CMSC 216
Introduction to Computer Systems
Lecture 21
Time Measurement and
Function Pointers

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

Administrivia

- Project 3 secret tests posted on Grace
- Project 4 secret tests on Grace soon
- Project 5 posted, public tests on Grace today
- Exam #2 next Thursday
  - Dynamic memory allocation (lecture 11) through today (lecture 21)
  - 02 and 03 (Jan's lectures) will be in CSIC 1115
  - Practice exam posted today
- Read Reek, Section 16.3 on time measurement, Section 13.3 on function pointers

Chapter 10, Bryant and O'Hallaron

SYSTEM-LEVEL I/O

Mixing buffered/unbuffered reads

buf.c:

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

#define BUFFER_SZ 100
static char buffer[BUFFER_SZ];

int main() {
  char line_s[11];
  char line_u[11] = {0};
  setbuffer(stdin, buffer, BUFFER_SZ);
  fgets(line_s, 11, stdin);
  read(STDIN_FILENO, line_u, 10);
  printf("L1: %s\n", line_s);
  printf("L2: %s\n", line_u);
  fgets(line_s, 11, stdin);
  read(STDIN_FILENO, line_u, 10);
  printf("L3: %s\n", line_s);
  printf("L4: %s\n", line_u);
  return 0;
}
```

data.txt:

```
aaaaaaaaaabbbbbbbbbbcccccccccc
ddddddddddeeeeeeeeeeffffffffff
gggggggggghhhhhhhhhhiiiiiiiiii
jjjjjjjjjjkkkkkkkkkkllllllllll
mmmmmmmmmmnnnnnnnnnnoooooooooo
```

Execution:

```
$ ./buf < data.txt
L1: aaaaaaaaaa
L2: kkkkkkkkkk
L3: bbbbbbbbbb
L4: llllllllll
```

Section 16.3, Reek

TIME MEASUREMENT

Time

- On a computer there are many ways to measure time
- Conceptual difference:
  - *wall time* is always running
  - *process time* is the time your program was running
    - process time doesn't count:
      - the time when your program was stopped for others
      - the time when your program stopped to wait for I/O
    - is composed of:
      - user time: when the OS is running your code
      - system time: when the OS is running system code (handling system calls)
- Difference in how to measure
  - interval time
  - clock cycles
- What time to use depends on what you are measuring

Timing a program

- To time an entire program in the shell:
  - **`time`** *program-name program-arguments*
  - runs the program and prints its execution time in the form (tcsh)

    `2.230u 0.260s 0:06.52 38.1% 0+0k 0+0io 80pf+0w`
    - the first two numbers are user and system time
    - the third number is wall time
    - the fourth is percentage of wall time: (user+system)/wall
    - the remainder are paging and I/O statistics
- This provides some idea about timing
  - but it's hard to know what to do about it - what functions are taking most of the time?

Representing time-of-day

- How to deal with
  - time zones
  - daylight saving time rules
  - computers moving between time zones
  - leap seconds?
- Answer:
  - UNIX keeps time internally as seconds and fractions of a second
    - 2^{32} bit values work nicely for 1ns accuracy for > 100 years
  - use a reference starting point
    - called the epoch - midnight Jan. 1, 1970
  - keep all time in UTC form until printed
    - no time zones or daylight saving time to deal with

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