

---

# Topic 6: Threads\*

# Threads - Νήματα

---

- Lightweight Processes (LWPs)

- share single address space,
- each has its own flow control

- Try to overcome penalties when it comes to context switching between different “flows” (or sequence) of execution

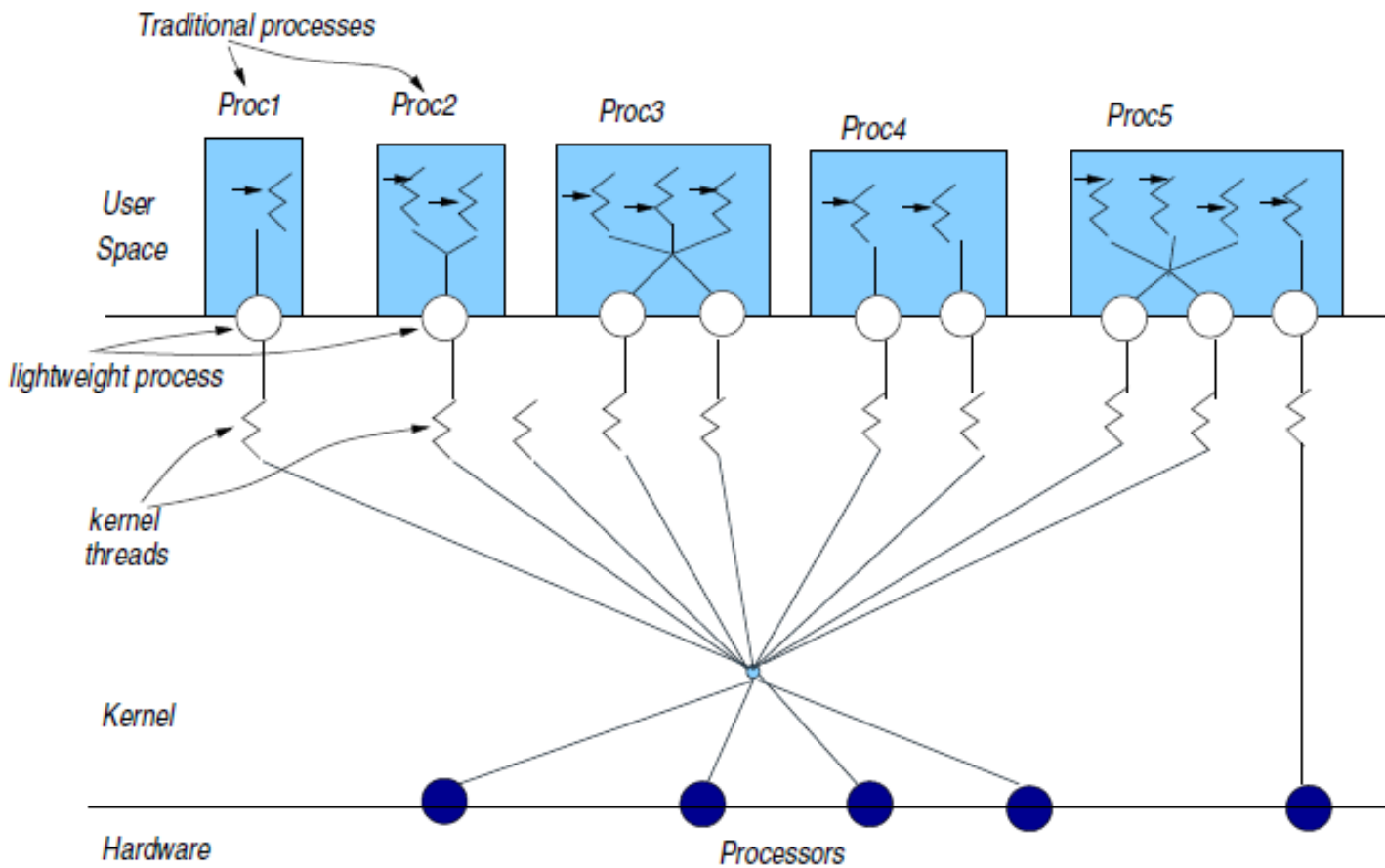
- separate process for every flow of control is costly

# Why Threads?

---

- 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)
  
- Question: In our network server example, if creating multiple processes is a performance issue, why not just use `select()`?
  
- What do threads give us over `select()`-based network server?

# Thread (Solaris) Model



# Threads Overview

---

- ◆ One or more threads may be executed in the context of a process.
- ◆ The entity that is being scheduled is the thread – **not** the process itself.
- ◆ In the presence of a single processor, threads are executed concurrently.
- ◆ If there are more than one processors, threads can be assigned to different kernel thread (and so different CPUs) and run in parallel.
- ◆ Any thread may create a new thread.

# Threads Overview

---

- ◆ All threads of a single process share the same address space (global, static, heap, file descriptors etc.) BUT they have their own PC, stack and set of registers.
- ◆ The kernel can perform context switches from one thread to another faster than from one process to another.
- ◆ The header `#include <pthread.h>` is required by all programs that use threads.
- ◆ Programs have to be compiled with the pthread library.
  - `gcc <filename>.c -lpthread`

# Threads Overview

---

- ◆ The functions of the pthread library do not set the value of the variable *errno* and so, we cannot use the function *perror()* for the printing of a diagnostic message.
- ◆ If there is an error in one of the thread functions, *strerror()* is used for the printing of the diagnostic code (which is the “function return” for the thread).

`char *strerror(int errnum)`

- returns a pointer to a string that describes the error code passed in the argument *errnum*.
- requires: `#include <string.h>`

# Threads vs. Processes

	Threads	Processes
Address space	Common. Any change made by one thread is visible to all (i.e., <code>malloc()/free()</code> )	Different for each process. After a <code>fork()</code> we have different address spaces
File descriptors	Common. Any two threads can use the same descriptor. One <code>close()</code> on this descriptor is sufficient	Two processes use copies of the file descriptors
<code>fork()</code>	On a <code>fork()</code> , only the thread that invoked the <code>fork</code> is duplicated.	
<code>exit()</code>	On <code>exit()</code> , all threads die together ( <code>pthread_exit</code> for the termination of a single thread).	
<code>exec()</code>	All threads disappear (the shared/common address space is replaced)	
Signals	This is somewhat more complex. See textbook.	



# POSIX Thread Management

---

POSIX function	description
<code>pthread_create</code>	create a thread
<code>pthread_self</code>	find out own thread ID
<code>pthread_equal</code>	test 2 thread IDs for equality
<code>pthread_exit</code>	exit thread without existing process
<code>pthread_detach</code>	set thread to release resources
<code>pthread_join</code>	wait for a thread
<code>pthread_cancel</code>	terminate another thread
<code>pthread_kill</code>	send a signal to a thread

# Thread creation

---

```
int pthread_create(pthread_t *thread,  
                  const pthread_attr_t *attr,  
                  void *(*start_func)(void *), void *arg);
```

- ◆ Creates a new thread with attributes specified by *attr* within a process.
- ◆ Thread executes the function at address *start\_func*
- ◆ Upon successful completion, `pthread create()` shall store the ID of the created thread in the location referenced by *thread*.
- ◆ Through *attr* we can change features of the thread but oftentimes we use the default attribute values, by setting *attr* to `NULL`.
- ◆ If successful, returns 0; otherwise, an error number shall be returned to indicate the error.

# Thread termination

---

`void pthread_exit (void *retval);`

- ◆ terminates the calling thread and makes the value *retval* available to any successful `pthread_join()` with the terminating thread.
- ◆ After a thread has terminated, the result of access to local (auto) variables of the thread is undefined.
- ◆ References to local variables of the exiting thread should **not** be used for the *retval* parameter value.

# pthread join - waiting for thread termination

---

```
int pthread_join (pthread_t thread,  
                 void ** retval );
```

- ◆ suspends execution of the calling thread until the target thread terminates (unless the target thread has already terminated).
- ◆ When a pthread\_join() returns successfully, the target thread has been terminated.
- ◆ On successful completion, the function returns 0.
- ◆ If retval is not NULL, then pthread\_join() copies the exit status of the target thread into the location pointed to by \*retval.

# Identifying – Detaching Threads

---

`pthread_t pthread_self(void);`

- ◆ Returns the thread-ID of the calling thread

`int pthread_detach(pthread_t thread);`

- ◆ indicates that the storage for the thread can be reclaimed only when the thread terminates.
- ◆ If thread has not terminated, `pthread_detach()` shall not cause it to terminate.
- ◆ If the call succeeds, `pthread_detach()` shall return 0; otherwise, an error number shall be returned.
- ◆ Calling `pthread_join()` on a detached thread fails.

# Χρήση pthread\_create, pthread\_exit, pthread\_join και pthread\_self

create\_a\_thread.c

```
#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());
    printf("Hanging on for a couple of secs!\n");
    sleep(2);
    printf("I am the newly created thread and about to
           exit! \n");
    pthread_exit((void *) 47);
}
```

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

```
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);
}
```

Χρήση pthread\_create, pthread\_exit, pthread\_join  
και pthread\_self

create\_a\_thread.c

---

```
/* Wait for thread */
```

```
if (err = pthread_join(thr, (void **) &status)) {  
    perror2("pthread_join", err); /* termination */  
    exit(1);  
}  
else printf("Just joined the two threads ->one!\n");  
printf("Thread %ld exited with code %d\n", thr, status);  
printf("Thread %ld just before exiting (Original)\n",  
        pthread_self());  
pthread_exit(NULL);  
}
```

# Run output

---

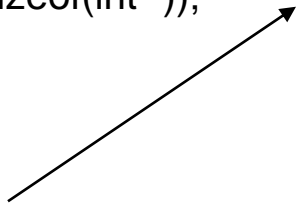
```
mema@browser> ./create_a_thread
I am original thread 8327046400 and I created thread 6097285120
I am the newly created thread 6097285120
Hanging on for a couple of secs!
I am the newly created thread and about to exit!
Just joined the two threads ->one!
Thread 6097285120 exited with code 47
Thread 8327046400 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)

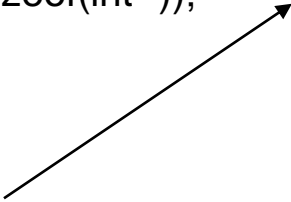


Which of the above options could be *safe* replacement for NULL as return value in whichexit function?  
(Or as a parameter to pthread\_exit?)

# 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

detached\_thread.c

```
#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
           argument %d\n", pthread_self(), *(int *) argp);
    printf("I will wait for some time in detached now..10 secs\n");
    sleep(10);
    printf("About to leave detached thread now.. Id\n",
           pthread_self());
    pthread_exit(NULL);
}
```

# Χρήση pthread\_detach

detached\_thread.c

```
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);
}
sleep(2);
printf("I am original thread %d and I created thread %d\n",
    pthread_self(), thr);
printf("I am the original and I am done! \n");
pthread_exit(NULL);
}
```

# Run output

---

```
mema@browser> ./detached_thread  
I am thread 6133919744 and I was called with argument 29  
I will wait for some time in detached now.. 10 secs  
I am original thread 8327046400 and I created thread 6133919744  
I am the original and I am done!  
About to leave detached thread now.. 6133919744  
mema@browser>
```

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

random\_sleeps.c

```
#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) {
    long sl = (long) 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, err;
    long sl;
    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> ./random_sleeps 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@browser> ./print_single  
hellohellohellohelloworld  
world  
world  
world  
world  
mema@browser>
```

# First attempt: multi\_hello.c

---

```
#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;
}
```

incprint.c

# 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 = 1
```

```
count = 1
```

```
count = 1
```

```
count = 1
```

```
count = 1
```

```
mema@browser>
```

# Counting words from two distinct files

wordcount1.c

```
#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)!

# Binary POSIX Mutexes

---

- ◆ When threads share common structures (resources), the POSIX library offers a simplified version of semaphores termed binary semaphores or mutexes.
- ◆ A binary semaphore can find itself in only two states: locked or unlocked.
- ◆ **int pthread\_mutex\_init(pthread\_mutex\_t \*mutex, const pthread\_mutexattr\_t \*mutexattr)**
  - Initializes the mutex-object pointed to by `mutex` according to the mutex attributes specified in `mutexattr`.
  - `pthread_mutex_init` always returns 0
- ◆ A mutex may also be initialized by setting its value by the macro
- ◆ **static pthread\_mutex\_t mymutex = PTHREAD\_MUTEX\_INITIALIZER;**
- ◆ Initialization of a mutex should occur **only once**

# Locking a mutex

---

- ◆ Locking a mutex is carried out by:

```
int pthread_mutex_lock(pthread_mutex_t * mutex)
```

- ◆ If the mutex is currently unlocked, it becomes locked and owned by the calling thread, and `pthread_mutex_lock` returns immediately.
- ◆ If successful, `pthread_mutex_lock` returns 0.
- ◆ If the mutex is already locked by another thread, `pthread_mutex_lock` blocks (or “suspends” for the user) the calling thread until the mutex is unlocked.

# Locking a mutex

---

- ◆ `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.

# Unlocking and destroying a mutex

---

```
int pthread_mutex_unlock ( pthread_mutex_t *mutex)
```

- ◆ If the mutex has been locked and owned by the calling thread, the mutex gets unlocked.
- ◆ Upon successful call, it returns 0.

```
int pthread_mutex_destroy(pthread_mutex_t *mutex)
```

- ◆ Destroys the mutex, freeing resources it might hold.
- ◆ In the LinuxThreads implementation, the call does nothing except checking that mutex is unlocked.
- ◆ Upon successful call, it returns 0.

# Counting (correctly) words in two files

wordcount2.c

```
/* 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 with 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(&counter_lock);
                total_words++;
                pthread_mutex_unlock(&counter_lock);
            }
            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> ./wordcount2 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>
```



# Another program that counts words in two files correctly

wordcount3.c

```
#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>
```

# Tips

---

- ◆ *pthread\_mutex\_trylock()* returns *EBUSY* if the mutex is already locked by another thread
- ◆ Every mutex must be initialized **only once**
- ◆ *pthread\_mutex\_unlock()* should be called **only** by the thread holding the mutex
- ◆ **NEVER** have *pthread\_mutex\_lock()* called by the thread that has **already locked** the mutex. A **deadlock** will occur
- ◆ If *EINVAL* is returned when trying to lock a mutex, then the mutex has not been initialized properly
- ◆ **NEVER** call *pthread\_mutex\_destroy()* on a locked mutex (*EBUSY*)

Using: pthread\_mutex\_init, pthread\_mutex\_lock,  
pthread\_mutex\_unlock, pthread\_mutex\_destroy

sync\_by\_mutex.c

```
.....  
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);
```

## sync\_by\_mutex.c

```
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@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

sum\_of\_squares.c

```
.....
#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 buffer not empty,  
αφαίρεση δεδομένων  
3) Ξεκλείδωμα mutex  
4) Επιστροφή στο 1)

Αυτό δουλεύει μια χαρά. Βλέπετε κάποιο πιθανό πρόβλημα;

# Condition Variables

---

- ◆ A condition variable is a synchronization mechanism that allows POSIX threads to suspend execution and relinquish the processors until some predicate on the shared data is satisfied.
- ◆ Basic operations on condition variables:
  - **signal** the condition (when the predicate becomes true)
  - **wait** for the condition, suspending execution
  - The waiting lasts until another thread “signals” (also called “notifies”) the condition
- ◆ A condition variable **must always be associated with a mutex** to avoid a race condition:
  - A thread prepares to wait on a condition variable and another thread signals the condition **just before** the first thread actually waits on the condition variable

# Initializing a condition variable

---

```
int pthread_cond_init(pthread_cond_t *cond,  
                      pthread_condattr_t *cond_attr)
```

- ◆ Initializes the condition variable `cond`, using the condition attributes specified in `cond_attr`, or default attributes of `cond_attr` is simply `NULL`
- ◆ Always returns 0
- ◆ The LinuxThreads implementation does not support attributes for condition variables (`cond_attr` is ignored).
- ◆ Variables of type `pthread_cond_t` can also be initialized statically, using the constant `PTHREAD_COND_INITIALIZER`.



# Waiting on a condition

---

```
int pthread_cond_wait (pthread_cond_t *cond  
                      pthread_mutex_t *mutex);
```

- ◆ atomically unlocks the mutex and waits for the condition variable cond to be signaled.
- ◆ Before calling pthread\_cond\_wait() the thread must have \*mutex locked
- ◆ The thread's execution is suspended and the thread does not consume any CPU time until the condition variable is signaled (via a call by another thread to pthread\_cond\_signal() or pthread\_cond\_broadcast())
- ◆ Before returning to the calling thread, pthread\_cond\_wait re-acquires mutex
- ◆ Always returns 0

# Signaling a condition variable

---

```
int pthread_cond_signal(pthread_cond_t *cond)
```

- ◆ Restarts one of the threads that are waiting on the condition variable `cond`
- ◆ If no threads are waiting on `cond`, nothing happens
- ◆ If several threads are waiting on `cond`, exactly one is restarted
- ◆ Always returns 0

# Broadcasting a condition variable

---

```
int pthread_cond_broadcast(pthread_cond_t *cond)
```

- ◆ Restarts all the threads that are waiting on the condition variable `cond`.
- ◆ Nothing happens if no threads are waiting on `cond`.
- ◆ Always returns 0.

# Destroying a condition variable

---

```
int pthread_cond_destroy(pthread_cond_t *cond)
```

- ◆ Destroys a condition variable `cond`, freeing the resources it might hold.
- ◆ No threads must be waiting on the condition variable when `pthread_cond_destroy()` is called
- ◆ In LinuxThreads, the call does nothing except checking that the condition has no waiting threads
- ◆ On success, the call returns 0
- ◆ In case some threads are waiting on `cond`, `pthread_cond_destroy` returns `EBUSY`
- ◆ No need to call this function for statically initialized condition variables

# Tips for using condition variables

---

- ◆ For every condition, use a single, distinctly-associated with the condition, condition variable
- ◆ Associate/use a **single, unique mutex** with **every condition variable**
- ◆ Lock the mutex before checking the condition protected by that mutex
- ◆ Always use the same mutex when changing variables of a condition
- ◆ Keep a mutex for the shortest possible time
- ◆ Do not forget to release locks at the end with `pthread_mutex_unlock()`

Use of: pthread\_cond\_init, pthread\_cond\_wait,  
pthread\_cond\_signal, pthread\_cond\_destroy

mutex\_condvar.c

```
#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.

counter.c

```
#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>**

# Thread safety

---

- ◆ **Problem:** a thread may call library functions that are not thread-safe creating spurious outcomes
- ◆ A function is “thread-safe,” if multiple threads can simultaneously execute invocations of the same function without *side-effects* (or interference of any type!)
- ◆ POSIX specifies that all functions (including all those from the Standard C Library) except those listed on next slide are implemented in a thread-safe manner
- ◆ Directive: the calls listed on the next slide should have thread-safe implementations denoted with the postfix `_r`

# System calls not required to be thread-safe

---

<i>asctime</i>	<i>basename</i>	<i>catgets</i>	<i>crypt</i>	<i>ctime</i>
<i>dbm_clearerr</i>	<i>dbm_close</i>	<i>dbm_delete</i>	<i>dbm_error</i>	<i>dbm_fetch</i>
<i>dbm_firstkey</i>	<i>dbm_nextkey</i>	<i>dbm_open</i>	<i>dbm_store</i>	<i>dirname</i>
<i>derror</i>	<i>drand48</i>	<i>ecvt</i>	<i>encrypt</i>	<i>endgrent</i>
<i>endpwent</i>	<i>endutxent</i>	<i>fcvt</i>	<i>ftw</i>	<i>gcvt</i>
<i>getc_unlocked</i>	<i>getchar_unlocked</i>	<i>getdate</i>	<i>getenv</i>	<i>getgrent</i>
<i>getgrgid</i>	<i>getgrname</i>	<i>gethostbyaddr</i>	<i>gethostbyname</i>	<i>getlogin</i>
<i>getnetbyaddr</i>	<i>getnetbyname</i>	<i>getnetent</i>	<i>getopt</i>	<i>getprotobynam</i>
<i>getprotobynumber</i>	<i>getprotoend</i>	<i>getpwent</i>	<i>getpwnam</i>	<i>getpwuid</i>
<i>getservbyname</i>	<i>getservbyport</i>	<i>getservent</i>	<i>getutxent</i>	<i>getutxid</i>
<i>getutxline</i>	<i>gmtime</i>	<i>hcreate</i>	<i>hdestroy</i>	<i>hsearch</i>
<i>inet_ntoa</i>	<i>l64a</i>	<i>lgamma</i>	<i>lgammaf</i>	<i>lgammal</i>
<i>localeconv</i>	<i>localtime</i>	<i>lrand48</i>	<i>mrand48</i>	<i>nftw</i>
<i>nLlanginfo</i>	<i>ptsname</i>	<i>putc_unlocked</i>	<i>putchar_unlocked</i>	<i>putenv</i>
<i>pututxline</i>	<i>rand</i>	<i>readdir</i>	<i>setenv</i>	<i>setgrent</i>
<i>setkey</i>	<i>setpwent</i>	<i>setutxent</i>	<i>strerror</i>	<i>strtok</i>
<i>ttyname</i>	<i>unsetenv</i>	<i>wcstombs</i>	<i>wctomb</i>	

# Thread safety

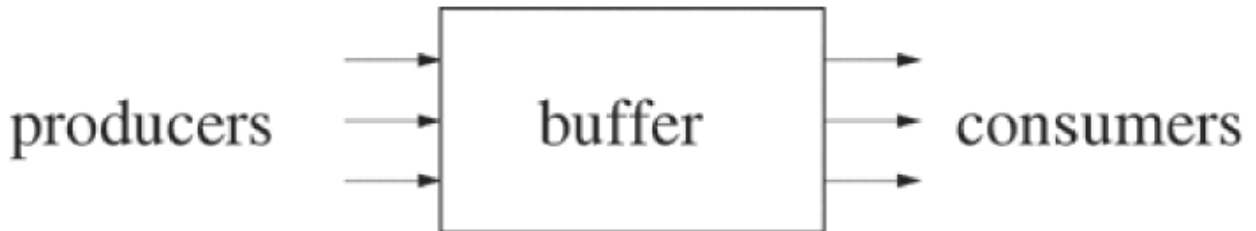
---

- ◆ An easy (“dirty”) way to safely use the above calls with threads is to invoke them in conjunction with mutexes (i.e., in mutually exclusive fashion)
- ◆ Can convert non-thread-safe functions to safe as follows:
  - 1) Lock a mutex, 2) call the function, 3) use data returned (e.g., pointer to a struct allocated somewhere), 4) unlock mutex
  - 1) Lock mutex, 2) call function 3) copy struct pointed to by returned pointer for later use 4) unlock
- ◆ Remember: `_r` at end of function name means it is re-entrant (i.e., thread-safe)



# Producer-Consumer Problem

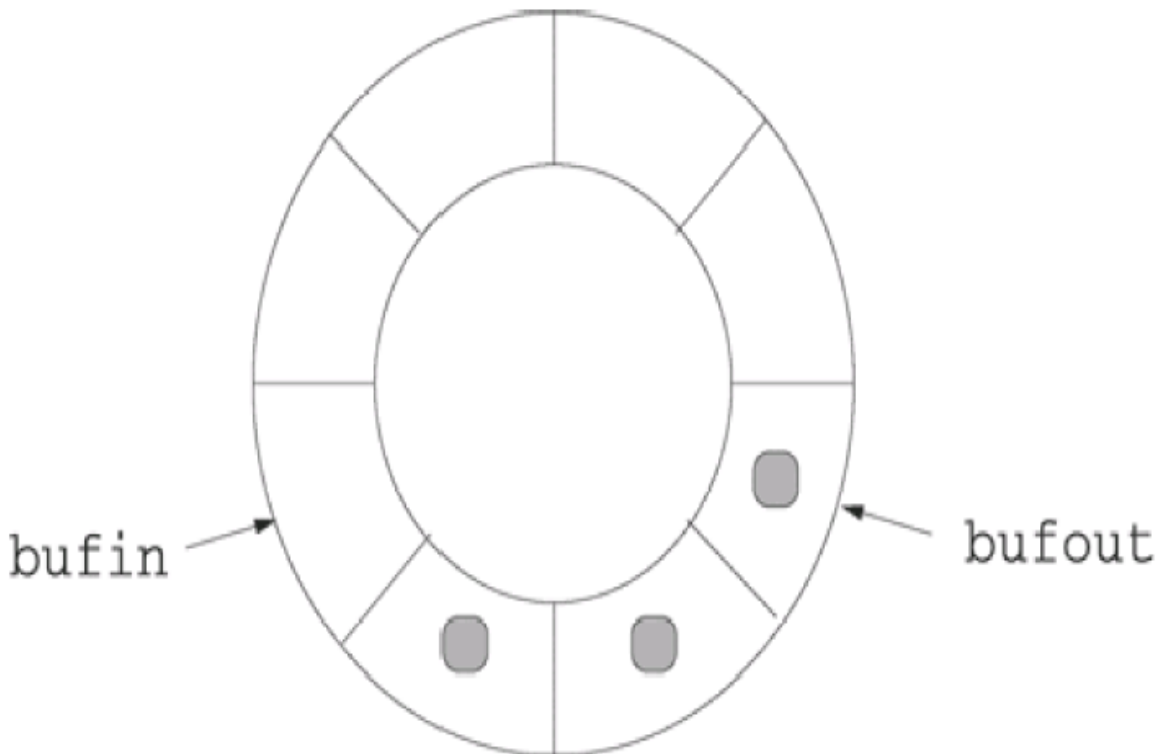
---



- ◆ Producers (P) insert data into buffer
- ◆ Consumers (C ) read data from the buffer
- ◆ **What do we want to avoid?**
  - C starts reading an object that a producer has not yet finished inserting
  - C reads an object from the buffer that does not exist
  - C reads an object that has already been removed from the buffer
  - P places an item in the buffer, when buffer is full
  - P overwrites an item in the buffer that has not yet been read by a consumer

# Example: Bounded cyclical buffer

---



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

# Solution to bounded buffer problem

```
#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 *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;
```

```
}
```

← Τι επιστρέφει αν δεν υπάρχουν δεδομένα?

---

```
int putitem(buffer_t item) { /* insert item into 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[buflen] = item;
        buflen = (buflen + 1) % BUFSIZE;
        totalitems++;
    } else
        erroritem = EAGAIN;
    if (error = pthread_mutex_unlock(&bufferlock))
        return error; /* unlock error more serious than no slot*/
    return erroritem;
}
```

# 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);
}
```

Problems??

# 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)

Solution:

**Use condition variables**

# Another producer-consumer example

// from [www.mario-konrad.ch](http://www.mario-konrad.ch), changed slightly

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

prod-cons.c

```
#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);  
}
```

```
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);  
}
```

---

```
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;
}
```

mema@browser> ./prod-cons

>> 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>

mema@browser> ./prod-cons

>> 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

**Outcome - usleep(0)**

---

>> 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>

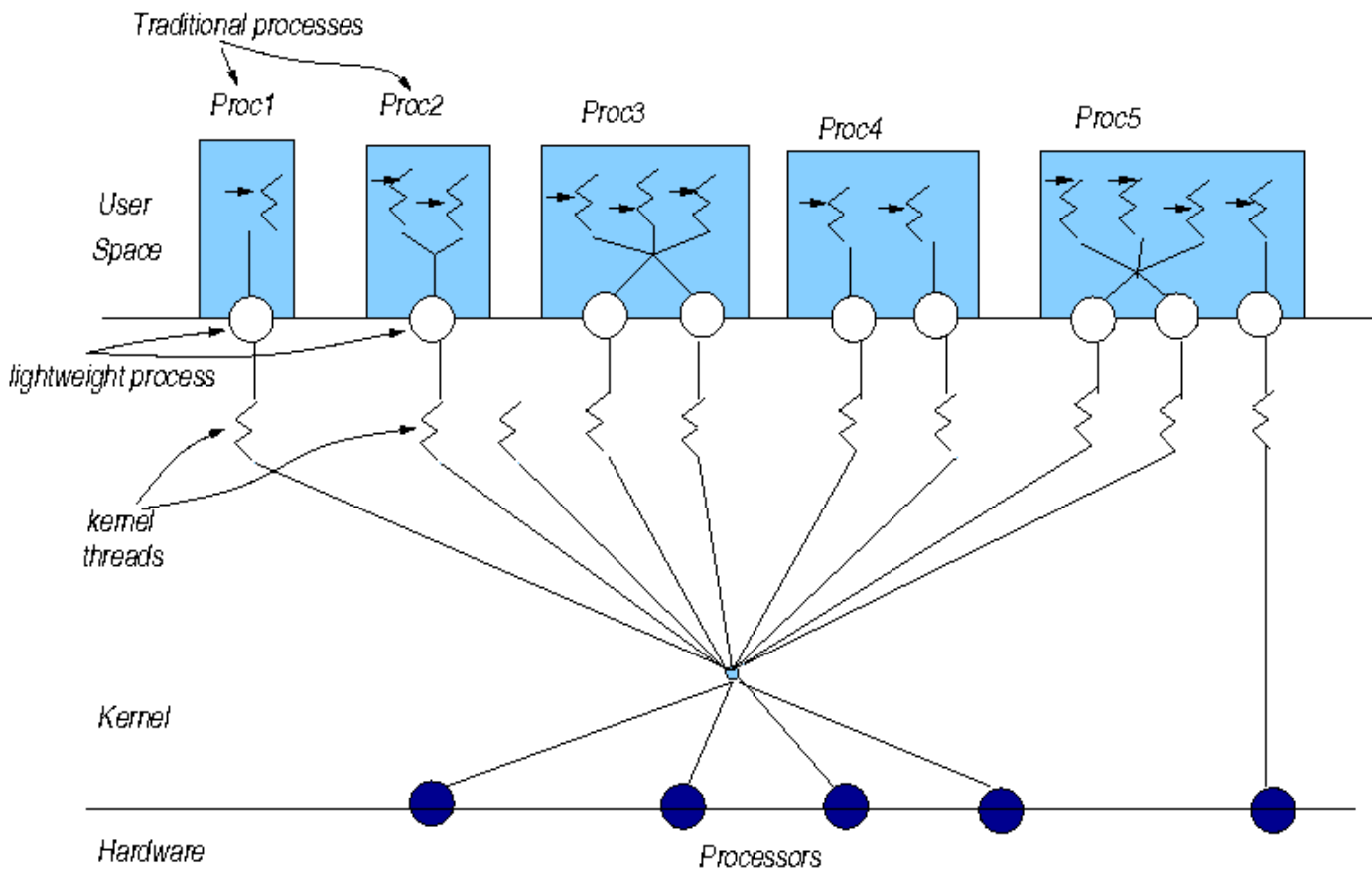
**Outcome - usleep(0)**  
**(cont'd)**

# Food for Thought

---

- Anything wrong with this example? Hint: what data are protected by mutexes?
- With one producer/consumer, can we use `if` instead while in the condition check?
- With multiple producers/consumers where multiple items can be added/removed at a time, does code need to be changed?

# Νήματα - Μοντέλο



Two-level multi-threaded Model