# Operating System (CSC 3150)

## Tutorial 3

*Lele Li:*
*221019053@link.cuhk.edu.cn*

# Target

In this tutorial, we will practice **Pthread** programming using c/c++.

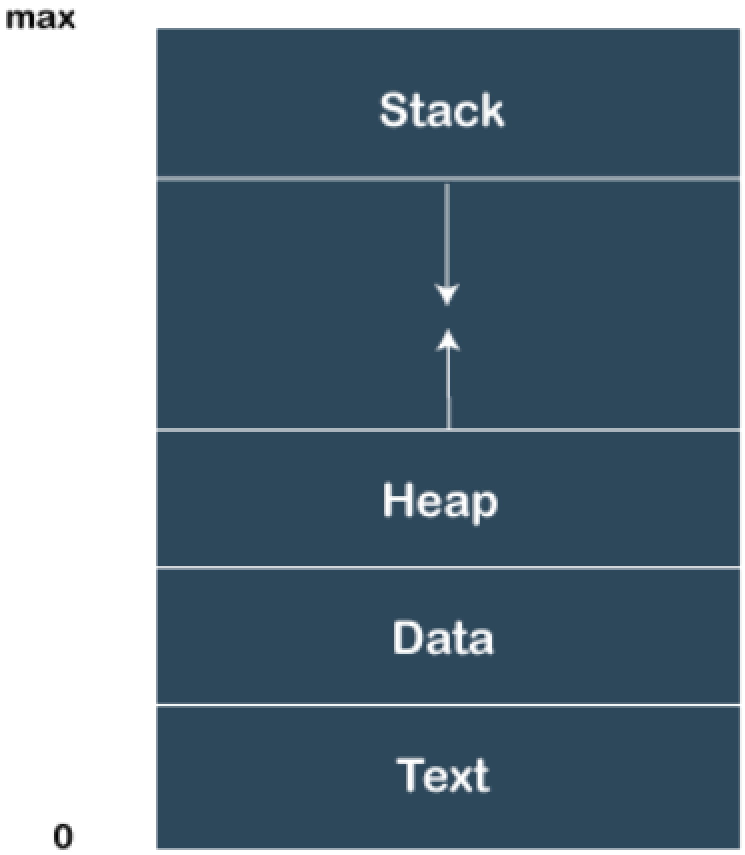
- Process
- Thread
- Pthread creation
- Pthread termination
- Pthread join
- Pthread mutex
- Pthread condition

# What is Process?

- A process is **an instance of a program** that is being executed.

### Features of Process

- Each time we create a process, we need to make a separate system call for each process to the OS. The **fork()** function creates the process.
- **Each process exists within its own address or memory space**.
- Each process **is independent and treated as an isolated process by the OS.**
- Processes need IPC (Inter-process Communication) in order to communicate with each other.

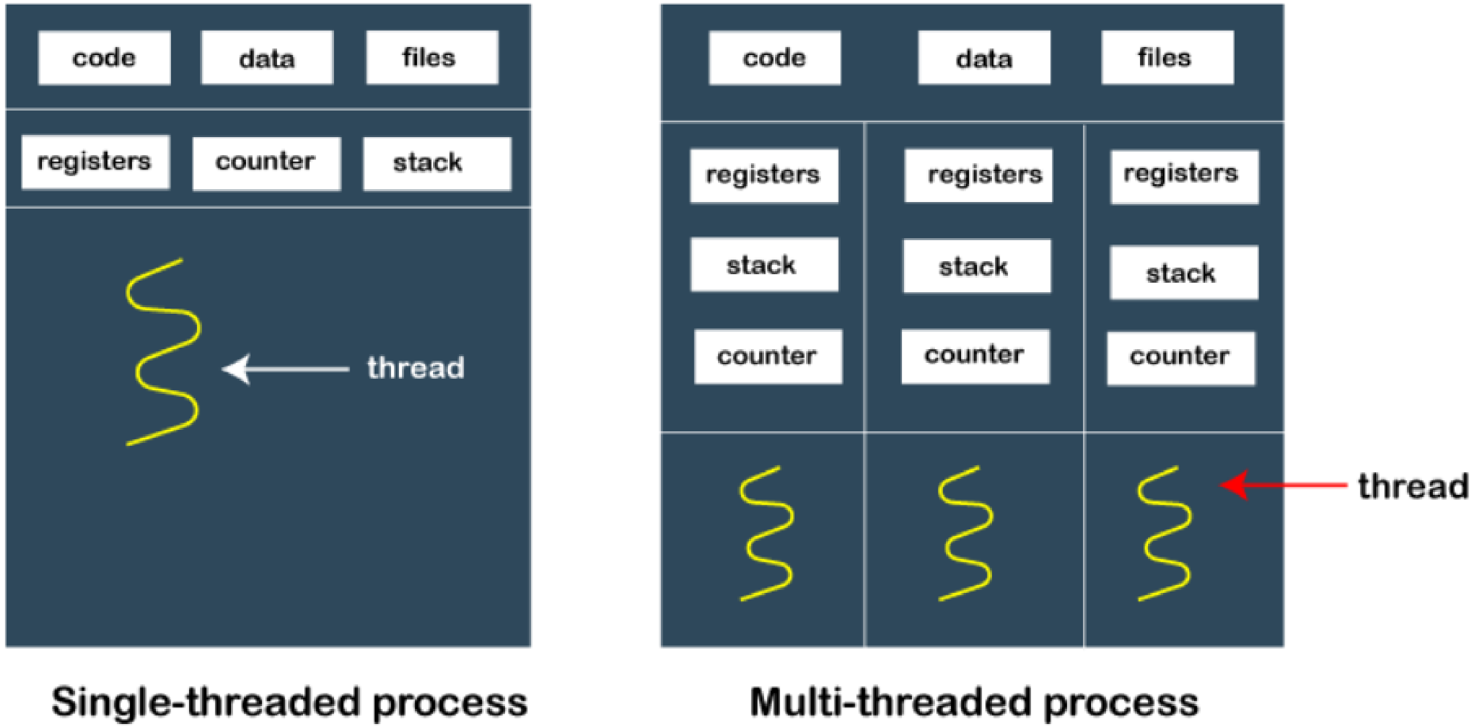


Process in the Memory

**What is a thread?**

**A thread is a single sequence stream within a process**. Because threads have some of the properties of processes, they are sometimes called *lightweight processes*.

- Threads are not independent from each other unlike processes. As a result, **threads shares with other threads their code section, data section and OS resources like open files and signals**. But, like processes, a thread has its own program counter (PC), a register set, and a stack space.
- Threads use and exist within these process resources. They are able to be scheduled by the operating system and run as independent entities within a process.
- **A process can have multiple threads**, all of which share the resources within a process and all of which execute within the same address space.

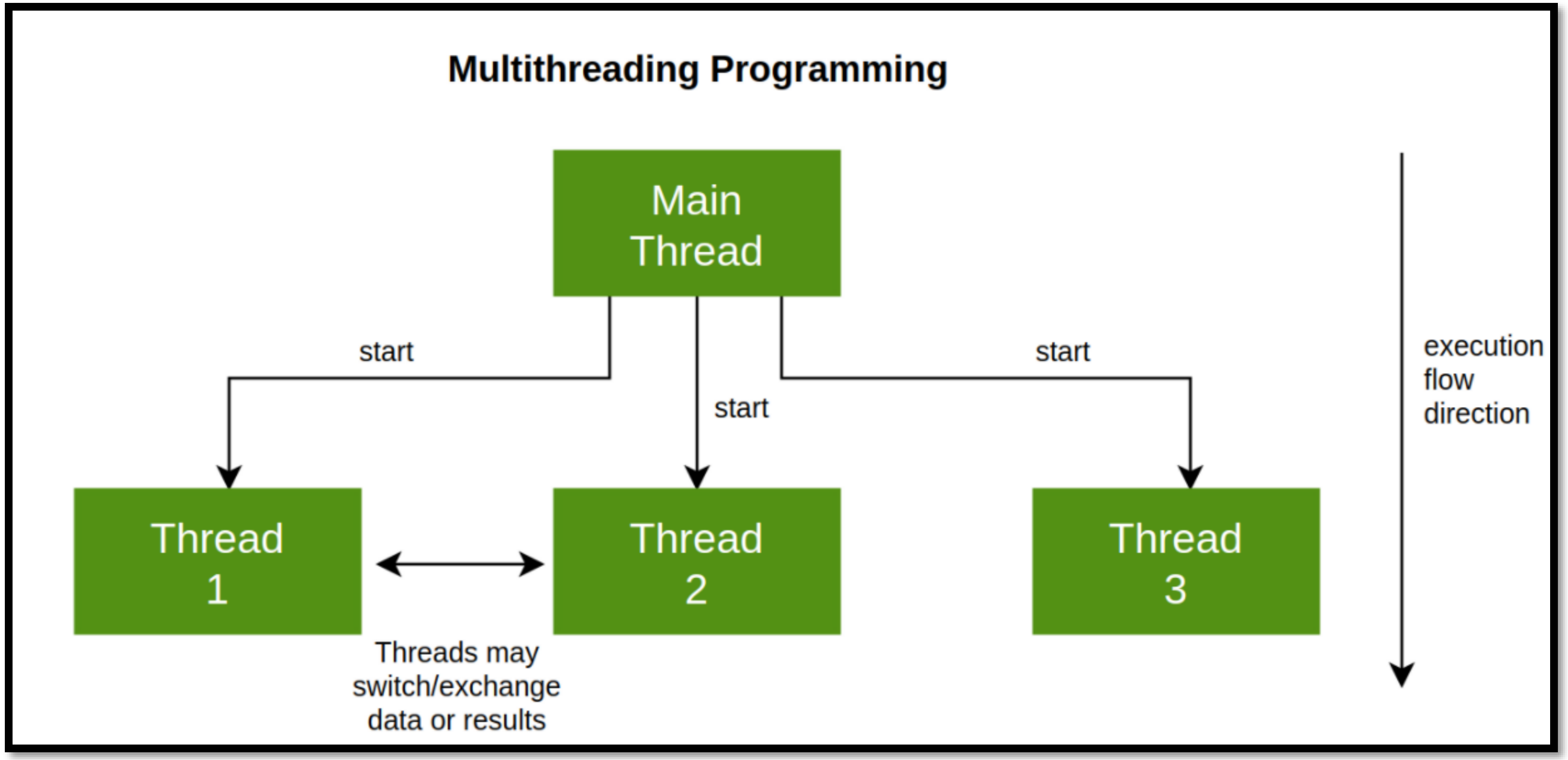


**Why Multithreading?**

Threads are popular way to improve application through parallelism. For example, in a browser, multiple tabs can be different threads.

Threads operate faster than processes due to following reasons:

1) Thread creation is much faster.
2) Context switching between threads is much faster.
3) Threads can be terminated easily
4) Communication between threads is faster.



## Can we write multithreading programs in C?

Unlike Java, multithreading is not supported by the C language standard. POSIX Threads (or Pthreads) is a POSIX standard for threads. Implementation of pthread is available with gcc compiler.

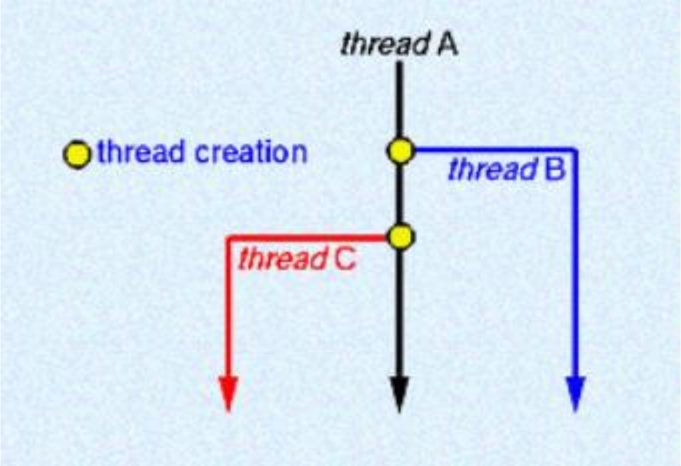
- To the software developer, the concept of a "procedure" that runs independently from its main program may best describe a thread.

- Pthread: POSIX Thread, a standard-based thread API for C.
- other options: openMP, std::thread

- When compiling Pthread in gcc/g++, should add option “-lpthread”.
  - Compile: gcc test.c –lpthread or g++ test.cpp -lpthread
  - Execution: ./a.out

# Pthread creation

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



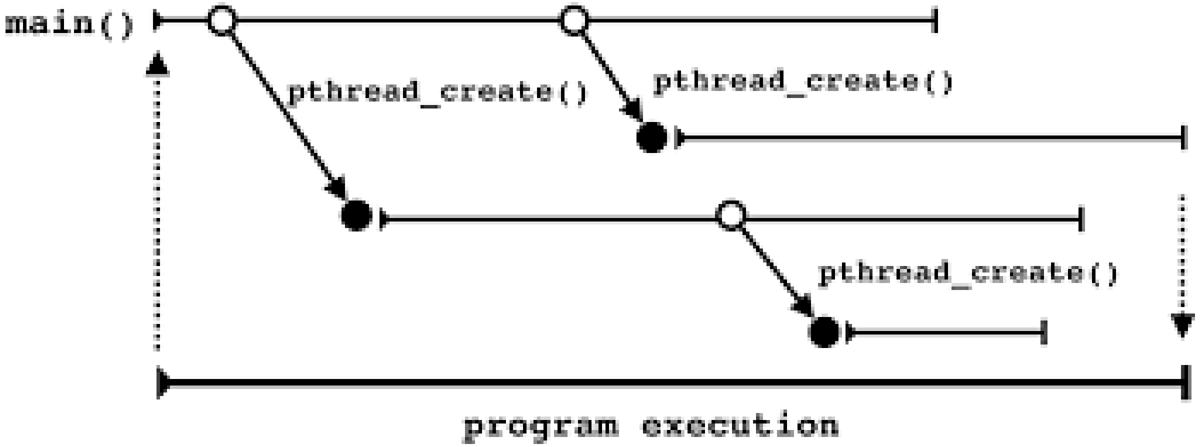
- **thread:** pointer to an unsigned integer value that returns the thread id of the thread created.
- The **attr** parameter is used to set thread attributes. You can specify a thread attributes object like scheduling policy, detached state, etc. Set NULL by default.

- The **start_routine** is the C routine that the thread will execute once it is created.
- **arg**: pointer to void that contains the arguments to the function defined in the earlier argument

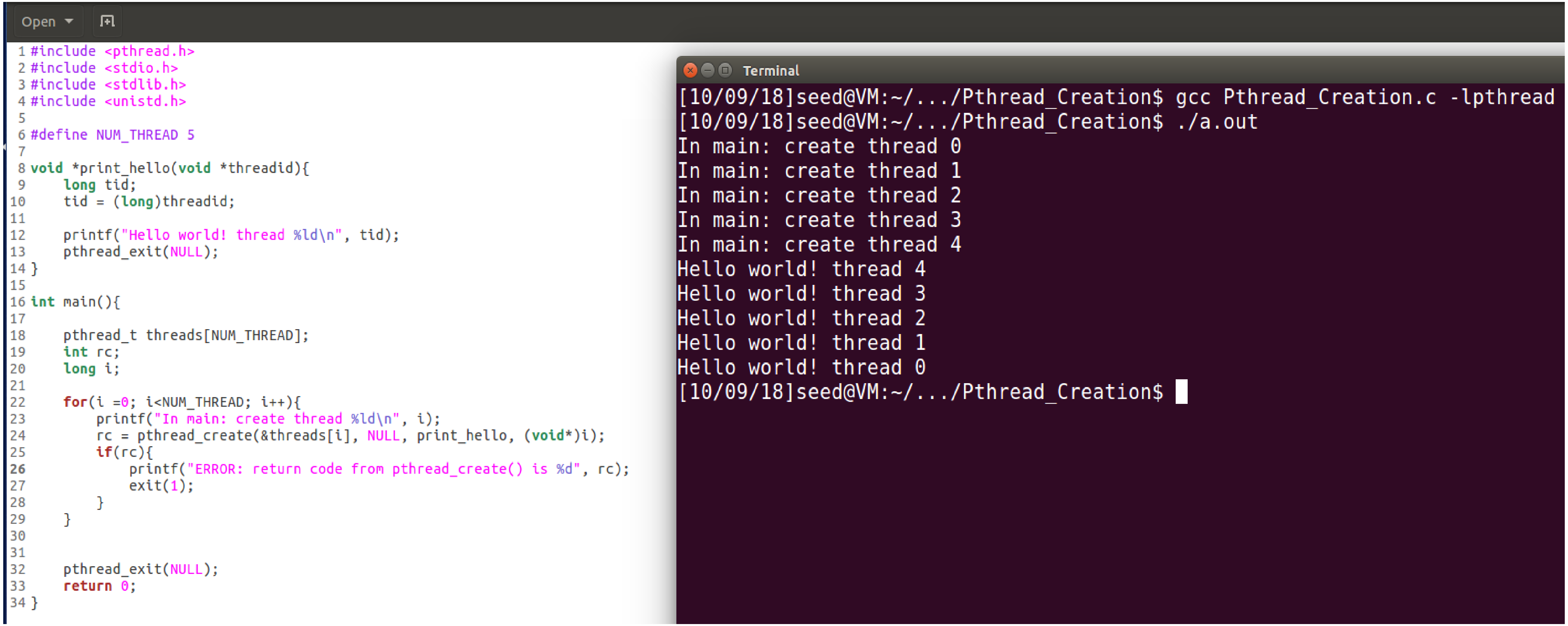
# Pthread creation

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

- Pthread is declared with type:
  - pthread_t (defined in “sys/types.h”)



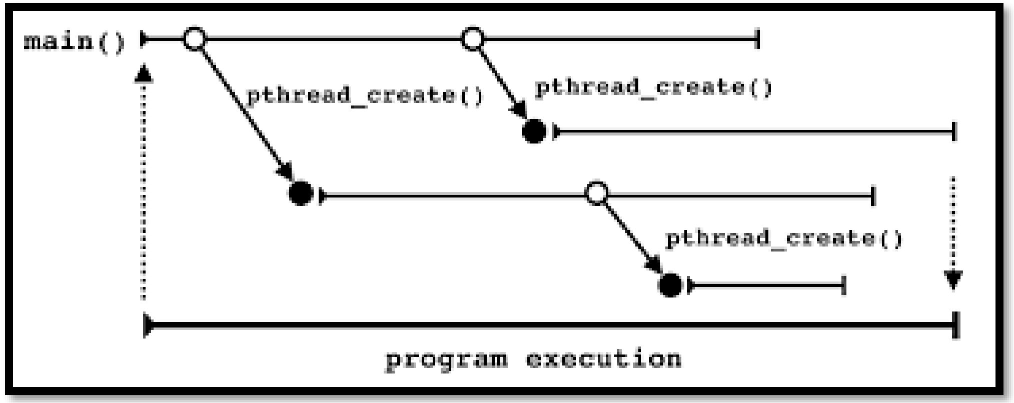
# Pthread creation



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

#define NUM_THREAD 5

void *print_hello(void *threadid){
    long tid;
    tid = (long)threadid;

    printf("Hello world! thread %ld\n", tid);
    pthread_exit(NULL);
}

int main(){

    pthread_t threads[NUM_THREAD];
    int rc;
    long i;

    for(i =0; i<NUM_THREAD; i++){
        printf("In main: create thread %ld\n", i);
        rc = pthread_create(&threads[i], NULL, print_hello, (void*)i);
        if(rc){
            printf("ERROR: return code from pthread_create() is %d", rc);
            exit(1);
        }
    }


    pthread_exit(NULL);
    return 0;
}
```

```
[10/09/18]seed@VM:~/.../Pthread_Creation$ gcc Pthread_Creation.c -lpthread
[10/09/18]seed@VM:~/.../Pthread_Creation$ ./a.out
In main: create thread 0
In main: create thread 1
In main: create thread 2
In main: create thread 3
In main: create thread 4
Hello world! thread 4
Hello world! thread 3
Hello world! thread 2
Hello world! thread 1
Hello world! thread 0
[10/09/18]seed@VM:~/.../Pthread_Creation$
```

# Pthread termination



- pthread_exit:
  - void pthread_exit(void *retval);

  **Parameters:** This method accepts a mandatory parameter **retval** which is the pointer to an integer that stores the return status of the thread terminated. The scope of this variable must be **global** so that any thread waiting to join this thread may read the return status.

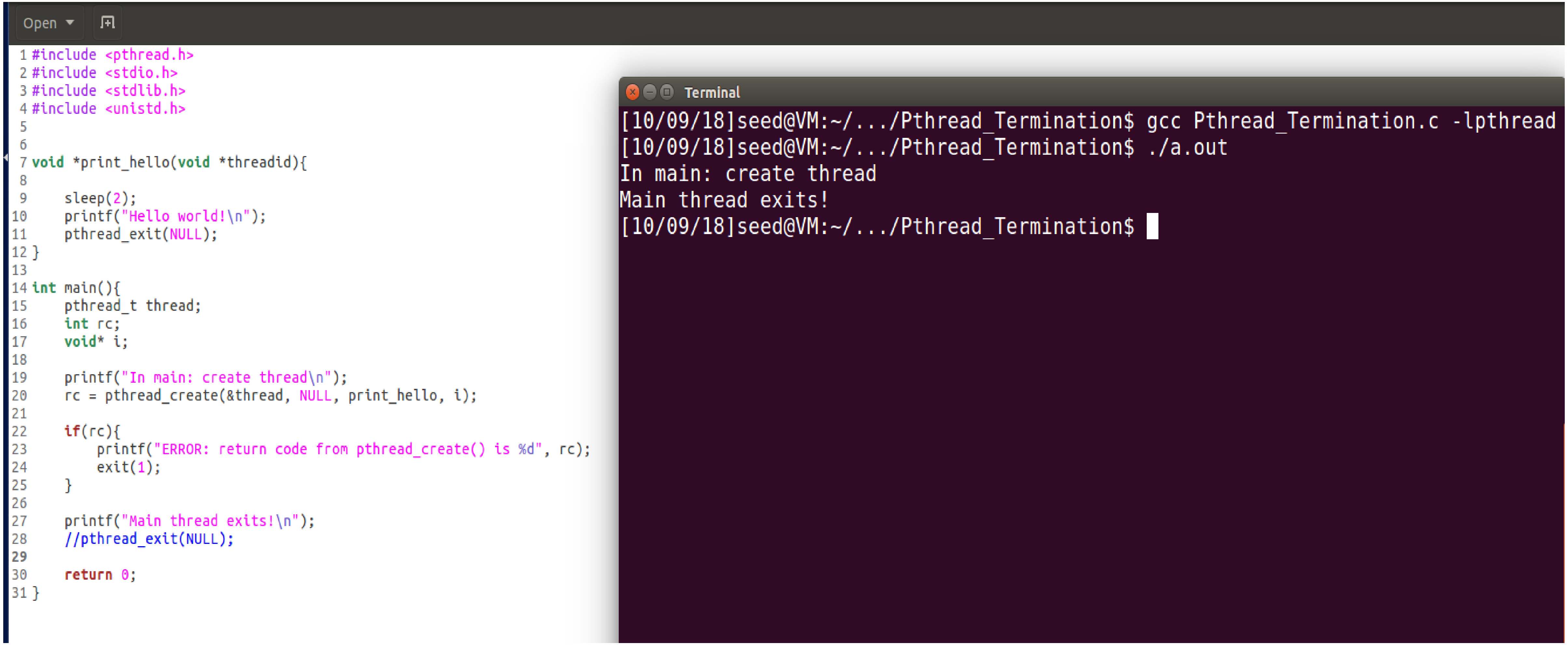
- This routine is used to **explicitly** exit a thread. Typically, the pthread_exit() routine is called after a thread has completed its work.

- If main() finishes before the **threads it has created, and exits with pthread_exit(),** the other threads will continue to execute. Otherwise, they will **be automatically terminated** when main() finishes.

- Recommendation: Use pthread_exit() to exit from all threads...especially main().

# Pthread termination – main thread without pthread_exit()

**Other threads would be automatically terminated** when main() finishes. Therefore, they might not have finished their work.

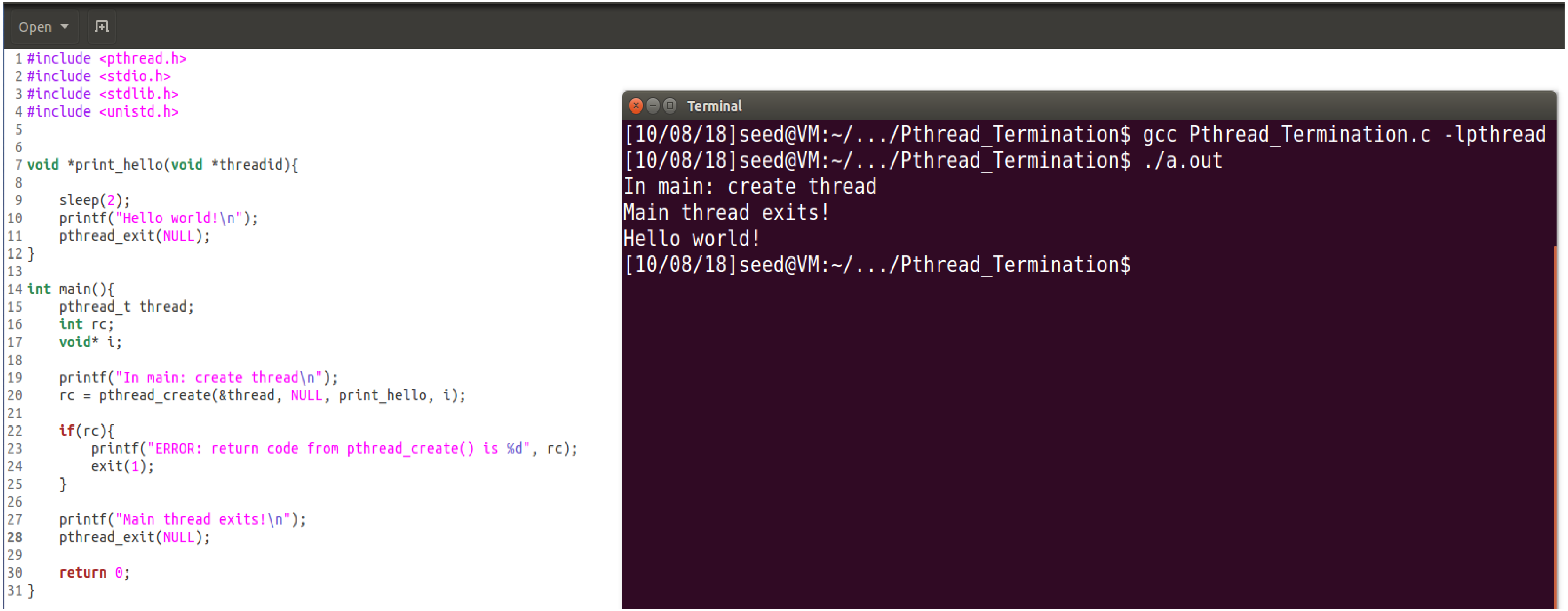


```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>


void *print_hello(void *threadid){

    sleep(2);
    printf("Hello world!\n");
    pthread_exit(NULL);
}

int main(){
    pthread_t thread;
    int rc;
    void* i;

    printf("In main: create thread\n");
    rc = pthread_create(&thread, NULL, print_hello, i);

    if(rc){
        printf("ERROR: return code from pthread_create() is %d", rc);
        exit(1);
    }

    printf("Main thread exits!\n");
    //pthread_exit(NULL);

    return 0;
}
```

```
[10/09/18]seed@VM:~/.../Pthread_Termination$ gcc Pthread_Termination.c -lpthread
[10/09/18]seed@VM:~/.../Pthread_Termination$ ./a.out
In main: create thread
Main thread exits!
[10/09/18]seed@VM:~/.../Pthread_Termination$
```

## Pthread termination – main with pthread_exit().

The other threads will continue to execute.



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


void *print_hello(void *threadid){

    sleep(2);
    printf("Hello world!\n");
    pthread_exit(NULL);
}

int main(){
    pthread_t thread;
    int rc;
    void* i;

    printf("In main: create thread\n");
    rc = pthread_create(&thread, NULL, print_hello, i);

    if(rc){
        printf("ERROR: return code from pthread_create() is %d", rc);
        exit(1);
    }

    printf("Main thread exits!\n");
    pthread_exit(NULL);

    return 0;
}
```

```
[10/08/18]seed@VM:~/.../Pthread_Termination$ gcc Pthread_Termination.c -lpthread
[10/08/18]seed@VM:~/.../Pthread_Termination$ ./a.out
In main: create thread
Main thread exits!
Hello world!
[10/08/18]seed@VM:~/.../Pthread_Termination$
```

# Pthread join - synchronization

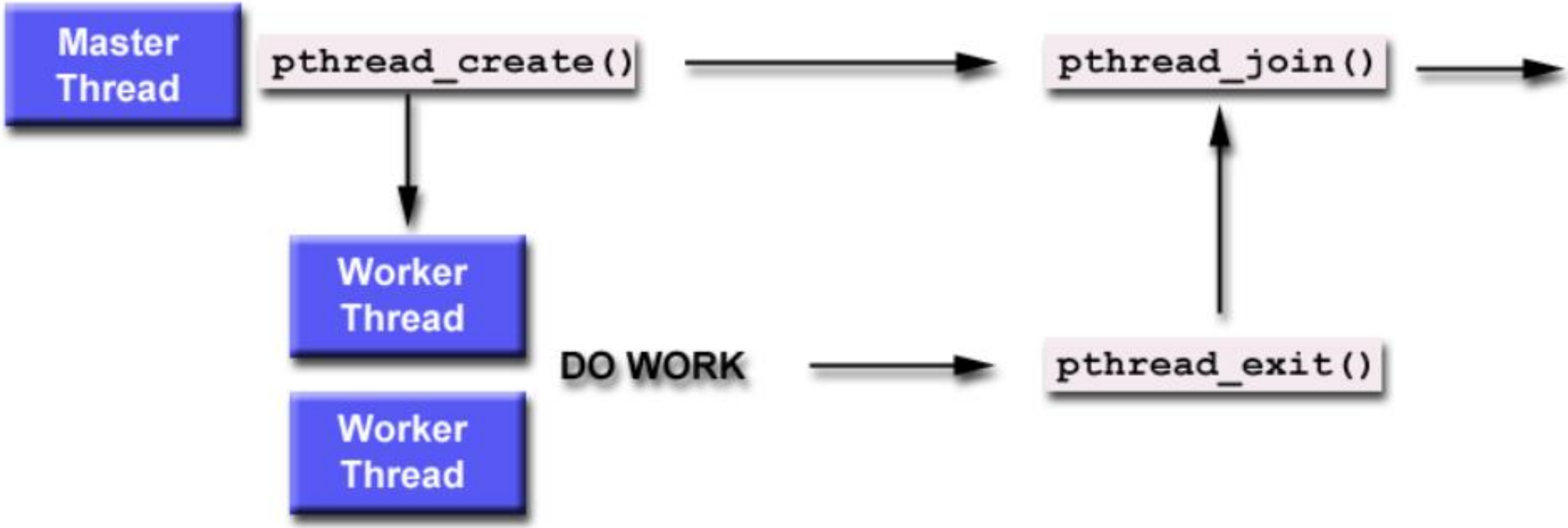
- pthread_join:
  - int pthread_join(pthread_t thread, void *retval);

- "Joining" is one way to accomplish ***synchronization*** between threads.

- The pthread_join() subroutine blocks the calling thread until the **specified thread terminates**.

- The programmer is able to obtain the target thread's termination return status if specified through pthread_exit(), in the status parameter.

"Joining" is one way to accomplish synchronization between threads.



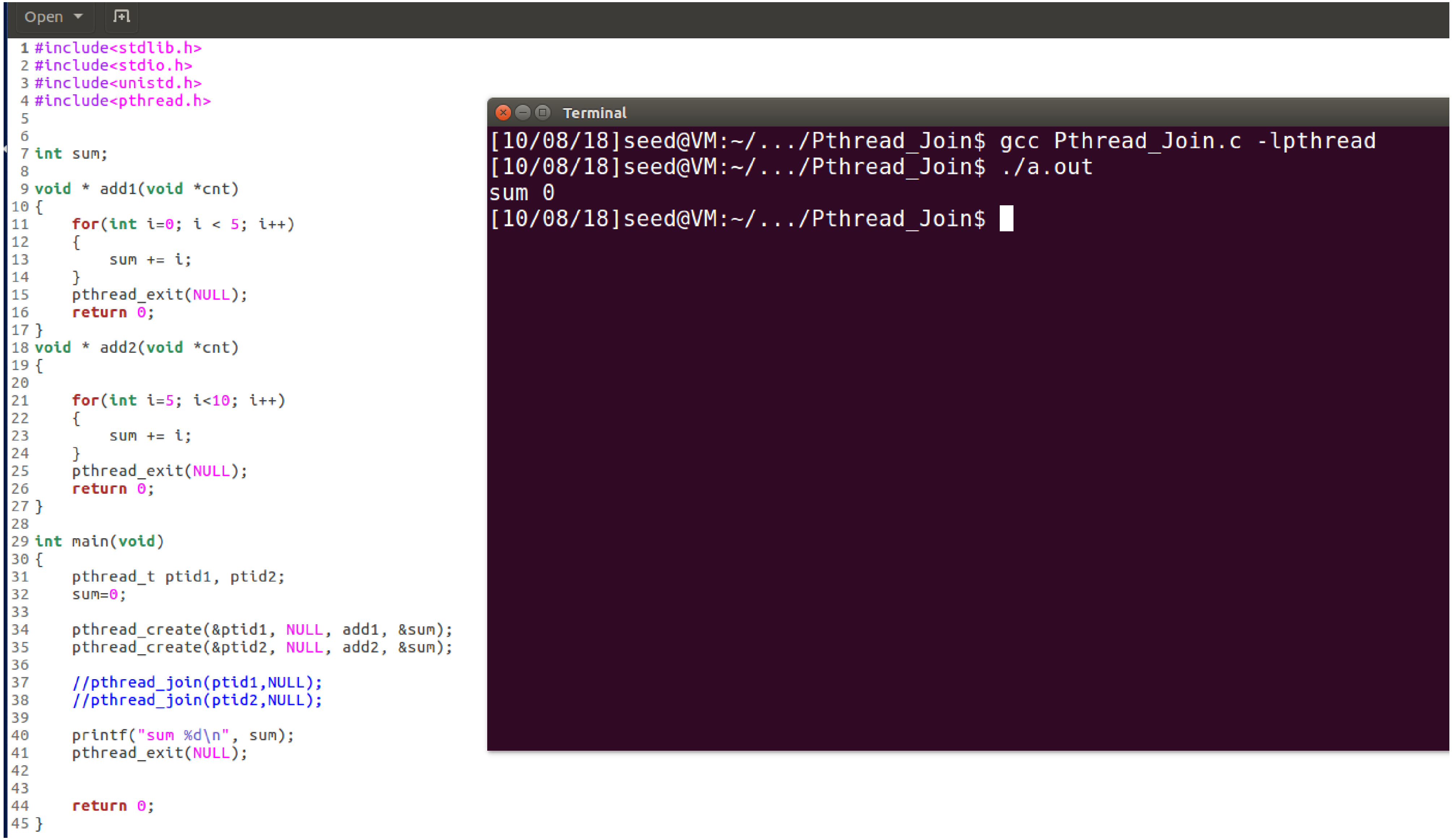
# Pthread join - synchronization

- Return value
  - On success, pthread_join() returns **0**;
  - On error, it returns an **error number**.

- It is impossible to join a detached thread.

- When a thread is created, one of its attributes defines whether it is joinable or detached. Detached means it can never be joined. (PTHREAD_CREATE_DETACHED or PTHREAD_CREATE_JOINABLE)

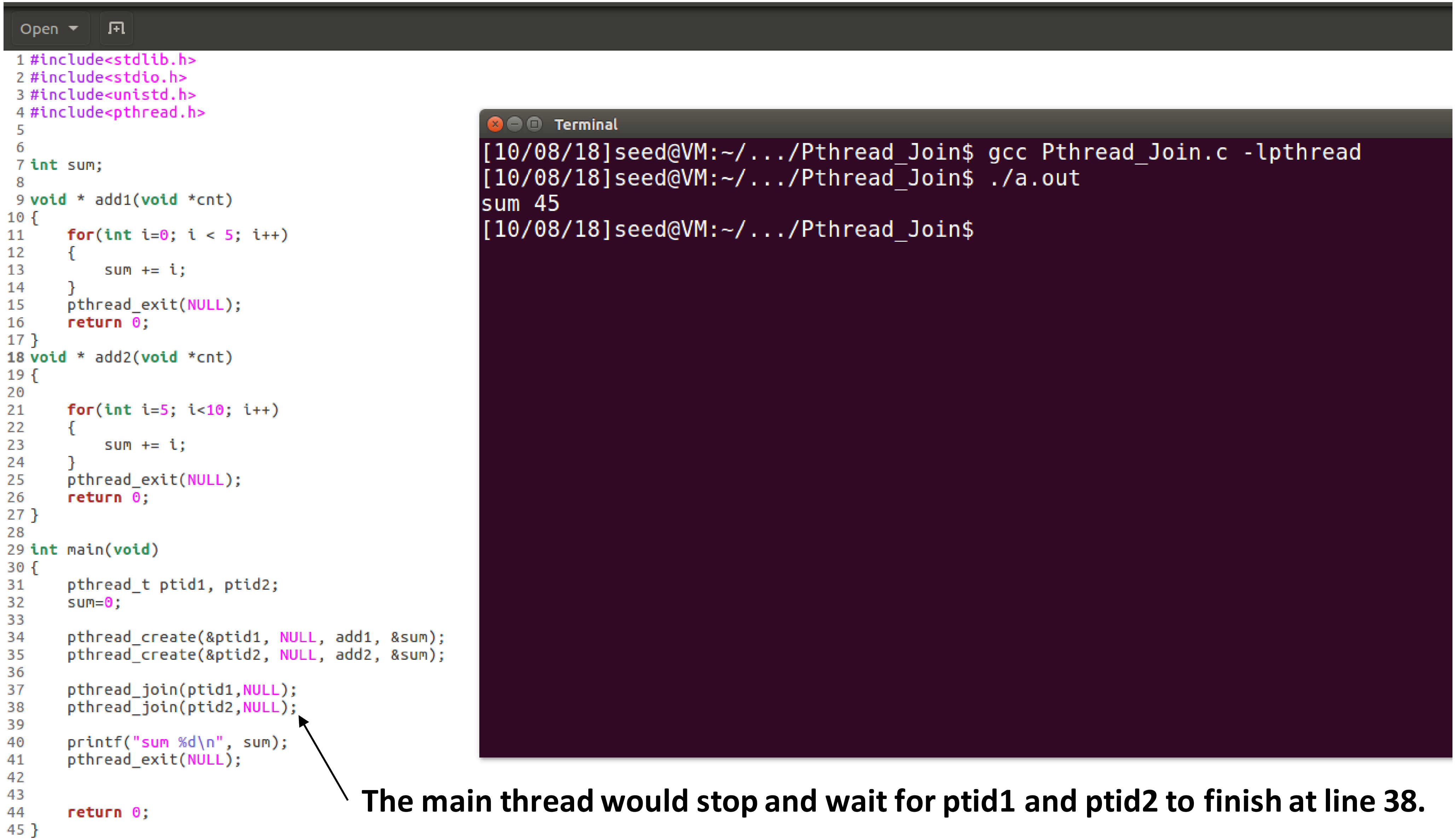
# Pthread join –  without calling pthread_join().



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


int sum;

void * add1(void *cnt)
{
    for(int i=0; i < 5; i++)
    {
        sum += i;
    }
    pthread_exit(NULL);
    return 0;
}
void * add2(void *cnt)
{

    for(int i=5; i<10; i++)
    {
        sum += i;
    }
    pthread_exit(NULL);
    return 0;
}

int main(void)
{
    pthread_t ptid1, ptid2;
    sum=0;

    pthread_create(&ptid1, NULL, add1, &sum);
    pthread_create(&ptid2, NULL, add2, &sum);

    //pthread_join(ptid1,NULL);
    //pthread_join(ptid2,NULL);

    printf("sum %d\n", sum);
    pthread_exit(NULL);


    return 0;
}
```

```
[10/08/18]seed@VM:~/.../Pthread_Join$ gcc Pthread_Join.c -lpthread
[10/08/18]seed@VM:~/.../Pthread_Join$ ./a.out
sum 0
[10/08/18]seed@VM:~/.../Pthread_Join$ 
```

# Pthread join – synchronization  with calling pthread_join().



```
 1 #include<stdlib.h>
 2 #include<stdio.h>
 3 #include<unistd.h>
 4 #include<pthread.h>
 5
 6
 7 int sum;
 8
 9 void * add1(void *cnt)
10 {
11     for(int i=0; i < 5; i++)
12     {
13         sum += i;
14     }
15     pthread_exit(NULL);
16     return 0;
17 }
18 void * add2(void *cnt)
19 {
20
21     for(int i=5; i<10; i++)
22     {
23         sum += i;
24     }
25     pthread_exit(NULL);
26     return 0;
27 }
28
29 int main(void)
30 {
31     pthread_t ptid1, ptid2;
32     sum=0;
33
34     pthread_create(&ptid1, NULL, add1, &sum);
35     pthread_create(&ptid2, NULL, add2, &sum);
36
37     pthread_join(ptid1,NULL);
38     pthread_join(ptid2,NULL);
39
40     printf("sum %d\n", sum);
41     pthread_exit(NULL);
42
43
44     return 0;
45 }
```

```
[10/08/18]seed@VM:~/.../Pthread_Join$ gcc Pthread_Join.c -lpthread
[10/08/18]seed@VM:~/.../Pthread_Join$ ./a.out
sum 45
[10/08/18]seed@VM:~/.../Pthread_Join$
```

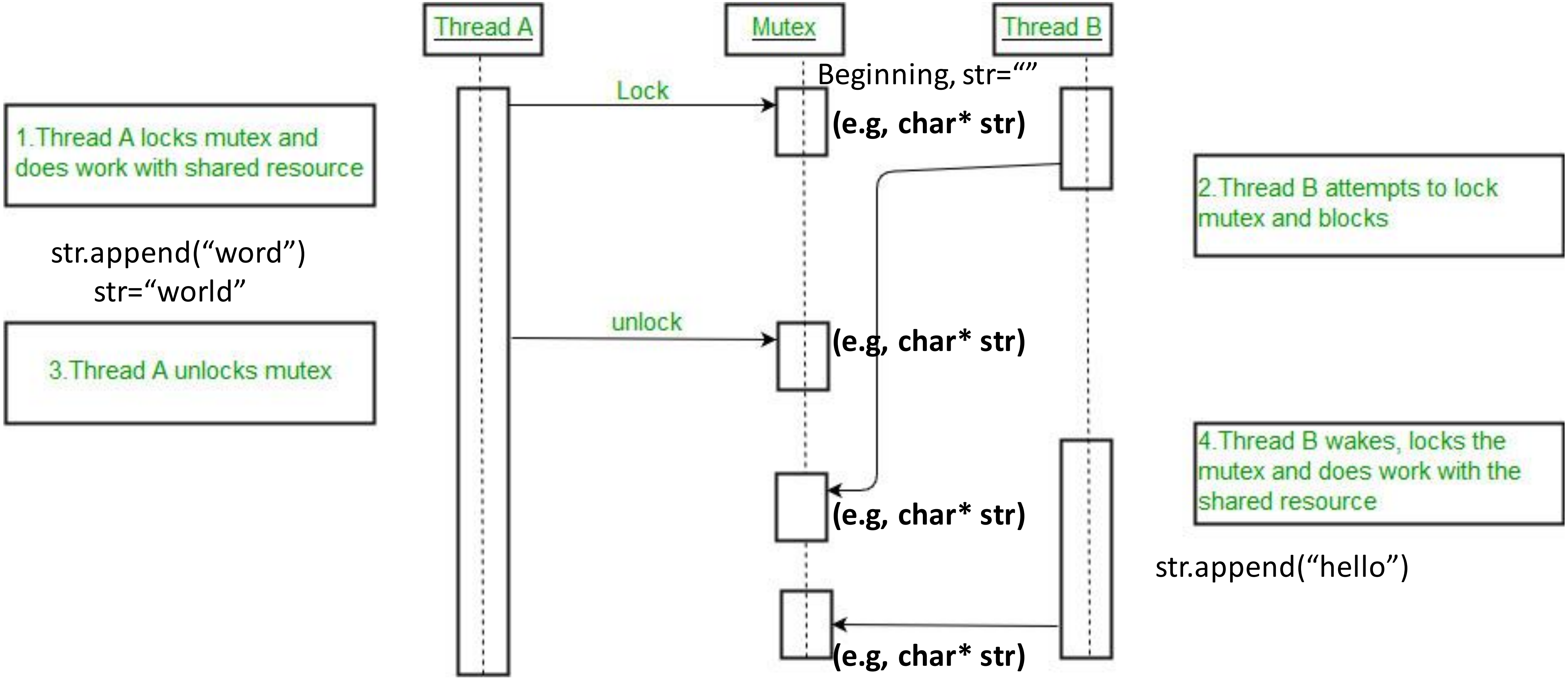
**The main thread would stop and wait for ptid1 and ptid2 to finish at line 38.**

# Pthread mutex – flag for privacy/security

- Mutex is an abbreviation for "mutual exclusion". Mutex variables are one of the primary means of implementing thread synchronization and for protecting shared data when multiple writes occur.

- **A mutex variable acts like a "lock" protecting access to a shared data resource.**

- A mutex variable acts like a "lock" protecting access to a shared data resource.



Thread A

Mutex

Thread B

Beginning, str=“”

Lock

**(e.g, char* str)**

1.Thread A locks mutex and does work with shared resource

2.Thread B attempts to lock mutex and blocks

str.append(“word”)

str=“world”

unlock

**(e.g, char* str)**

3.Thread A unlocks mutex

4.Thread B wakes, locks the mutex and does work with the shared resource

**(e.g, char* str)**

str.append(“hello”)

**(e.g, char* str)**

Finally, str=“world hello”

# Pthread mutex - flag

- Mutex should be declared with type:
  - pthread_mutex_t (defined in “sys/types.h”)

- Mutex should be initialized before it is used:
  - int pthread_mutex_init( pthread_mutex_t *mutex,
    const pthread_mutexattr_t *attr);
  - It initialises the mutex referenced by *mutex* with attributes specified by *attr*.
  - If *attr* is NULL, the default mutex attributes are used; the effect is the same as passing the address of a default mutex attributes object.
  - Upon successful initialisation, the state of the mutex becomes initialised and unlocked.

- Mutex should be free if it is no longer used:
  - int pthread_mutex_destroy(pthread_mutex_t *);

# Pthread mutex - flag

- Pthread mutex lock routines:
  - int pthread_mutex_lock(pthread_mutex_t *mutex);
  - int pthread_mutex_trylock(pthread_mutex_t *mutex);
  - int pthread_mutex_unlock(pthread_mutex_t *mutex);

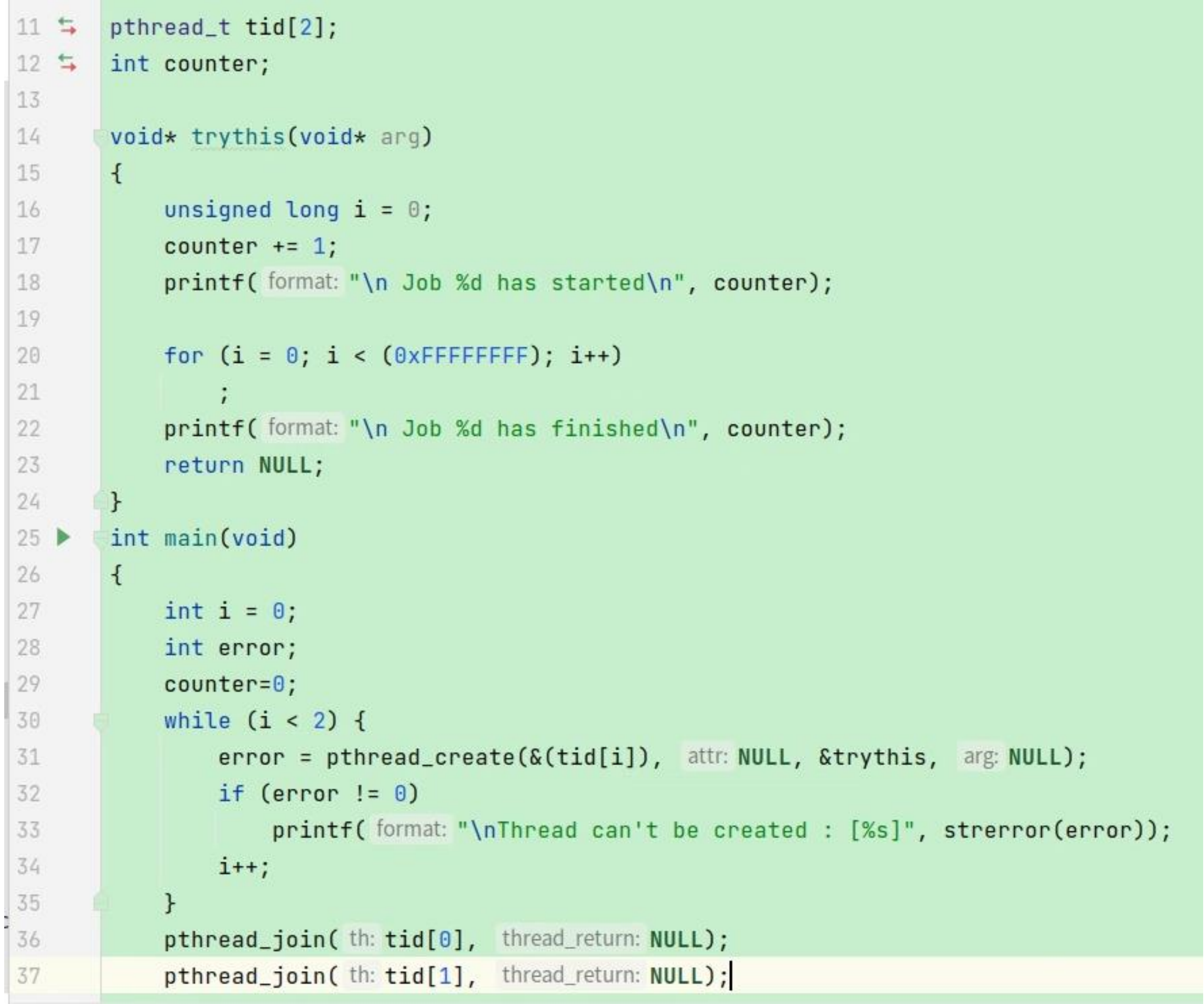
# Pthread mutex - flag

- The **pthread_mutex_lock()** routine is used by a thread to acquire a lock on the specified mutex variable. **If the mutex is already locked by another thread, this call will block the calling thread until the mutex is unlocked**.

- **pthread_mutex_trylock()** will **attempt to lock a mutex. However, if the mutex is already locked, the routine will return immediately with a "busy" error code**. This routine may be useful in preventing deadlock conditions, as in a priority-inversion situation.

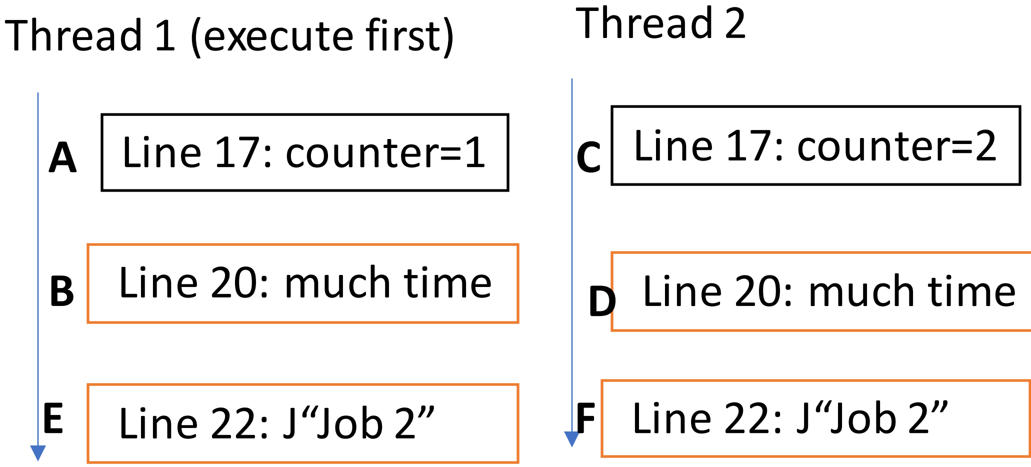
- **pthread_mutex_unlock()** will **unlock a mutex if called by the owning thread**. Calling this routine is required after a thread has completed its use of protected data if other threads are to acquire the mutex for their work with the protected data. An **error** will be returned if:
  - **If the mutex was already unlocked**
  - **If the mutex is owned by another thread**

Line 20 consume more time (e.g. 3seconds)



```
11  pthread_t tid[2];
12  int counter;
13
14  void* trythis(void* arg)
15  {
16      unsigned long i = 0;
17      counter += 1;
18      printf( format: "\n Job %d has started\n", counter);
19
20      for (i = 0; i < (0xFFFFFFFF); i++)
21          ;
22      printf( format: "\n Job %d has finished\n", counter);
23      return NULL;
24  }
25  int main(void)
26  {
27      int i = 0;
28      int error;
29      counter=0;
30      while (i < 2) {
31          error = pthread_create(&(tid[i]),  attr: NULL, &trythis,  arg: NULL);
32          if (error != 0)
33              printf( format: "\nThread can't be created : [%s]", strerror(error));
34          i++;
35      }
36      pthread_join( th: tid[0],  thread_return: NULL);
37      pthread_join( th: tid[1],  thread_return: NULL);
```

**Execution Order: A B C D E F**



Thread 1 (execute first)

A Line 17: counter=1

B Line 20: much time

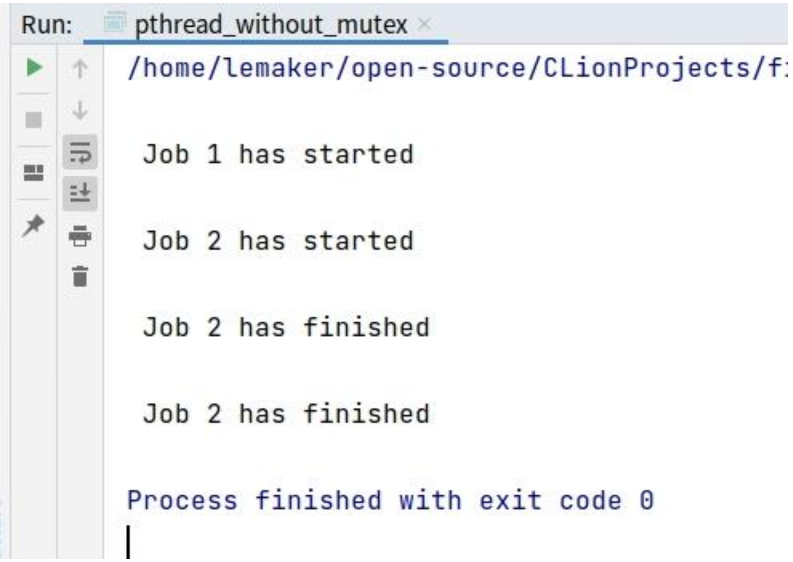
E Line 22: J“Job 2”

Thread 2

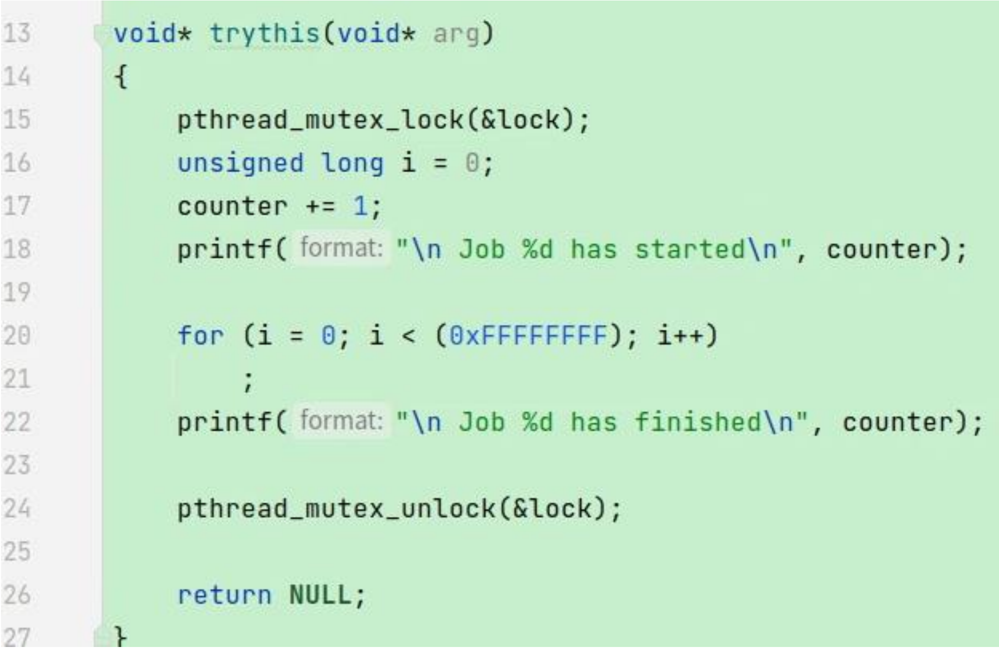
C Line 17: counter=2

D Line 20: much time

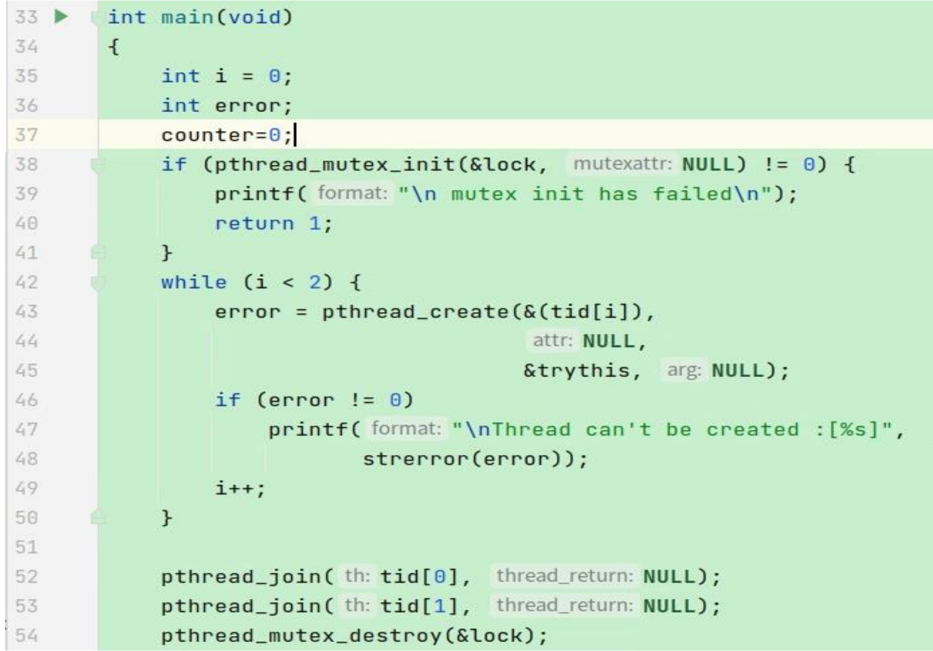
F Line 22: J“Job 2”



```
/home/lemaker/open-source/CLionProjects/f

 Job 1 has started

 Job 2 has started

 Job 2 has finished

 Job 2 has finished

Process finished with exit code 0
```



```
13  void* trythis(void* arg)
14  {
15      pthread_mutex_lock(&lock);
16      unsigned long i = 0;
17      counter += 1;
18      printf( format: "\n Job %d has started\n", counter);
19
20      for (i = 0; i < (0xFFFFFFFF); i++)
21          ;
22      printf( format: "\n Job %d has finished\n", counter);
23
24      pthread_mutex_unlock(&lock);
25
26      return NULL;
27  }
```



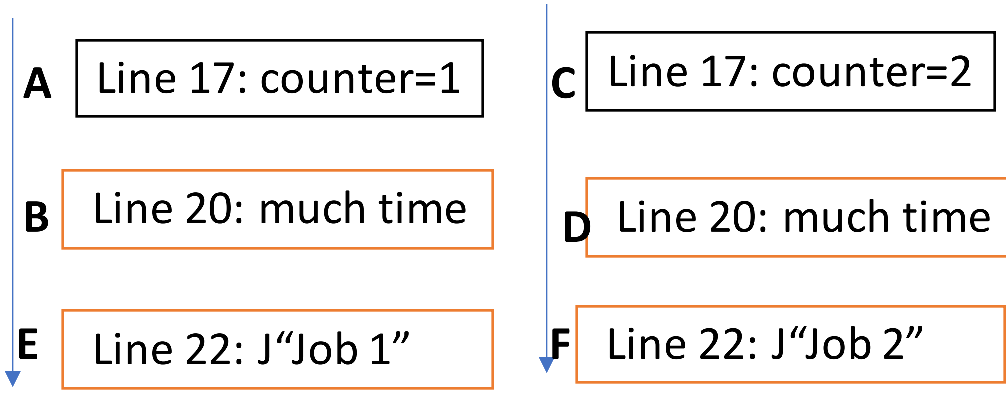
```
33  int main(void)
34  {
35      int i = 0;
36      int error;
37      counter=0;
38      if (pthread_mutex_init(&lock,  mutexattr: NULL) != 0) {
39          printf( format: "\n mutex init has failed\n");
40          return 1;
41      }
42      while (i < 2) {
43          error = pthread_create(&(tid[i]),
44                                  attr: NULL,
45                                  &trythis,  arg: NULL);
46          if (error != 0)
47              printf( format: "\nThread can't be created :[%s]",
48                     strerror(error));
49          i++;
50      }
51
52      pthread_join( th: tid[0],  thread_return: NULL);
53      pthread_join( th: tid[1],  thread_return: NULL);
54      pthread_mutex_destroy(&lock);
```

Line 20 consume more time (e.g. 3seconds)

**Execution Order: A B C D E F (impossible)**

**Execution Order: A B E C D F  or  C D F A B E**

Thread 1 (execute first)    Thread 2



A Line 17: counter=1

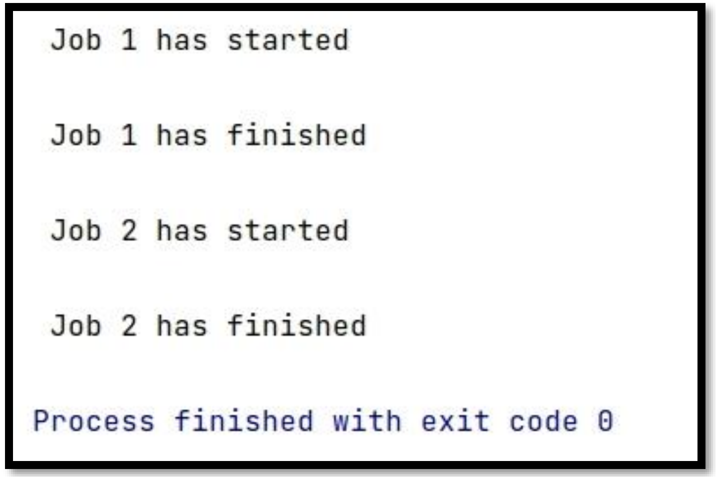
B Line 20: much time

E Line 22: J"Job 1"

C Line 17: counter=2

D Line 20: much time

F Line 22: J"Job 2"



```
 Job 1 has started

 Job 1 has finished

 Job 2 has started

 Job 2 has finished

Process finished with exit code 0
```

https://www.geeksforgeeks.org/mutex-lock-for-linux-thread-synchronization/

# Pthread condition - signals

- **Condition variables** provide yet another way for threads to synchronize.

- While **mutexes** implement synchronization by controlling thread access to data, condition variables allow threads **to synchronize based upon the actual value of data.**

- A condition variable is always **used in conjunction with a mutex lock**.

# Pthread condition - signals

- Condition variables must be declared with type: pthread_cond_t
  - pthread_cond_t (defined in “sys/types.h”)

- Condition variables must be initialized before it is used:
  - int pthread_cond_init(pthread_cond_t *, const pthread_condattr_t *);

- Condition variables should be freed if it is no longer used:
  - int pthread_cond_destroy(pthread_cond_t *);

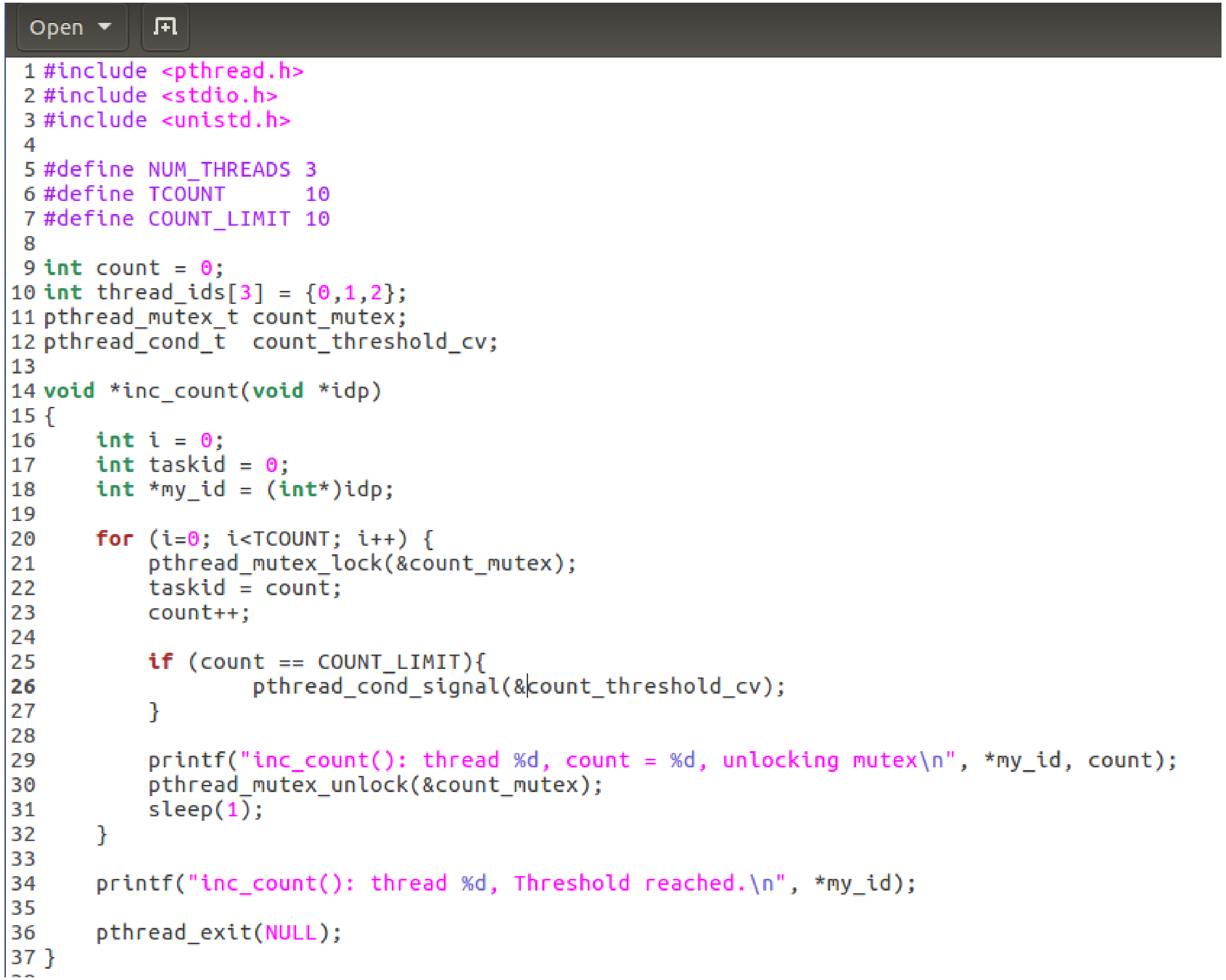
# Pthread condition - signals

- Pthread condition routines:
  - int pthread_cond_wait(pthread_cond_t *, pthread_mutex_t *);
  - int pthread_cond_signal(pthread_cond_t *);
  - int pthread_cond_broadcast(pthread_cond_t *);

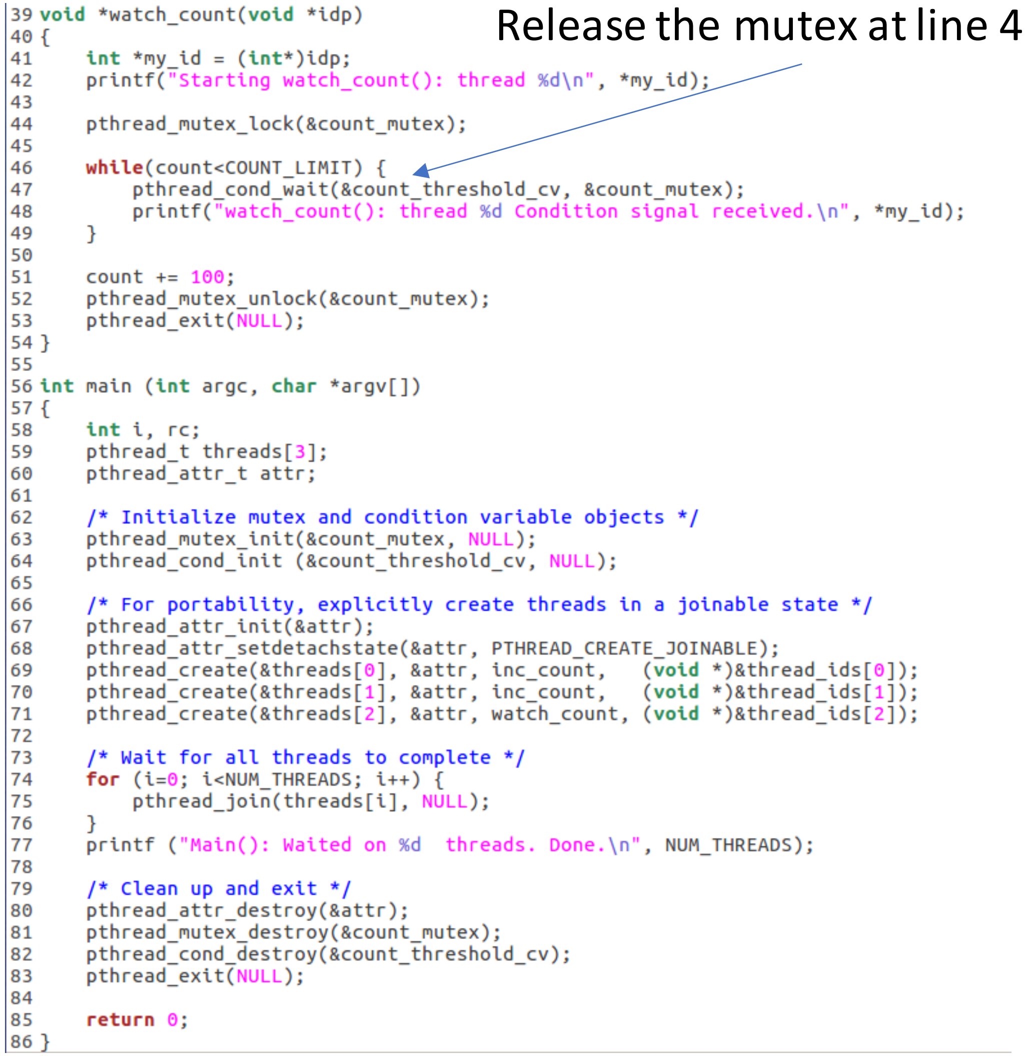
# Pthread condition - signals

- **pthread_cond_wait()** blocks the calling thread until **the specified condition** is signalled. This routine should be called **while mutex is locked**, and it will **automatically release the mutex while it waits**.
- **The pthread_cond_signal()** routine is used **to signal (or wake up) another thread which is waiting** on the condition variable. It should be called after mutex is locked, and must unlock mutex in order for pthread_cond_wait() routine to complete.
- **The pthread_cond_broadcast()** routine should be used instead of pthread_cond_signal() if more than one thread is in a blocking wait state.

# Pthread condition - signals



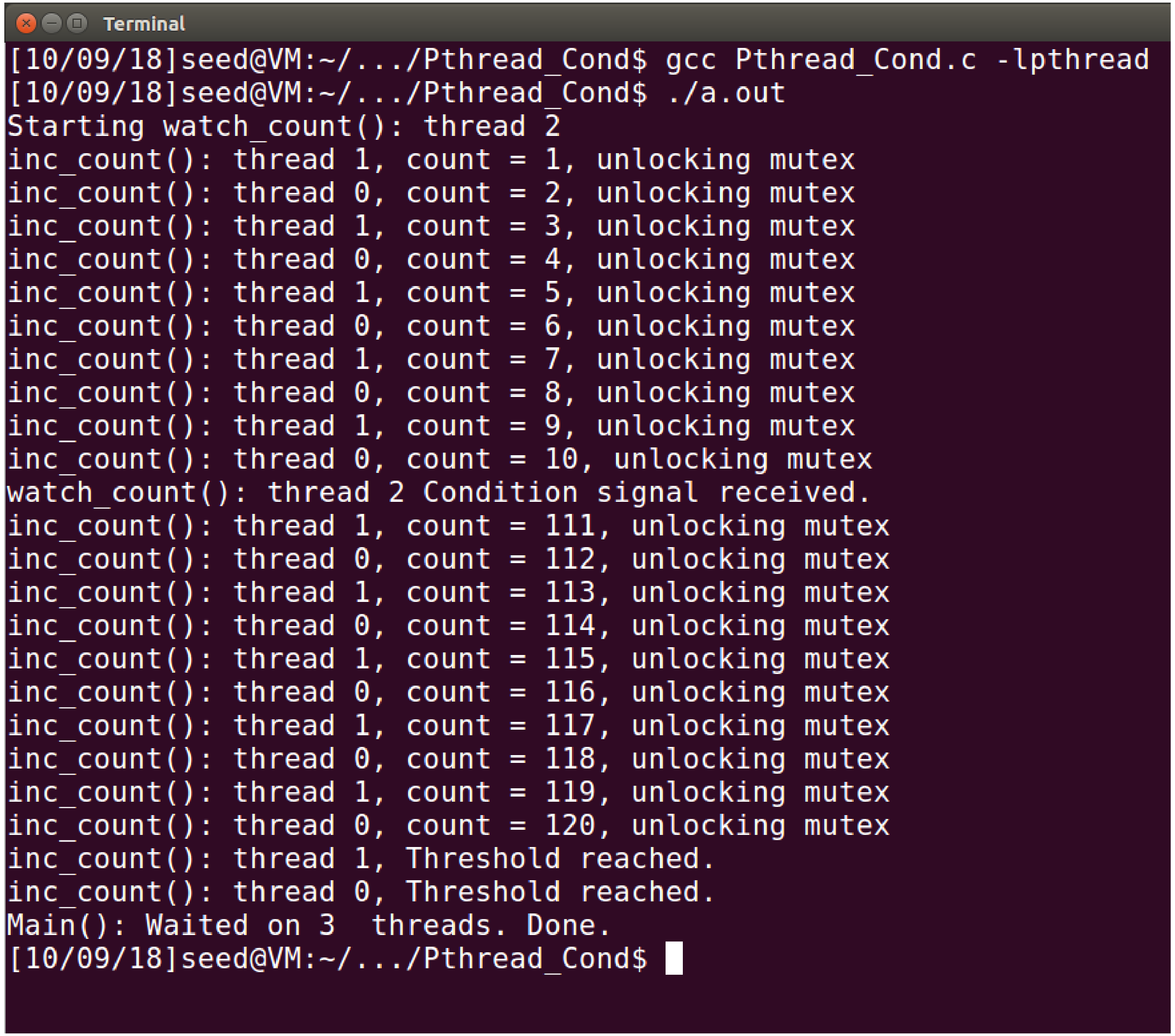
```c
 1 #include <pthread.h>
 2 #include <stdio.h>
 3 #include <unistd.h>
 4
 5 #define NUM_THREADS 3
 6 #define TCOUNT      10
 7 #define COUNT_LIMIT 10
 8
 9 int count = 0;
10 int thread_ids[3] = {0,1,2};
11 pthread_mutex_t count_mutex;
12 pthread_cond_t  count_threshold_cv;
13
14 void *inc_count(void *idp)
15 {
16     int i = 0;
17     int taskid = 0;
18     int *my_id = (int*)idp;
19
20     for (i=0; i<TCOUNT; i++) {
21         pthread_mutex_lock(&count_mutex);
22         taskid = count;
23         count++;
24
25         if (count == COUNT_LIMIT){
26                 pthread_cond_signal(&count_threshold_cv);
27         }
28
29         printf("inc_count(): thread %d, count = %d, unlocking mutex\n", *my_id, count);
30         pthread_mutex_unlock(&count_mutex);
31         sleep(1);
32     }
33
34     printf("inc_count(): thread %d, Threshold reached.\n", *my_id);
35
36     pthread_exit(NULL);
37 }
```



Release the mutex at line 47

```c
39 void *watch_count(void *idp)
40 {
41     int *my_id = (int*)idp;
42     printf("Starting watch_count(): thread %d\n", *my_id);
43
44     pthread_mutex_lock(&count_mutex);
45
46     while(count<COUNT_LIMIT) {
47         pthread_cond_wait(&count_threshold_cv, &count_mutex);
48         printf("watch_count(): thread %d Condition signal received.\n", *my_id);
49     }
50
51     count += 100;
52     pthread_mutex_unlock(&count_mutex);
53     pthread_exit(NULL);
54 }
55
56 int main (int argc, char *argv[])
57 {
58     int i, rc;
59     pthread_t threads[3];
60     pthread_attr_t attr;
61
62     /* Initialize mutex and condition variable objects */
63     pthread_mutex_init(&count_mutex, NULL);
64     pthread_cond_init (&count_threshold_cv, NULL);
65
66     /* For portability, explicitly create threads in a joinable state */
67     pthread_attr_init(&attr);
68     pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
69     pthread_create(&threads[0], &attr, inc_count,   (void *)&thread_ids[0]);
70     pthread_create(&threads[1], &attr, inc_count,   (void *)&thread_ids[1]);
71     pthread_create(&threads[2], &attr, watch_count, (void *)&thread_ids[2]);
72
73     /* Wait for all threads to complete */
74     for (i=0; i<NUM_THREADS; i++) {
75         pthread_join(threads[i], NULL);
76     }
77     printf ("Main(): Waited on %d  threads. Done.\n", NUM_THREADS);
78
79     /* Clean up and exit */
80     pthread_attr_destroy(&attr);
81     pthread_mutex_destroy(&count_mutex);
82     pthread_cond_destroy(&count_threshold_cv);
83     pthread_exit(NULL);
84
85     return 0;
86 }
```

# Pthread condition - signals



```
[10/09/18]seed@VM:~/.../Pthread_Cond$ gcc Pthread_Cond.c -lpthread
[10/09/18]seed@VM:~/.../Pthread_Cond$ ./a.out
Starting watch_count(): thread 2
inc_count(): thread 1, count = 1, unlocking mutex
inc_count(): thread 0, count = 2, unlocking mutex
inc_count(): thread 1, count = 3, unlocking mutex
inc_count(): thread 0, count = 4, unlocking mutex
inc_count(): thread 1, count = 5, unlocking mutex
inc_count(): thread 0, count = 6, unlocking mutex
inc_count(): thread 1, count = 7, unlocking mutex
inc_count(): thread 0, count = 8, unlocking mutex
inc_count(): thread 1, count = 9, unlocking mutex
inc_count(): thread 0, count = 10, unlocking mutex
watch_count(): thread 2 Condition signal received.
inc_count(): thread 1, count = 111, unlocking mutex
inc_count(): thread 0, count = 112, unlocking mutex
inc_count(): thread 1, count = 113, unlocking mutex
inc_count(): thread 0, count = 114, unlocking mutex
inc_count(): thread 1, count = 115, unlocking mutex
inc_count(): thread 0, count = 116, unlocking mutex
inc_count(): thread 1, count = 117, unlocking mutex
inc_count(): thread 0, count = 118, unlocking mutex
inc_count(): thread 1, count = 119, unlocking mutex
inc_count(): thread 0, count = 120, unlocking mutex
inc_count(): thread 1, Threshold reached.
inc_count(): thread 0, Threshold reached.
Main(): Waited on 3  threads. Done.
[10/09/18]seed@VM:~/.../Pthread_Cond$ 
```

# References

- https://www.baeldung.com/cs/async-vs-multi-threading
- https://www.javatpoint.com/process-vs-thread
- https://www.geeksforgeeks.org/multithreading-c-2/
- https://www.geeksforgeeks.org/condition-wait-signal-multi-threading/
- http://www.cs.unibo.it/~ghini/didattica/sistop/pthreads_tutorial/POSIX_Threads_Programming.htm
- https://www.geeksforgeeks.org/mutex-lock-for-linux-thread-synchronization/

Thank you