**Lecture 7**

**Semaphores and Mutexes**

The POSIX thread library contains functions for working with semaphores and mutexes. There is much more to say than what is mentioned here. A good place to find more information is https://linux.die.net.

The functions should all be compiled and linked with -pthread.

**What is a semaphore in LINUX?**

**(Library: #include <semaphore.h>)**

A semaphore is fundamentally an integer whose value is never allowed to fall below 0. There are two operations on a semaphore: wait and post. The post operation increments the semaphore by 1, and the wait operation does the following: If the semaphore has a value > 0, the semaphore is decremented by 1. If the semaphore has value 0, the caller will be blocked (busy-waiting or more likely on a queue) until the semaphore has a value larger than 0, and then it is decremented by 1. We declare a semaphore as:

**sem\_t sem;**

where sem\_t is a typedef defined in a header file as (apparently) a kind of unsigned char.

An example of this might be that we have a set of N interchangeable resources. We start with semaphore S = N. We use a resource, so there are now N-1 available (wait), and we return it when we are done (post). If the semaphore has value 0, there are no resources available, and we have to wait (until someone does a post).

Semaphores are thus used to coordinate concurrent processes.

This is what some people call a "counted semaphore". There is a similar notion called a "binary semaphore" which is limited to the values 0 and 1.

A semaphore may be named or unnamed. These notes assume we are using named semaphores.

**Semaphore Functions in C**

1. **int sem\_init(sem\_t \* sem, int *pshared*, unsigned int value);**

**Purpose:**

This initializes the semaphore \*sem.   
The initial value of the semaphore will be value. If *pshared* is 0, the semaphore is shared among all threads of a process (and hence need to be visible to all of them such as a global variable).  
 If *pshared* is not zero, the semaphore is shared but should be in shared memory.

**Notes:**

* On success, the return value is 0, and on failure, the return value is -1.
* An attempt to initialize a semaphore that has already been initialized results in undefined behavior.

1. **int sem\_wait(sem\_t \* sem);**

**Purpose:** This implements the wait function described above on the semaphore \*sem.

**Notes:**

* Here sem\_t is a typedef defined in the header file as (apparently) some variety of integer.
* On success, the return value is 0, and on failure, the return value is -1 (and the value of the semaphore is unchanged).
* There are related functions sem\_trywait() and sem\_timedwait().

1. **int sem\_post(sem\_t \* sem);**

**Purpose:** This implements the post function described above on the semaphore \*sem.

**Note:** On success, the return value is 0, and on failure, the return value is -1 (and the value of the semaphore is unchanged).

1. **int sem\_destroy(sem\_t \* sem);**

**Prototype:** int sem\_destroy(sem\_t \* sem);

**Purpose:** This destroys the semaphore \*sem, so \*sem becomes uninitialized.

**Notes:**

* On success, the return value is o, and on failure, the return value is -1.
* Destroying a semaphore on which other processes or threads are waiting (using sem\_wait()) or destroying an uninitialized semaphore will produce undefined results.

**What is a mutex in LINUX?**

**(Library: #include <pthread.h>)**

A mutex (named for "mutual exclusion") is a binary semaphore with an ownership restriction: it can be unlocked (the post operation) only by whoever locked it (the wait operation). Thus a mutex offers a somewhat stronger protection than an ordinary semaphore.

We declare a mutex as:

**pthread\_mutex\_t mutex;**

**mutex Functions in C**

1. **int pthread\_mutex\_init(pthread\_mutex\_t \* restrict mutex, const pthread\_mutexattr\_t \* restrict attr);**

**Purpose:** This initializes \*mutex with the attributes specified by attr. If attr is NULL, a default set of attributes is used. The initial state of \*mutex will be "initialized and unlocked".

**Notes:**

* If we attempt to initialize a mutex already initialized, the result is undefined.
* On success, the return value is 0, and on failure, the return value is a nonzero value indicating the type of error.
* In the prototype, the keyword restrict (part of the C99 standard) means that this pointer will be the only pointer to the object.

1. **int pthread\_mutex\_destroy(pthread\_mutex\_t \* restrict mutex);**

**Purpose:** This destroys the mutex object \*mutex, so \*mutex becomes uninitialized.

**Notes:**

* It is safe to destroy an unlocked mutex but not a locked mutex.
* The object \*mutex could be reused, i.e., reinitialized.
* On success, the return value is 0, and on failure, the return value is a nonzero value indicating the type of error.

1. **int pthread\_mutex\_lock(pthread\_mutex\_t \* mutex);**

**Purpose:** This locks \*mutex. If necessary, the caller is blocked until \*mutex is unlocked (by someone else) and then &mutex is locked. When the function call ends, \*mutex will be in a locked state.

**Notes:**

* Suppose we try to relock a locked mutex. Depending on the attributes of the mutex, we may have an error, or a count may be kept of how many times the caller has locked the same mutex (and thus will have to unlock it the same number of times).
* On success, the return value is 0, and on failure, the return value is a nonzero value indicating the type of error.

1. **int pthread\_mutex\_unlock(pthread\_mutex\_t \* mutex);**

**Purpose:** This unlocks \*mutex.

**Notes:**

* Suppose we try to unlock an unlocked mutex. Depending on the attributes of the mutex, we may have an error.
* On success, the return value is 0, and on failure, the return value is a nonzero value indicating the type of error.