Operating Systems (coe518) Lecture Notes

Tuesday, September 2, 2014

Announcements:

- Course management information distributed in class.
- Lecture notes distributed. In future, you are responsible for printing copies of the lecture notes. Draft versions will be available prior to each lecture; final versions will be available within 24 hours after the lecture.
- Labs start the week of September 8, 2014. (No labs this week.)

1 What is an Operating System?

For the purposes of this course, an Operating System is software that acts as an abstraction layer between the physical resources of the computer system and the application programs that run on the system.

The resources managed by an OS include:

- Memory
- CPU
- Processes
- Input/Output (I/O)
- Files and file system

The OS manages these resources so that they are used efficiently. For example, "busy-waiting loops" (aka "spin-locks") are rarely needed.

1.1 Views of an OS

1.1.1 End user (non-technical)

- Users (eg. Someone who writes documents with Microsoft *Word*) experience operating system differences at a "cosmetic" level.
- For example, if they change from Windows 7 to Mac OSX, they quickly adapt and learn that the functionality of the square red "X" button on the top right of a widow used in Windows 7 to close a window is accomplished with the red round button on the top left in Mac OSX.

1.1.2 Application programmer

- Non-GUI applications can often be written so that they can be compiled and run on different OSes with few changes.
- Example: a C programmer who restricts themselves to the standard C library.
- Applications (including GUI apps) written with an operating system independent language (such as Java) need no changes for different systems.
- GUI applications will generally be tightly integrated with the windowing system natively supported by the operating system.
- Application programmers should strive to isolate the OS-dependent code into small modules. To do this, the programmer needs to know the API's for different systems.

1.1.3 System programmer/administrator

- These professionals need the greatest knowledge of operating systems including:
  - How operating systems work internally
  - How to write device drivers.
  - How to tune and/or modify the kernel.

2 Operating Systems covered

- Microsoft Windows and (even!) MS-DOS
- UNIX and UNIX-based including:
  - Linux
  - Solaris
  - Mac OSX
  - Android (Google)
  - Chromium (Google)
  - BSD unix
  - Apple iOS4 (iPad, iPhone, iPod touch)
  - POSIX (IEEE standard. All unix Oses support it and Microsoft Windows is almost completely compliant.)

3 The hardware/software interface

- The OS abstracts away the detials of the underlying hardware from the programmer.
- Modern OSes also provide *security* but this requires additional hardware capabilities from the CPU. Specifically:
  - At lease 2 modes of operation (user and supervisor) enforced by the hadware. Only OS code is allowed to operate in supervisor mode.
  - Hardware memory management (paged and/or segmented *virutal memory*) that makes it possible for the OS to ensure that processes to not corrupt other processess.

4 Using the UNIX shell

Refer to the local guide (http://www.ee.ryerson.ca/guides/user/#SECTION00400)

5 Introduction to processes, I/O redirection, piping and the Shell

Condsider the following simple programs:

Convert to uppercase:

```
/*
 * File:   main.c
 * Author: Ken Clowes
 */

#include <stdio.h>
#include <stdlib.h>

/*
 *Description:  Reads characters from stdin and writes each character
 * as uppercase to stdout.  (Note, "caseless" characters
 * such as numbers or punctuation are unchanged.)
 */
int main(int argc, char** argv) {

    int ch;
    while((ch = getchar()) != EOF) {
        putchar(toupper(ch));
    }
    exit(0);
}
```

5.1.1 Remarks:

- Note that "ch" is an int, not a char. Why? Because "stdin" can be any file which can contain any characters (bytes). So a "special character" cannot be used to indicate "end-of-file". (Otherwise the file could not contain that byte!) The OS knows how many bytes are in the file and how many have been read. It must return something that is NOT a byte to indicate end of file.

```
/*
 * File:   main.c
 * Author: Ken Clowes
 */


#include <stdio.h>
#include <stdlib.h>

/*
 *Description:  Reads characters from stdin and writes
 * each character reversing its case (i.e. lower case is
 * converted to upper case and vice versa)
 * to stdout.  (Note, "caseless" characters such
 * as numbers or punctuation are unchanged.)
 */

int main(int argc, char** argv) {
    int ch;

    while((ch = getchar()) != EOF) {
        if(isupper(ch)) {
            putchar(tolower(ch));
        } else {
            putchar(toupper(ch));
        }
    }
    return (EXIT_SUCCESS);
}
```

Using re-direction:

- Given the command "toupper" we can write:
  - toupper #read keyboard and echo chars converted to uppercase on the screen
  - toupper < foo #Display file "foo" in upper case on the screen

•toupper > goo #read from keyboard and write chars in uppercase to file "goo"

•toupper < foo > goo #

•toupper < foo > tmp1; reverse < tmp1 > goo2

Using pipes (|)

- Cmd1 | Cmd2 connects the stdout of Cmd1 to the stdin of Cmd2
- Thus "toupper | reverse" converts stdin to lowercase on stdout.
- This is much more efficient that creating temporary intermediate files.

Using shell scripts:

Consider the "shell script" file "mycopy":

```
#!/usr/bin/sh
./reverse < $1 | ./reverse > $2
```

A file can now be copied with: "mycopy foo goo" to copy file "foo" to "goo".

6 Memory spaces

Programs need memory for different purposes:

- Memory for the program instructions (in machine language, usually generated by a compiler of source code written by the programmer.)
- Memory for local variables and passed parameters. (The "stack".)
- Memory for (initialized) global variables.
- Memory that is allocated dynamically (for example using "malloc").
- Possibly global read-only memory.

The operating system is responsible for allocating and managing these memory segemnts.

The following program illustrates that these kind of things so use different sections of memory.

```
/*
 * File:   main.c
 * Author: Ken Clowes
 * Desctipntion: Demonstrates various memory spaces.
 */

#include <stdio.h>
```

```
#include <stdlib.h>
int g1;
int g2;
char * gcp1;
char * gcp2;

void foo(int p, int q) {
    printf("Address of parameter p:   %08X\n", &p);
    printf("Address of parameter q:   %08X\n", &q);
    printf("\n");
}

/*
 *
 */
int main(int argc, char** argv) {
    int local_1;
    int local_2;

    gcp1 = "abc";
    printf("Address of main:          %08X\n", main);
    printf("Address of foo:           %08X\n", foo);
    printf("\n");

    printf("Address of g1:            %08X\n", &g1);
    printf("Address of g2:            %08X\n", &g2);
    printf("Address of gcp1:          %08X\n", &gcp1);
    printf("Address of gcp2:          %08X\n", &gcp2);
    printf("\n");

    printf("Address of local_1:       %08X\n", &local_1);
    printf("Address of local_2:       %08X\n", &local_2);
    foo(local_1, 5);
    printf("\n");

    printf("Address of string 'abc':  %08X\n", gcp1);
    printf("\n");
    gcp2 = (char *) malloc(strlen(gcp1) + 1);
    strcpy(gcp2, gcp1);
    printf("Address of copied string: %08X\n", gcp2);
    printf("\n");


    *gcp2 = 'x';
    printf("%s\n", gcp2);

//    *gcp1 = 'x';
    printf("Goodbye\n");

    return (EXIT_SUCCESS);
```

```
}
```

Posix process creation with “fork()”

```
/*
 * File:   main.c
 * Author: Ken Clowes
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int global = 5;

int main(int argc, char** argv) {
    int local = 5;
    int i;

    if (fork() == 0) {
        for (i = 0; i < 10; i++) {
            local++;
            global++;
            printf("CHILD local: %d, global: %d\n", local, global);
            usleep(500);
        }
        exit(EXIT_SUCCESS);
    }
    for (i = 0; i < 10; i++) {
            local--;
            global--;
            printf("PARENT local: %d, global: %d\n", local, global);
            usleep(500);
        }

    return (EXIT_SUCCESS);
}
```

7 Post lecture study guide

1. Read the Wikipedia article on operating systems.
2. Suppose you have to write 3 C compilers:
    - one for a PC running Windows 7

- one for a MacBook Pro running Mac OSX
- one for an iPad running iOS 3 on an ARM 32-bit microprocessor

Which two compilers would be the most similar? Why?

8 Prep for next lecture

- Try to remember how to use the C programming language. Can you still write the "Hello World" program?
- We will be using the Netbeans Integrated Development Environment in this course. It is free open-source and runs on all popular operating systems. I suggest you download and install it on your own computer.
- If you are using Microsoft Windows (XP, Vista or 7, 8), download and install cygwin (which allows most unix commands and system calls to work under Windows.)

1. Some puzzles

See if you can compile and run the following short C programs. Can you explain the output?

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char * argv[], char **envp) {
   int x = 1;
   char ch[4];
   printf("x = %d\n", x);
   ch[12] = '1';
   printf("x = %d\n", x);
   return EXIT_SUCCESS;
}
```

```c
main() { char *s="main() { char *s=%c%s%c; printf(s,34,s,34); }"; printf(s,34,s,34); }
```

Appendix Words

- kernel
- micro-kernel
- blocking vs. non-blocking I/O
- process

- rich client platform (RCP)
- thread
- device driver
- inode
- semaphore
- message passing
- deadlock
- segmentation error
- page fault