CSE130: Principles of Computer Systems Design

Teaching Assistant: Nilufar Ferdous

Threads

- A thread is a path of execution that is identified by a unique entity within a process.
- A process can have multiple threads of execution and they all share the same process address space and system state information.

Prerequisite:

- To use Posix thread (pthread), we need to include the following header:
  #include <pthread.h>
- To successfully compile the C++ code with Posix thread (pthread), we need to link in the Posix thread (pthread) library by specifying the following library option to g++:
  -lpthread

Thread Creation:

To create a Posix thread (pthread), we use the following function:

```
int pthread_create(
pthread_t *id,
const pthread_attr_t *attr,
void *(*exec)(void*),
void *arg
);
```

- If successful, pthread_create() returns 0.
- If unsuccessful, pthread_create() returns -1, gives error report

Parameters of Pthread ():

pthread_t *id: id is the Input pointer/address that will contain the address of the thread identifier on successful return from the function.
const pthread_attr_t *attr : Input pointer to a structure that provides additional parameters for creating a customthread.
*exec: Input-pointer/address to a global function that is to be executed in the thread of execution
*arg : Input-pointer/address to the argument that is to be passed to the function to be executed by the thread

ThreadId:

- To identify threads within a process, each thread is assigned a unique identifier. This is also known as the Thread Id.
- To get the identifier of a thread, the following function is used:
  pthread_t pthread_self(void);

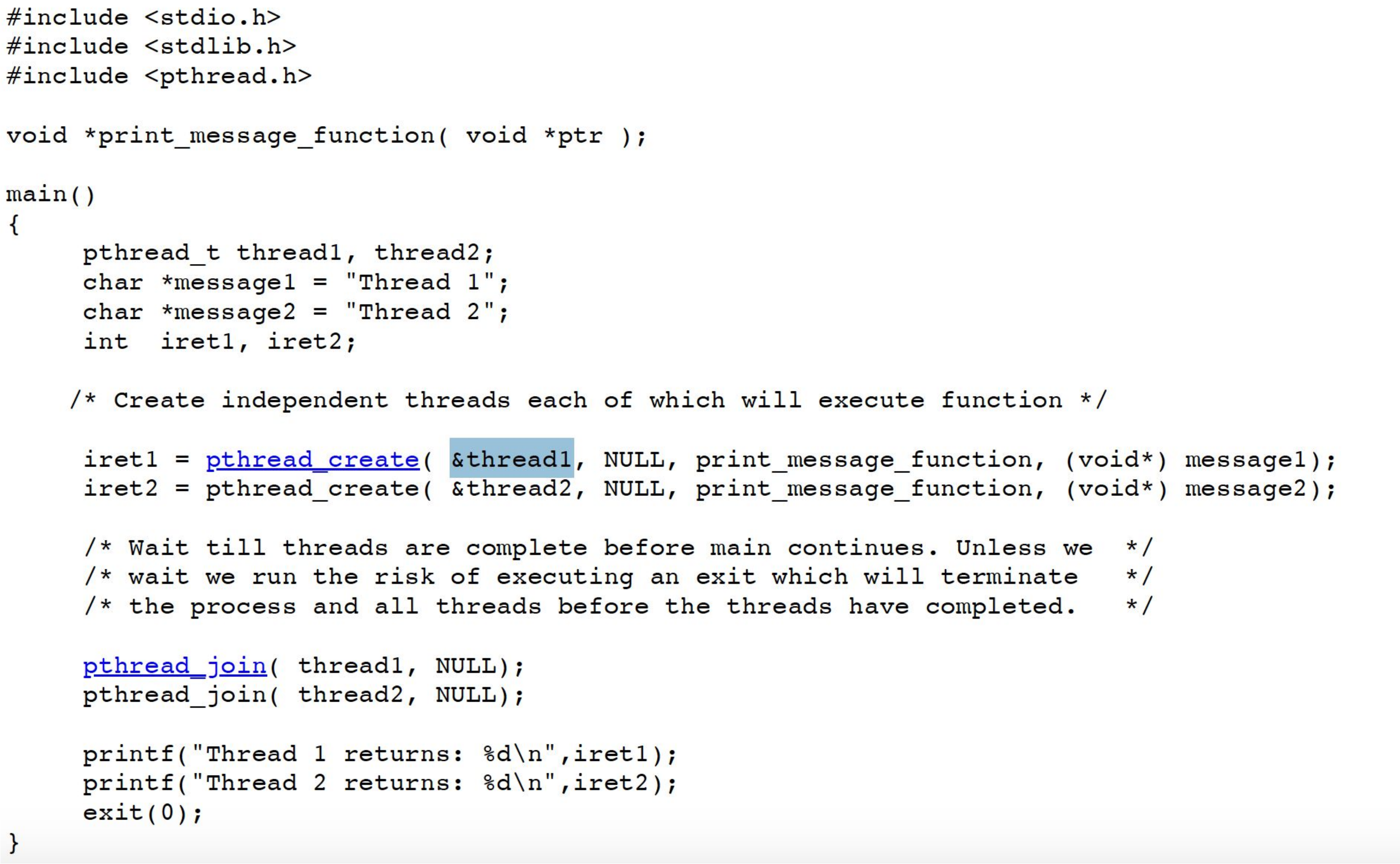
Join()

Allows the calling thread to wait for the ending of the target *thread*.

```
#define _OPEN_THREADS
#include <pthread.h>
int pthread_join(pthread_t thread, void **status);
```

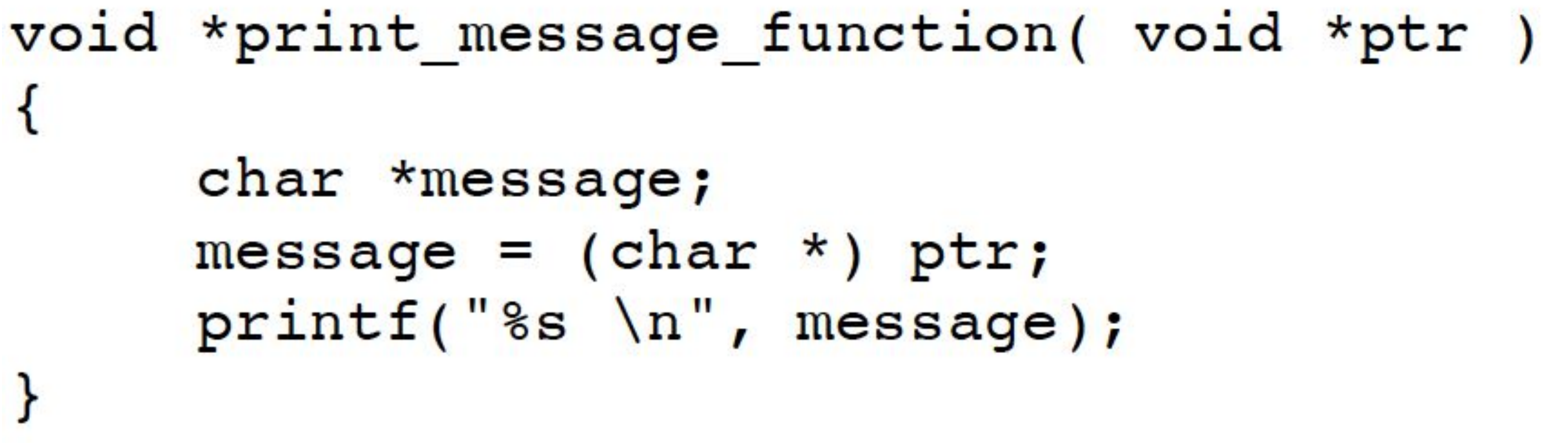
- *pthread_t* is the data type used to uniquely identify a thread.
- It is returned by pthread_create().
- If successful, pthread_join() returns 0.
- If unsuccessful, pthread_join() returns -1 and sets an error message

join:



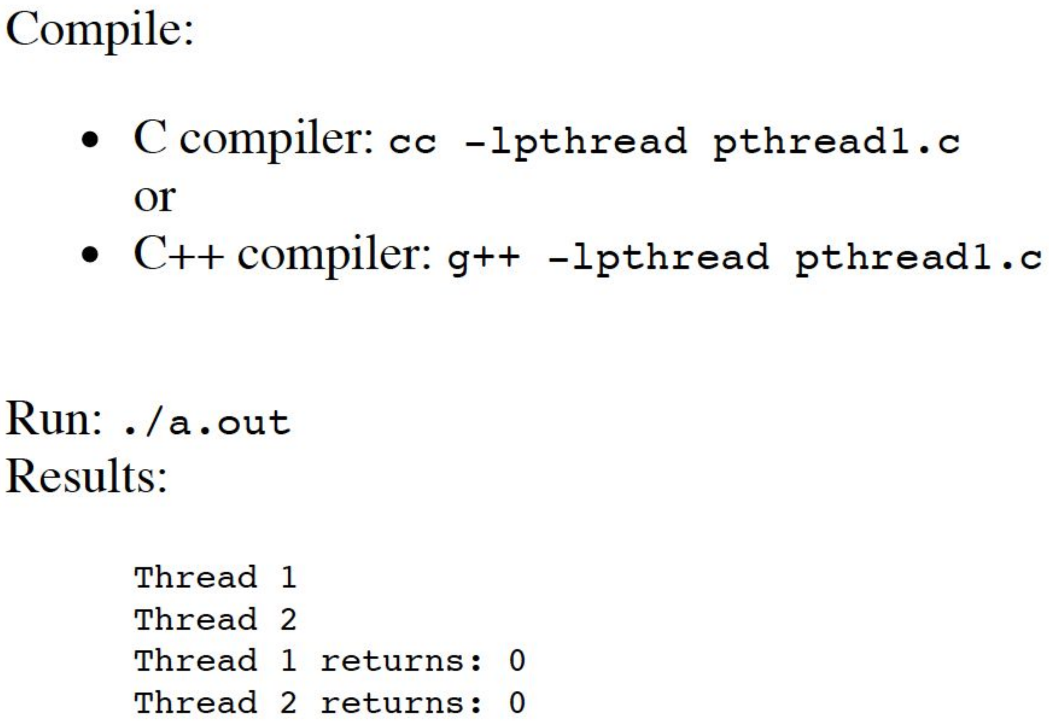
```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void *print_message_function( void *ptr );

main()
{
     pthread_t thread1, thread2;
     char *message1 = "Thread 1";
     char *message2 = "Thread 2";
     int  iret1, iret2;

    /* Create independent threads each of which will execute function */

     iret1 = pthread_create( &thread1, NULL, print_message_function, (void*) message1);
     iret2 = pthread_create( &thread2, NULL, print_message_function, (void*) message2);

     /* Wait till threads are complete before main continues. Unless we  */
     /* wait we run the risk of executing an exit which will terminate    */
     /* the process and all threads before the threads have completed.   */

     pthread_join( thread1, NULL);
     pthread_join( thread2, NULL);

     printf("Thread 1 returns: %d\n",iret1);
     printf("Thread 2 returns: %d\n",iret2);
     exit(0);
}
```

join:



```
void *print_message_function( void *ptr )
{
     char *message;
     message = (char *) ptr;
     printf("%s \n", message);
}
```

Output:



Compile:

- C compiler: `cc -lpthread pthread1.c`

  or
- C++ compiler: `g++ -lpthread pthread1.c`

Run: `./a.out`

Results:

```
Thread 1
Thread 2
Thread 1 returns: 0
Thread 2 returns: 0
```

Pthread Synchronization: Mutex

- When multiple threads access and manipulate a shared resource (ex: a variable for instance), the access to the shared resource needs to be controlled through a lock mechanism.
- Only one thread is allowed access to the shared resource at any point of time .
- The other threads are waiting to gain access the shared resource.
- In Posix thread (pthread) this is enforced by using a synchronization primitive called mutex lock.

Pthread_Mutex_init()

int pthread_mutex_init(pthread_mutex_t *mutex,const pthread_mutexattr_t *mattr);

- pthread_mutex_t*mutex: Input pointer to an instance of mutex lock object
- mattr : Input pointer to a structure that provides additional parameters for creating a custom mutex. This argument is usually specified as NULL in most cases

Mutex destroy(), Lock() & Unlock():

- Before one can discard an initialized instance of mutex lock object, the storage space allocated for the internal attributes needs to be deallocated using the following function:

  int pthread_mutex_destroy(pthread_mutex_t *mutex);

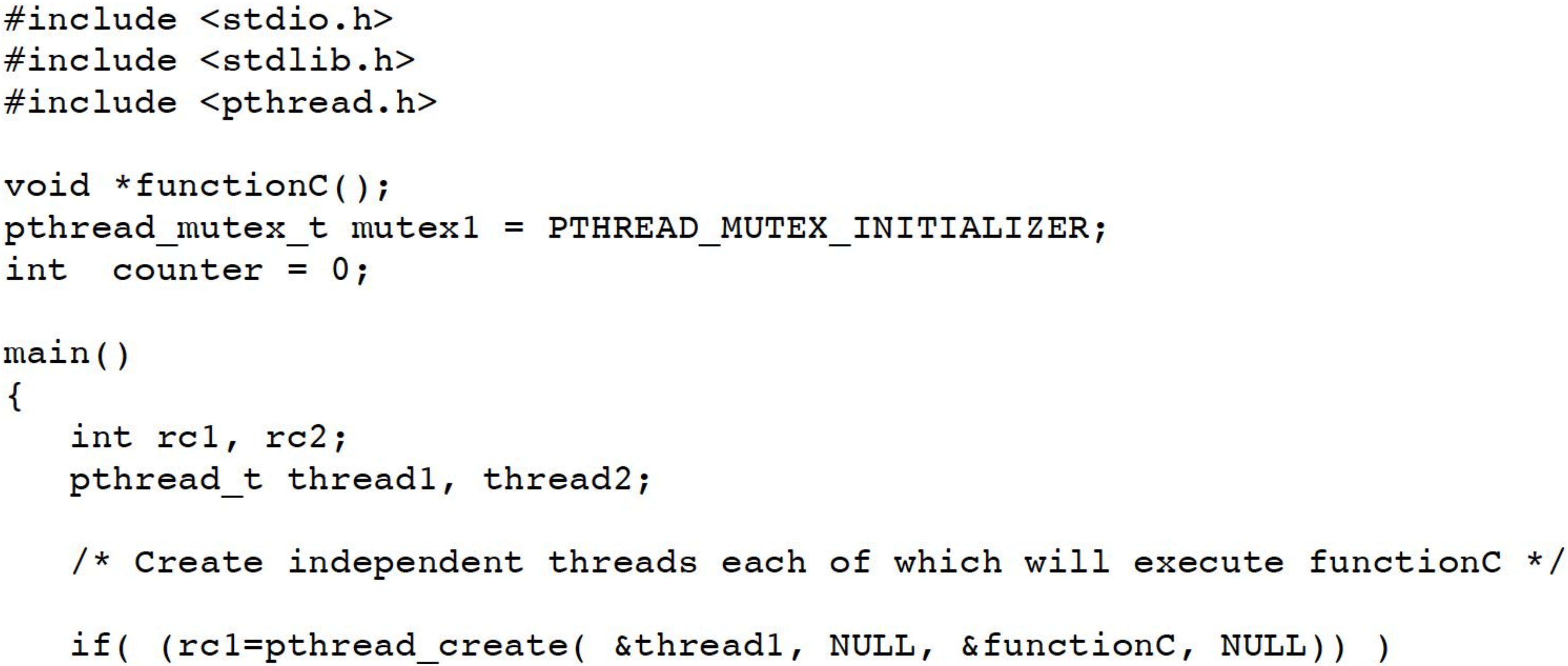
- To lock a mutex, use the following function:

  int pthread_mutex_lock(pthread_mutex_t *mutex);

- To unlock a mutex, use the following function:

  int pthread_mutex_unlock(pthread_mutex_t *mutex);

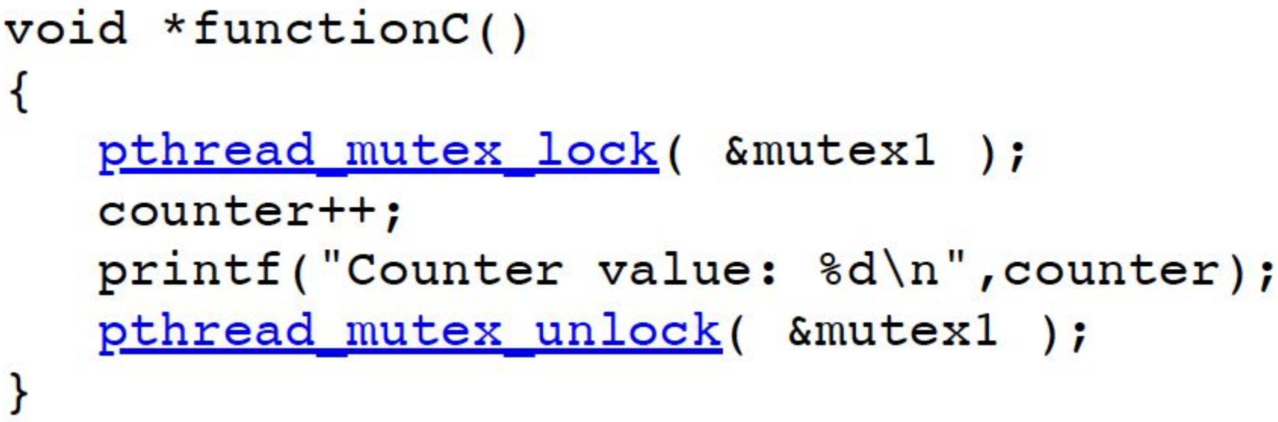
Mutex():



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

void *functionC();
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
int  counter = 0;

main()
{
   int rc1, rc2;
   pthread_t thread1, thread2;

   /* Create independent threads each of which will execute functionC */

   if( (rc1=pthread_create( &thread1, NULL, &functionC, NULL)) )
```

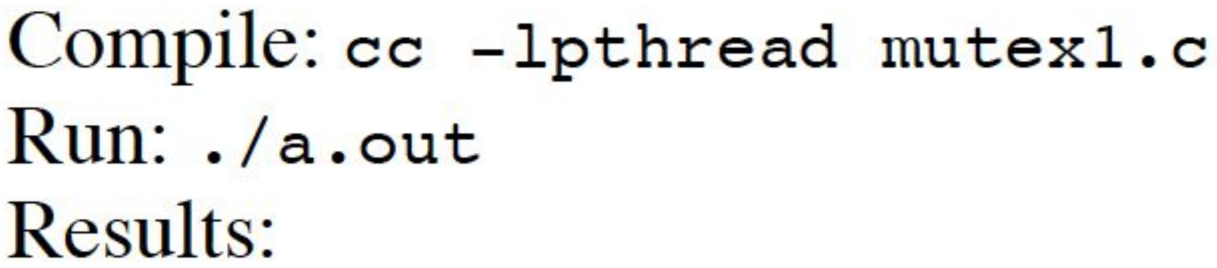
```
{
   printf("Thread creation failed: %d\n", rc1);
}

if( (rc2=pthread_create( &thread2, NULL, &functionC, NULL)) )
{
   printf("Thread creation failed: %d\n", rc2);
}

/* Wait till threads are complete before main continues. Unless we  */
/* wait we run the risk of executing an exit which will terminate   */
/* the process and all threads before the threads have completed.   */

pthread_join( thread1, NULL);
pthread_join( thread2, NULL);

exit(0);
```

Function ()



```
void *functionC()
{
   pthread_mutex_lock( &mutex1 );
   counter++;
   printf("Counter value: %d\n",counter);
   pthread_mutex_unlock( &mutex1 );
}
```

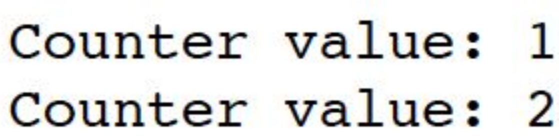
Output:



Compile: `cc -lpthread mutex1.c`

Run: `./a.out`

Results:



```
Counter value: 1
Counter value: 2
```

Condition Variable

- A condition variable is a variable of type pthread_cond_t and is used with the appropriate functions for waiting and later, process continuation.
- The condition variable mechanism allows threads to suspend execution and relinquish the processor until some condition is true.
- A condition variable must always be associated with a mutex to avoid a race condition.

Condition Variable: Creating, Destroying & Waiting Function

Creating :

- pthread_cond_init
- pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

Destroying:

- pthread_cond_destroy

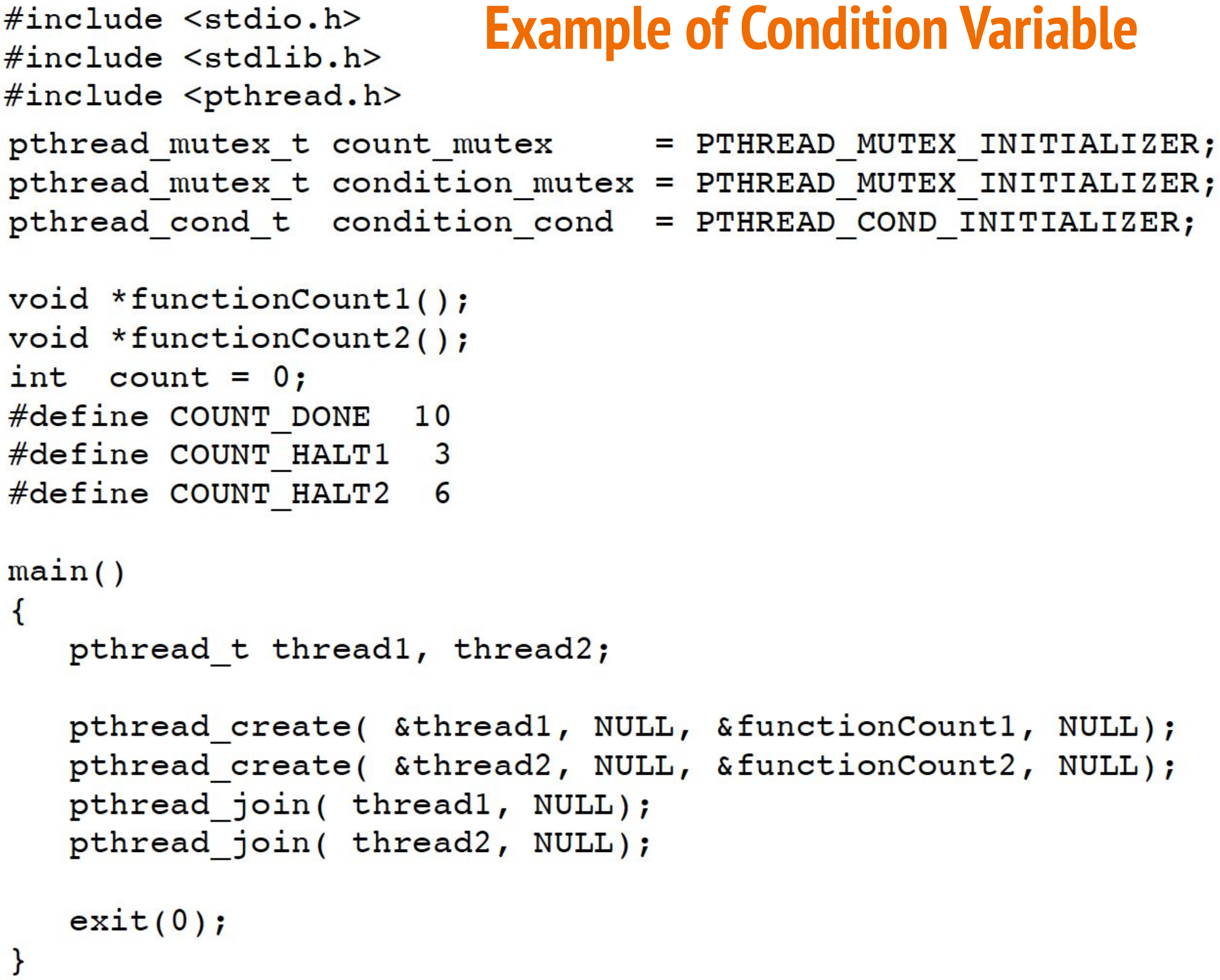
Waiting on condition:

- pthread_cond_wait
- pthread_cond_timedwait - place limit on how long it will block.

Waking thread based on condition:

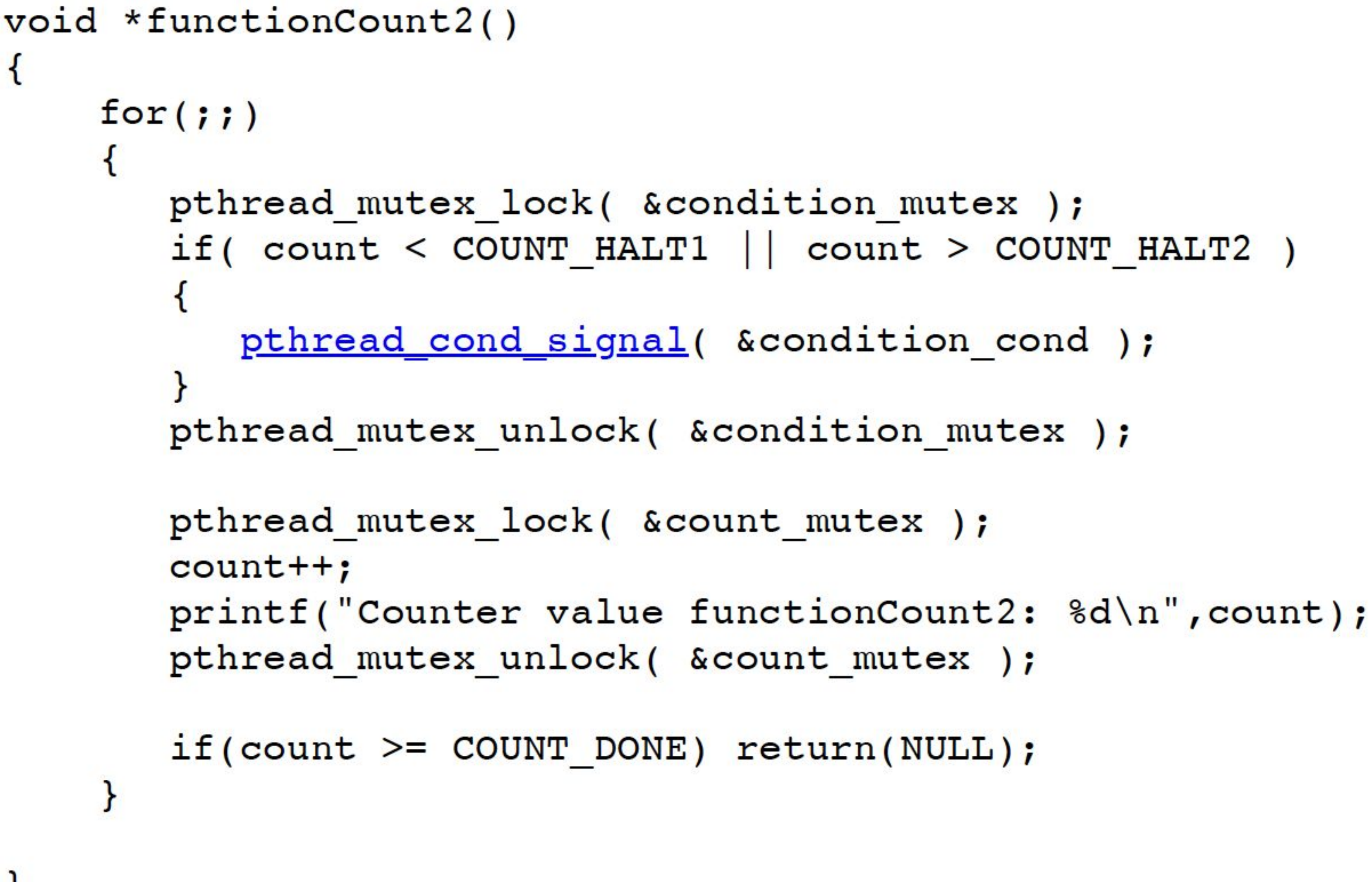
- pthread_cond_signal
- pthread_cond_broadcast - wake up all threads blocked by the specified condition variable

Example of Condition Variable



```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
pthread_mutex_t count_mutex     = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t condition_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  condition_cond  = PTHREAD_COND_INITIALIZER;

void *functionCount1();
void *functionCount2();
int  count = 0;
#define COUNT_DONE  10
#define COUNT_HALT1  3
#define COUNT_HALT2  6

main()
{
   pthread_t thread1, thread2;

   pthread_create( &thread1, NULL, &functionCount1, NULL);
   pthread_create( &thread2, NULL, &functionCount2, NULL);
   pthread_join( thread1, NULL);
   pthread_join( thread2, NULL);

   exit(0);
}
```

```
void *functionCount1()
{
   for(;;)
   {
      pthread_mutex_lock( &condition_mutex );
      while( count >= COUNT_HALT1 && count <= COUNT_HALT2 )
      {
         pthread_cond_wait( &condition_cond, &condition_mutex );
      }
      pthread_mutex_unlock( &condition_mutex );

      pthread_mutex_lock( &count_mutex );
      count++;
      printf("Counter value functionCount1: %d\n",count);
      pthread_mutex_unlock( &count_mutex );

      if(count >= COUNT_DONE) return(NULL);
    }
}
```



```
void *functionCount2()
{
    for(;;)
    {
       pthread_mutex_lock( &condition_mutex );
       if( count < COUNT_HALT1 || count > COUNT_HALT2 )
       {
          pthread_cond_signal( &condition_cond );
       }
       pthread_mutex_unlock( &condition_mutex );

       pthread_mutex_lock( &count_mutex );
       count++;
       printf("Counter value functionCount2: %d\n",count);
       pthread_mutex_unlock( &count_mutex );

       if(count >= COUNT_DONE) return(NULL);
    }

}
```

Question?

thank you