Getting Friendly with C

Eric Wu (ericwu@cs)

Before diving into C…

High level OS concepts

- Largely the same across modern operating systems
- What is an OS?

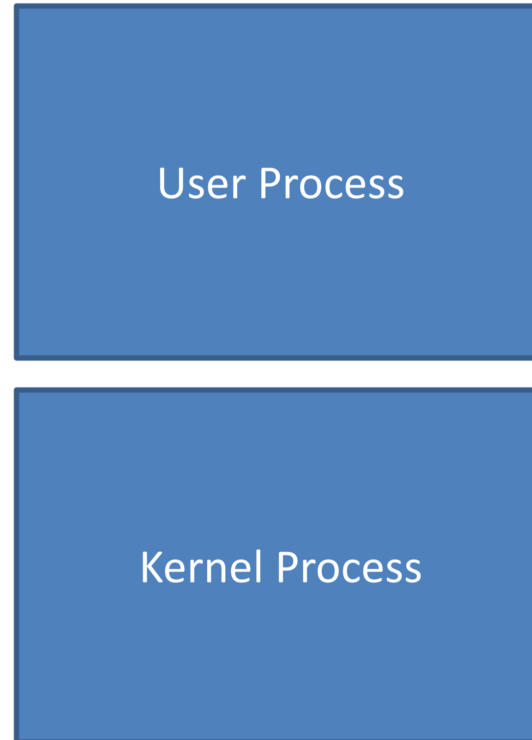
Before diving into C…

High level OS concepts

- Largely the same across modern operating systems
- What is an OS?
  - Software that manages hardware resources
  - Software that manages applications and enables them to use hardware resources

Structure of a Modern OS

- Two modes: user and kernel
  - Kernel mode executes OS tasks
  - User mode executes user (application) tasks
- User mode switches to kernel mode via a system call
  - System calls can only be executed in kernel mode



The System Call

- Can be called anywhere, but can only be executed in kernel space.
- Example system calls in Windows: CreateProcess(), CreateFile(), SetFileSecurity()
- Same above system calls in Linux: fork(), open(), chmod()

Getting Friendly with C

Differences with Java?

- Not object oriented
- Not type safe
- Explicit memory management
- Executables don’t require a VM

Getting Friendly with C

Java

```
import java.xyz;

class Point {
  public int x;
  public int y;

  public int foo(int a) {
    …
    Point p;
  }
}
```

C

```
#include “xyz.h”

struct Point {
  int x;
  int y;
};

int foo(int a) {
  …
  Point *p;
}
```

Pointers

```
int a = 5;
int b = 6;
int *pa = &a;     // value of pointer *pa is
                  // address of a

*pa = b;          // changes value of a to b
                  // (a == 6)

pa = &b;          // changes pointer *pa to
                  // point to address of b

// pointers are just another variable type!
```

struct and typedef

```
struct foo_s {     // defines a type that is referred
  int x;           // to as a “struct foo_s”
  int y;
};                 // don’t forget the semicolon

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

foo.x = 1;         // access the x field

// use typedef to create an alias
// allowing you to now use foo_t
// to declare variables instead.
typedef struct foo_s *foo_t;
foo_t foo_ptr;     // this is type (struct foo_s *)
```

Variable Scoping

- Dynamic (“heap”) memory

```
void foo() {
    // value of pointer *p exists until free()'d
    int *p = malloc(sizeof(int));
}
```

- Global memory

```
int g;
void foo() {
    // value of pointer *p always exists
    int *p = &g;
}
```

- Local (“stack”) memory

```
void foo() {
    // value of pointer *p exists until foo() returns
    int a;
    int *p = &a;
}
```

Functions and 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;
  // makes pt_fn point at some_fn()'s
  // location in memory
int a = (*pt_fn)(7, 'p');
  // calls some_fn and stores the result
  // in variable a
```

Common C Pitfalls (1)

- What's wrong and how to fix it?

```
char* get_city_name(double latitude, double longitude) {
  char city_name[100];
  …
  return city_name;
}
```

Common C Pitfalls (1)

- Problem: return pointer to statically allocated memory.
- Solution: allocate on the heap.

```
char* get_city_name(double latitude, double
longitude) {
  char* city_name = (char*)malloc(100);
  …
  return city_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: buffer overflow
- Solution: limit the size of the copied buffer

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

- 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: free() all variables that are allocated on the heap

```
char* buf = (char*)malloc(32);
strncpy(buf, "hello", 32);
printf("%s\n", buf);
free(buf);
buf = (char*)malloc(64);
…
```

- Are memory leaks important?
  - OS, web server, web browser, your projects?

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: NULL terminate the string!

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

- Or better:

```
char * foo = "Hi";
```

Double-quoted string literal syntax gets NULL-terminated automatically.

Java programmer gotchas (1)

```
{
  int i
  for(i = 0; i < 10; i++)
     …
```

NOT

```
{
  for(int i = 0; i < 10; i++)
     …
```

Java programmer gotchas (2)

- Uninitialized variables
  - catch with **`-Wall`** compiler option

```
#include <stdio.h>

int main(int argc, char* argv[])
{
  int i;
  factorial(i);
  return 0;
}
```

Java programmer gotchas (3)

- Error handling
  - No exceptions
  - Must look at return values

Conditional Compilation

```
#define CSE451

int main(int argc, char* argv)
{
  #ifdef CSE451
  printf("The best class ever!\n");
  #else
  printf("Some other random class...\n");
  #endif
  return 0;
}
```