Practical Programming

POSIX Thread Programming

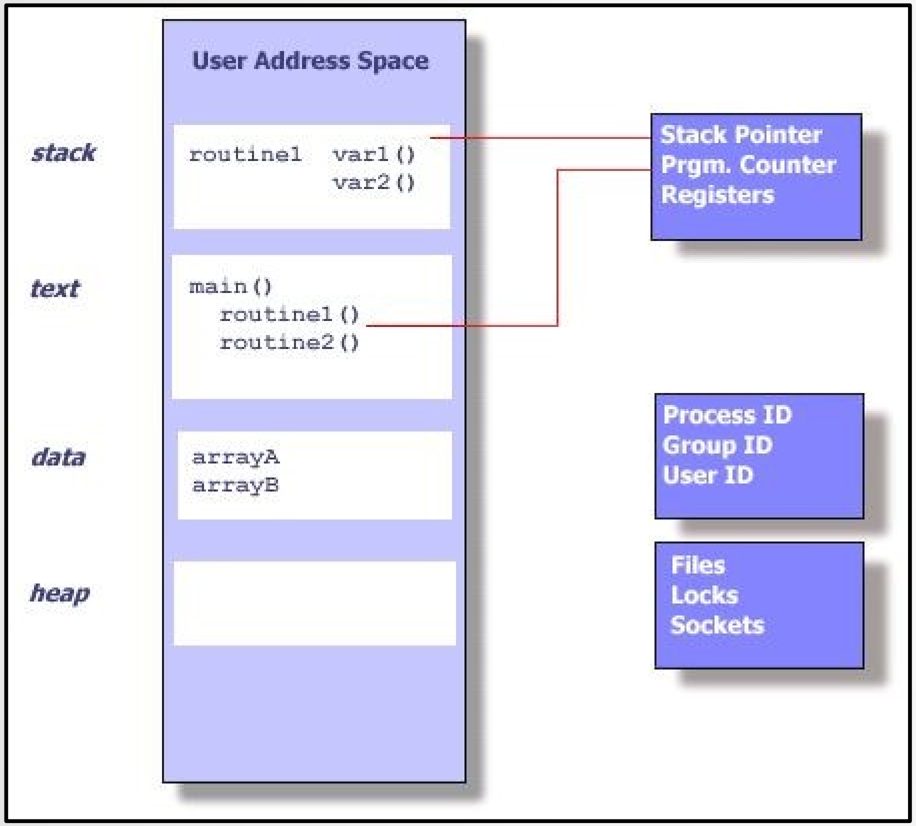
David Bouchet

david.bouchet.epita@gmail.com

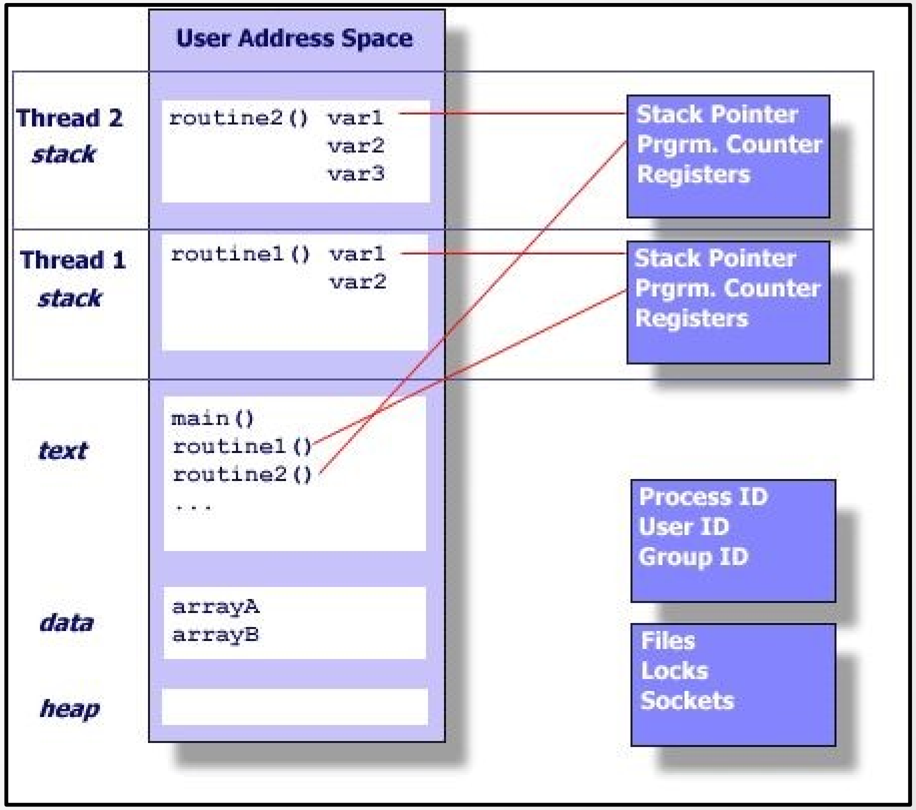
Threads?

- A process can have more than one execution flow at a time.
- Threads are light-weight processes sharing the same address space.
- Each thread has its own stack along with its own execution flow.

Threads?



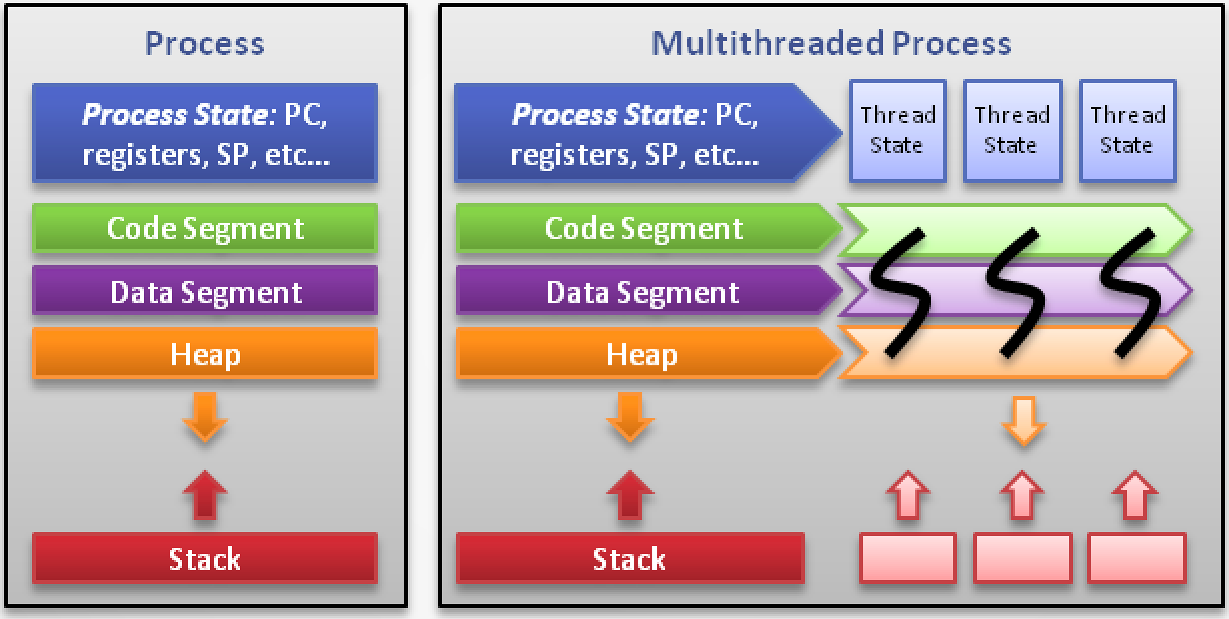
Mono-Threaded



Multi-Threaded

Source: https://computing.llnl.gov/tutorials/pthreads/

Threads?



Threads contain only necessary information, such as a stack (for local variables, function arguments, return values), a copy of the registers, program counter and any thread-specific data to allow them to be scheduled individually. Other data is shared within the process between all threads.

© Alfred Park, http://randu.org/tutorials/threads

Source: https://randu.org/tutorials/threads/

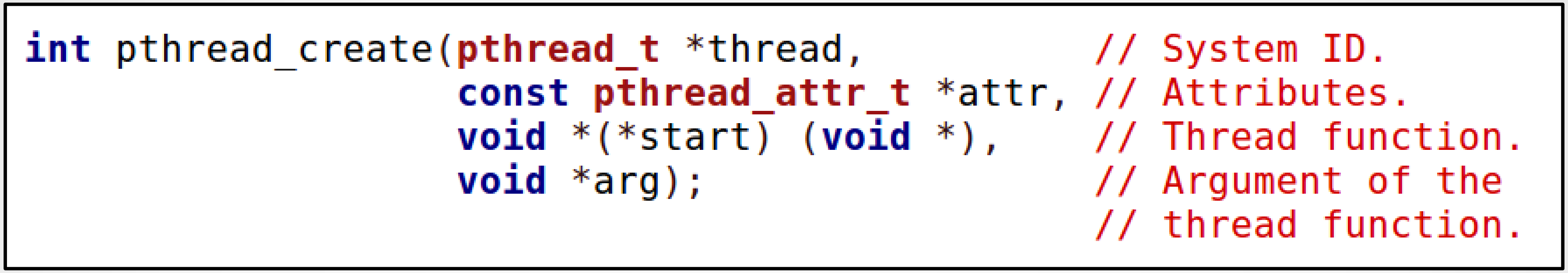
Multi-Threading Libraries

- **POSIX Threads (pthread)**
- SDL, BOOST
- C11 API (need better support)
- Higher-level parallelism: Intel’s TBB, OpenMP

Compiling POSIX Threads

- Require compiler option `-pthread`
- Require header “`pthread.h`”

Creating Threads

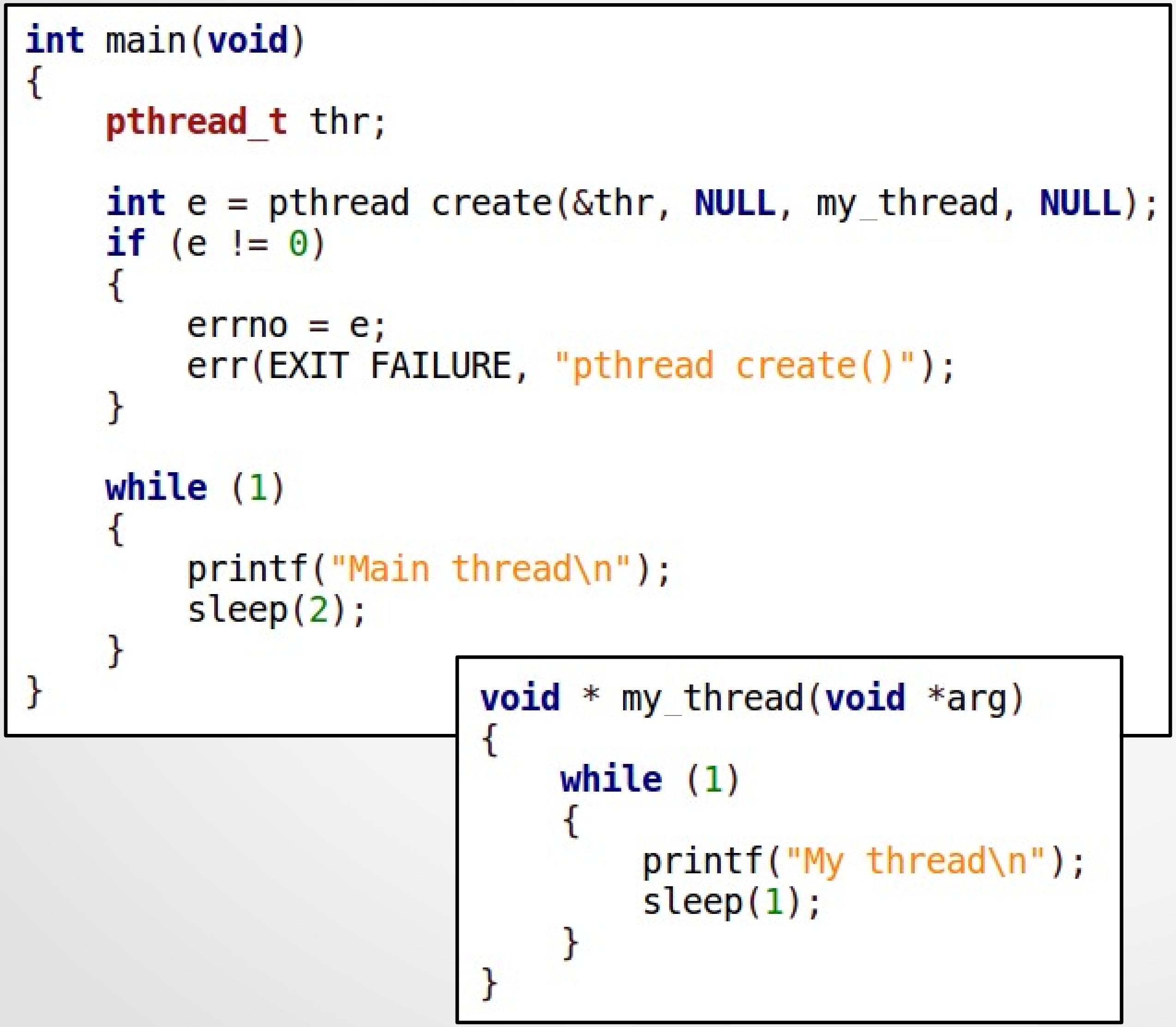


```
int pthread_create(pthread_t *thread,             // System ID.
                   const pthread_attr_t *attr, // Attributes.
                   void *(*start) (void *),    // Thread function.
                   void *arg);                 // Argument of the
                                               // thread function.
```

On success, *pthread_create()* returns 0;
On error, it returns an error number, and the
contents of **thread* are undefined.

pthread_create(3)

Creating Threads – Example

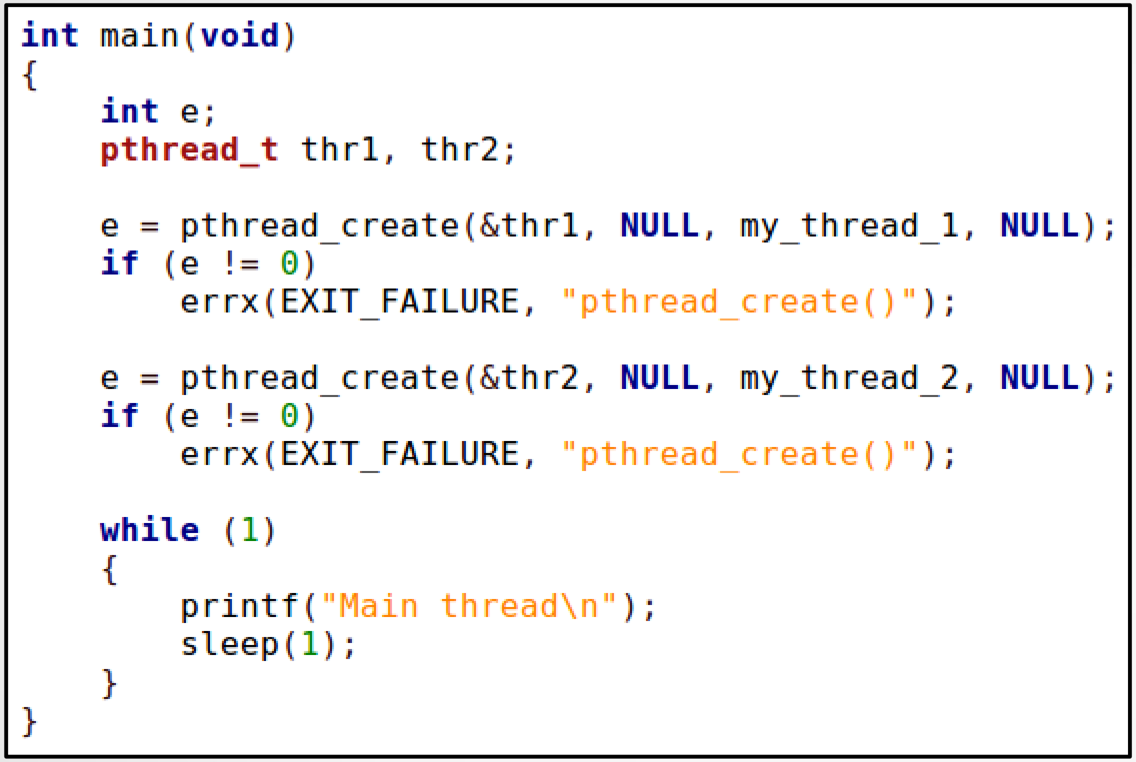


```c
int main(void)
{
    pthread_t thr;

    int e = pthread_create(&thr, NULL, my_thread, NULL);
    if (e != 0)
    {
        errno = e;
        err(EXIT_FAILURE, "pthread_create()");
    }

    while (1)
    {
        printf("Main thread\n");
        sleep(2);
    }
}
```

```c
void * my_thread(void *arg)
{
    while (1)
    {
        printf("My thread\n");
        sleep(1);
    }
}
```

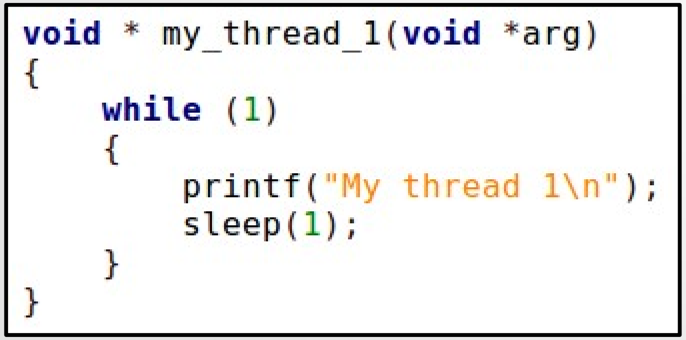
```
Main thread
My thread
My thread
Main thread
My thread
My thread
Main thread
My thread
My thread
Main thread
My thread
My thread
Main thread
My thread
My thread
Main thread
My thread
My thread
Main thread
My thread
My thread
^C
```

Creating Threads – Example

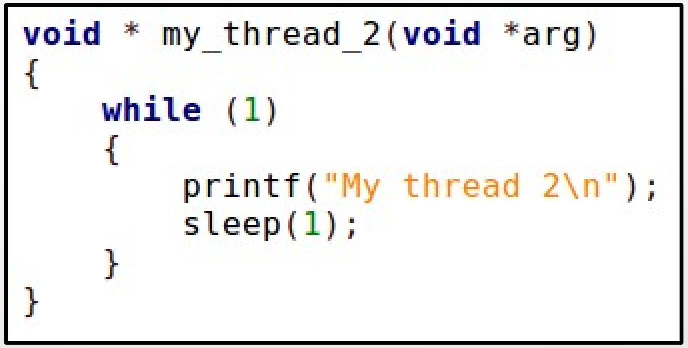


```c
int main(void)
{
    int e;
    pthread_t thr1, thr2;

    e = pthread_create(&thr1, NULL, my_thread_1, NULL);
    if (e != 0)
        errx(EXIT_FAILURE, "pthread_create()");

    e = pthread_create(&thr2, NULL, my_thread_2, NULL);
    if (e != 0)
        errx(EXIT_FAILURE, "pthread_create()");

    while (1)
    {
        printf("Main thread\n");
        sleep(1);
    }
}
```

The execution order cannot be predicted.



```c
void * my_thread_1(void *arg)
{
    while (1)
    {
        printf("My thread 1\n");
        sleep(1);
    }
}
```

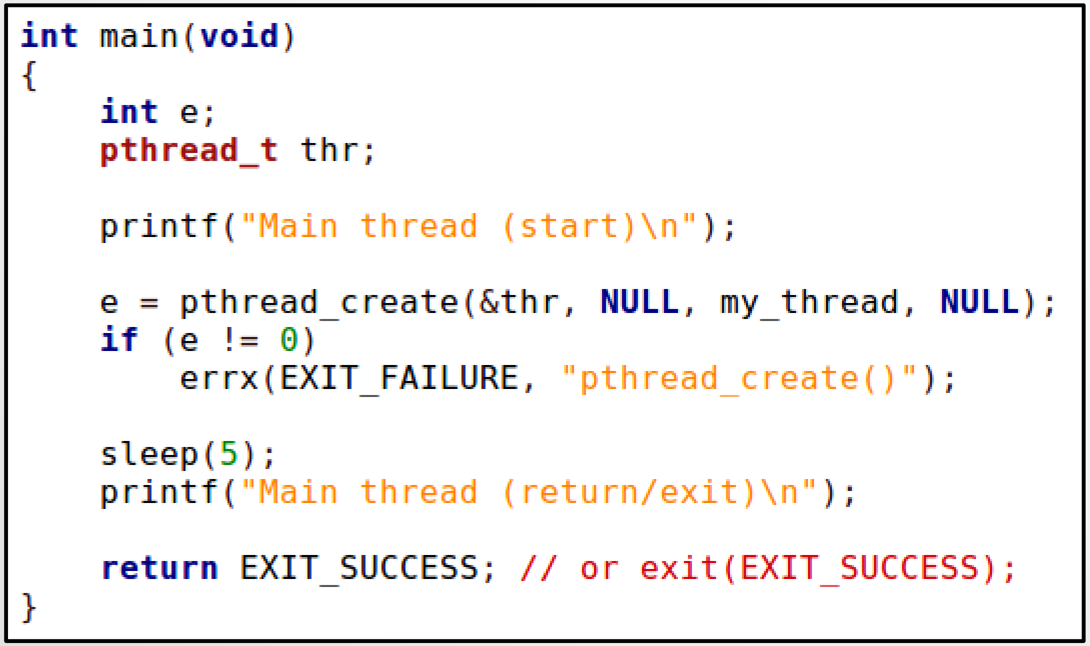


```c
void * my_thread_2(void *arg)
{
    while (1)
    {
        printf("My thread 2\n");
        sleep(1);
    }
}
```

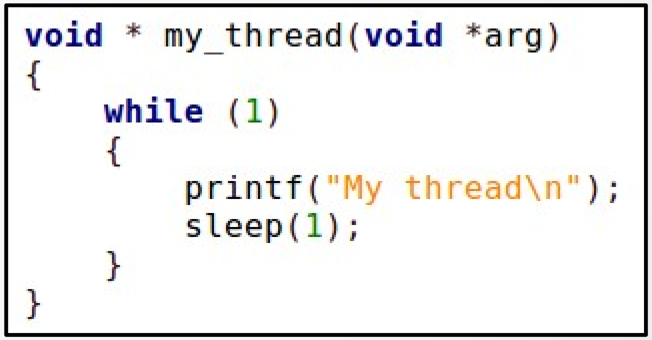
```
Main thread
My thread 1
My thread 2
Main thread
My thread 2
My thread 1
My thread 1
My thread 2
Main thread
My thread 1
My thread 2
Main thread
My thread 1
My thread 2
Main thread
My thread 1
Main thread
My thread 2
^C
```

Terminating Processes and Threads

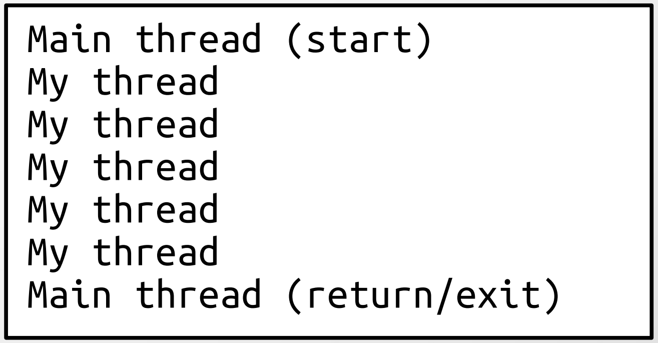
When the main thread ends (by using return or the *exit()* function), it causes the termination of the process and all of its threads.



```c
int main(void)
{
    int e;
    pthread_t thr;

    printf("Main thread (start)\n");

    e = pthread_create(&thr, NULL, my_thread, NULL);
    if (e != 0)
        errx(EXIT_FAILURE, "pthread_create()");

    sleep(5);
    printf("Main thread (return/exit)\n");

    return EXIT_SUCCESS; // or exit(EXIT_SUCCESS);
}
```

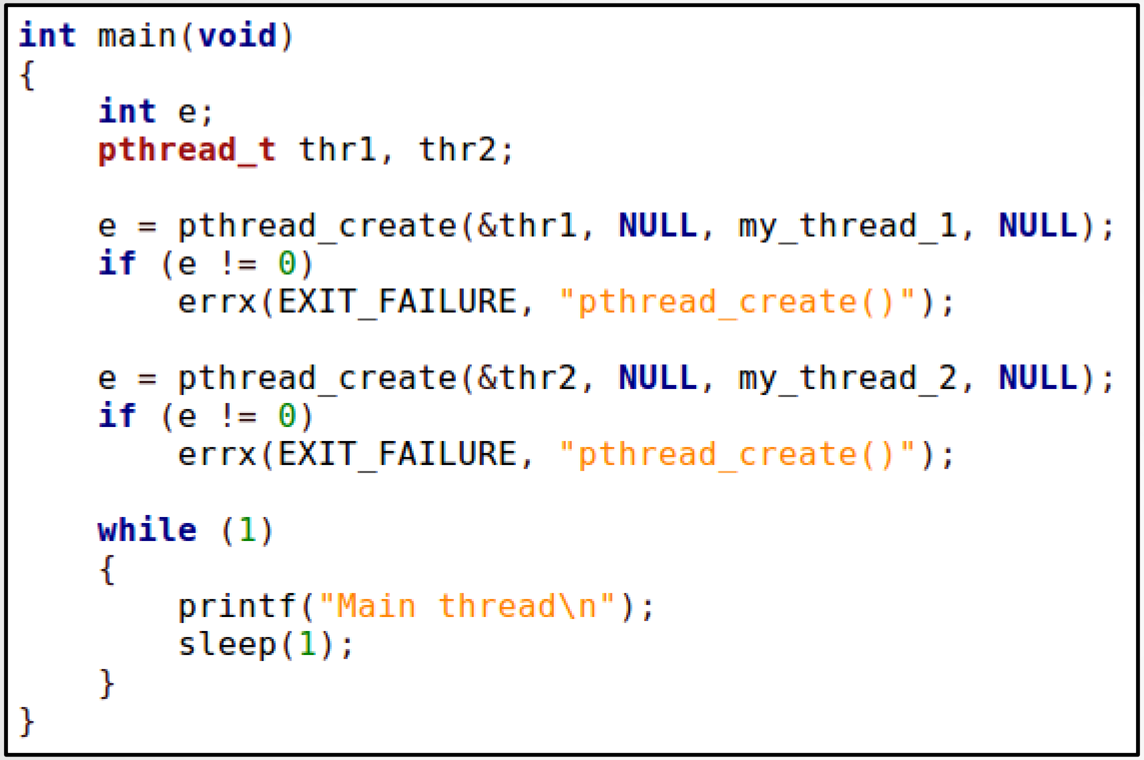


```c
void * my_thread(void *arg)
{
    while (1)
    {
        printf("My thread\n");
        sleep(1);
    }
}
```



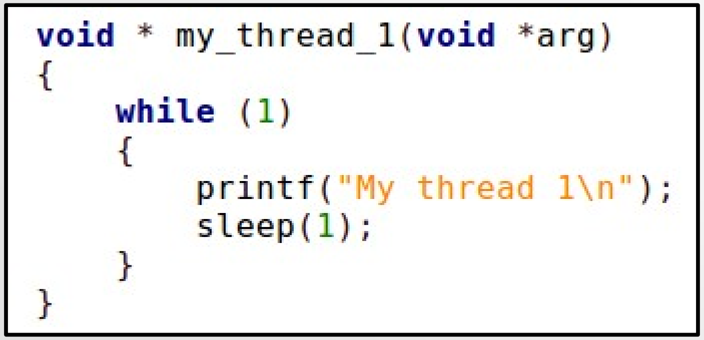
```
Main thread (start)
My thread
My thread
My thread
My thread
My thread
Main thread (return/exit)
```

Terminating Processes and Threads

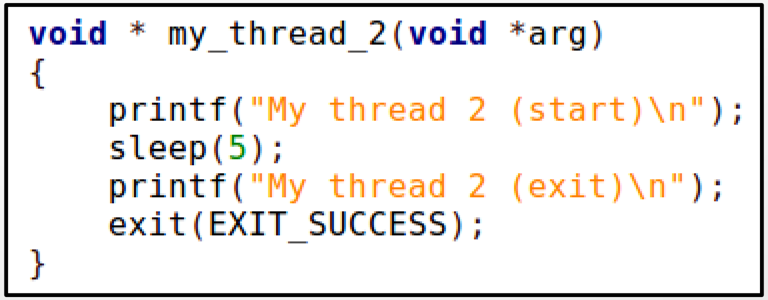


```c
int main(void)
{
    int e;
    pthread_t thr1, thr2;

    e = pthread_create(&thr1, NULL, my_thread_1, NULL);
    if (e != 0)
        errx(EXIT_FAILURE, "pthread_create()");

    e = pthread_create(&thr2, NULL, my_thread_2, NULL);
    if (e != 0)
        errx(EXIT_FAILURE, "pthread_create()");

    while (1)
    {
        printf("Main thread\n");
        sleep(1);
    }
}
```

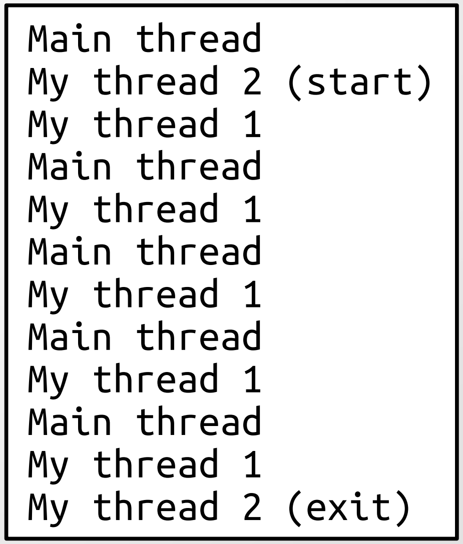
When any thread of a process is ended by the *exit()* function, it causes the termination of the process and all of its threads (main thread included).



```c
void * my_thread_1(void *arg)
{
    while (1)
    {
        printf("My thread 1\n");
        sleep(1);
    }
}
```



```c
void * my_thread_2(void *arg)
{
    printf("My thread 2 (start)\n");
    sleep(5);
    printf("My thread 2 (exit)\n");
    exit(EXIT_SUCCESS);
}
```



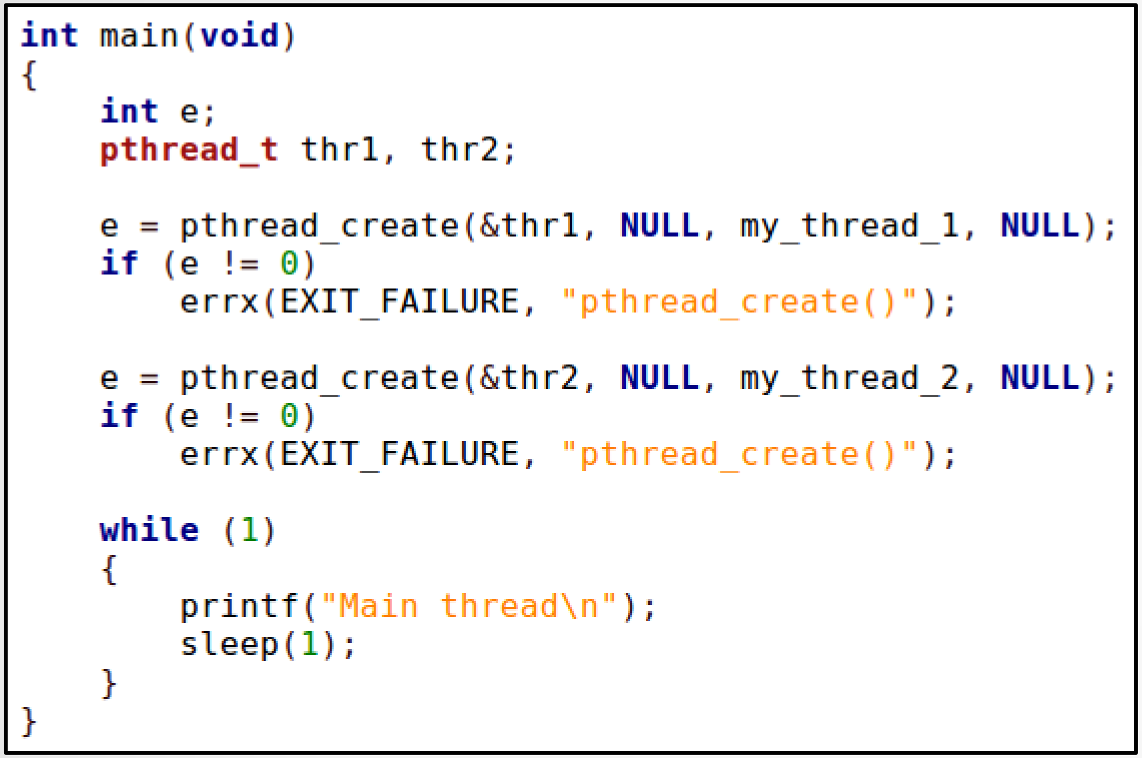
```
Main thread
My thread 2 (start)
My thread 1
Main thread
My thread 1
Main thread
My thread 1
Main thread
My thread 1
Main thread
My thread 1
My thread 2 (exit)
```

Terminating Threads Only

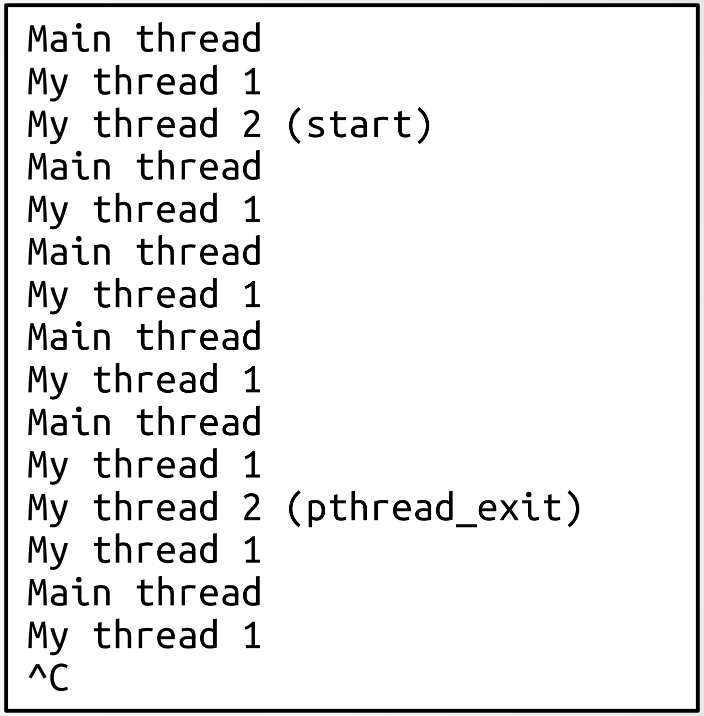
To terminate a thread (and not the process).

- You can use the *return* instruction (not in the main thread, or it causes the termination of the process).
- You can use the *pthread_exit()* function (even in the main thread).

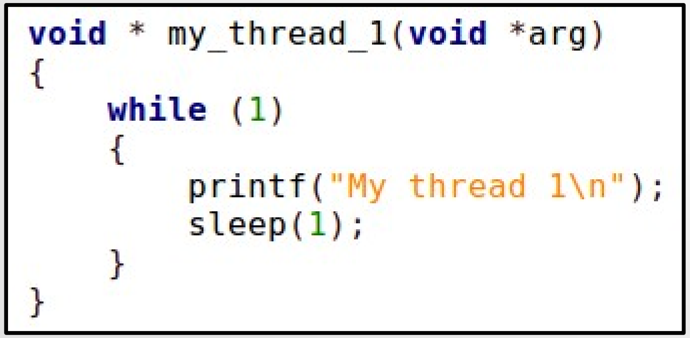
Terminating Threads Only



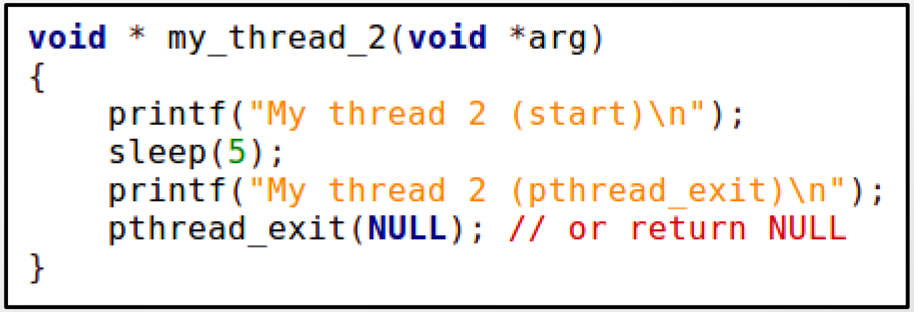
```c
int main(void)
{
    int e;
    pthread_t thr1, thr2;

    e = pthread_create(&thr1, NULL, my_thread_1, NULL);
    if (e != 0)
        errx(EXIT_FAILURE, "pthread_create()");

    e = pthread_create(&thr2, NULL, my_thread_2, NULL);
    if (e != 0)
        errx(EXIT_FAILURE, "pthread_create()");

    while (1)
    {
        printf("Main thread\n");
        sleep(1);
    }
}
```



```
Main thread
My thread 1
My thread 2 (start)
Main thread
My thread 1
Main thread
My thread 1
Main thread
My thread 1
Main thread
My thread 1
My thread 2 (pthread_exit)
My thread 1
Main thread
My thread 1
^C
```

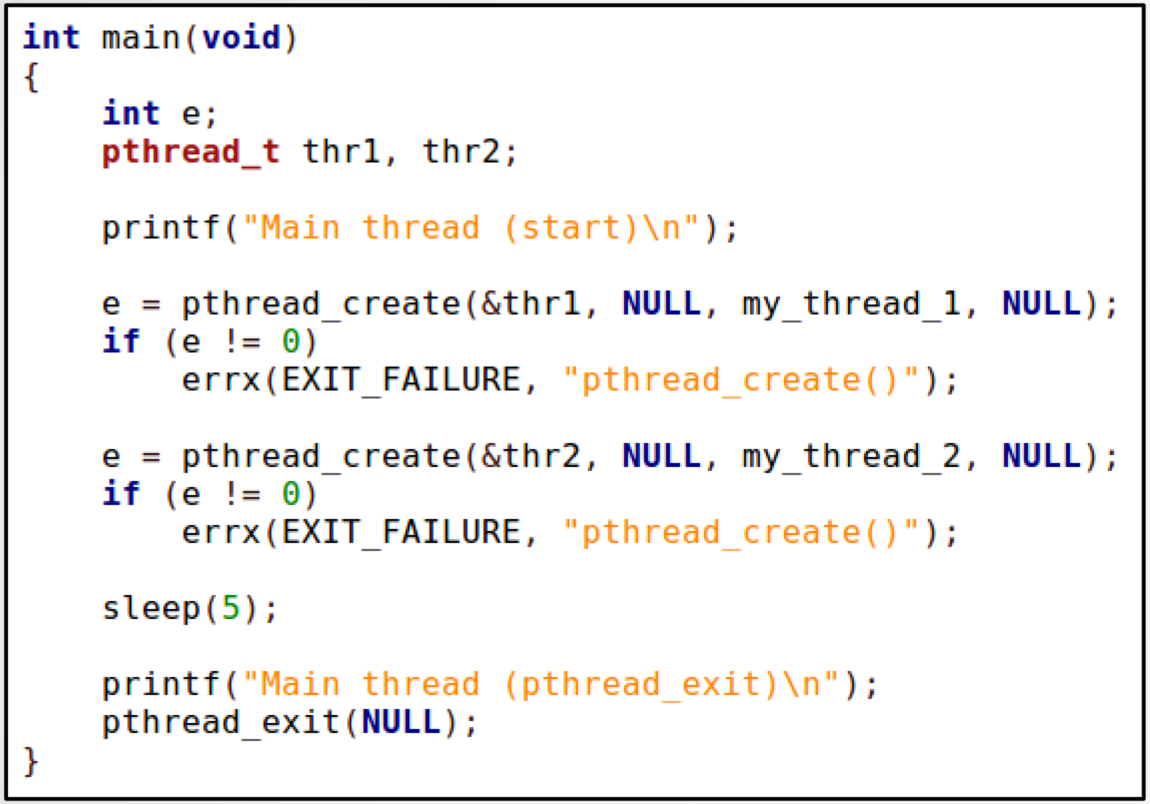


```c
void * my_thread_1(void *arg)
{
    while (1)
    {
        printf("My thread 1\n");
        sleep(1);
    }
}
```

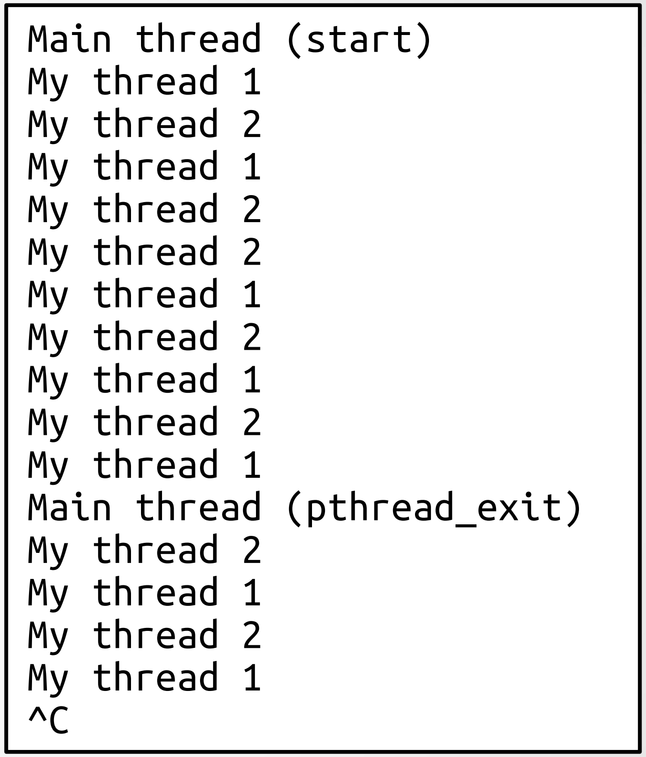


```c
void * my_thread_2(void *arg)
{
    printf("My thread 2 (start)\n");
    sleep(5);
    printf("My thread 2 (pthread_exit)\n");
    pthread_exit(NULL); // or return NULL
}
```

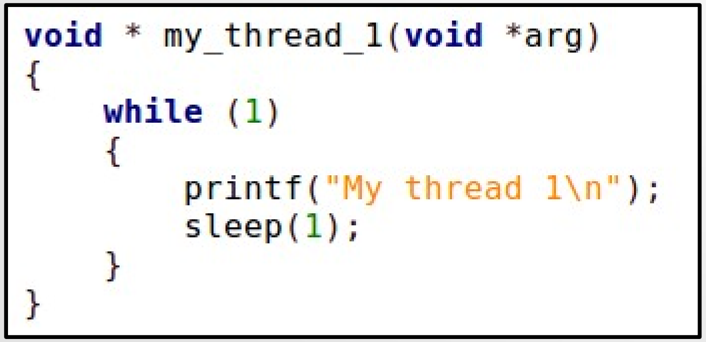
Terminating Threads Only



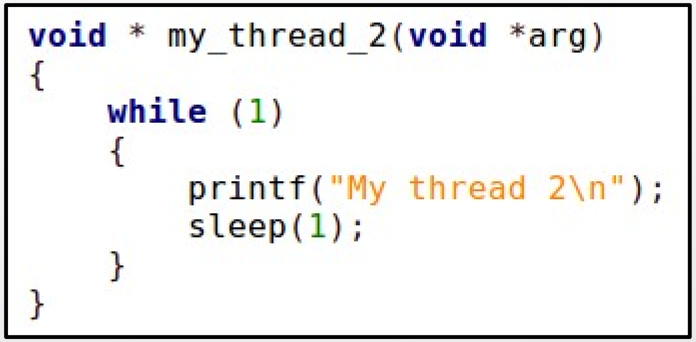
```c
int main(void)
{
    int e;
    pthread_t thr1, thr2;

    printf("Main thread (start)\n");

    e = pthread_create(&thr1, NULL, my_thread_1, NULL);
    if (e != 0)
        errx(EXIT_FAILURE, "pthread_create()");

    e = pthread_create(&thr2, NULL, my_thread_2, NULL);
    if (e != 0)
        errx(EXIT_FAILURE, "pthread_create()");

    sleep(5);

    printf("Main thread (pthread_exit)\n");
    pthread_exit(NULL);
}
```



```
Main thread (start)
My thread 1
My thread 2
My thread 1
My thread 2
My thread 2
My thread 1
My thread 2
My thread 1
My thread 2
My thread 1
Main thread (pthread_exit)
My thread 2
My thread 1
My thread 2
My thread 1
^C
```

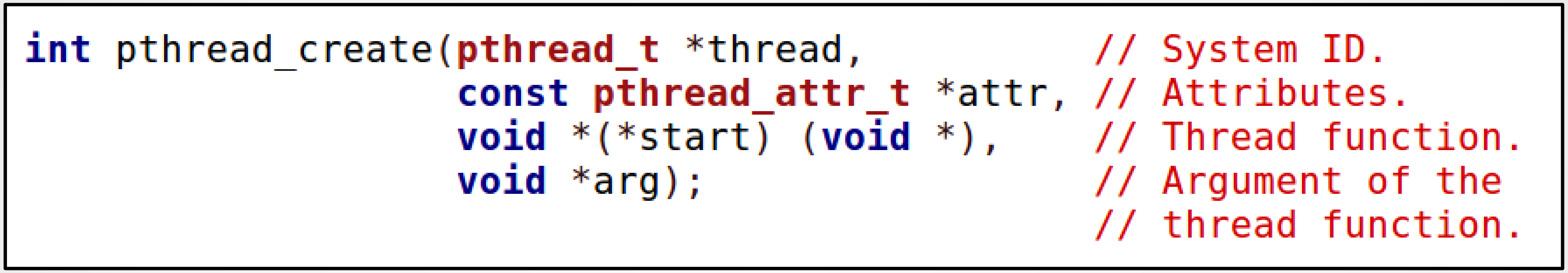


```c
void * my_thread_1(void *arg)
{
    while (1)
    {
        printf("My thread 1\n");
        sleep(1);
    }
}
```



```c
void * my_thread_2(void *arg)
{
    while (1)
    {
        printf("My thread 2\n");
        sleep(1);
    }
}
```

Passing Arguments to Threads

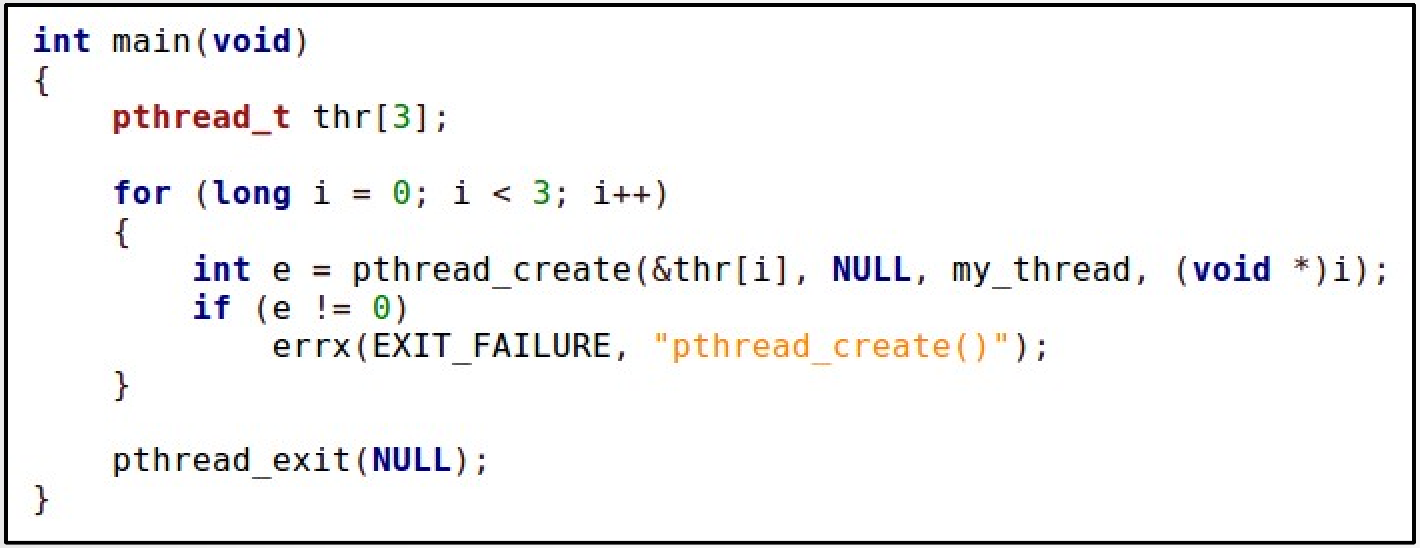


```
int pthread_create(pthread_t *thread,            // System ID.
                   const pthread_attr_t *attr, // Attributes.
                   void *(*start) (void *),    // Thread function.
                   void *arg);                 // Argument of the
                                               // thread function.
```

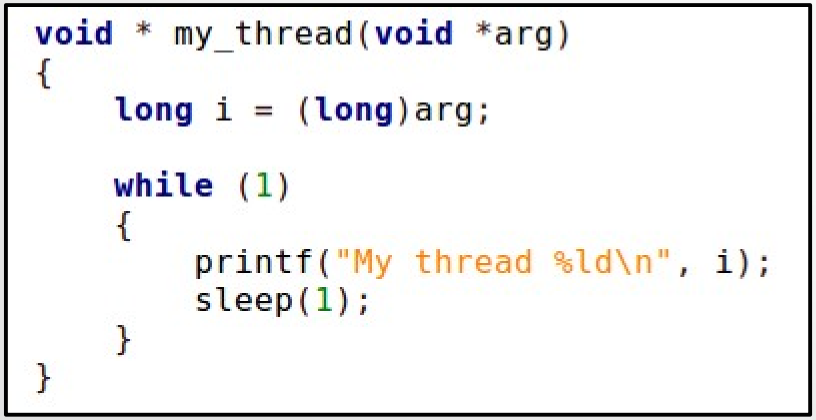
The **fourth argument** of the ***pthread_create()*** function is passed to the thread function.

Passing Arguments to Threads

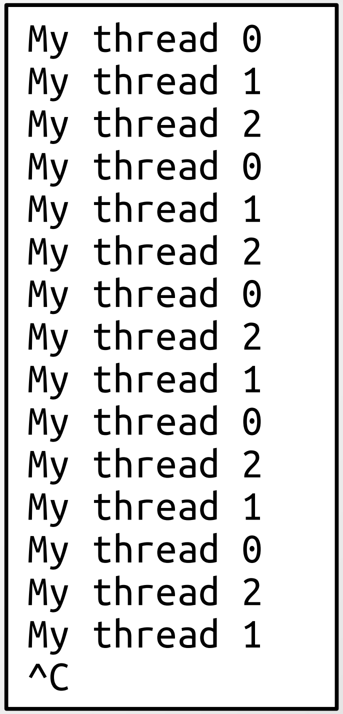
Example with the *long* type:



```c
int main(void)
{
    pthread_t thr[3];

    for (long i = 0; i < 3; i++)
    {
        int e = pthread_create(&thr[i], NULL, my_thread, (void *)i);
        if (e != 0)
            errx(EXIT_FAILURE, "pthread_create()");
    }

    pthread_exit(NULL);
}
```



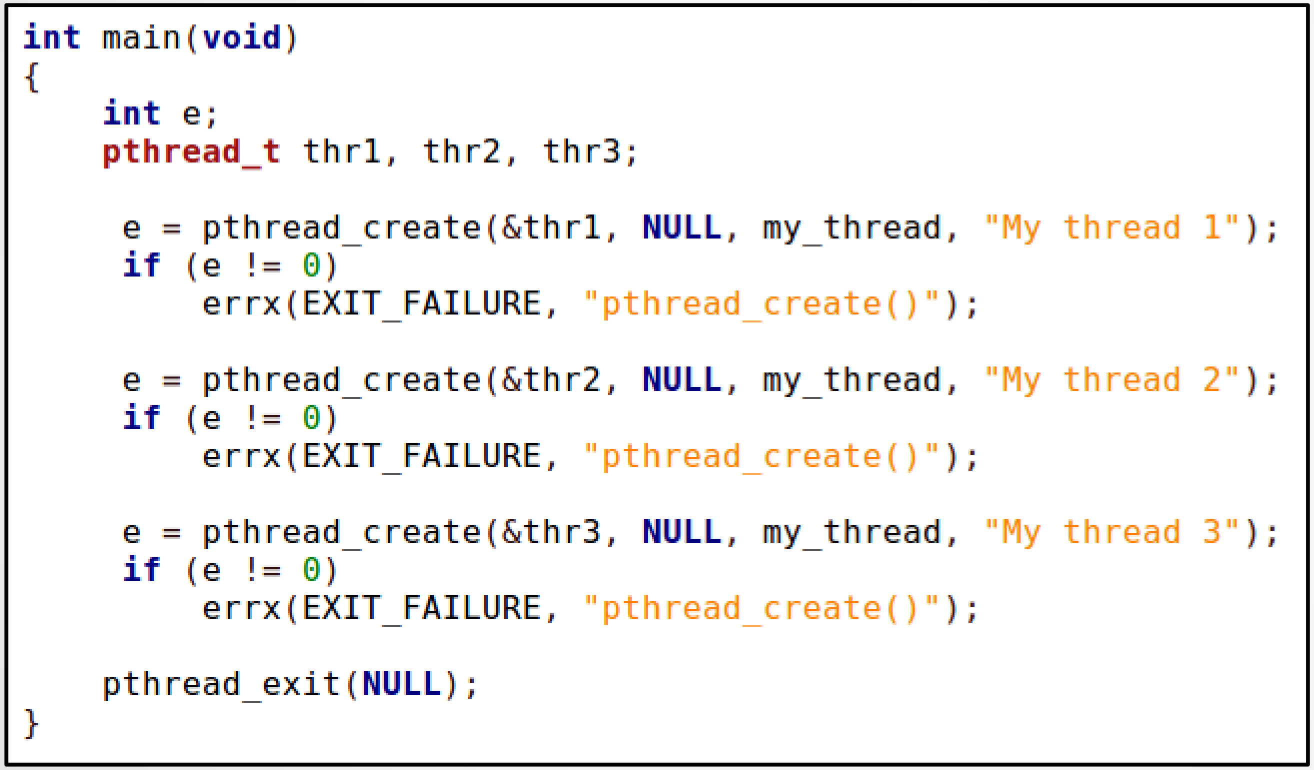
```c
void * my_thread(void *arg)
{
    long i = (long)arg;

    while (1)
    {
        printf("My thread %ld\n", i);
        sleep(1);
    }
}
```



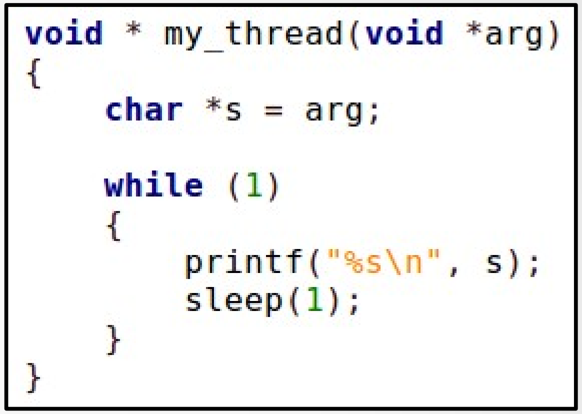
```
My thread 0
My thread 1
My thread 2
My thread 0
My thread 1
My thread 2
My thread 0
My thread 2
My thread 1
My thread 0
My thread 2
My thread 1
My thread 0
My thread 2
My thread 1
^C
```

Passing Arguments to Threads

Example with strings:



```
int main(void)
{
    int e;
    pthread_t thr1, thr2, thr3;

     e = pthread_create(&thr1, NULL, my_thread, "My thread 1");
     if (e != 0)
         errx(EXIT_FAILURE, "pthread_create()");

     e = pthread_create(&thr2, NULL, my_thread, "My thread 2");
     if (e != 0)
         errx(EXIT_FAILURE, "pthread_create()");

     e = pthread_create(&thr3, NULL, my_thread, "My thread 3");
     if (e != 0)
         errx(EXIT_FAILURE, "pthread_create()");

    pthread_exit(NULL);
}
```

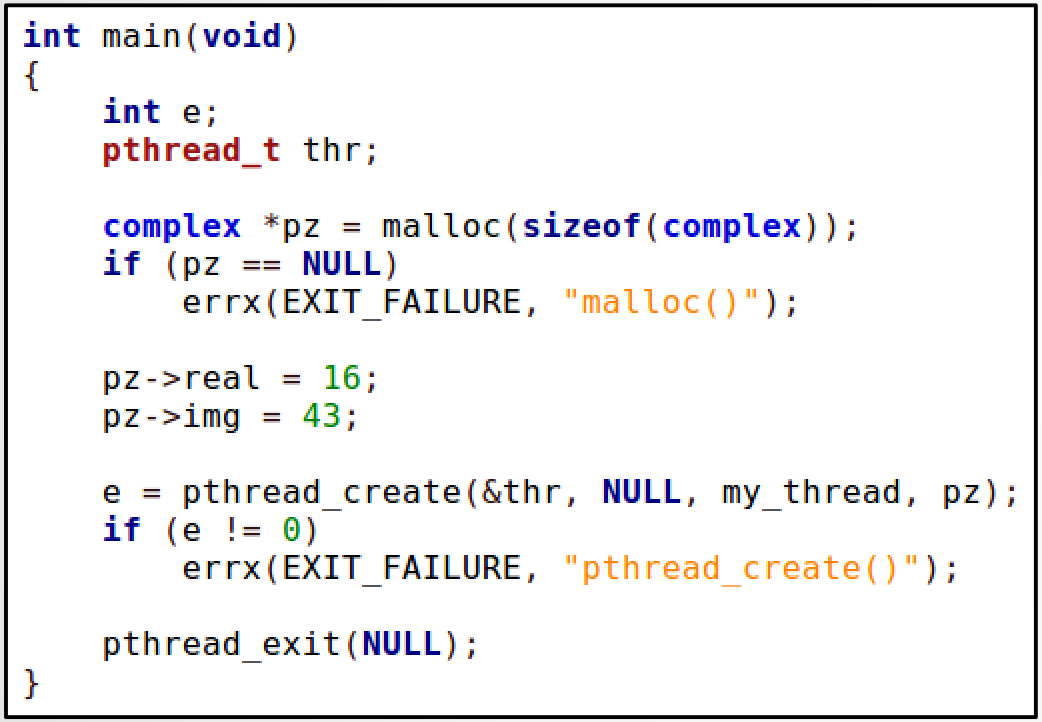


```
void * my_thread(void *arg)
{
    char *s = arg;

    while (1)
    {
        printf("%s\n", s);
        sleep(1);
    }
}
```

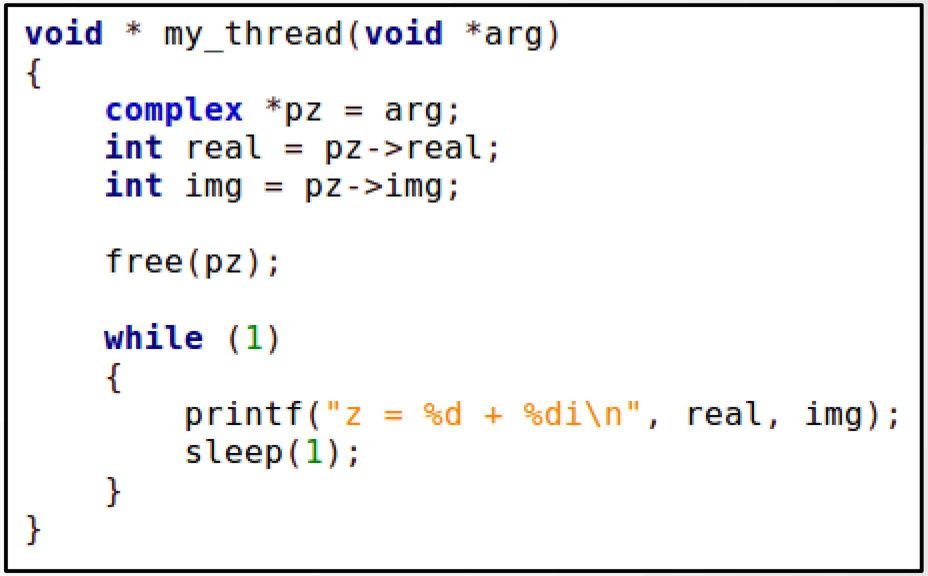
```
My thread 1
My thread 2
My thread 3
My thread 1
My thread 3
My thread 2
My thread 1
My thread 2
My thread 3
My thread 1
My thread 2
My thread 3
^C
```

Passing Arguments to Threads

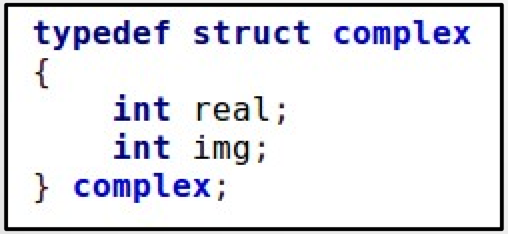
Example with *struct* and dynamic allocation:



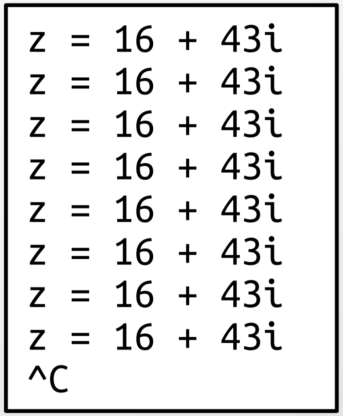
```c
int main(void)
{
    int e;
    pthread_t thr;

    complex *pz = malloc(sizeof(complex));
    if (pz == NULL)
        errx(EXIT_FAILURE, "malloc()");

    pz->real = 16;
    pz->img = 43;

    e = pthread_create(&thr, NULL, my_thread, pz);
    if (e != 0)
        errx(EXIT_FAILURE, "pthread_create()");

    pthread_exit(NULL);
}
```



```c
void * my_thread(void *arg)
{
    complex *pz = arg;
    int real = pz->real;
    int img = pz->img;

    free(pz);

    while (1)
    {
        printf("z = %d + %di\n", real, img);
        sleep(1);
    }
}
```



```c
typedef struct complex
{
    int real;
    int img;
} complex;
```



```
z = 16 + 43i
z = 16 + 43i
z = 16 + 43i
z = 16 + 43i
z = 16 + 43i
z = 16 + 43i
z = 16 + 43i
z = 16 + 43i
^C
```

Joining Threads

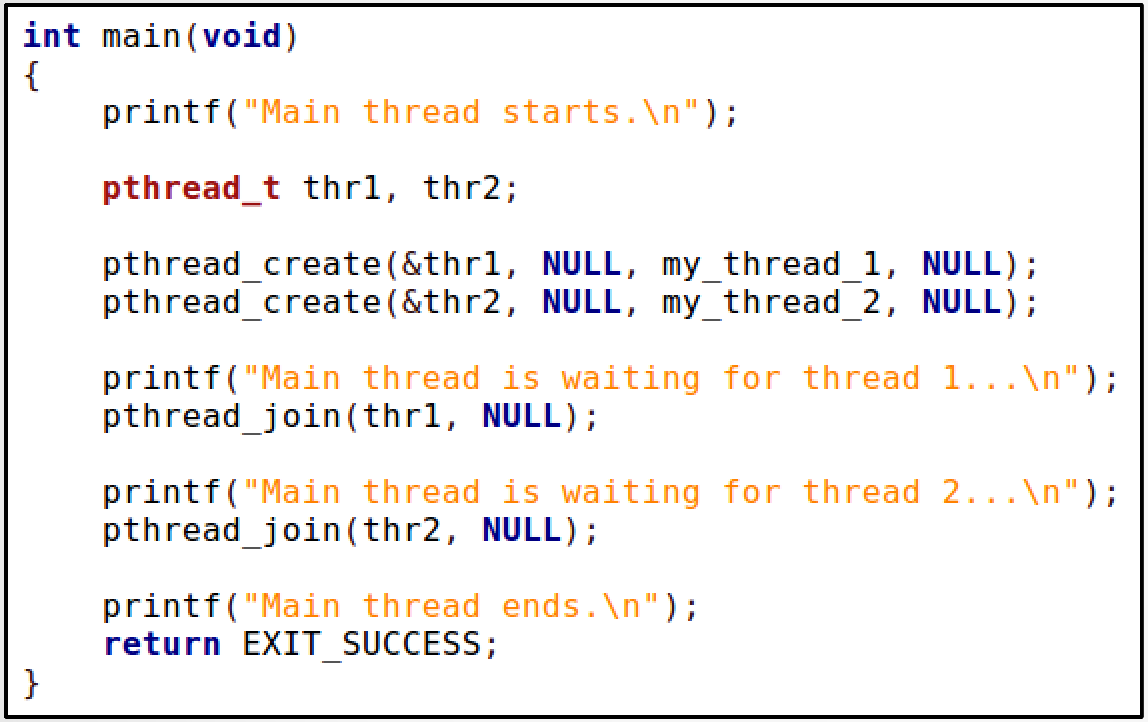
The following function can be used to wait for a specific thread and/or to return a value from this thread:

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

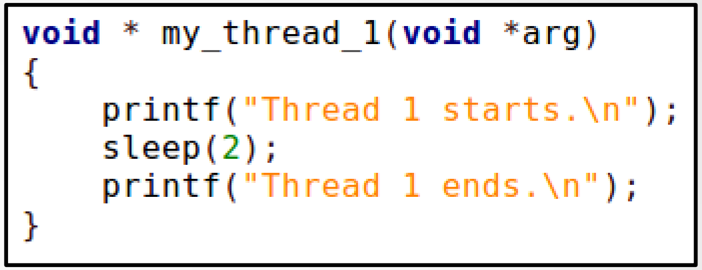
Any thread can join with any other thread in the process.

→ pthread_join(3)

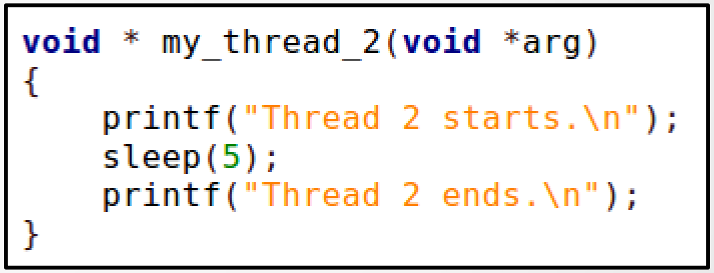
Waiting for Threads



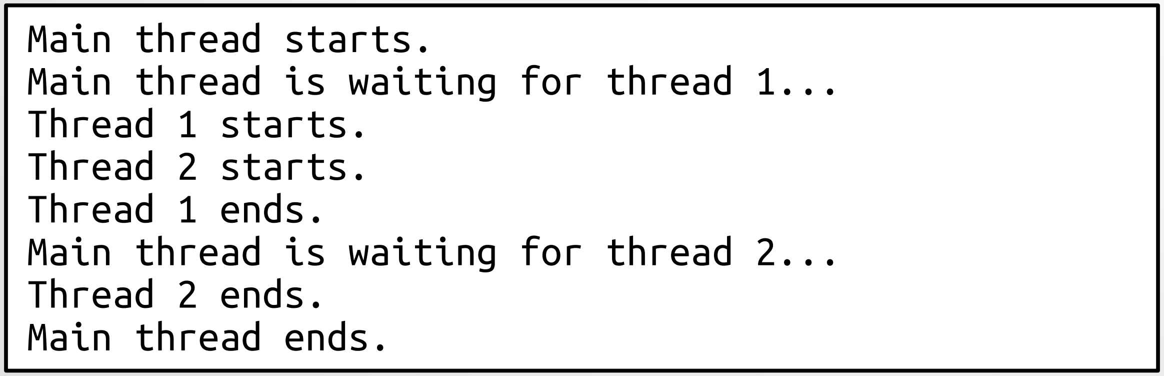
```
int main(void)
{
    printf("Main thread starts.\n");

    pthread_t thr1, thr2;

    pthread_create(&thr1, NULL, my_thread_1, NULL);
    pthread_create(&thr2, NULL, my_thread_2, NULL);

    printf("Main thread is waiting for thread 1...\n");
    pthread_join(thr1, NULL);

    printf("Main thread is waiting for thread 2...\n");
    pthread_join(thr2, NULL);

    printf("Main thread ends.\n");
    return EXIT_SUCCESS;
}
```



```
void * my_thread_1(void *arg)
{
    printf("Thread 1 starts.\n");
    sleep(2);
    printf("Thread 1 ends.\n");
}
```



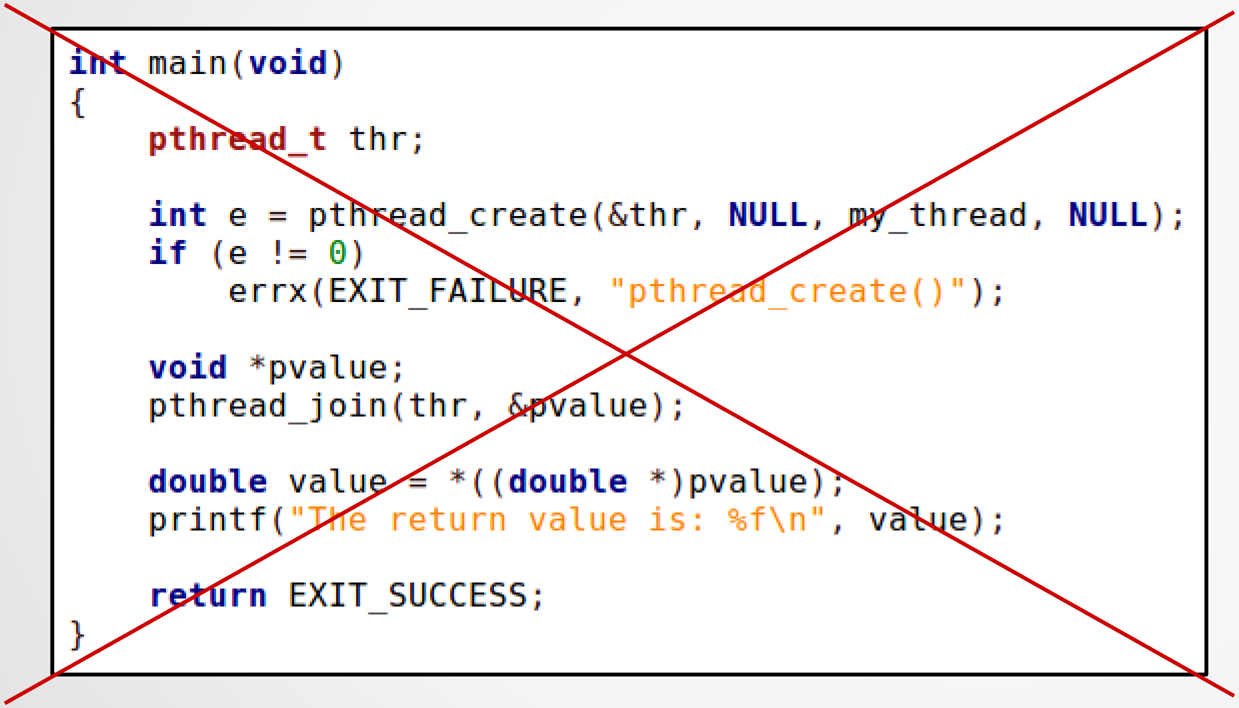
```
void * my_thread_2(void *arg)
{
    printf("Thread 2 starts.\n");
    sleep(5);
    printf("Thread 2 ends.\n");
}
```



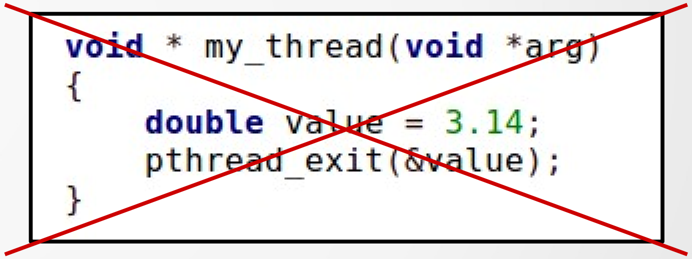
```
Main thread starts.
Main thread is waiting for thread 1...
Thread 1 starts.
Thread 2 starts.
Thread 1 ends.
Main thread is waiting for thread 2...
Thread 2 ends.
Main thread ends.
```

Returning Values from Threads

The wrong way!



```c
int main(void)
{
    pthread_t thr;

    int e = pthread_create(&thr, NULL, my_thread, NULL);
    if (e != 0)
        errx(EXIT_FAILURE, "pthread_create()");

    void *pvalue;
    pthread_join(thr, &pvalue);

    double value = *((double *)pvalue);
    printf("The return value is: %f\n", value);

    return EXIT_SUCCESS;
}
```



```c
void * my_thread(void *arg)
{
    double value = 3.14;
    pthread_exit(&value);
}
```

```
The return value is: 0.000000
```

Returning Values from Threads

The right way!

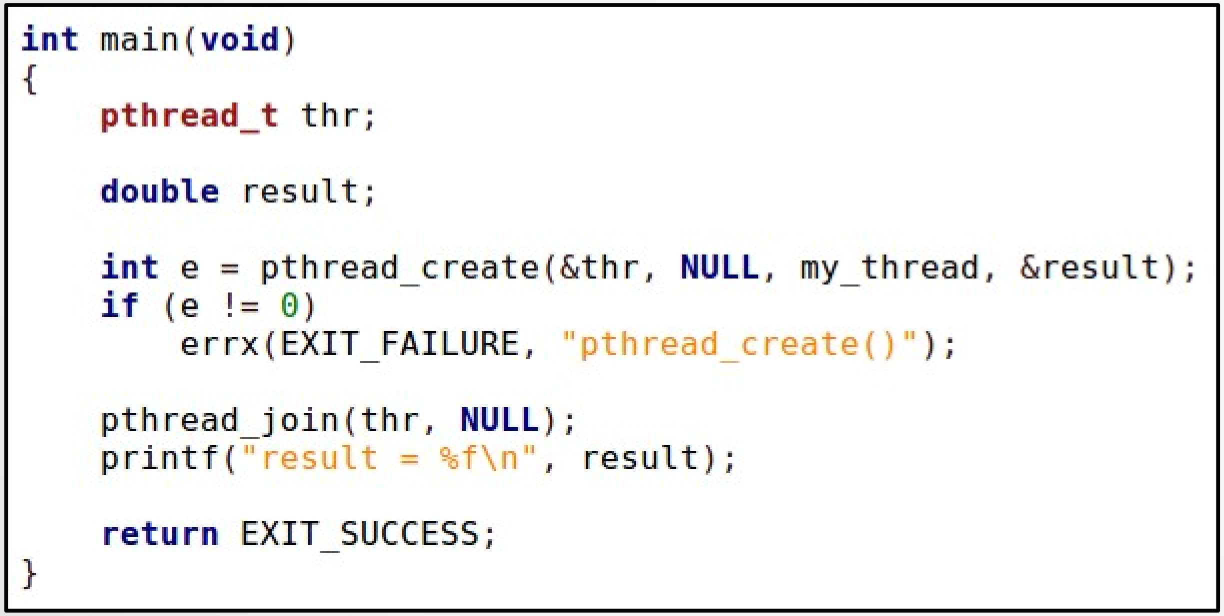
```
int main(void)
{
    pthread_t thr;

    int e = pthread_create(&thr, NULL, my_thread, NULL);
    if (e != 0)
        errx(EXIT_FAILURE, "pthread_create()");

    void *pvalue;
    pthread_join(thr, &pvalue);

    double value = *((double *)pvalue);
    free(pvalue);

    printf("The return value is: %f\n", value);

    return EXIT_SUCCESS;
}
```

```
void * my_thread(void *arg)
{
    double *value = malloc(sizeof(double));
    *value = 3.14;
    return value; // or pthread_exit(value)
}
```

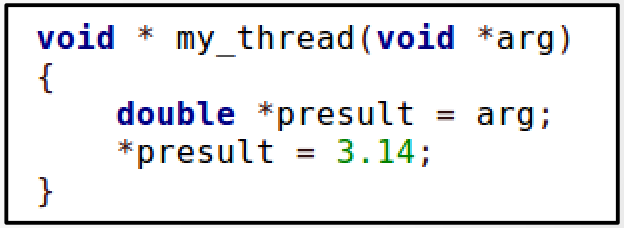
```
The return value is: 3.140000
```

Returning Values from Threads

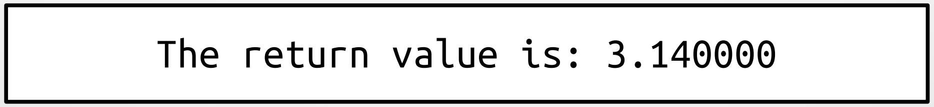
Another way...



```c
int main(void)
{
    pthread_t thr;

    double result;

    int e = pthread_create(&thr, NULL, my_thread, &result);
    if (e != 0)
        errx(EXIT_FAILURE, "pthread_create()");

    pthread_join(thr, NULL);
    printf("result = %f\n", result);

    return EXIT_SUCCESS;
}
```



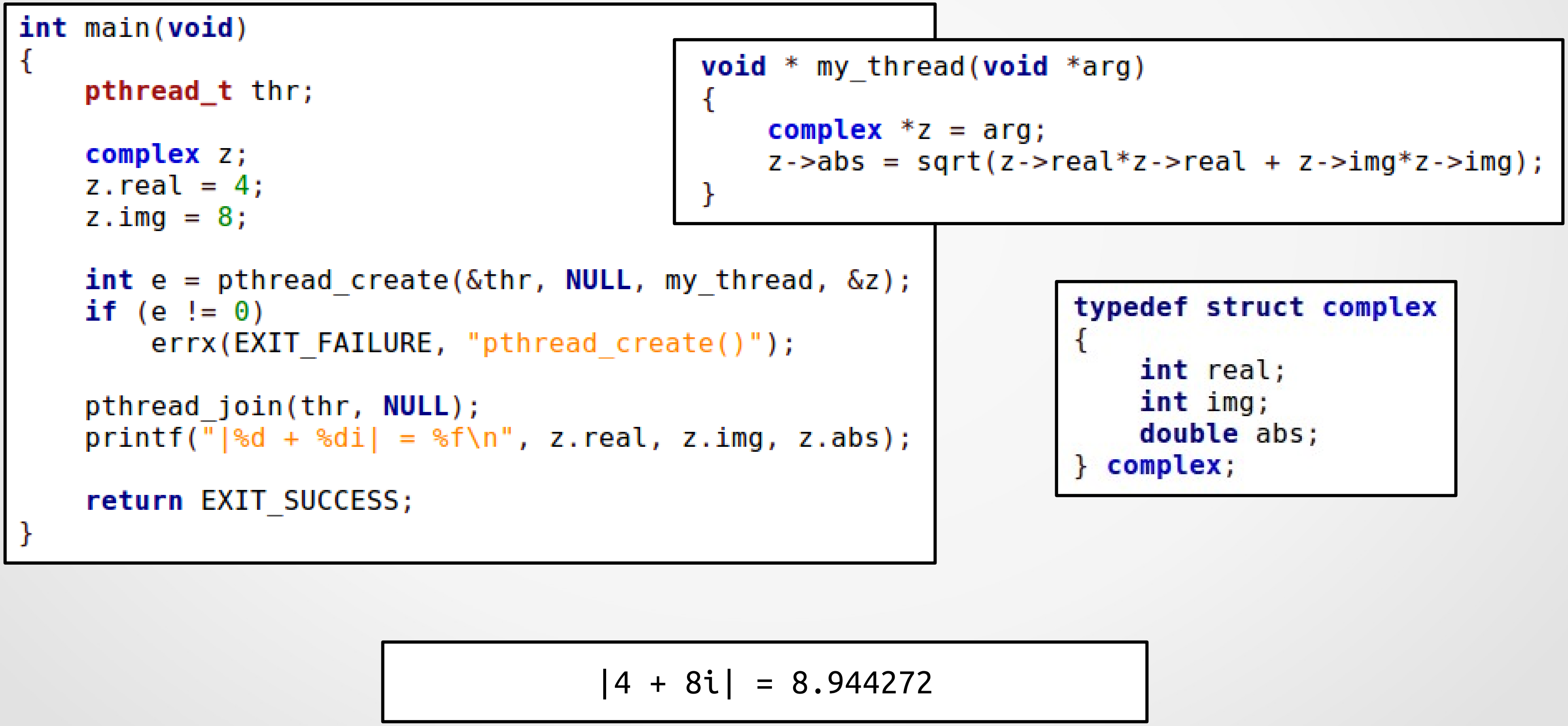
```c
void * my_thread(void *arg)
{
    double *presult = arg;
    *presult = 3.14;
}
```



```
The return value is: 3.140000
```

Returning Values from Threads

Another way with a structure...



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

    complex z;
    z.real = 4;
    z.img = 8;

    int e = pthread_create(&thr, NULL, my_thread, &z);
    if (e != 0)
        errx(EXIT_FAILURE, "pthread_create()");

    pthread_join(thr, NULL);
    printf("|%d + %di| = %f\n", z.real, z.img, z.abs);

    return EXIT_SUCCESS;
}
```

```
void * my_thread(void *arg)
{
    complex *z = arg;
    z->abs = sqrt(z->real*z->real + z->img*z->img);
}
```

```
typedef struct complex
{
    int real;
    int img;
    double abs;
} complex;
```

```
|4 + 8i| = 8.944272
```

Detaching Threads

A thread can be either *joinable* or *detached*.
By default, it is joinable.

A thread can be detached when:

- We do not need its return value.
- We do not need to wait for it.

In other words, a thread can be detached when we do not need to call the *phtread_join()* function for this thread.

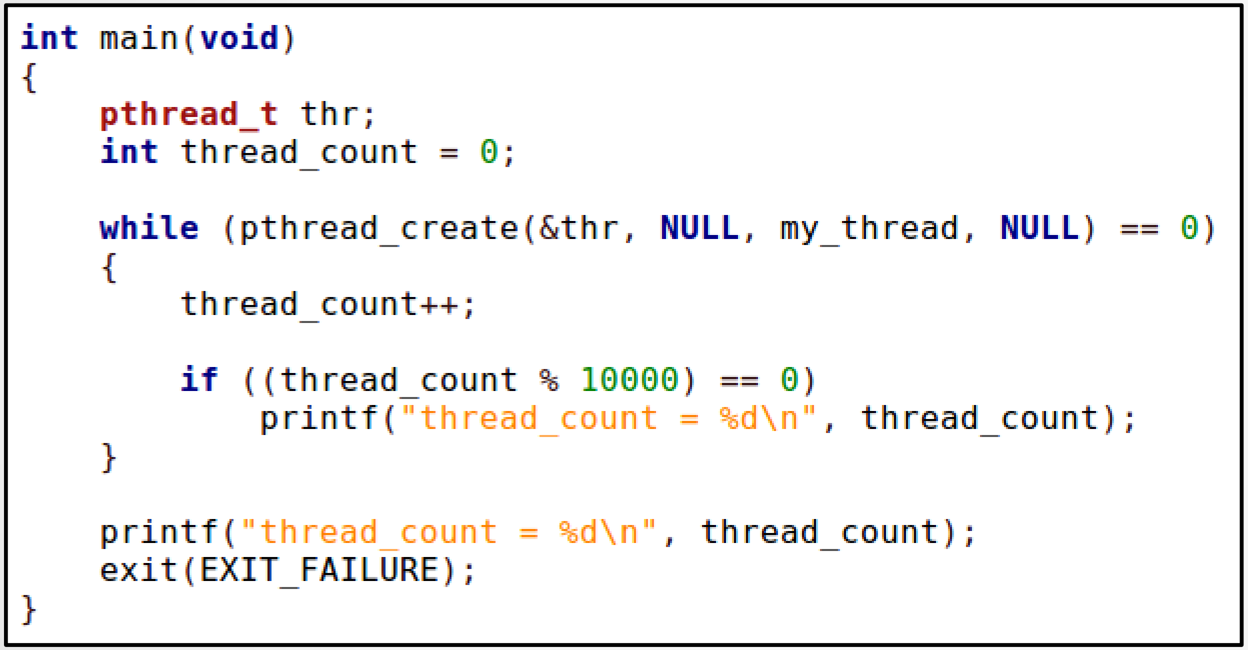
Detaching Threads

But why a thread should be detached?

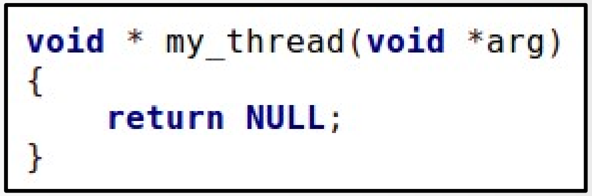
Can’t we just ignore its return value and never call the *pthread_join()* function?

Detaching Threads

Let us try this first program:



```c
int main(void)
{
    pthread_t thr;
    int thread_count = 0;

    while (pthread_create(&thr, NULL, my_thread, NULL) == 0)
    {
        thread_count++;

        if ((thread_count % 10000) == 0)
            printf("thread_count = %d\n", thread_count);
    }

    printf("thread_count = %d\n", thread_count);
    exit(EXIT_FAILURE);
}
```



```c
void * my_thread(void *arg)
{
    return NULL;
}
```

```
thread_count = 10000
thread_count = 20000
thread_count = 30000
thread_count = 32754
```

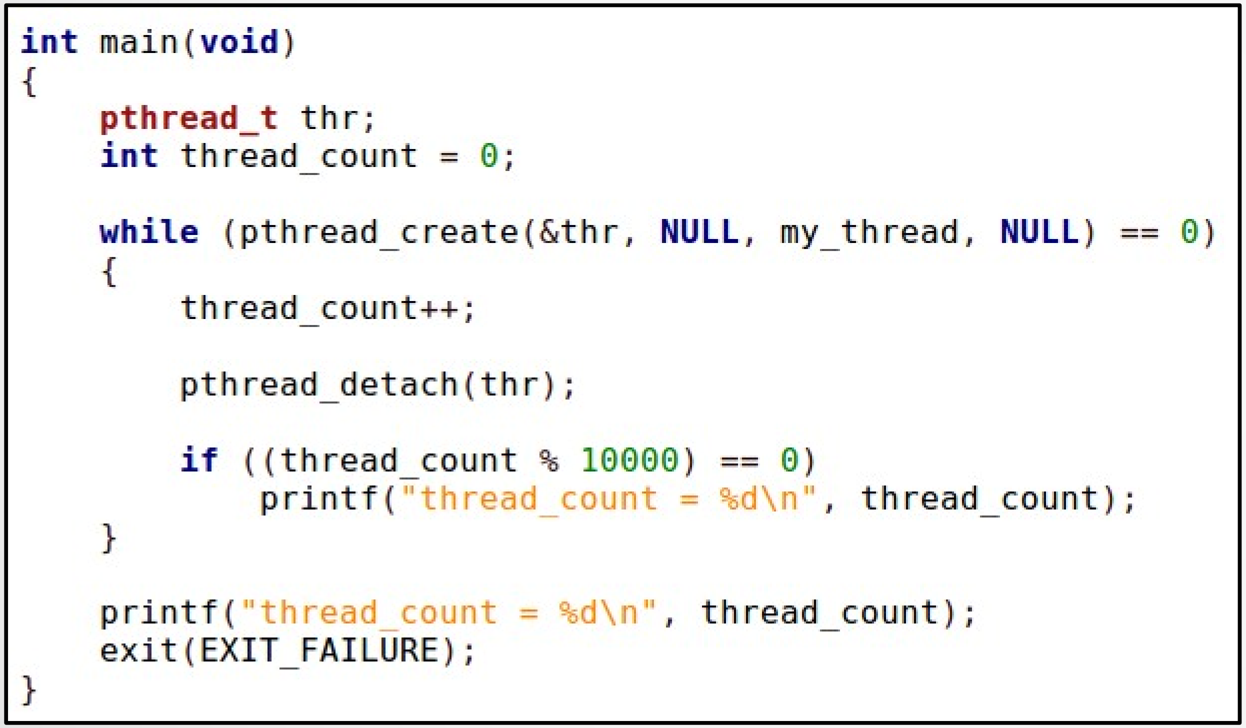
32,754 threads have been created before the *phtread_create()* function failed.

(This number may change according to your system.)

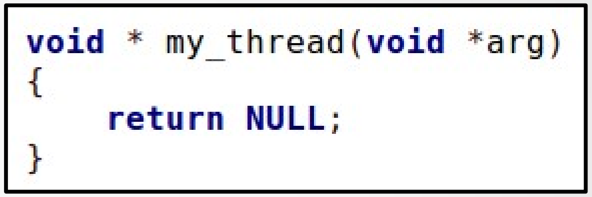
Detaching Threads

To detach a thread, we can use the *pthread_detach(3)* function.

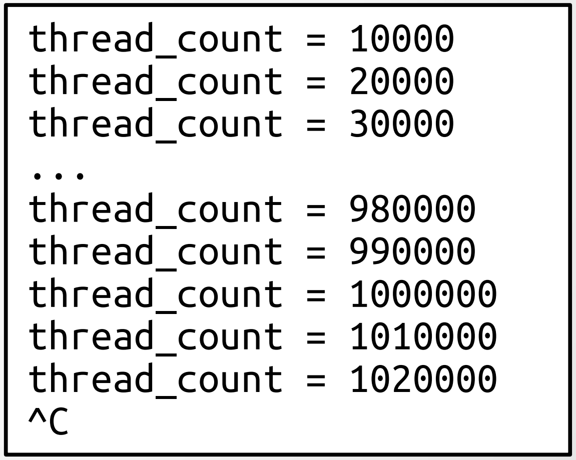
Let us try this second program:



```
int main(void)
{
    pthread_t thr;
    int thread_count = 0;

    while (pthread_create(&thr, NULL, my_thread, NULL) == 0)
    {
        thread_count++;

        pthread_detach(thr);

        if ((thread_count % 10000) == 0)
            printf("thread_count = %d\n", thread_count);
    }

    printf("thread_count = %d\n", thread_count);
    exit(EXIT_FAILURE);
}
```



```
void * my_thread(void *arg)
{
    return NULL;
}
```



```
thread_count = 10000
thread_count = 20000
thread_count = 30000
...
thread_count = 980000
thread_count = 990000
thread_count = 1000000
thread_count = 1010000
thread_count = 1020000
^C
```

The *phtread_create()* function never fails.

(When we stop the program, more than 1,000,000 threads have been created.)

Detaching Threads

**In the first program,
why is the number of joinable threads limited?**

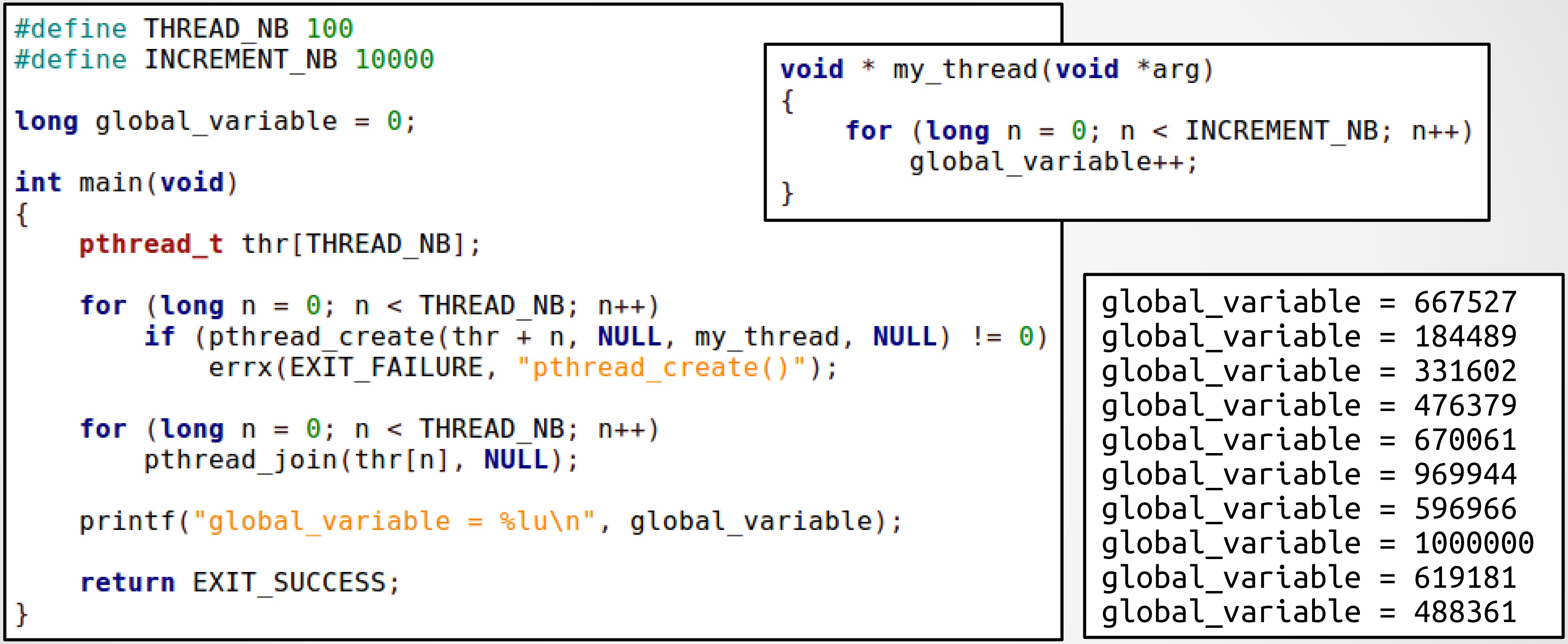
Because the resources used by a joinable thread (i.e. its stack, its return value) are released only when the *pthread_join()* function is called for this thread.

pthread_detach(3):

[…] Either pthread_join(3) or pthread_detach() should be called for each thread that an application creates, so that system resources for the thread can be released. (But note that the resources of all threads are freed when the process terminates.) [...]

Concurrent Memory Accesses

Let us run 10 times the following code:



```
#define THREAD_NB 100
#define INCREMENT_NB 10000

long global_variable = 0;

int main(void)
{
    pthread_t thr[THREAD_NB];

    for (long n = 0; n < THREAD_NB; n++)
        if (pthread_create(thr + n, NULL, my_thread, NULL) != 0)
            errx(EXIT_FAILURE, "pthread_create()");

    for (long n = 0; n < THREAD_NB; n++)
        pthread_join(thr[n], NULL);

    printf("global_variable = %lu\n", global_variable);

    return EXIT_SUCCESS;
}
```

```
void * my_thread(void *arg)
{
    for (long n = 0; n < INCREMENT_NB; n++)
        global_variable++;
}
```

```
global_variable = 667527
global_variable = 184489
global_variable = 331602
global_variable = 476379
global_variable = 670061
global_variable = 969944
global_variable = 596966
global_variable = 1000000
global_variable = 619181
global_variable = 488361
```

The expected value of *global_variable* is 1,000,000.

However, this value is always different.

Why?

Concurrent Memory Accesses

Almost all operations are not atomic

In the example: `global_variable++`

1. Fetch value from `global_variable`
2. Compute `global_variable + 1`
3. Write back result to `global_variable`

Modifications done to memory location between steps 1 and 3 are lost.

Critical Section

A section of code is said to be a critical section if the execution of this section cannot be interrupted without loss of consistency or determinism.

Mutex

Abstract entity used to enforce mutual exclusion.

Two basic operations: *lock* and *unlock*.

Lock: if the *mutex* is free, passes and takes it, otherwise waits until the actual owner of the *mutex* unlocks it.

Unlock: gives back the *mutex*.

Also trylock: Same as *lock* but does not wait.

Mutex – Type and Functions

```
// Declaration and Initialization
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

// Lock / Unlock
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);

// Destruction
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Mutex – Usage

Shared mutex variable: m

Thread code:

No Critical Section

Lock(m)

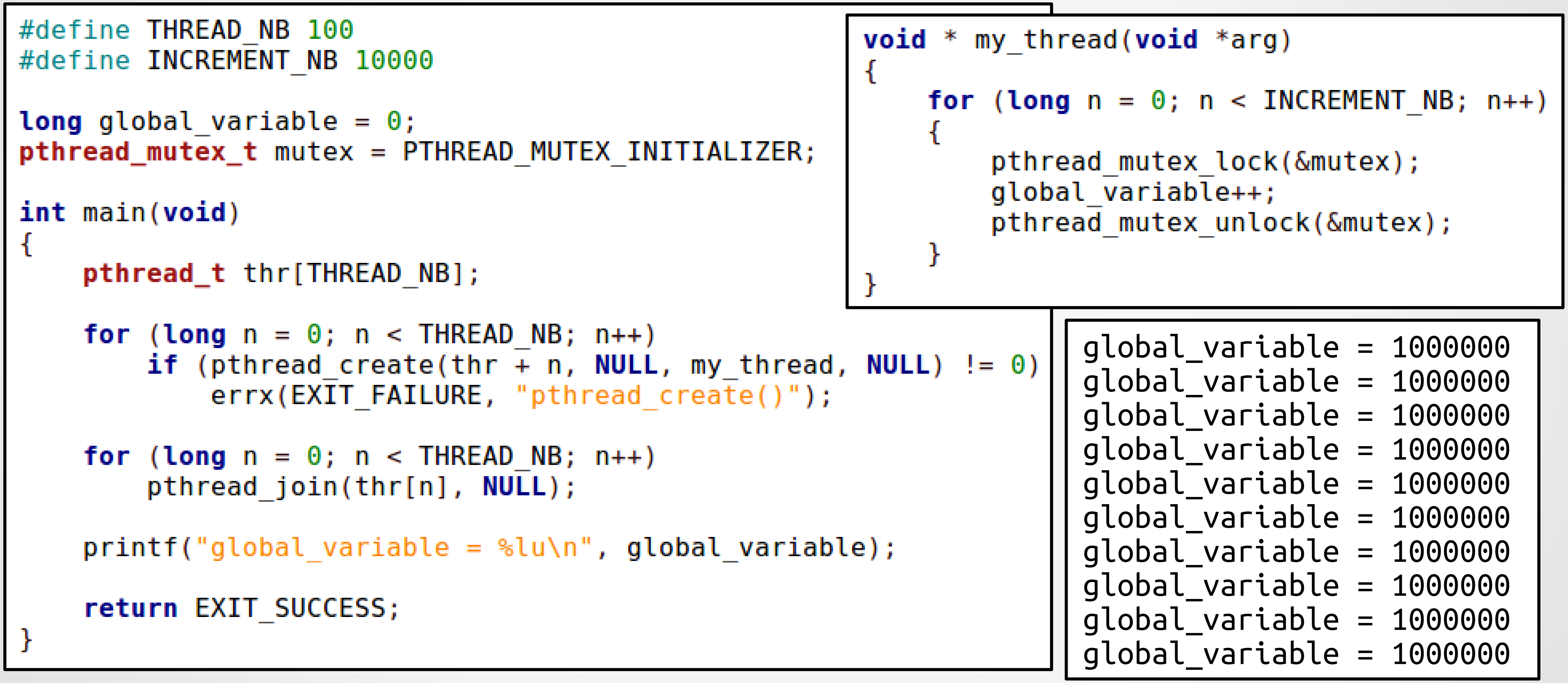
Critical Section

Unlock(m)

No Critical Section

Mutex – Example

Let us run the following code 10 times:



```c
#define THREAD_NB 100
#define INCREMENT_NB 10000

long global_variable = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

int main(void)
{
    pthread_t thr[THREAD_NB];

    for (long n = 0; n < THREAD_NB; n++)
        if (pthread_create(thr + n, NULL, my_thread, NULL) != 0)
            errx(EXIT_FAILURE, "pthread_create()");

    for (long n = 0; n < THREAD_NB; n++)
        pthread_join(thr[n], NULL);

    printf("global_variable = %lu\n", global_variable);

    return EXIT_SUCCESS;
}
```

```c
void * my_thread(void *arg)
{
    for (long n = 0; n < INCREMENT_NB; n++)
    {
        pthread_mutex_lock(&mutex);
        global_variable++;
        pthread_mutex_unlock(&mutex);
    }
}
```

```
global_variable = 1000000
global_variable = 1000000
global_variable = 1000000
global_variable = 1000000
global_variable = 1000000
global_variable = 1000000
global_variable = 1000000
global_variable = 1000000
global_variable = 1000000
global_variable = 1000000
```

The expected value of *global_variable* is 1,000,000.

Now, with the mutex, this value is always 1,000,000.

Deadlock (1)

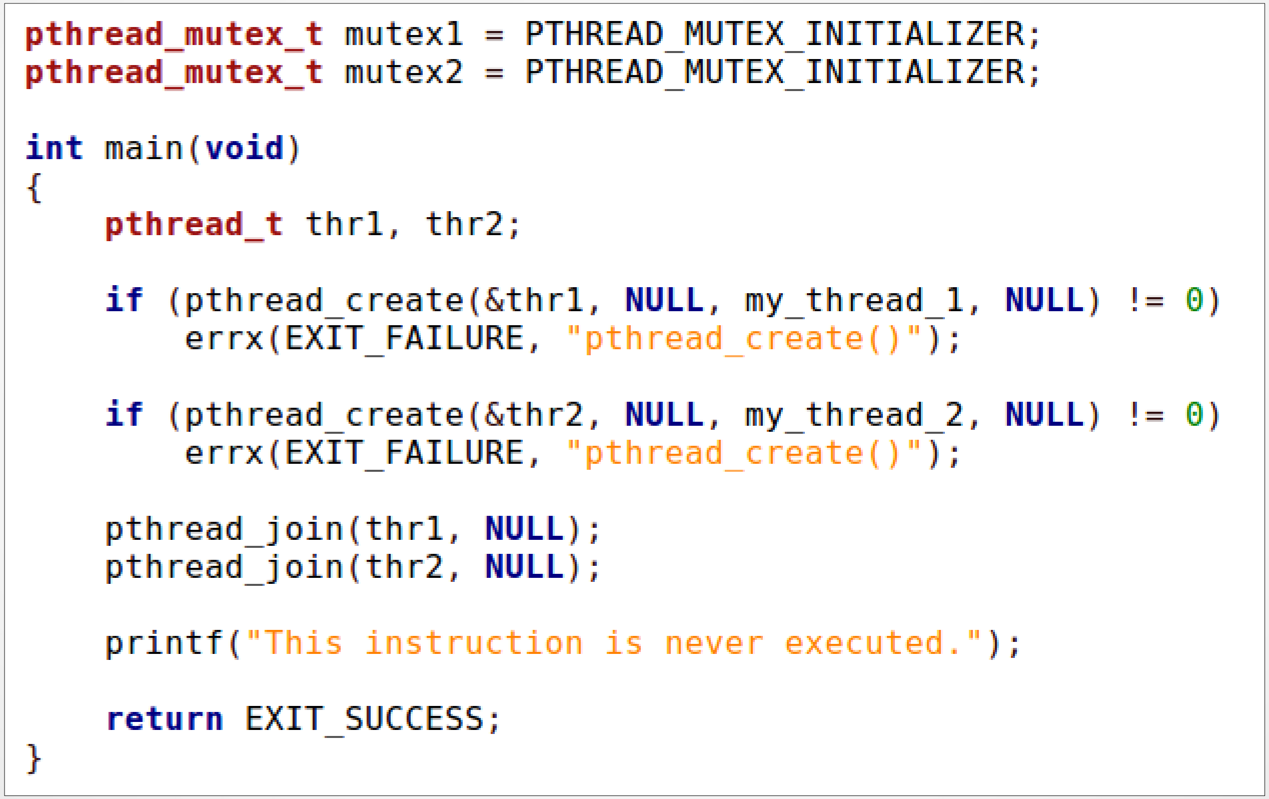
Thread 1 has locked Mutex 1

Thread 2 has locked Mutex 2

Thread 1 is waiting for Mutex 2

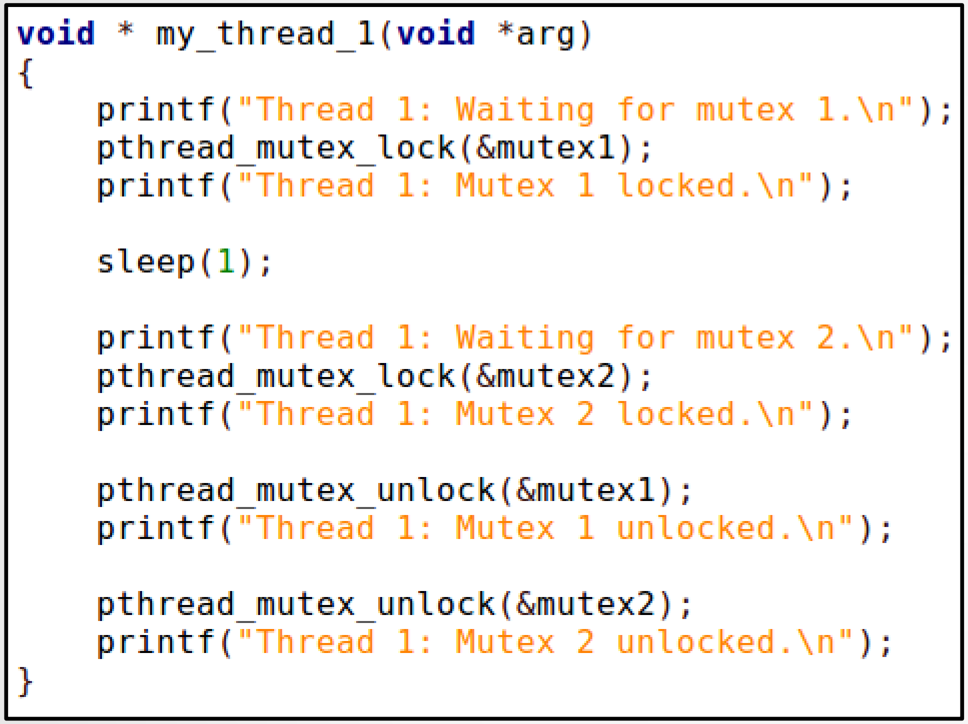
Thread 2 is waiting for Mutex 1

Deadlock (2)

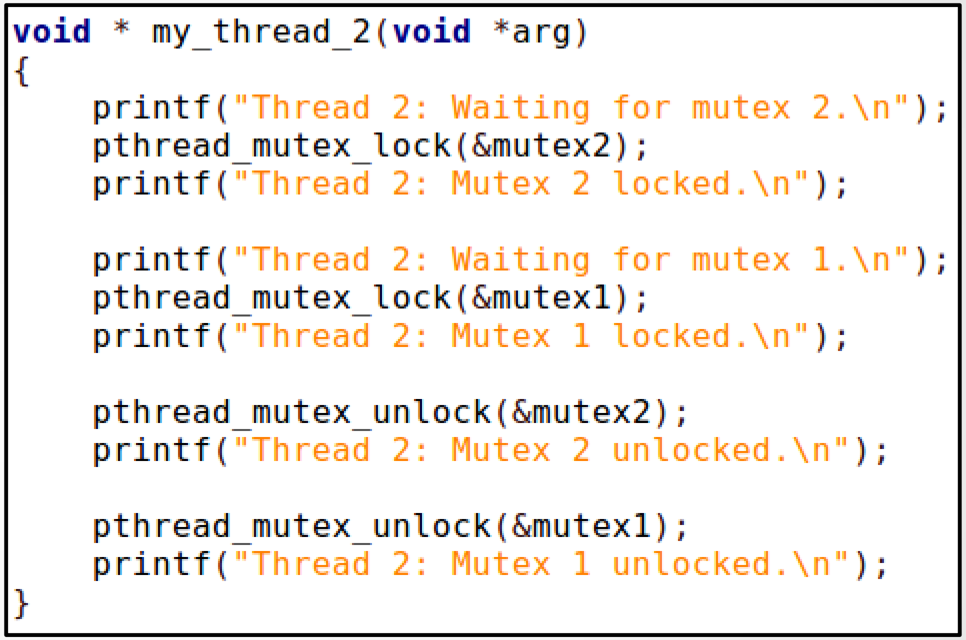


```
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t mutex2 = PTHREAD_MUTEX_INITIALIZER;

int main(void)
{
    pthread_t thr1, thr2;

    if (pthread_create(&thr1, NULL, my_thread_1, NULL) != 0)
        errx(EXIT_FAILURE, "pthread_create()");

    if (pthread_create(&thr2, NULL, my_thread_2, NULL) != 0)
        errx(EXIT_FAILURE, "pthread_create()");

    pthread_join(thr1, NULL);
    pthread_join(thr2, NULL);

    printf("This instruction is never executed.");

    return EXIT_SUCCESS;
}
```

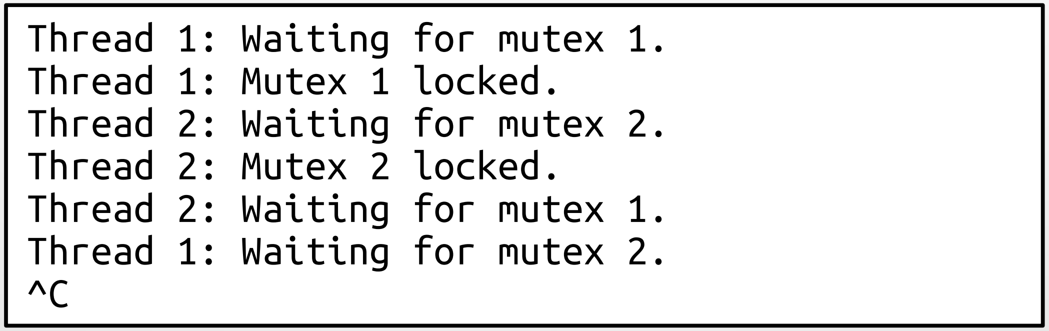
Deadlock (3)



```c
void * my_thread_1(void *arg)
{
    printf("Thread 1: Waiting for mutex 1.\n");
    pthread_mutex_lock(&mutex1);
    printf("Thread 1: Mutex 1 locked.\n");

    sleep(1);

    printf("Thread 1: Waiting for mutex 2.\n");
    pthread_mutex_lock(&mutex2);
    printf("Thread 1: Mutex 2 locked.\n");

    pthread_mutex_unlock(&mutex1);
    printf("Thread 1: Mutex 1 unlocked.\n");

    pthread_mutex_unlock(&mutex2);
    printf("Thread 1: Mutex 2 unlocked.\n");
}
```



```c
void * my_thread_2(void *arg)
{
    printf("Thread 2: Waiting for mutex 2.\n");
    pthread_mutex_lock(&mutex2);
    printf("Thread 2: Mutex 2 locked.\n");

    printf("Thread 2: Waiting for mutex 1.\n");
    pthread_mutex_lock(&mutex1);
    printf("Thread 2: Mutex 1 locked.\n");

    pthread_mutex_unlock(&mutex2);
    printf("Thread 2: Mutex 2 unlocked.\n");

    pthread_mutex_unlock(&mutex1);
    printf("Thread 2: Mutex 1 unlocked.\n");
}
```



```
Thread 1: Waiting for mutex 1.
Thread 1: Mutex 1 locked.
Thread 2: Waiting for mutex 2.
Thread 2: Mutex 2 locked.
Thread 2: Waiting for mutex 1.
Thread 1: Waiting for mutex 2.
^C
```

Condition Variables

Condition variables are another way to synchronize threads.

- A thread waits for a condition to be met.
- Another thread signals that the condition has been met.

A condition variable is always paired with a mutex.

Condition Variables – Type and Functions

```
// Declaration and Initialization
pthread_cont_t cont = PTHREAD_COND_INITIALIZER;

// Waiting for a condition to be met
int pthread_cond_wait(pthread_cond_t *cond);

// Signaling that a condition has been met
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);

// Destruction
int pthread_cond_destroy(pthread_cond_t *cond);
```

Condition Variables – In theory

Shared mutex variable: m
Shared condition variable: c

Thread 1:

```
    Do Work
    Lock(m)
    Wait(c)
    Unlock(m)
    Do Work
```

Thread 2:

```
    Do Work
    Lock(m)
    Signal(c)
    Unlock(m)
    Do Work
```

Condition Variables – In Practice

Shared mutex variable: m
Shared condition variable: c

Thread 1:

```
Do Work
Lock(m)
While(!condition)
    Wait(c)
Unlock(m)
Do Work
```

Thread 2:

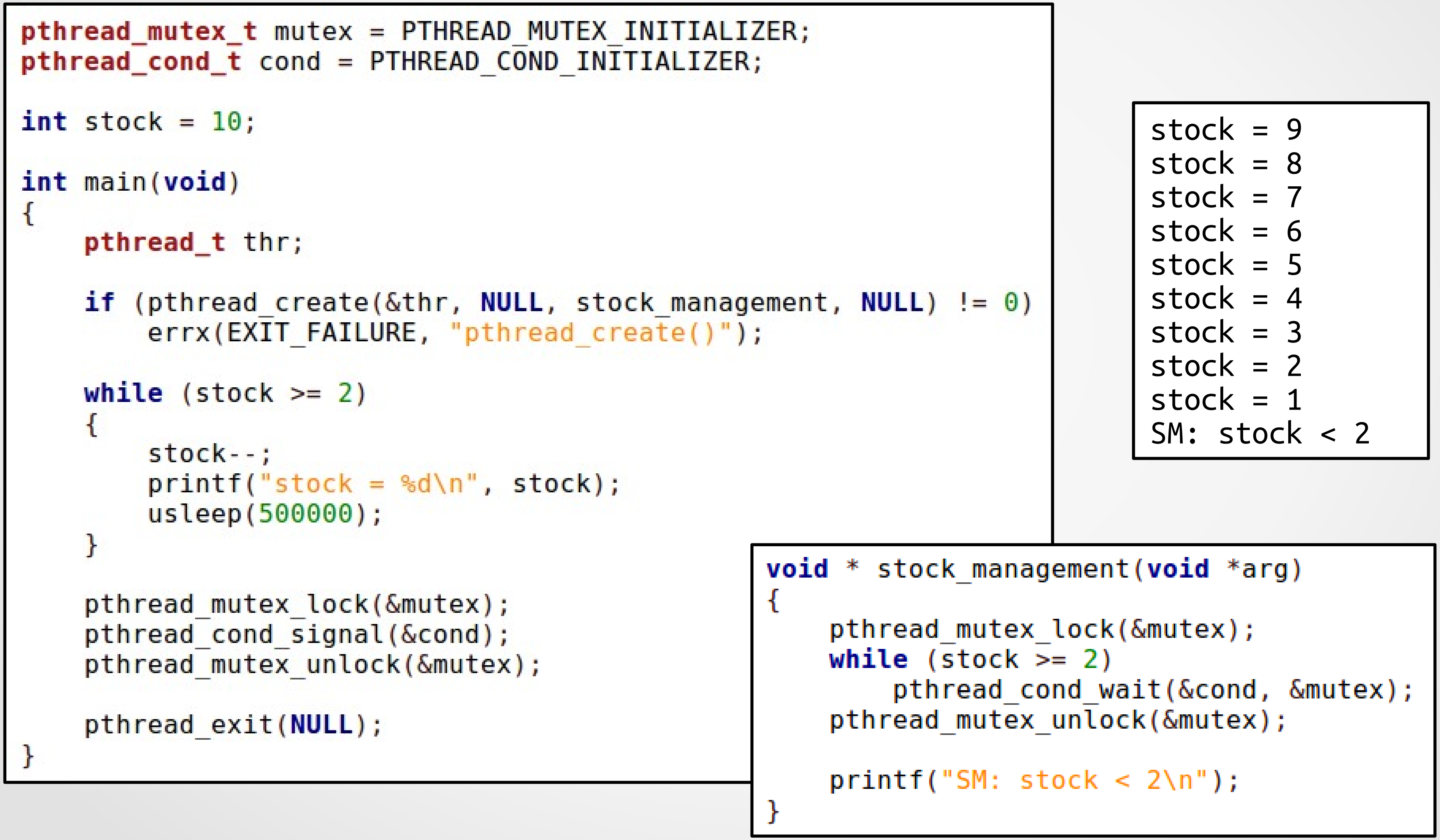
```
Do Work
Lock(m)
Signal(c)
Unlock(m)
Do Work
```

Condition Variables – Usage

Why should I wait for the condition in a *while* loop?

- **To prevent a bug:** the *pthread_cond_signal()* function was executed by mistake.
- **The *pthread_cond_wait()* function can return even if the condition is not met:** this behavior is allowed by the *pthread* library.

Condition Variables – Example

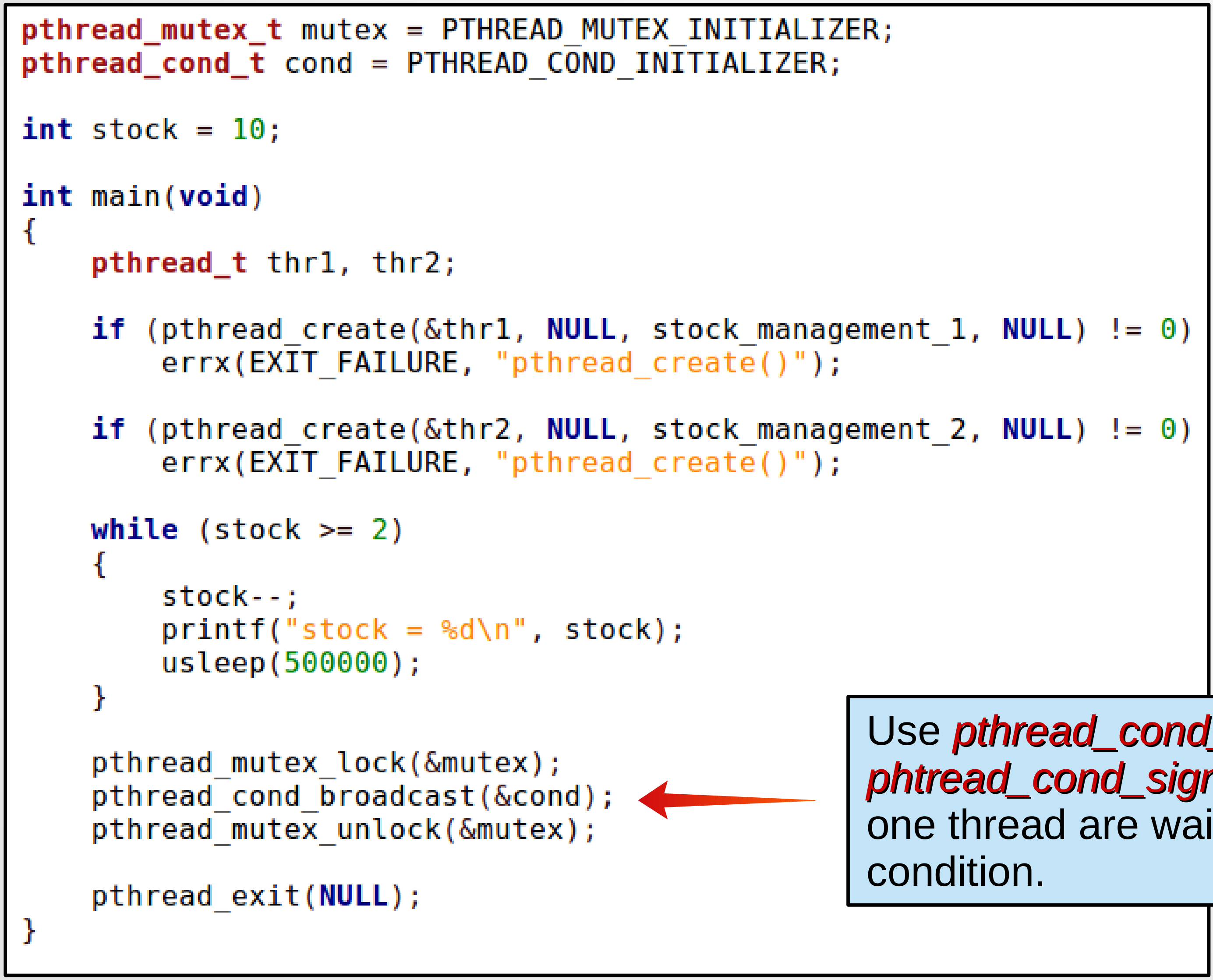


```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

int stock = 10;

int main(void)
{
    pthread_t thr;

    if (pthread_create(&thr, NULL, stock_management, NULL) != 0)
        errx(EXIT_FAILURE, "pthread_create()");

    while (stock >= 2)
    {
        stock--;
        printf("stock = %d\n", stock);
        usleep(500000);
    }

    pthread_mutex_lock(&mutex);
    pthread_cond_signal(&cond);
    pthread_mutex_unlock(&mutex);

    pthread_exit(NULL);
}
```

```
stock = 9
stock = 8
stock = 7
stock = 6
stock = 5
stock = 4
stock = 3
stock = 2
stock = 1
SM: stock < 2
```

```
void * stock_management(void *arg)
{
    pthread_mutex_lock(&mutex);
    while (stock >= 2)
        pthread_cond_wait(&cond, &mutex);
    pthread_mutex_unlock(&mutex);

    printf("SM: stock < 2\n");
}
```

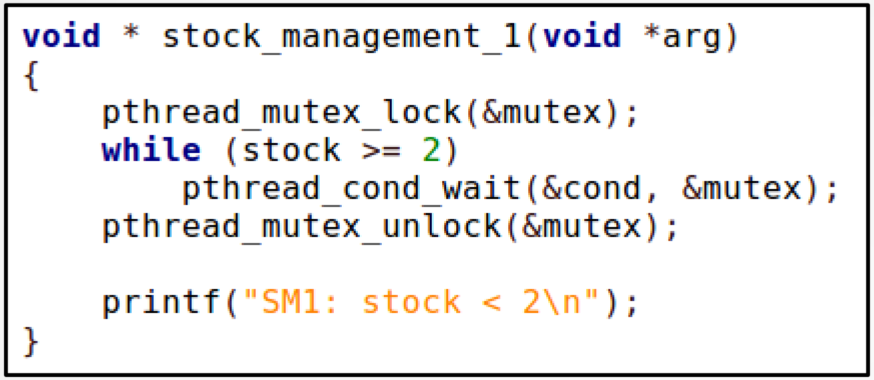
Condition Variables – Broadcast



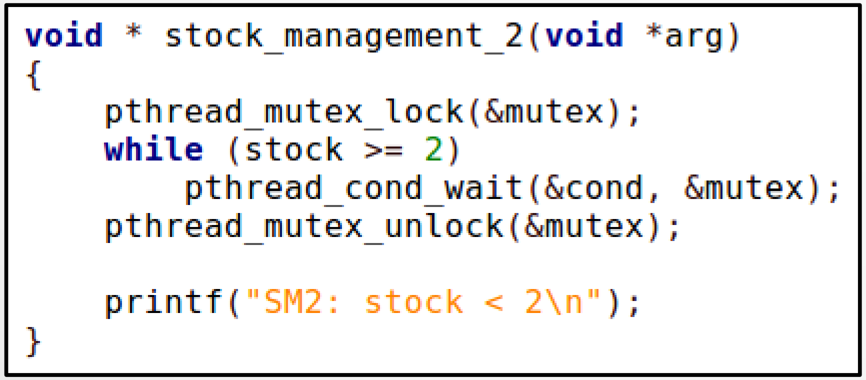
```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

int stock = 10;

int main(void)
{
    pthread_t thr1, thr2;

    if (pthread_create(&thr1, NULL, stock_management_1, NULL) != 0)
        errx(EXIT_FAILURE, "pthread_create()");

    if (pthread_create(&thr2, NULL, stock_management_2, NULL) != 0)
        errx(EXIT_FAILURE, "pthread_create()");

    while (stock >= 2)
    {
        stock--;
        printf("stock = %d\n", stock);
        usleep(500000);
    }

    pthread_mutex_lock(&mutex);
    pthread_cond_broadcast(&cond);
    pthread_mutex_unlock(&mutex);

    pthread_exit(NULL);
}
```

Use *pthread_cond_broadcast()* instead of *phtread_cond_signal()* when more than one thread are waiting for the same condition.

Condition Variables – Broadcast



```c
void * stock_management_1(void *arg)
{
    pthread_mutex_lock(&mutex);
    while (stock >= 2)
        pthread_cond_wait(&cond, &mutex);
    pthread_mutex_unlock(&mutex);

    printf("SM1: stock < 2\n");
}
```

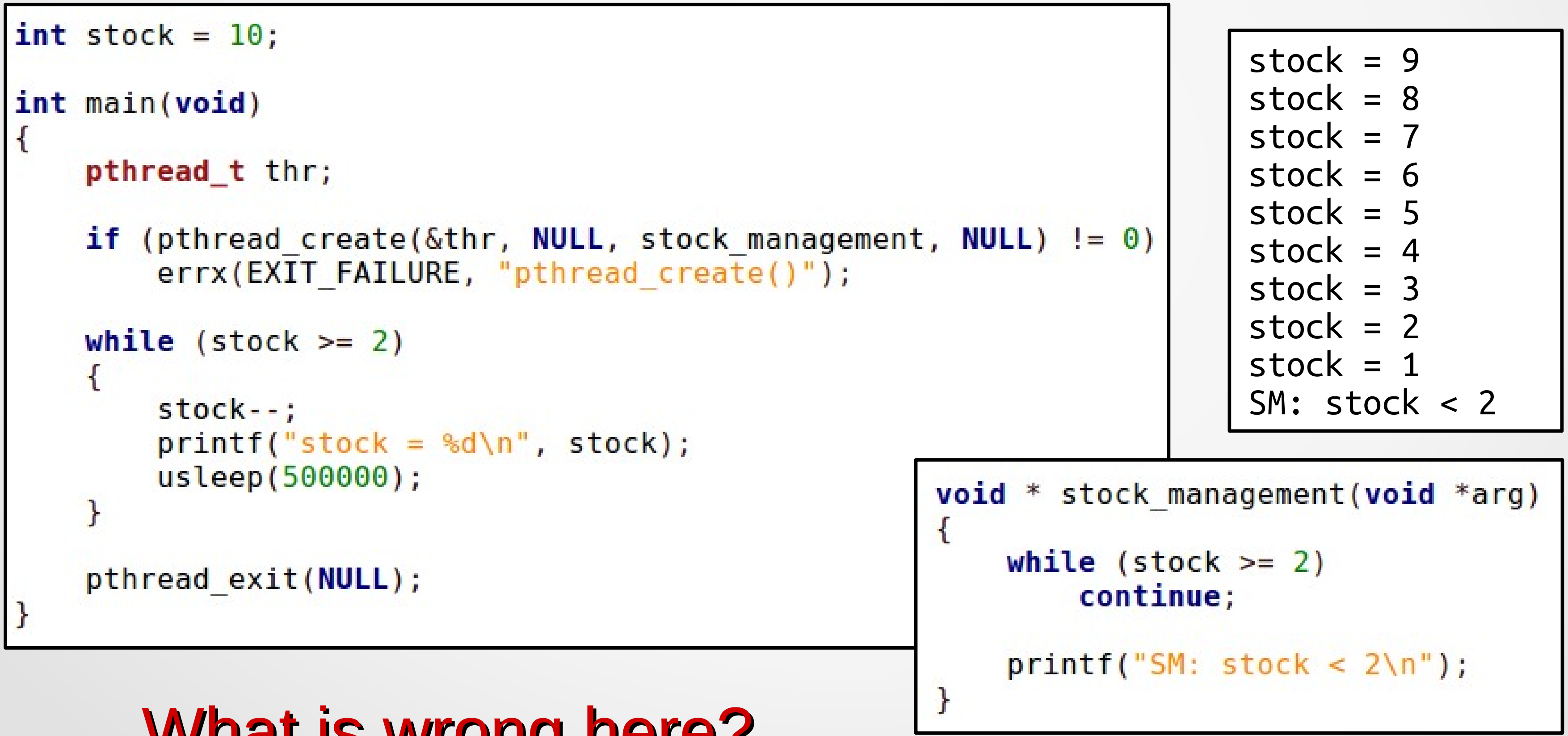


```c
void * stock_management_2(void *arg)
{
    pthread_mutex_lock(&mutex);
    while (stock >= 2)
        pthread_cond_wait(&cond, &mutex);
    pthread_mutex_unlock(&mutex);

    printf("SM2: stock < 2\n");
}
```

```
stock = 9
stock = 8
stock = 7
stock = 6
stock = 5
stock = 4
stock = 3
stock = 2
stock = 1
SM1: stock < 2
SM2: stock < 2
```

Why Condition Variables?

Why should we use condition variables?
Can’t we use *while* loops only?



```c
int stock = 10;

int main(void)
{
    pthread_t thr;

    if (pthread_create(&thr, NULL, stock_management, NULL) != 0)
        errx(EXIT_FAILURE, "pthread_create()");

    while (stock >= 2)
    {
        stock--;
        printf("stock = %d\n", stock);
        usleep(500000);
    }

    pthread_exit(NULL);
}
```

```
stock = 9
stock = 8
stock = 7
stock = 6
stock = 5
stock = 4
stock = 3
stock = 2
stock = 1
SM: stock < 2
```

```c
void * stock_management(void *arg)
{
    while (stock >= 2)
        continue;

    printf("SM: stock < 2\n");
}
```

What is wrong here?

Why Condition Variables?

Why should we use condition variables?
Can’t we use *while* loops only?

```
int stock = 10;

int main(void)
{
    pthread_t thr;

    if (pthread_create(&thr, NULL, stock_management, NULL) != 0)
        errx(EXIT_FAILURE, "pthread_create()");

    while (stock >= 2)
    {
        stock--;
        printf("stock = %d\n", stock);
        usleep(500000);
    }

    pthread_exit(NULL);
}
```

```
stock = 9
stock = 8
stock = 7
stock = 6
stock = 5
stock = 4
stock = 3
stock = 2
stock = 1
SM: stock < 2
```

```
void * stock_management(void *arg)
{
    while (stock >= 2)
        continue;

    printf("SM: stock < 2\n");
}
```

Too Much Resource Consuming

What is wrong here?

Semaphores

POSIX semaphores allow processes and threads to synchronize their actions.

A semaphore is an integer whose value is never allowed to fall below zero.

Two operations can be performed on semaphores:

- increment the semaphore value by one (*sem_post(3)*);
- and decrement the semaphore value by one (*sem_wait(3)*).

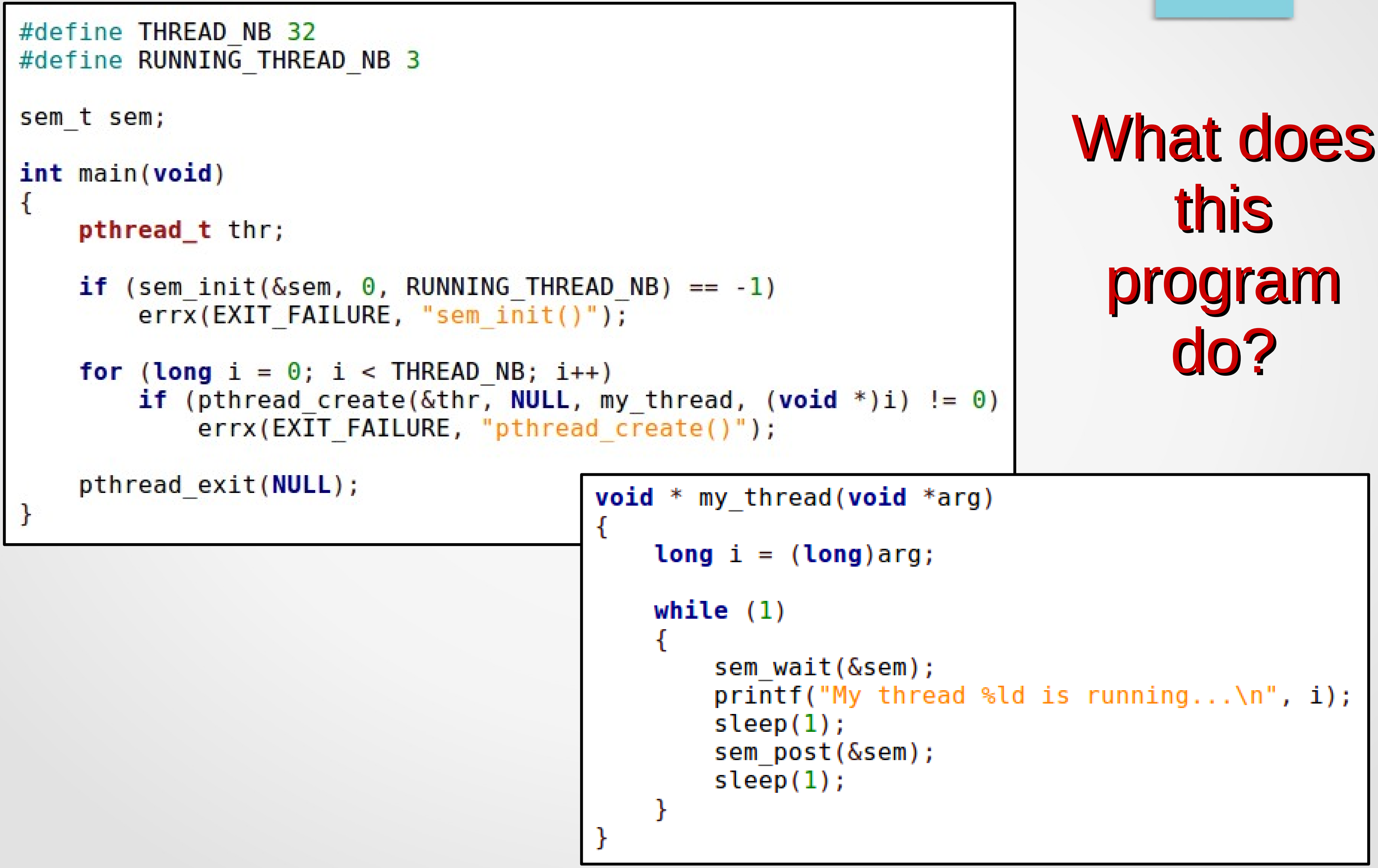
If the value of a semaphore is currently zero, then a *sem_wait(3)* operation will block until the value becomes greater than zero.

→ sem_overview(7)

Semaphores – Type and Functions

```
// Initialization (for threads of the same process)
int sem_init(sem_t *sem, 0, unsigned int value);

// Incrementing and Decrementing
int sem_post(sem_t *sem);
int sem_wait(sem_t *sem);

// Destruction
int sem_destroy(sem_t *sem);
```

Semaphores – Example

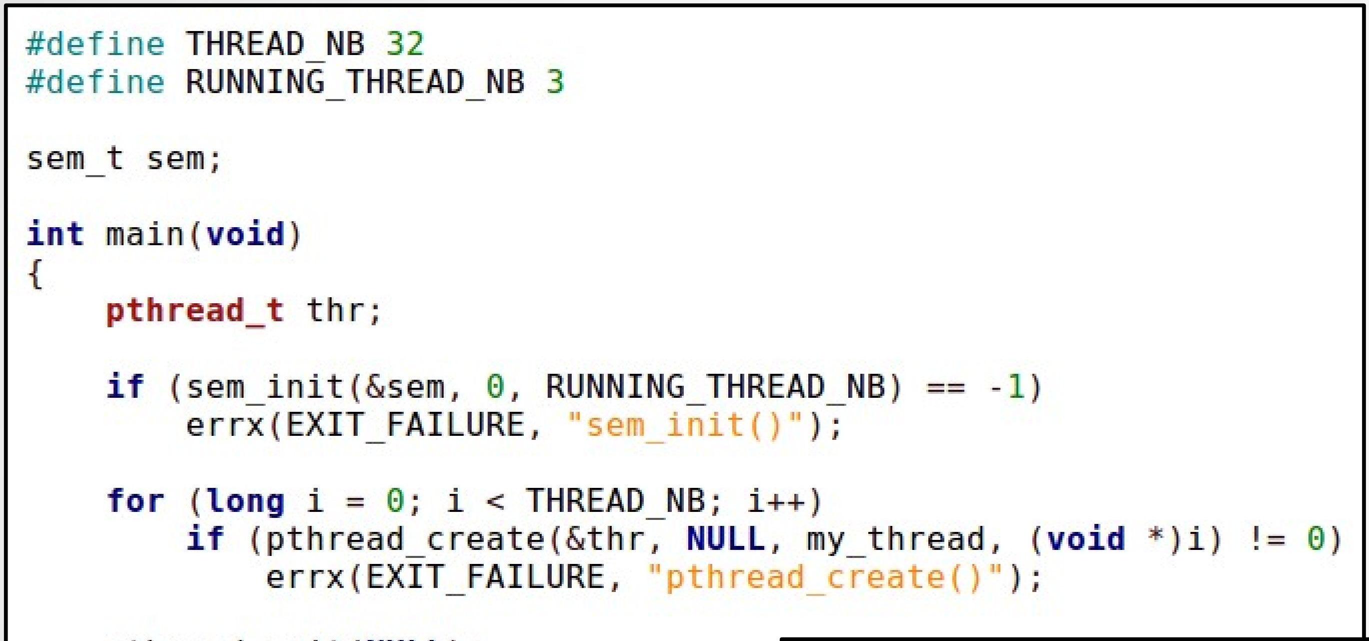


```c
#define THREAD_NB 32
#define RUNNING_THREAD_NB 3

sem_t sem;

int main(void)
{
    pthread_t thr;

    if (sem_init(&sem, 0, RUNNING_THREAD_NB) == -1)
        errx(EXIT_FAILURE, "sem_init()");

    for (long i = 0; i < THREAD_NB; i++)
        if (pthread_create(&thr, NULL, my_thread, (void *)i) != 0)
            errx(EXIT_FAILURE, "pthread_create()");

    pthread_exit(NULL);
}
```

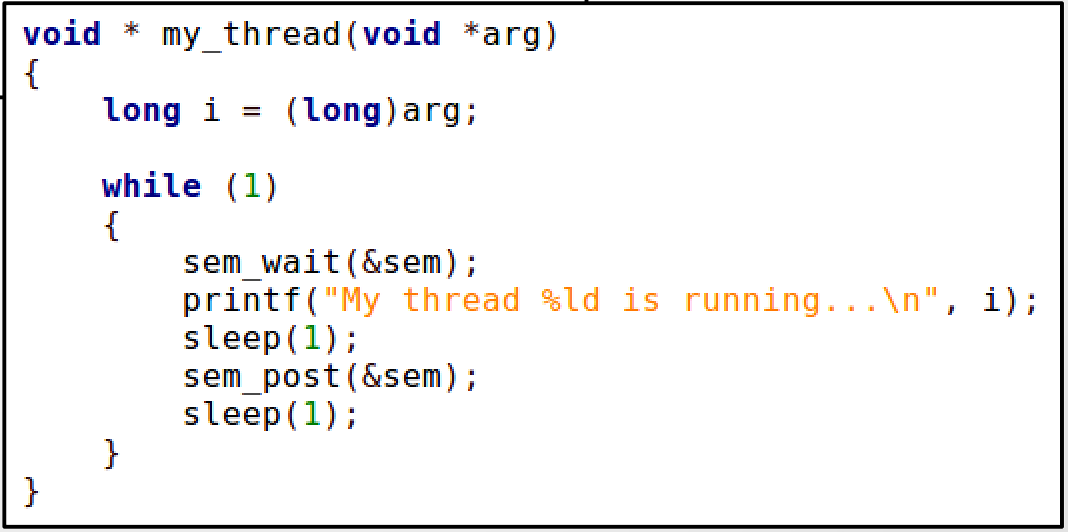
```c
void * my_thread(void *arg)
{
    long i = (long)arg;

    while (1)
    {
        sem_wait(&sem);
        printf("My thread %ld is running...\n", i);
        sleep(1);
        sem_post(&sem);
        sleep(1);
    }
}
```

What does this program do?

Semaphores – Example



```
#define THREAD_NB 32
#define RUNNING_THREAD_NB 3

sem_t sem;

int main(void)
{
    pthread_t thr;

    if (sem_init(&sem, 0, RUNNING_THREAD_NB) == -1)
        errx(EXIT_FAILURE, "sem_init()");

    for (long i = 0; i < THREAD_NB; i++)
        if (pthread_create(&thr, NULL, my_thread, (void *)i) != 0)
            errx(EXIT_FAILURE, "pthread_create()");

    pthread_exit(NULL);
}
```



```
void * my_thread(void *arg)
{
    long i = (long)arg;

    while (1)
    {
        sem_wait(&sem);
        printf("My thread %ld is running...\n", i);
        sleep(1);
        sem_post(&sem);
        sleep(1);
    }
}
```

What does this program do?

It creates 32 threads and limits the number of threads that are running to 3.