

ΛΕΙΤΟΥΡΓΙΚΑ ΣΥΣΤΗΜΑΤΑ



ΠΟΛΥΠΡΟΓΡΑΜΜΑΤΙΣΜΟΣ

Multiprogramming

- Είναι η υλοποίηση κώδικα που εκτελείται από πολλαπλά `threads` ταυτόχρονα.
 - Τι θα πει «ταυτόχρονα»?
 - Ο ρόλος των δεδομένων.

- Παρατήρηση:
 - Για τον πυρήνα, οι διαφορετικές διεργασίες είναι σαν διαφορετικά `threads`.

Race conditions

- Παράδειγμα:

```
int count=0;    /* static variable */  
...  
count = count + 1;
```

- Ερώτηση: αν η παραπάνω γραμμή κώδικα εκτελεστεί από 3 διαφορετικά threads (από 1 φορά), ποιά θα είναι η τιμή της count?
 - Απάντηση: εξαρτάται. ($1 \leq \text{count} \leq 3$).

Εξήγηση

```
count: defw 0
```

```
...
```

```
load A, $count
```

```
add A, A, #1
```

```
store A, $count
```

```
...
```

Thread 1

load

add

store

Thread 2

load

add

store

Thread 3

load

add

store

Race conditions/atomicity

- Ορισμός: Race condition έχουμε όταν για κάποιες πιθανές δρομολογήσεις έχουμε εσφαλμένη εκτέλεση (εσφαλμένη: ως προς αυτό που θέλαμε).
- Δηλαδή: Race conditions = bugs
- Ατομικότητα: η εκτέλεση ενός τμήματος κώδικα από ένα thread τη φορά (χωρίς διακοπή, ά-τομικά = χωρίς τομή).
- Άρα: race condition = παραβίαση της ατομικότητας.

Ορισμοί

- Ορισμός: ένα κομμάτι κώδικα λέγεται `reentrant` (επανεισχωρήσιμο) όταν μπορεί να εκτελείται από πολλαπλά `threads` ταυτόχρονα.
- Ορισμός: ένα κομμάτι κώδικα λέγεται `critical section` (κρίσιμο τμήμα), όταν δεν πρέπει να εκτελείται από περισσότερα του ενός `threads` ταυτόχρονα.
 - Τα κρίσιμα τμήματα πρέπει να εκτελούνται ατομικά.
- Αρα: στον πολυπρογραμματισμό, ο κώδικας αποτελείται από κομμάτια, που το καθένα είναι είτε `reentrant` είτε κρίσιμο τμήμα.

Reentrant κώδικας

- Αν σε ένα κομμάτι κώδικα, κάθε thread θα προσπελάσει ιδιωτική μόνο μνήμη, τότε αυτό είναι reentrant.
- Συνήθεις περιπτώσεις που κάνουν ένα κομμάτι κώδικα non-reentrant
 - Χρήση global μεταβλητών (πχ. errno).
 - Χρήση static μεταβλητών (πχ. buffers).
 - Κλήση non-reentrant συναρτήσεων.
- Αλλά ΠΡΟΣΟΧΗ: τα παραπάνω δεν κάνουν αυτόματα μια συνάρτηση non-reentrant!

Κρίσιμα τμήματα

- Τα κρίσιμα τμήματα είναι κομμάτια κώδικα που πρέπει να εκτελούνται ατομικά.
- Mutual exclusion (αμοιβαίος αποκλεισμός): μηχανισμός που χρησιμοποιεί ένα νήμα για να αποκλείσει σε άλλα νήματα την είσοδο σε κρίσιμο τμήμα.

Μηχανισμοί mutual exclusion

- Απενεργοποίηση `interrupts` (γενικά του `preemption`)
 - Μόνο στον scheduler του πυρήνα
 - Και επιλεκτικά σε device drivers (no 1 λόγος crash στα Windows).

- Locking (`locks`, `mutexes`, `semaphores`, `condition variables`) .
 - Η καλύτερη λύση
 - Αλλά, δε δουλεύει για device drivers/signal handlers.

Mutex

- Αρχικά του MUTual EXclusion.
- Γνωστά και ως locks (κλειδαριές).
- Έχουν δύο μεθόδους
 - Lock (ή acquire): κλείδωσε (απόκτησε)
 - Unlock (ή release): ξεκλείδωσε (αποδέσμευσε)
- Μόνο ένα thread μπορεί να κατέχει το mutex σε κάθε στιγμή.
 - Κλήση της lock σε δεσμευμένο mutex μπλοκάρει την διεργασία μέχρι το mutex να αποδεσμευτεί.

Παράδειγμα

```
int count=0;    /* static variable */  
Mutex m;  
...  
lock(&m);  
count = count + 1;  
unlock(&m);
```

Υλοποίηση mutex

```
int Mutex::trylock();
```

- Προσπαθεί να αποκτήσει το mutex
- Επιστρέφει 1 για επιτυχία, 0 για αποτυχία
- ΔΕΝ ΜΠΛΟΚΑΡΕΙ

```
class Mutex {  
    /* state!=0 when free  
    */  
    volatile int state;
```

Volatile: οδηγία προς τον compiler.

```
void lock() {  
    while( trylock() )  
        yield();  
}
```

```
void unlock() {  
    state = 1;  
}
```

Υλοποίηση της `trylock`

- Σε γλώσσα μηχανής, για ατομική εκτέλεση.
- Ειδική εντολή μηχανής (σε σύγχρονες CPUs)
 - TSL (Test and Set Lock), ή
 - TAS (Test And Set)
- Εναλλακτικά (σε μονοπύρηνια CPUs):
 - Εντολή EXCHANGE.

Παραλλαγές: busy waiting

- Επειδή η `yield` είναι ακριβή, `spin lock(MONO multicores)`.

```
void lock() {  
    int spin = 500; /* e.g. spin 500 times */  
    while(! trylock()) {  
        if(spin>0) spin--;  
        else yield();  
    }  
}
```

Ιδιότητες mutex

- (+) Πάρα πολύ φθηνή υλοποίηση (2-5 κύκλοι μηχανής) κλειδώματος/ξεκλειδώματος όταν δεν υπάρχει ανταγωνισμός μεταξύ νημάτων.
- (-) Μόνο το νήμα που κάλεσε lock μπορεί να καλέσει την unlock.
 - Αλλιώς υπάρχει περίπτωση race condition σε πολλαπλές CPUs.
 - Δες semaphores.
- (-) Κατάλληλα για μικρής μόνον διάρκειας αναμονή
 - Για μεγαλύτερη αναμονή, θέλουμε να μπλοκάρουμε τη διεργασία.

Μοντέλο μνήμης

Σε σύγχρονα συστήματα η σειρά μεταβολών στη μνήμη είναι ΔΙΑΦΟΡΕΤΙΚΗ από αυτήν του προγράμματος.

Παράδειγμα:

```
/* Αρχικά:  int x = 0,  y = 0; */
```

```
/* thread 1 */
```

```
x = 1;
```

```
y = 1;
```

```
/* thread 2 */
```

```
if(x==0 && y==1)
```

```
printf("It CAN happen!\n");
```

ΓΙΑΤΙ: Η **CPU** ή/και ο compiler μπορεί να αλλάξουν τη σειρά του εκτέλεσης του κώδικα.

Μοντέλο μνήμης: συνέχεια

- **Μοντέλο μνήμης:**

- Καθορίζει **φράγματα συγχρονισμού (synchronization barriers)**
 - πχ. mutex lock
- Οι τιμές των κοινών μεταβλητών γίνονται “ορατές” στα άλλα νήματα, όταν αυτά “διασχίζουν” το φράγμα.
- Γενικός κανόνας: ΠΑΝΤΑ να χρησιμοποιούνται mutex για προσπέλαση σε κοινά δεδομένα.

```
/* thread 1 */
mutex.lock();
x = 1;
y = 1;
mutex.unlock();
```

```
/* thread 2 */
mutex.lock();
if(x==0 && y==1)
    printf("It CANNOT happen!\n");
mutex.unlock();
```

- Προχωρημένες τεχνικές Java/C/C++:

- “atomic statements”
- πολύπλοκοι κανόνες (για υψηλής απόδοσης κώδικα)

Condition variables

- Κατάλληλα για μεγάλης διάρκειας αναμονή
- Μέθοδοι
 - **wait**(Mutex* m):
 - Ατομικά: ξεκλειδώνει το mutex m και κοιμίζει το νήμα στο condition variable cv.
 - Όταν το νήμα ξυπνά, πριν επιστρέψει ξανακλειδώνει το m.
 - **Signal()** (ή **notify**):
 - Ξυπνά *ένα* από τα κοιμισμένα νήματα του condition variable cv.
 - **Broadcast()** (ή **notifyAll**):
 - Ξυπνά *όλα* τα κοιμισμένα νήματα του condition variable cv.

Παράδειγμα

```
Mutex m;  
Condvar cv;  
bool critical=false;
```

Γιατί while και όχι if?

```
■ m.lock();           // try to enter C.S.  
  while(critical)     ■  
    cv.wait(&m);  
  critical = true;  
  m.unlock();
```

```
Long_Critical_Section(); // runs for a LONG time
```

```
m.lock();           // leave critical section  
critical = false;  
cv.signal();  
m.unlock();
```

Παράδειγμα (συν.)

- Μπορεί να γίνει απλούστερα;
- Μπορεί να γίνει αν η `wait` δεν εκτελεί ατομικά το «ξεκλείδωμα του `mutex` + κοίμισμα του νήματος»;
- Πότε κερδίζουμε πολύ σε χρόνο σε σχέση με τα `mutex`;
 - Π.χ. 1000 νήματα περιμένουν να μπουν σε ένα μακρύ `critical section`.

Υλοποίηση condition variables

```
class Condvar {
    Mutex mx;
    list<Pid_t> waitlist;

void wait(Mutex* l)
{
    mx.lock();
    waitlist.append(getpid());
    l->unlock();
    unlock_and_sleep(&mx);
    l->lock();
}
```

```
void signal()
{
    Pid_t pid=-1;
    mx.lock();
    if(! waitlist.empty())
        pid = waitlist.pop();
    mx.unlock();
    if(pid!=-1) wakeup(pid);
}

void broadcast()
{
    ... // άσκηση για τον
    // αναγνώστη
}
```

Άλλες δομές συγχρονισμού

- Semaphores
- Bounded buffers
- Barriers
- Read-write locks
- ...

Όλες υλοποιούνται από mutexes + condvars

Όλες υλοποιούνται ως monitors.

Monitors

- Hoare (1974) , Brinch-Hansen (1975) -μικροδιαφορές.
- Monitor = design pattern
- Ορισμός: Ένα αντικείμενο που
 - Έχει ένα μοναδικό mutex m.
 - Έναν αριθμό από (ιδιωτικά) condition variables.
 - Όλες οι public μέθοδοι:
 - Κλειδώνουν το mutex m αμέσως μετά την κλήση τους και το ξεκλειδώνουν αμέσως πριν την επιστροφή τους.
 - Όλες οι κλήσεις της wait σε condition variables, αναφέρονται στο mutex m και μόνον.

Monitors στη C++

```
class SomeClass {
    Mutex m;
    CondVar c; // μόνο ένα c.v.
    ...
public:
    int method1() {
        int ret;
        m.lock();
        ... // κώδικας
        m.unlock();
        return ret;
    };

    void method2(char* foo) {
        m.lock();
        ... // κώδικας
        m.unlock();
    };
};
```


Monitors στη Java

// ΚΑΘΕ αντικείμενο της Java είναι monitor

```
class SomeClass {  
    ...  
  
    synchronized public int method1() {  
        ...  
    }  
  
    synchronized public void method2(String a) {  
        ...  
    }  
}
```

Monitors στη C

- Αντίστοιχα με τη C++.
- **Monitor:** ένα σύνολο από συναρτήσεις της C, που
 - Όλες κλειδώνουν το ίδιο mutex *m* στην είσοδο και το ξεκλειδώνουν πριν την έξοδο.
 - Όλες οι κλήσεις των c.v. αφορούν το mutex *m*.
 - Το mutex *m* και τα c.v. δεν αναφέρονται σε άλλο κώδικα.

Semaphores

- Προτάθηκαν από τον Dijkstra (1965) .
- Μοιάζουν με `mutex`.
- Δύο λειτουργίες
- Ορολογία

Mutex		Semaphores			
lock	acquire	lock	acquire	down	P
unlock	release	unlock	release	up	V

Χρήση semaphores

- Φανταστείτε ένα **S ΣΑΚΟΥΛΙ** για φασόλια
 - **P: ΠΑΡΕ** ένα φασόλι (περίμενε αν πρέπει)
 - **V: ΒΑΛΕ** ένα φασόλι (δεν περιμένεις)
 - Σημείωση: ο αριθμός των φασολιών μπορεί να είναι αρνητικός (!)
- Αμοιβαίος αποκλεισμός:
 - Αρχικά, 1 φασόλι
 - Boolean semaphore
 - ΠΑΡΕ πριν το κρίσιμο τμήμα
 - ΒΑΛΕ μετά το κρίσιμο τμήμα

Semaphore s(1);

P(s);

... // critical section

V(s);

Υλοποίηση semaphores

Ως Monitor

```
class Semaphore {  
    int n;  
    Mutex m;  
    CondVar cv;  
};
```

```
void Semaphore(int N)  
{  
    n = N;  
}
```

```
void P() {  
    m.lock();  
    while( n <= 0 )  
        cv.wait(&m);  
    n--;  
    m.unlock();  
}
```

```
void V() {  
    m.lock();  
    n++;  
    cv.signal();  
    m.unlock();  
}
```

Χρήση Condition Variable

```
void P() {  
    m.lock();  
    while( n <= 0 )  
        cv.wait(&m);  
    n--;  
    m.unlock();  
}
```

```
void V() {  
    m.lock();  
    n++;  
    cv.signal();  
    m.unlock();  
}
```



Ερώτηση: γιατί `while` και όχι απλά `if` ?
Αποδείξτε ότι με `if` έχουμε race condition.

Μεθοδολογία Υλοποίησης

- **Monitor state**
 - Μεταβλητές
 - Αρχικοποίηση
- **Method = Condition + Action**
 - **Condition** : ένα boolean expression
 - **Action** (state change):
 - Αλλαγή στην τιμή του state.
- Για κάθε condition, ένα **Condition Variable**
 - trivial conditions?

Semaphore state

- `int n`
 - Initially $n = N$

Method	Condition	Action
P	$n > 0$	$n--;$
V	true	$n++;$

Template μεθόδου monitor

```
ReturnType  MethodName(args ...)  
{  
    ReturnType retval;  
    mutex.lock();  
        1. while( !(Condition) )  condvar.wait(&mutex);  
        2. Action ...(change state, compute retval)  
        3. Signal or Broadcast on changed conditions  
    mutex.unlock();  
    return retval;  
}
```


Template: υλοποίηση semaphores

```
class Semaphore {  
    int n;  
    Mutex m;  
    CondVar cv;  
  
    Semaphore(int N) {  
        n = N;  
    }  
}
```

```
void P() {  
    m.lock();  
    while( n<=0 )  
        cv.wait(&m);  
    n--;  
    /* no signalling */  
    m.unlock();  
}
```

```
void V() {  
    m.lock();  
    /* no condition */  
    n++;  
    cv.signal();  
    m.unlock();  
}
```

Reader-writer locks

- Παράδειγμα χρήσης: ένα `array`
- Χρειάζεται αμοιβαίος αποκλεισμός ανάμεσα σε:
 - Νήματα που μόνο διαβάζουν από το `array`;
 - Νήματα που γράφουν στο `array`;
 - Νήματα που άλλα διαβάζουν και άλλα γράφουν;
- Θέλουμε ένα `lock` που να επιτρέπει ταυτόχρονη ανάγνωση από πολλαπλά νήματα.
 - **rlock**: κλείδωσε ως reader
 - **wlock**: κλείδωσε ως writer
 - **unlock** (ή `runlock/wunlock`): ξεκλείδωσε (ως τί;)
- Εναλλακτική ορολογία: **Shared (SH-lock)** / **Exclusive (EX-lock)**

Σχεδίαση Reader-Writer Locks

State:

```
int takers    // takers > 0    - number of readers holding lock
              // takers == 0    - lock is free
              // takers == -1   - writer holds lock
```

Method	Condition	Action
rlock	takers >= 0	takers ++
wlock	takers == 0	takers = -1
unlock	true	if(takers>0) takers -- ; else takers = 0 ;

Πρώτη υλοποίηση RWLocks (monitor)

```
class RWLock {
    int takers;
    Mutex m;
    CondVar cv;

    RWLock() {
        takers = 0;
    }

    void rlock() {
        m.lock();
        while(takers<0)
            cv.wait(&m);
        takers++;
        m.unlock();
    }
}
```

```
void wlock() {
    m.lock();
    while(takers!=0)
        cv.wait(&m);
    takers = -1;
    m.unlock();
}

void unlock() {
    m.lock();
    if(takers>0) takers--;
    else takers=0;
    if(takers==0)
        cv.broadcast();
    m.unlock();
}
```

Πρόβλημα: δικαιοσύνη

- Πρόβλημα: οι `writers` μπορεί να μπλοκάρονται εσασεί από τους `readers`.
 - Πχ. μια δημοφιλής database
- Λύση: οι `writers` να έχουν προτεραιότητα.

Βελτίωση Reader-Writer Locks

State:

```
int takers    // takers > 0    – number of readers holding lock
              // takers == 0   – lock is free
              // takers == -1  – writer holds lock
int w         // number of waiting writers
```

rlock	takers >= 0 && w==0	takers ++
wlock	true	W ++
	takers == 0	takers = -1 ; w--
unlock	true	if(takers>0) takers -- ; else takers ++ ;

Υλοποίηση RWLocks με προτεραιότητα στους writers

```
class RWLock {  
    int takers; //as before  
    int w; //waiting writers  
    Mutex m;  
    CondVar rcv, wcv;
```

```
    RWLock() {  
        takers=0;  
        w = 0;  
    }
```

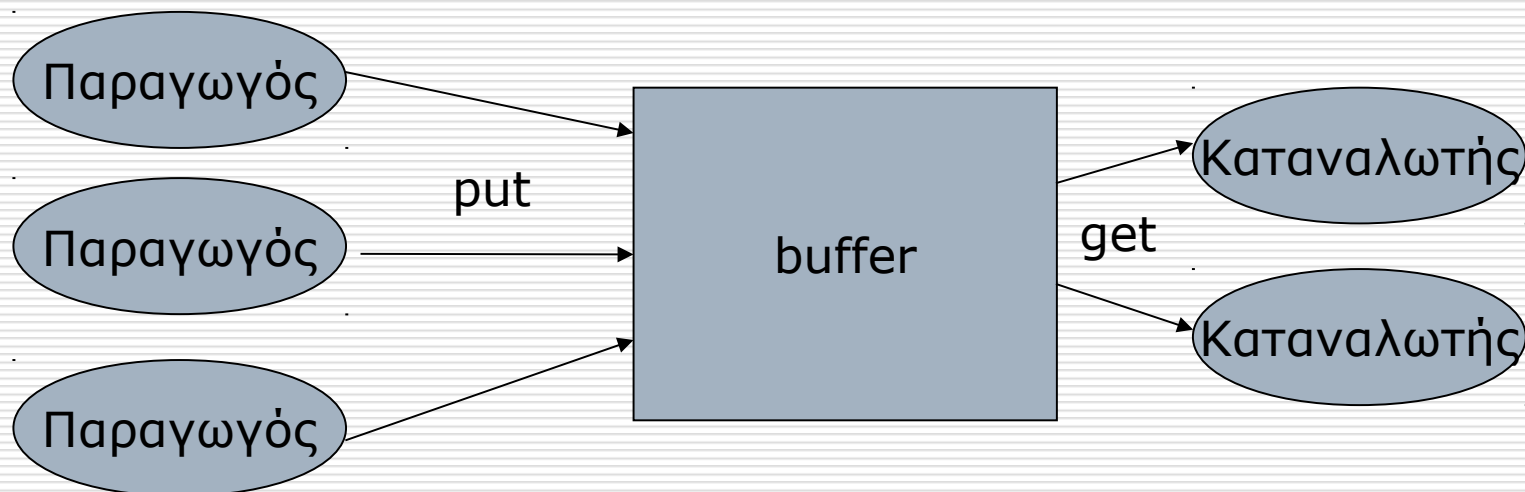
```
    void rlock() {  
        m.lock();  
        while(takers<0 || w>0)  
            rcv.wait(&m);  
        takers++;  
        m.unlock();  
    }
```

```
    void wlock() {  
        m.lock();  
        w++;  
        while(takers!=0)  
            wcv.wait(&m);  
        w--;  
        takers = -1;  
        m.unlock();  
    }
```

```
    void unlock() {  
        m.lock();  
        if(takers>0) takers--;  
        else takers=0;  
        if(takers==0) {  
            if(w>0) wcv.signal();  
            else rcv.broadcast();  
        }  
        m.unlock();  
    }
```

Buffers – producer/consumer

- Οι **buffers** είναι από τις πιο κεντρικές δομές σε ΛΣ
 - Είσοδος-έξοδος
 - Ασύγχρονη επικοινωνία (θυμηθείτε ορισμό «ασύγχρονης λειτουργίας»)
- Στη γενική περίπτωση έχουμε:



Είδη buffers

- **Container που υλοποιεί τον buffer:**
 - Αποθηκεύει κάποιο τύπο `T` (byte, int, pointer, κλπ)
 - Μέγιστο μέγεθος φραγμένο (πχ. array)
 - `C.push(T x)`: εισάγει το `x` στο container
 - `C.pop()`: εξάγει και επιστρέφει κάποιο στοιχείο
 - `C.full()` και `C.empty()`: boolean μέθοδοι

- **Παραδείγματα container:**
 - Queue (First In First Out – FIFO)
 - Stack (Last In First Out – LIFO)
 - Heap (Smallest out)
 - ...

Method	Condition	Action
put(x)	! C.full()	C.push(x)
get()	! C.empty()	retval = C.pop()

Παράδειγμα: ring buffer (κυκλικός buffer) χαρακτήρων

```
class Buffer {
    char buf[SIZE];
    int head, nelem;
    Mutex m;
    CondVar prod, cons;

    Buffer() head(0), nelem(0) {}

    void put(char c) {
        m.lock();
        while(nelem==SIZE)
            prod.wait(&m);
        buf[(head+nelem)%SIZE]=c;
        nelem++;
        cons.signal();
        unlock(&m);
    }

    char get() {
        char ret;
        m.lock();
        while(nelem==0)
            cons.wait(&m);
        ret=buf[head];
        head=(head+1)%SIZE;
        nelem--;
        prod.signal();
        m.unlock();
        return ret;
    }
}
```

Bounded buffers και semaphores

- Δύο semaphores
- Sfull : μετρά τις “γεμάτες θέσεις”
 - Αρχικά, Sfull \leftarrow 0
- Sempty: μετρά τις “άδειες θέσεις”
 - Αρχικά, Sempty \leftarrow SIZE

```
Container C(SIZE);
Semaphore Sfull(0), Sempty(SIZE);

void put(x) {
    Sempty.P();
    C.push(x); // εισαγωγή στο buffer
    Sfull.V();
}

char get() {
    char ret;
    Sfull.P();
    ret = C.pop();
    Sempty.V();
}
```

Reentrant locks

- Αν ένα νήμα προσπαθήσει να κλειδώσει ένα Mutex που το ίδιο έχει κλειδώσει προηγουμένως:
 - Deadlock
- Αν μια μέθοδος monitor καλείται μέσα από μια άλλη???
- Λύση: **reentrant locks (rlocks)**
 - ή **recursive locks**
 - Ένα ελεύθερο rlock δεν έχει ιδιοκτήτη.
 - Το νήμα που κλειδώνει ένα ελεύθερο rlock γίνεται ιδιοκτήτης
 - Ο ιδιοκτήτης μπορεί να κλειδώσει το rlock πολλές φορές
 - Για να ελευθερώσει το rlock ο ιδιοκτήτης,
 - Κλήσεις `unlock` == κλήσεις `lock`, ή
 - Μια κλήση της `disown`

Υλοποίηση reentrant locks

```
class RLock {
    Mutex m;
    volatile int count=0;
    volatile Pid_t owner=NO_PROC;

void unlock() {
    m.lock();
    count--;
    if(count==0) owner=NO_PROC;
    m.unlock();
}

int disown() {
    int retval;
    m.lock();
    retval = count;
    count=0;
    owner = NO_PROC;
    m.unlock();
    return retval;
}
```

```
void own(int c) {
    m.lock();
    while(owner!=NO_PROC &&
        owner!=GetPid())
    {
        m.unlock();
        yield();
        m.lock();
    }
    owner=GetPid();
    count += c;
    m.unlock();
}

void lock() { own(1); }
};
```

Reentrant locks και monitors

- Παραλλαγή monitor:
 - Αντί για mutex, reentrant lock.
 - Επιτρέπει σε μια μέθοδο του monitor να καλεί μια άλλη.
- Προσοχή στα `condition variables`
 - Για τη `wait`, νέα υλοποίηση

```
void Condvar::wait(RLock* l)
{
    int count;
    mx.lock();
    waitlist.append(getpid());
    count = l->disown(l);
    unlock_and_sleep(mx);
    l->own(count);
}
```

PThreads και δομές συγχρονισμού

Τύπος	API χειρισμού
<code>pthread_mutex_t</code>	<code>pthread_mutex_init(mx, attr)</code> <code>pthread_mutex_destroy(mx)</code> <code>pthread_mutex_lock(mx)</code> <code>pthread_mutex_unlock(mx)</code> <code>pthread_mutex_trylock(mx)</code>
<code>pthread_mutexattr_t</code>	<code>pthread_mutexattr_init(attr)</code> <code>pthread_mutexattr_destroy(attr)</code> <code>pthread_mutexattr_settype(attr, type)</code> <code>pthread_mutexattr_gettype(attr, *type)</code> type: <code>PTHREAD_MUTEX_NORMAL</code> <code>PTHREAD_MUTEX_RECURSIVE</code>

PThreads και δομές συγχρονισμού

Τύπος	API χειρισμού
<code>pthread_cond_t</code>	<code>pthread_cond_init(cv, attr)</code> <code>pthread_cond_destroy(cv)</code> <code>pthread_cond_wait(cv, mx)</code> <code>pthread_cond_timedwait(cv, mx, t)</code> <code>struct timespec *t;</code> <code>pthread_cond_signal(cv)</code> <code>pthread_cond_broadcast(cv)</code>
<code>pthread_condattr_t</code>	Ειδικές μόνο περιπτώσεις (συνήθως <code>attr==NULL</code>)

PThreads και δομές συγχρονισμού

Τύπος	API χειρισμού
<code>pthread_rwlock_t</code>	<code>pthread_rwlock_init(rw, attr)</code> <code>pthread_rwlock_destroy(rw)</code> <code>pthread_rwlock_rdlock(rw)</code> <code>pthread_rwlock_tryrdlock(rw)</code> <code>pthread_rwlock_timedrdlock(rw, t)</code> <code>pthread_rwlock_wrlock(rw)</code> <code>pthread_rwlock_trywrlock(rw)</code> <code>pthread_rwlock_timedwrlock(rw, t)</code> <code>pthread_rwlock_unlock(rw)</code>
<code>pthread_rwlockattr_t</code>	Ειδικές μόνο περιπτώσεις (συνήθως <code>attr==NULL</code>)

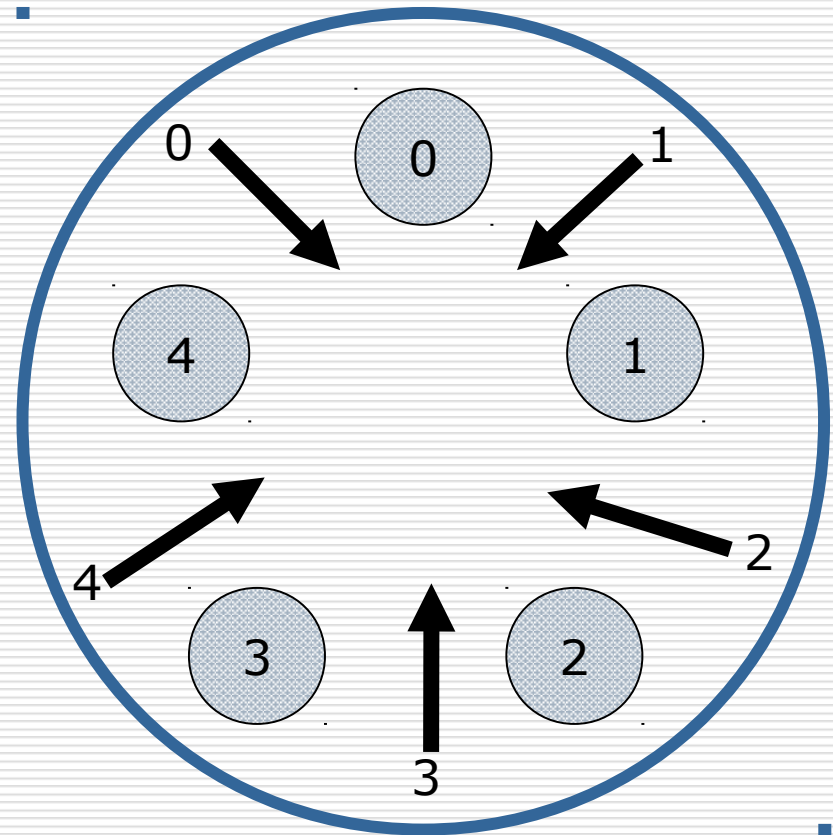
Προχωρημένος πολυπρογραμματισμός

- Πέρα από τον αμοιβαίο αποκλεισμό
 - Deadlock
 - Fairness (δικαιοσύνη)
 - Scheduling

Dining philosophers

Πρόβλημα

- Πέντε φιλόσοφοι γευματίζουν.
 - Ο καθένας περνά από τις φάσεις
 - THINKING
 - HUNGRY
 - EATING
 - Για να φάει ένας φιλόσοφος, πρέπει να πάρει και τα δύο κουτάλια.
-
- Κουτάλια = `mutexes`
 - Φιλόσοφοι = `νήματα`
 - Γενίκευση: N φιλόσοφοι



Δομή της λύσης

```
#define N 5    // αριθμός φιλοσόφων
#define PH(i)  ((i)%N)

void philosopher(int i)
{
    think(i);
    takeforks(i);    // πάρε τα πηρούνια
    eat(i);
    leaveforks(i);   // άφησε τα πηρούνια
}
```

Προβληματική λύση 1

```
Mutex fork[N];

void takeforks(int i) {
    fork[i].lock();
    fork[(i+1)%N].lock();
}

void leaveforks(int i) {
    fork[(i+1)%N].unlock();
    fork[i].unlock();
}
```

DEADLOCK

Προβληματική λύση 2

```
Mutex fork[N];

void takeforks(int i) {
    while(1) {
        fork[i].lock();

        if(fork[(i+1)%N].trylock())
            break;
        else
            fork[i].unlock();
    }
}
```

```
void leaveforks(int i) {
    fork[(i+1)%N].unlock();
    fork[i].unlock();
}
```

LIVELOCK

Προβληματική λύση 3

```
Mutex m;
```

```
void takeforks(int i) {  
    m.lock();  
}
```

```
void leaveforks(int i) {  
    m.unlock();
```

```
}
```

Χαμηλός
παραλληλισμός

Προβληματική λύση 4

```
Mutex m; CondVar ph[N];    void leaveforks(int i) {
enum PHIL {T, H, E};        m.lock();
PHIL s[N]={T,...,T}; //state s[i]=T;
                             ph[(i-1)%N].signal();
                             ph[(i+1)%N].signal();
                             m.unlock();
void takeforks(int i) {
    m.lock();
    s[i] = H;
    trytoeat(i);
    while(s[i]!=E) {
        ph[i].wait(&m);
        trytoeat(i);
    }
    m.unlock();
}

void trytoeat(int i) {
    if(s[i]==H && s[(i-1)%N]!=E
        && s[(i+1)%N]!=E)
        s[i]=E;
}
```

Starvation

Σωστή λύση

```
Mutex m; CondVar ph[N];  
enum PHIL {T, H, E};  
PHIL s[N]={T,...,T}; //state
```

```
void takeforks(int i) {  
    m.lock();  
    s[i] = H;  
    trytoeat(i);  
    if (s[i] != E)  
        ph[i].wait(&m);  
    m.unlock();  
}
```

To while είναι σωστότερο

```
void leaveforks(int i) {  
    m.lock();  
    s[i]=T;  
    trytoeat((i-1)%N);  
    trytoeat((i+1)%N);  
    m.unlock();  
}
```

```
void trytoeat(int i) {  
    if (s[i]==H && s[(i-1)%N]!=E  
        && s[(i+1)%N]!=E)  
    {  
        s[i]=E;  
        ph[i].signal();  
    }  
}
```

Deadlock

- Ορισμός: Έχουμε deadlock σε ένα σύνολο S από νήματα, όταν κάθε νήμα περιμένει, για να προχωρήσει, μια ενέργεια (unlock, signal κλπ) που μόνο κάποιο άλλο νήμα του συνόλου S μπορεί να εκτελέσει.

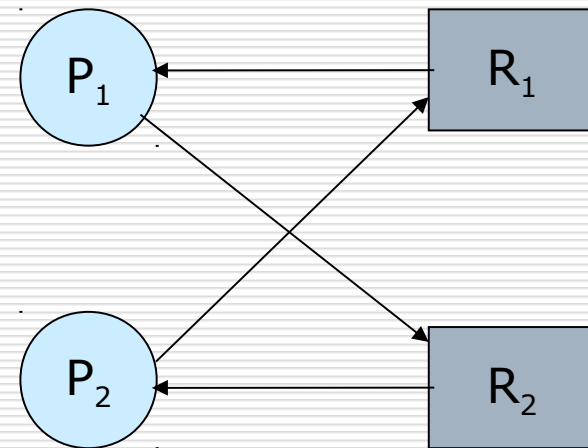
- Άρα:
 - Στο deadlock, όλα τα εμπλεκόμενα νήματα είναι «σταματημένα».
 - Μπορεί να εκτελούν κάποιο busy wait loop.
 - Κανένα νήμα δεν θα επανεκινήσει.

Αντιμετώπιση deadlock

- Στρουθοκαμηλισμός (να μην κάνουμε τίποτε).
 - Βασίζεται στο ότι (υποθετικά) deadlocks συμβαίνουν σπάνια.
 - Απλή υλοποίηση και γρήγορη εκτέλεση... ☺
 - Ματαθέτει το πρόβλημα σε άλλους (restart required...)
 - ΣΥΧΝΗ ΜΕΘΟΔΟΣ ΣΤΗΝ ΠΡΑΞΗ !!! :-)
- Ανίχνευση και θεραπεία.
 - Η ανίχνευση εισάγει overhead.
 - Η θεραπεία μπορεί να είναι:
 - Επανακατανομή των πόρων
 - Τερματισμός κάποιου(ων) νημάτος(ων)
- Πρόληψη.
 - Στατική: να γράφουμε **deadlock-free** κώδικα
 - Δυναμική: να έχουμε έναν “έξυπνο” scheduler
 - Δεν είναι πάντοτε εφικτή...

Ανίχνευση deadlock (διακριτοί πόροι)

- Έστω
 - νήματα P_1, \dots, P_k
 - πόροι R_1, \dots, R_m
- Γράφος $G(t)$: σε χρόνο t ,
 - $P_i \rightarrow R_j$: το νήμα P_i περιμένει τον πόρο R_j (πχ. ένα Mutex).
 - $R_j \rightarrow P_i$: ο πόρος R_j έχει αποκτηθεί από το νήμα P_i .
- Ο γράφος αλλάζει με το χρόνο.
- ΘΕΩΡΗΜΑ: Υπάρχει deadlock στο χρόνο t , αν και μόνο αν ο $G(t)$ περιέχει κύκλο.



Deadlock!

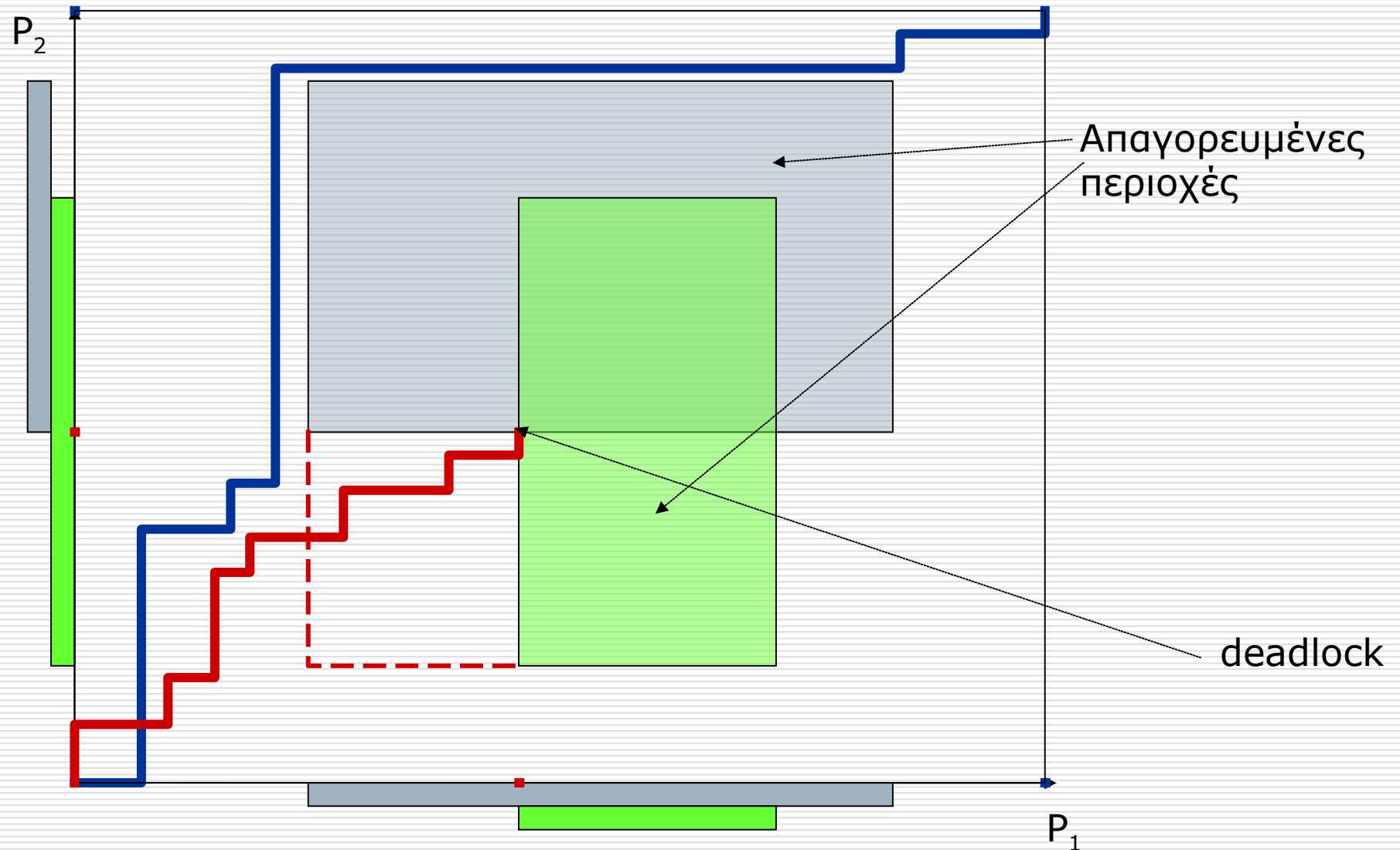
Ανίχνευση deadlock (μη διάκριτοι πόροι)

- Έστω n νήματα και m είδη πόρων
 - μη διακριτοί: πχ. σελίδες μνήμης, ελεύθερα pid, ...
- Σε κάποια χρονική στιγμή,
 - A_i : αριθμός ελεύθερων πόρων τύπου i
 - C_{ik} : αριθμός πόρων τύπου i δοσμένων στο νήμα k .
 - R_{ik} : αριθμός πόρων τύπου i που ζητά το νήμα k .
- Αλγόριθμος:
 - Αρχικά, $S = \{1, \dots, n\}$.
 - Ψάξε για $k \in S$ όπου $R_{ik} \leq A_i$ για κάθε $i=1..m$.
 - Αν υπάρχει,
 - $S := S - \{k\}$, $A_i := A_i + C_{ik}$, επανέλαβε (2),
 - αλλιώς τερμάτισε.
- Όλα τα νήματα που παραμένουν στο S είναι **deadlocked**.

Πρόληψη

- Δεν είναι πάντα εφικτή.
- Δυναμικά
 - Όταν κάθε νήμα “ξέρει” εκ των προτέρων τις τελικές του απαιτήσεις
 - Αλγόριθμος του “τραπεζίτη”.
- Στατικά
 - Όταν οι πόροι μπορούν να αποκτηθούν με δεδομένη σειρά.

Resource Trajectories



Αλγόριθμος τραπεζίτη

- Έστω n νήματα, m είδη πόρων
 - N_{ik} η μέγιστη μελλοντική απαίτηση πόρων τύπου i από το νήμα k .
 - Δεν γνωρίζουμε τους χρόνους/τη σειρά αίτησης και απελευθέρωσης των πόρων αυτών.
 - C_{ik} : αριθμός πόρων τύπου i στο νήμα k .
 - A_i : αριθμός ελεύθερων πόρων τύπου i .
- Αλγόριθμος τραπεζίτη: έλεγξε αν υπάρχει ενδεχόμενο deadlock
 - Αρχικά, $S = \{1, \dots, n\}$.
 - Ψάξε για $k \in S$ όπου $A_i \leq N_{ik}$ για κάθε $i=1..m$
 - Αν υπάρχει,
 - $S := S - \{k\}$, $A_i := A_i + C_{ik}$, επανέλαβε (2),
 - αλλιώς τερμάτισε.
- Αν $S \neq \emptyset$, υπάρχει ενδεχόμενο deadlock.

Διάταξη πόρων

- Έστω πόροι R_1, \dots, R_m .
- Κάθε νήμα δεσμεύει πόρους με αύξουσα σειρά,
 - π.χ. R_2, R_7, R_{11} .
- Θεώρημα: Δεν υπάρχει deadlock.
- Πρόβλημα: Μπορεί να μην είναι εφικτός στην υλοποίηση.
 - π.χ. βάση δεδομένων

- Εφαρμογή: Dining philosophers?

The Nested-Monitor (non?) Problem

Ιστορία:

- Ο A. Lister αναρωτήθηκε (CACM, 1974) τι συμβαίνει όταν ένα νήμα που έχει εισέλθει στο monitor A θέλει, πριν βγεί, να εισέλθει στο monitor B, όπου μπορεί να μπλοκάρει.
 - Περ.1: ξεκλειδώνει τα mutex των A και B
 - δύσκολη υλοποίηση και πιθανότητα deadlock!
 - Περ.2: ξεκλειδώνει μόνο το mutex του B.
 - πιθανότητα deadlock!
- Ο D. Parnas απάντησε ότι είναι "μη-πρόβλημα"
 - *"Δεν πρέπει να συγχέουμε τις απαιτήσεις (requirements—εδώ, ακεραιότητα των δεδομένων) με την υλοποίηση (εδώ, αμοιβαίος αποκλεισμός)"*
- Άλλες απαντήσεις (SIGOPS 1975 και μετά)
- Συμπέρασμα: καμιά καλή λύση!
 - άρα: δεν το κάνουμε!



Scheduling / Δρομολόγηση

Βασικές έννοιες

- Δρομολόγηση = η κατανομή της/των CPU στα νήματα των διεργασιών.
- Preemptive και non-preemptive scheduling.
 - Non-preemptive: διεργασία καλεί yield (αν "θέλει"!).
 - Preemptive: διεργασία μπορεί να διακοπεί από τον scheduler
 - Quantum (preemptive multitasking):
 - Μικρό quantum: καλύτερο responsiveness
 - Μεγάλο quantum: καλύτερο locality
 - quantum = 1: non-preemptive scheduling.
- Batch και online scheduling: ορισμοί.
- Είδος φόρτου
 - Compute-bound εκτέλεση: μεγάλα διαστήματα χωρίς I/O
 - I/O-bound εκτέλεση: συχνά I/O.

Στόχοι του scheduling

- **Fairness:** Κατανομή της/των CPU στα νήματα. Στόχος: όχι «ξεχασμένα» νήματα.
- **CPU Utilization:** κατανομή των νημάτων σε CPUs. Στόχος: όχι αδρανείς CPUs
- **Load balancing:** κατανομή έργου. Στόχος: απασχόληση όλου του συστήματος (πχ. περιφερειακά).
- **Policy enforcement:** προτεραιότητες, εγγυήσεις realtime κλπ.
- **Turnaround time (batch):** χρόνος μεταξύ υποβολής και ολοκλήρωσης.
- **Throughput (batch):** ρυθμός εκτέλεσης jobs
- **Response time (online):** καθυστέρηση απόκρισης στο χρήστη.

Αρμοδιότητες του scheduler

- **Scheduling** ☺
 - Σύμφωνα με τις ανάγκες
- **Signal dispatch**
 - Έκτακτη εκτέλεση signal handlers
- Υποστήριξη για δομές συγχρονισμού (**Mutexes/Condition variables**)
 - yield
 - ατομική εκτέλεση wait
 - Linux system calls:
 - "futex" – υποστήριξη για condition vars
 - "sched_yield" – current thread yields CPU
 - "getpriority/setpriority" – get/set niceness
 - "sched_*" syscalls – rtfm

Batch scheduling

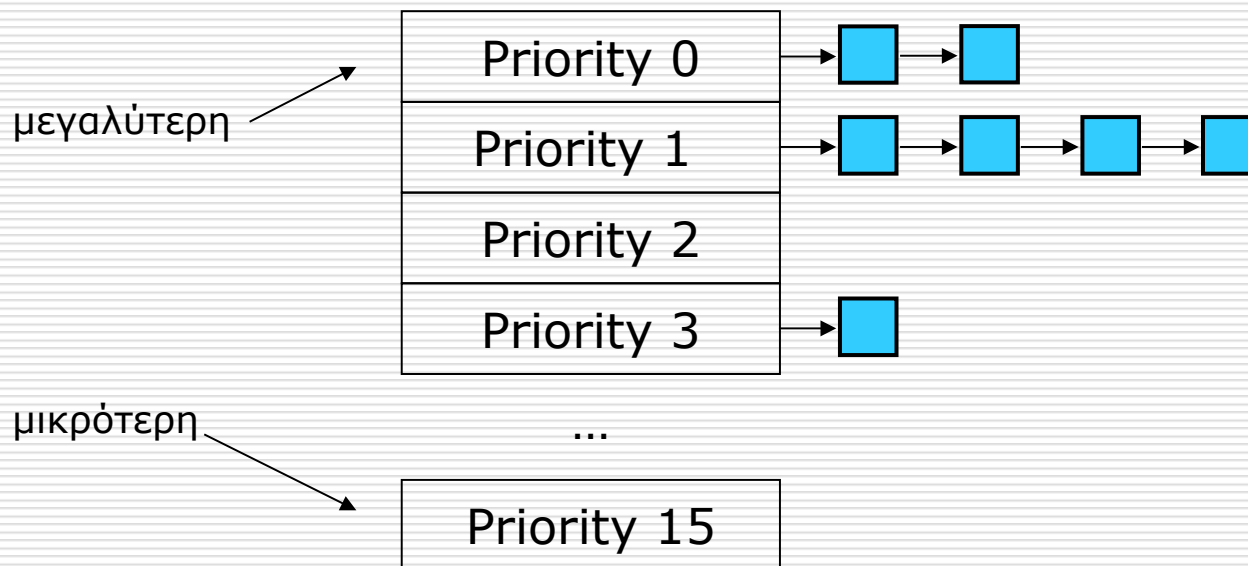
- Δρομολόγηση εργασιών.
- Ορίζουμε
 - $W_i(t)$: Συνολικός χρόνος παρουσίας εργασίας i (αναμονή+εκτέλεση)
 - S_i : Συνολικός χρόνος εκτέλεσης εργασίας i
 - $E_i(t)$: Χρόνος εκτέλεσης εργασίας i τη στιγμή t .
- **Αλγόριθμοι για batch λειτουργία**
 - FCFS (First Come First Served)
 - ή FIFO (First In First Out)
 - RR (Round-Robin)
 - SJF (Shortest Job First)
 - HRR (Highest Response Ratio)
 - SRT (Shortest Remaining Time)

Batch scheduling

Αλγόριθμος	Pre-emptive	Επιλογή	Πλεονέκτημα	Μειονέκτημα
FCFS	όχι	$\max(W_i(t))$	Απλή υλοποίηση, fairness	Μέτριο turnaround time
RR	ναι	Εκ περιτροπής	Απλή υλοποίηση, fairness	Μέτριο response time.
SJF	όχι	$\min(S_i)$	Απλή υλοποίηση, βέλτιστο turnaround time	Εκτίμηση S_i . Starvation.
HRR	όχι	$\max(W_i(t)/S_i)$	Πολύ καλό turnaround time	Εκτίμηση S_i .
SRT	ναι	$\min(S_i - E_i(t))$	Response time, fairness	Εκτίμηση S_i .

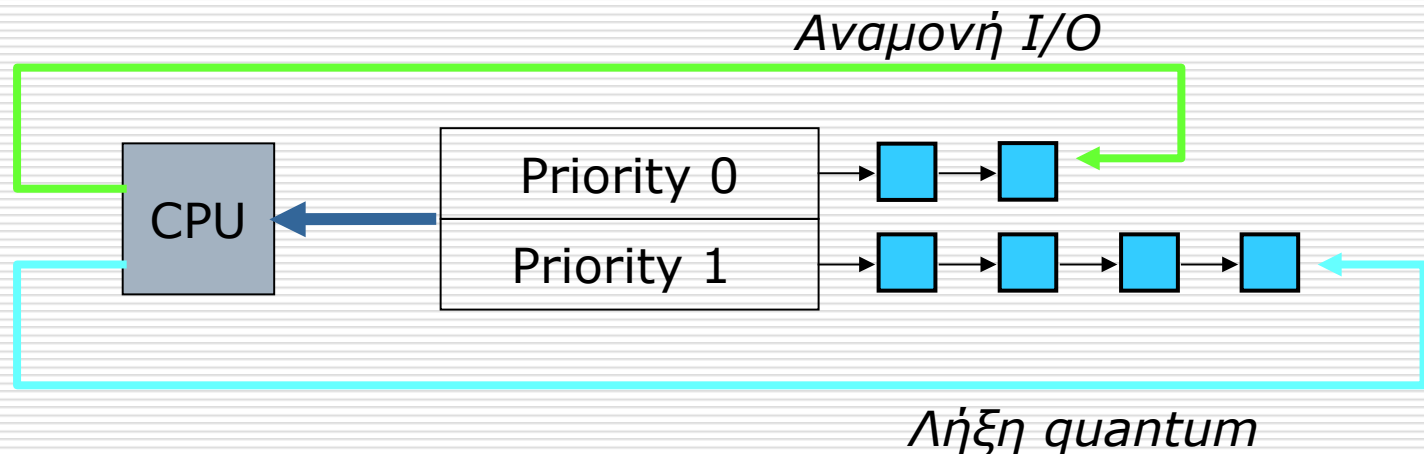
Online scheduling

- Προτεραιότητες
 - P_i : ακέραιος (ίσως και $\cdot 0$).
 - Δρομολογείται μια από τις διεργασίες με την «υψηλότερη» προτεραιότητα ($\max(P_i)$ ή $\min(P_i)$).
 - Υλοποίηση: πολλαπλές ουρές (άνυσμα από ουρές).



Παραλλαγές

- Round-Robin: όταν υπάρχει μόνο μια κλάση προτεραιοτήτων.
- Δυναμική: όταν η προτεραιότητα αλλάζει δυναμικά
 - CPU-bound διεργασίες σε χαμηλότερη προτεραιότητα.
 - I/O-bound διεργασίες σε υψηλότερη προτεραιότητα.
- Μια διεργασία που καταλαμβάνει όλο το quantum, χάνει προτεραιότητα (CPU-bound).
- Μια διεργασία που κοιμάται περιμένοντας I/O, κερδίζει προτεραιότητα.



Scheduling στην πράξη

- **Heuristic** (δεν υπάρχει αποδεκτό **standard**).
- **Linux/Solaris/WinNT**: Παραλλαγές δυναμικών προτεραιοτήτων.
- Λαμβάνουν υπόψιν
 - SMP
 - Χρήστη
 - Παλαιότητα εργασίας
 - ...



Υλοποίηση Scheduler

Ενδεικτικό API scheduler

- `spawn(newthread, func)` : αρχικοποιεί νέο νήμα
- `yield()` : καλεί τον scheduler
- `sleep(mutex)` : κοιμίζει το τρέχον νήμα
- `exit()` : τερματίζει το τρέχον νήμα
- `wakeup(thr_id)` : αφυπνίζει κάποιο νήμα
- `preemption_off()/preemption_on()` : (απ)ενεργοποίηση interrupts και κάθε διακοπής στην εκτέλεση

- Επίσης, ο scheduler χρησιμοποιεί μια “ουρά” (που μπορεί να είναι περίπλοκη δομή, με προτεραιότητες, κλπ)

Thread control block

```
enum State { INIT, READY, RUNNING, STOPPED, EXITED };
```

```
struct TCB {  
    ucontext_t context;  
    State state;  
    Bool dirty;    /* is this context up to date? */  
  
    TCB *prev, *next; /* for yield/gain */  
  
    /* other data */  
    PCB* owner_process;  
    Int priority;  
    ...  
};
```

Γενική λογική scheduler

“Yield” {

```
void yield() {  
    preempt_off();           /* disable interrupts */  
    curthread.next  
        = select_next();     /* scheduling algorithm */  
    currthread.next.prev = curthread;  
  
    if (curthread != curthread.next) /* do the swap */  
        swapcontext();  
  
    gain();  
}
```

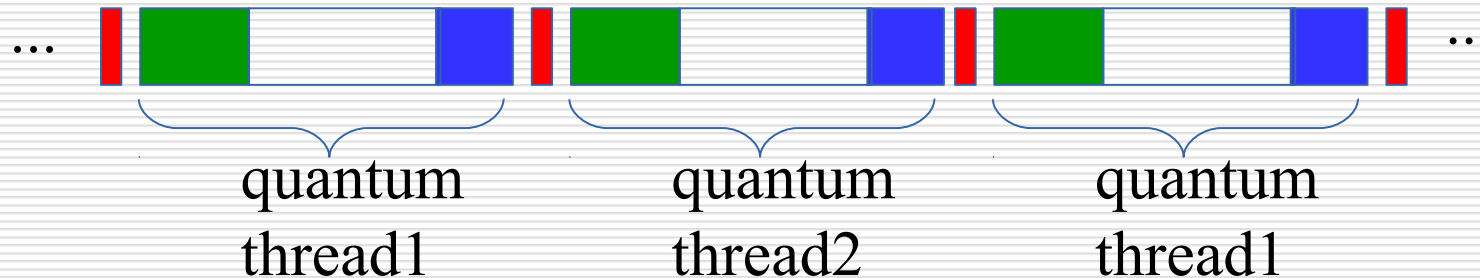
“Gain” {

```
void gain()  
{  
    if (curthread.prev.state == READY)  
        add_to_sched_list();  
    reset_quantum_timer(); /* preemptive sched */  
    preempt_on();          /* enable interrupts */  
}
```

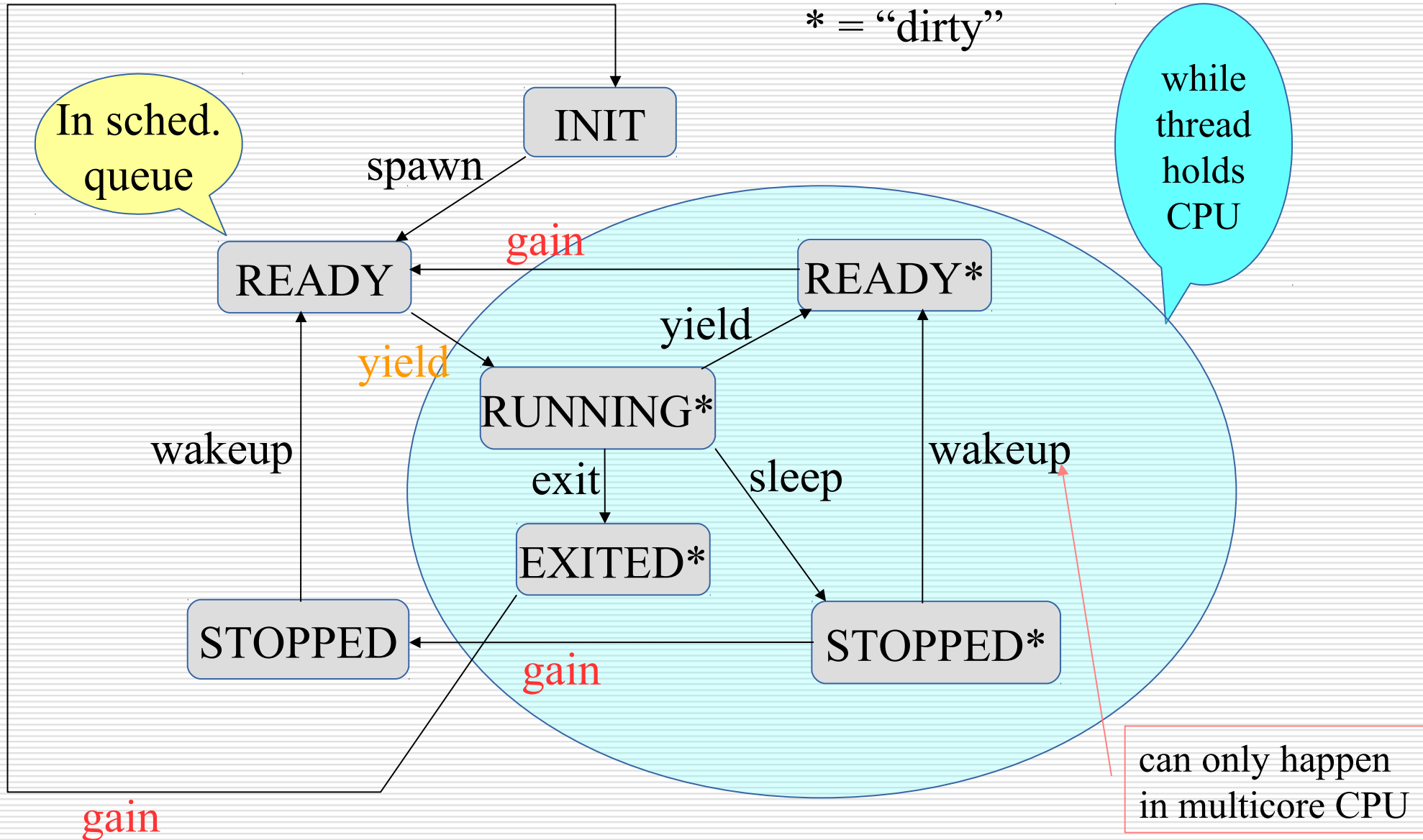
Εικόνα στο χρόνο

Ένα "quantum":

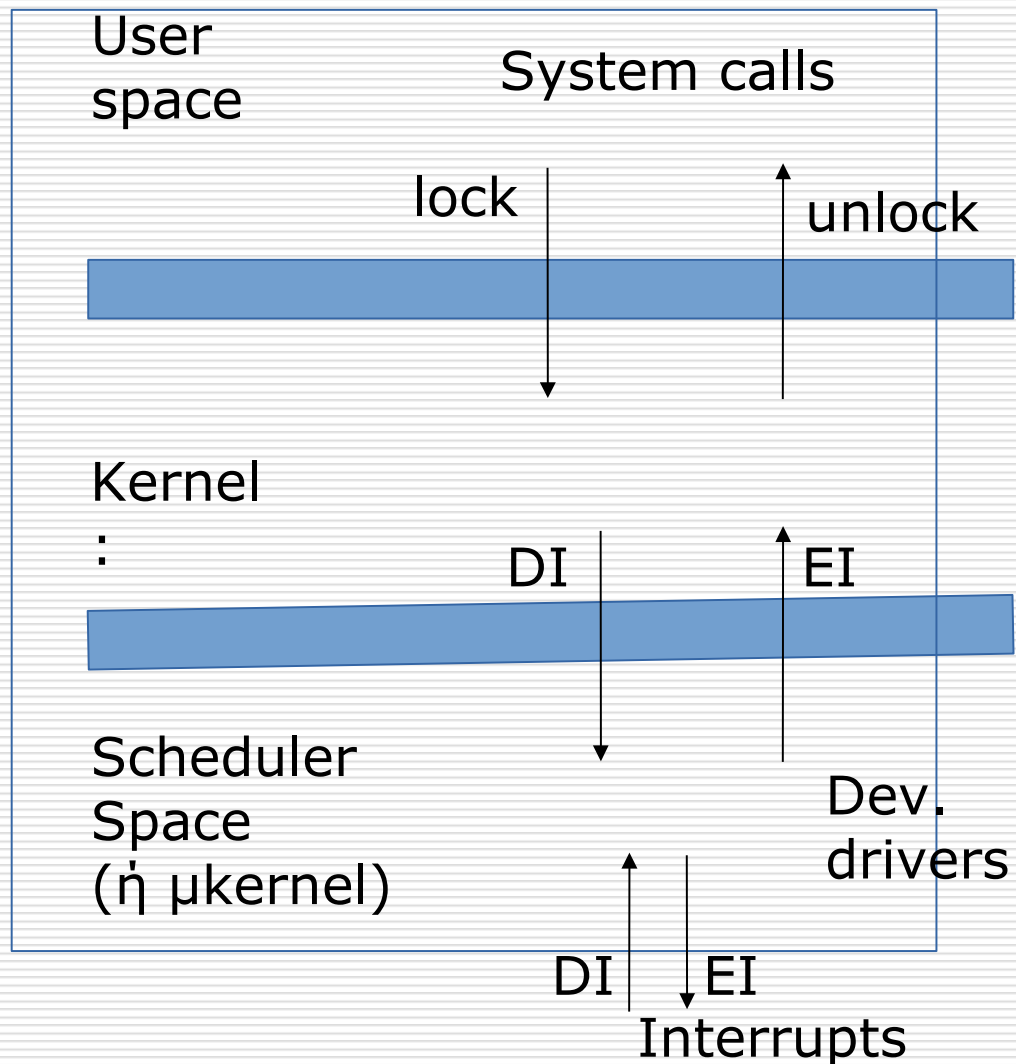
- είναι ο χρόνος ανάμεσα σε 2 **context switch**
- αρχίζει με **gain**
- τελειώνει με **yield**



Thread state transitions



Συγχρονισμός: 1 πυρήνας



- Η cpu εκτελεί:
 - Νήματα χρήστη
 - Interrupts
- 3 χώροι (shared data)
 - User space
 - Kernel (πλην scheduler)
 - Scheduler
- Είδη αμοιβαίου αποκλεισμού
 - **Thread-safe:** lock/unlock
 - **Async-call safe:** disable/enable preemption
 - **και τα 2** παραπάνω...

Self-deadlock από preemption

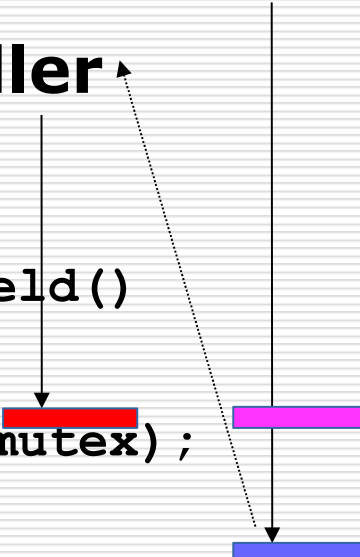
ThreadA

IntHandler

```
void yield()
{
    lock(mutex);

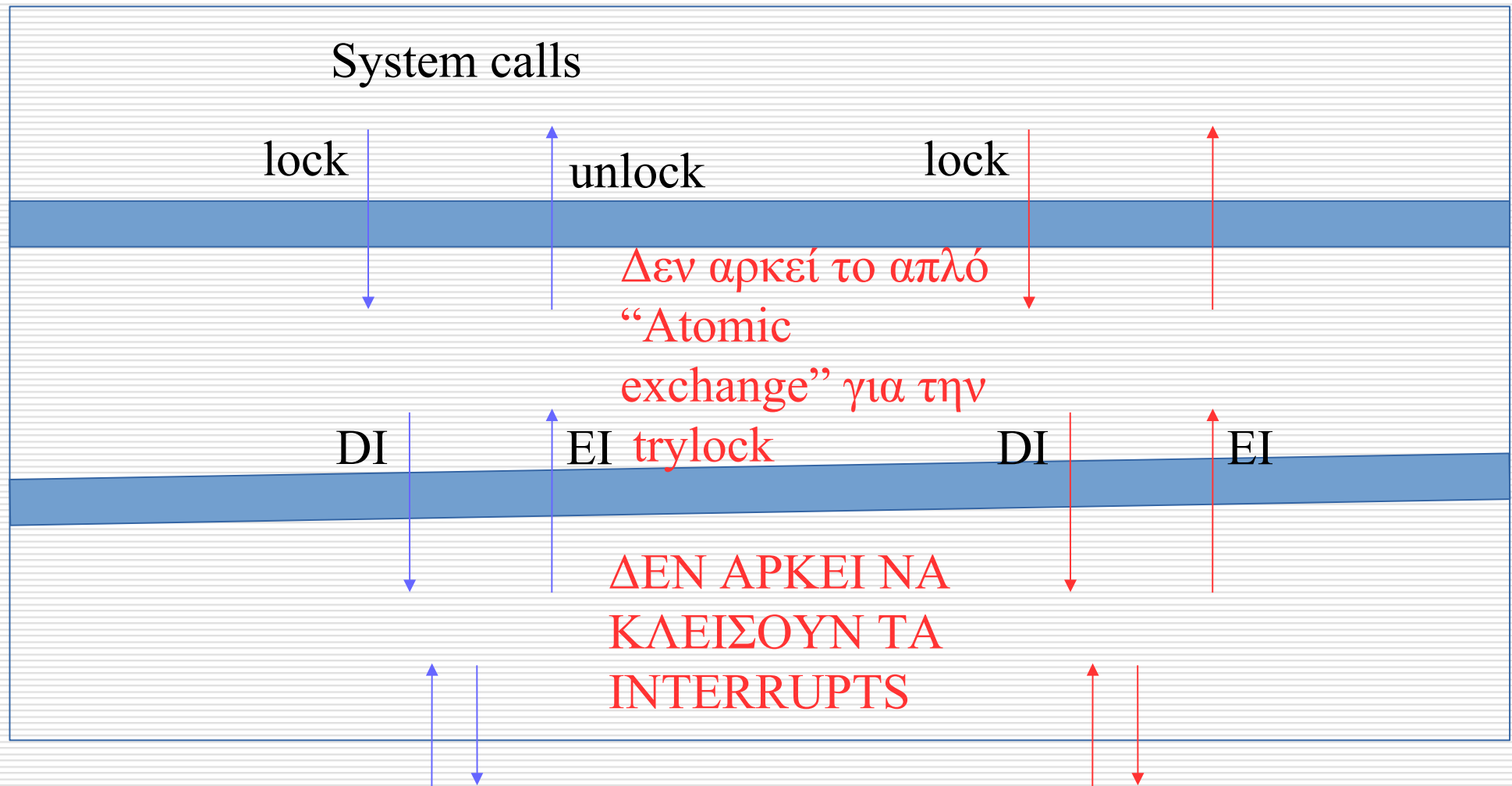
    ... /* Crit.Sec. */

    unlock(mutex);
    ...
}
```



- Έστω ότι το νήμα A καλεί `yield()`
 - π.χ. από κάποιο mutex
- Ενώ είναι μέσα στο C.S. κλειδώνει κάποιο lock
- Λήγει το quantum του
 - Preemption
 - Ο interrupt handler καλεί `yield()`
- **Deadlock...**
- Το ίδιο φαινόμενο και με τα signals σε διεργασίες του Unix.

Πολυπύρηνες CPU



Spin-locking

- Εντός του scheduler και device drivers
- Πολύ μικρή αναμονή
- Πάντα σε συνδυασμό με disable/enable int.
- Αλλά, λείπει κάτι ακόμη...

```
/* γενική ιδέα */  
typedef char Mutex;
```

```
Mutex mx = 0;
```

```
void spin_lock(Mutex *mutex) {  
    while(test_and_set(mutex))  
        while(*mutex);  
}
```

```
void spin_unlock(Mutex* mutex)  
{  
    *mutex = 0;  
}
```

Atomic operations

- Σε κάποιες περιπτώσεις, μπορούμε να κάνουμε updates σε κοινά δεδομένα, χωρίς locking.
 - πχ. αύξηση κατά 1, ανταλλαγή (swap), κλπ.
- **Atomic operations**
 - Το hardware τις εκτελεί εγγυημένα ατομικά
- Ο gcc υποστηρίζει τα παρακάτω atomic operations (το x είναι shared data, u,v είναι thread-private data)
 - Load (u = x)
 - Store (x = u)
 - Test-and-set (u=x; x=1) και Clear (x=0)
 - Exchange (v=u; u=x; x=v;)
 - Compare And Swap (CAS): (if (x==u) then x=v else u=x)
 - Fetch-and-Add : (u = x; x += v)
 - Add-and-Fetch : (x+=v; u = x)

Παρένθεση: γιατί atomic load/store?

- Σε πολυπύρηνες αρχιτεκτονικές υπάρχουν εντολές μηχανής που **δεν είναι ατομικές ως προς τη μνήμη**.
 - Δηλ. η εντολή είναι ατομική ως προς την εκτέλεσή της στον πυρήνα (δεν μπορεί να “διακοπεί” από κάποιο interrupt),
 - **αλλά**, κάνει πάνω από 1 προσπελάσεις στη μνήμη.
- Παραδείγματα εντολών μηχανής:
 - η εντολή exchange !
 - αποθήκευση ακεραίου πιο “φαρδιού” από το data bus
- Άρα, μπορεί να έχουμε το εξής “παράδοξο” (σε 16-bit CPU)

```
/* core 1 */
```

```
/* core 2 */
```

```
αρχικά x=0
```

```
LOAD R1, $10000001
```

```
STORE R1, $(x)
```

```
LOAD R2, $(x)
```

```
;ούτε R2==0 ούτε R2==0x10000001
```

Παράδειγμα χρήσης atomics: lock-free stack push

```
struct Node {
    Int key;
    Node* next;
}

Node* stack; /* shared */

void push(Node **stack, Node* elem)
{
    elem->next = *stack;
    while(CAS(stack, &elem->next, elem->next)) /*loop*/;
}
```


Speculative loads/stores

- Η CPU και/ή ο compiler, **προσπελαύνουν τη μνήμη** με σειρά άλλη από αυτήν που εμφανίζει ο κώδικας (loads νωρίτερα, stores αργότερα).
 - Παράδειγμα: ο compiler χρησιμοποιεί registers για κάποιες μεταβλητές
 - Παράδειγμα: κάποιος πυρήνας της CPU δεν ενημερώνει τα κοινά caches άμεσα
 - Κερδίζουμε τρομερά σε απόδοση, και
 - Το αποτέλεσμα είναι το ίδιο, αν δεν έχουμε πολλά threads/interrupts
- Αν όμως έχουμε πολλά threads, ή interrupt handlers, πρέπει να **ελέγξουμε** τη σειρά προσπέλασης στη μνήμη

Initially x==y==0

/ thread1 */*

x=1
y=1

/ thread2 */*

if(y==1)
 assert(x==1) */* may fail ! */*

Memory fences (ή barriers)

- Ειδικές “δηλώσεις” που οδηγούν τη CPU και τον compiler να διατηρήσουν ΚΑΠΟΙΑ ΣΧΕΤΙΚΗ ΣΕΙΡΑ.
- Συνδυασμός **ATOMIC OPERATIONS** με **FENCES**
 - π.χ. `__atomic_load_n(&x, __ATOMIC_ACQUIRE)`
- Στα πρόσφατα standards (C11/C++11), ορίζονται τα εξής είδη fences (που παραπλανητικά ονομάζονται “memory models”)
 - **RELAXED** : βασικά, σημαίνει “χωρίς fence”
 - **CONSUME** : ειδική περίπτωση του ACQUIRE
 - **ACQUIRE**: δες όλες τις αλλαγές πριν από ένα RELEASE
 - **RELEASE**: όρισε ότι το επόμενο ACQUIRE θα δει όλες τις μέχρι τώρα αλλαγές
 - **ACQUIRE-RELEASE**: συνδυασμός ACQUIRE + RELEASE
 - **SEQUENTIALLY CONSISTENT**: επιπλέον του ACQUIRE-RELEASE, όλα τα νήματα βλέπουν την ίδια σειρά στα atomic operations.

Spinlocks στον gcc

```
typedef char Mutex;
```

```
void spin_lock(Mutex* m)
{
    while(__atomic_test_and_set(m, __ATOMIC_ACQUIRE))
        while(__atomic_load_n(m, __ATOMIC_RELAXED));
}
```

```
void spin_unlock(Mutex* m)
{
    __atomic_clear(Mutex* m, __ATOMIC_RELEASE);
}
```

Mutex (gcc) via multicores

```
typedef char Mutex;
```

```
void Mutex_Lock(Mutex* m)
{
    int spin = 500;
    while(__atomic_test_and_set(m, __ATOMIC_ACQUIRE)) {
        while(__atomic_load_n(m, __ATOMIC_RELAXED))
            if(spin>0) spin--; else yield();
    }
}
```

```
void Mutex_Unlock(Mutex* m)
{
    __atomic_clear(m, __ATOMIC_RELEASE);
}
```

Το μέλλον...

Ήδη, τα atomic operations είναι μέρος του standard της C (2011) της C++ (2011) και της Java (από την Java7 και μετά)

Πηγές:

- <http://en.cppreference.com/w/> για τα standard της C και C++
 - http://en.cppreference.com/w/cpp/atomic/memory_order
- Linux docs (<https://www.kernel.org/doc/Documentation/>) :
 - atomic_ops.txt
 - local_ops.txt
 - memory-barriers.txt
 - preempt-locking.txt
 - volatile-considered-harmful.txt
 - Για προχωρημένους: RCU/*.txt