Introduction

Operating Systems
Based on: Three Easy Pieces by Arpaci-Dusseaux

Moshe Sulamy

Tel-Aviv Academic College

What is an Operating System?

Not a simple question.

What is an Operating System?

Not a simple question.

Users
Applications
Operating System
Hardware

- Middleware between user programs and hardware
- Abstracts and manages resources
    - CPU
    - Main memory
    - I/O Devices (disk, network card, mouse, keyboard, monitor, etc.)

Why study Operating Systems?

- We study Computer **Science**
  - Not a programming course...
- You use an operating system
  - The machine is (mostly) useless without an OS
  - Understand what you use
  - Why and how is the OS useful?
- Behavior of OS impacts entire machine
  - Understand system performance
  - Useful to know how computers work

Approach

- The course is about ideas and analysis
- We will not build an OS
    - But there will be coding

CPU wise, What happens when a program runs?

- Executes instructions
- The processor:
  - **Fetches** an instruction from memory
  - **Decodes** the instruction
  - **Executes** it
  - Moves on to the **next instruction**

- **Von Neumann** model of computing

What does the OS do?

- Abstraction
  - Virtual resources that correspond to hardware resources
  - Well-defined operations on these resources
    - CPU $\rightarrow$ Running program (process / thread)
    - Memory $\rightarrow$ Address space / virtual memory
    - Storage $\rightarrow$ Files, file system

What does the OS do?

- Resource Management
  - Share resources among running programs
  - Decide who gets how much and when
    - CPU $\rightarrow$ Who runs next?
    - Memory $\rightarrow$ Where is data in RAM and when to access it?
    - Storage $\rightarrow$ Where and how are files stored on disk?

Three Easy Pieces

- **Virtualization**
  - As if each program has resources to itself
- **Concurrency**
  - Juggling many things at once
- **Persistence**
  - Ability to store data beyond termination / computer shutdown

Virtualizing the CPU

cpu.c:

```c
int main(int argc, char *argv[])
{
    while (1) {
        spin(1); // returns after 1 second
        printf("%s\n", argv[1]);
    }
}
```

- The program loops and prints

Virtualizing the CPU

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

- Runs forever
- Halt program by pressing “Control-C”

Virtualizing the CPU

```
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
...
```

- We have **one** processor
- All four seem to be running **at the same time**

Virtualizing the CPU

- OS illusion
  - Single CPU $\rightarrow$ infinite number of <u>virtual</u> CPUs
  - Programs seemingly run at once

Virtualizing the CPU

- OS illusion
  - Single CPU $\rightarrow$ infinite number of <u>virtual</u> CPUs
  - Programs seemingly run at once

- Under the hood:
  - Multiple instances are started
  - OS picks <u>one</u> to run (use CPU)
  - After a while, OS kicks it off the CPU
  - Picks another one to run

Virtualizing the CPU

- **Context Switch**
  - Running program pauses, another is brought in
  - Program does not know when it is context-switched (in or out)
    - Illusion that it is alone on the CPU
    - Fetch-Decode-Execute cycle continues
  - Fast and frequent
    - Appears to be running at the same time

Virtualizing Memory

- Physical memory (**RAM**) - array of bytes
  - **Read** (load) - specify address to access data
  - **Write** or **update** (store) - also specify data to write
- Program code (instructions) is also in memory!

Virtualizing Memory

mem.c:

```c
int main(int argc, char *argv[])
{
    int p;
    printf("(%d) the address of p: %p\n",
        getpid(), &p);
    p = atoi(argv[1]);
    while (1) {
        spin(1);
        p = p + 1;
        printf("(%d) p: %p\n", getpid(), &p);
    }
}
```

Virtualizing Memory

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

- Increments and prints every second

Virtualizing Memory

```
prompt> ./mem 0 & ./mem 100 &
[1] 13526
[2] 13527
(13527) the address of p: 0x200000
(13526) the address of p: 0x200000
(13527) p: 101
(13526) p: 1
(13527) p: 102
(13526) p: 2
(13527) p: 103
(13526) p: 3
...
```

- Same address, different value
- As if each instance as its own private memory

Virtualizing Memory

- Program address is <u>not</u> physical address
  - It is a **virtual address**
- Each process accesses its own **virtual address space**
  - The OS (with hardware help) maps it onto the physical memory
  - Reference in one running program does not affect the other

Virtualizing Memory

- Program address is <u>not</u> physical address
  - It is a **virtual address**
- Each process accesses its own **virtual address space**
  - The OS (with hardware help) maps it onto the physical memory
  - Reference in one running program does not affect the other

- Each program seemingly has all physical memory to itself
  - No knowledge (or responsibility) of other programs
  - **Memory protection**

Concurrency

- OS is working on many things at once:
  - Each program has "its own" CPU and RAM
  - Many programs run at the same time
- Modern **multi-threaded** programs exhibit the same problems

- Concurrency is everywhere

Concurrency

threads.c:

```c
volatile int counter = 0;
int loops;

void* worker(void *arg) {
    for (int i = 0; i < loops; ++i)
        ++counter;
    return NULL;
}

int main(int argc, char *argv[]) {
    loops = atoi(argv[1]);
    pthread_t p1, p2;
    printf("Initial value : %d\n", counter);
    pthread_create(&p1, NULL, worker, NULL);
    pthread_create(&p2, NULL, worker, NULL);
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    printf("Final value   : %d\n", counter);
}
```

Concurrency

- The program creates two **threads**
  - Thread: a function running concurrently within the same address space
- Each thread executes `worker()`
  - Increments a global (shared) counter
- `loops`: how many times to increment the counter

Concurrency

`loops = 1000`:

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

`loops = 100000000`:

```
prompt> ./thread 100000000
Initial value : 0
Final value   : 196445738 // huh??
prompt> ./thread 100000000
Initial value : 0
Final value   : 197944967 // what the??
```

Concurrency

- Three instructions to increment a counter:
  - Load the value into a register
  - Increment the register
  - Store the register back into memory
- Does not execute **atomically** (without interference)
- A problem of **concurrency**

Persistence

- System memory (such as DRAM) is **volatile**
  - Data is lost when power goes away or the system crashes
- We need hardware and software to store data **persistently**

Persistence

- Hardware: **I/O device** such as a **hard drive** or **SSD**
- Software: **file system**
  - Manages the disk
  - Responsible for storing user files

- No private, virtualized disk
- Files can be viewd as virtualized disks.
  - Users want to **share** information that is in files

Persistence

```c
int main(int argc, char *argv[]) {
    int fd = open("/tmp/file", O_WRONLY|O_CREAT,
        S_IRWXU);
    assert(fd > -1);
    int rc = write(fd, "hello world\n", 12);
    assert(rc == 12);
    close(fd);
}
```

- The program makes three calls:
  - `open()`: opens (and creates) the file
  - `write()`: writes data to the file
  - `close()`: closes the file

Persistence

- These are **System calls**
  - Routed to part of the OS called the **file system**
  - Handles requests and returns error code
  - Like a **standard library** for OS operations
- The **file system**:
  - Figures out where on disk the new data will reside
  - Updates various structures
  - Issues I/O requests to the underlying storage device

Kernel

- The **kernel** is a core part of the OS:
  - Always in memory
  - Executed in response to events
    - External events (**interrupts**), e.g., clock
    - Requests from running programs (**system calls**)
- What we think of as “OS” is not always part of the kernel
  - e.g., the Unix Shell is an application
- The kernel **is not** a running program

Kernel - Event Handler

- An **event**: mouse is moved, key is pressed, network communication, division by zero, system call, etc.
- **Interrupts** the current program, executes kernel code
- The kernel defines a **handler** for each event type

Events

- **Interrupt** - asynchronous (external) event
  - For example: key pressed
  - Kernel stores it, can be checked later
- **Trap** - synchronous (internal) event
  - Also **exception** or **fault**
  - For example: division by zero, kernel terminates program
  - Not necessarily for errors!

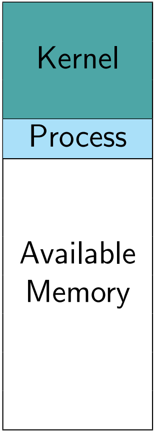
System Calls

- User program wants to invoke OS - places a **system call**
  - For example: open a file, allocate memory, get keyboard input
- Special instruction that causes a **trap**
- Calls a procedure in the **kernel**
  - The specific **event handler**

Physical Memory

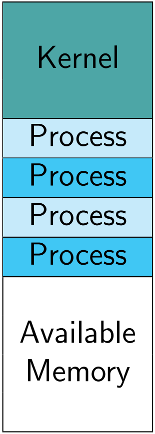
- The kernel resides in memory

Physical Memory



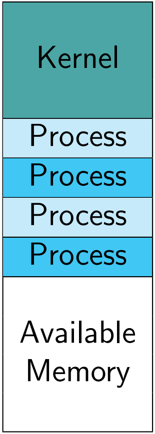
- The code & data of each running program (a **process**) is loaded into memory
  - RAM is divided: user, kernel
  - **System call**: invoke kernel code, then return to user code

Physical Memory



- Several processes can exist in parallel
  - **Memory protection**: each process is seemingly alone

Physical Memory



- Several processes can exist in parallel
  - **Memory protection**: each process is seemingly alone
- This is a major simplification
  - But it suffices for now

Design Goals

- Abstraction
  - Dividing into small, understandable pieces
  - Make the system convenient and easy to use
- Performance
  - Minimize the overhead of the OS
  - Provide virtualization without excessive overheads
- Protection
  - Malicious or bad behavior of one application does not harm others or the OS
  - **Isolation** of processes
- Reliability
  - The OS must run non-stop

Design Goals

- Other goals:
  - Energy efficiency
  - Security
  - Mobility

Some History

- Early OS: Just libraries
  - Commonly-used functions, e.g., low-level I/O
  - No abstraction, no virtualization
  - One program at a time
  - **Batch mode**

Some History

- Early OS: Just libraries
  - Commonly-used functions, e.g., low-level I/O
  - No abstraction, no virtualization
  - One program at a time
  - **Batch mode**
- Beyond Libraries: Protection
  - **User mode** with hardware restrictions
  - **System call**: instead of a library procedure
    - Raises privilege to **kernel mode**
    - OS has full access to hardware

Some History

- Era of Multiprogramming
  - Make better use of machine resources
  - Load a number of jobs, switch rapidly between them
  - **Context switch**, **concurrency**
  - **Memory protection** became important

Some History

- Era of Multiprogramming
  - Make better use of machine resources
  - Load a number of jobs, switch rapidly between them
  - **Context switch**, **concurrency**
  - **Memory protection** became important
- The Modern Era
  - The **PC**: the dominant force in computing

Summary

- The OS: abstraction & resource management
- Multiprogramming & concurrency via context switching and memory protection
- Kernel: OS code & data that is always in memory
    - Not a running program, but pieces of code executed in response to events
- System calls: user events to trigger kernel code to act on their behalf