Threads

Programs and Threads

- What is/are the advantage(s) of having multiple threads?
- What is/are the disadvantage(s) of having multiple threads?

Thread Safe Functions

- Not all functions are thread safe
- Some have a thread-safe alternative
- `chdir()` vs `opendir()`

POSIX Threads (pthreads)

- Popular implementation of threads
- Process address space shared by all threads
- `pthread` functions
  - return 0 or error number
  - use `strerror()`
- Thread ID (pthread_t)
  - `pthread_self()`
  - `pthread_equal()`

Thread Creation

- `pthread_create()`
- Data structures: thread ID, stack, PC value
- Can pass a single argument to a thread
  - How would you pass multiple values to a thread
- After a thread has been created, how to you know when it will be scheduled to run by the OS?

Thread Termination

- `pthread` termination
  - return from starting routine
  - `pthread_exit()`
  - main thread (process) ends
  - cancelled by another thread `pthread_cancel()`
- The main thread should call `pthread_join()` for each thread created
  - This is similar to calling `wait()` when forking child processes

Thread Synchronization: Mutex Locks

- A mutex variable is used to provide mutual exclusion and can be in a locked state or an unlocked state.
- Functions:

```
pthread_mutex_destroy
pthread_mutex_init
pthread_mutex_lock
pthread_mutex_trylock
pthread_mutex_unlock
```

Thread Synchronization: Condition Variables

- A condition variable is used to have a thread wait until some condition is satisfied
- Condition variables follow this strategy
  1. Lock a mutex
  2. Test the condition
  3. If true, unlock the mutex and exit
  4. If false, suspend the thread and unlock the mutex

Thread Synchronization: Condition Variables

- Functions:

```
pthread_cond_broadcast
pthread_cond_destroy
pthread_cond_init
pthread_cond_signal
pthread_cond_timedwait
pthread_cond_wait
```

Condition Variable Example

- Code that waits for the condition `x == y` to be true using mutex `m` and condition variable `v`

```
pthread_mutex_lock(&m)
while (x != y) {
    pthread_cond_wait(&v, &m);
}
// modify x or y if necessary
pthread_mutex_unlock(&m);
```

- Code that might run in a separate thread

```
pthread_mutex_lock(&m);
// modify x or y
pthread_cond_signal(&v);
pthread_mutex_unlock(&m);
```

Condition Variable Usage

- Condition variables are not associated with particular predicates and `pthread_cond_signal` can return due to spurious wakeups
- General rules for using condition variables
  1. Acquire the mutex before testing the predicate
  2. Test the predicate after returning from a `pthread_cond_wait` because `pthread_cond_signal` might have been caused by some unrelated variable updates
  3. Acquire the mutex before changing variables present in the predicate
  4. Hold the mutext for a only a short period of time
  5. Release the mutex explicitly with `pthread_mutex_unlock` or implicitly with `pthread_cond_wait`

Thread Synchronization: Read-Write Locks

- The reader-writer problem refers to the situation where a resource allows read access and write access and write access must be exclusive
- Strategies for reader-writer synchronization
  - Strong reader synchronization: give preference to readers, that is, grant access to readers as long as there is not an active writer
  - Strong writer synchronization: give preference to writers, that is, delay readers until all waiting or active writers are finished
- POSIX read-write locks allow multiple readers to acquire the lock provided that a writer does not hold the lock

Thread Synchronization: Read-Write Locks

- Functions:

```
pthread_rwlock_destroy
pthread_rwlock_init
pthread_rwlock_rdlock
pthread_rwlock_timedrdlock
pthread_rwlock_timedwrlock
pthread_rwlock_tryrdlock
pthread_rwlock_trywrlock
pthread_rwlock_wrlock
```

Threads and Signals

- All threads in a process share the process signal handlers, but each thread has its own signal mask that can be set with `pthread_sigmask`
- Signal types and delivery:
  - asynchronous: delivered to some thread that has the signal unblocked
  - synchronous: delivered to the thread that caused it
  - directed: delivered to the identified thread via `pthread_kill`
- A recommended strategy in multi-threaded applications is to dedicate particular threads for signal handling