Threads & Locks

Srinidhi Varadarajan

Topics

- **Thread Programming (Chapter 12)**
  - **Advantages/Disadvantages**
  - **Mutex Locks**
  - **Semaphore Locks**
  - **Condition Variables**
- **File Locking Mechanisms**

Advantages of threads

- **Lower context switching overhead**
- **Shared state.**
  - **Allows concurrent instances of the server to communicate easily with each other**
- **Linux supports the POSIX threads standard.**
  - **PTHREADS library**
  - **Portable across most UNIX platforms.**
  - **FSF project has largely ported pthreads to windows platforms as well.**

Disadvantages of Threads

- **Shared state**
  - **Global variables are shared between threads: Inadvertent modification of shared variables can be disastrous**
- **Many library functions are not thread safe.**
  - **Library functions that return pointers to internal static arrays are not thread safe. E.g. gethostbyname() used for DNS lookup**
- **Lack of robustness: If one thread crashes, the whole application crashes**

Thread state

- **Each thread has its own stack and local variables**
- **Globals are shared.**
- **File descriptors are shared. If one thread closes a file, all other threads can’t use the file**
- **I/O operations block the calling thread.**
  - **Some other functions act on the whole process. For example, the exit() function operates terminates the entire and all associated threads.**

Thread Synchronization: Mutex

- **How can a thread ensure that access/updates to shared variables is atomic?**
- **How can a thread ensure that it is the only thread executing some critical piece of code?**
  - **Need a mechanism for thread coordination and synchronization**
  - **Enter semaphores and mutex calls**
- **Mutex: Mutual Exclusion Lock.**
  - **Threads can create a mutex and initialize it. Before entering a critical region, lock the mutex.**
  - **Unlock the mutex after exiting the critical region**

Thread Synchronization: Semaphores

- **A mutex allows one thread to enter a critical region. A semaphore can allow some N threads to enter a critical region.**
  - **Used when there is a limited (but more than 1) number of copies of a shared resource.**
- **Can be dynamically initialized.**
  - **Thread calls a semaphore wait function before it enters a critical region.**
- **Semaphore is a generalization of a mutex.**

Conditional Variables

- **A set of threads use a mutex to allow serial access to a critical region.**
- **Once a thread enters a critical region, it needs to check for a condition to occur before proceeding.**
  - **This scenario is prone to deadlocks. A thread can't busy wait checking for the condition. Why? (Hint: what if the condition is set within a mutex protected region)**
- **Wasteful solution:**
  - **Thread enters mutex region, checks condition. If condition has not occurred, release mutex and repeat the process after some time**

Conditional Variables

- **A condition variable allows a thread to release a mutex and block on a condition atomically.**
- **When the condition is signaled, the thread is allowed to reacquire the mutex and proceed.**
  - **Two forms of signaling exist based on how many threads are blocked on the condition.**
  - **Either one thread may be allowed to proceed or all threads blocked on the condition are allowed to proceed.**

File Locking

- **File locking functions allow you to:**
  - **Lock entire files for exclusive use**
  - **Lock regions in a file**
  - **Test for locks held by other programs**
- **Function:**
  - **flock(int fd, int operation) where operation is:**
    - **LOCK_SH: Shared Lock**
    - **LOCK_EX: Exclusive Lock.**
    - **LOCK_UN: Unlock**
    - **LOCK_NB: Non blocking lock. Returns –ve result if lock can't be obtained**

Record Locking

- **The flock function locks the entire file. Record locking can be used to lock regions within a file**
- **Record locking uses the flock structure.**

```
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
struct flock {
off_t l_start; /* starting offset */
off_t l_len; /* len = 0 means until EOF */
pid_t l_pid; /* lock owner */
short l_type; /* F_RDLCK, F_WRLCK, F_UNLCK*/
short l_whence; /* type of l_start */
};
```

Record Locking

Type of lock desired: (l_type)

- **F_RDLCK: A shared read lock**
- **F_WRLCK: An exclusive write lock**
- **F_UNLCK: Unlocking a region**

Lock l_len bytes starting from (l_whence + l_start)

l_whence: SEEK_SET, SEEK_CUR, SEEK_END

To lock entire file set: l_start to 0, l_whence to SEEK_SET, and l_len to 0.

Record Locking

- **`int fcntl(int filedes, int cmd, struct flock *lock);`**

- **filedes: File descriptor**
- **cmd:**
  - **F_GETLK: Returns the lock struct of the lock preventing a file lock or sets the l_type to F_UNLCK on no obstruction**
  - **F_SETLK: Non-Blocking call to lock or unlock a region. Depends on the command inside the flock struct. Returns –1 if lock is held by someone else**
  - **F_SETLKW: Blocking version of F_SETLK**
- **struct flock *lock**

Record Locking: Example

```
struct flock lock;
FILE* myFile;
int fd;
if(( fd = creat("templock", FILE_MODE)) < 0 )
/* error */;
lock.l_len = 0;
lock.l_start = 0;
lock.l_whence = SEEK_SET;
lock.l_type = F_WRLCK;
fcntl(fd, F_SETLKW, lock);
myFile = fopen("mylog", "a");
fprintf(myFile, "Write\n");
fclose(myFile);
lock_l_type = F_UNLCK;
fcntl(fd, F_SETLKW, lock);
```