UNIVERSITY OF LEEDS

COMP2860 Operating Systems

Combined slides

Mantas Mikaitis and Timon Gutleb

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

Semester 1, 2025/26

Week 1: Introduction to OS

Objectives

- To introduce the structure of COMP2860 Operating Systems.
- Talk about the coursework.
- 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 COMP2860-OS: Staff

- James Dyer (labs)
- Timon Gutleb (Week 2)
- Mantas Mikaitis (me; Weeks 1, 3, 4, 5, 6)
- Tom Richardson (labs)

Structure of COMP2860-OS: Lectures

Week	Topic
1 (current)	Introduction to OS
2	OS processes
3	OS services
4	CPU scheduling
5	Memory protection, paging
6	Virtual memory & all content review
7	Lectures on security start with James Dyer (**OS assignment due this week.**)

Structure of COMP2860-OS: Times and Places

- Lectures Tue @ 9am in **Conference Auditorium 2**.
- Lectures Thu @ 11am in **Michael Saddler RBLT**
- Labs in **various venues around campus (check timetable)**.

You will find one of the lab slots in your timetable.

Structure of COMP2860-OS: Reading List

We will be mainly using the *Operating System Concepts* (OSC) 10th ed., 2018, and the 5th edition of the xv6 book (XV6), 2025. These slides are based on OSC (see the reference list).

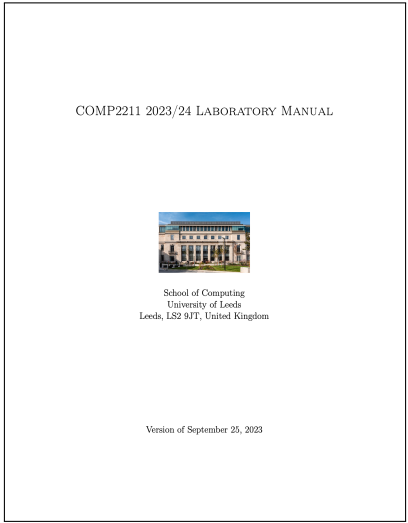
Week	Reading materials
1 (current)	**OS intro**: Chapter 1 OSC. Chapter 1 XV6.
2	**OS processes**: Chapter 3 OSC. Chapter 2 XV6.
3	**OS services**: Chapter 2 OSC.
4	**CPU scheduling**: Chapter 5 OSC.
5, 6	**Memory**: Chapters 9–10 OSC. Chapter 3 XV6.

Feel free to read other chapters too, if interested.

Optional reading: *Operating Systems: Three Easy Pieces* (free chapters online)

Structure of COMP2860-OS: Laboratories

- Lab assignments: we will use C and Bash.
- Download the lab manual from Minerva. Start working on the outlined steps.
- Week 1: reminder of C programming and command line interface.
- Week 2: introduction to the xv6 operating system.
- Week 3: programming in C to manipulate xv6 processes.
- Week 4: implementing `xargs` on xv6.
- Weeks 5–6: portfolio assignment to create a simple shell for xv6.



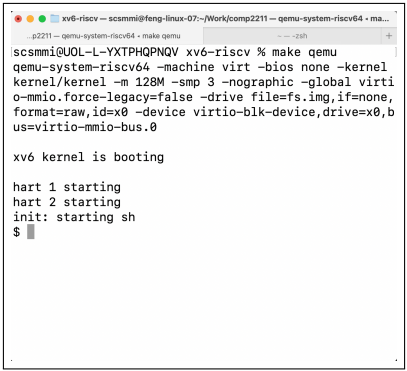
COMP2211 2023/24 Laboratory Manual

School of Computing
University of Leeds
Leeds, LS2 9JT, United Kingdom

Version of September 25, 2023

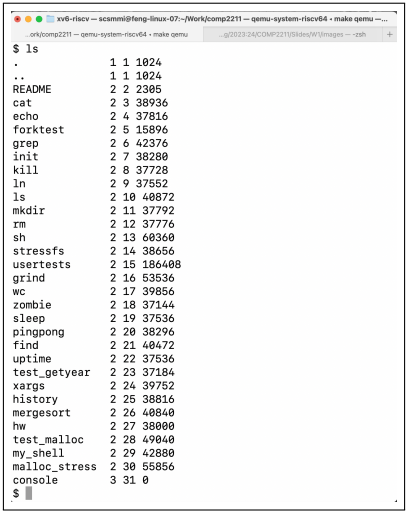
Structure of COMP2860-OS: xv6 operating system

- xv6 is a small teaching operating system created by MIT.
- The source code is readable and editable.
- It based on the RISC-V instruction set architecture.
- To run it on Intel/AMD CPUs we emulate RISC-V with **qemu**.
- It is written in C with parts in assembly.



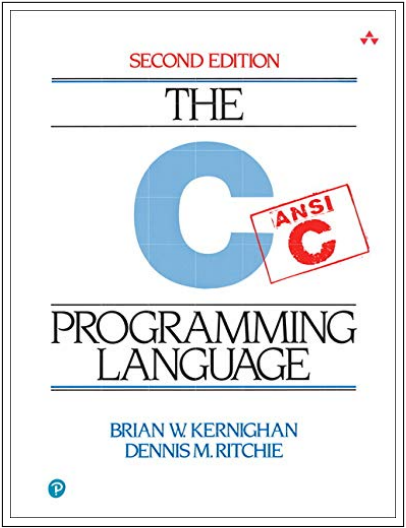
Structure of COMP2860-OS: xv6 operating system

- 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 **some bash commands**.
- In the labs you will extend your copy of xv6 to have more commands.



Structure of COMP2860-OS: Programming languages

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



Structure of COMP2860-OS: Assessment

Deadline 2pm Nov. 13 (W7). 20/25 marks needed to pass the portfolio.

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 1–4 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.

Exam in May 2026 (most likely pen and paper, closed book)

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

Structure of COMP2860-OS: Multiple-Choice Questions

The following three multiple-choice question tests are part of the portfolio (on Gradescope):

- Week 2: OS introduction.
- Week 4: OS processes.
- Week 6: Memory management.

You can complete these at any time—they are available on Gradescope right now.

Attempt as many times as needed until the deadline. Score 100% on all three to pass the portfolio.

Structure of COMP2860-OS: Communication and feedback

- Please email me with your questions.
- Come to the labs to ask questions.
- Provide questions on Vevox Q&A.

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 COMP2860-OS: 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 COMP2860-OS: What is in the exam?

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

Part II: Introduction to Operating Systems

Operating Systems: Main Definitions

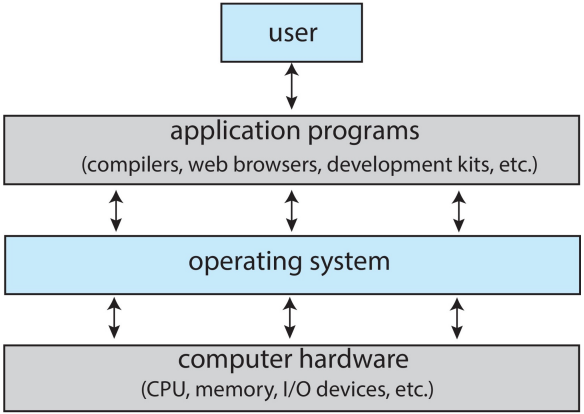
A **computer system** can be divided into four components:

- 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



Operating Systems: Main Definitions

User view:

- A laptop or a 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.

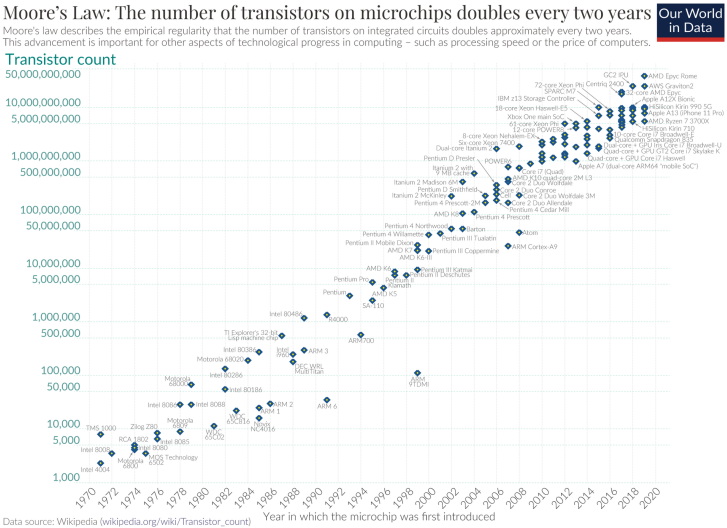
Operating Systems: Main Definitions

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



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**.

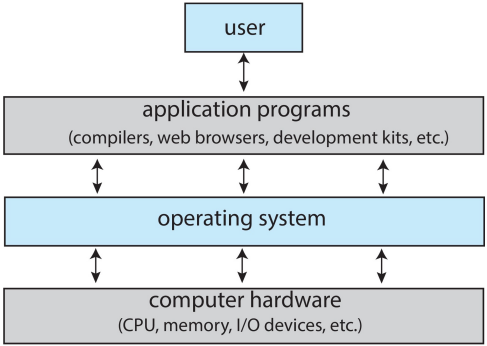
Novadays OS includes many things outside the immediate definition of what the OS does: browsers, photo viewers, word processors, ...

Operating System

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

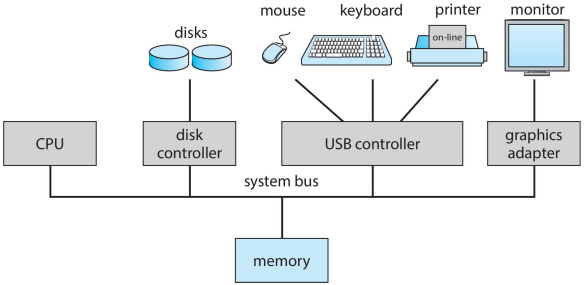
Structure of COMP2860-OS: Discussion with peers (5 minutes)

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?



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.

We now go deeper into various concepts within this system.

Interrupts

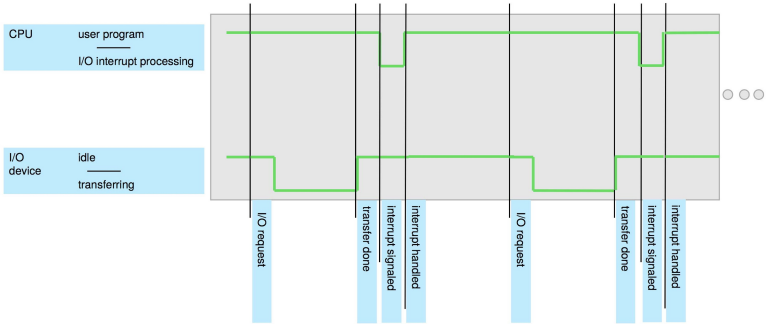
Consider a series of events within the system when we press a key on a keyboard. This constitutes input/output (I/O).

1. The keyboard activity informs the controller.
2. The controller reads to see what needs to be done and informs the keyboard driver.
3. The driver starts transfer of data from the keyboard.
4. Driver informs OS that a transfer has been done.

How does the controller inform the driver (CPU) that a key needs to be read? 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.



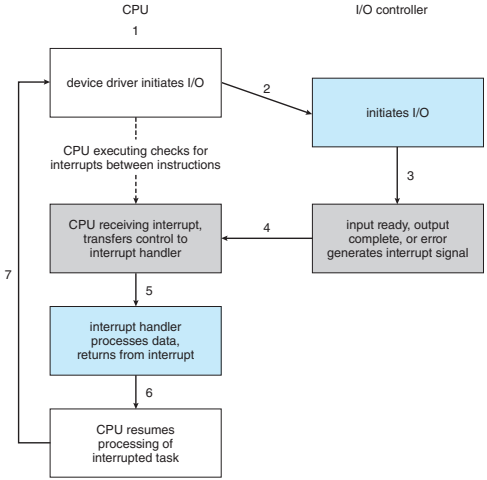
Interrupts

- 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 an 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.

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

Storage Structure: Secondary storage

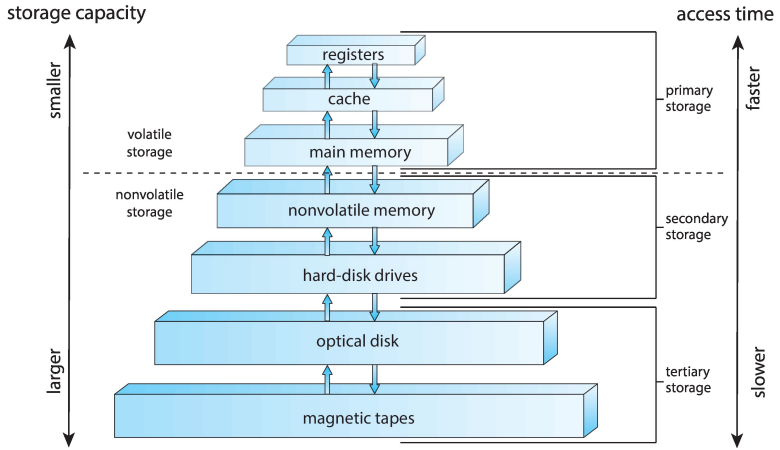
Data is stored in secondary storage, which preserves programs and data while the system is off.

- 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

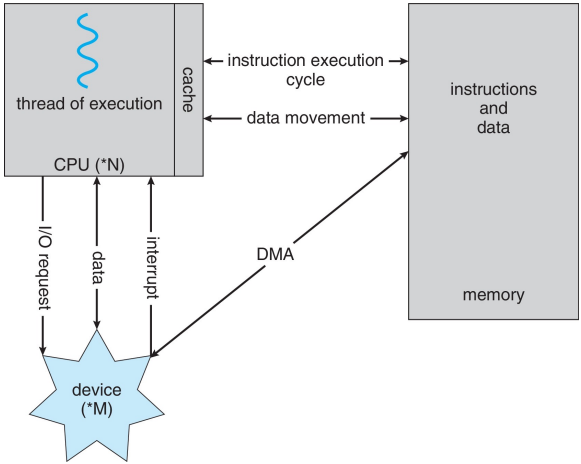
Interrupt driven memory access is fine for small requests, but moving a lot of data will not work very well (many interrupts needed).

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 executes program instructions and manages 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

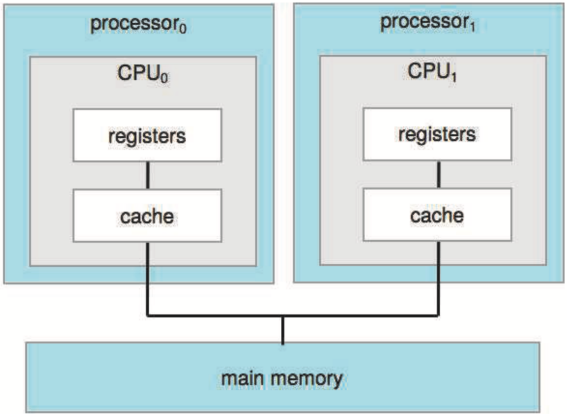
Multiprocessor systems dominate the computer landscape novadays.

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.

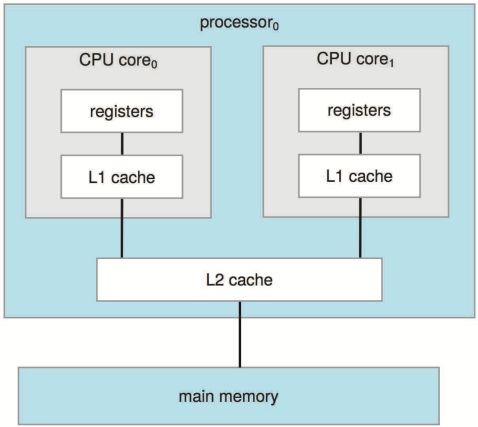
Ideally N processors should result in N times speed up. In reality it is less: there is some overhead in managing multiple processors that cooperate on some task. This overhead does not exist when only one processor is executing.

Symmetric multiprocessing (SMP)



Multiprocessor Systems

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



Multiprocessor Systems: Main Terms

- 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.

CPU—commonly used to refer to processors or to cores, interchangeably.

Single/multiprocessing: Discussion with peers (5 minutes)

Find out how many processors and cores 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 system.

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

Does anyone think they have the largest number of cores in the room?

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.

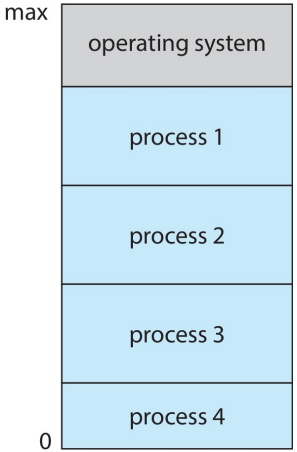
Multitasking

The switches between processes are very frequent to provide users with a fast **response time**. Multiprogramming is required for a good multitasking experience.

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

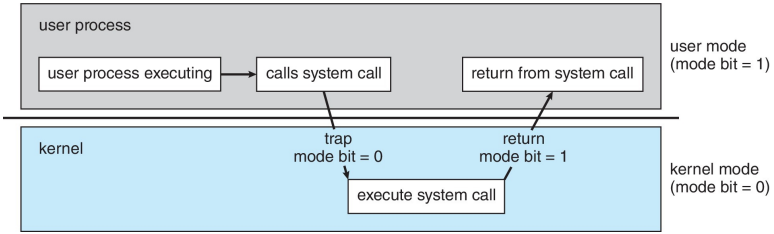
Main idea

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 a user process 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**.
- Provide ways for **process communication**.

Processes will be discussed in detail in Week 2.

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 later weeks.

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

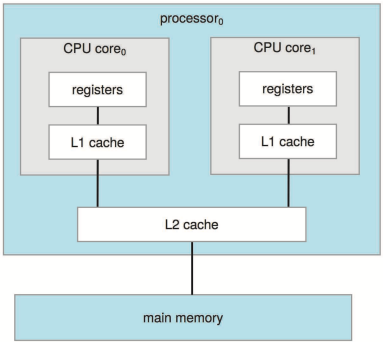
Level	1	2	3	4	5
Name	registers	cache	main memory	solid-state disk	magnetic disk
Typical size	< 1 KB	< 16MB	< 64GB	< 1 TB	< 10 TB
Implementation technology	custom memory with multiple ports CMOS	on-chip or off-chip CMOS SRAM	CMOS SRAM	flash memory	magnetic disk
Access time (ns)	0.25-0.5	0.5-25	80-250	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**.

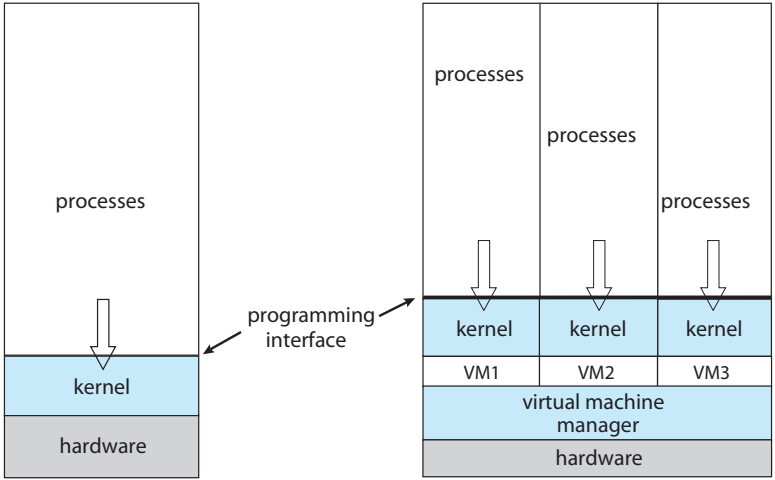
Emulation: guest OS compiled for a different hardware which has to be emulated to run on the hardware we have on our desk.

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.

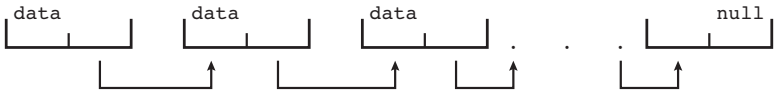
Virtualization



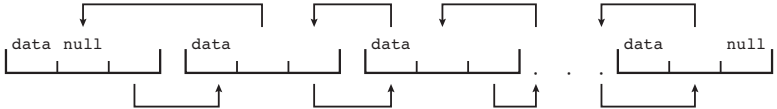
Kernel Data Structures

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:

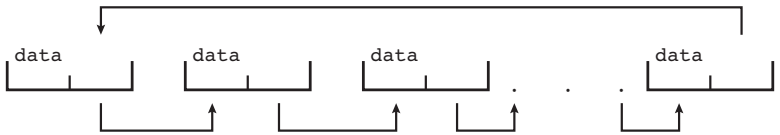


Doubly linked list:



Kernel Data Structures

Circularly linked list:

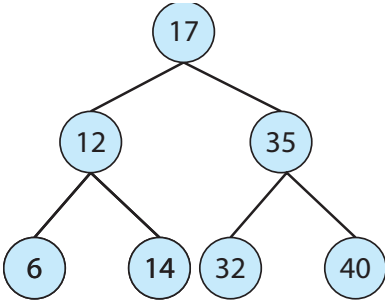


Lists of size n require at most n checks to find an item.

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).

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

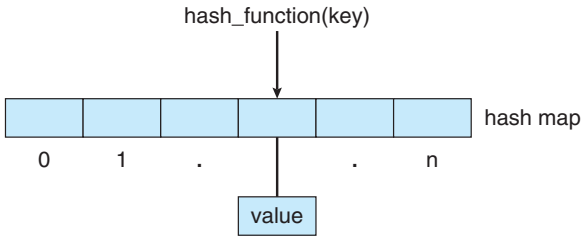


Tree search complexity

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.

Week 2: OS Processes

Objectives

In the 'OS processes' topic of this module we will answer:

- What is a process?
- How do operating systems represent and schedule processes?
- How are processes created and terminated?
- How can processes communicate with other processes?

Part I: Introduction to Processes

What is a Process?

Definition: Process

A **process** is a program **in execution**.

Early computers executed only **one program** at a time. The running program (= 'process') had complete control over resources.

Today, **multiple programs** may run simultaneously – computers can **multitask**. To achieve this, computers have to compartmentalize programs.

Note: Multiple processes of the same program

Invoking a program multiple times results in multiple processes.

More terminology

Early computers were **batch systems** which executed **jobs** submitted by users with minimal user interactivity.

Note: 'Job', 'Task' and 'Process'

These words are often used interchangeably. Only 'process' has consistent meaning across modern OS. 'Job' and 'task' can cause ambiguities between Windows and UNIX systems.

This was followed by **time-shared** systems: many users can access resources simultaneously. Even a single user can run several programs: file browser, web browser, editors and a lot of so-called 'daemons'.

Definition: 'Daemon' and 'Service'

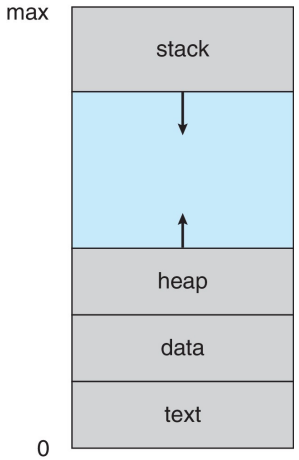
A **daemon** (also called '**service**') is a **background process** (no user interactions). Unix naming convention: append the letter 'd'. Examples: 'bluetoothd' (macOS), systemd (Linux).

The abstract memory layout of a process

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.



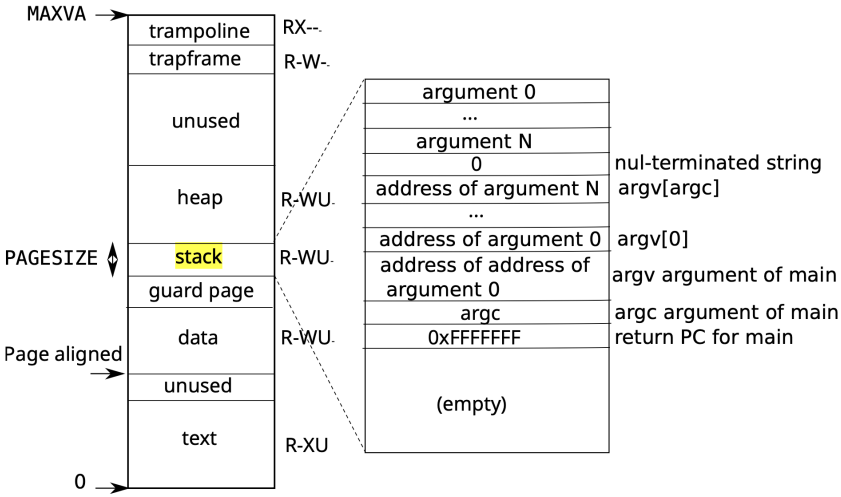
The abstract stack

The **stack** contains a small amount of data which can be **pushed** onto and **popped** using specific CPU instructions.

A typical example

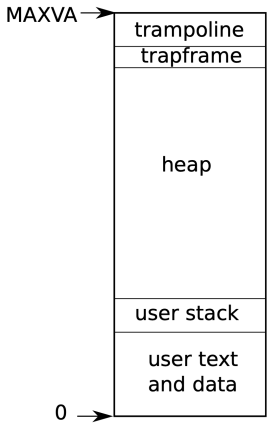
When a function is called, input arguments, local variables, and the return address are **pushed** onto the stack. Upon completing the function, data is **popped** from the stack: the last one is usually the return address to get back to the caller.

The xv6 stack

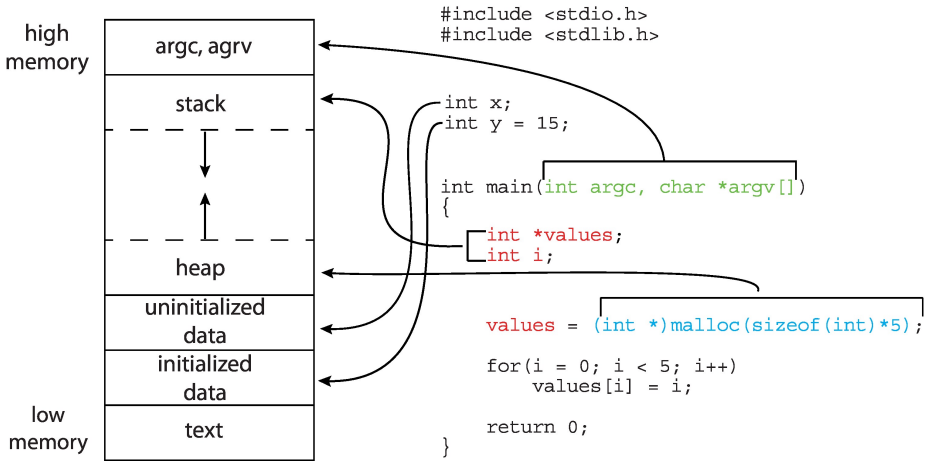


The heap

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



More detailed memory layout example

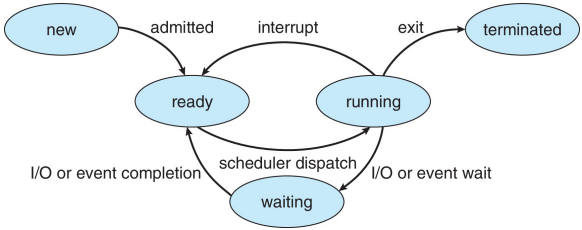


Part II: Process Scheduling

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: CPU cores

Usually only one process can be **running** on a core. Others may be **ready** or **waiting**.
Using multiple cores will be covered in the 'Parallel & Distr. Computing' part of COMP2860.

Process State Transitions (in the real world)

Throughout the OS part of this module we are largely considering operating systems in the abstract, leaning towards UNIX-like conventions.

This is necessary because we would otherwise get entangled in terminology spaghetti: Different operating systems use slightly different conventions **all the time**.

Process States (in the real world) - macOS manual

```
state     The state is given by a sequence of characters, for example, "RWNA".  The first character indicates the run state of the
          process:

          I       Marks a process that is idle (sleeping for longer than about 20 seconds).
          R       Marks a runnable process.
          S       Marks a process that is sleeping for less than about 20 seconds.
          T       Marks a stopped process.
          U       Marks a process in uninterruptible wait.
          Z       Marks a dead process (a "zombie").

          Additional characters after these, if any, indicate additional state information:

          +       The process is in the foreground process group of its control terminal.
          <       The process has raised CPU scheduling priority.
          >       The process has specified a soft limit on memory requirements and is currently exceeding that limit; such a process
                  is (necessarily) not swapped.
          A       the process has asked for random page replacement (VA_ANOM, from vadvise(2), for example, lisp(1) in a garbage
                  collect).
          E       The process is trying to exit.
          L       The process has pages locked in core (for example, for raw I/O).
          N       The process has reduced CPU scheduling priority (see setpriority(2)).
          S       The process has asked for FIFO page replacement (VA_SEQL, from vadvise(2), for example, a large image processing
                  program using virtual memory to sequentially address voluminous data).
          s       The process is a session leader.
          V       The process is suspended during a vfork(2).
          W       The process is swapped out.
          X       The process is being traced or debugged.
```

Process States (in the real world) - Linux source code

linux / include / linux / **sched.h**

Code | Blame | 2315 lines (1990 loc) · 66.5 KB

```
85   struct task_struct;
86   struct user_event_mm;
87
88   #include <linux/sched/ext.h>
89
90   /*
91    * Task state bitmask. NOTE! These bits are also
92    * encoded in fs/proc/array.c: get_task_state().
93    *
94    * We have two separate sets of flags: task->__state
95    * is about runnability, while task->exit_state are
96    * about the task exiting. Confusing, but this way
97    * modifying one set can't modify the other one by
98    * mistake.
99    */
100
101  /* Used in tsk->__state: */
102  #define TASK_RUNNING                    0x00000000
103  #define TASK_INTERRUPTIBLE              0x00000001
104  #define TASK_UNINTERRUPTIBLE            0x00000002
105  #define __TASK_STOPPED                  0x00000004
106  #define __TASK_TRACED                   0x00000008
107  /* Used in tsk->exit_state: */
108  #define EXIT_DEAD                       0x00000010
109  #define EXIT_ZOMBIE                     0x00000020
110  #define EXIT_TRACE                      (EXIT_ZOMBIE | EXIT_DEAD)
```

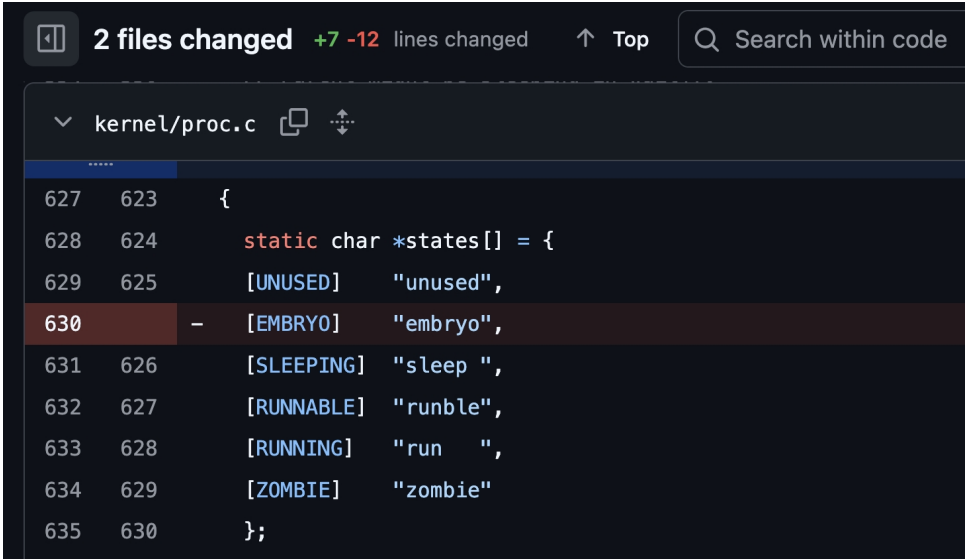
Process States (in the real world) - xv6 - Part 1

xv6-riscv / kernel / proc.c

Code Blame 687 lines (586 loc) · 14.2 KB

```
659   // Print a process listing to console.  For debugging.
660   // Runs when user types ^P on console.
661   // No lock to avoid wedging a stuck machine further.
662   void
663   procdump(void)
664   {
665     static char *states[] = {
666     [UNUSED]    "unused",
667     [USED]      "used",
668     [SLEEPING]  "sleep ",
669     [RUNNABLE]  "runble",
670     [RUNNING]   "run   ",
671     [ZOMBIE]    "zombie"
```

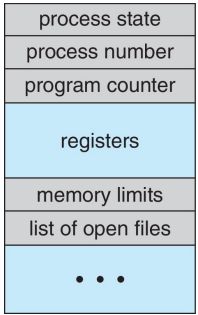
Process States (in the real world) - xv6 - Part 2



Process Control Block (PCB)

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

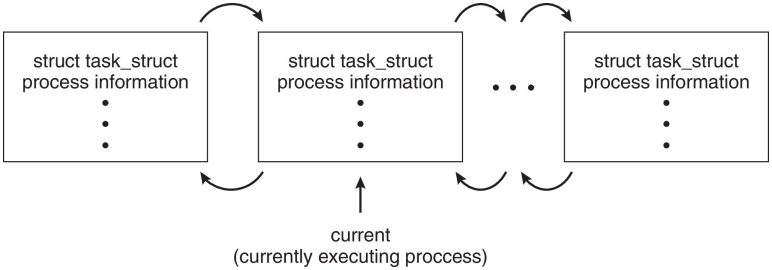
- **Process state**: e.g. new, running, waiting, ready, terminated
- **Program counter (PC)**: Where are we in the execution of the program?
- **CPU registers**: What data are we working on?
- **CPU-scheduling info**: priority and other scheduling parameters.
- **Memory-management info**: Memory locations assigned to a process.
- **Accounting info**: Resource utilization statistics.
- **I/O status info**: I/O allocated to a process, e.g. open files.



Note: Interrupts

Program counter and CPU registers need to be saved when interrupting a process.

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?

'Thread' is a common term when talking about programs in execution. We have assumed that a process performs a **single thread of execution**, read in sequence by one CPU core.

In such a single-threaded environment, the distinction between 'thread' and 'process' is largely unimportant but in a multi-threaded context these terms refer to different things.

Note: Parallel computing and multi-threading

Modern OS extend processes to be able to execute at different locations in the binary at once. The process control block and other parts of OS have to be expanded to make this work.
Parallel computing will be covered in the 'Parallel & Distr. Computing' part of COMP2860.

Process Scheduling

Reminder: Multiprogramming

A single program typically cannot keep the CPU busy at all times. The ability to run multiple programs and change between them is called **multiprogramming**. This increases CPU utilization by swapping which process gets CPU time.

Definition: Process Scheduler

The **process scheduler** is an integral part of an OS which meets the constraints posed by time-sharing and multi-tasking by selecting a process to run from a set of available processes.

We distinguish two types of processes:

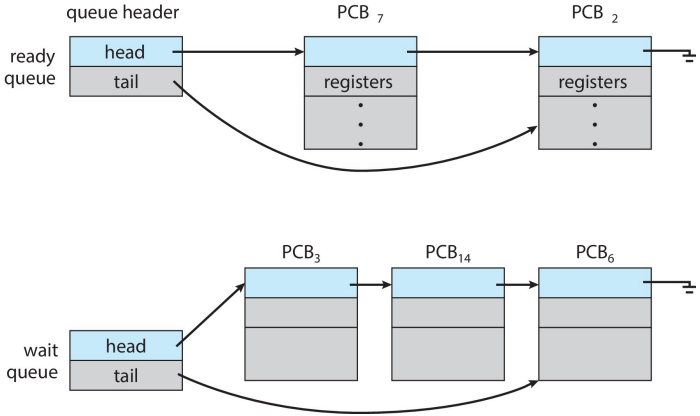
- **I/O bound**: most time spent *waiting* for memory.
- **CPU bound**: most time spent in execution.

The types of active processes affects the objectives of multiprogramming and time-sharing.

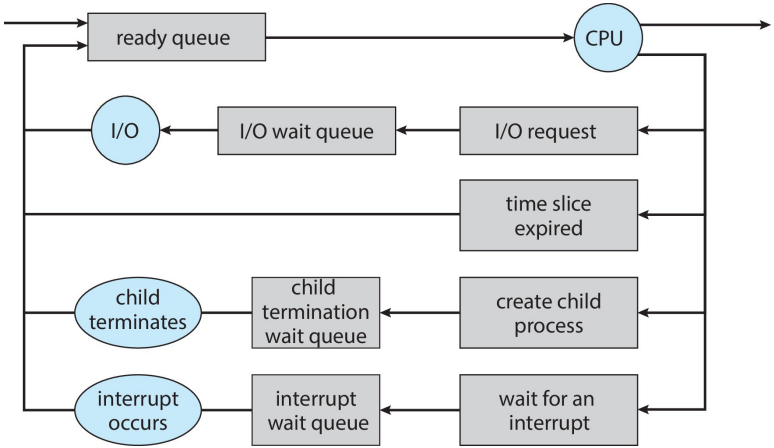
Ready and wait queues

Processes enter the system and are put into a **ready queue**, usually implemented as a linked list where each PCB links to the next.

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



An abstract queuing diagram



The CPU Scheduler

Definition: CPU Scheduler (also: short-term scheduler (STS))

The CPU scheduler selects from a set of **ready** processes and assigns a CPU core to it. This critically affects how efficiently a core is used and many different scheduling algorithms exist.

- **I/O bound processes** may execute for a few milliseconds before waiting for I/O.
- **CPU-bound processes** may require the CPU for extended durations.
- Processes are typically switched between very frequently (in less than 100 ms).

Definition: Memory Swapping

Memory swapping temporarily moves a process from memory to disk (saving state), reducing the **multiprogramming degree** and CPU competition. It can later be resumed.

Context switching: The basics

Reminder: Interrupts

Interrupts cause the CPU to pause the current process and execute some **kernel** routine.

Definition: Context Switch

Context switching performs a **state save** of the current process and a **state restore** of a different one.

Definition: Process context

The operational state data needed to resume a process is called its **context** and consists of things like the program counter, register contents, memory management info and its state.

Context switching: Additional notes

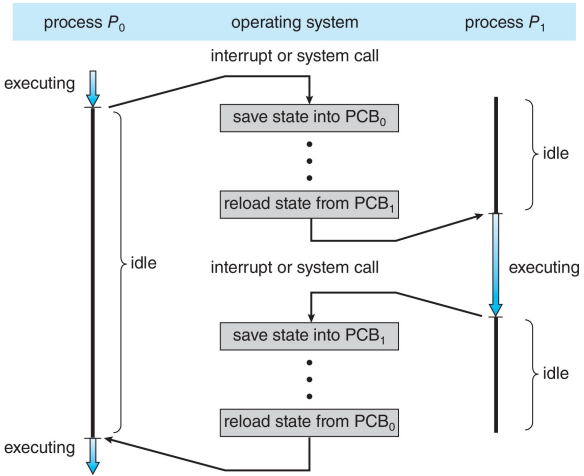
Note: Context vs. PCB

The exact details of how context switching is implemented depend on the OS but the main idea is that upon switching the **process context is stored in a process control block (PCB) data structure**.

Note: The CPU during context switches

Context switching is **pure overhead** as the CPU is not executing any process instructions while it occurs. Fast context switching is thus essential.

An abstract context switching diagram



Part III: Process Manipulation

Process Creation

Processes may create several new processes during execution.

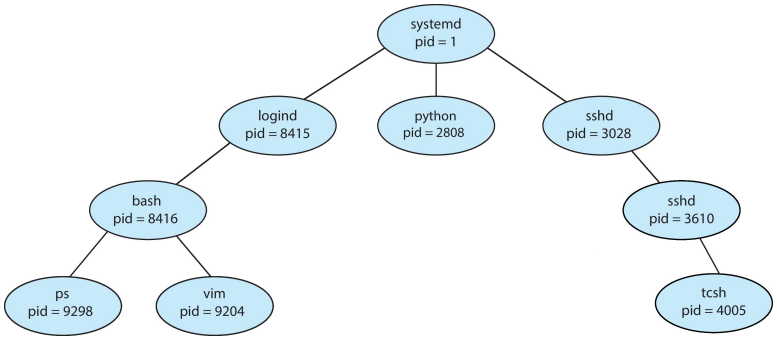
Definition: Parent and child processes

A process which has created other processes is called a **parent process** while the created processes are called **child processes**.

New processes can in turn create more, forming a **tree of processes**.

To keep track of specific processes unique **process identifiers (process ID or pid)** are used.

Process Creation: Linux example



The systemd process is created on boot. It then creates processes for various services.

Process Creation: System Resources

There are two options for allocating system resources to child processes. A child process can:

- obtain resources directly from the OS.
- share a subset of resources from the parent.

Restricting child process to a subset of the parent's resources can avoid overloading the system but may result in less flexibility.

Process Creation: Resource management options

When a process creates another process, there are two options for how to proceed:

- Parent and child processes can execute **concurrently** (not necessarily in parallel).
- The parent can wait until some or all of its child processes terminate.

The situation is analogous for the processes address space:

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

Process Creation: Examples

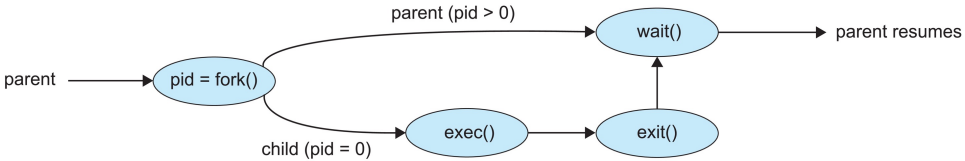
In UNIX, a new process is created by `fork()`:

- 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.

Process Creation: Concurrency example



Process Creation: UNIX example with C

```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;
}
```

Process Termination

A process terminates when it asks the OS to delete it using the `exit()` system call.

All the resources (e.g. physical and virtual memory, open files, I/O buffers) are reclaimed by the OS.

A parent process may forcibly terminate its created processes. This is done e.g. when:

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

Definition: Zombie processes

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

Part IV: Communicating Processes

Interprocess Communication (IPC)

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

Definition: Independent Process

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

Definition: Cooperating Process

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

Cooperating processes are useful scenarios such as:

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

Cooperating Process Models

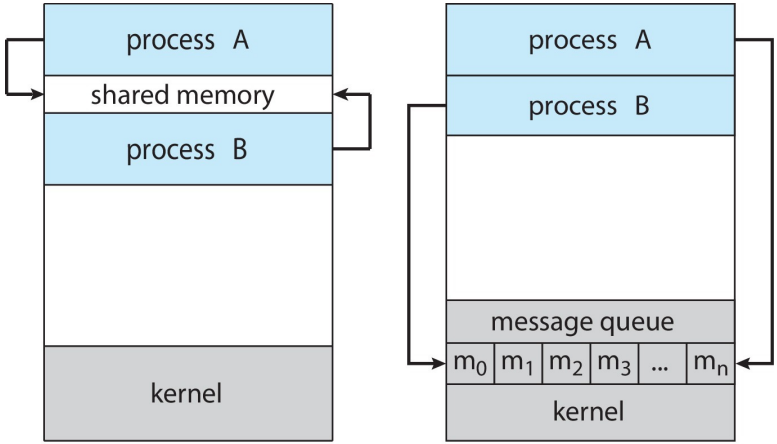
There are at least two approaches to cooperating processes implemented in operating systems:

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

The **shared-memory model** is potentially vulnerable to conflicts and race conditions may appear (two processes writing to the same piece of memory at the same time).

The **message-passing model** is useful when no conflict resolution is desired but it is slower, since each read/write requires kernel ops. Message-passing is required to communicate between different systems that do not share memory.

Abstract Diagram: Shared-memory and Message-passing models



Pipes

Pipelines, often just called **pipes**, were one of the earliest UNIX implementations of IPC.

Note: Pipes as an example of shared-memory model IPC

Pipes are an example of a **shared-memory model** implementation of process communication, with the basic idea being that processes are arranged such that the output of one process in the pipeline is written to the input of the next.

Different kinds of pipe designs are used depending on the specific use-case:

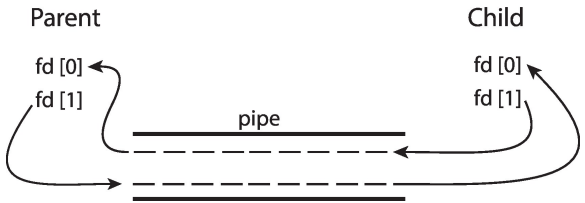
- Is there need for **bidirectional** or only **unidirectional** communication?
- If bidirectional is needed, does data need to travel in both directions at once?
- Do we require a relationship (e.g. parent-child) between communicating processes?
- Can we use pipes over network or only locally?

Ordinary Pipes

In the **producer-consumer model** of pipes, the producer process writes to the **write end** of the pipe while the consumer process reads from the **read end**.

These are **unidirectional** and we thus 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 which means `fork()` in the parent will make the child inherit these.



Named Pipes

While **ordinary pipes** provide a simple mechanism for processes to communicate, they only exist while processes exist. When they terminate, the pipe disappears.

Named pipes (also known as FIFOs = 'first in, first out') in UNIX provide extra functionality:

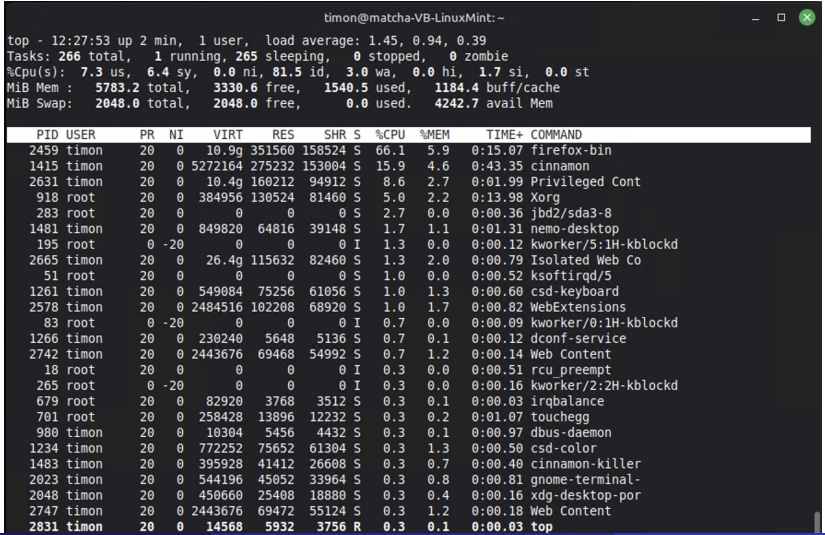
- Bidirectional communication.
- No parent-child relationship needed.
- Several processes can use the pipe for communication.
- The pipe remains active after communicating processes terminate.

Note that while FIFOs are **bidirectional** they only allow half-duplex transmission (i.e. one direction at a time).

Part V: A bit more from the real world

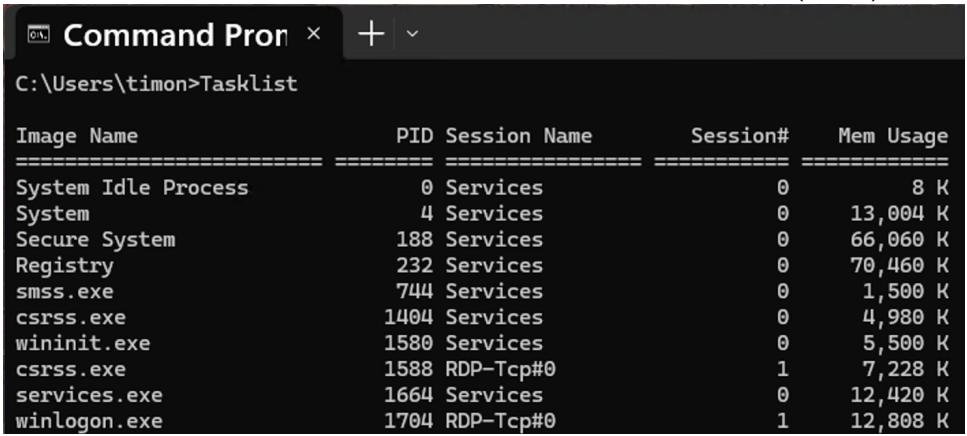
Processes in the real world - Part 1

'ps' or 'top' are typical UNIX-like commands to run to get info on processes.



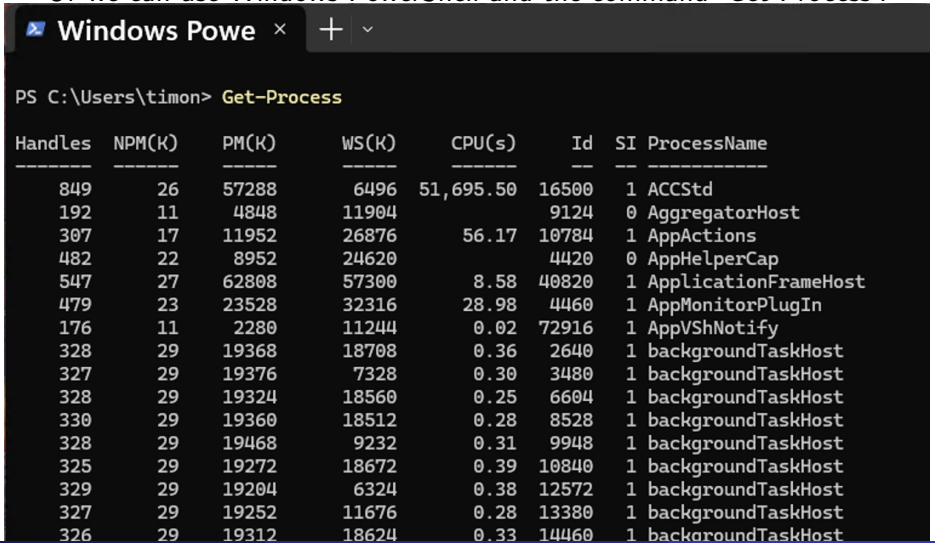
Processes in the real world - Part 2

In Windows we can use 'Tasklist' in the command prompt (CMD):



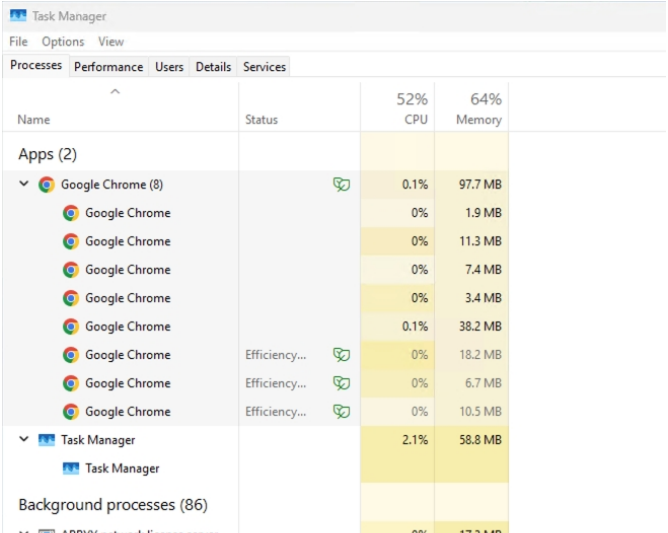
Processes in the real world - Part 3

Or we can use Windows PowerShell and the command 'Get-Process':



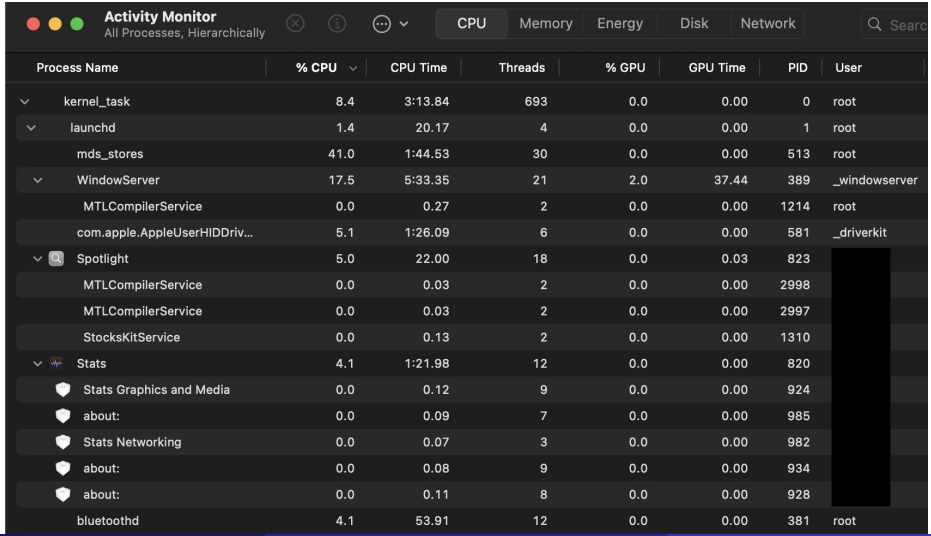
Processes in the real world - Part 4

Or you accept that you are in a Windows GUI session and just open Task Manager:



Processes in the real world - Part 4

GUI applications also exist in UNIX-like systems. Here is the macOS Activity Monitor:



Process Name	% CPU	CPU Time	Threads	% GPU	GPU Time	PID	User
kernel_task	8.4	3:13.84	693	0.0	0.00	0	root
launchd	1.4	20.17	4	0.0	0.00	1	root
mds_stores	41.0	1:44.53	30	0.0	0.00	513	root
WindowServer	17.5	5:33.35	21	2.0	37.44	389	_windowserver
MTLCompilerService	0.0	0.27	2	0.0	0.00	1214	root
com.apple.AppleUserHIDDriv...	5.1	1:26.09	6	0.0	0.00	581	_driverkit
Spotlight	5.0	22.00	18	0.0	0.03	823	[redacted]
MTLCompilerService	0.0	0.03	2	0.0	0.00	2998	
MTLCompilerService	0.0	0.03	2	0.0	0.00	2997	
StocksKitService	0.0	0.13	2	0.0	0.00	1310	
Stats	4.1	1:21.98	12	0.0	0.00	820	
Stats Graphics and Media	0.0	0.12	9	0.0	0.00	924	
about:	0.0	0.09	7	0.0	0.00	985	
Stats Networking	0.0	0.07	3	0.0	0.00	982	
about:	0.0	0.08	9	0.0	0.00	934	
about:	0.0	0.11	8	0.0	0.00	928	
bluetoothd	4.1	53.91	12	0.0	0.00	381	root

What are some reasons we might want to look at a table of processes?

- **System Performance**
  - Identify processes consuming high CPU or memory.
  - Detect disk I/O or network-heavy processes.
  - Find and terminate unresponsive or frozen applications.
- **Security and Auditing**
  - Spot suspicious or unauthorized processes.
  - Track which users are running which processes.
  - Detect processes with elevated privileges.
- **Development and Debugging**
  - Profile performance under load.
  - Understand parent-child relationships between processes.
- **System Administration**
  - Check if critical services are running.
  - Balance resource usage across users or processes.
  - Automate monitoring and alerts using scripts.

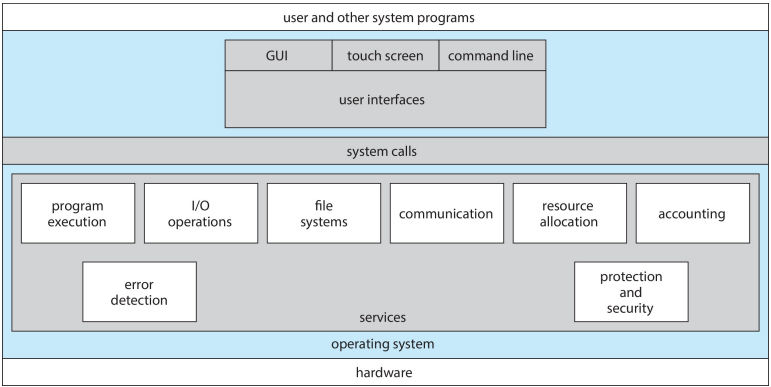
Week 3: OS Services

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 use 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.

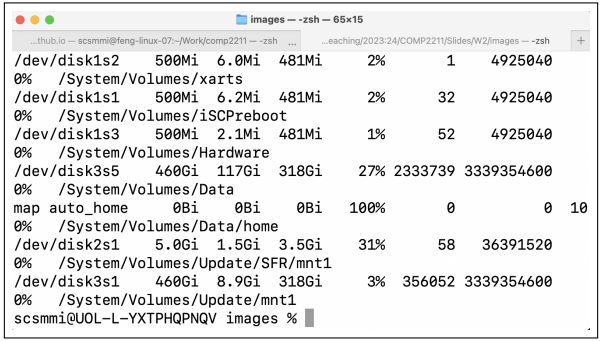
User and OS Interface: Command Interpreters

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 executables which the shell can invoke. The latter requires no modification to the shell when adding new commands.

User and OS Interface: Command Interpreters



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 GUIs—specialized commands only accessed through CLI.
- Easier to do repetitive tasks—commands can be recorded in a file and easily rerun (**shell scripts).**

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
```

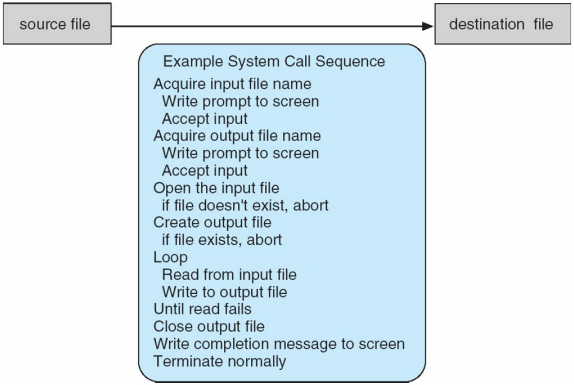
System Calls: an Example

```
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



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).

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>

ssize_t    read(int fd, void *buf, size_t count)
```

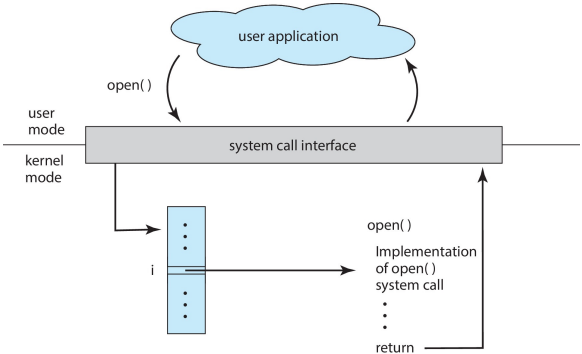
return value — function name — parameters

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.

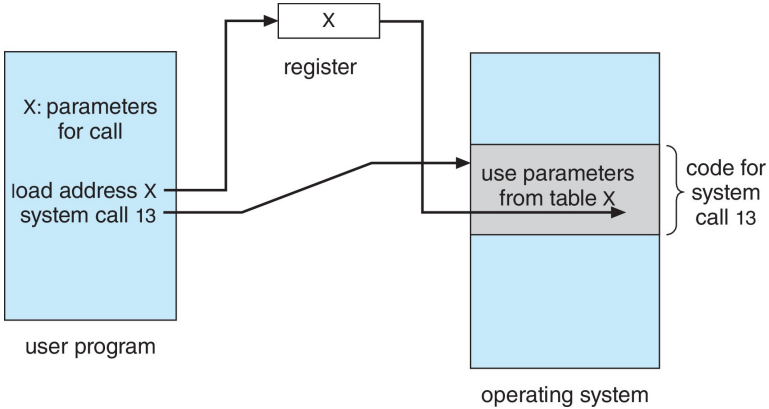
System Calls: Parameter Passing

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 system 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 permissions 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

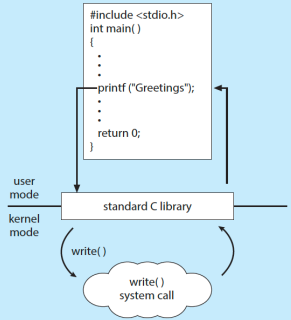
The following illustrates various equivalent system calls for Windows and UNIX operating systems.

	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

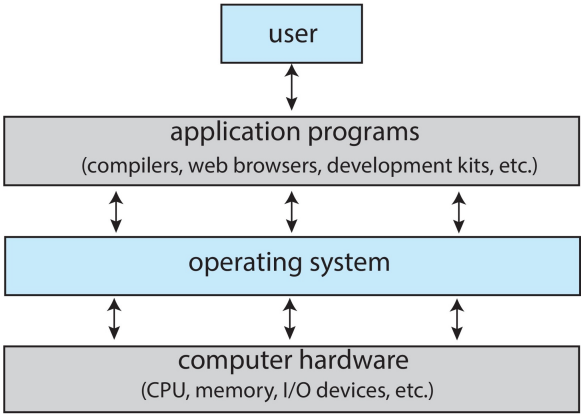
THE STANDARD C LIBRARY

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:



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**.



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.

Putting it together: confusing terminology?

Chapter 2 of OSC provides several terms that include "services":

- Operating System **services**: what functionality an OS provides at an abstract level.
- Operating System **system services**: an actual environment for using various OS services. Whole programs: e.g. a text editor, compiler.
- Operating System **system calls**: low-level calls to OS services. OS system services utilise these.

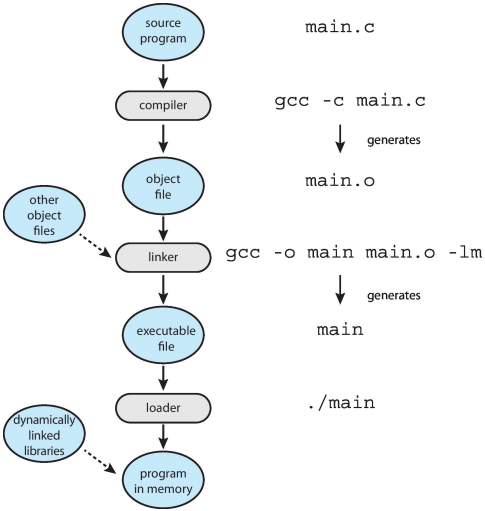
Part III: Code Compilation and Loading

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 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 system calls.
  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.

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

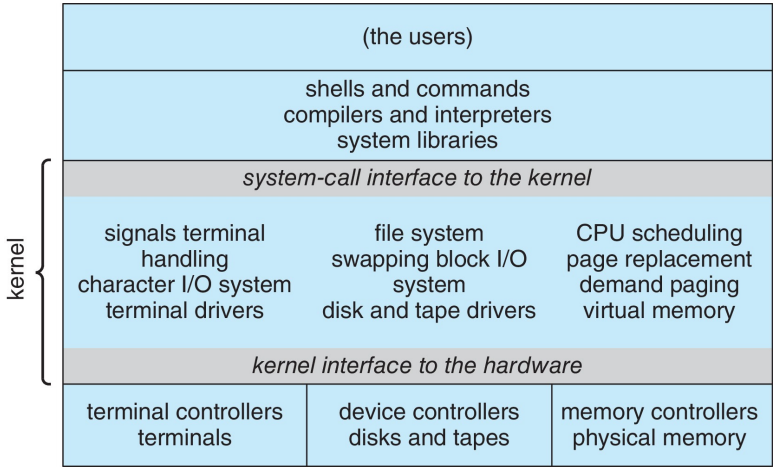
Code written faster, more compact, easier to understand and maintain. Compiler improvements easy to integrate by recompiling the OS. Easier to port the whole OS to new architectures.

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

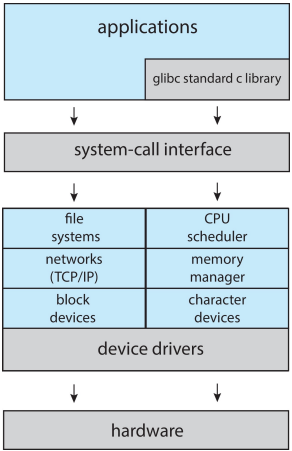


OS Structure

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 explains why they are still relevant.

OS Structure: Linux system



OS Structure

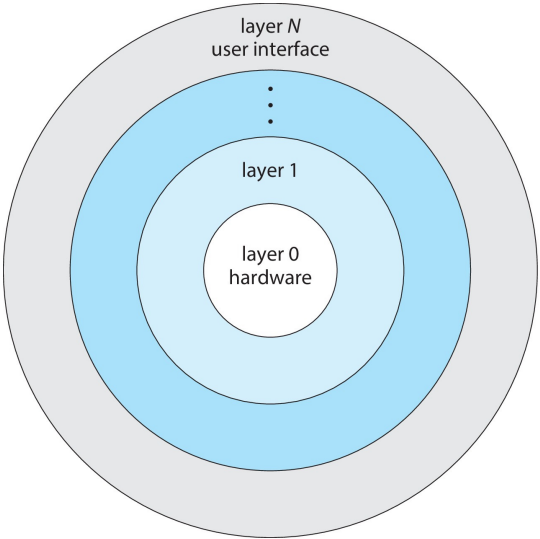
Monolithic kernels are said to be **tightly coupled** because changes in the system can affect all other parts.

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.

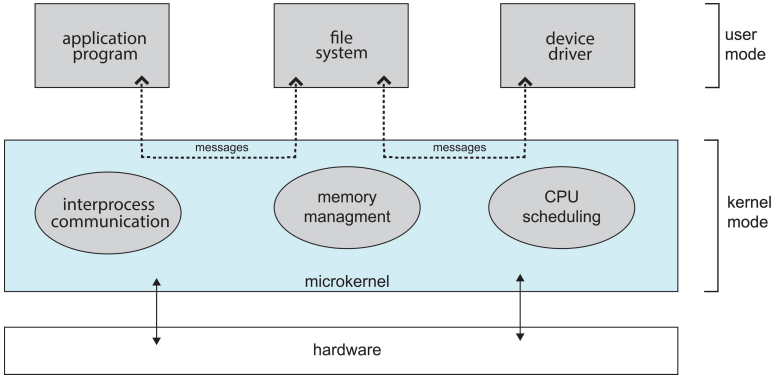
Remove all nonessential components from the kernel and implement them as user level programs—this results in a small kernel.

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



Week 4: CPU Scheduling

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**.

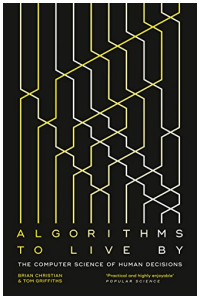
Part I: Description of the Problem

A General Problem

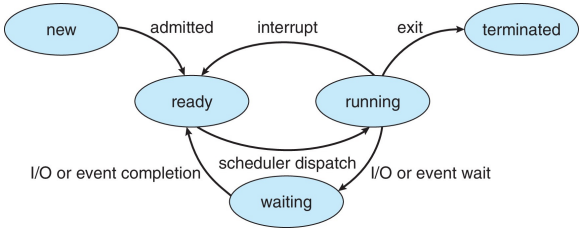
We are going to look at scheduling in operating systems, but it is a general problem (think about where else you can notice scheduling in everyday activities).

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

From Algorithms to Live By, Chapter 5 on Scheduling.

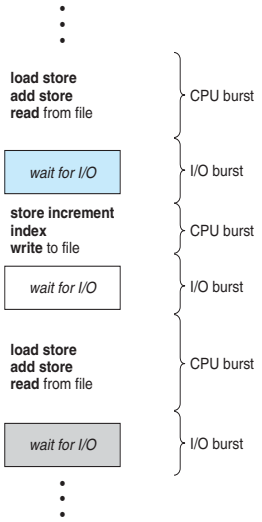


Why CPUs need scheduling?

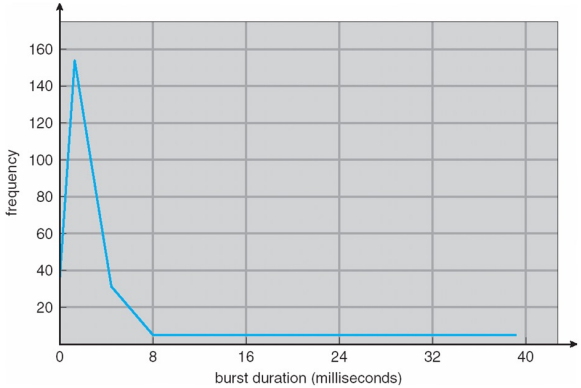


Why CPUs need scheduling?

- 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** (processes run as long as needed) while for 2 and 3 **preemptive** (running processes may be interrupted).

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** (suspend one processes, load another).
- 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 start of a response (not until the full output of the response is produced).

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

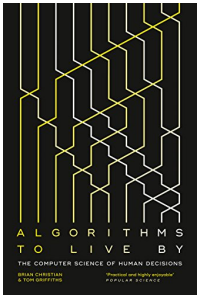
Convoy effect—short jobs can be held waiting by long jobs.

Note that **FCFS is nonpreemptive**.

Questions?

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

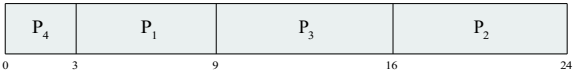
William James. From Algorithms to Live By, Chapter 5 on Scheduling.



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

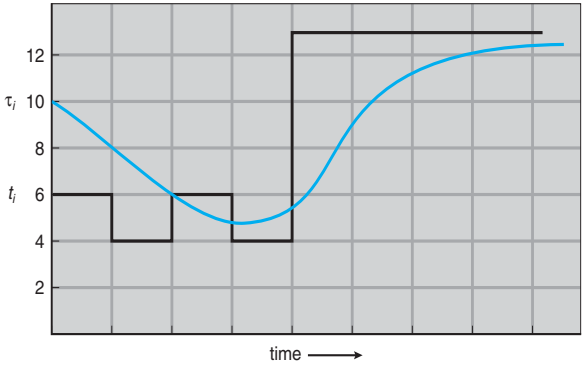
Predicting Lengths of Future CPU Bursts

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: $\alpha = 0.5$, $\tau_4 = 0.5t_3 + 0.25t_2 + 0.125t_1 + 0.0625\tau_0$.

Exponential average of past CPU bursts

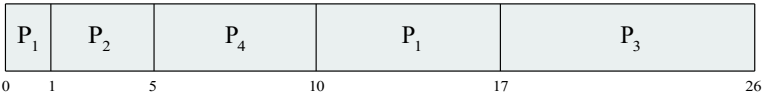
Each successive term has lower weighting than the newer ones, with the initial guess having the lowest.

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

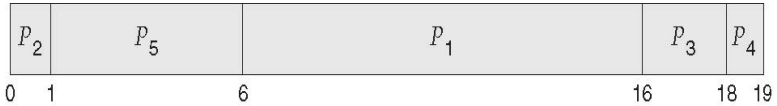
Priority scheduling

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



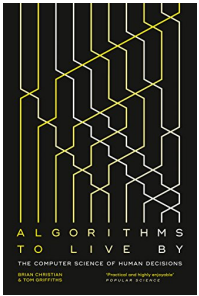
Preemptive priority scheduling

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.



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).

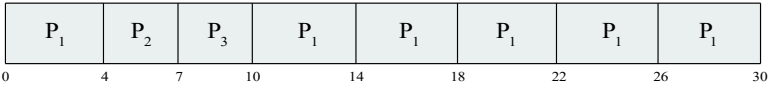
Round Robin (RR) Scheduling

- 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

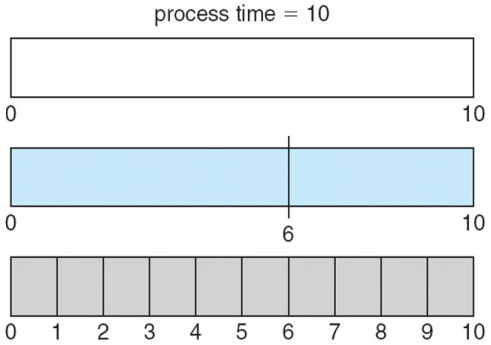
Take $q = 4$.

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



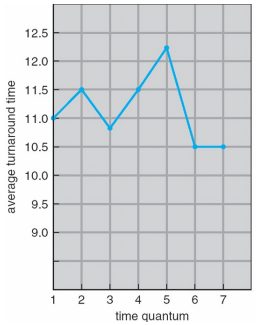
quantum	context switches
12	0
6	1
1	9

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.

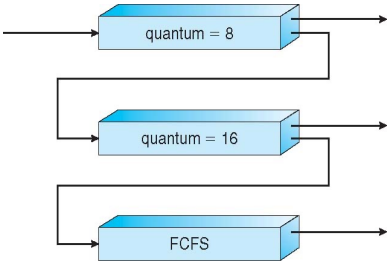
Multilevel Feedback Queue Scheduling (Example)

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.

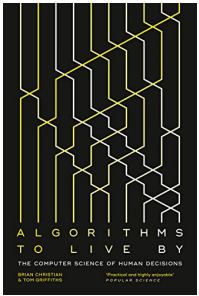
In practice?

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.



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.

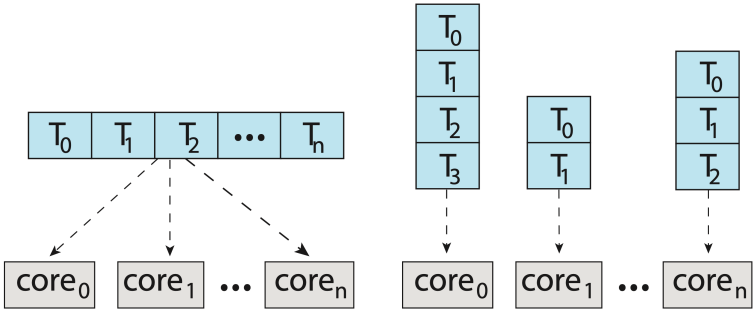
One approach to scheduling is to have one master processor handling scheduling (**assymetric multiprocessing**). Master becomes potential bottleneck.

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

Multi-Processor Scheduling: SMP

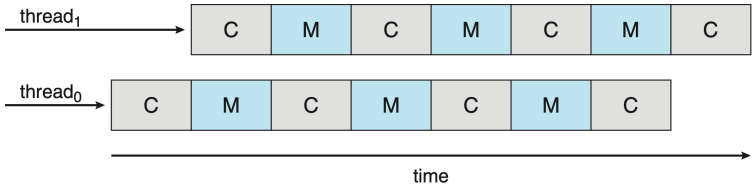
Two approaches in SMP

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

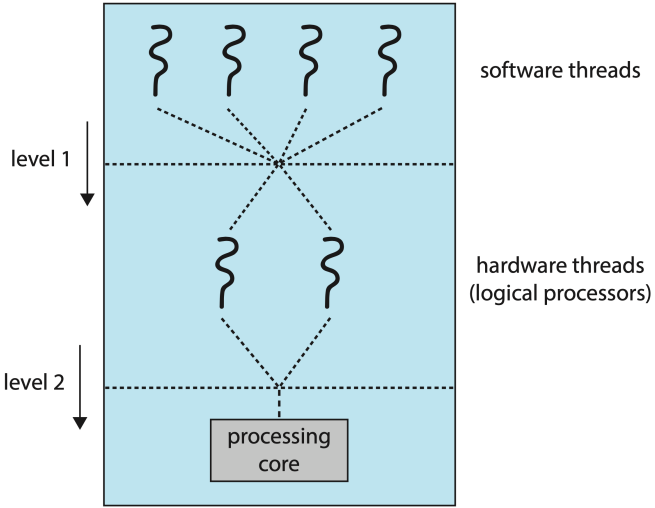


Multicore Processors

- 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.

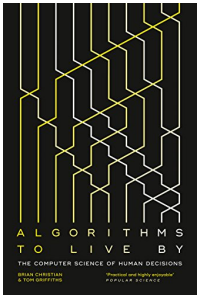
Implications on scheduling

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.



Part VI: Scheduling with Deadlines: Real-Time Processing

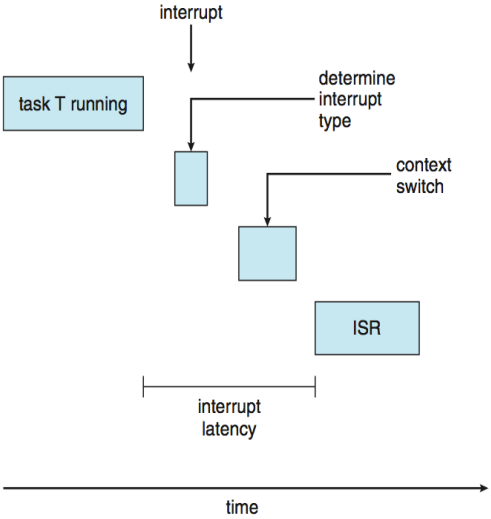
Real-Time CPU Scheduling

- 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.

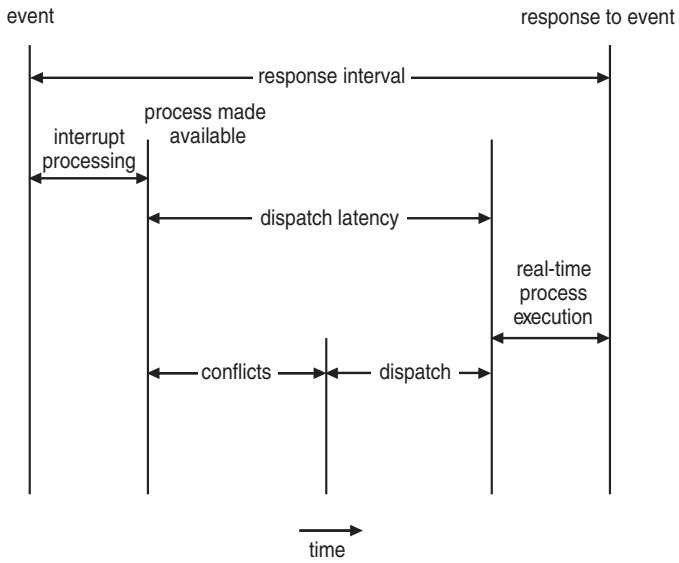
Hard real-time systems

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.

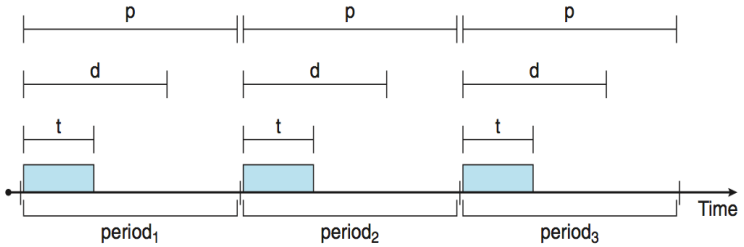
Additional scheduling features required for hard real-time.
Some definitions:

- 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

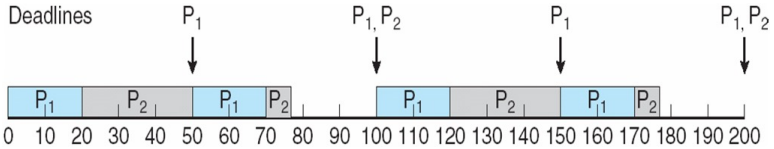
- Upon entering the system, each periodic task assigned priority $\propto \frac{1}{p}$.
- Rationale: prioritize processes that require CPU more often.

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:



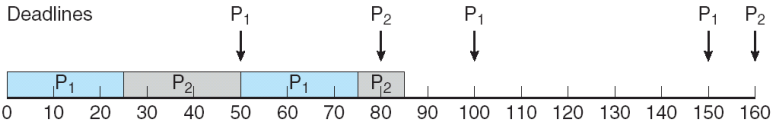
Rate-Monotonic Scheduling

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.

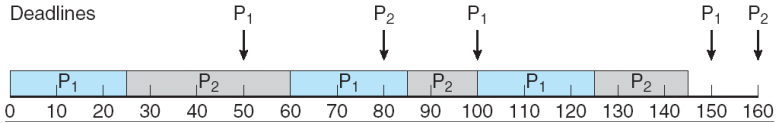
Rate-monotonic scheduler:



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

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.

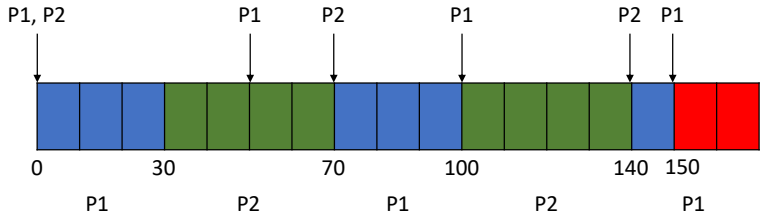
Process	p	t	d
P_1	50	30	50
P_2	70	40	70

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

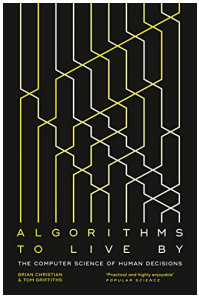
Scheduling occurs in two situations:

- 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.



Week 5: Memory Protection and Paging

Objectives

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

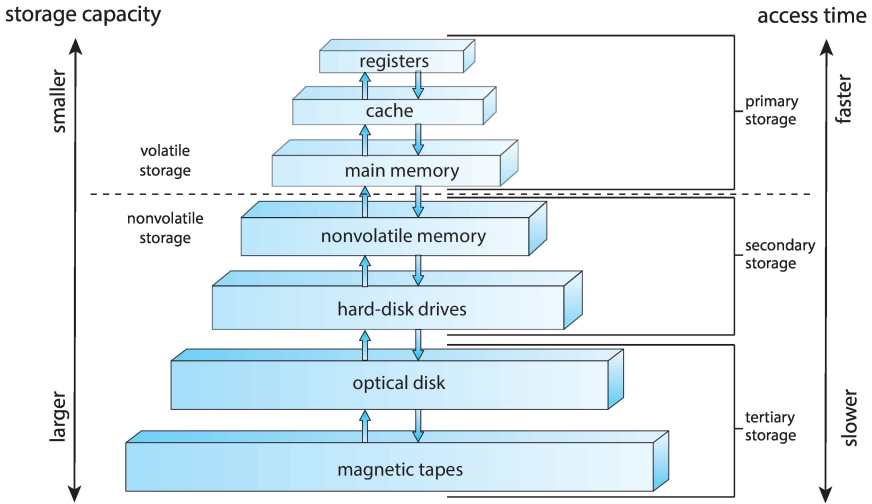
Part I: Description of the Problem

Memory operations

CPU can only access registers and main memory directly, with the *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

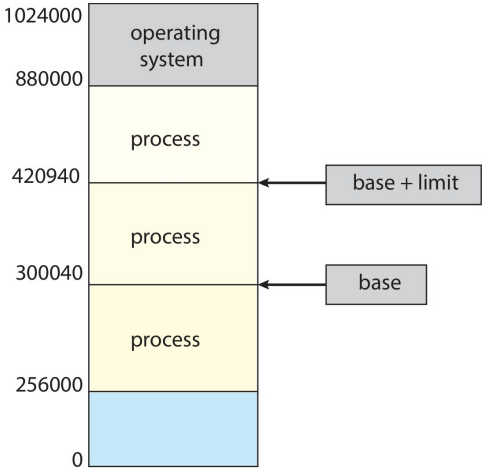


Memory protection

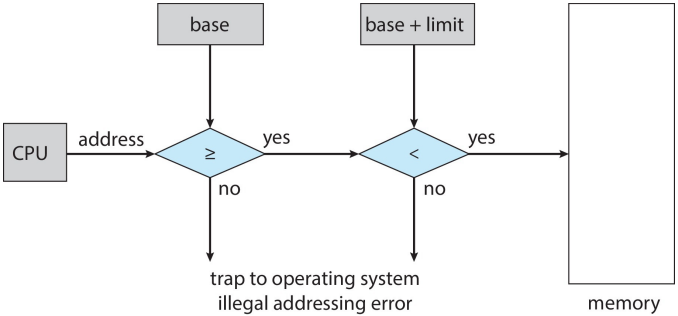
Why do we need memory protection

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

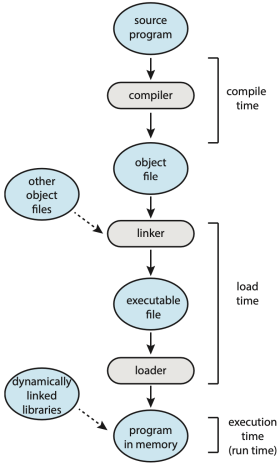
- Programs on disk have to be moved into memory eventually, for execution.
- We will place them in some location, not necessarily at address 0000.
- How to represent addresses before the decision of placement is made?
- Source programs contain **symbolic addresses** that the compiler binds to **relocatable 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 the most common set-up in operating systems today.

Address binding



Logical and physical address spaces

Logical address (virtual address)

Generated by the CPU (user processes).

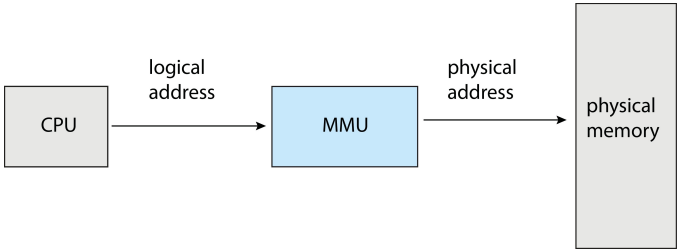
Physical address

Address that the memory unit works with.

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

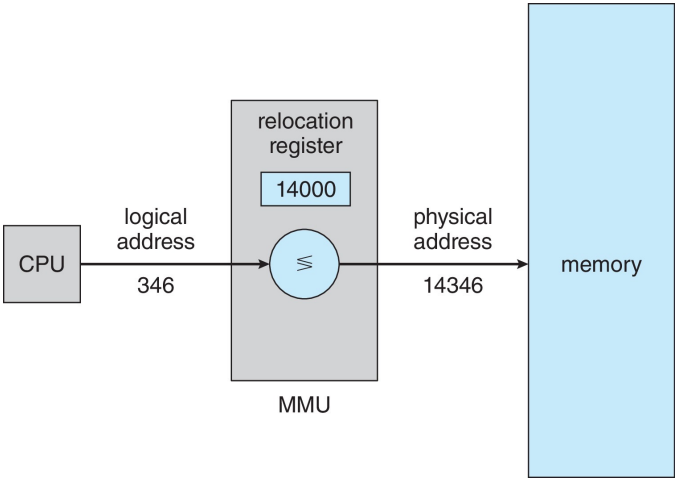
Compile-time and load-time binding results in equivalent logical and physical addresses. Execution-time binding makes processes think their address starts at 0000 and separate mapping is done to the physical address space.

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.

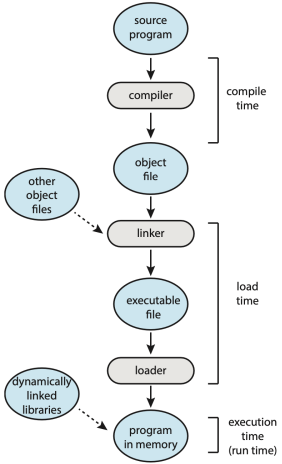
Dynamic linking

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:

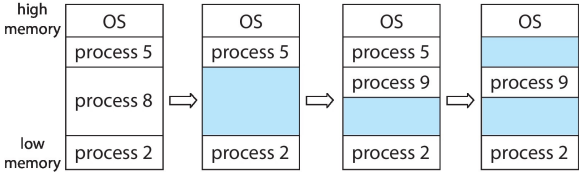
1. area for the operating system, and
2. area for processes, stored in single contiguous blocks.

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.

Part III: Paging

Motivation

Why use paging memory management technique

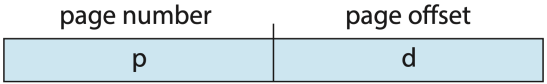
Previous techniques require memory to be contiguous, which introduces memory gaps and external fragmentation (requiring compaction). Paging allows processes to see contiguous memory space despite actual data stored in separate places in the physical memory.

Paging

A method that allows memory space of a process 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

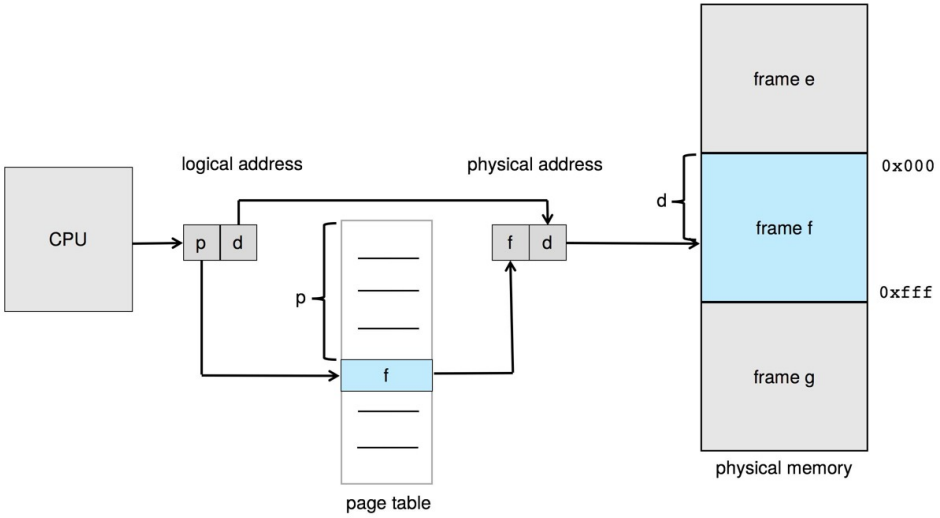
- Divide virtual memory space into fixed-size blocks, **pages**
- Divide physical memory space into fixed-size blocks, **frames**
- Keep track of free frames
- When a process requires N pages, N free frames have to be found
- **Page tables** map pages to frames (**virtual addresses** to **physical addresses**).



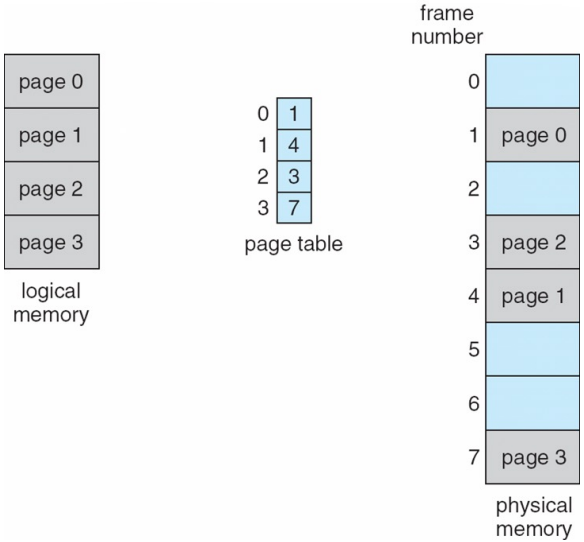
Virtual addresses

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.

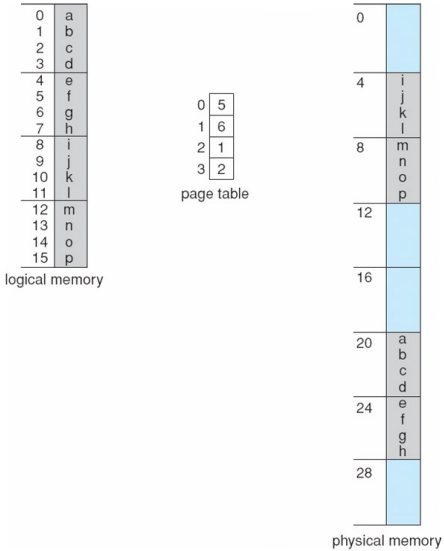
Page tables: diagram overview



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

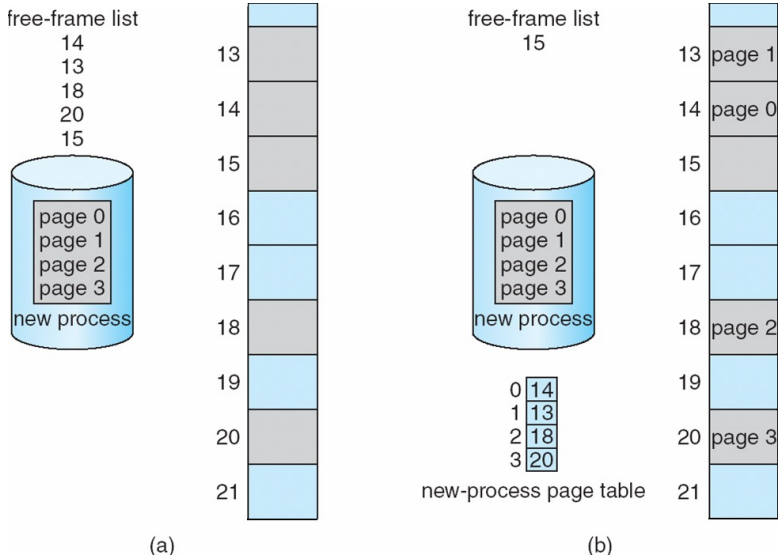
The main steps:

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 (PCB) of each process stores the address of the page table.
- When the 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.

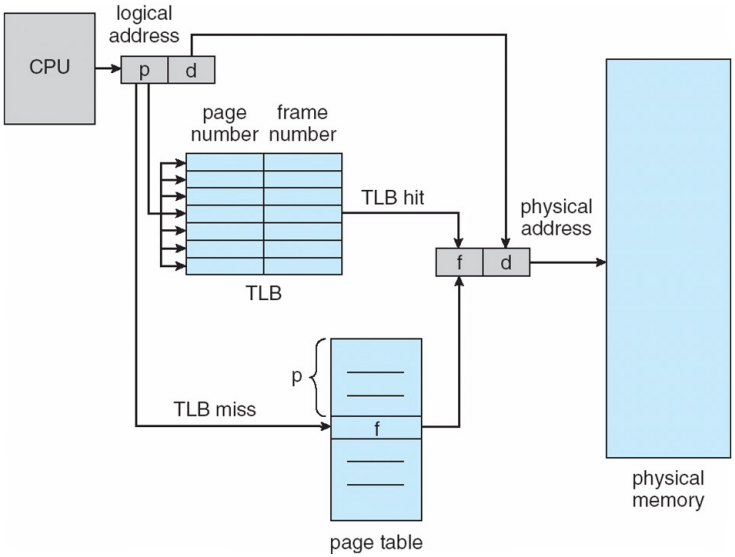
Translation look-aside buffer (TLB)

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

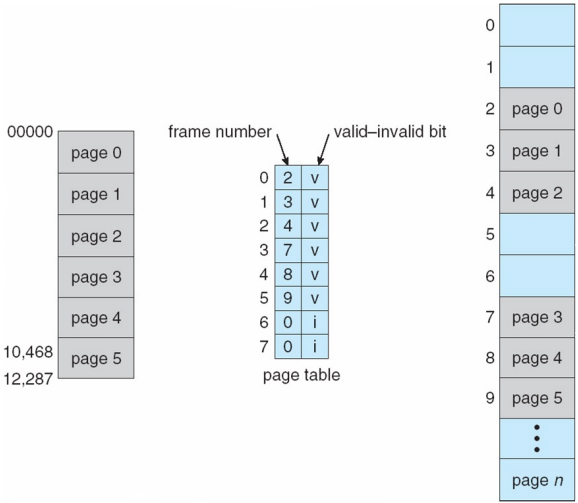
Modern CPUs provide multiple levels of TLBs. For example Intel Core i7 has 128-entry L1 instruction TLB and 64-entry data TLB. When TLB miss occurs, it checks in the 512-entry L2 TLB. In case of another TLB miss, page table in the main memory is looked at.

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.



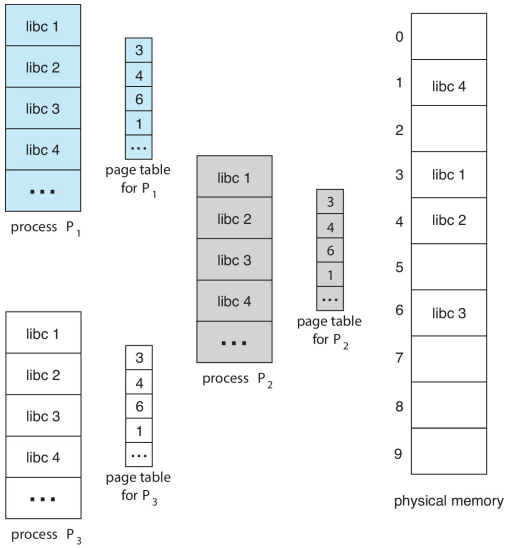
Protection with pages

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.

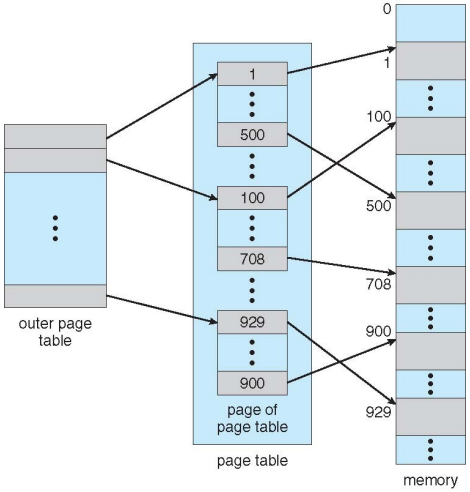
Shared pages

- 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.

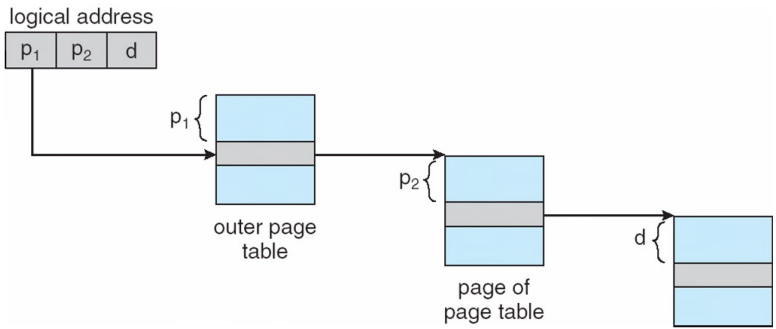


Hierarchical paging

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

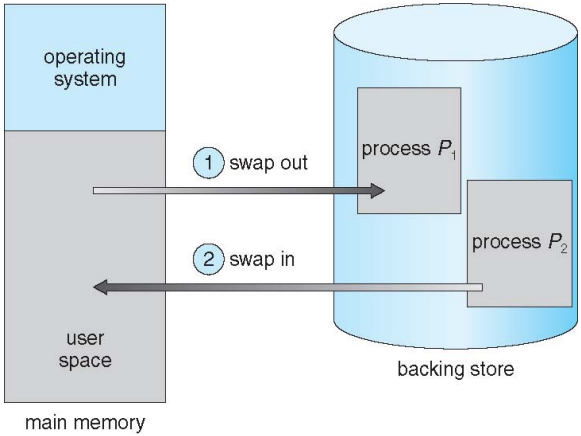
For 64-bit architectures even the two-level hierarchical page table requires too many entries. Other solutions are **hashed page tables** and **inverted page tables**—check OSC Chapter 9 for details.

Part IV: Memory Swapping

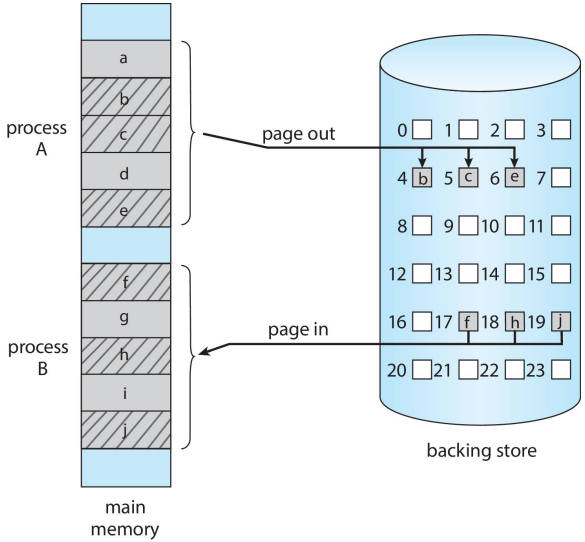
Standard swapping

- 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 the disc.

Standard swapping



Swapping with paging

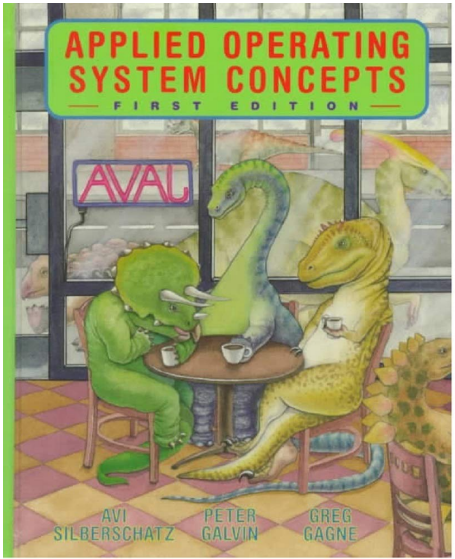


Swapping in mobile systems

- 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.

Week 6: Virtual Memory

Further reading?



Objectives

- Introduce **virtual memory**.
- Discuss demand paging.
- Consider why page replacement is essential.

Part I: Introduction to Virtual Memory

Virtual memory: introduction

- Memory management we discussed so far is stemming from a basic requirement: **instructions for several processes should be in memory for execution (multiprogramming)**.
- 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 so their code is almost never executed,
  - 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 (**CPU utilisation and throughput**), and
- less I/O needed when loading/swapping portions of programs.

Virtual memory: introduction

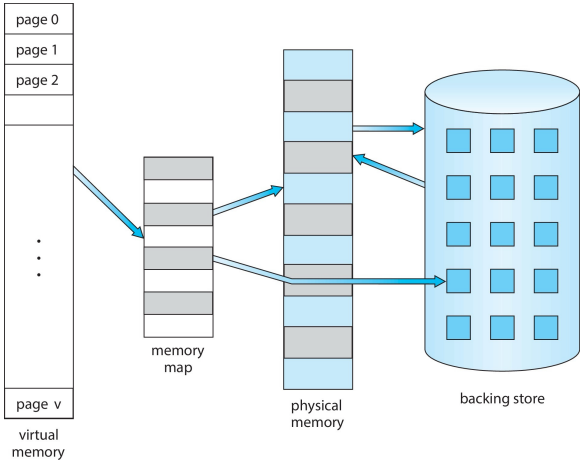
Virtual memory

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

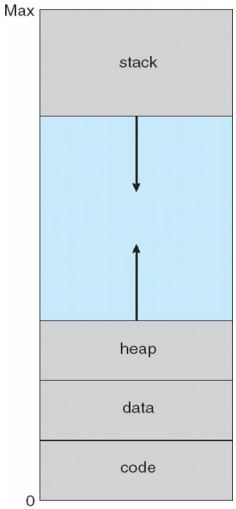
Virtual memory allows the programmer not to worry about the physical space limitations and concentrate on solving the problem in the virtual space. The virtual space starts at a certain address and is contiguous. MMU maps this space to the noncontiguous physical space.

Virtual memory: introduction



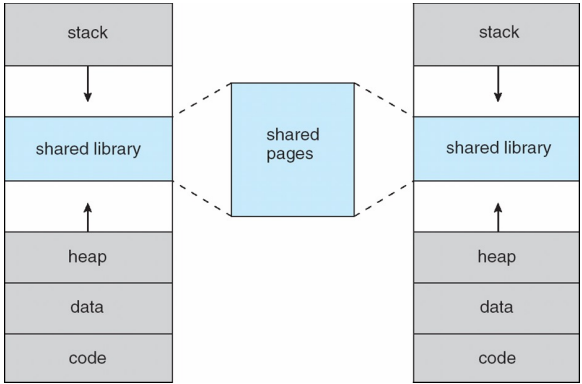
Virtual memory: introduction

- 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.
- What happens when stack meets heap? OS usually will have limits and cause an error to inform the user: segmentation fault, stack overflow, malloc failure.



Virtual memory: introduction

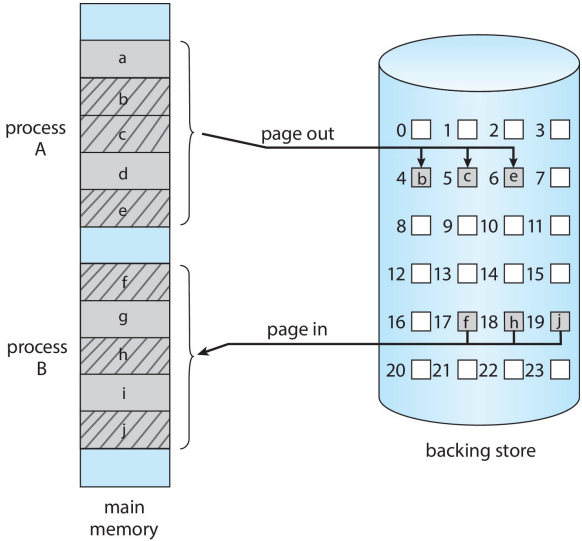
- Each process thinks that the shared memory is in their virtual address space, but the logical addresses are mapped to the same shared pages in the physical memory.
- System libraries: each process thinks it's in their virtual space, but pages are shared.
- Process shared-memory communication: each process thinks shared pages are in their virtual address space.
- `fork()` can set up shared pages; faster process forking



Part II: Demand Paging

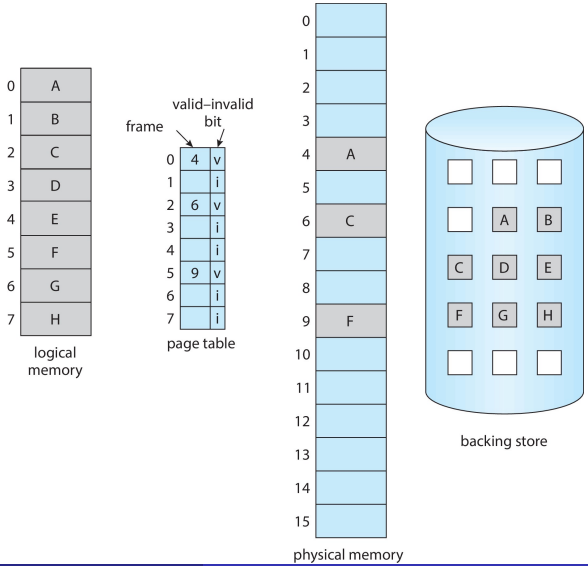
Demand paging: motivation

- When loading a process, should we bring all of its pages into the physical memory?
- A user may not need an entire large program.
- Instead, **bring pages in only when needed at run time**.
- Similar to paging with swapping (diagram on the right).
- Pages that are not accessed are never loaded.



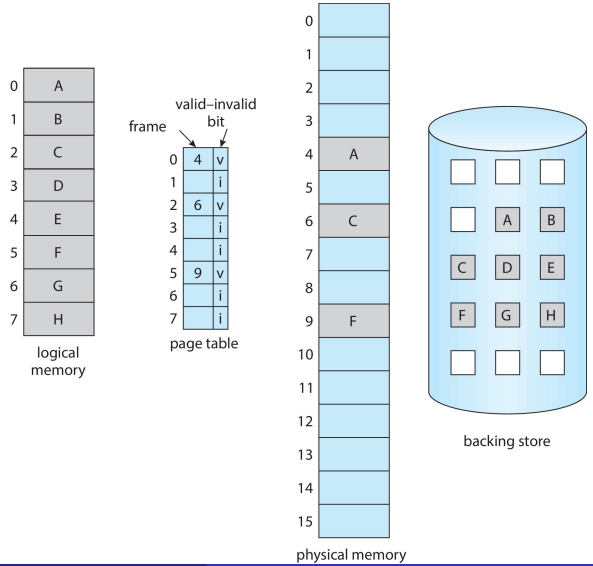
Demand paging: basic concepts

- 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.



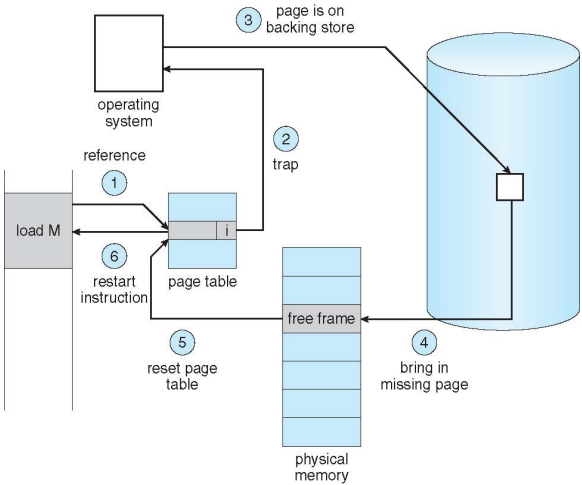
Demand paging: basic concepts

- If a process makes access to a page that is in the backing store, **page fault** occurs (OS trap interrupt 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: actions on page fault

1. If access is invalid (address out of bounds), terminate process.
2. Otherwise, find a free frame in memory.
3. Copy the page from disk to a new frame in memory.
4. When copied, modify page table to indicate the page is in.
5. Restart instruction—the process continues as normal.



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.

The main requirement of demand paging

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.

Worst-case example: a three-address instruction

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 sitting in that frame (consider what would happen if we did not do this).
- Free-frame list also is modified when the stack or heap segments of the process expand.
- On system start up, all available memory is placed on **free-frame list**.
- At some point it will become empty and need repopulating (see further).

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 process
7. Receive an interrupt from the disk I/O subsystem (I/O completed)
8. Save the registers and process state for the other process
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

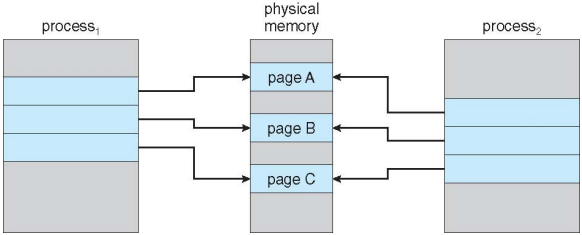
Copy-on-write

- 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.
- **These pages are marked copy-on-write.**
- Then, on write, a copy of the page will be made that is then written to by one of the processes—at this point parent and child memory states diverge.
- Pages that cannot be modified (such as executable code) are not marked copy-on-write and can be always shared.

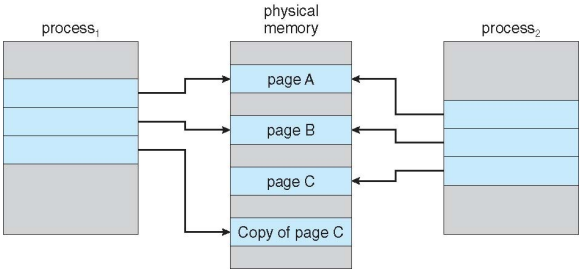
For interest

Look into how Linux `vfork()` differs from `fork()`.

Copy-on-write



Copy-on-write



Part III: Page Replacement

Consider an example

- Physical memory can fit 40 frames.
- User wants to start 8 processes, each needing 10 pages.
- Assume each uses only 5 pages mostly.
- Instead of loading 4 processes, load 8 processes, but only half of their pages (5).
- **Degree of multiprogramming**: $4 \rightarrow 8$ (processes active in the main memory).
- **Overallocation of memory**: 80 frames needed, but 40 available.
- CPU utilisation higher, throughput higher.

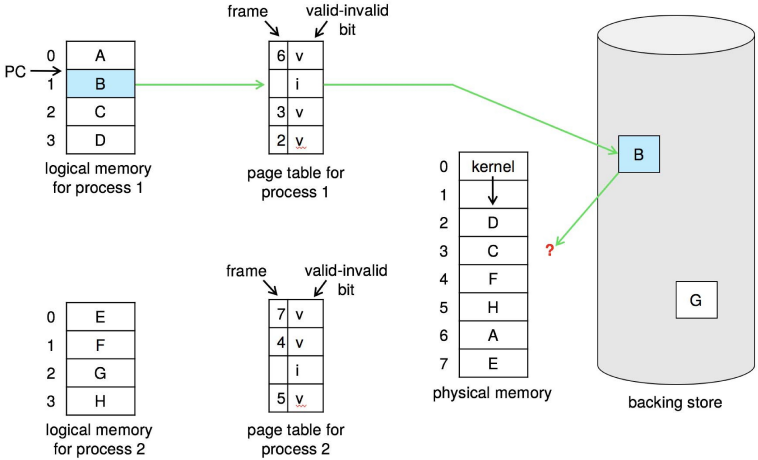
Potential problem

One of 8 processes suddenly needs more than 5 of its pages, but the **memory is full** with 40 frames in. **Page fault cannot be resolved?**

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?



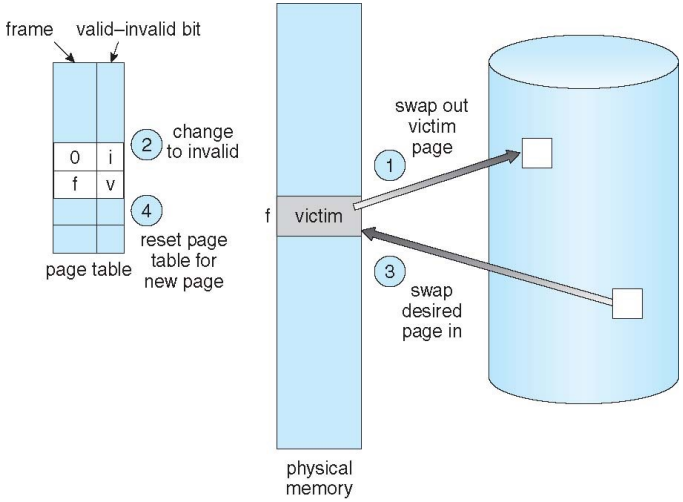
General page-replacement algorithm

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 **a 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

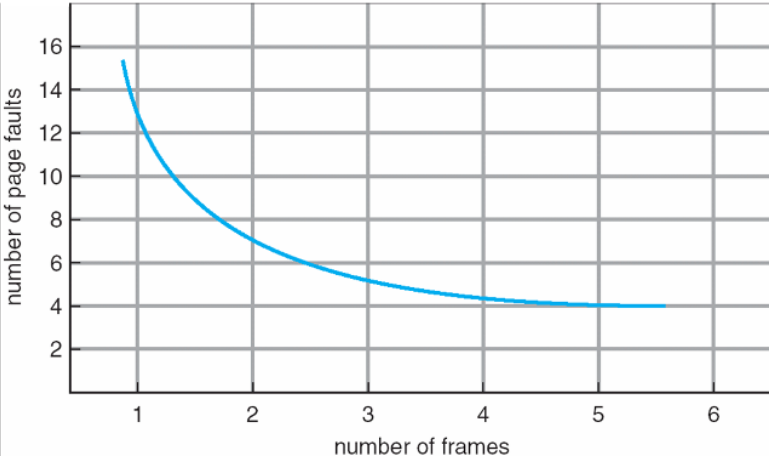
Some common algorithms for picking the **victim page**:

- **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

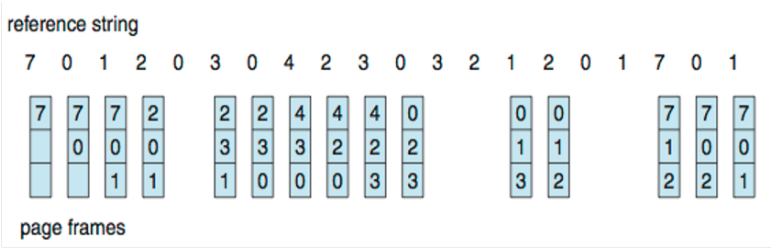
FIFO is the cheapest algorithm, but has an anomaly—increasing number of frames increases page faults. In general, any improvements to demand paging will yield large benefits because copying data from backing store take relatively long. Usually lowest page fault frame is a good measure to judge the algorithm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

Page faults as the main measurement of algorithm effectiveness



FIFO page replacement algorithm

- Assume 3 frames fit in memory.
- Consider page accesses in sequence as shown by the reference string.



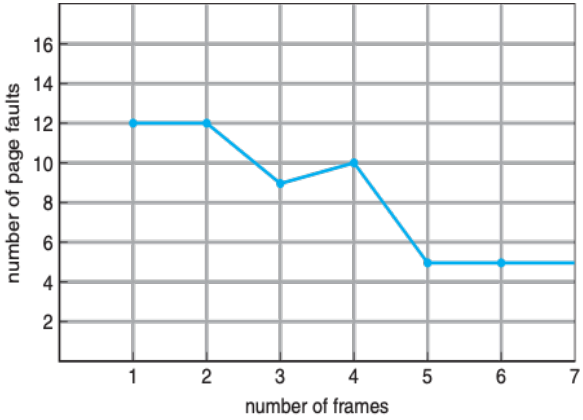
15 page faults.

Main downside with FIFO alg.

Old frame does not mean it is not used. If we move the page out, it can cause a page fault soon and it will be moved back in.

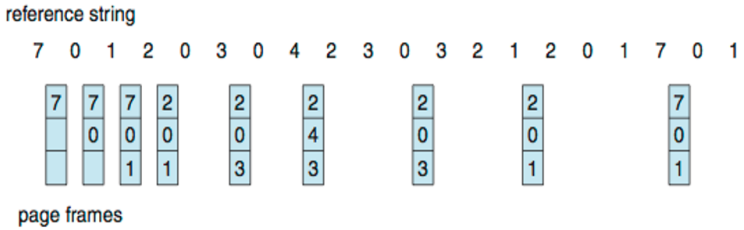
FIFO page replacement algorithm

There is an anomaly with FIFO—increasing physical memory size may not decrease the page fault rate. This is called **Balady's anomaly**.



OPT page replacement algorithm

- Replace the page that will not be referenced for the longest period of time.
- It is known as the best algorithm in terms of page fault reduction.



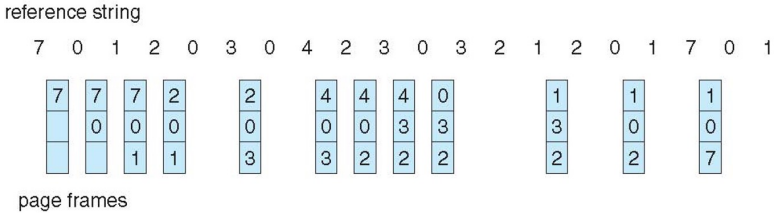
9 page faults.

Main downside with OPT alg.

Need to know future page accesses. **Similar issue to SJF scheduling algorithm which we have discussed in Week 4.**

LRU page replacement algorithm

- Replace the page in memory that was least recently used. 12 page faults.



Main downside with LRU alg.

Cannot see that a page will soon be used—still replaces it if it was not used for a long time. Still, better than FIFO, and generally LRU is considered a good page replacement algorithm.

OPT and LRU page-replacement algorithms

Do not suffer from Belady's anomaly. Implementation complex compared with FIFO.

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/6.828/2025/xv6/book-riscv-rev5.pdf
Version of Sep. 2, 2025

References II

R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau
Operating Systems: Three Easy Pieces. Version 1.10
https://pages.cs.wisc.edu/~remzi/OSTEP/
Arpaci-Dusseau Books. Nov., 2023