Operating Systems

Azam Asilian Bidgoli

Course Info

Instructor

Azam Asilian Bidgoli

Email:

aasilianbidgoli@wlu.ca

Offices:

N2076J, Science Building

Office hours:

Tuesdays: 13:30-14:30

Meeting times:

Tuesdays and Thursdays: 10:00-11:20

TA: TBA

Recommended Textbooks

- Operating Systems: Three Easy Pieces, Arpaci-Dusseau, Remzi H., et.al. 2018
  - https://pages.cs.wisc.edu/~remzi/OSTEP/

- Operating System Concepts, Abraham Silberschatz. et.al. 2018

Requirements

- Students must score **50%** or more in the weighted average of midterm and final exam to pass the course i.e. **(weighted marks in the midterm exam + weighted marks in the final exam) ≥ 30.** If the student passes the final exam, then the final score will be the sum of Grades in Assignments, Quizzes, Midterms, and Final exam as per the following distribution:

Assessment	**Weighting**
Four Assignments	20% (5% each)
Quizzes	20%
Midterm Exam	25%
Final Exam	35%
Class Participation (Bonus)	3%
Total	**103%**

Policies

1. Assignments will be accepted up to **2 days** late, but **15%** will be deducted for each day late, rounded up to the nearest day. No credit will be given for assignments submitted after 2 days.
2. Quizzes must be taken individually during lecture time. The number of quizzes will depend on time constraints. Each quiz's schedule will be announced one week in advance.
3. The assignments can be done in groups of up to **3 students**. You need to register in a group by
4. The submitted work will be checked for plagiarism. The submitted work checked for plagiarism and involved students will be awarded Zero. It is considered Academic Misconduct to provide your work to another student for any reason.

Policies

- Missed Midterm
  - If a midterm exam is missed, it will be graded as 0 unless there's a legitimate documented reason. The option to shift the exam's weight is reserved exclusively for students with confirmed exceptional circumstances. An exceptional circumstance refers to an unusual and significant event that is beyond the student's control or predictability, such as a severe accident or an urgent medical condition.

Course Topics

- Processes
- Scheduling
- Synchronization
- Concurrency
- Virtual Memory
- Filesystems

1. Introduction to Operating Systems

Portions of these slides include content from the work of:

Youjip Won, KIAT OS Lab
Ali Mashtizadeh, University of Waterloo

What 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.

Operating System (OS)

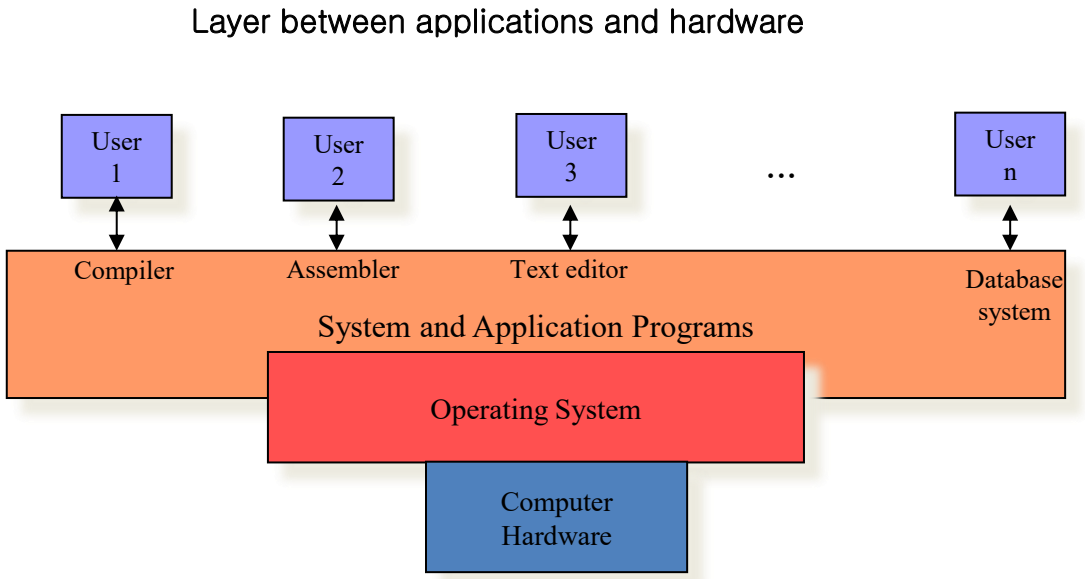
- Responsible for
  - Making it easy to **run** programs
  - Allowing programs to **share** memory
  - Enabling programs to **interact** with devices

OS is in charge of making sure the system operates correctly and efficiently.

Computer System Components

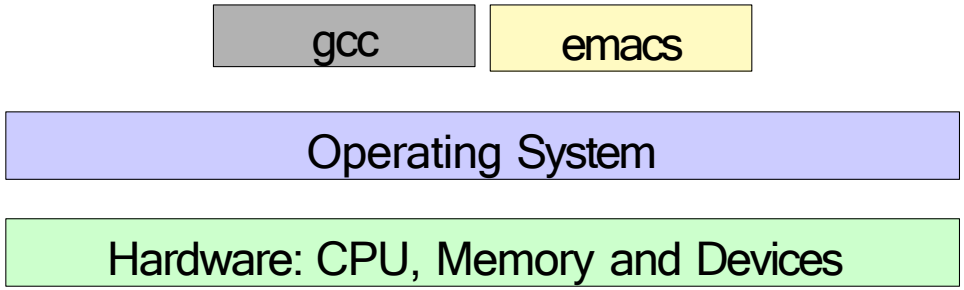
- Hardware
  - Provides basic computing resources (CPU, memory, I/O devices).
- Operating System
  - Controls and coordinates the use of hardware among application programs.
- Application Programs
  - Solve computing problems of users (compilers, database systems, video games, business programs such as banking software).
- Users
  - People, machines, other computers

Abstract View of System



- Makes hardware useful to the programmer
  - Usually: Provides abstractions for applications
  - Hides the messy, complex details of the hardware and instead offers clean, well-defined interfaces (abstractions) that applications can use to run smoothly.
  - Accesses hardware through low-level interfaces unavailable to applications
- Often: Provides protection
  - Prevents one process/user from clobbering another

Multitasking



- Idea: Run more than one process at once
  - When one process blocks (waiting for user input, IO, etc.) run another process
- Problem: What can ill-behaved process do?
  - Go into infinite loop and never relinquish CPU
  - Scribble over other processes' memory to make them fail
- OS provides mechanisms to address these problems
  - Preemption – take CPU away from looping process
  - Memory protection – protect process's memory from one another

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**.
- The OS creates a virtual layer that represents the physical resource in a way that allows multiple applications or processes to use the resource without interfering with each other. This is often referred to as virtualization.
- OS provides a layer of abstraction that simplifies the interaction between software applications and the hardware of the computer.

Virtualization

- One central question we will answer in this course is quite simple:

  How does the operating system virtualize resources?

1. what mechanisms and policies are implemented by the OS to attain virtualization?
2. how does the OS do so efficiently?
3. what hardware support is needed?

- *Why* the OS does this is not the main question, as the answer should be obvious:

  it makes the system easier to use.

The OS is a resource manager.

- The OS **manage 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

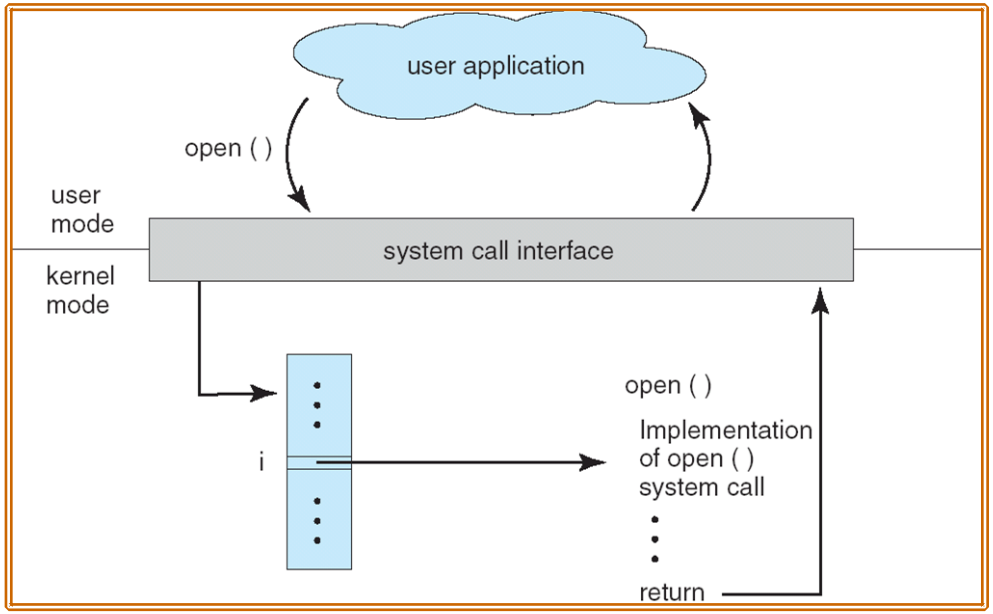
System call

- System call allows 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.
    - Run programs
    - Access memory
    - Access devices

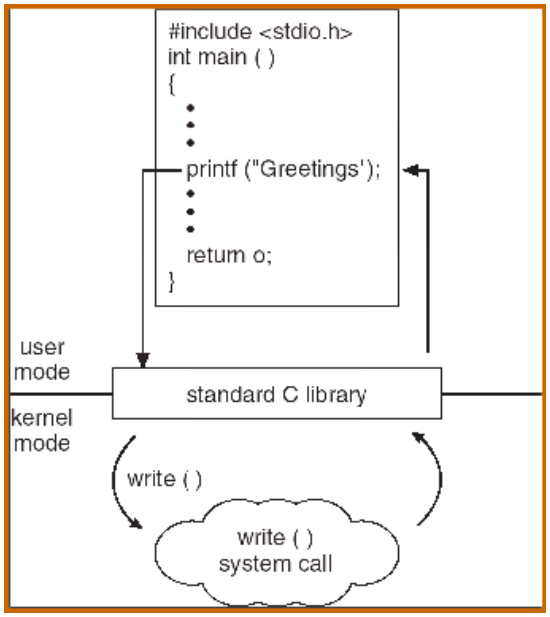
- Programming interface to the services provided by the OS
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs via a high-level Application Programming Interface (API) rather than direct system call use

System Calls

- Interface between applications and the OS.
  - Application uses an assembly instruction to trap into the kernel
  - Some higher level languages provide wrappers for system calls (e.g., C)
- System calls pass parameters between an application and OS via registers or memory
- Linux has over 300 system calls
  - read(), write(), open(), close(), fork(), exec(), ioctl(),…..



System call example



- Standard library implemented in terms of syscalls
  - *printf* – in libc, has same privileges as application
  - calls *write* – in kernel, which can send bits out serial port

***printf* is an API**: It provides a convenient and abstracted way to output formatted text, relying internally on system calls like write to perform the actual output operation.
***write* is a system call**: It is the fundamental building block used by many higher-level APIs for writing data to various outputs, including files and terminals. write(fileDescriptor, data, #bytes);

Examples of Windows and Unix System Calls

	Windows	Unix
Process Control	CreateProcess()	fork()
	ExitProcess()	exit()
	WaitForSingleObject()	wait()
File Manipulation	CreateFile()	open()
	ReadFile()	read()
	WriteFile()	write()
	CloseHandle()	close()

Virtualizing the CPU

- The system has 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 main(int argc, char *argv[])
8       {
9               char *str = argv[1];
10              while (1) {
11                      Spin(1);
12                      printf("%s\n", str);
13              }
14              return 0;
15      }
```

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!

Virtualizing Memory

- The physical memory is *an array of bytes*.
- A program keeps all of its data structures in memory.
  - **Read memory** (load):
    - Specify an address to be able to access the data
  - **Write memory** (store):
    - Specify the data to be written to the given address

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); // If this passes, the program continues
11          printf("(%d) address of p: %p\n",
12                  getpid(), (unsigned) p);  // a2: print out the
                                                 address of the memory
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) value of p: %d and addr pointed to by
                    p: %p\n", getpid(), *p, p); // a4
18          }
19          return 0;
20  }
```

Virtualizing Memory (Cont.)

- The output of the program `mem.c`

```
prompt> ./mem
(2134) memory address of p: 00200000
(2134) p: 1
(2134) p: 2
(2134) p: 3
(2134) p: 4
(2134) p: 5
^C
```

  - The newly allocated memory is at address `00200000`.
  - It updates the value and prints out the result.

Virtualizing Memory (Cont.)

- Running `mem.c` multiple times

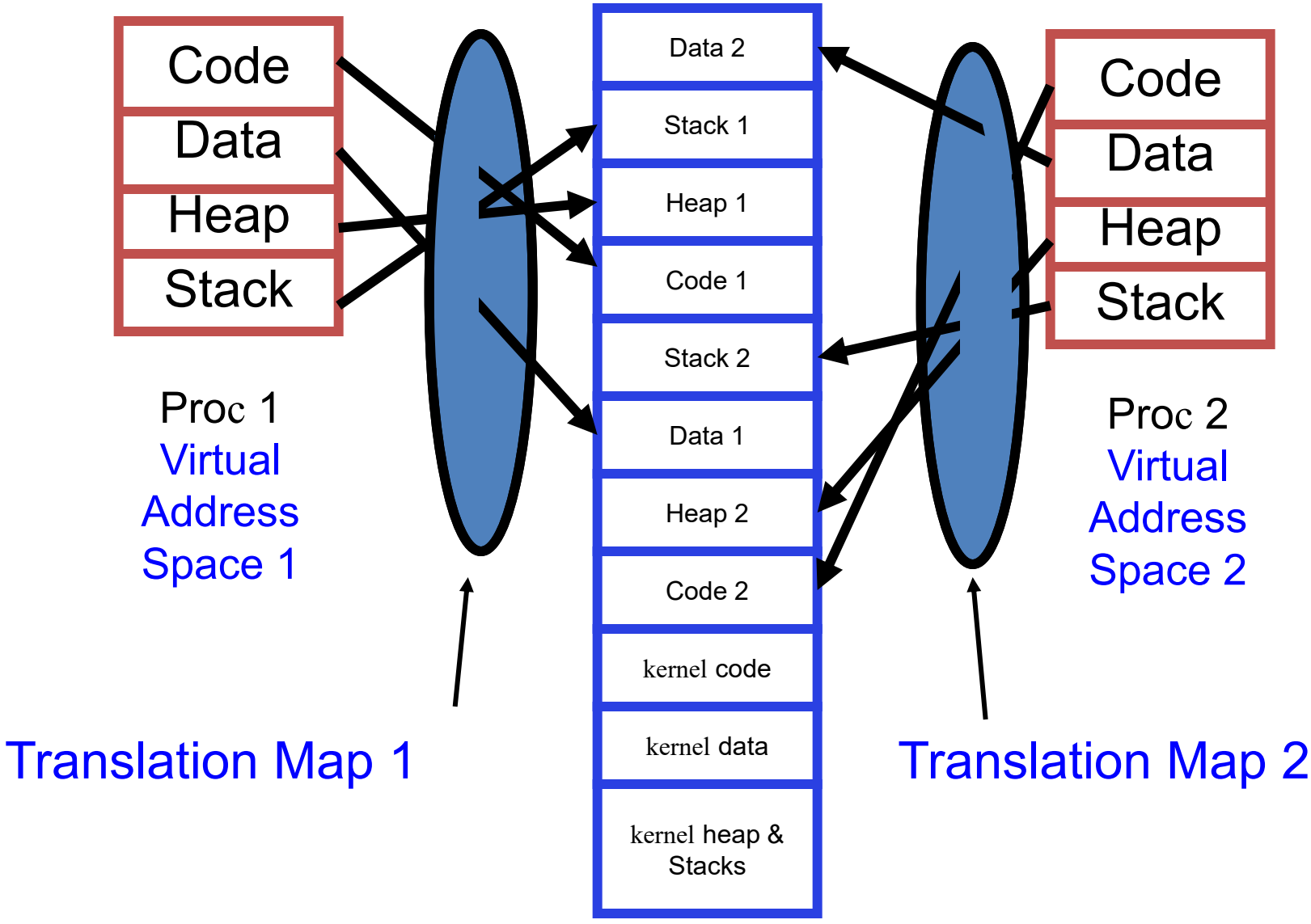
```
prompt> ./mem &; ./mem &
[1] 24113
[2] 24114
(24113) memory address of p: 00200000
(24114) memory address of p: 00200000
(24113) p: 1
(24114) p: 1
(24114) p: 2
(24113) p: 2
(24113) p: 3
(24114) p: 3
...
```

  - 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 `00200000` independently.

Virtualizing Memory (Cont.)

- Each process accesses its own private **virtual address space**.
  - The OS maps address space onto the physical memory.
  - A memory reference within one running program does not affect the address space of other processes.
  - Physical memory is a shared resource, managed by the OS.

Providing the Illusion of Separate Address Spaces



The problem of Concurrency

- The OS is juggling many things at once, first running one process, then another, and so forth.

- Modern multi-threaded programs also exhibit the concurrency problem.

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 main(int argc, char *argv[])
18      {
19
```

Concurrency Example (Cont.)

```
23              loops = atoi(argv[1]);
24              pthread_t p1, p2;
25              printf("Initial value : %d\n", counter);
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 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 thread.c -o thread
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 → take three instructions.
  1. Load the value of the counter from memory into register.
  2. Increment it
  3. Store it back into memory

- These three instructions do not execute atomically. → Problem of **concurrency** happen.

Concurrency

- Main question:

  HOW TO BUILD CORRECT CONCURRENT PROGRAMS

- When there are many concurrently executing threads within the same memory space, how can we build a correctly working program?
- What primitives are needed from the OS? What mechanisms should be provided by the hardware?
- How can we use them to solve the problems of concurrency?

Persistence

- Devices such as DRAM store values in a volatile.
- when power goes away or the system crashes, any data in memory is lost.
- *Hardware* and *software* are needed to store data persistently.
  - **Hardware**: I/O device such as a hard drive, solid-state drives(SSDs)
  - **Software**:
    - File system: the software in the operating system that usually manages the disk.
    - File system is responsible for storing any files the user creates.

Design Goals

- Build up **abstraction**
  - 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.
- **Protection** between applications
  - Isolation: Bad behavior of one does not harm other and the OS itself.