Operating Systems

Chapter 2: Introduction to Operating Systems

A running program executes instructions. We **fetch** -> **decode** -> **execute**. Then we move to the next one.

> This is the Von Neumann model of computing

Thus, this body is called the **Operating System**

The operating system **virtualizes** physical resources and create a general, easy to use **virtual machine**. These operating systems thereby provide **system calls** by which programs can interact with other parts. Thus, a **standard libraru**.

The operating system then manages these.

Virtualizing CPU

The operating system virtualizes the CPU to allow for multiple programs to "simultaneously" run. Do note that this isn't truly simultaneous.

The **policy** is the system by which the OS chooses which program to be running at any given time. This is mainly hueristics based on many factors.

Virtualizing Memory

Memory is just an array of bytes. **address** -> **data**. THe operating system virtualizes the memory in such a way that every program gets its own set of memory. This simplifies sharing memory.

Each process has its own **virtual address space** (or address space).

The OS manages the actual mapping of hardware memory.

Concurrency

Another thing the OS does is run multiple programs at the same time. Often, these programs need to access the same resources.

If programs were to directly access the same memory, they would end up with race conditions, wherein one would modify the memory before another, thus getting overwritten.

Persistence

The operating system must also handle persistence of data. Programs cannot exist only in memory – they must have access to some amount of persistent storage. Whether that be files or whatnot.

Some of these might be **input/output (I/O)**, as well as storage in **hard disk drives (HDDs)** or **solid-state drives (SSDs)**.

THe software in the OS manages this **file system** and any **files** the user creates.

Files, unlike memory and CPU, are not virtual and are expected to be shared between programs. (You wouldn't want a file written by nvim to be only accessable by nvim, would you?)

Design Goals

The basic goal of the OS is to create enough **abstractions** to make the system easy to use for most developers.

Some other goals are for high performance, or to minimize the overheads of the OS. Virtualization is nice, but it does maintain an overhead that might not be wanted in other places.

We also want to provide **protection** between applications, and the OS and appplications. If we are to **isolate** different processes from eachother, we can reduce unexpected interference and thus have more stable programs.

Do note that building a reliable operating system is hard.

Some History
Operating systems have to be understood with respect to their history.

Early Operating Systems: Just Libraries
In the begining operating systems were just libraries of code in which programmers could rely on rather than creating for themselves.

This is because many computers were super large and would still be ran by human operators.

We used **batch** processing in which jobs were setup and run in a batch by the operator.

Beyond Libraries: Protection
After the basic libraries, operating systems began to add feature upon feature. We now have system calls to provide access to the system and a file systel to allow for persistence.

Procedures and system calls are different allowing for different levels of protection. For example, the application has no ability to write/read files, but the system can through a syscal.

The OS does this, and then returns through a **return-from-trap**

The Era of Multiprogramming

The Modern Era

Closing

Chapter 4: The Abstraction: The Process
A **process** is just a **running program**. Program itself just sits on the disk – You can think of the program as the actual executable. The process as what is running.

In order to run more than one process at a time, we **virtualize** the CPU. This means to **time share** the cpu.

mechanisms – low level machinery. They are low level methods/protocls that implement a needed piece of functionality.

context switch – what gives the OS the ability to stop one program and start another.

schedueling policy – makes the decision of which programs should run.

4.1 The Abstraction: A Process
machine state – what a program can read or update when its running.

Composes of: *memory*, *registers*

Some of those registers are special. The **program counter (PC)**, **stack pointer**, **frame pointer**.

Programs have persistent storage device. (I/O information as well).

4.2 Process API
The process api has a few required features:

- Create – the method to create the process.

- Destroy – the method to destroy the process. This is forceful.
- Wait – To wait for a process to stop running.
- Miscellaneus Control – suspend, etc
- Status – get info from the process

4.3 Process Creation: A Little More Detail

1. The os **loads** the code (and static data) into memory

Modern OSes do this **lazily**. Older OSes do this **eagerly**.

See: **paging** and **swapping**

2. The os allocates for the **run-time stack** (or just stack)

3. The os allocates for the program's **heap**

4. Then the OS will do other initialization tasks. Setting up IO. Jumping into _start

4.4 Process States

Processes (obviously) have different states.

- Running – duh
- Ready – The program is ready to run, but its not runnning quite yet.
- Blocked – The program has performed an operation that makes it not ready until some other event takes place. Think I/O requests.

Being moved from ready to running means the process is **scheduled**

Being moved from running to ready means the process is **descheduled**

Processes which are blocked, will stay blocked until their event has occured.

4.5 Data Structures

The OS is a program, and like all programs, it has some data to keep.

The OS has a **process lsist** for all processes. Including states and other ifnromation.

```
// the registers xv6 will save and restore
// to stop and subsequently restart a process
struct context {
int eip;
int esp;
int ebx;
int ecx;
int edx;
int esi;
int edi;
int ebp;
};
// the different states a process can be in
enum proc_state { UNUSED, EMBRYO, SLEEPING,
RUNNABLE, RUNNING, ZOMBIE };
// the information xv6 tracks about each process
// including its register context and state
struct proc {
char *mem; // Start of process memory
uint sz; // Size of process memory
char *kstack; // Bottom of kernel stack
// for this process
```

```
enum proc_state state; // Process state
int pid; // Process ID
struct proc *parent; // Parent process
void *chan; // If !zero, sleeping on chan
int killed; // If !zero, has been killed
struct file *ofile[NOFILE]; // Open files
struct inode *cwd; // Current directory
struct context context; // Switch here to run process
struct trapframe *tf; // Trap frame for the
// current interrupt
};
```

Note that there are more states. For example **final/zombie** or **initial**.

You can reach zombie state if you are looking at the exit codes or something.

4.6 Summary

process = running program

Chapter 6: Mechanism: Limited Direct Execution

There are a few problems in virtualizing the CPU

- performance – how can we implement a virtualizer without adding overhead
- control – how do we run them whie maitaining control over the CPU

Control is especially important.

6.1 Basic Technique: Limited Direct Execution

The basic idea is to run the program directly on the CPU.

- create a process entry
- allocate memory
- load program
- setup stack (argc/argv)
- clear registers
- execute call main()
- ▸ run main()
- ▸ execute return from main
- free memory of process
- remove from process list

There are two problems. Namely,

1. How do we make sure the program we don't want it to do? While still running it efficiently.
2. How do we stop the process and switch to another?

This is where "limited" comes from. We have no control.

6.2 Problem 1: Restricted Operations

How can we want to run a restricted operation (I/O request, or gain access to resources)?

We could let the process do whatever it wants, but then we wouldn't be able to restrict permissions based on some factors (file permissions).

One approach is to create different processor modes. **user mode** and **kernel mode**.

- user mode – a process can't issue I/O requests, so it would get killed if it did so

- kernel mode – code in this mode can basically do whatever it wants

A process can execute a system call that raises into kernel mode, in which the OS runs its code.

System calls deal with **trap** instructions and **return from trap** instructions. A **trap** instruction jumps into kernel and raises the privlege level to kernel mode. The **return from trap** instruction, returns back to user mode.

The kernel doesn't let the user program specify which address to jump to. That would allow for arbitrary code execution. Instead a **trap table** maps traps to their **trap handlers** (or essentially their addresses).

Note that being able to find the trap rables is a **privleged** operation. You don't want any process to just be able to write their own trap table.

Thus, we have two phases in **LDE**:

1. Boot time – kernel initializes trap table; cpu remembers its location
2. Running a process – kernel sets up a few things (node on the process list, allocating memory, etc)

6.3 Problem 2: Switching Between Processes

Switching between processes is super complicated. If a process is running, that means the OS is not running.

A Cooperative Approach: Wait for System Calls

In the **cooperative** approach, the OS assumes that a process will give up the CPU periodically so that the OS can run something else.

- Systems like these have the **yield syscall** to allow for transfer back to the OS.
- Applications also transfer when they do illegal operations (divide by 0, access memory it can't, etc)

A Non-Cooperative Approach: The OS Takes Control

Aside from **rebooting the machine** there aren't many options to kill a process.

A quick approach to do this is to have a **timer interrupt** that raises an interupt every X amount of miliseconds. Thus, the OS regains control of the CPU.

Once the timer is started (in the boot sequence), the OS is safe to handle control over to different processes.

Saving and Restoring Context

Once the OS has control, a choice is made: (1) return back to the process (2) return to a different process. This choice is made by the **schedueler**

If a different one is selected, the OS executes a **context switch**. The OS saves a few register values from the current process, and restores the "new" process into memory.

The OS does the opposite for the current process.

During the context switch, the registers must be saved, and then registers are written to. The OS stores registers in software until they are needed.

6.4 Worried About Concurrency?