Systems Software

Week 9: Threading and Concurrency

Notes By: Jonathan McCarthy

Overview

- Threading
- Synchronisation and Concurrency
- POSIX Threading
- Thread Example
- Mutex file locking
- Locking Example

Introduction to Threading

- A thread can be thought of as the path of a programs execution.
- The programs that we have seen to date all ran in a single thread.
- If we are dealing with a large problem, this can be sub-devided into smaller parts and execute them in different threads concurrently. This is known as multithreading.

Threading in C

- C programming has multithreading support.
- A multithreaded program contains two or mote parts that will run concurrently in separate threads.
- Each thread has a separate path of execution.
- Multithreading could be described as multitasking.

Types of Multithreading

- There are two types of multitasking:
  - Process Based
  - Thread Based
- Process based multitasking allows a computer to run multiple applications at the same time (eg. Word and PowerPoint etc..)
- Thread based multitasking allows a C program to perform two or more tasks at once. This can make good use of the hardware the program is running on (eg. multicore CPU).

User Threads

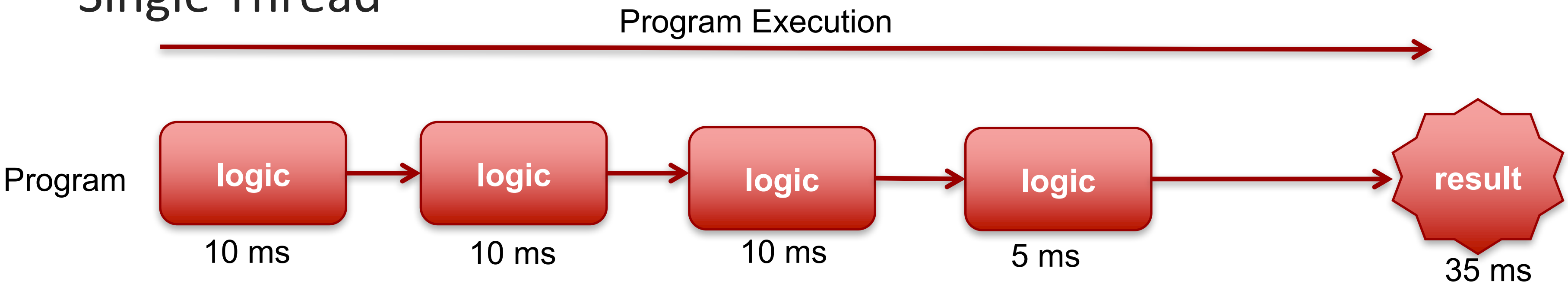
- User level threads are mostly at the application level where an application creates these threads to sustain its execution in the main memory.
- User threads work in isolation with kernel threads.
- These are easier to create since they do not have to refer any registers and context switching is much faster than a kernel level thread.
- User level thread, mostly can cause changes at the application level and has no impact on kernel threads.

Kernel Threads

- Kernel threads are mostly independent of the ongoing processes and are executed by the operating system.
- Kernel threads are used by the Operating System for management tasks etc…..
- Kernel threads are more expensive to create and manage and context switching of these threads are slow.
- Most of the kernel level threads can not be preempted by the user level threads.

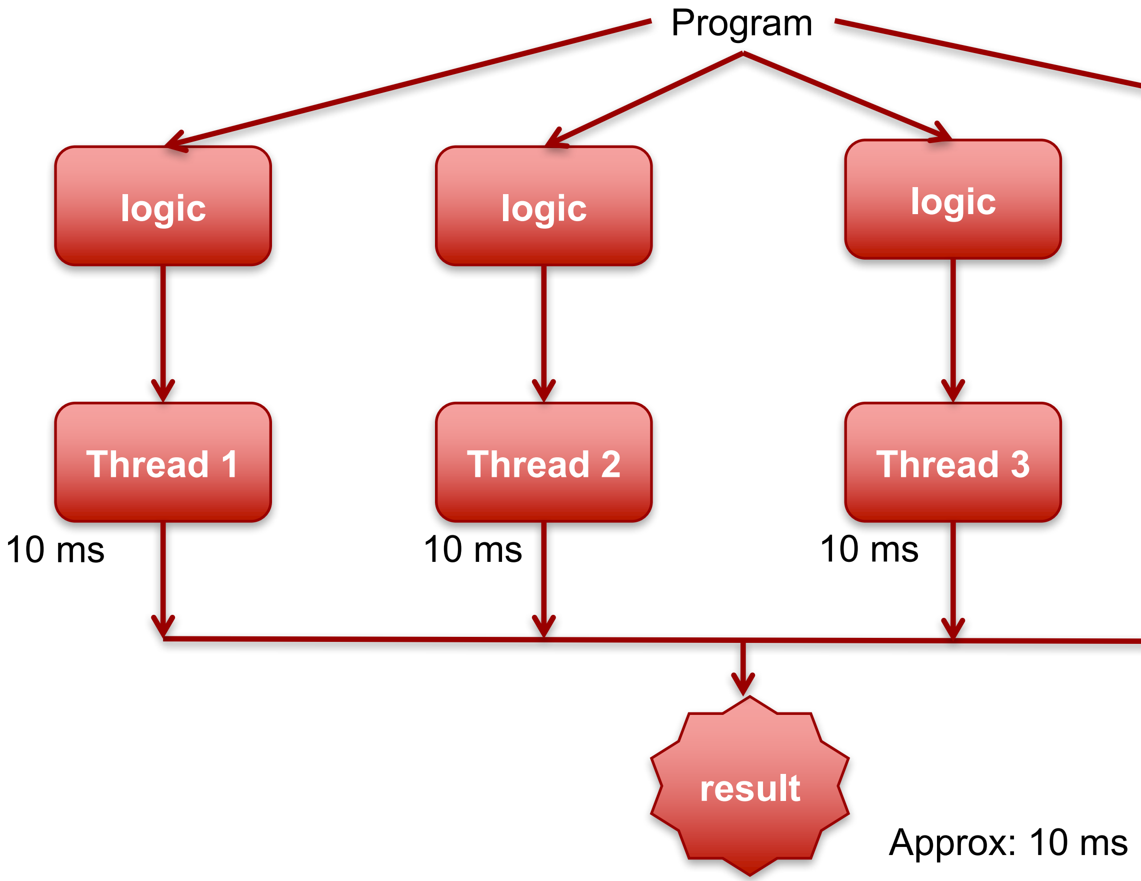
Multithreading Fundamentals

Single Thread

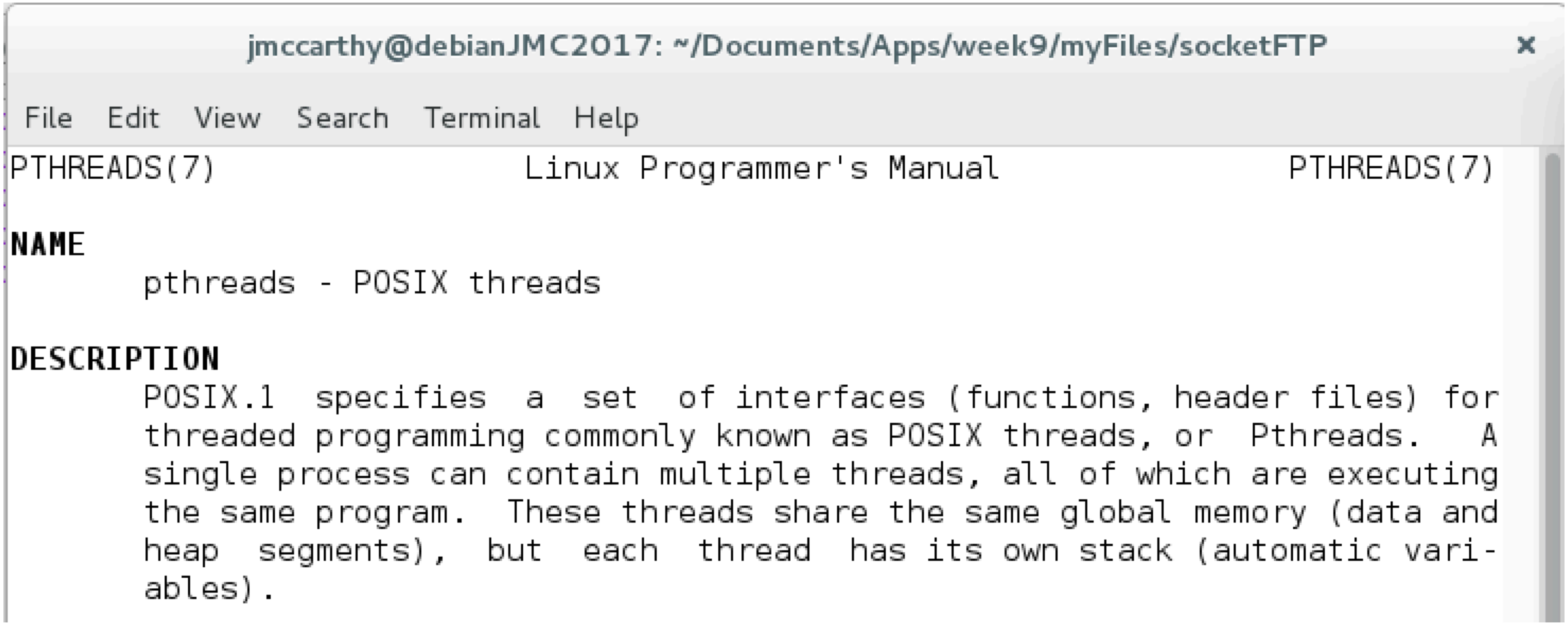


Multithreading Fundamentals

- Multithread



POSIX threads - pthreads



jmccarthy@debianJMC2017: ~/Documents/Apps/week9/myFiles/socketFTP
×
File Edit View Search Terminal Help

```
PTHREADS(7)                Linux Programmer's Manual               PTHREADS(7)

NAME
       pthreads - POSIX threads

DESCRIPTION
       POSIX.1  specifies  a  set  of interfaces (functions, header files) for
       threaded programming commonly known as POSIX threads, or  Pthreads.   A
       single process can contain multiple threads, all of which are executing
       the same program.  These threads share the same global memory (data and
       heap  segments),  but  each  thread  has its own stack (automatic vari-
       ables).
```

What can be shared in a thread?

- POSIX.1 also requires that threads share a range of other attributes (i.e., these attributes are process-wide rather than per-thread):
  - process ID
  - parent process ID
  - process group ID and session ID
  - controlling terminal
  - user and group IDs
  - open file descriptors
  - record locks (see fcntl(2))
  - signal dispositions
  - file mode creation mask (umask(2))

What can be shared in a thread?

- current directory (chdir(2)) and root directory (chroot(2))
- interval timers (setitimer(2)) and POSIX timers (timer_create(2))
- nice value (setpriority(2))
- resource limits (setrlimit(2))
- measurements of the consumption of CPU time (times(2)) and resources (getrusage(2))

Thread Primitives

Process Primitive	Thread Primitive	Description
fork	pthread_create	Create a new flow of control
waitpid	pthread_join	Get exit status
exit	pthread_exit	Exit current code execution
getpid	pthread_self	Get ID
abort	pthread_cancel	Request abort of execution

Creating a thread

- #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.

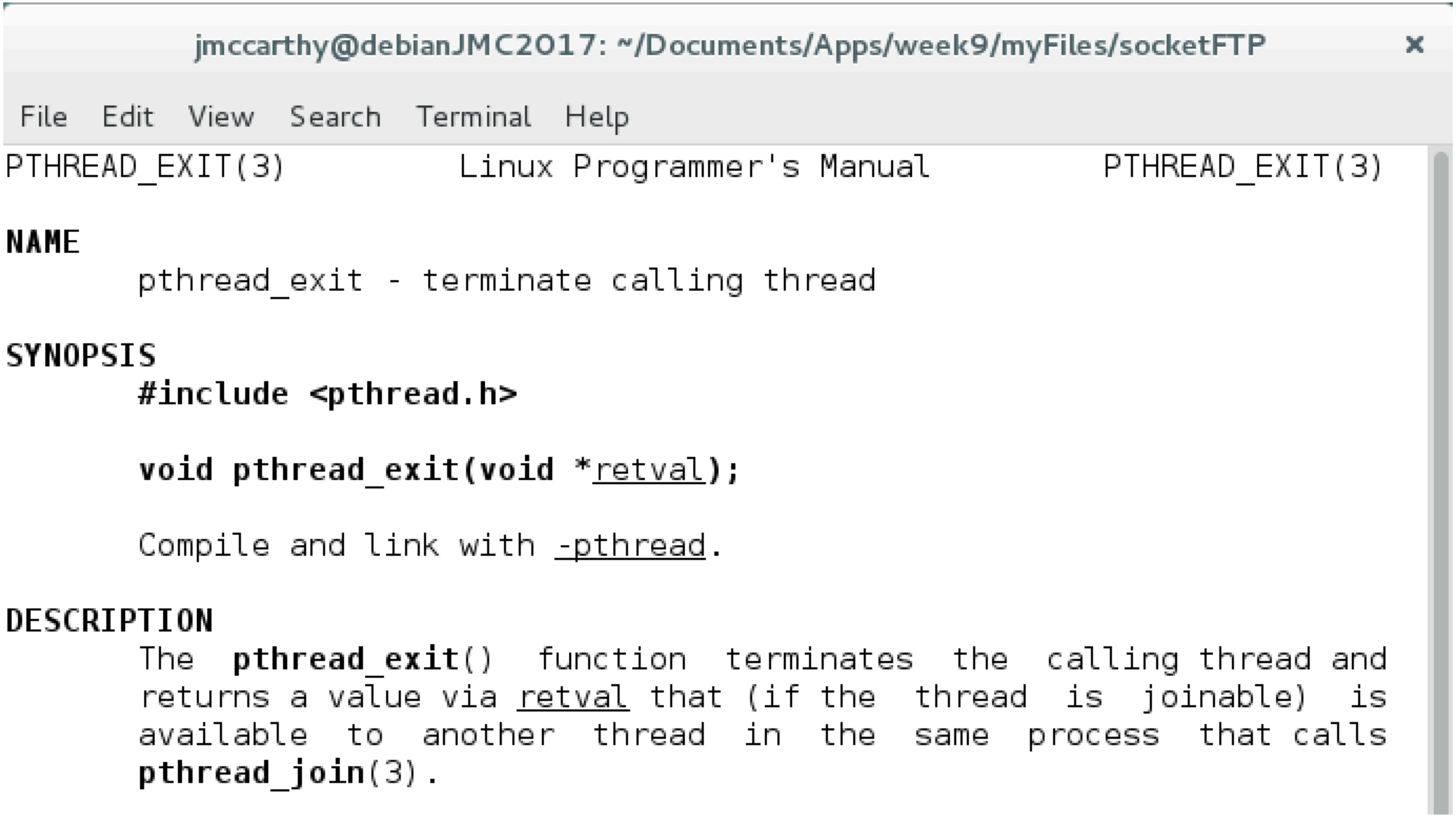
Creating a thread – where…

- **pthread_t *thread** --> sets and returns the id of the newly created thread
- **const pthread_attr_t *attr** --> attributes to configure the thread
- **void *(*start_routine) (void *)** --> function that will run in the thread
- **void *arg** --> variables to be passed to the function for use in the thread of execution

Thread Termination

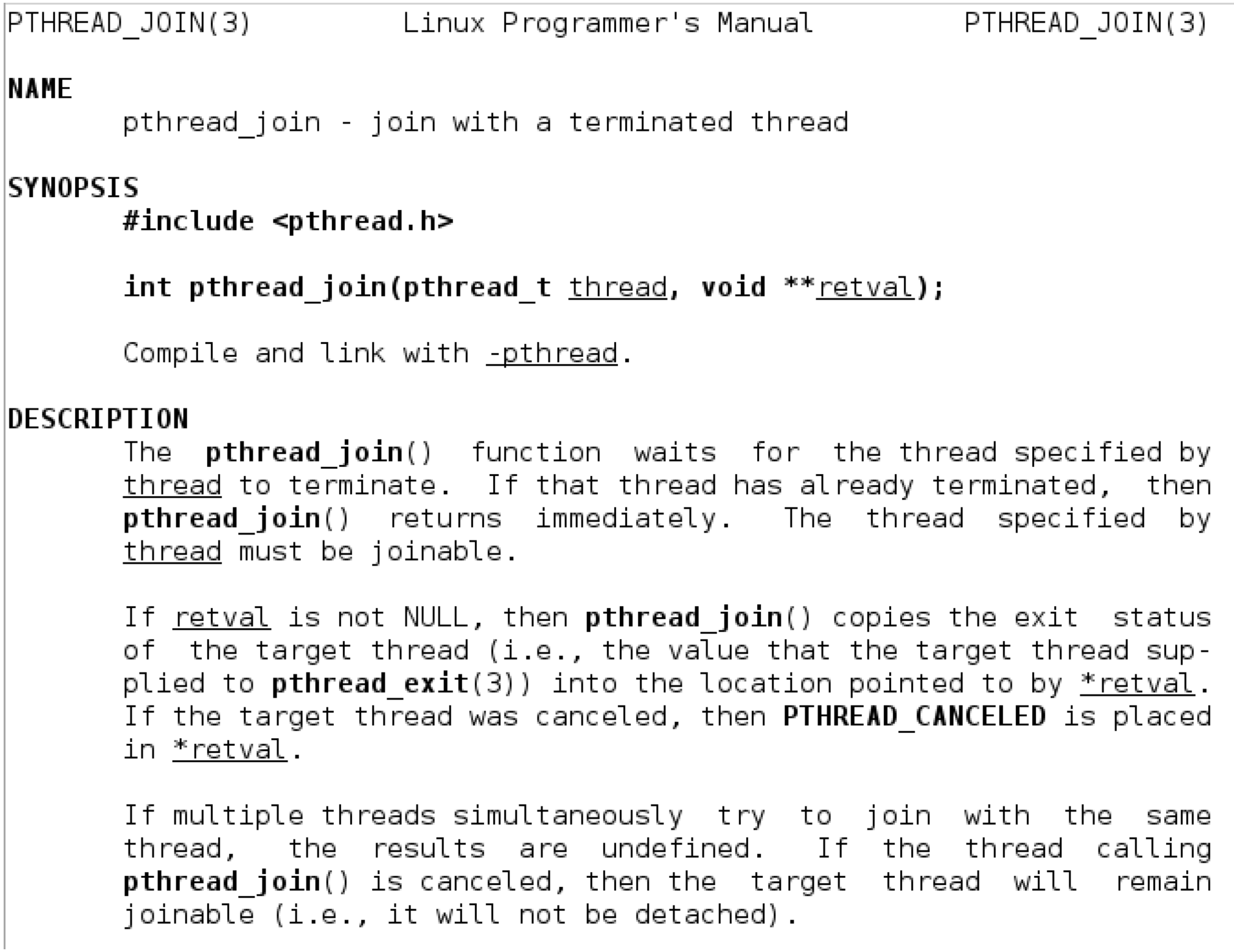
- There are three options for a thread to terminate:
- The thread can return from the start routine. This can return the threads exit code.
- The thread can be stopped by another thread in the same pool/process.
- The thread can call pthread_exit()

Thread Termination


jmccarthy@debianJMC2017: ~/Documents/Apps/week9/myFiles/socketFTP
×
File Edit View Search Terminal Help

```
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).
```

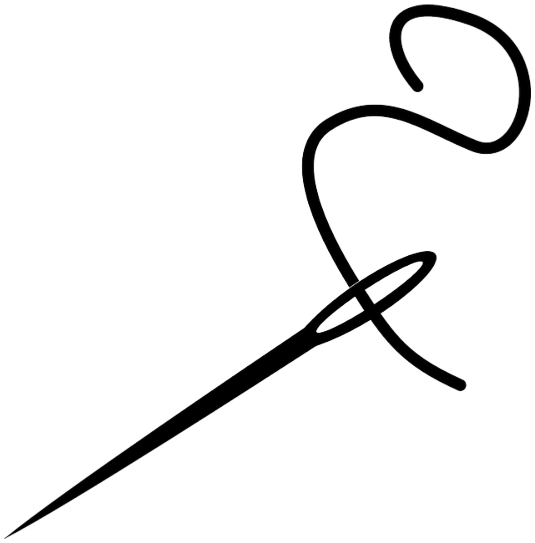
Thread Join

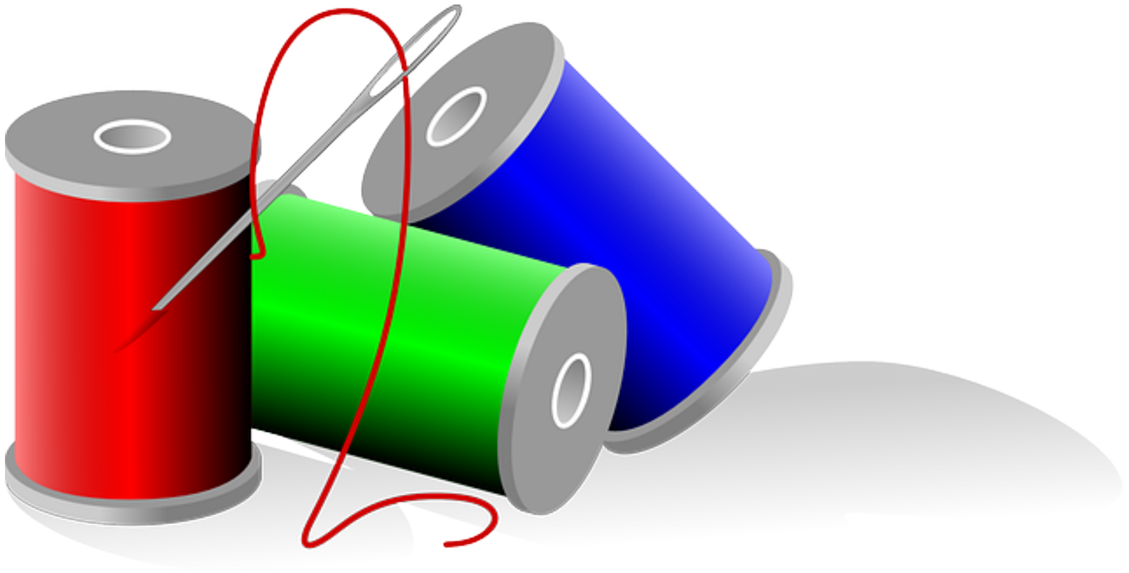


```
PTHREAD_JOIN(3)          Linux Programmer's Manual          PTHREAD_JOIN(3)

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 sup-
       plied to pthread_exit(3)) into the location pointed to by *retval.
       If the target thread was canceled, then PTHREAD_CANCELED is placed
       in *retval.

       If multiple threads simultaneously  try  to  join  with  the  same
       thread,   the  results  are  undefined.   If  the  thread  calling
       pthread_join() is canceled, then the  target  thread  will  remain
       joinable (i.e., it will not be detached).
```

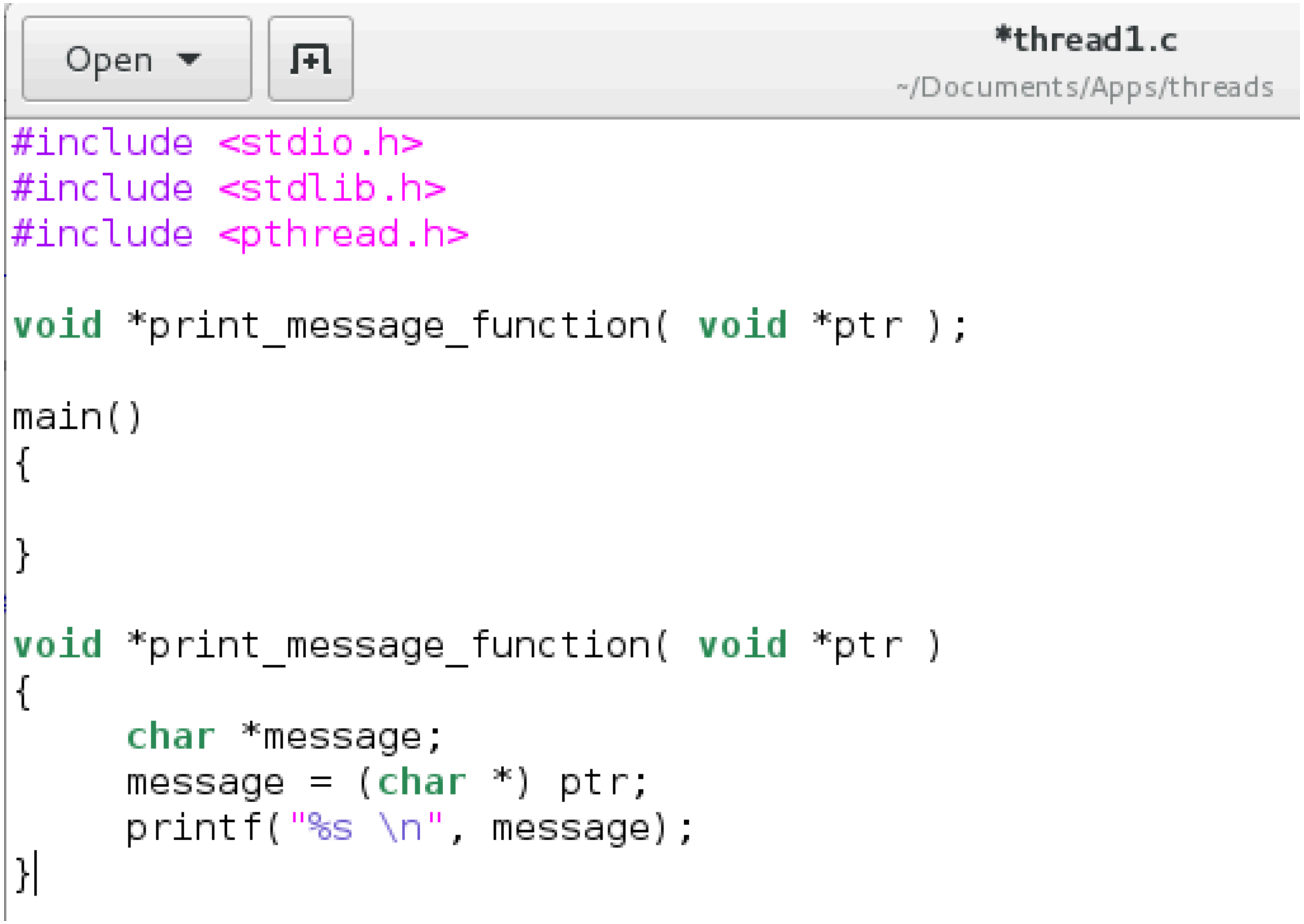
Simple Thread Example

- Create a simple C program to demonstrate two threads running concurrently.





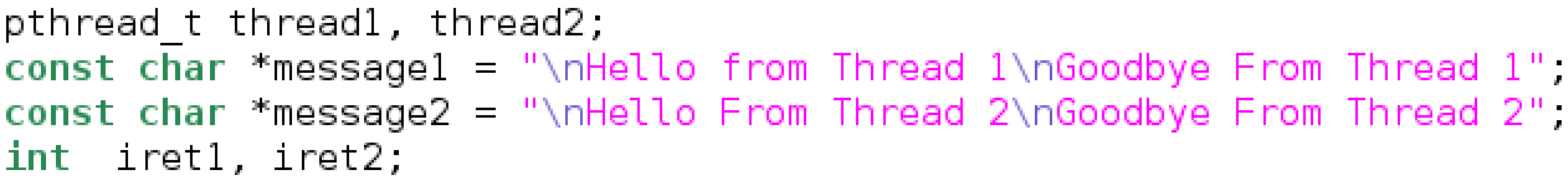
Example 1



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

void *print_message_function( void *ptr );

main()
{

}

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

Example 1 – main() code

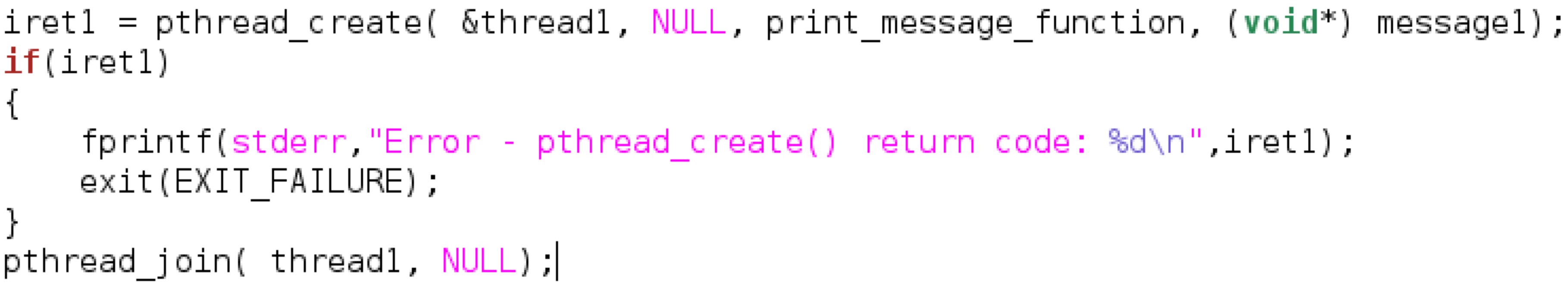
- // Initialise Variables



```
pthread_t thread1, thread2;
const char *message1 = "\nHello from Thread 1\nGoodbye From Thread 1";
const char *message2 = "\nHello From Thread 2\nGoodbye From Thread 2";
int  iret1, iret2;
```

Example 1 – main() – Thread 1

- // Thread 1

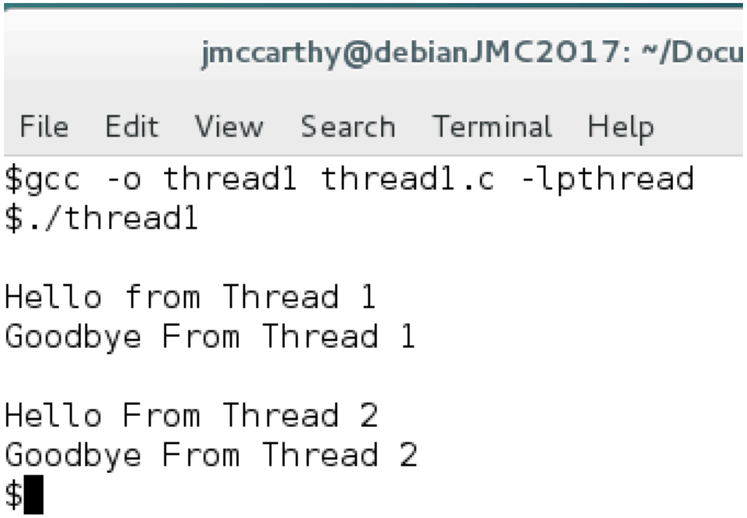


```
iret1 = pthread_create( &thread1, NULL, print_message_function, (void*) message1);
if(iret1)
{
    fprintf(stderr,"Error - pthread_create() return code: %d\n",iret1);
    exit(EXIT_FAILURE);
}
pthread_join( thread1, NULL);
```

Example 1 – main() – Thread 2

- // Thread 2

```
iret2 = pthread_create( &thread2, NULL, print_message_function, (void*) message2);
if(iret2)
{
    fprintf(stderr,"Error - pthread_create() return code: %d\n",iret2);
    exit(EXIT_FAILURE);
}
pthread_join( thread2, NULL);

exit(EXIT_SUCCESS);
```



```
$gcc -o thread1 thread1.c -lpthread
$./thread1

Hello from Thread 1
Goodbye From Thread 1

Hello From Thread 2
Goodbye From Thread 2
$
```

Thread Synchronisation

- If a program is using more than one thread, the threads may be sharing the same resources which can lead to inconsistencies in the program.
- Example:
- Create a program to manage the launch sequences for space shuttles. It must be possible to schedule the launch of multiple shuttles at once. The components needed to run the launch countdown is shared amongst the shuttles.

Protecting Shared Resources

- The process running in a given thread may need to access a resource that will be used by all other threads currently running.
- In certain circumstances this may not be desirable, if the thread reads the same data, potentially a problem could arise if the data changes.
- It is unsafe to facilitate concurrent access to a shared resource reads and modifies data.
- A mechanism to to block access to the resource is needed if a thread is using the resource.

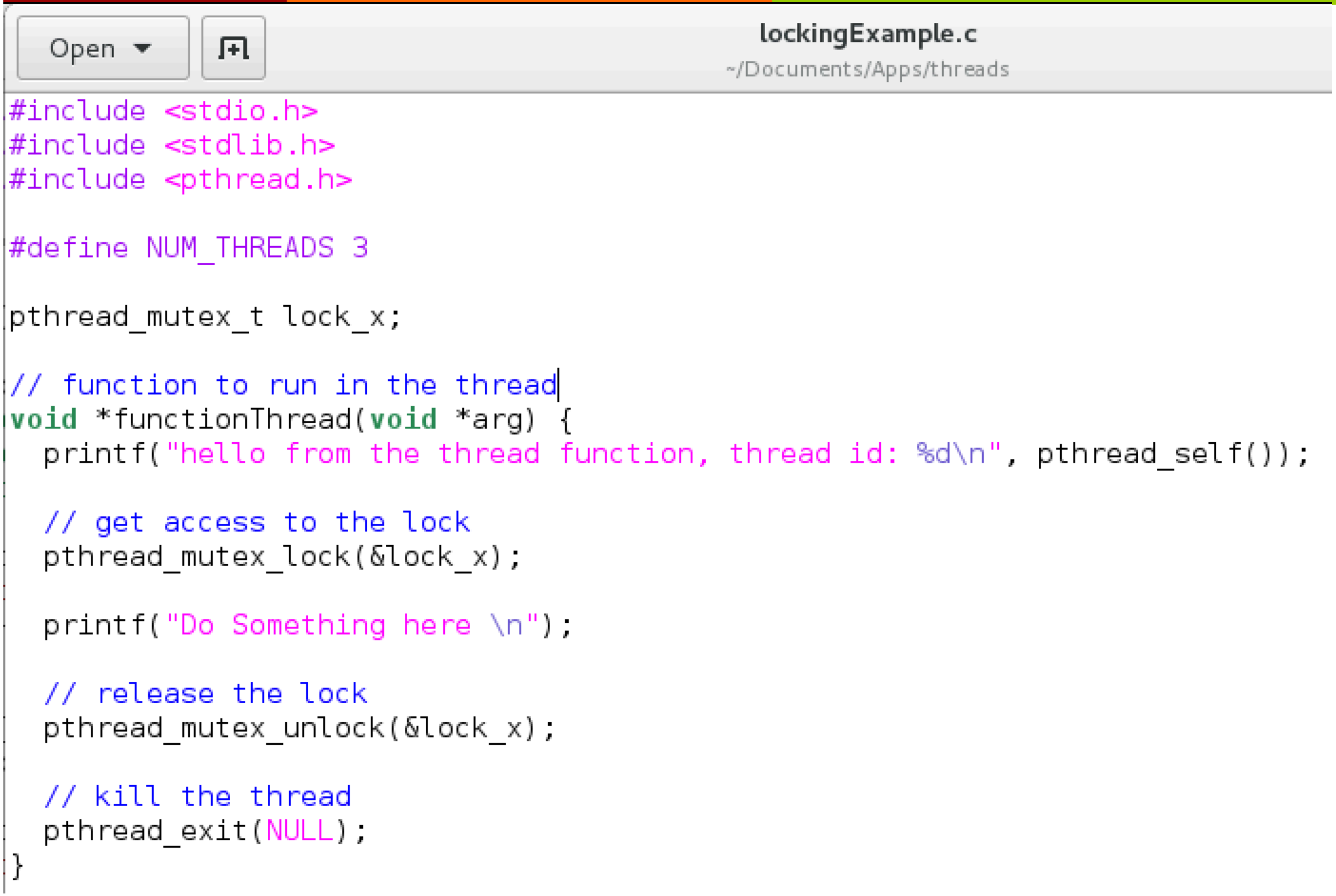
Mutex File Locking

- A mutex is a lock that can be attached to a given resource.
- If a thread needs to modify data in a shared resource, the thread must first obtain access to the lock.
- All other threads cannot use the shared resource until it has been released.
- Different algorithms can be used to control access to the shared resource. (FIFO Queues etc…)

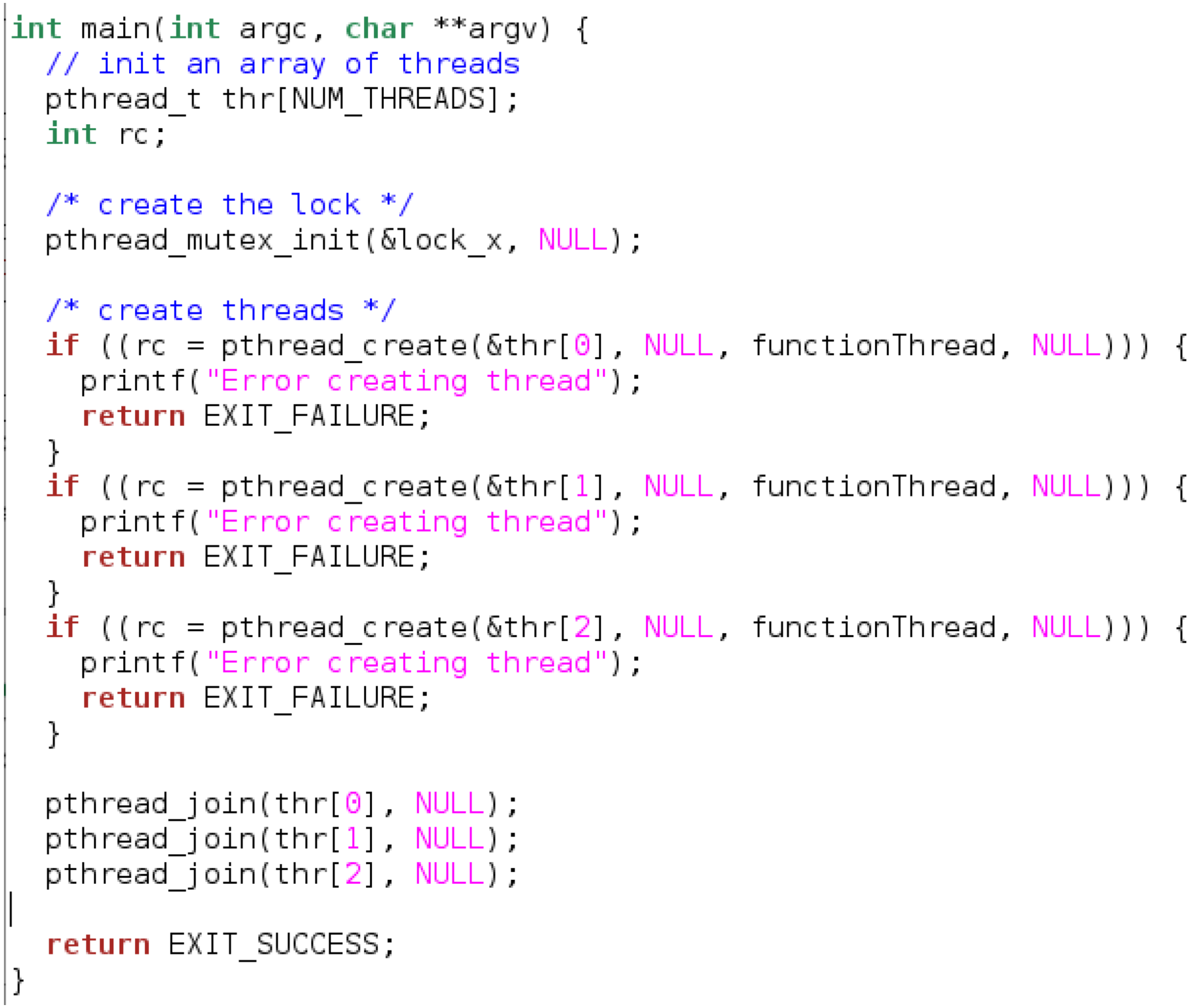
Mandatory File Locking

- Mandatory locking is kernel enforced file locking
- This differs from the standard cooperative file locking for sequential access.
- File locks are applied using the flock() and fcntl() system calls
- A process must check for locks on a file it wishes to update, before applying its own lock, updating the file and unlocking it again.
- Issues exist with this mechanism and should be avoided where possible.

Lock Example

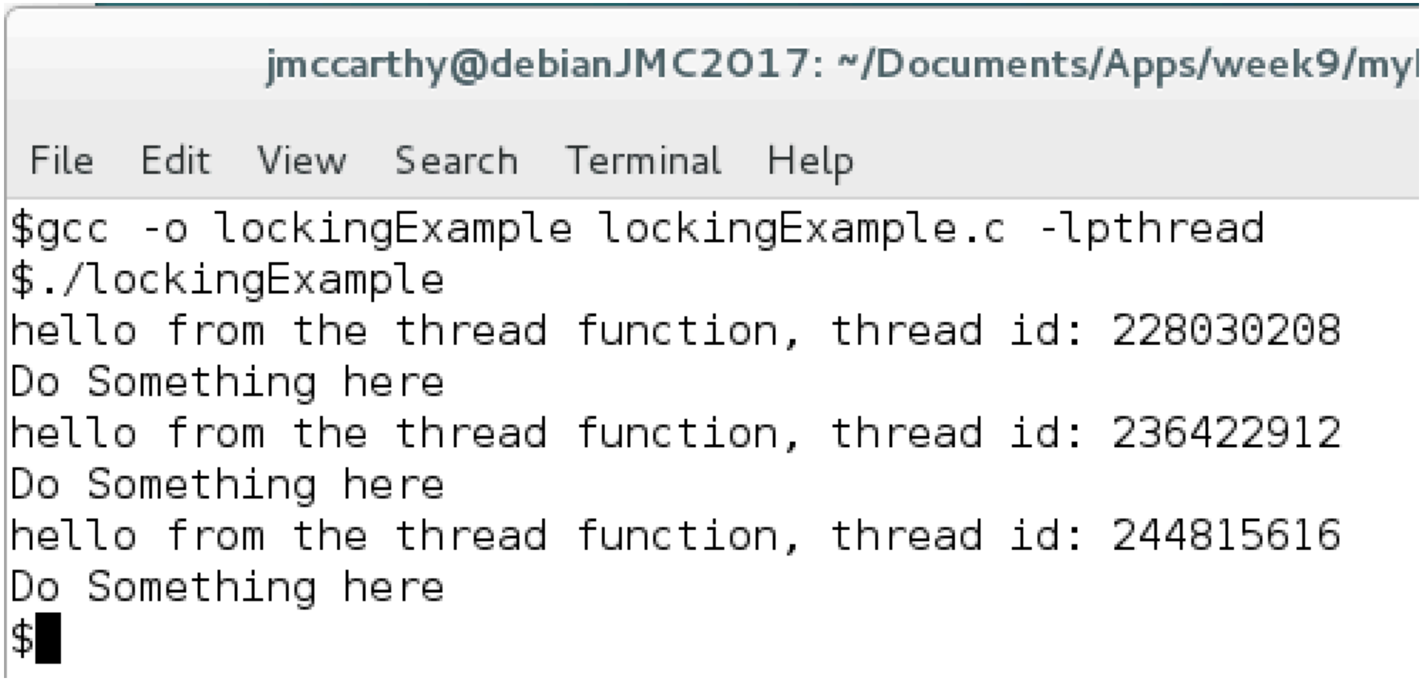


```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define NUM_THREADS 3

pthread_mutex_t lock_x;

// function to run in the thread
void *functionThread(void *arg) {
  printf("hello from the thread function, thread id: %d\n", pthread_self());

  // get access to the lock
  pthread_mutex_lock(&lock_x);

  printf("Do Something here \n");

  // release the lock
  pthread_mutex_unlock(&lock_x);

  // kill the thread
  pthread_exit(NULL);
}
```



```c
int main(int argc, char **argv) {
  // init an array of threads
  pthread_t thr[NUM_THREADS];
  int rc;

  /* create the lock */
  pthread_mutex_init(&lock_x, NULL);

  /* create threads */
  if ((rc = pthread_create(&thr[0], NULL, functionThread, NULL))) {
    printf("Error creating thread");
    return EXIT_FAILURE;
  }
  if ((rc = pthread_create(&thr[1], NULL, functionThread, NULL))) {
    printf("Error creating thread");
    return EXIT_FAILURE;
  }
  if ((rc = pthread_create(&thr[2], NULL, functionThread, NULL))) {
    printf("Error creating thread");
    return EXIT_FAILURE;
  }

  pthread_join(thr[0], NULL);
  pthread_join(thr[1], NULL);
  pthread_join(thr[2], NULL);

  return EXIT_SUCCESS;
}
```

Locking Example


jmccarthy@debianJMC2017: ~/Documents/Apps/week9/my
File Edit View Search Terminal Help

```
$gcc -o lockingExample lockingExample.c -lpthread
$./lockingExample
hello from the thread function, thread id: 228030208
Do Something here
hello from the thread function, thread id: 236422912
Do Something here
hello from the thread function, thread id: 244815616
Do Something here
$
```

Questions

Notes By: Jonathan McCarthy