## Lecture 4

### Thread Programming

January 21, 2005

### Threads vs. Processes

- `fork()` is expensive (time, memory)
- Interprocess communication is hard.
- Threads are ‘lightweight’ processes:
  - one process can contain several threads of execution.
  - all threads execute the same program (different stages).
  - all threads share instructions, global memory, open files, and signal handlers.
  - each thread has own thread ID, stack, program counter and stack pointer, errno, signal mask.
  - threads can communicate with shared memory.
  - threads have special synchronization mechanisms.

### Threads in C

- POSIX threads (pthreads): standard for Unix
- OS must support it (Linux)
- Programs must be linked with `-lpthread`

### Pthreads

- Creating a thread:
  ```c
  #include <pthread.h>
  int pthread_create(pthread_t *tid, pthread_attr_t *attr, void *(*start_routine)(void *), void *arg);
  ```
  - tid: thread id
  - attr: options
  - start_routine: function to be executed
  - arg: parameter to thread
Pthreads
- Stopping a pthread: a thread stops when
  - the process stops,
  - the parent thread stops,
  - its start_routine function return,
  - or it calls pthread_exit:

```c
#include <pthread.h>
void pthread_exit(void *retval);
```

Pthreads
- Threads must be waited for:

```c
#include <pthread.h>
int pthread_join(pthread_t tid, void **status);
```

Pthreads Example
```c
#include <pthread.h>

void *func(void *param) {
    int *p = (int *) param;
    printf("This is a new thread (%d)
", *p);
    return NULL;
}

int main () {
    pthread_t id;
    int x = 100;
    pthread_create(&id, NULL, func, (void *)&x);
    pthread_join(id, NULL);
}
```

Pthreads
- A thread can be joinable or detached.
- Detached: on termination all thread resources are released, does not stop when parent thread stops, does not need to be pthread_join().
- Default: joinable (attached), on termination thread ID and exit status are saved by OS.

Pthreads
- Creating a detached thread:

```c
#include <pthread.h>

pthread_t id;
pthread_attr_t attr;

pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
pthread_create(&id, &attr, func, NULL);

pthread_detach();
```

Pthreads
- A thread can join another:

```c
int pthread_join (pthread_t tid, void **status);
```
- Call waits until specified thread exits.
Pthreads

```c
int counter = 0;
void *thread_code (void *arg) {
    counter++;
    printf("Thread %u is number %d\n",
           pthread_self(), counter);
}
main () {
    int i; pthread_t tid;
    for (i = 0; i < 10; i++)
        pthread_create(&tid, NULL, thread_code, NULL);
}
```

Quiz

- A server creates a thread for each client. No more than n threads can be active (or n clients can be serviced). How can we let the main thread know that a thread terminated and that it can service a new client?

What About Global Variables?

- Thread startup:
  - acquire lock on the variable
  - increment variable
  - release lock
- Thread termination:
  - acquire lock on the variable
  - decrement variable
  - release lock

What About the Main Loop?

```c
active_threads = 0;
// start up first n threads for first n clients
// make sure they are running
while (1) {
    // have to lock/release active_threads;
    if (active_threads < n)
        // start up thread for next client
        busy_waiting(is_bad);
}
```

What About pthread_join?

- `pthread_join()` is kinda like `wait()`.
- Requires thread id, so we can wait for thread xy, but not for the ‘next’ thread.

What About Pthread?

- Mutual exclusion:
  ```c
  pthread_mutex_t counter_mtx = PTHREAD_MUTEX_INITIALIZER;
  ```
- Locking (blocking call):
  ```c
  pthread_mutex_lock(pthread_mutex_t *mutex);
  ```
- Unlocking:
  ```c
  pthread_mutex_unlock(pthread_mutex_t *mutex);
  ```
Condition Variables

- Allow one thread to wait/sleep for event generated by another thread.
- Allows us to avoid busy waiting.
- Condition variable is ALWAYS used with a mutex.

```
condition_variable_t foo = PTHREAD_COND_INITIALIZER;
```

Problem Revisited

- Each thread decrements active_threads when terminating and calls `pthread_cond_signal()` to wake up main loop.
- The main thread increments active_threads when a thread is started and waits for changes by calling `pthread_cond_wait`.
- All changes to active_threads must be ‘within’ a mutex.
- If two threads exit ‘simultaneously’, the second one must wait until the first one is recognized by the main loop.
- Condition signals are NOT lost.

```
int active_threads = 0;
while (1) {
    pthread_mutex_lock(&at_mutex);
    active_threads--;
    pthread_cond_signal(&at_cond);
    pthread_mutex_unlock(&at_mutex);
}
```

Condition Variables

```
pthread_cond_wait(pthread_cond_t *cptr, pthread_mutex_t *mpt);
```

```
pthread_cond_signal(pthread_cond_t *cptr);
```

Condition Variables

```
void *handler_fct(void *arg) {
    // handle client
    pthread_mutex_lock(&at_mutex);
    active_threads--;
    pthread_cond_signal(&at_cond);
    pthread_mutex_unlock(&at_mutex);
    return();
}
```

```
if active_threads == 0;
```

```
while (active_threads < n) {
    active_threads++;
    pthread_start(...);
}
```

Condition Variables

- Multiple ‘waiting’ threads: signal wakes up exactly one, but not specified which one.
- `pthread_cond_wait` atomically unlocks mutex.
- When handling signal, `pthread_cond_wait` atomically re-acquires mutex.
- Avoids race conditions: a signal cannot be sent between the time a thread unlocks a mutex and begins to wait for a signal.
Thread Safeness

- Some Unix primitives and library functions have internal race conditions and were not designed with threads in mind!
- Check on the man pages if library functions are thread-safe.
- Example: gethostbyname is not thread-safe, use it within a mutex protection.

Java Thread Programming

- Java thread is a class which inherits from Thread, you must overload its run() method:
  
  ```java
  public class MyThread extends Thread {
      private int argument;
      MyThread(int arg) {
          argument = arg;
      }
      public void run() {
          System.out.println("New thread created! Arg= " + argument);  
      }
  }
  ```

Java Thread Programming

- Starting a thread:
  
  ```java
  MyThread t = new MyThread(42);
  t.start();
  ```

- Stopping a thread: when run() returns, no 'join' needed for thread to finish, but you can:
  
  ```java
  MyThread t = new MyThread(42);
  t.start();
  ...
  t.join();
  ```

Java Thread Programming

- Monitor (similar to mutex):
  
  ```java
  public class AnotherClass {
      synchronized public void methodOne() {...}
      synchronized public void methodTwo() {...}
      public void methodThree() {...}
  }
  ```

- Each object contains one mutex which is locked when entering a synchronized method and unlocked when leaving.
- Two synchronized methods from same object cannot be executing simultaneously.
- Two different objects from the same class can be executing the same synchronized method simult.