운영체제 및 실습
- Concurrency -

한예진, 김수창, 이승준

Dankook University

Concurrency

1. Concurrency

2. Pthread Programming

3. Race Condition

4. Lock and Unlock

5. Conditional Variable

Concurrency

1. Concurrency

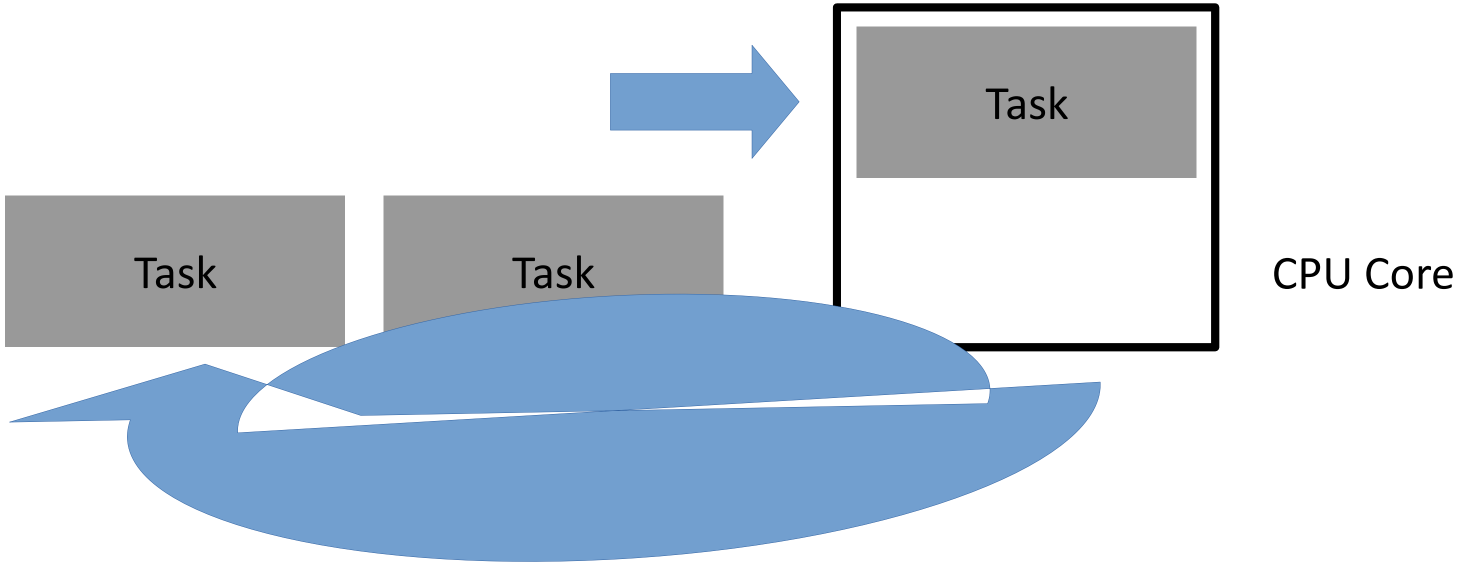
Concurrency means multiple computations are happening at the same time

Concurrency

1. Concurrency

즉, 하나의 일만 수행하는 것이 아닌 한 가지의 일을 쪼개어 실행

=> **여러가지 일을 동시에 실행하는 것처럼 보임**



Concurrency

1. Concurrency

Multi Process or Multi Thread

이전 시간엔 Multi Process를 배웠으니 이번 시간엔 Multi Thread에 대해서 알아볼 예정

Thread란?

스레드(thread)는 어떠한 프로그램 내에서, 특히 프로세스 내에서 실행되는 흐름의 단위를 말한다.

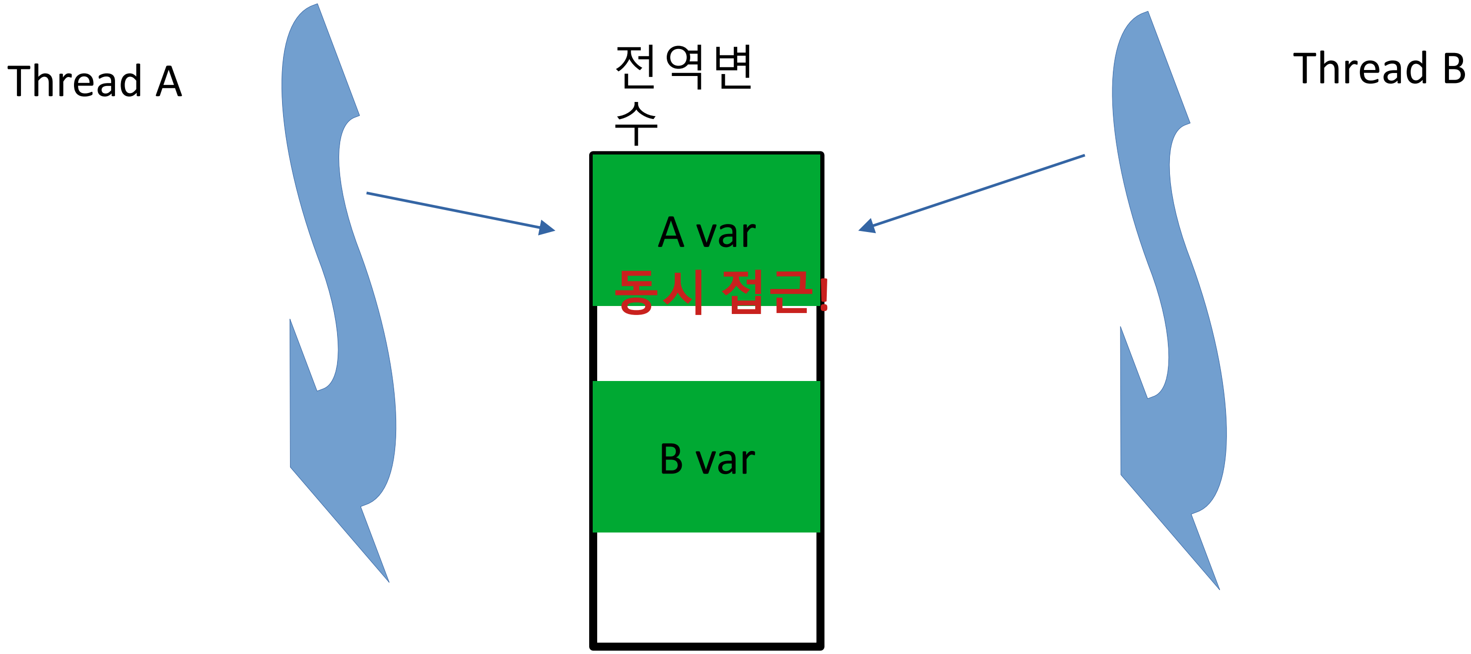
일반적으로 한 프로그램은 하나의 스레드를 가지고 있지만, 프로그램 환경에 따라 둘 이상의 스레드를 동시에 실행할 수 있다. 이러한 실행 방식을 멀티스레드(multithread)라고 한다.

Concurrency

1. Concurrency

Process는 Text 영역을 제외하고는 독립적인 메모리공간을 가지지만
Thread는 Stack을 제외하고는 다른 Thread와 공유한다.

가볍지만 Race Condition이 발생할 수 있음



Concurrency

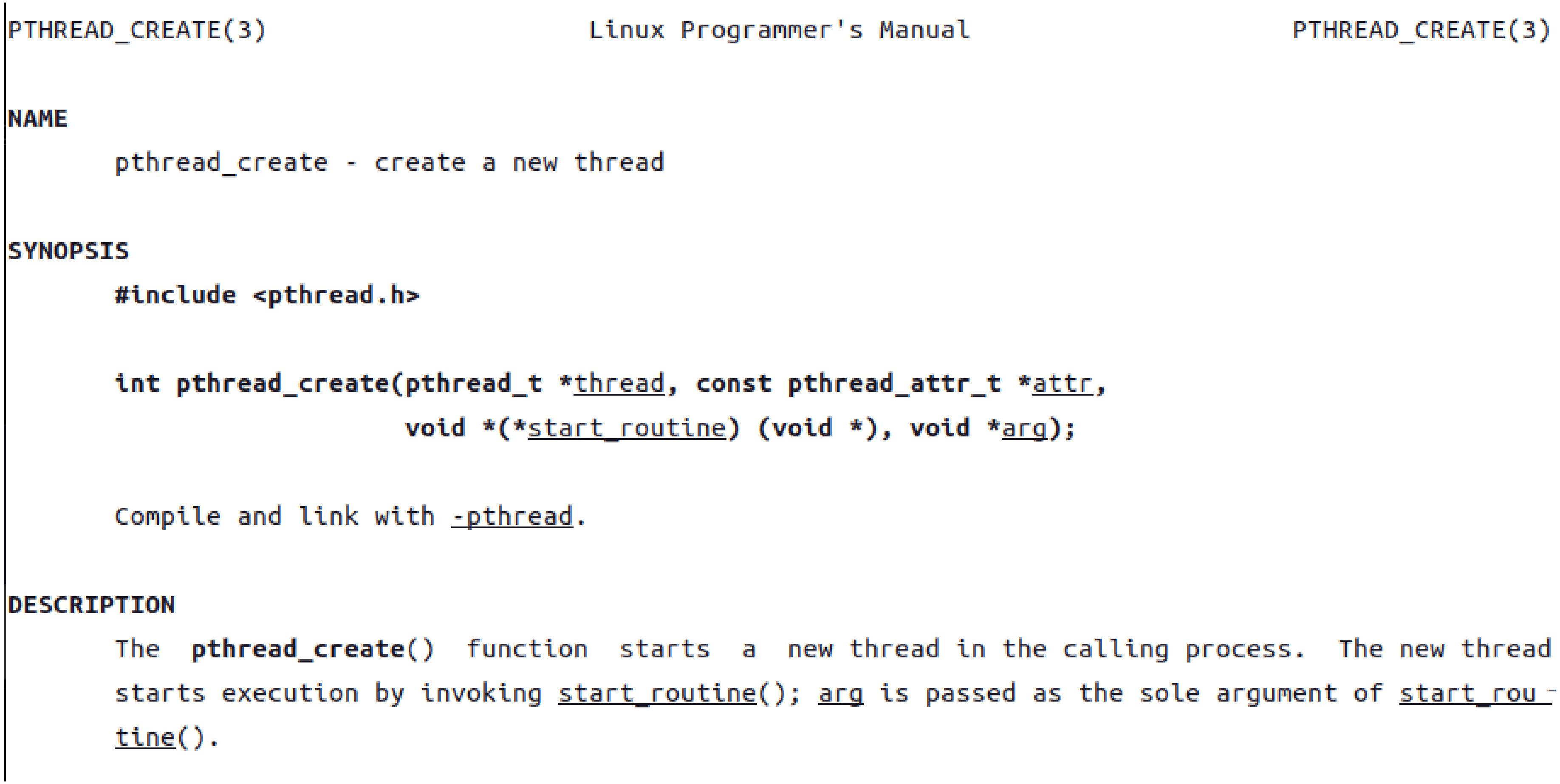
1. Concurrency

How To Use Thread on Linux?

Use POSIX Thread API

Concurrency

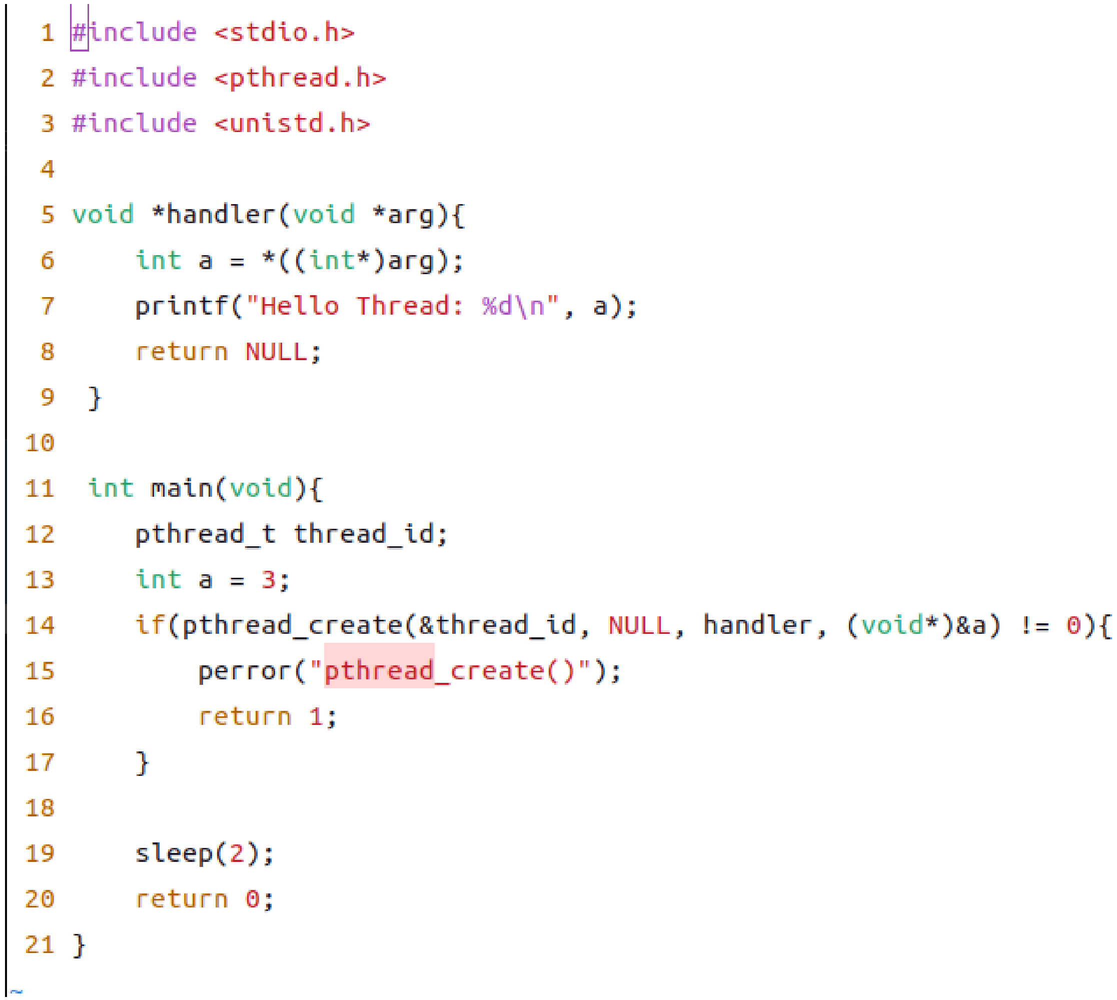
1. Concurrency



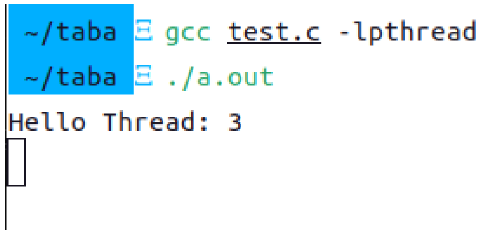
```
PTHREAD_CREATE(3)               Linux Programmer's Manual               PTHREAD_CREATE(3)

NAME
       pthread_create - create a new thread

SYNOPSIS
       #include <pthread.h>

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

       Compile and link with -pthread.

DESCRIPTION
       The  pthread_create()  function  starts  a  new thread in the calling process.  The new thread
       starts execution by invoking start_routine(); arg is passed as the sole argument of start_rou‐
       tine().
```

Concurrency

1. Concurrency



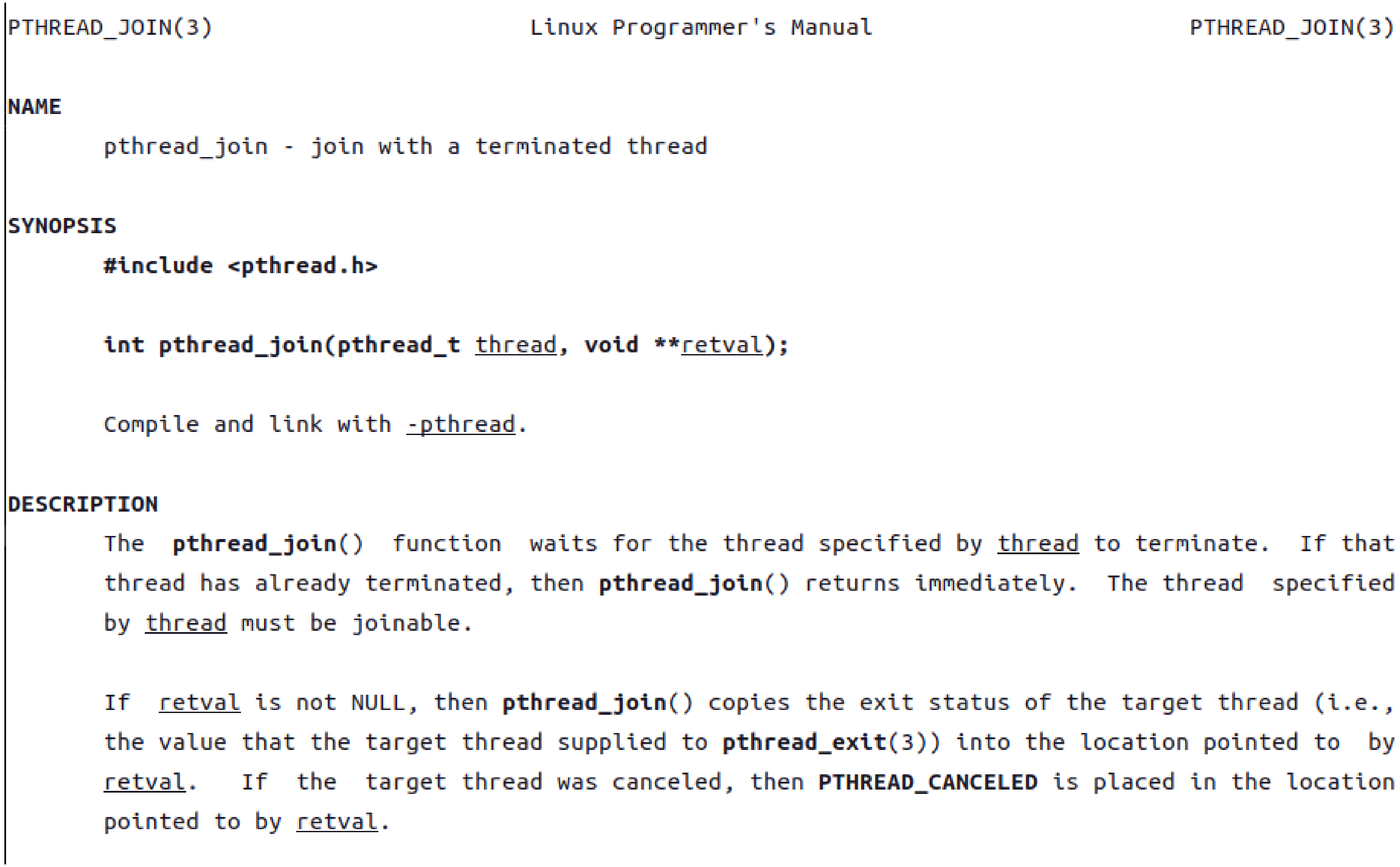
```c
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

void *handler(void *arg){
    int a = *((int*)arg);
    printf("Hello Thread: %d\n", a);
    return NULL;
 }

 int main(void){
    pthread_t thread_id;
    int a = 3;
    if(pthread_create(&thread_id, NULL, handler, (void*)&a) != 0){
        perror("pthread_create()");
        return 1;
    }

    sleep(2);
    return 0;
}
~
```



```
 ~/taba Ξ gcc test.c -lpthread
 ~/taba Ξ ./a.out
Hello Thread: 3
```

Concurrency

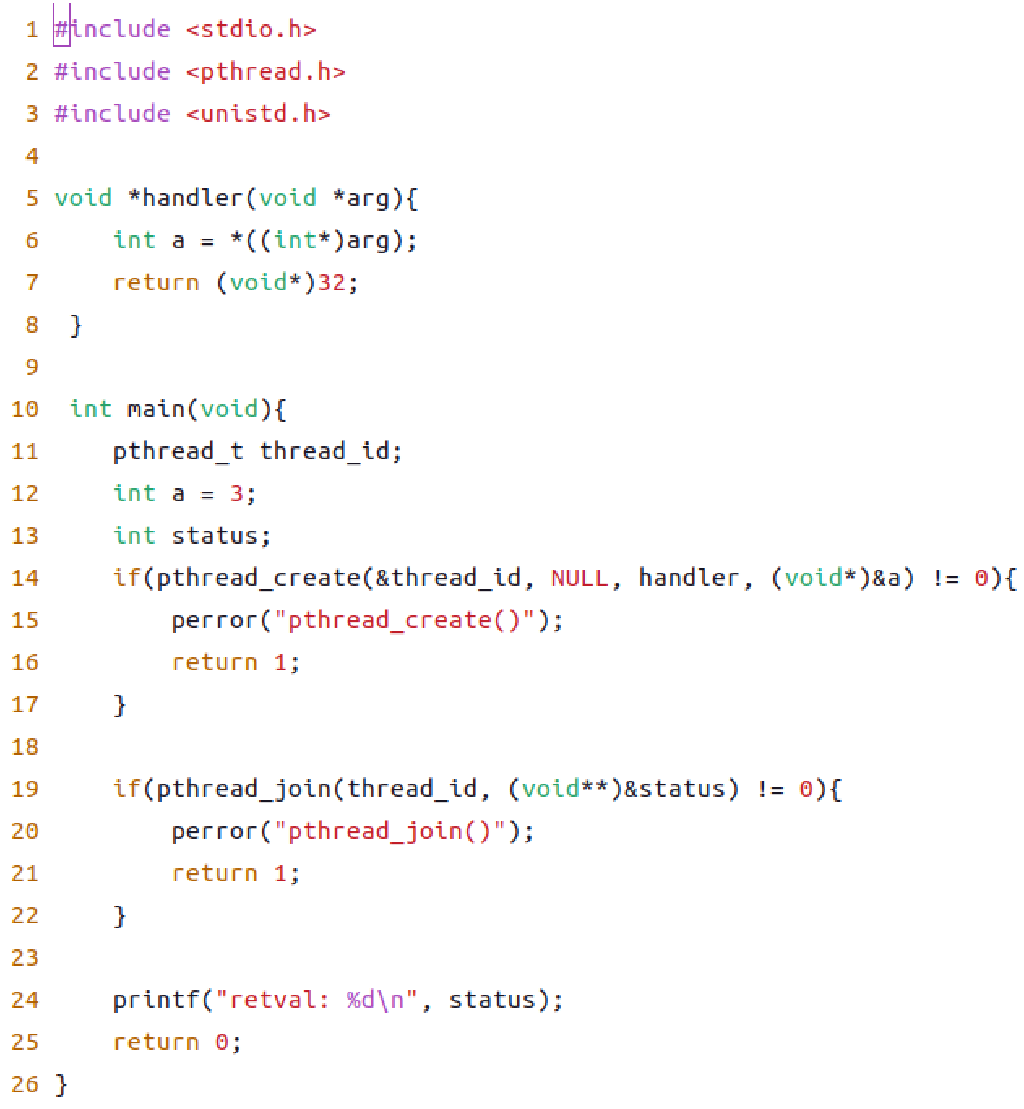
1. Concurrency



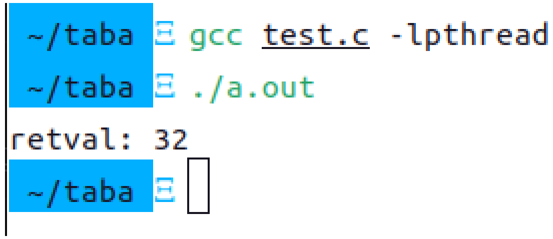
```
PTHREAD_JOIN(3)                Linux Programmer's Manual               PTHREAD_JOIN(3)

NAME
       pthread_join - join with a terminated thread

SYNOPSIS
       #include <pthread.h>

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

       Compile and link with -pthread.

DESCRIPTION
       The  pthread_join()  function  waits for the thread specified by thread to terminate.  If that
       thread has already terminated, then pthread_join() returns immediately.  The thread  specified
       by thread must be joinable.

       If  retval is not NULL, then pthread_join() copies the exit status of the target thread (i.e.,
       the value that the target thread supplied to pthread_exit(3)) into the location pointed to  by
       retval.   If  the  target thread was canceled, then PTHREAD_CANCELED is placed in the location
       pointed to by retval.
```

Concurrency

1. Concurrency



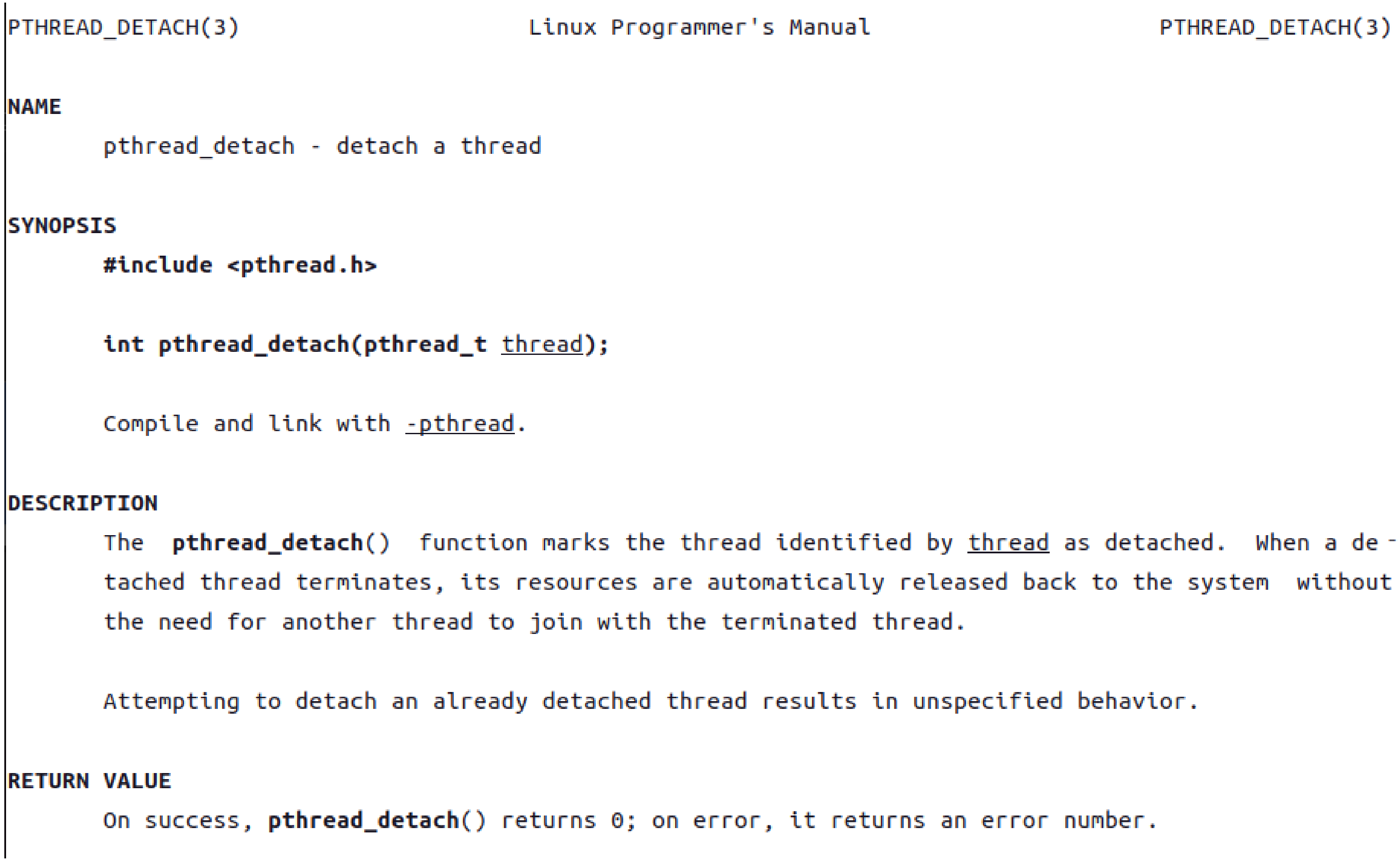
```c
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

void *handler(void *arg){
    int a = *((int*)arg);
    return (void*)32;
 }

 int main(void){
    pthread_t thread_id;
    int a = 3;
    int status;
    if(pthread_create(&thread_id, NULL, handler, (void*)&a) != 0){
        perror("pthread_create()");
        return 1;
    }

    if(pthread_join(thread_id, (void**)&status) != 0){
        perror("pthread_join()");
        return 1;
    }

    printf("retval: %d\n", status);
    return 0;
}
```



```
 ~/taba  gcc test.c -lpthread
 ~/taba  ./a.out
retval: 32
 ~/taba 
```

Concurrency

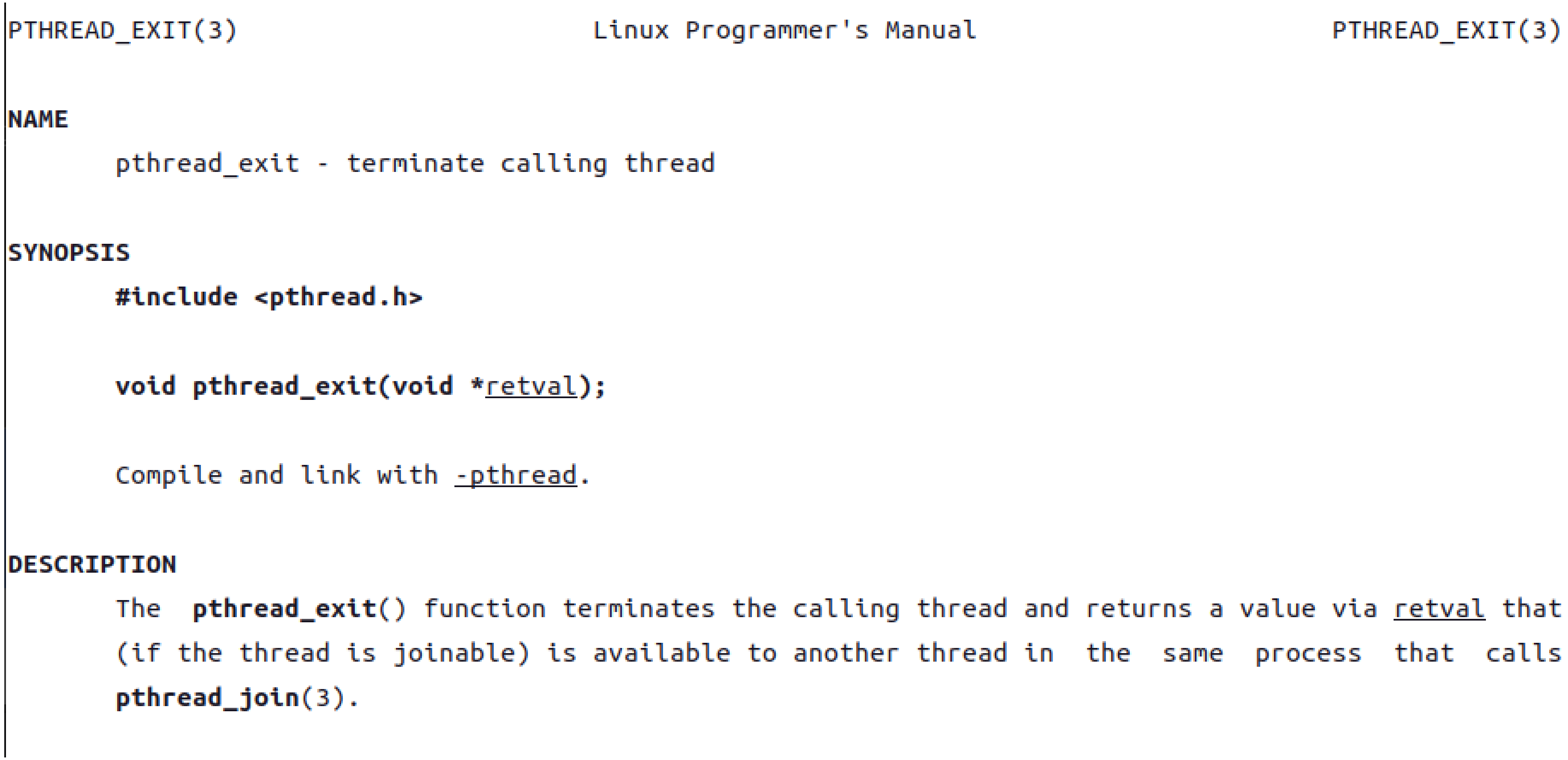
1. Concurrency



```
PTHREAD_DETACH(3)               Linux Programmer's Manual               PTHREAD_DETACH(3)

NAME
       pthread_detach - detach a thread

SYNOPSIS
       #include <pthread.h>

       int pthread_detach(pthread_t thread);

       Compile and link with -pthread.

DESCRIPTION
       The  pthread_detach()  function marks the thread identified by thread as detached.  When a de‐
       tached thread terminates, its resources are automatically released back to the system  without
       the need for another thread to join with the terminated thread.

       Attempting to detach an already detached thread results in unspecified behavior.

RETURN VALUE
       On success, pthread_detach() returns 0; on error, it returns an error number.
```

Concurrency

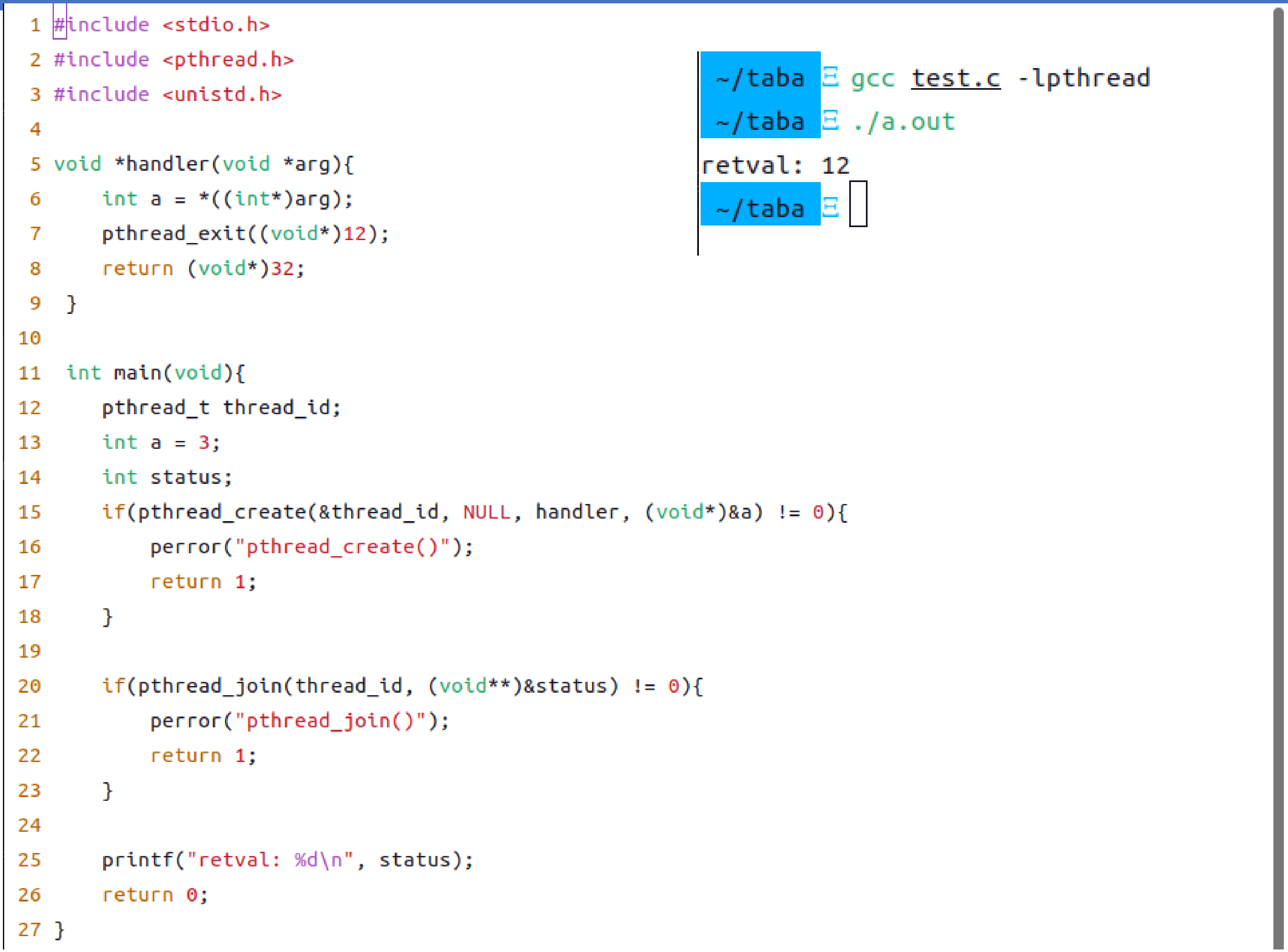
1. Concurrency



```
PTHREAD_EXIT(3)              Linux Programmer's Manual              PTHREAD_EXIT(3)

NAME
       pthread_exit - terminate calling thread

SYNOPSIS
       #include <pthread.h>

       void pthread_exit(void *retval);

       Compile and link with -pthread.

DESCRIPTION
       The  pthread_exit() function terminates the calling thread and returns a value via retval that
       (if the thread is joinable) is available to another thread in  the  same  process  that  calls
       pthread_join(3).
```

Concurrency

1. Concurrency

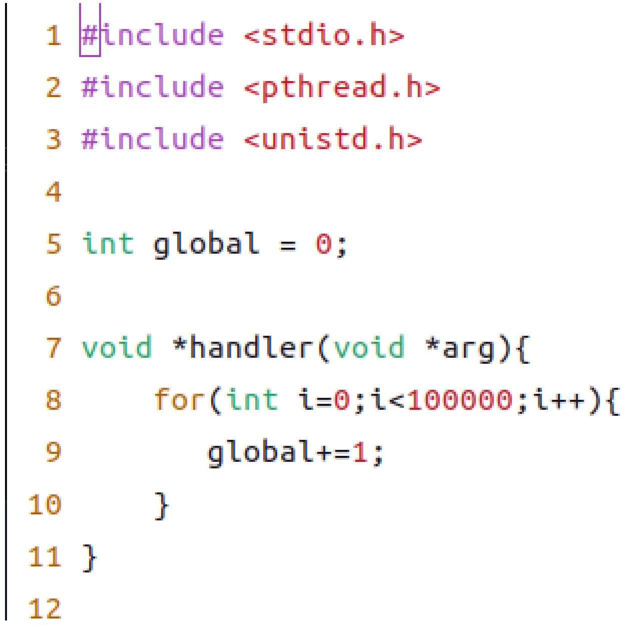


```c
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

void *handler(void *arg){
    int a = *((int*)arg);
    pthread_exit((void*)12);
    return (void*)32;
 }

 int main(void){
    pthread_t thread_id;
    int a = 3;
    int status;
    if(pthread_create(&thread_id, NULL, handler, (void*)&a) != 0){
        perror("pthread_create()");
        return 1;
    }

    if(pthread_join(thread_id, (void**)&status) != 0){
        perror("pthread_join()");
        return 1;
    }

    printf("retval: %d\n", status);
    return 0;
}
```

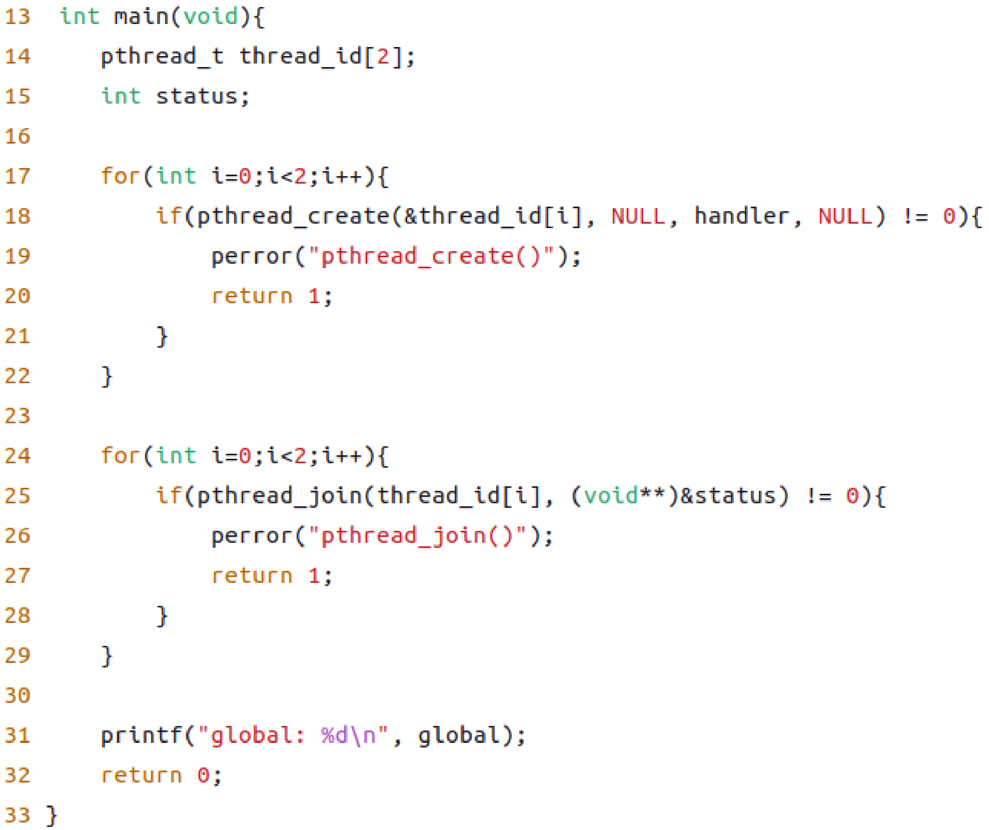
```
 ~/taba Ξ gcc test.c -lpthread
 ~/taba Ξ ./a.out
retval: 12
 ~/taba Ξ
```

Concurrency

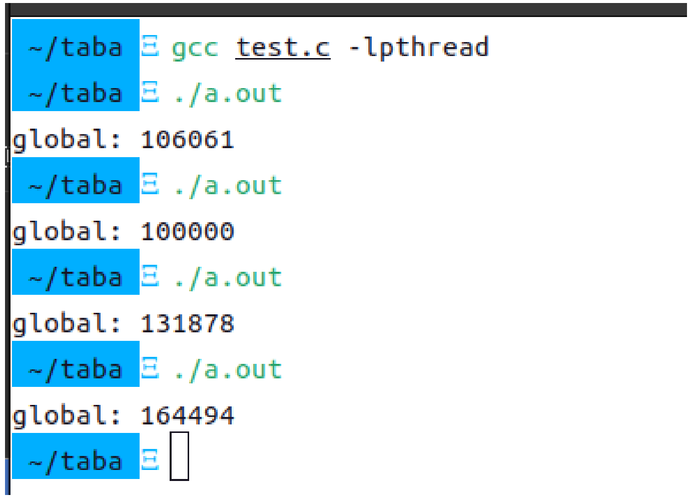
1. Concurrency



```c
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

int global = 0;

void *handler(void *arg){
    for(int i=0;i<100000;i++){
        global+=1;
    }
}

```



```c
int main(void){
    pthread_t thread_id[2];
    int status;

    for(int i=0;i<2;i++){
        if(pthread_create(&thread_id[i], NULL, handler, NULL) != 0){
            perror("pthread_create()");
            return 1;
        }
    }

    for(int i=0;i<2;i++){
        if(pthread_join(thread_id[i], (void**)&status) != 0){
            perror("pthread_join()");
            return 1;
        }
    }

    printf("global: %d\n", global);
    return 0;
}
```



```
 ~/taba Ξ gcc test.c -lpthread
 ~/taba Ξ ./a.out
global: 106061
 ~/taba Ξ ./a.out
global: 100000
 ~/taba Ξ ./a.out
global: 131878
 ~/taba Ξ ./a.out
global: 164494
 ~/taba Ξ 
```

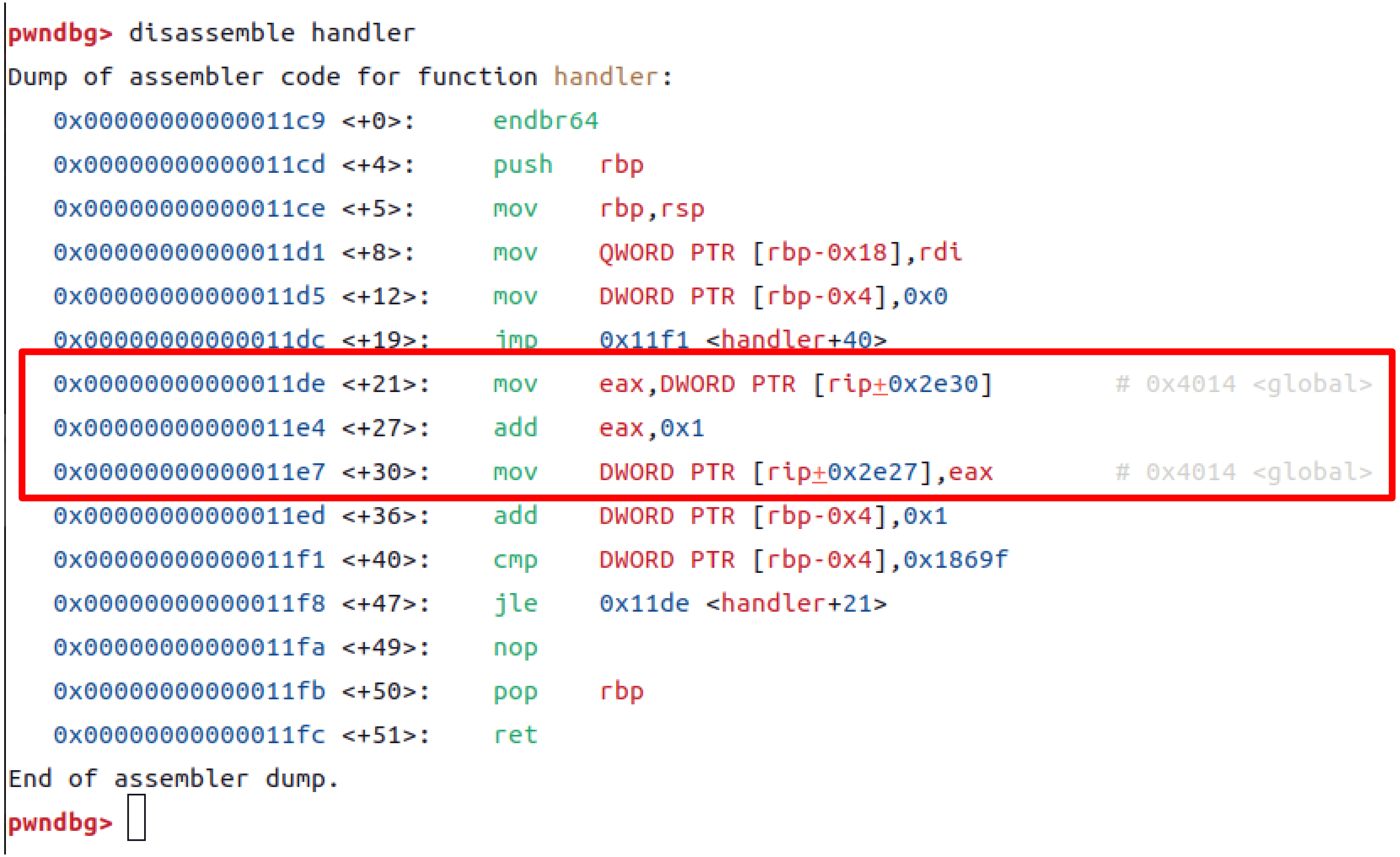
Why?

Concurrency

1. Concurrency

global += 1 assembly …

Have Three instructions

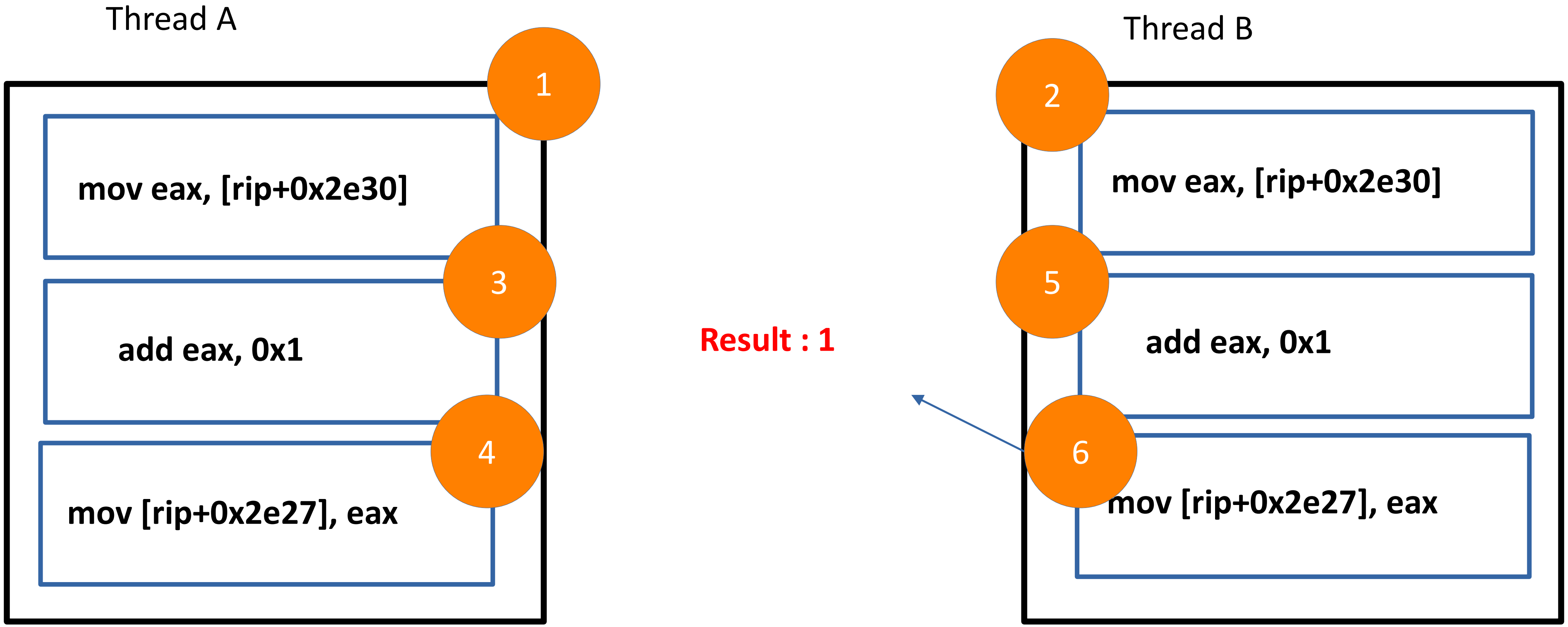


```
pwndbg> disassemble handler
Dump of assembler code for function handler:
   0x00000000000011c9 <+0>:     endbr64
   0x00000000000011cd <+4>:     push   rbp
   0x00000000000011ce <+5>:     mov    rbp,rsp
   0x00000000000011d1 <+8>:     mov    QWORD PTR [rbp-0x18],rdi
   0x00000000000011d5 <+12>:    mov    DWORD PTR [rbp-0x4],0x0
   0x00000000000011dc <+19>:    jmp    0x11f1 <handler+40>
   0x00000000000011de <+21>:    mov    eax,DWORD PTR [rip+0x2e30]        # 0x4014 <global>
   0x00000000000011e4 <+27>:    add    eax,0x1
   0x00000000000011e7 <+30>:    mov    DWORD PTR [rip+0x2e27],eax        # 0x4014 <global>
   0x00000000000011ed <+36>:    add    DWORD PTR [rbp-0x4],0x1
   0x00000000000011f1 <+40>:    cmp    DWORD PTR [rbp-0x4],0x1869f
   0x00000000000011f8 <+47>:    jle    0x11de <handler+21>
   0x00000000000011fa <+49>:    nop
   0x00000000000011fb <+50>:    pop    rbp
   0x00000000000011fc <+51>:    ret
End of assembler dump.
pwndbg>
```

Concurrency

1. Concurrency

Critical Section에 동시에 접근해서 생기는 문제!



Concurrency

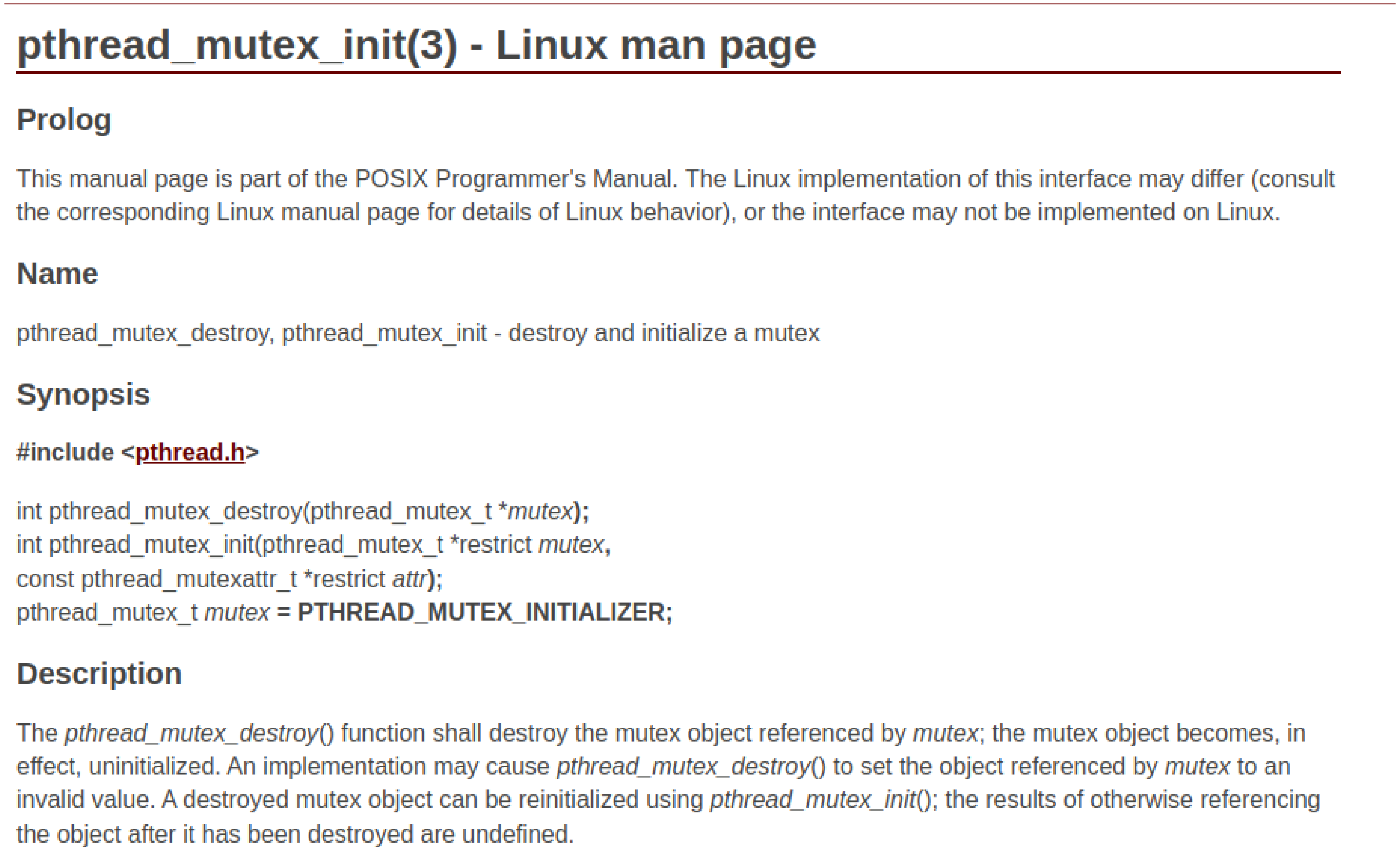
1. Concurrency

But, We Have Lock Mechanism!

pthread_mutex_init
pthread_mutex_destroy
pthread_mutex_lock
pthread_mutex_unlock

Concurrency

1. Concurrency



pthread_mutex_init(3) - Linux man page

Prolog

This manual page is part of the POSIX Programmer's Manual. The Linux implementation of this interface may differ (consult the corresponding Linux manual page for details of Linux behavior), or the interface may not be implemented on Linux.

Name

pthread_mutex_destroy, pthread_mutex_init - destroy and initialize a mutex

Synopsis

#include <pthread.h>

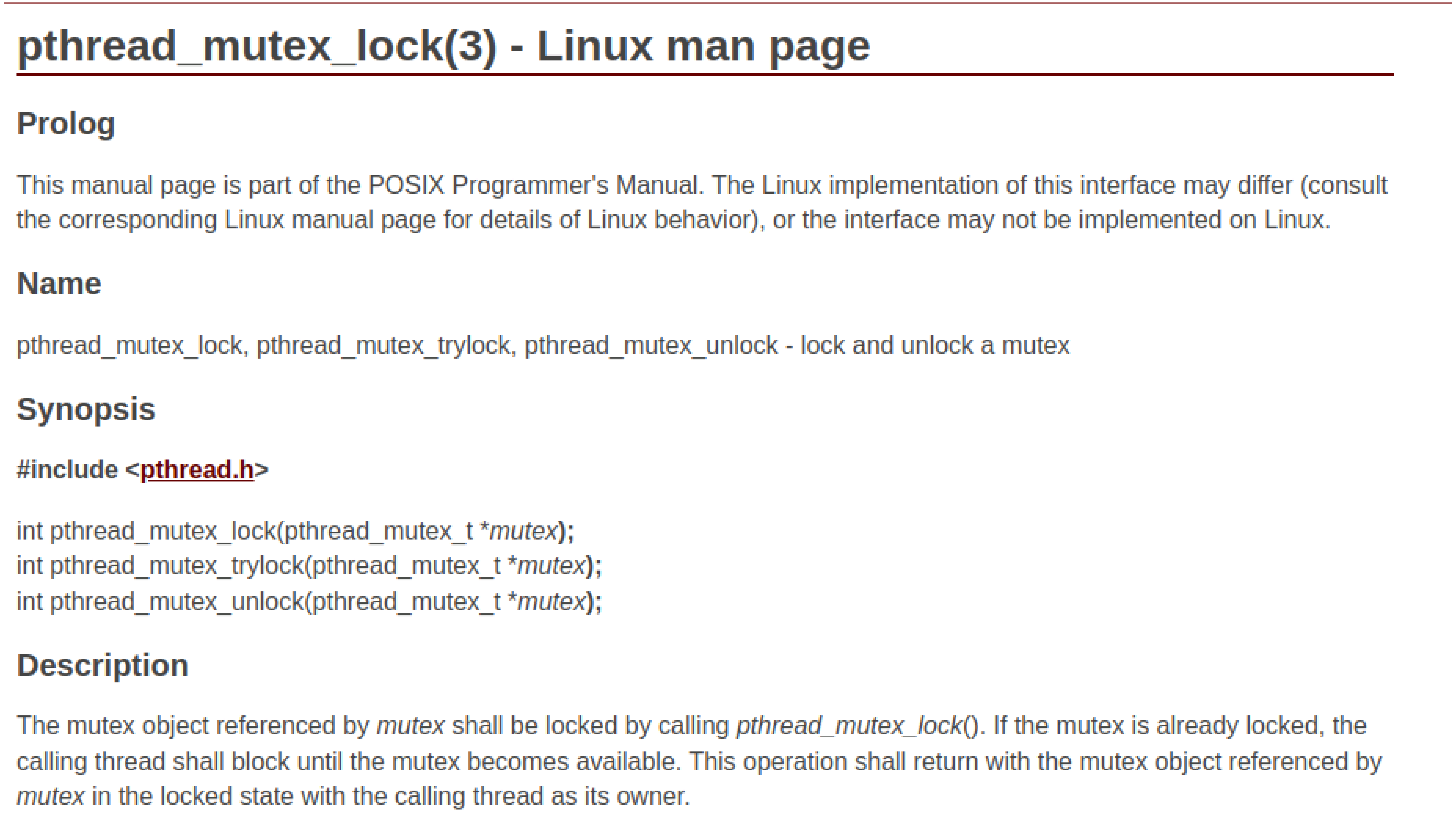
int pthread_mutex_destroy(pthread_mutex_t **mutex***);**
int pthread_mutex_init(pthread_mutex_t *restrict *mutex***,**
const pthread_mutexattr_t *restrict *attr***);**
pthread_mutex_t *mutex* **= PTHREAD_MUTEX_INITIALIZER;**

Description

The *pthread_mutex_destroy*() function shall destroy the mutex object referenced by *mutex*; the mutex object becomes, in effect, uninitialized. An implementation may cause *pthread_mutex_destroy*() to set the object referenced by *mutex* to an invalid value. A destroyed mutex object can be reinitialized using *pthread_mutex_init*(); the results of otherwise referencing the object after it has been destroyed are undefined.

Concurrency

1. Concurrency



pthread_mutex_lock(3) - Linux man page

Prolog

This manual page is part of the POSIX Programmer's Manual. The Linux implementation of this interface may differ (consult the corresponding Linux manual page for details of Linux behavior), or the interface may not be implemented on Linux.

Name

pthread_mutex_lock, pthread_mutex_trylock, pthread_mutex_unlock - lock and unlock a mutex

Synopsis

#include <pthread.h>

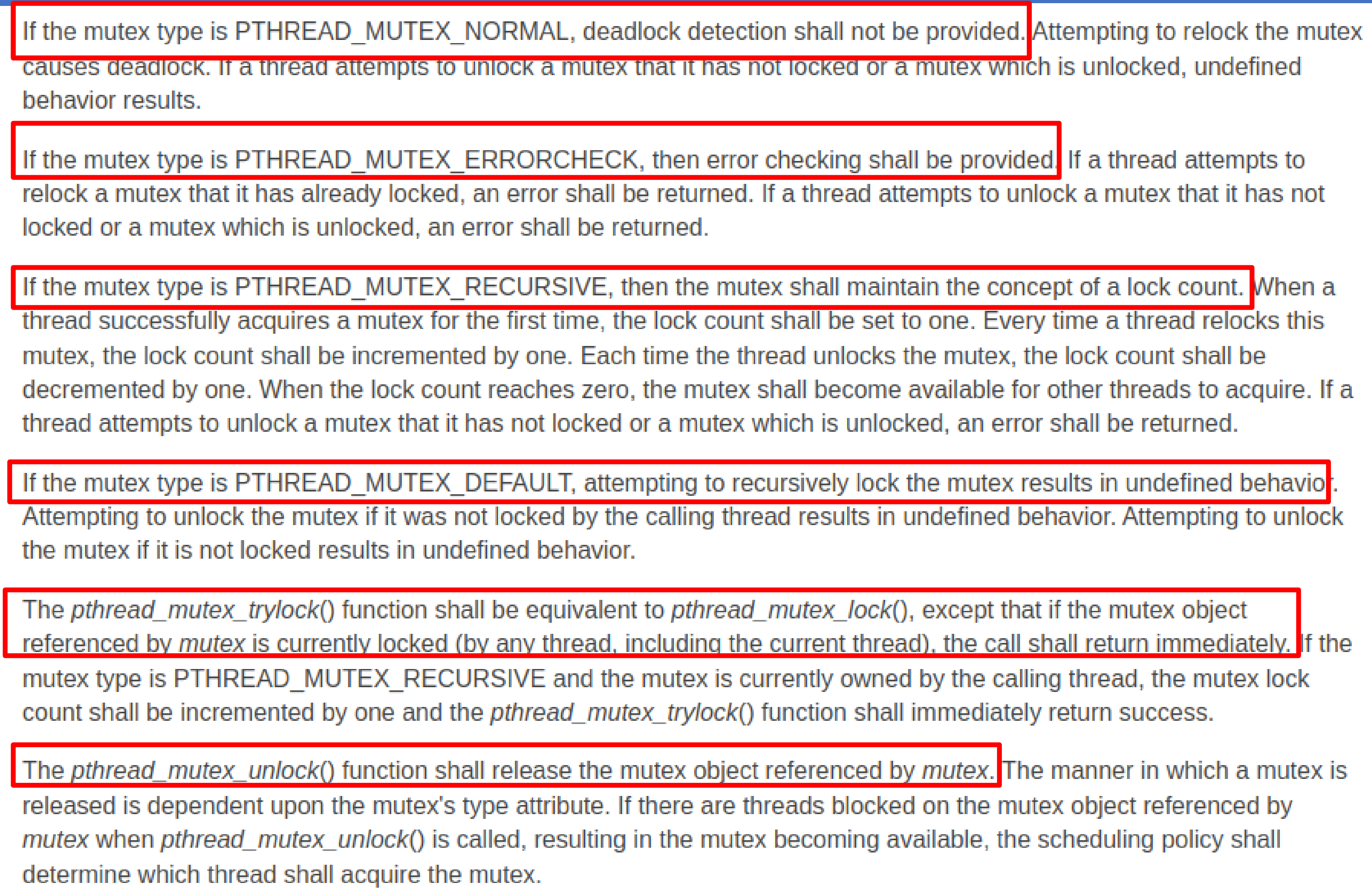
```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Description

The mutex object referenced by *mutex* shall be locked by calling *pthread_mutex_lock*(). If the mutex is already locked, the calling thread shall block until the mutex becomes available. This operation shall return with the mutex object referenced by *mutex* in the locked state with the calling thread as its owner.

Concurrency

1. Concurrency



If the mutex type is PTHREAD_MUTEX_NORMAL, deadlock detection shall not be provided. Attempting to relock the mutex causes deadlock. If a thread attempts to unlock a mutex that it has not locked or a mutex which is unlocked, undefined behavior results.

If the mutex type is PTHREAD_MUTEX_ERRORCHECK, then error checking shall be provided. If a thread attempts to relock a mutex that it has already locked, an error shall be returned. If a thread attempts to unlock a mutex that it has not locked or a mutex which is unlocked, an error shall be returned.

If the mutex type is PTHREAD_MUTEX_RECURSIVE, then the mutex shall maintain the concept of a lock count. When a thread successfully acquires a mutex for the first time, the lock count shall be set to one. Every time a thread relocks this mutex, the lock count shall be incremented by one. Each time the thread unlocks the mutex, the lock count shall be decremented by one. When the lock count reaches zero, the mutex shall become available for other threads to acquire. If a thread attempts to unlock a mutex that it has not locked or a mutex which is unlocked, an error shall be returned.

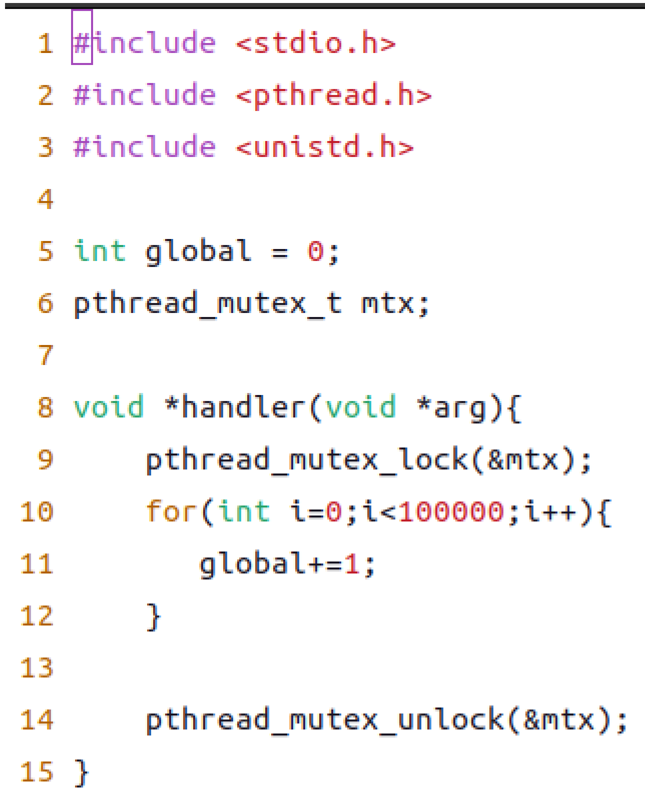
If the mutex type is PTHREAD_MUTEX_DEFAULT, attempting to recursively lock the mutex results in undefined behavior. Attempting to unlock the mutex if it was not locked by the calling thread results in undefined behavior. Attempting to unlock the mutex if it is not locked results in undefined behavior.

The *pthread_mutex_trylock*() function shall be equivalent to *pthread_mutex_lock*(), except that if the mutex object referenced by *mutex* is currently locked (by any thread, including the current thread), the call shall return immediately. If the mutex type is PTHREAD_MUTEX_RECURSIVE and the mutex is currently owned by the calling thread, the mutex lock count shall be incremented by one and the *pthread_mutex_trylock*() function shall immediately return success.

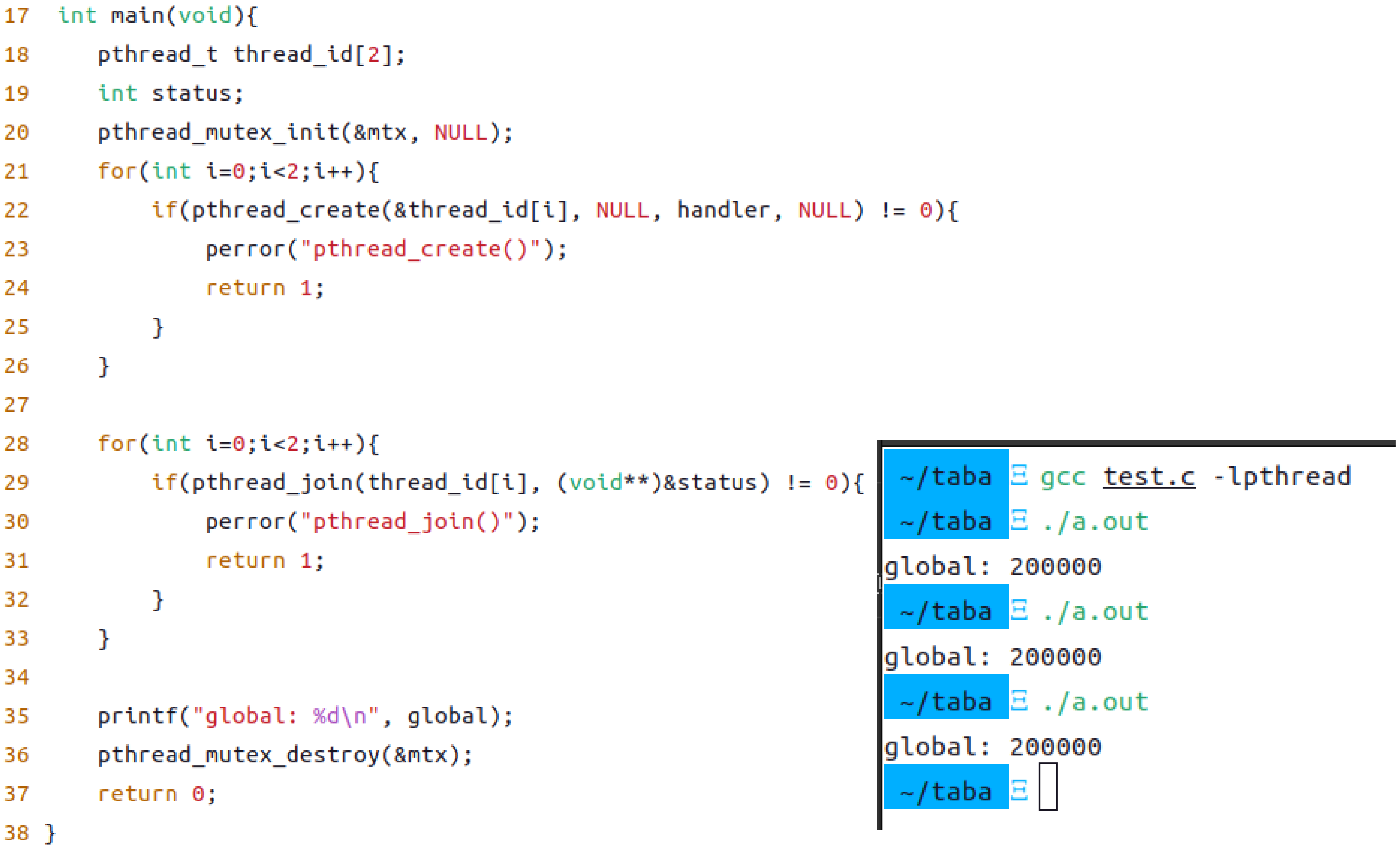
The *pthread_mutex_unlock*() function shall release the mutex object referenced by *mutex*. The manner in which a mutex is released is dependent upon the mutex's type attribute. If there are threads blocked on the mutex object referenced by *mutex* when *pthread_mutex_unlock*() is called, resulting in the mutex becoming available, the scheduling policy shall determine which thread shall acquire the mutex.

Concurrency

1. Concurrency



```c
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

int global = 0;
pthread_mutex_t mtx;

void *handler(void *arg){
    pthread_mutex_lock(&mtx);
    for(int i=0;i<100000;i++){
        global+=1;
    }

    pthread_mutex_unlock(&mtx);
}
```



```c
int main(void){
    pthread_t thread_id[2];
    int status;
    pthread_mutex_init(&mtx, NULL);
    for(int i=0;i<2;i++){
        if(pthread_create(&thread_id[i], NULL, handler, NULL) != 0){
            perror("pthread_create()");
            return 1;
        }
    }

    for(int i=0;i<2;i++){
        if(pthread_join(thread_id[i], (void**)&status) != 0){
            perror("pthread_join()");
            return 1;
        }
    }

    printf("global: %d\n", global);
    pthread_mutex_destroy(&mtx);
    return 0;
}
```

```
 ~/taba Ξ gcc test.c -lpthread
 ~/taba Ξ ./a.out
global: 200000
 ~/taba Ξ ./a.out
global: 200000
 ~/taba Ξ ./a.out
global: 200000
 ~/taba Ξ 
```

Concurrency

1. Concurrency

Lock Performance Problem

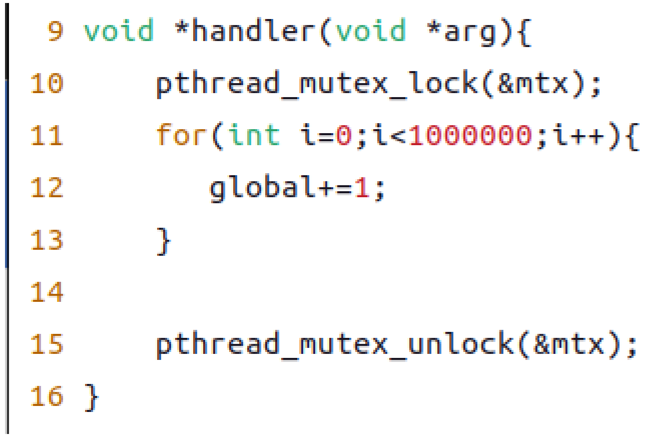
Lock을 어디에 배치해야 효율적일까 tradeoff

넓은 Critical Section? → Lock unLock 호출 횟수 감소 → 퍼포먼스 증가 → 다른 코드 실행 불가

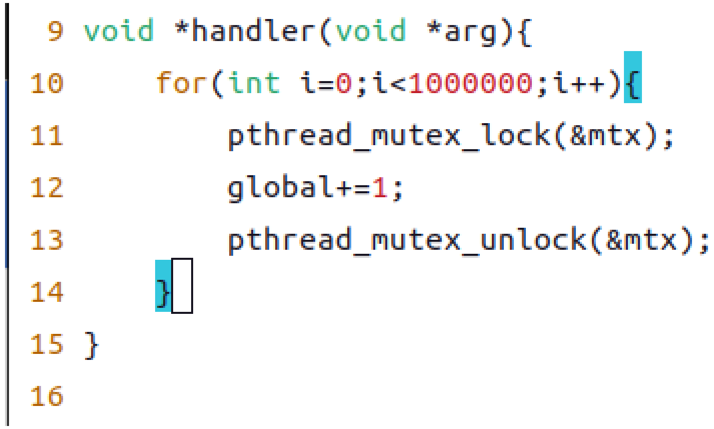
좁은 Critical Secrion? → Lock unLock 호출 횟수 증가 → 퍼포먼스 감소 → 다른 코드 실행 가능

Concurrency

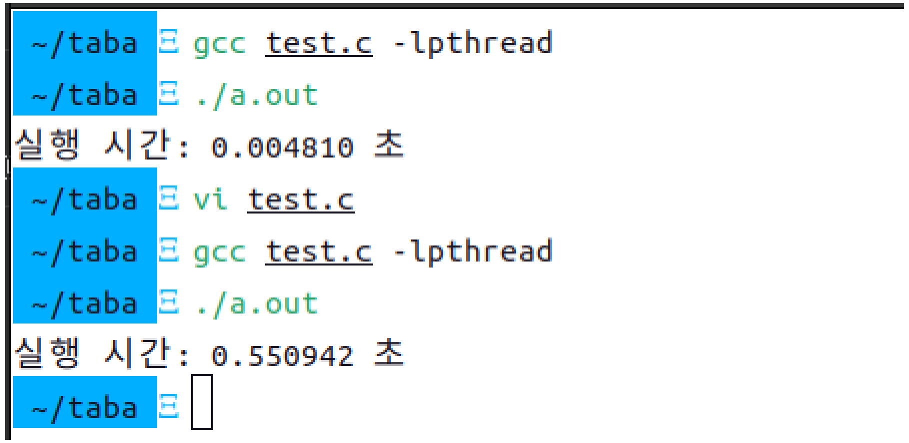
1. Concurrency



```
void *handler(void *arg){
    pthread_mutex_lock(&mtx);
    for(int i=0;i<1000000;i++){
        global+=1;
    }

    pthread_mutex_unlock(&mtx);
}
```



```
void *handler(void *arg){
    for(int i=0;i<1000000;i++){
        pthread_mutex_lock(&mtx);
        global+=1;
        pthread_mutex_unlock(&mtx);
    }
}

```



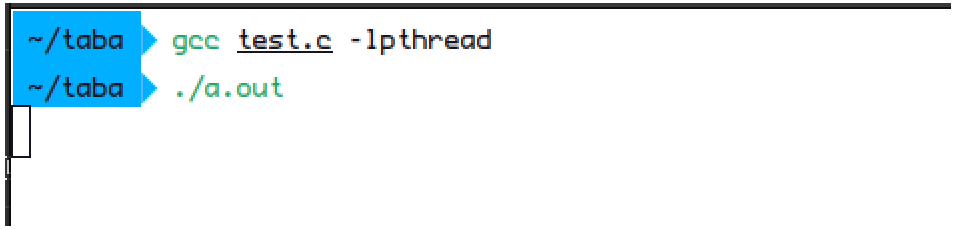
```
 ~/taba ㅌ gcc test.c -lpthread
 ~/taba ㅌ ./a.out
실행 시간: 0.004810 초
 ~/taba ㅌ vi test.c
 ~/taba ㅌ gcc test.c -lpthread
 ~/taba ㅌ ./a.out
실행 시간: 0.550942 초
 ~/taba ㅌ
```

Concurrency

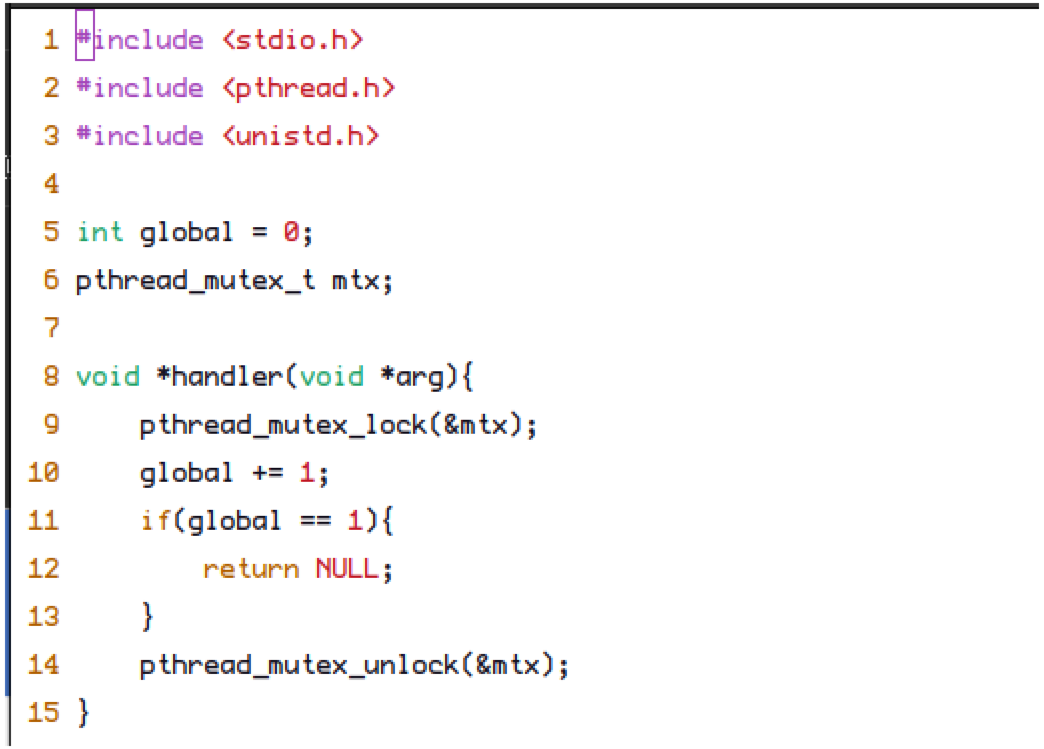
1. Concurrency

Deadlock 시나리오1

싸늘한 새벽에 일어난 당신, 심심함을 달래기위해 코드를 작성하는데 …



```
~/taba  gcc test.c -lpthread
~/taba  ./a.out
```



```c
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

int global = 0;
pthread_mutex_t mtx;

void *handler(void *arg){
    pthread_mutex_lock(&mtx);
    global += 1;
    if(global == 1){
        return NULL;
    }
    pthread_mutex_unlock(&mtx);
}
```

왜 프로그램이 종료되지 않는가!

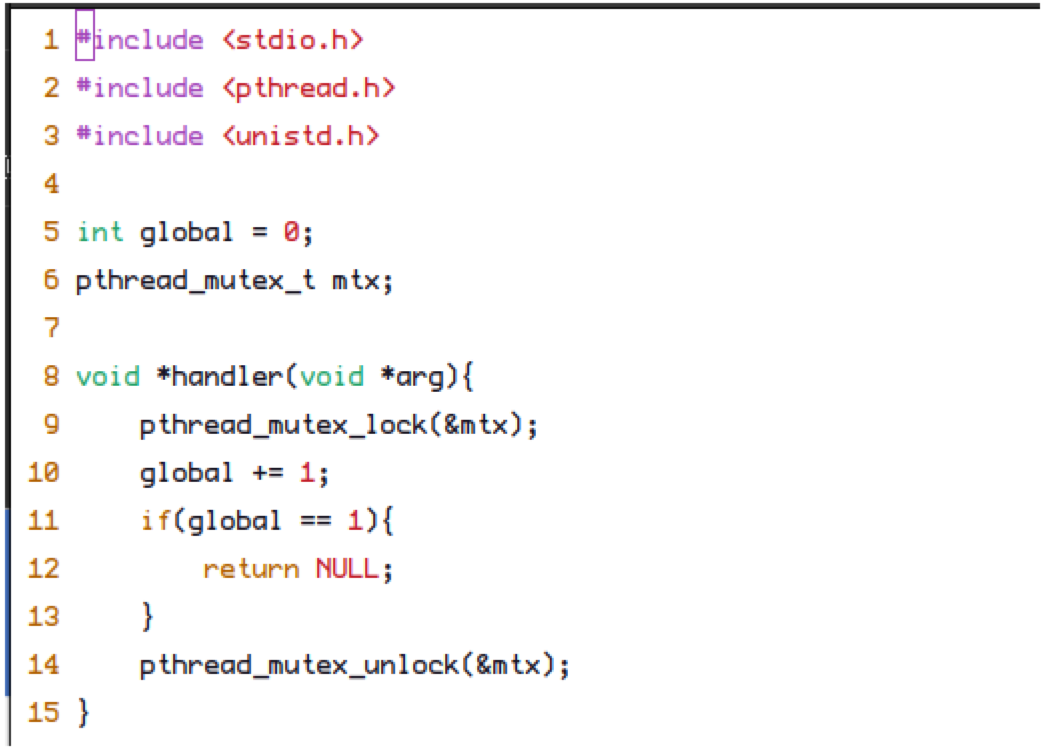
너무나도 골때리는 상황

Concurrency

1. Concurrency

mutex를 통해 Lock을 했으나 unlock을 하지 않고 함수를 종료...

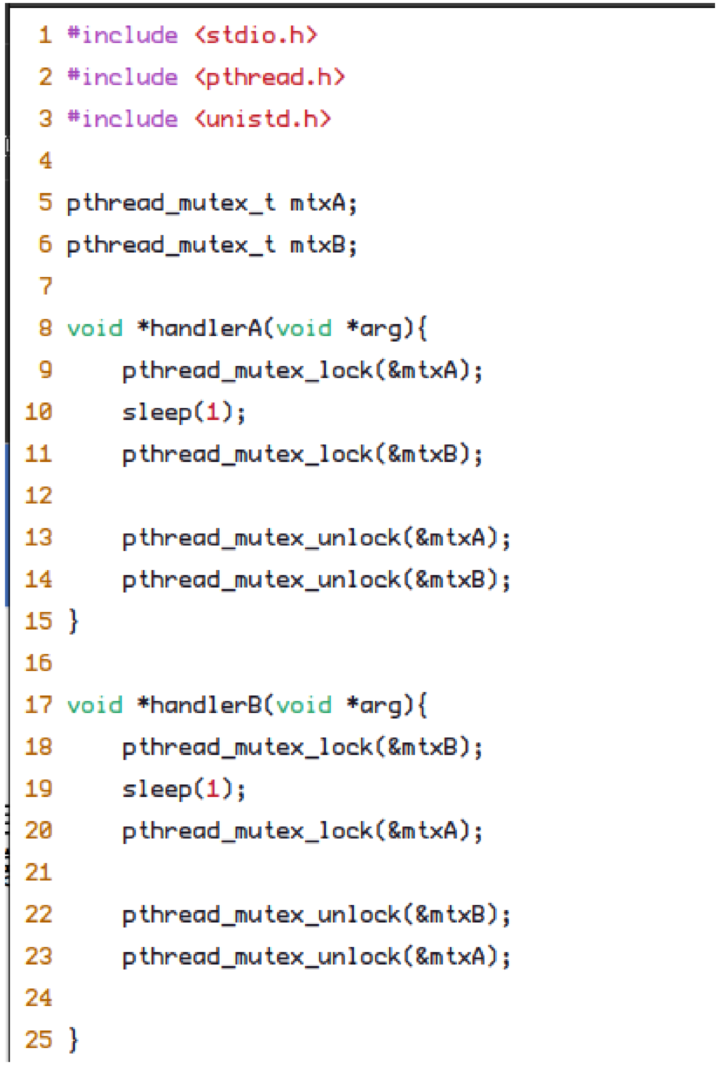
다른 스레드가 mutex를 참조했을 때
해당 mutex는 아직 lock되어있기에 무한 대기...



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

int global = 0;
pthread_mutex_t mtx;

void *handler(void *arg){
    pthread_mutex_lock(&mtx);
    global += 1;
    if(global == 1){
        return NULL;
    }
    pthread_mutex_unlock(&mtx);
}
```

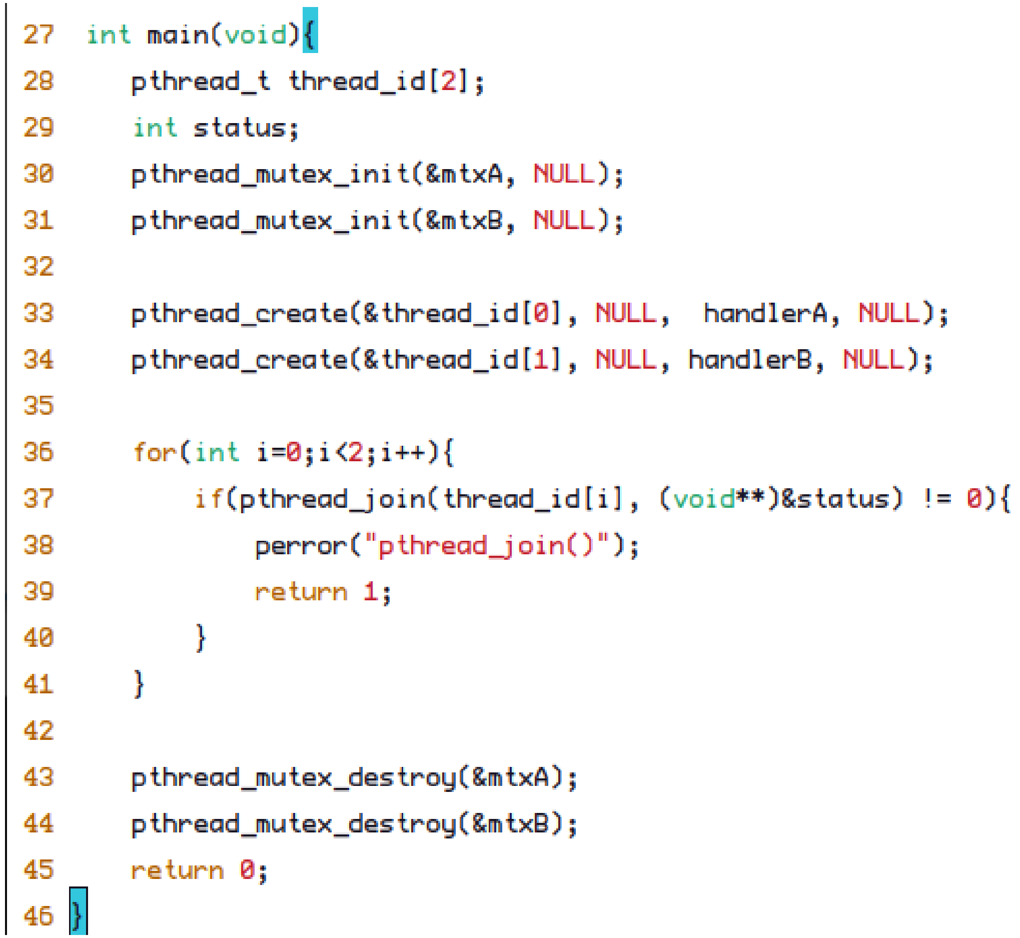
Concurrency

1. Concurrency



```c
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

pthread_mutex_t mtxA;
pthread_mutex_t mtxB;

void *handlerA(void *arg){
    pthread_mutex_lock(&mtxA);
    sleep(1);
    pthread_mutex_lock(&mtxB);

    pthread_mutex_unlock(&mtxA);
    pthread_mutex_unlock(&mtxB);
}

void *handlerB(void *arg){
    pthread_mutex_lock(&mtxB);
    sleep(1);
    pthread_mutex_lock(&mtxA);

    pthread_mutex_unlock(&mtxB);
    pthread_mutex_unlock(&mtxA);

}
```

서로가 서로의 mutex를 획득하려는 상황에서 자원의 겹침 발생



```c
int main(void){
    pthread_t thread_id[2];
    int status;
    pthread_mutex_init(&mtxA, NULL);
    pthread_mutex_init(&mtxB, NULL);

    pthread_create(&thread_id[0], NULL,  handlerA, NULL);
    pthread_create(&thread_id[1], NULL, handlerB, NULL);

    for(int i=0;i<2;i++){
        if(pthread_join(thread_id[i], (void**)&status) != 0){
            perror("pthread_join()");
            return 1;
        }
    }

    pthread_mutex_destroy(&mtxA);
    pthread_mutex_destroy(&mtxB);
    return 0;
}
```

Concurrency

1. Concurrency

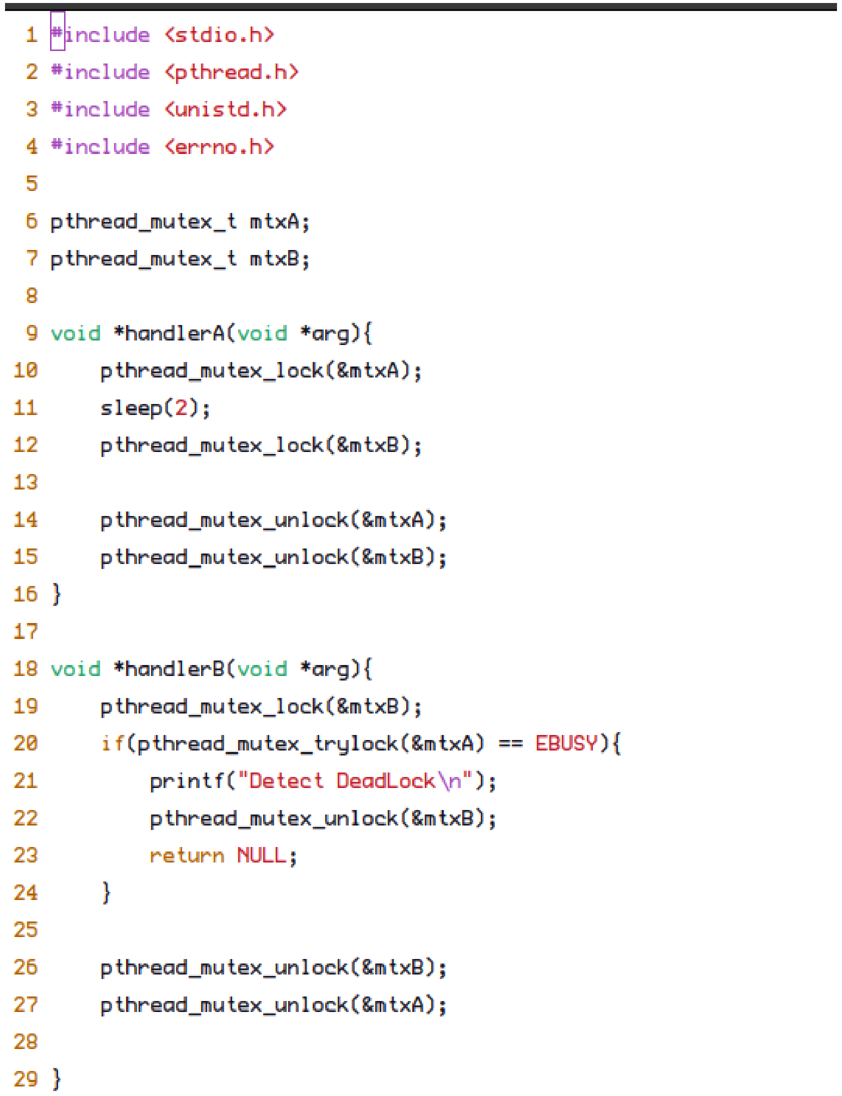
How to Prevent?

We can use ‘pthread_mutex_trylock’

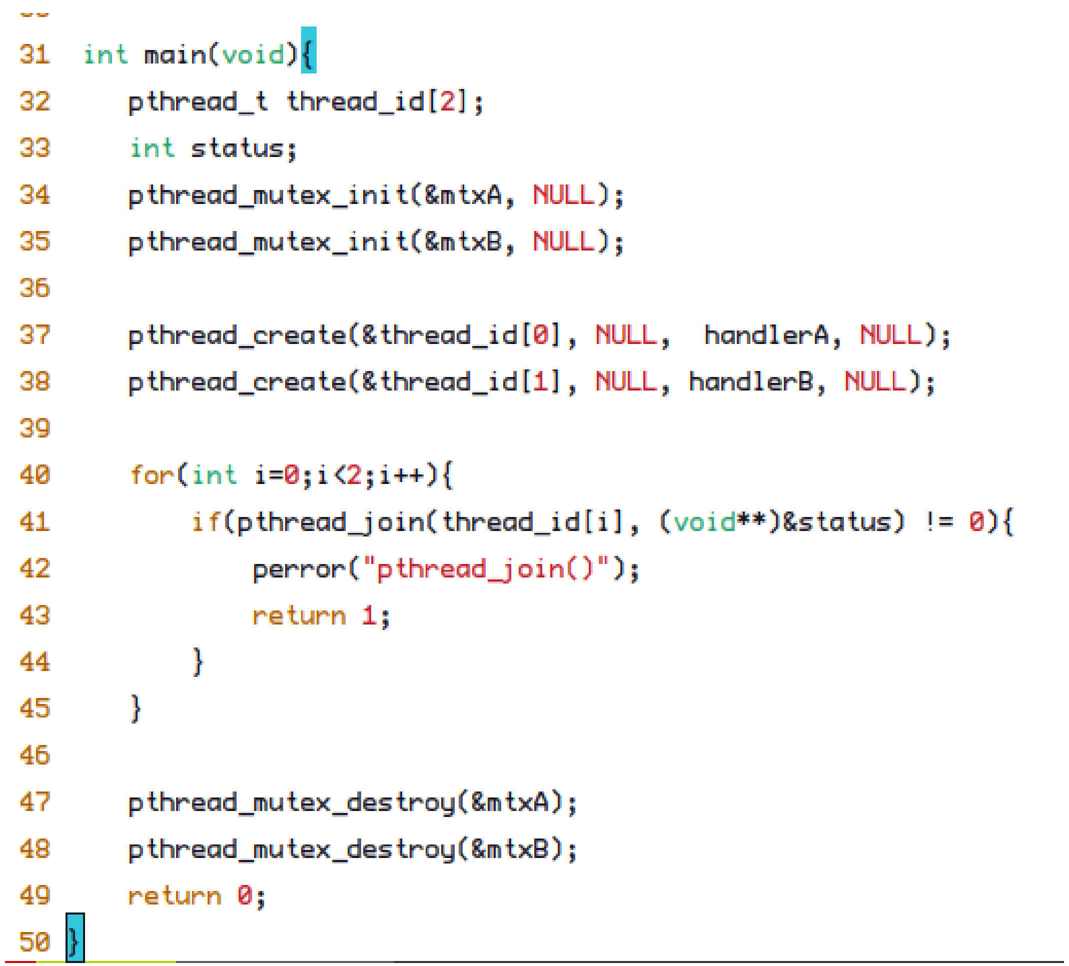
완벽한 해결책은 아니지만
해당 mutex가 lock상태인지 확인할 수 있음

Concurrency

1. Concurrency



```c
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#include <errno.h>

pthread_mutex_t mtxA;
pthread_mutex_t mtxB;

void *handlerA(void *arg){
    pthread_mutex_lock(&mtxA);
    sleep(2);
    pthread_mutex_lock(&mtxB);

    pthread_mutex_unlock(&mtxA);
    pthread_mutex_unlock(&mtxB);
}

void *handlerB(void *arg){
    pthread_mutex_lock(&mtxB);
    if(pthread_mutex_trylock(&mtxA) == EBUSY){
        printf("Detect DeadLock\n");
        pthread_mutex_unlock(&mtxB);
        return NULL;
    }

    pthread_mutex_unlock(&mtxB);
    pthread_mutex_unlock(&mtxA);

}
```



```c
int main(void){
    pthread_t thread_id[2];
    int status;
    pthread_mutex_init(&mtxA, NULL);
    pthread_mutex_init(&mtxB, NULL);

    pthread_create(&thread_id[0], NULL,  handlerA, NULL);
    pthread_create(&thread_id[1], NULL, handlerB, NULL);

    for(int i=0;i<2;i++){
        if(pthread_join(thread_id[i], (void**)&status) != 0){
            perror("pthread_join()");
            return 1;
        }
    }

    pthread_mutex_destroy(&mtxA);
    pthread_mutex_destroy(&mtxB);
    return 0;
}
```

Concurrency

1. Concurrency

Other Mechanism?

We can use semaphore, condition variable, ...

Concurrency

1. Concurrency

Conditional Variable

Mutex Lock and Unlock가 Concurrency 프로그래밍에서 유일하게 사용하는 것인가?

Mutex lock, unlock만으로 특정 변수가 Setup되었는지 어떻게 알것인가?

Lock, unlock을 반복적으로 사용해서 변수를 확인할것인가?
While loop로 busy wait할것인가?

이러한 문제를 어떻게 해결할까 → **Condition Variable**

1. Concurrency

Condition Variable은 말 그대로 조건 변수

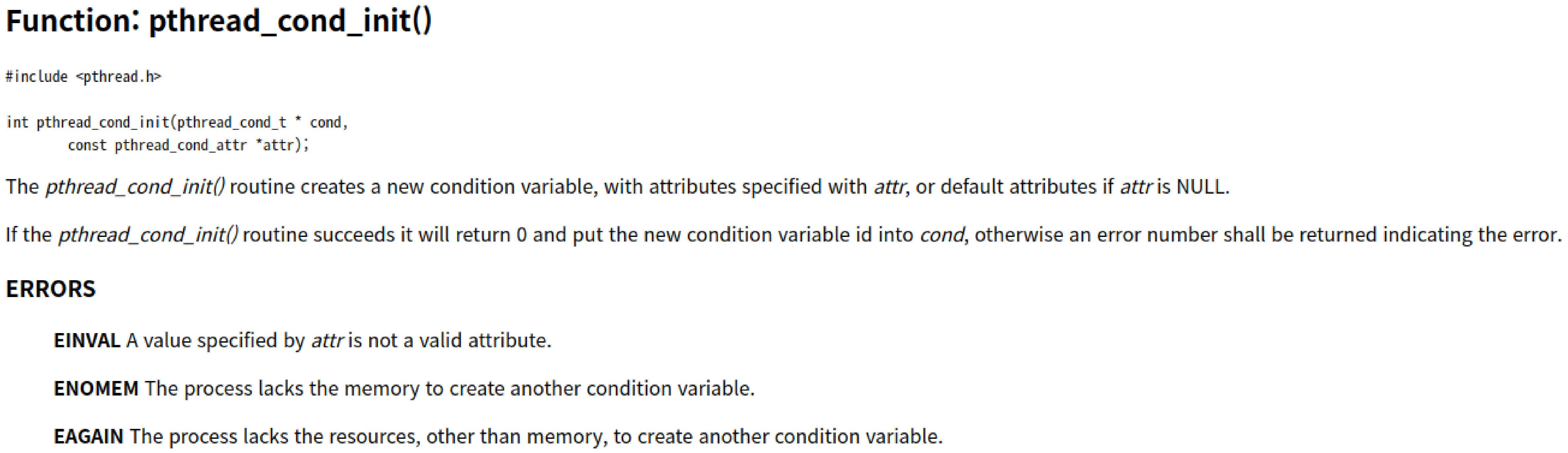
해당 조건 변수가 signal 알림을 탈 때까지 Wait

이 때 Wait은 CPU 자원을 소모하지 않음

Concurrency

1. Concurrency

How to use Condition Variable



Function: pthread_cond_init()

```
#include <pthread.h>

int pthread_cond_init(pthread_cond_t * cond,
        const pthread_cond_attr *attr);
```

The *pthread_cond_init()* routine creates a new condition variable, with attributes specified with *attr*, or default attributes if *attr* is NULL.

If the *pthread_cond_init()* routine succeeds it will return 0 and put the new condition variable id into *cond*, otherwise an error number shall be returned indicating the error.

ERRORS

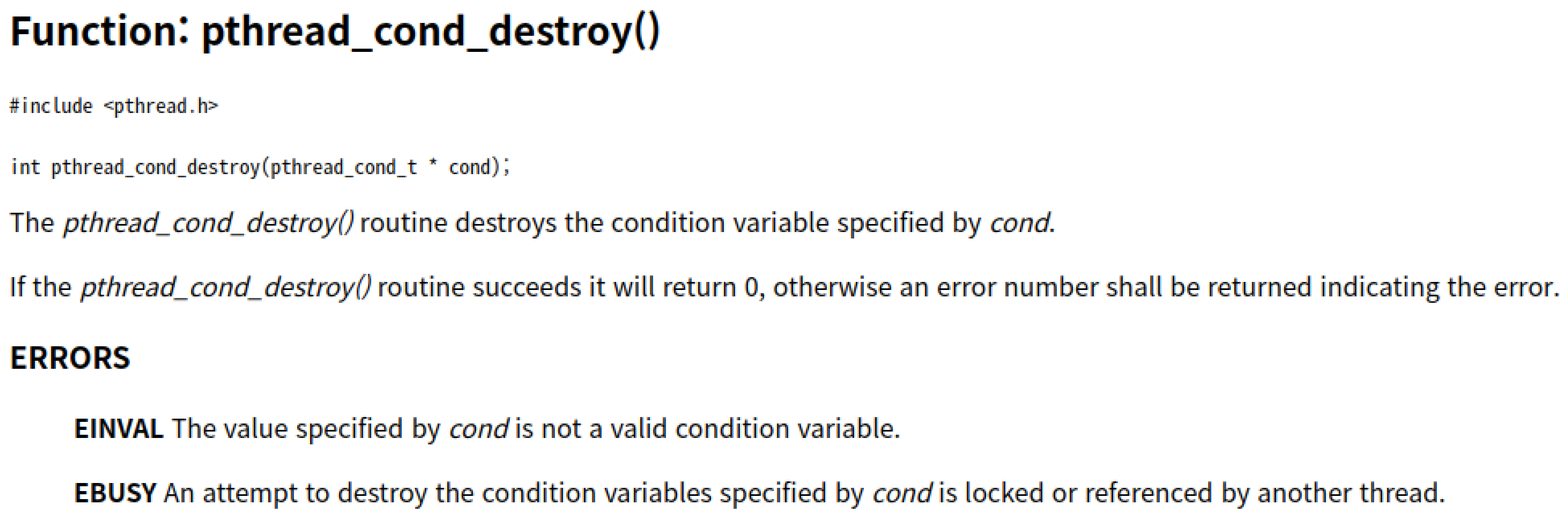
EINVAL A value specified by *attr* is not a valid attribute.

ENOMEM The process lacks the memory to create another condition variable.

EAGAIN The process lacks the resources, other than memory, to create another condition variable.

Concurrency

1. Concurrency



Function: pthread_cond_destroy()

```
#include <pthread.h>

int pthread_cond_destroy(pthread_cond_t * cond);
```

The *pthread_cond_destroy()* routine destroys the condition variable specified by *cond*.

If the *pthread_cond_destroy()* routine succeeds it will return 0, otherwise an error number shall be returned indicating the error.

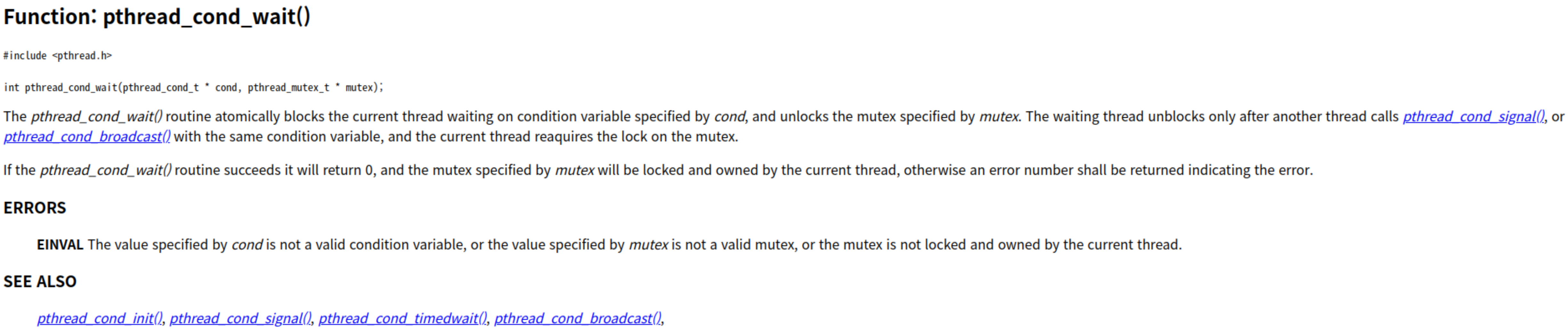
ERRORS

EINVAL The value specified by *cond* is not a valid condition variable.

EBUSY An attempt to destroy the condition variables specified by *cond* is locked or referenced by another thread.

Concurrency

1. Concurrency



Function: pthread_cond_wait()

```
#include <pthread.h>

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

The *pthread_cond_wait()* routine atomically blocks the current thread waiting on condition variable specified by *cond*, and unlocks the mutex specified by *mutex*. The waiting thread unblocks only after another thread calls *pthread_cond_signal()*, or *pthread_cond_broadcast()* with the same condition variable, and the current thread reaquires the lock on the mutex.

If the *pthread_cond_wait()* routine succeeds it will return 0, and the mutex specified by *mutex* will be locked and owned by the current thread, otherwise an error number shall be returned indicating the error.

ERRORS

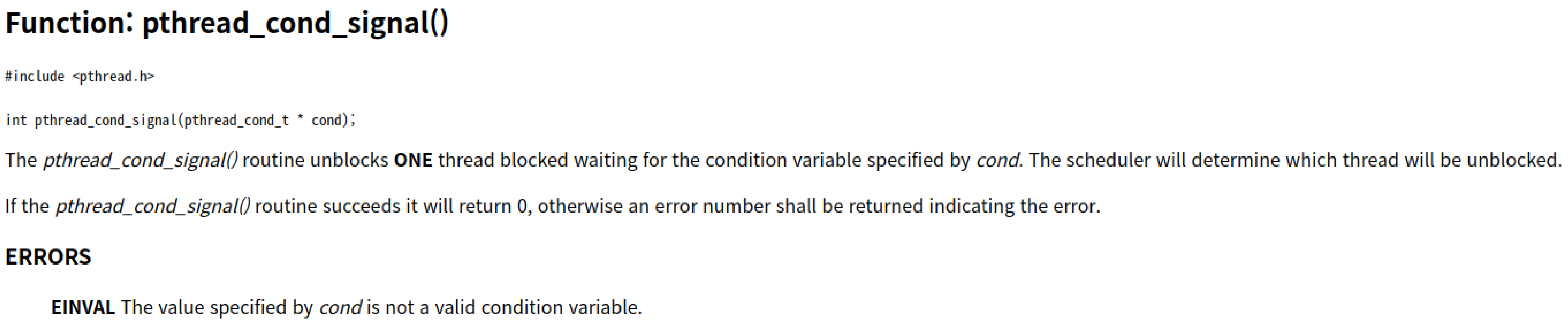
EINVAL The value specified by *cond* is not a valid condition variable, or the value specified by *mutex* is not a valid mutex, or the mutex is not locked and owned by the current thread.

SEE ALSO

pthread_cond_init(), *pthread_cond_signal()*, *pthread_cond_timedwait()*, *pthread_cond_broadcast()*,

Concurrency

1. Concurrency



Function: pthread_cond_signal()

```
#include <pthread.h>

int pthread_cond_signal(pthread_cond_t * cond);
```

The *pthread_cond_signal()* routine unblocks **ONE** thread blocked waiting for the condition variable specified by *cond*. The scheduler will determine which thread will be unblocked.

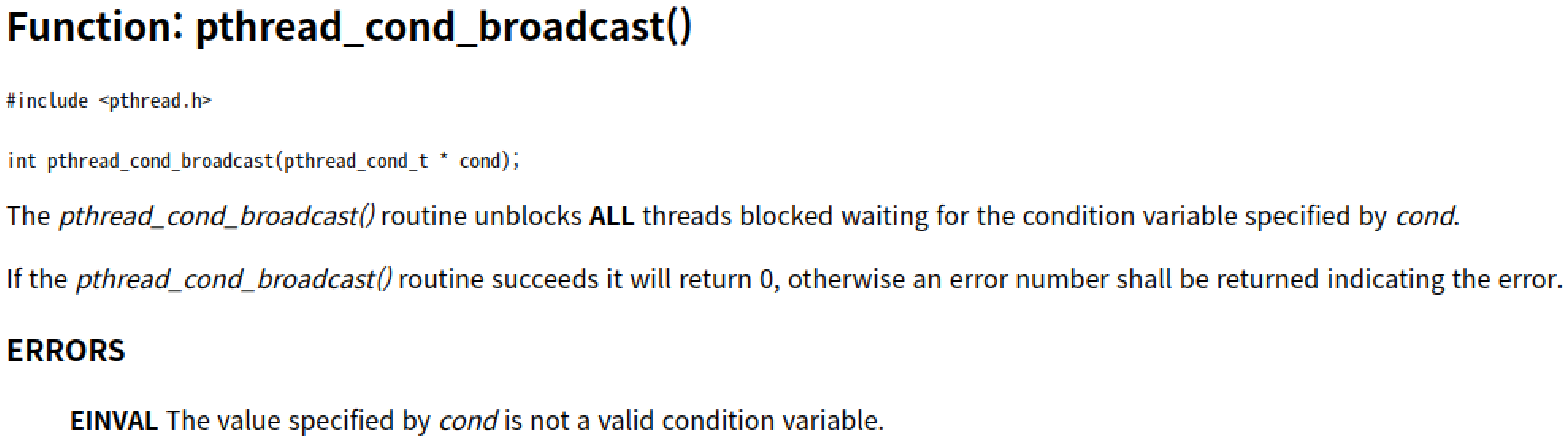
If the *pthread_cond_signal()* routine succeeds it will return 0, otherwise an error number shall be returned indicating the error.

ERRORS

EINVAL The value specified by *cond* is not a valid condition variable.

Concurrency

1. Concurrency



Function: pthread_cond_broadcast()

```
#include <pthread.h>

int pthread_cond_broadcast(pthread_cond_t * cond);
```

The *pthread_cond_broadcast()* routine unblocks **ALL** threads blocked waiting for the condition variable specified by *cond*.

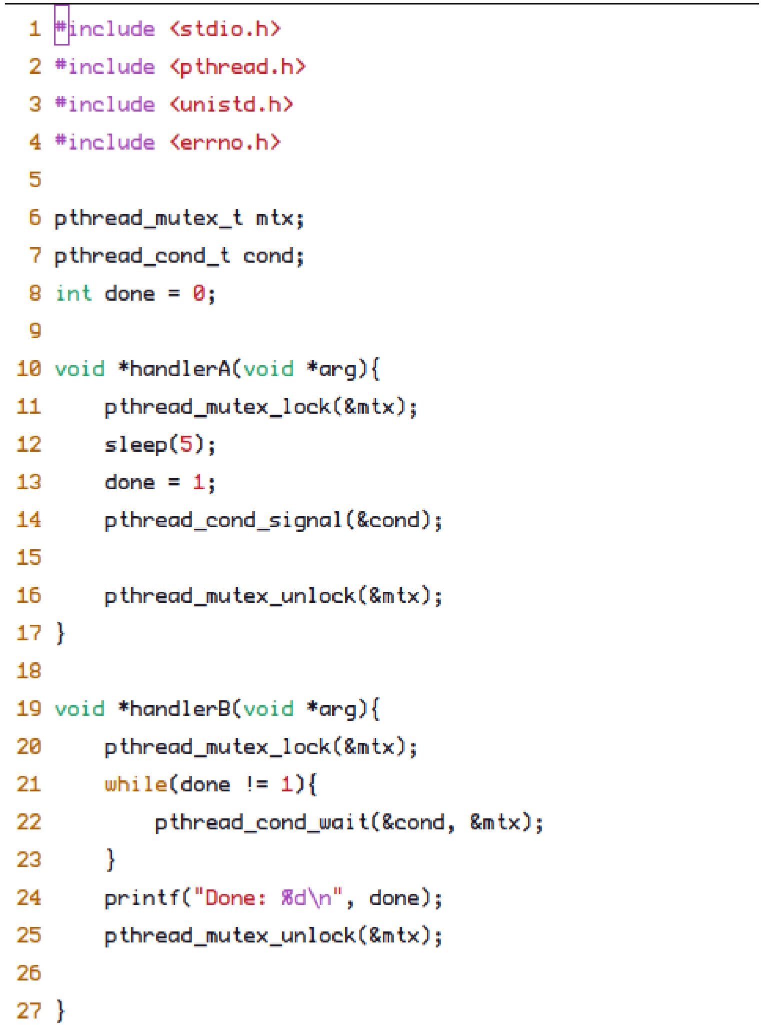
If the *pthread_cond_broadcast()* routine succeeds it will return 0, otherwise an error number shall be returned indicating the error.

ERRORS

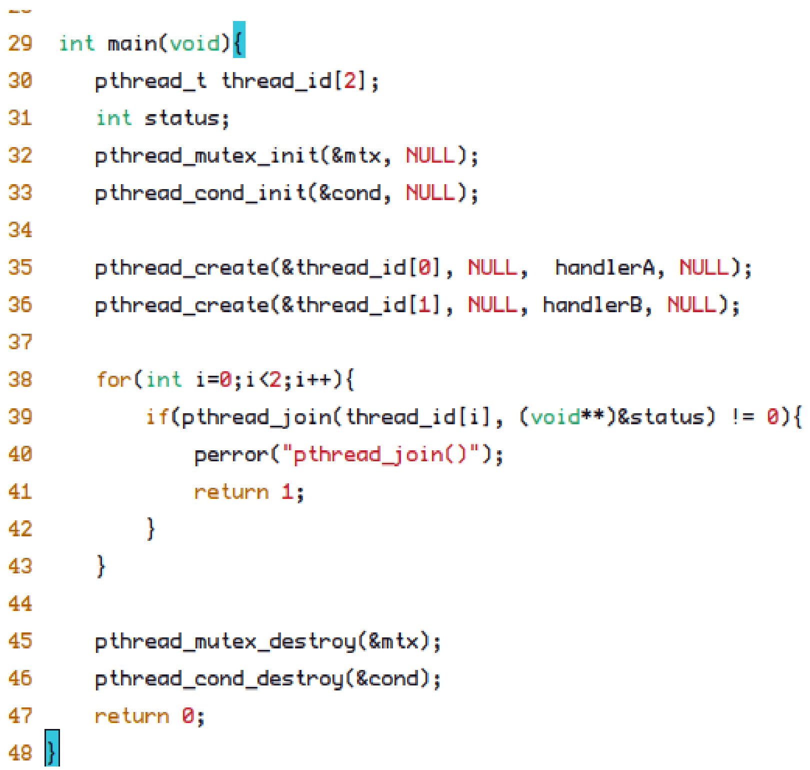
EINVAL The value specified by *cond* is not a valid condition variable.

Concurrency

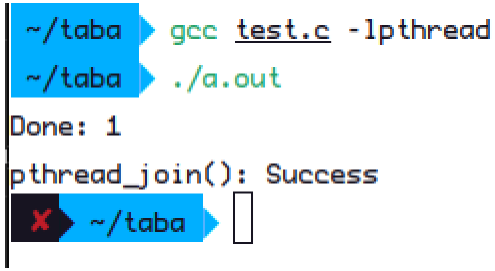
1. Concurrency



```c
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#include <errno.h>

pthread_mutex_t mtx;
pthread_cond_t cond;
int done = 0;

void *handlerA(void *arg){
    pthread_mutex_lock(&mtx);
    sleep(5);
    done = 1;
    pthread_cond_signal(&cond);

    pthread_mutex_unlock(&mtx);
}

void *handlerB(void *arg){
    pthread_mutex_lock(&mtx);
    while(done != 1){
        pthread_cond_wait(&cond, &mtx);
    }
    printf("Done: %d\n", done);
    pthread_mutex_unlock(&mtx);

}
```



```c
int main(void){
    pthread_t thread_id[2];
    int status;
    pthread_mutex_init(&mtx, NULL);
    pthread_cond_init(&cond, NULL);

    pthread_create(&thread_id[0], NULL,  handlerA, NULL);
    pthread_create(&thread_id[1], NULL, handlerB, NULL);

    for(int i=0;i<2;i++){
        if(pthread_join(thread_id[i], (void**)&status) != 0){
            perror("pthread_join()");
            return 1;
        }
    }

    pthread_mutex_destroy(&mtx);
    pthread_cond_destroy(&cond);
    return 0;
}
```



```
~/taba  gcc test.c -lpthread
~/taba  ./a.out
Done: 1
pthread_join(): Success
✘  ~/taba
```

Concurrency

1. Concurrency

Why we use while loop condition?

Why not use if condition?

Because of **Spurious wakeup**

Concurrency

Question