CMSC 313
Introduction to Computer Systems
Lecture 20
Function Pointers and Threads

Alan Sussman
als@cs.umd.edu

Administrivia

- Project 3 secret tests posted on Grace, and results and grades also posted on submit and grades server
- Project 5 public tests on submit server
  - project due Nov. 19 at 6PM
- Exam #2 on Monday
  - Punt rule
  - Dynamic memory allocation (lecture 10) through data representation (lecture 19)
- Read Section 13.3 of Reek and Chapter 13 of Bryant and O'Hallaron

Section 13.3, Reek

FUNCTION POINTERS

Function Pointers

- Each function is located somewhere in memory; this means we can create a pointer to it
- Declared like this:
  - `void (*fp)(int);`
    - **fp** is a pointer to a function that returns **void** and has a single parameter (which is an **int**)
  - `int *(*fp2)(char *, int);`
    - **fp2** is a pointer to a function that returns a pointer to an **int**, and has 2 parameters (a pointer to **char**, and an **int**)

Using function pointers

```
void print_decimal(unsigned int i) {
  printf("%u\n", i);
}
void print_hex(unsigned int i) {
  printf("%x\n", i);
}
void print_octal(unsigned int i) {
  printf("%o\n", i);
}

...

void (*fp)(unsigned int);
fp = print_hex;
fp(16);  /* prints "10" */
fp = &print_octal;
fp(16);  /* prints "20" */
fp = print_decimal;
(*fp)(16);  /* prints "16" */
```


Using `typedef` with function pointers

- To make things a bit more clear, we can use **`typedef`** to create a specific function pointer type
- Example:

```
typedef char *(*Str_func)(char *);

char *strdup(char *str) { ... }
...
Str_func sf = strdup;
char *copy = sf(str);
```


Understanding complex declarations

- Even people who've programmed in C for a long while may have trouble deciphering this declaration:

```
int *(*f[8])(char *);
```

- The program **`cdecl`** can be of use here:

```
$ cdecl
Type `help' or `?' for help
cdecl> explain int *(*f[8])(char *);
declare f as array 8 of pointer to function
(pointer to char) returning pointer to int
```

- In other words, **`f`** is an array containing 8 function pointers, each of which can point to a function that takes a **`char *`** as an argument and returns an **`int *`**

Chapter 13, 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
```