ΤΑΥΤΟΧΡΟΝΕΣ ΔΙΕΡΓΑΣΙΕΣ

Σε πολλές περιπτώσεις έχουμε ταυτόχρονη (παράλληλη) εκτέλεση πολλαπλών διεργασιών (συστήματα πολυπρογραμματισμού – πολυδιεργασικά συστήματα)

- Είτε ως ξεχωριστά προγράμματα διαφορετικών χρηστών
- Είτε ως διεργασίες του ίδιου προγράμματος ενός χρήστη
- Είτε ως μέρη/διεργασίες μία σύνθετης εφαρμογής (application software)
- Είτε ως διεργασίες του ευρύτερου λειτουργικού συστήματος (system software)

Π.χ. ορισμός ταυτόχρονης (παράλληλης) εκκίνησης διεργασιών σε UCSD Pascal:

```
PROGRAM parallel;
var……

procedure <proc1>
begin
  ......
end

procedure <proc2>
begin
  ......
end

begin   {κύριο πρόγραμμα}
.........
start (proc1)
start (proc2)
.........
end  {κύριο πρόγραμμα}
```

ΔΙΑΔΙΕΡΓΑΣΙΑΚΗ ΕΠΙΚΟΙΝΩΝΙΑ

ΣΥΝΘΗΚΕΣ ΑΝΤΑΓΩΝΙΣΜΟΥ ΚΟΙΝΩΝ ΠΟΡΩΝ

Κρίσιμα Τμήματα – Αμοιβαίος Αποκλεισμός

(critical sections – mutual exclusion)

Οταν έχουμε ταυτόχρονη εκτέλεση πολλαπλών διεργασιών, σε πολλές περιπτώσεις συμβαίνει κάποιες από αυτές να χρειάζεται (απαιτούν) να προσπελάσουν ταυτόχρονα έναν κοινό πόρο του συστήματος, μία κοινή μεταβλητή ή μία κοινή δομή δεδομένων (π.χ. προσπέλαση περιφερειακών συσκευών, προσπέλαση δομών/πινάκων συστήματος, προσπέλαση κοινών buffers ή ευρετηρίων κ.λ.π.)

Σχεδόν όλα τα προβλήματα ταυτόχρονης προσπέλασης κοινών πόρων ανάγονται τελικά (πρωταρχικά) στο πρόβλημα ταυτόχρονης προσπέλασης μίας κοινής μεταβλητής ή δομής δεδομένων

Π.χ.: Πρόσβαση και αύξηση τιμής μία κοινής μεταβλητής X

Κώδικας διεργασίας 0:	**Κώδικας διεργασίας 1**:
begin	**begin**
1. tmp0 = X;	1. tmp1 = X;
2. tmp0 = tmp0 +1;	2. tmp1 = tmp1 +1;
3. X = tmp0;	3. X = tmp1;
end	**end**

Αν εκτελεστούν διακοπτόμενα εντολή-προς-εντολή, η τελική τιμή της X δεν θα είναι η σωστή

[ο κώδικας αύξησης της τιμής μίας κοινής μεταβλητής X – π.χ. X=X+1 αποτελεί κρίσιμο τμήμα – δεν επιτρέπεται να εκτελείται ταυτόχρονα από παραπάνω από 1 διεργασίες]

Γενικός ορισμός του προβλήματος (Mutual Exclusion)

- Εστω Ν διεργασίες κάθε μία από τις οποίες εκτελεί μία άπειρη ανακύκλωση, που μπορεί αν διαιρεθεί σε δύο τμήματα εντολών: (α) το κρίσιμο τμήμα (critical section) και (β) το μή κρίσιμο τμήμα (non-critical section)
- Οταν οι διεργασίες αυτές τρέχουν παράλληλα (ταυτόχρονα) πρέπει να ισχύει αμοιβαίος αποκλεισμός όσον αφορά την εκτέλεση κάθε κρίσιμου τμήματος (εκτελέσεις εντολών από τις κρίσιμες περιοχές δύο ή περισσοτέρων διεργασιών δεν πρέπει να επικαλύπτονται χρονικά)
- Μία διεργασία μπορεί να σταματήσει μόνο μέσα στο μή κρίσιμο τμήμα της. Αν συμβεί αυτό, δεν θα πρέπει να επηρεάσει τις άλλες διεργασίες.
- Η συνολική εκτέλεση των Ν διεργασιών δεν πρέπει να καταλήξει σε αδιέξοδο (έστω και για κάποιες από αυτές (αν περισσότερες από μία διεργασίες προσπαθήσουν ταυτοχρονα να εισέλθουν στην κρίσιμη περιοχή τους, τότε μία από αυτές πρέπει να το πετύχει.
- Καμία διεργασία δεν πρέπει να υποσιτιστεί (starvation – επ' αόριστον αναμονή). Αν μία διεργασία ξεκινήσει την προσπάθειά της να εισέλθει στην κρίσιμη περιοχή της, θα πρέπει τελικά κάποτε να το πετύχει.
- Στην περίπτωση έλλειψης συναγωνισμού (contention) για ταυτόχρονη είσοδο στις κρίσιμες περιοχές, μία διεργασία που επιθυμεί να ει΄σελθει στην κρίσιμη περιοχή της, θα πρέπει α το πετύχει με την ελάχιστη δυνατή επιβάρυνση (overhead)

Οι 3 βασικές συνθήκες καλής λύσης σε προβλήματα συντονισμού-αμοιβαίου αποκλεισμού

1. Δύο διεργασίες δεν πρέπει να βρίσκονται ποτέ ταυτόχρονα στα κρίσιμα τμήματά τους.
2. Διεργασία που δεν βρίσκεται σε κρίσιμο τμήμα δεν πρέπει να αναστέλλει άλλες διεργασίες (από το να μπούν σε αυτό).
3. Δεν επιτρέπεται η επ' αόριστον αναμονή μίας διεργασίας, για να εισέλθει στο κρίσιμο τμήμα της

- *Δεν επιτρέπονται υποθέσεις σε ότι αφορά την ταχύτητα ή το πλήθος των επεξεργαστών*

ΕΠΙΤΕΥΞΗ ΑΜΟΙΒΑΙΟΥ ΑΠΟΚΛΕΙΣΜΟΥ

☞ ***Απενεργοποίηση Διακοπών σε περιβάλλον χρήστη***

- Δεν ενδείκνυται (κακή λύση) – δεν χρησιμοποιείται

☞ ***Μεταβλητές Κλειδώματος – Προσπάθεια 1***

- 1 μεταβλητή - μεταβιβάζεται το πρόβλημα στην μεταβλητή κλειδώματος – απαράδεκτη

free = true;

Κώδικας διεργασίας 0:	**Κώδικας διεργασίας 1:**
while true {	while true {
1. ……	1. ……
2. while not free; /*wait*/	2. while not free; /*wait*/
3. free = false;	3. free = false;
4. critical_section();	4. critical_section();
5. free = true;	5. free = true;
6. ……	6. ……
7. }	7. }

☞ ***Μεταβλητές Κλειδώματος – Προσπάθεια 2***

- 2 μεταβλητές – συμβαίνει το ίδιο – απαράδεκτη

turn1 = false; turn2 = false;

Κώδικας διεργασίας 0:	**Κώδικας διεργασίας 1:**
while true {	while true {
1. ……	1. ……
2. while turn2; /*wait*/	2. while turn1; /*wait*/
3. turn1 = true;	3. turn2 = true;
4. critical_section();	4. critical_section();
5. turn1 = false;	5. turn2 = false;
6. ……	6. ……
7. }	7. }

☞ ***Αυστηρή Εναλλαγή – Προσπάθεια 3***

- Ναι μεν (συνθήκη 1) αλλά… (συνθήκες 2,3)

Κώδικας διεργασίας 0:	**Κώδικας διεργασίας 1**:
while (true) { 1. …… 2. while (turn !=0); /*wait*/ 3. critical_section(); 4. turn = 1; 5. …... 6. }	while (true) { 1. …… 2. while (turn !=1); /*wait*/ 3. critical_section(); 4. turn = 0; 5. ……. 6. }

ΛΥΣΕΙΣ DEKKER – PETERSON

☞ ***Συνδυασμός 'κλειδώματος' και 'εναλλαγής'***

☞ ***Δήλωση 'πρόθεσης εισόδου'***

```
int turn;
int interested[2];
```

Κώδικας διεργασίας 0:	**Κώδικας διεργασίας 1**:
while true { 1. ..... 2. interested[0] = true; 3. turn = 0; 4. while (turn == 0 && interested[1] == true) ; 5. critical_section(); 6. interested[0] = false; 7. ..... }	while true { 1. ..... 2. interested[1] = true; 3. turn = 1; 4. while (turn == 1 && interested[0] == true) ; 5. critical_section(); 6. interested[1] = false; 7. ..... }

→ **Πρόβλημα:** η Ενεργός Αναμονή (busy waiting)

Test and Set Lock (TSL)

Χρήση κοινής μεταβλητής κλειδώματος ('flag') ΑΛΛΑ: Διάβασμα της τιμής (σε καταχωρητή) και θέση της (π.χ. flag=1) σε «ατομική/αδιαίρετη εντολή»

Κώδικας διεργασίας 0:	**Κώδικας διεργασίας 1:**
1. In: tsl register, flag 2. cmp register, #0 3. jnz In 4. critical_section(); 5. mov flag, #0	1. In: tsl register, flag 2. cmp register, #0 3. jnz In 4. critical_section(); 5. mov flag, #0
Κώδικας διεργασίας 0:	**Κώδικας διεργασίας 1**:
1. In: if tsl(flag) goto In; 2. critical_section(); 3. flag=0;	1. In: if tsl(flag) goto In; 2. critical_section(); 3. flag=0;

→ **Πρόβλημα:** η Ενεργός Αναμονή (busy waiting)

Εναλλακτική Λύση:

Αναστολή Διεργασιών (αντί ενεργούς αναμονής) όταν δεν μπορούν να μπουν στο κρίσιμο τμήμα τους

1ο Παράδειγμα/Τεχνική: 'Απενεργοποίηση και Αφύπνιση'

Απενεργοποίηση και Αφύπνιση / SLEEP()-WAKEUP()

Παράδειγμα:

Το πρόβλημα Παραγωγού-Καταναλωτή (ή αλλοιώς: της περιορισμένης ενδιάμεσης μνήμης): Ενδιάμεση 'κοινή' μνήμη 'ασύγχρονης' επικοινωνίας δύο διεργασιών (shared buffer) μεγέθους Ν. Μέσω αυτής (τοποθετώντας τα σε αυτήν με τη σειρά) μία Διεργασία Δ1 'στέλνει' δεδομένα στην Διεργασία Δ2 (η Δ2 πάιρνει ένα-ένα τα δεδομένα από την κοινή μνήμη όταν τα χρειάζεται)

Τί γίνεται,

(a) αν η Δ1 θέλει να στείλει ένα μήνυμα (να το τοποθετήσει στην κοινή μνήμη) αλλά αυτή είναι γεμάτη γιατί η Δ2 δεν έχει πάρει (καταναλώσει) τίποτα από αυτήν ;
(b) αν η Δ2 χρειάζεται να πάρει ένα επόμενο μήνυμα από την κοινή μνήμη αλλά αυτή είναι άδεια γιατί η Δ1 δεν έχει στείλει (παράγει) ακόμα κανένα (το επόμενο) μήνυμα ;

Λύση:

(a) Αν η Δ1 βρίσκει γεμάτη τη μνήμη (Ν μηνύματα) απενεργοποιείται
(b) Αν η Δ2 βρίσκει άδεια τη μνήμη (0 μηνύματα) απενεργοποιείται
(c) Αν η Δ1 βρίσκει άδεια τη μνήμη (0 μηνύματα) βάζει το μήνυμα και ξυπνά τη Δ2
(d) Αν η Δ2 βρίσκει γεμάτη τη μνήμη (Ν μηνύματα) παίρνει ένα μήνυμα και ξυπνά τη Δ1

Πρόβλημα: (πρόσβαση κοινών μεταβλητών/μετρητών)

→ *κίνδυνος να μείνουν και οι 2 απενεργοποιημένες για πάντα*

Το Πρόβλημα Παραγωγού-Καταναλωτή με χρήση SLEEP()-WAKEUP()

```
#include "prototypes.h"
#define N 100                       /* πλήθος θέσεων στην ενδιάμεση μνήμη */
int count = 0;                      /* πλήθος δεδομένων στην ενδιάμεση μνήμη */


void producer(void)
{
 int item;

 while (TRUE)  {                    /* επαναλάμβανε συνεχώς */

   produce_item(&item);             /* παραγωγή νέου δεδομένου */

   if (count == N)  sleep();        /* απενεργοποίηση αν η ενδιάμεση μνήμη
                                       είναι γεμάτη */
   enter_item(item);                /* τοποθέτησε το νέο δεδομένο στην
                                       ενδιάμεση μνήμη */
   count = count + 1;               /* αύξησε το πλήθος των δεδομένων στην
                                       ενδιάμεση μνήμη */
   if (count == 1)                  /* είναι η ενδιάμεση μνήμη άδεια; */
      wakeup(consumer);
 }
}


void consumer(void)
{
 int item;

 while (TRUE)  {                    /* επαναλάμβανε συνεχώς */

   if (count == 0)  sleep();        /* απενεργοποίηση αν η ενδιάμεση μνήμη
                                       είναι γεμάτη */
   remove_item(item);               /* απόσυρση δεδομένου από την
                                       ενδιάμεση μνήμη */
   count = count - 1;               /* μείωσε το πλήθος των δεδομένων στην
                                       ενδιάμεση μνήμη */
   if (count == N-1)                /* είναι η ενδιάμεση μνήμη γεμάτη; */
      wakeup(producer);

   consume_item(item);              /* εκτύπωσε το δεδομένο */
 }
}
```

ΣΗΜΑΦΟΡΟΙ (WAIT/SIGNAL ή DOWN/UP)

Ακέραια μεταβλητή (π.χ.: S)

Συγχρονισμός μέσω σημάτων (και καταμέτρησής τους)

Θεμελειώδεις λειτουργίες (WAIT/SIGNAL ή DOWN/UP)

WAIT (S) : if (S>0) S=S-1; "continue";
else "απενεργοποίηση (εισαγωγή σε ουρά αναμονής)"

SIGNAL (S) : S=S+1; "continue";
Αν η ουρά αναμονής δεν είναι άδεια «ξυπνά/απενεργοποιεί» μία από αυτές που είχαν ανασταλεί/απενεργοποιηθεί η οποία θα συνεχίσει ολοκληρώνοντας το 'WAIT'

Οι λειτουργία αρχικοποίησης SEMINIT (S, value)

Αδαίρετη/ατομική υλοποίηση των παραπάνω

Εφαρμογή στο πρόβλημα Παραγωγού - Καταναλωτή

Δυαδικοί Σημαφόροι:
σημαφόροι που χρησιμοποιούνται για την εξασφάλιση αμοιβαίου αποκλεισμού στην χρήση κρίσιμων τμημάτων
Αρχική τιμή: '1' Λαμβάνουν τιμές '0' και '1';

mutex:=1	
Κώδικας διεργασίας 0: while true { 1. …….. 2. WAIT(&mutex); 3. critical_section(); 4. SIGNAL(&mutex); 5. …….. }	**Κώδικας διεργασίας 1**: while true { 1. …….. 2. WAIT(&mutex); 3. critical_section(); 4. SIGNAL(&mutex); 5. …….. }

Λύση στο Πρόβλημα Παραγωγού-Καταναλωτή με χρήση Σημαφόρων

```
#include "prototypes.h"
#define N 100                       /* πλήθος θέσεων στην ενδιάμεση μνήμη */
typedef int semaphore;              /* οι σημαφόροι αποτελούν ειδικό τύπο*/
semaphore mutex = 1;                /* ελέγχει την πρόσβαση στην κρίσιμη περιοχή */
semaphore empty = N;                /* πλήθος άδειων θέσεων της ενδιάμεσης  μνήμης */
semaphore full = 0;                 /* πλήθος κατειλημμένων θέσεων της ενδ. μνήμης */

void producer(void)
{
 int item;

 while (TRUE)  {                    /* TRUE ισοδυναμεί με 1 */

   produce_item(&item);             /* παραγωγή νέου δεδομένου προς τοποθέτηση
                                       στην ενδιάμεση μνήμη */
   wait(&empty);                    /* μείωσε το πλήθος των άδειων θέσεων */

   wait (&mutex);                   /* εισαγωγή σε κρίσιμη περιοχή */
   enter_item(item);                /* τοποθέτησε το νέο δεδομένο στην ενδ. μνήμη */
   signal(&mutex);                  /* εγκατάλειψη κρίσιμης περιοχής */

   signal(&full);                   /* αύξησε το πλήθος των κατειλημμένων θέσεων */
 }
}


void consumer(void)
{
 int item;

 while (TRUE)  {                    /* επαναλάμβανε συνεχώς */

   wait(&full);                     /* μείωσε το πλήθος των κατειλημμένων
                                       θέσεων */
   wait(&mutex);                    /* εισαγωγή σε κρίσιμη περιοχή */
   remove_item(item);               /* απόσυρε δεδομένο από την ενδ. μνήμη */
   signal(&mutex);                  /* εγκατάλειψη κρίσιμης περιοχής */

   signal(&empty);                  /* αύξησε το πλήθος των άδειων θέσεων */

   consume_item(item);              /* χρησιμοποίησε το δεδομένο */
 }
}
```

ΠΡΟΒΛΗΜΑΤΑ – ΑΣΚΗΣΕΙΣ #1

Ελέξτε για τα παρακάτω προγράμματα αν ικανοποιούν τι 3 συνθήκες καλής λύσης σε προβλήματα συντονισμού διαδικασιών

Πρόγραμμα 1.1

```
        shared int busy = 0, trying = -1;

start:  while (busy == 1)  { };
        trying = GetPid();                  // Η GetPid() επιστρέφει την ταυτότητα
                                            της διεργασίας, ένα θετικό ακέραιο
        if (busy) goto start;
        busy = 1;
        if (trying != GetPid()) goto start;

        /* είσοδος στην κρίσιμη περιοχή */
        ...
        /* έξοδος από την κρίσιμη περιοχή*/

        busy = 0;
```

Πρόγραμμα 1.2

```
        shared int busy = 0, trying = -1;

start:  trying = GetPid();                  // Η GetPid() επιστρέφει την ταυτότητα
                                            της διεργασίας, ένα θετικό ακέραιο
        if (busy) goto start;
        busy = 1;
        if (trying != GetPid()) {  busy = 0;  goto start;  }

        /* είσοδος στην κρίσιμη περιοχή */
        ...
        /* έξοδος από την κρίσιμη περιοχή*/

        busy = 0;
```

ΠΡΟΒΛΗΜΑΤΑ – ΑΣΚΗΣΕΙΣ #2

ΑΣΚΗΣΗ 2.1

Προγραμματείστε Ν αμαξοστοιχίες που πρέπει να διασχίσουν αποκλειστικά μία μοναδική γέφυρα ή σιδηροδρομική γραμμή χωρίς να συγκρουστούν

typedef int semaphore;
semaphore bridge_free = 1;

Κώδικας αμαξοστοιχίας 1:	**Κώδικας αμαξοστοιχίας 2**:
while true { 1. Διαδρομή_πριν_Γέφυρα (); 2. WAIT(&bridge_free); 3. Διάσχιση_Γέφυρας (); 4. SIGNAL(&bridge_free); 5. Διαδρομή_μετά_Γέφυρα (); 6. }	while true { 1. Διαδρομή_πριν_Γέφυρα (); 2. WAIT(&bridge_free); 3. Διάσχιση_Γέφυρας (); 4. SIGNAL(&bridge_free); 5. Διαδρομή_μετά_Γέφυρα (); 6. }
Κώδικας αμαξοστοιχίας ...Χ...:	**Κώδικας αμαξοστοιχίας Ν**:
while true { 1. Διαδρομή_πριν_Γέφυρα (); 2. WAIT(&bridge_free); 3. Διάσχιση_Γέφυρας (); 4. SIGNAL(&bridge_free); 5. Διαδρομή_μετά_Γέφυρα (); 6. }	while true { 1. Διαδρομή_πριν_Γέφυρα (); 2. WAIT(&bridge_free); 3. Διάσχιση_Γέφυρας (); 4. SIGNAL(&bridge_free); 5. Διαδρομή_μετά_Γέφυρα (); 6. }

Συνθήκες Λειτουργίας Σημαφόρων:

S>=0	**S = S0 + # signal(S) - # wait(S)**	**# wait(S) <= # signal(S) + S_0**

ΑΣΚΗΣΗ 2.2

Θεωρείστε ένα σύνολο διεργασιών που διαχειρίζονται έναν κοινό λογαριασμό τραπέζης χρησιμοποιώντας κοινή μνήμη. Η κοινή μεταβλητή `balance` υποδηλώνει το ποσό που περιέχει ο λογαριασμός ανά πάσα στιγμή.

Ερώτημα 1

Έστω ότι για να πραγματοποιήσουν αναληψεις, οι διεργασίες εκτελούν τον ακόλουθο κώδικα:

```
shared int balance;
boolean withdraw(int amount)
{
        if (balance – amount >= 0) {
                balance = balance – amount;
                return OK;
        }
        else {
                error("Δεν επιτρέπεται ανάληψη τόσο μεγάλου ποσού!");
                return NOT_OK;
}
```

όπου η παράμετρος amount αποθηκεύει το ποσό της εκάστοτε ανάληψης.
Υποθέστε ότι η διεργασία Α θέλει να κάνει ανάληψη το ποσό των 500 Ευρώ από ένα λογαριασμό που περιέχει 800 Ευρώ, ενώ η Β θέλει να κάνει ανάληψη το ποσό των 400 Ευρώ από τον ίδιο λογαριασμό (δηλαδή, η διεργασία Α καλεί τη διαδικασία `withdraw(500)`, ενώ η Β καλεί τη `withdraw(400)`.

Δεδομένου ότι οι ταχύτητες των δύο διεργασιών είναι αυθαίρετες και ότι οι δύο κλήσεις στην `withdraw` γίνονται ταυτόχρονα, περιγράψτε όλα τα δυνατά σενάρια και εξηγήστε ποιες είναι οι δυνατές τιμές που μπορεί να έχει η μεταβλητή `balance`, όταν και οι δύο διαδικασίες θα έχουν τερματίσει.

Ερώτημα 2

Περιγράψτε τις εξής δύο διαδικασίες:
(1) `void deposit(int amount)`, και
(2) `void withdraw(int amount)`,
για τις οποίες θα πρέπει να ισχύουν τα ακόλουθα:

- ✓ Η `deposit` καταθέτει το ποσό `amount` στον κοινό λογαριασμό.
- ✓ Η `withdraw` κάνει ανάληψη το ποσό `amount` από τον λογαριασμό.
- ✓ Και οι δύο διαδικασίες θα πρέπει να μπορούν να εκτελεστούν από πολλές διεργασίες ταυτόχρονα, χωρίς να προκύπτουν προβλήματα και το τελικό αποτέλεσμα θα πρέπει να είναι σωστό.
- ✓ Δεν επιτρέπονται υποθέσεις σε ότι αφορά την ταχύτητα ή το πλήθος των επεξεργαστών.

Θα πρέπει να περιγράψετε δύο λύσεις:
α) μια λύση βασισμένη σε σημαφόρους,
γ) μια λύση βασισμένη στη χρήση της ατομικής εντολής `TSL()`

ΑΣΚΗΣΗ 2.3

Εστω ένα σύστημα στο οποίο ένα σύνολο από διεργασίες-πελάτες (clients) επικοινωνούν με μία διεργασία-εξυπηρετητή (server) μέσω μίας κοινής μεταβλητής X. Κάθε πελάτης μπορεί να παράγει επαναληπτικά αιτήσεις. Κάθε φορά που ένας πελάτης παράγει μία αίτηση πρέπει (α) να ενεργοποιηθεί ο εξυπηρετητής ώστε να εξυπηρετήσει την αίτηση, ενώ (β) ενδιάμεσα κανένας άλλος πελάτης δεν θα πρέπει να μπορεί να παράγει αιτήσεις. Ο εξυπηρετητής με τη σειρά του αφού εξυπηρετήσει την αίτηση θα πρέπει να ενεργοποιήσει στη συνέχεια κάποιον άλλον πελάτη (ή ίσως και τον ίδιο) για την παραγωγή μίας νέας αίτησης.

```
Semaphore client_request =1;
Semaphore server_on = 0;
int true = 1;
```

```
/* Κώδικας εκτελούμενος επαναληπτικά από κάθε πελάτη */
------------
Client_i_
------------

while(true)
{

wait(client_request);

produce_request();

signal(server_on);

}
```

```
/* Κώδικας εκτελούμενος επαναληπτικά από τον εξυπηρετητή */
------------
Server
------------

while(true)
{

wait (server_on);

process_request();

signal (client_request);

}
```

ΠΑΡΑΚΟΛΟΥΘΗΤΕΣ (monitors)

- ☞ *Συλλογή από διαδικασίες, μεταβλητές, δεδομένα κ.λ.π. ομαδοποιημένες σε έναν ειδικό τύπο ενότητας ή πακέτου*

- ☞ *Οι διεργασίες μπορούν να καλέσουν τις διαδικασίες ενός παρακολουθητή αλλά δεν μπορούν να προσπελάσουν άμεσα τις εσωτερικές δομές του*

- ☞ *Μία μόνο διεργασία κάθε χρονική στιγμή μπορεί να είναι ενεργή σε έναν παρακολουθητή*

- Τί άλλο χρειάζεται επιπλέον για την επίτευξη αμοιβαίου αποκλεισμού ? Τί λείπει σε σχέση με το μηχανισμό των «σημαφόρων» ?

- Μηχανισμοί αναστολής (όταν π.χ. μία διεργασία μέσα σε παρακολουθητή δεν μπορεί να συνεχίσει) και αφύπνισης διεργασιών

CWAIT(c): *αναστέλλει την καλούσα διεργασία (όταν π.χ. αυτή δεν μπορεί να συνεχίσει – π.χ. ο παραγωγός όταν βρεί την ενδιάμεση μνήμη γεμάτη) και επιτρέπει σε μία άλλη διεργασία να χρησιμοποιήσει τον παρακολουθητή. Συνδέεται δε, με μία μεταβλητή συνθήκης (π.χ. 'c')*

CSIGNAL(c): *αφυπνίζει την απενεργοποιημένη συζηγή της (π.χ. ο καταναλωτής όταν καταναλώσει ένα από τα μηνύματα της γεμάτης ενδιάμεσης μνήμης) και εξέρχεται του παρακολουθητή. Αν στην μεταβλητή συνθήκης αναμένουν πολλές διεργασίες, ο χρονοδρομολογητής αποφασίζει ποια θα αφυπνισθεί*

ΔΟΜΗ ΕΝΟΣ ΠΑΡΑΚΟΛΟΥΘΗΤΗ

```
monitor example

    integer i;
    condition c;

    procedure producer(x);
    .
    .
    .
    end;

    procedure consumer(x);
    .
    .
    .
    end;

end monitor;
```

- *Το πρόβλημα του Παραγωγού-Καταναλωτή με χρήση «Παρακολουθητών»*

Το πρόβλημα Παραγωγού-Καταναλωτή με χρήση Παρακολουθητή

```
monitor ProducerConsumer

        condition full, empty;
        integer count;

        procedure enter;
        begin
          if count=N then cwait(full);
          enter_item;
          count:=count+1;
          if count=1 then csignal(empty);
        end;

        procedure remove;
        begin
          if count=0 then cwait(empty);
          remove_item;
          count:=count-1;
          if count=N-1 then csignal(full);
        end;

        count:=0;

end monitor;


procedure producer;
begin
  while true do
  begin
     produce_item;
     ProducerConsumer.enter;
  end
end;


procedure consumer;
begin
  while true do
  begin
      ProducerConsumer.remove;
      consume_item;
  end
end;
```

Το Πρόβλημα των Αναγνωστών & Εγγραφέων

Πολλοί αναγνώστες και εγγραφείς θέλουν να προσπελάσουν μία Β.Δ. – οι μεν για να διαβάσουν κάτι, οι δε για να γράψουν. Επιτρέπεται ταυτόχρονα να διαβάζουν περισσότερο από ένας. Δεν επιτρέπεται όμως να γράφουν ταυτόχρονα περισσότεροι από έναν. Όταν επίσης κάποιος γράφει δεν επιτρέπεται κανένας άλλος ούτε καν να διαβάζει.

```
semaphore mutex = 1;
semaphore db = 1;
int rc = 0;

void reader(void)
{
   while (TRUE)  {
      wait(&mutex);
      rc = rc +1;
      if (rc==1)
         wait(&db);
      signal(&mutex);

      read_database();

      wait(&mutex);
      rc = rc -1;
      if (rc==0)
          signal(&db);
      signal(&mutex);

      use_data_read();
      }
 }

void writer(void)
{
  while (TRUE)  {
     think_up_data();
     wait(&db);
     write_database();
     signal(&db);
   }
}
```

Το Πρόβλημα του Κοιμώμενου Κουρέα

Το κουρείο διαθέτει έναν κουρέα, μία θέση για κούρεμα και 'ν' θέσεις αναμονής για τους πελάτες. Οταν δεν έχει πελάτες ο κουρέας κοιμάται. Αν έρθει κάποιος πελάτης πρέπει να τον ξυπνήσει. Οι επόμενοι που θα έρθουν αναμένουν στις σχετικές θέσεις ή εκτός εάν δεν υπάρχουν κενές θέσεις οπότε φεύγουν.

```
#define CHAIRS 5
semaphore customers = 0;
semaphore barbers = 0;
semaphore mutex = 1;
int waiting = 0;

void barber(void)
{
   while (TRUE)  {
      wait(&customers);
      wait(&mutex);
      waiting = waiting – 1;
      signal(&barbers);
      signal(&mutex);
      cut_hair();
    }
 }

void customer(void)
{
     wait(&mutex);
      if (waiting < CHAIRS)  {
          waiting = waiting + 1;
          signal(&customers);
          signal(&mutex);
          wait(&barbers);
          get_haircut();
       }
       else  signal(&mutex);
    }
 }
```

Το πρόβλημα των ‘μπαμπουίνων’

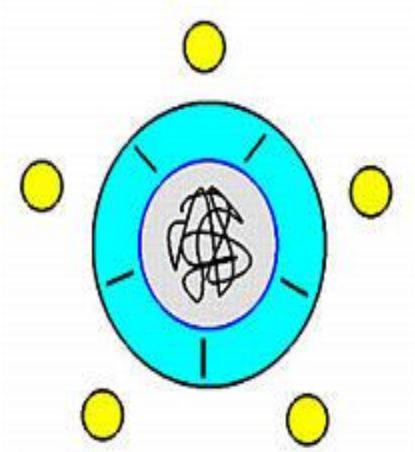
```
#define TRUE 1
#define FALSE 0
typedef int semaphore;
semaphore mutex=1;
semaphore west=1;
semaphore east=1;
share int west_count=0;
share int east_count=0;
```

Για να διαπεράσουν μία χαράδρα οι ‘μπαμπουίνοι’ έχουν δέσει ένα σχοινί από τη μία άκρη έως την άλλη και το χρησιμοποιούν όποτε το χρειάζονται. Προφανώς την ώρα που έχει μπεί ένας μπαμπουίνος να το χρησιμοποιήσει για να περάσει προς τη μία κατεύθυνση δεν πρέπει να επιτραπεί η είσοδος σε κανέναν από την αντίθετη κατεύθυνση

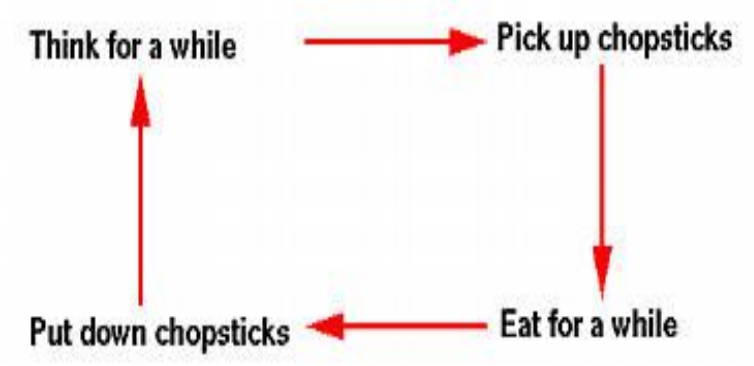
```
void east(void)  {
   while (TRUE)  {
       wait(&east);
       east_count++;
       if (east_count==1)
           wait(&mutex);
       signal(&east);
       move_east();
       wait(&east);
       east_count--;
       if (east_count==0)
           signal(&mutex);
       signal(&east);
   }
}
```

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

Το πρόβλημα των φιλοσόφων



(κινέζικη παραλλαγή)



Το πρόβλημα των φιλοσόφων – Λύση 1

έστω N=5

```
Philosopher(int i)  {
  while TRUE {
  think();                    /* σκέψου */
  take_fork(i);               /* πάρε αριστερό πηρούνι) */
  take_fork((i mod N)+1);     /* πάρε δεξί πηρούνι) */
  eat();                      /* επιτέλους! */
  put_fork(i);                /* άφησε αριστερό πηρούνι) */
  put_fork((i mod N)+1);      /* άφησε δεξί πηρούνι) */
  }
}
```

προβλήματα? starvation, αν όλοι πάρουν το αριστερό πηρούνι τους ταυτόχρονα

Το πρόβλημα των φιλοσόφων – Λύση 2

έστω N=5

```
Philosopher(int i) {
  while TRUE {
○ down(mutex);
  think();                    /* σκέψου */
  take_fork(i);               /* πάρε αριστερό πηρούνι) */
  take_fork((i mod N)+1);     /* πάρε δεξί πηρούνι) */
  eat();                      /* επιτέλους! */
  put_fork(i);                /* άφησε αριστερό πηρούνι) */
  put_fork((i mod N)+1);      /* άφησε δεξί πηρούνι) */
○ up(mutex);
  }
}
```

προβλήματα? μόνο ένας τη φορά μπορεί να τρώει

Το πρόβλημα των φιλοσόφων – Λύση 3

semaphore mutex, s[N] (μια σημαφόρος ανά φιλόσοφο)

```
Philosopher(int i) {      /* i η ταυτότητα φιλοσόφου */
  while TRUE {
    think();              /* σκέψου */
    take_forks(i);        /* πάρε πηρούνια ή αναστολή */
    eat();                /* φαγητό */
    put_forks(i);         /* άφησε τα πηρούνια */
  }
}
```


Το πρόβλημα των φιλοσόφων – Λύση 3 …συν.

```
take_forks(int i) {
  down(mutex);              /* αρχίζει κρίσιμη περιοχή */
  state[i]=HUNGRY;          /* ο i πεινάει */
  test(i);                  /* δοκίμασε να πάρεις πηρούνια*/
  up(mutex);                /*τέλος κρίσιμης περιοχής*/
  down(s[i]);               /*αναστολή αν δε βρεις πηρούνια*/
{
put_forks(int i) {
  down(mutex);              /* αρχίζει κρίσιμη περιοχή */
  state[i]=THINKING;        /* ο i σκέφτεται */
  test(LEFT);               /*μπορεί ο αριστερά να φάει; */
  test(RIGHT);              /*μπορεί ο δεξιά να φάει; */
  up(mutex);                /*τέλος κρίσιμης περιοχής*/
}
test(int i) {
  if (state[i]=HUNGRY AND state[LEFT]<>EATING AND
  state[RIGHT]<>EATING) then
    state[i]=EATING; up(s[i]);
}
```

ΓΡΑΦΟΙ ΠΡΟΤΕΡΑΙΟΤΗΤΩΝ (ΓΡΑΦΗΜΑΤΑ ΣΥΓΧΡΟΝΙΣΜΟΥ)

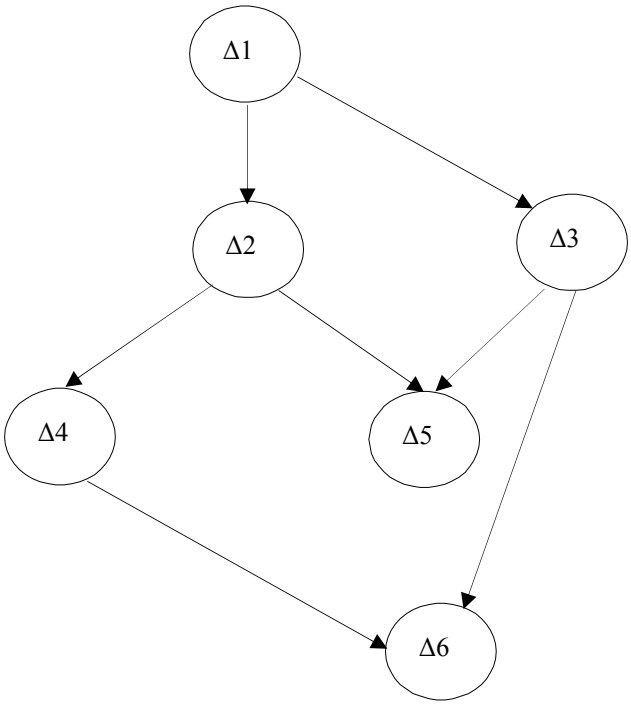
Θεωρήστε ότι πρέπει να συγχρονίσετε την εκτέλεση των διαδικασιών Δ1, Δ2, Δ3, Δ4, Δ5 και Δ6 σύμφωνα με τους παρακάτω περιορισμούς:

- Η Δ1 εκτελείται πριν από τις Δ2 και Δ3.
- Η Δ2 εκτελείται πριν από τις Δ4 και Δ5.
- Η Δ3 εκτελείται πριν από την Δ5.
- Η Δ6 εκτελείται μετά από τις Δ3 και Δ4.

Η έκφραση «η διαδικασία Δi εκτελείται πριν από τη διαδικασία Δj» σημαίνει ότι η εκτέλεση της Δi πρέπει να ολοκληρωθεί πριν αρχίσει η εκτέλεση της Δj. Αναλόγως η έκφραση «η διαδικασία Δi εκτελείται μετά από τη διαδικασία Δj» σημαίνει ότι η εκτέλεση της Δi μπορεί να αρχίσει αφού ολοκληρωθεί η εκτέλεση της Δj.

Για το συγχρονισμό των διαδικασιών Δ1, …, Δ6 σύμφωνα με τα παραπάνω, έχετε στη διάθεσή σας σημαφόρους. Να δώσετε ένα «παράλληλο» πρόγραμμα (με χρήση εντολών **cobegin** και **coend**) που να υλοποιεί την εκτέλεση των Δ1, …, Δ6 σύμφωνα με τους παραπάνω περιορισμούς με χρήση σημαφόρων.

ΓΡΑΦΟΣ ΠΡΟΤΕΡΑΙΟΤΗΤΩΝ



***Α' Εναλλακτική Λύση*:** Χρήση σημαφόρων Σij για τον έλεγχο των προτεραιοτήτων Δi → Δj (χρησιμοποιούμε μία σημαφόρο Σij – με αρχική τιμή '0' – για τον έλεγχο κάθε σχέσης προτεραιότητας Δi → Δj):

```
var Σ12, Σ13, Σ24, Σ25, Σ35, Σ36, Σ46: semaphores;
  Σ12:= Σ13:= Σ24:= Σ25:= Σ35:= Σ36:= Σ46:= 0;
cobegin
  begin Δ1; V(Σ12); V(Σ13); end;
  begin P(Σ12); Δ2; V(Σ24); V(Σ25); end;
  begin P(Σ13); Δ3; V(Σ35); V(Σ36);end;
  begin P(Σ24); Δ4; V(Σ46); end;
  begin P(Σ25); P(Σ35); Δ5; end;
  begin P(Σ46); P(Σ36); Δ6; end;
coned
```

Β' Εναλλακτική Λύση: Χρήση σημαφόρων Σi για τον έλεγχο των μη-αρχικών διαδικασιών Δi (χρησιμοποιούμε μία σημαφόρο Σi για τον έλεγχο κάθε μη-αρχικής διαδικασίας, δηλαδή για κάθε διαδικασία Δi εκτός της Δ1, και της δίνουμε αρχική τιμή ανάλογη με τον αριθμό των διαδικασιών Δj τις οποίες πρέπει να περιμένει πριν αρχίσει την εκτέλεσή της):

```
var Σ2, Σ3, Σ4, Σ5, Σ6: semaphores;
  Σ2:= 0, Σ3:= 0, Σ4:= 0, Σ5:= -1, Σ6:= -1;
cobegin
  begin Δ1; V(Σ2); V(Σ3); end;
  begin P(Σ2); Δ2; V(Σ4); V(Σ5); end;
  begin P(Σ3); Δ3; V(Σ5); V(Σ6);end;
  begin P(Σ4); Δ4; V(Σ6); end;
  begin P(Σ5); Δ5; end;
  begin P(Σ6); Δ6; end;
coend
```