Threads, MUTEX & Semaphores

Threads

Introduction

1. In the traditional Unix model, when a process needs something performed by another entity, it forks a child process. There are problems with a fork:

➢ Fork is expensive. Memory is copied from the parent to the child, all the descriptors are duplicated in the child. Current implementation use a technique called copy-on-write, which avoids a copy of the parent’s data space to the child until the child needs its own copy.

➢ Inter process communication is required to pass information between the parent and child after the fork.

2. Threads help with both problems. Threads are sometimes called lightweight processes. That is a thread creation can be 10-100 times faster than process creation time.

Threads and processes

All threads within a process share the same global memory. This makes the sharing of information easy between the threads, but along with this simplicity comes the problem of synchronization.

All threads within a process share:

- Process instructions
- Most data
- Open files (e.g. descriptors)
- Signal handlers
- Current working directory
- User and group Ids

Threads and processes

But each thread has its own:

- Thread ID
- Set of registers, including program counter and stack pointer
- Stack (for local variables and return addresses)
- Errno
- Signal mask
- Priority

Thread Usage Reasons

1. Simplifying programming model by decomposing application into quasi-parallel threads.
2. Thread creation is up to 100 times faster than process creation.
3. Performance is better when there is I/O besides CPU bursts. Overlapping those activities speed up application.
4. Threads are useful on multiple CPUs systems, where real parallelism is possible.

pthread Library

Linux implements the POSIX standard thread API (known as *pthreads*). All thread functions and data types are declared in the header file <pthread.h>.

The pthread functions are not included in the standard C library. Instead, they are in **libpthread**, so you should add **-lpthread** to the command line when you link your program.

pthread_create

- When a program is started by exec, a single thread is created, called the initial thread or main thread. Additional threads are created by **pthread_create**.

```
#include <pthread.h>
int pthread_create(pthread_t *tid, const pthread_attr_t *attr,
                                        void *(*func) (void *), void *arg);
```

Returns 0 if OK, positive Exxx value on Error

- Each thread within a process is identified by a thread ID, whose data type is pthread_t. On successful creation of a new thread, its ID is returned through the pointer tid.

Thread Creation

Each thread has numerous attributes: its **priority**, its initial **stack size**, whether it should be **daemon** thread or not, and so on. When a thread is created, we can specify these attributes by initializing a **pthread_attr_t** variable that overrides the default. We normally take the default, we specify the **attr** argument as a **null** pointer.

When we create a thread, we specify a function for it to execute, called its thread start function. The thread starts by calling this function and then terminates either explicitly (by calling **pthread_exit**) or implicitly (by letting this function return).

The address of the function is specified as the **func** argument, and this function is called with a single pointer argument, **arg**.

pthread_join

We can wait for a given thread to terminate by calling **pthread_join**. Comparing threads to Unix processes, **pthread_create** is similar to **fork**, and **pthread_join** is similar to **waitpid**.

```
#include <pthread.h>
int pthread_join(pthread_t tid, void **status);
```

Returns 0 if OK, positive Exxx value on error

pthread_self

Each thread has an ID that defines it with in a given process. The thread ID is returned by pthread_create. A thread fetches this value for itself using **pthread_self**.

```
#include <pthread.h>
pthread_t pthread_self(void);
```

Returns thread ID of calling thread

pthread_exit

One way for a thread to terminate is to call pthread_exit.

```
#include <pthread.h>
void pthread_exit(void *status);
```

Does not return to caller

If the thread is not detached, its thread ID and exit status are Retained for a later pthread_join by some other thread in the calling process.

Thread Attributes

Thread attributes provide a mechanism for fine-tuning the behavior of individual threads.

```
int pthread_attr_init ( pthread_attr_t   *attr ) ;
int pthread_attr_destroy ( pthread_attr_t  *attr) ;
int pthread_attr_setdetachstate ( pthread_attr_t  *attr,
                                                    int   detachstate ) ;
 int pthread_attr_setschedpolicy (pthread_attr_t  *attr, int  policy ) ;
int pthread_attr_setschedparam ( pthread_attr_t  *attr,
                                                         struct sched_param *param);
int pthread_attr_setinheritsched ( pthread_attr_t *attr, int  inherit );
Int pthread_attr_setscope ( pthread_attr_t  *attr , int  scope );
```

Cancellation

```
int  pthread_cancel ( pthread_t  thread ) ;
int  pthread_setcancelstate ( int state , int *oldstate ) ;
int  pthread_setcanceltype ( int  type , int *oldtype ) ;
void pthread_testcancel ( void ) ;
```

Cancellation is the mechanism by which a thread can terminate the execution of another thread . More precisely a thread can send cancellation request to another thread. Depending on its setting the target thread can then either ignore the request , honor it immediately or defer it till it reaches a cancellation point.
When a thread eventually honors a cancellation request , it performs as if pthread_exit(PTHREAD_CANCEL) has been called at that point .

Synchronization and Critical-sections

Multi-threaded programs are concurrent and share the same process space and can access the same data structures.
Threads are scheduled by OS and are executed at random. When threads are executing (racing to complete) they may give unexpected results (race condition).

Race Condition is a situation in which an unfortunate order of execution causes undesirable behavior .

Thread Synchronization

The threads library provides three synchronization mechanisms:

- 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.
- joins - Make a thread wait till others are complete (terminated).
- condition variables - data type pthread_cond_t
- **Waking and Suspending threads (RTL)**

Joins

A join is performed when one wants to wait for a thread to finish.

A thread calling routine may launch multiple threads then wait for them to finish to get the results. One wait for the completion of the threads with a join.

int   pthread_join (pthread th , void **thread_return) ;

POSIX SEMAPHORES

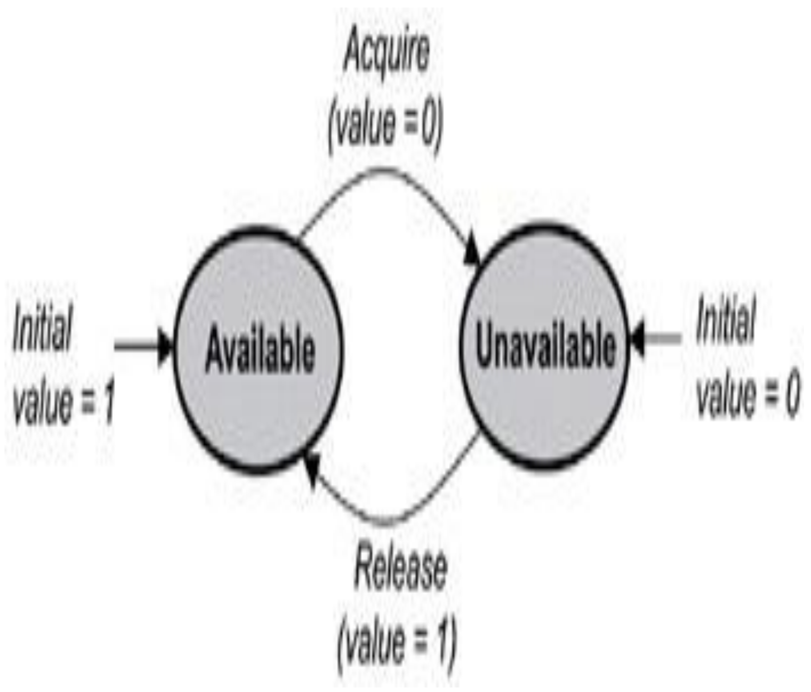
A semaphore is a counter that can be used to synchronize multiple threads. Linux guarantees that checking or modifying the value of a semaphore can be done safely, without creating a race condition. Each semaphore has a counter value, which is a non-negative integer.

A semaphore supports two basic operations:
A ***wait*** operation decrements the value of the semaphore by 1. If the value is already zero, the operation blocks until the value of the semaphore becomes positive.When the semaphore’s value becomes positive, it is decremented by 1 and the wait operation returns.

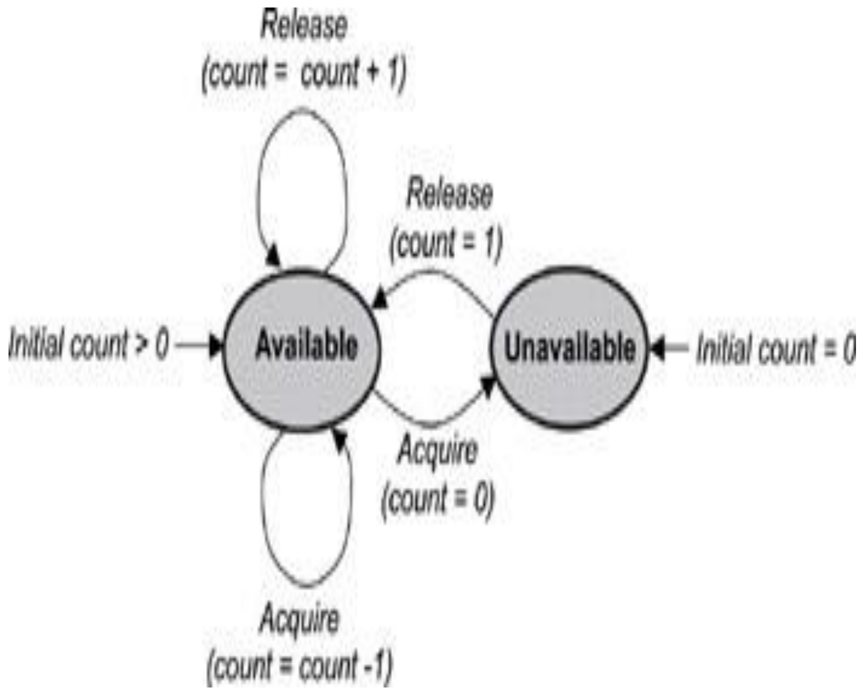
….POSIX SEMAPHORES

A ***post*** operation increments the value of the semaphore by 1. If the semaphore was previously zero and other threads are blocked in a wait operation on that semaphore, one of those threads is unblocked and its wait operation completes.

SEMAPHORES



The state diagram of a binary semaphore.



The state diagram of a counting semaphore

….POSIX SEMAPHORES

```
#include < semaphore .h >
int sem_init ( sem_t *sem , int pshared, unsigned int val ) ;
int sem_wait ( sem_t *sem ) ;
int sem_trywait ( sem_t *sem ) ;
int sem_post ( sem_t *sem ) ;
int sem_getvalue ( sem_t *sem , int *sval ) ;
int sem_destroy ( sem_t *sem ) ;
```

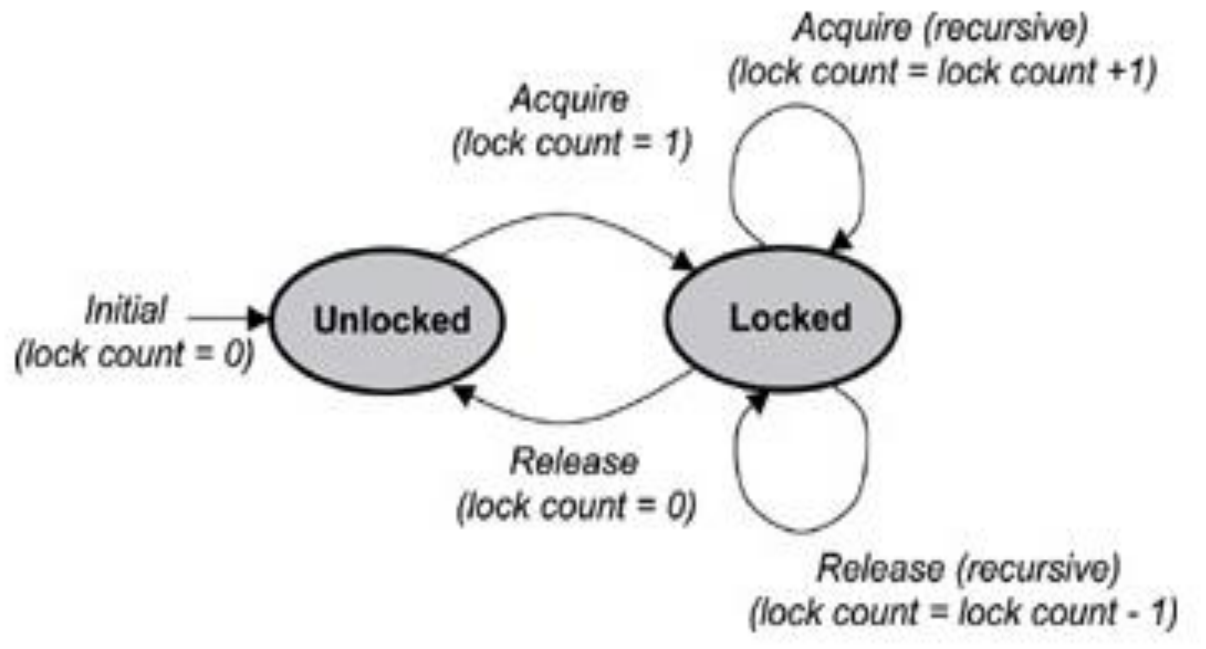
POSIX Mutexes

A mutex is a MUTual EXclusion device, and is useful for protecting shared data structures from concurrent modifications, and implementing critical sections.

A mutex has two possible states: unlocked (not owned by any thread), and locked (owned by one thread).

```
#include < pthread .h >
int pthread_mutex_init ( pthread_mutex_t *mutex,
      const pthread_mutex  attr_t *mutexattr ) ;
int pthread_mutex_lock ( pthread_mutex_t  *mutex );
int pthread_mutex_unlock (pthread_mutex_t  *mutex);
int pthread_mutex_destroy (pthread_mutex_t  *mutex);
int pthread_trylock ( pthread_mutex_t  *mutex ) ;
```

POSIX Mutexes



pthread_mutex_init()

The mutex variable should be declared and initialized only once as given below:

```
pthread_mutex_t mutex;
pthread_mutex_init (&mutex, NULL);
```

Another way to create a mutex with default attributes is to initialize it with the special value PTHREAD_MUTEX_INITIALIZER. No additional call to pthread_mutex_init is necessary.

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

pthread_mutex_lock()

A shared global variable x can be protected by a mutex as follows:

```
int x;
pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;
```

All accesses and modifications to *x* should be bracketed by calls to **pthread_mutex_lock** and **pthread_mutex_unlock** as follows:

```
pthread_mutex_lock(&mut);
 /* operate on x */
pthread_mutex_unlock(&mut);
```

Deadlocks

Deadlocks can occur when two (or more) threads are each blocked, waiting for a condition to occur that only the other one can cause.

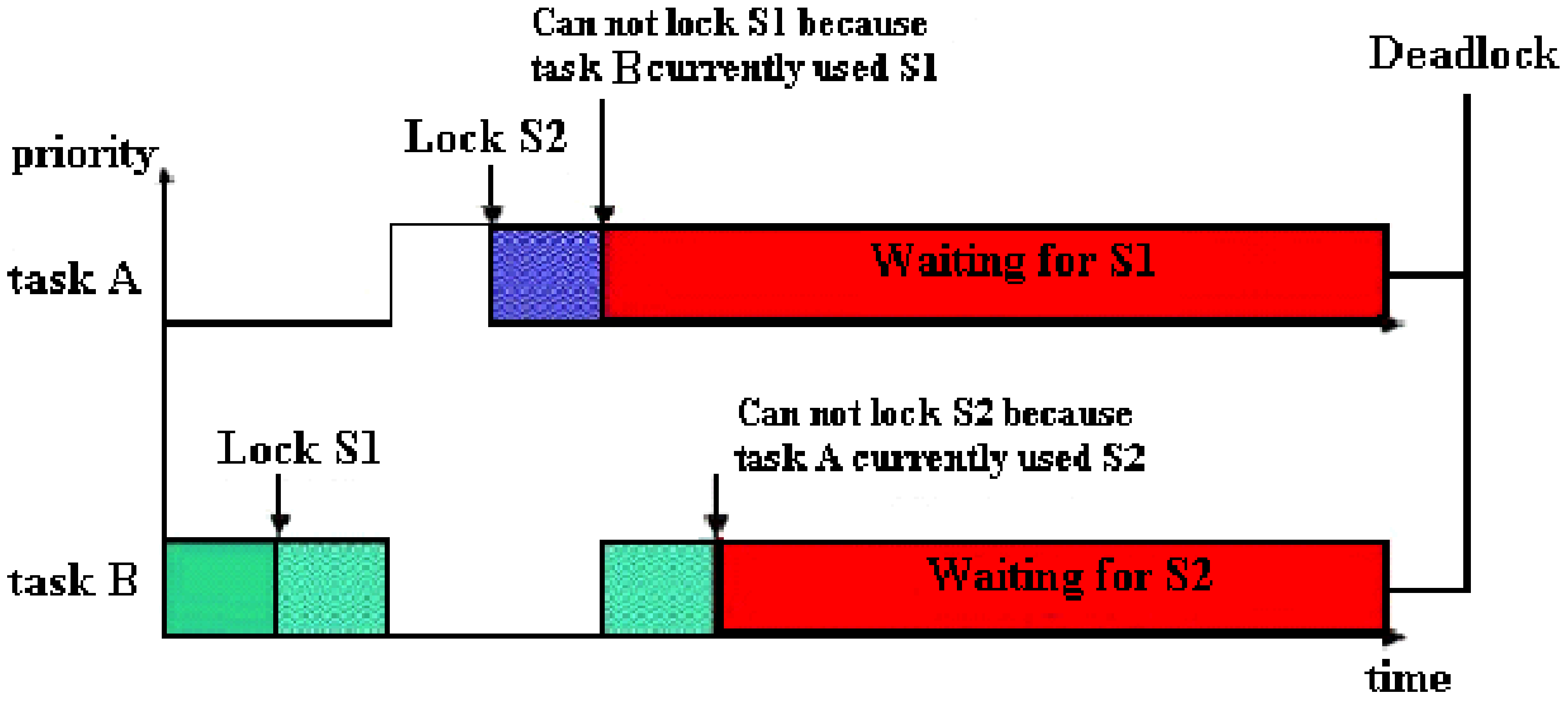
For instance, if thread A is blocked on a condition variable waiting for thread B to signal it, and thread B is blocked on a condition variable waiting for thread A to signal it, a deadlock has occurred Because neither thread will ever signal the other.

You should take care to avoid the possibility of such situations because they are quite difficult to detect.

Deadlocks

Semaphores can also result in deadlocks. Assume two tasks share two resources and lock them in different orders. One task (Task B) locks the first resource using semaphore S1. Then, in the ordinary course of events a second task (Task A) runs and locks the second resource with semaphore S2. It needs to use the first resource (S1), but cannot because it is locked by the first task (Task B). All it can do is pass control back to the first task (Task B), which then attempts to use the second resource (S2). It cannot use it because the second task (Task A) has locked it, so it passes control back to the second resource. But the second resource cannot run, so it passes control back to the first task, endlessly. The two tasks are deadlocked because each is waiting on a semaphore locked by the other task.

Two tasks reaching deadlock



Can not lock S1 because task B currently used S1
Deadlock
priority
Lock S2
task A
Waiting for S1
Can not lock S2 because task A currently used S2
Lock S1
task B
Waiting for S2
time