CS214 Recitation Sec.7

Nov. 7, 2017

Topics

1. Synchronization
2. Mutex Lock
3. Semaphore
4. Homework 6 - Threads Synchronization

Example Code

An example code using threads:

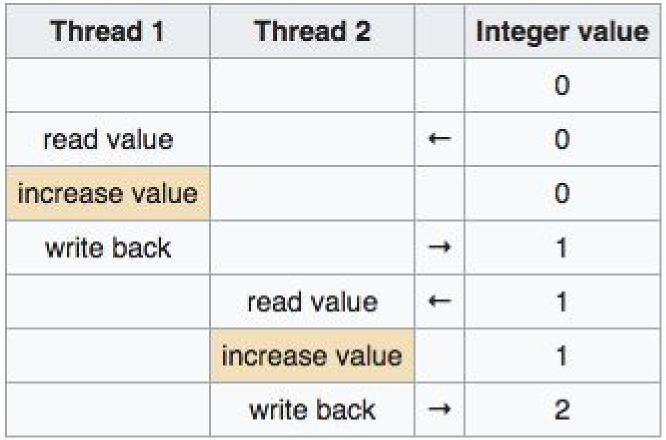
What is the output?

What is the intended output?

```c
#include <stdio.h>
#include <pthread.h>
// Compile with -pthread
// gcc race.c -pthread -o race

int sum = 0; //shared

void *countgold(void *param) {
    int i; //local to each thread
    for (i = 0; i < 10000000; i++) {
        sum += 1;
    }
    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("The sum is %d\n", sum);
    return 0;
}
```

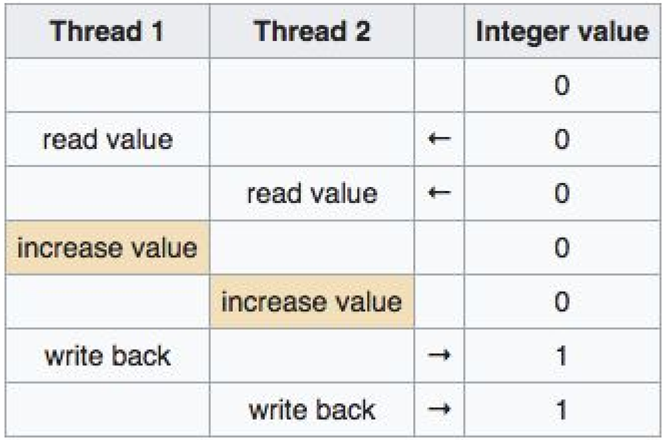
Race condition

Ideal output:



Thread 1	Thread 2		Integer value
			0
read value		←	0
increase value			0
write back		→	1
	read value	←	1
	increase value		1
	write back	→	2

Real output:



Thread 1	Thread 2		Integer value
			0
read value		←	0
	read value	←	0
increase value			0
	increase value		0
write back		→	1
	write back	→	1

Race condition

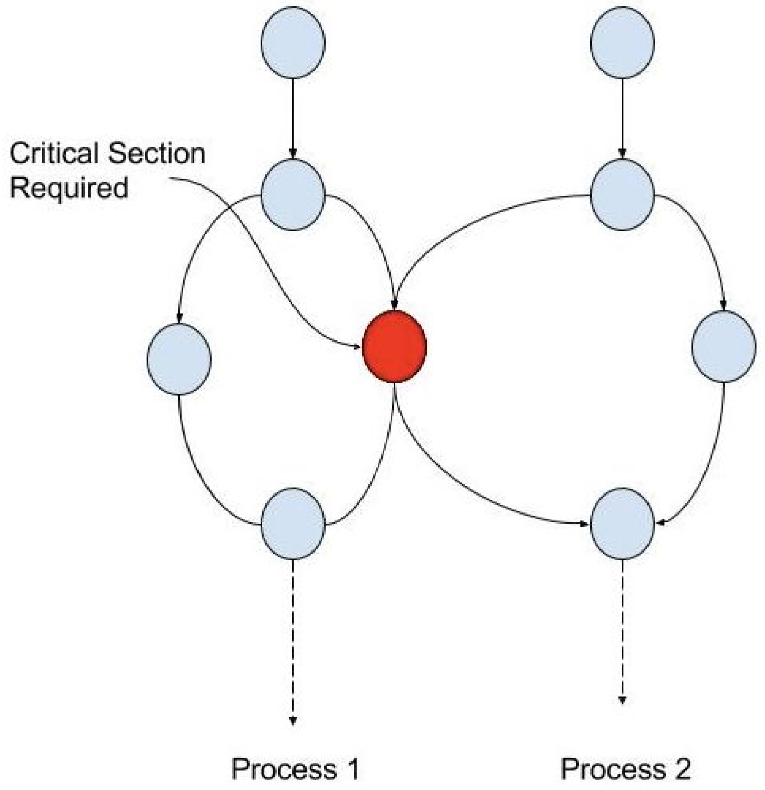
A race condition is the behavior that the output is *dependent on the sequence of other uncontrollable events*. It becomes a bug when events do not happen in the order the programmer intended.

The term originates with the idea of two events *racing each other* to influence the output.

How do we guarantee correct interaction between threads?
Using **Synchronization**!

Critical sections

When a process/thread is accessing shared resources that *can only be operated by one process/thread at a time*. The process/thread is said to be in a **critical section**.



Synchronization

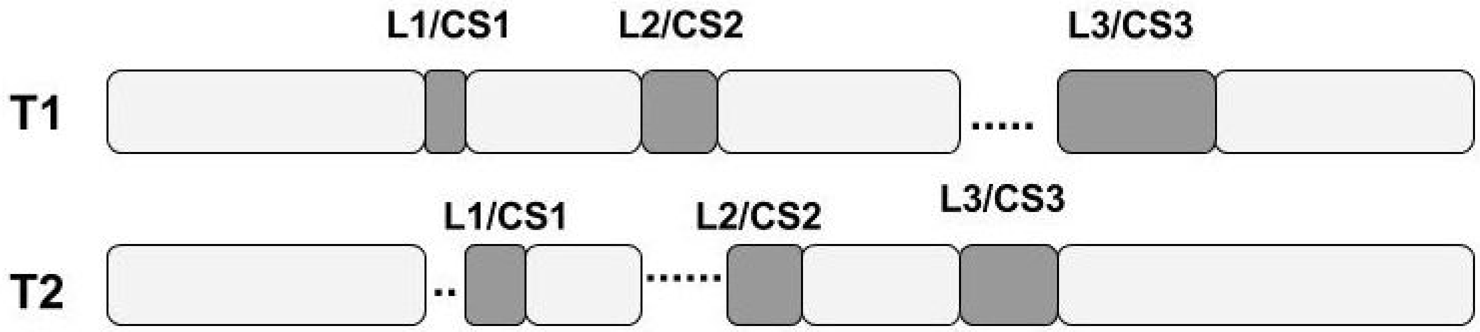
Several methods to implement synchronization:

- **Mutual exclusion** (Mutex locks) : used for *exclusive access* to a shared resource (critical section)   operations: *lock*, *unlock*
- **Semaphores** : generalization of mutexes: *counting number* of available “resources” operations: wait for an available resource (*decrement*), notify availability (*increment*)
- **Conditional variables** : *wait for a specific event* to happen, tied to a mutex for exclusive access   operations: *wait* for event, *signal occurrence* of event

Mutex lock

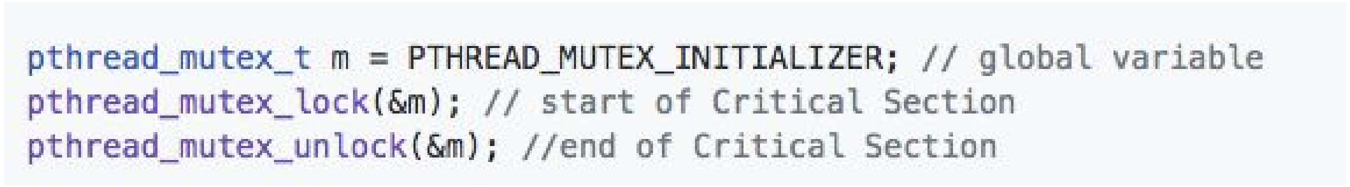
If one thread is currently inside a critical section we would like another thread to wait until the first thread is complete. For this purpose we can use a **mutex lock**

operations: *lock*, *unlock*



Mutex lock (Cont.)

Simply three lines to use mutex locks:



```
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER; // global variable
pthread_mutex_lock(&m); // start of Critical Section
pthread_mutex_unlock(&m); //end of Critical Section
```

Once we are finished with the mutex, we should also call *pthread_mutex_destroy(&m)* to destroy the mutex lock

Mutex locks (Cont.)

If I lock a mutex, does it stop all other threads?

No, the other threads will continue.

It's only when a thread attempts to lock a mutex that is already locked, will the thread have to wait.

As soon as the original thread unlocks the mutex, the second (waiting) thread will acquire the lock and be able to access the critical section.

Mutex locks (Cont.)

If one thread locks a mutex can another thread unlock it?

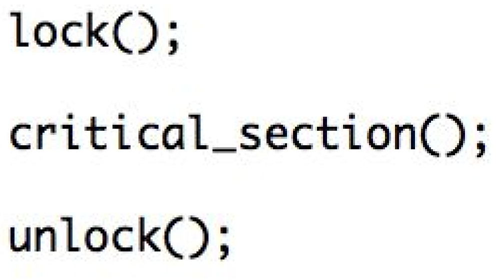
No. The same thread must unlock it.

Mutex locks (Cont.)

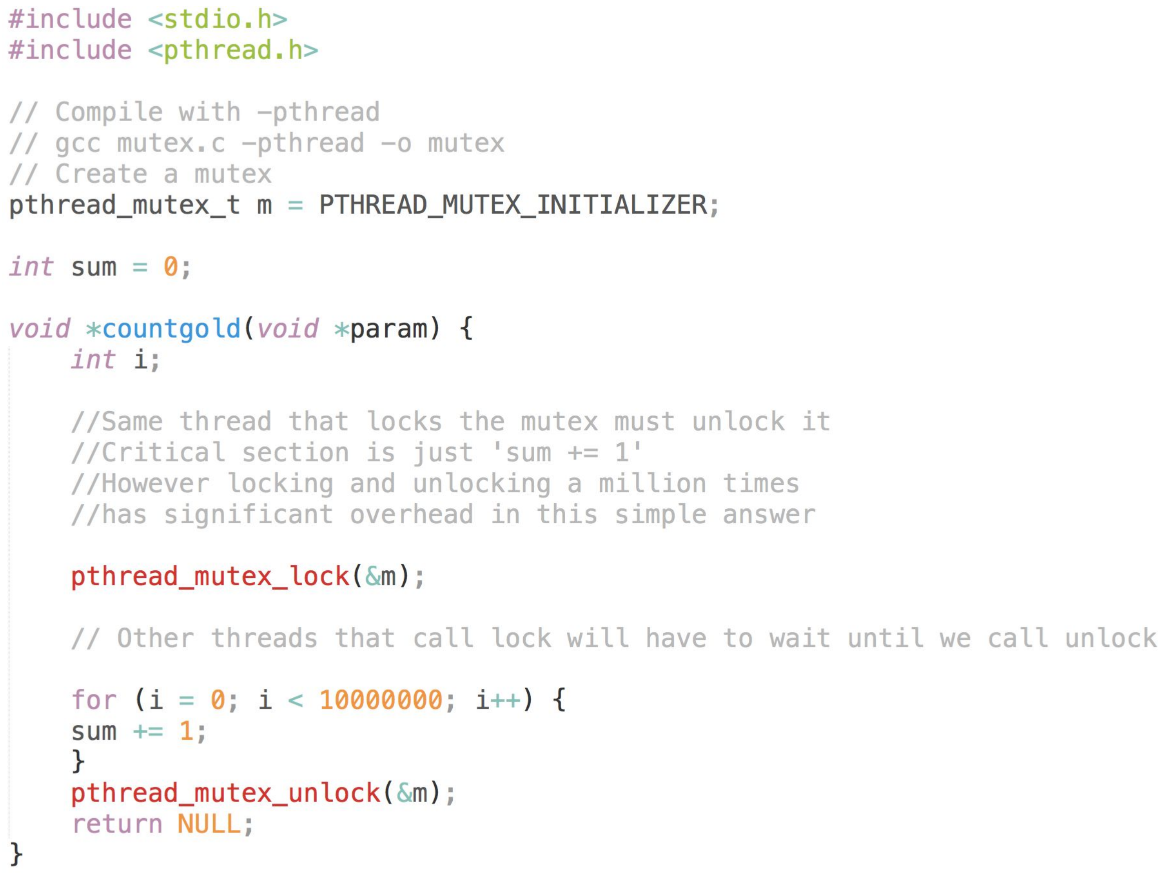
Can I use two or more mutex locks?

Yes. In fact it's common to have one lock per data structure that you need to update.

Mutex example



```
lock();

critical_section();

unlock();
```



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

// Compile with -pthread
// gcc mutex.c -pthread -o mutex
// Create a mutex
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;
}
```

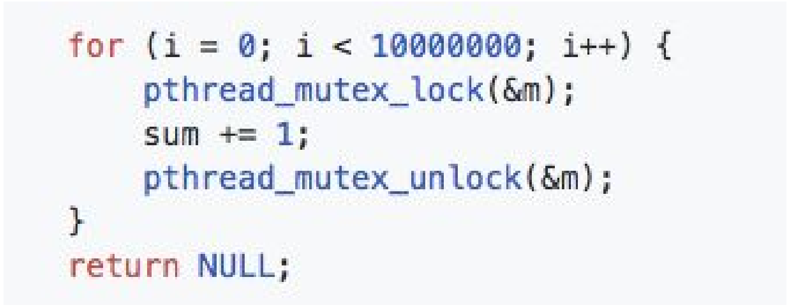
Mutex example (Cont.)

What is the output?

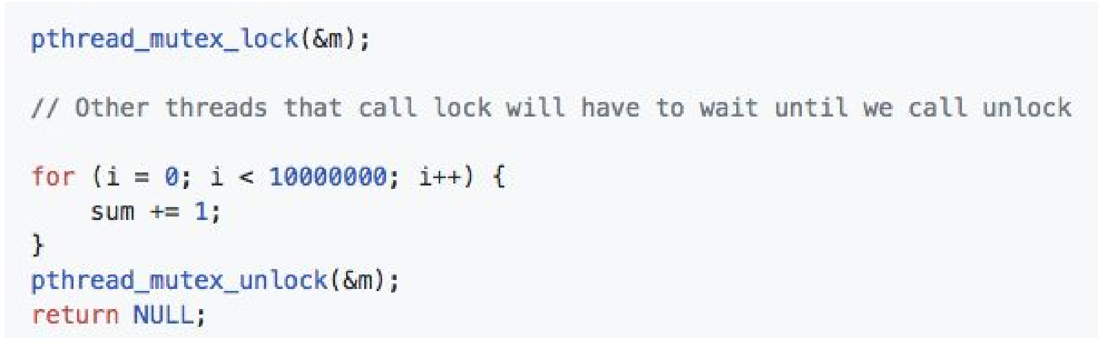
```
int main() {
    pthread_t tid1, tid2;
    pthread_create(&tid1, NULL, countgold, NULL);
    pthread_create(&tid2, NULL, countgold, NULL);

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

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

Mutex example (Cont.)

Which is faster?



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



```
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;
```

Mutex - Discussion

Mutual exclusion changes scheduling between threads

- Previously: Schedule could be anything
- With mutual exclusion: Schedule is *constrained*

Q: Since scheduling is constrained, which thread goes first, Thread 1 or Thread 2?
A: We still have no clue

- mutex only ensures two threads aren’t in critical section at one time
- otherwise *scheduling is still arbitrary*
- and that’s fine with us

Semaphore

A non-negative global integer synchronization variable which is maintained by *OS kernel* (Not supported by Mac OS X)

Manipulated by *wait* and post operations:

- signal availability of a resource

```
post(s): [ s++; ]
```

- wait for resource only if none available

```
wait(s): [ while (s == 0) wait(); s--; ]
```

Semaphore (Cont.)

Post *increments* the semaphore and immediately returns.

```
post(s): [ s++; ]
```

Wait will *wait if the count is zero*.

If the count is non-zero the semaphore *decrements* the count and immediately returns

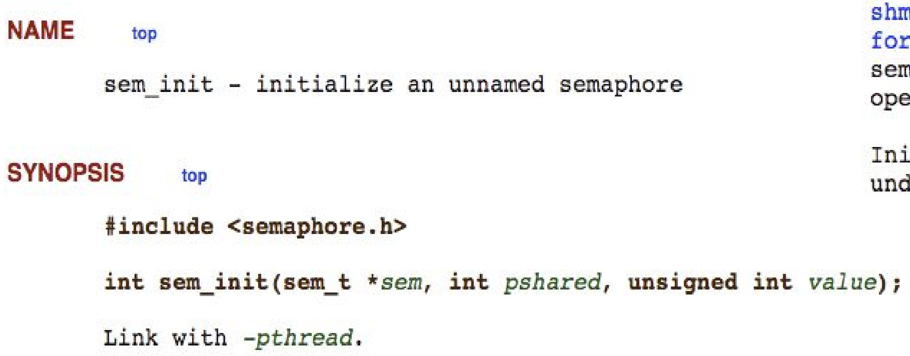
```
wait(s): [ while (s == 0) wait(); s--; ]
```

OS kernel guarantees that the two operations are executed indivisibly. It means that **only one wait or post** operation at a time can modify s

Semaphore (Cont.)

How to create a semaphore:

sem_init

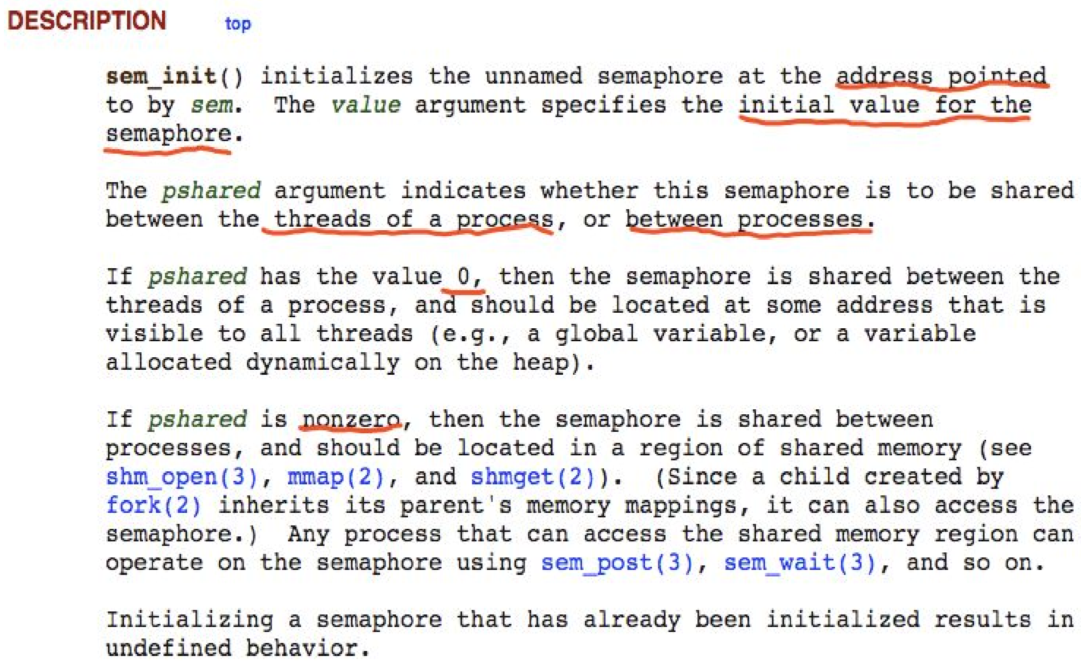


NAME top

```
sem_init - initialize an unnamed semaphore
```

SYNOPSIS top

```
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned int value);

Link with -pthread.
```



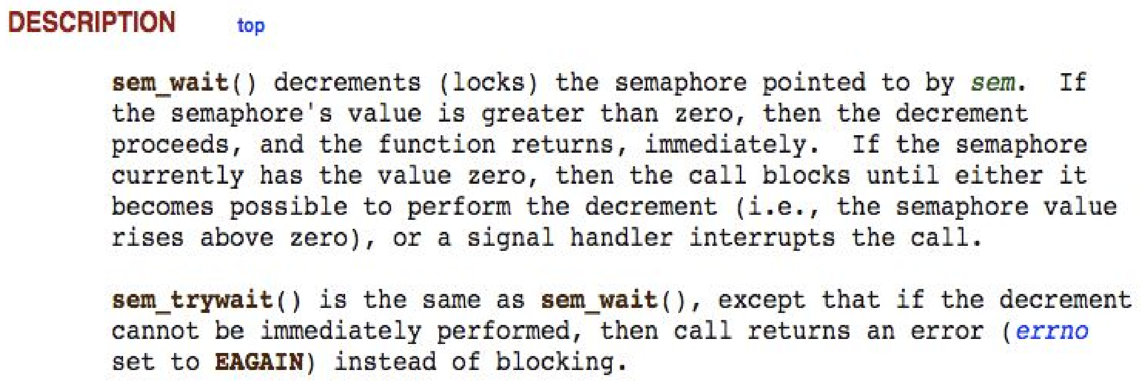
DESCRIPTION top

```
sem_init() initializes the unnamed semaphore at the address pointed
to by sem.  The value argument specifies the initial value for the
semaphore.

The pshared argument indicates whether this semaphore is to be shared
between the threads of a process, or between processes.

If pshared has the value 0, then the semaphore is shared between the
threads of a process, and should be located at some address that is
visible to all threads (e.g., a global variable, or a variable
allocated dynamically on the heap).

If pshared is nonzero, then the semaphore is shared between
processes, and should be located in a region of shared memory (see
shm_open(3), mmap(2), and shmget(2)).  (Since a child created by
fork(2) inherits its parent's memory mappings, it can also access the
semaphore.)  Any process that can access the shared memory region can
operate on the semaphore using sem_post(3), sem_wait(3), and so on.

Initializing a semaphore that has already been initialized results in
undefined behavior.
```

Semaphore (Cont.)

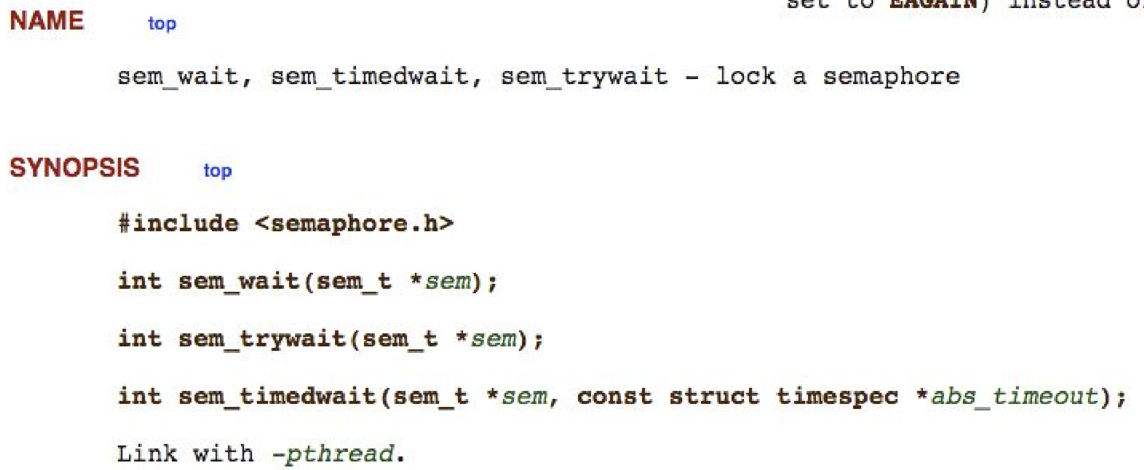
How to create a semaphore:

sem_wait



DESCRIPTION top

```
sem_wait() decrements (locks) the semaphore pointed to by sem.  If
the semaphore's value is greater than zero, then the decrement
proceeds, and the function returns, immediately.  If the semaphore
currently has the value zero, then the call blocks until either it
becomes possible to perform the decrement (i.e., the semaphore value
rises above zero), or a signal handler interrupts the call.

sem_trywait() is the same as sem_wait(), except that if the decrement
cannot be immediately performed, then call returns an error (errno
set to EAGAIN) instead of blocking.
```



NAME top

```
       sem_wait, sem_timedwait, sem_trywait - lock a semaphore
```

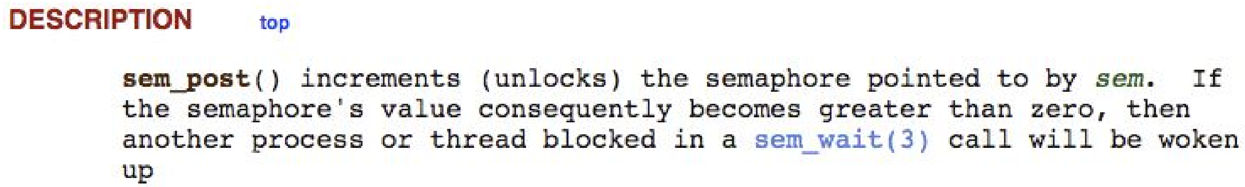
SYNOPSIS top

```
       #include <semaphore.h>

       int sem_wait(sem_t *sem);

       int sem_trywait(sem_t *sem);

       int sem_timedwait(sem_t *sem, const struct timespec *abs_timeout);

       Link with -pthread.
```

Semaphore (Cont.)

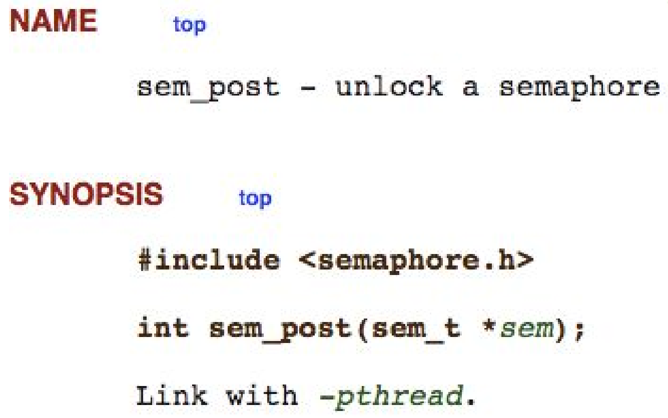
How to create a semaphore:

sem_post



DESCRIPTION top

sem_post() increments (unlocks) the semaphore pointed to by *sem*. If the semaphore's value consequently becomes greater than zero, then another process or thread blocked in a sem_wait(3) call will be woken up



NAME top

```
sem_post - unlock a semaphore
```

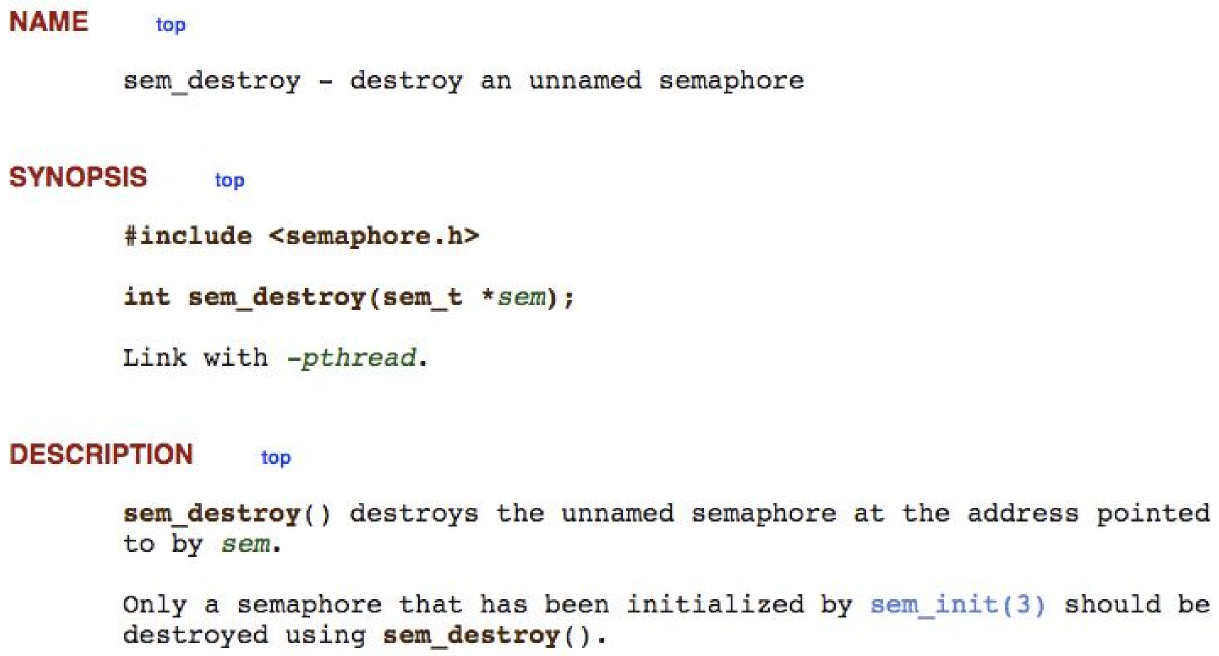
SYNOPSIS top

```
#include <semaphore.h>

int sem_post(sem_t *sem);

Link with -pthread.
```

Semaphore (Cont.)

How to create a semaphore:

sem_destroy



NAME top

```
       sem_destroy - destroy an unnamed semaphore
```

SYNOPSIS top

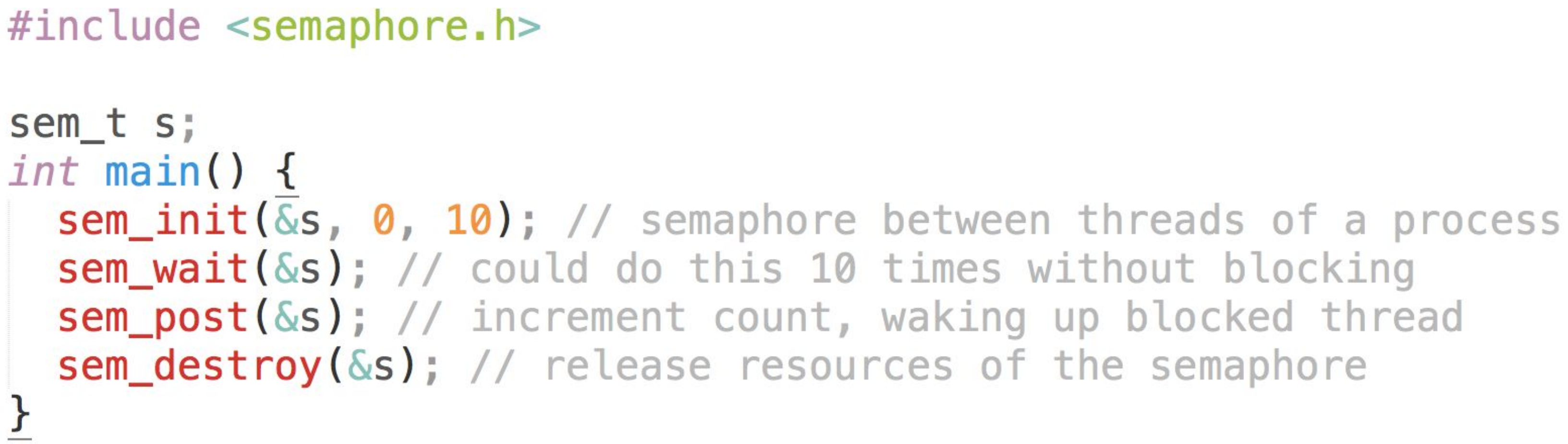
```
       #include <semaphore.h>

       int sem_destroy(sem_t *sem);

       Link with -pthread.
```

DESCRIPTION top

```
       sem_destroy() destroys the unnamed semaphore at the address pointed
       to by sem.

       Only a semaphore that has been initialized by sem_init(3) should be
       destroyed using sem_destroy().
```

Semaphore (Cont.)

How to create a semaphore:



```c
#include <semaphore.h>

sem_t s;
int main() {
  sem_init(&s, 0, 10); // semaphore between threads of a process
  sem_wait(&s); // could do this 10 times without blocking
  sem_post(&s); // increment count, waking up blocked thread
  sem_destroy(&s); // release resources of the semaphore
}
```

Semaphores (Cont.)

Can I call wait and post from different threads?

Yes! Unlike a mutex, the increment and decrement can be from different threads.

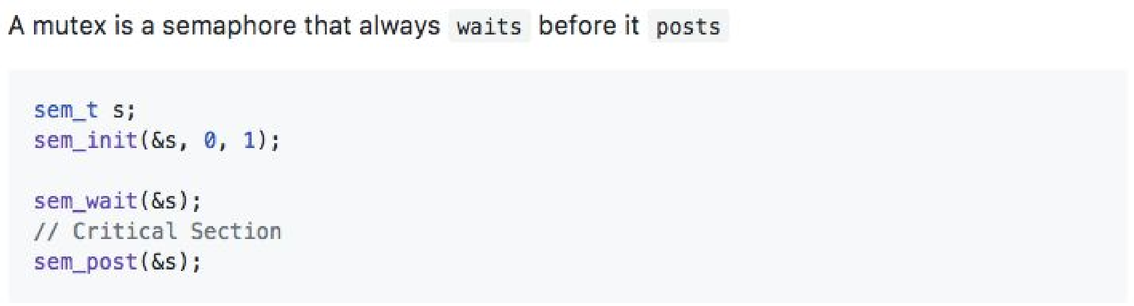
Semaphores are the generalization of mutexes. And mutexes can be regarded as binary semaphore (semaphore whose value is always 0 or 1)

Semaphores (Cont.)

Can I use a semaphore instead of a mutex?

Yes. A semaphore is greater than mutex. To use a semaphore:

- Initialize the semaphore with a count of *one*
- Replace ...*lock* with *sem_wait*
- Replace ...*unlock* with *sem_post*



A mutex is a semaphore that always `waits` before it `posts`

```
sem_t s;
sem_init(&s, 0, 1);

sem_wait(&s);
// Critical Section
sem_post(&s);
```

Semaphores example

Not supported by Mac OS X

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

// Compile with -pthread
// gcc sem.c -pthread -o sem
// Create a binary semaphore

int sum = 0;
sem_t s;

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


    for (i = 0; i < 10000000; i++) {
        sem_wait(&s);
        sum += 1;
        sem_post(&s);
    }

    return NULL;
}
```

```c
int main() {
    sem_init(&s, 0, 1);
    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("The sum is %d\n", sum);
    sem_destroy(&s); // release resources of the semaphore
    return 0;
}
```

HW6 - Threads Synchronization

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

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

where each line is printed by a different thread. Have **thread 0** print out the even lines (2 stars, 4 stars, etc) and **thread 1** print out the odd ones (1 star, 3 stars, etc).

HW6 - Threads Synchronization (Cont.)

At first, just run them with no synchronization. You'll get an interleaving of rows ... likely you'll get a batch of rows and then another batch of rows.

Next add a pair of mutexes to trade off control between the threads. They should trade off using the mutexes to synchronize between them.

Sometimes thread 0 may be scheduled before thread 1 and you get the rows printed in the wrong order. You need to make sure the first thread gets the mutex first - but you can't, really since you do not have control of the scheduler.

HW6 - Threads Synchronization (Cont.)

Change the mutex into a binary semaphore. This way if the wrong thread starts first, it will block until the other thread gets to run.

Thread 1 ought to notify/produce and thread 0 ought to wait/consume. This way you can trade control in an intentional manner.