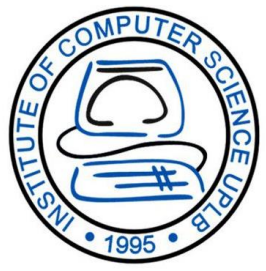
CMSC 125: Operating Systems

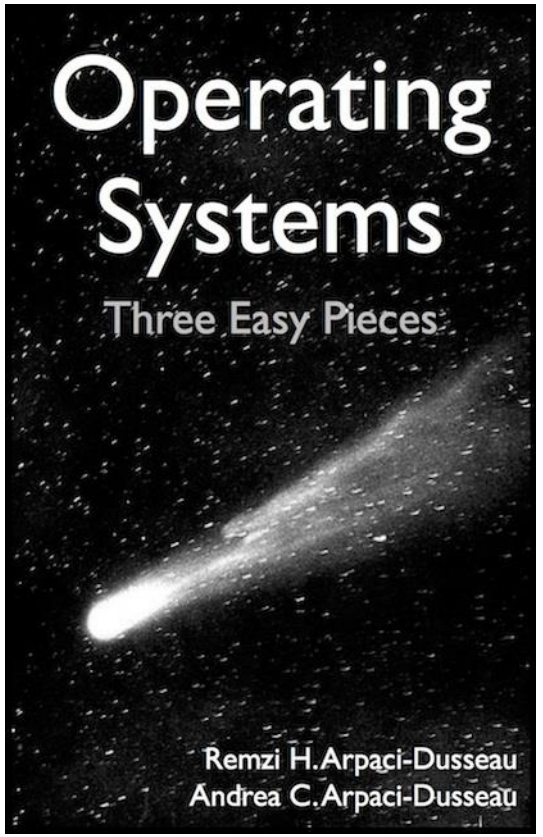
- Instructor: **Joseph Anthony C. Hermocilla**
- Email: jchermocilla@up.edu.ph
- Web: https://jachermocilla.org



Resources

Book: https://pages.cs.wisc.edu/~remzi/OSTEP/

Slides Template:
https://pages.cs.wisc.edu/~remzi/OSTEP/Educators-Slides/Youjip/



Acknowledgement

- This lecture slide set was initially developed for Operating System course in Computer Science Dept. at Hanyang University. This lecture slide set is for OSTEP book written by Remzi and Andrea at University of Wisconsin.

2. Introduction to Operating Systems

Operating System: Three Easy Pieces

What a happens when a program runs?

- A running program executes instructions.
  1. The processor **fetches** an instruction from memory.
  2. **Decode**: Figure out which instruction this is
  3. **Execute**: i.e., add two numbers, access memory, check a condition, jump to function, and so forth.
  4. The processor moves on to the **next instruction** and so on (until completion).

This is the Von Neumann model of computing.

Operating System (OS)

- Responsible for
  - Making it easy to **run** programs
  - Allowing programs to **share** memory
  - Enabling programs to **interact** with devices

OS is in charge of making sure the system operates correctly, efficiently, and in an easy-to-use manner.

Virtualization

- The primary way to realize the goals presented in the previous slide is through **virtualization**.
- The OS takes a physical resource and transforms it into a virtual form of itself.
  - **Physical resource**: Processor, Memory, Disk …
  - The virtual form is more general, powerful and easy-to-use.
  - Sometimes, we refer to the OS as a **virtual machine**.

System calls

- System calls allow user **to tell the OS what to do**
  - The OS provides some **interface** (APIs, standard library).
  - A typical OS exports a few hundred system calls for tasks such as the following:
    - Run programs
    - Access memory
    - Access devices files

The OS as a resource manager

- The OS manages **resources** such as *CPU*, *memory,* and *disk*.
- The OS allows
  - Many programs to run → Sharing the CPU
  - Many programs to *concurrently* access their own instructions and data → Sharing memory
  - Many programs to access devices → Sharing disks
- Management goals: fairness, efficiency, etc.

Examples

Virtualizing the CPU

- A system can have a single processor (CPU)..
- But can have a very large number of virtual CPUs.
  - Turning a single CPU into a seemingly infinite number of CPUs.
  - Allowing many programs to seemingly run at once → **Virtualizing the CPU**

Virtualizing the CPU (Cont.)

```
1       #include <stdio.h>
2       #include <stdlib.h>
3       #include <sys/time.h>
4       #include <assert.h>
5       #include "common.h"
6
7       int
8       main(int argc, char *argv[])
9       {
10              if (argc != 2) {
11                      fprintf(stderr, "usage: cpu <string>\n");
12                      exit(1);
13              }
14              char *str = argv[1];
15              while (1) {
16                      Spin(1); // Repeatedly checks the time and
                                 returns once it has run for a second
17                      printf("%s\n", str);
18              }
19              return 0;
20      }
```

Simple Example(`cpu.c`): Code That Loops and Prints

Virtualizing the CPU (Cont.)

- Execution result 1.

```
prompt> gcc -o cpu cpu.c -Wall
prompt> ./cpu "A"
A
A
A
^C
prompt>
```

Run forever; Only by pressing “Control-c” can we halt the program

Virtualizing the CPU (Cont.)

- Execution result 2.

```
prompt> ./cpu A & ; ./cpu B & ; ./cpu C & ; ./cpu D &
[1] 7353
[2] 7354
[3] 7355
[4] 7356
A
B
D
C
A
B
D
C
A
C
B
D
...
```

Even though we have only one processor, all four of programs seem to be running at the same time!

Policies and Mechanisms

- As a resource manager, an OS needs to implement mechanisms and enforce policies
- **Mechanism**
  - Example: How do we allow multiple programs to run at once given that we only have a single processor?
    - timer
- **Policy**
  - Example: Which program should run first? Which should run next?
    - round-robin, shortest job first

Virtualizing Memory

- The **physical memory** is *an array of bytes*.
- Each location(slot) has an address
- A program keeps all of its data structures in memory which is accessed through various instructions
  - **Read memory** (load instruction):
    - Specify an address to be able to access the data
  - **Write memory** (store instruction):
    - Specify the data to be written to the given address
- Important: Instruction/code that manipulate data structures are also in memory!

Virtualizing Memory (Cont.)

- A program that Accesses Memory (`mem.c`)

```
1       #include <unistd.h>
2       #include <stdio.h>
3       #include <stdlib.h>
4       #include "common.h"
5
6       int
7       main(int argc, char *argv[])
8       {
9               int *p = malloc(sizeof(int));  // a1: allocate some
                                                  memory
10              assert(p != NULL);
11              printf("(%d) address of p: %08x\n",
12                      getpid(), (unsigned) p);  // a2: print out the
                                                  address of the memmory
13              *p = 0;  // a3: put zero into the first slot of the memory
14              while (1) {
15                      Spin(1);
16                      *p = *p + 1;
17                      printf("(%d) p: %d\n", getpid(), *p); // a4
18              }
19              return 0;
20      }
```

Virtualizing Memory (Cont.)

- The output of the program `mem.c`

```
prompt> ./mem 0
(2134) memory address of p: 0x200000
(2134) p: 1
(2134) p: 2
(2134) p: 3
(2134) p: 4
(2134) p: 5
^C
```

  - The newly allocated memory is at address `0x200000`.
  - It updates the value and prints out the result.

- Note: Disable Address Space Layout Randomization(ASLR) first before running.
  - In linux:
    - `#echo 0 > /proc/sys/kernel/randomize_va_space`

Virtualizing Memory (Cont.)

- Running `mem.c` multiple times

```
prompt> ./mem &; ./mem &
[1] 24113
[2] 24114
(24113) memory address of p: 0x200000
(24114) memory address of p: 0x200000
(24113) p: 1
(24114) p: 1
(24114) p: 2
(24113) p: 2
(24113) p: 3
(24114) p: 3
...
```

  - It is as if each running program has its **own private memory**.
    - Each running program has allocated memory at the same address.
    - Each seems to be updating the value at `0x200000` independently.

Virtualizing Memory (Cont.)

- Each process accesses its own private **virtual address space**.
  - The OS maps (virtual) address space onto the physical memory.
  - A memory reference within one running program does not affect the address space of other processes.
  - As far as a running program is concerned, it has physical memory all to itself!
  - Physical memory is a shared resource, managed by the OS.

Problems brought about by Concurrency

- The OS is juggling many things at once, first running one process, then another, and so forth.
  - Will this lead to data inconsistencies and incorrect operation? It surely will!

- Also, modern multi-threaded programs also exhibit the concurrency problem.
  - Such as client-server applications, banking and finance applications, games

Concurrency Example

- A Multi-threaded Program (`thread.c`)

```
1       #include <stdio.h>
2       #include <stdlib.h>
3       #include "common.h"
4
5       volatile int counter = 0;
6       int loops;
7
8       void *worker(void *arg) {
9               int i;
10              for (i = 0; i < loops; i++) {
11                      counter++;
12              }
13              return NULL;
14      }
15
16      int
17      main(int argc, char *argv[])
18      {
19              if (argc != 2) {
20                      fprintf(stderr, "usage: threads <value>\n");
21                      exit(1);
22              }
```

Concurrency Example

- A Multi-threaded Program (`thread.c`)

```
1         #include <stdio.h>
2         #include <stdlib.h>
3         #include "common.h"
4
5         volatile int counter = 0;
6         int loops;
7
8         void *worker(void *arg) {
9                 int i;
10                for (i = 0; i < loops; i++) {
11                        counter++;
12                }
13                return NULL;
14  }
15  ...
```

Concurrency Example (Cont.)

```
16      int
17      main(int argc, char *argv[])
18      {
19              if (argc != 2) {
20                      fprintf(stderr, "usage: threads <value>\n");
21                      exit(1);
22              }
23              loops = atoi(argv[1]);
24              pthread_t p1, p2;
25              printf("Initial value : %d\n", counter);
26
27              Pthread_create(&p1, NULL, worker, NULL);
28              Pthread_create(&p2, NULL, worker, NULL);
29              Pthread_join(p1, NULL);
30              Pthread_join(p2, NULL);
31              printf("Final value : %d\n", counter);
32              return 0;
33      }
```

- The main program creates **two threads**.
  - Thread: a function running within the same memory/address space. Each thread start running in a routine called `worker()`.
  - `worker()`: increments a counter

Concurrency Example (Cont.)

- `loops` determines how many times each of the two workers will **increment the shared counter** in a loop.
  - `loops`: 1000.

```
prompt> gcc -o thread thread.c -Wall -pthread
prompt> ./thread 1000
Initial value : 0
Final value : 2000
```

  - `loops`: 100000.

```
prompt> ./thread 100000
Initial value : 0
Final value : 143012 // huh??
prompt> ./thread 100000
Initial value : 0
Final value : 137298 // what the??
```

Why is this happening?

- Increment a shared counter (`counter++`) → takes three instructions.
  1. Load the value of the counter from memory into register.
  2. Increment it
  3. Store it back into memory

```
.L9:
        mov     eax, DWORD PTR counter[rip]
        add     eax, 1
        mov     DWORD PTR counter[rip], eax
```

- These three instructions do not execute atomically(all at once) → **concurrency** problem occurs.
- Important: The threads here are in the same program's address space, thus sharing access to the same data (`counter`). This is different from the previous discussion on virtualizing memory.

```
ostep-code/intro  gcc -S -masm=intel threads.c
```

Persistence

- Devices such as DRAM store values in a volatile manner – lost when power is removed
- *Hardware* and *software* are both needed to store data persistently.
  - **Hardware**: I/O device such as a hard drive(HDD), solid-state drives(SSDs)
  - **Software**:
    - File system manages the disk.
    - File system is responsible for storing any files the user creates.
- The OS does not create a private, virtualized disk for each program/application
  - Assumes that files will be shared across programs or users
  - Example: Your C source code is used by the text editor and the compiler

Persistence (Cont.)

- Create a file (`/tmp/file`) that contains the string "hello world"

```
1       #include <stdio.h>
2       #include <unistd.h>
3       #include <assert.h>
4       #include <fcntl.h>
5       #include <sys/types.h>
6
7       int
8       main(int argc, char *argv[])
9       {
10              int fd = open("/tmp/file", O_WRONLY | O_CREAT
                                | O_TRUNC, S_IRWXU);
11              assert(fd > -1);
12              int rc = write(fd, "hello world\n", 13);
13              assert(rc == 13);
14              close(fd);
15              return 0;
16      }
```

`open()`, `write()`, and `close()` system calls are routed to the part of OS called the file system, which handles the requests

Persistence (Cont.)

- What OS does in order to write to disk?
  - Figure out **where** on disk this new data will reside
  - **Issue I/O** requests to the underlying storage device (via a device driver)
- File system handles system crashes during write
  - **Journaling** or **copy-on-write**
  - Carefully ordering writes to disk to get back to a usable state after failed writes – during brownouts!
- Efficiency is also important
  - I/O is slower than main memory access
  - Employs different data structures (lists, B-trees)

Design Goals

- Build up some **abstractions**
  - To make the system convenient and easy to use.
- Provide **high performance**
  - Minimize the overhead of the OS.
  - OS must strive to provide virtualization without excessive overhead.
    - Overhead sources: Extra time and extra space
- **Protection** between applications and OS
  - Isolation: Bad behavior of one does not harm other and the OS itself.

Design Goals (Cont.)

- High degree of **reliability**
  - The OS must also run non-stop.
- Other issues
  - **Energy-efficiency**
  - **Security**
  - **Mobility**

Some History

Early Operating Systems: Just Libraries

- Just a library of commonly used functions (ex. Low-level I/O)
- Old mainframes run one program at a time controlled by a human operator
- **batch processing** – jobs run by "batch"
- Non-interactive

Beyond Libraries: Protection

- Realized that code run on behalf of the OS is “special”
- OS code can be treated differently than application code
  - More privileged
- **Systems call** was invented/introduced by the Atlas Computing System
  - Added special pair of hardware instructions and hardware state
  - Makes the transition into the OS a more formal, controlled process
- A system call differs from an ordinary procedure call in that it simultaneously raises the hardware privilege level when control is transferred (during jumps) – via a **trap** instruction
  - Changes the execution state to a more privileged level called the **Kernel mode**
- **User mode** – restricted/limited execution

Multiprogramming Era

- Introduction of the **minicomputer**
- Increased developer activity as more and more people have access to hardware
- **Multiprogramming** – OS loads a number of jobs into memory and switch rapidly between them
  - Improves **CPU utilization**
  - Switch to another job when one job is waiting for I/O operation to complete
- Concept of **memory protection** and **concurrency problems** were introduced
- Introduction of the UNIX operating system

Modern Era

- Introduction of the **personal computer**
- Disk Operating System (DOS) was introduced – very limited and poorly-designed
  - No protection, single-user
- During the late 90's and early 2000's - has forgotten the lessons during the minicomputer era
- Has improved now!
  - Introduction of Linux
  - Improvements in Windows and Mac OS
  - Mobile Devices and Networks