CSE 451: Operating Systems

Section 1

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
* Operating systems *support* all other areas of computer science

facebook

Photos @ Facebook

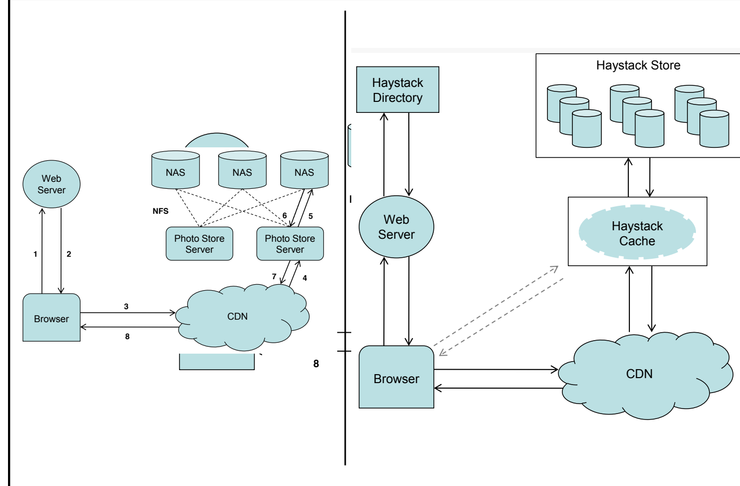
	April 2009	Current
Total	15 billion photos 60 billion images 1.5 petabytes	65 billion photos 260 billion images 20 petabytes
Upload Rate	220 million photos / week 25 terabytes	1 billion photos / week 60 terabytes
Serving Rate	550,000 images / sec	1 million images / sec

facebook

NFS based Design

- Typical website
  - Small working set
  - Infrequent access of old content
  - ~99% CDN hit rate
- Facebook
  - Large working set
  - Frequent access of old content
  - 80% CDN hit rate

facebook



Who are we?

* Pete Hornyack
* Elliott Brossard

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

What is this section for?

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

Office hours

- Monday 12:30 – 1:20 (Ed)
- Tuesday 12:30 – 1:20 (Elliott)
- Wednesday 10:30 – 11:20 (Pete)
- Wednesday 1:30 – 2:20 (Elliott)
- Friday 3:30 – 4:20 (Pete)

Use the discussion board!

- If you remember anything from this section, remember this!
  - The TAs get an e-mail notification every time somebody posts
  - Your classmates may have quicker / better answers than we do

Collaboration

- If you talk or collaborate with anybody, or access any websites for help, *name them* when you submit your project
- Review the CSE policy on collaboration:
  - http://www.cs.washington.edu/education/courses/cse451/12sp/overview.html#Policies

Outline

- ~~Introduction~~
- C language “features”
- C pitfalls
- Project 0

Why C?

* Why not write OS in Java?
  * Interpreted Java code runs in a virtual machine; what does the VM run on?
* *Precision*
  * Instructions
  * Timing
  * Memory
* What about Android?

C language features

* Pointers
* Pass-by-value vs. pass-by-reference
* Structures
* Typedefs
* Explicit memory management

Pointers

```
int x = 5;
int y = 6;

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

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

px = &y;       // changes px to point to
               // y's memory location
```


Pointer tutorials

* *Pointer Fun C*
  * http://www.youtube.com/watch?v=mnXkiAKbUPg
* UW ACM tutorial: *A C++ Crash Course*
  * Review slides 23-29:
    * http://flatline.cs.washington.edu/orgs/acm/tutorials/intro-c++/c++-tutorial-pt1.ppt
* More tutorials linked from project page
  * http://www.cs.washington.edu/education/courses/cse451/12sp/projects.html

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')
```


Arrays and pointer arithmetic

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

```
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

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
}
```


Pass-by-reference for returning values

```
bool queue_remove(
  queue* q, queue_element** elem_ptr)
{
  queue_element* elem = ...;
  ...
  *elem_ptr = elem;
  return true;
}
```

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;
     // Creates an alias "foo" for
     // "struct foo_s"

foo* new_foo =
   (foo*)malloc(sizeof(foo));
     // Allocates 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;
     // equivalent to (*new_foo).x = 2;
```


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

Common C pitfalls (1)

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

```
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);
  ...
  return name;
}
```


Common C pitfalls (2)

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

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


Common C pitfalls (2)

* Problem: potential buffer overflow
* Solution:

```
#define BUF_SIZE 32
char* buf = (char*)malloc(BUF_SIZE);
strncpy(buf, argv[1], BUF_SIZE);
```

* Why are buffer overflow bugs dangerous?

Common C pitfalls (3)

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

```
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 can it be fixed?

```
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";`

Common C pitfalls (5)

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

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


Project 0

- Description is on course web page now
- Due Wednesday April 4, 11:59pm
- Work individually
  - Remaining projects are in groups of 3: e-mail your groups to us by 11:00am on Monday

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: tools

- Editing
  - Choose your favorite: emacs, vi, Eclipse…
  - Refer to *man pages* for system and library calls
- Navigation
  - ctags
    - http://www.cs.washington.edu/education/courses/cse451/12sp/tutorials/tutorial_ctags.html
  - cscope

Project 0: tools

* Compiling
  * gcc and Makefiles
* Debugging
  * valgrind
  * gdb
    * http://www.cs.washington.edu/education/courses/cse451/12sp/projects.html

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 add a queue destroy function:

```
void queue_destroy(queue* q) {
    queue_link* cur;
    queue_link* next;
    if (q) {
        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

Remember:

Use the discussion board!