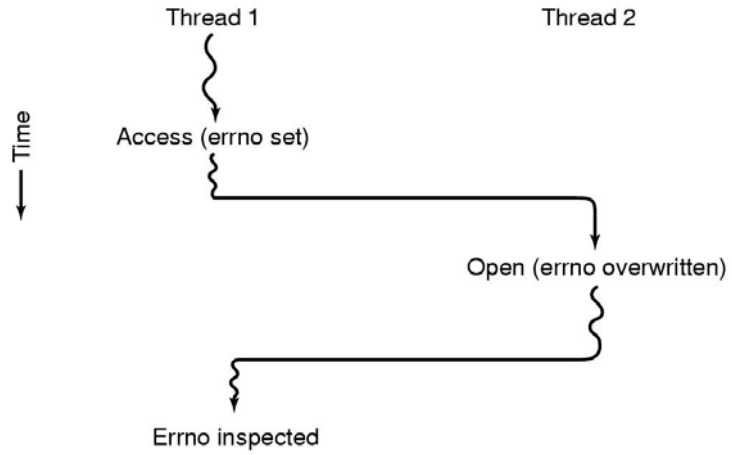
- Στόχος: μίμηση της λειτουργίας των νημάτων πυρήνα
  - Βελτίωση απόδοσης των νημάτων επιπέδου χρήστη
- Αποφυγή μη αναγκαίων μεταβάσεων από κατάσταση χρήστη σε κατάσταση πυρήνα
- Ο πυρήνας αναθέτει εικονικούς επεξεργαστές σε κάθε διεργασία
  - Επιτρέπει στο σύστημα χρόνου εκτέλεσης (runtime system) να κατανέμει τα νήματα σε επεξεργαστές
- Πρόβλημα:
  - Βασίζονται εγγενώς στον πυρήνα (κατώτερα στρώματα)
  - Οι καλούμενες διεργασίες βρίσκονται στο χώρο χρήστη (ανώτερο στρώμα)

Αναδυόμενα νήματα (Pop-Up Threads)



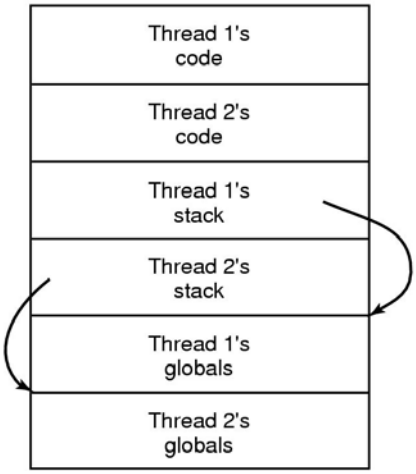
Εικόνα 2-18. Δημιουργία νέου νήματος κατά την άφιξη ενός μηνύματος. (a) Πριν την άφιξη του μηνύματος. (b) Μετά την άφιξη του μηνήματος.

Μετατροπή μονονηματικού κώδικα σε πολυνηματικό (1)



Εικόνα 2-19. Διενέξεις μεταξύ νημάτων για τη χρήση μίας καθολικής μεταβλητής.

Μετατροπή μονονηματικού κώδικα σε πολυνηματικό (2)



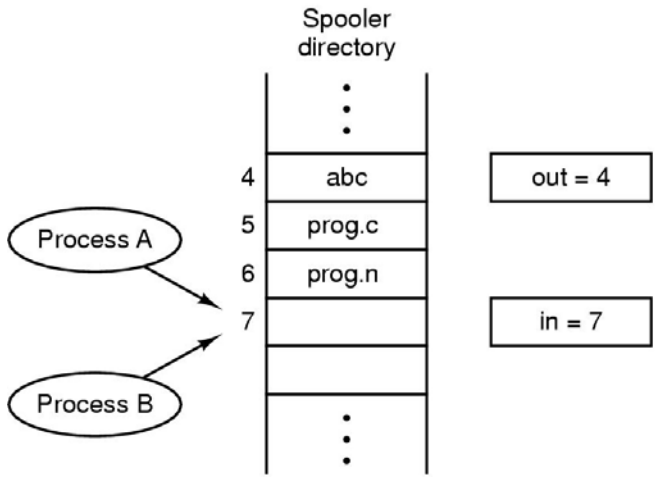
Εικόνα 2-20. Τα νήματα είναι δυνατόν να έχουν ιδιωτικές καθολικές μεταβλητές

2.3. ΔΙΑΔΙΕΡΓΑΣΙΑΚΗ ΕΠΙΚΟΙΝΩΝΙΑ (INTERPROCESS COMMUNICATION)

Διαδιεργασιακή επικοινωνία

1. Με **ποιο τρόπο επικοινωνούν μεταξύ τους** δύο διεργασίες;
   - Μία σωλήνωση (pipe) επιτρέπει για παράδειγμα την επικοινωνία μεταξύ δύο διεργασιών.
   - Πολλά ΛΣ δημιουργούν «κοινόχρηστους» χώρους διευθύνσεων.
2. Πώς μπορούμε να εξασφαλίσουμε ότι μία διεργασία **δεν εμποδίζεται από μία άλλη**, όταν η 1η εκτελεί μία *κρίσιμη λειτουργία*;
3. Πως εξασφαλίζουμε το **σωστό συγχρονισμό** μεταξύ διεργασιών που πρέπει να επικοινωνήσουν;

Συνθήκες ανταγωνισμού



Εικόνα 2-21. Δύο διεργασίες που θέλουν ταυτόχρονα να προσπελάσουν κοινή μνήμη.

Οι διεργασίες Α και Β αποφασίζουν ταυτόχρονα να τοποθετήσουν στην ουρά εκτύπωσης ένα αρχείο προς εκτύπωση η κάθε μία.

• **Εκτελείται η Α:**

- Η Α διαβάζει ποια είναι η επόμενη διαθέσιμη θέση (*in* = *7*) και αποθηκεύει αυτή την τιμή σε μία τοπική μεταβλητή (next_free_slot = 7).

• Γίνεται μία διακοπή ρολογιού **και εκτελείται η Β**.

- Η Β διαβάζει επίσης ποια είναι η επόμενη διαθέσιμη θέση (*in* = *7*) και αποθηκεύει αυτή την τιμή σε μία τοπική μεταβλητή (next_free_slot = 7).
- Αποθηκεύει το αρχείο της στη θέση 7.
- Αυξάνει την global μεταβλητή *in* κατά ένα και γίνεται (in = next_free_slot + 1 = 7+1 =8). (Η Α δεν είχε προλάβει πριν να αυξήσει την in).

• Γίνεται μία διακοπή ρολογιού **και συνεχίζει η Α**.

- Διαβάζει την τοπική μεταβλητή της (*next_free_slot* = *7*) και τοποθετεί το αρχείο της στη θέση 7 **διαγράφοντας το αρχείο της Β!**
- Αυξάνει την global μεταβλητή *in* κατά ένα και γίνεται (in = next_free_slot + 1 = 7+1 =8).

• Τίποτα ασυνήθιστο δεν έχει συμβεί για τον print daemon. Όλοι λένε ότι η επόμενη κενή θέση είναι η 8!

Κρίσιμες περιοχές (1)

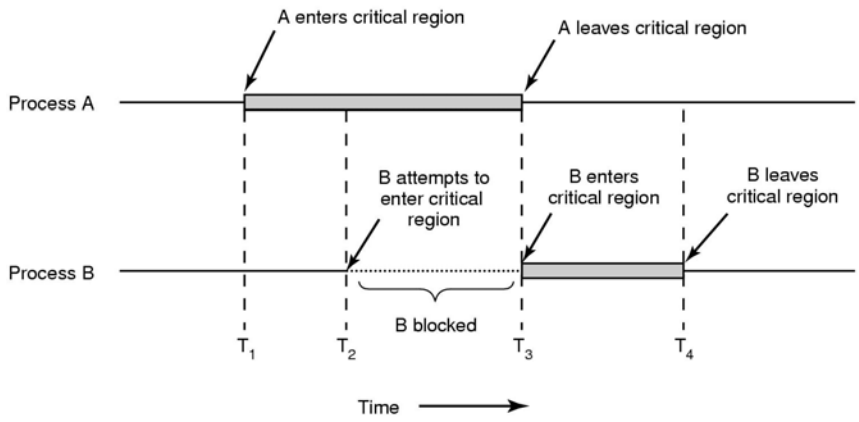
Κρίσιμη περιοχή = τμήμα προγράμματος που γίνεται προσπέλαση κοινόχρηστης μνήμης.

Συνθήκες για την αποφυγή την ανταγωνισμού:

1. Δύο διεργασίες δεν βρίσκονται ***ποτέ ταυτόχρονα*** στις κρίσιμες περιοχές τους **(αμοιβαίος αποκλεισμός)**.
2. Δεν επιτρέπονται παραδοχές σχετικά με την ταχύτητα ή το πλήθος των CPU.
3. Αν μία διεργασία δεν βρίσκεται σε κρίσιμο τμήμα, δεν επιτρέπεται να μπλοκάρει άλλες διεργασίες.
4. Δεν επιτρέπεται η επ' αόριστο αναμονή μίας διεργασίας, για την είσοδό της στην κρίσιμη περιοχή της.

Αμοιβαίος αποκλεισμός (mutual exclusion): Εάν μία διεργασία χρησιμοποιεί μία κοινόχρηστη μεταβλητή σε μία δεδομένη στιγμή, τότε καμία άλλη διεργασία δεν χρησιμοποιεί την κοινόχρηστη μεταβλητή (ή γενικότερα τον κοινόχρηστο πόρο) την ίδια στιγμή.

Κρίσιμες περιοχές (2)



Εικόνα 2-22. Αμοιβαίος αποκλεισμός με τη χρήση κρίσιμων περιοχών.

Αμοιβαίος αποκλεισμός μέσω αναμονής με απασχόληση (busy waiting)

Μέθοδοι για την επίτευξη αμοιβαίου αποκλεισμού:

- Απενεργοποίηση διακοπών
- Μεταβλητές κλειδώματος
- Αυστηρή εναλλαγή
- Η λύση του Peterson
- Η εντολή TSL

Απενεργοποίηση διακοπών

- Όταν μπαίνει μία διεργασία στην κρίσιμη περιοχή της απενεργοποιεί όλες τις διακοπές. Τις ενεργοποιεί ξανά μόλις βγει από την κρίσιμη περιοχή της.
- Επιτρέπεται στις διεργασίες να απενεργοποιούν τις διακοπές ρολογιού.
- Μία κακόβουλη διεργασία μπορεί να μην επαναφέρει ποτέ τις διακοπές (συνεπώς να μην μπορέσει ποτέ να δοθεί η CPU σε άλλη διεργασία!)

Μεταβλητές κλειδώματος

- Χρήση κοινόχρηστης μεταβλητής κλειδώματος (lock variable).
- Κάθε διεργασία ελέγχει τη μεταβλητή κλειδώματος:
  - Αν είναι 0 μπαίνει στην κρίσιμη περιοχή και την αλλάζει σε 1.
  - Αν είναι 1, περιμένει για λίγο και ξαναδοκιμάζει.
- Πρόβλημα:
  - Η Α διαβάζει την μεταβλητή και τη βρίσκει 0. Συμβαίνει μία διακοπή ρολογιού.
  - Η Β διαβάζει τη μεταβλητή και τη βρίσκει 0. Την αλλάζει σε 1 και μπαίνει στην κρίσιμη περιοχή της. Συμβαίνει διακοπή ρολογιού.
  - Η Α συνεχίζει από εκεί που είχε μείνει. Αλλάζει την μεταβλητή σε 1 (*παρόλο που είναι ήδη δεν το ξέρει*) και μπαίνει και αυτή στην κρίσιμη περιοχή της!

- Το πρόβλημα οφείλεται στη διαφορά χρόνου μεταξύ της ανάγνωσης της μεταβλητής και την αλλαγής της τιμής της. Εάν στο μεσοδιάστημα συμβεί μία διακοπή, ενδέχεται να δημιουργηθεί η παραπάνω κατάσταση.

Αυστηρή εναλλαγή

```
while (TRUE) {
    while (turn != 0)        /* loop */ ;
    critical_region( );
    turn = 1;
    noncritical_region( );
}
```

(a)

```
while (TRUE) {
    while (turn != 1)        /* loop */ ;
    critical_region( );
    turn = 0;
    noncritical_region( );
}
```

(b)

Εικόνα 2-23. Μία λύση στο πρόβλημα των κρίσιμων περιοχών. (a) Διεργασία 0. (b) Διεργασία 1. Και στις δύο περιπτώσεις, προσέξτε το ; στο τέλος της εντολής while.

- Η μεταβλητή turn δείχνει ποια διεργασία έχει σειρά να εισέλθει στην κρίσιμη περιοχή της.
- Η διεργασία 0 μπαίνει όταν η μεταβλητή turn γίνει 0. Μόλις ολοκληρώσει την κρίσιμη περιοχή της, αλλάζει την turn σε 1 για να μπορέσει να μπει και η διεργασία 1 στην κρίσιμη περιοχή της.

- **Πρόβλημα:** Όταν οι διεργασίες δεν έχουν αντίστοιχη διάρκεια εκτέλεσης στην κρίσιμη περιοχή τους, ενδέχεται μία διεργασία που δεν βρίσκεται στην κρίσιμη περιοχή της να εμποδίζει μία άλλη διεργασία να μπει στην κρίσιμη περιοχή της. Π.χ.:
  - Η Δ0 βρίσκει τη μεταβλητή ίση με 0 και μπαίνει στην κρίσιμη περιοχή της. Ολοκληρώνει γρήγορα τη δουλειά της και αλλάζει την turn σε 1.
  - Η Δ1 βρίσκεται στην μη-κρίσιμη περιοχή της για πολύ ώρα (εκτελεί πολλούς υπολογισμούς ή αναμένει Ε/Ε).
  - Η Δ0 ολοκλήρωσε την μη-κρίσιμη περιοχή της και θέλει να μπει στην κρίσιμη περιοχή της. Δυστυχώς δεν μπορεί, εφόσον η turn είναι 1 (***παρόλο που η διεργασία Δ1 δεν βρίσκεται εκείνη τη στιγμή στην κρίσιμη περιοχή της!)***.

Η λύση του Peterson

```
#define FALSE  0
#define TRUE   1
#define N      2                        /* number of processes */

int turn;                               /* whose turn is it? */
int interested[N];                      /* all values initially 0 (FALSE) */

void enter_region(int process);         /* process is 0 or 1 */
{
     int other;                         /* number of the other process */

     other = 1 – process;               /* the opposite of process */
     interested[process] = TRUE;        /* show that you are interested */
     turn = process;                    /* set flag */
     while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process)          /* process: who is leaving */
{
     interested[process] = FALSE;       /* indicate departure from critical region */
}
```

Εικόνα 2-24. Η λύση του Peterson για την επίτευξη αμοιβαίου αποκλεισμού.

- Συνδυάζει την μεταβλητή κλειδώματος με την εναλλαγή της εκτέλεσης των διεργασιών.
- Πριν μπει μία διεργασία στην κρίσιμη περιοχή της καλεί την enter_region.
  - Δηλώνει ότι «ενδιαφέρεται» να μπει στην κρίσιμη περιοχή της (interested[process]=TRUE)
  - «Παίρνει» τη σειρά της (turn = process).
  - Εφόσον είναι η σειρά της αλλά η άλλη διεργασία ενδιαφέρεται, τότε περιμένει.

- Όταν φεύγει από την κρίσιμη περιοχή της, καλεί την leave_region.
  - Δηλώνει ότι πλέον δεν ενδιαφέρεται (ώστε να μπορέσει η άλλη διεργασία να μπει στην κρίσιμη περιοχή της).

Η εντολή TSL (1)

```
enter_region:
        TSL REGISTER,LOCK       | copy lock to register and set lock to 1
        CMP REGISTER,#0         | was lock zero?
        JNE enter_region        | if it was nonzero, lock was set, so loop
        RET                     | return to caller; critical region entered

leave_region:
        MOVE LOCK,#0            | store a 0 in lock
        RET                     | return to caller
```

Εικόνα 2-25. Είσοδος και έξοδος από την κρίσιμη περιοχή με την εντολή TSL.

Η ανάγνωση της λέξης LOCK και η αποθήκευση τιμής σε αυτή είναι ***αδιαίρετη πράξη***.

- Πρέπει να υποστηρίζεται από το υλικό (Εντολή της CPU)
- TSL (Test and Set Lock), δηλαδή έλεγξε και θέσε τη μεταβλητή lock=1.
- Η ανάγνωση της λέξης lock και η αποθήκευση τιμής σε αυτή είναι **αδιαίρετη πράξη!**
- Η CPU η οποία εκτελεί αυτή την εντολή, **κλειδώνει το δίαυλο της μνήμης από άλλες CPU** μέχρι να τελειώσει την εργασία της.

Η εντολή TSL (2)
Χρήση της XCHG

```
enter_region:
        MOVE REGISTER,#1            | put a 1 in the register
        XCHG REGISTER,LOCK          | swap the contents of the register and lock variable
        CMP REGISTER,#0             | was lock zero?
        JNE enter_region            | if it was non zero, lock was set, so loop
        RET                         | return to caller; critical region entered

leave_region:
        MOVE LOCK,#0                | store a 0 in lock
        RET                         | return to caller
```

Εικόνα 2-26. Είσοδος και έξοδος από την κρίσιμη περιοχή με την εντολή XCHG.

- Εναλλακτική εντολή της TSL είναι η XCHG (ανταλλάσει τα περιεχόμενα δύο καταχωρητών).
- Είναι επίσης **αδιαίρετη εντολή**.

Αναμονή με απασχόληση και το πρόβλημα της αντιστροφής προτεραιοτήτων

- Οι διεργασίες X και Y έχουν την ίδια κρίσιμη περιοχή (η Y έχει υψηλή προτεραιότητα και η X χαμηλή προτεραιότητα)
- Η X έχει μπει στην κρίσιμη περιοχή της, αλλά δεν πρόλαβε να καλέσει την leave_region( )
- Γίνεται μία διακοπή (interrupt) και εκτελείται μία τρίτη διεργασία (η Z). Και οι δύο (Y και X) είναι έτοιμες για εκτέλεση (ready).
- Γίνεται μία διακοπή (interrupt) και καλείται η Y (έχει υψηλή προτεραιότητα). Η Y κάνει αναμονή με απασχόληση μέχρι να μπορέσει να μπει στην κρίσιμη περιοχή της.
- Η X όμως δεν μπορεί να χρονοπρογραμματιστεί εφόσον εκτελείται η Y!

- Οι προηγούμενες λύσεις είναι σωστές αλλά υλοποιούν αναμονή με απασχόληση (συνεπώς, σπατάλη της CPU).
- Επιπλέον, αυτό μπορεί να οδηγήσει στο να μην εκτελεστεί μία διεργασία με υψηλή προτεραιότητα (Y), απλά επειδή μία άλλη με χαμηλή προτεραιότητα (X) δεν χρονοπρογραμματίζεται ώστε να μπορέσει να καλέσει την leave_region (πρόβλημα αντιστροφής προτεραιοτήτων) .

Λήθαργος και αφύπνιση (sleep & wakeup)

- Για την ***αποφυγή της αναμονής με απασχόληση***, γίνεται χρήση του ζεύγους κλήσεων συστήματος sleep( ) και wakeup( ).
- sleep( )
  - Μπλοκάρει («κοιμίζει») την καλούσα διεργασία, μέχρι κάποια άλλη να την ξυπνήσει.
- wakeup(process)
  - Ξυπνάει μία διεργασία.

Το πρόβλημα παραγωγού-καταναλωτή (1)

- Γνωστό και ως *το πρόβλημα της περιορισμένης προσωρινής μνήμης (bounded-buffer problem)*.
- Δύο (ή περισσότερες) διεργασίες μοιράζονται μία σταθερού μήκους προσωρινή μνήμη (ένα buffer).
- Η διεργασία - παραγωγός τοποθετεί δεδομένα στον buffer.
  - Εάν ο buffer έχει γεμίσει, ο παραγωγός πρέπει να **κοιμηθεί**, μέχρις ότου αδειάσει τουλάχιστον μία θέση στον buffer.
- Η διεργασία – καταναλωτής αφαιρεί δεδομένα από τον buffer.
  - Εάν ο buffer είναι τελείως άδειος, ο καταναλωτής πρέπει να **κοιμηθεί**, μέχρις ότου γεμίσει τουλάχιστον μία θέση στον buffer.
- Είναι ευθύνη της διεργασίας παραγωγού (καταναλωτή) να **ξυπνήσει** την διεργασία καταναλωτή (παραγωγό).

Το πρόβλημα παραγωγού-καταναλωτή

```
#define N 100                                   /* number of slots in the buffer */
int count = 0;                                  /* number of items in the buffer */

void producer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          item = produce_item( );               /* generate next item */
          if (count == N) sleep( );             /* if buffer is full, go to sleep */
          insert_item(item);                    /* put item in buffer */
          count = count + 1;                    /* increment count of items in buffer */
          if (count == 1) wakeup(consumer);     /* was buffer empty? */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          if (count == 0) sleep( );             /* if buffer is empty, got to sleep */
          item = remove_item( );                /* take item out of buffer */
          count = count – 1;                    /* decrement count of items in buffer */
          if (count == N – 1) wakeup(producer); /* was buffer full? */
          consume_item(item);                   /* print item */
     }
}
```

Εικόνα 2-27. Το πρόβλημα παραγωγού-καταναλωτή με μία μοιραία συνθήκη συναγωνισμού.

- Συνθήκη συναγωνισμού:
  - (Εκτελείται ο καταναλωτής).
    - Ο buffer είναι άδειος (count=0). Ο καταναλωτής διαβάζει το περιεχόμενο της count για να δει αν είναι 0.
  - (Γίνεται interrupt → εκτελείται ο παραγωγός).
    - Ο παραγωγός τοποθετεί ένα στοιχείο στον buffer και αυξάνει την count (count= count+1 = 0+1 = 1).
    - Εφόσον η count ήταν 0 καλεί την weakup( ) για να ξυπνήσει τον καταναλωτή. Όμως ***ο καταναλωτής δεν βρίσκεται ακόμα σε λήθαργο. (δεν πρόλαβε να μπει σε λήθαργο λόγω του interrupt). Το σήμα wakeup χάνεται!***
  - (Γίνεται interrupt → εκτελείται ο καταναλωτής).
    - Ο καταναλωτής είχε βρει την τιμή της count=0. Άρα καλεί την sleep(). ***Και οι δύο διεργασίες είναι σε λήθαργο, λόγω του σήματος weakup που χάθηκε!***

Σηματοφόροι ή σημαφόροι (Semaphores)

- *Νέος τύπος μεταβλητής*
  - 0: δεν έχουν αποθηκευτεί σήματα αφύπνισης.
  - x>0: εκκρεμούν x σήματα αφύπνισης.
- Λειτουργία **down**:
  - Ελέγχει την τιμή του σημαφόρου.
    - Εάν >0, τον μειώνει κατά 1 και συνεχίζει.
    - Εάν =0 τότε η καλούσα διεργασία κοιμάται χωρίς να κάνει την μείωση κατά 1 (*η μείωση εκκρεμεί*).
  - Ο έλεγχος, η αλλαγή τιμής και η πιθανή κλήση της sleep είναι μία ***αδιαίρετη ενέργεια (atomic action)***.

- Ο σημαφόρος λειτουργεί ως «κουμπαράς» των σημάτων που σε άλλη περίπτωση θα χάνονταν…

Σηματοφόροι ή σημαφόροι (Semaphores)

- Λειτουργία **up**:
  - Αυξάνει την τιμή του σημαφόρου κατά 1.
  - Εάν κάποια λειτουργία down δεν είχε ολοκληρωθεί (*εκκρεμεί από πριν μία λειτουργία down*) τότε ολοκληρώνεται αυτή η λειτουργία (μειώνει κατά 1).
  - Μετά από μία λειτουργία up σε ένα σημαφόρο που έχει μπλοκάρει κάποιες διεργασίες, θα εξακολουθήσει να έχει την τιμή 0 αλλά *οι διεργασίες που βρίσκονται σε λήθαργο* ***θα έχουν μειωθεί κατά μία****!*
  - Οι λειτουργίες αύξησης του σημαφόρου και αφύπνισης της διεργασίας είναι επίσης **αδιαίρετες**.

- Οι λειτουργίες up και down υλοποιούνται ως κλήσεις συστήματος και το ΛΣ απενεργοποιεί τις διακοπές μέχρι να εκτελεστούν.
  - Η απενεργοποίηση των διακοπών γίνεται **μόνο για λίγα μsec**. (σε αντίθεση με τις εντολές TSL και XCHG οι οποίες κλειδώνουν το δίαυλο)

Σηματοφόροι (Semaphores)

```
#define N 100                                   /* number of slots in the buffer */
typedef int semaphore;                          /* semaphores are a special kind of int */
semaphore mutex = 1;                            /* controls access to critical region */
semaphore empty = N;                            /* counts empty buffer slots */
semaphore full = 0;                             /* counts full buffer slots */

void producer(void)
{
     int item;

     while (TRUE) {                             /* TRUE is the constant 1 */
          item = produce_item( );                /* generate something to put in buffer */
          down(&empty);                          /* decrement empty count */
          down(&mutex);                          /* enter critical region */
          insert_item(item);                     /* put new item in buffer */
          up(&mutex);                            /* leave critical region */
          up(&full);                             /* increment count of full slots */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                             /* infinite loop */
          down(&full);                           /* decrement full count */
          down(&mutex);                          /* enter critical region */
          item = remove_item( );                 /* take item from buffer */
          up(&mutex);                            /* leave critical region */
          up(&empty);                            /* increment count of empty slots */
          consume_item(item);                    /* do something with the item */
     }
}
```

Εικόνα 2-28. Επίλυση του προβλήματος παραγωγού-καταναλωτή με τη χρήση σηματοφόρων.

- Δύο χρήσεις σημαφόρων:
  - Ο mutex εξασφαλίζει τον αμοιβαίο αποκλεισμό.
  - Ο empty εγγυάται ο παραγωγός μπλοκάρεται όταν ο buffer είναι άδειος, ενώ ο full εγγυάται ο καταναλωτής μπλοκάρεται όταν ο buffer είναι γεμάτος.

Τα Mutex

```
mutex_lock:
        TSL REGISTER,MUTEX      | copy mutex to register and set mutex to 1
        CMP REGISTER,#0         | was mutex zero?
        JZE ok                  | if it was zero, mutex was unlocked, so return
        CALL thread_yield       | mutex is busy; schedule another thread
        JMP mutex_lock          | try again
ok:     RET                     | return to caller; critical region entered

mutex_unlock:
        MOVE MUTEX,#0           | store a 0 in mutex
        RET                     | return to caller
```

Εικόνα 2-29. Υλοποίηση των διαδικασιών *mutex_lock* and *mutex_unlock*.

- Κατάλληλα για την περίπτωση που η καταμέτρηση των σημαφόρων δεν χρειάζεται (αλλά μόνο ο αμοιβαίος αποκλεισμός).

Χρήση mutex σε νήματα

- Όταν τα νήματα υλοποιούνται στο χώρο του χρήστη, τα mutex μπορούν να χρησιμοποιηθούν για αμοιβαίο αποκλεισμό των νημάτων.
- Κλήσεις συστήματος για τη δημιουργία, έλεγχο και καταστροφή των mutex σε pthreads
- Κλήσεις συστήματος για τη δημιουργία, έλεγχο και καταστροφή μεταβλητών συνθήκης (condition variables)
  - Οι μεταβλητές συνθήκης επιτρέπουν ή απαγορεύουν την πρόσβαση στην κρίσιμη περιοχή ανάλογα με κάποια συνθήκη (αν δεν ικανοποιείται η συνθήκη, δεν επιτρέπεται η πρόσβαση στο critical region).

Χρήση των Mutex σε νήματα Pthreads (1)

Thread call	Description
Pthread_mutex_init	Create a mutex
Pthread_mutex_destroy	Destroy an existing mutex
Pthread_mutex_lock	Acquire a lock or block
Pthread_mutex_trylock	Acquire a lock or fail
Pthread_mutex_unlock	Release a lock

Εικόνα 2-30. Μερικές από τις κλήσεις Pthreads που σχετίζονται με τα mutex.

- Συγχρονισμός και αμοιβαίος αποκλεισμός σε νήματα επιπέδου χρήστη (user space).

- Η κλήση mutex_trylock είτε αποκτά τον έλεγχο του κλειδώματος, είτε επιστρέφει κάποιο σφάλμα (όμως δεν μπλοκάρει το νήμα!)
    - Συνεπώς επιτρέπει στο πρόγραμμα να εξετάσει πιθανές εναλλακτικές (π.χ το νήμα να μπει σε αναμονή με απασχόληση ή να μπλοκάρει)

Χρήση των Mutex σε νήματα Pthreads (2)

Thread call	Description
Pthread_cond_init	Create a condition variable
Pthread_cond_destroy	Destroy a condition variable
Pthread_cond_wait	Block waiting for a signal
Pthread_cond_signal	Signal another thread and wake it up
Pthread_cond_broadcast	Signal multiple threads and wake all of them

Εικόνα 2-31. Μερικές από τις κλήσεις Pthreads που σχετίζονται με μεταβλητές συνθήκης.

- Οι μεταβλητές συνθήκης επιτρέπουν ή απαγορεύουν την πρόσβαση στην κρίσιμη περιοχή ανάλογα με κάποια συνθήκη (αν δεν ικανοποιείται η συνθήκη, δεν επιτρέπεται η πρόσβαση στο critical region).
- Οι μεταβλητές συνθήκης (condition variables), σε αντίθεση με τα mutex **δεν έχουν μνήμη** (αν σταλεί ένα σήμα σε μία condition variable για την οποία δεν περιμένει κανένα νήμα, τότε το σήμα θα χαθεί!)
    - Είναι ευθύνη των προγραμματιστών να μην χάνονται σήματα.

Χρήση των Mutex σε νήματα Pthreads (3)

```
#include <stdio.h>
#include <pthread.h>
#define MAX 1000000000                  /* how many numbers to produce */
pthread_mutex_t the_mutex;
pthread_cond_t condc, condp;
int buffer = 0;                         /* buffer used between producer and consumer */

void *producer(void *ptr)               /* produce data */
{   int i;

    for (i= 1; i <= MAX; i++) {
        pthread_mutex_lock(&the_mutex);   /* get exclusive access to buffer */
        while (buffer != 0) pthread_cond_wait(&condp, &the_mutex);
        buffer = i;                       /* put item in buffer */
        pthread_cond_signal(&condc);      /* wake up consumer */
        pthread_mutex_unlock(&the_mutex);/* release access to buffer */
    }
    pthread_exit(0);
}

void *consumer(void *ptr)               /* consume data */
{   int i;

    for (i = 1; i <= MAX; i++) {
        pthread_mutex_lock(&the_mutex);   /* get exclusive access to buffer */
        while (buffer ==0 ) pthread_cond_wait(&condc, &the_mutex);
        buffer = 0;                       /* take item out of buffer */
        pthread_cond_signal(&condp);      /* wake up producer */
        pthread_mutex_unlock(&the_mutex);/* release access to buffer */
    }
    pthread_exit(0);
}

int main(int argc, char **argv)
{
    pthread_t pro, con;
    pthread_mutex_init(&the_mutex, 0);
    pthread_cond_init(&condc, 0);
    pthread_cond_init(&condp, 0);
    pthread_create(&con, 0, consumer, 0);
    pthread_create(&pro, 0, producer, 0);
    pthread_join(pro, 0);
    pthread_join(con, 0);
    pthread_cond_destroy(&condc);
    pthread_cond_destroy(&condp);
    pthread_mutex_destroy(&the_mutex);
}
```

Εικόνα 2-32. Χρήση νημάτων για την επίλυση του προβλήματος παραγωγού-καταναλωτή.

Ελεγκτές (Monitors) (1)

- Οι σηματοφόροι και τα mutex λειτουργούν σε χαμηλό επίπεδο
- Οι ελεγκτές (monitors) είναι μία λύση για συγχρονισμό μεταξύ διεργασιών και αποφυγή συνθηκών συναγωνισμού, σε υψηλότερο επίπεδο.
  - Αφορά γλώσσες προγραμματισμού υψηλότερου επιπέδου, όπως η java.
- Μέσα σε ένα ελεγκτή (monitor) περιλαμβάνονται πολλές διεργασίες. Κάθε φορά όμως μπορεί να είναι ενεργή μόνο μία διαδικασία!
- Χρησιμοποιούν μεταβλητές συνθήκης και κλήσεις wait και signal για μπλοκάρισμα και αφύπνιση.

Ελεγκτές (Monitors) (2)

```
monitor example
        integer i;
        condition c;

        procedure producer( );
        .
        .
        .
        end;

        procedure consumer( );
        .           .           .
        end;
end monitor;
```

Εικόνα 2-33. Παράδειγμα ελεγκτή.

Ελεγκτές (Monitors) (3)

```
monitor ProducerConsumer
        condition full, empty;
        integer count;

        procedure insert(item: integer);
        begin
                if count = N then wait(full);
                insert_item(item);
                count := count + 1;
                if count = 1 then signal(empty)
        end;

        function remove: integer;
        begin
                if count = 0 then wait(empty);
                remove = remove_item;
                count := count - 1;
                if count = N - 1 then signal(full)
        end;

        count := 0;
end monitor;

procedure producer;
begin
        while true do
        begin
                item = produce_item;
                ProducerConsumer.insert(item)
        end
end;

procedure consumer;
begin
        while true do
        begin
                item = ProducerConsumer.remove;
                consume_item(item)
        end
end;
```

Εικόνα 2-34. Σε κάθε χρονική στιγμή χρησιμοποιείται μία μόνο διαδικασία του ελεγκτή.

- Ο μεταγλωττιστής της γλώσσας θα αναλάβει τον αμοιβαίο αποκλεισμό και το συγχρονισμό των διεργασιών σε χαμηλό επίπεδο, χρησιμοποιώντας κάποια από τις τεχνικές που αναφέρθηκαν προηγουμένως.
- Τώρα όμως ο προγραμματιστής δεν ασχολείται με τις λεπτομέρειες υλοποίησης της ΔΔΕ. Απλά χρησιμοποιεί τους ελεγκτές.
- Τα προγραμματιστικά σφάλματα που θα μπορούσαν να οδηγήσουν σε αδιέξοδα (π.χ. να βρεθούν όλες οι διεργασίες σε κατάσταση sleep) ελαχιστοποιούνται.

Δημιουργία ελεγκτών σε Java (1)

```
public class ProducerConsumer {
    static final int N = 100;       // constant giving the buffer size
    static producer p = new producer( );    // instantiate a new producer thread
    static consumer c = new consumer( ); // instantiate a new consumer thread
    static our_monitor mon = new our_monitor( );      // instantiate a new monitor

    public static void main(String args[ ]) {
       p.start( );       // start the producer thread
       c.start( );       // start the consumer thread
    }

    static class producer extends Thread {
       public void run( )  {// run method contains the thread code
          int item;
          while (true) {      // producer loop
             item = produce_item( );
             mon.insert(item);
          }
       }
       private int produce_item( ) { ... }      // actually produce
    }


     . . .
```

Εικόνα 2-35. Μία λύση του προβλήματος Π-Κ σε Java με τη χρήση synchronized διαδικασιών.

- Η χρήση της λέξης synchronized κατά τη δήλωση μιας μεθόδου εξασφαλίζει ότι εάν ένα νήμα της διεργασίας ξεκινήσει μία synchronized μέθοδο, δεν θα επιτραπεί σε κάποιο άλλο νήμα να εκκινήσει οποιαδήποτε μέθοδο έχει δηλωθεί ως synchronized, μέχρι να ολοκληρώσει το 1ο νήμα τη synchronized μέθοδό του.

Δημιουργία ελεγκτών σε Java (2)

. . .

```
static class consumer extends Thread {
   public void run( ) {run method contains the thread code
      int item;
      while (true) {      // consumer loop
         item = mon.remove( );
         consume_item (item);
      }
   }
   private void consume_item(int item) { ... }// actually consume
}

static class our_monitor {  // this is a monitor
   private int buffer[ ] = new int[N];
   private int count = 0, lo = 0, hi = 0;  // counters and indices

   public synchronized void insert(int val) {
      if (count == N) go_to_sleep( );     // if the buffer is full, go to sleep
      buffer [hi] = val; // insert an item into the buffer
      hi = (hi + 1) % N;        // slot to place next item in
      count = count + 1;        // one more item in the buffer now
      if (count == 1) notify( );        // if consumer was sleeping, wake it up
   }
```

. . .

Εικόνα 2-35. Μία λύση του προβλήματος Π-Κ σε Java με τη χρήση synchronized διαδικασιών

Δημιουργία ελεγκτών σε Java (3)

```
. . .

        public synchronized int remove( ) {
            int val;
            if (count == 0) go_to_sleep( );     // if the buffer is empty, go to sleep
            val = buffer [lo]; // fetch an item from the buffer
            lo = (lo + 1) % N;        // slot to fetch next item from
            count = count – 1;        // one few items in the buffer
            if (count == N – 1) notify( ); // if producer was sleeping, wake it up
            return val;
        }
        private void go_to_sleep( ) { try{wait( );} catch(InterruptedException exc) {};
    }
}
```

Εικόνα 2-35. Μία λύση του προβλήματος Π-Κ σε Java με τη χρήση synchronized διαδικασιών

Μεταβίβαση μηνυμάτων (1)

- Σε ένα κατανεμημένο σύστημα, δεν είναι δυνατό να χρησιμοποιηθούν προγραμματιστικές τεχνικές χαμηλού επιπέδου.
- Χρησιμοποιείται η μεταβίβαση μηνυμάτων (message passing)
  - Κλήσεις send(destination, &message) και
  - receive(source, &message).
  - Για το συγχρονισμό απαιτούνται μηνύματα επιβεβαίωσης (acknowledgement messages)

Το πρόβλημα Π-Κ με μεταβίβαση μηνυμάτων (1)

```
#define N 100                                   /* number of slots in the buffer */

void producer(void)
{
     int item;
     message m;                                 /* message buffer */

     while (TRUE) {
          item = produce_item( );               /* generate something to put in buffer */
          receive(consumer, &m);                /* wait for an empty to arrive */
          build_message(&m, item);              /* construct a message to send */
          send(consumer, &m);                   /* send item to consumer */
     }
}

  . . .
```

Εικόνα 2-36. Το πρόβλημα Π-Κ με Ν μηνύματα.

- Αρχικά ο καταναλωτής παράγει Ν κενά μηνύματα και τα στέλνει στον παραγωγό.
- Ο παραγωγός κοιτάζει αν έχει κενό μήνυμα, το γεμίζει και το στέλνει στον καταναλωτή.
- Ο καταναλωτής κοιτάζει αν έλαβε γεμάτο μήνυμα, το καταναλώνει και απελευθερώνει ένα για τον παραγωγό.

Το πρόβλημα Π-Κ με μεταβίβαση μηνυμάτων (2)

. . .

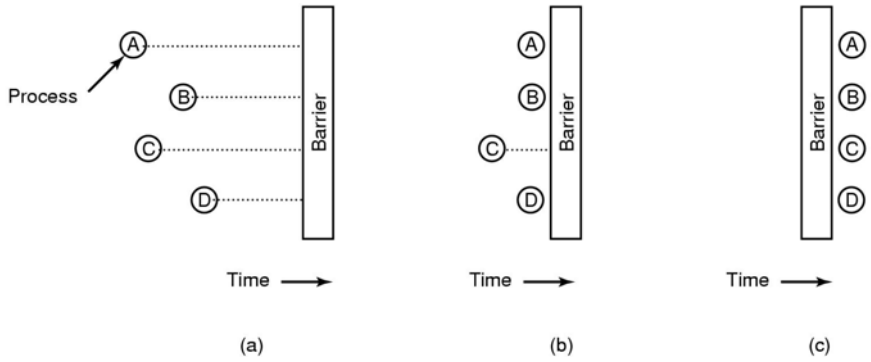
```
void consumer(void)
{
     int item, i;
     message m;

     for (i = 0; i < N; i++) send(producer, &m);  /* send N empties */
     while (TRUE) {
          receive(producer, &m);          /* get message containing item */
          item = extract_item(&m);        /* extract item from message */
          send(producer, &m);             /* send back empty reply */
          consume_item(item);             /* do something with the item */
     }
}
```

Εικόνα 2-36. Το πρόβλημα Π-Κ με Ν μηνύματα.

Φράγματα (Barriers) (1)

- Μέθοδος συγχρονισμού που αφορά ομάδες διεργασιών.
- Ένα φράγμα σταματά όσες διεργασίες της ομάδας έχουν ολοκληρώσει κάποια φάση.
- Κάθε διεργασία που φτάνει στο φράγμα, εκτελεί την εργασία της και καλεί την κλήση barrier για να μπλοκαριστεί.
- Όταν όλες οι διεργασίες της ομάδας έχουν καλέσει την barrier, τότε μπορούν να συνεχίσουν.

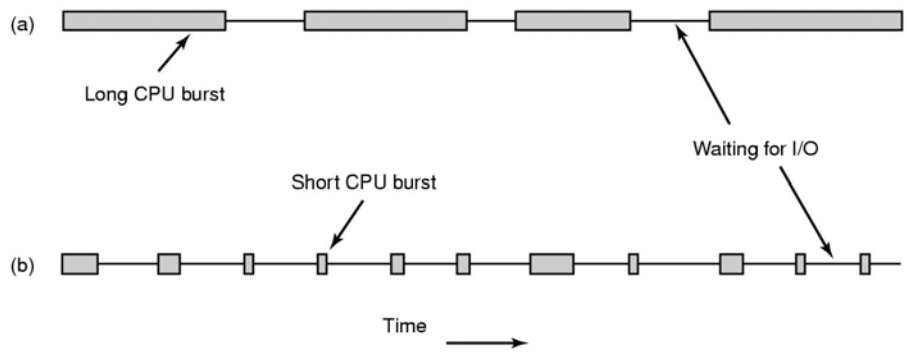
Φράγματα (Barriers)



Εικόνα 2-37. Χρήση φράγματος. (a) Οι διεργασίες πλησιάζουν το φράγμα. (b) Όλες οι διεργασίες εκτός από μία έχουν μπλοκαριστεί στο φράγμα. (c) Όταν φτάσει και η τελευταία στο φράγμα, επιτρέπεται πλέον σε όλες να περάσουν.

2.4. ΧΡΟΝΟΠΡΟΓΡΑΜΜΑΤΙΣΜΟΣ (SCHEDULING)

Η χρονοπρογραμματιστική συμπεριφορά των διεργασιών



Εικόνα 2-38. Τα διαστήματα χρήσης της CPU εναλλάσσονται με περιόδους αναμονής για Ε/Ε. (a) Μια διεργασία εξαρτημένη από τη CPU. (b) Μία διεργασία εξαρτημένη από την Ε/Ε.

Πότε γίνεται χρονοπρογραμματισμός

1. Όταν μία διεργασία που εκτελείται τερματίσει.
2. Όταν περισσότερες από μία διεργασίες είναι έτοιμες για εκτέλεση (ready).
3. Όταν μία διεργασία που εκτελείται μπλοκάρεται από Ε/Ε, σημαφόρο κτλ.
4. Όταν συμβαίνει μία διακοπή Ε/Ε (interrupt).
5. Το ρολόι του συστήματος προκαλεί περιοδικές διακοπές (π.χ. στα 50Hz). Σε κάθε διακοπή λαμβάνεται μία απόφαση χρονοπρογραμματισμού.

Προεκτοπιστικοί και μη-προεκτοπιστικοί αλγόριθμοι χρονοπρογραμματισμού

- *Μη προεκτοπιστικοί αλγόριθμοι* (non-preemtive)
  - Σε κάθε διακοπή ρολογιού επιλέγουν μία διεργασία η οποία θα εκτελεστεί μέχρι να επιστρέψει εθελοντικά τη CPU ή μέχρι να μπλοκαριστεί για Ε/Ε.
  - Δεν λαμβάνονται αποφάσεις χρονοπρογραμματισμού στις διακοπές ρολογιού.
- *Προεκτοπιστικοί αλγόριθμοι* (preemtive)
  - Σε κάθε διακοπή ρολογιού επιλέγουν μία διεργασία η οποία θα εκτελεστεί για ένα μέγιστο χρονικό διάστημα.
  - Εάν δεν έχει ολοκληρωθεί σε αυτό το χρονικό διάστημα, αναστέλλεται και θα συνεχίσει κάποια άλλη στιγμή.

Κατηγορίες αλγορίθμων χρονοπρογραμματισμού

- (Α) Αλγόριθμοι για περιβάλλοντα δέσμης (batch)
- (Β) Αλγόριθμοι για αλληλεπιδραστικά περιβάλλοντα (interactive)
- (Γ) Αλγόριθμοι για περιβάλλοντα πραγματικού χρόνου (real-time)

Στόχοι αλγόριθμων χρονοπρογραμματισμού

Για όλα τα συστήματα

- **Δικαιοσύνη:** Να εκχωρείται σε κάθε διεργασία ένα μερίδιο της CPU.
- **Επιβολή της πολιτικής:** Να παρακολουθείται εάν εφαρμόζεται η καθορισμένη πολιτική.
- **Ισορροπία:** Να διατηρούνται ενεργά όλα τα τμήματα του συστήματος.

Για τα συστήματα δέσμης

- **Διεκπεραιωτική ικανότητα:** Μεγιστοποίηση των εργασιών που ολοκληρώνονται ανά ώρα.
- **Χρόνος διεκπεραίωσης:** Ελαχιστοποίηση του χρόνου μεταξύ υποβολής και ολοκλήρωσης μιας διεργασίας.
- **Αξιοποίηση της CPU:** Συνεχής απασχόληση της CPU.

Για τα αλληλεπιδραστικά συστήματα

- **Χρόνος απόκρισης:** Ταχύτατη απόκριση στις αιτήσεις.
- **Τήρηση αναλογικότητας:** Ικανοποίηση των λογικών προσδοκιών των χρηστών.

Για τα συστήματα πραγματικού χρόνου

- **Τήρηση προθεσμιών:** Αποφυγή απώλειας δεδομένων.
- **Προβλεψιμότητα:** Αποφυγή υποβιβασμού ποιότητας πολυμέσων.

Εικόνα 2-39. Μερικοί στόχοι των αλγορίθμων χρονοπρογραμματισμού για διαφορετικές συνθήκες.

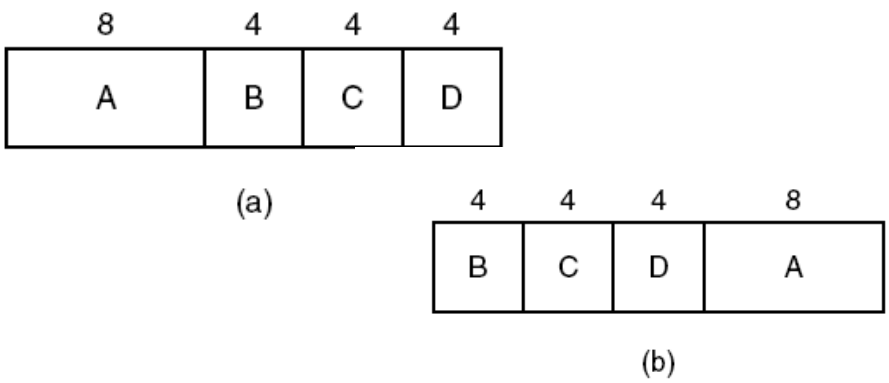
(Α) Χρονοπρογραμματισμός σε συστήματα δέσμης

- **Με βάση τη σειρά άφιξης** (First-come first-served)
- **Με βάση τη μικρότερη διάρκεια** (Shortest job first)
- **Με βάση τη μικρότερη υπολειπόμενη διάρκεια** (Shortest remaining Time next)

(Α) Χρονοπρογραμματισμός σε συστήματα δέσμης
(Με βάση τη σειρά άφιξης)

- Η εξυπηρέτηση με βάση τη σειρά άφιξης δημιουργεί μία μοναδική ουρά FIFO (First-In-First-Out).
- Οι νέες διεργασίες τοποθετούνται στο τέλος της λίστας.
- Εάν η εκτελούμενη εργασία μπλοκαριστεί, πηγαίνει στο τέλος της ουράς και ο αλγόριθμος συνεχίζει.
- Πολύ απλός αλγόριθμος. Η υλοποίηση μπορεί να γίνει με μία **συνδεδεμένη λίστα**.
- *Μειονέκτημα αλγορίθμου:*
  - Όταν υπάρχουν λίγες διεργασίες εξαρτημένες από τη CPU και πολλές εξαρτημένες από Ε/Ε γίνεται σπατάλη της CPU.

(Α) Χρονοπρογραμματισμός σε συστήματα δέσμης
(Εξυπηρέτηση με βάση τη διάρκεια)



Εικόνα 2-40. Παράδειγμα χρονοπρογραμματισμού με βάση τη μικρότερη διάρκεια (Shortest Job First).
(a) Εκτέλεση 4 εργασιών με βάση την αρχική σειρά.
(b) Εκτέλεση με βάση τη μικρότερη διάρκεια.

- Στη γενική περίπτωση, έστω διεργασίες Α, Β, Γ και Δ με χρόνους εκτέλεσης α, β, γ, και δ αντίστοιχα.
  - Η Α θα ολοκληρωθεί σε α sec, η Β σε (α+β) sec, η Γ σε (α+β+γ) sec, η Δ σε (α+β+γ+δ) sec.
  - Ο μέσος χρόνος διεκπεραίωσης είναι (4α+3β+2γ+δ)/4

1. Στην (a) περίπτωση έχουμε:
   - Χρόνο διεκπεραίωσης για τις διεργασίες Α, Β, Γ, Δ, 8, 12, 16 και 20 sec αντίστοιχα.
   - Ο μέσος χρόνος εκτέλεσης είναι 14 sec
2. Στη (b) περίπτωση έχουμε:
   - Χρόνο διεκπεραίωσης για τις διεργασίες Α, Β, Γ, Δ, 4, 8, 12 και 20 sec αντίστοιχα.
   - Ο μέσος χρόνος εκτέλεσης είναι 11 sec

(Β) Χρονοπρογραμματισμός σε αλληλεπιδραστικά συστήματα

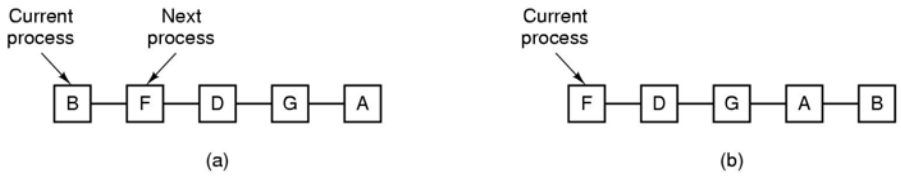
- **Χρονοπρογραμματισμός εκ περιτροπής** (Round-robin scheduling)
- **Χρονοπρογραμματισμός με βάση την προτεραιότητα** (Priority scheduling)
- **Πολλαπλές ουρές** (Multiple queues)
- **Εξυπηρέτηση με βάση τη μικρότερη διάρκεια** (Shortest process next)
- **Εγγυημένος χρονοπρογραμματισμός** (Guaranteed scheduling)
- **Χρονοπρογραμματισμός με λοταρία** (Lottery scheduling)
- **Χρονοπρογραμματισμός δίκαιης διανομής** (Fair-share scheduling)

(Β) Σε αλληλεπιδραστικά συστήματα ***(Round-Robin Scheduling)*** (1/2)

- Από τους παλαιότερους και δικαιότερους αλγορίθμους.
- Σε κάθε διεργασία εκχωρείται ένα ***κβάντο χρόνου*** (quantum time).
  - Εάν δεν έχει ολοκληρωθεί, μπλοκάρεται και πηγαίνει στο τέλος της λίστας.
- Ζητήματα:
  - *Πόσο διαρκεί το κβάντο χρόνου;*
  - *Πόσο διαρκεί η εναλλαγή μεταξύ των διεργασιών (process switch);*

- Εάν το κβάντο χρόνου είναι σχετικά μικρό σε σχέση με το χρόνο εναλλαγής διεργασιών, χάνεται πολύς χρόνος για την εναλλαγή, άρα μειώνεται η αποδοτικότητα της CPU.
- Εάν το κβάντο είναι πολύ μεγάλο σε σχέση με το χρόνο εναλλαγής διεργασιών, θα αυξηθεί ο χρόνος αναμονής των χρηστών και ο χρόνος διεκπεραίωσης.

(Β) Σε αλληλεπιδραστικά συστήματα ***(Round-Robin Scheduling)*** (1/2)



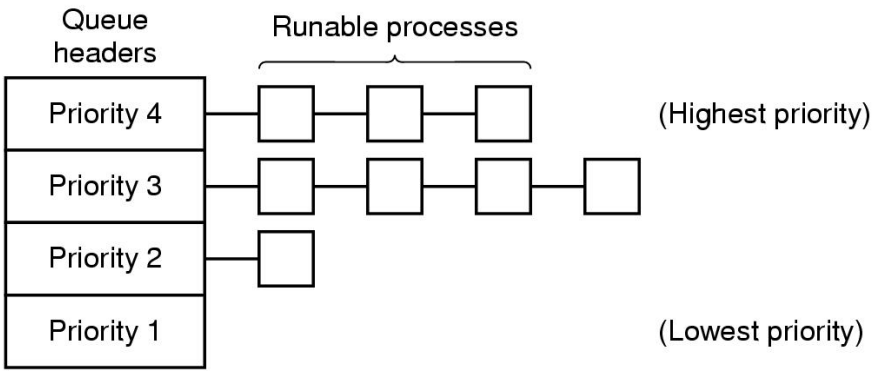
Εικόνα 2-41. Χρονοπρογραμματισμός εκ περιτροπής . (a) Η λίστα των εκτελέσιμων διαδικασιών. (b) Η λίστα των εκτελέσιμων διαδικασιών μετά από τη χρήση του κβάντου που αντιστοιχεί στη διεργασία Β.

(Β) Σε αλληλεπιδραστικά συστήματα
(Με βάση την προτεραιότητα) (1/2)

- Σε κάθε διεργασία αντιστοιχίζεται μία προτεραιότητα.
- Κάθε στιγμή εκτελείται μία διαθέσιμη (runable) διεργασία με την υψηλότερη προτεραιότητα.
- Η προτεραιότητα των διεργασιών μπορεί να μειώνεται μετά από κάθε διακοπή ρολογιού, ώστε να μπορέσουν να εκτελεστούν και οι υπόλοιπες.
- Εναλλακτικά, κάθε διεργασία μπορεί να έχει ένα μέγιστο κβάντο χρόνου.
- Οι προτεραιότητες μπορεί να ανατεθούν στατικά η δυναμικά.
  - Εντολή *nice* στο Unix.

- Απλός αλγόριθμος δυναμικής προτεραιότητας:
  - Αναθέτει σε κάθε διεργασία προτεραιότητα = 1/f όπου f είναι το ποσοστό του τελευταίου κβάντου που χρησιμοποίησε η διεργασία.

(Β) Σε αλληλεπιδραστικά συστήματα
(Με βάση την προτεραιότητα) (2/2)



Εικόνα 2-42. Ένας αλγόριθμος χρονοπρογραμματισμού με τέσσερις κλάσεις προτεραιοτήτων.

(Γ) Χρονοπρογραμματισμός σε συστήματα πραγματικού χρόνου

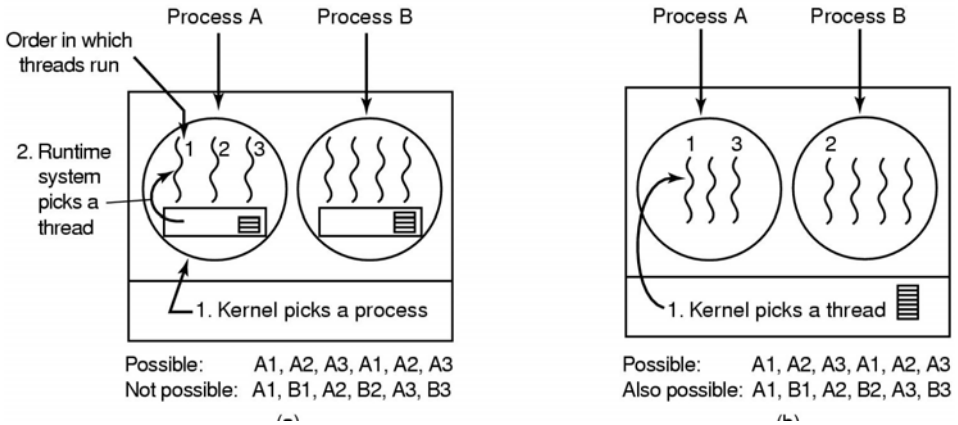
- Χρονοπρογραμματίσιμα συστήματα πραγματικού χρόνου (Schedulable real-time system)
- Έστω ότι ισχύει:
  - Συμβαίνουν m περιοδικά γεγονότα
  - Το γεγονός i συμβαίνει με περίοδο P_i και απαιτεί C_i δευτερόλεπτα
- Τότε το σύστημα είναι δυνατό να χειριστεί το συνολικό φορτίο εργασίας εάν ισχύει:

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

Πολιτική και μηχανισμός

- Διαχωρισμός μεταξύ τι επιτρέπεται να γίνει και πως γίνεται
  - Μία διεργασία γνωρίζει ποιες θυγατρικές διεργασίες της είναι σημαντικές και χρειάζονται προτεραιότητα
- Ο αλγόριθμος χρονοπρογραμματισμού είναι παραμετροποιήσιμος
  - Ο μηχανισμός βρίσκεται στον πυρήνα
- Οι παράμετροι συμπληρώνονται από τις διεργασίες επιπέδου χρήστη
  - Η πολιτική τίθεται από τις διεργασίες χρήστη

Χρονοπρογραμματισμός νημάτων

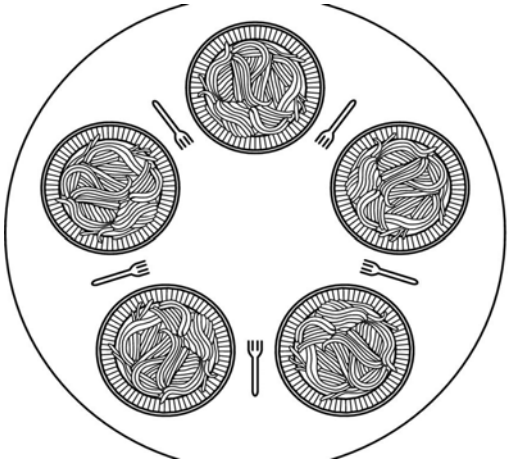


Εικόνα 2-43. (a) Πιθανός χρονοπρογραμματισμός νημάτων επιπέδου χρήστη και (b) επιπέδου πυρήνα

Κβάντο χρόνου διεργασιών 50 msec και νήματος 5 msec, κάθε φορά που τους εκχωρείται η CPU.

2.5. ΚΛΑΣΙΚΑ ΠΡΟΒΛΗΜΑΤΑ ΔΙΑΔΙΕΡΓΑΣΙΑΚΗΣ ΕΠΙΚΟΙΝΩΝΙΑΣ

Το πρόβλημα του δείπνου των φιλοσόφων (1)



Εικόνα 2-44. Ώρα φαγητού στο Τμήμα Φιλοσοφίας.

Το πρόβλημα του δείπνου των φιλοσόφων (2)

```
#define N 5                                     /* number of philosophers */

void philosopher(int i)                         /* i: philosopher number, from 0 to 4 */
{
    while (TRUE) {
        think( );                               /* philosopher is thinking */
        take_fork(i);                           /* take left fork */
        take_fork((i+1) % N);                   /* take right fork; % is modulo operator */
        eat( );                                 /* yum-yum, spaghetti */
        put_fork(i);                            /* put left fork back on the table */
        put_fork((i+1) % N);                    /* put right fork back on the table */
    }
}
```

Εικόνα 2-45. Μία μη λύση στο πρόβλημα του δείπνου των φιλοσόφων.

Το πρόβλημα του δείπνου των φιλοσόφων (3)

```
#define N              5                    /* number of philosophers */
#define LEFT           (i+N–1)%N            /* number of i's left neighbor */
#define RIGHT          (i+1)%N              /* number of i's right neighbor */
#define THINKING       0                    /* philosopher is thinking */
#define HUNGRY         1                    /* philosopher is trying to get forks */
#define EATING         2                    /* philosopher is eating */
typedef int semaphore;                      /* semaphores are a special kind of int */
int state[N];                               /* array to keep track of everyone's state */
semaphore mutex = 1;                        /* mutual exclusion for critical regions */
semaphore s[N];                             /* one semaphore per philosopher */

void philosopher(int i)                     /* i: philosopher number, from 0 to N–1 */
{
     while (TRUE) {                         /* repeat forever */
          think( );                         /* philosopher is thinking */
          take_forks(i);                    /* acquire two forks or block */
          eat( );                           /* yum-yum, spaghetti */
          put_forks(i);                     /* put both forks back on table */
     }
}
  . . .
```

Εικόνα 2-46. Μία ικανοποιητική λύση στο πρόβλημα.

Το πρόβλημα του δείπνου των φιλοσόφων (4)

```
. . .
void take_forks(int i)                  /* i: philosopher number, from 0 to N–1 */
{
        down(&mutex);                   /* enter critical region */
        state[i] = HUNGRY;              /* record fact that philosopher i is hungry */
        test(i);                        /* try to acquire 2 forks */
        up(&mutex);                     /* exit critical region */
        down(&s[i]);                    /* block if forks were not acquired */
}

. . .
```

Εικόνα 2-46. Μία ικανοποιητική λύση στο πρόβλημα.

Το πρόβλημα του δείπνου των φιλοσόφων (5)

```
. . .
void put_forks(i)                          /* i: philosopher number, from 0 to N–1 */
{
        down(&mutex);                      /* enter critical region */
        state[i] = THINKING;               /* philosopher has finished eating */
        test(LEFT);                        /* see if left neighbor can now eat */
        test(RIGHT);                       /* see if right neighbor can now eat */
        up(&mutex);                        /* exit critical region */
}

void test(i) /* i: philosopher number, from 0 to N–1 */
{
        if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
                state[i] = EATING;
                up(&s[i]);
        }
}
```

Εικόνα 2-46. Μία ικανοποιητική λύση στο πρόβλημα.

Το πρόβλημα των αναγνωστών - γραφέων (1)

```
typedef int semaphore;                  /* use your imagination */
semaphore mutex = 1;                    /* controls access to 'rc' */
semaphore db = 1;                       /* controls access to the database */
int rc = 0;                             /* # of processes reading or wanting to */

void reader(void)
{
    while (TRUE) {                      /* repeat forever */
        down(&mutex);                   /* get exclusive access to 'rc' */
        rc = rc + 1;                    /* one reader more now */
        if (rc == 1) down(&db);         /* if this is the first reader ... */
        up(&mutex);                     /* release exclusive access to 'rc' */
        read_data_base( );              /* access the data */
        down(&mutex);                   /* get exclusive access to 'rc' */
        rc = rc – 1;                    /* one reader fewer now */
        if (rc == 0) up(&db);           /* if this is the last reader ... */
        up(&mutex);                     /* release exclusive access to 'rc' */
        use_data_read( );               /* noncritical region */
    }
}
```

. . .

Εικόνα 2-47. Μία λύση στο πρόβλημα αναγνωστών - γραφέων.

Το πρόβλημα των αναγνωστών - γραφέων (2)

. . .

```
void writer(void)
{
     while (TRUE) {                  /* repeat forever */
          think_up_data( );          /* noncritical region */
          down(&db);                 /* get exclusive access */
          write_data_base( );        /* update the data */
          up(&db);                   /* release exclusive access */
     }
}
```

Εικόνα 2-47. Μία λύση στο πρόβλημα αναγνωστών - γραφέων.