CSE 451: Operating Systems

Section 1
Intro, C programming, project 0
Introduction

- My name is Elliott and I am a fifth-year masters student in computer science.
- I graduated last year with a degree in computer science and math.
- I accepted an offer from Google to start as a software engineer in August with the Dremel team.
- I’m very passionate about C++ programming and distributed systems.
  - Favorite classes: graphics, OS, distributed systems.
  - I have been a TA for CSE 451, CSE 333, and CSE 351.
- My office hours are Wednesday 10:30-11:20, Thursday 11:30 to 12:20, or by appointment/when I’m in 002.
- Contact: discussion board or by email (snowden@cs)
Why are you here?

- You had an opening in your schedule and needed a 400-level elective
- You had heard how awesome Ed is as a lecturer and wanted to experience him first-hand
- You want to learn about the part of computer science that facilitates all others
Far-reaching implications

- Operating systems techniques apply to all other areas of computer science
- Data structures
- Caching
- Concurrency
- Virtualization

- Operating systems *support* all other areas of computer science
Course Tools

* Use whatever works best for you: the CSE home VM, attu, the instructional Linux machines, or your own Linux installation
* The second project requires the use of VMWare Player/VirtualBox, which are available for all major operating systems and are also present on Windows lab machines
* The forkbomb server (more on this next week) can be used for kernel compilation
Course Tools

- We’ll be using the GNU C Compiler (gcc) for compiling C code in this course, which is available on pretty much every platform except Windows (unless through Cygwin).
- For an editor, use whatever makes you comfortable; Emacs, Vim, gedit, and Eclipse are good choices.
Discussion board

-The discussion board is an invaluable tool; use it!
-Jim (my TA partner in crime) and I both receive email alerts whenever there is a new post, so prefer the discussion board to email since then the rest of the class can benefit from your questions as well
Collaboration

- If you talk or collaborate with anybody, or access any websites for help, *name them* when you submit your project.
- See the [course policy](#) for more details.
- Okay: discussing problems and techniques to solve them with other students.
- Not okay: looking at/copying other students’ code.
Most modern operating systems are still written in C

Why not Java?
- Interpreted Java code runs in a virtual machine, so what does the VM run on?

C is precise in terms of
- Instructions (semantics are clear)
- Timing (can usually estimate number of cycles to execute code)
- Memory (allocations/deallocations are explicit)
C language features

- Pointers
- Pass-by-value vs. pass-by-reference
- Structures
- Typedefs (aliasing)
- Explicit memory management
Pointers

int x = 5;
int y = 6;

int* px = &x;  // declare a pointer to x
    // with value as the
    // address of x

*px = y;  // change value of x to y
    // (x == 6)

px = &y;  // change px to point to
    // y’s memory location

// For more review, see the CSE 333 lecture
// and section slides from autumn 2012
**Function pointers**

```c
int some_fn(int x, char c) { ... }  
    // declare and define a function
int (*pt_fn)(int, char) = NULL;
    // declare a pointer to a function
    // that takes an int and a char as
    // arguments and returns an int
pt_fn = some_fn;
    // assign pointer to some_fn()'s
    // location in memory
int a = pt_fn(7, 'p');
    // set a to the value returned by
    // some_fn(7, 'p')
```
extern void (*signal(int, void(*)(int)))(int);

What is going on here?

signal() is "a function that takes two arguments, an integer and a pointer to a function that takes an integer as an argument and returns nothing, and it returns a pointer to a function that takes an integer as an argument and returns nothing."*

*See this StackOverflow post
Case study: signals

* We can make this a lot clearer using a typedef:

```c
// Declare a signal handler prototype
typedef void (*SigHandler)(int signum);
// signal could then be declared as 
extern SigHandler signal(
    int signum, SigHandler handler);
```

* Much improved, right?
Arrays and pointer arithmetic

* Array variables can often be treated like pointers, and vice-versa:

```c
int foo[2];  // foo acts like a pointer to
             // the beginning of the array
*(foo + 1) = 5;  // the second int in the
                 // array is set to 5
```

* Don’t use pointer arithmetic unless you have a good reason to do so
Passing by value vs. reference

```c
int doSomething(int x) {
    return x + 1;
}

void doSomethingElse(int* x) {
    *x += 1;
}

void foo(void) {
    int x = 5;
    int y = doSomething(x); // x==5, y==6
    doSomethingElse(&x);    // x==6, y==6
}
```
bool Initialize(int arg1, int arg2, ErrorCode* error_code) {
// If initialization fails, set an error
// code and return false to indicate
// failure.
if (!...) {
    *error_code = ...;
    return false;
}
// ... Do some other initialization work
return true;
}
// Define a struct referred to as
// "struct ExampleStruct"
struct ExampleStruct {
  int x;
  int y;
};  // Don’t forget the trailing ‘;’!

// Declare a struct on the stack
struct ExampleStruct s;

// Set the two fields of the struct
s.x = 1;
s.y = 2;
typedef struct ExampleStruct ExampleStruct;
    // Creates an alias “ExampleStruct” for “struct ExampleStruct”

ExampleStruct* new_es = (ExampleStruct*) malloc(sizeof(ExampleStruct));
    // Allocates an ExampleStruct struct on the heap; new_es points to it

new_es->x = 2;
    // “->” operator dereferences the pointer and accesses the field x;
    // equivalent to (*new_es).x = 2;
Explicit memory management

Allocate memory on the heap:

```c
void* malloc(size_t size);
```

* Note: may fail!
  * But not necessarily when you would expect...

* Use `sizeof()` operator to get size

Free memory on the heap:

```c
void free(void* ptr);
```

* Pointer argument comes from previous `malloc()` call
Common C pitfalls (1)

What’s wrong and how can it be fixed?

```c
char* city_name(float lat, float long) {
    char name[100];
    ...
    return name;
}
```
Common C pitfalls (1)

**Problem:** returning pointer to local (stack) memory

**Solution:** allocate on heap

```c
char* city_name(float lat, float long) {
    // Preferrably allocate a string of just the right size
    char* name = (char*) malloc(100);
    ...
    return name;
}
```
Common C pitfalls (2)

What’s wrong and how can it be fixed?

```c
char* buf = (char*) malloc(32);
strcpy(buf, argv[1]);
```
Common C pitfalls (2)

Problem: potential buffer overflow

Solution:

```
static const int kBBufferSize = 32;

char* buf = (char*) malloc(kBBufferSize);
strncpy(buf, argv[1], kBBufferSize);
```

Why are buffer overflow bugs dangerous?
Common C pitfalls (3)

What’s wrong and how can it be fixed?

```c
char* buf = (char*) malloc(32);
strncpy(buf, "hello", 32);
printf("%s\n", buf);

buf = (char*) malloc(64);
strncpy(buf, "bye", 64);
printf("%s\n", buf);

free(buf);
```
Problem: memory leak

Solution:

```c
char* buf = (char*) malloc(32);
strncpy(buf, "hello", 32);
printf("%s\n", buf);
free(buf);

buf = (char*) malloc(64);
...```

4/4/13
Common C pitfalls (4)

* What’s wrong (besides ugliness) and how can it be fixed?

```c
char foo[2];
foo[0] = 'H';
foo[1] = 'i';
printf("%s\n", foo);
```
Problem: string is not NULL-terminated

Solution:
char foo[3];
foo[0] = 'H';
foo[1] = 'i';
foo[2] = '\0';
printf("%s\n", &foo);

Easier way: char* foo = "Hi";
Another bug in the previous examples?
Not checking return value of system calls / library calls!

```c
char* buf = (char*) malloc(BUF_SIZE);
if (!buf) {
    fprintf(stderr, "error!\n");
    exit(1);
}
strncpy(buf, argv[1], BUF_SIZE);
... 
```
Project 0

* Description is on course web page
* Due Wednesday April 10, 11:59pm
* Work individually
  * Remaining projects are in groups of 2. When you have found a partner, one of you should fill out this Catalyst survey by Monday at 11:59pm: https://catalyst.uw.edu/webq/survey/elliottb/198212
Project 0 goals

- Get re-acquainted with C programming
- Practice working in C / Linux development environment
- Create data structures for use in later projects
Valgrind

- Helps find all sorts of memory problems
  - Lost pointers (memory leaks), invalid references, double frees

- Simple to run:
  - valgrind ./myprogram
  - Look for “definitely lost,” “indirectly lost” and “possibly lost” in the LEAK SUMMARY

- Manual:
Before you can check the queue for memory leaks, you should probably add a queue destroy function:

```c
void queue_destroy(queue* q) {
    queue_link* cur;
    queue_link* next;
    if (q != NULL) {
        cur = q->head;
        while (cur) {
            next = cur->next;
            free(cur);
            cur = next;
        }
        free(q);
    }
}
```
Project 0 testing

* The test files in the skeleton code are incomplete
  * Make sure to test *every* function in the interface (the `.h` file)
  * Make sure to test corner cases

* Suggestion: write your test cases *first*
Project 0 tips

- Part 1: queue
  - First step: improve the test file
  - Then, use valgrind and gdb to find the bugs

- Part 2: hash table
  - Write a thorough test file
  - Perform memory management carefully

- You’ll lose points for:
  - Leaking memory
  - Not following submission instructions

- Use the discussion board for questions about the code