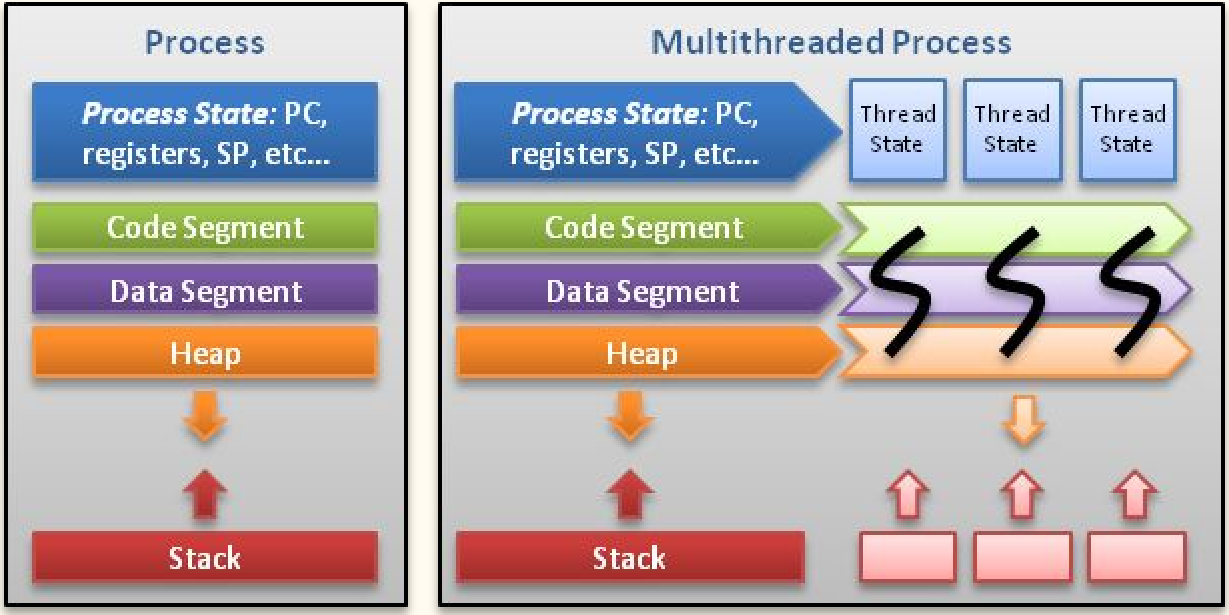
Recitation 9

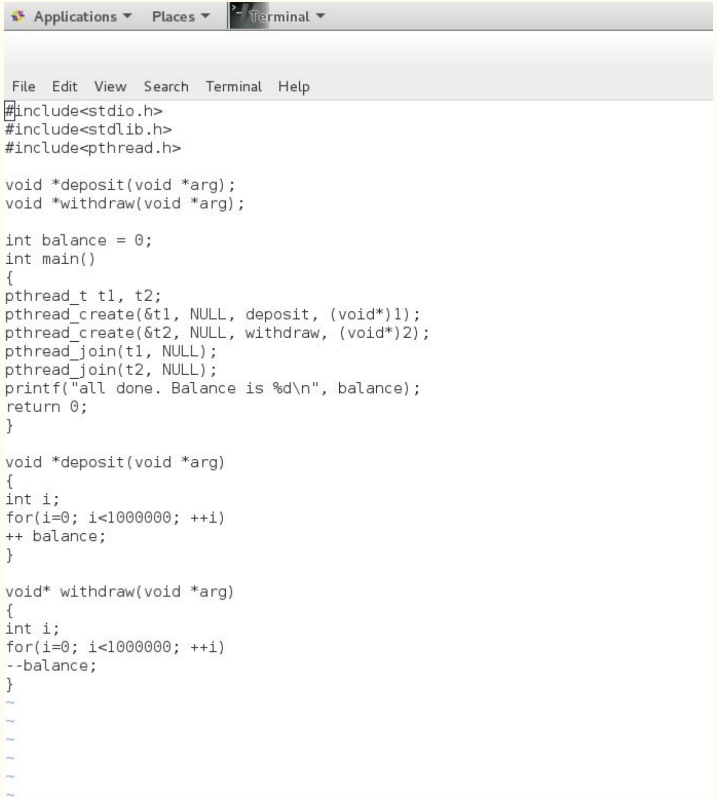
Synchronization

Single Thread v/s Multithreaded Programs

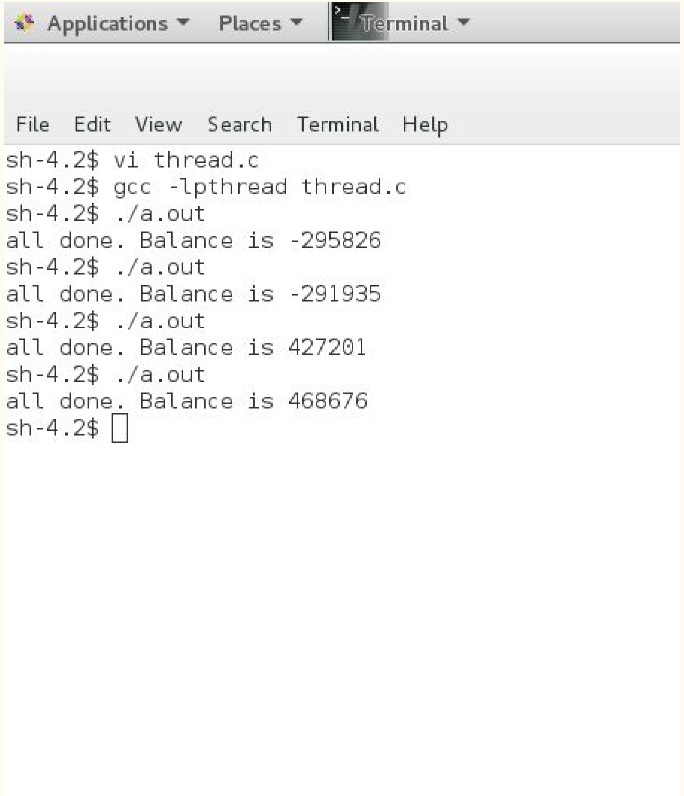


Threads contain only necessary information, such as a stack (for local variables, function arguments, return values), a copy of the registers, program counter and any thread-specific data to allow them to be scheduled individually. Other data is shared within the process between all threads.

Synchronization Problems



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

void *deposit(void *arg);
void *withdraw(void *arg);

int balance = 0;
int main()
{
pthread_t t1, t2;
pthread_create(&t1, NULL, deposit, (void*)1);
pthread_create(&t2, NULL, withdraw, (void*)2);
pthread_join(t1, NULL);
pthread_join(t2, NULL);
printf("all done. Balance is %d\n", balance);
return 0;
}

void *deposit(void *arg)
{
int i;
for(i=0; i<1000000; ++i)
++ balance;
}

void* withdraw(void *arg)
{
int i;
for(i=0; i<1000000; ++i)
--balance;
}
~
~
~
~
~
~
```

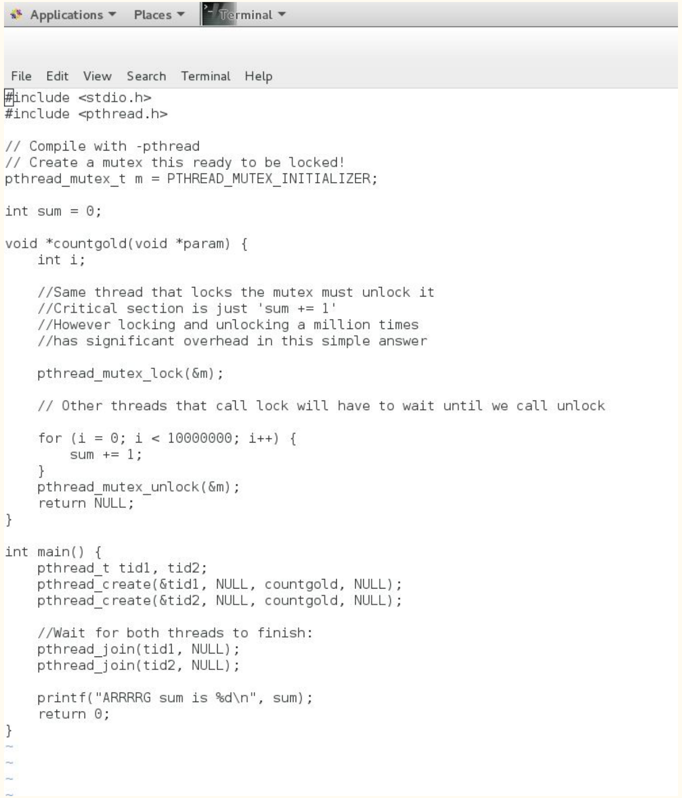


```
sh-4.2$ vi thread.c
sh-4.2$ gcc -lpthread thread.c
sh-4.2$ ./a.out
all done. Balance is -295826
sh-4.2$ ./a.out
all done. Balance is -291935
sh-4.2$ ./a.out
all done. Balance is 427201
sh-4.2$ ./a.out
all done. Balance is 468676
sh-4.2$ 
```

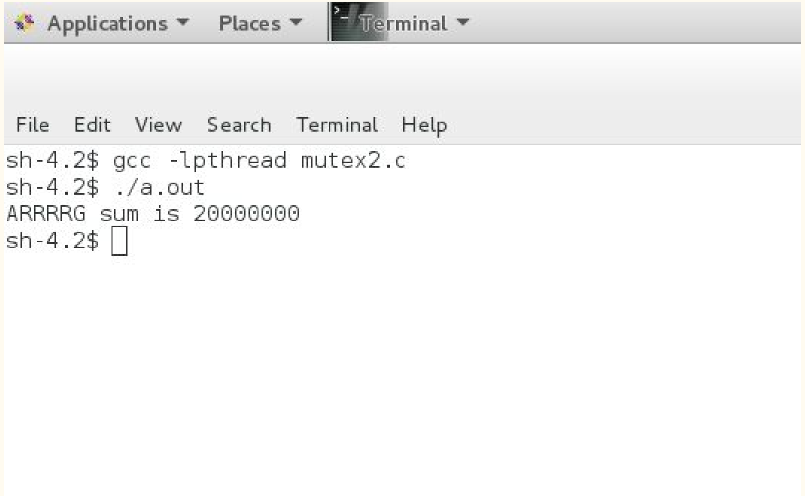
Race Condition

1. Definition: A timing dependent error that involves a shared resource.
2. Can be very bad:
   a. Non-deterministic: Don’t know what the output will be and it is likely to be different across different runs.
   b. Hard to detect: Too many possible schedules.
   c. Hard to debug

Mutex

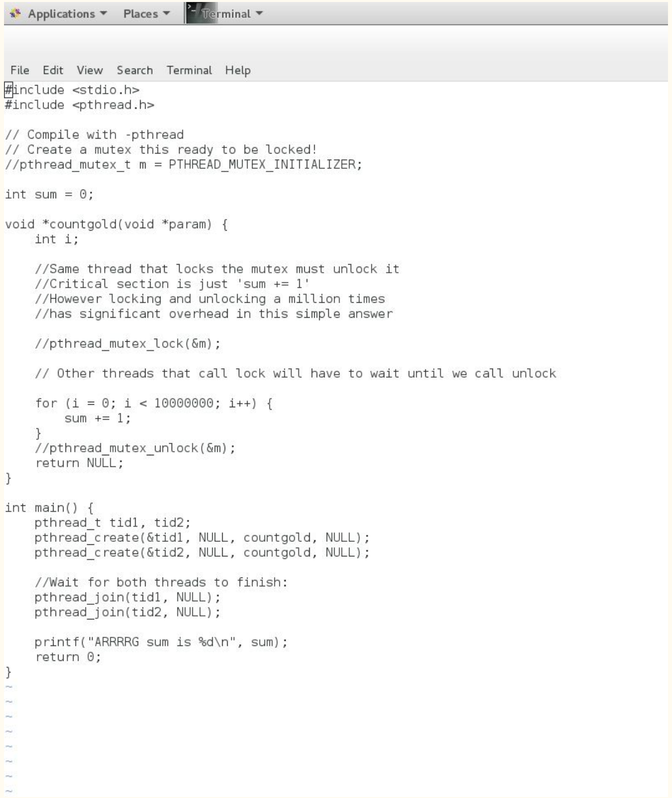


```c
#include <stdio.h>
#include <pthread.h>

// Compile with -pthread
// Create a mutex this ready to be locked!
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;

int sum = 0;

void *countgold(void *param) {
    int i;

    //Same thread that locks the mutex must unlock it
    //Critical section is just 'sum += 1'
    //However locking and unlocking a million times
    //has significant overhead in this simple answer

    pthread_mutex_lock(&m);

    // Other threads that call lock will have to wait until we call unlock

    for (i = 0; i < 10000000; i++) {
        sum += 1;
    }
    pthread_mutex_unlock(&m);
    return NULL;
}

int main() {
    pthread_t tid1, tid2;
    pthread_create(&tid1, NULL, countgold, NULL);
    pthread_create(&tid2, NULL, countgold, NULL);

    //Wait for both threads to finish:
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    printf("ARRRRG sum is %d\n", sum);
    return 0;
}
~
~
~
~
```

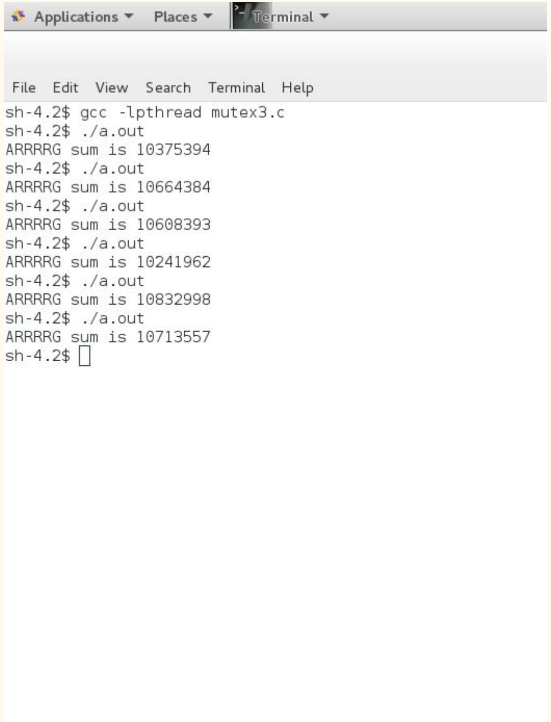


```
sh-4.2$ gcc -lpthread mutex2.c
sh-4.2$ ./a.out
ARRRRG sum is 20000000
sh-4.2$ 
```

Without Mutex

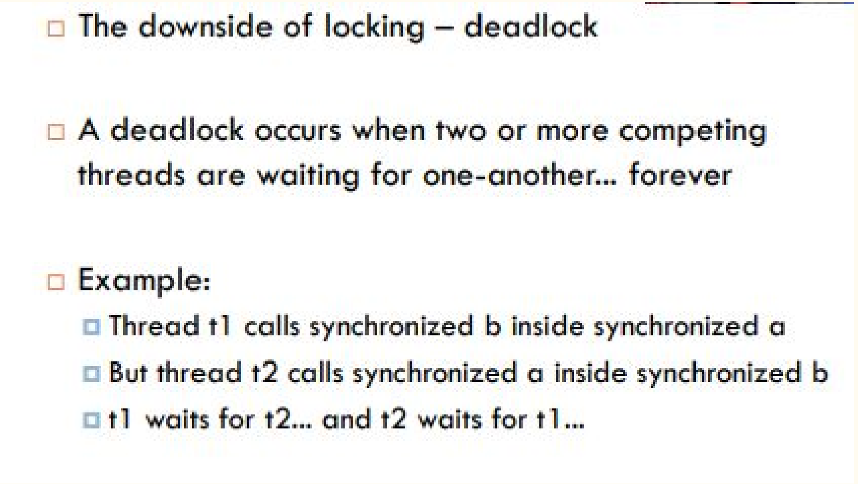


```
#include <stdio.h>
#include <pthread.h>

// Compile with -pthread
// Create a mutex this ready to be locked!
//pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;

int sum = 0;

void *countgold(void *param) {
    int i;

    //Same thread that locks the mutex must unlock it
    //Critical section is just 'sum += 1'
    //However locking and unlocking a million times
    //has significant overhead in this simple answer

    //pthread_mutex_lock(&m);

    // Other threads that call lock will have to wait until we call unlock

    for (i = 0; i < 10000000; i++) {
        sum += 1;
    }
    //pthread_mutex_unlock(&m);
    return NULL;
}

int main() {
    pthread_t tid1, tid2;
    pthread_create(&tid1, NULL, countgold, NULL);
    pthread_create(&tid2, NULL, countgold, NULL);

    //Wait for both threads to finish:
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    printf("ARRRRG sum is %d\n", sum);
    return 0;
}
~
~
~
~
~
~
~
~
```

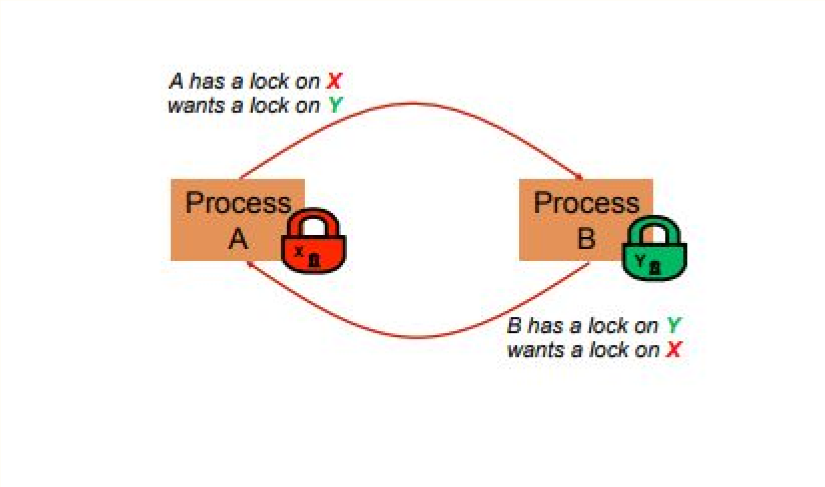


```
sh-4.2$ gcc -lpthread mutex3.c
sh-4.2$ ./a.out
ARRRRG sum is 10375394
sh-4.2$ ./a.out
ARRRRG sum is 10664384
sh-4.2$ ./a.out
ARRRRG sum is 10608393
sh-4.2$ ./a.out
ARRRRG sum is 10241962
sh-4.2$ ./a.out
ARRRRG sum is 10832998
sh-4.2$ ./a.out
ARRRRG sum is 10713557
sh-4.2$ 
```

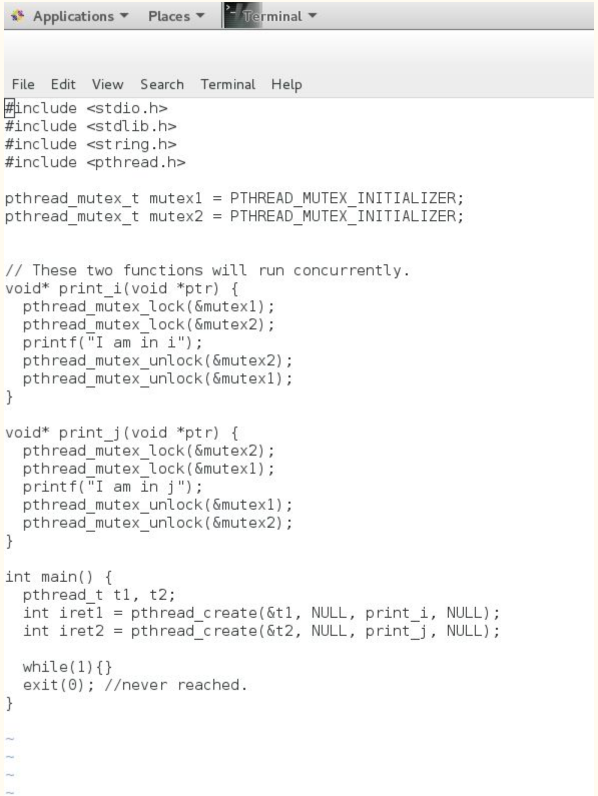
Deadlocks



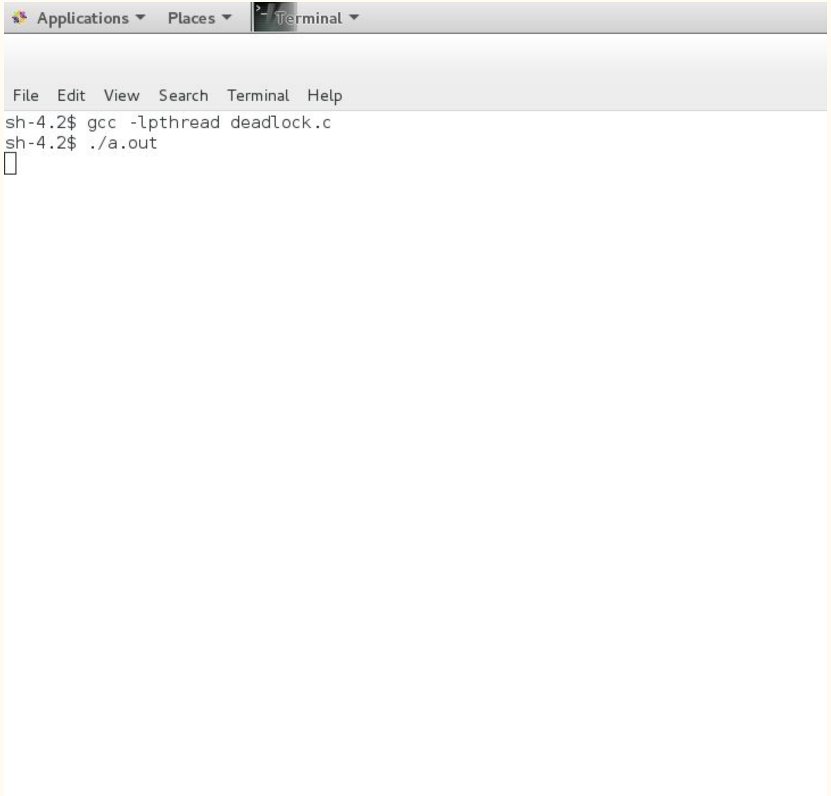
- The downside of locking – deadlock
- A deadlock occurs when two or more competing threads are waiting for one-another… forever
- Example:
  - Thread t1 calls synchronized b inside synchronized a
  - But thread t2 calls synchronized a inside synchronized b
  - t1 waits for t2… and t2 waits for t1…



Deadlocks

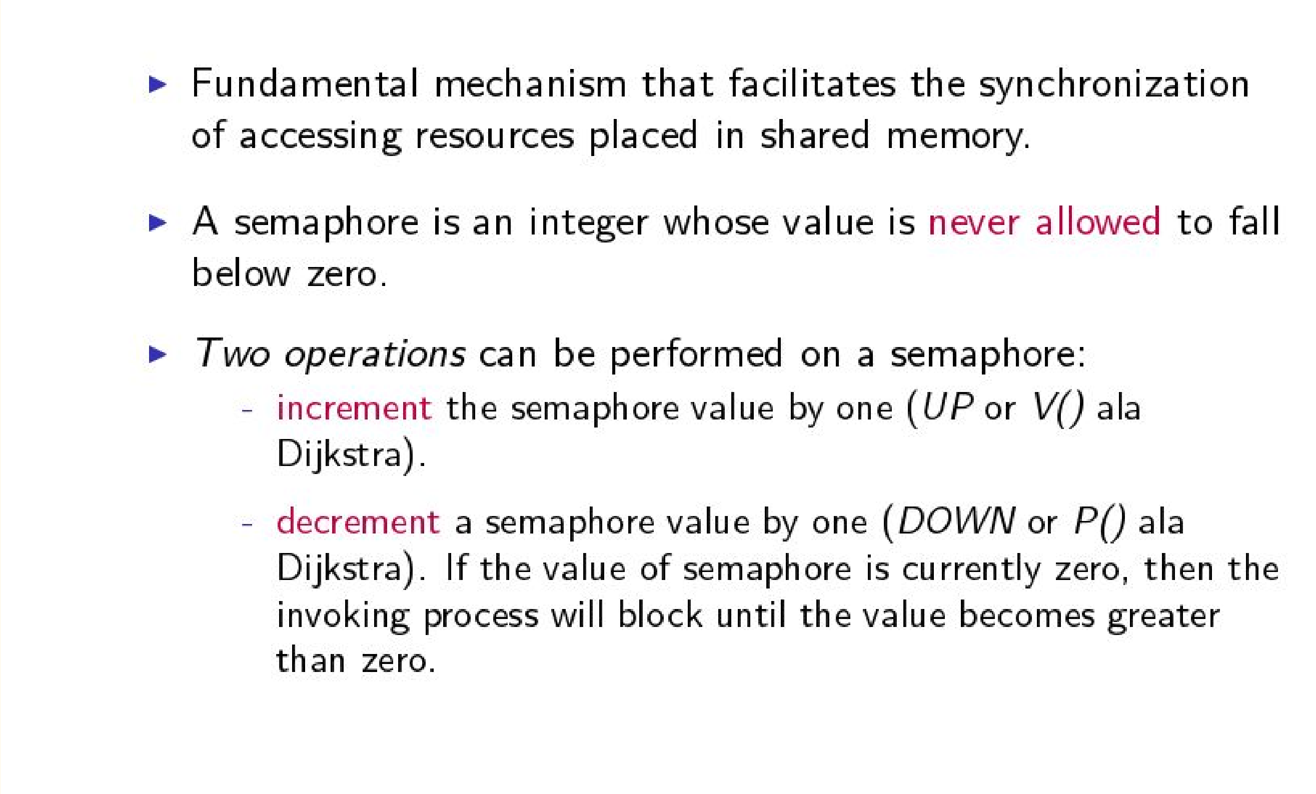


```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>

pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t mutex2 = PTHREAD_MUTEX_INITIALIZER;


// These two functions will run concurrently.
void* print_i(void *ptr) {
  pthread_mutex_lock(&mutex1);
  pthread_mutex_lock(&mutex2);
  printf("I am in i");
  pthread_mutex_unlock(&mutex2);
  pthread_mutex_unlock(&mutex1);
}

void* print_j(void *ptr) {
  pthread_mutex_lock(&mutex2);
  pthread_mutex_lock(&mutex1);
  printf("I am in j");
  pthread_mutex_unlock(&mutex1);
  pthread_mutex_unlock(&mutex2);
}

int main() {
  pthread_t t1, t2;
  int iret1 = pthread_create(&t1, NULL, print_i, NULL);
  int iret2 = pthread_create(&t2, NULL, print_j, NULL);

  while(1){}
  exit(0); //never reached.
}

~
~
~
~
```



```
sh-4.2$ gcc -lpthread deadlock.c
sh-4.2$ ./a.out
```

Semaphores

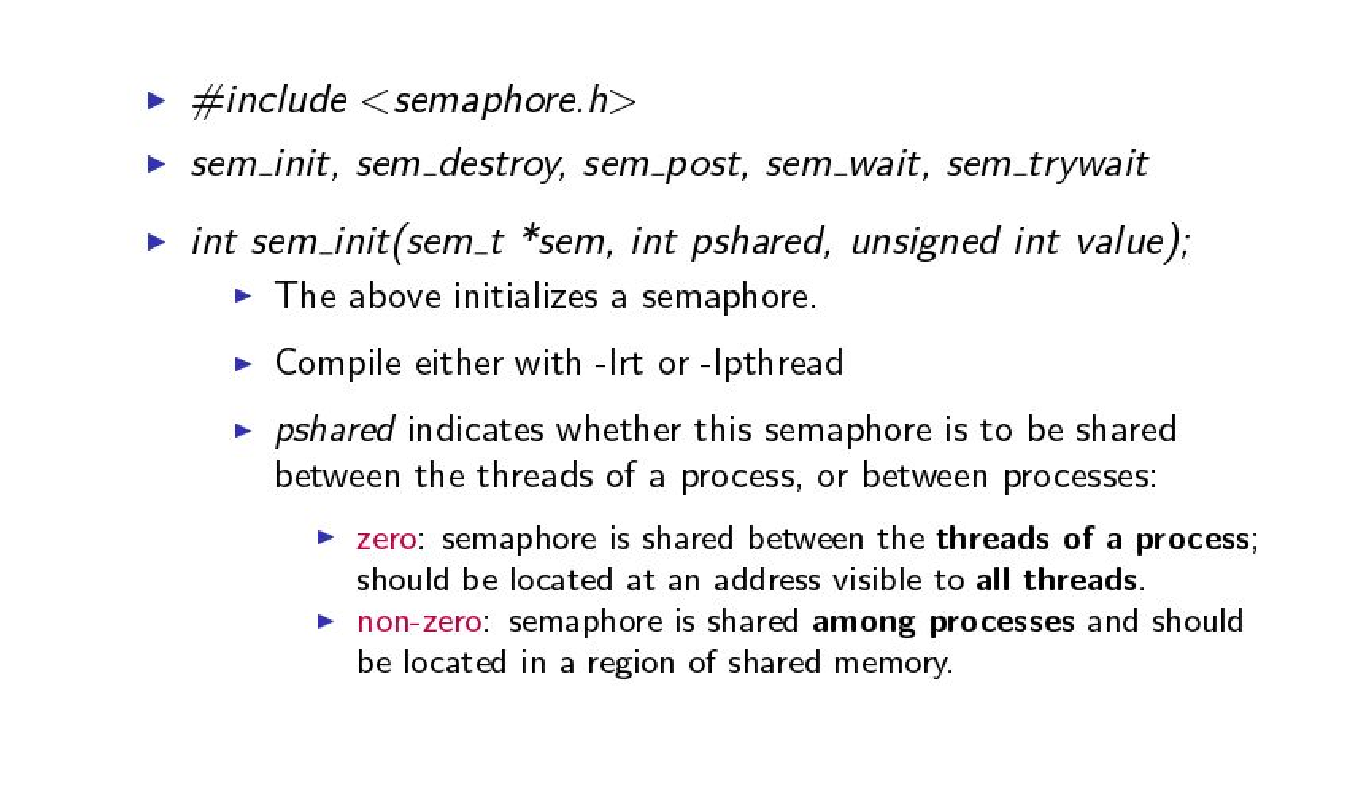


- Fundamental mechanism that facilitates the synchronization of accessing resources placed in shared memory.

- A semaphore is an integer whose value is never allowed to fall below zero.

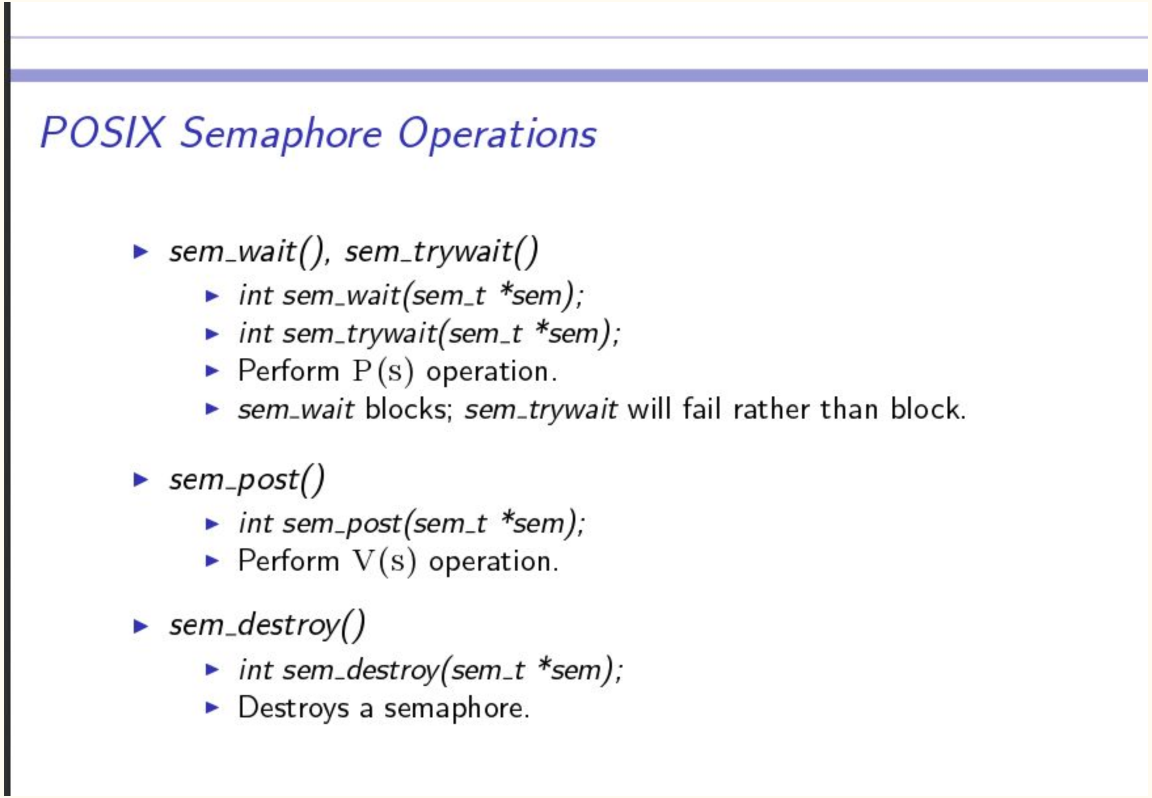
- *Two operations* can be performed on a semaphore:
  - increment the semaphore value by one (*UP* or *V()* ala Dijkstra).
  - decrement a semaphore value by one (*DOWN* or *P()* ala Dijkstra). If the value of semaphore is currently zero, then the invoking process will block until the value becomes greater than zero.

POSIX Semaphores



- *#include <semaphore.h>*
- *sem_init, sem_destroy, sem_post, sem_wait, sem_trywait*
- *int sem_init(sem_t *sem, int pshared, unsigned int value);*
    - The above initializes a semaphore.
    - Compile either with -lrt or -lpthread
    - *pshared* indicates whether this semaphore is to be shared between the threads of a process, or between processes:
        - zero: semaphore is shared between the **threads of a process**; should be located at an address visible to **all threads**.
        - non-zero: semaphore is shared **among processes** and should be located in a region of shared memory.

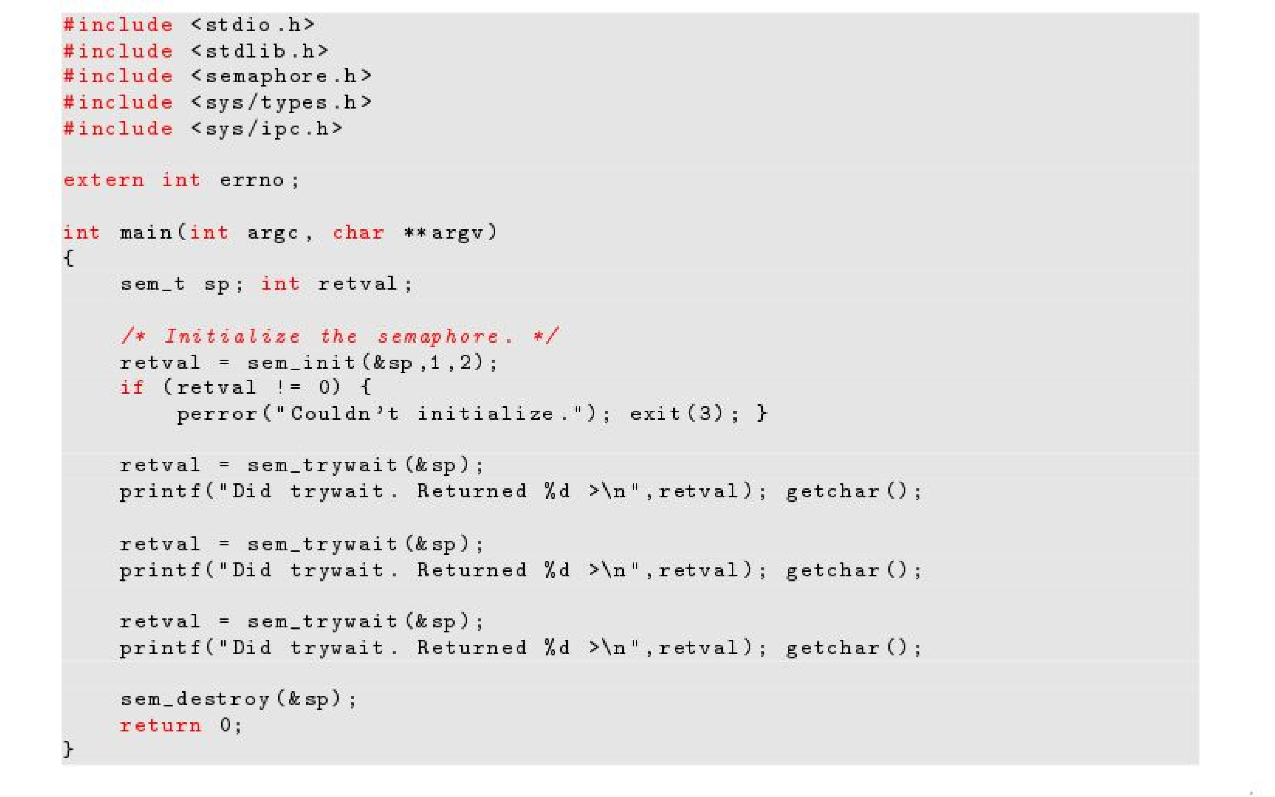
POSIX Semaphore Operations



POSIX Semaphore Operations

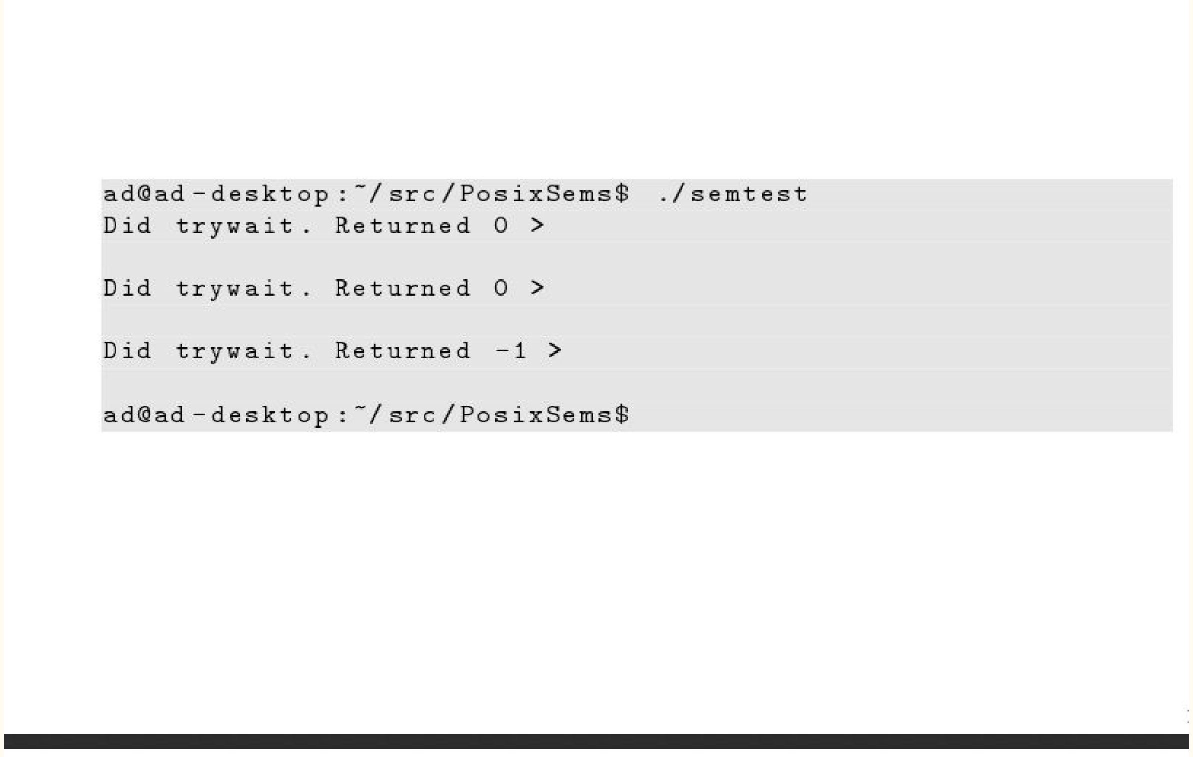
- *sem_wait(), sem_trywait()*
  - *int sem_wait(sem_t *sem);*
  - *int sem_trywait(sem_t *sem);*
  - Perform $\mathrm{P}(\mathrm{s})$ operation.
  - *sem_wait* blocks; *sem_trywait* will fail rather than block.
- *sem_post()*
  - *int sem_post(sem_t *sem);*
  - Perform $\mathrm{V}(\mathrm{s})$ operation.
- *sem_destroy()*
  - *int sem_destroy(sem_t *sem);*
  - Destroys a semaphore.

Creating and Using A Posix Semaphore



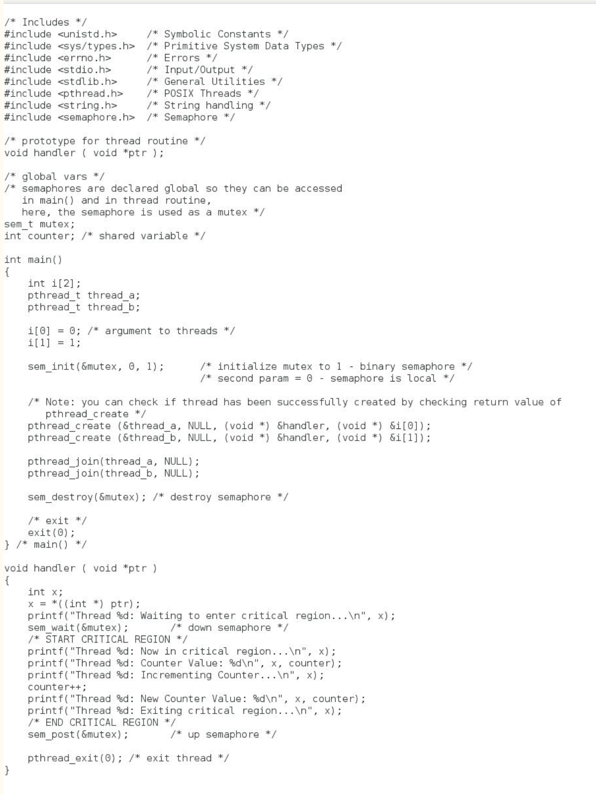
```c
#include <stdio.h>
#include <stdlib.h>
#include <semaphore.h>
#include <sys/types.h>
#include <sys/ipc.h>

extern int errno;

int main(int argc, char **argv)
{
    sem_t sp; int retval;

    /* Initialize the semaphore. */
    retval = sem_init(&sp,1,2);
    if (retval != 0) {
        perror("Couldn’t initialize."); exit(3); }

    retval = sem_trywait(&sp);
    printf("Did trywait. Returned %d >\n",retval); getchar();

    retval = sem_trywait(&sp);
    printf("Did trywait. Returned %d >\n",retval); getchar();

    retval = sem_trywait(&sp);
    printf("Did trywait. Returned %d >\n",retval); getchar();

    sem_destroy(&sp);
    return 0;
}
```

Executing the program



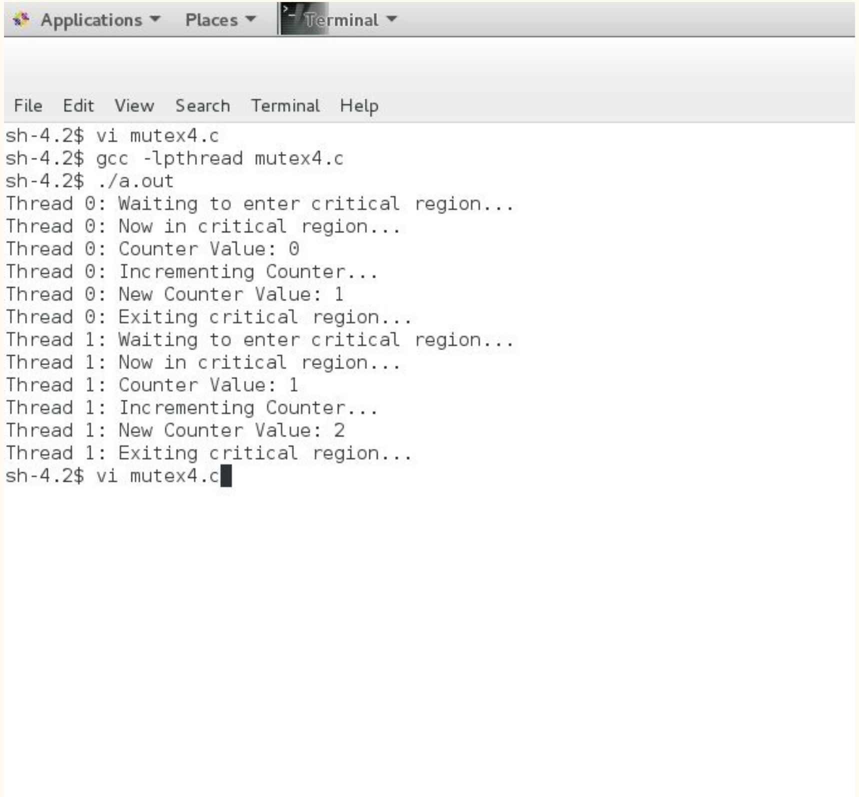
```
ad@ad-desktop:~/src/PosixSems$ ./semtest
Did trywait. Returned 0 >

Did trywait. Returned 0 >

Did trywait. Returned -1 >

ad@ad-desktop:~/src/PosixSems$
```

Another Example



```c
/* Includes */
#include <unistd.h>     /* Symbolic Constants */
#include <sys/types.h>  /* Primitive System Data Types */
#include <errno.h>      /* Errors */
#include <stdio.h>      /* Input/Output */
#include <stdlib.h>     /* General Utilities */
#include <pthread.h>    /* POSIX Threads */
#include <string.h>     /* String handling */
#include <semaphore.h>  /* Semaphore */

/* prototype for thread routine */
void handler ( void *ptr );

/* global vars */
/* semaphores are declared global so they can be accessed
   in main() and in thread routine,
   here, the semaphore is used as a mutex */
sem_t mutex;
int counter; /* shared variable */

int main()
{
    int i[2];
    pthread_t thread_a;
    pthread_t thread_b;

    i[0] = 0; /* argument to threads */
    i[1] = 1;

    sem_init(&mutex, 0, 1);      /* initialize mutex to 1 - binary semaphore */
                                 /* second param = 0 - semaphore is local */

    /* Note: you can check if thread has been successfully created by checking return value of
       pthread_create */
    pthread_create (&thread_a, NULL, (void *) &handler, (void *) &i[0]);
    pthread_create (&thread_b, NULL, (void *) &handler, (void *) &i[1]);

    pthread_join(thread_a, NULL);
    pthread_join(thread_b, NULL);

    sem_destroy(&mutex); /* destroy semaphore */

    /* exit */
    exit(0);
} /* main() */

void handler ( void *ptr )
{
    int x;
    x = *((int *) ptr);
    printf("Thread %d: Waiting to enter critical region...\n", x);
    sem_wait(&mutex);       /* down semaphore */
    /* START CRITICAL REGION */
    printf("Thread %d: Now in critical region...\n", x);
    printf("Thread %d: Counter Value: %d\n", x, counter);
    printf("Thread %d: Incrementing Counter...\n", x);
    counter++;
    printf("Thread %d: New Counter Value: %d\n", x, counter);
    printf("Thread %d: Exiting critical region...\n", x);
    /* END CRITICAL REGION */
    sem_post(&mutex);       /* up semaphore */

    pthread_exit(0); /* exit thread */
}
```

Output



```
sh-4.2$ vi mutex4.c
sh-4.2$ gcc -lpthread mutex4.c
sh-4.2$ ./a.out
Thread 0: Waiting to enter critical region...
Thread 0: Now in critical region...
Thread 0: Counter Value: 0
Thread 0: Incrementing Counter...
Thread 0: New Counter Value: 1
Thread 0: Exiting critical region...
Thread 1: Waiting to enter critical region...
Thread 1: Now in critical region...
Thread 1: Counter Value: 1
Thread 1: Incrementing Counter...
Thread 1: New Counter Value: 2
Thread 1: Exiting critical region...
sh-4.2$ vi mutex4.c
```

Homework

Write a function that uses threads to synchronize printing between them to print out a triangle. Make two threads, one to print out odd rows and one to print out even rows. The goal is to print out:

```
*
**
***
****
```

where each line is printed by a different thread.