CMSC 313
Introduction to Computer Systems
Lecture 21
Threads

Alan Sussman
als@cs.umd.edu

Administrivia

- Project 5 due tomorrow at 6PM
  - questions?
  - Do NOT use the submit server to test your projects; follow the instructions in the project specification that explain how to test on Grace
- Exam #2 returned next week
- Project 6 assigned next week
- Read Chapter 13 of Bryant and O’Hallaron

Threads example

```c
#define THREAD_CT 2
void *print_stuff(void *ptr) {
  int i, id = * (int *) ptr;
  for (i = 0; i < 5; i++) {
    printf("Thread %d, loop %d\n", id, i);
    sleep(rand() % 2); /* sleep 0 or 1 seconds */
  }
  printf("Thread %d exiting\n", id);
  return NULL;
}

int main() {
  pthread_t tids[THREAD_CT];
  int i, ids[THREAD_CT];
  for (i = 0; i < THREAD_CT; i++) {
    ids[i] = i + 1;
    pthread_create(&tids[i], NULL, print_stuff, &ids[i]);
    printf("Thread 0 created thread %d\n", i + 1);
  }
  for (i = 0; i < THREAD_CT; i++) {
    pthread_join(tids[i], NULL);
    printf("Thread 0 reaped thread %d\n", i + 1);
  }
  return 0;
}
```
Execution of the example

$ ./simple
Thread 0 created thread 1
Thread 0 created thread 2
Thread 1, loop 0
Thread 2, loop 0
Thread 2, loop 1
Thread 1, loop 1
Thread 2, loop 2
Thread 1, loop 2
Thread 2, loop 3
Thread 2, loop 4
Thread 2 exiting
Thread 1, loop 3
Thread 1, loop 4
Thread 1 exiting
Thread 0 reaped thread 1
Thread 0 reaped thread 2

Return value example

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void *get_rand_num(void *args) {
    int *nump = malloc(sizeof(int));
    srand(pthread_self());
    *nump = rand();
    return nump;
}

int main() {
    pthread_t tid;
    void *ptr = NULL;

    pthread_create(&tid, NULL, get_rand_num, NULL);
    pthread_join(tid, &ptr);
    printf("Random number: %d\n", * (int *) ptr);
    return 0;
}
```

Bad return value example

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void *get_rand_num(void *args) {
    int num;
    srand(pthread_self());
    num = rand();
    return #}

int main() {
    pthread_t tid;
    void *ptr = NULL;

    pthread_create(&tid, NULL, get_rand_num, NULL);
    pthread_join(tid, &ptr);
    printf("Random number: %d\n", * (int *) ptr);
    return 0;
}
```

Thread detachment

- Threads are by default joinable, meaning they can be reaped and killed by other threads, and a thread's memory resources stay until it is reaped
- We can detach threads so that they cannot be reaped or killed by other threads, and memory resources are automatically freed upon termination
- To avoid memory leaks, threads should either be reaped by another thread, or detached
Thread detachment, cont.

```c
int pthread_detach(pthread_t tid);
-- detaches the thread tid
-- returns 0 on success, nonzero on error
```

- A thread can detach itself:
  ```
  pthread_detach(pthread_self());
  ```
- We might use detachment if we have a constantly active process, like a mail daemon or web server, which shouldn't need to take time to reap terminated threads

Threads memory model

- Remember that threads run within the context of a single process
- Each thread has its own context, including a thread ID, stack, stack pointer, PC, CCs, and registers; everything else is shared
- It is possible for one thread to access another thread's stack if a pointer is made accessible

Threads memory model, cont.

- Global variables are always shared; there is only one of each global variable
- Automatic local variables are thread-local; each thread has its own copy of these in its stack
- Static local variables are also shared, just as globals
  ```
  -- remember, static variables are just globals with restricted scope
  ```

Thread synchronization

- What will the following code output?
  ```c
  #define LOOPS 10000000
  static int count = 0;
  void *counter(void *args) {
    int i;   for (i = 0; i < LOOPS; i++)     count++;   printf("Executed %d times\n", i);   return NULL; }
  int main() {
    pthread_t tids[2];   pthread_create(&tids[0], NULL, counter, NULL);   pthread_create(&tids[1], NULL, counter, NULL);   pthread_join(tids[0], NULL);   pthread_join(tids[1], NULL);   printf("Count: %d\n", count);   return 0;
  }
  ```
Thread synchronization, cont.

- The first two lines will be: 
  **Executed 10000000 times**
- And the last will be:
  **Count: 11398345**
- Or maybe:
  **Count: 12398354**
- Or even...
  **Count: 15892348**
- But almost definitely not 
  "**Count: 20000000**" Why not?

Thread synchronization errors

- We might expect the increment to be an atomic operation, but it isn't
  - `i++` requires loading `i`'s value into a register, updating that value by adding 1 to it, then storing the result back into `i`'s location in memory
- Consider this schedule of events:
  - Thread A loads `i`'s value
  - Thread B loads `i`'s value
  - A updates register
  - B updates register
  - A stores register value in `i`
  - B stores register value in `i`
- What is the value stored in `i` now?
  - Only one more than it was before!

Thread synchronization errors, cont.

- In this example, we need to ensure that when one thread is in the middle of its load-update-store set of instructions, the other thread isn't in its own set
- For a given thread, these instructions constitute a **critical section** that should not have other threads' critical sections interleaved within them

Semaphores

- To properly implement a critical section, we can use semaphores
- These are special global variables, with nonnegative integer values, that can only be changed by two operations
  - in most literature, called `P` and `V`
  - we'll restrict ourselves to discussion of binary semaphores here, but semaphores can be more than just 0/1 values
- Used to ensure that only one thread is in a critical section at a time
Semaphores, cont.

- \( P(s) \), or wait operation
  - if \( s \) is 0, waits until \( s \) is 1
  - if \( s \) is 1, sets \( s \) to 0 (decrements \( s \))
- \( V(s) \), or post operation
  - if \( s \) is 0, sets \( s \) to 1 (increments \( s \)), and restarts
    exactly one of the processes waiting for \( s \) to be 1
  - should not be called except after a \( P(s) \) call
- All elements of these operations are indivisible (i.e., these operations are atomic)

Using semaphores

- Three functions, all in \(<semaphore.h>\)
- Each return 0 on success, -1 on error
  
  \[
  \text{int sem_init(sem_t *s, 0, unsigned int value);} \\
  \text{-- initializes the semaphore pointed at by s to value} \\
  \text{-- you must initialize semaphores before use} \\
  \text{-- the second argument will always be 0 for our purposes} \\
  \]
  
  \[
  \text{int sem_wait(sem_t *s);} \\
  \text{-- performs the } P(s) \text{ operation} \\
  \]
  
  \[
  \text{int sem_post(sem_t *s);} \\
  \text{-- performs the } V(s) \text{ operation} \\
  \]

Example using semaphores

```c
#define LOOPS 10000000
static int count = 0;
static sem_t mutex;

void *counter(void *args) {
    int i;
    for (i = 0; i < LOOPS; i++) {
        sem_wait(&mutex);
        count++;
        sem_post(&mutex);
    }
    printf("Executed %d times\n", i);
    return NULL;
}

int main() {
    pthread_t tids[2];
    sem_init(&mutex, 0, 1);
    pthread_create(&tids[0], NULL, counter, NULL);
    pthread_create(&tids[1], NULL, counter, NULL);
    pthread_join(tids[0], NULL);
    pthread_join(tids[1], NULL);
    printf("Count: \d\n", count);
    return 0;
}
```