# CMSC 216
# Introduction to Computer Systems
# Lecture 23
# Time, Function Pointers & Concurrent Programming

Larry Herman & Alan Sussman

## Notes

- Project 6 due Monday, 8PM
- Project 7 out Monday, due 5/9
- Exam 2 returned yesterday
  - Mean: 76 Median: 77 25%: 66 75%: 88
- Read Bryant and O'Hallaron Section 13.3 on function pointers, and start on Chapter 12 on Concurrent Programming



Section 16.3, Reek

## TIME MEASUREMENT



## Date and time functions

- There are several functions in **<time.h>** for working with times.
  - most use a type **time_t** that contains an encoded representation of a time
  - several use the following **tm** structure defined in **<time.h>** that has fields that can be extracted from a **time_t** variable:

```
struct tm {
  int tm_sec;   /* seconds */
  int tm_min;   /* minutes */
  int tm_hour;  /* hours */
  int tm_mday;  /* day of the month */
  int tm_mon;   /* month */
  int tm_year;  /* year */
  int tm_wday;  /* day of the week */
  int tm_yday;  /* day in the year */
  int tm_isdst; /* daylight saving time */
};
```

# Date and time functions

**`clock_t clock(void);`**
- returns the process time since the start of program execution
- to convert to time, divide by **`CLOCKS_PER_SEC`** (also in **`time.h`**)

**`time_t time(time_t *val);`**
- fills **`val`** with the current time (in an implementation-dependent format)

**`char *ctime(time_t *val);`**
- returns a character representation of the passed time
- Example: **`Sun Oct 28 09:02:48 2007\n\0`**

**`double difftime(time_t time1, time_t time2);`**
- returns the number of seconds between **`time1`** and **`time2`**

**`struct tm *gmtime(time_t val);`**

**`struct tm *localtime(time_t val);`**
- converts a time to UTC or local time, in the form of a **`struct tm`**



# Adding timing calls to your program

- Wall time

  **`int gettimeofday(struct timeval *tv, struct timezone *tz);`**
  - **`tv`** is a structure of time **`tv_sec`** and **`tv_usec`** ($10^{-6}$ seconds)
  - **`tz`** is no longer used (just pass **`NULL`**)
- Process time

  **`int getrusage(int who, struct rusage *usage);`**
  - **`who`** is **`RUSAGE_SELF`** or **`RUSAGE_CHILDREN`**
    - **`RUSAGE_CHILDREN`** is all terminated children
  - **`rusage`** contains fields for

    **`struct timeval ru_utime; /* user time used */`**

    **`struct timeval ru_stime; /* system time used */`**
    - and fields for various other OS statistics



# Adding timing calls, cont.

- Include **`<sys/time.h>`** to use **`gettimeofday()`**
- Include **`<sys/time.h>`**, **`<sys/resource.h>`**, and **`<unistd.h>`** to use **`getrusage()`**



# Example measuring time

```
#include <sys/time.h>
#include <sys/resource.h>
#include <unistd.h>

int main()  {
  struct rusage start_ru, end_ru;
  struct timeval start_wall, end_wall;

  gettimeofday(&start_wall, NULL);
  getrusage(RUSAGE_SELF, &start_ru);

  /* code to time */

  gettimeofday(&end_wall, NULL);
  getrusage(RUSAGE_SELF, &end_ru);

  /* compute difference */

  return 0;
}
```

# Calculating the difference of 2 times

- Not trivial, as two fields are involved in each struct timeval, but not too complicated
- Example (calculating **`end - start`**):

```
struct timeval tv_delta(struct timeval start,
                        struct timeval end)
{
  struct timeval delta = end;
  delta.tv_sec -= start.tv_sec;
  delta.tv_usec -= start.tv_usec;
  if (delta.tv_usec < 0) {
    delta.tv_usec += 1000000;
    delta.tv_sec--;
  }
  return delta;
}
```



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 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()`**