

Processes & Synchronization

Chapter 4

<< Semaphores >>

Introduction

Busy-waiting protocols for process synchronization:

- **Quite complex**
- **Inefficient implementation**

Need for special tools for process synchronization

Semaphores: **widely used as synchronization tools in threads and parallel programming libraries**

Introduction (cont.)

The basic idea come from train semaphores:

- a signal flag that indicates whether or not the track ahead is clear or occupied by another train.
- As the train proceeds, semaphores are set and cleared.
- So, a mechanism which ensures mutual exclusive occupancy of critical section of track.
- Similarly, in concurrent programs semaphores provide a basic signalling mechanism to implement mutual exclusion and condition

Semaphores first described by Dijkstra in 1968

Syntax amd Semantics

- A semaphore is a special kind of shared variable that is manipulated only by two atomic operations, P and V.
- The value of a semaphore is a nonnegative integer.

$P(s): < \text{await } (s > 0) \ s = s - 1; >$

$V(s): < s = s + 1; >$

- A general semaphore is one that can take on any nonnegative value.
- A binary semaphore is one whose value is always either 0 or 1.

Syntax and Semantics (Cont.)

- **A semaphore has the following properties:**
- **P(S) and V(S) are atomic instructions:**
- **no instructions can be interleaved between the test that $S > 0$ and the decrement of S**
- **A semaphore must be given a non-negative initial value.**
- **The V(S) operation must awaken one of the suspended processes. The definition does not specify which process must be awakened.**
- **Usually in implementations of semaphores, processes are awakened in the order they were delayed.**

Syntax and Semantics (Cont.)

Blocking Implementation

- $P(S)$: if $(S > 0)$ $S = S - 1$; “continue”;
else “block” (insert the process in a waiting queue)
- $V(S)$: $S = S + 1$; “continue” ...and... --> if the waiting queue is not empty, awaken one of the blocked processes in order to complete its ‘P’ operation

Busy Waiting Implementation

- $P(S)$: while $S \leq 0$ skip; $S = S - 1$; “continue”;
- $V(S)$: $S = S + 1$; “continue”;

Critical Sections: Mutual Exclusion

```
sem mutex = 1;

process CS[i = 1 to n] {
    while (true) {
        P(mutex);
        critical section;
        V(mutex);
        noncritical section;
    }
}
```

Semaphore solution to the critical section problem

Invariant: $\#CS + \text{mutex} = 1$

Barriers: Signaling Events

Two-process Barrier:

- The basic idea is to use one semaphore for each synchronization flag

```
sem arrive1 = 0, arrive2 = 0;
process Worker1 {
    ...
    V(arrive1);    /* signal arrival      */
    P(arrive2);    /* wait for other process */
    ...
}
process Worker2 {
    ...
    V(arrive2);    /* signal arrival      */
    P(arrive1);    /* wait for other process */
    ...
}
```

- arrive1, arrive 2: signaling semaphores
- Signaling semaphore s:
 - A process signals an event by executing V(s)
 - Other processes wait for the event by executing P(s)

Barriers: Use of a coordinator

```
semaphore here=0, go[1:2] = {0,0}
co
    while 1 {
        beforebarrier1;
        V(here); P(go[1]);
    }
//
    while 1 {
        beforebarrier2;
        V(here); P(go[2]);
    }
// -- coordinator
    while 1 {
        for [i=1,2] {
            P(here)
        };
        for [i=1,2] {
            V(go[i])
        }
    }
oc
```

Producers and Consumers: Split

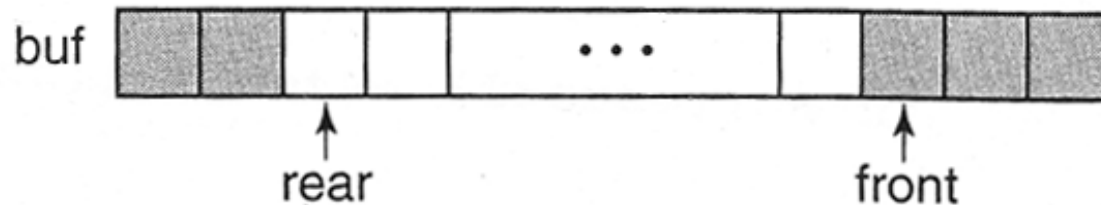
Binary Semaphores

```
type T buf;      /* a buffer of some type T */
sem empty = 1, full = 0;
process Producer[i = 1 to M] {
    while (true) {
        ...
        /* produce data, then deposit it in the buffer */
        P(empty);
        buf = data;
        V(full);
    }
}
process Consumer[j = 1 to N] {
    while (true) {
        /* fetch result, then consume it */
        P(full);
        result = buf;
        V(empty);
        ...
    }
}
```

- Empty and full: *split binary semaphores* because at most one of these two can be 1 at a time.

Bounded Buffers: Resource Counting

- Now the shared buffer can hold n elements
- So, the producers can run ahead of the consumers (up to some limit).
- Put items in the *rear* and fetch items in the *front* of the array.



- $\text{Buf}[\text{rear}] = \text{data}; \text{rear} = (\text{rear} + 1) \% n;$
- $\text{Results} = \text{buf}[\text{front}]; \text{front} = (\text{front} + 1) \% n;$

Bounded Buffers: One Producer and Consumer

```
type T buf[n];          /* an array of some type T */
int front = 0, rear = 0;
sem empty = n, full = 0; /* n-2 <= empty+full <= n */

process Producer {
    while (true) {
        ...
        produce message data and deposit it in the buffer;
        P(empty);
        buf[rear] = data; rear = (rear+1) % n;
        V(full);
    }
}

process Consumer {
    while (true) {
        fetch message result and consume it;
        P(full);
        result = buf[front]; front = (front+1) % n;
        V(empty);
        ...
    }
}
```

- **empty, full: semaphores used as resource counters**
- **Each counts the number of units of a resource**

Bounded Buffers: Multiple Producers and Consumer

```
typeT buf[n];          /* an array of some type T */
int front = 0, rear = 0;
sem empty = n, full = 0; /* n-2 <= empty+full <= n */
sem mutexD = 1, mutexF = 1; /* for mutual exclusion */

process Producer[i = 1 to M] {
    while (true) {
        ...
        produce message data and deposit it in the buffer;
        P(empty);
        P(mutexD);
        buf[rear] = data; rear = (rear+1) % n;
        V(mutexD);
        V(full);
    }
}

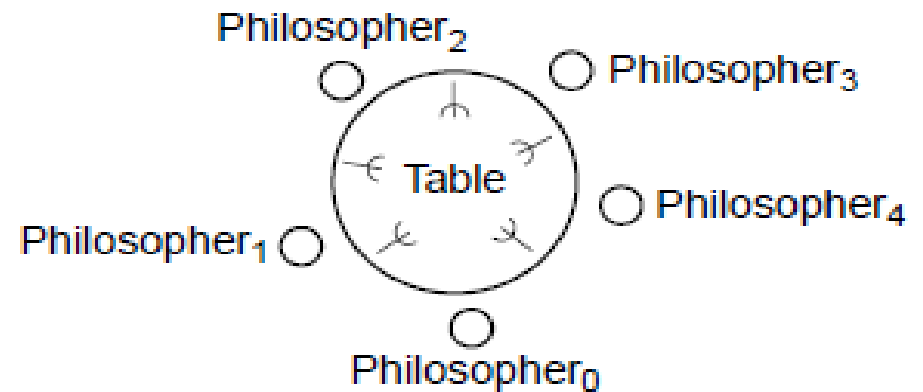
process Consumer[j = 1 to N] {
    while (true) {
        fetch message result and consume it;
        P(full);
        P(mutexF);
        result = buf[front]; front = (front+1) % n;
        V(mutexF);
        V(empty);
        ...
    }
}
```

- Semaphores mutexD, mutexF enforce mutual exclusion for deposit and fetch operations respectively.

The Dining Philosophers

- Five philosophers thinking or eating.
- Each should use two forks to eat

```
Process Philosophers [i=0 to 4]{  
    while(true) {  
        think;  
        acquire forks;  
        eat;  
        release forks;  
    }  
}
```



- Deadlock is possible:
 - All philosophers pick up their left fork .
 - All wait forever for their right fork
- A necessary condition for deadlock: circular waiting.

The Dining Philosophers (Cont.)

```
sem fork[5] = {1, 1, 1, 1, 1};
process Philosopher[i = 0 to 3] {
    while (true) {
        P(fork[i]); P(fork[i+1]); # get left fork then right
        eat;
        V(fork[i]); V(fork[i+1]);
        think;
    }
}
process Philosopher[4] {
    while (true) {
        P(fork[0]); P(fork[4]); # get right fork then left
        eat;
        V(fork[0]); V(fork[4]);
        think;
    }
}
```

- Circular waiting cannot occur

The Readers / Writers Problem

- Two kinds of processes – readers and writers share a database.
- Readers wait until no writers are inside.
- Many readers may read the database at the same time
- Writers wait until no readers or other writers are accessing.
- Write operations are executed in isolation.
- Two solutions for this problem:
 - Mutual exclusion solution
 - Condition synchronization solution (Passing the baton).

Readers / Writers as an Exclusion Problem

- Overconstrained solution: only one reader can read the database at the same time

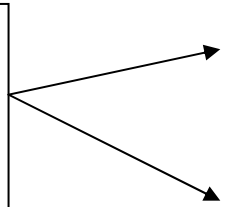
```
sem rw = 1;
process Reader[i = 1 to M] {
    while (true) {
        ...
        P(rw);    # grab exclusive access lock
        read the database;
        V(rw);    # release the lock
    }
}
process Writer[j = 1 to N] {
    while (true) {
        ...
        P(rw);    # grab exclusive access lock
        write the database;
        V(rw);    # release the lock
    }
}
```

Readers / Writers as an Exclusion Problem (Cont.)

- Outline of readers and writers solution

```
int nr = 0;          # number of active readers
sem rw = 1;          # lock for reader/writer exclusion
process Reader[i = 1 to M] {
    while (true) {
        ...
        < nr = nr+1;
          if (nr == 1) P(rw); # if first, get lock
        >
        read the database;
        < nr = nr-1;
          if (nr == 0) V(rw); # if last, release lock
        >
    }
}
process Writer[j = 1 to N] {
    while (true) {
        ...
        P(rw);
        write the database;
        V(rw);
    }
}
```

Critical
sections



Readers / Writers as an Exclusion Problem (Cont.)

```
int nr = 0;           # number of active readers
sem rw = 1;           # lock for access to the database
sem mutexR = 1;       # lock for reader access to nr
process Reader[i = 1 to m] {
    while (true) {
        ...
        P(mutexR);
        nr = nr+1;
        if (nr == 1) P(rw); # if first, get lock
        V(mutexR);
        read the database;
        P(mutexR);
        nr = nr-1;
        if (nr == 0) V(rw); # if last, release lock
        V(mutexR);
    }
}
process Writer[j = 1 to n] {
    while (true) {
        ...
        P(rw);
        write the database;
        V(rw);
    }
}
```

- Readers' preference: If some readers are accessing the DB, both another reader and a writer arrive, the new reader get preference over the writer.
- Continual stream of readers can permanently prevent writers from accessing the database

Readers / Writers as a Condition Synchronization Problem (Cont.)

```
int nr = 0, nw = 0;
## RW:  (nr == 0  $\vee$  nw == 0)  $\wedge$  nw <= 1
process Reader[i = 1 to m] {
  while (true) {
    ...
    <await (nw == 0) nr = nr+1;>
    read the database;
    <nr = nr-1;>
  }
}
process Writer[j = 1 to n] {
  while (true) {
    ...
    <await (nr == 0 and nw == 0) nw = nw+1;>
    write the database;
    <nw = nw-1;>
  }
}
```

Passing the Baton

- A process within the CS holding a baton passes it to another process
- Code for the signal procedure:

```
if (nw == 0 and dr > 0) {  
    dr = dr-1; V(r); # awaken a reader, or  
}  
elseif (nr == 0 and nw == 0 and dw > 0) {  
    dw = dw-1; V(w); # awaken a writer, or  
}  
else  
    V(e); # release the entry lock
```

Passing the Baton (Cont.)

```
int nr = 0,    ## RW: (nr == 0 or nw == 0) and nw <= 1
    nw = 0;
sem e = 1,    # controls entry to critical sections
    r = 0,    # used to delay readers
    w = 0;    # used to delay writers
              # at all times 0 <= (e+r+w) <= 1
int dr = 0,    # number of delayed readers
    dw = 0;    # number of delayed writers

process Reader[i = 1 to M] {
  while (true) {
    # <await (nw == 0) nr = nr+1;>
    P(e);
    if (nw > 0) { dr = dr+1; V(e); P(r); }
    nr = nr+1;
    SIGNAL;      # see text for details
    read the database;
    # <nr = nr-1;>
    P(e);
    nr = nr-1;
    SIGNAL;
  }
}

process Writer[j = 1 to N] {
  while (true) {
    # <await (nr == 0 and nw == 0) nw = nw+1;>
    P(e);
    if (nr > 0 or nw > 0) { dw = dw+1; V(e); P(w); }
    nw = nw+1;
    SIGNAL;
    write the database;
    # <nw = nw-1;>
    P(e);
    nw = nw-1;
    SIGNAL;
  }
}
```

e, r, w: split binary semaphores

Passing the Baton – Simplified Code (Cont.)

```

int nr = 0,    ## RW: (nr==0 or nw==0) and nw<=1
    nw = 0;
sem e = 1,    # controls entry to critical sections
r = 0,    # used to delay readers
w = 0;    # used to delay writers
           # at all times 0 <= (e+r+w) <= 1
int dr = 0,    # number of delayed readers
    dw = 0;    # number of delayed writers

process Reader[i = 1 to M] {
    while (true) {
        # <await (nw == 0) nr = nr+1;>
        P(e);
        if (nw > 0) { dr = dr+1; V(e); P(r); }
        nr = nr+1;
        if (dr > 0) { dr = dr-1; V(r); }
        else V(e);
        read the database;
        # <nr = nr-1;>
        P(e);
        nr = nr-1;
        if (nr == 0 and dw > 0) { dw = dw-1; V(w); }
        else V(e);
    }
}

process Writer[j = 1 to N] {
    while (true) {
        # <await (nr == 0 and nw == 0) nw = nw+1;>
        P(e);
        if (nr > 0 or nw > 0) { dw = dw+1; V(e); P(w); }
        nw = nw+1;
        V(e);
        write the database;
        # <nw = nw-1;>
        P(e);
        nw = nw-1;
        if (dr > 0) { dr = dr-1; V(r); }
        elseif (dw > 0) { dw = dw-1; V(w); }
        else V(e);
    }
}

```

Alternative Scheduling Policies

- Last solution gives readers preference.
- To give writers preference
 - New readers are delayed if a writer is waiting and,
 - Changing the reader's first statement
If ($nw > 0$ or $dw > 0$) { $dr = dr + 1$; $V(e)$; $P(r)$;}
– A delayed reader is awakened only if no writer is waiting.
 - Switching the order of the first two arms of the if statement in writers:
If ($dw > 0$) { $dw = dw - 1$; $V(w)$;}
Else if ($dr > 0$) { $dr = dr - 1$; $V(r)$;}
Else $V(e)$;

Fair Access

- Delay a new reader when a writer is waiting;
 - Delay a new writer when a reader is waiting;
 - Awaken one waiting writer (if any) when a reader finishes;
 - Awaken all waiting readers (if any) when a writer finishes; otherwise awaken one waiting writer (if any).
-
- Conclusion:
 - Passing the baton: control over the order in which processes use the resources
 - Cannot control the order in which processes delayed on the entry semaphore are awakened.

Resource Allocation and Scheduling

- Multiple processes are competing for use of units of some shared resource. Decide which processes acquire the resource units.
- Request for units:
 - Request (parameters):
<await (request can be satisfied) take units;>
- Release of reserved units
 - Release (parameters):
<return units; >

Resource Allocation and Scheduling (Cont.)

- Use of the “passing the baton” technique
- Request(parameters);
 P(e);
 if (request cannot be satisfied) DELAY;
 take units;
 SIGNAL;
- Release(parameters);
 P(e);
 return units;
 SIGNAL;

Shortest-Job-Next (SJN)

Allocation

- Several processes compete for use of a single shared resource.
- Requesting by calling *request (time, id)*, where time is an integer that specifies how long the process will use the resource and id is an integer that identifies the requesting process.
- Free resource: allocated to the delayed requesting process (if any) that has the minimum value for *time*
- Freeing resource by executing *release*
- SJN policy minimizes average job completion time
- However, a process can be delayed forever if there is a continual stream of requests specifying shorter usage times.
- Use of aging technique: a process that has been delayed a long time is given preference.

Shortest-Job-Next (SJN)

Allocation (Cont.)

- The coarse grained solution:
 - Bool free = true; #shared variable
 - request(time, id): < await (free) free=false; >
 - release(): < free = true; >

Shortest-Job-Next (SJN)

Allocation (Cont.)

- Use of the “passing the baton” technique
- Request (time, id):
P(e);
if (!free) DELAY;
free = false;
SIGNAL;
- Release():
P(e);
free=true;
SIGNAL;

SJN: Solution using semaphores

```
bool free = true;
sem e = 1, b[n] = ([n] 0); # for entry and delay
typedef Pairs = set of (int, int);
Pairs pairs =  $\emptyset$ ;
## SJN: pairs is an ordered set  $\wedge$  free  $\Rightarrow$  (pairs ==  $\emptyset$ )
request(time,id):
    P(e);
    if (!free) {
        insert (time,id) in pairs;
        V(e);          # release entry lock
        P(b[id]);      # wait to be awakened
    }
    free = false;
    V(e);              # optimized since free is false here
release():
    P(e);
    free = true;
    if (P !=  $\emptyset$ ) {
        remove first pair (time,id) from pairs;
        V(b[id]);      # pass baton to process id
    }
    else V(e);
```

Private semaphore:

Exactly one process
executes P operations on
that semaphore

Pairs:

Set of records (time, id),
ordered by the values of
time fields

SJN: Generalizing the Solution

- Resources have more than one unit
- Request and release operations have the *amount* parameter to indicate the number of units requested or returned.
- Solution:
 - Replace free by an integer *avail* that records the number of available units.
 - In request(), test if $amount \leq avail$,
 - If so, allocate *amount* units.
 - Otherwise, record how many units are required before delaying
 - In release()
 - increase *avail* by *amount*
 - determine whether the delayed process that has the minimum value for *time* can have its request satisfied
 - If so, awaken it
 - If not, execute V(e)
- What if more than one pending request can be satisfied?
 - The signaling protocol at the end of request should be the same as the one at the end of release

An example: The Sleeping Barber

- Customers who need a haircut enter the waiting room. If the room is full, the customer comes back later. If the barber is busy but there is a vacant chair the customer takes a seat. If the waiting room is empty and the barber is daydreaming the customer sits in the barber chair and wakes the barber.
- When the barber finishes cutting a customer's hair, the barber fetches another customer from the waiting room. If the waiting room is empty the barber daydreams.

The Sleeping Barber (Cont.)

```
Semaphore mutex=1;  
Semaphore customers=0, cutting=0, barber=0;  
int waiting=0, numChairs= ..;
```

```
process Customer[i=1..M]  
  while (true) {  
    P(mutex);  
    if (waiting<numChairs){  
      waiting++;  
      V(customers);  
      V(mutex);  
      P(barber);  
      V(cutting);  
    }  
    else V(mutex);  
  }  
end Customer;
```

```
process SleepingBarber  
  while (true) {  
    P(customers);  
    P(mutex);  
    waiting--;  
    V(barber);  
    V(mutex);  
    P(cutting);  
  }  
end SleepingBarber;
```

The Sleeping Barber (Cont.)

- This example illustrates the use of semaphores for both mutual exclusion and condition synchronization.
- The customers and barber use semaphores to control each other's flow through the barber shop.
- This interaction is an example of a client-server relationship.
- The customer and barber rendezvous to interact: each waits at a certain point for the other to arrive.

ΑΣΚΗΣΕΙΣ

ΑΣΚΗΣΗ 1

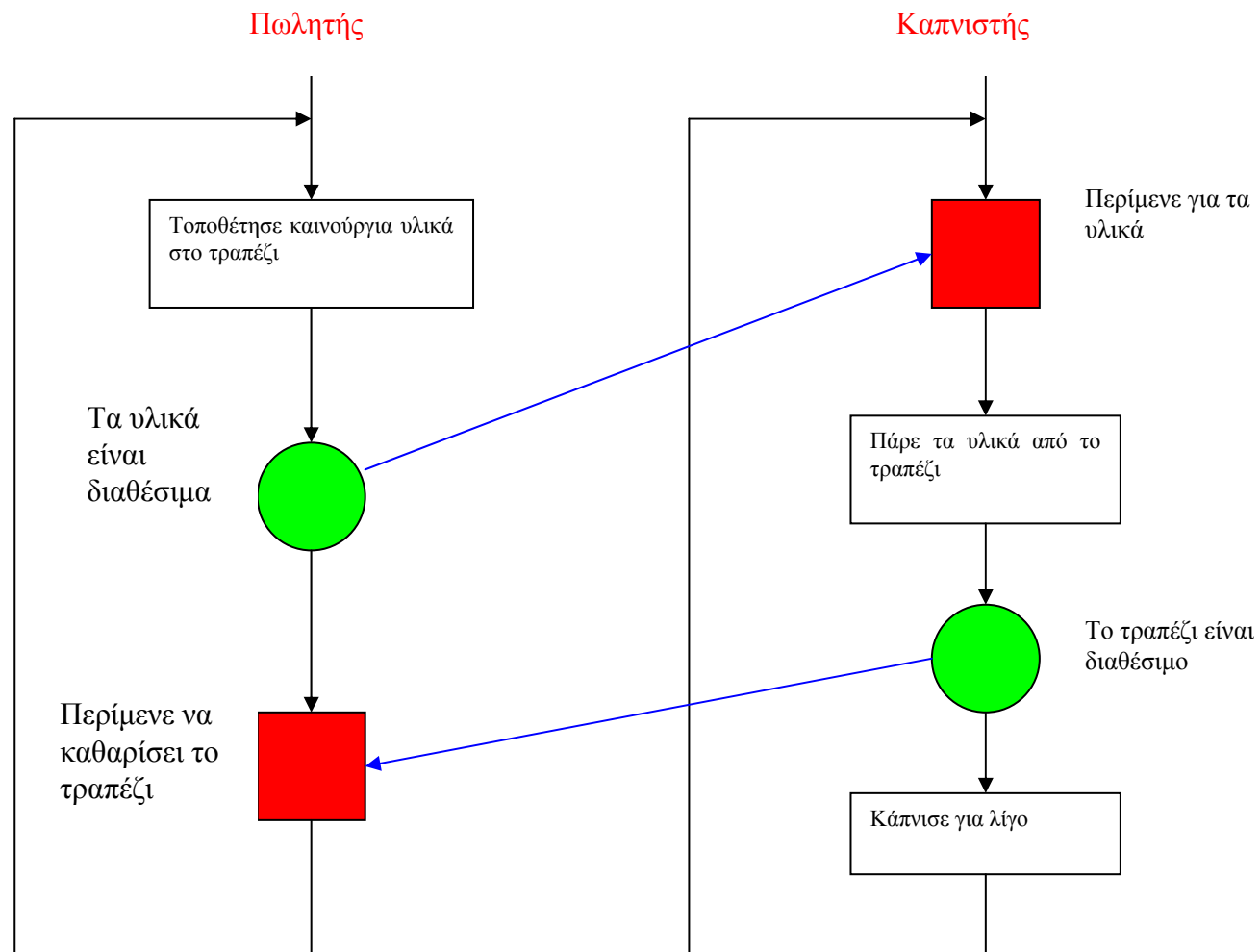
- Εστω ένα σύστημα στο οποίο ένα σύνολο από διεργασίες-πελάτες (clients) επικοινωνούν με μία διεργασία-εξυπηρετητή (server) μέσω μίας κοινής μεταβλητής X.
- Κάθε πελάτης μπορεί να παράγει επαναληπτικά αιτήσεις. Κάθε φορά που ένας πελάτης παράγει μία αίτηση πρέπει
 - να ενεργοποιηθεί ο εξυπηρετητής ώστε να εξυπηρετήσει την αίτηση, ενώ
 - 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.

<pre>Semaphore client_request = 1; Semaphore server_on = 0; int true = 1;</pre>	
<pre>/* Κώδικας εκτελούμενος επαναληπτικά από κάθε πελάτη */ ----- Client_i_ ----- while(true) { P(client_request); produce_request(); V(server_on); }</pre>	<pre>/* Κώδικας εκτελούμενος επαναληπτικά από τον εξυπηρετητή */ ----- Server ----- while(true) { P(server_on); process_request(); V (client_request); }</pre>

ΑΣΚΗΣΗ 2

- Τρεις μανιώδεις καπνιστές βρίσκονται στο ίδιο δωμάτιο μαζί με ένα πωλητή ειδών καπνιστού.
- Για να φτιάξει και να χρησιμοποιήσει τσιγάρα, κάθε καπνιστής χρειάζεται τρία συστατικά: καπνό, χαρτί και σπίρτα.
- Όλα αυτά τα παρέχει ο πωλητής σε αφθονία. Ένας καπνιστής έχει το δικό του καπνό, ένας δεύτερος το δικό του χαρτί και ο τρίτος τα δικά του σπίρτα.
- Η δράση ξεκινά όταν ο πωλητής τοποθετεί στο τραπέζι δύο από τα απαραίτητα υλικά, επιτρέποντας έτσι σε έναν από τους καπνιστές να καπνίσει.
- Όταν ο κατάλληλος καπνιστής τελειώσει το κάπνισμα, ο πωλητής θα πρέπει να αφυπνίζεται, ώστε να τοποθετεί στο τραπέζι δύο ακόμη από τα υλικά του – τυχαία – με αποτέλεσμα να επιτρέπει και σε άλλον καπνιστή να καπνίσει.
- Γράψτε κατάλληλα προγράμματα για τους καπνιστές και τους πωλητές, με σκοπό την επίλυση του συγκεκριμένου προβλήματος. Η λύση σας θα πρέπει να βασίζεται στη χρήση σημαφόρων.

ΑΣΚΗΣΗ 2



ΑΣΚΗΣΗ 2

```
S0=0, S1=0, S2=0, table=0: semaphore; /* Αρχικοποίηση στο 0 */
Πωλητής
while (1) {
    j=random(2); // Δες ποιόν καπνιστή θα εξυπηρετήσεις. Το j θα
είναι 0,1 ή 2 //
    Τοποθέτησε τα κατάλληλα υλικά στο τραπέζι;
    // Αν j=0, τοποθέτησε χαρτί και σπίρτα, αν j=1, τοποθέτησε καπνό και χαρτί
και αν j=2, τοποθέτησε σπίρτα και καπνό //
    if (j== 0) V(S0)
    else if (j==1) V(S1)
    else V(S2);          // Ειδοποίησε τον καπνιστή j //
    P(table); // Περίμενε μήπως ο καπνιστής δεν έχει αδειάσει το
τραπέζι //
}
```

ΑΣΚΗΣΗ 2

Καπνιστής j

while (1) {

 P(Sj); // Περίμενε σήμα από τον πωλητή //

 Πάρε τα υλικά από το τραπέζι;

 V(table); // Ειδοποίησε τον πωλητή ότι το τραπέζι άδειασε //

 delay() // Κάπνισε για λίγο //

}

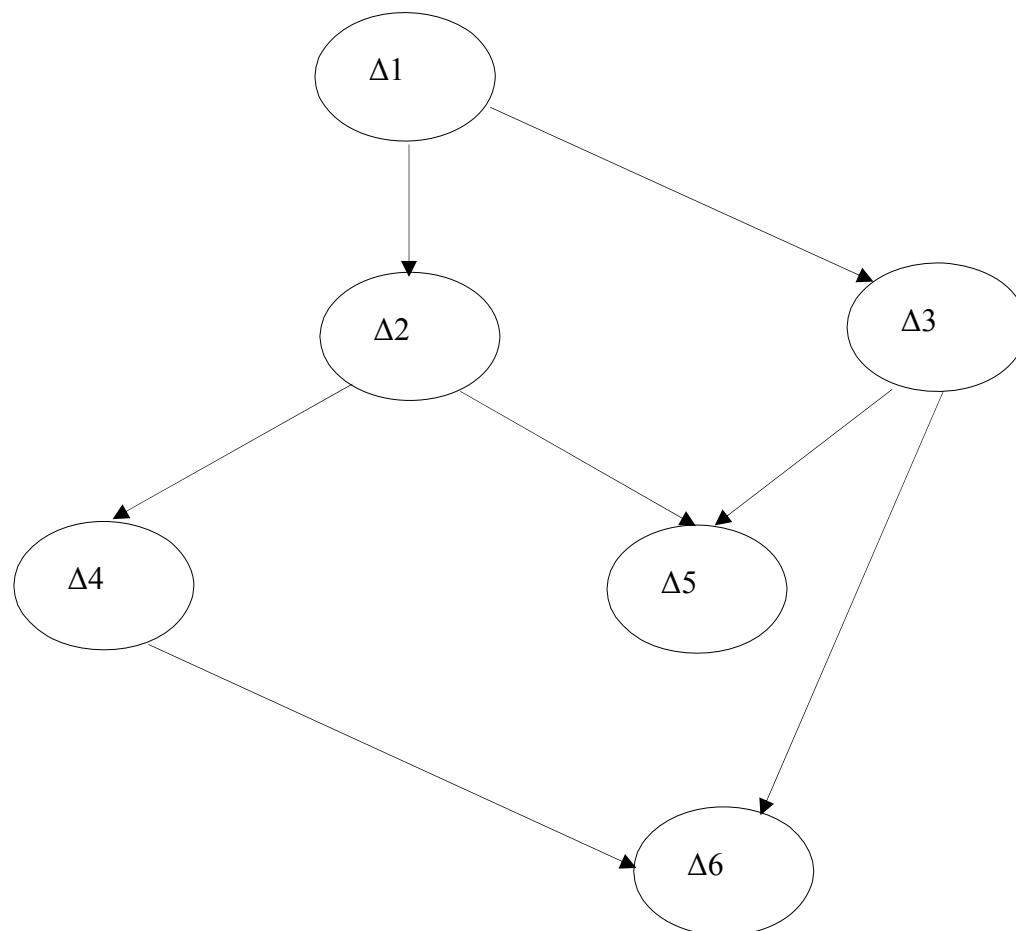
ΑΣΚΗΣΗ 3

<pre>#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;</pre>	<p><i>Για να διαπεράσουν μία χαράδρα οι 'μπαμπούνοι' έχουν δέσει ένα σχοινί από τη μία άκρη έως την άλλη και το χρησιμοποιούν όποτε το χρειάζονται. Προφανώς την ώρα που έχει μπει ένας μπαμπούνος να το χρησιμοποιήσει για να περάσει προς τη μία κατεύθυνση δεν πρέπει να επιτραπεί η είσοδος σε κανέναν από την αντίθετη κατεύθυνση</i></p>
<pre>void east(void) { while (TRUE) { P(east); east_count++; if (east_count==1) P(mutex); V(east); move_east(); P(east); east_count--; if (east_count==0) V(mutex); V(east); } }</pre>	<pre>void west(void) { while (TRUE) { P(west); west_count++; if (west_count==1) P(mutex); V(west); move_west(); P(west); west_count--; if (west_count==0) V(mutex); V(west); } }</pre>

ΑΣΚΗΣΗ 4

- Θεωρήστε ότι πρέπει να συγχρονίσετε την εκτέλεση των διαδικασιών $\Delta 1$, $\Delta 2$, $\Delta 3$, $\Delta 4$, $\Delta 5$ και $\Delta 6$ σύμφωνα με τους παρακάτω περιορισμούς:
 - Η $\Delta 1$ εκτελείται πριν από τις $\Delta 2$ και $\Delta 3$.
 - Η $\Delta 2$ εκτελείται πριν από τις $\Delta 4$ και $\Delta 5$.
 - Η $\Delta 3$ εκτελείται πριν από την $\Delta 5$.
 - Η $\Delta 6$ εκτελείται μετά από τις $\Delta 3$ και $\Delta 4$.
- 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 $\Delta 1, \dots, \Delta 6$ 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 $\Delta 1, \dots, \Delta 6$ σύμφωνα με τους παραπάνω περιορισμούς με χρήση σημαφόρων.

ΑΣΚΗΣΗ 4



ΑΣΚΗΣΗ 4

- *A' Εναλλακτική Λύση:* Χρήση σηματοφόρων Σ_{ij} για τον έλεγχο των προτεραιοτήτων $\Delta_i \rightarrow \Delta_j$ (χρησιμοποιούμε μία σηματοφόρο Σ_{ij} – με αρχική τιμή ‘0’ – για τον έλεγχο κάθε σχέσης προτεραιότητας $\Delta_i \rightarrow \Delta_j$):

var $\Sigma_{12}, \Sigma_{13}, \Sigma_{24}, \Sigma_{25}, \Sigma_{35}, \Sigma_{36}, \Sigma_{46}$: **semaphores**;

$\Sigma_{12} := \Sigma_{13} := \Sigma_{24} := \Sigma_{25} := \Sigma_{35} := \Sigma_{36} := \Sigma_{46} := 0$;

cobegin

begin Δ_1 ; V(Σ_{12}); V(Σ_{13}); **end**;

begin P(Σ_{12}); Δ_2 ; V(Σ_{24}); V(Σ_{25}); **end**;

begin P(Σ_{13}); Δ_3 ; V(Σ_{35}); V(Σ_{36}); **end**;

begin P(Σ_{24}); Δ_4 ; V(Σ_{46}); **end**;

begin P(Σ_{25}); P(Σ_{35}); Δ_5 ; **end**;

begin P(Σ_{46}); P(Σ_{36}); Δ_6 ; **end**;

coend

ΑΣΚΗΣΗ 4

- **B' Εναλλακτική Λύση:** Χρήση σημαφόρων Σ_i για τον έλεγχο των μη-αρχικών διαδικασιών Δ_i
- Χρησιμοποιούμε μία σημαφόρο Σ_i για τον έλεγχο κάθε μη-αρχικής διαδικασίας, δηλαδή για κάθε διαδικασία Δ_i εκτός της Δ_1 , και της δίνουμε αρχική τιμή ανάλογη με τον αριθμό των διαδικασιών Δ_j τις οποίες πρέπει να περιμένει πριν αρχίσει την εκτέλεσή της):

var $\Sigma_2, \Sigma_3, \Sigma_4, \Sigma_5, \Sigma_6$: **semaphores**;

$\Sigma_2 := 0, \Sigma_3 := 0, \Sigma_4 := 0, \Sigma_5 := -1, \Sigma_6 := -1;$

cobegin

begin $\Delta_1; V(\Sigma_2); V(\Sigma_3);$ **end**;

begin $P(\Sigma_2); \Delta_2; V(\Sigma_4); V(\Sigma_5);$ **end**;

begin $P(\Sigma_3); \Delta_3; V(\Sigma_5); V(\Sigma_6);$ **end**;

begin $P(\Sigma_4); \Delta_4; V(\Sigma_6);$ **end**;

begin $P(\Sigma_5); \Delta_5;$ **end**;

begin $P(\Sigma_6); \Delta_6;$ **end**;

coend

ΑΣΚΗΣΗ 5

- Το τετράγωνο ενός ακεραίου αριθμού N μπορεί να υπολογιστεί με τον ακόλουθο τύπο (αθροίζοντας δηλαδή τους πρώτους N περιττούς ακεραίους) :
- $$N^2 = \sum_{k=0}^{N-1} 2k + 1$$
- Θεωρείστε ότι για τον παραπάνω υπολογισμό συνεργάζονται 3 διεργασίες P,Q,R εκτελώντας η κάθε μία τον παρακάτω κώδικα (π.χ. για N=5):

shared int N=5; shared int Sqr=0;		
<u>Process P</u>	<u>Process Q</u>	<u>Process R</u>
loopP: if (N==0) goto endP; N = N - 1; goto loopP; endP:print(Sqr);	loopQ: Sqr = Sqr+ 2*N; goto loopQ;	loopR: Sqr = Sqr + 1; goto loopQ;

ΑΣΚΗΣΗ 5

shared semaphore P=1; shared semaphore Q=0;shared semaphore R=0;shared int N=5; shared int Sqr=0;		
<u>Process P</u> loopP: if (N==0) goto endP; P(P); N = N - 1; V(Q); goto loopP; endP: P(P); print(Sqr);	<u>ProcessQ</u> loopQ: P(Q); Sqr=Sqr+2*N; V(R); goto loopQ;	<u>ProcessR</u> loopR: P(R); Sqr=Sqr+1; V(P); goto loopR;

ΑΣΚΗΣΗ 6

- Θεωρείστε τον ακόλουθο κώδικα για δύο διαδικασίες P1 και P2 που κάνουν προσπέλαση σε μια κοινή (διαμοιραζόμενη) ακέραια μεταβλητή A.
- Η διαδικασία P1 χρησιμοποιεί την τοπική ακέραια μεταβλητή B, ενώ η διαδικασία P2 χρησιμοποιεί την τοπική ακέραια μεταβλητή C.
- Η μεταβλητή A αρχικά τίθεται ίση με 15.

cobegin

P1: begin

B = A * 2;

A = B;

end

P2: begin

C = A + 1;

A = C;

end

coend

Θέλουμε να χρησιμοποιήσουμε μια δυαδική σημαφόρο S ώστε να εξασφαλίσουμε ότι μετά την ολοκλήρωση της εκτέλεσης των δύο διαδικασιών η μεταβλητή A θα λάβει τις τιμές 31 ή 32. Ποιο/ποια από τα παρακάτω προγράμματα εξασφαλίζουν την απαίτηση αυτή;

ΑΣΚΗΣΗ 6

```
(A) var S: semaphore;  
    S = 1;  
cobegin  
P1: begin  
P(S);  
B = A * 2;  
A = B;  
    end  
P2: begin  
C = A + 1;  
A = C;  
V(S)  
    end  
coend
```

```
(B) var S: semaphore;  
    S = 0;  
cobegin  
P1: begin  
B = A * 2;  
A = B;  
V(S)  
    end  
P2: begin  
P(S);  
C = A + 1;  
A = C;  
    end  
coend
```

```
(Γ) var S: semaphore;  
    S = 1;  
cobegin  
P1: begin  
P(S);  
B = A * 2;  
A = B;  
V(S)  
    end  
P2: begin  
P(S);  
C = A + 1;  
A = C;  
V(S);  
    end  
coend
```

```
(Δ) var S: semaphore;  
    S = 0;  
cobegin  
P1: begin  
P(S);  
B = A * 2;  
A = B;  
V(S)  
    end  
P2: begin  
P(S);  
C = A + 1;  
A = C;  
V(S);  
    end  
coend
```

ΑΣΚΗΣΗ 7

- Θέλουμε να υλοποιήσουμε ένα τραπεζικό *σύστημα επεξεργασίας συναλλαγών* (transaction processing system) με χρήση σηματοφόρων.
- Το σύστημα αποτελείται από τρεις ταυτόχρονες διαδικασίες, μία για *ανάγνωση*, μία για *επεξεργασία* και μία για *απάντηση*.
- Η διαδικασία για *ανάγνωση* διαβάζει τις εισερχόμενες αιτήσεις και τις τοποθετεί σε *απομονωτές (buffers) εισόδου*.
- Η διαδικασία για *επεξεργασία* διαβάζει μια αίτηση από τους απομονωτές εισόδου, εκτελεί την απαιτούμενη επεξεργασία και αποθηκεύει τα αποτελέσματα στους *απομονωτές εξόδου*.
- Τέλος, η διαδικασία για *απάντηση* παίρνει τα αποτελέσματα από τους απομονωτές εξόδου και τα προωθεί στον παραλήπτη.
- Έχουμε N απομονωτές εισόδου που ο καθένας μπορεί να περιέχει μία αίτηση και M απομονωτές εξόδου που ο καθένας μπορεί να περιέχει τα αποτελέσματα της επεξεργασίας μιας αίτησης.
- Θεωρούμε αμοιβαίο αποκλεισμό των διαδικασιών *ανάγνωσης* και *επεξεργασίας* για την πρόσβαση στους απομονωτές εισόδου και αμοιβαίο αποκλεισμό των διαδικασιών *επεξεργασίας* και *απάντησης* για την πρόσβαση στους απομονωτές εξόδου.
- Η προσπάθεια της διαδικασίας *ανάγνωσης* να τοποθετήσει μια αίτηση στους απομονωτές εισόδου όταν είναι όλοι γεμάτοι, σημαίνει καθυστέρηση της διαδικασίας μέχρι να ελευθερωθεί κάποιος.
- Η προσπάθεια της διαδικασίας *επεξεργασίας* να διαβάσει μια αίτηση από κάποιο απομονωτή εισόδου όταν όλοι είναι άδειοι, σημαίνει καθυστέρηση της διαδικασίας μέχρι να τοποθετηθεί κάποια αίτηση σε έναν απομονωτή εισόδου.
- Η προσπάθεια της διαδικασίας *επεξεργασίας* να αποθηκεύσει τα αποτελέσματα επεξεργασίας μιας αίτησης σε απομονωτή εξόδου όταν όλοι είναι γεμάτοι, σημαίνει καθυστέρηση της διαδικασίας μέχρι να αδειάσει ένας απομονωτής εξόδου.
- Η προσπάθεια της διαδικασίας *απάντησης* να πάρει αποτελέσματα από κάποιον απομονωτή εξόδου όταν είναι όλοι άδειοι, σημαίνει καθυστέρηση της διαδικασίας μέχρι να αποθηκευτούν τα αποτελέσματα επεξεργασίας κάποιας αίτησης σε απομονωτή εξόδου.

ΑΣΚΗΣΗ 7

```
var mutexin, mutexout, fullin, emptyin, fullout, emptyout: semaphore;
mutexin = mutexout = 1;
fullin = fullout = 0;
emptyin = N;
emptyout = M;
cobegin
    Read: repeat
        Διάβασε μια αίτηση που φτάνει στο σύστημα;
        P(emptyin); P(mutexin);
        Τοποθέτησε την αίτηση σε έναν απομονωτή εισόδου;
        V(mutexin); V(fullin);
        until false;
    Process: repeat
        P(fullin); P(mutexin);
        Διάβασε μια αίτηση από έναν απομονωτή εισόδου;
        V(mutexin); V(emptyin);
        Επεξεργάσου την αίτηση;
        P(emptyout); P(mutexout);
        Αποθήκευσε τα αποτελέσματα σε έναν απομονωτή εξόδου;
        V(mutexout); V(fullout);
        until false;
    Answer: repeat
        P(fullout); P(mutexout);
        Πάρε αποτελέσματα από έναν απομονωτή εξόδου;
        V(mutexout); V(emptyout);
        Προώθησε τα αποτελέσματα στον παραλήπτη;
        until false;
coend.
```

ΑΣΚΗΣΗ 8

- Ένα βιντεοκλάμπ έχει N πελάτες και έναν υπάλληλο.
- Στο βιντεοκλάμπ λειτουργεί μια αυτόματη μηχανή την οποία και χρησιμοποιούν οι πελάτες για να επιστρέψουν τα DVD που έχουν δανειστεί.
- Η αυτόματη μηχανή (θεωρήστε ότι στο βιντεοκλάμπ υπάρχει μία μόνο αυτόματη μηχανή που αρχικά είναι άδεια) έχει περιορισμένη χωρητικότητα M (όπου $M < N$), δηλαδή «γεμίζει» με DVD όταν επιστραφούν από τους πελάτες του βιντεοκλάμπ M σε πλήθος DVD.
- Όταν η μηχανή «γεμίσει» τότε ένα ηχητικό σήμα ειδοποιεί τον υπάλληλο που κοιμάται για να την «αδειάσει» από τα DVD που επέστρεψαν οι πελάτες.
- Κατά το διάστημα που ο υπάλληλος αδειάζει τη μηχανή, η μηχανή τίθεται εκτός λειτουργίας (δηλ. δεν μπορεί να τη χρησιμοποιήσει ένας πελάτης να επιστρέψει DVD).
- Ο υπάλληλος ενεργοποιεί πάλι τη μηχανή (για να είναι διαθέσιμη στους πελάτες) όταν την αδειάσει από όλα τα DVD. Υποθέστε ότι ένας πελάτης μπορεί να επιστρέφει κάθε φορά μόνο ένα DVD στη μηχανή.
- Γράψτε κατάλληλο κώδικα για τους πελάτες και τον υπάλληλο του βιντεοκλάμπ, με σκοπό την επίλυση του συγκεκριμένου προβλήματος. Η λύση σας θα πρέπει να βασίζεται στη χρήση σηματοφόρων.

ΑΣΚΗΣΗ 8

```
var full: semaphore;    machine: semaphore;    no_DVD: shared integer;  
full := 0;  
machine := 1;  
no_DVD := 0;
```

Υπάλληλος

```
while(true) do {  
    P (full);  
    no_DVD := 0;  
    V(machine); }
```

Πελάτης i

```
while(true) do {  
    P(machine);  
    no_DVD := no_DVD +1 ;  
    if (no_DVD == M)  
        V(full);  
    else  
        V(machine);}
```

ΑΣΚΗΣΗ 9

- Θεωρήστε το εξής πρόβλημα:
- Οι οκτώ αθλητές μιας ομάδας *windsurf* πρέπει να προπονηθούν για τη συμμετοχή τους σε ένα σημαντικό τουρνουά.
- Για να προετοιμαστούν κατάλληλα, τους έχει επιβληθεί εντατικό πρόγραμμα προπονήσεων, το οποίο σε καθημερινή βάση περιλαμβάνει τα ακόλουθα στάδια:
 - κατ' ιδίαν συνεργασία με έναν προπονητή για την εφαρμογή κάποιας συγκεκριμένης τεχνικής πλεύσης,
 - προπόνηση (κατά την οποία επιβλέπεται επίσης από κάποιον προπονητή),
 - ξεκούραση στο κυλικείο – και στη συνέχεια επανάληψη της ίδιας διαδικασίας (στάδια (α), (β) και (γ)) με τη συνεργασία και προπόνηση πάνω σε μια άλλη τεχνική κ.ο.κ.
- Η ομάδα είναι ερασιτεχνική με αποτέλεσμα για την προπόνηση (στάδιο (β)) να μην προβλέπεται ούτε ατομικός προπονητής, ούτε προσωπικός εξοπλισμός προπόνησης για κάθε αθλητή.
- Πιο συγκεκριμένα, έχουν διατεθεί στην ομάδα συνολικά δυο προπονητές και τέσσερις πλήρεις εξοπλισμοί προπόνησης, τους οποίους οι αθλητές μοιράζονται μεταξύ τους.
- Για να μπορέσει ένας αθλητής να προπονηθεί (στάδιο (β)) χρειάζεται έναν από τους τέσσερις διαθέσιμους εξοπλισμούς.
- Επιπλέον, κατά τη διάρκεια της προπόνησης, απαραίτητη είναι και η επίβλεψη των αθλητών από έναν προπονητή.
- Για όλους τους αθλητές που βρίσκονται στο στάδιο της προπόνησης, μόνο ένας εκ των δυο προπονητών είναι διαθέσιμος, τον οποίο και διαμοιράζονται.
- Φυσικά, αν κανένας αθλητής δεν βρίσκεται στο στάδιο αυτό (στάδιο (β) – προπόνηση) και οι δυο προπονητές είναι διαθέσιμοι για κατ' ιδίαν συνεργασία (στάδιο (α) – το οποίο αντίθετα είναι ατομικό / δηλαδή ο κάθε προπονητής συνεργάζεται με έναν μόνο αθλητή κατ' ιδίαν κάθε φορά).
- Οι αθλητές ξεκουράζονται στο κυλικείο (στάδιο (γ)), το οποίο όμως μπορεί να εξυπηρετήσει το πολύ πέντε αθλητές ταυτόχρονα.

ΑΣΚΗΣΗ 9

Κοινές μεταβλητές & σημαφόροι

semaphore trainer=2, equipment=4, service=5;

semaphore mutex=1;

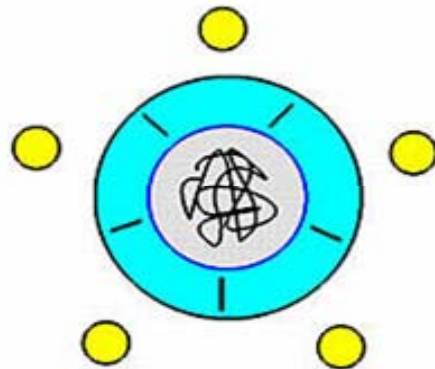
int training_counter=0;

Διεργασία-Αθλητής

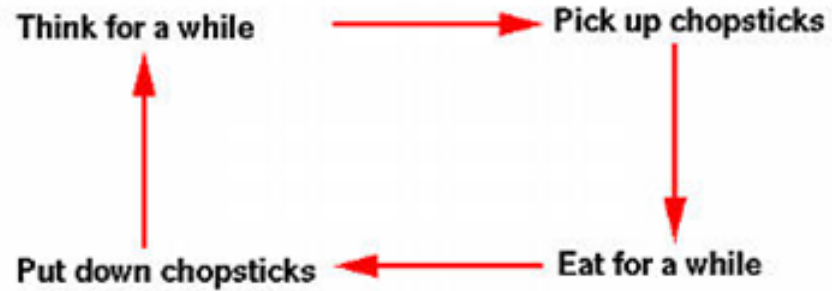
```
while(TRUE){  
    P(trainer);  
    Collaborate_with_trainer();  
    V(trainer);  
    P(equipment);  
    P(mutex);  
    training_counter=training_counter+1;  
    if (training_counter==1) P(trainer);  
    V(mutex);  
    Training();  
    P(mutex);  
    training_counter=training_counter-1;  
    if (training_counter==0) V(trainer);  
    V(mutex);  
    V(equipment);  
    P(service);  
    Rest();  
    V(service);  
}
```

ΑΣΚΗΣΗ 10

Το πρόβλημα των φιλοσόφων



(κινέζικη παραλλαγή)



ΑΣΚΗΣΗ 10

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 */
    }
}
```

ΑΣΚΗΣΗ 10

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]);
}
```