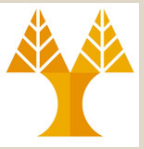
ΕΠΛ421 - Προγραμματισμός Συστημάτων

Διάλεξη 20: Συγχρονισμός Νημάτων και Προχωρημένες Έννοιες (Synchronization & Advanced Topics)

(Κεφάλαιο 11-12 - Stevens & Rago)

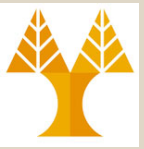
Δημήτρης Ζεϊναλιπούρ

Περιεχόμενο Διάλεξης



A. Ασυνέπεια Μνήμης (Data Inconsistency)
B. Συγχρονισμός Νημάτων με δυαδικούς σηματοφόρους
C. Αποφυγή Αδιεξόδων και Reader/Writer Locks
D. Linux Threading: Native POSIX Thread Library
E. Άλλα Σχετικά Θέματα
   - Thread Pools
   - Thread-Safe Libraries
   - Μέγιστος Αριθμός Νημάτων – sysconf()

Α. Ασυνέπεια Μνήμης



- Εφόσον τα νήματα έχουν κοινή μνήμη μπορεί να δημιουργηθούν προβλήματα συνέπειας μνήμης.
- **Ασυνέπεια Μνήμης (Memory Inconsistency).**

  Λάθη μνήμης τα οποία δημιουργούνται όταν ένα νήμα **πραγματοποιεί κάποια λειτουργία** (Read, Write, Delete) πάνω σε κάποια περιοχή μνήμης (μεταβλητή, δομή δεδομένων, κτλ) ενώ ένα άλλο νήμα προσπαθεί να **μεταβάλει** αυτή την περιοχή μνήμης (με Write ή Delete).
- Αυτό μπορεί να συμβεί εύκολα (π.χ., καθώς ένα νήμα εκτελείται γίνεται **μεταγωγή περιβάλλοντος (context switching)** με αποτέλεσμα να ανακτήσει την μονάδα επεξεργασίας κάποιο άλλο νήμα. Συμβαίνει ακόμα ευκολότερα σε συστήματα SMP.

Α. Ασυνέπεια Μνήμης

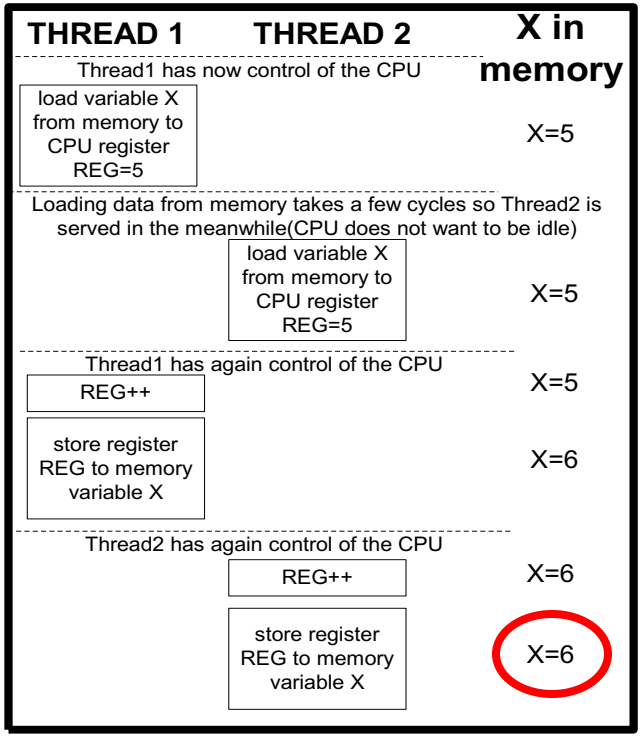
- Το ακόλουθο παράδειγμα δείχνει πως η μνήμη θα βρεθεί σε κάποια **ασυνεπή κατάσταση**.
- Αριστερά δημιουργείται το λανθασμένο αποτέλεσμα (δηλ. 6) ενώ οποιαδήποτε σειριακή και ορθή εκτέλεση των δυο νημάτων (δεξιά) θα δημιουργούσε το αποτέλεσμα 7.

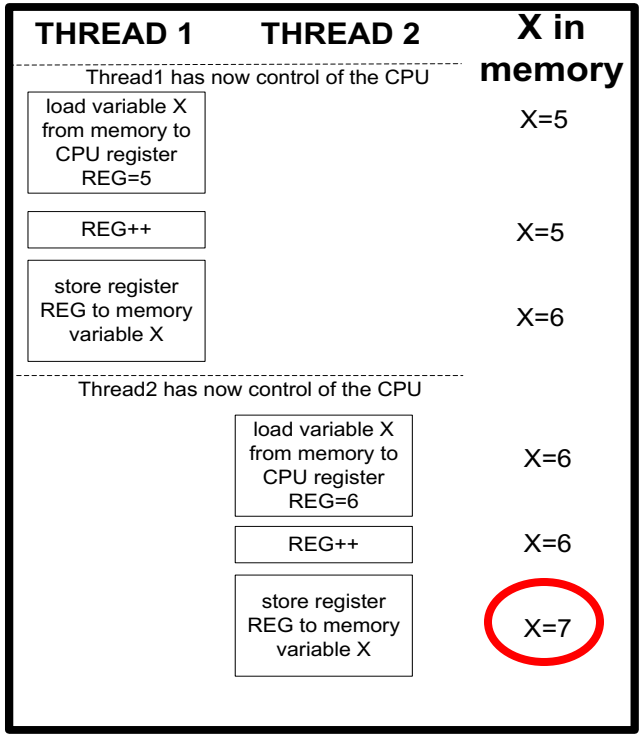
Η πράξη X++ δεν είναι ατομική (atomic). Η κλήση της μεταφράζεται ως εξής:

Α. Φόρτωσε το X από την κύρια μνήμη σε κάποιο καταχωρητή REG του επεξεργαστή.

Β. Αύξησε τιμή του REG κατά μια μονάδα.

C. Φύλαξε την τιμή του REG στην μεταβλητή μνήμης X.



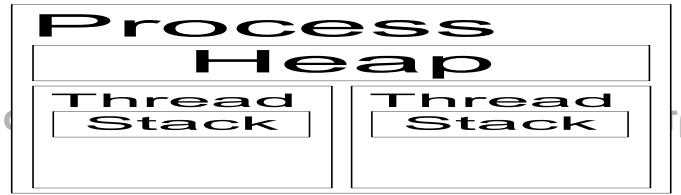


Εάν το X++ ήταν ατομική πράξη τότε δεν θα υπήρχε πρόβλημα ταυτοχρονίας

Α. Ασυνέπεια Μνήμης

Πότε ΔΕΝ τίθεται θέμα Ασυνεπούς Μνήμης?

- **Readonly:** Όταν τα νήματα κάνουν αποκλειστικά ανάγνωση δεδομένων (δύσκολο να μας το εγγυηθεί κάποιος)
- **Const:** Εάν πρόκειται για μια **καθολική σταθερά** (define ή const) που δεν μπορεί να αλλάξει τιμή έτσι και αλλιώς.
- **LocalVars:** Εάν ένα νήμα χρησιμοποιεί μόνο **τοπικές μεταβλητές**
  - Η συζήτηση μας γενικά υποθέτει ότι ένα νήμα δεν θα **προσπελάζει την στοίβα** άλλου νήματος αυθαίρετα – πχ. από κάποιο λάθος pointer.
  - Για να απομονώσουμε (την μνήμη) των νημάτων μεταξύ τους ρητά, μπορούμε να χρησιμοποιήσουμε την **pthread_key_create()** (Κεφ. 12.6), το οποίο δημιουργεί την έννοια της **προσωπικής μνήμης νήματος** (thread-private data) αλλά δεν θα μελετηθεί σε αυτό το μάθημα.



Παράδειγμα Ασυνέπεια Μνήμης

Ένα νήμα έχει πρόσβαση σε οποιαδήποτε διεύθυνση της διεργασίας στην οποία ανήκει

```
#include <pthread.h> /* Pthread related functions*/
#include <string.h> /* For strerror */
#include <stdio.h> /* For fopen, printf */
#include <errno.h> /* For errno variable */
#define perror2(s, e) fprintf(stderr, "%s: %s\n", s, strerror(e))
void *runner1(); /* function prototype of thread's code */

int *address;

int main(int argc, char *argv[]) {
   int err;   // error code
    pthread_t tid;  /* thread id  table*/

    if (err = pthread_create( &tid, NULL, &runner1, NULL)) {
            perror2("pthread_create", err);  exit(1);
    }
    sleep(1);

    // Αλλαγή της τιμής X που βρίσκετε στην στοίβα του νήματος
    *address = 7;
    while (1)      sleep(1);
    pthread_exit(0);
}

void *runner1() {X
            int X=0;   // Η μεταβλητή X είναι στην στοίβα του νήματος
            address = &X;  /* Καταχώρηση διεύθυνσης X σε καθολική μεταβλητή.
                              για να γνωρίζει ο γονέας την διεύθυνση της τοπικής
                              μεταβλητής X και να μπορεί να την αλλάξει στην συνέχεια*/
            while (1) {
                    printf("Thread.X=%d\n", X); sleep(1);
            }
   pthread_exit( 0 );
}
```

Αυτό το πρόγραμμα δείχνει πως ένα νήμα μπορεί να αλλάξει τα δεδομένα (μεταβλητή X) στην στοίβα ενός άλλου thread. Επομένως πρέπει να είμαστε προσεκτικοί με την προστασία των δεδομένων ενός νήματος !

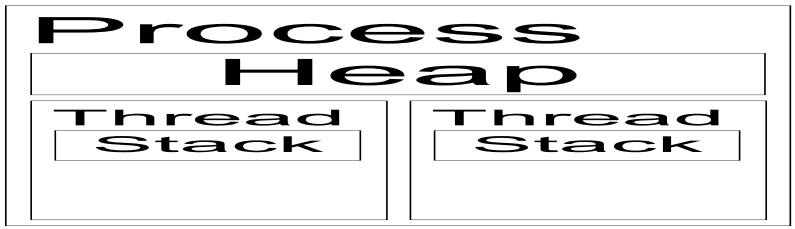
Δεδομένα Εξόδου

```
$execute
Thread.X=0
Thread.X=0
Thread.X=7
Thread.X=7
Thread.X=7
```

Α. Ασυνέπεια Μνήμης

Πότε **τίθεται** θέμα Ασυνέπειας Μνήμης?

- **Heap: Δεδομένα στην Σωρό:** εάν κάποια αντικείμενο είναι στην σωρό (και η διεύθυνση του αντικειμένου έχει περάσει στα νήματα τα οποία προβαίνουν σε αλλαγές του αντικειμένου) τότε μπορεί να δημιουργηθούν λάθη στην τιμή του αντικειμένου.
- **Static: Μεταβλητή Static στην συνάρτηση runner(),** όπου runner() είναι η συνάρτηση εκτέλεσης **του νήματος.**
  - Γνωρίζουμε ότι οι static μεταβλητές αποθηκεύονται στο DATA segment μαζί με άλλες καθολικές μεταβλητές.
  - Συνεπώς, δημιουργούνται μόνο μια φορά μεταξύ διαδοχικών κλήσεων, και διαμοιράζονται μεταξύ όλων των νημάτων-παιδιών (δες παράδειγμα στην επόμενη διαφάνεια).



Α. Παράδειγμα Ασυνέπεια Μνήμης Προβλήματα από **static** μεταβλητές

```
#include <pthread.h> /* Pthread related functions*/
#include <string.h> /* For strerror */
#include <stdio.h> /* For fopen, printf */
#include <errno.h> /* For errno variable */
#define perror2(s, e) fprintf(stderr, "%s: %s\n", s, strerror(e))
#define THREADS 5
void *runner1(); /* function prototype of thread's code */

int main(int argc, char *argv[]) {
    int err, i;     // error code and loop counter
    pthread_t tid[THREADS];  /* thread id  table*/

    for (i=0; i<THREADS; i++) {
            /* create thread */
            if (err = pthread_create( &tid[i], NULL, &runner1, NULL)) {
                error2("pthread_create", err);  exit(1);
            }
    }
   pthread_exit(0);
}

void *runner1() {
   static int x;
   while (1) {
            printf("%d %d\n", pthread_self(), x++); sleep(3);
    }
    pthread_exit( 0 );
}
```

Όσες διαδοχικές κλήσεις και εάν γίνουν στην runner (όσα νήματα και αν δημιουργηθούν) θα υπάρχει μόνο μια (1) μεταβλητή x στην μνήμη.

.....

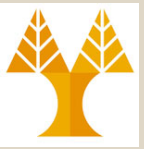
6686216	190
6686488	191
6686352	192
6686624	192
6686216	193
6686488	194

.....

Παρατηρούμε ότι κάποιες τιμές εμφανίζονται διπλά!, μια πιθανή εκτέλεση που οδήγησε στο λάθος είναι η πιο κάτω:

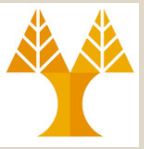
Thread1.Load X (191)
Thread2.Load X (191)
Thread1.X++ (192 στον Καταχωρητή - CPU)
Thread1.Store X (192 στην RAM)
Thread1.printf (192 οθόνη)
Thread2.X++ (192 στον Καταχωρητή - CPU)
Thread2.Store X (192 στην RAM)
Thread2.printf (192 οθόνη)

Β. Συγχρονισμός Νημάτων με Δυαδικούς Σηματοφόρους



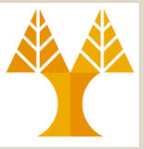
- Για τον συγχρονισμό μεταξύ νημάτων που προτίθενται να προσπελάσουν κοινούς πόρους, η **βιβλιοθήκη POSIX** παρέχει μια απλοποιημένη εκδοχή σηματοφόρων, τους **δυαδικούς σηματοφόρους (Mutexes).**
- Ένας δυαδικός σηματοφόρος μπορεί να βρίσκεται σε μια από δυο πιθανές καταστάσεις: ***Κλειδωμένος*** ή ***Ξεκλείδωτος***
- Η πράξη κλειδώματος/ξεκλειδώματος είναι **ατομική** και υλοποιείται μέσα στην βιβλιοθήκη (επομένως δεν χρειάζεται η χρήση των σηματοφόρων πυρήνα πράγμα το οποίο είναι αρκετά πιο ακριβό).

Β. Δυαδικοί Σηματοφόροι



- **Για να στατικό (static) ορισμό ενός Mutex:**
  pthread_mutex_t name = **PTHREAD_MUTEX_INITIALIZER**
- **Για δυναμικό (dynamic) ορισμό ενός Mutex:**
  int pthread_mutex_init(pthread_mutex_t *mutex, pthread_mutexattr_t *attr)
  - Επιστρέφουν 0 ή τον αριθμό λάθους
  - Μέσω attr μπορούμε να ορίσουμε κάποια χαρακτηριστικά αλλά το αφήνουμε NULL (περισσότερα στο Κεφ. 12.4)
- **Για να καταστρέψουμε ένα Mutex**
  int pthread_mutex_destroy(pthread_mutex_t *mutex)
  - Η κλήση θα πετύχει μόνο εφόσον είναι ξεκλείδωτο το mutex.
  - Δεν χρειάζεται να απελευθερώνετε ένα mutex το οποίο δηλώθηκε στατικά.

Β. Δυαδικοί Σηματοφόροι



- Όλες οι πιο κάτω συναρτήσεις επιστρέφουν 0 σε επιτυχία ή τον αριθμό του λάθους.
- Για να κλειδώσουμε ένα Mutex :

  int pthread_mutex_lock(pthread_mutex_t *mutex)
  - Εάν είναι κλειδωμένο το mutex τότε το καλών νήμα τίθεται σε αναστολή (δηλαδή κάνει block) και η συνάρτηση επιστρέφει μόλις κατορθώσει να κλειδώσει το mutex.

- Εάν ένα νήμα δεν έχει την πολυτέλεια να κάνει block (για να κλειδώσει ένα Mutex) τότε μπορεί να χρησιμοποιήσει την:

  int pthread_mutex_trylock(pthread_mutex_t *mutex)
  - Η συνάρτηση επιχειρεί να κλειδώσει το mutex. Αν αποτύχει τότε δεν κάνει block αλλά επιστρέφει απλά EBUSY (ένδειξη ότι απότυχε το lock)

- Για να ξεκλειδώσουμε ένα mutex :

  int pthread_mutex_unlock(pthread_mutex_t *mutex)
  - Το ξεκλείδωμα γίνετε μόνο εάν το lock έγινε από το ίδιο νήμα.

Types of Locks
(Τύποι Κλειδαριών)

Binary Locks (Δυαδικές Κλειδαριές)

```
lock_item(X):
B: if LOCK(X) = 0          (* item is unlocked *)
       then LOCK(X) ← 1    (* lock the item *)
   else
       begin
       wait (until LOCK(X) = 0
           and the lock manager wakes up the transaction);
       go to B
       end;
unlock_item(X):
       LOCK(X) ← 0;        (* unlock the item *)
       if any transactions are waiting
           then wakeup one of the waiting transactions;
```

// Waiting Queue for Object X (outside the lock_item() function)

//Notify next in Queue

Παράδειγμα Δυαδικού Σηματοφόρου

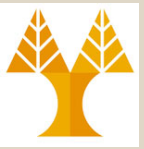
Ας δούμε ξανά το παράδειγμα με την static μεταβλητή

```
#include <pthread.h> /* Pthread related functions*/
#include <string.h> /* For strerror */
#include <stdio.h> /* For fopen, printf */
#include <errno.h> /* For errno variable */
#define perror2(s, e) fprintf(stderr, "%s: %s\n", s, strerror(e))
#define THREADS 10
void *runner1(); /* function prototype of thread's code */

pthread_mutex_t mtx;          /* Mutex for Synchronization */
// int x;  // Για παρουσίαση της λειτουργίας των σηματοφόρων, το
// x θα μπορούσε να είναι δηλωμένο και εδώ αντί με static μέσα στην συνάρτηση.
int main(int argc, char *argv[]) {
   int err, i;// error code and loop counter
   pthread_t tid[THREADS];  /* thread id  table*/
   pthread_mutex_init(&mtx, NULL);          /* Initialized Mutex */

   for (i=0; i<THREADS; i++) {    /* create thread */
            if (err = pthread_create( &tid[i], NULL, &runner1, NULL)) {
                  perror2("pthread_create", err);  exit(1);
            }
   }
   for (i=0; i<THREADS; i++) {    /* join threads */
            if (err = pthread_join( tid[i], NULL)) {
                  perror2("pthread_join", err);  exit(1);
            }
   }
   pthread_mutex_destroy(&mtx);   pthread_exit(0);
}
```

```
void *runner1() {
   static int x;
   int err;
   while (1) {   /* Lock Mutex */
         if (err = pthread_mutex_lock(&mtx)) {
               perror2(pthread_mutex_lock); exit(1);
         }
         x++;
         if (err = pthread_mutex_unlock(&mtx)) {
               /* Unlock Mutex */
               perror2(pthread_mutex_lock); exit(1);
         }
         printf("%d\n", x);
    }
    pthread_exit( 0 );
}
```

Τώρα η αύξηση του μέτρηση x είναι ατομική πράξη (παρόλο που το x είναι static). Επομένως θα πάρουμε το σωστό **σειριακά-διατεταγμένο (serializable)** πλάνο εκτέλεσης (δηλ., τα νήματα εκτελούνται με οποιαδήποτε σειρά χωρίς ωστόσο να δημιουργείται πρόβλημα ασυνέπειας μνήμης)

Β. Δυαδικοί Σηματοφόροι



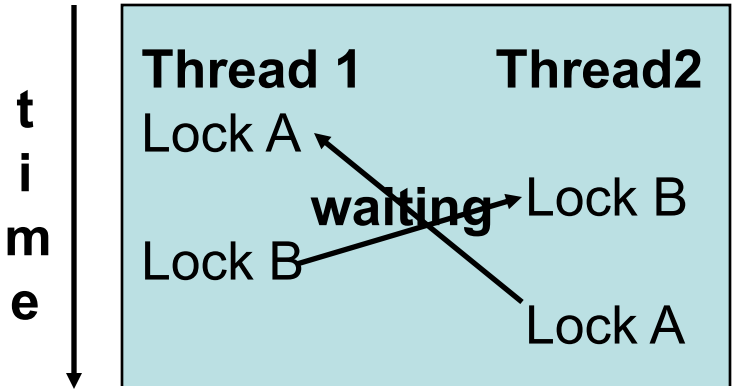
- Η βιβλιοθήκη **pthread** ΔΕΝ διατάσσει από μόνη της **σειριακά (serialize)** την πρόσβαση σε δεδομένα, αλλά αυτό το πετυχαίνουμε εμείς με την σωστή διάταξη των **lock()** και **unlock()**.
- Το mutex **τίθεται (lock)** από τον προγραμματιστή προτού να γίνει η πρόσβαση στο **critical section (κρίσιμη περιοχή)** (κάποιο κοινό πόρο) και το mutex **απελευθερώνεται** αμέσως μετά (**unlock**).
- Εάν επιτρέψουμε σε ένα νήμα να έχει πρόσβαση σε κοινό πόρο **χωρίς να κλειδώσει** το mutex τότε αυτό μπορεί να οδηγήσει σε κατάσταση **ασυνεπούς μνήμης.**
- Η έννοια των **thread mutexes** υπάρχει σε όλες τις γλώσσες. π.χ., στη Java υπάρχει η λέξη κλειδί **synchronized** που χρησιμοποιείται σε ονόματα συναρτήσεων ή και αντικειμένων:
  - π.χ. public synchronized int get() { …. }
  - ή synchronized (someobject) { }

C. Αποφυγή Αδιεξόδων

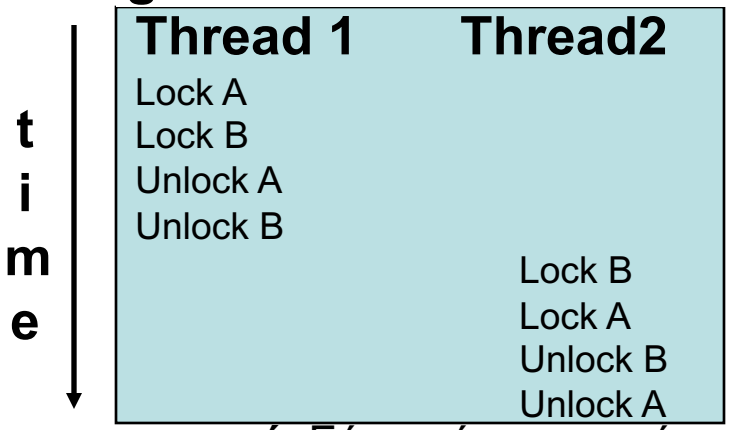
Αδιέξοδο (Deadlock)

Κατάσταση κατά την οποία δυο (ή περισσότερα) νήματα **τίθενται σε αναστολή (κάνουν block)**, για **απεριόριστο χρονικό** διάστημα, επειδή προσπαθούν να **κλειδώσουν ένα mutex (lock)** το οποίο είναι ήδη κλειδωμένο από το άλλο νήμα.

Deadlock

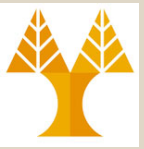


Rigorous 2PL: NO Deadlock



Η αποφυγή αδιεξόδων είναι ευθύνη του προγραμματιστή. Εάν πρόκειται για ένα μεγάλο αριθμό από mutexes και δεν μπορούμε να τα κλειδώσουμε με μια σειρά τότε έχουμε δυο εναλλακτικές επιλογές: α) Στην απλούστερη περίπτωση, κάθε νήμα δοκιμάζει **trylock(),** εάν αποτύχει τότε αφήνει όλα τα locks και δοκιμάζει ξανά αργότερα. β) Εναλλακτικά μπορεί κάποιος coordinator να φτιάξει ένα κατευθυνόμενο **Wait-for-Graph G(mutexes, dependencies)** και να βρίσκει εάν υπάρχουν κύκλοι μέσα στον G (οπόταν υπάρχει deadlock). Άλλες τεχνικές μπορεί να λαμβάνουν υπόψη και θέματα **starvation** (να μην τιμωρείται συνεχώς ο ίδιος) κτλ.

C. Reader / Writer Locks (Shared-Exclusive Locks)



- **Όμοια με Binary Mutexes με την διαφορά ότι επιτρέπουν στο mutex να βρίσκεται σε 3 καταστάσεις αντί 2 (locked, unlocked**)
- Ένα reader/writer lock μπορεί να βρίσκεται:
  - Locked in Read mode (Shared Mode)
  - Locked in Write Mode (Exclusive Mode)
  - Unlocked
- Σε ένα **Reader/Writer Lock**, όταν κλειδώνεται ένα **Write-Lock** τότε κανένα άλλο lock δεν επιτρέπεται να υπάρχει ταυτόχρονα.
- Ωστόσο, πολλά **read locks** μπορεί να **συνυπάρχουν**. Αν υπάρξει αίτηση για **write-lock**, καθώς ήδη υπάρχουν κάποια **read-locks**, τότε το **write** κάνει block μέχρι να **τερματίσουν όλα τα read-locks**.
- **Είναι ιδανικά για περιπτώσεις όπου αντικείμενα διαβάζονται πιο συχνά από ότι ενημερώνονται.** Π.χ. μια συνάρτηση find() σε μια δομή ζητά Read Lock ενώ το update() ζητά Write Lock.
- Μελετήστε το παράδειγμα ουράς στην σελίδα 380 (Stevens & Rago)
- Οι συναρτήσεις μοιάζουν πολύ με τα binary mutexes που είδαμε έως τώρα. (π.χ. **pthread_rwlock_init**(pthread_rwlock_t *lock, pthread_rwlockattr_t *attr) αντί int **pthread_mutex_init** (pthread_mutex_t *mutex, pthread_mutexattr_t *attr)

Types of Locks
(Τύποι Κλειδαριών)

Shared/Exclusive (or Read/Write Locks).

S(X)

```
read_lock(X):
B: if LOCK(X) = "unlocked"
     then begin LOCK(X) ← "read-locked";
          no_of_reads(X) ← 1
          end
   else if LOCK(X) = "read-locked"
     then no_of_reads(X) ← no_of_reads(X) + 1
   else begin
        wait (until LOCK(X) = "unlocked"
          and the lock manager wakes up the transaction);
        go to B
        end;
```

// Grant first Lock

// More than one Shared Locks granted

// Waiting Queue for Object X as X is currently locked.

X(X)

```
write_lock(X):
B: if LOCK(X) = "unlocked"
     then LOCK(X) ← "write-locked"
   else begin
        wait (until LOCK(X) = "unlocked"
          and the lock manager wakes up the transaction);
        go to B
        end;
```

// Initialize Lock(X) to "write-locked" (only 1 write lock is granted)

// Waiting Queue for Object X as X is currently locked.

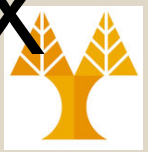
Rel(X)

```
unlock (X):
   if LOCK(X) = "write-locked"
     then begin LOCK(X) ← "unlocked";
          wakeup one of the waiting transactions, if any
          end
   else it LOCK(X) = "read-locked"
     then begin
          no_of_reads(X) ← no_of_reads(X) − 1;
          if no_of_reads(X) = 0
            then begin LOCK(X) = "unlocked";
                 wakeup one of the waiting transactions, if any
                 end
          end;
```

// If "write-locked" then unloc

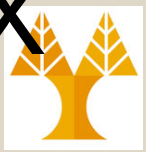
// If "read-locked" then decrease no_of_reads(X) counter and reset to "unlocked if necessary

D. Linux Threading: Native POSIX Thread Library (NPTL)



- Το γεγονός ότι η διαχείριση νημάτων στο Linux γίνεται στο **user space** δημιουργεί διάφορα προβλήματα διαχείρισης και επίδοσης νημάτων (θυμηθείτε πχ ότι κάθε Thread κατακρατεί ένα PID).
- Η **Redhat** έχει αναπτύξει το **Native POSIX Thread Library (NPTL)** το οποίο προσφέρει κάποια υποστήριξη από τον ίδιο τον πυρήνα.
- Το NPTL έχει σχεδιαστεί για να μεγιστοποιήσει την επίδοση σε επεξεργαστές Symmetric Multi-Processing (SMP)
  - SMP: Αρχιτεκτονική όπου 2 ή περισσότεροι ίδιοι επεξεργαστές συνδέονται με κοινόχρηστη μνήμη
  - Τα SMPs έχουν επισκιάσει τους uni-processors στην σημερινή αγορά επεξεργαστών.

D. Linux Threading: Native POSIX Thread Library (NPTL)



- Στο NPTL (πάλι 1:1 μοντέλο) δεν υπάρχει η έννοια του **νήματος διαχειριστή** (ο οποίος είναι υπεύθυνος για συγχρονισμό των νημάτων, διανομή σημάτων, κτλ)
  - Αυτές οι λειτουργίες αναλαμβάνονται από τον πυρήνα το οποίο οδηγεί σε αυξημένη επίδοση.
- Προβλέπεται ότι τα NPTL θα αντικαταστήσουν πλήρως τα LinuxThreads στο μέλλον, αν και ήδη οι εκδόσεις του Kernel > 2.6.x έρχονται με το NPTL εγκατεστημένο.
- Για περισσότερες πληροφορίες για το NPTL ελέγξετε το σύνδεσμο: http://www-128.ibm.com/developerworks/linux/library/l-threading.html?ca=dgr-lnxw07LinuxThreadsAndNPTL

D. Linux Threading: Native POSIX Thread Library (NPTL)

- **Επεξεργαστής και Βιβλιοθήκες Threading δυο Υπολογιστών του Π.Κ.**

cs4054.in.cs.ucy.ac.cy

- Intel(R) Pentium(R) 4 CPU 3.60GHz (1 Core με Hyperthreading, δηλ το λειτουργικό το βλέπει σαν 2 Cores για χρονοδρομολόγηση νημάτων).
- **Έκδοση Πυρήνα Λ.Σ. :**
  - `dzeina@cs4054> uname -a`

    `Linux cs4054.in.cs.ucy.ac.cy 2.6.20-1.2307.fc5smp #1 SMP Sun Mar 18 21:02:16 EDT 2007 i686 i686 i386 GNU/Linux`
- **Έκδοση Pthread Library:**
  - `dzeina@cs4054> getconf GNU_LIBPTHREAD_VERSION`

    `NPTL 2.4`

astarti.cs.ucy.ac.cy

- 4x Intel(R) Xeon(TM) CPU 1.40GHz (από το 2002) χωρίς τεχνολογία Hyperthreading και χωρίς πολύ-πυρήνες (επομένως κάθε επί μέρους επεξεργαστής έχει λιγότερη υπολογιστική ισχύ).
- **Έκδοση Πυρήνα Λ.Σ. :**
  - `dzeina@astarti> uname -a`

    `Linux astarti 2.4.20-18.8smp #1 SMP Thu May 29 07:20:32 EDT 2003 i686 i686 i386 GNU/Linux`
- **Έκδοση Pthread Library:**
  - `dzeina@astarti> getconf GNU_LIBPTHREAD_VERSION`

    `linuxthreads-0.10` (Εφόσον η έκδοση του kernel είναι 2.4.x, είναι αναμενόμενο ότι υποστηρίζει μόνο τα παλιά (user-level) **linuxthreads**)

Ε. Άλλα Σχετικά Θέματα
Μέγιστος Αριθμός Νημάτων – sysconf()

#include <unistd.h>

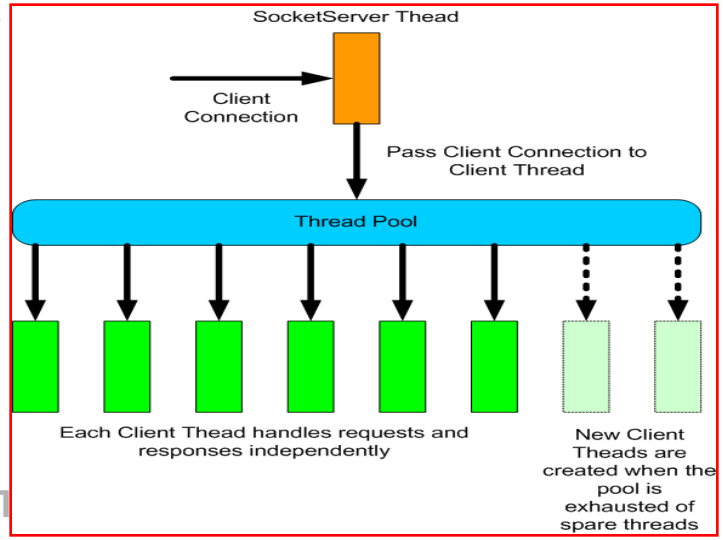
- Η συνάρτηση **int sysconf(name)** μας επιστρέφει το όριο που σχετίζεται με την σταθερά *name.*
- Για να βρούμε τον μέγιστο αριθμό από νήματα που μπορούν να εκτελεστούν στο σύστημα χρησιμοποιούμε την σταθερά **_SC_THREAD_THREADS_MAX**
- Για περισσότερες σταθερές που υποστηρίζονται από την **sysconf** συμβουλευθείτε τον πιο κάτω σύνδεσμο http://www.opengroup.org/onlinepubs/009695399/basedefs/unistd.h.html

Άλλα Σχετικά Θέματα
Thread Pools

- Έστω ότι έχουμε ένα **πολυνηματικό** Web-Server (π.χ. Apache) ο οποίος δέχεται αιτήσεις από πελάτες και δημιουργεί ένα νέο νήμα ανά αίτηση.
- Αυτό έχει τα εξής προβλήματα
  1. Πρέπει να **δημιουργήσουμε** το νήμα το οποίο είναι ακριβό (βέβαια πιο φθηνό από μια νέα διεργασία). Στην συνεχεία πρέπει να **αποδεσμεύσουμε** τα δεδομένα που σχετίζονται με τον νήμα και τέλος να το καταστρέψουμε.
  2. Δεν ξέρουμε τον **μέγιστο αριθμό νημάτων** ανά πάσα στιγμή, επομένως μπορεί η διεργασία του webserver να ξεπεράσει το **επιτρεπτό όριο μνήμης** με αποτέλεσμα να **χάσουμε την υπηρεσία**.

Άλλα Σχετικά Θέματα
Thread Pools

- Η βασική ιδέα του Thread-pool είναι να δημιουργήσουμε ένα αριθμό από threads κατά την διάρκεια της εκκίνησης της υπηρεσίας.
- Τα **νήματα αυτά περιμένουν** μέχρι να έρθει μια νέα αίτηση, την **διεκπεραιώνουν** και στην συνέχεια γίνονται και πάλι **διαθέσιμες στο pool** (χωρίς να καταστρέφονται)
- Εάν κάποια στιγμή χρειαστούμε περισσότερα νήματα τότε **αυξάνουμε δυναμικά** τον αριθμό μέχρι να φτάσουμε ένα **όριο** που θέτει ο χρήστης και από εκεί **δεν δεχόμαστε** άλλες αιτήσεις.



Άλλα Σχετικά Θέματα
Thread Safe Libraries

- **Thread-Safe Library:** Μια βιβλιοθήκη η οποία έχει σχεδιαστεί με τέτοιο τρόπο που να υποστηρίζει πολυνηματικές εφαρμογές.
- Μια συνάρτηση λέγεται ότι είναι **thread-safe (reentrant)** εάν μπορεί να κληθεί από πολλαπλά νήματα και το αποτέλεσμα της συνάρτησης εξακολουθεί να είναι ορθό

  π.χ., καλώντας **την printf()** από πολλαπλά νήματα **δεν μας δημιουργεί** κάποιο **λάθος** αποτέλεσμα!
- Το **μεγαλύτερο ποσοστό** των συναρτήσεων POSIX, με μερικές εξαιρέσεις (σχήμα 12.9), είναι **thread safe!**
- Οι **thread-safe συναρτήσεις** υλοποιούνται με χρήση **mutexes** τα οποία ελέγχουν την ταυτοχρονία.