CSE 451: Operating Systems

Section 1

Why are you here?

Why are you here?

- Because you want to work for Microsoft and hack on the Windows kernel?

Why are you here?

- ~~Because you want to work for Microsoft and hack on the Windows kernel?~~
- Because it fulfills a requirement and fits your schedule?

Who cares about operating systems?

- Operating systems techniques apply to all other areas of computer science
  - Data structures; caching; concurrency; virtualization…

Who cares about operating systems?

- 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

Who are we?

- Peter Hornyack
- Abdul Salama

Who are we?

- Peter Hornyack
- Abdul Salama
- What do we know?

Who are we?

- Peter Hornyack
- Abdul Salama

- What do we know?
- Why are we here?

What is this section for?

- Projects
- Questions!
- Extensions beyond lecture / textbook material

Office Hours

Outline

- ~~Introduction~~
- C vs. Java
- C language “features”
- C pitfalls
- Project 0

Motivation: Why C?

- Why not write OS in Java?

Motivation: Why C?

- Why not write OS in Java?
  - Interpreted Java code runs in VM; what does VM run on?

Motivation: Why C?

- Why not write OS in Java?
  - Interpreted Java code runs in VM; what does VM run on?
- Precision:
  - Instructions
  - Timing
  - Memory

C vs. Java: Compilation

Java	C
• Packages `"import java.xyz"`	• Header files `"#include xyz.h"`
• .class files	• .o files

• jar program
  • .jar files
	• linker program
	  • Executable files
	  • libc

C vs. Java: Constructs

Java	C
∗ Classes	∗ Structures
∗ Public or private members	∗ All members "public"
∗ Methods	∗ Functions
∗ Instantiated with class, or may be static	∗ Implicitly "static"
∗ References	∗ Pointers

C Language Features

∗Pointers

∗Pass-by-value vs. pass-by-reference

∗Structures

∗Typedefs

∗Explicit memory management

Pointers

```
int a = 5;
int b = 6;
int *pa = &a;  // declares a pointer to a
               // with value as the
               // address of a
*pa = b;       // changes value of a to b
               // (a == 6)
pa = &b;       // changes pa to point to
               // b's memory location (on
               // stack)
```


Function Pointers

```
int some_fn(int x, char c) { ... }
     // declares and defines a function
int (*pt_fn)(int, char) = NULL;
     // declares a pointer to a function
     // that takes an int and a char as
     // arguments and returns an int
pt_fn = &some_fn;
     // assigns pointer to some_fn()'s
     // location in memory
int a = (*pt_fn)(7, 'p');
     // sets a to the value returned by
     // some_fn(7, 'p')
```

Pointer Arithmetic

- Array variables are really just pointers:

```
int foo[2];     // foo is 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

Pass-By-Value vs. Pass-By-Reference

```
int doSomething(int x) {
   return x+1;
}
void doSomethingElse(int *x) {
   *x += 1;
}
void foo() {
   int x = 5;
   int y = doSomething(x);  // x==5, y==6
   doSomethingElse(&x);     // x==6, y==6
}
```


Structures

```
struct foo_s {      // defines a type that
   int x;           // is referred to as a
   int y;           // "struct foo_s".
};                  // don't forget this ;

struct foo_s foo;   // declares a struct
                    // on the stack

foo.x = 1;          // sets the x field
                    // of the struct to 1
```


Typedefs

```
typedef struct foo_s *foo_t;
     // creates an alias "foo_t" for
     // pointer to foo_s struct

foo_t new_foo =
   (foo_t)malloc(sizeof(struct foo_s));
     // allocate a foo_s struct on the
     // heap; new_foo points to it

new_foo->x = 2;
     // "->" operator dereferences the
     // pointer and accesses the field x
```

Explicit Memory Management

- Allocate memory on the heap:

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

  - Note: may fail!
  - Use `sizeof()` operator to get size

- Free memory on the heap:

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

  - Pointer argument comes from previous `malloc()` call

Common C Pitfalls (1)

- What's wrong and how to fix it?

```
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

```
char* city_name(float lat, float long) {
   char* name =
      (char*)malloc(100*sizeof(char));
   ...
   return name;
}
```


Common C Pitfalls (2)

- What's wrong and how to fix it?

```
char* buf = (char*)malloc(32);
strcpy(buf, argv[1]);
```

Common C Pitfalls (2)

- Problem: potential buffer overflow
- Solution:

```
int buf_size = 32;
char* buf =
   (char*)malloc(buf_size*sizeof(char));
strncpy(buf, argv[1], buf_size);
```

- Why are buffer overflow bugs important?

Common C Pitfalls (3)

- What's wrong and how to fix it?

```
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);
```


Common C Pitfalls (3)

- Problem: memory leak
- Solution:

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

buf = (char*)malloc(64);
…
```


Common C Pitfalls (4)

- What's wrong (besides ugliness) and how to fix it?

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

Common C Pitfalls (4)

- 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";`

Project 0

- Description is on course web page now
- Due Friday October 8, 11:59pm
- Work individually
  - Remaining projects in groups of 3

Project 0 Goals

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

Project 0 Tips

- Try these tools:
  - man pages
  - valgrind
  - gdb
- Write your test cases **first**

Project 0 Tips

- Part 1: queue
  - To find bugs, try valgrind, then gdb
- Part 2: hash table
  - Perform memory management carefully
    - Check using valgrind