CSC209: Software Tools and Systems Programming

Week 12: Threads [1]

Kianoosh Abbasi

[1] Slides are mostly taken from Andi Bergen's in summer 2021.

Why Do We Use Multiple Processes?

- Isolation: each process is protected from the others
- Parallelism: if our computer has multiple processors, then each processor can be running a process!

But:

- `fork` is a heavyweight system call
- Communication between processes requires pipes, or sockets, or signals, etc.
- So, what if we had multiple *threads* of execution, all within a single process?

Threads vs. Processes

- Both processes and threads allow an application to perform multiple concurrent tasks
- Processes don't share memory. Threads do
- Process creation with `fork` is slow. Thread creation is much faster
- Pthreads is the API we use for managing threads

Threads

- Threads belong to a process and share the same PID and parent PID
- Threads of a process also share the heap and global variables
- Each thread also has unique attributes
  - Its own thread ID
  - Its own `errno` variable
  - Its own stack for local variables and function calls

Pthread Return Values

- Many function calls we have seen so far return `0` for success and `-1` for failure
- Pthreads functions are different
  - They return `0` for success and a positive integer for failure
  - If a function fails, we can store the positive integer into `errno` and then call `perror`

Creating New Threads

```
int pthread_create(pthread_t *thread,
  const pthread_attr_t *attr,
  void *(*start)(void *), void *arg);
```

- `thread` is where the ID of the new thread gets stored
- `attr` specifies attributes for the thread; use `NULL` for defaults
- `start` is a pointer to a function
- The new thread runs `start` with argument `arg`
- `arg` typically points to a heap or global variable

Thread Termination

A thread terminates when one of the following happens

- Its `start` function returns
- It calls `pthread_exit`
- It gets canceled by another thread using `pthread_cancel`
- Any thread in the process calls `exit`
  - So don't call `exit` unless you want the entire process to terminate!

Thread Joining

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

- This is similar to using `waitpid` to wait for a process to terminate
- `pthread_join` waits for a thread to terminate, or returns immediately if the thread has already terminated

Creating and Joining

Compile Pthreads programs with `gcc -pthread`.

```
errno = pthread_create(&t1, NULL, thread_func, "Hello\n");
if (errno != 0) {
  perror("pthread_create"); exit(1);
}
printf("main() before joining...\n");
errno = pthread_join(t1, &res);
if (errno != 0) {
  perror("pthread_join"); exit(1);
}
```

Thread Joining...

Differences between `pthread_join` and `waitpid`:

- No thread hierarchy. Any thread can use `pthread_join` to wait for any other thread
- There is no way to "join with any thread"
- There is no equivalent to `WNOHANG`

Detaching a Thread

```
int pthread_detach(pthread_t thread);
```

- We can detach a thread if we don't want to obtain its exit status
- If we detach a thread, it is cleaned-up automatically
- We must detach or join every thread

Disadvantages of Threads

When using threads:

- Only thread-safe functions can be used
- A bug in one thread can damage other threads
- Its difficult to use signals with threads
- All threads in a process must run the same program

Access to Shared Variables

- Threads can easily share information using global variables
- But we run into trouble unless we synchronize access to those variables
- *Critical section*: code that should be accessed by only one thread at a time

Access to Shared Variables...

```
for (j = 0; j < loops; j++) {
    loc = glob;
    loc = loc + 1;
    glob = loc;
}
```

- Suppose that `loops` is one million and that two threads run this
- The expected final value is 2000000
- But that probably isn't what we'll get!

Access to Shared Variables...

- Here's a possible (problematic) execution path
  1. Thread 1 increments `glob` 2000 times. On iteration 2001, it obtains the value of `glob`, but then ...
  2. Thread 2 increments `glob` 1000000 times, and terminates
  3. Now thread 1 takes over, but writes 2001 into `glob`!

Mutexes

- *Mutex*: mutual exclusion
- A mutex can be used to ensure that only one thread accesses a variable at a time
- A mutex is always in one of two states: locked or unlocked
- When unlocked, a thread can lock the mutex
- Any thread that tries to obtain a locked mutex is blocked until the mutex is unlocked
- Only the thread that locked the mutex is allowed to unlock it

Mutexes. . .

The pattern for using a mutex:

```
pthread_mutex_lock(&mtx);
... access shared resource
pthread_mutex_unlock(&mtx);
```

Condition Variables

- A *condition variable* (CV) Allows a thread to notify other threads that a shared resource has changed
- A CV also allows threads to block waiting for such notification
- Without using a CV, threaded programs can be very inefficient
  - e.g. they may loop quickly to poll a variable value

Condition Variables...

CVs have three core operations.

- `signal`: wakes up at least one thread waiting for the CV
- `broadcast`: wakes up all threads waiting for the CV
- `wait`: waits (blocks) until signaled by a CV

Condition Variables...

- A CV always has an associated mutex
- The mutex must be locked by a thread before it calls `pthread_cond_wait`
- `pthread_cond_wait` unlocks the mutex, blocks the thread, and (when the thread is later signaled) relocks the mutex
- Unlocking the mutex and blocking the thread are *atomic*
  - This means that no other thread can `signal` the CV between the time that the mutex is unlocked and the time that the thread blocks on the CV