Concurrency: Pthreads Tutorial

Based on notes by Andrae Muys

The Process Model

- All Unix operating systems are multi-tasking
- Process model permits a user to run multiple processes simultaneously: concurrency
- Unix has one of the most powerful and flexible multi-programming models
- Traditional Unix model creates processes by using the fork() system call

Fork()

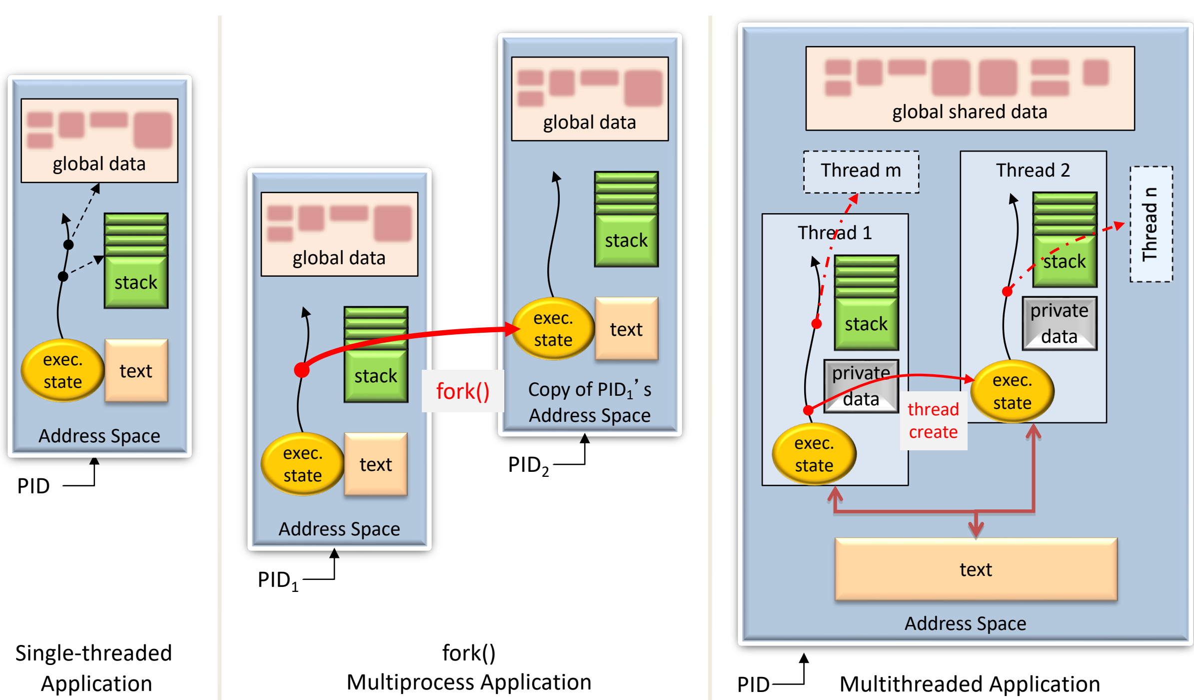
- Fork() produces a second copy of the calling process
- Processes are identified by process ID (pid)
- Second copy is identical to the original except:
  - Return value of fork() in the child = 0,
  - Return value of fork() in the parent = the child's pid

Normal Use of Fork()

```
// do parent stuff
ppid = fork();
If (ppid < 0) {
        fork_error_function();
} else if (ppid == 0) {
        child_function();
} else {
        parent_function();
}
```

Fork()

- Fork() works because it returns two ***completely independent*** copies of the original process
- Each process has its own address space
  - Its own copies of the **same** variables



Issues with Separate Processes

- Advantage: independence provides memory protection and stability, however…
- Challenges when multiple processes work on parts of the same task/problem
- One can use pipes or other inter-process communication (IPC), but there are inefficiencies:
  - Cost of switching between multiple processes is relatively high
  - Synchronization variables, shared between multiple processes, are typically slow
  - Often severe limits on the number of processes the scheduler can handle efficiently

Threads Alternative

- To avoid previous problems, threads or Light Weight Processes (LWP) can be very useful
- Threads:
  - Share a common address space
  - Are often scheduled internally in a process
  - Avoids many of the inefficiencies of multiple processes
- A popular API for threading an application is pthreads
  - Also known as POSIX threads (e.g., P1003.1c, or ISO/IEC 9945-1:1990c)

Benefits of Threads

- Support concurrency
- A number of situations where threads can simplify writing clean and efficient programs:
  - Blocking IO
  - Multiple processors/cpus
  - User interface
  - Responsive application servers

Blocking I/O

- Programs that do a lot of IO have three options:
  1. They can either do the IO serially, waiting for each to complete before commencing the next
  2. They can use asynchronous IO, dealing with all the complexity of asynchronous signals, polling or selects
  3. They can use synchronous IO, and just spawn a separate thread/process for each IO call
- In this case threading can significantly improve both performance and code complexity

Multiple Processors

- If you are using a threads library that supports multiple processors, you can gain significant performance improvements by running threads on each processor
  - CPU sockets with many cores are common now
- This is particularly useful when your program is compute bound

User Interface

- By separating the user interface, and the program engine into different threads you can allow the UI to continue to respond to user input even while long operations are in progress
  - E.g., music player involves UI, audio processing, file and network IO

Application Servers

- Servers that serve multiple clients can be made more responsive by the appropriate use of **concurrency**
  - Traditionally has been achieved by using the fork() system call
- However, in some cases, especially when dealing with large caches, threads can help improve the memory utilization, or even permit concurrent operation where fork() is unsuitable

Data Races

- Multiple threads share a common address space
- The problem can be worse on some hardware, where ‘a = data’ is non-atomic
  - E.g., when data is loaded into ‘a’, you could end up with the low order bits of the old data, and the high order bits of the new ‘data’
- Load, update, store operations can become interleaved for different threads

```
THREAD 1                    THREAD 2
a = data;                   b = data;
a++;                        b--;
data = a;                   data = b;
```

Now if this code is executed serially (THREAD 1, the THREAD 2) there isn't a problem. However, threads execute in an arbitrary order, so consider this:

```
THREAD 1                    THREAD 2
a = data;
                            b = data;
a++;
                            b--;
data = a;
                            data = b;
/* data = data - 1 !!!!!!! */
```

- Data could end up +1, 0, -1
- No way to know which as it is completely non-deterministic

Data Races (cont'd)

- Can use atomic variables for some shared access
- But the general solution to this is to provide functions that will block a thread if another thread is accessing data that it is using
- Pthreads library uses a data type called a mutex to achieve this

***

Thread Creation

- The function pthread_create() creates a new thread
- *pthread_t* is an opaque type which acts as a handle for the new thread
- *attributes* is another opaque data type which allows you to fine tune various parameters, to use the defaults pass NULL
- *thread_function* is the function the new thread is executing, the thread will terminate when this function terminates, or it is explicitly killed
- *arguments* is a void* pointer which is passed as the only argument to the *thread_function*

```
#include <pthread.h>

int
pthread_create (pthread_t *thread_id, const pthread_attr_t *attributes,
                void *(*thread_function)(void *), void *arguments);
```

Thread Exit

- Terminates when the thread function returns, or the thread can call pthread_exit() which terminates the calling thread explicitly
- Thread can provide its status is the return value of the thread
- Note the *thread_function* returns a void *, so calling return (void *) is the equivalent of this function

```
#include <pthread.h>

int
pthread_exit (void *status);
```

Thread Join

- One thread can wait on the termination of another by using pthread_join()
- Can retrieve status from the terminated thread

```
int
pthread_join (pthread_t thread, void **status_ptr);
```

Thread ID

- A thread can get its own thread id, by calling pthread_self()

```
pthread_t
pthread_self ();
```

- Two thread ids can be compared using pthread_equal()
- Returns zero if the threads are different threads, non-zero otherwise

```
int
pthread_equal (pthread_t t1, pthread_t t2);
```

Mutexes

- Mutexes have two basic operations
  - lock and unlock
- If a mutex is *unlocked* and a thread calls lock, the mutex locks and the **thread continues**.
- If, however, the mutex is *locked*, the **thread blocks** until the thread 'holding' the lock calls unlock
- There are five basic functions dealing with mutexes

```
#include <pthread.h>

pthread_mutex_t *mutex;
```

Mutexes

- Initialize a mutex and attributes for it
- Just pass NULL as the second parameter to use the default attributes
  - Attributes are not required to be implemented
- Attributes can control the priority and sharing behavior
- Returns zero if successful

```
int
pthread_mutex_init (pthread_mutex_t *m, const pthread_mutexattr_t *attr);
```

- Deallocates any memory or other resources associated with the mutex
- Should be unlocked (otherwise returns EBUSY)

```
int
pthread_mutex_destroy (pthread_mutex_t *m);
```

Mutex Lock/Unlock

- Locks the mutex
- Try either acquires the lock if it is available, or returns EBUSY
- Return zero if successful

```
int
pthread_mutex_lock (pthread_mutex_t *m);

int
pthread_mutex_trylock (pthread_mutex_t *m);
```

- Unlocks the mutex

```
int
pthread_mutex_unlock (pthread_mutex_t *m);
```

Example

```
THREAD 1                          THREAD 2
pthread_mutex_lock (&mut);
                                  pthread_mutex_lock (&mut);
a = data;                         /* blocked */
a++;                              /* blocked */
data = a;                         /* blocked */
pthread_mutex_unlock (&mut);      /* blocked */
                                  b = data;
                                  b--;
                                  data = b;
                                  pthread_mutex_unlock (&mut);


/* data is fine.  The data race is gone. */
```

- Fix data race with mutexes

Condition Variables

- Mutexes allow one to avoid data races
- But while they allow one to protect an operation, they don't permit one to wait until another thread completes an arbitrary activity
  - I.e., wait for a condition to be true
- Condition Variables solve this problem
- There are six operations, which can be done on a condition variable

```
#include <pthread.h>

pthread_cond_t *cond;
```

Condition Variables

- Initialization
- Just pass NULL as the second parameter to use the default attributes (e.g. private vs pshared)

```
int
pthread_cond_init (pthread_cond_t *m, const pthread_condattr_t *attr);
```

- Deallocation of resources for a condition variable

```
int
pthread_cond_destroy (pthread_cond_t *cond);
```

Condition Variables Wait

- Waiting: this function always blocks

```
int
pthread_cond_wait (pthread_cond_t *cond, pthread_mutex_t *mut);
```

- Note that it releases the mutex before it blocks, and then re-acquires it before it returns -- this is very important.
- Also note that re-acquiring the mutex can block for a little longer, so the condition that was signaled will need to be rechecked after the function returns.
- Pseudo-code description:

```
pthread_cond_wait (cond, mut)
begin
        pthread_mutex_unlock (mut);
        block_on_cond (cond);
        pthread_mutex_lock (mut);
end
```

Condition Variables Signal

- This wakes up *at least one* thread blocked on the condition variable. Remember that they must each re-acquire the mutex before they can return, so they will exit the block one at a time

```
int
pthread_cond_signal (pthread_cond_t *cond);
```

- This wakes up all of the threads blocked on the condition variable. Note again they will exit the block one at a time

```
int
pthread_cond_broadcast (pthread_cond_t *cond);
```

CV Signal Test

- "Always test your predicate; and then test it again!"
- Predicate may not always be true
  - Intercepted wakeups: race condition for another thread acquiring the the mutex first
  - Loose predicates: signal based on loose conditions; may have been accidentally signaled
  - Spurious wakeups: hard to make wakeups completely predictable on some systems (e.g., interrupts). Rare
- Without re-test one could have seemingly random application errors
- Lost wakeup:
  - Signal/Broadcast lost if another thread is between the test of the condition and the call to pthread_cond_wait() without the mutex lock

Condition Variables Timeout

- Waiting with timeout
- Identical to pthread_cond_wait(), except it has a timeout. This timeout is an absolute time of day

```
int
pthread_cond_timedwait (pthread_cond_t *cond, pthread_mutex_t *mut,
                        const struct timespec *abstime);
```

- If an *abstime* has passed, then pthread_cond_timedwait() returns ETIMEDOUT

- Timeout structure

```
struct timespec {
        time_t tv_sec;
        long tv_nsec;
};
```

Semaphores

- Another synchronization primitive
- Specified as another POSIX standard
- Initialized with an integer value *n*
  – e.g., binary semaphore *n* = 1 or counting semaphore *n* > 1
- Two operations defined on it
  – "Down", i.e., sem_wait()
  – "Up": i.e., sem_post()
- Can do mutual exclusion or schedule ordering of operations

```
#include <semaphore.h>
#include <time.h>

int
sem_wait (sem_t *s);
int
sem_trywait (sem_t *s);
int
sem_timedwait (sem_t *s, const struct timespec *abstime);

int
sem_post (sem_t *s);
```

Semaphores (2)

- Initialize with an integer value, e.g., 1
- Behavior of the two operations:

```
int sem_wait(sem_t *s) {
    wait until value of s > 0
    decrement the value of s by 1
}
```

```
int sem_post(sem_t *s) {
    increment the value of s by 1
    if there are 1 or more
       threads waiting, wake 1
}
```

- Can be used to control thread ordering, e.g., initialize semaphore with *zero*

```
// THREAD 1                    // THREAD 2
// do prep work
sem_post(s);
                               sem_wait(s);
                               // finish work
```

Semaphores (3)

- **Named** and **unnamed** semaphores
  - Named semaphores provide shared access between multiple processes
  - Unnamed semaphores provide multiple accesses in a single process or between related processes
- Some semaphore functions are specific to operate on named or unnamed semaphores
  - E.g., sem_init() and sem_destroy() operate on unnamed ones *only*
- Named semaphores are *persistent*, need to call the sem_unlink() after a system restart
- After calling sem_unlink(), need the sem_open() to establish new semaphores
- Named semaphore e.g. , removing persistence:
  - sem = sem_open("/semaphore", O_CREAT, 0644, 1)
  - sem_close(sem)
  - sem_unlink("/semaphore") # (error if return value == -1)

```
#include <semaphore.h>

sem_t *
sem_open (const char *name, int oflag, mode_t mode,
          unsigned int value);
```