Thread Programming (Linux)

Thread Programming

- http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html#BASICS

Example

```
#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); }

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

Compile

Compile:

C compiler: cc -lpthread pthread1.c

or

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

Run: ./a.out

Results:

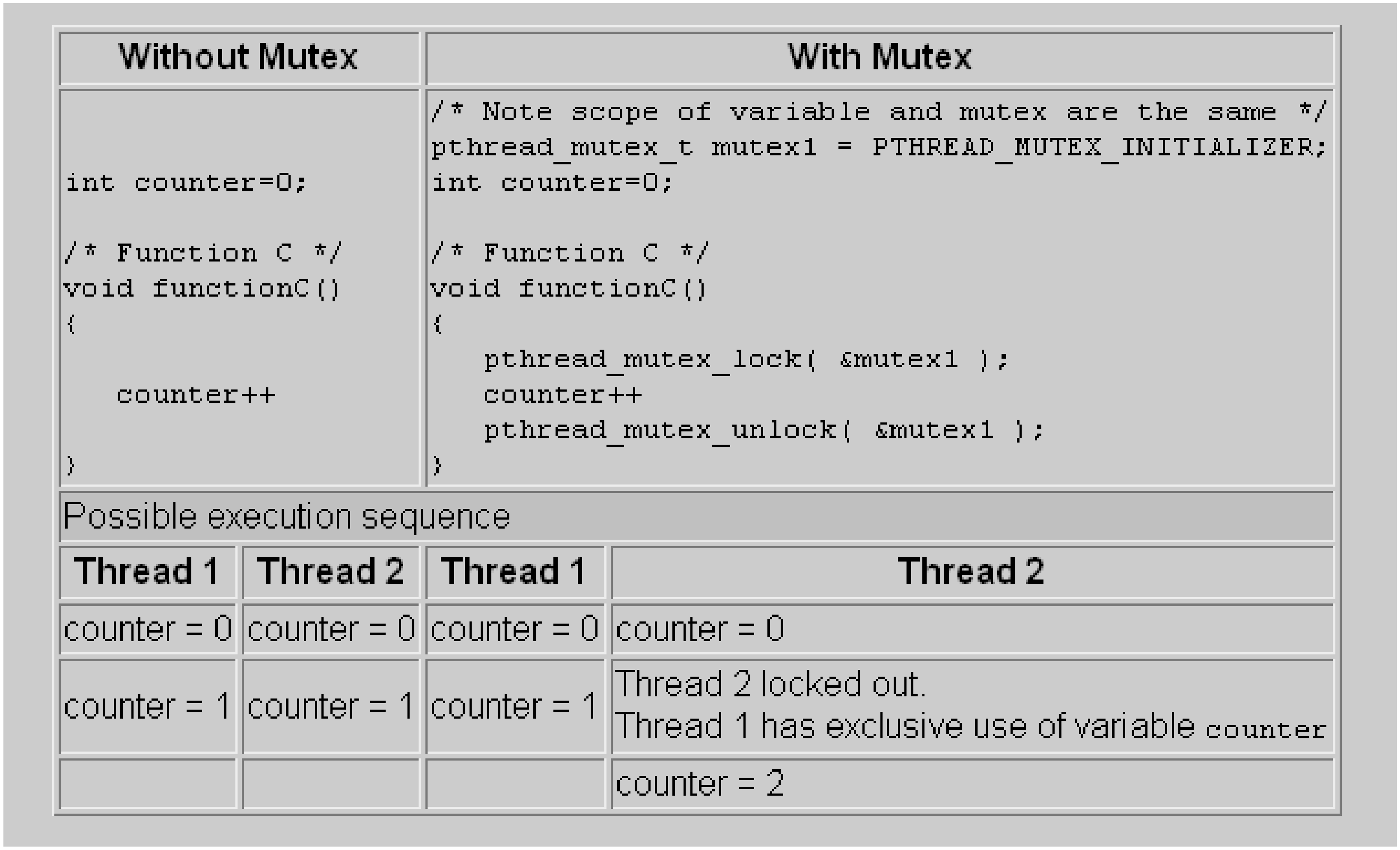
Thread 1 Thread 2 Thread 1 returns: 0 Thread 2 returns: 0

Details:

Thread Synchronization

Mutexes - Mutual exclusion lock: Block access to variables by other threads. This enforces exclusive access by a thread to a variable or set of variables.

Mutexes



Without Mutex

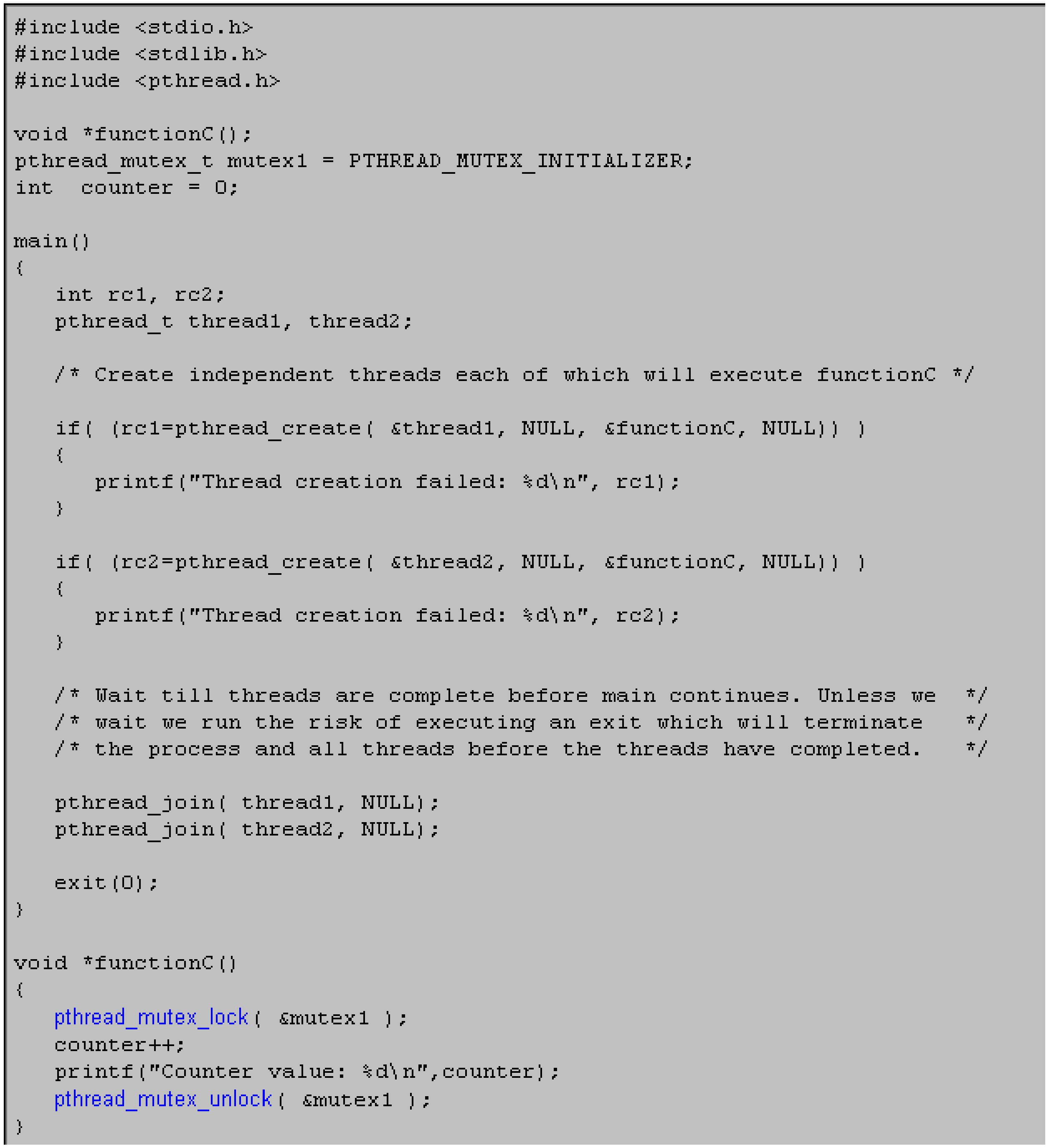
```
int counter=0;

/* Function C */
void functionC()
{

   counter++

}
```

With Mutex

```
/* Note scope of variable and mutex are the same */
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
int counter=0;

/* Function C */
void functionC()
{
   pthread_mutex_lock( &mutex1 );
   counter++
   pthread_mutex_unlock( &mutex1 );
}
```

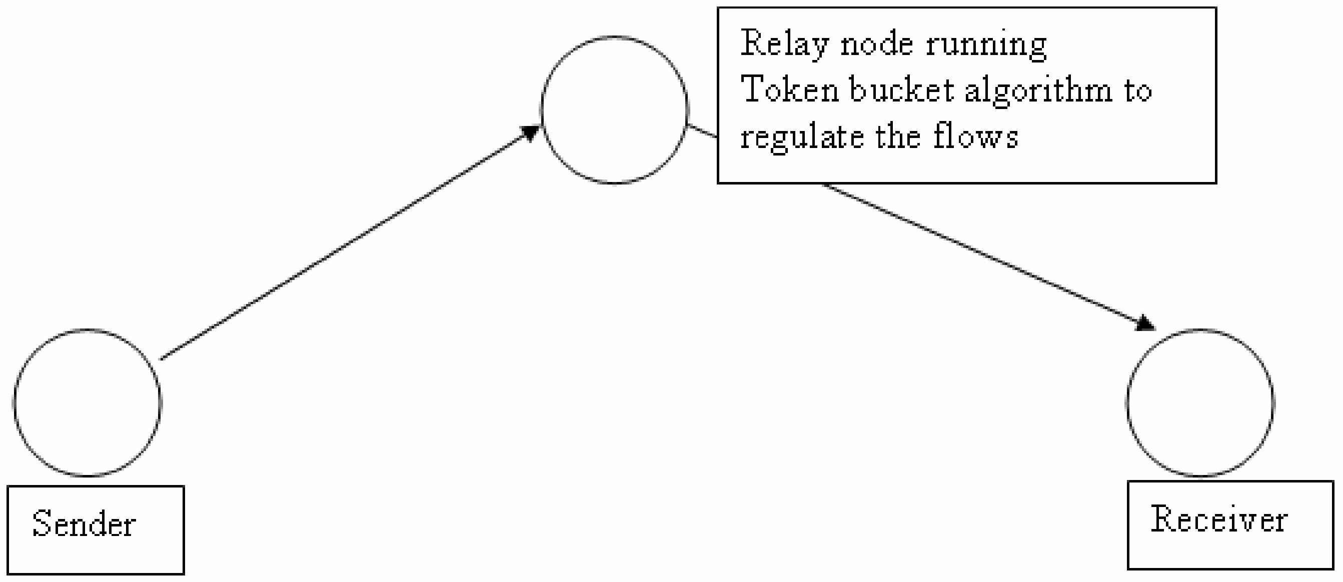
Possible execution sequence

Thread 1	Thread 2	Thread 1	Thread 2
counter = 0	counter = 0	counter = 0	counter = 0
counter = 1	counter = 1	counter = 1	Thread 2 locked out. Thread 1 has exclusive use of variable `counter`
			counter = 2



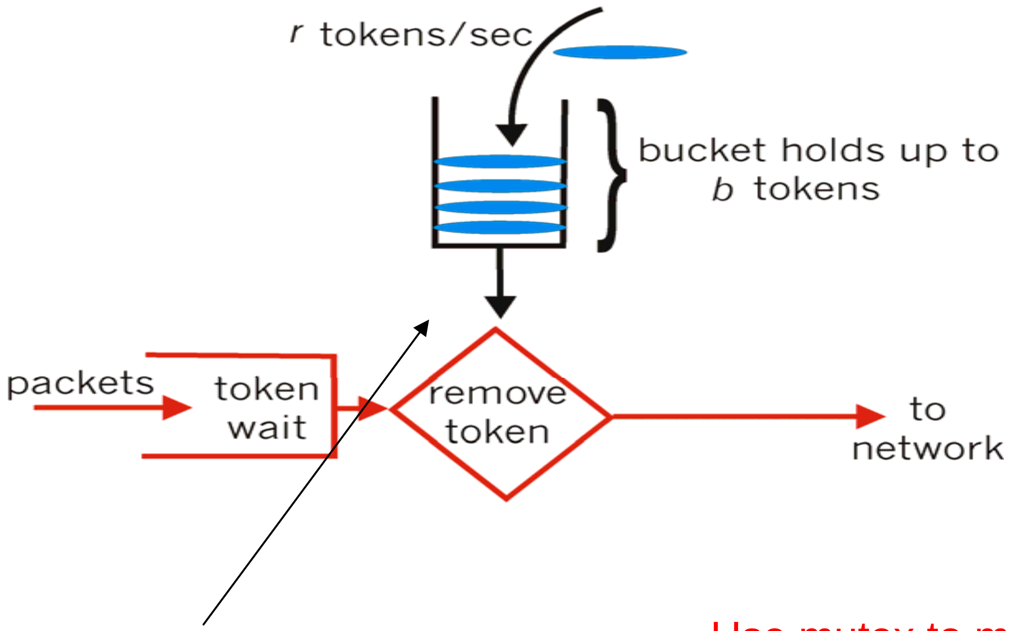
```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

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);
}

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

Homework 4


Relay node running
Token bucket algorithm to
regulate the flows
Sender
Receiver

Implement Token Bucket Algorithm at Relay Node

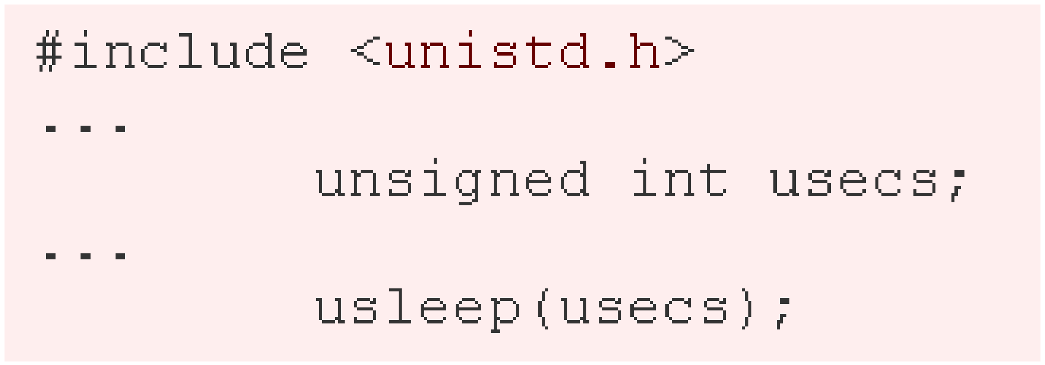
Thread 1: Filling the bucket by increasing the counter value at every interval of time. Stop when the counter value reaches b.



Thread 2 (main thread): For every arrival packet, remove a token from a bucket and sends the packet out. If the bucket is empty, discards the arrival packet.

Use mutex to modify the value of the counter!

To control the sending rate or the token filling rate



```
#include <unistd.h>
...
        unsigned int usecs;
...
        usleep(usecs);
```