

CS1550 Fall 2015

Cyrus Ramavarapu

May 6, 2016

Contents

1	Overview	2
1.1	Operating Systems Manage Resources and Abstract Details	2
2	Basics and Terminology	3
2.1	How to support multiprogramming	3
2.2	A Process's Address Space	3
2.3	OS Design Schemes	5
2.3.1	Monolithic OS	5
2.3.2	Microkernel OS	5
2.3.3	Which is better?	5
2.3.4	Virtual Machines	5
3	Processes	6
3.1	Handling Processes	6
3.2	How to choose a process	7
3.3	User Threads and Kernel Threads	7
4	Scheduling	9
4.1	How to pick?	9
4.1.1	When to schedule?	10
4.1.2	Where to schedule?	10
4.1.3	Classes of scheduling	10
4.2	Scheduling Algorithms	10
4.2.1	Metrics	10
4.2.2	Algorithm 1: First Come, First Serve	10

Chapter 1

Overview

A brief overview of the course.

1.1 Operating Systems Manage Resources and Abstract Details

Resources: CPU Time, Memory, I/O Devices, Security

- Operating system needs its own resources to make decisions
- A layered structure between request and resources

Detail Abstraction: Sharing

- device context and method calling
- unified interface for application devices
- exclusive access 1 process on a constrained computer (virtual memory)

Different types of OS open up choices in scheduling algorithms, etc depending on load, tasks, resources

- Mainframes
- Realtime Has deadlines on tasks
 - **Soft** - (Can miss dealines) A dvd player and its FPS
 - **Hard** - if a deadline is missed, might as well have not tried (nuclear plant auto-pilot)
- Embedded A Car, A linking library to abstract I/O
- Server Linux is still a server class OS

We will look at medium sized systems since they are constrained enough so we can't be naive, but they have enough resources so that we can share.

In computer science, often the same problem will be solved, historically, twice - Paradigm shifts

Computer time is expensive

Von Neumann Architecture - data and code occupied a unified memory.

We will start with so few resources that we can't share.

Chapter 2

Basics and Terminology

2.1 How to support multiprogramming

Can I divide RAM?

- A process could access an address that is not in its RAM block
- Protect a block and check?

Where is the best place to do checks? **Hardware can help perform OS tasks** Do I pack tightly in memory? Many Processes but there are issues with **dynamic allocation**

2.2 A Process's Address Space

INSERT FIGURE OF ADDRESS SPACE

All address in range are your process's. None belong to another process.

A large region to support dynamic allocation.

Exclusive Access (This is not real)

- Does not make sense for same reason as dividing RAM – we don't have enough resources
- **The lie of virtual memory**

The OS is a layer between user and resources. Insert CS0449 Diagram. We make system calls from userspace to kernel space. All requests for a resources must go through the OS – you should **not** be able to side step OS.

- Hardware provides us with a Partitioned Instruction Set (Some for Userspace, Some for Kernel Space)
- Some instructions are safe (add int)
- Some instructions are priveleged (kernel mode) – Idea of a **mode bit**

If you try a priveleged instruction in user mode, an **exception** is raised and OS sends a **signal** and terminates the process.

Syscalls are mechanistically different than function calls – mode changes. Can't jump and link to our different address space.

A syscall is an interrupt. In x86 you put a trap in eax. These are **software interrupts**. Later we will talk about **hardware interrupts**.

Interrupt Vector – Indexed by ints. When interrupt occurs, goes to correct index. Gets address (to syscall handler?)

Syscall table – grab new address for code of the syscall

OS only needs priveleged mode to change machine state. Not everything in OS is priveleged.

The OS is not the same as other processe. It does not compete for CPU time. It does not schedule itself. It is basically **pure overhead**.

The OS does not need to exist, but we are afraid a process may misbehave. As a result, the OS exists out of practical necessity. We don't really want this, but we have code (OS) and it needs resources. However, the OS only runs when it needs to: Reacting to events. This takes time.

Think back to CS0447 and assembly. On a function call, everything had to be returned to the original state. A clean up needed to be done. Similarly, the OS needs to make room for its code. The **caller context** state will have to be saved and restarted. **A context switch**.

A syscall does not save context. **The OS does it before the syscall**. Context will be saved it to RAM and put at the top of the caller's stack.

- Safe
 1. Code was interrupted (caller) – can't execute until OS returns
 2. RAM is a shared resource as a whole – address space abstraction, pieces of memory are mine

We believe that a single context switch is **optimized** (from a hardware/software end). Only way to go faster is to have fewer context switches. If we have two solutions and one uses **fewer** context switches, we will say it goes **faster**.

Resources to protect and share:

- CPU Time – Preemption
- Memory – Virtual Memory
- I/O – Spooling
- Security – *'Tis black magic...* (Cyrus)

Memory trade off – cost, speed, capacity

There are also hardware interrupts. However the actual action is that of a software interrupt. Think about a **bus signal**. It has some basic steps that allow the OS to react.

2.3 OS Design Schemes

There are two big types of OS Designs: **Monolithic** and **Microkernel**

2.3.1 Monolithic OS

INSERT FIGURE HERE

Think about the OS as another application which controls everything. **Monolithic Design** is how we normally write an application. In this design, the OS is privileged. Consider scheduling: need a data structure, scheduling algorithm. This is a lot of code. In a monolithic design all of this can be done **without privilege!** Privileged instructions came when you make context switch, set up memory space, etc. This low level state is not doable by unprivileged instructions.

In a Monolithic OS:

- Code to maintain scheduler code
- Privileged code to do context work

All of this bundled together in a monolithic OS.

2.3.2 Microkernel OS

INSERT FIGURE HERE

In comparison, a microkernel OS strives to pare down OS size by extracting unprivileged code into separate processes. **Servers** communicate with the microkernel to get right answer. The microkernel then goes back and acts on it.

2.3.3 Which is better?

It depends.

- Context Switches: The microkernel makes more context switches. The monolithic kernel only needs to make 2.
- Code surface: The microkernel has less code – smaller attack surface. Also less code is easier to validate. Therefore system wide effects are less likely.
- Crashing: When a crash occurs the OS runs last. In a microkernel if a user server crashes, just need to pick another server.
- Speed: **A monolithic kernel is FAST**

Linux is a monolithic kernel. Windows (NT Line) is a sort of microkernel/monolithic hybrid.

2.3.4 Virtual Machines

Java has JVM between application and OS. Comparatively, VMware runs as a guest OS (A different architecture). Between hardware - hypervisor. In all cases there is some notion of **resource management**. Virtual machines are not a new idea. They grant exclusive access and as a result came back because they could. We now have a lot of resources and can run many more systems. Think about a webserver.

Chapter 3

Processes

3.1 Handling Processes

Multiprogramming - a single program will not saturate a CPU. The OS needs to decide which process to run. The CPU seems to only get a continuous stream of instructions. INSERT FIGURE. Every time we stop a process we introduce work. We can do this work during I/O.

RAM does not get faster with more transistors. The improvement is linear in speed; however, RAM capacity grows exponentially.

Ahmdahl's Law – Diminishing returns in increasing speed. I/O is idle process time.

The hope is to interleave the waits of a process with the runs of another.

Process - a running program and its associated data.

Ready - A process is ready if it has everything it needs to run except CPU time.

Pseudo-Parallelism - Juggling processes. Blocked means you are just a choice for scheduling.

In a process life-cycle, a process can leave ready state by calling **exit()**. Processor time is fast in compared to I/O time. OS can block (A program is blocked waiting for I/O). Since the program is not ready, it is not a candidate for scheduling. OS can instead run another process. Hardware interrupt will tell OS that a blocked process got resources. The process then becomes ready.

Batch System - The only way to stop a process is to **exit()**.

If you choose to block, you are vulnerable to programs that neither exit or block. Consider the following: *jump: jmp jump* (infinite loop that does nothing). Greedy process with CPU time.

Remember: The OS is not a ready process that is scheduled. A greedy process starves even the OS from CPU time, but this is tied to the greedy process.

One solution is to create an even in the long program – a **Syscall**. Assumes programmer did this (**yield()** syscall for example). This gives us a **Cooperative multitasking system** which is not the ultimate goal. A lot depends on how the process is written.

We can't do this in code – do this in **hardware internal clock**. A hardware interrupt due to a time.

Taking away a resource is called preemption. This gives us a **Preemptive Multitasking System**.

3.2 How to choose a process

Table or Linked List containing:

- | | | |
|---|---|--|
| <ul style="list-style-type: none"> • Process Management <ul style="list-style-type: none"> – State – Priority – PID – PPID – signal handlers – stats (start time/total CPU usage) | <ul style="list-style-type: none"> • File Management <ul style="list-style-type: none"> – File descriptor – Root directory – Current Working Directory – UID – GID | <ul style="list-style-type: none"> • Memory Management <ul style="list-style-type: none"> – Page table pointer – Pointers to text on text stack data |
|---|---|--|

Thread – a stream of instructions and associated state. A thread is different than a process if we have more than one of them. Tasks can communicate between multiple separate threads (different processes) via traditional I/O OS.

3.3 User Threads and Kernel Threads

pthread abstracts system threading implementation.

Kernel Threading – OS knows about threads.

User Threading – OS knows nothing about threads and only has a process table.

User threading depends upon library – **This is not pthreads**. Pthreads exists regardless of user/kernel threading.

If kernel thread: pthread.create() is a syscall. If user threading: library that is linked against

Kernel threading costs context switches – therefore user threading may be faster.

Synchronization is not the problem right now

Issues with preemption:

- in kernel threading we can switch to a new thread or process (schedule)
- in user threading schedule can only choose between processes (the greedy thread does not know of preemption – kernel would have to do it and kernel can't see it). A process can yield(). It only hurts one process (The one containing the greedy thread)
- in kernel threading pthread.yield() maybe a NOOP

In user threading we can customize scheduling decisions.

User Threading - a thread calls scanf and the whole process computation is blocked. However, in kernel threading other threads can keep working.

There is the nonblocking I/O or syscall. It immediately returns and more than likely your data is not ready.

In the user threading library, scanf can make a different syscall of the nonblocking type. The behavior of this syscall is along the lines of "try again or try again and I'll get it. This can be in the library. - the thread table is there so we can simulate "blocking" and switch to a different thread.

Examples:

Select() on a file descriptor, checks if the data is ready.

A videogame is drawing state with select and read, in a loop.

Select is a unix syscall. In linux there is poll() and epoll().

Between User threading and Kernel threading try to hybridize. **Scheduler Activation** (upcall) is pure hybrid because the OS is told what to do by process.

You need to have kernel threading and link user threading on top (Another hybrid aproach).

More Syscalls? Linux 2.6 had faster kernel threading than Linux 2.4 – due to a bad implementation.

Chapter 4

Scheduling

4.1 How to pick?

Scheduling is the process of choosing which of the **READY** processes/threads get to run next.

We can use the CPU intensely or be more I/O bound as a process:

- A CPU intensive process is **cpu bound**
- An I/O bound process will do a burst of CPU work and then I/O

Note: Both types of processes can have the same **wall clock run time**

If two process are CPU bound back to back, it will take $2t$ where t is 1 process time. It is hard/impractical to interleave two CPU bound processes because switching processes is not free. If you don't switch properly you lose illusion of independent program.

A CPU bound program will likely get pre-empted often. This leads to losing a processor's time due to the switch. Running two CPU intensive programs at the "same" time may not be better than sequentially running. As mentioned above, it is not easy to schedule a bunch of CPU bound processes together.

In a preemptive system, a process can get preempted and end up exiting (unlikely) or being blocked (I/O bound block a lot).

If the system is careful with the preemption timer, an I/O bound process may never see preemption – it will automatically block because it has to do I/O.

Can overlap CPU bursts and I/O blocks of two processes.
It only makes sense if I/O bound processes are much greater in number than CPU bound processes. **Reasonable: interactive programs tend to be I/O bound.阿姆达尔定律 (Amdahl's Law).**

Over time CPUs increase exponentially in speed (See Moore's law..although the quantum limit is rapidly being reached). On the other hand, I/O speeds grow linearly. As a result, a CPU bound program will in "time" become an I/O bound process because we gain the ability to do a lot of computation faster than we can read from a disk.

4.1.1 When to schedule?

- Software
 - Process Creation
 - Process Exit
 - Blocked
- Hardware Interrupts
 - I/O interrupt
 - clock interrupt

4.1.2 Where to schedule?

We assume a **Von Neumann Architecture** where a process can be prevented to run by denying it RAM. Once a process is in RAM, CPU scheduler picks a process.

4.1.3 Classes of scheduling

Admission Schedule selects jobs to go into RAM. In an interactive system, this might be the user.

If admission scheduler did not do a great job, there is also a **Memory Scheduler** which can kick out a job from RAM. The problem is that a process may have made progress before the memory scheduler gets to it. The process has state that has to be saved. Therefore the memory scheduler will do a "temporary" eviction and hold the state of the process on disk. When system is better, the memory scheduler can bring process and its state back into RAM.

CPU scheduling – Batch scheduling can be used for non-interactive systems for jobs that can run overnight. (I like to think of scheduling jobs running on a cluster - Cyrus). This type of system can be preemptive.

4.2 Scheduling Algorithms

4.2.1 Metrics

Throughput – *Number of Jobs/Unit Time* or in Frequency (Hz)

Turnaround Time – Time from job submission to job completion. This is **not** execution time! The execution time is the time a process has **available** to run.

Average Turn Around Time – Average of all turn around times for a set of jobs.

Fairness – comparable processes get comparable service. Of course, comparable is not really defined. You can be egalitarian and say all processes are created equal...but are they?

Big-O – Asymptotic worst case run time. The key is that it is **Asymptotic**.

Implementation difficulty – we like easy things even if they are only close to perfect. Easy is also relative and hard to define.

4.2.2 Algorithm 1: First Come, First Serve

Queue	4	3	6	3	Runtime
	A	B	C	D	Process

After FCFS (first come, first serve):

Queue	4	3	6	3	Runtime
	A	B	C	D	Process

Such change, much wow! Also this algorithm is $\mathcal{O}(1)$ – so yay!

Throughput: $4 \text{ jobs}/16 \text{ time} = 1/4$

Average turn around: $40/4 = 10$

Process	Completion	Arrival	Δt
A	4	0	4
B	7	0	7
A	13	0	13
D	16	0	16
Sum			40

No matter what is the schedule, the through put is the same. We don't consider reality and delays.