Unix System Programming - Chapter 13

Neal Nelson

The Evergreen State College

May 2010

USP Chapter 13 - Thread Synchronization

- Section 13.1 - POSIX Synchronization Facilities
- Section 13.2 - Mutex Locks
- Section 13.3 - At-Most-Once and At-Least-Once Execution
- Section 13.4 - Condition Variables
- Section 13.5 - Signal Handling and Threads
- Section 13.6 - Readers and Writers
- Section 13.7 - A strerror_r implementation
- Section 13.8 - Deadlocks and Other Pesky Problems
- Section 13.9 - Exercise: Multiple Barriers

POSIX Synchronization Facilities

- Mutex Locks - Short locks for mutual exclusion to shared data
- Condition Variables - Signals and Waits like Java
- Read-write locks - Readers and Writers Mutual Exclusion

Mutex Locks

- Initialize - Static, or dynamic init function. Static init uses default attributes. Use dynamic init for dynamically allocated mutex variables or if you need to set non-default attributes.
- pthread_mutex_lock function.
- pthread_mutex_unlock function.
- Mutexes are efficient thread synchronization.
- Meant for short hold-time locks.
- Maybe a spinlock is used, the USP usage guidance implies that.
- Not interrupted by signals except termination or thread_exit.
- Use for making changes to data structures in which the state of the structure is temporarily inconsistent, like updating pointers in shared linked lists.

Mutex Lock Examples - Shared counter, counter.c

- counter.c - shared counter.
- randsafe.c - protecting unsafe library functions.
- doneflag.c - synchronization with flags.
- globalerror.c - synchronizing global error settings.
- sharedsum.c - synchronizing global variables.
- listlib_r.c - thread safe linked list data structure from Ch 2.

Mutex Lock Example - counter.c shared counter

- counter.c encapsulates locking within functions to facilitate proper locking protocols.
- Static initialization at the top.

Mutex Lock Example - listlib_r.c thread-safe list library

- A thread-safe linked list data structure for the Chapter 2 listlib.
- The program really just wraps the Chapter 2 listlib functions. The program needs to have an include "listlib.h" at the top.
- Note the unlocking in the error exits as well as normal exits
- errno is saved across locking and unlocking calls.

At-Most-Once, and At-Least-Once Execution

- Normally you initialize synchronization in main before you create threads.
- Sometimes this is not possible, so we need a special mechanism to allow many threads to get just one thing done at most.
- That's what the `pthread_once` function is for.
- The `pthread_once` is called by each thread, but the thread runtime system makes sure that the first call happens and the rest do not.
- `pthread_once` takes a `once_control` parameter and an init_routine that you provide. The once_control needs to be initialized.
- See the `printinitonce.c` example USP program 13.10 p462

POSIX Condition Variables

- Condition variables are used to allow a process to wait until arbitrary condition is satisfied.
- Use condition variables to synchronize on events of indefinite duration, like waiting for input.
- POSIX condition variables are of type `pthread_cond_t` and must be initialized either statically or dynamically (USP p467)
- POSIX condition variables are not associated with any particular condition. They are just a special identifier, like a mutex.
- The `pthread_cond_wait` function takes a condition variable and a mutex as parameters and automatically suspends the calling thread and unlocks the mutex.
- To properly synchronize on an event you need both the condition variable and the mutex.

Waiting for Condition Variables

- Let m be a mutex and v be a condition variable.
- The sequence of actions we need when waiting for a condition such as x==y is as follows (note the condition is reversed):

```
pthread_mutex_lock(&m);
while (x!=y);
   pthread\_cond\_wait(&v, &m);
pthread\_mutex\_unlock(&m);
```

- The mutex protects the variables x and y from interference between testing of the condition and subsequent suspension.
- Other threads that use x and y must protect them by using the same mutex.
- The mutex must be unlocked when the thread is suspended so that other threads can use x and y.
- The `pthread_cond_wait` returns with the mutex re-acquired when it has been notified.
- The thread must test the condition again before proceeding - thus the test wait is done in a loop.

Signaling Condition Variables

- A thread suspended by a `pthread_cond_wait` has the illusion of uninterrupted mutex ownership.
- But while suspended the waiting thread does not own the mutex and it can be acquired by other threads.
- Releasing the mutex while a thread is suspended on a condition variable wait is key to letting other threads access and change the event conditions.
- For example, we want x and y released from the mutex so that another thread can change the x and the y. Maybe x will equal y upon awakening!
- Another thread that modifies x or y can call `pthread_cond_signal` to notify other threads of the change.

Signaling Condition Variables

- `pthread_cond_signal` takes a condition variable and wakes up at least one of the threads waiting in the corresponding condition variable queue.
- `pthread_cond_signal` in effect moves the suspended process from the condition variable queue to the head of the mutex queue so the thread will wake up holding the mutex as required.
- Here's some sample code signaling the condition variable *v* from above. The same *m* mutex is used here to protect access to the variable *x*.

```
pthread_mutex_lock(&m);
x++;
pthread_cond_signal(&v);
pthread_mutex_unlock(&m);
```

- The signal happens before the mutex unlock to avoid the Mars Rover priority inversion problem of a low priority process grabbing the mutex before the signal can happen.

Condition Variables vs Waiting Conditions

- Condition variables are not directly associated with waiting conditions.
- That is why event waiting must be done in a loop.
- When a thread wakes up from a condition variable wait, it already has the mutex lock, so it can immediately check the waiting condition safely.
- A thread waiting on a condition variable might be woken up even though the waiting condition was not satisfied.
- In the signal example above, if the code included y++ as well as x++, then x and y would both have changed, the signal that x or y changed would be sent, but the condition was still not satisfied.
- Threads can awaken waiting threads blocked on condition variables by using a `pthread_cond_broadcast` function that unblocks *all* threads blocked on a condition variable.
- The `phtread_cond_signal` is only guaranteed to unblock at least one.

A Thread-safe Barrier

- Program 13.13 tbarrier.c USP p472, a thread-safe barrier using condition variables.
- The `limit` variable specifies how many threads must arrive at the barrier (execute the `waitvbarrier`) before the threads are released from the barrier.
- The `count` variable specifies how many threads are currently waiting at the barrier.
- Both `limit` and `count` are declared global for sharing, and static (file scope) so they must be accessed through the `initvarrier` and `waitbarrier` functions in the same file.
- `limit` is only initialized once. All other attempts return EINVAL
- When `count` reaches limit, all threads are woken up.
- Only the last thread actually wakes up all the others.
- Look at all the berror checking.

Signal Handling and Threads

- Thread and Signal interaction is complicated because threads can operate asynchronously with signals.
- All threads in a process share the process signal handlers.
- Each thread has its own signal mask.
- Threads operate asynchronously with signals.
- But, *a recommended strategy for dealing with signals in multi-threaded processes is to dedicate particular threads to signal handling* (USP p475).

Types of signals

- **Asynchronous** - delivered to some thread that has it unblocked, from anywhere, at any time.
- If several threads have an asynchronous signal unblocked, the thread runtime system system selects one of them to handle the signal (USP p473).
- **Synchronous** - delivered to the thread that caused it, like SIGFPE floating point exception, an in-line exception.
- **Directed** - delivered to the identified thread by pthread_kill
- pthread_kill is directed to a particular thread, but may affect the entire process.
- A pthred_kill can kill an entire process for signals that cannot be blocked or ignored, like SIGKILL.
- Each thread has its own signal blocking mask.

Signal Masks for Threads

- Signal handlers are process-wide, but each thread has its own signal mask.
- Recall, sigmasks specify which signals are to be blocked and sigset objects are used to specify signal sets when setting the masks.
- Use `pthread_sigmask` function to examine or set a thread signal mask; it works as a generalization of `sigprocmask`.
- Do not use `sigprocmask` for multi-threaded programs, but you can use it in main *before* threads are created.

Dedicated Threads for Signal Handling - Yeah!

- Signal handlers are process-wide, but in a multi-threaded application signal masks are thread specific.
- In a single threaded application, the causative signal is blocked on entry to the signal handler.
- In a multi-threaded application another signal of the same type can be delivered to another thread that has the signal unblocked.
- So, it is possible to have multiple threads executing within the same signal handler.

Dedicated Threads for Signal Handling (USP p475)

- *a recommended strategy for dealing with signals in multi-threaded processes is to dedicate particular threads to signal handling* (USP p475, their emphasis).
- The main thread blocks all signals before creating the threads.
- The signal mask is inherited from the crating thread, so all threads have the signal initially blocked.
- The thread dedicated to handling the signal the executes a `sigwait` on that signal (See Section 8.5) or uses `pthread_sigmask` to unblock the signal.
- `sigwait` is not restricted to async-signal-safe functions, so maybe it's easier than unblocking signals using a `pthread_sigmask` call.

Signal Handling with Threads

- No signal handler is needed because the thread itself can entirely handle the signal
- The signal is delivered to the thread because it has unblocked that particular signal, either using `pthread_sigmask`, or using `sigwait`.
- The delivery of the signal to the thread or the `sigwait` removes the signal from those pending, so the signal handler never gets called.
- The sigwait function does not automatically unblock signals (see USP section 8.5.3 p282).
- The end result is that at the process level the signal is always blocked or delivered to some thread, so at the process level the default signal handler for the signal is never taken!
- In this way thread signal handling can be slipped in like a shim before the traditional process signal handling gets involved.

Example - signalthread.c and computethreadsig.c

- Draw the structure chart for the program.
- Blah, Blah code walkthrough.

Readers and Writers in POSIX pthreads

- The Reader-Writer problem:
- Writers must have exclusive access
- Readers may share access
- Example - Sharing read access to a file simultaneously
- A reader-writer lock has type `pthread_rwlock_t`.

Strong Reader vs Strong Writer

- Two variations of Reader-Writer problem:
- Strong reader - always gives precedence to readers ahead of waiting writers.
- Strong writer - always give precedence to all waiting writers over any waiting readers.
- Airline reservation system - strong writer to keep schedules current.
- Library reference database - strong readers to postpone non-critical updates until system activity is low.
- POSIX does not specify Strong reader or Strong writer, so it depends on the implementation.

POSIX Reader-Writers functions

- `pthread_rwlock_rdlock` - blocking read lock request
- `pthread_rwlock_tryrdlock` - non-blocking read lock request
- `pthread_rwlock_rwlock` - blocking write lock request
- `pthread_rwlock_tryrwlock` - non-blocking write lock request
- `pthread_rwlock_unlock` - unlock
- All return 0 if successful, nonzero error code if not.
- The trylocks return EBUSY if the lock could not be acquired because it was already held.

Initialization and destruction

- Reader-writer locks have no static initialization
- `pthread_rwlock_init` - dynamic initialization.
- Be sure to execute this only once, otherwise POSIX behavior is undefined.
- `pthread_rwlock_destroy`
- Referencing a lock that has been destroyed has undefined behavior
- You can re-initialize a destroyed lock.

Listlib with reader-writer locks

- See listing listlibrw_r.c
- A thread-safe listlib but now using reader-writer locks
- The code essentially wraps the original listlib functions from USP Program 2.7 p44.
- The code for is very similar to the mutex version of the thread-safe listlib (USP Program 13.9 p460).
- Mutex locks are more efficient, but all accesses to the list must be serialized.
- Reader-writer locks allow multiple readers into the `getdata` function.
- Mutex locks are probably better here because the list locks are only held for a very short time (USP Exercise 13.29 p480).
- Reader-writer locks are better for read operations that take a considerable amount of time, such as accessing a disk (USP Exercise 13.29 p480).

Thread-safe and async-safe strerror and perror

- Later, maybe.

Deadlocks and other Pesky problems

- POSIX does not require implementations to detect deadlock, so you the programmer must carefully prove your program deadlock-free.
- A common problem: *failing to release locks when threads fail.*
- Threads with priorities can complicate matters.
- The Mars Pathfinder problem and experience.
- Leaving in the debugging code with dynamic triggering if needed to spot problems while the software is online.
- Worth reading this section.

Multiple Barriers

- Exercise Section 13.9 - An implement of multiple barriers
- Uses mutex locks and condition variables
- Extends the thread-safe barrier program: USP Program 13.13 p 472
- Maintain an array or linked list of barriers
- What do you maintain for each barrier in the list nodes?
- Should you have a distinct set of mutexes and condition variables for each barrier or can you re-use them?

Done

- Done.