CSE 451: Operating Systems

Section 1

Intro, C programming, project 0

Slides adapted by slides created by jasnyder

Far-reaching implications

- Concepts and techniques learned in lecture / through projects apply to all other areas of computer science
  - Data structures
  - Caching
  - Concurrency
  - Virtualization
- OSes *support* all other areas of computer science

Course tools

- Assn 0: Any computer with C development tools (002, attu, your *nix box)
- Assn 1: Use the course VM inside an emulator (VMware, Qemu etc.) on your computer or a lab computer
- Can compile on forkbomb.cs.washington.edu (faster)

Course tools

* We'll be using the GNU C Compiler (gcc) for compiling C code in this course, which is available on every platform except Windows (Cygwin lovers proceed at your own risk)
* For an editor, use whatever you are most comfortable with; emacs, vim, gedit, and Eclipse are good choices (ed and butterflies also options)

Discussion board

* The discussion board is an invaluable tool; use it!
* Andrew (my TA partner in crime) and I both receive email alerts whenever there is a new post. Response time should be by the end of the day, typically faster than that.
* For anything non-personal use the discussion board.

Collaboration

* If you talk or collaborate with anybody, or access any websites for help, *name them* in your project submission
* See the course policy for more restrictions
* Okay: discussing problems and techniques to solve them with other students
* Not okay: looking at/copying other students' code. Googling solutions. Using code from Wikipedia.
* We will pass your code through plagiarism detection software (MOSS, Deckard, etc.)

C programming

- Most modern operating systems are still written in C
- Why not Java?
  - Interpreted Java code runs in a virtual machine, so what language is the VM built in?
- C is precise in terms of
  - Instructions (semantics are clear)
  - Timing (can usually estimate number of cycles needed to execute code)
  - Memory (allocations/de-allocations are explicit)

C language features

- Pointers
- Pass-by-value vs. pass-by-reference
- Structs
- Typedefs (aliasing)
- Malloc/free

Pointers

```
int iX = 5;
int iY = 6;

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

*piX = iY;      // change value of iX to iY
                // (iX == 6)

piX = &iY;      // change piX to point to
                // iY's memory location
// For more review, see the CSE 333 lecture
// and section slides
```

Function pointers

```
int functionate(int iHerp, char cDerp) { ... }
            // declare and define a function
int (*pfFoo)(int, char) = NULL;
            // declare a pointer to a function
            // that takes an int and a char as
            // arguments and returns an int
pfFoo = functionate;
            // assign pointer to functionate()'s
            // location in memory
iX = pfFoo(7, 'p');
            // set iX to the value returned by
            // functionate(7, 'p')
```

Case study: signal()

```
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 (`signal()`) returns a pointer to a function that takes an integer as an argument and returns nothing." (from StackOverflow)

Case study: signal

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

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

Arrays and pointer arithmetic

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

```
int aiFoo[2];     // foo acts like a pointer to
                  // the beginning of the array
*(aiFoo + 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

```
int doSomething(int iFoo) {
   return iFoo + 1;
}

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

void example(void) {
   int iX = 5;
   int iY = doSomething(iX);   // iX==5, iY==6
   doSomethingElse(&iX);       // iX==6, iY==6
}
```

Returning addl. information

```
int initialize(int iArg1, int iArg2,
    int* piErrorCode) {
  // If initialization fails, set an error
  // code and return false to indicate
  // failure.
  if (...) {
    *piErrorCode = ...;
    return EXIT_FAILURE;
  }
  // ... Do some other initialization work
  return EXIT_SUCCESS;
}
```

Structs

```
// Define a struct referred to as
// "struct s2DPoint"
struct s2DPoint {
  int iX;
  int iY;
};  // Don’t forget the trailing ‘;’!

// Declare a struct on the stack
struct s2DPoint foo;

// Set the two fields of the struct
foo.iX = 1;
foo.iY = 2;
```

Typedefs

```
typedef struct s2DPoint 2DPoint;
     // Creates an alias “2DPoint” for
     // “struct s2DPoint”

2DPoint* poBar =
    (2DPoint*) malloc(
       sizeof(2DPoint));
     // Allocates space for a 2DPoint struct
     // on the heap; poBar points to it

poBar->iX = 2;
     // “->” operator dereferences the
     // pointer and accesses the field iX;
     // equivalent to (*poBar).iX = 2;
```

Memory management

* Allocate memory on the heap:

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

  * Note: malloc may fail!
    * But not necessarily when you would expect…
  * Use `sizeof()` operator to get the size of a type/struct

* 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 can it be fixed?

```
char* city_name(float fLat, float fLong) {
  char sName[100];
  ...
  return sName;
}
```

Common C pitfalls (1)

- Problem: returning pointer to local (stack) memory (also: using floats)
- Solution: allocate on heap

```
char* city_name(double fLat, double fLong) {
  // Preferably allocate a string of
  // just the right size
  char* sName =
      (char*) malloc(100*sizeof(char));
  ...
  return sName;
}
```

Common C pitfalls (2)

* What's wrong and how can it be fixed?

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

Common C pitfalls (2)

* Problem: potential buffer overflow
* Solution:

```
static const int BUFFER_SIZE = 32;

char* sBuf = (char*) malloc(BUFFER_SIZE);
strncpy(sBuf, argv[1], BUFFER_SIZE);
```

* Why are buffer overflow bugs dangerous?

Common C pitfalls (3)

- What's wrong and how can it be fixed?

```
char* sBuf = (char*) malloc(BUFFER_SIZE);
Strncpy(sBuf, sHello, BUFFER_SIZE);
printf("%s\n", sBuf);

sBuf = (char*) malloc(2*BUFFER_SIZE);
strncpy(sBuf, sLongHello, 2*BUFFER_SIZE);
printf("%s\n", sBuf);

free(sBuf);
```

Common C pitfalls (3)

- Problem: memory leak
- Solution:

```
char* sBuf = (char*) malloc(BUFFER_SIZE);
strncpy(sBuf, sHello, BUFFER_SIZE);
printf("%s\n", sBuf);
free(sBuf);

buf = (char*) malloc(2*BUFFER_SIZE);
...
```

Common C pitfalls (4)

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

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

Common C pitfalls (4)

- Problem: string is not NULL-terminated
- Solution:

```
char sFoo[3];
sFoo[0] = 'H';
sFoo[1] = 'i';
sFoo[2] = '\0';
printf("%s\n", sFoo);
```

- Easier way: `char* sFoo = "Hi”;`

Common C pitfalls (5)

- Another bug in the previous examples?
  - Not checking the return value of system calls / library calls!

```
char* sBuf = (char*) malloc(BUFFER_SIZE);
if (sBuf == 0) {
    fprintf(stderr, "error!\n");
    return EXIT_FAILURE;
}
strncpy(sBuf, argv[1], BUFFER_SIZE);
...
```

Project 0

- Description is on course web page
- Due Friday January 17th, 11:59pm
- Work individually
  - Remaining projects are in groups of 2. When you have found a partner, one of you should fill out the survey on Catalyst (forthcoming by email)

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:
  * http://valgrind.org/docs/manual/manual.html

Project 0 memory leaks

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

```
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