
Topic 6: Threads*

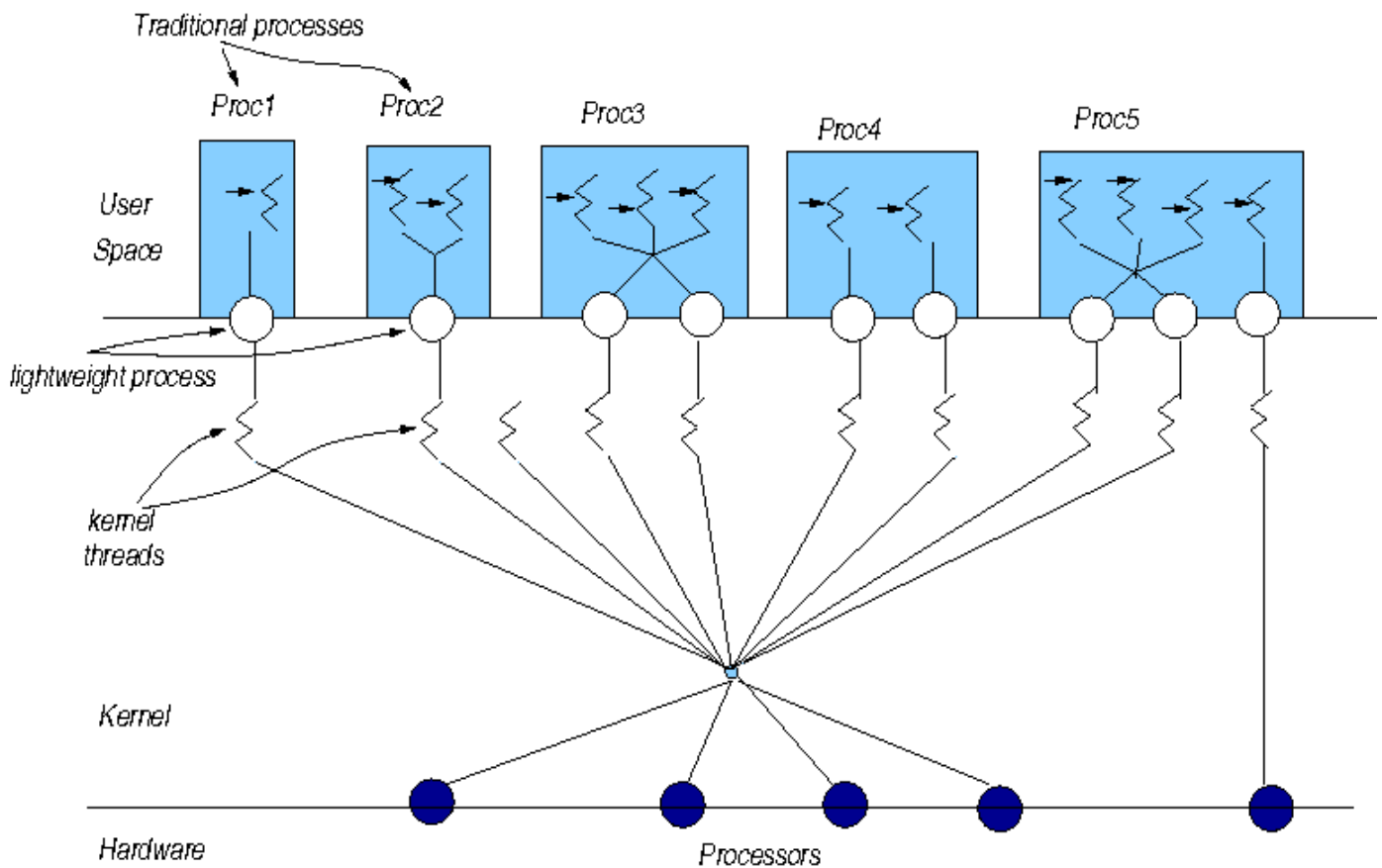
K24: Systems Programming

Instructor: Mema Roussopoulou

Threads - Νήματα

- Lightweight Processes (LWPs)
 - share single address space,
 - each has its own flow control
- Try to overcome penalties
 - separate process for every flow of control is costly
- Offer a more efficient way to develop apps
 - on uniprocessor, one thread blocks on a syscall (e.g., read), another grabs CPU and does useful processing (faster user interface, lower program execution time)
 - on multiprocessor, separate thread on each CPU (program finishes faster)

Νήματα - Μοντέλο



Two-level multi-threaded Model

Νήματα

- Ένα ή περισσότερα νήματα μπορούν να εκτελούνται στο πλαίσιο μίας διεργασίας
- Η βασική μονάδα που χρονοδρομολογείται από το λειτουργικό σύστημα είναι το νήμα και όχι η διεργασία
- Τα νήματα των διεργασιών εκτελούνται ψευδο-παράλληλα, αλλά μπορούν να εκτελεσθούν και πραγματικά παράλληλα σε συστήματα με πολλούς επεξεργαστές
- Οποιοδήποτε νήμα μίας διεργασίας μπορεί να δημιουργήσει άλλα νήματα
- Όλα τα νήματα μίας διεργασίας μοιράζονται τον ίδιο χώρο διευνύνσεων, καθώς και άλλα χαρακτηριστικά της διεργασίας (π.χ. κώδικας, περιγραφείς αρχείων κ.λ.π.) αλλά το καθένα έχει τη δική του στοίβα εκτέλεσης και δείκτη προγράμματος
- Το λειτουργικό σύστημα μπορεί να εκτελέσει πολύ γρηγορότερα την εναλλαγή νημάτων σε σχέση με την εναλλαγή διεργασιών

Πολύ σημαντικό

Νήματα (συν.)

- Όλες οι συναρτήσεις βιβλιοθήκης που ακολουθούν και χρησιμοποιούνται για διαχείριση νημάτων απαιτούν

```
#include <pthread.h>
```

Τα προγράμματα στα οποία χρησιμοποιούνται οι συναρτήσεις αυτές πρέπει να μεταγλωττίζονται με την εντολή

```
gcc -o <filename> <filename>.c -pthread
```

έτσι ώστε στο δημιουργούμενο εκτελέσιμο να “φορτώνεται” και η βιβλιοθήκη για τα νήματα

- Οι συναρτήσεις βιβλιοθήκης για διαχείριση νημάτων δεν θέτουν τιμή, σε περίπτωση λάθους, στην εξωτερική μεταβλητή `errno`, συνεπώς δεν μπορεί να χρησιμοποιηθεί η συνάρτηση βιβλιοθήκης `perror` για την εκτύπωση κατάλληλου διαγνωστικού μηνύματος
- Σε περίπτωση λάθους κατά την κλήση κάποιας συνάρτησης διαχείρισης νημάτων, μπορεί να χρησιμοποιηθεί η συνάρτηση βιβλιοθήκης `strerror`, για την εκτύπωση κατάλληλου διαγνωστικού μηνύματος που αντιστοιχεί στον κωδικό του λάθους που επέστρεψε η συνάρτηση για το νήμα
- Συνάρτηση βιβλιοθήκης `strerror`
 - `char *strerror(int errnum)`
 - Επιστρέφει μία συμβολοσειρά που περιγράφει το λάθος το οποίο αντιστοιχεί στον κωδικό λάθους `errnum`
 - Απαίτηση: `#include <string.h>`

Σημαντικό

Threads vs. Processes

	Νήματα	Διεργασίες
Χώρος Δεδομένων	Κοινός. Ότι αλλάζει το 1 νήμα το βλέπουν/ αλλάζουν και τα άλλα (πχ malloc/free)	Ξεχωριστός. Ξεχωριστός χώρος διευθύνσεων μετά το fork
Περιγραφείς Αρχείων	Κοινοί. 2 νήματα χρησιμοποιούν ίδιο περιγραφέα. Αρκεί 1 close σε αυτόν.	Αντίγραφα. 2 διεργασίες χρησιμοποιούν αντίγραφα του περιγραφέα.
fork	Μόνο το νήμα που κάλεσε fork αντιγράφεται.	
exit	Όλα τα νήματα πεθαίνουν μαζί (pthread_exit για τερματισμό μόνο ενός νήματος)	
exec	Όλα τα νήματα εξαφανίζονται (αντικαθίσταται ο κοινός χώρος διευθύνσεων)	
Σήματα	Περίπλοκο. Κοιτάξτε βιβλίο. Κάποια πάνε στο νήμα που τα προκάλεσε, κάποια στο πρώτο που δεν έχει μπλοκάρει...	

Δημιουργία και Τερματισμός Νημάτων

- Συνάρτηση βιβλιοθήκης `pthread_create`

- `int pthread_create(pthread_t *thread, pthread_attr_t *attr, void *(*start)(void *), void *arg)`
- Δημιουργεί ένα καινούργιο νήμα το οποίο εκτελεί τη συνάρτηση με διεύθυνση `start` και παράμετρο το `arg`
- Η ταυτότητα του νήματος επιστρέφεται στο `*thread`
- Μέσω του `attr` μπορούμε να ορίσουμε κάποια χαρακτηριστικά για το νήμα, αλλά συνήθως αφήνουμε τα default, δίνοντάς του την τιμή `NULL`
- Επιστρέφει 0 σε επιτυχία

Χαρακτηριστικό νήματος.

- Συνάρτηση βιβλιοθήκης `pthread_exit`

- `void pthread_exit(void *retval)`
- Τερματίζει το νήμα που την καλεί
- Εφ' όσον το νήμα είναι συνεχώσιμο, που είναι η default περίπτωση, ο κωδικός εξόδου `retval` είναι διαθέσιμος σε οποιοδήποτε άλλο νήμα της διεργασίας που περιμένει, μέσω της συνάρτησης `pthread_join` η οποία θα περιγραφεί στη συνέχεια, τον τερματισμό του συγκεκριμένου νήματος

(Μπορεί να γίνει join)

Αποφύγετε επιστροφή δείκτη σε αυτόματες μεταβλητές.
Προτιμήστε δείκτη σε δεδομένα δημιουργημένα από `malloc`

Συναρτήσεις pthread_join, pthread_detach, pthread_self


- Συνάρτηση βιβλιοθήκης pthread_join
 - `int pthread_join(pthread_t thread, void **retaddr)` Θυμίζει την `waitpid` σε διεργασίες
 - Περιμένει τον τερματισμό του συνενώσιμου νήματος με ταυτότητα `thread`
 - Ο κωδικός εξόδου του νήματος που τερμάτισε, όπως δόθηκε με την `pthread_exit`, επιστρέφεται στο `*retaddr`
 - Επιστρέφει 0 σε επιτυχία

Έλεγχος ισότητας pthread_t με pthread_equal
- Συνάρτηση βιβλιοθήκης pthread_self
 - `pthread_t pthread_self()`
 - Επιστρέφει την ταυτότητα του νήματος που την καλεί
- Συνάρτηση βιβλιοθήκης pthread_detach
 - `int pthread_detach(pthread_t thread)`
 - Μετατρέπει το νήμα με ταυτότητα `thread` από συνενώσιμο σε αποσπασμένο
 - Επιστρέφει 0 σε επιτυχία
 - Ένα αποσπασμένο νήμα ελευθερώνει αμέσως τους πόρους που έχει δεσμεύσει μόλις τερματίσει, ενώ ένα συνενώσιμο το κάνει αυτό μόνο όταν κάποιο άλλο νήμα ζητήσει να συνενωθεί μαζί του, μέσω της `pthread_join`
 - Η κλήση της `pthread_join` για ένα αποσπασμένο νήμα αποτυγχάνει

Σημαντικό: ΔΕΝ τερματίζεται το νήμα. Απλά δηλώνει ότι θα αποδεσμευτούν οι πόροι του ΟΤΑΝ τερματίσει

Χρήση pthread_create, pthread_exit, pthread_join και pthread_self

```
#include <stdio.h>
#include <string.h> /* For strerror */
#include <stdlib.h> /* For exit */
#include <pthread.h> /* For threads */
#define perror2(s,e) fprintf(stderr, "%s: %s\n", s, strerror(e))
void *thread_f(void *argp){ /* Thread function */
    printf("I am the newly created thread %ld\n",
           pthread_self());
    pthread_exit((void *) 47);
}
main(){
    pthread_t thr;
    int err, status;
    /* New thread */
    if (err = pthread_create(&thr, NULL, thread_f, NULL)) {
        perror2("pthread_create", err);
        exit(1);
    }
    printf("I am original thread %ld and I created
           thread %ld\n", pthread_self(), thr);
    /* Wait for thread */
    if (err = pthread_join(thr, (void **) &status)) {
        perror2("pthread_join", err); /* termination */
        exit(1);
    }
    printf("Thread %ld exited with code %d\n", thr, status);
    printf("Thread %ld just before exiting (Original)\n",
           pthread_self());
    pthread_exit(NULL); }
```



Not recommended way of “exit”ing..
avoid using automatic values; use
malloc(ed) structs to return status

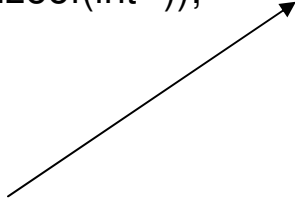
Run output

```
mema@browser> ./create_a_thread  
I am the newly created thread 134558720  
I am original thread 134557696 and I created thread 134558720  
Thread 134558720 exited with code 47  
Thread 134557696 just before exiting (Original)  
mema@browser>
```

whichexit() can be executed as a thread

```
void *whichexit(void *arg){
    int n;
    int np1[1];
    int *np2;
    char s1[10];
    char s2[] = "I am done";
    n = 3;
    np1 = n;
    np2 = (int *)malloc(sizeof(int *));
    *np2 = n;
    strcpy(s1,"Done");
    return(NULL);
}
```

1. **n**
2. **&n**
3. **(int *)n**
4. **np1**
5. **np2**
6. **s1**
7. **s2**
8. **“This works”**
9. **strerror(EINTR)**



Which of the above options could be *safe* replacement for NULL as return value in whichexit function?
(Or as a parameter to pthread_exit?)

1. No, return value is a pointer not an int
2. No – automatic variable
3. Might work (but not in all impls- avoid)
4. No – automatic variable
5. Yes – dynamically allocated
6. No – automatic variable
7. No – automatic storage
8. Yes – In C, string literals have static storage
9. No – The string produced by strerror might not exist

Χρήση pthread_detach

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <pthread.h>
#define perror2(s,e) fprintf(stderr,"%s: %s\n",s,strerror(e))

void *thread_f(void *argp){ /* Thread function */
    int err;
    if (err = pthread_detach(pthread_self())) { /* Detach thread */
        perror2("pthread_detach", err);
        exit(1);
    }
    printf("I am thread %d and I was called with\n",
           pthread_self(), *(int *) argp);
    pthread_exit(NULL);
}
main(){
    pthread_t thr;
    int err, arg = 29;
    /*New Thread */
    if (err = pthread_create(&thr,NULL,thread_f,(void *) &arg)){
        perror2("pthread_create", err);
        exit(1);
    }
    printf("I am original thread %d and I created thread %d\n",
           pthread_self(), thr);
    pthread_exit(NULL);
}
```

Print “detached” thread

Run output

```
mema@browser> ./detached_thread
```

```
I am thread 134558720 and I was called with argument 29
```

```
I am original thread 134557696 and I created thread 134558720
```

```
mema@browser>
```

Create n threads that wait for a random number of seconds and then terminate

```
#include <stdio.h>
#include <string.h> /* For strerror */
#include <stdlib.h> /* For exit */
#include <pthread.h> /* For threads */
#define perror2(s,e) fprintf(stderr, "%s: %s\n", s, strerror(e))
#define MAX_SLEEP 10

void *sleeping(void *arg) {
    int sl = (int) arg;
    printf("thread %ld sleeping %d seconds ...\n",
        pthread_self(), sl);
    sleep(sl); /* Sleep a number of seconds */
    printf("thread %ld waking up\n", pthread_self());
    pthread_exit(NULL);
}

main(int argc, char *argv[]){
    int n, i, sl, err;
    pthread_t *tids;
    if (argc > 1) n = atoi(argv[1]); /* Make integer */
    else exit(0);

    if (n > 50) { /* Avoid too many threads */
        printf("Number of threads should be no more
            than 50\n"); exit(0); }
    if ((tids = malloc(n * sizeof(pthread_t))) == NULL) {
        perror("malloc"); exit(1); }
```

```
srandom((unsigned int) time(NULL)); /* Initialize generator */
for (i=0 ; i<n ; i++) {
    /* Sleeping time 1..MAX_SLEEP */
    sl = random() % MAX_SLEEP + 1;
    if (err = pthread_create(&tids+i, NULL,
                           sleeping, (void *) sl)) {
        /* Create a thread */
        perror2("pthread_create", err);  exit(1);}
    }
for (i=0 ; i<n ; i++) {
    /* Wait for thread termination */
    if (err = pthread_join(&tids+i, NULL)) {
        perror2("pthread_join", err);
        exit(1);
    }
}
printf("all %d threads have terminated\n", n);
}
```

Sample output

```
mema@browser> ./create_many_threads 12
thread 134685184 sleeping 8 seconds ...
thread 134559232 sleeping 3 seconds ...
thread 134685696 sleeping 7 seconds ...
thread 134686208 sleeping 1 seconds ...
thread 134558720 sleeping 2 seconds ...
thread 134559744 sleeping 2 seconds ...
thread 134560256 sleeping 5 seconds ...
thread 134560768 sleeping 8 seconds ...
thread 134561280 sleeping 5 seconds ...
thread 134684672 sleeping 4 seconds ...
thread 134686720 sleeping 2 seconds ...
thread 134687232 sleeping 8 seconds ...
thread 134686208 waking up
thread 134558720 waking up
thread 134559744 waking up
thread 134686720 waking up
thread 134559232 waking up
thread 134684672 waking up
thread 134560256 waking up
thread 134561280 waking up
thread 134685696 waking up
thread 134685184 waking up
thread 134560768 waking up
thread 134687232 waking up
all 12 threads have terminated
mema@browser>
```


Going from a single-threaded program to multi-threading

```
#include <stdio.h>
#define NUM 5
```

```
void print_mesg(char *);
```

```
int main(){
    print_mesg("hello");
    print_mesg("world\n");
}
```

```
void print_mesg(char *m){
    int i;
    for (i=0; i<NUM; i++){
        printf("%s", m);
        fflush(stdout);
        sleep(1);
    }
}
```

```
mema@bowser> ./print_single
hellohellohellohelloworld
world
world
world
world
mema@bowser>
```

First attempt..

```
#include <stdio.h>
#include <pthread.h>
#define NUM 5

main()
{
    pthread_t t1, t2;
    void *print_mesg(void *);

    pthread_create(&t1, NULL, print_mesg, (void *)"hello ");
    pthread_create(&t2, NULL, print_mesg, (void *)"world\n");
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
}

void *print_mesg(void *m)
{
    char *cp = (char *)m;
    int i;
    for (i=0; i<NUM; i++){
        printf("%s", cp);
        fflush(stdout);
        sleep(2);
    }
    return NULL;
}
```

Output of 4 runs

```
mema@browser> ./multi_hello
```

```
hello world
```

```
hello world
```

```
hello world
```

```
hello world
```

```
hello world
```

```
mema@browser> ./multi_hello
```

```
hello world
```

```
world
```

```
hello hello world
```

```
hello world
```

```
hello world
```

```
mema@browser> ./multi_hello
```

```
world
```

```
hello world
```

```
hello hello world
```

```
hello world
```

```
hello world
```

```
mema@browser> ./multi_hello
```

```
world
```

```
hello hello world
```

```
hello world
```

```
world
```

```
hello world
```

```
hello mema@browser>
```

Another Example: Synchronization attempt (via *sleep()*)

```
#include <stdio.h>
#include <pthread.h>
#define NUM 5
int counter=0;
main(){
    pthread_t t1;
    void *print_count(void *);
    int i;

    pthread_create(&t1, NULL, print_count, NULL);
    for(i=0; i<NUM; i++){
        counter++;
        sleep(1);
    }
    pthread_join(t1, NULL);
}

void *print_count(void *m){
    /* counter is a shared variable */
    int i;
    for (i=0; i<NUM; i++){
        printf("count = %d\n", counter);
        sleep(1);
        /*changing this to something else has an effect */
    }
    return NULL;
}
```

Output

```
mema@browser> ./incprint
count = 0
count = 1
count = 2
count = 3
count = 4
mema@browser> ./incprint
count = 1
count = 1
count = 3
count = 4
count = 5
mema@browser> ./incprint
count = 1
count = 2
count = 3
count = 4
count = 5
mema@browser> ./incprint
count = 1
count = 1
count = 3
count = 4
count = 5
mema@browser>
```

Changing the *sleep(1)* to *sleep(0)* within *print_count()* we get the following:

```
mema@browser> ./incprint
count = 0
count = 0
count = 0
count = 0
count = 0
mema@browser>
```

Counting words from two distinct files

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <ctype.h>
int total_words;

int main(int ac, char *av[]){
    pthread_t t1, t2;
    void *count_words(void *);
    if (ac != 3 ) {
        printf("usage: %s file1 file2 \n", av[0]);
        exit(1); }
    total_words=0;

    pthread_create(&t1, NULL, count_words, (void *)av[1]);
    pthread_create(&t2, NULL, count_words, (void *)av[2]);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("Main thread with ID: %ld reports %5d total words\n",
        pthread_self(), total_words);
}
```

```
void *count_words(void *f){
    char *filename = (char *)f;
    FILE *fp;  int c, prevc = '\0';
    printf("In thread with ID: %ld counting words.. \n",
        pthread_self());
    if ( (fp=fopen(filename,"r")) != NULL ){
        while ( ( c = getc(fp) )!= EOF ){
            if ( !isalnum(c) && isalnum(prevc) )
                total_words++;
            prevc = c;
        }
        fclose(fp);
    } else perror(filename);
    return NULL;
}
```

Output

```
mema@browser> wc file1 file2
```

```
1      4     15 file1
```

```
1      4     17 file2
```

```
2      8     32 total
```

```
mema@browser> ./wordcount1 file1 file2
```

```
In thread with ID: 134559232 counting words..
```

```
In thread with ID: 134558720 counting words..
```

```
Main thread with ID: 134557696 reports 8 total words
```

```
mema@browser> ./wordcount1 file1 file2
```

```
In thread with ID: 134559232 counting words..
```

```
In thread with ID: 134558720 counting words..
```

```
Main thread with ID: 134557696 reports 6 total words
```

```
mema@browser>
```


Concurrent Access

Potential Problem:

Thread1

.....

.....

total_words++

.....

Thread2

.....

.....

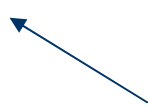
total_words++

.....

Race-condition: total_words might not have *consistent value* after executing the above two assignments.

Never allow concurrent access to data without protection (when at least one access is write)!

Δυναδικοί Σηματοφορείς

- Για το συγχρονισμό μεταξύ νημάτων που προτίθενται να προσπελάσουν κοινούς πόρους, η βιβλιοθήκη των POSIX νημάτων παρέχει μία απλοποιημένη εκδοχή σηματοφόρων, τους δυναδικούς σηματοφόρους (mutexes)
 - Ένας δυναδικός σηματοφόρος μπορεί να βρίσκεται σε μία από δύο πιθανές καταστάσεις, να είναι κλειδωμένος ή ξεκλειδωτός
 - Συνάρτηση βιβλιοθήκης `pthread_mutex_init`
 - `int pthread_mutex_init(pthread_mutex_t *mutex, pthread_mutexattr_t *attr)`
 - Αρχικοποιεί δυναμικά το σηματοφόρο `*mutex`
 - Μέσω του `attr` μπορούμε να ορίσουμε κάποια χαρακτηριστικά για το σηματοφόρο, αλλά συνήθως αφήνουμε τα default, δίνοντάς του την τιμή `NULL`
 - Ένας σηματοφόρος μπορεί να αρχικοποιηθεί και στατικά δίνοντάς του σαν τιμή τη σταθερά `PTHREAD_MUTEX_INITIALIZER`
 - Επιστρέφει πάντοτε 0
- `static pthread_mutex_t koko =
PTHREAD_MUTEX_INITIALIZER`

Αρχικοποίηση ΠΑΝΤΑ ΑΚΡΙΒΩΣ 1 φορά

Δυναδικοί Σηματοφορείς (συν)

- Συνάρτηση βιβλιοθήκης `pthread_mutex_lock`
 - `int pthread_mutex_lock(pthread_mutex_t *mutex)`
 - Κλειδώνει το σηματοφόρο `*mutex` εφ' όσον είναι ξεκλειδωτος
 - Αν ο σηματοφόρος είναι κλειδωμένος από άλλο νήμα, τότε το καλούν νήμα τίθεται σε αναστολή και η συνάρτηση επιστρέφει όταν κατορθώσει να κλειδώσει το σηματοφόρο, αφού τον ξεκλειδώσει το νήμα που τον κλειδωσε
 - Επιστρέφει 0 σε επιτυχία
- Συνάρτηση βιβλιοθήκης `pthread_mutex_unlock`
 - `int pthread_mutex_unlock(pthread_mutex_t *mutex)`
 - Ξεκλειδώνει το σηματοφόρο `*mutex` εφ' όσον έχει κλειδωθεί προηγουμένως από το ίδιο νήμα
 - Επιστρέφει 0 σε επιτυχία
- Συνάρτηση βιβλιοθήκης `pthread_mutex_destroy`
 - `int pthread_mutex_destroy(pthread_mutex_t *mutex)`
 - Καταστρέφει το σηματοφόρο `*mutex` εφ' όσον είναι ξεκλειδωτος
 - Δεν είναι αναγκαίο να καταστρέφονται οι σηματοφόροι που έχουν αρχικοποιηθεί στατικά
 - Επιστρέφει 0 σε επιτυχία

Υπάρχει και
`pthread_mutex_trylock`



Δυαδικοί Σηματοφορείς (συν)

Function:

int *pthread_mutex_trylock* (*pthread_mutex_t* **mutex*);

behaves identically to *pthread_mutex_lock*, except that it does not block the calling thread if the *mutex* is already locked by another thread.

Instead, *pthread_mutex_trylock* returns immediately with the error code *EBUSY*

If *pthread_mutex_trylock* returns the error code *EINVAL*, the mutex was not initialized properly.

Counting (corretly) words in two files

```
/* add all header files */
```

```
int          total_words;
pthread_mutex_t counter_lock =
    PTHREAD_MUTEX_INITIALIZER;
```

```
int  main(int ac, char *av[])
{
    pthread_t t1, t2;
    void *count_words(void *);
    if ( ac != 3 ) {
        printf("usage: %s file1 file2 \n", av[0]);
        exit(1); }
    total_words=0;
    pthread_create(&t1, NULL, count_words, (void *)av[1]);
    pthread_create(&t2, NULL, count_words, (void *)av[2]);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("Main thread wirth ID %ld reporting %5d
        total words\n", pthread_self(),total_words);
}
```

```
void *count_words(void *f){
    char *filename = (char *)f;
    FILE *fp; int c, prevc = '\0';

    if ( (fp=fopen(filename,"r")) != NULL ){
        while ( ( c = getc(fp) )!= EOF ){
            if ( !isalnum(c) && isalnum(prevc) ){
                pthread_mutex_lock(&cou
                total_words++;
                pthread_mutex_unlock(&c
                }
                prevc = c;
            }
            fclose(fp);
        } else perror(filename);
        return NULL;
    }
```

Another program that counts words in two files correctly

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
#include <pthread.h>
```

```
#include <ctype.h>
```

```
#define EXIT_FAILURE 1
```

```
struct arg_set{  
    char *fname;  
    int  count;  
};
```

```
int  main(int ac, char *av[]) {  
    pthread_t t1, t2;  
    struct arg_set args1, args2;  
    void *count_words(void *);
```

```
    if ( ac != 3 ) {  
        printf("usage: %s file1 file2 \n", av[0]);  
        exit (EXIT_FAILURE);  
    }
```

```
args1.fname = av[1]; args1.count = 0;
    pthread_create(&t1, NULL,
                  count_words, (void *) &args1);

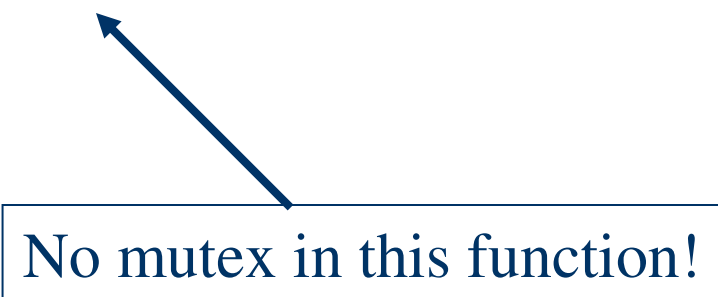
args2.fname = av[2]; args2.count = 0;
    pthread_create(&t2, NULL,
                  count_words, (void *) &args2);

pthread_join(t1, NULL);
pthread_join(t2, NULL);

printf("In file  %-10s there are %5d words\n",
      av[1], args1.count);
printf("In file  %-10s there are %5d words\n",
      av[2], args2.count);
printf("Main thread %ld reporting %5d
      total words\n", pthread_self(),
      args1.count+args2.count);
}
```

```
void *count_words(void *a) {
    struct arg_set *args = a;
    FILE *fp; int c, prevc = '\0';
    printf("Working within Thread with ID %ld
           and counting\n",pthread_self());

    if ( (fp=fopen(args->fname,"r")) != NULL ){
        while ( ( c = getc(fp) )!= EOF ){
            if ( !isalnum(c) && isalnum(prevc) ){
                args->count++;
            }
            prevc = c;
        }
        fclose(fp);
    } else perror(args->fname);
    return NULL;
}
```



No mutex in this function!

```
mema@browser> ./twordcount3 \  
    /etc/dictionaries-common/words \  
    /etc/dictionaries-common/ispell-default  
Working within Thread with ID 1210238064 and counting  
Working within Thread with ID 1218630768 and counting  
In file /etc/dictionaries-common/words there are 123261 words  
In file /etc/dictionaries-common/ispell-default there are 3 words  
Main thread 1210235216 reporting 123264 total words  
mema@browser>
```

ΠΡΟΣΟΧΗ ΣΤΑ ΠΑΡΑΚΑΤΩ:

- ♦ Η *pthread_mutex_trylock* επιστρέφει *EBUSY* αν ο σηματοφορέας είναι ήδη κλειδωμένος από άλλο νήμα
- ♦ Κάθε δυαδικός σηματοφορέας πρέπει να αρχικοποιηθεί ΑΚΡΙΒΩΣ 1 φορά
- ♦ Η *pthread_mutex_unlock* να καλείται ΜΟΝΟ από το νήμα που έχει κλειδωμένο το δυαδικό σηματοφορέα
- ♦ ΠΟΤΕ να μην καλείτε τη *pthread_mutex_lock* από το νήμα που έχει **ΗΔΗ** κλειδώσει το σηματοφορέα (στο είδος σηματοφορέων που αναφέραμε προκαλεί αδιέξοδο)
- ♦ Αν λάβετε σφάλμα *EINVAL* σε προσπάθεια κλειδώματος, τότε δεν έχετε αρχικοποιήσει το σηματοφορέα
- ♦ ΠΟΤΕ κλήση της *pthread_mutex_destroy* σε κλειδωμένο σηματοφορέα (*EBUSY*)

Using: pthread_mutex_init, pthread_mutex_lock,
pthread_mutex_unlock, pthread_mutex_destroy

.....

```
pthread_mutex_t mtx;           /* Mutex for synchronization */
char buf[25];                  /* Message to communicate */
void *thread_f(void *);        /* Forward declaration */
```

```
main() {
    pthread_t thr;
    int err;

    printf("Main Thread %ld running \n",pthread_self());
    pthread_mutex_init(&mtx, NULL);

    if (err = pthread_mutex_lock(&mtx)) { /* Lock mutex */
        perror2("pthread_mutex_lock", err); exit(1); }
    printf("Thread %d: Locked the mutex\n", pthread_self());

    /* New thread */
    if (err = pthread_create(&thr, NULL, thread_f, NULL)) {
        perror2("pthread_create", err); exit(1); }
    printf("Thread %ld: Created thread %d\n", pthread_self(), thr);

    strcpy(buf, "This is a test message");
    printf("Thread %ld: Wrote message \"%s\" for thread %ld\n",
        pthread_self(), buf, thr);
```

```
if (err = pthread_mutex_unlock(&mtx)) { /* Unlock mutex */
    perror2("pthread_mutex_unlock", err); exit(1);
}
printf("Thread %ld: Unlocked the mutex\n", pthread_self());

if (err = pthread_join(thr, NULL)) { /* Wait for thread */
    perror2("pthread_join", err); exit(1); } /* termination */

printf("Exiting Threads %ld and %ld\n", pthread_self(), thr);

if (err = pthread_mutex_destroy(&mtx)) { /* Destroy mutex */
    perror2("pthread_mutex_destroy", err); exit(1); }
pthread_exit(NULL);
}
```

Shall block here

```
void *thread_f(void *argp){ /* Thread function */
    int err;
    printf("Thread %ld: Just started\n", pthread_self());
    printf("Thread %ld: Trying to lock the mutex\n", pthread_self())

    if (err = pthread_mutex_lock(&mtx)) { /* Lock mutex */
        perror2("pthread_mutex_lock", err); exit(1); }

    printf("Thread %ld: Locked the mutex\n", pthread_self());
    printf("Thread %ld: Read message \"%s\"\n", pthread_self(), buf);

    if (err = pthread_mutex_unlock(&mtx)) { /* Unlock mutex */
        perror2("pthread_mutex_unlock", err); exit(1); }
    printf("Thread %ld: Unlocked the mutex\n", pthread_self());

    pthread_exit(NULL);
}
```

```
mema@browser> ./sync_by_mutex
Main Thread 1209685296 running
Thread 1209685296: Locked the mutex
Thread 1209689200: Just started
Thread 1209689200: Trying to lock the mutex
Thread 1209685296: Created thread 1209689200
Thread 1209685296: Wrote message "This is a test message"
\ for thread 1209689200
Thread 1209689200: Locked the mutex
Thread 1209689200: Read message "This is a test message"
Thread 1209689200: Unlocked the mutex
Thread 1209685296: Unlocked the mutex
Exiting Threads 1209685296 and 1209689200
mema@browser>
```

```
mema@linux01> ./sync_by_mutex
Main Thread -1217464640 running
Thread -1217464640: Locked the mutex
Thread -1217464640: Created thread -1217467536
Thread -1217464640: Wrote message "This is a test message"
for thread -1217467536
Thread -1217464640: Unlocked the mutex
Thread -1217467536: Just started
Thread -1217467536: Trying to lock the mutex
Thread -1217467536: Locked the mutex
Thread -1217467536: Read message "This is a test message"
Thread -1217467536: Unlocked the mutex
Exiting Threads -1217464640 and -1217467536
mema@linux01>
```


Sum the squares of n integers using m threads

.....

```
#include <pthread.h>
#define perror2(s, e) fprintf(stderr, "%s: %s\n", s, strerror(e))
#define LIMITUP 100

pthread_mutex_t mtx;    /* Mutex for synchronization */
int n, nthr, mtxfl;     /* Variables visible by thread function */
double sqsum;          /* Sum of squares */
void *square_f(void *); /* Forward declaration */

main(int argc, char *argv[]){
    int i, err;
    pthread_t *tids;
    if (argc > 3) {
        n = atoi(argv[1]);    /* Last integer to be squared */
        nthr = atoi(argv[2]); /* Number of threads */
        mtxfl = atoi(argv[3]); /* with lock (1)? or without lock (0) */
    } else exit(0);

    if (nthr > LIMITUP) { /* Avoid too many threads */
        printf("Number of threads should be up to 100\n"); exit(0); }
    if ((tids = malloc(nthr * sizeof(pthread_t))) == NULL) {
        perror("malloc"); exit(1); }
```

```
sqsum = (double) 0.0; /* Initialize sum */
pthread_mutex_init(&mtx, NULL); /* Initialize mutex */

for (i=0 ; i<nthr ; i++) {
    if (err = pthread_create(tids+i, NULL, square_f, (void *) i)) {
        /* Create a thread */
        perror2("pthread_create", err); exit(1); } }

for (i=0 ; i<nthr ; i++)
    if (err = pthread_join(*(tids+i), NULL)) {
        /* Wait for thread termination */
        perror2("pthread_join", err); exit(1); }
```

```

if (!mtxfl) printf("Without mutex\n");
else printf("With mutex\n");

printf("%2d threads: sum of squares up to %d is %12.9e\n",
    nthr,n,sqsum);
sqsum = (double) 0.0; /* Compute sum with a single thread */
for (i=0 ; i<n ; i++)
    sqsum += (double) (i+1) * (double) (i+1);
printf("Single thread: sum of squares up to %d is %12.9e\n",
    n, sqsum);
printf("Formula based: sum of squares up to %d is %12.9e\n",
    n, ((double) n)*(((double) n)+1)*(2*((double) n)+1)/6);
pthread_exit(NULL);
}

void *square_f(void *argp){ /* Thread function */
    int i, thri, err;
    thri = (int) argp;

    for (i=thri ; i<n ; i+=nthr) {
        if (mtxfl) /* Is mutex used? */
            if (err = pthread_mutex_lock(&mtx)) { /* Lock mutex */
                perror2("pthread_mutex_lock", err); exit(1); }

        sqsum += (double) (i+1) * (double) (i+1);

        if (mtxfl) /* Is mutex used? */
            if (err = pthread_mutex_unlock(&mtx)) { /*Unlock mutex */
                perror2("pthread_mutex_unlock", err); exit(1); } }
    pthread_exit(NULL); }

```

Execution outcome

mema@browser> ./sum_of_squares 12345678 99 1

With mutex

99 threads: sum of squares up to 12345678 is 6.272253963e+20

Single thread: sum of squares up to 12345678 is 6.272253963e+20

Formula based: sum of squares up to 12345678 is 6.272253963e+20

mema@browser> ./sum_of_squares 12345678 99 0

Without mutex

99 threads: sum of squares up to 12345678 is 4.610571900e+20

Single thread: sum of squares up to 12345678 is 6.272253963e+20

Formula based: sum of squares up to 12345678 is 6.272253963e+20

Synchronization & Performance

- ♦ Two threads, A and B
- ♦ A reads data from net and places in buffer, B reads data from buffer and computes with it

A: 1) Ανάγνωση δεδομένων
2) Κλείδωμα mutex
3) Τοποθέτηση στην ουρά
4) Ξεκλείδωμα mutex
5) Επιστροφή στο 1)

B: 1) Κλείδωμα mutex
2) If buf not empty,
αφαίρεση δεδομένων
3) Ξεκλείδωμα mutex
4) Επιστροφή στο 1)

Αυτό δουλεύει μια χαρά. Βλέπετε κάποιο πιθανό πρόβλημα;

Μεταβλητές Συνθήκης

- Εκτός από τους δυαδικούς σηματοφόρους, η βιβλιοθήκη των POSIX νημάτων παρέχει και ένα δεύτερο μέσο συγχρονισμού νημάτων, τις μεταβλητές συνθήκης
- Ένα νήμα είναι δυνατόν, μέσω μίας μεταβλητής συνθήκης, να αναστείλει την εκτέλεσή του μέχρις ότου ικανοποιηθεί κάποια συνθήκη
- Η ικανοποίηση της συνθήκης γίνεται γνωστή στο νήμα που έχει αναστείλει την εκτέλεσή του από άλλο νήμα μέσω κατάλληλου μηχανισμού ενημέρωσης
- Μία μεταβλητή συνθήκης είναι πάντοτε συνυφασμένη με ένα δυαδικό σηματοφόρο
- Συνάρτηση βιβλιοθήκης `pthread_cond_init`
 - `int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *cond_attr)`
 - Αρχικοποιεί δυναμικά τη μεταβλητή συνθήκης `*cond`
 - Μέσω του `cond_attr` μπορούμε να ορίσουμε κάποια χαρακτηριστικά για τη μεταβλητή συνθήκης, αλλά συνήθως αφήνουμε τα default, δίνοντάς του την τιμή `NULL`
 - Μία μεταβλητή συνθήκης μπορεί να αρχικοποιηθεί και στατικά δίνοντάς της σαν τιμή τη σταθερά `PTHREAD_COND_INITIALIZER`
 - Επιστρέφει πάντοτε `0`

Συνάρτηση `pthread_cond_wait`

- `int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex)`
- Ξεκλειδώνει το σηματοφόρο `*mutex` και αναστέλλει την εκτέλεση του καλούντος νήματος μέχρι να ενεργοποιηθεί και πάλι από κάποιο άλλο νήμα, μέσω της συνάρτησης `pthread_cond_signal` ή της `pthread_cond_broadcast`, των οποίων η περιγραφή θα ακολουθήσει, με βάση τη μεταβλητή συνθήκης `*cond`
- Το καλούν νήμα πρέπει πριν την κλήση της `pthread_cond_wait` να έχει κλειδώσει το σηματοφόρο `*mutex`
- Πριν επιστρέψει η `pthread_cond_wait`, κλειδώνει και πάλι το σηματοφόρο `*mutex`
- Το νήμα που θα ενεργοποιήσει το καλούν νήμα πρέπει πριν την κλήση της συνάρτησης ενεργοποίησής, π.χ. της `pthread_cond_signal`, να κλειδώσει το σηματοφόρο `*mutex` και μετά την κλήση να τον ξεκλειδώσει

Σημαντικό



Avoiding Lost Signals

- ◆ We want to avoid the following scenario:

T1 unlocks mutex;

Context switch;

T2 locks mutex, checks condition, about to wait

Context switch;

T1 signals on condition variable;

Context switch;

- ◆ T2 has not yet called `pthread_cond_wait`:
SIGNAL LOST
- ◆ OS does not accumulate/store signals
- ◆ When signal occurs, OS checks for all processes waiting on that signal and wakes them up

Συναρτήσεις: pthread_cond_signal, pthread_cond_broadcast, pthread_cond_destroy

- Συνάρτηση βιβλιοθήκης pthread_cond_signal
 - `int pthread_cond_signal(pthread_cond_t *cond)`
 - Ενεργοποιεί κάποιο νήμα που έχει αναστείλει την εκτέλεσή του με βάση τη μεταβλητή συνθήκης *cond
- Συνάρτηση βιβλιοθήκης pthread_cond_broadcast
 - `int pthread_cond_broadcast(pthread_cond_t *cond)`
 - Ενεργοποιεί όλα τα νήματα που έχουν αναστείλει την εκτέλεσή τους με βάση τη μεταβλητή συνθήκης *cond
- Συνάρτηση βιβλιοθήκης pthread_cond_destroy
 - `int pthread_cond_destroy(pthread_cond_t *cond)`
 - Καταστρέφει τη μεταβλητή συνθήκης *cond, εφ' όσον κανένα νήμα δεν έχει αναστείλει την εκτέλεσή του με βάση αυτήν
 - Δεν είναι αναγκαίο να καταστρέφονται οι μεταβλητές συνθήκης που έχουν αρχικοποιηθεί στατικά

Διαδικασία Χρήσης Μεταβλητών Συνθήκης

- ♦ Χρήση **ΕΝΟΣ ΜΟΝΟ** *mutex* με **κάθε** μεταβλητή συνθήκης
- ♦ Απόκτηση *mutex* πριν ελέγξετε κάποια συνθήκη
- ♦ Χρήση πάντα του ίδιου *mutex* για αλλαγές των μεταβλητών στη συνθήκη
- ♦ Να κρατάτε το *mutex* για μικρό μόνο διάστημα
- ♦ Απελευθέρωση στο τέλος με *pthread_mutex_unlock*

Use of: pthread_cond_init, pthread_cond_wait,
pthread_cond_signal, pthread_cond_destroy

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <pthread.h>
#define perror2(s, e) fprintf(stderr, "%s: %s\n", s, strerror(e))
```

```
pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
```

```
pthread_cond_t cvar;          /* Condition variable */
char buf[25];                 /* Message to communicate */
void *thread_f(void *);       /* Forward declaration */
```

```
main(){
    pthread_t thr; int err;
    /* Initialize condition variable */
    pthread_cond_init(&cvar, NULL);

    if (err = pthread_mutex_lock(&mtx)) { /* Lock mutex */
        perror2("pthread_mutex_lock", err); exit(1); }
    printf("Thread %d: Locked the mutex\n", pthread_self());

    /* New thread */
    if (err = pthread_create(&thr, NULL, thread_f, NULL)) {
        perror2("pthread_create", err); exit(1); }
    printf("Thread %d: Created thread %d\n", pthread_self(), thr);
```

```
printf("Thread %d: Waiting for signal\n", pthread_self());
pthread_cond_wait(&cvar, &mtx);          /* Wait for signal */
printf("Thread %d: Woke up\n", pthread_self());
printf("Thread %d: Read message \"%s\"\n",
      pthread_self(), buf);

if (err = pthread_mutex_unlock(&mtx)) { /* Unlock mutex */
    perror2("pthread_mutex_unlock", err); exit(1); }
printf("Thread %d: Unlocked the mutex\n", pthread_self());

if (err = pthread_join(thr, NULL)) { /* Wait for thread */
    perror2("pthread_join", err); exit(1); } /* termination */
printf("Thread %d: Thread %d exited\n", pthread_self(), thr);

if (err = pthread_cond_destroy(&cvar)) {
    /* Destroy condition variable */
    perror2("pthread_cond_destroy", err); exit(1); }
pthread_exit(NULL);
}
```

```
void *thread_f(void *argp){    /* Thread function */
    int err;

    printf("Thread %d: Just started\n", pthread_self());
    printf("Thread %d: Trying to lock the mutex\n", pthread_self());

    if (err = pthread_mutex_lock(&mtx))    /* Lock mutex */
        perror2("pthread_mutex_lock", err); exit(1); }
    printf("Thread %d: Locked the mutex\n", pthread_self());

    strcpy(buf, "This is a test message");

    printf("Thread %d: Wrote message \"%s\"\n",
        pthread_self(), buf);
    pthread_cond_signal(&cvar);    /* Awake other thread */
    printf("Thread %d: Sent signal\n", pthread_self());

    if (err = pthread_mutex_unlock(&mtx)) {    /* Unlock mutex */
        perror2("pthread_mutex_unlock", err); exit(1); }
    printf("Thread %d: Unlocked the mutex\n", pthread_self());

    pthread_exit(NULL);
}
```

Execution output

```
mema@browser> ./mutex_condvar
Thread 1210546512: Locked the mutex
Thread 1210549360: Just started
Thread 1210549360: Trying to lock the mutex
Thread 1210546512: Created thread 1210549360
Thread 1210546512: Waiting for signal
Thread 1210549360: Locked the mutex
Thread 1210549360: Wrote message "This is a test message"
Thread 1210549360: Sent signal
Thread 1210549360: Unlocked the mutex
Thread 1210546512: Woke up
Thread 1210546512: Read message "This is a test message"
Thread 1210546512: Unlocked the mutex
Thread 1210546512: Thread 1210549360 exited
mema@browser>
```

Three threads increase the value of a global variable while a fourth thread suspends its operation until a *maximum* value is reached.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define perror2(s, e) fprintf(stderr, "%s: %s\n", s, strerror(e))

#define COUNT_PER_THREAD 8 /* Count increments by each thread */
#define THRESHOLD 19      /* Count value to wake up thread */

int count = 0; /* The counter */
int thread_ids[4] = {0, 1, 2, 3}; /* My thread ids */

pthread_mutex_t mtx; /* mutex */
pthread_cond_t cv; /* the condition variable */

void *incr(void *argp){
    int i, j, err, *id = argp;
    for (i=0 ; i<COUNT_PER_THREAD ; i++) {
        if (err = pthread_mutex_lock(&mtx)) { /* Lock mutex */
            perror2("pthread_mutex_lock", err); exit(1); }
        count++; /* Increment counter */
        if (count == THRESHOLD) { /* Check for threshold */
            pthread_cond_signal(&cv); /* Signal suspended thread */
            printf("incr: thread %d, count = %d, threshold reached\n",
                *id, count);
        }
    }
}
```

```

printf("incr: thread %d, count = %d\n", *id, count);

    if (err = pthread_mutex_unlock(&mtx)) { /* Unlock mutex */
        perror2("pthread_mutex_unlock", err); exit(1); }
    for (j=0 ; j < 1000000000 ; j++); /* For threads to alternate */
}
pthread_exit(NULL);
}

void *susp(void *argp){
    int err, *id = argp;
    printf("susp: thread %d started\n", *id);

    if (err = pthread_mutex_lock(&mtx)) { /* Lock mutex */
        perror2("pthread_mutex_lock", err); exit(1);
    }
    while (count < THRESHOLD) { /* If threshold not reached */
        pthread_cond_wait(&cv, &mtx); /* suspend */
        printf("susp: thread %d, signal received\n", *id);
    }

    if (err = pthread_mutex_unlock(&mtx)) { /* Unlock mutex */
        perror2("pthread_mutex_unlock", err); exit(1);
    }

    pthread_exit(NULL); }

```

Always use a while loop and re-check the condition after receiving signal and returning from pthread_cond_wait; Why?


```

main() {
    int i, err;
    pthread_t threads[4];
    pthread_mutex_init(&mtx, NULL); /* Initialize mutex */
    pthread_cond_init(&cv, NULL); /* and condition variable */

    for (i=0 ; i<3 ; i++)
        if (err = pthread_create(&threads[i], NULL, incr,
                                (void *) &thread_ids[i])) {
            /* Create threads 0, 1, 2 */
            perror2("pthread_create", err); exit(1);
        }
    if (err = pthread_create(&threads[3], NULL, susp,
                            (void *) &thread_ids[3])) {
        /* Create thread 3 */
        perror2("pthread_create", err); exit(1); }

    for (i=0 ; i<4 ; i++)
        if (err = pthread_join(threads[i], NULL)) {
            perror2("pthread_join", err); exit(1);
        };
    /* Wait for threads termination */
    printf("main: all threads terminated\n");
    /* Destroy mutex and condition variable */
    if (err = pthread_mutex_destroy(&mtx)) {
        perror2("pthread_mutex_destroy", err); exit(1); }
    if (err = pthread_cond_destroy(&cv)) {
        perror2("pthread_cond_destroy", err); exit(1); }
    pthread_exit(NULL); }

```

```
mema@browser> ./counter
incr: thread 0, count = 1
incr: thread 1, count = 2
incr: thread 2, count = 3
susp: thread 3 started
incr: thread 0, count = 4
incr: thread 2, count = 5
incr: thread 1, count = 6
incr: thread 1, count = 7
incr: thread 0, count = 8
incr: thread 2, count = 9
incr: thread 1, count = 10
incr: thread 0, count = 11
incr: thread 2, count = 12
incr: thread 1, count = 13
incr: thread 0, count = 14
incr: thread 2, count = 15
incr: thread 1, count = 16
incr: thread 0, count = 17
incr: thread 2, count = 18
incr: thread 0, count = 19, threshold reached
incr: thread 0, count = 19
susp: thread 3, signal received
incr: thread 2, count = 20
incr: thread 1, count = 21
incr: thread 0, count = 22
incr: thread 2, count = 23
incr: thread 1, count = 24
main: all threads terminated
mema@browser>
```

Ασφάλεια Νημάτων

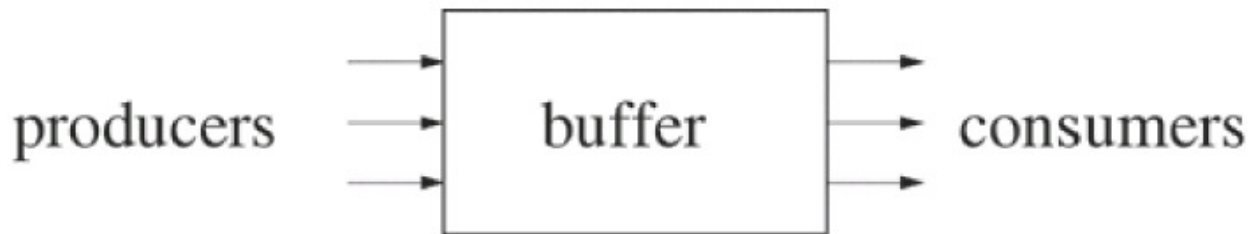
- ♦ Πρόβλημα επειδή πολλά νήματα μπορούν να καλέσουν συναρτήσεις που δεν είναι ασφαλείς
- ♦ Αποτέλεσμα πολλών συναρτήσεων συστήματος όχι ασφαλές

asctime	fcvt	getpwnam	nl_langinfo
basename	ftw	getpwuid	ptsname
catgets	gcvt	getservbyname	putc_unlocked
crypt	getc_unlocked	getservbyport	putchar_unlocked
ctime	getchar_unlocked	getservent	putenv
dbm_clearerr	getdate	getutxent	pututxline
dbm_close	getenv	getutxid	rand
dbm_delete	getgrent	getutxline	readdir
dbm_error	getgrgid	gmtime	setenv
dbm_fetch	getgrnam	hcreate	setgrent
dbm_firstkey	gethostbyaddr	hdestroy	setkey
dbm_nextkey	gethostbyname	hsearch	setpwent
dbm_open	gethostent	inet_ntoa	setutxent
dbm_store	getlogin	l64a	strerror
dirname	getnetbyaddr	lgamma	strtok
derror	getnetbyname	lgammaf	ttyname
drand48	getnetent	lgammal	unsetenv
ecvt	getopt	localeconv	wcstombs
encrypt	getprotobyname	localtime	wctomb

Ασφάλεια Νημάτων

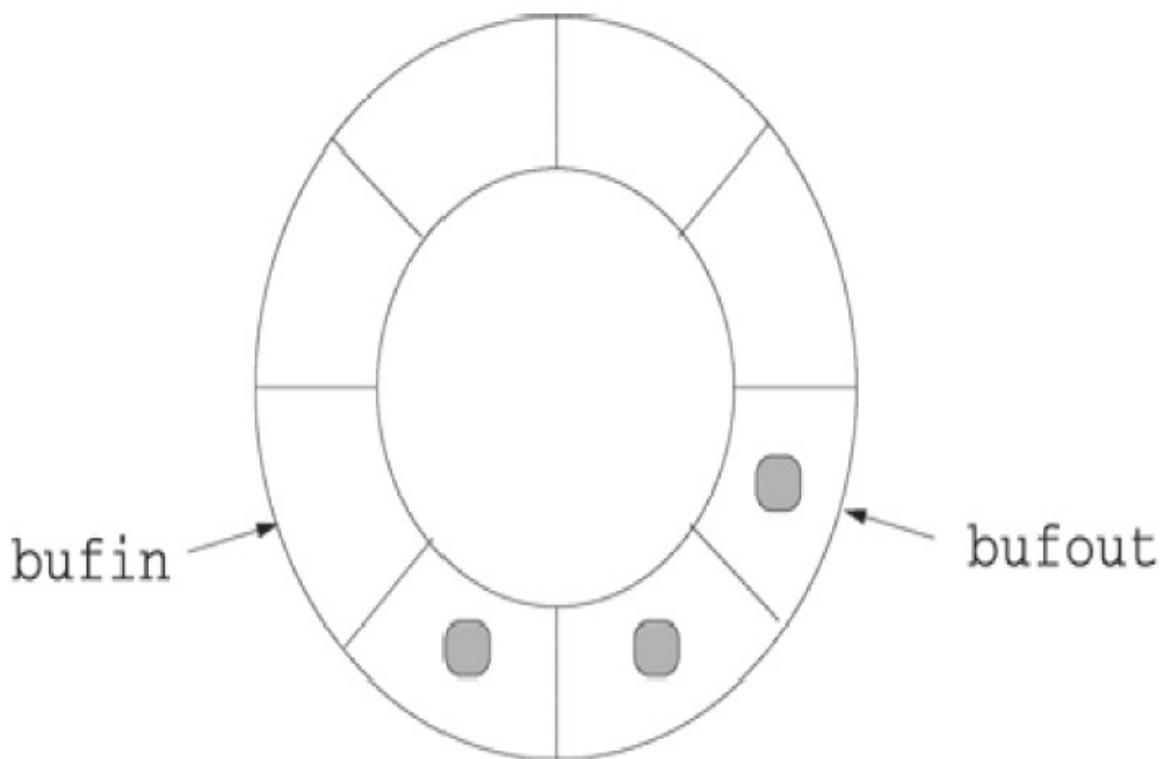
- ♦ Μπορείτε εύκολα να μετατρέψετε τις συναρτήσεις αυτές σε ασφαλείς με χρήση mutexes

Παράδειγμα Παραγωγού Καταναλωτή



- ◆ Παραγωγοί (Π) εισάγουν δεδομένα σε καταχωρητή
- ◆ Καταναλωτές (Κ) διαβάζουν δεδομένα από καταχωρητή
- ◆ Τι να αποφύγουμε?
 - Κ να διαβάσει αντικείμενο που ο Π δεν έχει ολοκληρώσει την εισαγωγή του
 - Κ αφαιρεί αντικείμενο που δεν υπάρχει
 - Κ αφαιρεί αντικείμενο που έχει ήδη αφαιρεθεί
 - Π προσθέτει αντικείμενο ενώ ο καταχωρητής δεν έχει χώρο
 - Π γράφει πάνω σε αντικείμενο που δεν έχει αφαιρεθεί

Χρήση Κυκλικού Καταχωρητή Περιορισμένου Μεγέθους



bufin: points at next available slot for storing item
bufout: points at slot where next reader should read from

Προστασία με Δυναδικούς Σηματοφορείς

```
#include <errno.h>
#include <pthread.h>
#include "buffer.h"
static buffer_t buffer[BUFSIZE];
static pthread_mutex_t bufferlock = PTHREAD_MUTEX_INITIALIZER;
static int bufin = 0;
static int bufout = 0;
static int totalitems = 0;

int getitem(buffer_t *item) { /* remove item from buffer and put in *item */
    int error;
    int erroritem = 0;
    if (error = pthread_mutex_lock(&bufferlock)) /* no mutex, give up */
        return error;
    if (totalitems > 0) { /* buffer has something to remove */
        *item = buffer[bufout];
        bufout = (bufout + 1) % BUFSIZE;
        totalitems--;
    } else
        erroritem = EAGAIN;
    if (error = pthread_mutex_unlock(&bufferlock))
        return error; /* unlock error more serious than no item */
    return erroritem;
}

int putitem(buffer_t item) { /* insert item in the buffer */
    int error;
    int erroritem = 0;
    if (error = pthread_mutex_lock(&bufferlock)) /* no mutex, give up */
        return error;
    if (totalitems < BUFSIZE) { /* buffer has room for another item */
        buffer[bufin] = item;
        bufin = (bufin + 1) % BUFSIZE;
        totalitems++;
    } else
        erroritem = EAGAIN;
    if (error = pthread_mutex_unlock(&bufferlock))
        return error; /* unlock error more serious than no slot */
    return erroritem;
}
```

Πόσα υπάρχουν στο buffer.
Χρειάζεται?

Τι επιστρέφει αν δεν
υπάρχουν δεδομένα?

fetching items from the buffer

The following piece of code attempts to retrieve 10 items from the buffer[8] ring...

```
int error, i, item;

for (i=0; i<10; i++){
    while ( (error = getitem(&item)) && (error== EAGAIN)) ;
    if (error) break;
    printf("Retrieved item %d: %d\n", i, item);
}
```

Problem??

- 1) busy waiting
- 2) producers might get blocked --
(readers might continuously grab lock first)

Προστασία με Μεταβλητές Συνθήκης (1)

```
// from www.mario-konrad.ch, changed slightly
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#define POOL_SIZE 6
typedef struct {
    int data[POOL_SIZE];
    int start;
    int end;
    int count;
} pool_t;

int num_of_items = 15;
pthread_mutex_t mtx;
pthread_cond_t cond_nonempty;
pthread_cond_t cond_nonfull;
pool_t pool;

void initialize(pool_t * pool) {
    pool->start = 0;
    pool->end = -1;
    pool->count = 0;
}

void place(pool_t * pool, int data) {
    pthread_mutex_lock(&mtx);
    while (pool->count >= POOL_SIZE) {
        printf(">> Found Buffer Full \n");
        pthread_cond_wait(&cond_nonfull, &mtx);
    }
    pool->end = (pool->end + 1) % POOL_SIZE;
    pool->data[pool->end] = data;
    pool->count++;
    pthread_mutex_unlock(&mtx);
}
```

Προστασία με Μεταβλητές Συνθήκης (2)

```
int obtain(pool_t * pool) {
    int data = 0;
    pthread_mutex_lock(&mtx);
    while (pool->count <= 0) {
        printf(">> Found Buffer Empty \n");
        pthread_cond_wait(&cond_nonempty, &mtx);
    }
    data = pool->data[pool->start];
    pool->start = (pool->start + 1) % POOL_SIZE;
    pool->count--;
    pthread_mutex_unlock(&mtx);
    return data;
}

void * producer(void * ptr)
{
    while (num_of_items > 0) {
        place(&pool, num_of_items);
        printf("producer: %d\n", num_of_items);
        num_of_items--;
        pthread_cond_signal(&cond_nonempty);
        usleep(300000);
    }
    pthread_exit(0);
}

void * consumer(void * ptr)
{
    while (num_of_items > 0 || pool.count > 0) {
        printf("consumer: %d\n", obtain(&pool));
        pthread_cond_signal(&cond_nonfull);
        usleep(500000);
    }
    pthread_exit(0);
}
```

main()

```
int main(){
    pthread_t cons, prod;

    initialize(&pool);
    pthread_mutex_init(&mtx, 0);
    pthread_cond_init(&cond_nonempty, 0);
    pthread_cond_init(&cond_nonfull, 0);

    pthread_create(&cons, 0, consumer, 0);
    pthread_create(&prod, 0, producer, 0);

    pthread_join(prod, 0);
    pthread_join(cons, 0);

    pthread_cond_destroy(&cond_nonempty);
    pthread_cond_destroy(&cond_nonfull);
    pthread_mutex_destroy(&mtx);
    return 0;
}
```

outcome

```
mema@browser> ./producer-consumer
>> Found Buffer Empty
producer: 15
consumer: 15
producer: 14
consumer: 14
producer: 13
producer: 12
consumer: 13
producer: 11
consumer: 12
producer: 10
producer: 9
consumer: 11
producer: 8
producer: 7
consumer: 10
producer: 6
consumer: 9
producer: 5
producer: 4
consumer: 8
producer: 3
producer: 2
consumer: 7
producer: 1
consumer: 6
consumer: 5
consumer: 4
consumer: 3
consumer: 2
consumer: 1
mema@browser>
```

Outcome - usleep(0)

```
mema@browser> ./producer-consumer
>> Found Buffer Empty
producer: 15
consumer: 15
producer: 14
producer: 13
producer: 12
producer: 11
producer: 10
producer: 9
>> Found Buffer Full
consumer: 14
producer: 8
>> Found Buffer Full
consumer: 13
producer: 7
>> Found Buffer Full
consumer: 12
producer: 6
>> Found Buffer Full
consumer: 11
producer: 5
>> Found Buffer Full
consumer: 10
producer: 4
>> Found Buffer Full
consumer: 9
producer: 3
>> Found Buffer Full
consumer: 8
producer: 2
>> Found Buffer Full
consumer: 7
producer: 1
consumer: 6
consumer: 5
consumer: 4
consumer: 3
consumer: 2
consumer: 1
mema@browser>
```

Food for Thought

- Where was the cond signal performed in relation to mutex lock/unlock in this example?
- Exercise: think about whether this is OK with X producers, X consumers, X CPUs, for $X \geq 1$.