# CMSC 216
# Introduction to Computer Systems
# Lecture 22
# Concurrent Programming

Jan Plane & Alan Sussman
{jplane, als}@cs.umd.edu

## Administrivia

- Project 4 secret tests on Grace
- Project 5 posted, public tests on Grace
- Exam #2 Thursday
  - Dynamic memory allocation (lecture 11) through today (lecture 21)
  - 02 and 03 (Jan's lectures) will be in CSIC 1115
  - Practice exam posted, with answers
- Read Chapter 12 on Concurrent Programming



Chapter 12, Bryant and O'Hallaron

## CONCURRENT PROGRAMMING



## Concurrency

- We have seen concurrency in our programs before, with processes, as well as ways for processes to communicate with each other (signals, pipes)
- Since processes have separate virtual address spaces, working with common data requires considerable communication and synchronization overhead
- Concurrency can provide speedups, however, if implemented properly on a multicore system

# Threads

- The use of threads allows all the threads to access common memory inside a process
- Each thread has a separate thread context, including:
  - thread ID
  - stack
  - stack pointer
  - program counter
  - registers
  - condition codes
- Threads share:
  - heap memory
  - global/static memory
  - open files
  - shared libraries
  - virtual address space



# Thread model

- Threads are scheduled similarly to processes; a thread that performs an I/O operation may be scheduled out of the processor while another thread is scheduled in
- No parent-child relationship; the main (first) thread creates a peer thread, which are both then in the thread pool
- Context switches between threads are much less expensive than those between processes



# Posix threads

- The standard interface for C threads
- Example of a Pthreads program:

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

struct point {
  int x, y;
};
static void *print_point(void *pointp);

int main() {
  pthread_t tid;
  struct point pt = {3, 5};
  pthread_create(&tid, NULL, print_point, &pt);
  pthread_join(tid, NULL);
  return 0;
}

static void *print_point(void *pointp) {
  struct point arg = * (struct point *) pointp;
  printf("Point: (%d, %d)\n", arg.x, arg.y);
  return NULL;
}
```



# Compiling Pthreads code

- To compile the example program, you need to tell **`gcc`** to use the pthread library when linking your executable
- To do this, use the **`-l`** switch to **`gcc`**:

  ```
  gcc -o threadex threadex.c -lpthread
  ```

  - This same switch is needed to use many other libraries (math functions, for example)
  - If you forget this, your program will not compile, and you will get an error like this:

    ```
    /tmp/cc3VuzAe.o(.text+0x3a): In function `main':
    : undefined reference to `pthread_create'
    ```

  - You can add this flag to the **`LDFLAGS`** variable in your **`Makefile`** to have it work correctly
- All of the functions we'll talk about that begin with "**`pthread_`**" require including **`<pthread.h>`**

## Creating a thread

- Use:

```
int pthread_create(pthread_t *tid,
                   pthread_attr_t *attr,
                   void *(*func)(void *),
                   void *arg);
```

  - **tid** is a pointer to an allocated (dynamic or otherwise) **pthread_t** that will have the thread ID of the new thread placed in it
  - **attr** is a pointer that can be used to change the attributes of the new thread (but we'll usually just use **NULL**)
  - **func** is a function pointer to the new thread's routine
  - **arg** is a pointer that will be passed to the new thread's routine when the thread is created; this is the way you pass arguments to a thread
  - returns 0 on success, nonzero on error



## Obtaining your own thread ID

- **pthread_t pthread_self(void);**
  - returns thread ID of caller
  - similar to **getpid()**
- There is no parent-child relationship between threads, so there is no counterpart to **getppid()**



## Thread termination

- Threads can be terminated in one of four ways:
  - Implicit termination: the thread routine returns; usually what we'll use
  - Explicit termination: the thread calls **pthread_exit()**
  - Process exit: any thread calls **exit()**, which terminates the process and all associated threads; maybe not what you really want
  - Thread cancellation: another thread calls **pthread_cancel()** to terminate a specific thread



## Thread termination functions

- **void pthread_exit(void *val);**
  - terminates current thread, with a thread return value equal to the pointer **val**
  - the return value can be obtained by the reaping thread (discussed soon)
- **int pthread_cancel(pthread_t tid);**
  - requests termination of thread with thread ID **tid**
  - returns 0 on success, nonzero on error

# Thread reaping

- When a thread has terminated, information about it, including the thread return value, is still kept in memory until reaped by another thread
- Threads are reaped by **`pthread_join()`**:

  **`int pthread_join(pthread_t tid, void **val);`**

  – reaps thread with thread ID **`tid`**

  – blocks until thread **`tid`** terminates

  – frees memory resources held by thread **`tid`**

  – returns 0 on success, nonzero on error

  – on success, **`*val`** is the return value of the terminated thread



# A simple example

```
#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;
}
```



# Questions

```
#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;
}
```

- Why do we create an array called **`ids`**? Why not just pass in **`&i`** as the argument?
- *The value of **`i`** changes; if it changes before the new thread can access the memory, it's a problem.*
- What will the output be? Do we know what the first line to be printed out will be? How about the last?
- *Most of the output will be interleaved. The first line will probably, but not definitely, be the creation of thread 1. The last line will always be the reaping of thread 2.*



# 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

```
#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

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

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

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.

**`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?

```
#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!