UNIVERSITY OF LEEDS

COMP2211 Operating Systems

Combined slides

Mantas Mikaitis

School of Computer Science, University of Leeds, Leeds, UK

Semester 1, 2024/25

Objectives

- To introduce the structure of COMP2211.
- Talk about the coursework and exam.
- Describe organisation of a computer system and interrupts.
- Discuss the components in a modern multiprocessor system.
- Introduce user mode and kernel mode.

Part I: Introduction to the Module

Structure of COMP2211: Staff

- Mantas Mikaitis (me)
- Tom Richardson

Structure of COMP2211: Lectures

Week	Topic
1 (current)	Introduction to OS
2	OS services
3	Processes
4	Xv6: Live coding and Q&A from the xv6 book
5	**Reading week. No scheduled labs or lectures**
6	Threads and concurrency. **Assignment due this week.**
7	Scheduling
8	Process synchronisation
9	Memory management
10	Catch up
11	Module review

Structure of COMP2211: Times and Places

- Lectures Mon @ 2pm, Thu @ 12pm in **Michael Sadler RBLT (LG.X04)**.
- Labs Mon-Tue-Thu-Fri in **Bragg 2.05**.

$\boxed{\text{You will find one of the lab slots in your timetable.}}$

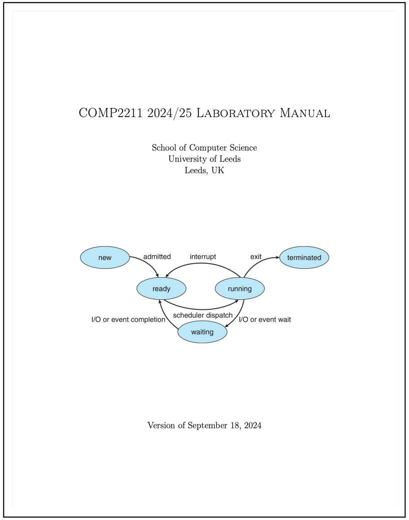
Structure of COMP2211: Reading List

We will be mainly using the *Operating System Concepts* (OSC) 10th ed., 2018, and the 3rd edition of the xv6 book (XV6), 2022. These slides are based on OSC (see the reference list).

Week	Reading materials
1 (current)	Chapter 1 OSC. Chapter 1 XV6.
2	Chapter 2 OSC. Chapter 2 XV6.
3	Chapter 3 OSC.
4	Reread Chapters 1–2 XV6.
5	Reread Chapters 1–3 OSC.
6	Chapter 4 OSC.
7	Chapter 5 OSC.
8	Chapter 6 OSC.
9	Chapters 9–10 OSC. Chapter 3 XV6.
10	Reread Chapters 4–6 OSC.
11	Reread Chapters 9–10 OSC.

Structure of COMP2211: Laboratories

- Laboratories will be in C and Bash.
- Download the lab manual from Minerva.
- Weeks 1–3 contain introduction to xv6 operating system.
- Weeks 4–6: you will work on a 40% assignment.
- Weeks 7–11 contain further formative assessment exercises.



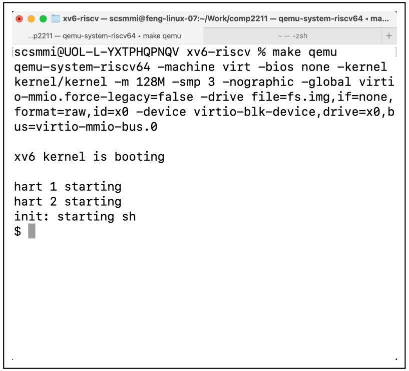
COMP2211 2024/25 Laboratory Manual

School of Computer Science
University of Leeds
Leeds, UK

Version of September 18, 2024

Structure of COMP2211: Xv6 operating system

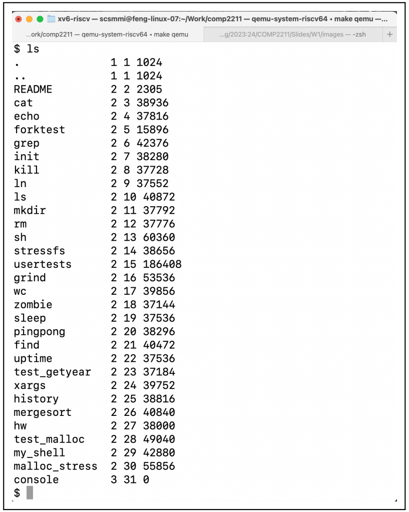
- XV6 is a small teaching operating system from MIT.
- The source code is readable and editable.
- It runs on RISC-V architectures.
- To run it on Intel/AMD CPUs we emulate RISC-V with **qemu**.
- It is written in C (94.4%) with some RISC-V assembly (3.4%).


xv6-riscv — scsmmi@feng-linux-07:~/Work/comp2211 — qemu-system-riscv64 ◂ ma...
...p2211 — qemu-system-riscv64 ◂ make qemu
~ — -zsh

```
scsmmi@UOL-L-YXTPHQPNQV xv6-riscv % make qemu
qemu-system-riscv64 -machine virt -bios none -kernel
kernel/kernel -m 128M -smp 3 -nographic -global virti
o-mmio.force-legacy=false -drive file=fs.img,if=none,
format=raw,id=x0 -device virtio-blk-device,drive=x0,b
us=virtio-mmio-bus.0

xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$
```

Structure of COMP2211: Xv6 operating system

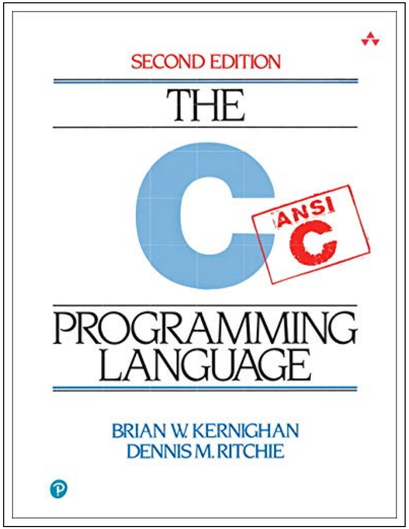
- We use it by entering commands, similarly as with the Unix machines in the 2.05 lab, but more limited.
- The command line interpreter recognizes **bash commands**.
- In the labs you will extend your copy of xv6 to have more commands.


xv6-riscv — scsmmi@feng-linux-07:~/Work/comp2211 — qemu-system-riscv64 ◂ make qemu —...
...ork/comp2211 — qemu-system-riscv64 ◂ make qemu
...g/2023:24/COMP2211/Slides/W1/images — -zsh

```
$ ls
.              1 1 1024
..             1 1 1024
README         2 2 2305
cat            2 3 38936
echo           2 4 37816
forktest       2 5 15896
grep           2 6 42376
init           2 7 38280
kill           2 8 37728
ln             2 9 37552
ls             2 10 40872
mkdir          2 11 37792
rm             2 12 37776
sh             2 13 60360
stressfs       2 14 38656
usertests      2 15 186408
grind          2 16 53536
wc             2 17 39856
zombie         2 18 37144
sleep          2 19 37536
pingpong       2 20 38296
find           2 21 40472
uptime         2 22 37536
test_getyear   2 23 37184
xargs          2 24 39752
history        2 25 38816
mergesort      2 26 40840
hw             2 27 38000
test_malloc    2 28 49040
my_shell       2 29 42880
malloc_stress  2 30 55856
console        3 31 0
$
```

Structure of COMP2211: Programming languages

- Mainly C.
- A lot of character handling.
- Pointers and double pointers.
- Arrays of characters and strings.
- Dynamic memory allocation.
- Xv6 specific process creation and command execution.



Structure of COMP2211: Vevox

We will be using Vevox in the lectures a lot, to do quizzes.

Let's try it.

Structure of COMP2211: Assessment

Deadline 2pm Nov. 7 (W6): 40% module mark

Task: write your own command line interpreter (Shell) for xv6 that can perform various commands, such as `ls` or `cd`, redirect standard output to files, and other advanced features.

- The formative exercises on weeks 1–3 and reading of Chapters 1–2 of the xv6 book are essential for this assignment.
- The assignment will be automatically marked on Gradescope by running a set of commands and checking whether the output from the submitted shell is as expected.

January paper-based 2-hour exam: 60% module mark

Reading lecture slides and OSC is essential to succeed. Engaging with laboratory material can also help mastering the learning outcomes.

Structure of COMP2211: Communication and feedback

- Please email me with your questions.
- Ask staff in the lab, who can forward me questions/comments without revealing your identity.
- Feedback: informal mid-module and formal end-of-module surveys.

Feedback welcome

Feel free to leave me feedback after lectures and labs and I will try to implement changes as we go along. For example, tell me: present slower, explain a specific topic again or differently, supply slides in different colour theme, do less/more quizzes, discuss this piece of code in class, ...

Structure of COMP2211: Connection between labs and lectures?

- Lectures cover general OS concepts.
- Laboratories focus on xv6, which has used some of those concepts.
- Do not look for every concept from the lectures to be in xv6.
- You will notice the theory being of use in your further studies, interviews, placements, graduate roles, other modules, ...

Structure of COMP2211: What is in the exam?

All topics addressed in the lectures can appear in the January exam. Material appearing in the lectures is examinable based on the OSC contents of appropriate Chapters/Sections.

Structure of COMP2211: Quiz (5 minutes)

Part II: Introduction to Operating Systems

Operating Systems: Main Definitions

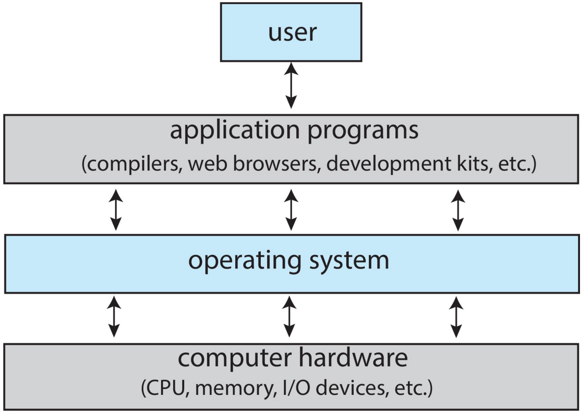
A **computer system** can be divided into four components:

- Hardware
- Operating system
- Application programs
- User

OS is a resource allocator

Hardware (CPU, memory, mouse, keyboard, ...) are resources. Multiple applications running on the system compete for them. Operating System coordinates hardware use among users and applications.

Operating Systems: Main Definitions


user
application programs
(compilers, web browsers, development kits, etc.)
operating system
computer hardware
(CPU, memory, I/O devices, etc.)

Operating Systems: Main Definitions

User view:

- Laptop or PC that consists of monitor, keyboard, mouse.
- One user that wants to use all of the resources.
- OS designed for **ease of use** rather than **resource utilization**.
- Many users interact with mobile devices: touch screen, voice recognization.

Embedded systems

Some computers have little or no user view: home appliances, various devices in cars, and other specialized computers that almost work on their own.

Operating Systems: Main Definitions

System view:

- Resource allocator, involved with hardware intimately.
- Manages CPU time, memory space, storage space, I/O access.
- Faces several requests—has to decide who gets the resources and who waits (users, applications).
- Responsible for overall efficient operation of the system.

Control program

Different view of OS. Manages the control of programs to prevent errors and improper use of the hardware.

Operating Systems: Main Definitions

Operating systems arose due to the growth of complexity of computer hardware.

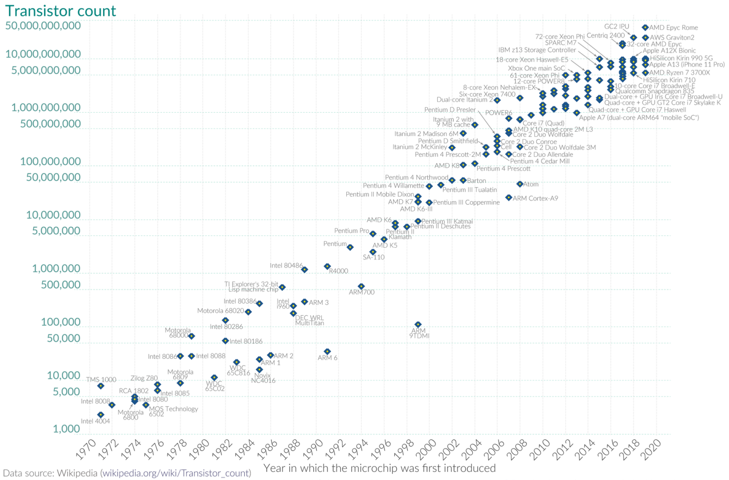
Moore's Law correctly predicted in the 1960s that the number of transistors on an integrated circuit would double every 18 months.

The size shrank and the functionality has grown—now the uses are very varied and OS is essential to manage the complexity.

Operating Systems: Main Definitions

Moore's Law: The number of transistors on microchips doubles every two years

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important for other aspects of technological progress in computing – such as processing speed or the price of computers.



Data source: Wikipedia (wikipedia.org/wiki/Transistor_count)
OurWorldinData.org – Research and data to make progress against the world's largest problems.

Operating Systems: Main Definitions

A common definition of operating systems is that it is the one program that always runs, a **kernel**. Alongside are system programs, associated with OS but not part of kernel, and **application programs**.

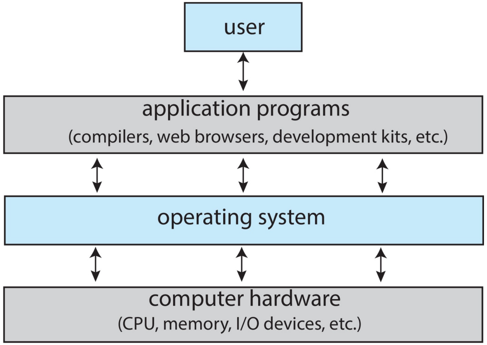
Novadays OS includes many things outside the immediate definition of what the OS does: browsers, photo viewers, word processors, ...

Operating System

- A **kernel** (always running).
- **Middleware** frameworks that allow development of applications.
- **System programs** that aid in various OS tasks.

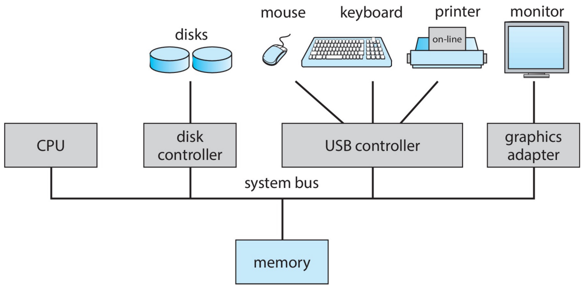
Structure of COMP2211: Discussion with peers (5 minutes)

Consider the main components of a computer system below, again. Discuss with your peers how each of those would look in a **washing machine system**: User interaction? Applications? OS? Hardware resources? Programming them?


user
application programs
(compilers, web browsers, development kits, etc.)
operating system
computer hardware
(CPU, memory, I/O devices, etc.)

Part III: Concept of Interrupts

Computer-System Organization



- Many devices competing for memory access.
- OS uses **device drivers** to talk to various controllers.
- Memory has a **memory controller** which also does some managing to keep up with many reads and writes at once.

$\boxed{\text{We now go deeper into various concepts within this system.}}$

Interrupts

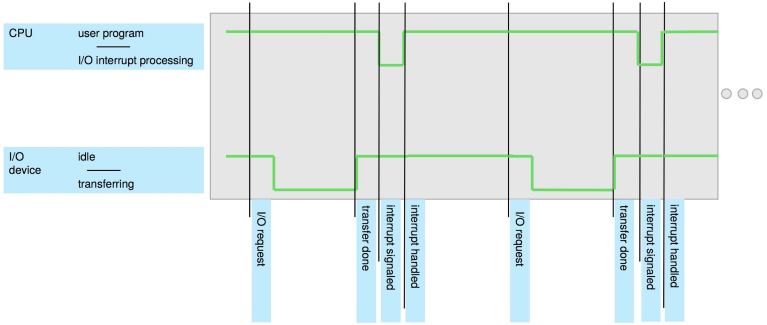
Consider a series of events within the system when we press a key on a keyboard. This constitutes input/output (I/O).

1. Device driver writes to appropriate registers (memory locations) in the device controller.
2. The controller reads to see what needs to be done (e.g. read a character from the keyboard).
3. Controller starts transfer of data from the keyboard.
4. Controller informs the driver that a transfer has been done.
5. Driver gives control to OS, to read the data.

How does the controller inform the driver (CPU) that it has finished an operation? Through an **interrupt**.

Interrupts

- Hardware may trigger interrupts at CPU at any time.
- CPU then stops what it is doing and checks if it can service the interrupt, through the **interrupt service routine**.
- When serviced, CPU goes back to what it was doing.



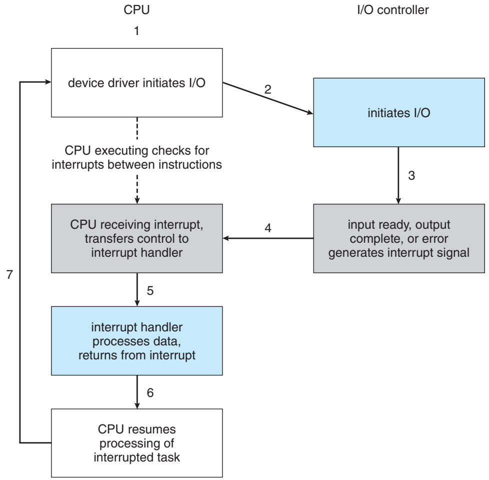
Interrupts

- CPU may need to store the current **program counter**, for example into the **link register** or **stack**—to get back to what it was doing before the interrupt.
- The interrupt routine has to save any CPU state that it will be changing, and return it back to what it was once finished.
- Once done, the program counter and the original program execution continue.

Program Counter (PC) register

Stores the address of the next instruction that the CPU will execute.

Interrupts


CPU
I/O controller
1
device driver initiates I/O
2
initiates I/O
CPU executing checks for interrupts between instructions
3
CPU receiving interrupt, transfers control to interrupt handler
4
input ready, output complete, or error generates interrupt signal
7
5
interrupt handler processes data, returns from interrupt
6
CPU resumes processing of interrupted task

Interrupts

Requirements for an effecting interrupt system:

- Capability to defer interrupts when something more important is being done on CPU.
- Efficient way to service interrupts—can't take too long to respond.
- Structure of **high and low priority interrupts**.

Interrupts: Advanced Material

- When interrupt occurs, correct interrupt service routine needs to be discovered.
- There can be hundreds to search through, but we need to be fast.
- Instead of searching, a table of pointers to interrupt service routines can be used: **interrupt vector**.
- A unique ID on interrupts is indexing this vector, which sends CPU straight to the correct interrupt service routine.

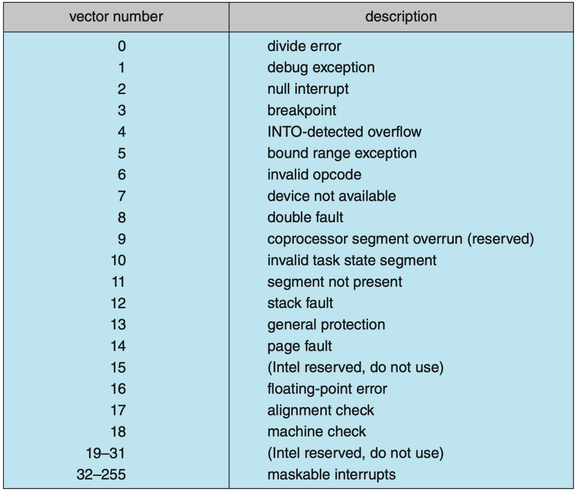
Interrupt state save and restore

Assume that an interrupt service routine for keyboard input interrupt is using CPU registers R1–R7 for internal operations. Before doing anything, R1 to R7 have to be stored in memory (pushed to stack), and once the interrupt is finished, they should be restored. If this is not done, the CPU will return to its previous state but will potentially crash because the carpet was moved from under its feet—the registers suddenly changed!

Interrupts: Advanced Material

- In reality, the vectors get too big and a hybrid approach is used, **interrupt chaining**.
- Interrupt routines point to the next interrupt routine: we loop through them until the right one is found.
- **Nonmaskable interrupts**: unrecoverable errors, such as memory read/write faults.
- **Maskable interrupts**: CPU can turn these off before starting some critical code section.

Interrupts: Intel processor event-vector table



vector number	description
0	divide error
1	debug exception
2	null interrupt
3	breakpoint
4	INTO-detected overflow
5	bound range exception
6	invalid opcode
7	device not available
8	double fault
9	coprocessor segment overrun (reserved)
10	invalid task state segment
11	segment not present
12	stack fault
13	general protection
14	page fault
15	(Intel reserved, do not use)
16	floating-point error
17	alignment check
18	machine check
19–31	(Intel reserved, do not use)
32–255	maskable interrupts

Interrupts: Common Terms

Raise an interrupt: ask CPU to stop what it is doing and do something for me.

Catch an interrupt: CPU discovers someone wants processing time.

Dispatch the interrupt: call **interrupt handler**.

Clear the interrupt: call the correct interrupt service routine.

Interrupts: Vevox quiz (5 minutes)

Part IV: Storage and Memory

Storage Structure

- CPU can only load instructions from memory.
- Programs therefore must be loaded into memory to run.
- Usually programs loaded to **random-access memory** (RAM).
- Computers use other memory as well. Since RAM is volatile (contents lost when power is off) we cannot trust it to hold for example the **bootstrap program**, which runs on power on.
- Read-only EEPROM memory—slow and rarely changed memory that preserves contents on power off.
- Iphones for example use EEPROM to store serial numbers and other hardware information.

Storage Structure: Reminder of Units

A **bit** is a basic unit of storage. A **byte** is 8 bits.

A **word** is one or more bytes, varies between computer architectures. Register width and instruction size usually constitutes how large is a word.

Kilobytes, megabytes, gigabytes, terabytes, petabytes, ...

Or in fact, correct International Organization for Standardization (ISO) binary prefixes adopted in 2008 are:

Kibibytes, mebibytes, gibibytes, tebibytes, pebibytes, ...

1024, 1024^3, ... bytes.

Storage Structure

- Memory is laid out in arrays of **bytes**, which have **addresses**.
- CPU interacts with memory by **load** and **store** instructions addressing specific bytes or words.
- Bytes or words are moved between the CPU registers and the memory.
- Similarly, CPU loads instructions from memory automatically, addressed by the **program counter**.

Storage Structure

- You may have heard about **von Neumann architecture**.
- **Instruction-execution cycle**: fetch instruction, execute, repeat.
- First CPU fetches an instruction from a program in memory, to an **instruction register**.
- Then it decodes the instruction and executes it in the hardware.
- The result may be stored back in memory.

Main memory

Ideally we would like the programs and data to be in fast RAM memory. This is not possible due to **volatility** of the memory and relatively small size.

Storage Structure: Secondary storage

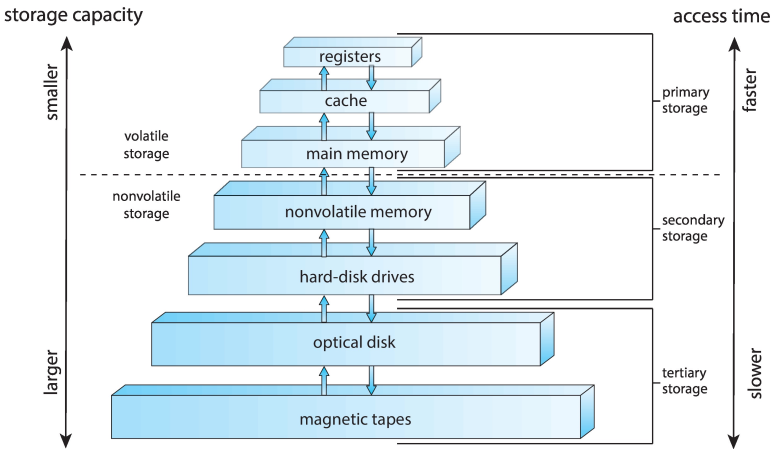
Data is stored in secondary storage, which preserves programs and data while the system is off.

- Hard-disk drives (HDD).
- Nonvolatile memory (NVM) devices.

Other storage

CDs, cache, Blu-ray disks, magnetic tapes, ...

Storage Structure: Hierarchy



Operating Systems have to balance all of these storage types for the whole system to work efficiently and reliably.

I/O Structure

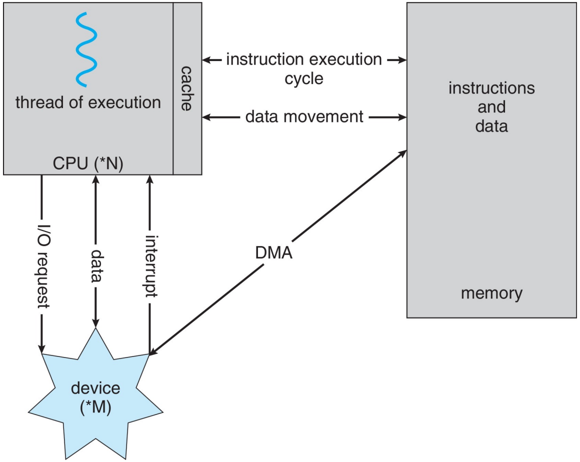
Interrupt driven memory access is fine for small requests, but moving a lot of data will not work very well.

Direct Memory Acces (DMA) is used to offload work from the CPU.

Device controller directly transfers data to and from the device and main memory, without holding the CPU while doing so.

One interrupt is generated per transfer, to tell the device driver that the operation has completed. Better than interrupt for every byte.

I/O Structure: Direct Memory Access



Part V: Single and Multi-processor Systems

Single-Processor Systems

Single processor containing one CPU with a single processing core - many years ago.

The **core** is the main piece of hardware within a CPU that executed program instructions and managed register storage locally.

General purpose or **domain specific**: can run general programs or can run a limited set of operations optimized for some task/s.

A computer system may have one general purpose single-processor CPU and multiple domain-specific processors that accelerate some specific tasks. From the perspective of OS, this system is still a single processor.

Multiprocessor Systems

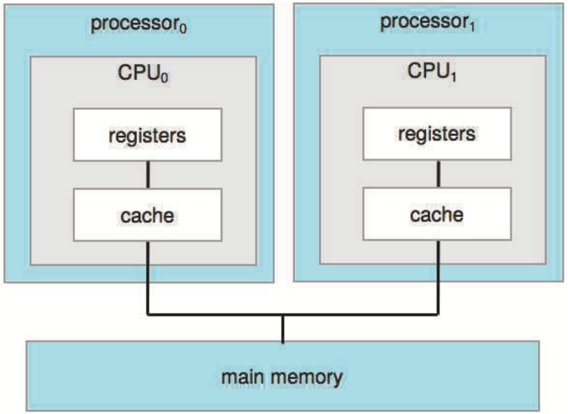
Multiprocessor systems dominate the computer landscape novadays.

Two or more processors, each with a single-core CPU.

The main goal is to increase **throughput**—how much work can we do in a certain amount of time.

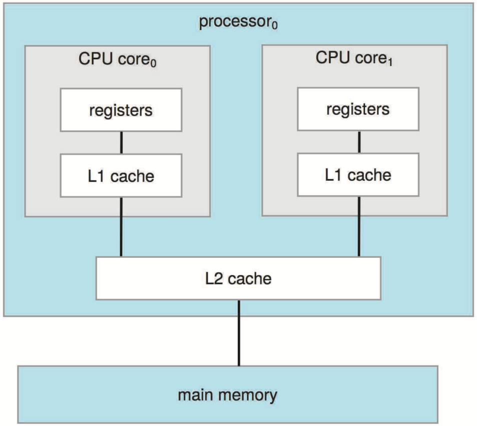
Ideally N processors should result in N times speed up. In reality it is less: there is some overhead in managing multiple processors that cooperate on some task. This overhead does not exist when only one processor is executing.

Symmetric multiprocessing (SMP)



Multiprocessor Systems

The definition gets more complicated today: **multicore** systems.



Multiprocessor Systems: Main Terms

- CPU — The hardware that executes instructions.
- Processor—A physical chip that contains one or more CPUs.
- Core — The basic computation unit of the CPU.
- Multicore — Including multiple computing cores on the same CPU.
- Multiprocessor — Including multiple processors.

Single/multiprocessing: Discussion with peers (5 minutes)

Find out how many processors and CPUs there are in your chosen personal device (laptop, mobile phone).

Discuss with your peers and compare.

At the end volunteers welcome to tell us the details about their CPU.

Does anyone in the room have a device with one CPU or even one core?

Part VI: Key Concepts for Operating Systems

Operating-System Operations

As noted earlier, **bootstrap program** is a key component that starts a computer:

- Stored in nonvolatile memory.
- Initializes CPU registers, device controllers, memory contents.
- Loads and starts executing the OS: locate the **kernel** and load into the main memory.

Once the kernel is loaded, it can start providing services to the system and its users.

System daemons also run "always", alongside the kernel, and provide various system services.

Once the kernel and daemons are running, the OS is waiting for I/O device requests and other tasks to do. It can sit quietly if nothing is happening.

Operating-System Operations

Back to interrupts:

- **Hardware interrupts**: we looked at these. I/O interrupts and other devices.
- **Trap interrupts**: software-generated interrupt: for example, division by zero or invalid memory address being accessed.

Multiprogramming

A single program cannot keep CPU or I/O devices busy at all times—the ability to run multiple programs and change between them is **multi programming**.

It increases CPU utilization by swapping which program in execution (a **process**) gets the CPU time.

When one process stops executing and starts waiting for I/O to finish, CPU is allocated to another process that is ready to run.

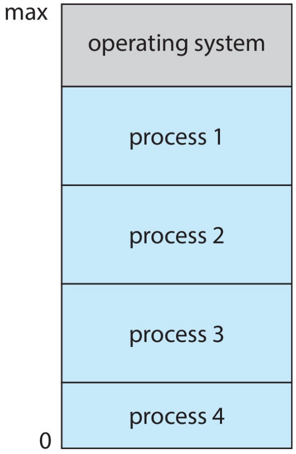
Multitasking

Similar to multiprogramming, but the switches between processes are very frequent to provide users with a fast **response time**.

Processes

Having multiple running programs, processes, requires some form of memory management. We also need a set of rules for deciding which process gets run (**scheduling**). Processes should also not interfere with other processes. These issues will be addressed in later lectures.

Multiprogramming: Memory Layout



User and Kernel Modes of Operation

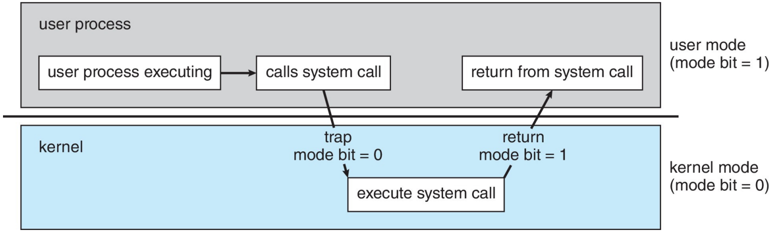
Main idea

Incorrect or malicious program should not be able to break the OS, execute code that belongs to OS services, or take over the hardware resources.

To avoid these problems, OS can execute code in **user mode** and in **kernel mode (also known as system/supervisor/privileged mode)**.

OS services and the kernel are executed in the system mode, while user programs are in user mode. Once program requests some important resources, it can go into the kernel mode for some specific tasks, **system calls**.

User and Kernel Modes of Operation



At system boot time, the hardware starts in kernel mode.

Also, on interrupts, the hardware switches to kernel mode.

In general, whenever OS gains control, we are in kernel mode.

Timer: Periodic Interrupts from OS

For the OS to maintain control over the CPU we need protection against user program getting stuck in infinite loop or similar.

Timer is set to interrupt the computer after a specified period.

Period can be fixed or variable.

OS sets up the timer before transferring control to user programs. When the timer interrupt occurs OS gets control and can decide whether to abort the program or let it run longer.

Instructions that set up the timer are **privileged instructions**—hardware operations that can only be executed in kernel mode.

Process Management

What is a process

A program is compiled and stored in the main memory, as a set of instructions. When the CPU is going through those instructions and executing them one by one, the program takes a form of a **running process**. Concept of processes is fundamental to OS resource management.

Example of processes: a compiler that is compiling some code; a word processor that has a document open; a social media app open on a smartphone.

Process Management

- Processes need resources: CPU, memory, I/O, files, initialization data.
- A program is not a process—it's a **passive entity**.
- Processes instead are **active entities**.
- A **single-threaded process** has one **program counter** specifying the next instruction to execute.
- Sequential execution: CPU executes instructions one at a time, until the process terminates.
- Two processes can be associated to the same program, but are considered separate entities, separate execution sequences.
- **Multithreaded processes** have multiple program counters—we will address **threads** later in the module.
- Typically many processes exist, some belong to OS executing in kernel mode, some to user, executing in user mode: OS multiplexes between these processes on single or multicore CPUs.

Process Management

OS undertakes following activities in relation to processes:

- Creating and deleting processes.
- Scheduling processes and threads on the CPUs.
- Suspending or resuming processes.
- Provide **process synchronization**—we address this later in the module.
- Provide ways for **process communication**—also later in the module.

We will get back to processes in Week 3.

Memory Management

- **Main memory** is central to the operation of a modern computer system.
- Main memory can be very large, holding thousands to billions of bytes, each addressed separately.
- CPU and I/O devices can target those bytes (read/write).
- Apart from registers and caches, main memory is the only other memory directly accessible by the CPU.
- For CPU to access other data, such as in various disks, it has to be transferred to the RAM first.
- Data and instructions therefore first travel to the RAM.

Memory Management: Program Execution

1. Load a program into the main memory and let CPU know the start address.
2. As program executes, instructions and data are accessed by addressing the main memory.
3. When a program terminates, space is freed and a new program may take its place in the main memory.
4. Several programs are usually in memory, which creates a need for **memory management**.

OS memory management

Keep track of used memory blocks and which process is using them. Allocate/deallocate memory space. Decide which processes to move in and out of memory. We will get back to this in Week 9.

You can notice the complexity of work of OS is growing.

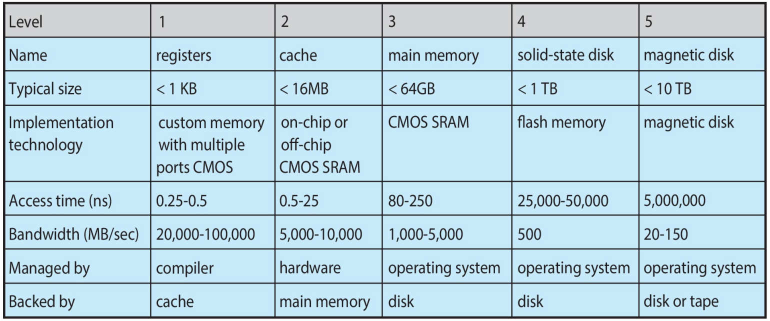
Cache Management

Caching

Caching is a technique used to speed-up access to commonly read/written information—the core idea is to copy blocks of information from slower to faster memory and then access it from that faster memory. This state is temporary and caching is happening very frequently at all levels: hardware, OS, software.

When we need a particular piece of data, we first check the cache. If found, we don't go to slower memory. Otherwise we copy the data from the RAM to the cache—assume it will be needed again soon.

Cache Management



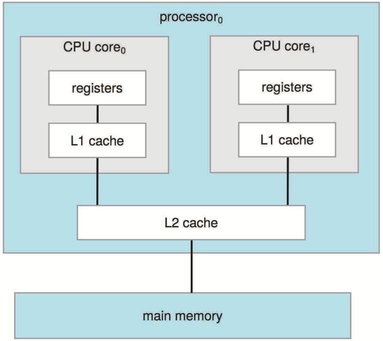
Level	1	2	3	4	5
Name	registers	cache	main memory	solid-state disk	magnetic disk
Typical size	< 1 KB	< 16MB	< 64GB	< 1 TB	< 10 TB
Implementation technology	custom memory with multiple ports CMOS	on-chip or off-chip CMOS SRAM	CMOS SRAM	flash memory	magnetic disk
Access time (ns)	0.25-0.5	0.5-25	80-250	25,000-50,000	5,000,000
Bandwidth (MB/sec)	20,000-100,000	5,000-10,000	1,000-5,000	500	20-150
Managed by	compiler	hardware	operating system	operating system	operating system
Backed by	cache	main memory	disk	disk	disk or tape

Cache is smaller than main memory. **Cache replacement policy** is an important consideration in OS and can increase performance significantly: what should we keep in cache and what should we move to memory?

Cache Coherence

Cache coherence

In multiprocessor environment, each processor may have a separate cache. If both contain the same memory copied from the RAM, and one of them updates it, what happens to the other? Cache coherency is needed to make sure the data is not outdated in one of the copies. In distributed systems problem severe: same data in different computers.



Security and Protection

- OS protects hardware and memory resources from what can be considered unauthorized access by processes and users.
- **Protection**: mechanism for controlling access to certain resources defined by the computer system.
- **Security**: defense of the system against internal and external attacks: denial-of-service, worms, viruses, identity theft, theft of service, ...

Operating System Security Measures

Keep track of user IDs; each user IDs has associated resources that they can access; group ID allow set of users to be assigned permissions.

Virtualization

Run another OS within the main OS, run applications on that **guest OS**.

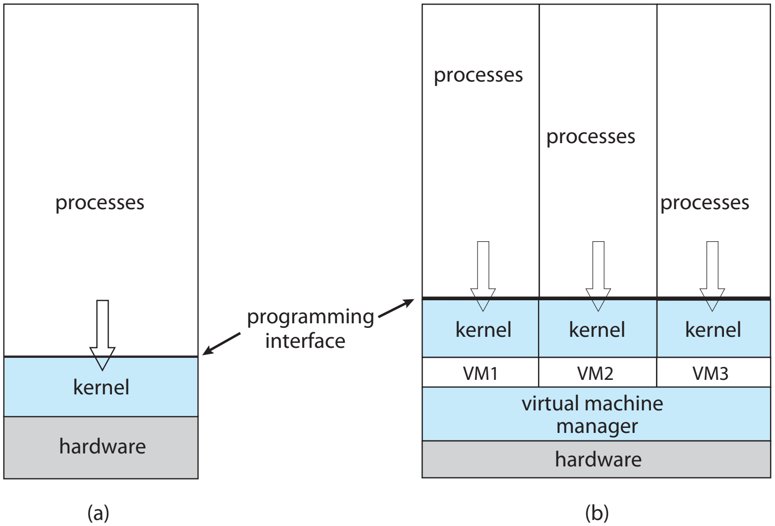
Emulation: guest OS compiled for a different hardware which has to be emulated to run on the hardware we have on our desk.

Virtualization: guest OS compiled for our hardware, and run **natively**, using that CPUs instruction set. This is faster than emulation, but limited.

XV6

In the labs we are emulating RISC-V architecture and running xv6 by translating RISC-V instructions that xv6 uses to Intel/AMD instructions which the lab computers understand.

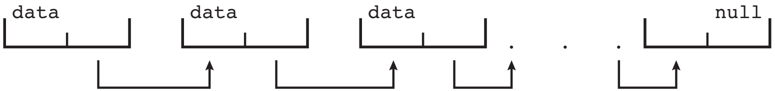
Virtualization



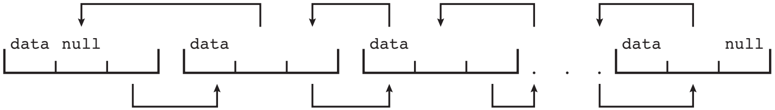
Kernel Data Structures

We finish this week with a look (a reminder?) of the various data structures that are used in the kernel to store and manage data.

Singly linked list:

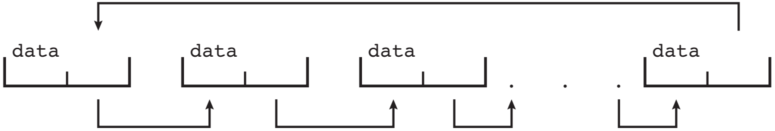


Doubly linked list:



Kernel Data Structures

Circularly linked list:



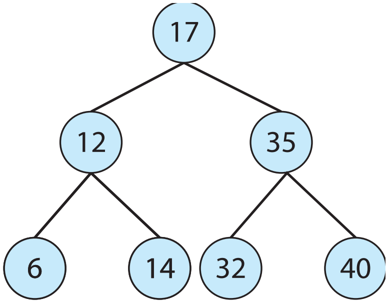
Lists of size *n* require at most *n* checks to find an item.

A **stack** is a common data structure in OS: last-in first-out (LIFO) structure which **pushes** things at the top and **pops** them from the top of the list. Interrupt routines push registers and pop them back to restore the previous state of the CPU.

Queue similarly uses first-in first-out idea (FIFO).

Kernel Data Structures

Trees introduce hierarchy: **parent-child structure** between data points.

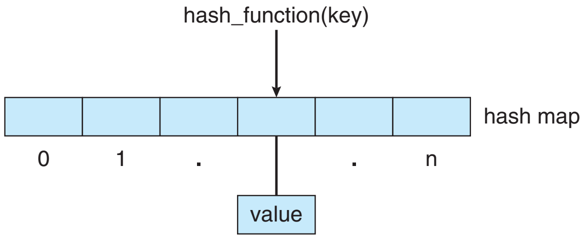


Tree search complexity

Unbalanced tree of n nodes can require up to n comparisons to find the data. Balanced tree can improve this by requiring $\log(n)$ (height of left and right subtrees differ by at most 1).

Kernel Data Structures

Hash maps can allow a search cost of at most 1.



Ideally, each unique key is mapped to unique value, which can be used as an index to a table containing data we want.

Hash collisions can occur: different keys map to the same value.

COMP2211: Progress

Week	Topic
1	~~Introduction to OS~~
2	OS services
3	Processes
4	Xv6: Live coding and Q&A from the xv6 book
5	**Reading week. No scheduled labs or lectures**
6	Threads and concurrency. **Assignment due this week.**
7	Scheduling
8	Process synchronisation
9	Memory management
10	Catch up
11	Module review

Progress

Week	Topic
1	Introduction to OS
2 (current)	OS services
3	Processes
4	Xv6: Live coding and Q&A from the xv6 book
5	**Reading week. No scheduled labs or lectures**
6	Threads and concurrency. **Assignment due this week.**
7	Scheduling
8	Process synchronisation
9	Memory management
10	Catch up
11	Module review

Structure of COMP2211: Reading List

We will be mainly using the *Operating System Concepts* (OSC) 10th ed., 2018, and the 3rd edition of the xv6 book (XV6), 2022. These slides are based on OSC (see the reference list).

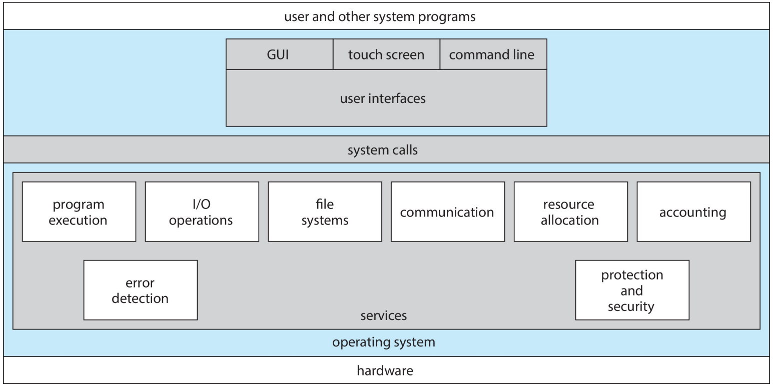
Week	Reading materials
1	Chapter 1 OSC. Chapter 1 XV6.
2 (current)	Chapter 2 OSC. Chapter 2 XV6.
3	Chapter 3 OSC.
4	Reread Chapters 1–2 XV6.
5	Reread Chapters 1–3 OSC.
6	Chapter 4 OSC.
7	Chapter 5 OSC.
8	Chapter 6 OSC.
9	Chapters 9–10 OSC. Chapter 3 XV6.
10	Reread Chapters 4–6 OSC.
11	Reread Chapters 9–10 OSC.

Objectives

- Identify services that OS provides.
- Discuss **system calls**.
- Compare monolithic, layered, microkernel, modular and hybrid approach to OS design.

Part I: Description of the Problem

Operating-System Services



Operating-System Services

Set of features helpful most directly to the user:

- **User interface (UI)**: graphical, touch-screen, command-line interface, ...
- **Program execution**: load programs from memory and run them; end execution.
- **I/O operations**: access data from files or I/O devices. For efficiency and protection, users cannot do so directly: OS services do that for us.
- **File-system manipulation**: read, write, create, delete, search files. Access permission control.
- **Communications**: Communicate between processes. **Shared memory** or **message passing** communication.
- **Error detection**: Errors occur in CPU, memory, I/O devices, user programs: OS has to detect, correct, report errors. Sometimes may decide to halt the system.

Operating-System Services

Other features are mainly for the operation of the system:

- **Resource allocation**: Multiple processes are running on the system and they should be allocated resources: CPU cycles, main memory, file storage, I/O device access.
- **Logging**: Keep track of which programs what resources.
- **Protection and security**: Protect information between different users on the system. Make sure processes do not interfere. Control resource access. Security from outside: control access to the system.

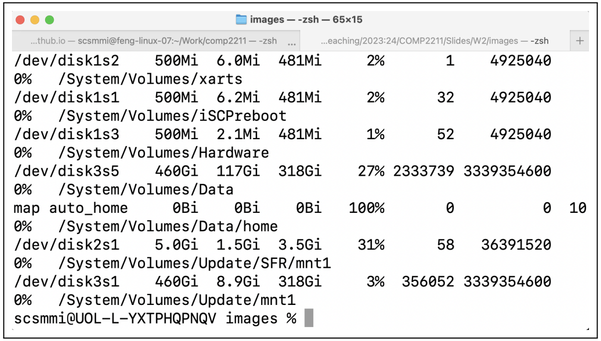
User and OS Interface: Command Interpreters

Command Interpreters

On UNIX and Linux systems multiple options: C shell, Bourne-Again shell, Korn shell. The main function is to execute user supplied commands, which usually modify files on the system.

The commands can be built into the shell or they can be a separately stored executable which the shell can invoke. The latter requires no modification to the shell when adding new commands.

User and OS Interface: Command Interpreters


images — -zsh — 65×15
...thub.io — scsmmi@feng-linux-07:~/Work/comp2211 — -zsh
...eaching/2023:24/COMP2211/Slides/W2/images — -zsh

```
/dev/disk1s2     500Mi  6.0Mi  481Mi     2%        1     4925040
0%   /System/Volumes/xarts
/dev/disk1s1     500Mi  6.2Mi  481Mi     2%       32     4925040
0%   /System/Volumes/iSCPreboot
/dev/disk1s3     500Mi  2.1Mi  481Mi     1%       52     4925040
0%   /System/Volumes/Hardware
/dev/disk3s5     460Gi  117Gi  318Gi    27% 2333739 3339354600
0%   /System/Volumes/Data
map auto_home      0Bi    0Bi    0Bi   100%        0           0  10
0%   /System/Volumes/Data/home
/dev/disk2s1     5.0Gi  1.5Gi  3.5Gi    31%       58    36391520
0%   /System/Volumes/Update/SFR/mnt1
/dev/disk3s1     460Gi  8.9Gi  318Gi     3%  356052 3339354600
0%   /System/Volumes/Update/mnt1
scsmmi@UOL-L-YXTPHQPNQV images %
```

User and OS Interface: Graphical User Interface

Instead of entering commands directly, we could use a **Graphical User Interface**—a mouse-based window-and-menu interface.

Users move mouse and click on images that represent files, executables, directories, to interact with them.

First GUI appeared in 1973.

Apple made GUI (desktop) widespread in the 1980s.

On UNIX systems traditionally command line dominated, but open-source GUIs exist: KDE, GNOME, ...

Touchscreen is a form of GUI common on mobile devices.

User and OS Interface: When is Command Line Interface better?

- Command-line is usually faster, but requires specialized knowledge.
- **System administrators** for example would choose command line over a GUI for most tasks.
- Not everything is available in GUI—specialized commands only accessed through CLI.
- Easier to do repetitive tasks—commands can be recorded in a file and easily rerun (**shell scripts).**

Vevox quiz

Part II: System Calls

System Calls

System calls: a well-defined interface to the services of an operating system, used by programmers and users.

Usually written in C or C++, but assembly also used.

Consider an example task of reading a file and writing its contents to another file. As a UNIX command it may look like this:

```
cp in.txt out.txt
```

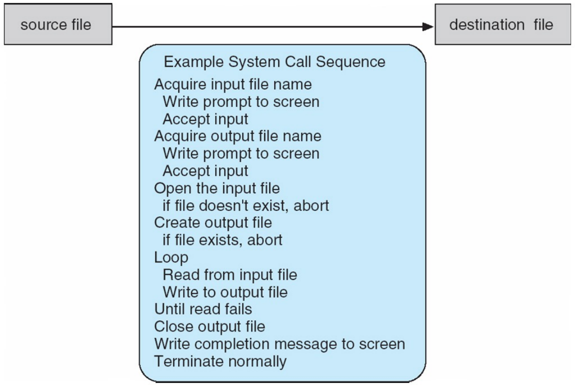
System Calls: an Example

```
cp in.txt out.txt
```

In this simple task, many OS services are employed:

- Entering the command, or moving a mouse to select files, causes sequence of I/O system calls.
- Then, files need to be opened: another set of system calls.
- Errors need to be detected: input file not existent, output file already exists with the same name.
- Can ask user if they want to replace the output file—requires set of system calls.
- When both files are open, we loop by reading bytes from one to another (system calls).
- Each read must return possible error conditions: end-of-file, hardware failure to read, ...
- Once done, files should be closed (system calls).

System Calls: an Example



Example System Call Sequence

```
Acquire input file name
  Write prompt to screen
  Accept input
Acquire output file name
  Write prompt to screen
  Accept input
Open the input file
  if file doesn't exist, abort
Create output file
  if file exists, abort
Loop
  Read from input file
  Write to output file
Until read fails
Close output file
Write completion message to screen
Terminate normally
```

Application Programming Interface (API)

Most programmers will not see this level of complexity of numerous OS services being in use.

APIs hide this away behind a set of standard functions which are made available to programmers, for performing common tasks when developing applications.

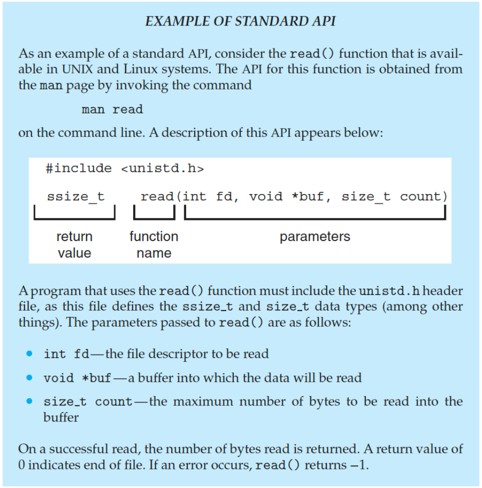
Input and output parameters are specified for each API function.

Common APIs:

- Windows API.
- POSIX API (UNIX, Linux, macOS).
- Java API (Applications based on the Java Virtual Machine).

$\boxed{\text{APIs provide code portability and eases the task of using OS services.}}$

Application Programming Interface (API)



EXAMPLE OF STANDARD API

As an example of a standard API, consider the `read()` function that is available in UNIX and Linux systems. The API for this function is obtained from the `man` page by invoking the command

```
man read
```

on the command line. A description of this API appears below:

```
#include <unistd.h>
```

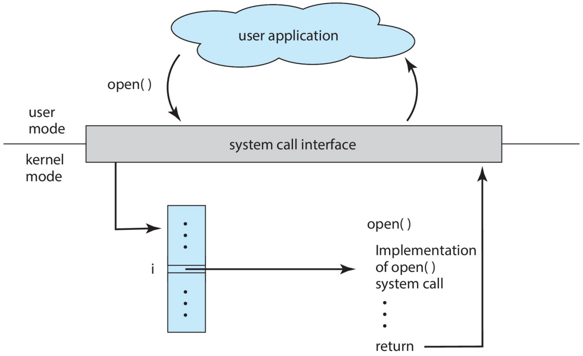
$$\underbrace{\texttt{ssize\_t}}_{\text{return value}}\quad\underbrace{\texttt{read}}_{\text{function name}}\underbrace{\texttt{(int fd, void *buf, size\_t count)}}_{\text{parameters}}$$

A program that uses the `read()` function must include the `unistd.h` header file, as this file defines the `ssize_t` and `size_t` data types (among other things). The parameters passed to `read()` are as follows:

- `int fd`—the file descriptor to be read
- `void *buf`—a buffer into which the data will be read
- `size_t count`—the maximum number of bytes to be read into the buffer

On a successful read, the number of bytes read is returned. A return value of 0 indicates end of file. If an error occurs, `read()` returns −1.

Application Programming Interface (API)



System call interface

This is an abstraction that allows programmer not to think about the details of system calls being used in API functions. Only need to obey the API and understand the effects of calling it.

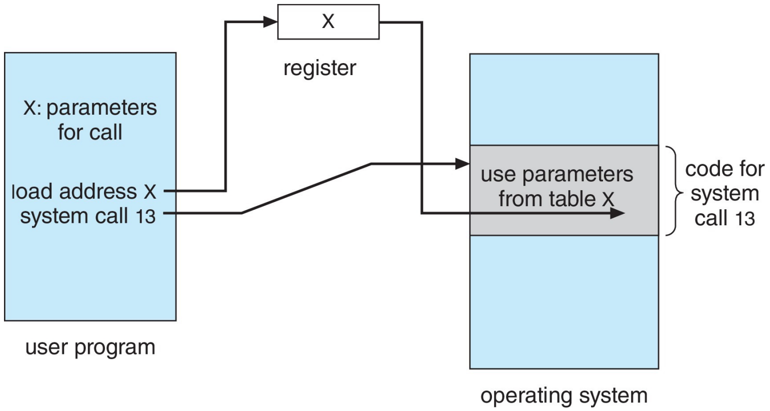
System Calls: Parameter Passing

System calls require various information, for example, files, devices, addresses in memory, lengths of byte streams, ...

Three methods to pass parameters to OS:

- Through registers.
- Store in a table and the address to it is passed through a register.
- Pushed to a stack by a program and popped off the stack by OS.

System Calls: Parameter Passing



System Calls: Types

We can roughly group system calls into six categories:

1. Process control
2. File management
3. Device management
4. Information maintenance
5. Communications
6. Protection

Next we discuss each of these categories.

System Calls: Process Control

- Running program needs to halt execution.
- If the termination is abnormal, some log files are usually generated.
- **Debugger** may use those logs to aid programmer in fixing problems.
- **Bugs** are usually discovered this way in the code.
- When a process is running, it may want to **load** and **execute** other programs.
- Create, terminate, duplicate, wait for processes.
- Get information about a process.
- Where data is shared among processes, **locking** is provided to assure no clashes.

We will go into detail in later weeks.

System Calls: File Management

Common system calls that deal with files:

- **Create** and **delete** files.
- **Open** files for reading and writing.
- Similar operations are required for directories.
- Determine and set attributes: file name, type, protection codes, ...

System Calls: Device Management

Processes may need resources to execute: main memory (RAM), disk drives, access to files, ...

Resources available can be granted, but usually processes will have to wait for them.

We can think of resources as **devices**: physical or virtual.

OS provides systems calls for interacting with these:

- Request and release a device.
- Similar to open and close system calls for files.
- Once we have the device allocated to us, we can read and write.
- File handling and general device handling is so similar that UNIX merge the two.

System Calls: Information Maintenance

There are system calls for transferring information between OS and user programs:

- Time and date calls.
- Version of OS.
- Amount of free memory or disk space.
- Memory **dump** also goes into this category.
- Other debugging info usually provided: single step, runtime profiling, program counter recording, various information about processes.

System Calls: Communication

Processes need to communicate, and there are two main methods: **message-passing model** and **shared-memory model**.

Message-Passing Process Communication Model

Processes exchange messages with one another to transfer information. Before communication takes places, connection must be opened. Computer **host name** and **process name** are used to identify the possibly remote parties for communication. System calls to establish or abort communication are available. Other system calls to receive and send messages are also available.

Shared-memory Model

Processes use system calls to create and gain access to regions of memory owned by other processes. Normally, OS prevents process from accessing memory allocated to other processes. In shared-memory model processes have to agree to remove this obstruction.

System Calls: Protection

OS should provide services for protecting computer system resources.

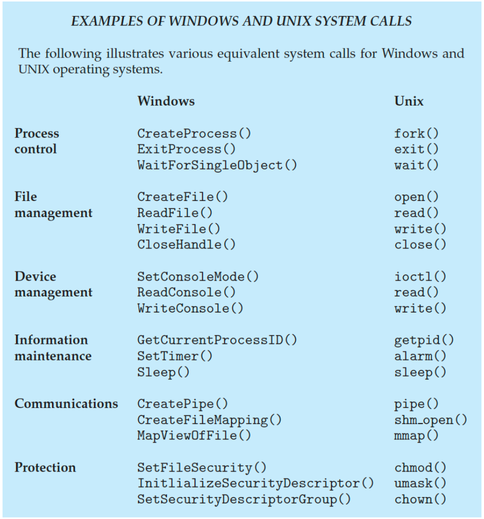
Traditionally this was to protect one user from another on an instance of some OS.

With Internet all systems started to get concerned about protection.

System calls include setting permission on files and disks.

Allow/deny access (for particular users).

System Calls: Example System Calls on Windows and UNIX



EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

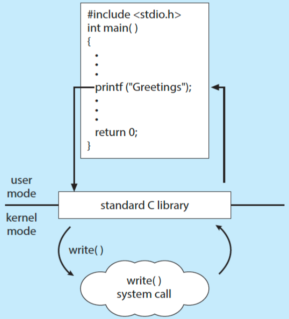
The following illustrates various equivalent system calls for Windows and UNIX operating systems.

	Windows	**Unix**
Process control	`CreateProcess()`	`fork()`
	`ExitProcess()`	`exit()`
	`WaitForSingleObject()`	`wait()`
File management	`CreateFile()`	`open()`
	`ReadFile()`	`read()`
	`WriteFile()`	`write()`
	`CloseHandle()`	`close()`
Device management	`SetConsoleMode()`	`ioctl()`
	`ReadConsole()`	`read()`
	`WriteConsole()`	`write()`
Information maintenance	`GetCurrentProcessID()`	`getpid()`
	`SetTimer()`	`alarm()`
	`Sleep()`	`sleep()`
Communications	`CreatePipe()`	`pipe()`
	`CreateFileMapping()`	`shm_open()`
	`MapViewOfFile()`	`mmap()`
Protection	`SetFileSecurity()`	`chmod()`
	`InitlializeSecurityDescriptor()`	`umask()`
	`SetSecurityDescriptorGroup()`	`chown()`

System Calls: Example System Calls on Windows and UNIX

THE STANDARD C LIBRARY

The standard C library provides a portion of the system-call interface for many versions of UNIX and Linux. As an example, let's assume a C program invokes the `printf()` statement. The C library intercepts this call and invokes the necessary system call (or calls) in the operating system—in this instance, the `write()` system call. The C library takes the value returned by `write()` and passes it back to the user program:

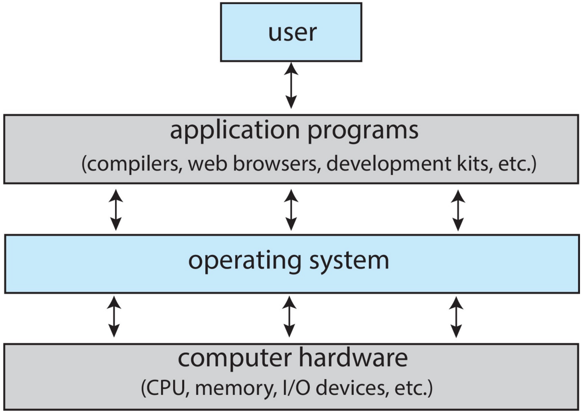


```
#include <stdio.h>
int main( )
{
   •
   •
   •
   printf ("Greetings");
   •
   •
   •
   return 0;
}
```

System Services

OS System Services

This is separate from **system calls** within the OS. **System services** are sitting between the OS and the Application Programs. Also called **system utilities**.


user
application programs
(compilers, web browsers, development kits, etc.)
operating system
computer hardware
(CPU, memory, I/O devices, etc.)

System Services

Some system services are interfaces to system calls, but some are more complex.

Examples:

- File management: create, delete, copy, rename, print, list files/directories.
- Status information. Can be simple: time, date, memory space, users. Could be more complex things about performance or debugging.
- File modification: text editors, text searching utilities, text transformation.
- Programming language support: compilers, assemblers, debuggers. interpreters.
- Program loading and execution.

Application programs supplied with OS are usually higher level tools that utilize many **system services**: browsers, word processors and text formatters, spreadsheets.

Most users view OS through application programs and system services, not system calls.

Vevox quiz

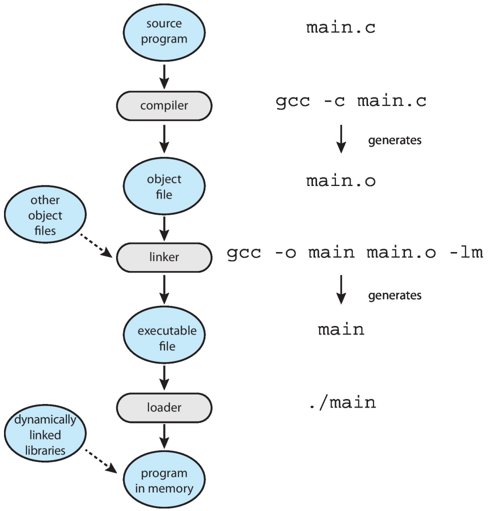
Part III: Code Compilation and Loading

Linkers and Loaders

Usually a program resides on a disk as a **binary executable file**. To run it, the executable has to be copied to memory and placed in the context of some process.

- **Relocatable object file**: source code compiled into object files suitable to be moved into a particular memory location.
- The **linker** combines these objects into a **binary executable**.
- Linker may include standard libraries, such as `math.h` in C.
- A **loader** loads the executable into memory to be run on CPU.
- **Relocation** assigns final addresses to various parts of the executable after it is placed in memory.

Linkers and Loaders



Linkers and Loaders

- Consider running `./main` on command line.
- The shell first creates a new process using the `fork()` system call.
- The shell then invokes the loader with `exec()` passing it the name of the executable: `main`.
- The **loader** loads the program into main memory using the **address space of the new process**.
- Similar process occurs in GUI by double clicking the executable with the mouse.

Linkers and Loaders: Dynamic Linking

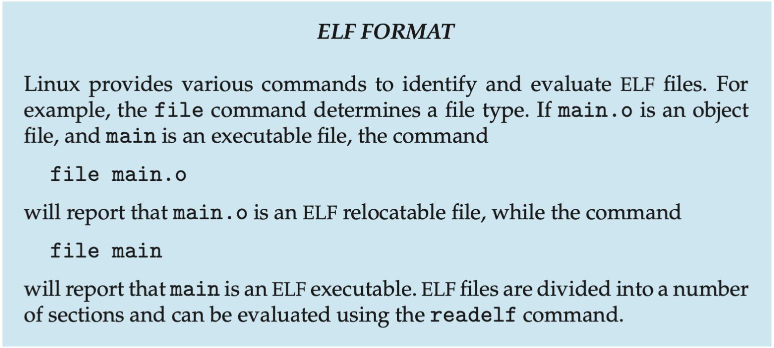
In the above we assume that libraries are linked into the executable and then loaded into memory together with the rest of the program code—even if the code will end up not calling those libraries.

Dynamic Linking

Link libraries dynamically when the program is being loaded into memory. Avoid linking and loading libraries that will end up not being used in the program. Instead the library is loaded when, and if, it is required during run time. Possible memory space improvements.

Linkers and Loaders: ELF format

Object files and executables typically have a standard format. It holds machine code and various metadata about functions and variables in the program. Unix and Linux use the ELF format.



ELF FORMAT

Linux provides various commands to identify and evaluate ELF files. For example, the `file` command determines a file type. If `main.o` is an object file, and `main` is an executable file, the command

```
file main.o
```

will report that `main.o` is an ELF relocatable file, while the command

```
file main
```

will report that `main` is an ELF executable. ELF files are divided into a number of sections and can be evaluated using the `readelf` command.

One data point of interest is an **entry point**—address of the instruction to execute upon the start of the program.

Why Applications are OS Specific

- Applications compiled for one system (OS-hardware combination) usually will not work on a different system.
- Each OS has unique system calls.
- Possible solutions:
    1. Use interpreted languages like Python, Ruby: interpreter on each system goes through the source code and executes correct instructions and system calls. Interpreter can be limited.
    2. Use language like Java that runs on Java Virtual Machine (JVM): virtual machine is ported to different systems and programmers use the universal interface of the JVM rather than the specific OS.
    3. Compile code (such as C) for every different configuration.

In general this is still a difficult problem and there is no ultimate solution. Porting is required.

Part IV: Design and Structure of Operating Systems

OS Design and Implementation

Design of an operating system is a major undertaking and there is no complete solution that could generate an OS automatically given requirements.

Internal structure can vary widely, based on the purpose of OS.

User goals and **system goals** first are outlined.

User: OS easy to learn and use, reliable, fast, safe.

System: easy to design, implement, maintain; efficient, reliable, error free.

$\boxed{\text{There can be many interpretations of these vague requirements}}$

General principles are known (we are learning them in this module), but designing one is a creative task that relies on many human decisions and **software engineering**.

OS Design and Implementation: Policy and Mechanism

Separation of **policies** (what) and **mechanisms** (how) is an important concept.

- Example policy: interrupt OS regularly; Mechanism: timer interrupts.
- Good approach as we can change policies later and mechanisms are in place: for example, change the timer interrupt frequency.

Linux example

The standard Linux kernel has a specific CPU scheduler—but we can change it to support a different policy in how we schedule different jobs on the system.

OS Design and Implementation: Languages

OS is a collection of many programs, implemented by many people over years—general statements hard to make but there are some common points.

- Kernel: assembly, C.
- Higher level routines: C, C++, other.
- For example, Android is mostly C and some assembly.
- Android system libraries C or C++.
- Android APIs: Java.

Advantages of High Level Languages

Code written faster, more compact, easier to understand and maintain. Compiler improvements easy to integrate by recompiling the OS. Easier to port the whole OS to new architectures.

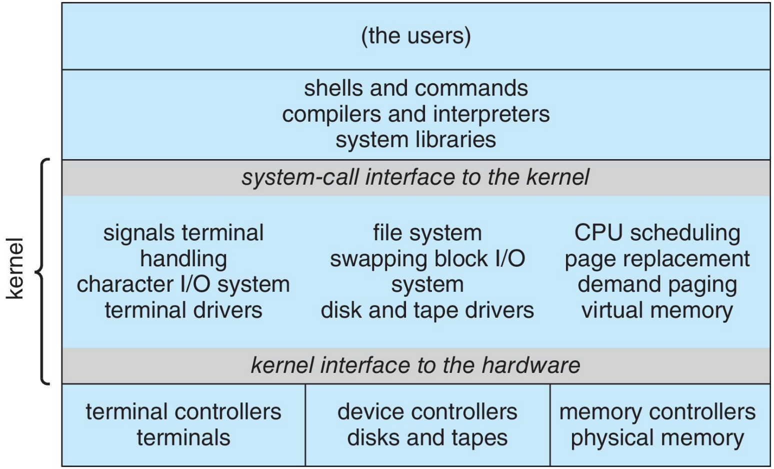
Vevox quiz

OS Structure

Monolithic kernel: Place all the functions of the kernel into a single, static binary file that runs a single address space. Not much structure or no structure at all.

Original UNIX system used this approach: it had a kernel and the system programs. It has evolved over the years with some structure.

OS Structure: Traditional UNIX system

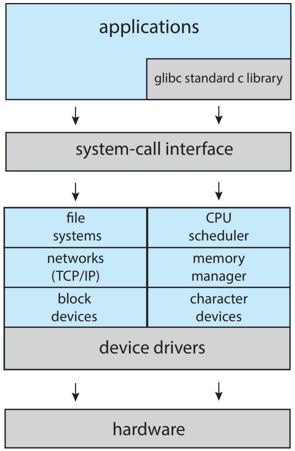


OS Structure

Monolithic kernels are simple in concept, but are difficult to implement and extend as everything is in one big kernel rather than structured.

They have performance advantage, which explain why they are still relevant.

OS Structure: Linux system



OS Structure

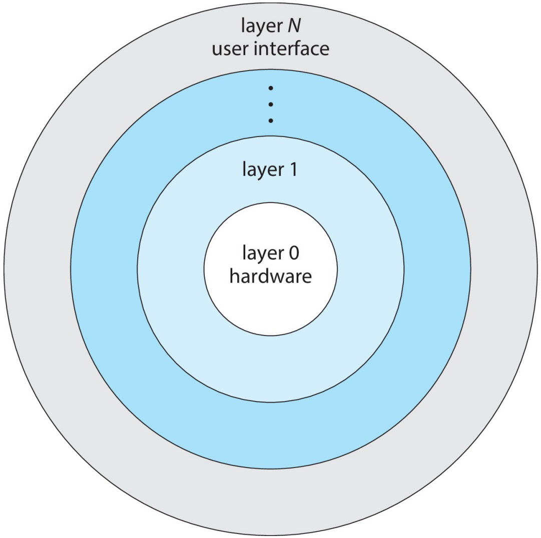
Monolithic kernels are said to be **tightly coupled** because changes in the system can affect all other parts.

We can instead take a **loosely coupled** approach where the kernel is structured into parts doing specific and limited functions.

Layered system: highest layer is user interface, while lowest layer is hardware. Layers can only call functions from the layer below.

Debugging is easier in this—debug first layer without affecting the rest of the system, once done, move up the layer.

OS Structure: Layered approach



OS Structure

Kernels can be modularized using the **microkernel** approach.

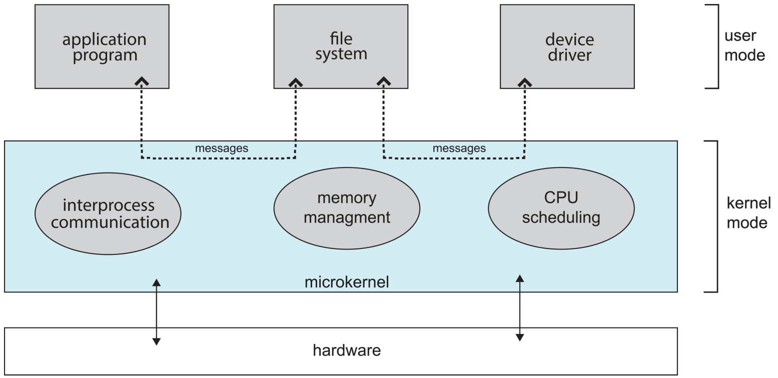
Remove all nonessential components from the kernel and implement them as user level programs—this results in a small kernel.

When the operating system needs to be extended, new services are added in user space rather than modifying the kernel. Kernel modifications require fewer changes since it is small.

It is also easier to port to another OS and provides more security since most services run in user mode.

Performance suffers compared to one big kernel—different parts have to communicate.

OS Structure: Microkernel



Vevox Quiz

COMP2211: Progress

Week	Topic
1	~~Introduction to OS~~
2	~~OS services~~
3	Processes
4	Xv6: Live coding and Q&A from the xv6 book
5	**Reading week. No scheduled labs or lectures**
6	Threads and concurrency. **Assignment due this week.**
7	Scheduling
8	Process synchronisation
9	Memory management
10	Catch up
11	Module review

Progress

Week	Topic
1	Introduction to OS
2	OS services
3 current	Processes
4	Xv6: Live coding and Q&A from the xv6 book
5	**Reading week. No scheduled labs or lectures**
6	Threads and concurrency. **Assignment due this week.**
7	Scheduling
8	Process synchronisation
9	Memory management
10	Catch up
11	Module review

Reading List

We will be mainly using the *Operating System Concepts* (OSC) 10th ed., 2018, and the **4th edition of the xv6 book (XV6)**, 2024. These slides are based on OSC (see the reference list).

Week	Reading materials
1	Chapter 1 OSC. Chapter 1 XV6.
2	Chapter 2 OSC. Chapter 2 XV6.
3 (current)	Chapter 3 OSC.
4	Reread Chapters 1–2 XV6.
5	Reread Chapters 1–3 OSC.
6	Chapter 4 OSC. **CW deadline**
7	Chapter 5 OSC.
8	Chapters 6–8 OSC.
9	Chapters 9–10 OSC. Chapter 3 XV6.
10	Reread Chapters 4–6 OSC.
11	Reread Chapters 9–10 OSC.

Objectives

- Study the concept of processes.
- OS representation and scheduling of processes.
- Creation and termination of processes.
- Study the methods for interprocess communication.

Part I: Introduction to Processes

What is a Process?

- Early computers: only one program executed at a time.
- One program had complete control over resources.
- Today: multiple programs in memory, all executed through **multitasking**.
- This evolution required compartmentalization of various programs.

Process

Process is a program in execution. One program invoked multiple times results in multiple processes.

A Concept of Processes

- Early computers were **batch systems**: execute **jobs** submitted by users. Minimum interaction.
- This was followed by **time-shared** systems: **user programs** or **tasks**. Potentially interacting.
- Even one user can run several tasks: browser, email, editor.

Jobs

Calling running programs **jobs** has historical significance, as most of the OS concepts were developed around job processing. You may see the term used to this day, but **process** is a modern replacement

A Concept of Processes

The status of the current activity of some process is represented by

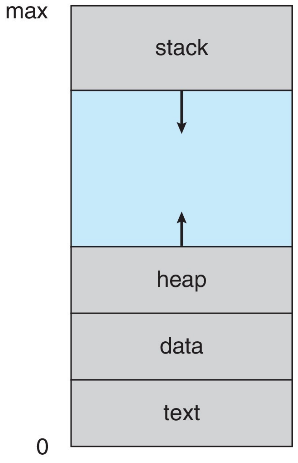
- Current value of the **program counter**: where are we in the execution of the binary?
- Contents of the CPU **registers**: what data are we working on?

A Concept of Processes: Memory Layout

Each process has its own memory layout:

- **Text section**: executable code.
- **Data**: global variables.
- **Heap section**: Memory that grows and shrinks dynamically during execution.
- **Stack**: Structure for temporary values (function parameters, return addresses, and local variables).

Text and data sections are fixed size.
Heap and stack change size.



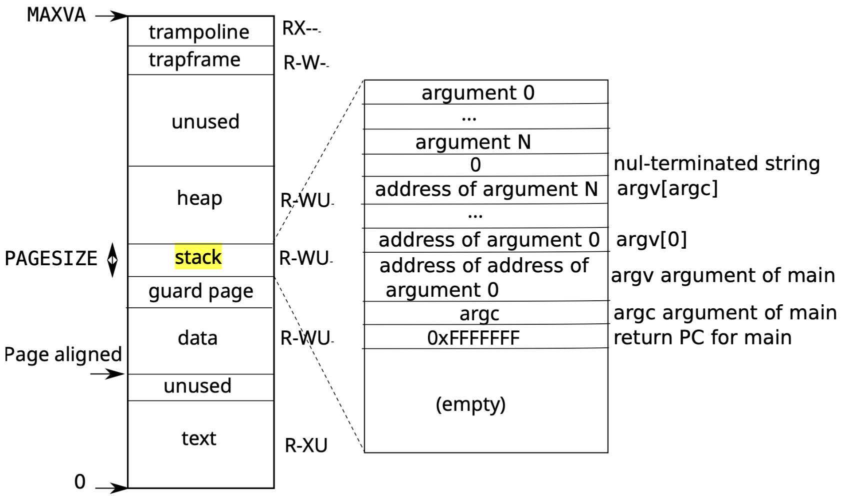
A Concept of Processes: Stack

Stack contains potentially a small amount of data which is **pushed** and **popped** using specific CPU instructions.

For example, when a function is called, input arguments, local variables, and the return address are **pushed** onto the stack.

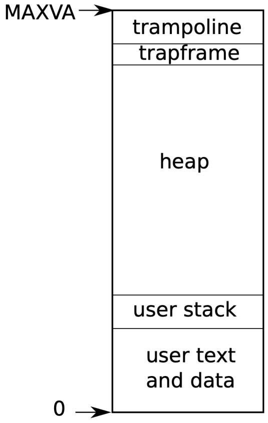
Upon completing the function, data is **popped** from the stack: the last one is usually the return address to get back to caller.

A Concept of Processes: xv6 stack

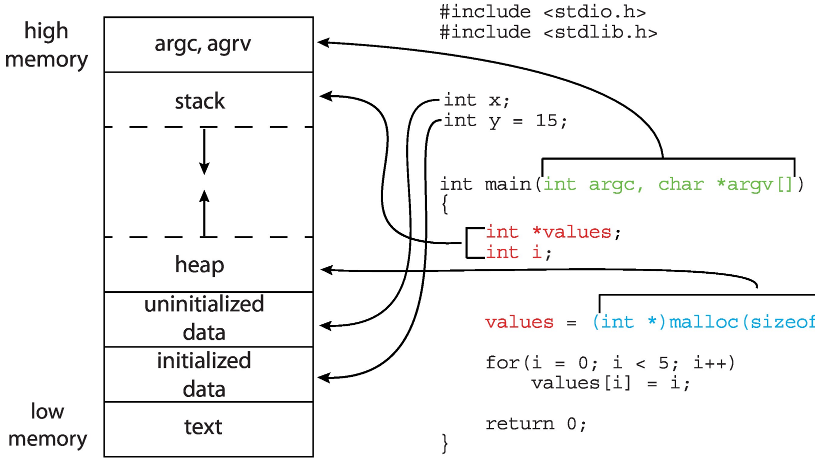


A Concept of Processes: Heap

- The **heap** can be grown dynamically.
- In C `malloc` and `free` do that for us.
- Usually heap and stack grow toward each other—overlap watched by OS.
- On the right is the xv6 process memory, which is slightly different.



A Concept of Processes: Memory Layout Example



```
#include <stdio.h>
#include <stdlib.h>

int x;
int y = 15;

int main(int argc, char *argv[])
{
    int *values;
    int i;

    values = (int *)malloc(sizeof(int)*5);

    for(i = 0; i < 5; i++)
        values[i] = i;

    return 0;
}
```

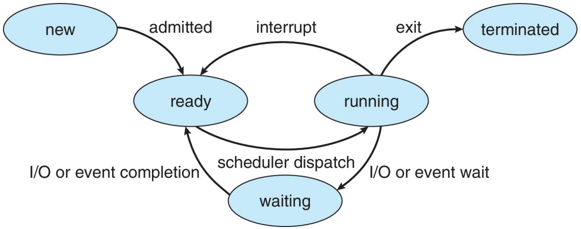
Vevox Quiz

Part II: Scheduling of Processes

Process State Transitions

Processes change **states** during execution.

- **New**: Process has been created.
- **Running**: CPU is reading process' instructions.
- **Waiting**: Waiting for an event (for example, I/O completion).
- **Ready**
- **Terminated**

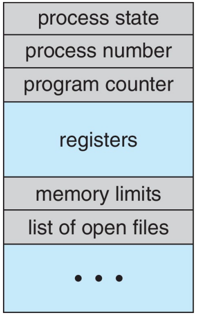


Note that only one process can be **running** on a core. Others may be **ready** or **waiting**.

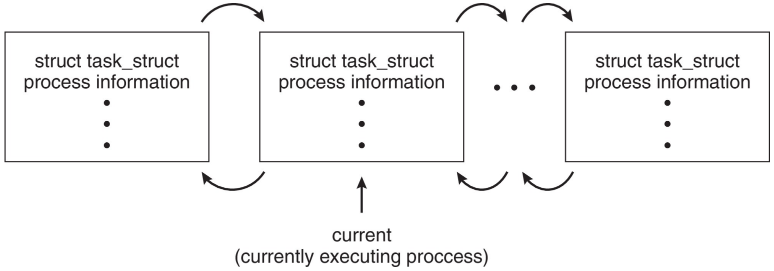
Process Control Block (PCB)

OS keeps track of processes using a **process control block (PCB)**.

- **Process state**
- **Program counter (PC)**
- **CPU registers**: along with the PC, these have to be saved when process is interrupted.
- **CPU-scheduling information**: priority and other scheduling parameters.
- **Memory-management information**: location of various memories assigned to the process (Week 9).
- **Accounting information**: resource utilization statistics.
- **I/O status information**: I/O devices allocated to the process, open files, ...



Process Control Block (PCB): Linux Example



```
long state;                     /* state of the process */
struct sched_entity se;         /* scheduling information */
struct task_struct *parent;     /* this process's parent */
struct list_head children;      /* this process's children */
struct files_struct *files;     /* list of open files */
struct mm_struct *mm;           /* address space */
```

Processes or Threads?

You may have heard of **threads** more rather than processes.

Common term in parallel programming frameworks, such as GPU programming.

Here we implied that a process performs a **single thread of execution**.

A single thread of instructions being read in sequence by CPU.

Multi-threading

Nowadays Operating Systems extend processes to be able to perform multiple tasks at once—for example, two cores executing at different locations in the binary of a process. PCB and other parts of OS have to be expanded. See Week 6 for detail.

Process Scheduling

The objective of **multi-programming** is to maximize CPU utilization.

Time-sharing (or multitasking) adds another requirement—the switching between processes should be frequent enough for users to interact with the running programs.

Process Scheduler

Integral part of operating systems which meets the constraints posed by time-sharing and multi-tasking by selecting a process to run from a set of available processes.

Multiprocessing

Each CPU core can run one process at a time; N CPUs can run N processes. If more processes than cores are created, some will have to *wait*. **Degree of multiprogramming** defines the number of processes currently in memory.

Process Scheduling

Two types of processes:

- **I/O bound**: most time spent *waiting* for memory.
- **CPU bound**: most time spent in execution.

The types of processes going through the system will affect the objectives of multiprogramming and time-sharing.

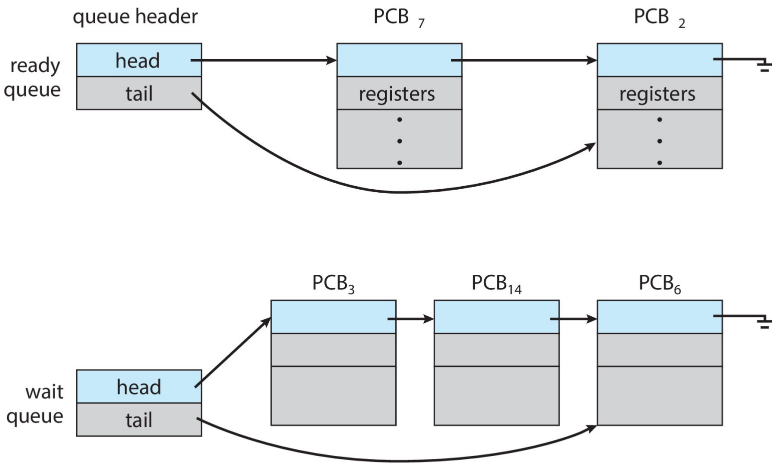
Process Scheduling: Queues

Processes enter the system and are put into a **ready queue**.

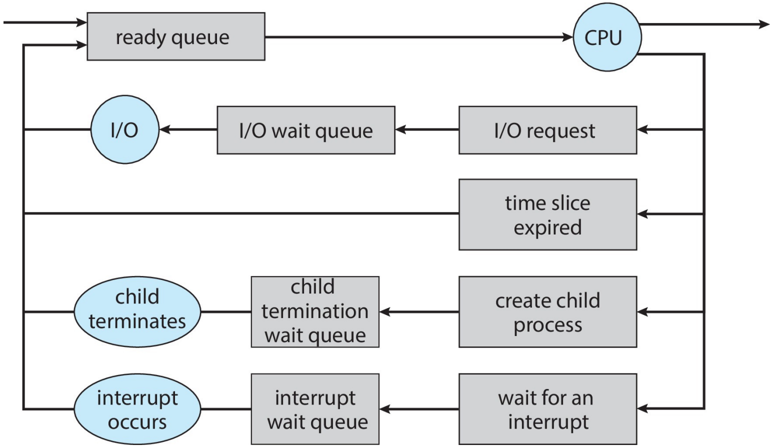
Queue is usually a linked list, where each PCB links to the next.

There may be other queues, for example **wait queue** for processes waiting I/O.

Process Scheduling: Queues



Process Scheduling: Queuing Diagram



Process Scheduling: CPU Scheduler

Role of a scheduler: from a set of **ready** select one and run on CPU.

- Scheduler is working frequently;
- I/O bounds processes may execute for a few milliseconds before waiting for I/O.
- CPU-bound processes may require CPU for extended durations, but scheduler unlikely to grant it.
- Typically designed to switch processes very frequently (less than every 100 milliseconds).

Memory Swapping

This technique may decide to move a process from memory to disk, reduce the **degree of multiprogramming** and thus reduce the active contention for the CPU. Later the process can be returned to memory and continued where it left off (state save/restore required).

Process Scheduling: Context Switching

- Remember **interrupts** cause CPU to pause current task and do some **kernel** routine.
- CPU needs to save the current **context** of a process and later restore it for continuing running it.
- Context is represented in the PCB of a process.
- Register contents, state of the process, memory management information.

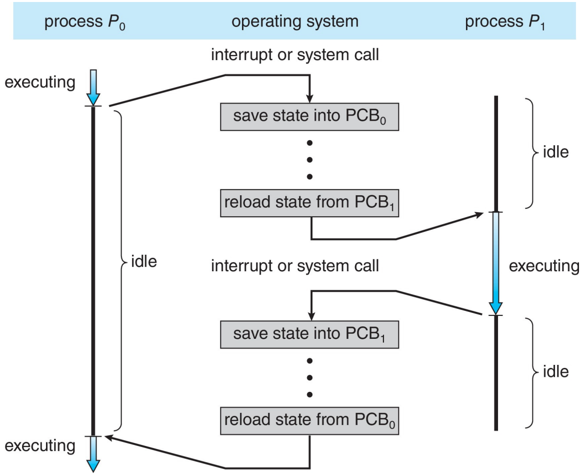
Context Switch

Perform a **state save** of the current process and a **state restore** of a different process.

Context switch is pure overhead as CPU is not executing any process instructions.

Typical speed: several microseconds. Depends on size of state needed to save/restore.

Process Scheduling: Context Switching



Vevox Quiz

Part III: Process Manipulation

Operations on Processes: Creation

Processes may create several new processes during execution.

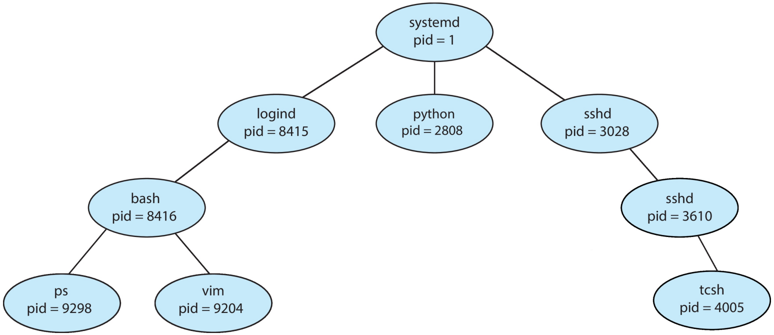
Creating process is called a **parent process** and the new processes are called **children processes**.

New processes can in turn create more—this forms a **tree of processes**.

Process identifier (pid) is usually used to identify each process.

Operations on Processes: Creation

Linux example



systemd created on boot; it starts the processes for various services.

Operations on Processes: Creation

Options for resources for the child processes:

- Obtain directly from OS.
- Share a subset of resources from a parent.

Restricting child process to a subset of the parent's resources avoids overloading the system through creation of many child processes.

Operations on Processes: Creation

When a process creates another process, two possibilities:

- Parent and child execute **concurrently** (not necessarily in parallel).
- Parent waits until some or all of its children terminate.

Address space also has two possibilities:

- Parent and child have the same program and data (xv6 `fork`).
- The child has a new program loaded into it (xv6 `fork` and then `exec`).

Operations on Processes: Creation

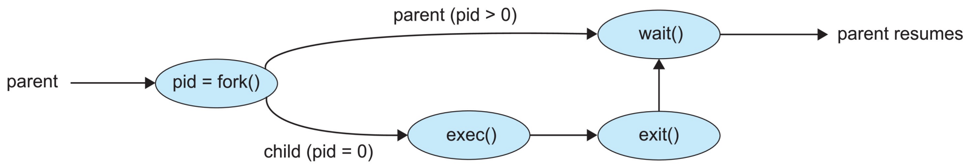
In UNIX, a new process is created by `fork()`:

- New process has a copy of address space of the original process.
- Both processes continue execution at the instruction after the `fork`.
- `fork()` returns zero in the child and PID of the child in the parent.

After the `fork()` usually `exec()` is called:

- Process' memory space is replaced by a new program.
- Load a binary file into memory.
- Destroy the memory image of the program containing `exec` system call.
- Parent can then create more children, or wait until termination of current ones.

Operations on Processes: Creation



Operations on Processes: Creation in UNIX with C

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
       fprintf(stderr, "Fork Failed");
       return 1;
    }
    else if (pid == 0) { /* child process */
       execlp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
       /* parent will wait for the child to complete */
       wait(NULL);
       printf("Child Complete");
    }

    return 0;
}
```

Operations on Processes: Termination

Process terminates when it asks OS to delete it using `exit()` system call.

All the resources: physical and virtual memory, open files, I/O buffers are reclaimed by the OS.

A parent may forcibly terminate its created processes:

- If the child process has exceeded its usage of some allocated resources.
- The job that the child is doing is no longer required.
- The parent is exiting and it is required to terminate the sub-tree of processes before exiting (**cascading termination**).

Zombie processes

Parents may call `wait()` to wait for their children to terminate. The processes that have terminated but whose parents have not yet called `wait()` are called zombie processes—we have to keep them in the system to return the status to the parent eventually.

Part IV: Communicating Processes

Interprocess Communication

Active processes on the system can either be **Independent** or **cooperating**.

A process is **independent** if it does not share any data while executing.

A process is **cooperating** if it can affect or be affected by other processes.

Process cooperation useful in a few scenarios:

- **Information sharing**: for example, copy-paste between programs.
- **Computation speedup**: split big tasks into multiple subtasks.
- **Modularity**: The system may be designed to have separate processes or threads working cooperatively to achieve some function.

When processes cooperate, they require **interprocess communication (IPC)**.

Interprocess Communication

Two fundamental concepts:

- **Shared-memory model**: agree a region of memory to share among cooperating processes. Read and write there to exchange info.
- **Message-passing model**: use a message-passing protocol to send and receive information.

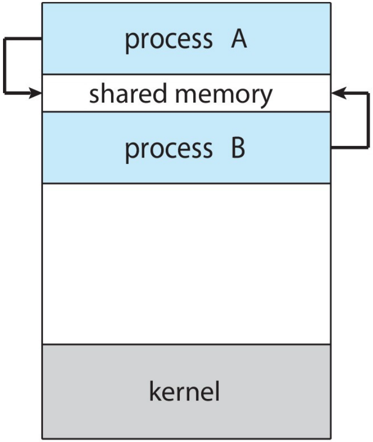
Both are implemented in operating systems.

Message-passing model is useful when no conflict resolution is desired. However, it is slower, since each read/write requires kernel ops.

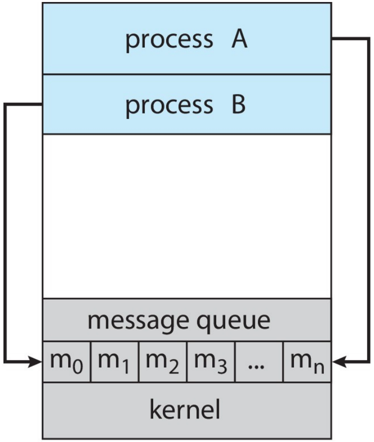
With share-memory model, conflicts and race conditions may appear (two processes write).

Message-passing is required to communicate between different systems that do not share memory.

Interprocess Communication



(a)



(b)

Interprocess Communication: Pipes

Pipes were one of the earliest UNIX mechanisms for interprocess communication.

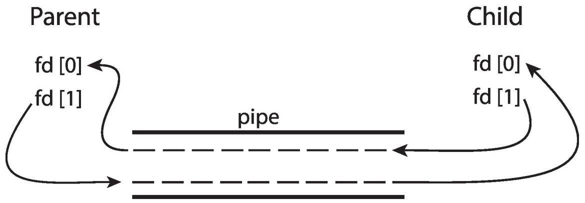
An example of shared-memory model of communication.

Four key design considerations:

- **Bidirectional** or **unidirectional** communication?
- If bidirectional, can data travel both directions at once?
- Do we need a relationship (parent-child) between communicating processes?
- Can we use pipes over network or locally only?

Interprocess Communication: Ordinary Pipes

- **Produced-consumer** model: producer writes to the **write end** of the pipe while the consumer reads from the **read end**.
- **Unidirectional**: we need two pipes for communicating back to the producer.
- In UNIX this is constructed using `pipe(int p[])` where `p[0]` is the read end of the pipe and `p[1]` is the write end.
- `p[0]` and `p[1]` are special types of *files* in UNIX, thefore `fork()` in the parent will make the child inherit these.



Interprocess Communication: Named Pipes

Ordinary pipes provide a simple mechanism for processes to communicate, but they only exist until processes exist and communicate. When they terminate, the pipe disappears.

Named pipes (FIFOs) in UNIX provide extra functionality:

- Bidirectional communication.
- No parent-child relationship needed.
- Several processes can use the pipe for communication.
- Pipe remains active after communicating processes terminate.

Named pipes are bidirectional, but provide only half-duplex transmission (only one direction at a time).

COMP2211: Progress

Week	Topic
1	~~Introduction to OS~~
2	~~OS services~~
3	~~Processes~~
4	Xv6: Live coding and Q&A from the xv6 book
5	**Reading week. No scheduled labs or lectures**
6	Threads and concurrency. **Assignment due this week.**
7	Scheduling
8	Process synchronisation
9	Memory management
10	Catch up
11	Module review

Progress

Week	Topic
1	Introduction to OS
2	OS services
3	Processes
4	Xv6: Live coding and Q&A from the xv6 book
5	Reading week.
6 (current)	Threads and concurrency. **Assignment due this week.**
7	Scheduling
8	Process synchronisation
9	Memory management
10	Catch up
11	Module review

Reading List

We will be mainly using the *Operating System Concepts* (OSC) 10th ed., 2018, and the **4th edition of the xv6 book (XV6)**, 2024. These slides are based on OSC (see the reference list).

Week	Reading materials
1	Chapter 1 OSC. Chapter 1 XV6.
2	Chapter 2 OSC. Chapter 2 XV6.
3	Chapter 3 OSC.
4	Reread Chapters 1–2 XV6.
5	Reread Chapters 1–3 OSC.
6 (current)	Chapter 4 OSC.
7	Chapter 5 OSC.
8	Chapters 6–8 OSC.
9	Chapters 9–10 OSC. Chapter 3 XV6.
10	Reread Chapters 4–6 OSC.
11	Reread Chapters 9–10 OSC.

Objectives

- Discuss the motivation, benefits, and challenges in designing **multithreaded processes**.
- Talk about the basic components of a **thread**.
- Describe mechanisms for threading.

Vevox big quiz (15–20 minutes)

Part I: The Concept of Threads

Threads: Introduction

In week 3 we have explored the concept of **processes**, which assumes that it is a running program that has a single thread of control.

For interest, xv6 supports only single-threaded processes. See p. 29 of the xv6 book.

In modern computing most operating systems provide capabilities for processes to have **multiple threads of control**.

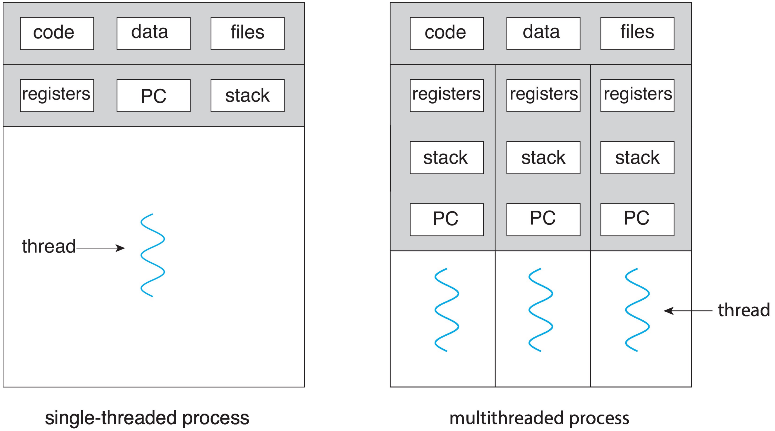
Threads: Introduction

What is a Thread?

A basic unit of CPU utilization; it comprises a **thread ID**, a **program counter (PC)**, a **register set**, and a **stack**.

- Same threads within a process share: code section, data section, and other resources (for example open files).
- A traditional process has a single thread of control.
- Processes with multiple threads can perform more than one task at a time.

Single and Multithreaded Processes



Multithreading

Examples:

- Application creating photo thumbnails from a collection of images uses a different thread for each image.
- A web browser displays images or text in one thread and retrieves network data in another.
- Word processor has a thread to display UI, a thread to respond to keystrokes, and a thread for spellchecking.

Multicore Systems and Threads

Applications can be designed for threads to run in parallel on multicore systems.

Example: Multithreading in a Web Server

In some situations, applications may be required to perform several similar tasks.

For example, a web server accepts client requests for web pages, images, sound.

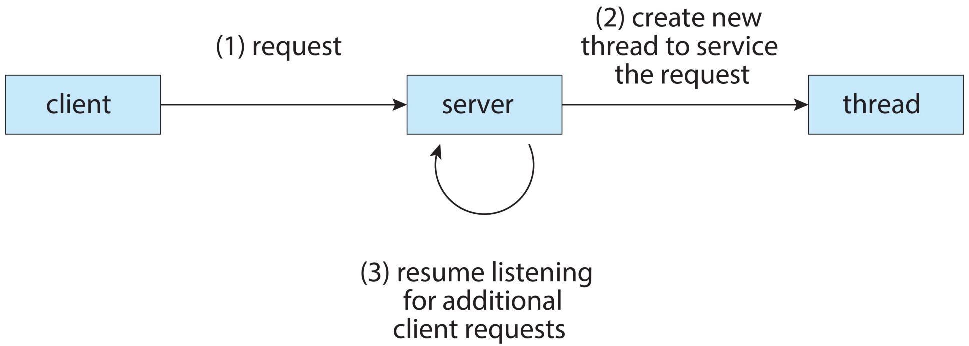
Several users may request access at the same time.

Running a single thread, the web server would hold other users, potentially for prolonged periods.

One approach: tree of processes, with the root being the server and children being user requests—time consuming process creation and resource use significant.

Since similar tasks are performed on requests, **multithreading is a more efficient solution**.

Example: Multithreading in a Web Server


(1) request
(2) create new thread to service the request
client
server
thread
(3) resume listening for additional client requests

Multithreading

Other motivating aspects:

- Most OS kernels are multithreaded; for example, during Linux boot time, threads are created for managing devices, memory management, or interrupt handling.
- Various applications that parallelize well can take advantage of multithreading: sorting, tree algorithms, graph algorithms.
- Data mining, graphics, artificial intelligence: people aim to design algorithms to **exploit multicore architectures**.
- Sometimes problems are **embarrassingly parallel**, without data dependencies, such as adding two vectors together. These problems can be easily solved across many cores.

Sign up for *COMP3221* next year to get into the details.

Multithreading: Benefits

Four major categories:

1. **Responsiveness**: if one part of an application blocks, other threads can continue working.
2. **Resource sharing**: threads share memory resources of a process to which they belong.
3. **Economy**: allocating resources when creating processes is costly; context switch is also costly. Threads are cheaper in both aspects.
4. **Scalability**: Multi-threaded processes can exploit multiple cores, whereas a single-threaded ones can only run on one core.

Part II: Introduction to Parallel Computation

Multicore Programming

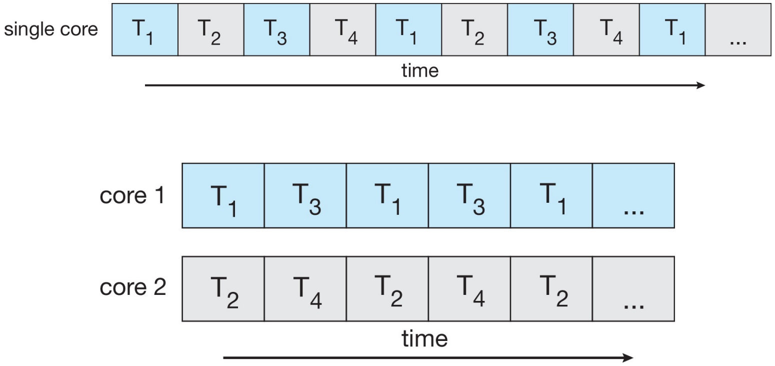
Recall that a **multicore** environment is an environment in which single processing chip contains multiple computing cores.

The communication within cores on the same chip is very fast.

Multithreaded programming provides a mechanism for an efficient use of multicores.

Multicore Programming: Concurrency or Parallelism?

Concurrency (top) and **parallelism** (bottom) refer to different concepts.



Multicore Programming: Concurrency or Parallelism?

On single core, concurrency means **interleaving the execution of threads in time**.

On a multicore system, concurrency means that some threads can execute simultaneously, which means there is parallelism.

We can have concurrency without parallelism.

Historical perspective

Before multi-processor/core architectures became prevalent, most systems had a single processor and operating systems were designed to *provide illusion of parallelism by rapidly switching processes*.

Multicore Programming: Key Challenges in Designing Programs

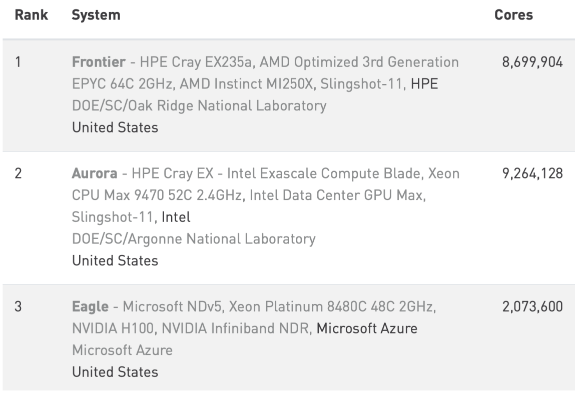
System designers and application programmers are pressured to make better use of multiple cores—this is ongoing.

The systems are growing in size, both at large (warehouse computers) and small (laptops, phones) scale.

- Operating Systems have to accommodate multicore hardware.
- Old single-threaded applications have to be ported.
- New algorithms have to be developed from scratch (see the whole topic of **parallel algorithms**).

Multicore Programming: Key Challenges in Designing Programs

For example, see the TOP500 list: `https://top500.org/lists/top500/list/2024/06/`.



Rank	System	Cores
1	**Frontier** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE DOE/SC/Oak Ridge National Laboratory United States	8,699,904
2	**Aurora** - HPE Cray EX - Intel Exascale Compute Blade, Xeon CPU Max 9470 52C 2.4GHz, Intel Data Center GPU Max, Slingshot-11, Intel DOE/SC/Argonne National Laboratory United States	9,264,128
3	**Eagle** - Microsoft NDv5, Xeon Platinum 8480C 48C 2GHz, NVIDIA H100, NVIDIA Infiniband NDR, Microsoft Azure Microsoft Azure United States	2,073,600

$\sim$ 9 million cores. Photo source: Wikipedia.

Multicore Programming: Key Challenges in Designing Programs

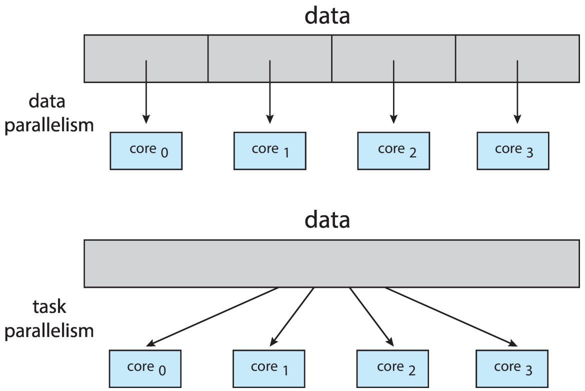
1. **Identifying tasks**: examine applications and find workloads that can be divided, ideally into independent tasks.
2. **Balance**: make sure parallel tasks perform similar amounts of work.
3. **Data splitting**: The data accessed and manipulated by parallel tasks have to be divided.
4. **Data dependency**: Do tasks depend on output data from other tasks; **synchronization** may be required.
5. **Testing and debugging**: Program running on N cores has many execution paths. Debugging more difficult than in a single-threaded case.

Of interest

Many people believe that an entirely new software design approach will be needed in the future. Computer Science educators often talk about teaching software development through increased emphasis on parallel programming.

Question: how many of you have done some parallel code development/debugging/reading?

Multicore Programming: Types of Parallelism



Top: **data parallelism**; bottom: **task parallelism**.

Multicore Programming: Types of Parallelism

Data parallelism: distribute data across N cores, each to receive a *subset* of the whole data.

Example: sum a vector of size K. Single-core would get elements from 0 to $K-1$ and sum them in series. On $N=2$ core system, core 1 would get elements 0 to $K/2-1$ and core 2 would get elements $K/2$ to $K-1$.

Task parallelism: distribute different tasks (operations) across multiple cores. Each task may require part or the whole of the input data.

In practice you may likely see a hybrid approach.

Multicore Programming: Amdahl's Law

What speedup can we expect when we add additional cores to an application that contain both **serial** and **parallel** parts?

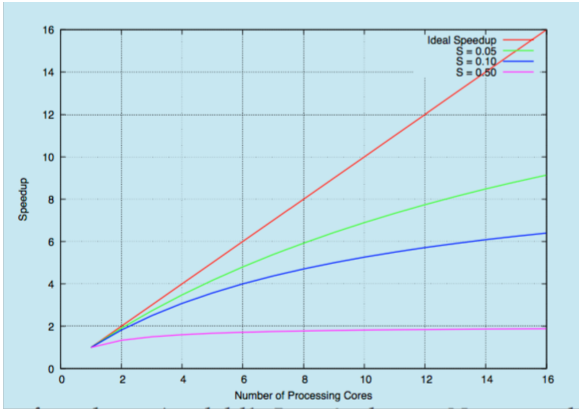
$$speedup \leq \frac{1}{S + \frac{1-S}{N}}$$

where N is the number of cores, S is a portion of the application that must be performed serially.

Example: $S = 0.25$ (25% of the application is serial and 75% is parallel). If $N = 2$ (2 cores), the speedup is no greater than $1.6\times$.

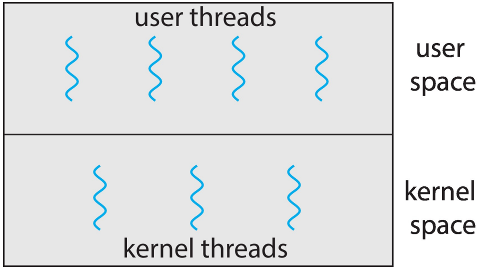
If we now set $N = 4$ (4 cores), the upper bound on the speedup is $2.28\times$ (not $4\times$!).

Multicore Programming: Amdahl's Law



Part III: Libraries for Multithreading

Multithreading Models

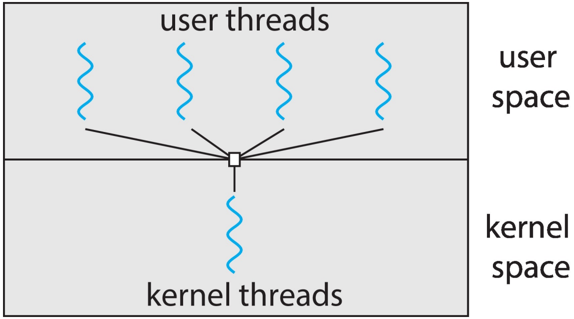


Support may be provided for threads at user level and kernel level: **user threads** and **kernel threads**.

User threads work in user mode and kernel threads require direct OS support.

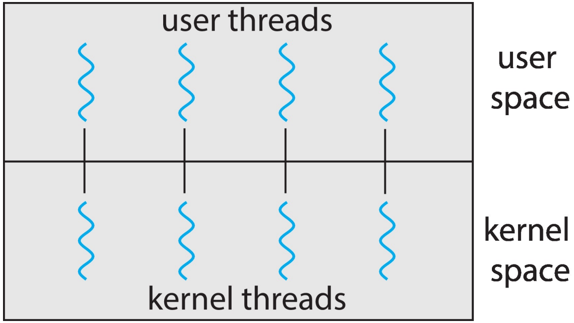
$\boxed{\text{What is the relationship between user and kernel threads?}}$

Multithreading Models: Many-to-One



Disadvantage: entire process blocks if one thread calls a blocking system call. Multiple threads are unable to run in parallel → very few systems implement this.

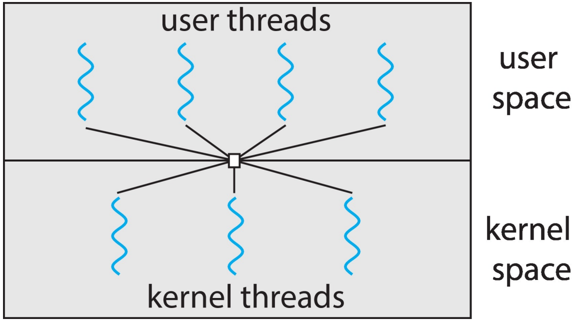
Multithreading Models: One-to-One



Advantages: another thread can run when one calls a blocking system call. Parallel processing doable.

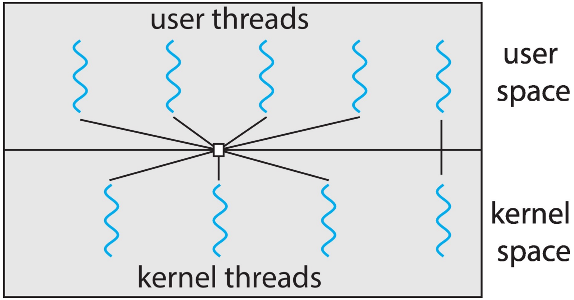
Disadvantage: Each user thread requires creating a kernel thread. Performance may suffer.

Multithreading Models: Many-to-Many



Advantage: Number of kernel threads customizable according to an application or machine requirements. Number limited; does not depend on how many user threads. Kernel can run threads in parallel.

Multithreading Models: two-level-threads



Combined approach which allows specific user threads to be assigned a kernel thread, but still multiplex between other user and kernel threads.

Thread Libraries

- A thread library provides an API for managing threads.
- Two approaches: entirely in user or in kernel modes.
- Three main libraries: POSIX Pthreads, Windows thread library, and Java thread API.
- For Pthreads and Windows data declared globally is shared among threads.

Synchronous and asynchronous threading

In the **asynchronous threading** parent continues working concurrently with any children threads created. In the **synchronous threading** parent waits for children to complete: for example, parent may combine the results from children.

Thread Libraries: Pthreads

- Pthreads is a standard API for thread creation and synchronization.
- Various IEEE standards address it.
- It is a **specification not an implementation**.
- OS designers can implement the specification their own way.
- The C program on the right is a standard way to use pthreads.
- `runner` executes in a separate thread from `main`.
- `sum` is shared between both threads.

Look into Windows and Java threading.

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

int sum;
void *runner(void *param);

int main(int argc, char *argv[]) {
  pthread_t tid;
  pthread_attr_t attr;

  // Create a thread.
  pthread_attr_init(&attr);
  pthread_create(&tid, &attr, runner, argv[1]);

  // Wait for the thread to exit.
  pthread_join(tid,NULL);

  printf("sum = %d\n",sum);
}

void *runner(void *param) {
  int i, upper = atoi(param);
  sum = 0;
  for (i = 1; i <= upper; i++)
    sum += i;
  pthread_exit(0);
}
```

Implicit Threading

Designing parallel programs manually is potentially a cumbersome task.

To better support the design of concurrent and parallel applications is to automate the identification and creation of threads.

Offload this work from developers to compilers: **implicit threading**.

Advantage of implicit threading

Developer identifies **tasks** that can run in parallel. The environment determines the low-level details of thread creation and management.

Implicit Threading: Thread Pools

Manual thread creation has two problems:

- Thread creation may be costly; threads are discarded once finished.
- Number of threads is unbounded: may exhaust system resources.

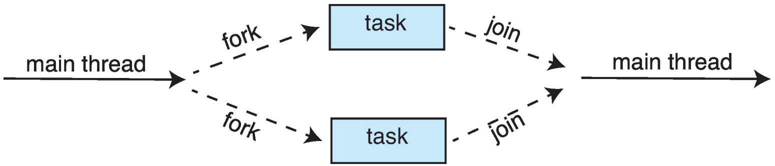
Thread pools

Create a number of threads at startup and make them available for doing work. The application requests resources from the thread pool: if there is a thread available, it is allocated; otherwise wait until a thread is placed back in the pool.

Several advantages of thread pools:

- Servicing a request with existent thread may be faster than creating and deleting threads.
- Thread pool limits the number of threads.
- Thread pool can be configured based on available resources, or adjusted dynamically based on what the applications are doing.

Implicit Threading: Thread Pools



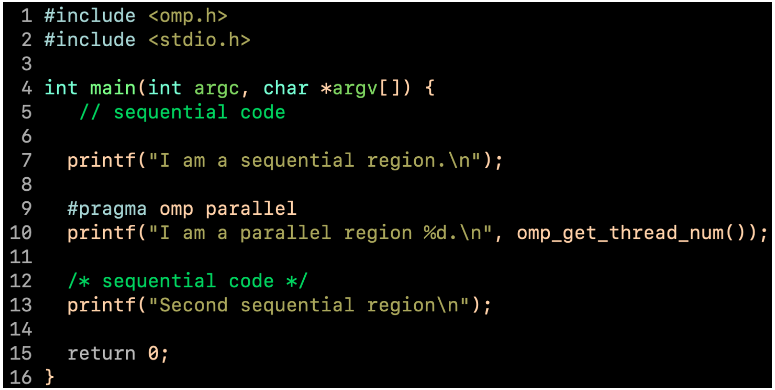
Fork-join model works for implicit threading as well:

- A library manages threads and assignment of tasks to threads.
- Threads are not created directly during fork. Parallel tasks are identified and designated to threads.
- Join mechanism provides the synchronicity.

Implicit Threading: Example Libraries

- Fork-join library available in the Java API.
- **OpenMP** is a way to augment C, C++, Fortran programs to identify what can be parallelized.
- **Grand Central Dispatch** is Apple technology for implicit threading.
- **Intel Thread Building Blocks** supports designing parallel C++ programs.
- **CUDA** allows to program NVIDIA Graphic Cards to perform massively parallel computations.

Implicit Threading: OpenMP example in C



```c
#include <omp.h>
#include <stdio.h>

int main(int argc, char *argv[]) {
   // sequential code

  printf("I am a sequential region.\n");

  #pragma omp parallel
  printf("I am a parallel region %d.\n", omp_get_thread_num());

  /* sequential code */
  printf("Second sequential region\n");

  return 0;
}
```

Implicit Threading: OpenMP example in C

```
#pragma omp parallel for
for (i = 0; i < N; i++) {
   c[i] = a[i] + b[i];
}
```

- Take two arrays a and b of size N. We want to add the array and produce a new array c.
- This is an **embarrassingly parallel** problem.
- OpenMP pragma will divide the work among the threads it creates.
- Different parts of the vectors will be added in parallel.

Other OpenMP features

Developers can choose several levels of parallelism. For example, set the number of threads manually. It also allows to say whether data is shared or private among threads.

Activity: Discuss with Peers (5 minutes)

```
#pragma omp parallel for
for (i = 0; i < N; i++) {
   c[i] = a[i] + b[i];
}
```

- Assume that the number of available cores is 4.
- Take the length of the array $N = 40$.

Questions:

1. What is the maximum number of for loop iterations that will be executed in parallel?
2. How many iterations will each core run, assuming each iteration runs in the same amount of time and all cores start at the same time?

Part IV: Threading Issues

Threading Issues: `fork` and `exec` calls

We have seen `fork()` and `exec()` in a small UNIX system: create a child and load and execute a binary.

This becomes difficult in a multithreaded environment: does `fork` duplicate the thread or the whole set of threads within a process?

Different `fork` functions may be provided to achieve either.

The `exec` call typically works as usual: entire process is replaced by the specified program.

$\boxed{\text{Which \texttt{fork} to use depends on the application.}}$

Threading Issues: Signal Handling

- **Signalling** in UNIX is used to inform processes of events.
-
    - Occurrence of some event generates a signal.
    - The signal is delivered to a process.
    - Process handles the signal.
- Example events: division by zero, illegal memory access, CTRL-C key combination.

$\boxed{\text{Which thread should a signal be delivered to?}}$

Various methods exist, for example: deliver to all threads; assign one thread to deal with signals.

On UNIX:

```
kill(pid_t pid, int signal)
```

Threads will either accept or block the signal. First thread to accept receives it.

Threading Issues: Thread Cancellation

We may want to terminate threads before they complete, called **target threads**.

For example, threads looking through a database for something can stop when one finds the item.

Problems can occur if the target thread is updating shared data while being cancelled.

`pthreads` for example allows threads to disable cancellation for some time.

COMP2211: Progress

Week	Topic
1	~~Introduction to OS~~
2	~~OS services~~
3	~~Processes~~
4	~~Xv6: Live coding and Q&A from the xv6 book~~
5	~~Reading week. No scheduled labs or lectures~~
6	~~Threads and concurrency.~~
7	Scheduling
8	Process synchronisation
9	Memory management
10	Catch up
11	Module review

Progress

Week	Topic
1	Introduction to OS
2	OS services
3	Processes
4	Xv6: Live coding and Q&A from the xv6 book
5	Reading week.
6	Threads and concurrency.
7 (current)	Scheduling
8	Process synchronisation
9	Memory management
10	Catch up
11	Module review

Reading List

We will be mainly using the *Operating System Concepts* (OSC) 10th ed., 2018, and the **4th edition of the xv6 book (XV6)**, 2024. These slides are based on OSC (see the reference list).

Week	Reading materials
1	Chapter 1 OSC. Chapter 1 XV6.
2	Chapter 2 OSC. Chapter 2 XV6.
3	Chapter 3 OSC.
4	Reread Chapters 1–2 XV6.
5	Reread Chapters 1–3 OSC.
6	Chapter 4 OSC.
7 (current)	Chapter 5 OSC.
8	Chapters 6–8 OSC.
9	Chapters 9–10 OSC. Chapter 3 XV6.
10	Reread Chapters 4–6 OSC.
11	Reread Chapters 9–10 OSC.

Objectives

- To introduce **CPU scheduling**, which is the basis for **multiprogrammed** operating systems.
- To describe various **CPU-scheduling algorithms** and understand pros and cons of each.
- To discuss **evaluation criteria** for selecting a CPU-scheduling algorithm for a particular system.
- To understand challenges with scheduling in **multiprocessor** and **real-time systems**.

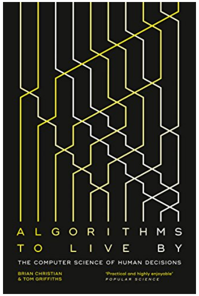
Part I: Description of the Problem

A General Problem

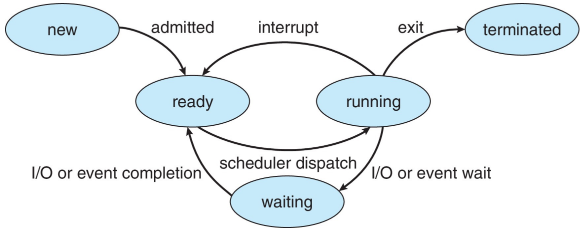
We are going to look at scheduling in operating systems, but it is a general problem (think about where else you can notice scheduling in everyday activities).

"So what to do, and when, and in what order? Your life is waiting."

From Algorithms to Live By, Chapter 5 on Scheduling.

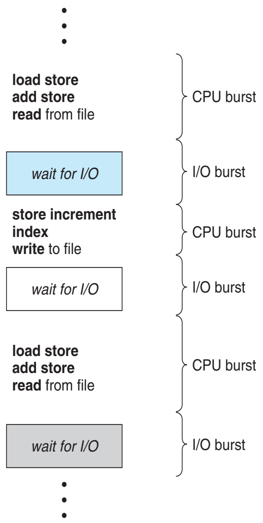

ALGORITHMS
TO LIVE BY
THE COMPUTER SCIENCE OF HUMAN DECISIONS
BRIAN CHRISTIAN & TOM GRIFFITHS
'Practical and highly enjoyable' POPULAR SCIENCE

Why CPUs need scheduling?

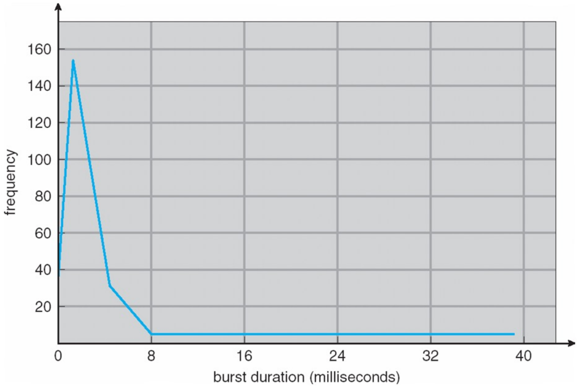


Why CPUs need scheduling?

- Processes go through multiple phases of CPU-IO over their lifetime.
- Maximum CPU utilization through **multiprogramming**.
- When processes wait for IO, CPU can be used for something.
- What to run next? There is a need for **scheduling**.



Typical CPU Burst Lengths



Usually many short and a few long CPU bursts.

Part II: Introduction to Process Scheduling

CPU Scheduler

CPU utilization

When CPU becomes idle, OS finds work (**ready process queue**).

- **CPU scheduler** selects a process from the ready queue and allocates CPU to it.
- Queue may be ordered in various ways.
- CPU scheduling decisions may take place when a process changes state:
  1. running $\rightarrow$ waiting,
  2. running $\rightarrow$ ready,
  3. waiting $\rightarrow$ ready,
  4. terminates.
- For 1 and 4, scheduling is **nonpreemptive** (run as long as needed) while for 2 and 3 **preemptive** (may interrupt a running process).

Challenges with Preemptive Scheduling

A few scenarios that cause problems:

1. Process 1 is writing data, is preempted by process 2 that reads the same data.
2. Process 1 asks kernel to do some important changes, process 2 interrupts while they are being done.

Disabling interrupts

Irrespective of the challenges, most modern operating systems are fully preemptive when running in kernel mode, but disable interrupts on certain small areas of code.

Dispatcher

Dispatcher gives control of the CPU to the scheduled process.

- **Switching context**.
- Switching to **user mode** (kernel tasks in **supervisor mode**).
- Jumping to the proper location in the previously interrupted user program (set the **Program Counter** register).

Dispatch latency

Time it takes for the dispatcher to stop one process and start another running.

Scheduling Criteria

- **CPU utilization**—reduce amount of time CPU is idle.
- **Throughput**—number of processes completed per time unit.
- **Turnaround time**—amount of time to execute a particular process.
- **Waiting time**—amount of time a process has been waiting in the ready queue.
- **Response time**—amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment).

When designing a scheduler

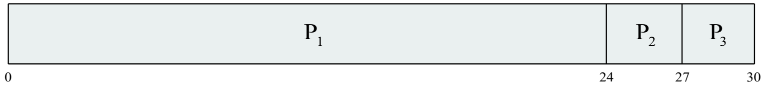
It is desirable to maximize CPU utilization and throughput and to minimize turnaround time, waiting time, and response time.

Part III: Scheduling Algorithms

First-Come, First-Served (FCFS) Scheduling

Process	Burst time
P_1	24
P_2	3
P_3	3

If processes arrive in sequence we have the following schedule:

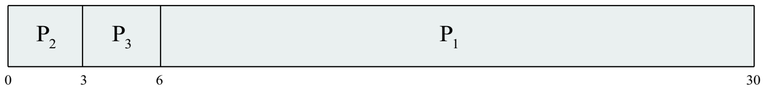


Waiting time for $P_1 = 0$, $P_2 = 24$, and $P_3 = 27$.

Average waiting time: $\frac{0+24+27}{3} = 17$.

First-Come, First-Served (FCFS) Scheduling

If processes arrive instead as P_2, P_3, P_1:



Waiting time for $P_1 = 6$, $P_2 = 0$, and $P_3 = 3$.

Average waiting time: $\frac{6+0+3}{3} = 3$.

Substantial reduction from the previous case but in general not good.

Issue with FCFS

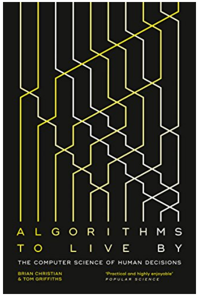
Convoy effect—short jobs can be held waiting by long jobs.

Note that **FCFS is nonpreemptive**.

Questions?

"there's nothing so fatiguing as the eternal hanging of an uncompleted task,"

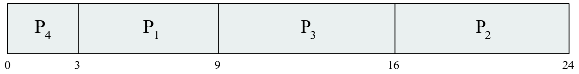
William James. From Algorithms to Live By, Chapter 5 on Scheduling.


ALGORITHMS TO LIVE BY
THE COMPUTER SCIENCE OF HUMAN DECISIONS
BRIAN CHRISTIAN & TOM GRIFFITHS
'Practical and highly enjoyable' POPULAR SCIENCE

Shortest-Job-First (SJF)

- Append each process with the length of next CPU burst.
- Schedule jobs with shortest time.
- SJF is optimal, but difficult to know future CPU burst lengths.
- Ties broken with FCFS scheduling.
- Better name **shortest-next-CPU-burst**.

Process	Next burst time
P_1	6
P_2	8
P_3	7
P_4	3



Average waiting time: $\frac{3+16+9+0}{4} = 7$.

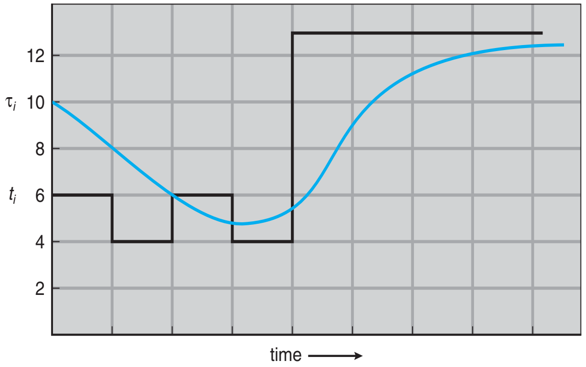
Predicting Lengths of Future CPU Bursts

Make an assumption

Next CPU burst likely similar to the past bursts.

- t_n—actual length of the CPU burst n.
- τ_{n+1}—predicted value of the next burst.
- $0 \leq \alpha \leq 1$.
- $\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$.
- We can tune this model through α (usually set to 0.5).

Example Prediction of CPU Bursts



CPU burst (t_i)		6	4	6	4	13	13	13	...
"guess" (τ_i)	10	8	6	6	5	9	11	12	...

Predicting Lengths of Future CPU Bursts

Model of CPU Burst Lengths

$$\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$$

- $\alpha = 0$, $\tau_{n+1} = \tau_n$—recent history does not count.
- $\alpha = 1$, $\tau_{n+1} = t_n$—only the actual last CPU burst counts.
- Expand the formula:
  $\tau_{n+1} = \alpha t_n + (1-\alpha)\alpha t_{n-1} + \cdots + (1-\alpha)^j \alpha t_{n-j} + \cdots + (1-\alpha)^{n+1}\tau_0$.
- Example: $\alpha = 0.5$, $\tau_4 = 0.5 t_3 + 0.25 t_2 + 0.125 t_1 + 0.0625\tau_0$.

Exponential average of past CPU bursts

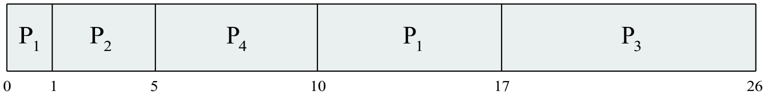
Each successive term has lower weighting than the newer ones, with the initial guess having the lowest.

Shortest-remaining-time-first

If we allow SJF to be preemptive, it can interrupt a currently running process if it would run longer than some new process.

Consider

Process	Arrival time	Next burst time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5



Average waiting time is 6.5—standard SJF would result in 7.75.

Shortest-remaining-time-first (Question, 3min)

Take 3 minutes to schedule the following processes with a **preemptive shortest-job-first scheduler**. Feel free to discuss with your peers. Volunteers for the solution welcome at the end.

Process	Arrival time	Next burst time
P_1	0	8
P_2	1	9
P_3	2	7
P_4	3	2
P_5	4	3

Shortest-remaining-time-first (Question, 3min)

Take 3 minutes to schedule the following processes with a **preemptive shortest-job-first scheduler**. Feel free to discuss with your peers. Volunteers for the solution welcome at the end.

Process	Arrival time	Next burst time
P_1	0	8
P_2	1	9
P_3	2	7
P_4	3	2
P_5	4	3

Answer

P1 runs 0 to 3; P4 interrupts, runs 3 to 5; P5 runs 5 to 8; P1 continues, runs 8 to 13; P3 then runs; finally P2 is run.

Priority Scheduling

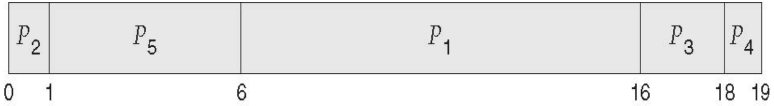
Priority scheduling

Shortest-job-first is a specific case of general scheduler that decides by priorities.

- A priority (integer) associated with each process.
- CPU allocated to a process of highest priority.
- **Starvation**—low priority processes may not execute.
- **Aging**—increase the priority proportional to waiting time.
- **Internal priorities**—time limits, memory requirements, ratio of average I/O burst.
- **External priorities**—importance of the process, type and amount of funds being paid for the CPUs, who is asking to run the process, and other.

Priority Scheduling

Process	Burst time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2



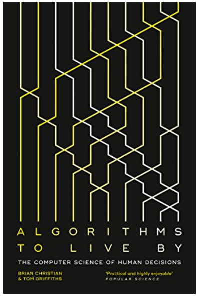
Preemptive priority scheduling

Priorities may change while a process is running.

Questions?

"they wrote up a fix and beamed the new code across millions of miles to Pathfinder. What was the solution they sent flying across the solar system? Priority inheritance."

From Algorithms to Live By, Chapter 5 on Scheduling.


ALGORITHMS TO LIVE BY
THE COMPUTER SCIENCE OF HUMAN DECISIONS
BRIAN CHRISTIAN & TOM GRIFFITHS
'Practical and highly enjoyable' POPULAR SCIENCE

Round Robin (RR) Scheduling

- **Time quantum** (q) is defined.
- CPU scheduler assigns the CPU to each process for an interval of up to 1 quantum.
- Queue treated as First-In-First-Out.
- Interrupts every quantum to schedule next process.
- **RR is therefore preemptive**.
- No process allocated for more than q in a row (unless there is only one).

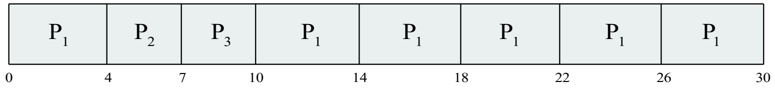
Round Robin (RR) Scheduling

- If there are n processes waiting, each process is guaranteed to get $1/n$ of CPUs time in chunks of time quantum q.
- Each process must wait no longer than $(n-1) \times q$ time units until its next turn to run.

Round Robin (RR) Scheduling

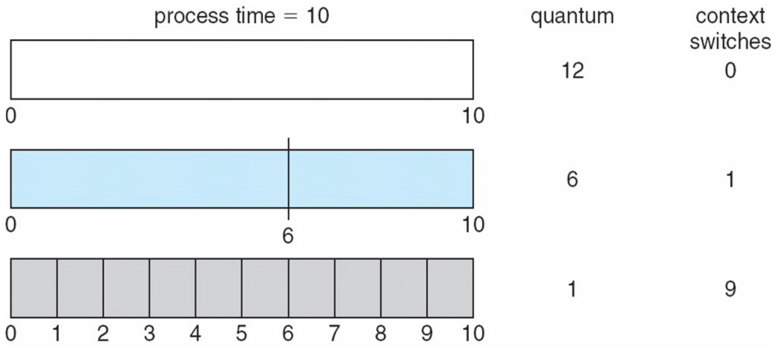
Take $q = 4$.

Process	Burst time
P_1	24
P_2	3
P_3	3



- Small quantum—too many interrupts will reduce performance.
- Big quantum—scheduler similar to FCFS.
- Need a balance (according to OSC, usually $q = 10$ to 100 ms).
- Context switch around 10 microseconds (small fraction of q).

Round Robin (RR) Scheduling

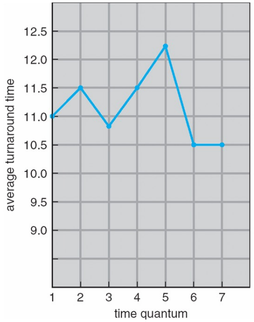


Round Robin (RR) Scheduling

- **Turnaround time** depends on the size of the quantum.
- However, it does not necessarily improve with the size of q.

Rule of Thumb

80% of CPU bursts should be shorter than q.



process	time
P_1	6
P_2	3
P_3	1
P_4	7

Round Robin (RR) Scheduling (Question, 3min)

Take 3 minutes to schedule the following process queue with a **round robin scheduler** with $q = 3$. Feel free to discuss with your peers. Volunteers for the solution welcome at the end.

Process	Burst time
P_1	5
P_2	12
P_3	3
P_4	1

Round Robin (RR) Scheduling (Question, 3min)

Take 3 minutes to schedule the following process queue with a **round robin scheduler** with $q = 3$. Feel free to discuss with your peers. Volunteers for the solution welcome at the end.

Process	Burst time
P_1	5
P_2	12
P_3	3
P_4	1

Answer

P1 runs 0 to 3; P2 runs 3 to 6; P3 runs 6 to 9; P4 runs 9 to 10; P1 runs 10 to 12; P2 runs 12 to 15; P2 15 to 18; P2 runs 18 to 21.

Part IV: Optimizations of Process Scheduling

Multilevel Queue Scheduling

- With previous algorithms, it takes $\mathcal{O}(n)$ to search the queue.
- Assign processes to different queues, by priority.
- Can also assign to queues by process types:
  1. Queue for **background processes** (for example, batch processing)
  2. Queue for **foreground processes** (interactive)
- Each queue can have different scheduling algorithms, depending on needs.
- Scheduling may be required among queues: commonly fixed-priority preemptive scheduling.

Multilevel Queue Scheduling

Example queues in decreasing priority level:

1. Real-time precesses
2. System processes
3. Interactive processes
4. Batch processes

Multilevel priority queue

No process in a lower priority queue runs while there are processes waiting in the higher priority queues. High priority queues preempt lower priority ones.

Time slicing

Another possibility is to allocate time among queues. Example: 80% to foreground queue and 20% to the background queue.

Multilevel Feedback Queue Scheduling

Dynamic queueing

Instead of fixing processes to queues, allow them to move.

Multilevel feedback queue defined by

- number of queues,
- a scheduling alg. for each queue,
- a method to upgrade a process to higher priority queue,
- a method to downgrade a process, and
- a method to determine which queue to assign process at the start.

Multilevel feedback queue

Most general CPU scheduling algorithm due to many parameters in the definition.

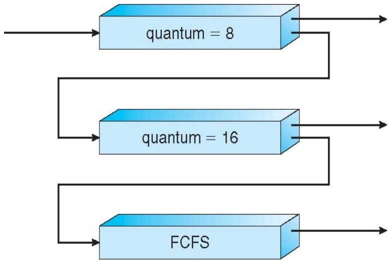
Multilevel Feedback Queue Scheduling (Example)

Three queues (from the top):

- Q0—RR with $q = 8$ ms.
- Q1—RR with $q = 16$ ms.
- Q2—FCFS.

Scheduling:

1. A new job enters Q0 and gets 8 ms.
2. Not finished in 8 ms—move to Q1.
3. Not finished in queue 1 in another 16 ms—move to Q2.
4. Scheduled in FCFS in Q2 when queue 0 and 1 empty.



Starvation in Q2

To prevent starvation we may move old processes to Q0/1.

Advantages and Disadvantages of Scheduling Algorithms

Algorithm	(dis)advantages
FCFS	Convoy effect a problem—long jobs hold the queue.
SJF	Need to predict future CPU burst lengths.
Preemptive SJF	Better average waiting time than SJF.
Priority scheduler	Starvation.
RR	Need to tune time quantum to avoid expensive context switch.
Multilevel queue	Faster search than $\mathcal{O}(n)$.
Multilevel feedback queue	Configuration can be expensive. Starvation.

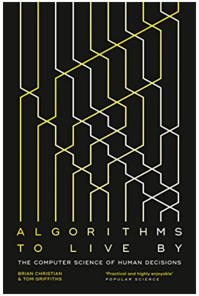
Practice

There is no perfect algorithm for all cases. It is a tradeoff based on requirements of the system and usually a combination of scheduling algorithms is implemented (See OSC OS examples [1, Sec. 5.7]).

Questions?

"In fact, the weighted version of Shortest Processing Time is a pretty good candidate for best general-purpose scheduling strategy in the face of uncertainty."

From Algorithms to Live By, Chapter 5 on Scheduling.


ALGORITHMS TO LIVE BY
THE COMPUTER SCIENCE OF HUMAN DECISIONS
BRIAN CHRISTIAN & TOM GRIFFITHS
'Practical and highly enjoyable' POPULAR SCIENCE

Vevox quiz

Part V: Remarks on Multi-Processor Scheduling

Multi-Processor Scheduling

Traditionally term **multi-processor** referred to systems with multiple physical cores. Now we use it to describe systems with either several physical or virtual cores/threads.

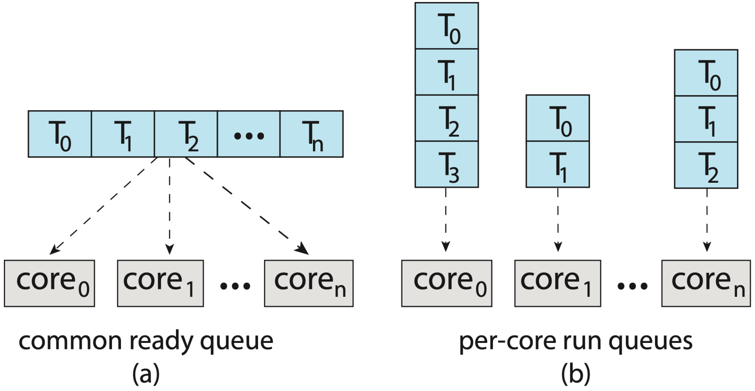
One approach to scheduling is to have one master processor handling scheduling (**assymetric multiprocessing**). Master becomes potential bottleneck.

Another is **symmetric multiprocessing (SMP)**—each processor handles its scheduling. Most common (Windonws, Linux, macOS, Android, iOS).

Multi-Processor Scheduling: SMP

Two approaches in SMP

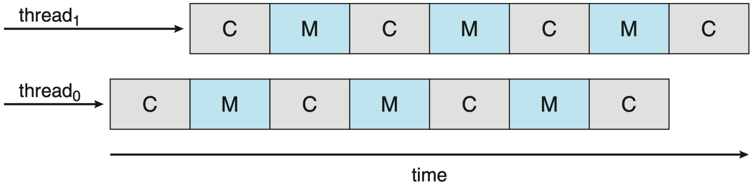
1) Common ready queue—each processor takes processes/threads from that queue (potential clashes). 2) Each processor has its own queue.



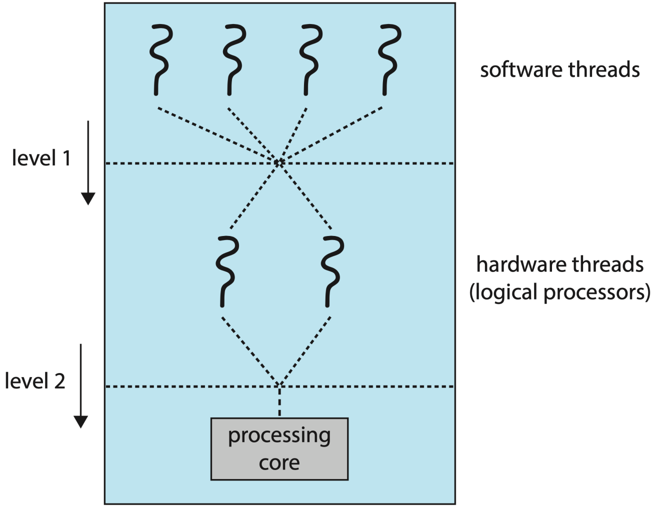
T_0 T_1 T_2 ••• T_n
core_0 core_1 ••• core_n
common ready queue
(a)
T_0 T_1 T_2 T_3
T_0 T_1
T_0 T_1 T_2
core_0 core_1 ••• core_n
per-core run queues
(b)

Multicore Processors

- Relatively recent trend is to place multiple cores on chip (**multicore**).
- Speed and energy efficiency.
- **Memory stall**—cores spend significant amount of time for memory (since these days cores are much faster than memory).
- **Multithreading**—hardware assisted mutliple threads per core.
- When one thread is in memory stall, work on another.
- OS sees different hardware threads as separate CPUs.



Multicore Processors: Two Levels of Scheduling



Load balancing

- With SMP we need to utilize all CPUs efficiently.
- Load balancing attempts even distribution.
- Only necessary on systems with separate queues for each CPU.
- **Push migration**—a task checks the load on each CPU and moves threads from CPU to CPU to avoid imbalance.
- **Pull migration**—idle processor pulls waiting tasks from busy processors.

Processor Affinity

- When a thread runs on a core, the cache is “warmed up” for that thread.
- We say that a task has affinity for the processor it’s running on.
- When a task is moved, say due to load balancing, we have a big overhead in terms of cache.
- Invalidating and repopulating caches is expensive.
- **Soft affinity**—OS will attempt to keep the process on the same core, but load balancing can move it.
- **Hard affinity**—processes specify a list of processes on which to run.
- Usually both methods are available.

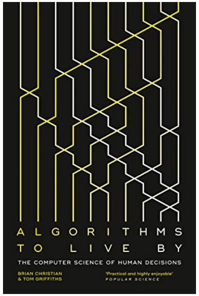
Implications on scheduling

Load balancing and processor affinity both may have implications on scheduling.

Questions?

"the Linux core team, several years ago, replaced their scheduler with one that was less "smart" about calculating process priorities but more than made up for it by taking less time to calculate them."

From Algorithms to Live By, Chapter 5 on Scheduling.


ALGORITHMS TO LIVE BY
THE COMPUTER SCIENCE OF HUMAN DECISIONS
BRIAN CHRISTIAN & TOM GRIFFITHS
'Practical and highly enjoyable' POPULAR SCIENCE

Part VI: Scheduling with Deadlines: Real-Time Processing

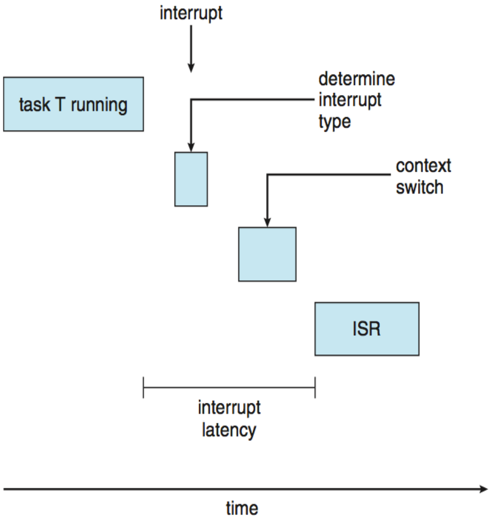
Real-Time CPU Scheduling

- Real-time systems categorized into two:
  1. **Soft real-time**: guarantee preference for critical processes.
  2. **Hard real-time**: guarantee completion by deadline.
- Two types of latencies affect performance:
  1. **Interrupt latency**: time from arrival to interrupt service routine.
  2. **Dispatch latency**: time for dispatcher to stop current process and start another.

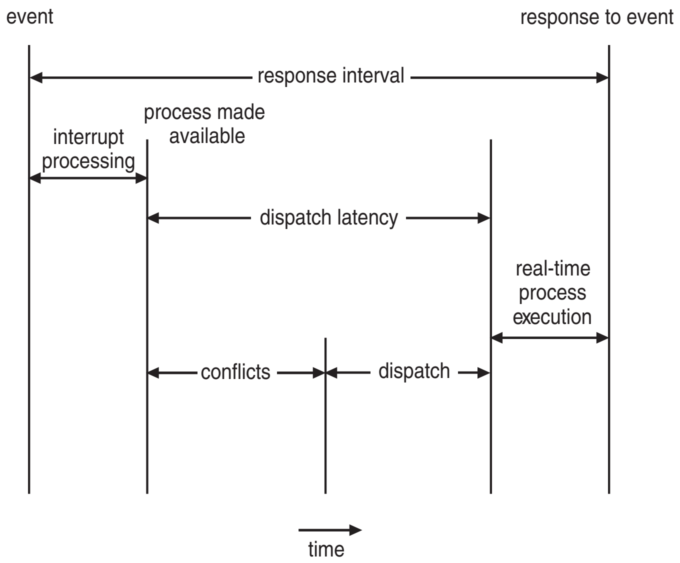
Hard real-time systems

Various latencies should be bounded to meet the strict requirements of these systems.

Real-Time CPU Scheduling



Real-Time CPU Scheduling



Priority-Based Scheduling

Real-time systems

It is essential to have a priority-based preemptive scheduling for real-time systems. Usually real-time processes have highest priority.

Priority-based preemptive scheduling gives us soft real-time functionality.

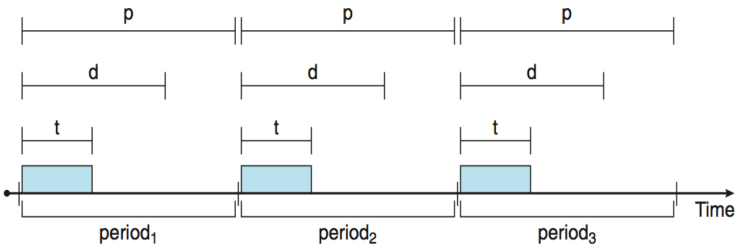
Additional scheduling features required for hard real-time.
Some definitions:

- Processes are periodic—require CPU at constant intervals.
- Processing time t, deadline d, period p. Here $0 \leq t \leq d \leq p$.

Admission control

Schedulers take advantage of these details and assign priorities based on deadlines and period. **Admission control** algorithm may reject the request as impossible to service by the required deadline.

Priority-Based Scheduling



Rate-Monotonic Scheduling

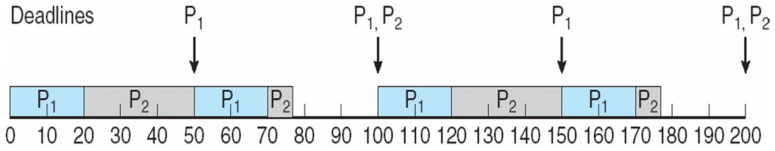
- Upon entering the system, each periodic task assigned priority $\propto \frac{1}{p}$.
- Rationale: prioritize processes that require CPU more often.

Example:

Process	p	t	d
P_1	50	20	50
P_2	100	35	100

P_1 has higher priority since the period is shorter.

Rate-monotonic scheduler:



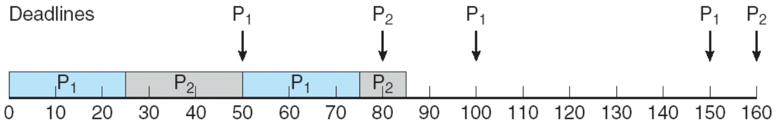
Rate-Monotonic Scheduling

Now we make the requirements more strict for P_2:

Process	p	t	d
P_1	50	25	50
P_2	80	35	80

P_1 has higher priority since the period is shorter.

Rate-monotonic scheduler:

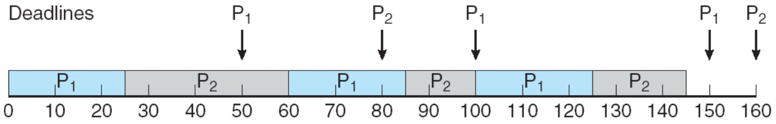


P_2 failed to complete by $d = 80$! The total CPU utilization is $25/50 + 35/80 = 0.94$, but the problem was that the scheduler starts P_1 again before P_2 completes.

Earliest-Deadline-First Scheduling

I think you have been using this one in the past weeks! ☺

Priorities not fixed in advance—the earlier the deadline, the higher priority.



At time 50 process P_2 is not preempted by P_1 because its next deadline (80) is earlier than process P_1's next deadline at time 100.

EDF Scheduling

No requirement of the period, just the deadline, therefore processes do not need to be periodic as with rate-monotonic scheduling.

Earliest-Deadline-First Scheduling (Question, 5min)

Take 5 minutes to schedule the following process queue with a **Earliest-Deadline-First Scheduling**. Feel free to discuss with your peers. Volunteers for the solution welcome at the end.

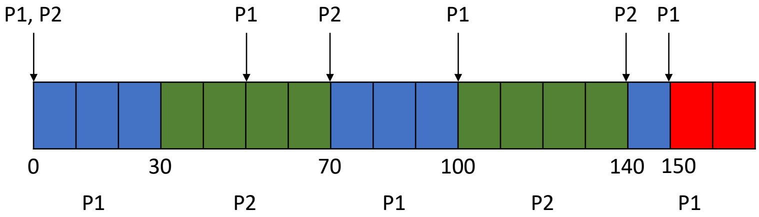
Process	p	t	d
P_1	50	30	50
P_2	70	40	70

Don't forget the aforementioned **admission control**.

Earliest-Deadline-First Scheduling (Question, 5min)

Take 5 minutes to schedule the following process queue with a **Earliest-Deadline-First Scheduling**. Feel free to discuss with your peers. Volunteers for the solution welcome at the end.

Process	p	t	d
P_1	50	30	50
P_2	70	40	70



Scheduling in XV6

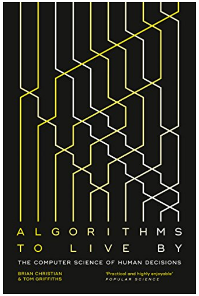
Scheduling occurs in two situations:

- Running process runs `sleep` or `wait`.
- XV6 periodically forces scheduling (round-robin with quantum of $\sim$ 100 ms).
- Scheduler exists as a separate thread per CPU.
- Queue of up to 64 processes available.
- See `kernel/proc.c` for further detail. Scheduler in the function `void scheduler(void)`.

Questions?

“there’s no choice but to treat that unimportant thing as being every bit as important as whatever it’s blocking.”

From Algorithms to Live By, Chapter 5 on Scheduling.


ALGORITHMS TO LIVE BY
THE COMPUTER SCIENCE OF HUMAN DECISIONS
BRIAN CHRISTIAN & TOM GRIFFITHS
'Practical and highly enjoyable' POPULAR SCIENCE

COMP2211: Progress

Week	Topic
1	~~Introduction to OS~~
2	~~OS services~~
3	~~Processes~~
4	~~Xv6: Live coding and Q&A from the xv6 book~~
5	~~Reading week. No scheduled labs or lectures~~
6	~~Threads and concurrency.~~
7	~~Scheduling~~
8	Process synchronisation
9	Memory management
10	Catch up
11	Module review

Progress

Week	Topic
1	Introduction to OS
2	OS services
3	Processes
4	Xv6: Live coding and Q&A from the xv6 book
5	Reading week.
6	Threads and concurrency.
7	Scheduling
8 (current)	Process synchronisation
9	Memory management
10	Catch up
11	Module review

Reading List

We will be mainly using the *Operating System Concepts* (OSC) 10th ed., 2018, and the **4th edition of the xv6 book (XV6)**, 2024. These slides are based on OSC (see the reference list).

Week	Reading materials
1	Chapter 1 OSC. Chapter 1 XV6.
2	Chapter 2 OSC. Chapter 2 XV6.
3	Chapter 3 OSC.
4	Reread Chapters 1–2 XV6.
5	Reread Chapters 1–3 OSC.
6	Chapter 4 OSC.
7	Chapter 5 OSC.
8 (current)	Chapter 6 OSC (Chapters 7 and 8 additional).
9	Chapters 9–10 OSC. Chapter 3 XV6.
10	Reread Chapters 4–6 OSC.
11	Reread Chapters 9–10 OSC.

Objectives

- Discuss why we need process synchronisation.
- Present various solutions: hardware and API level.
- Discuss new challenges that those solutions introduce.

Part I: Description of the Problem

Process Synchronisation: Motivation

- By now we know that OS typically consists of **many processes/threads** running either **concurrently** or **in parallel**.
- Threads often share data.
- OS continually updates various data structures to support multithreading.

$\boxed{\text{Multiple threads may want to update shared data at the same time.}}$

- If access to shared data is not controlled, we may get corrupted data values.
- **Process synchronisation** involves methods for control of access to shared data to avoid such issues.
- This week we will learn to **recognize** the need for process synchronisation.

Process Synchronisation: A Few Reminders

Recall **cooperating processes** that can affect or be affected by other processes.

Shared data through shared memory or message passing. Concurrent access may cause data inconsistency.

Last week we studied **scheduling**, which is the key in achieving concurrency:

- Scheduler rapidly switches between processes.
- One process may be interrupted at any time by another.
- Processes in reality may be interrupted hundreds of times, pausing what they are doing.

Parallelism involves multiple instruction streams running at the same time.

Example 2-thread code in Pthreads: who can spot a problem?

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

int x = 0;
void *runner1();

int main(int argc, char *argv[]) {
  pthread_t tid1, tid2;
  pthread_attr_t attr;

  // Create two threads.
  pthread_attr_init(&attr);
  pthread_create(&tid1, &attr, runner1, NULL);
  pthread_create(&tid2, &attr, runner1, NULL);

  // Wait for the threads to exit.
  pthread_join(tid1,NULL);
  pthread_join(tid2,NULL);

  printf("x = %d\n",x);
}

void *runner1() {
  x=x+1;
  pthread_exit(0);
}
```

Race conditions

A **Race condition** arises when several processes manipulate data concurrently.

Outcome of execution depends on a particular order.

It is usually difficult to predict the order of execution and *it may change on different runs* of a multithreaded application.

In our example we need to make sure that only one thread can manipulate `x` at any time.

We need some way of **synchronizing processes/threads**.

Race conditions

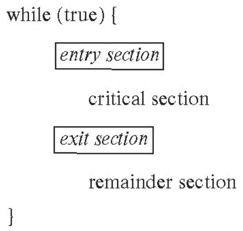
- **Race conditions** can arise in OS as different parts manipulate shared resources.
- **Race conditions** also arise in multithreaded user applications.
- Increasing use of multicore systems makes this an important problem.
- Applications are being developed to run in parallel on many cores, sharing data among different parts.
- Mechanisms to secure against **race conditions** is an important part of the systems these days.

The Critical-Section Problem

Consider a system with n processes/threads $P_0, P_1, \ldots, P_{n-1}$.

- Each process has a **critical section** of code.
- In that section, *shared data* is being accessed (shared among at least two processes).
- When a process is executing instructions in the **critical section**, no other process can do so.
- Each process *must request permission* to enter their critical section.

  - The **entry section** implements the request.
  - The **exit section** may do some tidying up after the critical section.
  - The rest of the code is called the **remainder section**.



```
while (true) {
        entry section
            critical section
        exit section
            remainder section
}
```

The Critical-Section Problem

A solution to this problem should meet the following.

1. **Mutual exclusion**: Only one P_i can be in a critical section.
2. **Progress**: If one process asks to execute its *critical section*, only processes not in their *remainder section* can participate.
3. **Bounded waiting**: There should be a limit on a number times other processes can enter their *critical sections*, when some other process is waiting. Avoid the problem of process **starvation** (see W7 on scheduling).

Part II: Software Solutions to Process Synchronisation

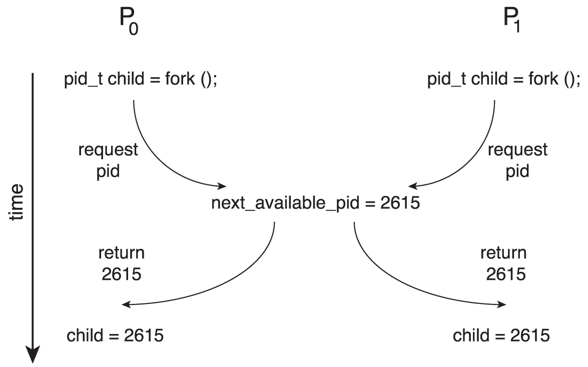
Race Conditions in the Kernel: File Opening/Closing

Consider a *kernel data structure* that maintains **a list of open files** in the system.

It must be modified when a new file is opened/closed.

If two processes open/close files simultaneously, there may be a **race condition** on this data structure.

Race Conditions in the Kernel: Getting PIDs



The Critical-Section Problem

The **critical-section problem** can be solved in single-core environment by disabling interrupts during the execution of critical code.

One sequence of instructions would be run and we would know that nothing interferes with the section that modifies the *shared data*.

In a multiprocessor environment this is not going to work since two or more cores can write to a shared location at the same time.

The Critical-Section Problem in the Kernel

Two approaches are used in kernels:

1. **Preemptive kernels**
2. **Nonpreemptive kernels**

The latter does not allow processes to be interrupted while they are in kernel mode. Safe from race conditions on kernel data structures.

Preemptive kernels need to solve the critical-section problem.

Preemptive vs nonpreemptive kernels

Why would anyone favour preemptive kernels if they have this problem? More responsive since there is less risk of lengthy processes holding the CPU.

Basic Software Solution to the Critical-Section Problem

- **Peterson's solution**.
- No guarantee this works on modern architectures.
- Good starting point to understand solutions.
- Restricted to two processes: P_0 and P_1.
- When talking about P_i we use P_j to refer to the other process.

```
while (true){

        flag[i] = true;
        turn = j;
        while (flag[j] && turn = = j)
                ;

           /* critical section */

        flag[i] = false;

        /* remainder section */

}
```

Basic Software Solution to the Critical-Section Problem

- Variables `turn` and `flag` shared.
- `turn` indicates whose turn it is to enter the crit. sec.
- `flag[i]` indicates whether process *i* is ready to enter the crit. sec.
- All 3 conditions of the crit. sec. problem solution are met.
- See OSC p. 263 for proof.

```
while (true){

        flag[i] = true;
        turn = j;
        while (flag[j] && turn = = j)
                ;

            /* critical section */

        flag[i] = false;

        /* remainder section */

}
```

Peterson's Solution on Modern Architectures: Instruction Reordering

- This may not work on modern architectures due to **instruction reordering**.
- Instruction reordering is performed when there are no **data dependencies** between them.
- On single-threaded applications reordering does not impact the final result.
- On multi-threaded applications it may change the final result.
- See the example on the right: last two lines in Thread 2 may be reordered.

Consider shared data between two threads:

```
boolean f = false
int x = 0
```

Thread 1:

```
while (!f)
    ;
print x;
```

Thread 2:

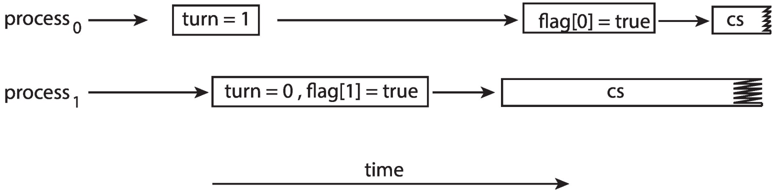
```
x = 1;
f = true;
```

Peterson's Solution on Modern Architectures: Instruction Reordering

- Consider what happens if the first two lines of the while loop are reodered.

```
while (true){

      flag[i] = true;
      turn = j;
      while (flag[j] && turn = = j)
            ;

         /* critical section */

      flag[i] = false;

      /* remainder section */

}
```

Peterson's Solution on Modern Architectures: Instruction Reordering



Vevox Quiz

Part III: Hardware Solutions to Process Synchronisation

Hardware Support for Synchronisation

Software-based solutions fall short in solving the synchronisation problems arising in shared data among threads.

Reason: a programmer provides code in certain order; modern architectures reorder for performance, when there are no data dependencies.

We will now look at three hardware instructions created specifically for this problem.

Hardware Support for Synchronisation: Memory Barriers

Memory model

How a computer architecture determines what memory guarantees it will provide.

Two memory model categories:

1. **Strongly ordered**: memory modifications by one processor are known to other processors *immediately*.
2. **Weakly ordered**: mem. modifications *may not be immediately* visible.

Kernel developer assumptions on memory models

Memory models vary; developers cannot assume what visibility of memory modifications will there be in a shared-memory multiprocessor.

Hardware Support for Synchronisation: Memory Barriers

This issue is addressed by computer architectures providing instructions that force changes in shared-memory to be propagated to all processors.

This way all threads running on other processors will know about the memory modifications.

Memory barriers/fences

When a processor meets a **memory barrier**, it makes sure that any memory operations are completed before starting any subsequent ones (even if reordering has been taking place). This way other threads see the latest data.

Hardware Support for Synchronisation: Memory Barriers

```
boolean f = false
int x = 0
```

Thread 1:

```
while (!f)
    ;
print x;
```

Thread 2:

```
x = 1;
f = true;
```

```
boolean f = false
int x = 0
```

Thread 1:

```
while (!f)
    barrier();
print x;
```

Thread 2:

```
x = 1;
barrier();
f = true;
```

Example on the right uses memory barriers to synchronize read/write of `x`.

Hardware Support for Synchronisation: Memory Barriers

Let's return to **Peterson's critical-section problem solution**. Threads 1 and 2 run the following code and share `flag` and `turn`. Does not work if first two writes reordered, so add a barrier there.

```
while (true) {
  flag[i] = true;
  turn = j;
  while (flag[j] && turn == j)
    ;

    /* critical section */

  flag[i] = false;

    /* remainder section */
}
```

```
while (true) {
  flag[i] = true;
  barrier();
  turn = j;
  while (flag[j] && turn == j)
    ;

    /* critical section */

  flag[i] = false;

    /* remainder section */
}
```

Hardware Support for Synchronisation: Hardware Instructions

Modern hardware provides special hardware instructios that allow:

1. **Test-and-modify** contents of memory.
2. **Compare-and-swap** two words of memory.

These instructions allow us to resolve the critical section problem.

Hardware Support for Synchronisation: Test-and-set instruction

```
boolean test_and_set(boolean *target) {
  current_val = *target;
  *target = true;
  return current_val;
}
```

- This is a definition of the instruction. Implemented in hardware.
- These steps happen **atomically—test-and-set** cannot be interrupted.
- In a multiprocessor, **test-and-set** also happens sequentially in arbitrary order.
- If `target` stores 1, we will keep keep it unchanged.
- If `target` stores 0, we will return 0 but change `target` to 1.

Hardware Support for Synchronisation: Test-and-set instruction

Test-and-set can be used to implement *mutual exclusion* (critical-section problem requirement 1).

Each thread initializes a shared `lock=0`. Only one thread can get the lock and execute its critical section.

```
do {
  while (test_and_set(&lock))
    ;

  /* critical section */

  lock = 0;

  /* remainder section */
} while (true);
```

Hardware Support for Synchronisation: Compare-and-swap instruction

```
int compare_and_wap(int *val, int expected, int new) {
  int temp = *val;

  if (*val == expected)
    *val = new;

  return temp;
}
```

- This is a definition of the instruction. Implemented in hardware.
- These steps happen **atomically—compare-and-set** cannot be interrupted.
- In a multiprocessor, **compare-and-set** also happens sequentially in arbitrary order.
- Set `val` to `new` only if the value is what we expected.

Hardware Support for Synchronisation: Compare-and-swap instruction

Compare-and-swap can be used to implement *mutual exclusion*.

Each thread initializes a shared `lock=0`. Only one thread can swap the lock to 1 and execute its critical section.

```
while (1) {
  while (compare_and_swap(&lock, 0, 1) != 0)
    ;

  /* critical section */

  lock = 0;

  /* remainder section */
}
```

- First process to call **compare-and-swap** will set `lock=1`.
- Then it will enter its critical section because comparison returned 0.
- Other calls to **compare-and-swap** won't succeed since lock now is not equal to the expected value of 0.
- The process exiting its critical section will release the lock, allowing others to execute their critical sections.

Hardware Support for Synchronisation

Remember the three requirements of the critical-section problem solution:

1. **Mutual exclusion**: Only one P_i can be in a **critical section**.
2. **Progress**: If one process asks to execute its **critical section**, only processes not in their **remainder section** can participate.
3. **Bounded waiting**: There should be a limit on a number times other processes can enter their critical sections, when some other process is waiting. Avoid the problem of process **starvation** (see W7 on scheduling).

Test-and-set and compare-and-swap

The solutions presented above do not meet the **bounded waiting requirement**: a thread may be stuck at the atomic instructions, waiting, while other threads keep getting the lock. See OSC for a solution.

Hardware Support for Synchronisation: Atomic Variables

compare-and-swap instruction is often used to build other tools for synchronisation.

Atomic variables provide atomic operation on basic data types; only one thread at a time can modify them.

They can be used to avoid **race conditions** on single shared variables, when multiple threads are updating them.

Most systems that support **atomic variables** also provide *atomic data types* and operations on them.

Hardware Support for Synchronisation: Atomic Variables

Below we show how an integer can be atomically incremented using **compare-and-swap**.

```
void increment(atomic_int *var) {
  int temp;
  do {
    temp = *var;
  } while (temp != compare_and_swap(var, temp, temp+1));
}
```

Compare-and-swap will only increment `temp` when it hasn't changed since we set it.

Atomic variables

These variables are useful in operating systems for limited uses, such as updating single-variable features like counters.

Part IV: Other Solutions to Process Synchronisation

Other Solutions to Process Synchronisation

Hardware-based solutions are complicated and too low-level for application programmers to access.

Higher-level software tools are usually available in operating systems.

- **Mutex locks**
- **Semaphores**

Mutex Locks

- A process must acquire a mutex lock before entering a critical section.
- It releases the lock when it exits the critical section.

```
while (true) {
  /* acquire lock */
     critical section
  /* release lock */
  remainder section
```

Mutex Locks

The `acquire()` function acquires the lock:

```
acquire() {
  while (!available)
    ; /* busy wait */
  available = false;
}
```

The `release()` function releases it:

```
release() {
  available = true;
}
```

Implementations must assure calls to these are *atomic*.

Example 2-thread code in Pthreads: Use of a Mutex

Modify our example code on Slide 8 to use mutex locks.

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

int x = 0;
void *runner1();
pthread_mutex_t x_mutex;

int main(int argc, char *argv[]) {
  pthread_t tid1, tid2;
  pthread_attr_t attr;
  pthread_mutex_init(&x_mutex, NULL);

  // Create two threads.
  pthread_attr_init(&attr);
  pthread_create(&tid1, &attr, runner1, NULL);
  pthread_create(&tid2, &attr, runner1, NULL);

  // Wait for the threads to exit.
  pthread_join(tid1,NULL);
  pthread_join(tid2,NULL);

  printf("x = %d\n",x);
}

void *runner1() {
  pthread_mutex_lock(&x_mutex);
  x=x+1;
  pthread_mutex_unlock(&x_mutex);
  pthread_exit(0);
}
```

Semaphores

Mutex lock disadvantages

Mutex locks require **busy waiting**: while a process is in its **critical section**, other processes loop continuously trying to acquire the lock, until the process holding it releases it.

Another method can provide more sophisticated ways for synchronisation:

- A **semaphore** S is an integer variable.
- S is only accessed through atomic operations `wait()` and `signal()`.

```
wait(S) {
  while (S <= 0)
      ;
  S--;
}
```

```
signal(S) {
  S++;
}
```

Semaphores: Use for Access Control to Resources

We can use semaphores to control access to a limited number of resources.

- Initialize S to the number of resources available.
- Processes that wish to use one of the resources call `wait()`: decrement S.
- Processes that release resources call `signal()`: increment S.
- When S=0 all resources have been taken and processes need to block until some become available.
- Because semaphore modifications are atomic, no two processes can capture the same resource.

Semaphores: Use for Synchronisation

- Take two concurrent processes P_1 and P_2.
- Each has some code S_1 and S_2, respectively.
- Suppose we want $S1$ to go first followed by $S2$ (not guaranteed when code runs in parallel).
- Create a semaphore `synch=0`.

In P_1 we do:

```
S1;
signal(synch);
```

In P_2 we do:

```
wait(synch);
S2;
```

Implementing Semaphores

- When `wait()` is called and the process must wait for the semaphore, it will pause itself (**no busy waiting**).
- The process goes to the *waiting queue* and the scheduler selects another.
- The process is restarted when some other process calls `signal()`.
- Goes from the *waiting queue* to the *ready queue*.
- The scheduler may pick it up for running on the CPU.

See OSC Section 6.6.2 for the pseudo code that achieves this.

Vevox Quiz

Part V: Other Problems in Synchronisation

Liveness

- Using synchronisation tools to coordinate processes introduces a possibility for processes to *wait indefinitely*.
- Recall three requirements for a solution to the critical-section problem.
- **Mutual exclusion**, **progress**, **bounded waiting**.
- Waiting for an undefined amount of time for a lock violates some of these.
- **Liveness**: a system must ensure that processes can make progress.
- A process waiting indefinitely means our operating system produces a *liveness failure*.

$\boxed{\text{Providing semaphores and mutex locks opens up a possibility for liveness failure}}$

Liveness: Deadlock (Initially S=Q=1)

P_0

```
wait(S)
wait(Q)
    .
    .
    .
signal(S)
signal(Q)
```

P_1

```
wait(Q)
wait(S)
    .
    .
    .
signal(Q)
signal(S)
```

Deadlock

Every process in the set is waiting for an event that can only be caused by another process in that same set. Since that other process is also waiting, no progress will be made and the set is **deadlocked**.

See OSC Chapter 8 for further detail (optional reading).

Liveness: Priority Inversion

- Consider a scenario where a higher-priority process needs to read/modify kernel data held by a mutex lock by a lower-priority process.
- High priority process waits for lower-priority one: a scheduling problem (see **priority-based schedulers** in Week 7).

Priority inversion example

Take processes A, B, C with priorities $A > B > C$. Imagine that process A wants a semaphore S, which is held by C. Then imagine that process B preempts C and gets scheduled since it is of higher priority. B has affected how long a higher-priority process A must wait for the semaphore.

- Avoid by implementing a **priority-inheritance protocol**.
- If a process accesses a resource needed by a higher-priority process, it inherits that higher-priority.
- The priority is reduced back to the original priority when the resource is released.

COMP2211: Progress

Week	Topic
1	~~Introduction to OS~~
2	~~OS services~~
3	~~Processes~~
4	~~Xv6: Live coding and Q&A from the xv6 book~~
5	~~Reading week. No scheduled labs or lectures~~
6	~~Threads and concurrency.~~
7	~~Scheduling~~
8	~~Process synchronisation~~
9	Memory management
10	Catch up
11	Module review

Progress

Week	Topic
1	Introduction to OS
2	OS services
3	Processes
4	Xv6: Live coding and Q&A from the xv6 book
5	Reading week. No scheduled labs or lectures
6	Threads and concurrency.
7	Scheduling
8	Process synchronisation
9 (current)	Memory management
10	Catch up
11	Module review

Objectives

- Discuss why memory management is required in Operating Systems.
- Introduce **paging**.
- Introduce **virtual memory**.

Reading List

We will be mainly using the *Operating System Concepts* (OSC) 10th ed., 2018, and the **4th edition of the xv6 book (XV6)**, 2024. These slides are based on OSC (see the reference list).

Week	Reading materials
1	Chapter 1 OSC. Chapter 1 XV6.
2	Chapter 2 OSC. Chapter 2 XV6.
3	Chapter 3 OSC.
4	Reread Chapters 1–2 XV6.
5	Reread Chapters 1–3 OSC.
6	Chapter 4 OSC.
7	Chapter 5 OSC.
8	Chapters 6–8 OSC.
9 (current)	Chapters 9–10 OSC. Chapter 3 XV6.
10	Reread Chapters 4–6 OSC.
11	Reread Chapters 9–10 OSC.

Part I: Description of the Problem

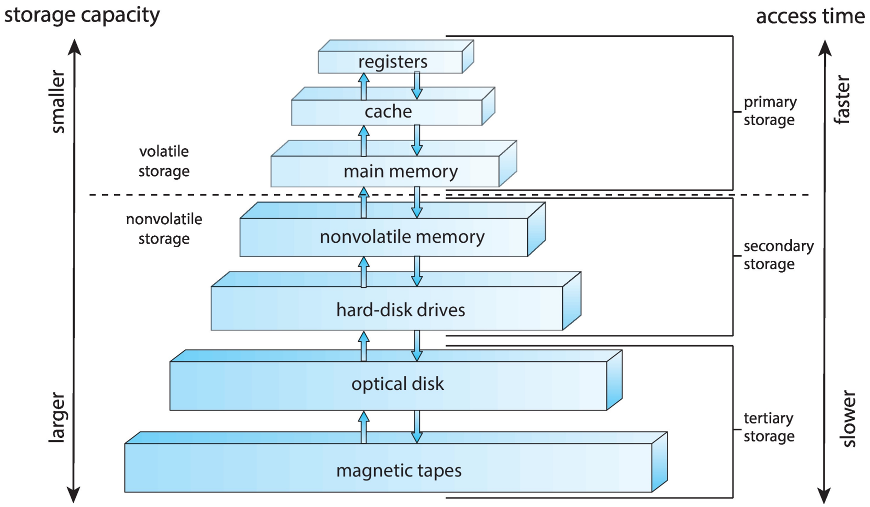
Introduction

Memory operations

CPU can only access registers and main memory directly, with cache in between.

- Programs loaded from disk to memory, within process' memory structure.
- CPU sends to the memory unit either
  - address and read requests, or
  - address, data, and write requests.
- Register access 1 clock cycle.
- Main memory: multiple cycles (LD/ST instructions). Causing a **stall** in CPU.
- Cache helps reduce stall times.

Introduction

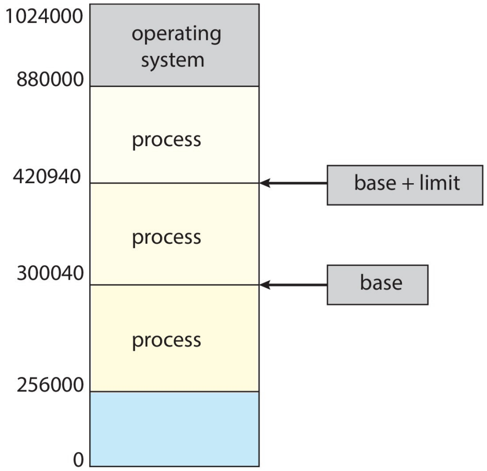


Memory protection

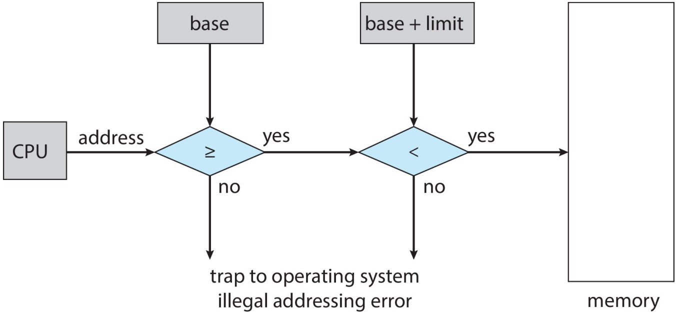
Why we need memory protection

Processes should only be able to access their addresses space. We do not want them to be able to impact each other (or the OS) directly.

- Each process memory is limited by the `limit`.
- Addresses have to lie between a limited range, starting at `base` address.
- Error if `address` > `base+limit`.



Memory protection



Base and limit registers

These registers can be accessed only by **privileged instructions** in kernel mode. Only the OS can modify them, protecting users from modifying the size of the address space.

Address binding

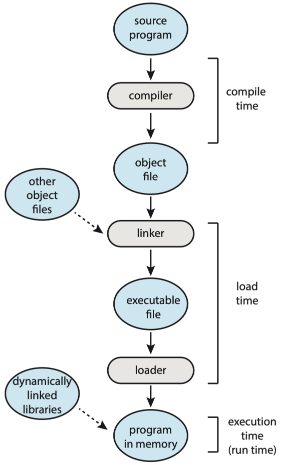
- Programs on disk have to be moved into memory eventually, for execution.
- We will place them in some location, not necessarily at address 0000.
- How to represent addresses before the decision of placement is made?
- Source programs contain **symbolic addresses** that the compiler bind to **relocatable addresses**, relative to some reference address that is set later.
- Linker or loader will bind these to absolute addresses.

Address binding

Address binding can happen at various stages of program lifetime:

- Compilation: if placement memory location is known, compiler can produce absolute addresses within the binary. Recompilation needed if location changes.
- Loading: take **relocatable code** from the compiler and transfer relative addresses to absolute addresses.
- Execution: if processes can move in memory during execution, then address binding has to be delayed until this time. This is a most common set-up in operating systems today.

Address binding



Logical and physical address spaces

Logical address (virtual address)

Generated by the CPU.

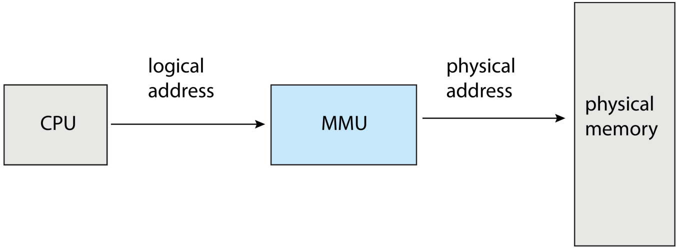
Physical address

Address that the memory unit works with.

Sets of addresses available: **logical address space** and **physical address space**.

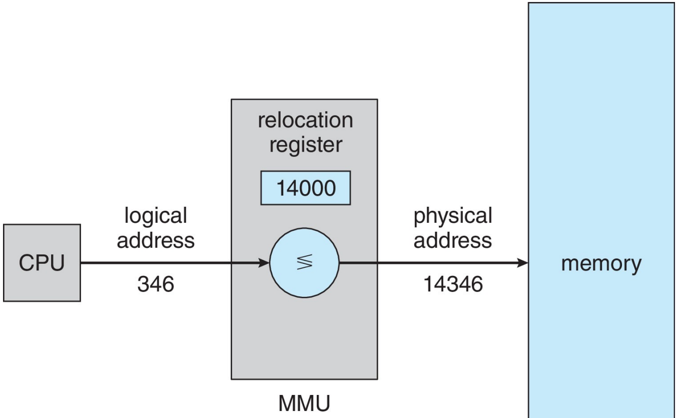
Compile-time and load-time binding results in equivalent logical and physical addresses. Execution-time binding makes processes think their address starts at 0000 and separate mapping is done to the physical address space.

Memory-management unit



- Base register now called **relocation register**.
- Value of **relocation register** is added to every (virtual) address generated by user program.
- User program never sees actual physical addresses.
- **Execution-time address binding** done on memory accesses, by the **MMU**.

Memory-management unit



Dynamic loading

Load routines when needed

- Entire program does not need to be copied in memory.
- Load some parts of it only when they are called.
- Memory utilization is improved because routines that are not used are never loaded.
- All code kept in **relocatable load format** on disk.
- Rarely called large routines do not need to be in memory for the lifetime of the process.

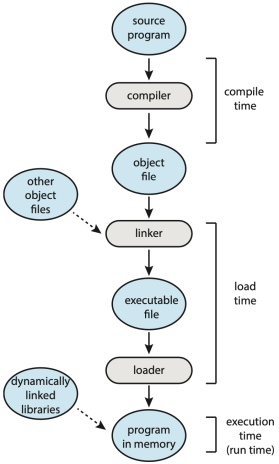
Dynamic linking

Static linking

System libraries and program code are combined into a binary executable.

- Dynamic linking postpones this until execution.
- Commonly used with system libraries: do not put them into the executable at all until called.
- Allows to share system libraries among process (**Dynamically Linked Libraries - DLL**, or shared libraries).
- Helps versioning: a new version of DLL can be updated in memory and all programs that reference it will dynamically link to the new version - no need to relink.

Loading and linking



Part II: Contiguous Memory Allocation

Contiguous memory allocation

Contiguous memory allocation

OS and processes have to live in memory in order to all run concurrently, requiring efficient allocation of their memory areas. Contiguous allocation of the memory space is one way to implement this.

Partition memory into two parts:

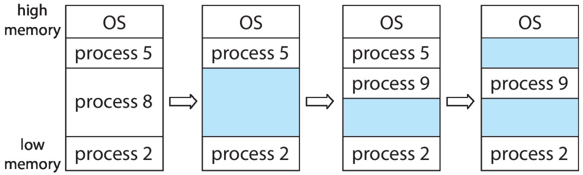
1. area for the operating system, and
2. area for processes, stored in single contiguous blocks.

Need for memory protection

Need to protect OS and processes, that are stored in the same memory space, from each other.

Memory allocation

- Keep track of free and occupied partitions.
- At start, memory is one big free block.
- Allocate **variable size partitions** as required.
- **Memory holes** of variable size form.
- When a process exits, it leaves a memory hole that is merged with adjacent holes.



Memory allocation

Given an allocation request of N bytes, how to determine which memory hole to return?

- **First-fit**: allocate the first free memory area of size N or larger.
- **Best-fit**: Search the list of free memory areas and allocate the smallest that fits N bytes (best case scenario is we find a memory hole of N bytes).
- **Worst-fit**: Allocate the largest free memory area available of size at least N.

Internal fragmentation

Allocated memory is larger than N, the requested size. The extra bytes in the allocated block are unused. Arises when memory is split into fixed-sized partitions.

External fragmentation

N bytes exist to satisfy the request for memory, but the space is not contiguous. Memory is broken into many small pieces.

Reducing external fragmentation

Compaction

Shuffle memory contents to merge all free memory holes into one contiguous space. Dynamic relocation is required for this to work, during execution. Require moving the program and data and updating the relocation register to reflect the change in the starting address.

Vevox quiz

Part III: Paging

Motivation

Why use paging memory management technique

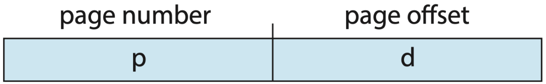
Previous techniques require memory to be contiguous, which introduces memory gaps and external fragmentation (requiring compaction). Paging allows processes to see contiguous memory space despite actual data stored in separate places in the physical memory.

Paging

A method that allows processes memory space to be fractured, but still look contiguous from process' perspective. Paging is implemented in collaboration between OS and the hardware. It is in widespread use today.

Basic method

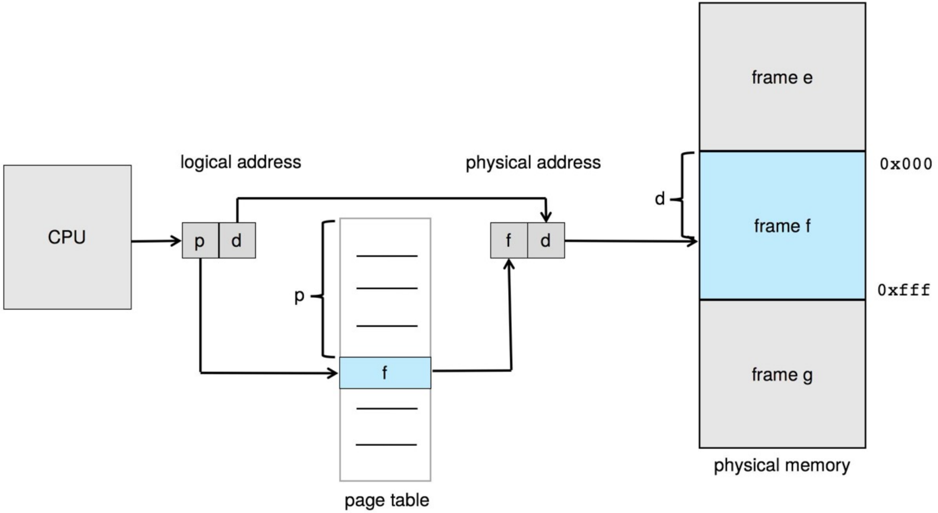
- Divide physical memory space into fixed-size blocks, **frames**
- Divide virtual memory space into fixed-size blocks, **pages**
- Keep track of free frames
- When program requires N pages, N free frames have to be found
- **Page tables** map pages to frames (**virtual addresses** to **physical addresses**).



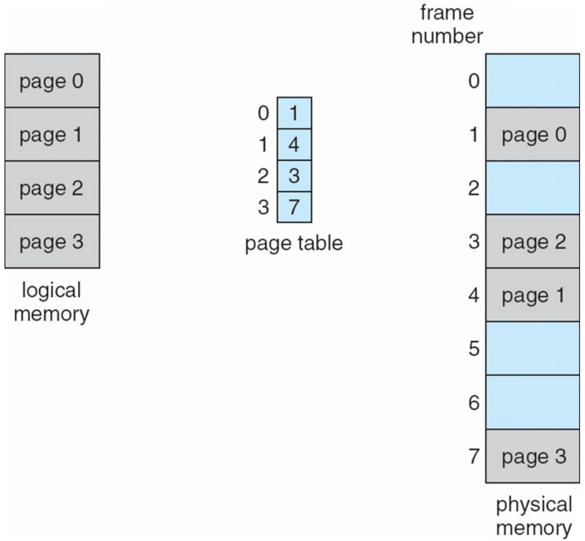
Virtual addresses

CPU generates addresses comprised of page number p and the page offset d. Entry p in the page table contains a base address of the corresponding frame in the physical memory. Then the offset d allows to find a specific memory address within the frame.

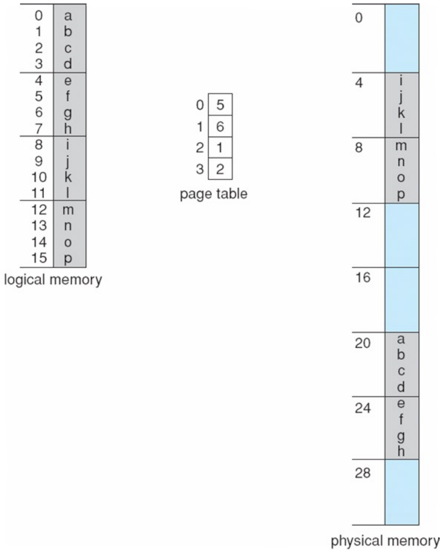
Hardware support



Paging example



Example (4-bit logical address: 2-bit page number, 2-bit address offset)



Paging

Key remarks about paging:

- No external fragmentation: any free frame can be allocated to a process.
- Internal fragmentation present: for example, page size is 4 bytes and process requests 10 bytes of memory; we will allocate 3 pages (12 bytes).

Page sizes

Today pages are typically 4 or 8 KB in size. Some systems support two page sizes which includes an option for **huge pages**.

Frame allocation to pages

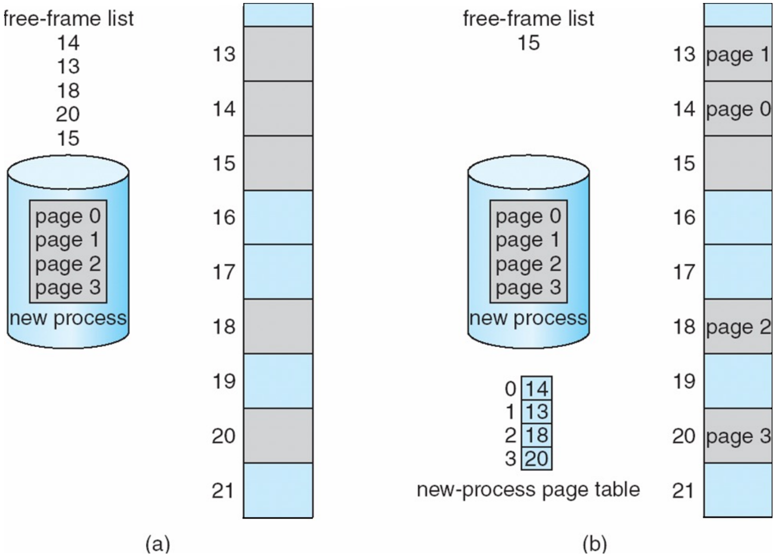
The main steps:

1. Process requiring execution arrives in the system.
2. Size in terms of pages is determined: $\lceil \frac{size}{pagesize} \rceil$ (number of pages required).
3. Each page requires a frame in physical memory.
4. If a process requires N pages, N frames need to be available.
5. First page is loaded into an allocated frame.
6. Frame number is written to the page table.
7. Repeat two previous steps until all pages are copied into frames and the page table of the process is fully set up. Process can then start and use logical addresses.

Programmer's view of memory

Paging allows programmer's view of memory to be separated from the physical memory. Programmer sees a contiguous area of memory available for a particular program. The program in reality is scattered across physical memory, with frames sitting amongst frames of other programs.

Frame allocation to pages (a: before allocation, b: after allocation)



Paging: implementation

- Each process has a page table stored in the main memory.
- Process Control Block of each process stores the address of the page table.
- When scheduler selects a process, it must set up the hardware page table by getting the copy from the memory.
- Simple hardware page table
    - Set of high-speed registers.
    - Translation of logical addresses fast.
    - Context switch time increases because these registers have to be exchanged.

PTRB register

Most CPUs support large tables (for example, 2^{20} entries)—register approach not feasible. Page table kept in memory and **page table base register (PTBR)** points to it. Context switch requires only one register swap.

Translation look-aside buffers

Page tables in memory

When page tables are stored in memory, process data/instruction access requires two memory operations, one for the page tables and another for the actual data.

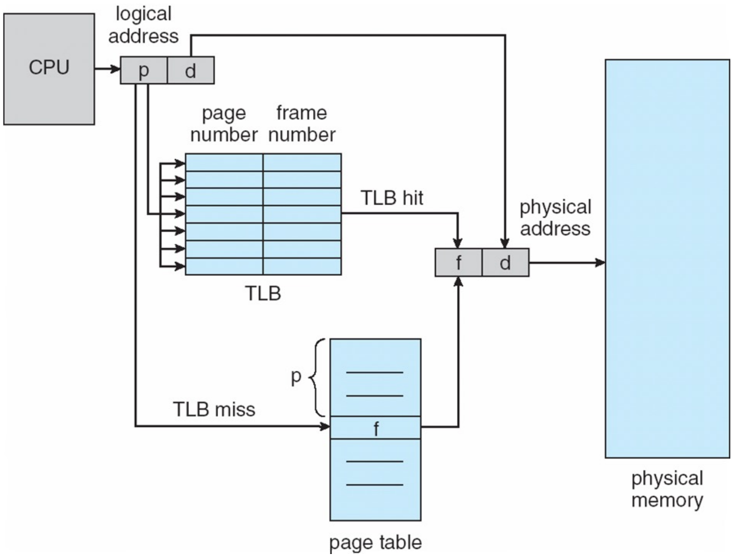
Translation look-aside buffer (TLB)

This technique is also called **associative memory**. Fast memory for storing commonly accessed frame addresses, typically 32 to 1024 entries. When CPU accesses memory, MMU first checks if page number is in the TLB and uses it if so. Otherwise (**TLB miss**) it gets the frame number from memory and updates the TLB.

TLBs in practice

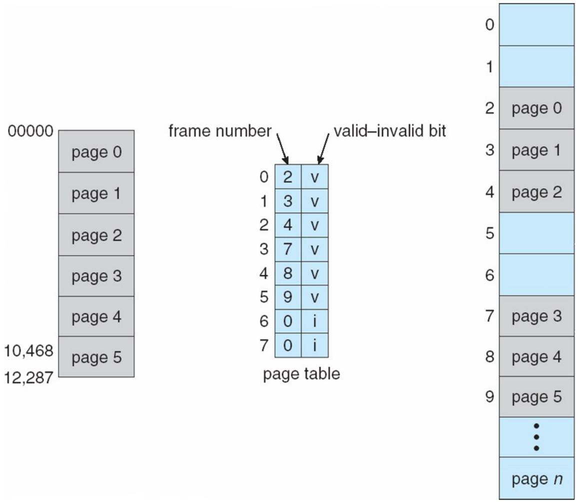
Modern CPUs provide multi levels of TLBs. For example Intel Core i7 has 128-entry L1 instruction TLB and 64-entry data TLB. When TLB miss occurs, it checks in the 512-entry L2 TLB. In case of another TLB miss, page table in the main memory is looked at.

Translation look-aside buffers



Memory protection with pages

- Page table can store read-only or write-only bits to mark restrictions in certain pages.
- *valid* bit indicates that the page is in the logical address space of the process.
- *invalid* indicates that it is out of bounds of the logical address space.
- Access to pages marked *invalid* will result in error.



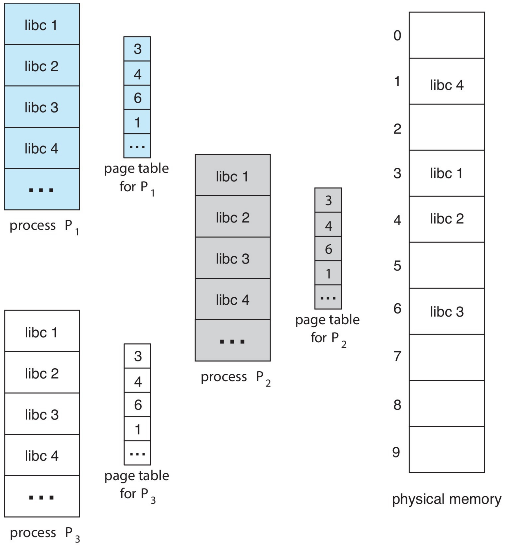
Protection with pages

Page-table length register (PTLR)

Instead of storing unused pages we may have a PTLR register that stores the size of the page table. This can be used for memory protection together with the PTBR.

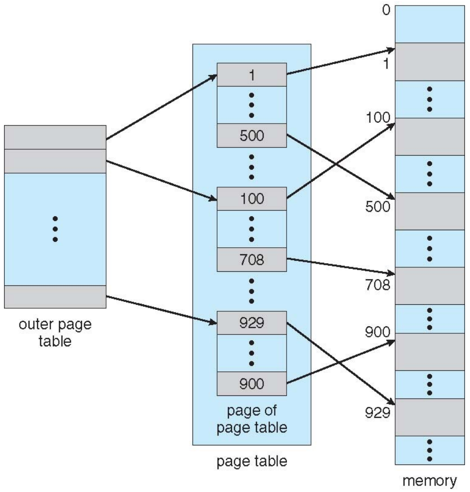
Shared pages

- Paging allows for efficient sharing of code between processes.
- **Reentrant code** means that the code of the library is not self-modifying.
- All processes can read it at the same time.
- For example, with shared pages, no need to have the copy of `libc` C standard library in each processes' memory.

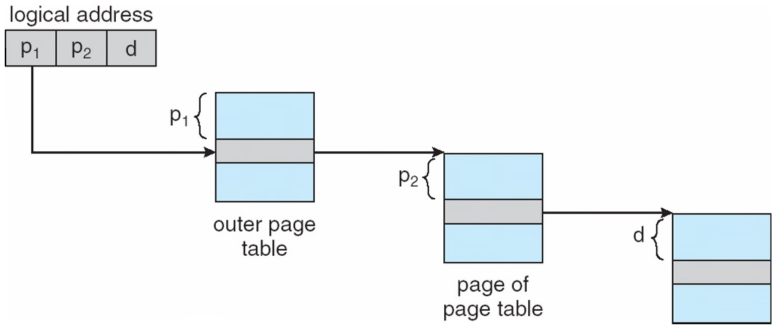


Hierarchical paging

Page tables in practice would be too large; they are split up in multiple layers to reduce size.



Hierarchical paging address translation



64-bit architectures

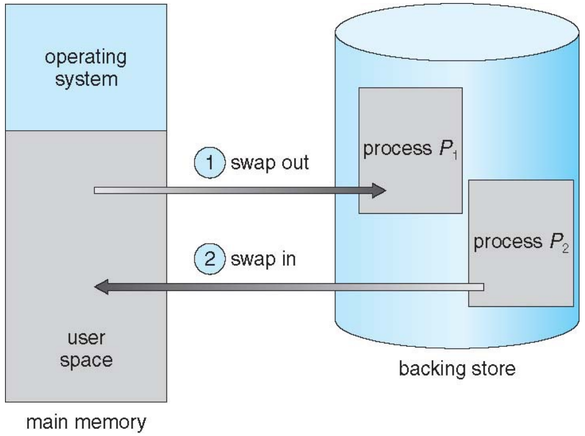
For 64-bit architectures even the two-level hierarchical page table requires too many entries. Other solutions are **hashed page tables** and **inverted page tables**—check OSC Chapter 9 for details.

Part IV: Memory Swapping

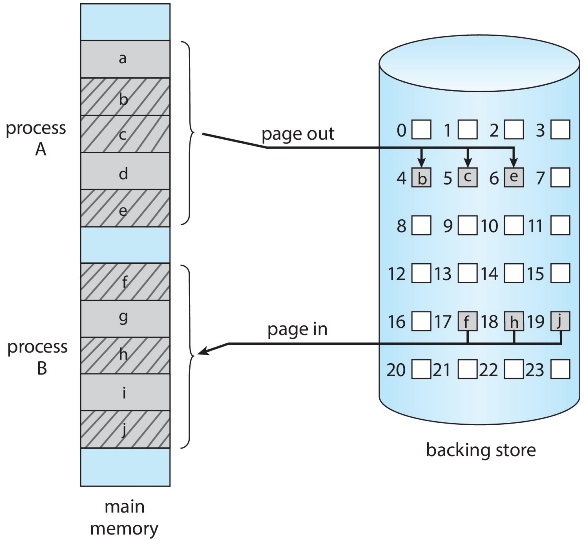
Standard swapping

- Process instructions and data have to reside in memory for execution.
- When not executing, a process can be swapped out into disc and brought back into memory later.
- Why? When main memory is at its limit, to make space for something with higher priority.
- **Swapping** allows for the total memory to look larger than the available physical memory.
- If N processes are executing, but only $N/2$ can fit into memory, the users still see that all N are working even though half of them are swapped out into disc.

Standard swapping



Swapping with paging



Swapping in mobile systems

- Most OS on PCs and high-performance computers support swapping with pages.
- On mobile systems typically this is not available.
- On mobile systems flash memory available instead of large hard drives.
- Space limitation and flash memory degradation due to writing operations make swapping not practical.
- Instead, iOS for example asks applications to release some memory.
- May forcibly terminate some processes when out of main memory.

Vevox quiz

Part V: Virtual Memory

Virtual memory: introduction

- Memory management we discussed so far is stemming from a basic requirement: **instructions should be in memory for execution**.
- Downside: the size of the program limited to the size of physical memory.
- The entire program may not be needed; examples:
  - code for error conditions; errors rarely occur,
  - large arrays when not all elements are used, and
  - generally, features of large programs that are rarely used.

Store the program in memory only partially?

Benefits:

- Program sizes not constrained by physical memory size,
- more programs can be ran concurrently, and
- less I/O needed when loading/swapping programs.

Virtual memory: introduction

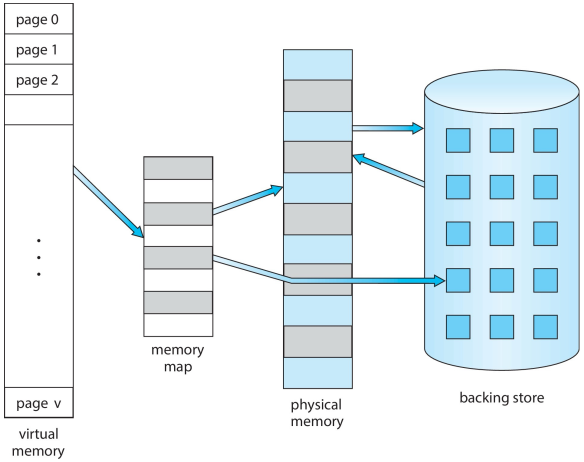
Virtual memory

Provide an extremely large memory space for the programmer's view: **logical space**. This space is implemented by a combination of smaller **physical memory** space and **large (but slow) disk** space, by moving pages between the disk and the physical space as required.

Programmer's view

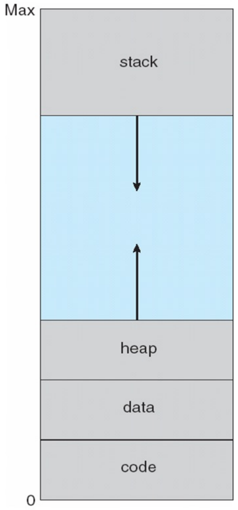
Virtual memory allows the programmer not to worry about the physical space limitations and concentrate on solving the problem in the virtual space. The virtual space starts at a certain address and is contiguous. MMU maps this space to the noncontiguous physical space.

Virtual memory: introduction

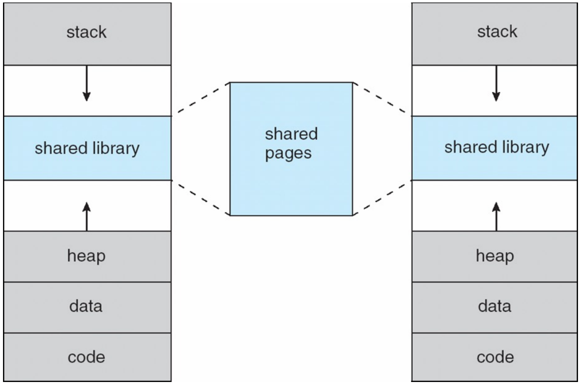


Virtual memory: introduction

- Commonly stack placed at the top addresses of logical space; grows down.
- Heap grows upwards.
- Hole in the middle - no physical memory used until pages are requested by, for example, `malloc` or dynamic linking.



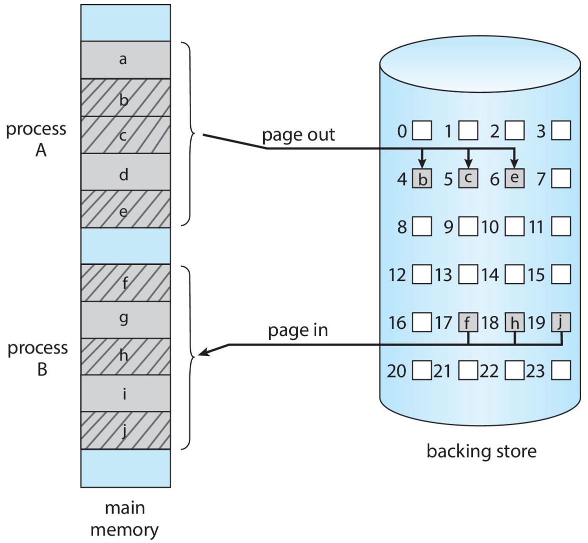
Virtual memory: introduction



Each process thinks that the shared memory is in their virtual address space, but the logical addresses are mapped to the same shared pages in the physical memory.

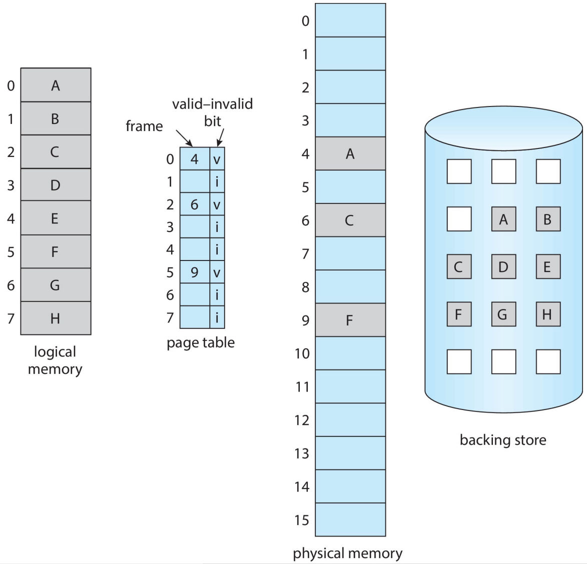
Demand paging: motivation

- When loading a process, should we bring all of its memory into the physical memory?
- A user may not need an entire large program.
- Instead, bring a page only when needed
- Similar to paging with swapping (diagram on the right).



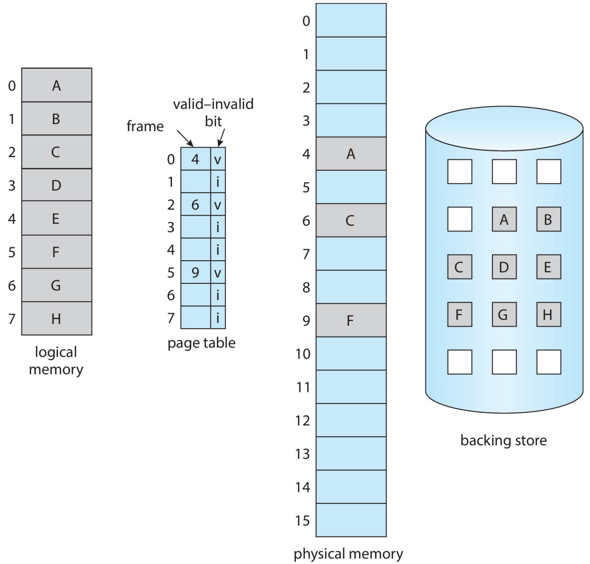
Demand paging: basic concepts

- With demand pages, a process in execution will have pages both in memory and in the backing storage (disk).
- We need some way of distinguishing them.
- Valid-invalid bit marker can be used here.
- When a page is set to valid, OK.
- When invalid, it may be out of address space or it may not be in memory.

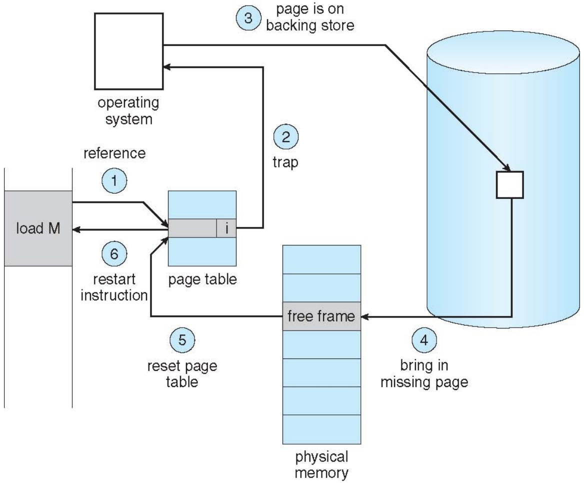


Demand paging: basic concepts

- If a process makes access to a page that is in the backing store, **page fault** occurs.
- We need to bring in that page into memory and set its valid bit to 1.
- Restart the memory instruction that caused the error.



Demand paging



Demand paging

- Extreme case: start process without any pages in memory.
- Let page faults occur and bring in only the required pages.
- **Pure demand paging** - never bring in a page until it is addressed.

Demand paging main requirement

The ability to restart the instruction after the page fault. If a page fault occurs on instruction fetch, we must refetch once the page has been loaded. If a page fault occurs when fetching an operand, we must fetch and decode the instruction again and then fetch the operand.

Example

Consider instruction `ADD A B C` which contains three memory locations. The steps are: 1) Fetch instruction, 2) Fetch A, 3) Fetch B, 4) Add A and B, 5) Store result in C. If page fault occurs when writing C, we will need to get the page in and restart the five steps.

Demand paging

- On page faults, desired page has to be brought into memory.
- It needs a frame in memory to fit the page in.
- A list of free frames is usually maintained: **free-frame list**.
- **Zero-fill-on-demand** is used to zero-out the previous data.
- Free-frame list also is modified when the stack or heap segments of the process expand.
- On system start up, available memory is placed on **free-frame list**.

$$\begin{tikzcd}
\text{head} \arrow[r] & \boxed{7} \arrow[r] & \boxed{97} \arrow[r] & \boxed{15} \arrow[r] & \boxed{126} & \cdots \arrow[r] & \boxed{75}
\end{tikzcd}$$

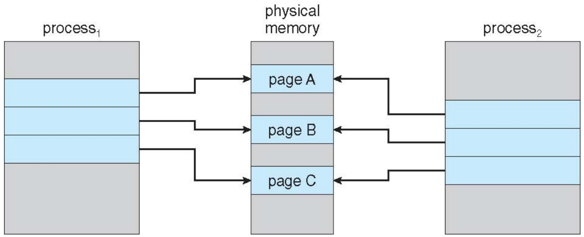
Demand paging: steps to take on page fault

1. Trap to the operating system
2. Save the user registers and process state
3. Determine that the interrupt was a page fault
4. Check that the page reference was legal and find the page on the disk
5. Issue a read from the disk to a free frame:
    1. Wait in a queue for this device until the read request is serviced
    2. Wait for the device seek and/or latency time
    3. Begin the transfer of the page to a free frame
6. While waiting, allocate the CPU to some other user
7. Receive an interrupt from the disk I/O subsystem (I/O completed)
8. Save the registers and process state for the other user
9. Determine that the interrupt was from the disk
10. Correct the page table and other tables to show page is now in memory
11. Wait for the CPU to be allocated to this process again
12. Restore the user registers, process state, and new page table, and then resume the interrupted instruction

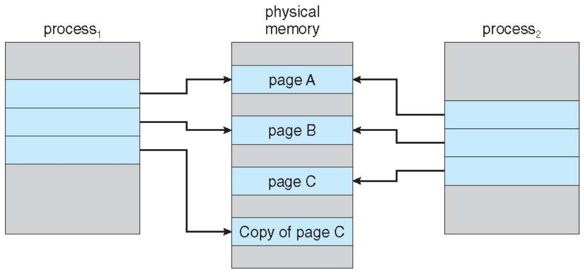
Copy-on-write

- Remember that `fork()` creates a copy of the calling process.
- Traditionally all pages have to be copied and assigned to a new process.
- But usually forked processes run `exec()` immediately, loading a new executable.
- A technique called **copy-on-write** avoids copying all pages on `fork()`.
- Instead, pages will be shared until the child process tries to write to one of them.
- On write, a copy of the page will be made that is then written to by the child process.

Copy-on-write



Copy-on-write

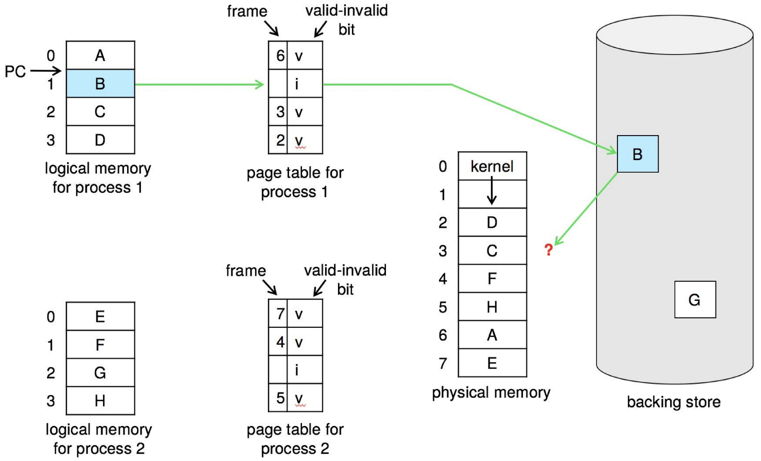


Part VI: Page Replacement

No free frames?

- What should OS do if there are no free frames to copy a certain page to?
- Terminate the process?
- Use standard swapping to copy some other process out into memory and free up pages: not used in modern OS because the whole process needs to be moved.
- Most OS now combine swapping pages (instead of whole processes) with **page replacement**.

No free frames?



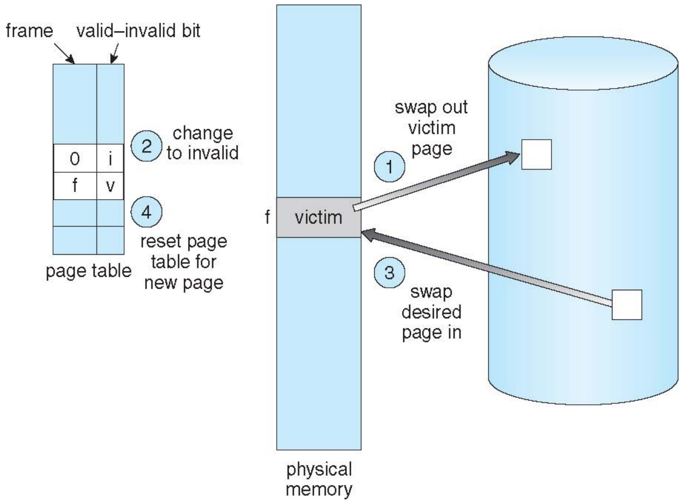
General page-replacement algorithm

1. Find required page on backing storage.
2. Get a free frame:
    1. If there is one available, use it.
    2. Apply page replacement algorithm to select **victim frame**.
    3. Copy the **victim frame** to the backing storage.
    4. Change page and frame tables to reflect the new state.
3. Move the original required page into the newly freed frame. Change page and frame tables.
4. Continue running the process.

Costs

Notice that there is performance penalty because two disk operations are required to move out one page and move in another page. **Dirty bit** is attached to each page to mark when it differs from its copy in the backing store—if dirty bit is not set, we don't need to copy it.

General page-replacement algorithm



Page-replacement algorithms

Some common algorithms for picking the **victim page**:

- **First-In-First-Out (FIFO)**: pick the oldest page in memory (arrived earliest).
- **OPT**: pick a page that will not be used for the longest period of time. Requires future knowledge. Not used in practice, but useful when comparing algorithms.
- **Least recently used (LRU)**: look back in the past and pick a page that has not been used the longest.

Page replacement

FIFO is the cheapest algorithm, but has an anomaly—increasing number of frames increases page faults. In general, any improvements to demand paging will yield large benefits because copying data from backing store take relatively long. Usually lowest page fault frame is a good measure to judge the algorithm on.

Vevox quiz

COMP2211: Progress

Week	Topic
1	~~Introduction to OS~~
2	~~OS services~~
3	~~Processes~~
4	~~Xv6: Live coding and Q&A from the xv6 book~~
5	~~Reading week. No scheduled labs or lectures~~
6	~~Threads and concurrency.~~
7	~~Scheduling~~
8	~~Process synchronisation~~
9	~~Memory management~~
10	~~Catch up~~
11	Module review

References I

A. Silberschatz, P. B. Galvin, and G. Gagne
Operating System Concepts. 10th edition
Wiley. 2018

A. Silberschatz, P. B. Galvin, and G. Gagne
Operating System Concepts. 10th edition. Accompanying slides
https://www.os-book.com/OS10/slide-dir/index.html
2020

R. Cox, F. Kaashoek, and R. Morris
xv6: a simple, Unix-like teaching operating system
https://pdos.csail.mit.edu/6.828/2024/xv6/book-riscv-rev4.pdf
Version of August 31, 2024

References II

R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau
Operating Systems: Three Easy Pieces. Version 1.0
https://pages.cs.wisc.edu/~remzi/OSTEP/
Arpaci-Dusseau Books. Aug., 2018