# 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 διευθύνσεων, 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

```
cc -o <filename> <filename>.c -lpthread
```

  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

# Νήματα vs Διεργασίες

| | 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 |
| Περιγραφητές Αρχείων | Κοινοί. 2 νήματα χρησιμοποιούν ίδιο περιγραφητή. Αρκεί 1 close σε αυτόν | Αντίγραφα. 2 διεργασίες χρησιμοποιούν αντίγραφα του περιγραφητή. |
| 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 κεφάλαιο 13.5 ebook. Κάποια στο νήμα που τα προκάλεσε, κάποια στο πρώτο που δεν το 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
- Συνάρτηση βιβλιοθήκης pthread_exit
  - `void pthread_exit(void *retval)`
  - Τερματίζει το νήμα που την καλεί
  - Εφ' όσον το νήμα είναι συνενώσιμο, 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

Χαρακτηριστικό νήματος. (Μπορεί να γίνει join)

Αποφύγετε επιστροφή δείκτη σε αυτόματες μεταβλητές. Προτιμήστε δείκτη σε δεδομένα δημιουργημένα από malloc (κεφάλαιο 12.3.4 ebook)
    ```

# 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 ΌΤΑΝ τερματίσει

# Χρήση pthread_create, pthread_exit, pthread_join και pthread_self

```
/* File: create_a_thread.c */
#include <stdio.h>                                          /* For printf */
#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))
                            /* Error message printing function */
void *thread_f(void *);                         /* Forward declaration */

main()
{  pthread_t thr;
   int err, status;
   if (err = pthread_create(&thr, NULL, thread_f, NULL)) {
                                                          /* New thread */
      perror2("pthread_create", err);
      exit(1); }
   printf("I am original thread %d and I created thread %d\n",
          pthread_self(), thr);
   if (err = pthread_join(thr, (void **) &status)) {
                                                     /* Wait for thread */
      perror2("pthread_join", err);                      /* termination */
      exit(1); }
   printf("Thread %d exited with code %d\n", thr, status);
   pthread_exit(NULL); }

void *thread_f(void *argp)                         /* Thread function */
{  printf("I am the newly created thread %d\n", pthread_self());
   pthread_exit((void *) 47);

}
```

ΛΑΘΟΣ ΤΡΟΠΟΣ: Δείτε Επόμενη Διαφάνεια, παράδειγμα 12.14 του ebook

```
$ ./create_a_thread
I am the newly created thread 1026
I am original thread 1024 and I created thread 1026
Thread 1026 exited with code 47
$
```

# Παράδειγμα 12.14 ebook

```
void *whichexit(void *arg) {
   int n;
   int np1[1];
   int *np2;
   char s1[10];
   char s2[] = "I am done";
   n = 3;
   np1[0] = n;
   np2 = (int *)malloc(sizeof(int *));
   *np2 = n;
   strcpy(s1, "Done");
   return(NULL);
}
```

Ποια από τα παρακάτω μπορούν να χρησιμοποιηθούν ως τιμή επιστροφής?

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

1. Όχι. Απαιτείται δείκτης
2. Όχι. Το n είναι αυτόματη μεταβλητή
3. Όχι. Μετατροπή ακεραίου σε δείκτη και το αντίστροφο είναι ασφαλές ΜΟΝΟ για το 0.
4. Όχι. Ίδιο με 2.
5. Σωστό. Δυναμική μνήμη
6. Όχι. Ίδιο με 2.
7. Όχι. Ίδιο με 2.
8. Σωστό. Σταθερά string έχουν static αποθήκευση
9. Όχι. Μη εγγυημένο ότι το αποτέλεσμα του strerror θα υπάρχει αφού τερματίσει το νήμα

# Χρήση pthread_detach

```
/* File: detached_thread.c */
#include <stdio.h>                                      /* For printf */
#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))
                                 /* Error message printing function */
void *thread_f(void *);                      /* Forward declaration */

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

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 argument %d\n",
         pthread_self(), *(int *) argp);
   pthread_exit(NULL); }
```

Τυπώνεται. Η pthread_detach δεν τερματίζει το νήμα

```
$ ./detached_thread
I am original thread 1024 and I created thread 1026
I am thread 1026 and I was called with argument 29
$
```

* Να γραφεί ένα πρόγραμμα C που να δημιουργεί ένα αριθμό από νήματα, καθένα από τα οποία να καθυστερεί να τερματίσει για ένα τυχαίο αριθμό δευτερολέπτων.

```
/* File: random_sleeps.c */
#include <stdio.h>                                            /* For printf */
#include <string.h>                                         /* For strerror */
#include <stdlib.h>                             /* For srandom, random, exit */
#include <pthread.h>                                         /* For threads */
#define MAX_SLEEP 10                      /* Maximum sleeping time in seconds */
#define perror2(s, e) fprintf(stderr, "%s: %s\n", s, strerror(e))
                                          /* Error message printing function */
void *sleeping(void *);                                /* Forward declaration */

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 up to 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++) {
      sl = random() % MAX_SLEEP + 1;   /* Sleeping time 1..MAX_SLEEP */
      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++)
      if (err = pthread_join(*(tids+i), NULL)) {
                                            /* Wait for thread termination */
         perror2("pthread_join", err); exit(1); }
   printf("all %d threads have terminated\n", n); }

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

Όχι σωστό. Όμοιο πρόβλημα με πρόγραμμα create_a_thread.c

# Εκτέλεση προγράμματος

```
$ ./random_sleeps 10
thread 1026 sleeping 8 seconds ...
thread 2051 sleeping 5 seconds ...
thread 3076 sleeping 1 seconds ...
thread 4101 sleeping 9 seconds ...
thread 5126 sleeping 6 seconds ...
thread 6151 sleeping 9 seconds ...
thread 7176 sleeping 2 seconds ...
thread 8201 sleeping 3 seconds ...
thread 9226 sleeping 3 seconds ...
thread 10251 sleeping 1 seconds ...
thread 3076 awakening
thread 10251 awakening
thread 7176 awakening
thread 8201 awakening
thread 9226 awakening
thread 2051 awakening
thread 5126 awakening
thread 1026 awakening
thread 6151 awakening
thread 4101 awakening
all 10 threads have terminated
$
```

# Δυαδικοί Σηματοφορείς

- Για το συγχρονισμό μεταξύ νημάτων που προτίθενται να προσπελάσουν κοινούς πόρους, η βιβλιοθήκη των POSIX νημάτων παρέχει μία απλοποιημένη εκδοχή σηματοφόρων, τους δυαδικούς σηματοφόρους (mutexes)
- Ένας δυαδικός σηματοφόρος μπορεί να βρίσκεται σε μία από δύο πιθανές καταστάσεις, να είναι κλειδωμένος ή ξεκλείδωτος
- 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

# Δυαδικοί Σηματοφορείς (συν)

- Συνάρτηση βιβλιοθήκης pthread_mutex_lock
  - `int pthread_mutex_lock(pthread_mutex_t *mutex)`
  - Κλειδώνει το σηματοφόρο *mutex εφ' όσον είναι ξεκλείδωτος
  - 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 κλείδωσε
  - Επιστρέφει 0 σε επιτυχία

Υπάρχει και pthread_mutex_trylock

- 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 ξεκλείδωτος
  - Δεν είναι αναγκαίο να καταστρέφονται οι σηματοφόροι που έχουν αρχικοποιηθεί στατικά
  - Επιστρέφει 0 σε επιτυχία

# ΠΡΟΣΟΧΗ ΣΕ

- Η pthread_mutex_trylock επιστρέφει EBUSY αν ο σηματοφορέας είναι ήδη κλειδομένος 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)

- Χρήση των συναρτήσεων pthread_mutex_init, pthread_mutex_lock, pthread_mutex_unlock και pthread_mutex_destroy

```
/* File: sync_by_mutex.c */
#include <stdio.h>                                     /* For printf */
#include <string.h>                          /* For strcpy, strerror */
#include <stdlib.h>                                      /* For exit */
#include <pthread.h>                                  /* For threads */
#define perror2(s, e) fprintf(stderr, "%s: %s\n", s, strerror(e))
                                  /* Error message printing function */
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;
   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());
   if (err = pthread_create(&thr, NULL, thread_f, NULL)) {
                                                       /* New thread */
      perror2("pthread_create", err); exit(1); }
   printf("Thread %d: Created thread %d\n", pthread_self(), thr);
   strcpy(buf, "This is a test message");
   printf("Thread %d: Wrote message \"%s\" for thread %d\n",
         pthread_self(), buf, thr);
   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_mutex_destroy(&mtx)) {        /* Destroy mutex */
      perror2("pthread_mutex_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());
   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());
   pthread_exit(NULL); }
```

```
$ ./sync_by_mutex
Thread 1024: Locked the mutex
Thread 1026: Just started
Thread 1026: Trying to lock the mutex
Thread 1024: Created thread 1026
Thread 1024: Wrote message "This is a test message" for thread 1026
Thread 1024: Unlocked the mutex
Thread 1026: Locked the mutex
Thread 1026: Read message "This is a test message"
Thread 1026: Unlocked the mutex
Thread 1024: Thread 1026 exited
$
```

- *Να γραφεί ένα πρόγραμμα $C$ που να δημιουργεί ένα δεδομένο αριθμό νημάτων, τα οποία να αναλαμβάνουν να υπολογίσουν, για δεδομένο $n$, το $\sum_{i=1}^{n} i^2$, έχοντας καθένα από τα νήματα την ευθύνη για τον υπολογισμό περίπου ίσου πλήθους όρων του αθροίσματος.*

```
/* File: sum_of_squares.c */
#include <stdio.h>                                              /* For printf */
#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))
                                         /* Error message printing function */
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]); }                             /* Are we going to lock? */
   else          exit(0);
   if (nthr > 50) {                                  /* Avoid too many threads */
      printf("Number of threads should be up to 50\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 (err = pthread_mutex_destroy(&mtx)) {        /* Destroy mutex */
       perror2("pthread_mutex_destroy", 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); }
```

```
$ ./sum_of_squares 123456 23 1
With mutex
23 threads:    sum of squares up to 123456 is 6.272210524e+14
Single thread: sum of squares up to 123456 is 6.272210524e+14
Formula based: sum of squares up to 123456 is 6.272210524e+14
$ ./sum_of_squares 123456 23 0
Without mutex
23 threads:    sum of squares up to 123456 is 5.740972751e+14
Single thread: sum of squares up to 123456 is 6.272210524e+14
Formula based: sum of squares up to 123456 is 6.272210524e+14
$
```

# Ελεγχοι Συνθήκης???

- Τι πρόβλημα έχει το παρακάτω:

```
flag = 1
while (flag) {
  pthread_mutex_lock(&mut);
  if (x == y) {
    flag = 0;
    commands….
  }
  pthread_mutex_unlock(&mut);
}
```

# 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 ενεργοποίησης, π.χ. της pthread_cond_signal, να κλειδώσει το σηματοφόρο *mutex και μετά την κλήση να τον ξεκλειδώσει
- Επιστρέφει πάντοτε 0

- 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
  - Επιστρέφει πάντοτε 0
- 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
  - Επιστρέφει πάντοτε 0
- 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
  - Επιστρέφει 0 σε επιτυχία

# Διαδικασία Χρήσης Μεταβλητών Συνθήκης

- Χρήση ΕΝΌΣ ΜΟΝΟ mutex με κάθε μεταβλητή συνθήκης
- Απόκτηση mutex πριν ελέγξετε κάποια συνθήκη
- Σε αποτυχία, pthread_cond_wait
- Μετά από την επιστροφή της pthread_cond_wait επανέλεγχος συνθήκης (μπορεί να επιστρέψει λανθασμένα, πχ από σήμα)
- 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

- Χρήση των συναρτήσεων pthread_cond_init, pthread_cond_wait, pthread_cond_signal και pthread_cond_destroy

```c
/* File: mutex_condvar.c */
#include <stdio.h>                                          /* For printf */
#include <string.h>                               /* For strcpy, strerror */
#include <stdlib.h>                                           /* For exit */
#include <pthread.h>                                       /* For threads */
#define perror2(s, e) fprintf(stderr, "%s: %s\n", s, strerror(e))
                                       /* Error message printing function */
pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
                                             /* Mutex for synchronization */
pthread_cond_t cvar;                                /* Condition variable */
char buf[25];                                     /* Message to communicate */
void *thread_f(void *);                            /* Forward declaration */

main()
{  pthread_t thr; int err;
   pthread_cond_init(&cvar, NULL); /* Initialize condition variable */
   if (err = pthread_mutex_lock(&mtx)) {                   /* Lock mutex */
      perror2("pthread_mutex_lock", err); exit(1); }
   printf("Thread %d: Locked the mutex\n", pthread_self());
   if (err = pthread_create(&thr, NULL, thread_f, NULL)) {
                                                          /* New thread */
      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); }
```

```
$ ./mutex_condvar
Thread 1024: Locked the mutex
Thread 1026: Just started
Thread 1026: Trying to lock the mutex
Thread 1024: Created thread 1026
Thread 1024: Waiting for signal
Thread 1026: Locked the mutex
Thread 1026: Wrote message "This is a test message"
Thread 1026: Sent signal
Thread 1026: Unlocked the mutex
Thread 1024: Woke up
Thread 1024: Read message "This is a test message"
Thread 1024: Unlocked the mutex
Thread 1024: Thread 1026 exited
$
```

- *Να γραφεί ένα πρόγραμμα C που να δημιουργεί τέσσερα νήματα, τα τρία από τα οποία να αυξάνουν ένα καθολικό μετρητή, ενώ το τέταρτο να αναστέλλει την εκτέλεσή του μέχρι ο μετρητής να φτάσει συγκεκριμένη τιμή.*

```
/* File: counter.c */
#include <stdio.h>                                        /* For printf */
#include <stdlib.h>                                         /* For exit */
#include <pthread.h>                                     /* For threads */
#define perror2(s, e) fprintf(stderr, "%s: %s\n", s, strerror(e))
                                   /* Error message printing function */
#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;                                        /* My mutex */
pthread_cond_t cv;                               /* My 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 < 1000000 ; j++); }     /* For threads to alternate */
                                                     /* on mutex lock */
   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); }

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); }
```

# Εκτέλεση Προγράμματος

```
$ ./counter
incr: thread 0, count = 1
incr: thread 1, count = 2
incr: thread 0, count = 3
incr: thread 2, count = 4
incr: thread 0, count = 5
incr: thread 1, count = 6
susp: thread 3 started
incr: thread 2, count = 7
incr: thread 1, count = 8
incr: thread 2, count = 9
incr: thread 1, count = 10
incr: thread 2, count = 11
incr: thread 1, count = 12
incr: thread 2, count = 13
incr: thread 0, count = 14
incr: thread 2, count = 15
incr: thread 0, count = 16
incr: thread 2, count = 17
incr: thread 0, count = 18
incr: thread 1, count = 19, threshold reached
incr: thread 1, count = 19
susp: thread 3, signal received
incr: thread 0, 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
$
```

# Ασφάλεια Νημάτων

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

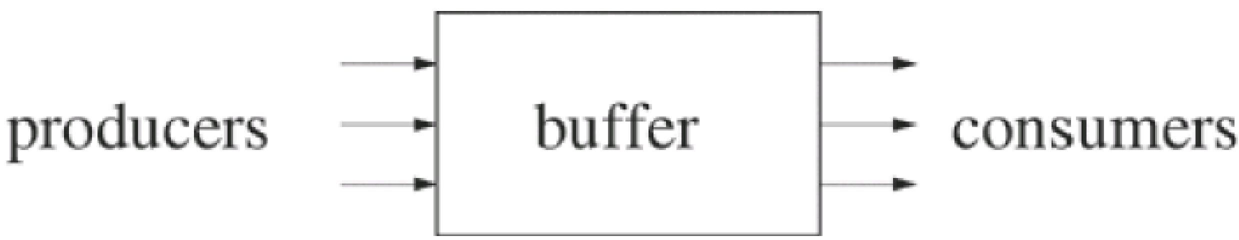
| | | | |
|---|---|---|---|
| asctime | fcvt | getpwnam | nl_langinfo |
| basename | ftw | getpwuid | ptsname |
| catgets | gcvt | getservbyname | putc_unlocked |
| crypt | getc_unlocked | getservbyport | putchar_unlocked |
| ctime | getchar_unlocked | getservent | putenv |
| dbm_clearerr | getdate | 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 |
| dlerror | getnetbyname | lgammaf | ttyname |
| drand48 | getnetent | lgammal | unsetenv |
| ecvt | getopt | localeconv | wcstombs |
| encrypt | getprotobyname | localtime | wctomb |

# Ασφάλεια Νημάτων (2)

- Μπορείτε εύκολα να μετατρέψετε τις συναρτήσεις αυτές σε ασφαλείς με χρήση δυαδικών σηματοφορέων (παράδειγμα 13.17 του ebook)

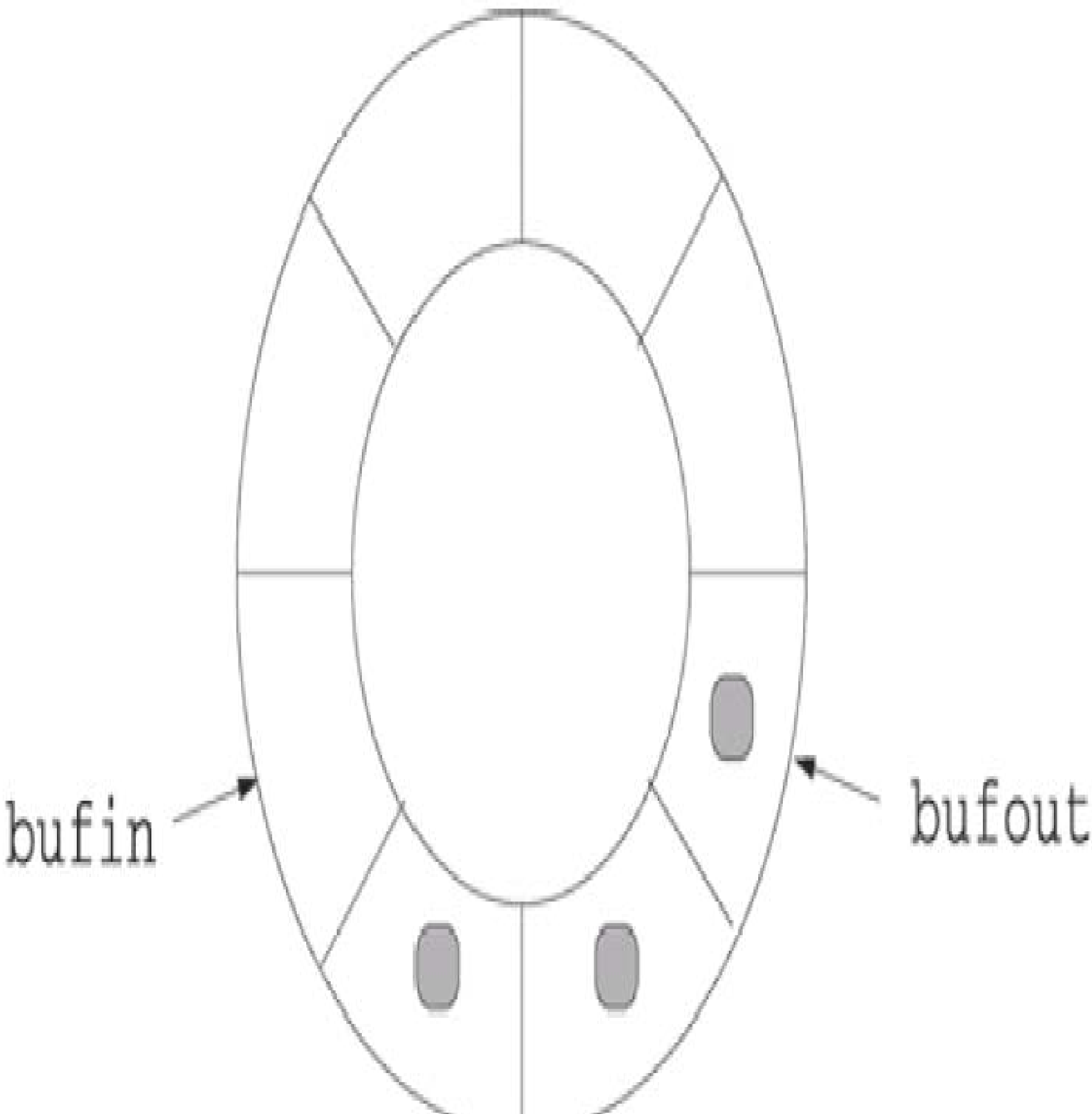
```
int perror_r(const char *s) {
  int error1;
  int error2;
  sigset_t maskblock;
  sigset_t maskold;

  if ((sigfillset(&maskblock) == -1) ||
     (sigprocmask(SIG_SETMASK, &maskblock, &maskold) == -1))
    return errno;
  if (error1 = pthread_mutex_lock(&lock)) {
    (void)sigprocmask(SIG_SETMASK, &maskold, NULL);
    return error1;
  }
  perror(s);
  error1 = pthread_mutex_unlock(&lock);
  error2 = sigprocmask(SIG_SETMASK, &maskold, NULL);
  return error1 ? error1 : error2;
}
```

# Παράδειγμα Παραγωγού Καταναλωτή (κεφ. 16 ebook)




- Παραγωγοί (Π) εισάγουν δεδομένα σε καταχωρητή
- Καταναλωτές (Κ) διαβάζουν δεδομένα από καταναλωτή
- **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



# Προστασία με Δυαδικούς 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;
```

Πόσα υπάρχουν στο buffer? Χρειάζεται?

```
int getitem(buffer_t *itemp) {  /* remove item from buffer and put in *itemp */
   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 */
      *itemp = 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;
}
```

Τι επιστρέφει αν δεν υπάρχουν δεδομένα?

```
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;
}
```

# Προστασία με Μεταβλητές Συνθήκης (1)

```
#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 pthread_cond_t items = PTHREAD_COND_INITIALIZER;
static pthread_cond_t slots = PTHREAD_COND_INITIALIZER;
static int totalitems = 0;

int getitem(buffer_t *itemp) { /* remove an item from buffer and put in itemp */
   int error;
   if (error = pthread_mutex_lock(&bufferlock))
      return error;
   while ((totalitems <= 0) && !error)
      error = pthread_cond_wait (&items, &bufferlock);
   if (error) {
      pthread_mutex_unlock(&bufferlock);
      return error;
   }
   *itemp = buffer[bufout];
   bufout = (bufout + 1) % BUFSIZE;
   totalitems--;
   if (error = pthread_cond_signal(&slots)) {
      pthread_mutex_unlock(&bufferlock);
      return error;
   }
   return pthread_mutex_unlock(&bufferlock);
}
```

# Προστασία με Μεταβλητές Συνθήκης (2)

```
int putitem(buffer_t item) {            /* insert an item in the buffer */
  int error;
  if (error = pthread_mutex_lock(&bufferlock))
    return error;
  while ((totalitems >= BUFSIZE) && !error)
    error = pthread_cond_wait (&slots, &bufferlock);
  if (error) {
    pthread_mutex_unlock(&bufferlock);
    return error;
  }
  buffer[bufin] = item;
  bufin = (bufin + 1) % BUFSIZE;
  totalitems++;
  if (error = pthread_cond_signal(&items)) {
    pthread_mutex_unlock(&bufferlock);
    return error;
  }
  return pthread_mutex_unlock(&bufferlock);
}
```