Συγχρονισμός

Λειτουργικά Συστήματα
6ο εξάμηνο ΣΗΜΜΥ
ακ. έτος 2022-2023

`http://www.cslab.ece.ntua.gr/courses/os`

Εργαστήριο Υπολογιστικών Συστημάτων
ΕΜΠ

Απρίλιος 2023

Περιεχόμενα

- Το πρόβλημα του συγχρονισμού
  - Πότε/γιατί χρειάζεται δύο ή περισσότερες διεργασίες να συγχρονιστούν;
- Μηχανισμοί και προγραμματιστικές δομές συγχρονισμού
  - Κλειδώματα (locks)
  - Σημαφόροι (semaphores)
  - Παρακολουθητές (monitors)
- Προβλήματα συγχρονισμού (και λύσεις)
  - Σειριοποίηση (ordering)
  - Παραγωγός - Καταναλωτής
  - Αναγνώστες - Εγγραφείς
  - Συνδαιτημόνες συγγραφείς (dining philosophers)
- Επιπλέον ζητήματα
  - Αλληλεπίδραση με το υλικό
  - Στρατηγικές συγχρονισμού
  - Αδιέξοδα

Μοιραζόμενοι πόροι μεταξύ διεργασιών

- Μοιραζόμενοι πόροι
  - Συνεργασία διεργασιών (χώρος χρήστη)
    π.χ. πρόβλημα παραγωγού-καταναλωτή (βλέπε συνέχεια)
  - Πρόσβαση σε πόρους του ΛΣ (χώρος πυρήνα)
    πχ πρόσβαση στο σύστημα αρχείων
- Πρόσβαση διεργασιών σε μοιραζόμενους πόρους/δεδομένα
  - Πιθανή δημιουργία ασυνεπειών

Σημείωση: ο όρος διεργασία περιλαμβάνει και τα νήματα

Παράδειγμα: Παραγωγός-Καταναλωτής

- Η διεργασία παραγωγός προσθέτει σε έναν buffer ένα αντικείμενο
- Η διεργασία καταναλωτής αφαιρεί από τον buffer ένα αντικείμενο
- Ο buffer έχει πεπερασμένο μέγεθος N
- Ο παραγωγός περιμένει όταν ο buffer είναι γεμάτος
- Ο καταναλωτής περιμένει όταν ο buffer είναι άδειος
- Μπορεί να υπάρχουν πολλοί παραγωγοί και καταναλωτές

Παράδειγμα: Παραγωγός-Καταναλωτής

- δομή: Κυκλικός πίνακας (μεγέθους N)
- **Καταναλωτής:** αφαιρεί δεδομένα από το "τέλος"

```
#define NEXT(x) ((x + 1) % N)
item_t buffer[N]; int in=0, out=0, count=0;
```

Παράδειγμα: Παραγωγός-Καταναλωτής

- δομή: Κυκλικός πίνακας (μεγέθους N)
- **Καταναλωτής:** αφαιρεί δεδομένα από το "τέλος"

```
#define NEXT(x) ((x + 1) % N)
item_t buffer[N]; int in=0, out=0, count=0;
```

```
void enqueue(item_t item){
    item_t item;


    buffer[in] = item;
    in = NEXT(in);
    count++;
}
```

```
item_t dequeue(void){
    item_t item;


    item = buffer[out];
    out = NEXT(out);
    count--;
    return item
}
```

Παράδειγμα: Παραγωγός-Καταναλωτής

- δομή: Κυκλικός πίνακας (μεγέθους N)
- **Καταναλωτής:** αφαιρεί δεδομένα από το "τέλος"

```
#define NEXT(x) ((x + 1) % N)
item_t buffer[N]; int in=0, out=0, count=0;
```

```
void enqueue(item_t item){
    item_t item;
    while (count == N)
        ; // wait
    buffer[in] = item;
    in = NEXT(in);
    count++;
}
```

```
item_t dequeue(void){
    item_t item;
    while (count == 0)
        ; // wait
    item = buffer[out];
    out = NEXT(out);
    count--;
    return item
}
```

Παράδειγμα: Παραγωγός-Καταναλωτής

- δομή: Κυκλικός πίνακας (μεγέθους N)
- **Καταναλωτής:** αφαιρεί δεδομένα από το "τέλος"

```
#define NEXT(x) ((x + 1) % N)
item_t buffer[N]; int in=0, out=0, count=0;
```

```
void enqueue(item_t item){
    item_t item;
    while (count == N)
        ; // wait
    buffer[in] = item;
    in = NEXT(in);
    count++;
}
```

```
item_t dequeue(void){
    item_t item;
    while (count == 0)
        ; // wait
    item = buffer[out];
    out = NEXT(out);
    count--;
    return item
}
```

Κατάσταση συναγωνισμού

race condition

t	`count++`	`count--`
1	LOAD r, count	
2	ADD 1, r	
3		LOAD r, count
4		SUB 1, r
5	STORE r, count	
6		STORE r, count

Κατάσταση συναγωνισμού

race condition

t	`count++`	`count--`
1	LOAD r, count	
2	ADD 1, r	
3		LOAD r, count
4		SUB 1, r
5	STORE r, count	
6		STORE r, count

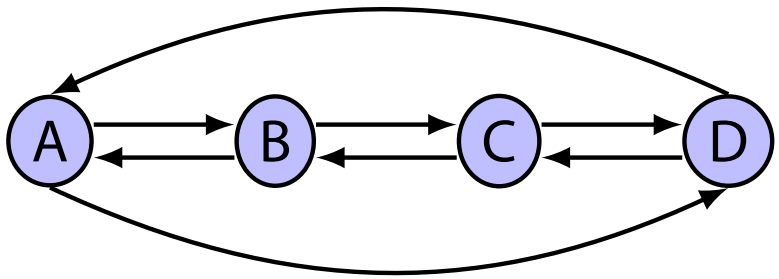
Αν για $\boldsymbol{t} = 0$, ***count*** $= \boldsymbol{c}$, τότε για $\boldsymbol{t} = 7$, ***count*** $= \boldsymbol{c} - 1$

$\Rightarrow$ Ασυνέπεια στη δομή του κυκλικού πίνακα

Κατάσταση συναγωνισμού: ***κατάσταση κατά την οποία το αποτέλεσμα ενός υπολογισμού εξαρτάται από την σειρά που πραγματοποιούνται οι προσπελάσεις.***

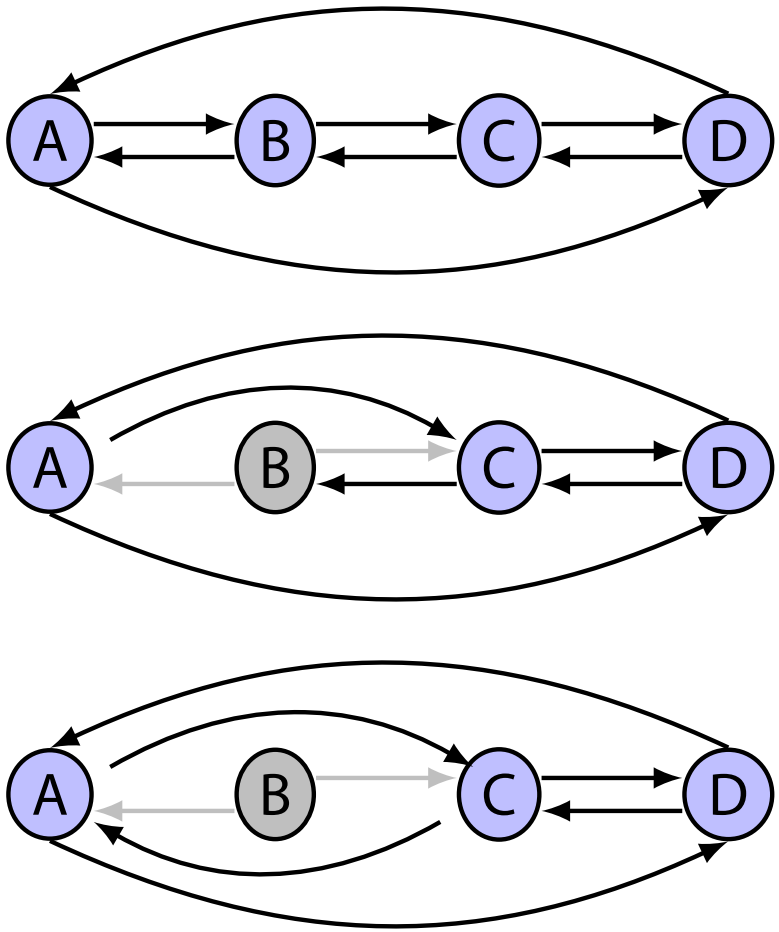
Παράδειγμα: Διπλά συνδεδεμένη λίστα

```
struct list_head {
    struct list_head *next;
    struct list_head *prev;
};

list_del(prev, next)
{
    prev->next = next;
    next->prev = prev;
}
```



Παράδειγμα: Διπλά συνδεδεμένη λίστα

```
struct list_head {
    struct list_head *next;
    struct list_head *prev;
};

list_del(prev, next)
{
    prev->next = next;
    next->prev = prev;
}
```

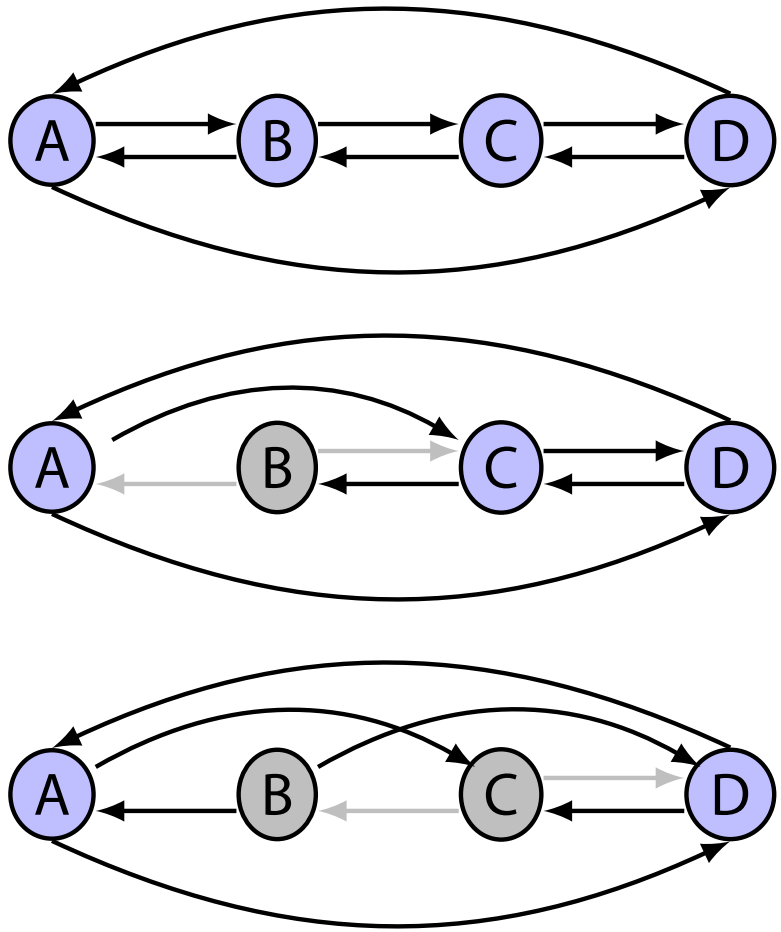


Διαγραφή B (1 διεργασία)

1. `t0: prev->next = next;`
2. `t0: next->prev = prev;`
3. OK!

Παράδειγμα: Διπλά συνδεδεμένη λίστα

```
struct list_head {
    struct list_head *next;
    struct list_head *prev;
};

list_del(prev, next)
{
    prev->next = next;
    next->prev = prev;
}
```

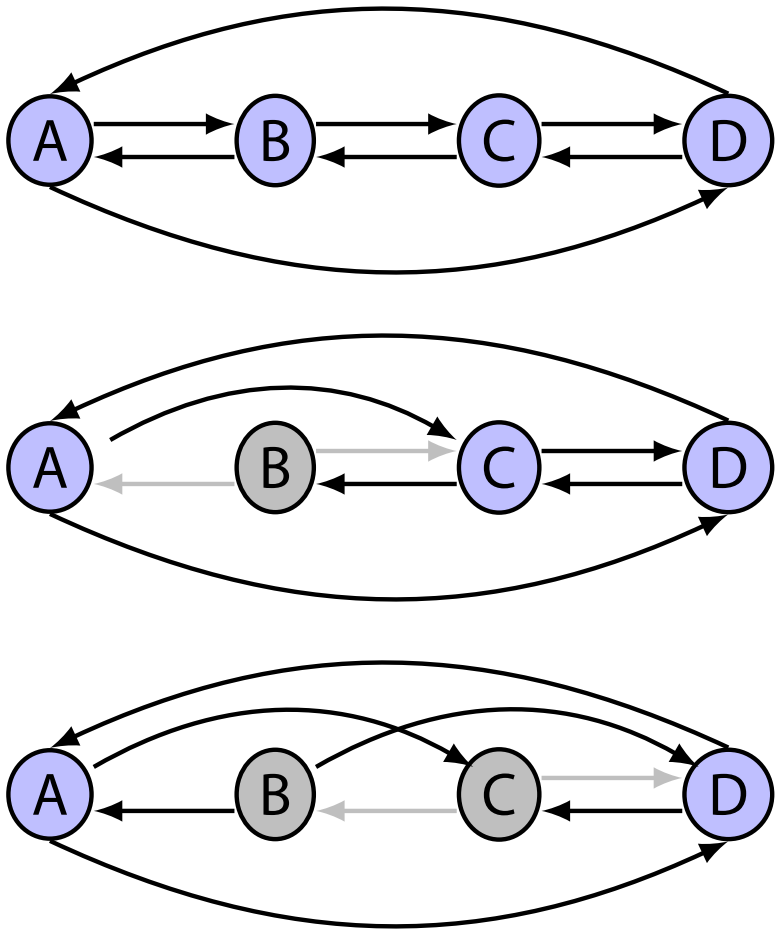


Διαγραφή B,C (2 διεργασίες)

1. `t0: prev->next = next;`
2. `t1: prev->next = next;`

Παράδειγμα: Διπλά συνδεδεμένη λίστα

```
struct list_head {
    struct list_head *next;
    struct list_head *prev;
};

list_del(prev, next)
{
    prev->next = next;
    next->prev = prev;
}
```



Διαγραφή B,C (2 διεργασίες)

1. `t0: prev->next = next;`
2. `t1: prev->next = next;`
3. ΑΣΥΝΕΠΕΙΑ

Πρόσβαση σε μοιραζόμενες πόρους/δεδομένα

- Πιθανή δημιουργία ασυνεπειών
  - Κοινοί πόροι / δεδομένα
  - Υπάρχει ενδεχόμενο εγγραφών
  - Η σημασιολογία του αλγορίθμου απαιτεί την **ατομική** εκτέλεση > 1 στοιχειωδών λειτουργιών

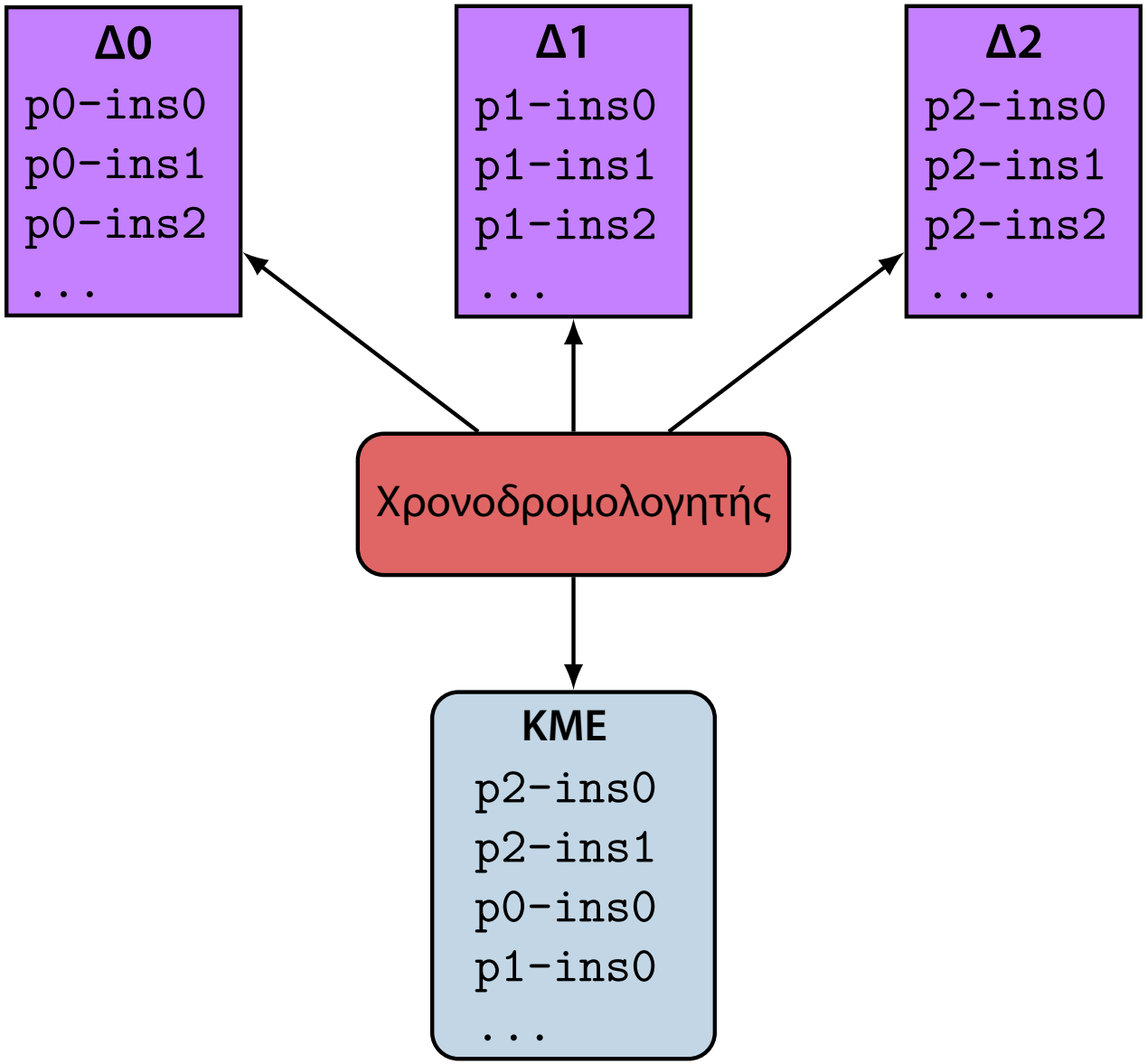
- **Διακοπτή χρονοδρομολόγηση (preemptive scheduling)**
  Ο χρονοδρομολογητής μπορεί να διακόψει την τρέχουσα διεργασία σε οποιοδήποτε σημείο.

- **Πολυεπεξεργασία (συστήματα μοιραζόμενης μνήμης)**
  Ταυτόχρονη πρόσβαση σε κοινές δομές δεδομένων.

- Ανάγκη για συγχρονισμό διεργασιών
  - Χώρος χρήστη
  - Χώρος πυρήνα

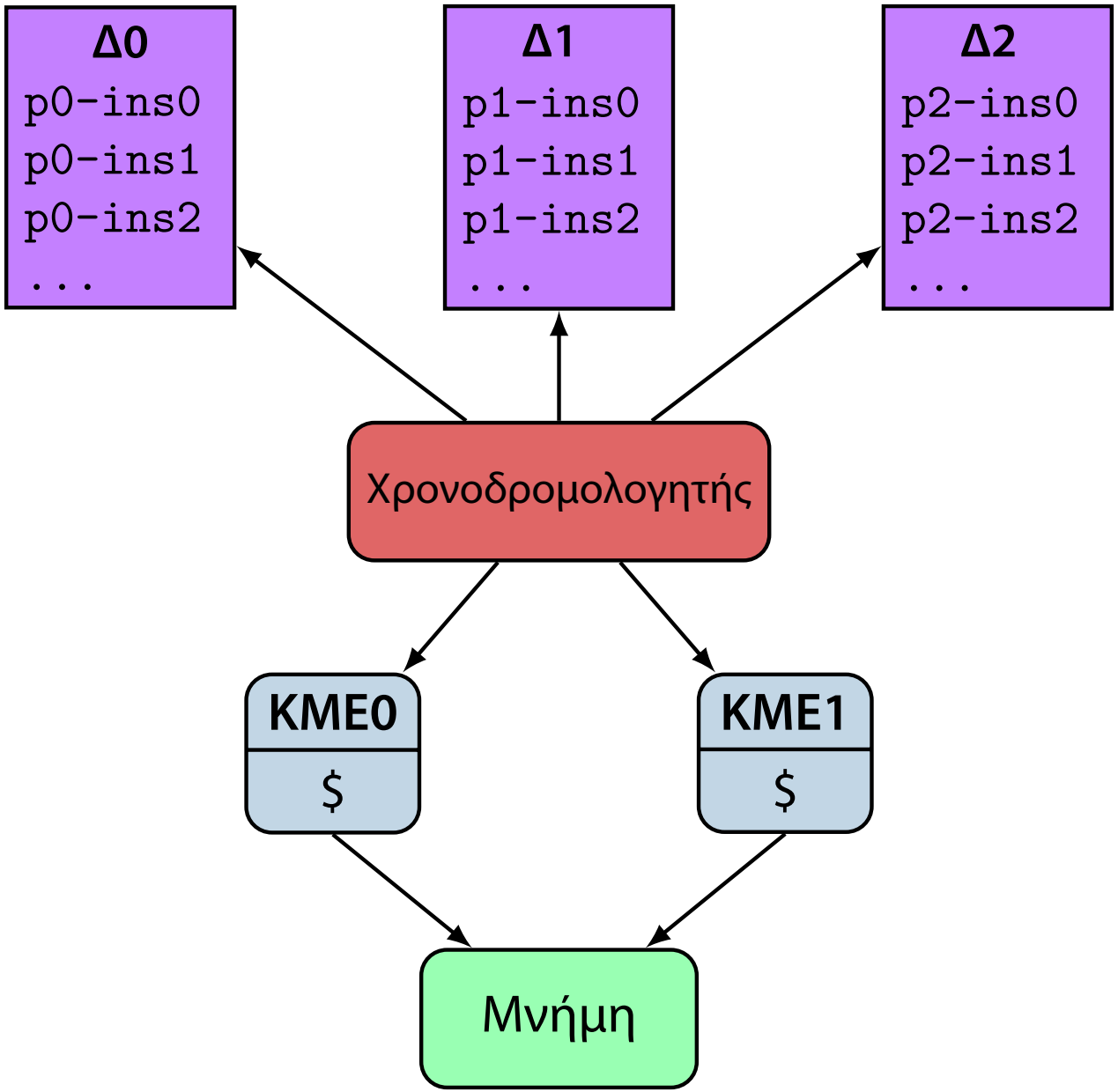
1 επεξεργαστής, Διακοπτή χρονοδρομολόγηση

(χώρος χρήστη)



>1 επεξεργαστές, πολυεπεξεργασία

(χώρος χρήστη)



Κρίσιμο τμήμα (ΚΤ)

critical section

- Λύση στο πρόβλημα της ταυτόχρονης πρόσβασης σε μοιραζόμενους πόρους
- Διαισθητικά: ορίζονται τμήματα κώδικα στα οποία μπορεί να βρίσκεται το πολύ μια διεργασία τη φορά
- Επιτρέπει την ατομική εκτέλεση τμημάτων
- Απαιτήσεις:
  - Αμοιβαίος αποκλεισμός (mutual exclusion)
  - Πρόοδος
  - Πεπερασμένη αναμονή
- Παραδοχές
  - μη-μηδενικές ταχύτητες εκτέλεσης διεργασιών
  - **καμία** παραδοχή για την σχετική ταχύτητα των διεργασιών

Κλειδώματα

Locks

Ορισμός κρίσιμων τμημάτων:

- Είσοδος στο ΚΤ: κλείδωμα (lock)
- Έξοδος από το ΚΤ: ξεκλείδωμα (unlock)

```
do {
    lock(mylock);
            /* critical section */
        code
    unlock(mylock);
    remainder section
} while (TRUE);
```

Κλειδώματα

Locks

Ορισμός κρίσιμων τμημάτων:

- Είσοδος στο ΚΤ: κλείδωμα (lock)
- Έξοδος από το ΚΤ: ξεκλείδωμα (unlock)

```
do {                                    do {
    lock(mylock);                           lock(mylock);
           /* critical section */                   /* critical section
        code                                    other code
    unlock(mylock);                         unlock(mylock);
    remainder section                       other remainder section
} while (TRUE);                         } while (TRUE);
```

- μεταβλητή lock: ορίζει instances του ΚΤ

Μια πρώτη λύση στο πρόβλημα του κρίσιμου τμήματος

για 2 διεργασίες

```
bool flag[2] = { FALSE, FALSE };
do {
    flag[me] = TRUE;
    while (flag[other])
        ;
        critical section
    flag[me] = FALSE;
    remainder section
} while (TRUE);
```

Μια πρώτη λύση στο πρόβλημα του κρίσιμου τμήματος

για 2 διεργασίες

```
bool flag[2] = { FALSE, FALSE };
do {
    flag[me] = TRUE;
    while (flag[other])
        ;
        critical section
    flag[me] = FALSE;
    remainder section
} while (TRUE);
```

t	me=0	me=1
1	flag[0] = TRUE	
2		flag[1] = TRUE

Μια πρώτη λύση στο πρόβλημα του κρίσιμου τμήματος

για 2 διεργασίες

```
bool flag[2] = { FALSE, FALSE };
do {
    flag[me] = TRUE;
    while (flag[other])
        ;
        critical section
    flag[me] = FALSE;
    remainder section
} while (TRUE);
```

t	me=0	me=1
1	flag[0] = TRUE	
2		flag[1] = TRUE

Αδιέξοδο!

(αδύνατο να συνεχιστεί η εκτέλεση)

Η λύση του Peterson

```
do {
    flag[me] = TRUE;
    turn = other;
    while (flag[other] && turn == other)
        ;
        critical section
    flag[me] = FALSE;
    remainder section
} while (TRUE);
```

- Περιορίζεται σε δύο διεργασίες
- Θεωρητικά ικανοποιεί τις απαραίτητες συνθήκες
- Πρακτικά δεν μπορεί να χρησιμοποιηθεί

Ζητήματα Υλοποίησης

στην πράξη...

- Ο μεταγλωττιστής μπορεί να αναδιατάξει τις εντολές
- Ο Επεξεργαστής μπορεί να εκτελέσει εντολές εκτός σειράς (Out of Order Execution)
- Σε πολυεπεξεργαστικά συστήματα μοιραζόμενης μνήμης, δεν είναι προφανές πότε θα φανούν οι αλλαγές που έκανε ένας επεξεργαστής σε μια θέση μνήμης στους υπόλοιπους.

Μηχανισμοί Συγχρονισμού

- Ατομικές εντολές (Atomic Operations)
- Περιστροφικά Κλειδώματα (Spinlocks)
- Κλειδώματα αμοιβαίου αποκλεισμού (Mutexes)
- Σημαφόροι (Semaphores)
- Ελεγκτές/Παρακολουθητές (Monitors)

Ατομικές Εντολές

Atomic Operations

- Εγγυημένες να λειτουργούν ***ατομικά***
- Υλοποιούνται στο υλικό
- Χρησιμοποιούνται για τη δημιουργία μηχανισμών συγχρονισμού

Παραδείγματα:

- `TestAndSet`:
  Θέτει την τιμή μιας μεταβλητής σε TRUE και επιστρέφει την προηγούμενη τιμή.
- `Swap`:
  Ανταλλάσσει την τιμή δύο μεταβλητών

Spinlocks

- Υλοποίηση αμοιβαίου αποκλεισμού με:
  - Ατομικές εντολές
  - Βρόχους ενεργού αναμονής (busy-wait loops)

Spinlocks

- Υλοποίηση αμοιβαίου αποκλεισμού με:
  - Ατομικές εντολές
  - Βρόχους ενεργού αναμονής (busy-wait loops)

Παραδείγματα:

```
do {
  while (TestAndSet(lock))
     ;
        critical section
  UnSet(lock);
  remainder section
} while (TRUE);
```

Spinlocks

- Υλοποίηση αμοιβαίου αποκλεισμού με:
  - Ατομικές εντολές
  - Βρόχους ενεργού αναμονής (busy-wait loops)

Παραδείγματα:

```
do {
 while (TestAndSet(lock))
    ;
       critical section
 UnSet(lock);
 remainder section
} while (TRUE);
```

```
do {
 key = TRUE;
 while (key)
    Swap(lock,key);
       critical section
 UnSet(lock);
 remainder section
} while (TRUE);
```

Spinlocks

- Υλοποίηση αμοιβαίου αποκλεισμού με:
  - Ατομικές εντολές
  - Βρόχους ενεργού αναμονής (busy-wait loops)

Παραδείγματα:

```
do {
 while (TestAndSet(lock))
    ;
       critical section
 UnSet(lock);
 remainder section
} while (TRUE);
```

```
do {
 key = TRUE;
 while (key)
    Swap(lock,key);
       critical section
 UnSet(lock);
 remainder section
} while (TRUE);
```

Δεν ικανοποιούν την απαίτηση πεπερασμένης αναμονής

Υλοποίηση spinlock

που ικανοποιεί όλες τις απαιτήσεις για το κρίσιμο τμήμα

```
#define NEXT(x) ((x + 1) % N)
bool waiting[N], lock, key;

Lock:
    Set(waiting[me]);
    key = TRUE;
    while ((Test(waiting[me]) && key)
        key = TestAndSet(&lock);
    UnSet(waiting[me]);

Unlock:
    for (j = NEXT(me); ; j = NEXT(j)){
        if (j == me){
            UnSet(lock); break;
        }
        if (Test(waiting[j])){
            UnSet(waiting[j]); break;
        }
    }
```

Τακτικές αμοιβαίου αποκλεισμού

Ενεργός Αναμονή

Καταναλώνει υπολογιστικούς πόρους, όσο μια άλλη διεργασία βρίσκεται στο κρίσιμο τμήμα.

- Η ΚΜΕ δεν μπορεί να χρησιμοποιηθεί για άλλες εργασίες
- Κατανάλωση ενέργειας
- Κατάλληλη για μικρά διαστήματα αναμονής

Αμοιβαίος αποκλεισμός με αναστολή εκτέλεσης (Mutex)

- Αν το ΚΤ είναι κατειλημμένο η διεργασία αναστέλλει την λειτουργία της και τίθεται υπό αναμονή (state = WAITING).
- Ο επεξεργαστής ελευθερώνεται και ο Χρονοδρομολογητής επιλέγει άλλη διεργασία.
- Κατά την έξοδο από το ΚΤ, ενεργοποιείται μια υπό αναμονή διεργασία (state = READY)
- Κατάλληλη για μεγάλα διαστήματα αναμονής

Σημαφόροι

Semaphores

- Τα spinlocks και mutexes λύνουν το πρόβλημα του κρίσιμου τμήματος, αλλά:
  - Είναι ”low-level” primitives και δυσχεραίνουν τον προγραμματισμό
  - Δεν επιλύουν πιο σύνθετα προβλήματα συγχρονισμού (π.χ. > 1 διεργασίες στο κρίσιμο τμήμα)
- Οι **σημαφόροι** επινοήθηκαν από τον Dijkstra
- Αποτελούν προγραμματιστικές δομές συγχρονισμού υψηλότερου επιπέδου
- Είναι πιο ευέλικτοι από τα locks και μπορούν να χρησιμοποιηθούν για επίλυση πιο σύνθετων προβλημάτων (βλ. συνέχεια)

Σημαφόροι

Λειτουργίες

- Ο σημαφόρος είναι ένας ακέραιος αριθμός στον οποίο επιτρέπεται η πρόσβαση μόνο με τρεις αυστηρά ορισμένες λειτουργίες: ***αρχικοποίηση***, `wait` και `signal`.
- Λειτουργίες:
  - Αρχικοποιείται σε μία ακέραιη τιμή. Μετά την αρχικοποίηση επιτρέπονται **μόνο** η `wait()` και η `signal()`
  - `wait()`: Ο σημαφόρος μειώνεται κατά 1. Αν πλέον έχει τιμη < 0, η διεργασία περιμένει (”κολλάει”).
  - `signal()`: Ο σημαφόρος αυξάνεται κατά 1. Αν πλέον έχει τιμή $<= 0$ μία διεργασία που περιμένει στο σημαφόρο θα πρέπει να ειδοποιηθεί (να ”ξεκολλήσει”).

Σημαφόροι

Λειτουργίες

```
wait (S) {
 S--;
 if (S < 0)
     block on semaphore
}
```

```
signal (S) {
 S++;
 if (S <= 0)
    unblock one process or threa
    that is blocked on semaphore
}
```

Σημαφόροι

Λειτουργίες

```
wait (S) {
 S--;
 if (S < 0)
     block on semaphore
}
```

```
signal (S) {
 S++;
 if (S <= 0)
     unblock one process or threa
     that is blocked on semaphore
}
```

Όλες οι αλλαγές στην τιμή του σημαφόρου είναι ατομικές!

Σημαφόροι

Υλοποίηση με ενεργό αναμονή – δεν υποστηρίζει αρνητικές τιμές

```
wait (S) {
  while (1) {
     lock(S_lock);
     if (S > 0) {
        S--;
        unlock(S_lock);
        break;
     }
     unlock(S_lock);
  }
}
```

```
signal (S) {
  lock(S_lock);
  S++;
  unlock(S_lock);
}
```

Σημαφόροι

Υλοποίηση χωρίς ενεργό αναμονή - υποστηρίζει αρνητικές τιμές

```
typedef struct {
  int value;
  lock s_lock;
  struct process *list;
} Semaphore;
```

```
wait (Semaphore *S) {
  lock(S->s_lock);
  S->value--;
  if (S->value < 0) {
     add proc to S->list;
     unlock(S->s_lock);
     block();
  }
  unlock(S->s_lock);
}
```

```
signal (Semaphore *S) {
  lock(S->s_lock);
  S->value++;
  if (S->value <= 0) {
     remove P from S->list;
     wakeup(P);
  }
  unlock(S->s_lock);
}
```

Σημαφόροι

Σημειώσεις

- Μια διεργασία δεν μπορεί να ξέρει την τιμή του σημαφόρου και συγκεκριμένα δεν ξέρει αν θα χρειαστεί να περιμένει ή όχι σε περίπτωση που καλέσει την `wait()`
- Δεν υπάρχει κάποιος κανόνας για το ποιο νήμα από αυτά που περιμένουν, θα συνεχίσει την λειτουργία του μετά από κλήση της `signal()`
- Ένα mutex ισοδυναμεί με έναν σημαφόρο αρχικοποιημένο στο 1 (unlocked)
- Οι λειτουργίες στους σημαφόρους παραδοσιακά αναφέρονται και ως ***P()*** (wait()) και ***V()*** (signal()).

Πρόβλημα σειριοποίησης (ordering)

- Θέλουμε η διεργασία P_1 να εκτελέσει τη ρουτίνα S_1 πριν η διεργασία P_2 εκτελέσει τη ρουτίνα S_2.
- Αρχικοποιούμε το σημαφόρο `sema` στην τιμή 0.

```
// P1
S1();
signal(sema);
```

```
// P2
wait(sema);
S2();
```

Πρόβλημα Παραγωγού-Καταναλωτή

```
#define NEXT(x) ((x + 1) % N)
item_t buffer[N]; int in=0, out=0, count=0;
```

```
void enqueue(item_t item){

    item_t item;
    while (count == N)
        ; // wait
    buffer[in] = item;
    in = NEXT(in);
    count++;

}
```

```
item_t dequeue(void){

    item_t item;
    while (count == 0)
        ; // wait
    item = buffer[out];
    out = NEXT(out);
    count--;

    return item
}
```

Πρόβλημα Παραγωγού-Καταναλωτή

```
#define NEXT(x) ((x + 1) % N)
item_t buffer[N]; int in=0, out=0, count=0;
sema_t mutex = semaphore(1);
```

```
void enqueue(item_t item){
    wait(mutex);
    item_t item;
    while (count == N)
        ; // wait
    buffer[in] = item;
    in = NEXT(in);
    count++;
    signal(mutex);
}
```

```
item_t dequeue(void){
    wait(mutex);
    item_t item;
    while (count == 0)
        ; // wait
    item = buffer[out];
    out = NEXT(out);
    count--;
    signal(mutex);
    return item
}
```

Πρόβλημα Παραγωγού-Καταναλωτή

```
#define NEXT(x) ((x + 1) % N)
item_t buffer[N]; int in=0, out=0, count=0;
sema_t mutex = semaphore(1);
```

```
void enqueue(item_t item){
    wait(mutex);
    item_t item;
    while (count == N)
        ; // wait
    buffer[in] = item;
    in = NEXT(in);
    count++;
    signal(mutex);
}
```

```
item_t dequeue(void){
    wait(mutex);
    item_t item;
    while (count == 0)
        ; // wait
    item = buffer[out];
    out = NEXT(out);
    count--;
    signal(mutex);
    return item
}
```

ΑΔΙΕΞΟΔΟ!

(deadlock)

Πρόβλημα Παραγωγού-Καταναλωτή

(Σωστή λύση)

```
#define NEXT(x) ((x + 1) % N)
item_t buffer[N]; int in=0, out=0, count=0;
sema_t mutex = semaphore(1);
sema_t items = semaphore(0);
sema_t space = semaphore(N);

void enqueue(item_t item){
    item_t item;

    buffer[in] = item;
    in = NEXT(in);
    count++;

}
```

```
item_t dequeue(void){
    item_t item;

    item = buffer[out];
    out = NEXT(out);
    count--;

    return item
}
```

Πρόβλημα Παραγωγού-Καταναλωτή

(Σωστή λύση)

```
#define NEXT(x) ((x + 1) % N)
item_t buffer[N]; int in=0, out=0, count=0;
sema_t mutex = semaphore(1);
sema_t items = semaphore(0);
sema_t space = semaphore(N);
```

```
void enqueue(item_t item){
    item_t item;
    wait(space);
    wait(mutex);
    buffer[in] = item;
    in = NEXT(in);
    count++;
    signal(mutex);
    signal(items);
}
```

```
item_t dequeue(void){
    item_t item;
    wait(items);
    wait(mutex);
    item = buffer[out];
    out = NEXT(out);
    count--;
    signal(mutex);
    signal(space);
    return item
}
```

Πρόβλημα Αναγνωστών - Εγγραφέων

readers-writers problem

- Δύο κλάσεις προσβάσεων:
  - **Αναγνώστες**: Δεν αλλάζουν τα δεδομένα
  - **Εγγραφείς**: Αλλάζουν τα δεδομένα
- Κανόνες:
  - Επιτρέπονται πολλαπλοί Αναγνώστες ταυτόχρονα
  - Οι Εγγραφείς πρέπει να έχουν αποκλειστική πρόσβαση
- Χρησιμοποιείται για δομές που δεν αλλάζουν συχνά
- Προτεραιότα:
  - Στους Εγγραφείς
  - Στους Αναγνώστες
  - Πιθανή λιμοκτονία (starvation) και στις δύο περιπτώσεις

Πρόβλημα Αναγνωστών - Εγγραφέων

προτεραιότητα στους αναγνώστες

```
sema_t mutex = semaphore(1);
sema_t write = semaphore(1);
int readcount = 0;
```

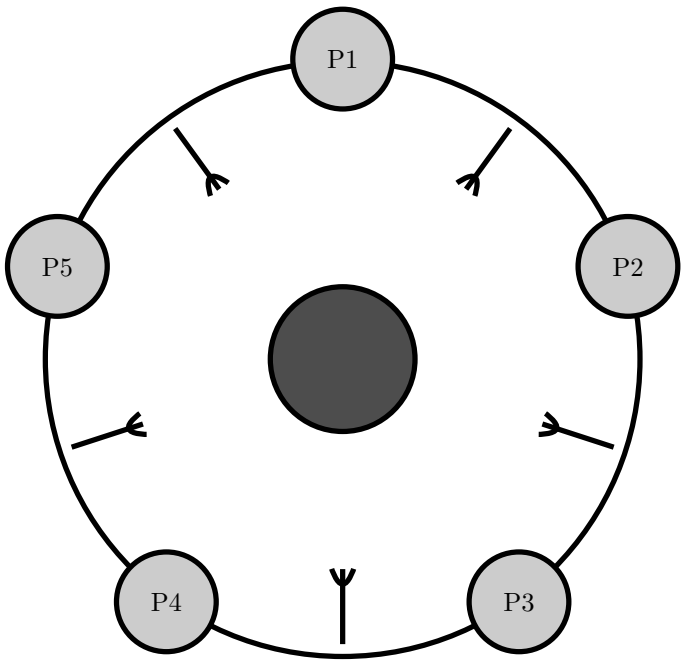
```
read_lock:
    wait(mutex);
    if (++readcount == 1)
        wait(write);
    signal(mutex);

read_unlock:
    wait(mutex);
    if (--readcount == 0)
        signal(write);
    signal(mutex);
```

```
write_lock:
    wait(write);

write_unlock:
    signal(write);
```

Το πρόβλημα των συνδαιτυμόνων φιλοσόφων

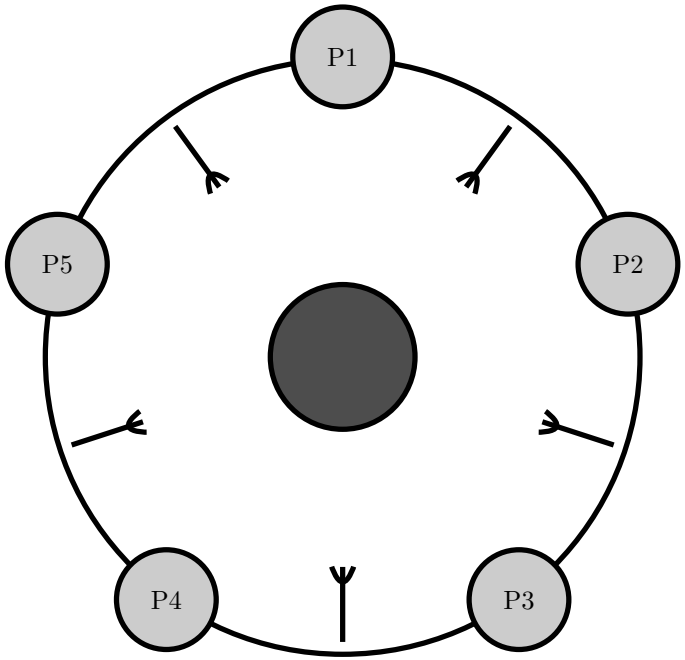
Dining philosophers problem

- 5 φιλόσοφοι
  (τρώνε και σκέφτονται)
- 5 ξυλάκια φαγητού
- Όταν ένας φιλόσοφος πεινάει, χρειάζεται 2 ξυλάκια για να φάει
- Δεν μπορούν 2 φιλόσοφοι να χρησιμοποιούν το ίδιο ξυλάκι
- Αποφυγή αδιεξόδου
- Αποφυγή λιμοκτονίας



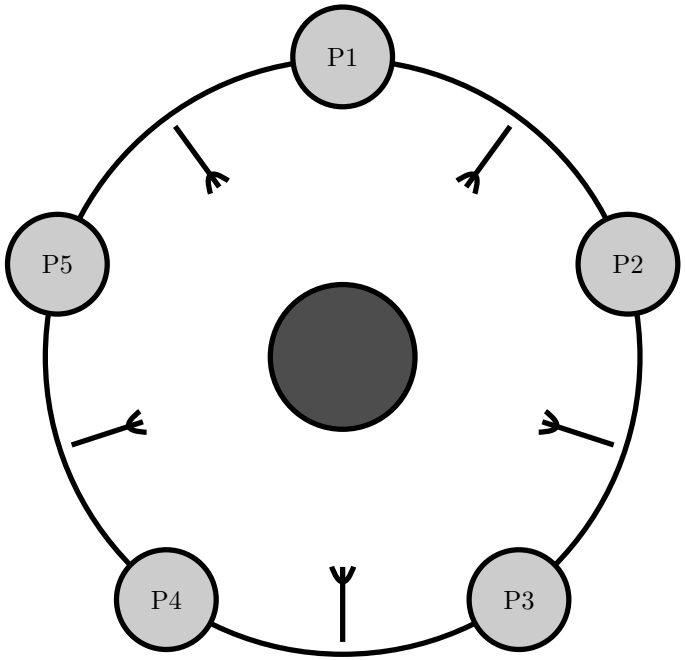
Λύση στο πρόβλημα των συνδαιτυμόνων φιλοσόφων

```
#define NEXT(x) ((x+1) % N)
sema_t F[N]; //forks
do {
  think();
  wait(F[i]);
  wait(F[NEXT(i)]);
  eat();
  signal(F[NEXT(i)]);
  signal(F[i]);
} while (TRUE);
```



Λύση στο πρόβλημα των συνδαιτυμόνων φιλοσόφων

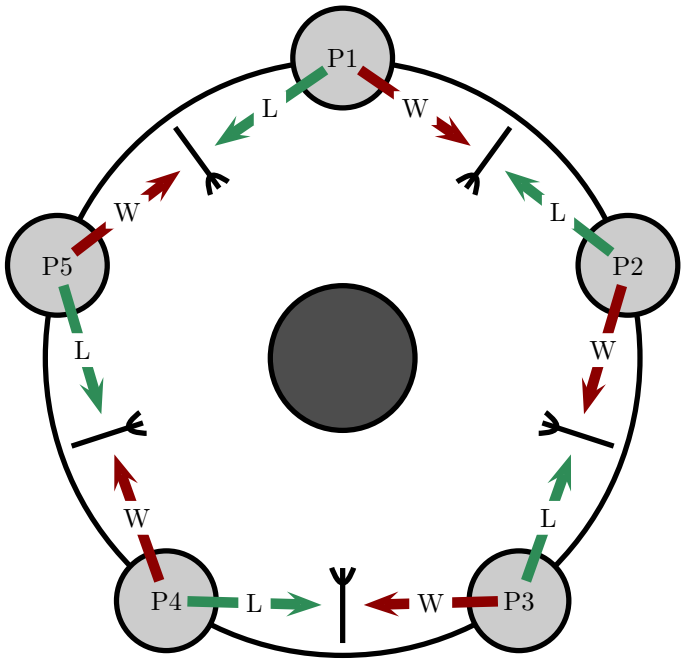
```
#define NEXT(x) ((x+1) % N)
sema_t F[N]; //forks
do {
  think();
  wait(F[i]);
  wait(F[NEXT(i)]);
  eat();
  signal(F[NEXT(i)]);
  signal(F[i]);
} while (TRUE);
```



1. P1 παίρνει F1
2. P2 παίρνει F2
3. P3 παίρνει F3
4. P4 παίρνει F4
5. P5 παίρνει F5

Λύση στο πρόβλημα των συνδαιτυμόνων φιλοσόφων

```
#define NEXT(x) ((x+1) % N)
sema_t F[N]; //forks
do {
  think();
  wait(F[i]);
  wait(F[NEXT(i)]);
  eat();
  signal(F[NEXT(i)]);
  signal(F[i]);
} while (TRUE);
```



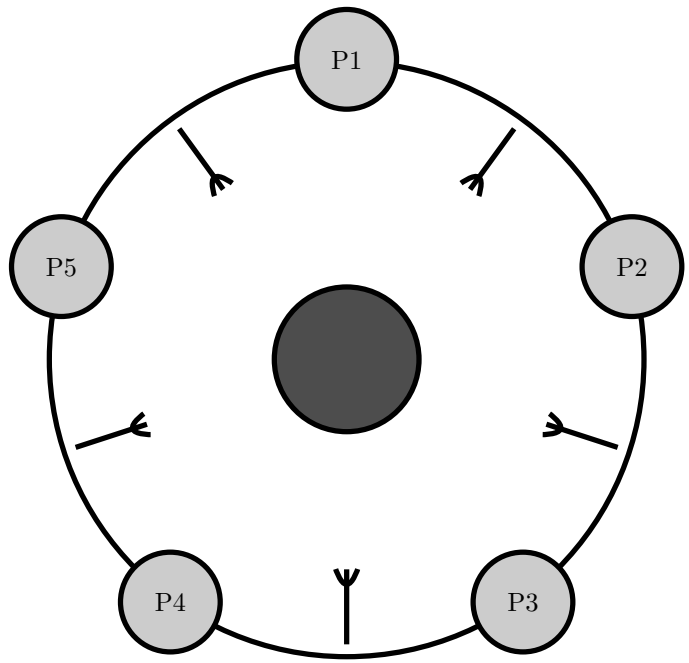
1. P1 παίρνει F1
2. P2 παίρνει F2
3. P3 παίρνει F3
4. P4 παίρνει F4
5. P5 παίρνει F5

$\Rightarrow$ Αδιέξοδο!

Λύσεις στο πρόβλημα των συνδαιτυμόνων φιλοσόφων

(σωστές)

Λύσεις

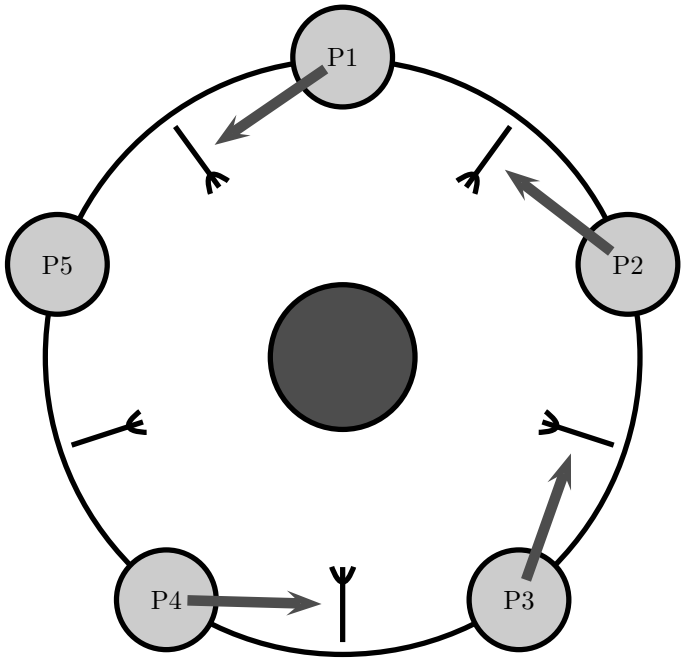


Λύσεις στο πρόβλημα των συνδαιτυμόνων φιλοσόφων

(σωστές)

Λύσεις

1. Χρήση σημαφόρου, αρχικοποιημένου στην τιμή 4, ώστε σε κάθε περίπτωση να μην διεκδικούν και οι 5 φιλόσοφοι τα ξυλάκια.

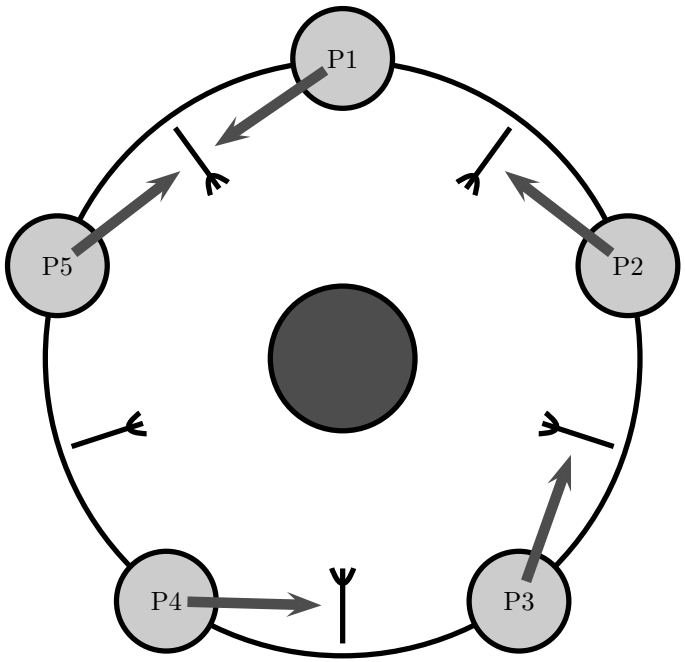


Λύσεις στο πρόβλημα των συνδαιτυμόνων φιλοσόφων

(σωστές)

Λύσεις

1. Χρήση σημαφόρου, αρχικοποιημένου στην τιμή 4, ώστε σε κάθε περίπτωση να μην διεκδικούν και οι 5 φιλόσοφοι τα ξυλάκια.
2. Σε περίπτωση που ένας από τους φιλόσοφους ξεκινάει να πάρει ένα ξυλάκι με διαφορετική σειρά από ότι οι υπόλοιποι, τότε δεν μπορεί να δημιουργηθεί αδιέξοδο.



Ελεγκτές/Παρακολουθητές

(Monitors)

- Οι σημαφόροι παρέχουν βολικό και αποτελεσματικό μηχανισμό συγχρονισμού
- **Ωστόσο:** βάρος χρήσης στον προγραμματιστή (που δεν είναι ποτέ καλή ιδέα ☺)

Ελεγκτές/Παρακολουθητές

(Monitors)

- Οι σημαφόροι παρέχουν βολικό και αποτελεσματικό μηχανισμό συγχρονισμού
- **Ωστόσο:** βάρος χρήσης στον προγραμματιστή (που δεν είναι ποτέ καλή ιδέα ☺)

Ελεγκτές/Παρακολουθητές:

- Υψηλού επιπέδου σχήματα συγχρονισμού που παρέχονται από τις γλώσσες προγραμματισμού
- Αφηρημένος τύπος δεδομένων (abstract data type – ADT)
  - Κρύβει λεπτομέρειες υλοποίησης
  - Δεν επιτρέπει απευθείας πρόσβαση στα δεδομένα (ιδιωτικά δεδομένα)
  - Ο προγραμματιστής ορίζει λειτουργίες (μεθόδους)

Ελεγκτές/Παρακολουθητές

(παράδειγμα δομής)

```
monitor MyMonitor {
    // shared variables
    int a;

    // methods
    procedure P_1(...) {
        ...
    }
    procedure P_2(...) {
        ...
    }

    // initialization
    initialize() {
        a = ...
        ...
    }
}
```

Ελεγκτές/Παρακολουθητές

Μηχανισμοί που παρέχονται

Αμοιβαίος Αποκλεισμός (mutual exclusion)

Μεταβλητές συνθήκης (condition variables)

Ελεγκτές/Παρακολουθητές

Μηχανισμοί που παρέχονται

Αμοιβαίος Αποκλεισμός (mutual exclusion)

- Όταν μια διεργασία εκτελεί μια λειτουργία, τότε ***η διεργασία βρίσκεται μέσα στον παρακολουθητή***
- Είναι εγγυημένο ότι μόνο μια διεργασία μπορεί να είναι ενεργή μέσα στον παρακολουθητή

Μεταβλητές συνθήκης (condition variables)

Ελεγκτές/Παρακολουθητές

Μηχανισμοί που παρέχονται

Αμοιβαίος Αποκλεισμός (mutual exclusion)

- Όταν μια διεργασία εκτελεί μια λειτουργία, τότε ***η διεργασία βρίσκεται μέσα στον παρακολουθητή***
- Είναι εγγυημένο ότι μόνο μια διεργασία μπορεί να είναι ενεργή μέσα στον παρακολουθητή

Μεταβλητές συνθήκης (condition variables)

- Μηχανισμός, ώστε μια διεργασία να αναστέλλει την λειτουργία της έως ότου ικανοποιηθεί μια συνθήκη (πχ Παραγωγός-Καταναλωτής)
- Λειτουργίες:

wait(): Η διεργασία αδρανοποιείται μέχρι να κληθεί η signal().
signal(): Ενεργοποιεί (ακριβώς) μια αδρανοποιημένη διεργασία.
(Πότε ;)

Παράδειγμα Παρακολουθητή

Παραγωγός-Καταναλωτής

```
#define NEXT(x) ((x + 1) % N)
monitor ProducerConsumer {

} // end ProducerConsumer
```

Παράδειγμα Παρακολουθητή

Παραγωγός-Καταναλωτής

```
#define NEXT(x) ((x + 1) % N)
monitor ProducerConsumer {
   item_t buffer[N];
   int in=0, out=0, count=0;
   condition full, empty;
     void enqueue(item_t item){   item_t dequeue(void){









     }
                                        }
} // end ProducerConsumer
```

Παράδειγμα Παρακολουθητή

Παραγωγός-Καταναλωτής

```
#define NEXT(x) ((x + 1) % N)
monitor ProducerConsumer {
   item_t buffer[N];
   int in=0, out=0, count=0;
   condition full, empty;
     void enqueue(item_t item){   item_t dequeue(void){
       item_t item;
       if (count == N)
           full.wait();
       buffer[in] = item;
       in = NEXT(in);
       if (++count == 1)
           empty.signal();
     }
                                  }
} // end ProducerConsumer
```

Παράδειγμα Παρακολουθητή

Παραγωγός-Καταναλωτής

```
#define NEXT(x) ((x + 1) % N)
monitor ProducerConsumer {
   item_t buffer[N];
   int in=0, out=0, count=0;
   condition full, empty;
      void enqueue(item_t item){
        item_t item;
        if (count == N)
            full.wait();
        buffer[in] = item;
        in = NEXT(in);
        if (++count == 1)
            empty.signal();
      }
      item_t dequeue(void){
          item_t item;
          if (count == 0)
              empty.wait();
          item = buffer[out];
          out = NEXT(out);
          if (count-- == N)
              full.signal();
          return item;
      }
} // end ProducerConsumer
```

Παράδειγμα Παρακολουθητή

Πρόβλημα των συνδαιτυμόνων φιλοσόφων

```
monitor DiningPhilosophers {
  enum {T, H, E} state[5];
  condition cond[N];

  void pickup(int i) {
      state[i] = H;
      test(i);
      if (state[i] != E)
          cond[i].wait();
  }

  void putdown(int i) {
      state[i] = T;
      test(PREV(i));
      test(NEXT(i));
   }

  void test(int i){
      if (state[i] == H &&
          state[PREV(i)] != E && state[NEXT(i)] != E){
              state[i] = E;
              cond[i].signal();
     }
  }
} // end of DiningPhilosophers
```

Ατομικές Συναλλαγές

Atomic Transactions

- Η έννοια της συναλλαγής (transaction) χρησιμοποιείται στις βάσεις δεδομένων
- Σύνολο εντολών που υλοποιεί μια λογική λειτουργία
- Ορίζεται από τον προγραμματιστή
- Μπορεί να:
  - Επιτύχει και να κατοχυρωθεί (COMMIT)
  - Αποτύχει και να ακυρωθεί (ABORT, ROLLBACK)
- Μοντέλο μνήμης με συναλλαγές (Transactional Memory)

Στρατηγικές συγχρονισμού σε ΛΣ (και όχι μόνο)

(Εξέλιξη)

- Απενεργοποίηση διακοπών
- Ένα κεντρικό κλείδωμα (BKL)

Στρατηγικές συγχρονισμού σε ΛΣ (και όχι μόνο)

(Εξέλιξη)

- Απενεργοποίηση διακοπών
  Δεν αρκεί για πολυεπεξεργαστικά συστήματα
- Ένα κεντρικό κλείδωμα (BKL)
  Δεν έχει καλή κλιμάκωση

Στρατηγικές συγχρονισμού σε ΛΣ (και όχι μόνο)

(Εξέλιξη)

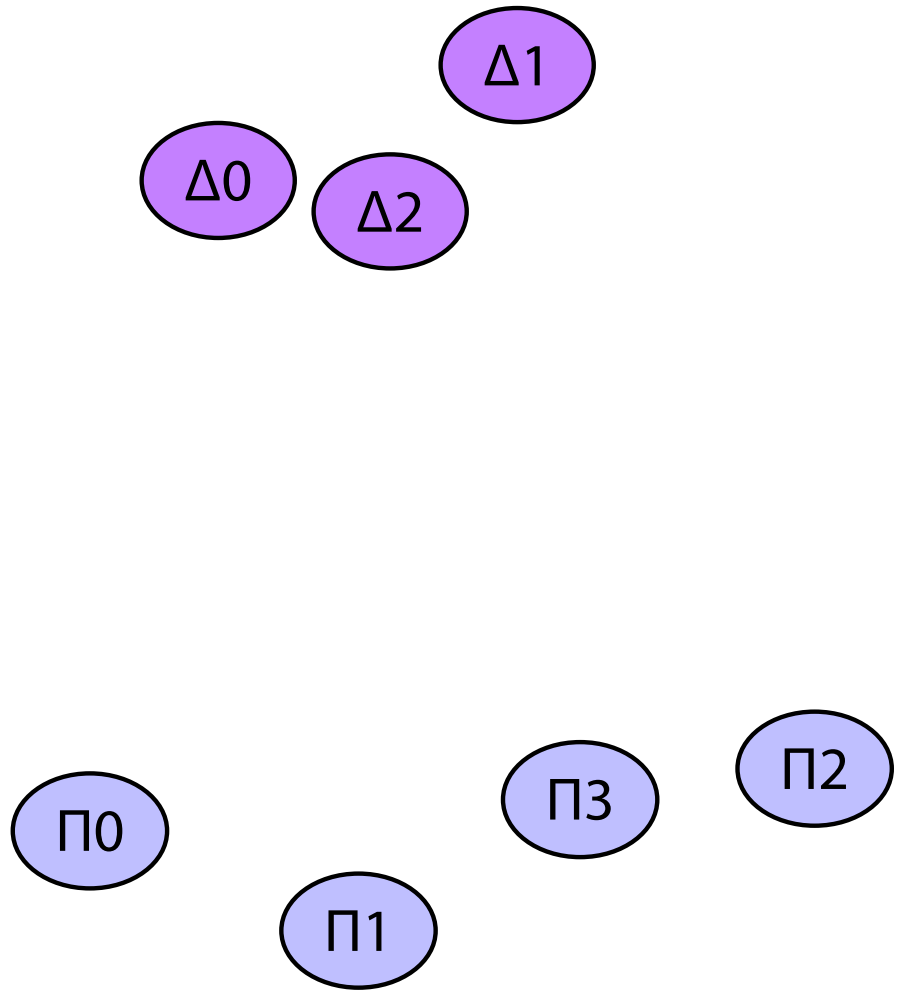
- Απενεργοποίηση διακοπών
  Δεν αρκεί για πολυεπεξεργαστικά συστήματα
- Ένα κεντρικό κλείδωμα (BKL)
  Δεν έχει καλή κλιμάκωση

- Σπάσιμο σε περισσότερα κλειδώματα
  (fine-grain locking)
- Διαμοιρασμός των δεδομένων και ανταλλαγή μηνυμάτων
  (partitioning, message passing)

Κεντρικό και πολλαπλά κλειδώματα για κοινούς πόρους

Big lock vs fine-grain locking

- Θεωρούμε:
  - Διεργασίες (Δx)
  - Πόρους (Πx)

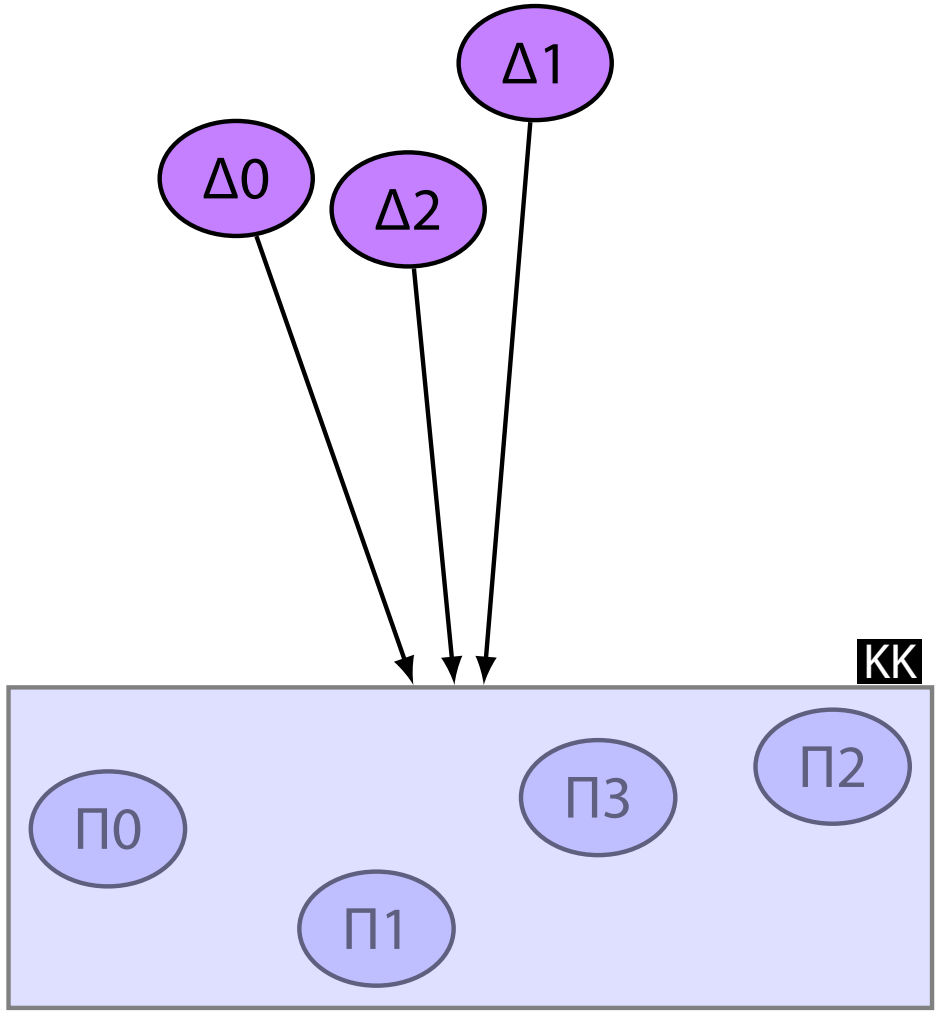


Κεντρικό και πολλαπλά κλειδώματα για κοινούς πόρους

Big lock vs fine-grain locking

- Θεωρούμε:
  - Διεργασίες (Δx)
  - Πόρους (Πx)

- **Κεντρικό κλείδωμα (ΚΚ):**
  Όποια διεργασία πάρει το ΚΚ μπορεί να χρησιμοποιήσει όλους τους πόρους



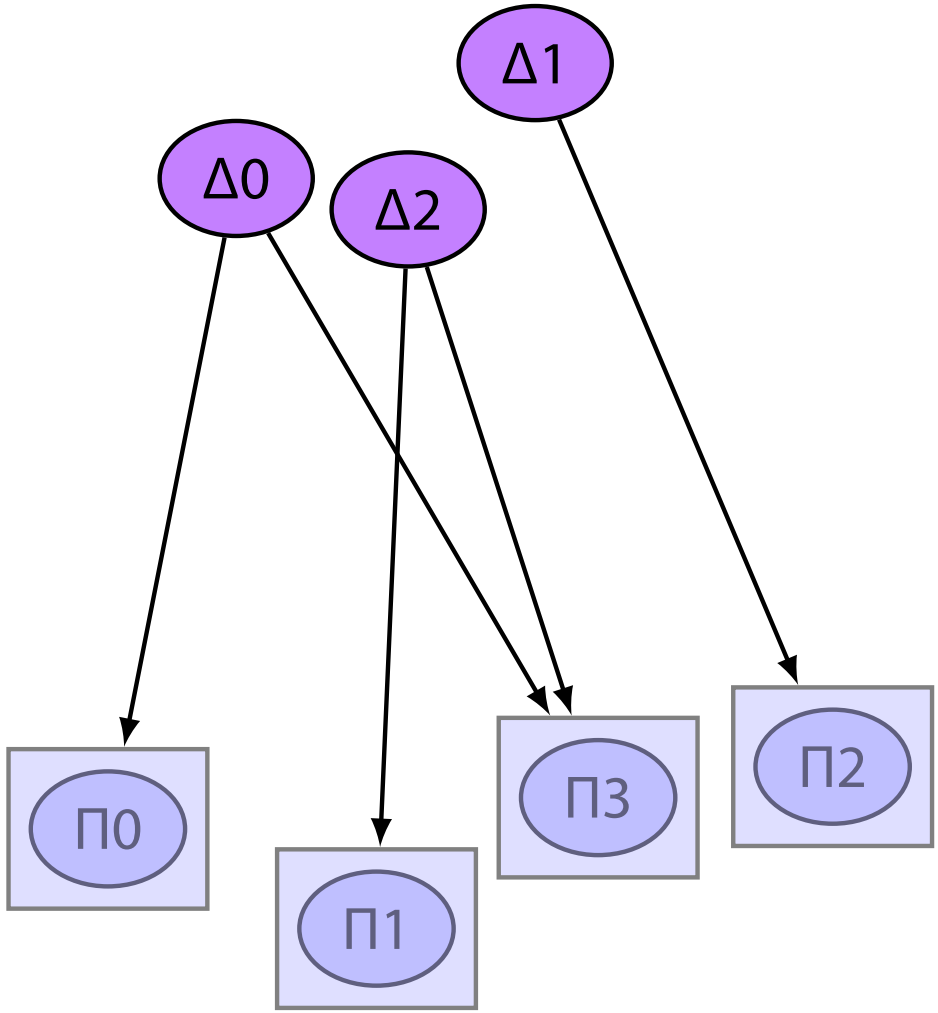
Κεντρικό και πολλαπλά κλειδώματα για κοινούς πόρους

Big lock vs fine-grain locking

- Θεωρούμε:
  - Διεργασίες (Δx)
  - Πόρους (Πx)

- **Κεντρικό κλείδωμα (ΚΚ):**
  Όποια διεργασία πάρει το ΚΚ μπορεί να χρησιμοποιήσει όλους τους πόρους

- **Πολλαπλά κλειδώματα (fine-grain locking)**
  1 κλείδωμα ανά διαθέσιμο πόρο
  \+ Δυνατότητα παραλληλίας



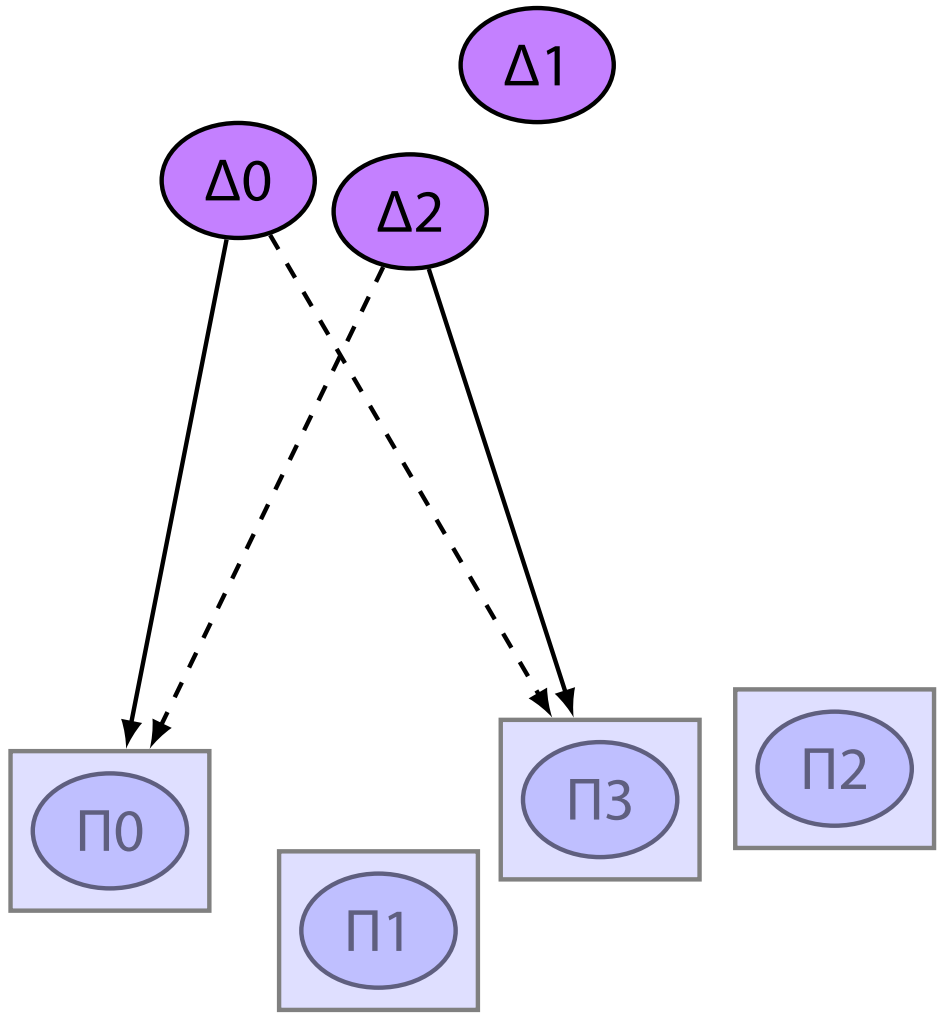
Κεντρικό και πολλαπλά κλειδώματα για κοινούς πόρους

Big lock vs fine-grain locking

- Θεωρούμε:
  - Διεργασίες (Δx)
  - Πόρους (Πx)

- **Κεντρικό κλείδωμα (ΚΚ):**
  Όποια διεργασία πάρει το ΚΚ μπορεί να χρησιμοποιήσει όλους τους πόρους

- **Πολλαπλά κλειδώματα (fine-grain locking)**
  1 κλείδωμα ανά διαθέσιμο πόρο
  + Δυνατότητα παραλληλίας
  - Κίνδυνος αδιεξόδου
    (Δ0 έχει Π0 και περιμένει Π3,
    Δ2 έχει Π3 και περιμένει Π0)

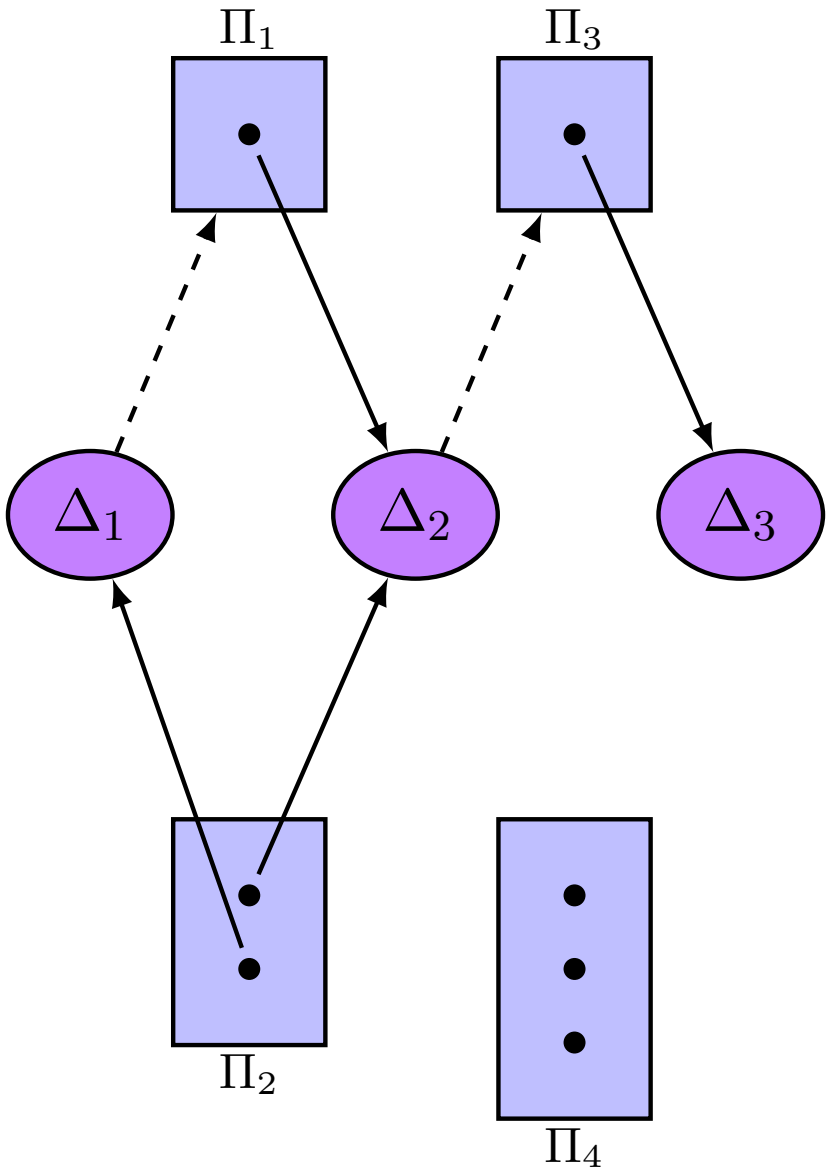


Αναγκαίες συνθήκες για δημιουργία αδιεξόδου

- **Αμοιβαίος αποκλεισμός**
  Ένας τουλάχιστον πόρος να μην υποστηρίζει διαμοιραζόμενο τρόπο λειτουργίας
- **Κράτημα και αναμονή**
  Κατά την απόκτηση πρόσβασης στους πόρους που χρειάζεται οι διεργασίες θα πρέπει να μην απελευθερώνουν πόρους που έχουν δεσμεύσει και να περιμένουν να απελευθερωθούν οι υπόλοιποι
- **Χωρίς Διακοπή**
  Οι πόροι μπορούν να απελευθερωθούν μόνο εθελοντικά από τις διεργασίες που τους έχουν δεσμεύσει
- **Κυκλική Αναμονή**
  P_1 αναμένει πόρο της P_2
  ...
  P_{n-1} αναμένει πόρο της P_n
  P_n αναμένει πόρο της P_1

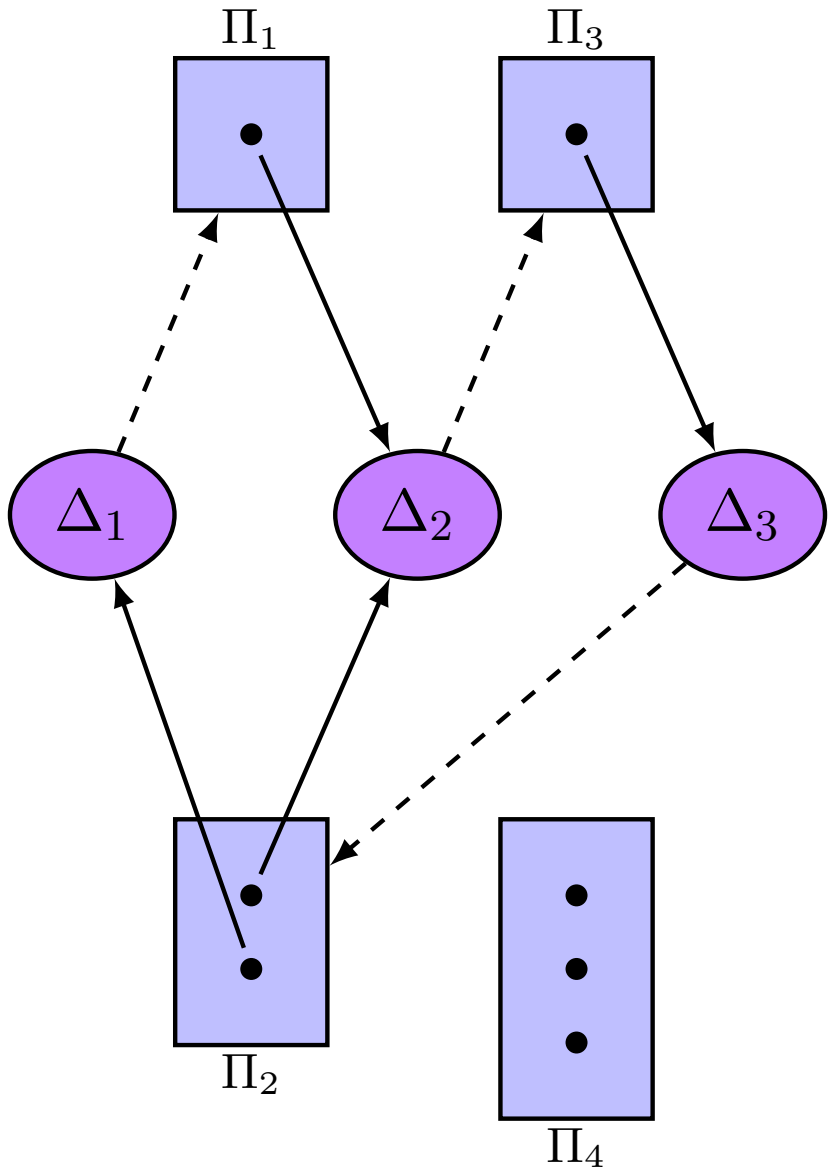
Γράφος εκχώρησης πόρων

- Πόροι Πx (στιγμιότυπα)
- Διεργασίες Δx
- Ακμή αίτησης $\Delta x \rightarrow \Pi x$
- Ακμή ανάθεσης $\Pi x \rightarrow \Delta x$
- Αφορά συγκεκριμένη χρονική στιγμή του συστήματος
- Αν δεν υπάρχει κύκλος δεν υπάρχει αδιέξοδο
- Αν υπάρχει κύκλος τότε μπορεί να υπάρχει αδιέξοδο



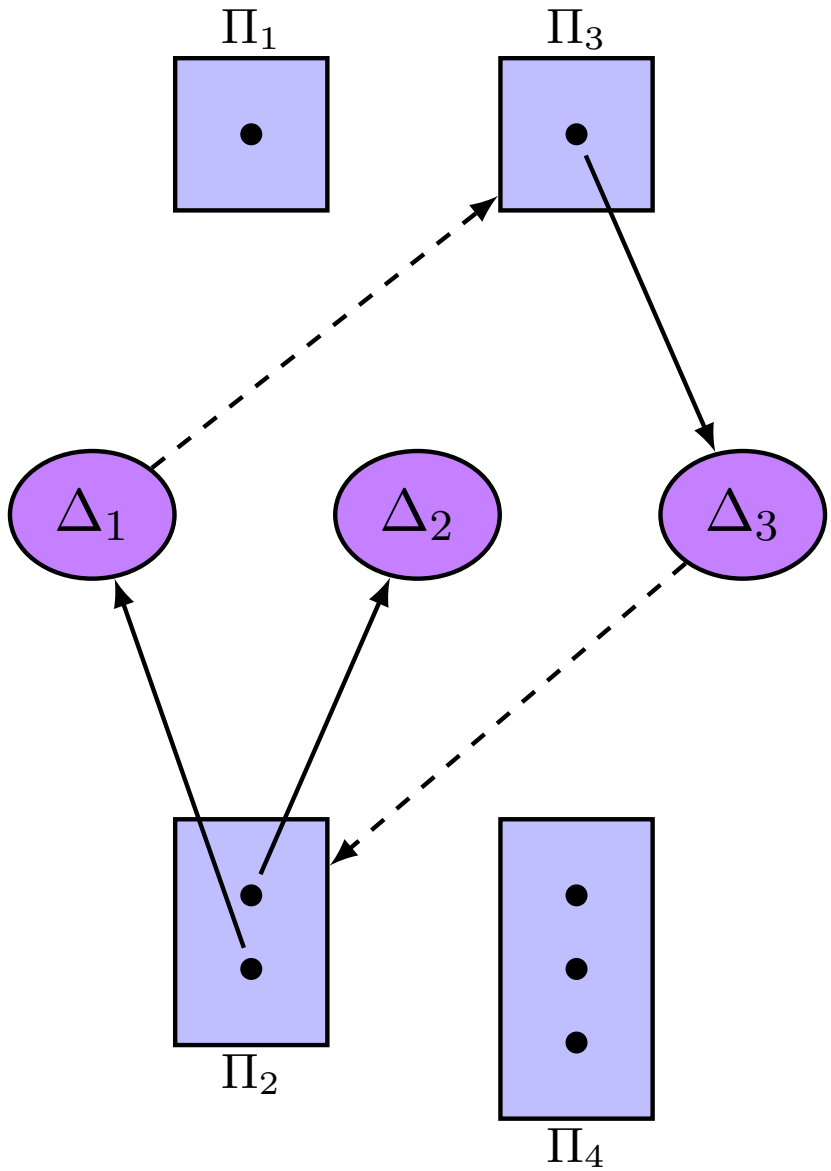
Γράφος εκχώρησης πόρων

- Πόροι Πx (στιγμιότυπα)
- Διεργασίες Δx
- Ακμή αίτησης $\Delta x \rightarrow \Pi x$
- Ακμή ανάθεσης $\Pi x \rightarrow \Delta x$
- Αφορά συγκεκριμένη χρονική στιγμή του συστήματος
- Αν δεν υπάρχει κύκλος δεν υπάρχει αδιέξοδο
- Αν υπάρχει κύκλος τότε μπορεί να υπάρχει αδιέξοδο



Γράφος εκχώρησης πόρων

- Πόροι Πx (στιγμιότυπα)
- Διεργασίες Δx
- Ακμή αίτησης $\Delta x \rightarrow \Pi x$
- Ακμή ανάθεσης $\Pi x \rightarrow \Delta x$
- Αφορά συγκεκριμένη χρονική στιγμή του συστήματος
- Αν δεν υπάρχει κύκλος δεν υπάρχει αδιέξοδο
- Αν υπάρχει κύκλος τότε μπορεί να υπάρχει αδιέξοδο



Αποτροπή αδιεξόδων

αφετηρία: αναγκαίες συνθήκες

- **Αμοιβαίος αποκλεισμός**
  Χρήση πόρων που μπορούν να διαμοιραστούν (πχ αρχεία για ανάγνωση)
- **Κράτημα και αναμονή & Χωρίς Διακοπή**
  Εναλλακτικά πρωτόκολλα
- **Κυκλική Αναμονή**
  Διάταξη πόρων και πραγματοποίηση αιτήσεων με συγκεκριμένη σειρά

Τέλος