Operating Systems: Three Easy Steps

Kunwar Shaanjeet Singh Grover

Contents

Chapter 1

Introduction

1.1 Virtualisation

The OS takes a **physical** resources (such as the processor , or memory, or a disk) and transforms it into a more general, powerful, and easy-to-use **virtual** for of itself. Thus, we sometimes refer to the operating system as a **virtual machine**. This general technique of transforming is called **virtualisation**.

1.1.1 CPU Virtualisation

Turning a single CPU into a seemingly infinite number of CPUs and thus allowing many programs to seemingly run at once is what we call **virtualizing the CPU**.

1.1.2 Memory Virtualisation

Each process accesses its own **virtual address spaces**, which the OS somehow maps onto the physical memory of the machine. Exactly how this is accomplished is what we study.

1.2 Concurrency

Concurrency is a conceptual term to refer to a host of problems that arise, and must bf addressed when working on many things at once (i.e. concurrently) in the same program.

1.3 Presistence

The software in the operating system that usually manages the disk is called the **file system**; it is this responsible for storing any files the user creates in a

reliable and efficient manner on the disks of the system. The file system is the part of OS in charge of managin persisten data. What techniques are needed to do so? What mechanisms and policies are required to do so with a high probability?

Chapter 2

CPU Virtualisation

2.1 The Abstraction: The Process

Process: A running program.

> **Crux Of The Problem:** Although there are only a few physical CPUs available, how can the OS provide the ullusion of nearly-endless supply of said CPUs?

The OS creates this illusion by **virtualizing** the CPU. By running one process, then stopping it and running another, and so forth, the OS can promote the illusion that many virtual CPUs exist when in fact there is only one physical CPU (or a few). This technique is called **time-sharing**.

To do this, OS requires some low-level machinery and some high-level intelligence. The low-level machinery is called **mechanisms**. For example, stopping one program and starting another on a given CPU. On top of these mechanisms, there is high-level intelligence call **policies**. Policies are algorithms for making decisions within the OS.

Mechanisms: Provides answer to a *how* question.
Policies: Provides answer to a *which* question.

2.2 Mechanism: Limited Direct Execution

> **The Crux:** How to efficiently virtualize the CPU with control?

The OS must virtualize the CPU in an efficient manner while retaining control over the system. To do so, both hardware and operating-system support

will be required. The OS will often use a judicious bit of hardware support in order to accomplish its work effectively.

2.2.1 Basic Technique: Limited Direct Execution

The *direct execution* part of the idea: just run the program directly on the CPU. Thus, when the OS wishes to start a program, it creates a process entry for it in a process list, allocates some memory for it, loads the program code into memory, locates its entery point (main()), jumps to it, and starts running the user's code.

This approach gives rise to a few problems:

- How to make sure that the program does not do anything that we dont want it to do while still maintaining efficency?
- How do we stop a process and switch to another process, i.e. how to implement **time sharing** we require to virtualize the CPU?

Problem #1: Restricted Operations

What if the process wishes to perform some kind of restricted operation, such as issuing an I/O request to a disk, or gaining access to more system resources such as CPU or memory?

> **The Curx: How to perform restricted operations**
> A process must be able to perform I/O and some other restricted operations, but without giving the process complete control over the system.

Two differend modes:

- **User Mode:** Code that runs in user mode is restricted in what it can do. For example, when running in user mode, a process can't issure I/O requests; doing so would result in the processor raising an exception.
- **Kernal Mode:** Mode the operating system (or kernal) runs in. In this mode, code that runs can do whatever it likes, including privileged operations such as issuing I/O requests and executing restricted instructions.

All modern hardware provides the ability for user programs to perform a **system call**. System calls allow the kernal to carfully expose certain key pieces of functionality to user programs such as accessing the file system, creating and destroying the processes, communicationg with other processes, and allocating more memory.

System calls To execute a system call, a program must execute a special **trap** instruction. This instruction jumps into the kernal and raises privilege level to kernal mode; once in the kernal, the system can now perform the restricted operations needed and thus the required work for the calling process. When finished, the OS calls a special **return-from-trap** instruction, which returns into the calling user program while simultaneously reducing the privilege level back to user mode.

When executing a trap, the caller's registers must be saved in order to return correctly when the OS issues the return-from-trap instruction.

On x86, the processor will push the program counter, flags and a few other registers onto a per-process **kernal stack**; return-from-trap will pop these values from the stack and resume execution.

How does the trap know which code to run inside the OS? Clearly, the calling process can't specify and address to jump to; doing so would allow programs to jump anywhere into the kernal which is not secure. The kernal must control what code executes upon a trap.

The kernal does so by setting up a **trap table** at boot time. When the machine boots up, it does so in kernal mode. One of the first things the OS does is to tell the hardware what code to run when certain exceptional events occur. The OS informs the hardware of these locations of these **trap handlers**, usually with some kind of special instruction. Once the hardware is informed, it remembers the location of these handlers until the machine is rebooted, and thus hardware knows what to do when system calls and other exceptional events take place.

To perform the exact system call, a **system-call number** is usually asigned to each system call. The user code has to place the desired system-call number in a register or at a specific location in the stack; the OS, when handeling the system call inside the trap handler, examines the number, ensures it is valid, and, if it is, executes the corresponding code. This level of indirection serves as a form of **protection**; user code cannot specify an exact address to jump to, but rather must request a particular service via a number.

Telling the hardware where the trap tables are is a **privileged** operationg, otherwise you could make your own trap tables and compromise the system.

How all this works:

1. At boot time, the kernal initializes the trap table, and the CPU remembers its location for subsequent use. The kernal does so via a privileged instruction.

2. The kernal sets up a few things (allocating memory, etc.) before using a return-from-trap instruction to start the execution of the process. When the process wishes to issue a system call, it traps back into the OS, which handles it and once again returns control via a return-from-trap to the process. The process completes its work and returns from main().

Problem #2: Switching Between Processes

When a process is running on the CPU, this by definition means the OS is *not* running. If OS isnt running how can it change a process? (it cant).

> **The Crux: How to regain control of the CPU?**
> How can the operating system **regain control** of the CPU so that it can switch between processes?

Cooperative Approach: Wait for system calls In this approach, the OS regains control of the CPU by waiting for a system call or an illegal operation of some kind to take place, as control is passed to the OS during these exceptions.

Non-Cooperative Approach: The OS takes control If we get stuck in an infinite loop in the cooperative approach, the only way out is to reboot the machine.

> **The Crux: How to gain control without cooperation**
> How can the OS gain control of the CPU even if processes are not being cooperative? What can the OS do to ensure a rogue process does not take over the machine?

Timer interrupt: A timer device is programmed to raise an interrupt after a fixed delay; when the interrupt is raised, the currently running process is halted, and a pre-configured **interrupt handler** in the OS runs. Now the OS has regained control of the CPU, and thus can stop the current process and start a different one.

At boot time, the OS starts the interrupt timer. This is a privileged operation.

Saving and Restoring Context The OS has regained control now. It can decide to switch or not. This decision is made by the **scheduler**.

If the decision is to switch, the OS executes a **context switch**. A context switch saves register values of the currently-executing process and restores register values for the soon-to-be-executing process. After this, the OS executes a return-from-trap instruction and executes the new process.

By switching stacks pointers (stack pointer is a register), the kernel enters call to the switch code in the context of one process (the interrupted one) amd returns in the context of another (the soon-to-be-executing one). By switching stack is actually how the process is switched.

There are two types of register saves/restores that happen here:

- When the timer interrupt occurs. The *user registers* of the running process are implicitly saved by the *hardware*, using the kernal stack of the process.
- When OS performs the context switch. The *kernal registers* are explicitly saved by the *software* (i.e. th OS), but this time into memory in the process structure of the process.

More info on the last two points here

2.3 Scheduling

In this section we will understand **policies** that OS scheduling employs.

> **The Crux: How To Develop Scheduling Policy**
>
> How should we develop a basic framework for thinking about scheduling policies? What are the key assumptions? What metrics are important? What basic approaches have been used in earliest of computer systems?

We will make some assumptions and continue to relax them to get to an optimal policy. Assumptions:

1. Each job runs for the same amount of time.
2. All jobs arrive at the same time.
3. Once started, each job runs to completion
4. All jobs only use the CPU (i.e., they perform no I/O)
5. The run-time of each job is known

For now, we will use a single metric to determine optimal policy: **turnaround time**.

$$T_{turnaround} = T_{completion} - T_{arrival}$$

2.3.1 First In, First Out (FIFO)

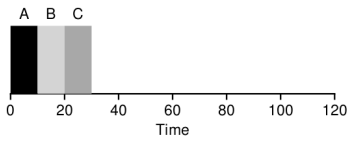

Figure 2.1: FIFO Simple Example

For our current assumptions, this is optimal. But if we relax the assumption that job running time are same, we can get bad turnaround time:

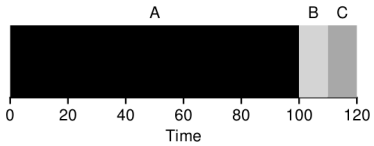

Figure 2.2: Why FIFO is Not that Great

2.3.2 Shortest Job First (SJF)

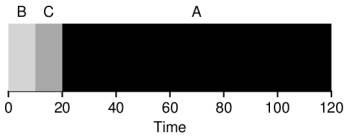

Figure 2.3: SJF Simple Example

It can be proven that under these assumptions, SJF will give the best turnaround time.

Now, lets relax the assumption that all jobs arrive at the same time. Now, we can build a worst case for SJF:

2.3.3 Shortest Time-To-Completion First (STCF)

To address this concern, we need to relax assumption 3 (jobs started, must run to completion). SJF by our defination was a **non-preemptive** scheduler, and thus suffered from the problems in figure 2.4.

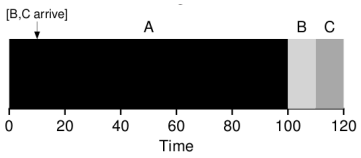

Figure 2.4: SJF With Late Arrivals from B and C

To improve upon that, we add preemption to SJF, known as STCF. Any time a new job enters the system, the STCF schedular determines which of the remaining jobs has least time left, and schedules that. It can be proven that under the current assumptions, this is the best schedular (w.r.t turnaround time)

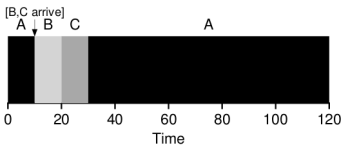

Figure 2.5: STCF Simple Example

New Metric: Response Time: If our only metric was turnaround time and we knew job runtime, STCF would be a great policy. But inroduction of time-sharing machines, required interactive performance from sustem as well. Thus a new metric was needed: **response time**

$$T_{response} = T_{firstrun} - T_{arrival}$$

2.3.4 Round Robin

To improve upon response time, we introduce **Round-Robin (RR)** scheduling. Basic idea: Instead of running a job to completion, RR runs a job for a **time slice** (or **scheduling quantum**) and then switches to the next job in the run queue.

The length of time-slice is obviously a multiple of timer-interupt.

The length of time slice is critical for RR. The shorter it is, the better performance of RR under response-time metric. However, too small time-slice can result in time taken to context switch to dominate overall performance. Thus, time slices hould be made long enough to **amortize** the cost of switching without making it so long that system is no longer responsive.

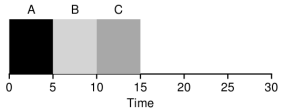

Figure 2.6: SJF (Bad response time)

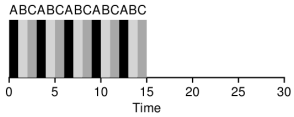

Figure 2.7: RR (Good response time)

RR is one of the *worst* policies for turnaround time because it extends the program to as long as possible.

Incorporating I/O

We now relax assumption that there is no I/O. When a job initiates an I/O request, because the currently-running job won't be using the CPU during the I/O; it is **blocked** waiting for the I/O completion. The CPU is idle during the time of the I/O.

The scheduler incorporates I/O by treating each CPU burst as a job. the scheduler makes sure processes that are "interactive" get run frequently. While those interactive jobs are performing I/O, other CPU-intensive jobs run, thus better utilizing the processor.

2.3.5 The Multi-Level Feedback Queue

In this subsection, we remove the assumption that we know job time and we will describe the most well-known approach to scheduling, **MLFQ**.

> **The Crus: How To Schedule Without Perfect Knowledge?**
>
> How can we design a schedular that both minimizes response time for interactive jobs while also minimizing turnaround time without a *prior* knowledge of job length?

MLFQ has a number of distinct **queues**, each assigned a different **priority level**. MLFQ uses priorities to decide which job should run at a given time.

Basic rules for MLFQ:

1. If Priority(A) > Priority(B), A runs (B doesn't)
2. If Prority(A) = Priority(B), A & B run in RR.

Rather than giving a fixed priority to each job, MLFQ *varies* the priority of a job based on its *observed behavior*.

If a job repeatedly relinquishes the CPU while waiting for input from the keyboard, MLFQ will keep its priority high, as this is how an interactive process might behave. If a job uses the CPU intensively for long periods of time, MLFQ will reduce its priority. In this way, MLFQ will try to *learn* about processes as they run, and thus use the *history* of the job to predict its *future* behavior.

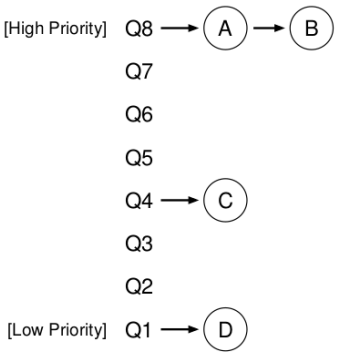

Figure 2.8: MLFQ Example

Attempt #1: How to Change Priority

Keep in mind: Our workload contains a mix of interactive jobs taht are short running (and may frequently relinquish the CPU), and some longer running "CPU-bound" jobs that need a lot of CPU time but where the response time isn't important. Here is the first attempt at an algoritm:

3. When a job enters the system, it is placed at highest priority (the topmost queue)
4. - If a job uses up an entire time slice while running, its priority is *reduced*.
   - If a job gives up the CPU before the time slice is up, it staus at the *same* priority level.

2.3.6 Problems with current MLFQ

There is a problem of **starvation:** if there are "too many" interactive jobs in the system, they will combine to consume *all* CPU time, and thus long-running jobs will *never* receive any CPU time.

Another problem is that the schedular can be **gamed**. Gaming a schedular refers to the idea to tricking the schedular in giving more time than the job's share. Before the time-slice is over, the job could issue and I/O and remain at the same priority. Doing so, allows the program to gain a higher percentage of CPU time.

2.3.7 Attemp #2: The Priority Boost

We add a new rule to handle the flaws.

5. After some time period S, move all the jobs in the system to the topmost queue.

This solves the problem of starvation. S has to be set properly.

2.3.8 Attempt #3: Better Accounting

We rewrite Rule 4 to make it anti-gaming.

4. Once a job uses its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced.

MLFQ: Summary

1. If Priority(A) > Priority(B), A runs (B doesn't)
2. If Priority(A) = Priority(b), A & B run in round-robin fashion using the time slice of the given queue.
3. When a job enters the system, it is placed at the highest priority.
4. Once a job uses up its time allotment at a given level, its priority is reduced.
5. After some period S, move all jobs to the topmost queue.