01. Introduction

Ch. 1, 2

Course Structure

Objectives

- To describe the basic organisation of computer systems
- To give an abstract view of the operating system
- To introduce some key concepts in (operating) systems
- To give a brief tour of the major functions of the operating system

- Recall Part 2 of Introduction to Microprocessors in IA Digital Electronics
  - Fetch-Decode-Execute cycle, Pipelining

Outline

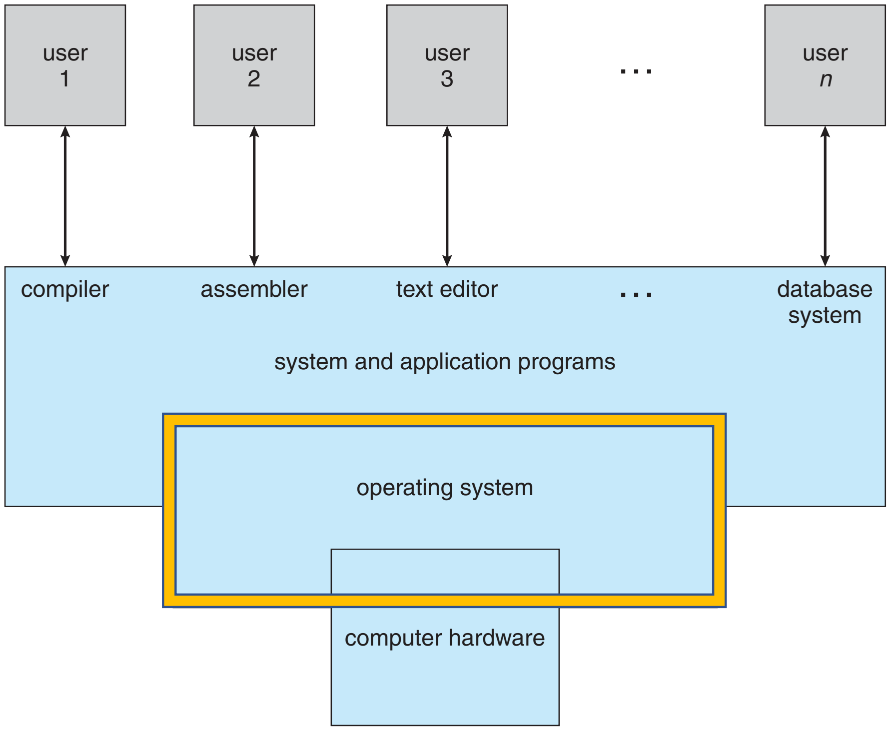
- System organisation
- System operation
- Concepts
- What is an Operating System?

Outline

- System organisation
  - Hardware resources
  - Fetch-execute cycle
  - Buses
- System operation
- Concepts
- What is an Operating System?

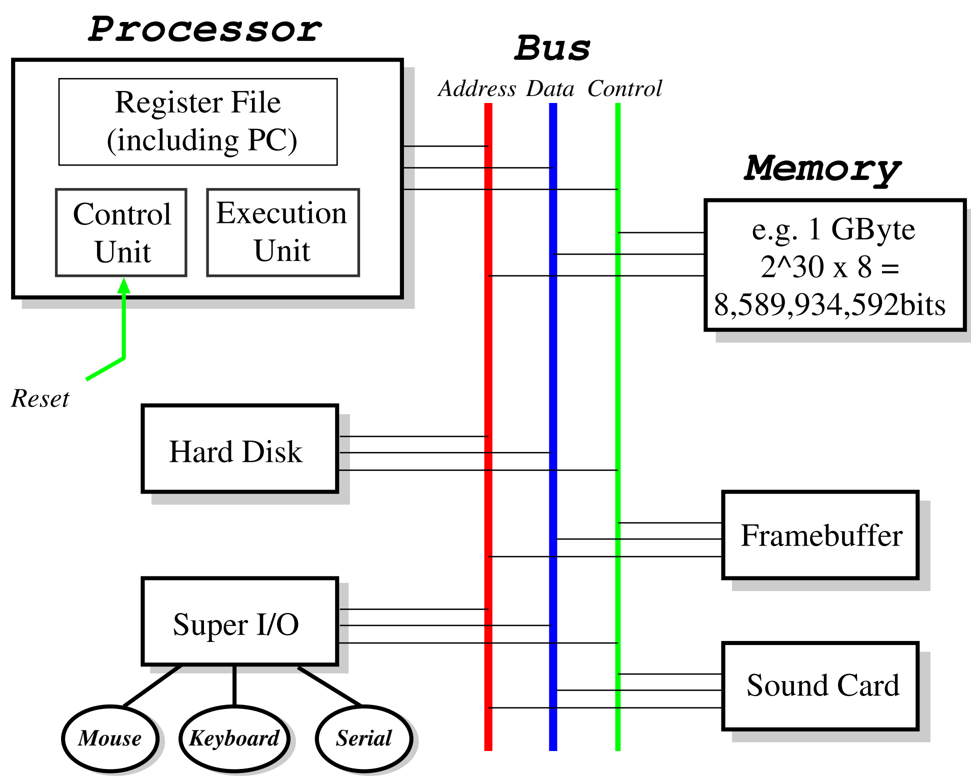
Computer system organisation

1. **Hardware** provides basic computing resources: CPU, memory, I/O devices
2. **Operating system** controls and coordinates use of those resources
3. **Application programs** define how those resources are used to solve the computing problems of the users
4. **Users** motivate the whole thing!

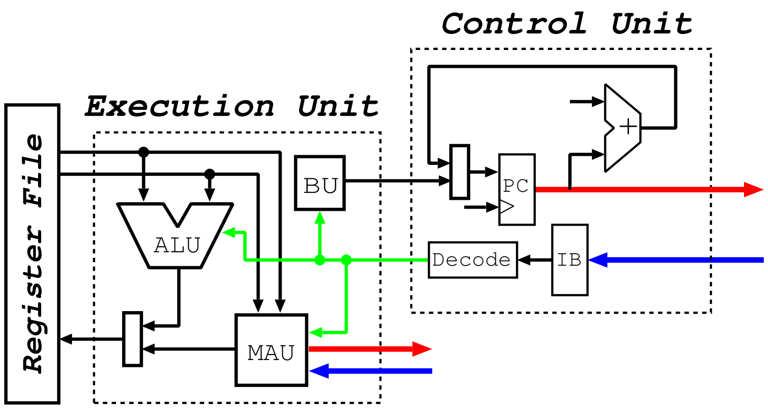


Hardware resources

- **Processor** (**CPU**) executes programs using
    - **Memory** to store both programs & data, effectively a large byte-addressed array,
    - **Devices** for input and output, and
    - **Bus** to transfer information between
- CPUs operate on data obtained from input devices and held in memory
    - CPUs and devices are concurrently active, competing for memory cycles and bus access
- Computer logically
    - Reads values from main memory into registers,
    - Performs operations, and
    - Stores results back

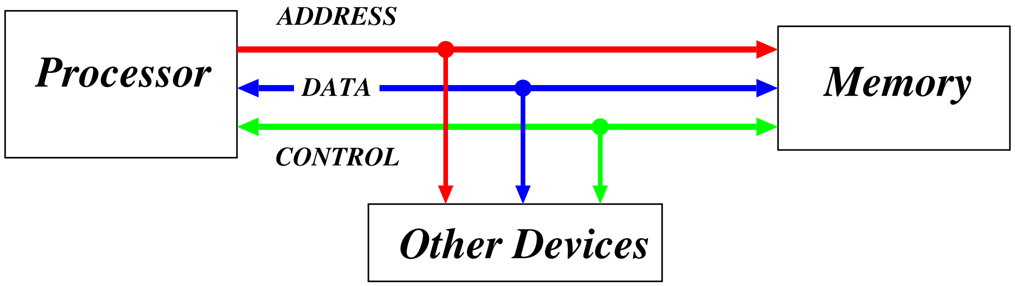


Fetch-Execute Cycle



- CPU repeatedly
  - Fetches & decodes next instruction,
  - Generating control signals and operand information
- Inside the **Execution Unit** (EU), control signals select the **Functional Unit** (FU) (“instruction class”) and operation
  - If **Arithmetic Logic Unit** (ALU), read one/two registers, perform operation, (probably) write result back
  - If **Branch Unit** (BU), test condition and (maybe) add value to PC
  - If **Memory Access Unit** (MAU), generate address (“addressing mode”) and use bus to read/write value

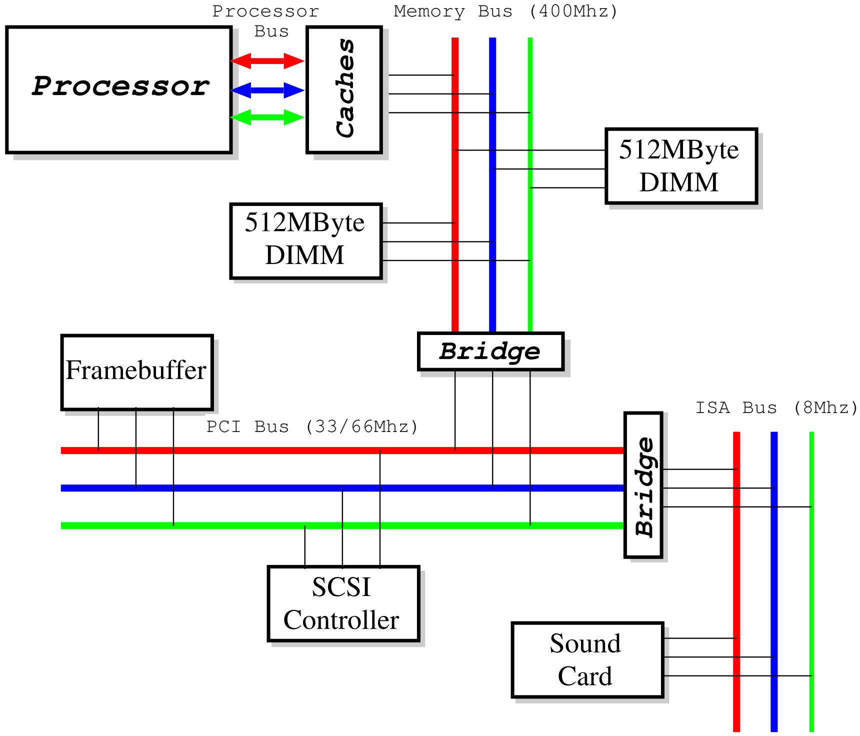
Buses



- Shared communication wires
  - Don't need wires everywhere!
  - Low cost, versatile
  - Potential bottleneck
- Typically comprises:
  - **address lines** determine how many devices on bus,
  - **data lines** determine how many bits transferred at once, and
  - **control** lines indicate target devices and selected operations
- Operates in a initiator-responder manner, e.g.,
  - Initiator decides to read data
  - Initiator puts address onto bus and asserts read
  - Responder reads address from bus, retrieves data, and puts onto bus
  - Initiator reads data from bus

Bus hierarchy

- Different buses with different characteristics
  - E.g., data width, max number of devices, max length
  - Most are synchronous, i.e. share a clock signal
- **Processor bus** is the fastest and often the widest for CPU to talk to cache
- **Memory bus** to communicate with memory
- **PCI buses** to communicate with devices
  - Other legacy buses also seen: ISA, EISA etc
- **Bridges** forwards from one side to the other
  - E.g., to access a device on ISA bus, CPU generates magic [physical] address which is sent to memory bridge, then to PCI bridge, and then to ISA bridge, and finally to ISA device

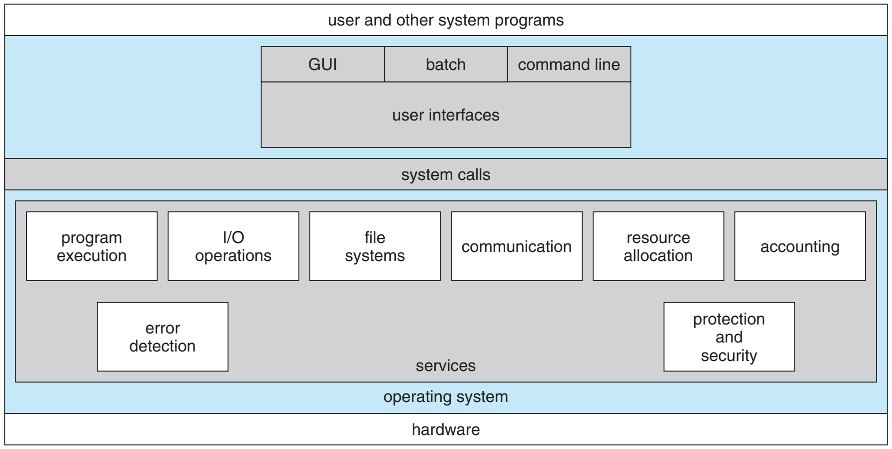


Outline

- System organisation
- System operation
  - Booting
  - Interrupts
  - Storage
- Concepts
- What is an Operating System?

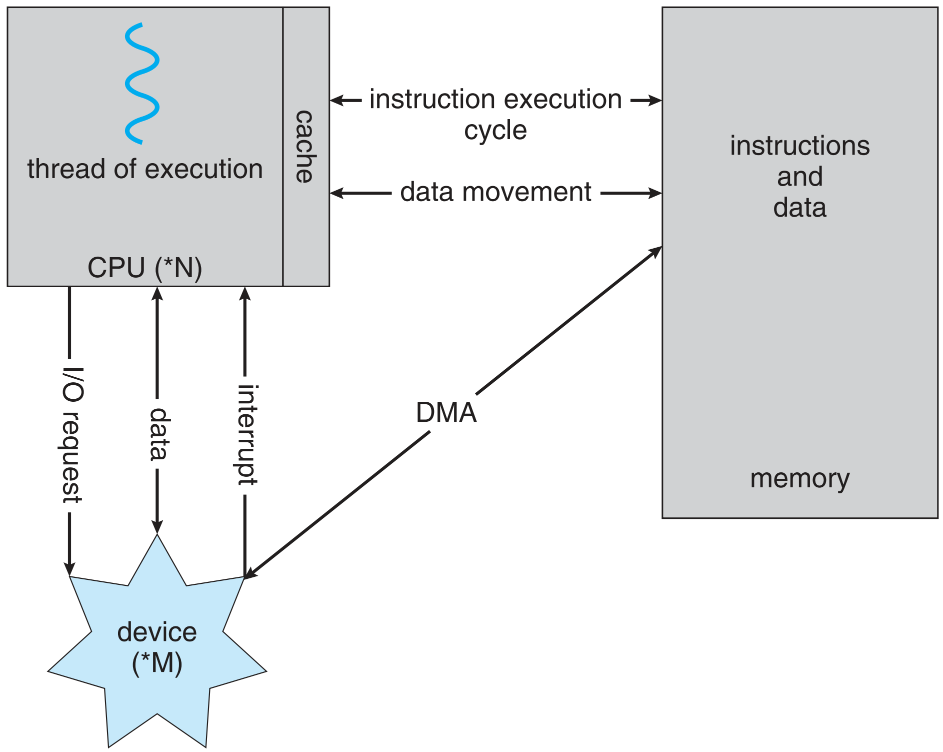
Booting the computer

- **Bootstrap program** (bootloader) executes when machine powered on
  - Traditionally ROM containing BIOS, now more complex UEFI
  - Initialises all parts of the system: memory, device controllers
  - Finds, loads, and executes the kernel, possibly in stages
- Operating system starts in stages
  - **Kernel** enables processes to be created, devices to be read/written, file system to be accessed
  - Then system processes start, beginning with *init* on Unix



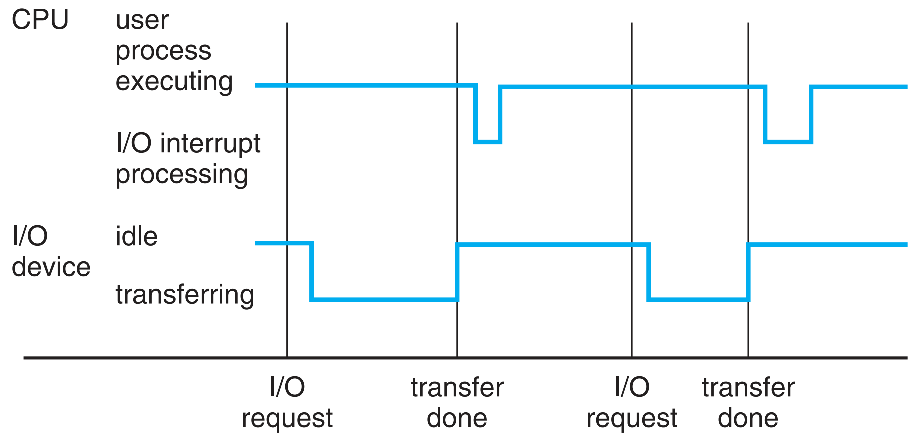
System operation

- I/O devices and CPU execute concurrently
- Each device controller
  - responsible for a particular device type
  - has a local buffer
- CPU moves data from/to main memory to/from local buffers
  - I/O is from the device to local buffer of controller
- Device controller informs CPU that it has finished its operation by raising an **interrupt**
  - OS is interrupt driven



Interrupts

- Device controllers communicate with CPU via **interrupts**
  - Controller controls interaction between device and local buffer
  - CPU moves data between main memory and device buffer



- Interrupts decouple CPU requests from device responses
  - Reading a block of data from a hard-disk might take 2ms, which could be 5×10^6 clock cycles!
- Controller informs CPU it is finished by **raising an interrupt**

Interrupt handling

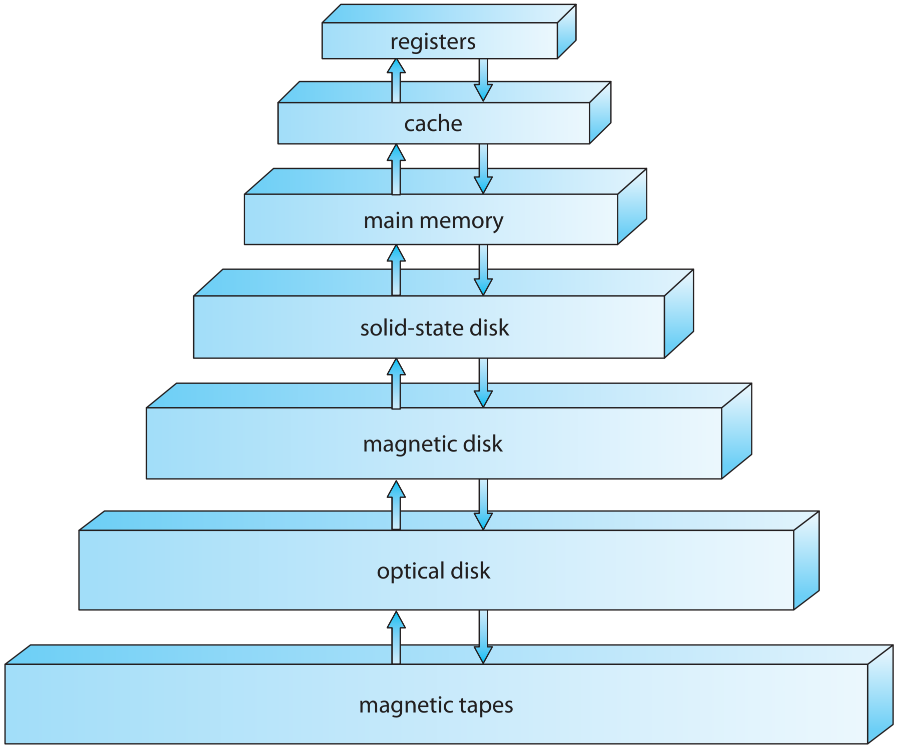
- A raised interrupt must be handled
  - Transfer control to the **interrupt service routine** (**ISR**) via
  - The **interrupt vector**, a table containing addresses of all the ISRs
  - Interrupt architecture saves the address of the interrupted instruction
  - After reading from device, CPU resumes using a special instruction, e.g., *rti*
- Interrupts can happen at any time
  - Typically deferred to an instruction boundary
  - ISRs must not trash registers, and must know where to resume
  - CPU thus typically saves values of all (or most) registers, restoring on return
- A **trap** or **exception** is a software-generated interrupt
  - Can be caused either by an error or a deliberate user request

Storage definitions

- Basic unit of computer storage is the **bit**, containing either 0 or 1
- A **byte** (or **octet**) is 8 bits, typically the smallest convenient chunk of storage
  - E.g., most computers can move a byte in memory but not a single bit
- A **word** is a given computer architecture's native unit of data, one or more bytes
  - E.g., a computer with 64-bit registers and 64-bit memory addressing typically has 64-bit (8-byte) words
- Storage generally measured and manipulated collections of bytes
  - A **kilobyte** (**KB**) is 1,024 bytes
  - A **megabyte** (**MB**) is $1{,}024^2$ bytes
  - A **gigabyte** (**GB**) is $1{,}024^3$ bytes
  - A **terabyte** (**TB**) is $1{,}024^4$ bytes
  - A **petabyte** (**PB**) is $1{,}024^5$ bytes
- Manufacturers often round so a megabyte is 1 million bytes and a gigabyte is 1 billion bytes

Storage hierarchy

- Storage systems organized in hierarchy
  - Speed, cost, volatility
- **Main memory** that the CPU can access directly
  - Large, random access, typically volatile
- **Secondary storage** extends main memory
  - Very large, non-volatile
  - **Hard disks** (**HDs**), rigid metal or glass platters covered with magnetic recording material divided logically into tracks, which are subdivided into sectors
  - **Solid-state disks** (**SSDs**), faster than hard disks, non-volatile
- **Device Driver** for each device controller to manage I/O provides a uniform interface between controller and kernel



Storage performance

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

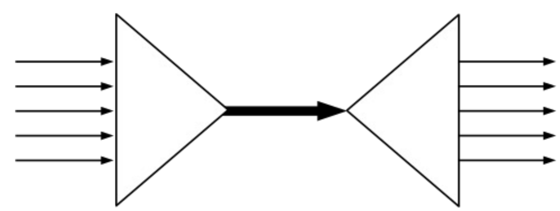
Outline

- System organisation
- System operation
- Concepts
  - Layering, multiplexing
  - Latency, bandwidth, jitter
  - Caching, buffering
  - Bottlenecks, tuning, 80/20 rule
  - Data structures
- What is an Operating System?

Layering, multiplexing

- **Layering** is a means to manage complexity by controlling interactions between components:
  - arrange components in a stack and restrict a component at layer X from
  - relying on any other component except the one at layer X-1 and
  - providing service to any component except the one at layer X+1
- **Multiplexing** is where one resource is being consumed by multiple consumers simultaneously
  - Traditionally, the combination of multiple (analogue) signals into a single signal over a shared medium

Internet	OSI
Application	Application
	Presentation
	Session
Transport	Transport
Internet	Network
Physical	Data Link
	Physical



Latency, bandwidth, jitter

- Different metrics of concern to systems designers
  - **Latency** is how long something takes
    E.g., “This read took 3ms”
  - **Bandwidth** is the rate at which something occurs
    - E.g., “This disk transfers data at 2Gb/s”
  - **Jitter** is the variation (statistical dispersal) in latency (frequency)
    - E.g., “Scheduling was periodic with jitter 50 μsec”
- Be aware
  - is it the absolute or relative value that matters, and
  - is the distribution of values also of interest

Caching, buffering

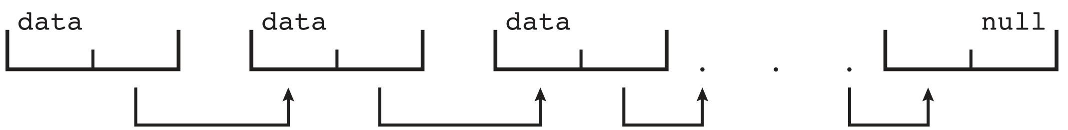
- Often need to handle two components operating at different speeds (latencies, bandwidths) – so-called **impedance mismatch**
- **Caching**, where a small amount of higher-performance storage is used to mask the performance impact of a larger lower-performance component. Relies on locality in time (finite resource) and space (non-zero cost)
  - E.g., CPU has registers, L1 cache, L2 cache, L3 cache, main memory
- **Buffering**, where memory of some kind is introduced between two components to soak up small, variable imbalances in bandwidth
  - E.g., A hard disk will have on-board memory into which the disk controller reads data, and from which the OS reads data out
  - No use if long-term average bandwidth of one component simply exceeds the other!

Bottlenecks, tuning, the 80/20 rule

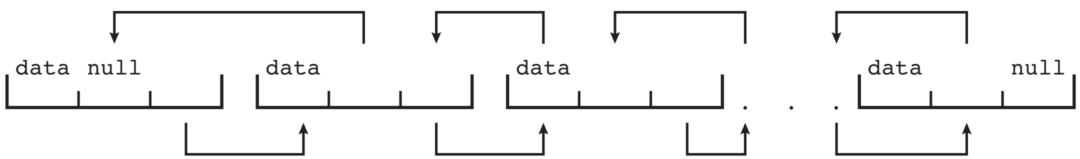
- The **bottleneck** is typically the most constrained resource in a system
- Performance optimisation and tuning focuses on determining and eliminating bottlenecks
  - Often introducing new ones in the process
- A perfectly balanced system has all resources simultaneously bottlenecked
  - Impossible to actually achieve
  - Often find that optimising the common case gets most of the benefit anyway
- Means that measurement is a prerequisite to performance tuning!
  - The 80/20 rule — 80% time spent in 20% code
  - No matter how much you optimise a very rare case, it will make no difference

Common data structures

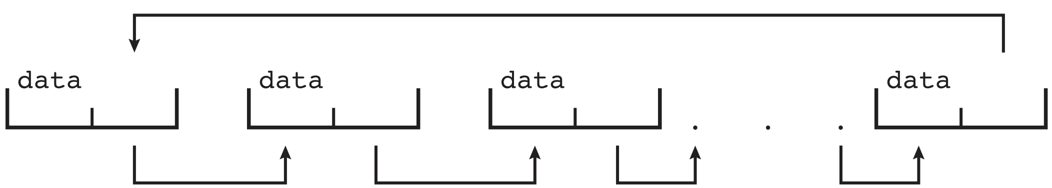
Linked list



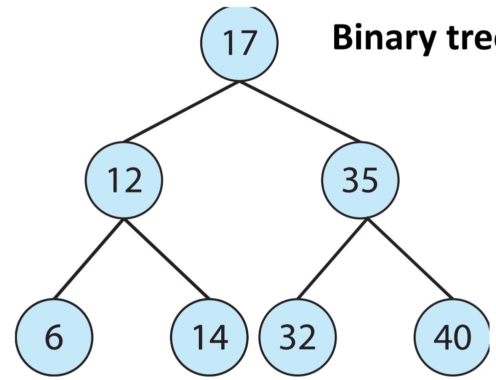
Doubly-linked list



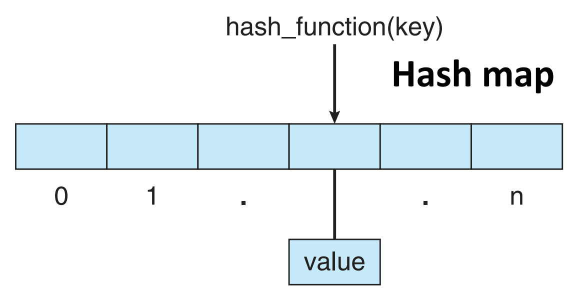
Circularly-linked list



Binary tree



Hash map



Outline

- System organisation
- System operation
- Concepts
- What is an Operating System?
  - Resource protection
  - CPU, memory, I/O

What is an Operating System?

- Just a program – a piece of software that (efficiently) provides
  - **Control**, over the execution of all other programs
  - **Multiplexing**, of resources between programs
  - **Abstraction**, over the complexity and low-level details
  - **Extensibility**, enabling evolution to meet changing demands and constraints
- Typically involves libraries and tools provided as part of the OS
  - Kernel – but also a *libc*, a language runtime, a web browser, …
  - Thus no-one really agrees precisely what the OS is
  - In this course we will focus on the **kernel**
- OS provides **mechanisms** that are used to implement **policies**
  - Policies may be deliberately designed, or accidents of implementation

Resource management

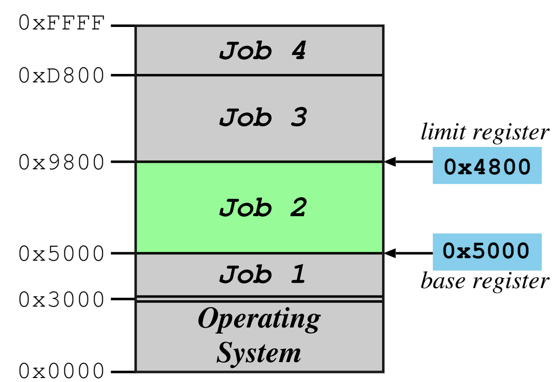
- Running program executes instructions sequentially to completion using resources
- **CPU**
  - OS multiplexes many running programs (threads) over the CPU(s)
  - Lifecycle management, synchronisation, communication
- **Memory**
  - Running programs require code and data in memory
  - Tracking memory ownership, managing de/allocation
- **Storage**
  - Abstracting different storage media and their characteristics
  - Creating, deleting, manipulating files, directories and free space
- **I/O Subsystem**
  - Abstracting peculiarities of different devices
  - Providing device drivers, managing I/O buffering, caching, spooling

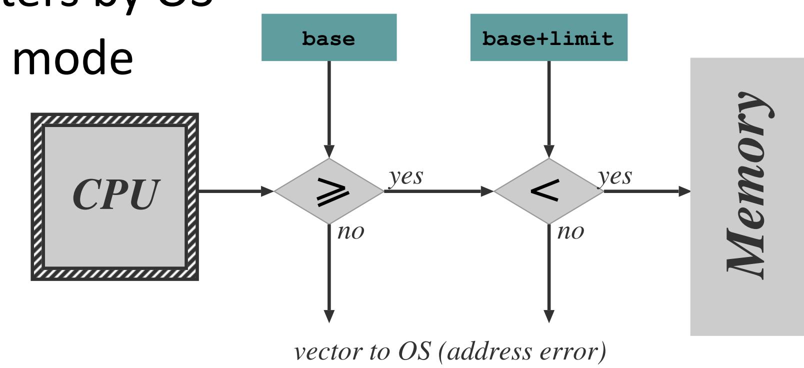
Protecting the CPU

- Need to ensure that the OS stays in control, able to prevent any application from “hogging” the CPU the whole time
- Means using a timer, usually a countdown timer, e.g.,
  - Set timer to initial value (e.g. 0xFFFF)
  - Every tick (nowadays programmable), timer decrements value
  - When value hits zero, interrupt
- Ensures the OS runs periodically provided
  - only OS can load timer, and
  - timer interrupt cannot be masked
- Also enables implementation of time-sharing

Protecting memory

- Define a base and a limit for each program, and protect access outside allowed range
- Have hardware check every memory reference:
  - Access out of range causes exception, vectored into OS
  - Only allow update of base and limit registers by OS
  - Can disable memory protection in kernel mode (but this is a bad idea)
- In reality, more complex protection hardware is used





Protecting I/O

- Initially, tried to make IO instructions privileged:
  - Applications can't mask interrupts (that is, turn one or many off)
  - Applications can't control IO devices
- Unfortunately, some devices are accessed via memory, not special instructions
  - Applications can rewrite interrupt vectors
- Hence protecting IO relies on memory protection mechanisms

Summary

- System organisation
  - Hardware resources
  - Fetch-execute cycle
  - Buses
- System operation
  - Booting
  - Interrupts
  - Storage
- Concepts
  - Layering, multiplexing
  - Latency, bandwidth, jitter
  - Caching, buffering
  - Bottlenecks, tuning, 80/20 rule
  - Data structures
- What is an Operating System?
  - Resource protection
  - CPU, memory, I/O

02. Protection

9th ed: Ch. 2.7+, 14, 15, 16

10th ed: Ch. 2.7+, 16, 17, 19

Objectives

- To describe the evolution of the operating system
- To understand how the OS protects itself from user programs
- To understand how the OS protects user programs from each other
- To know some different ways the OS can be structured
- To be aware of some security considerations

Outline

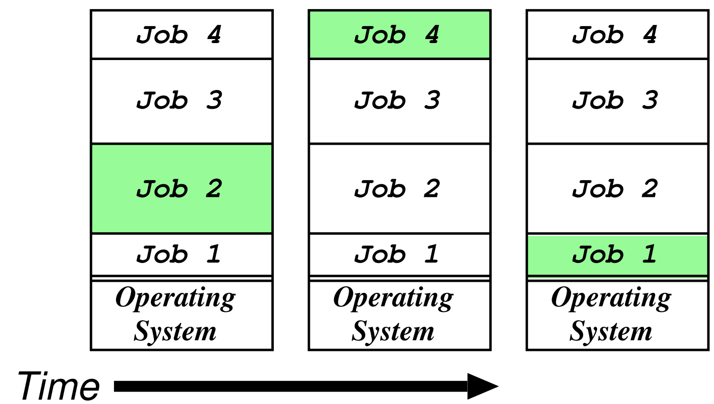
- OS evolution
- Kernels
- Security

Outline

- OS evolution
  - Single-tasking
  - Dual-mode operation
- Kernels
- Security

Operating system evolution

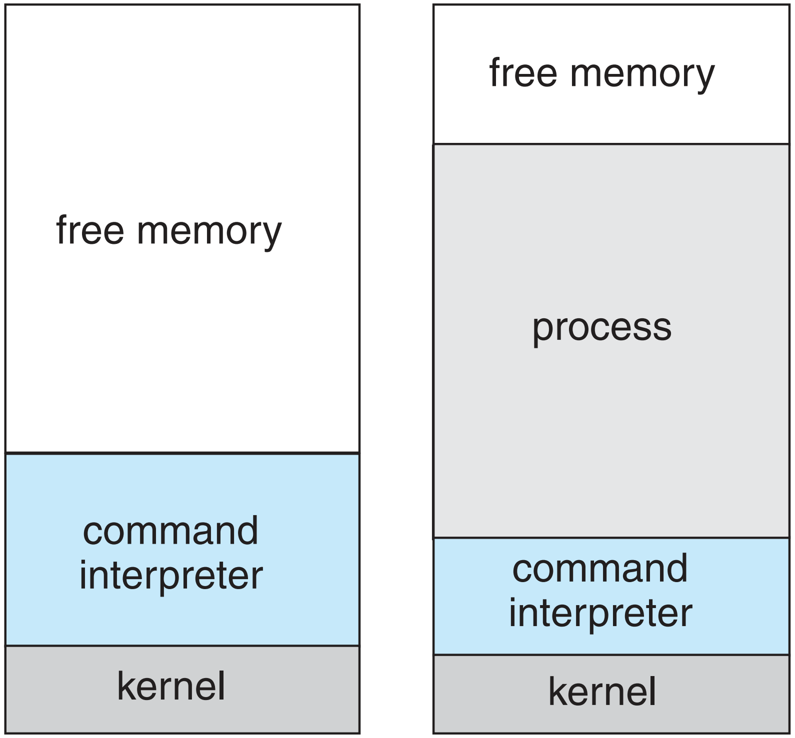
- **Open shop**: One machine, one CPU, one user, one program – the user is the programmer is the operator, all programming is in machine code
  - E.g., EDSAC, 1947—1955
- **Batch systems**: tape drives collate and run a set of programs in a batch, increasing efficiency
  - Spooling allowed overlap of I/O with computation
- **Multiprogramming**: one machine, one CPU, one running program but many loaded programs
  - **Job scheduling**: select jobs to load and then which resident job to run



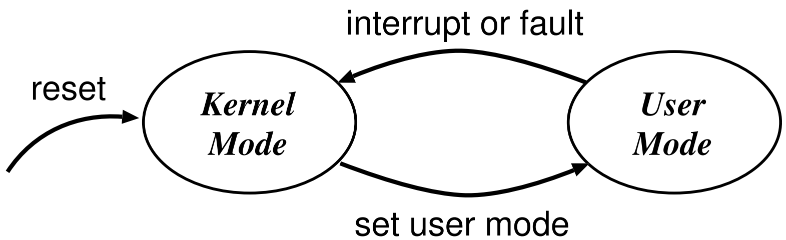
- **Timesharing**: switching jobs so frequently that users have the illusion many jobs are running simultaneously
  - **CPU scheduling**: select which job to run from many that are ready
  - Enables interactive computing

Single-tasking OS: MS-DOS

- Command interpreter receives input from user
  - Program is loaded, overwriting much of the command interpreter
  - Instruction pointer set to start of program
- Once finished, termination causes command interpreter stub to reload command interpreter
  - Exit error code available to user



Dual-mode operation



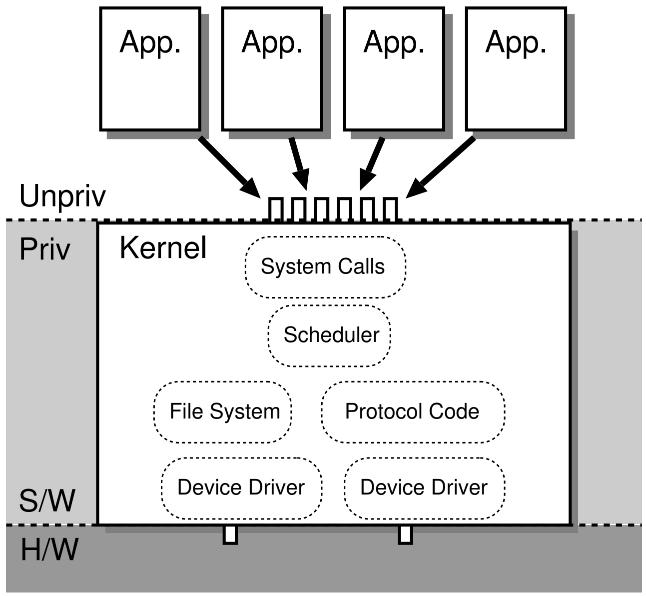
- Allows OS to stop malicious or buggy code from doing bad things
- Use hardware – a **mode bit** – to distinguish (at least) two modes of operation
  - **User mode** when executing on behalf of a user (i.e. application programs)
  - **Kernel mode** when executing on behalf of the OS
  - Some instructions designated as privileged, only executable in kernel mode
- Increasingly CPUs support multi-mode operations
  - i.e. virtual machine manager (VMM) mode for guest VMs
- Often “nested” e.g., x86 rings 0—3; further inside can do strictly more
  - Not ideal, but disjoint/overlapping permissions is complex

Outline

- OS evolution
- Kernels
  - System calls
  - Microkernels
  - Virtualisation
- Security

Kernels

- Protection prevents applications doing IO – kernel does it for them
  - Thus we need an unprivileged instruction to transition from user to kernel mode
  - Generally called a **trap** or a **software interrupt** since operates similarly to (hardware) interrupt
- OS services are accessible via **system calls**
  - Invoked by a trap with OS having vectors to handle
  - Vector enforces code run when mode switch occurs
  - Prevents application from switching to kernel mode and then just doing whatever it likes
- Alternative is for OS to emulate for application, and check every instruction before execution as used in some virtualisation systems, e.g., QEMU

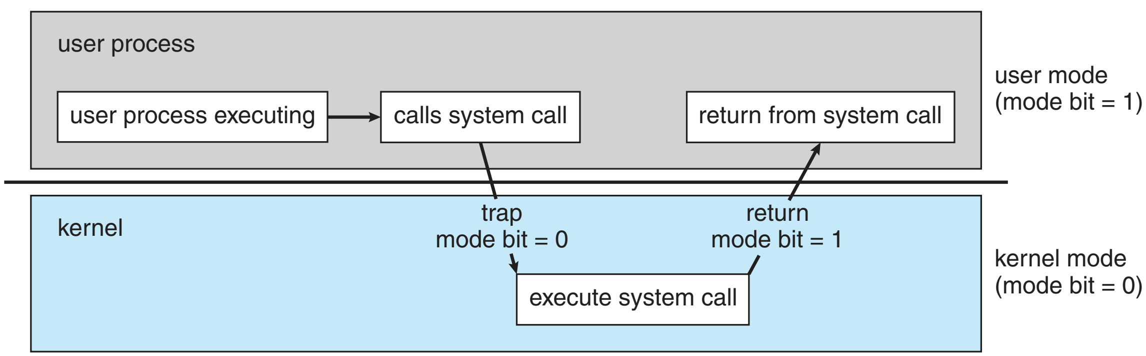


System calls

- Provide a (language agnostic) standard interface to the OS services
- Accessed via a high-level (language specific) Application Programming Interface (API) rather than called directly
  - E.g., glibc

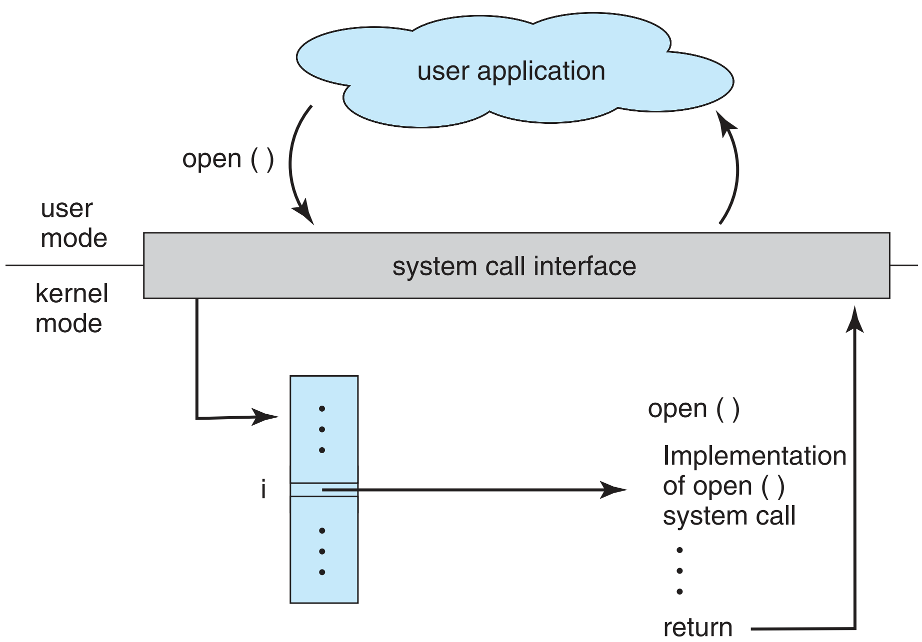
```rust
#[inline(always)]
pub unsafe fn syscall4(mut n: usize,
                       a1: usize,
                       a2: usize,
                       a3: usize,
                       a4: usize)
                       -> usize
{
    llvm_asm!("int $$0x80"
          : "+{eax}"(n)
          : "{ebx}"(a1) "{ecx}"(a2) "{edx}"(a3) "{esi}"(a4)
          : "memory" "cc"
          : "volatile");
    n
}
```

Raw system calls in Rust
https://github.com/strake/system-call.rs/



System call invocation

- Typically each system call is associated by a number that indexes a **system call table**
  - Invoked by putting the relevant number and any required parameters in the right places and trapping
  - Return status and any values made available to application in user space
- Usually managed by run-time support library, a set of functions built into libraries automatically linked by your compiler

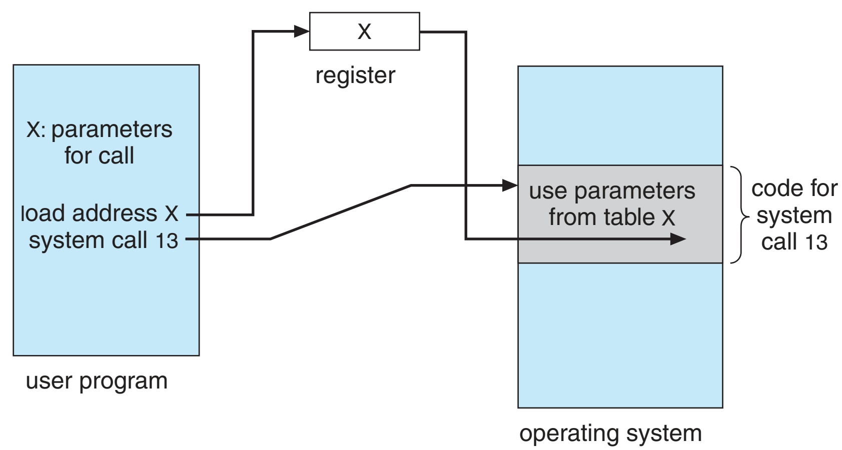


System call parameters

- Three main ways to pass parameters:
    1. Load into registers
    2. Place onto stack for the kernel to pop off
    3. Place into a block of memory and put the block's address into a register
- One of the latter two usually preferred
    - Registers limited in number and size

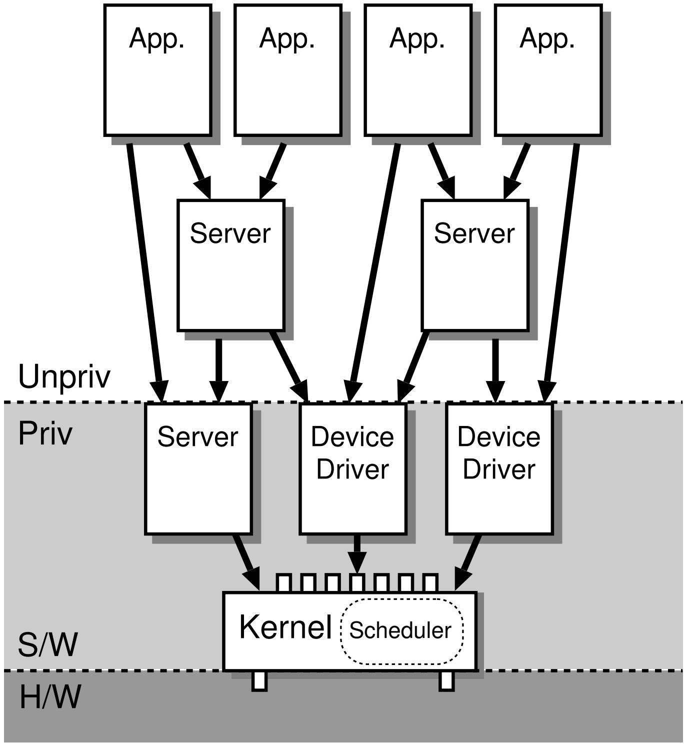
```
int
open(const char *path, int oflag, ...);

ssize_t
read(int fildes, void *buf, size_t nbyte);
```



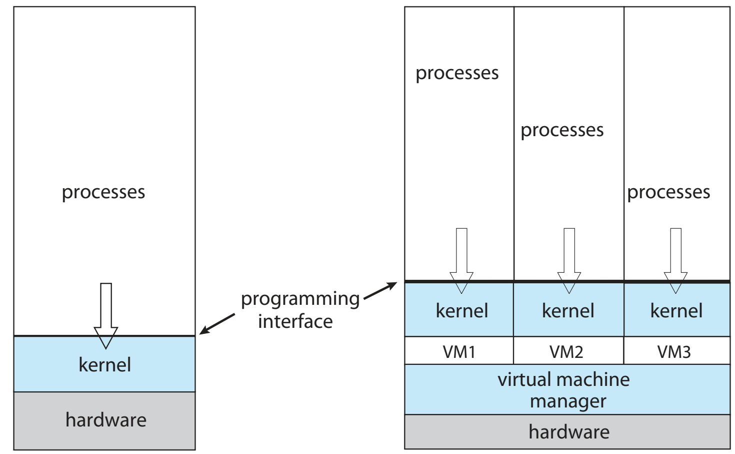
Microkernels

- OS interfaces must be extremely stable
  - Makes them difficult to extend with new calls
  - Even more difficult to remove calls
- Alternative is **microkernels**
  - Move OS services into local, sometimes privileged, servers
  - Increases modularity and extensibility
- **Message passing** used to access servers
  - Replaces trapping so must be extremely efficient
- Many common OSs blur the distinction between kernel and microkernel, e.g.,
  - Linux has kernel modules and some servers
  - Windows NT 3.5 originally a microkernel but performance concerns caused NT 4.0 to move services back into the kernel



Virtualisation

- More recently, trend towards encapsulating applications differently
  - Make the system appear to be supporting just one application
  - Particularly relevant when building systems using microservices
  - Protection, or isolation at a different level



- **Virtualisation**: allows operating systems to be run alongside each other above a **hypervisor**
  - Type 1 runs directly on the host hardware, possibly using hardware extensions (VT-x)
  - Type 2 runs above a full OS kernel
  - Can support cross-architecture using **emulation** (slow) or **interpretation** (if not natively compiled)

Virtual machines, containers

- Virtual Machines encapsulate an entire running system, including the OS, and then boot the VM over a hypervisor
  - E.g., Xen, VMWareESX, Hyper-V
- Containers expose functionality in the OS so that each container acts as a separate entity even though they all share the same underlying OS functionality
  - E.g., Linux Containers, FreeBSD Jails, Solaris Zones
- Use cases include
  - Laptops and desktops running multiple OSes for exploration or compatibility
  - Developing apps for multiple OSes without having multiple systems
  - QA testing applications without having multiple systems
  - Executing and managing compute environments within datacenters

Outline

- OS evolution
- Kernels
- Security
  - Principle of least privilege
  - Domain of protection
  - Access matrix
  - Access Control Lists (ACLs)
  - Capabilities
  - Authentication

Security

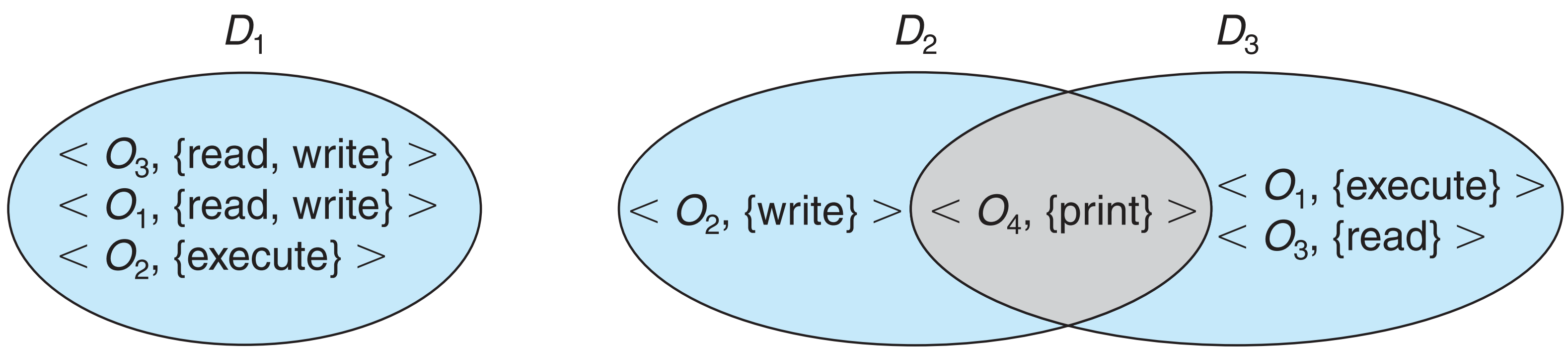
- Defence of the system against internal and external attacks
  - Huge range of attacks, including denial-of-service, worms, viruses, identity theft, theft of service
- Systems generally first distinguish among users, to determine who can do what
  - User identities (user IDs, security IDs) include name and associated number, one per user
  - User ID then associated with all files, processes of that user to determine access control
  - Group identifier (group ID) allows set of users to be defined and controls managed, then also associated with each process, file
- **Privilege escalation** allows user to change to effective ID with more rights

Principle of least privilege

- Objects should be given just enough privileges to perform their tasks
  - **Hardware objects** (e.g., devices) and **software objects** (e.g., files, programs, semaphores)
- Properly set permissions can limit damage if object has a bug and gets abused
  - Can be **static** (during life of system, during life of process)
  - Or **dynamic** (changed by process as needed) by domain switching, privilege escalation
- Compartmentalization a derivative concept regarding access to data
  - Process of protecting each individual system component through the use of specific permissions and access restrictions
  - More granular, more complex, more protective
- **Covert channels** leak information using side-effects
  - Hardware include wire tapping or receiving electromagnetic radiation from devices
  - Software include page fault statistics or input-dependent timing
  - E.g., lowest layer of recent OCaml TLS library had to be written in C to avoid the garbage collector becoming a covert channel

Domain of protection

- Domain limits access to (and operations on) objects
  - *access-right* = < *object-name, rights-set* > where *rights-set* is a subset of all valid operations that can be performed on *object-name*
  - A domain is then a set of *access-rights*
  - In UNIX a domain is a user id



Access matrix

- A matrix of **domains** (**subjects**, **principals**) against **objects**
  - Rows represent domains, columns represent objects
  - $M_{i,j}$ = operations a process in domain i can invoke on object j
  - Operations can include adding/deleting entries in matrix
- Example of separation of policy from mechanism

domain \ object	F_1	F_2	F_3	laser printer	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	switch control
D_3		read	execute					
D_4	write		write		switch			

Implementing the access matrix

- The access matrix is a table of triples < *domain, object, rights-set* >
  - For a domain to invoke an operation on an object involves searching to see if that operation is in any *rights-set* for the pair < *domain*, *object* >
- Table is large so may not fit in memory – but sparse
- Two common representations
  1. By **object**, storing list of subjects and rights with each object – **Access Control List** (**ACL**)
  2. By **subject**, storing list of objects and rights with each subject – **Capabilities**

Access Control Lists (ACLs)

- Each column is an access list for one object
  - Results in a per-object ordered list of < *domain*, *rights-set* >
- Often used in storage systems
  - System naming scheme provides for ACL to be inserted in naming path, e.g., files
- If ACLs stored on disk, check is in software so use only on low duty cycle – for higher duty cycle must cache results of check
  - E.g., ACL checked when file opened for read or write, or when code file is to be executed
- In (e.g.) UNIX, access control is by program, allowing arbitrary policies

Capabilities

- Each row is a capability for one domain
  - Indicates operations permitted on a set of objects
- To execute operation M on object O_j, process requests operation and passes capability as parameter
  - Possession of capability means access is allowed
  - Capability is a protected object, maintained by the OS and unmodifiable by the application – like a “secure pointer”
- Hardware capabilities, e.g., CHERI
  - Have special machine instructions to modify (restrict) capabilities
  - Support passing of capabilities on procedure (program) call
- Software capabilities
  - Protected by encryption
  - Nice for distributed systems

Authentication

- User to system: required as protection systems depend on user ID
  - Typically established through use of *password* (or passphrase or key)
  - Need to be managed, kept secure
  - Hashed with a salt (easy to compute, hard to invert)
  - Multi-factor authentication adds a second (or more) component
  - Failed access attempts usually logged
- System to user: avoid user talking to the wrong computer / program
  - In the old days with directly wired terminals, make login character same as terminal attention, or always do a terminal attention before trying login
  - E.g., Windows NT's Ctrl-Alt-Del to login — no-one else can trap it
  - (When your bank phones, how do you know it's them?)

Summary

- OS evolution
  - Single-tasking
  - Dual-mode operation
- Kernels
  - System calls
  - Microkernels
  - Virtualisation
- Security
  - Principle of least privilege
  - Domain of protection
  - Access matrix
  - Access Control Lists (ACLs)
  - Capabilities
  - Authentication

03. Processes

Ch. 1.6, 3

Objectives

- To understand the concept of a process vs a program, and the need for context switching
- To distinguish the states in a process' lifecycle
- To know some of the state required for process management

Outline

- What is a process?
- Process lifecycle
- Inter-Process Communication (IPC)

Outline

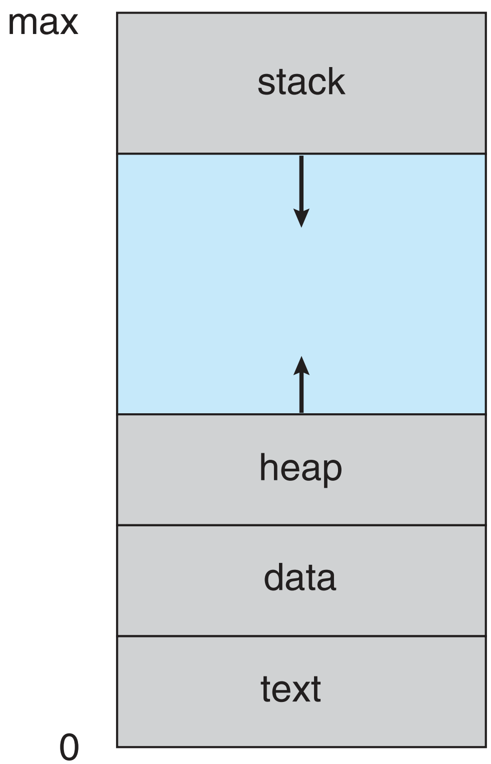
- What is a process?
  - Process Control Block (PCB)
  - Threads of execution
  - Context switching
- Process lifecycle
- Inter-Process Communication (IPC)

What is a process?

- The computer is there to execute programs, not the OS!
- Process ≠ Program
  - A **program** is static, on-disk
  - A **process** is dynamic, a program in execution
  - On a batch system, might refer to jobs instead of processes – nowadays generally used interchangeably
- Process is the **unit of protection** and **resource allocation**
  - So you may have multiple processes running created from a single program

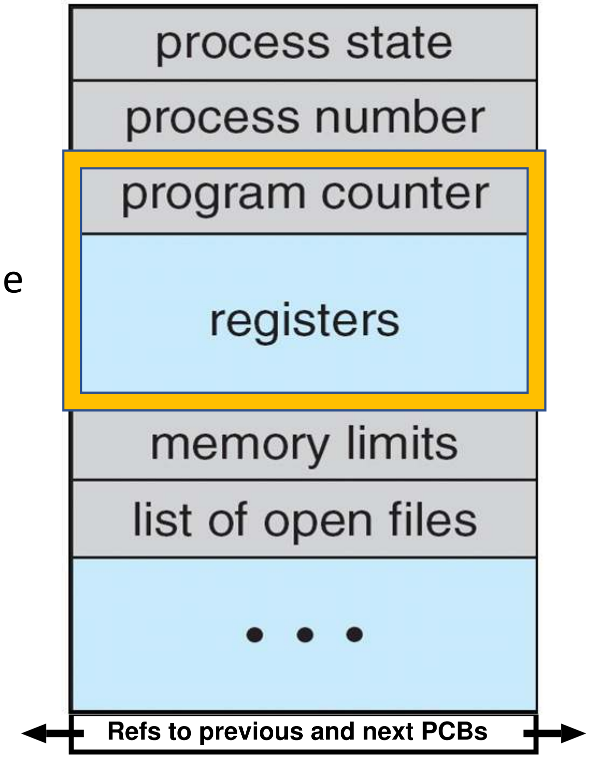
What is a process?

- Each process executed on a virtual processor has
  - **Text** containing the program code
  - **Data** containing global variables
  - **Heap** containing memory allocating during runtime
  - …plus one or more **threads of execution**
- Each thread has
  - **Program counter** indicating current instruction
  - **Stack** for temporary variables, parameters, return addresses, etc.



Process Control Block (PCB)

- Data structure representing a process, containing
  - **Process ID** or **number** – uniquely identifies the process
  - **Current process state** – running, waiting, etc
  - **CPU scheduling information** – priorities, scheduling queue pointers
  - **Memory-management information** – memory allocated to the process
  - **Accounting information** – CPU used, clock time elapsed since start, time limits
  - **I/O status information** – I/O devices allocated to process, list of open files
- Highlighted **process context** is the machine environment while the process is running
  - **Program counter**, location of instruction to next execute
  - **CPU registers**, contents of all process-centric registers

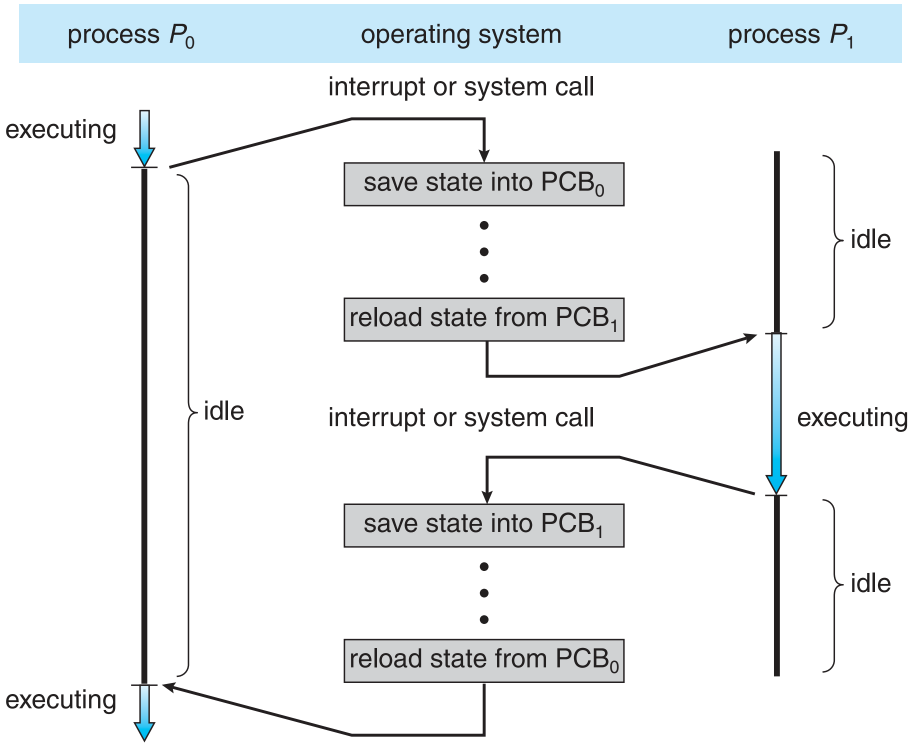


Threads of execution

- A **thread** represents an individual execution context
  - One process may have many threads
  - OS visible threads are **kernel threads**, whether executing in kernel or user space
- Each thread has an associated **Thread Control Block** (TCB)
  - Contains thread metadata: saved context (registers, including stack pointer), scheduler info, program counter, etc.
- A scheduler determines which thread to run
  - Changing the running thread involves a **context switch**
  - If between threads in different processes, the process state also switches

Context switching

- Switching between processes means
  - Saving the context of the currently executing process (if any), and
  - Restoring the context of that being resumed
- Wasted time! No useful work is carried out while switching
- How much time depends on hardware support
  - From nothing, to
  - Save/load multiple registers to/from memory, to
  - Complete hardware "task switch"

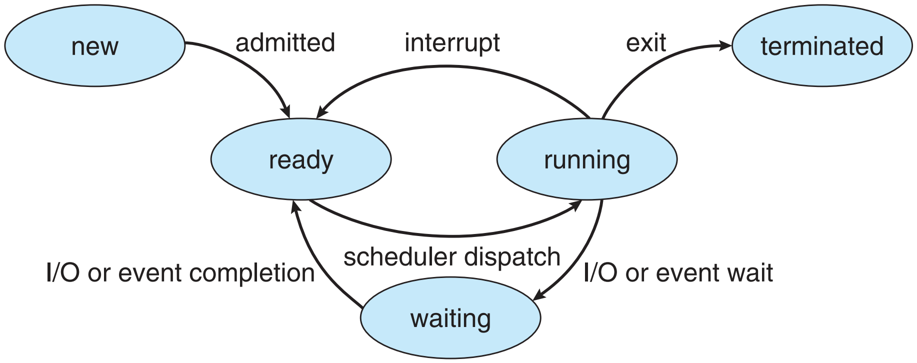


Outline

- What is a process?
- Process lifecycle
  - Process states
  - Process creation
  - Process termination
- Inter-Process Communication (IPC)

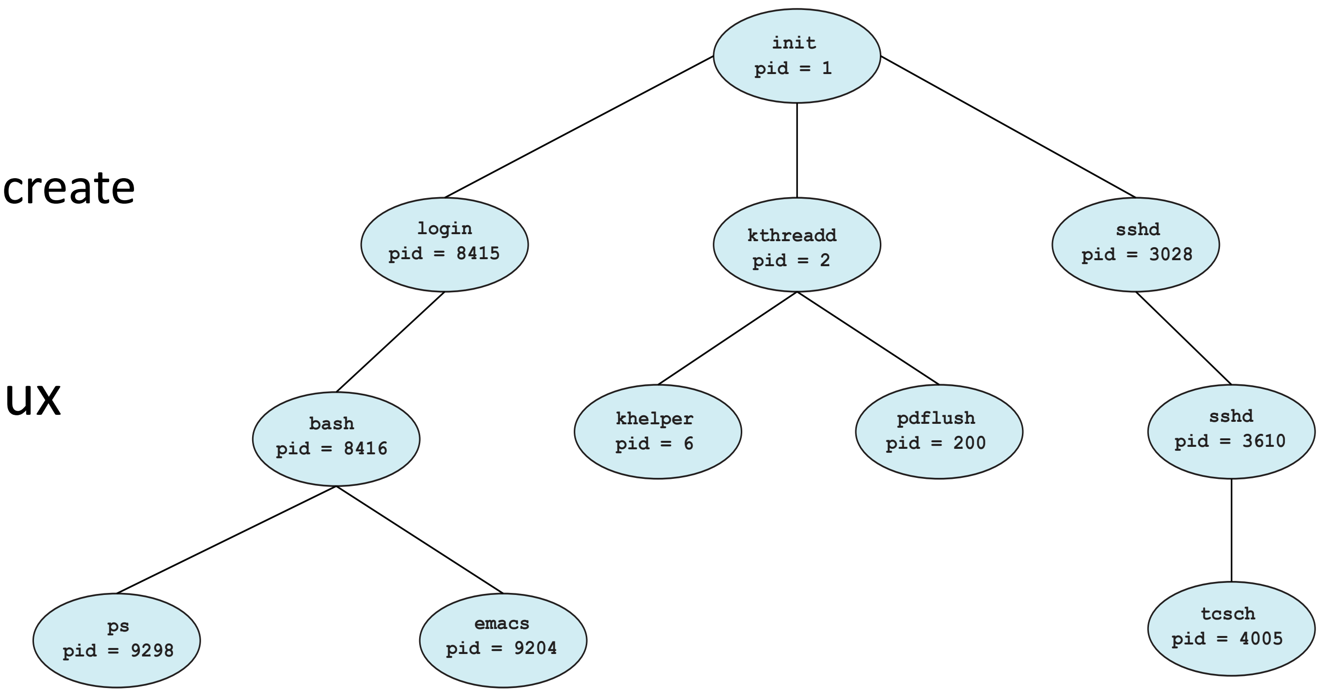
Process states

- **New**: process is being created
- **Running**: process instructions are being executed on the CPU
- **Ready**: process is ready to run, and is waiting for the CPU
- **Waiting** (**Blocked**): process has stopped executing, and is waiting for an event to occur
- **Terminated** (**Exit**): process has finished executing



Process creation

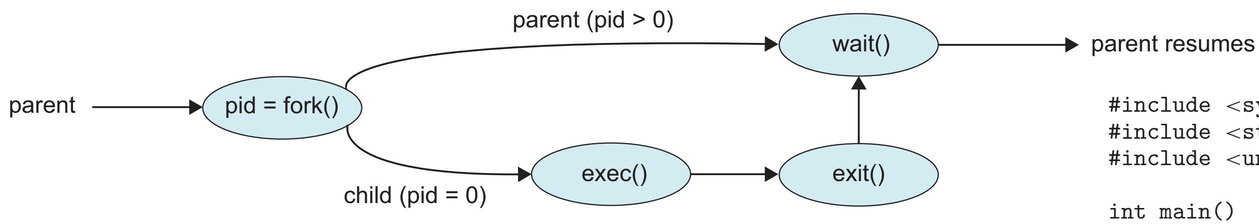
- Most systems are hierarchical
  - Parent processes create child processes
  - Forms a *tree*
- E.g., a possible Linux process tree



Process creation

- How are resources shared?
  1. Parent and children share all resources
  2. Children share subset of parent's resources
  3. Parent and child share no resources
- How is the child's memory initialised?
  1. Child starts with a duplicate of the parent and then modifies it
  2. Child explicitly has a program loaded into it
- How is execution of parent and children handled?
  1. Parent and children execute concurrently
  2. Parent waits until children terminate

Process creation




```
#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;
}
```

- E.g., on Unix
  - *fork* clones a child process from parent,
  - then *execve* replaces child's memory space with a new program,
  - meanwhile parent *waits* until child *exits*
- Alternative approach in NT/2K/XP
  - *CreateProcess* explicitly includes name of program to be executed

Process termination

1. Process performs an illegal operation, e.g.,
   - Makes an attempt to access memory without authorisation
   - Attempts to execute a privileged instruction
2. Parent terminates child (*abort*, *kill*), e.g. because
   - Child has exceeded allocated resources
   - Task assigned to child is no longer required
   - **Cascading termination** – parent is exiting and OS requires children must also exit
3. Process executes last statement and asks the OS to delete it (*exit*)
   - Parent *waits* and obtains status data from child
   - If parent didn't wait, process is a **zombie**
   - If parent terminated without waiting, process is an **orphan**

Outline

- What is a process?
- Process lifecycle
- Inter-Process Communication (IPC)
  - Message passing vs shared memory
  - Signals
  - Pipes
  - Shared memory segments

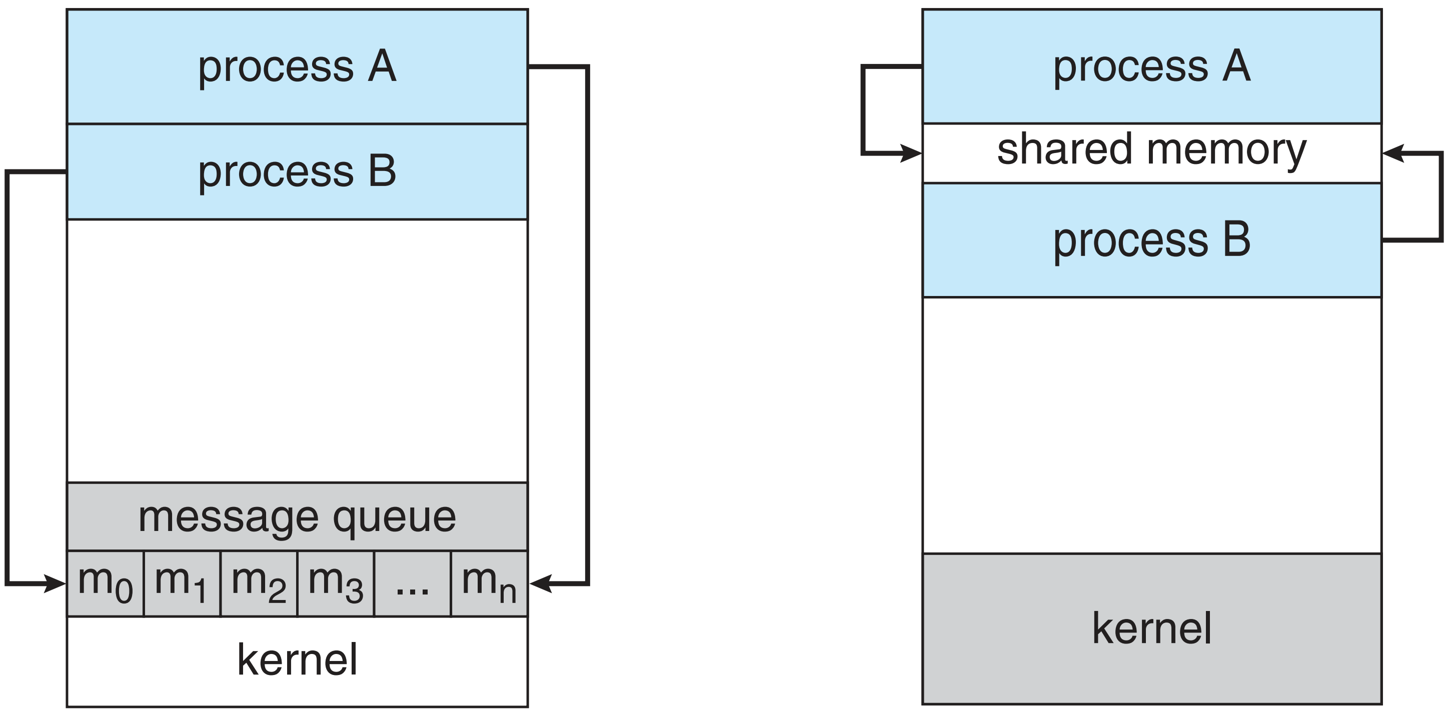
Inter-Process Communication (IPC)

- All communications require some protocol, with data transfer
  - …in a commonly-understood format (**syntax**)
  - …having mutually-agreed meaning (**semantics**)
  - …taking place according to agree rules (**synchronisation**)
  - (Ignore problems of discovery, identification, errors, etc. for now)
- Communication between hosts is IB Computer Networking
  - Separate hosts means handling reliability and asynchrony
- Communication between threads is IB Concurrent & Distributed Systems
  - Shared data structures allows corruption, deadlock, etc.
- IPC basic requirement: access to shared memory on same host

Message passing vs Shared memory

- Two fundamental models for IPC
- **Shared memory**
  - Communicating processes establish some part of memory both can access
  - Requires removing usual restriction that processes have memory protection
- **Message passing**
  - Processes send messages to each other mediated by the kernel
  - Requires support for processes to
    - name each other or a shared mailbox (direct vs indirect communication)
    - send and receive synchronously or asynchronously (blocking vs non-blocking)
    - buffer messages to match rates if non-blocking (zero, finite, unbounded buffers)

Message passing vs Shared memory

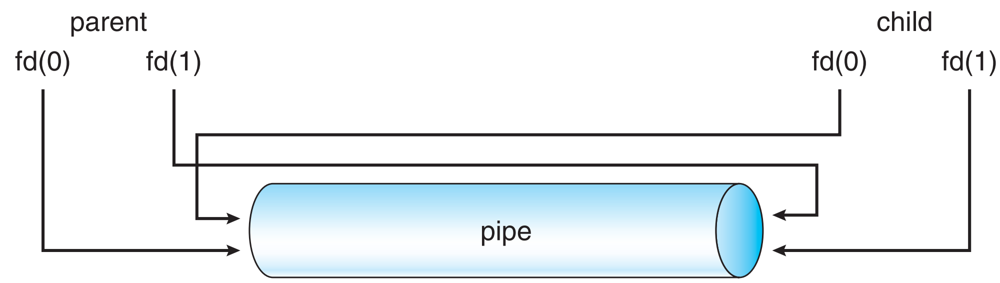


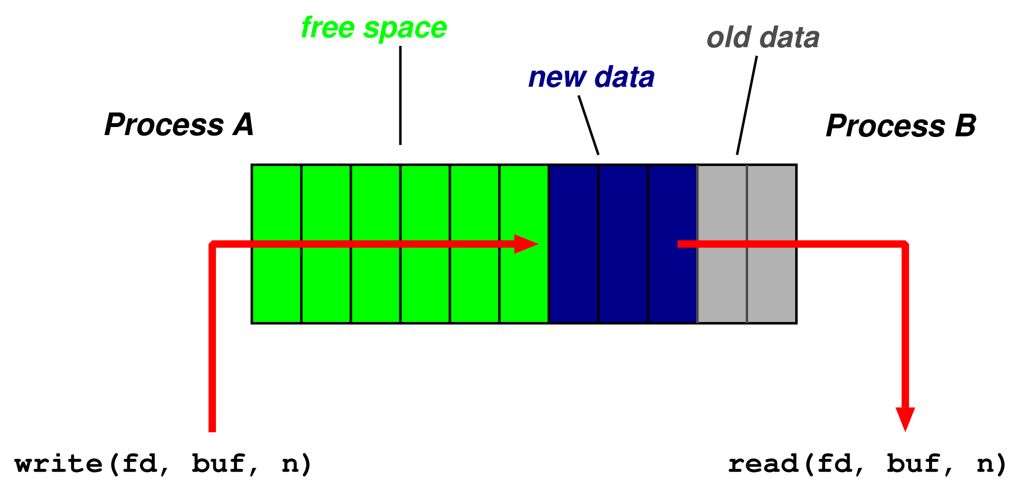
Signals

- Simple message passing: asynchronous notifications on another process
  - *kill* system call sends a signal to a specified process/es
  - *sigaction* examines or changes a **signal handler** disposition (terminate, ignore, etc)
  - *pause* suspends process until signal is caught
- Each signal mapped to an integer, different between architectures
  - https://www.man7.org/linux/man-pages/man7/signal.7.html
- Among the more commonly encountered:
  - SIGHUP: hangup detected on terminal / death of controlling process (1)
  - SIGINT: terminal interrupt (2)
  - SIGILL: illegal instruction (4)
  - SIGKILL: terminate the process [cannot be caught or ignored] (9)
  - SIGTERM: politely terminate process (15)
  - SIGSEGV: segmentation fault — process made an invalid memory reference
  - SIGUSR1/2: two user defined signals [system defined numbers]

Pipes

- Simple form of shared memory IPC
  - *pipe* returns a pair of file descriptors, (fd[0], fd[1])
  - *fork* creates child process
- Parent and child can now communicate
  - *read*/*write* on the pair of (read, write) fds
- **Named pipes** (**FIFOs**) extend beyond parent/child relation
  - Appear as files in the filesystem





Shared memory segments

- Obtain a segment of memory shared between two (or more) processes
  - *shmget* to get a segment
  - *shmat* to attach to it
- Simply read and write via pointers into the shared memory segment
  - Need to impose controls to avoid collisions when simultaneously reading and writing
- When finished,
  - *shmdt* to detach and
  - *shmctl* to destroy once you know no-one still using it

Summary

- What is a process?
  - Process Control Block (PCB)
  - Threads of execution
  - Context switching
- Process lifecycle
  - Process states
  - Process creation
  - Process termination
- Inter-Process Communication (IPC)
  - Message passing vs shared memory
  - Signals
  - Pipes
  - Shared memory segments

04. Scheduling

9th ed: Ch. 6

10th ed: Ch. 5

Objectives

- To introduce CPU scheduling, the basis for multi-programmed operating systems, and the CPU I/O burst cycle
- To distinguish pre-emptive and non-preemptive scheduling
- To understand some different metrics used to make scheduling decisions
  - Utilisation, Throughput
  - Turnaround time, Waiting time, Response time

Outline

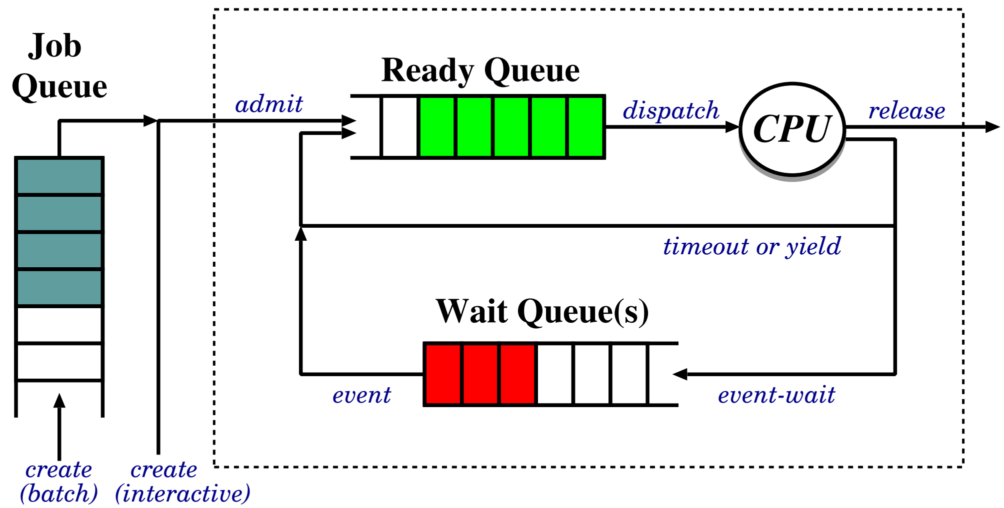
- Queues
- Scheduling
- Multiple processor scheduling

Outline

- Queues
  - CPU I/O burst cycle
  - CPU scheduler vs job scheduler
  - Idling
- Scheduling
- Multiple processor scheduling

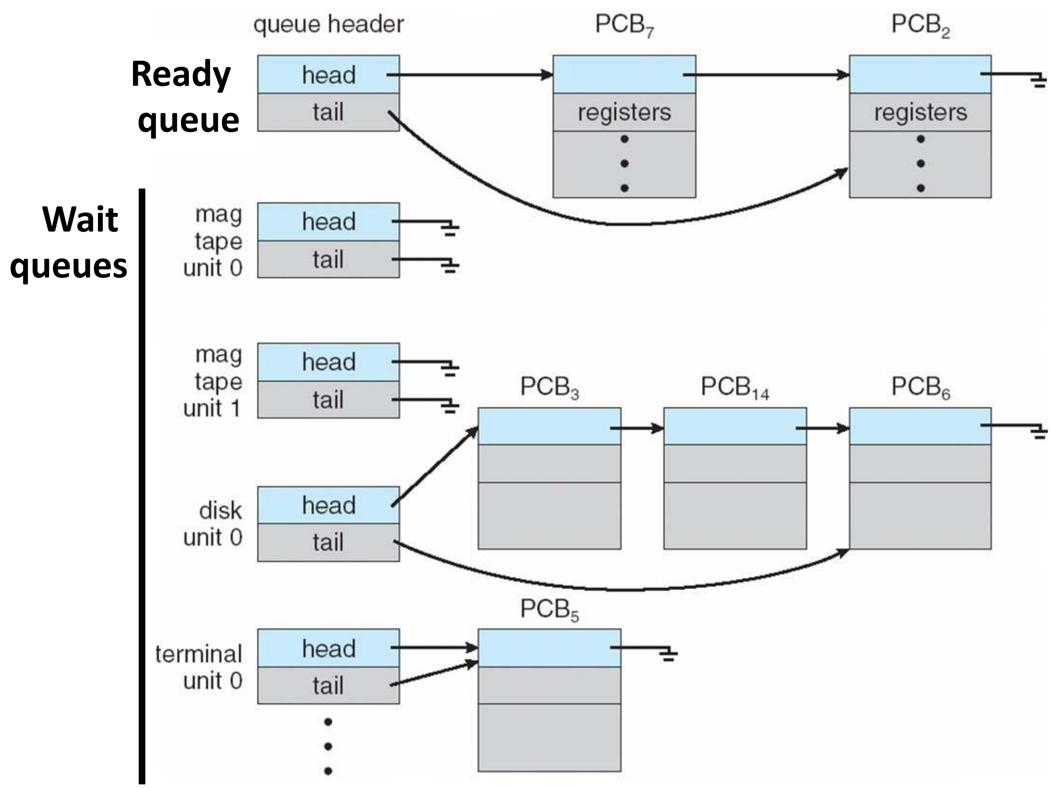
Queues

- **Job Queue**: batch processes awaiting admission
- **Ready Queue**: processes in main memory, ready and waiting to execute
- **Wait Queue(s)**: set of processes waiting for e.g., I/O devices or other processes

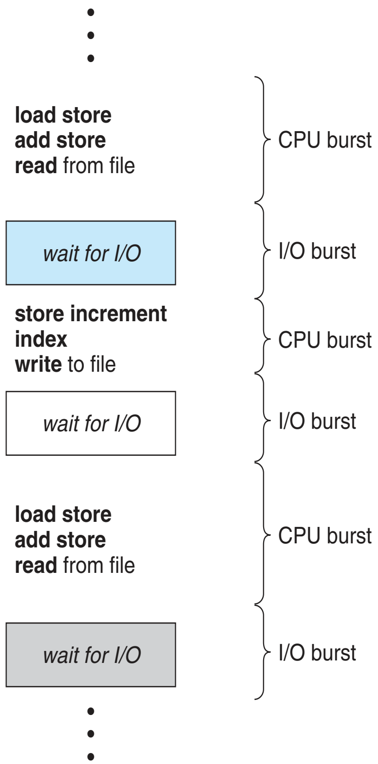


Queues

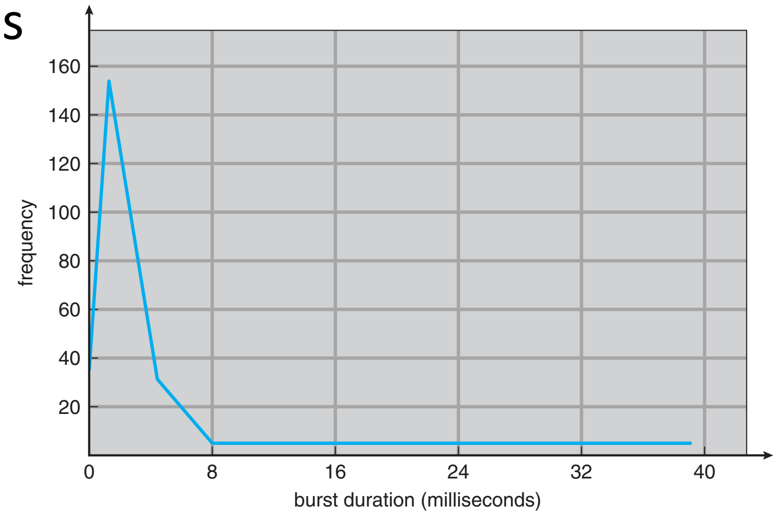
- For example,
  - Two processes (7, 2) in the Ready queue
  - No processes waiting for either magnetic tape unit
  - Three processes (3, 14, 6) waiting for the disk
  - One process (5) waiting for the terminal
- ...etc



CPU I/O Burst Cycle



- Process execution interleaves CPU execution with waiting for I/O
- Maximising CPU utilization means **multiprogramming**
  - Need something to do while waiting for I/O
- CPU burst distribution helps parameterise scheduling
  - Often (*hyper-*)*exponential*
- **I/O-bound**
  - Many short CPU bursts
- **CPU-bound**
  - Fewer long CPU bursts



Schedulers

- Short-term or **CPU scheduler**
  - Selects which process should be executed next and allocates it to the CPU
  - Sometimes the only scheduler in a system
  - Invoked frequently (milliseconds) so must be fast
- Long-term or **Job scheduler**
  - Controls the degree of multiprogramming
  - Selects which processes should be brought into the ready queue
  - Invoked infrequently (seconds, minutes) so may be slow
  - Strives for good process mix between CPU- and I/O-bound processes

Idling

- Will assume there's always something to do – but what if there isn't?
  - An important question on a modern (interactive) machine
- Three options:
  1. Busy wait in the scheduler: short-response times but ugly, inefficient
  2. Halt CPU until interrupted: saves energy but increases latency
  3. Invent an **idle process**:
     - nice uniform structure and could do some housekeeping
     - …but consumes resources and might slow interrupt response

Outline

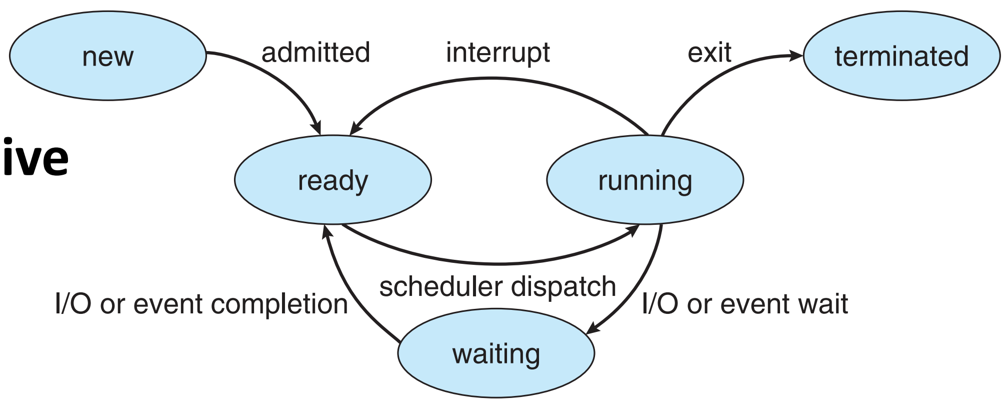
- Queues
- Scheduling
  - Dispatcher
  - Pre-emptive vs non-preemptive
  - Criteria
- Multiple processor scheduling

Dispatcher

- After scheduler, the **Dispatcher** gives control of the CPU to the selected process by
  - Switching context,
  - Switching to user mode,
  - Executing the user process from the selected location
- **Dispatch latency** is the time it takes to complete this stop/start procedure
- Two important questions:
  1. When to make a scheduling decision to select the next process?
  2. How to order the queue – which process to select next?

When to enter the scheduler?

- When can the scheduling decision be made? When
  1. ...a running process blocks (*running* → *waiting*)
  2. ...a running process terminates (*running* → *terminated*)
  3. ...a timer expires (*running* → *ready*)
  4. ...a waiting process unblocks (*waiting* → *ready*)



- If the scheduler is only invoked under 1 and 2, it is **non-preemptive**
  - Running process decides if/when to enter scheduler
- Otherwise, it is **pre-emptive**
  - OS can force scheduler entry

Pre-emptive vs Non-preemptive

- **Non-preemptive** scheduling
  - Typically uses an explicit *yield* system call or similar so running process can enter the scheduler, alongside implicit yields when, e.g., performing IO
  - Simple to implement: no timers required, process holds CPU as long as desired
  - Open to denial-of-service: malicious or buggy process can refuse to yield
- **Pre-emptive** scheduling
  - Hardware support for regular timer interrupts required to ensure scheduler entered
  - Precludes denial-of-service: the OS simply pre-empts a long-running process
  - More complex to implement: Timer management, concurrency issues
- Almost all modern schedulers are **pre-emptive**

Scheduling Criteria

- Typically there will be more than one process *runnable* – how to decide which one to pick?
- Many different metrics may be used, with different trade-offs and leading to different operating regimes
- Data structures introduce time and space overheads
  - …of measurement and computation for the metric
  - …of selecting the "best" next process

Scheduling Criteria

- **Turnaround time**, minimising the time for any process to complete
  - Aims to minimise total time from process submission to completion across all states
- **Waiting time**, minimising the time a process sits in the Ready queue
  - Scheduler only controls time in the Ready queue – rest is up to the process
  - But may penalise IO heavy processes that spend a long time in the wait queue
- **Response time**, minimising the time to *start* responding
  - In interactive/time-sharing systems, users may prefer to total efficiency
  - But may penalise longer running sessions under heavy load

Scheduling Criteria

- **CPU utilisation**, maximising the time the CPU is actively in use
  - Aims to keep the (expensive) CPU as busy as possible
  - But may penalise I/O heavy processes as they appear to leave the CPU idle
- **Throughput**, maximising the rate at which processes complete execution
  - Aims to get useful work done at the highest possible rate
  - But may penalise long-running processes as short-run processes will be preferred
- Typically want to maximise utilisation and throughput, and minimise turnaround, waiting and response times
  - …but what exactly – optimise the average? Minimise the maximum?
  - What about the distribution, e.g., variance, confidence intervals?

Outline

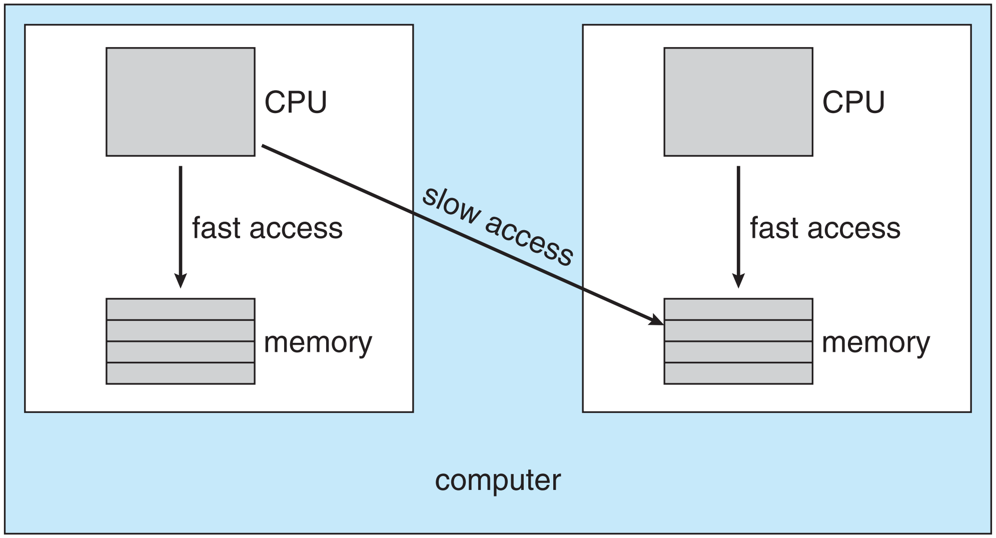
- Queues
- Scheduling
- Multiple processor scheduling
  - NUMA
  - Load balancing, multicore, virtualisation

Multiple processor scheduling

- Everything becomes more complex when multiple CPUs are available
  - Assume homogeneous processors within a multiprocessor
- Asymmetric multiprocessing
  - Only one processor accesses the system data structures
  - Alleviates the need for data sharing
- Symmetric multiprocessing (SMP) – currently the most common
  - Each processor is self-scheduling
  - All processes can be in a single ready queue, or each processor has its own private ready queue
- Processor affinity when a process has affinity for which processor it runs
  - Soft affinity indicates preference
  - Hard affinity indicates constraint
  - Variations including processor sets

Non-Uniform Memory Access (NUMA)

- Affects CPU scheduling as it means different CPUs have faster or slower access to parts of memory
  - E.g., because have combined CPU and memory boards
- Memory placement then affects affinity
- Costs of switching to a different CPU could be very much higher than without NUMA



Load balancing, multicore, virtualisation

- SMP means OS needs to keep all CPUs loaded for efficiency
- **Load balancing** attempts to keep workload evenly distributed
  - **Push migration** has a periodic task check load on each CPU and push tasks off overloaded CPUs onto other CPUs
  - **Pull migration** has idle CPUs pull waiting tasks off busy CPUs
- Recent trends include
  - **Multicore**, placing multiple CPU cores on same physical chip, increasing speed and efficiency
  - **Hyperthreading**, increasing the number of threads per core so that one thread can make progress while another is stalled on memory read
  - **Virtualisation** challenges OS scheduler as hypervisor and guests are all scheduling against each other

Summary

- Queues
  - CPU I/O burst cycle
  - CPU scheduler vs job scheduler
  - Idling
- Scheduling
  - Dispatcher
  - Pre-emptive vs non-preemptive
  - Criteria
- Multiple processor scheduling
  - NUMA
  - Load balancing, multicore, virtualisation

05. Scheduling Algorithms

9th ed: Ch. 6

10th ed: Ch. 5

Objectives

- To understand how to apply several common scheduling algorithms
  - FCFS, SJF, SRTF
  - Priority
  - Round Robin
  - Multilevel Queues
- To understand use of measurement and prediction for unknown scheduling parameters

Outline

- First-Come First-Served (FCFS)
- Shortest Job First (SJF)
- Shortest Remaining Time First (SRTF)
- Priority scheduling
- Round Robin (RR)

Outline

- First-Come First-Served (FCFS)
  - Convoy effect
- Shortest Job First (SJF)
- Shortest Remaining Time First (SRTF)
- Priority scheduling
- Round Robin (RR)

First-Come First-Served (FCFS)

- Schedule depends purely on the order in which processes arrive
- Simplest possible scheduling algorithm
- Not terribly robust to different **arrival processes**

- E.g., suppose processes with the following burst times arrive in the order P_1, P_2, P_3

Process	Burst Time
P_1	24
P_2	3
P_3	3

First-Come First-Served (FCFS)

- Then the Gantt chart is

- The waiting times are

Process	Burst Time	Waiting Time
P_1	24	0
P_2	3	24
P_3	3	27

- This gives an average per-process waiting time of $\frac{0+24+27}{3} = 17$

The Convoy Effect

- Now suppose the same processes arrive in the order P_2, P_3, P_1
- Then the Gantt chart and waiting times are:

Process	Burst Time	Waiting Time
P_1	24	6
P_2	3	0
P_3	3	3

- Gives an average per-process waiting time of $(6 + 0 + 3)/3 = 3$
- First case is an example of the **Convoy Effect**
  - Short-run processes getting stuck behind long-run processes
  - Consider one CPU-bound and many IO-bound processes

Outline

- First-Come First-Served (FCFS)
- Shortest Job First (SJF)
- Shortest Remaining Time First (SRTF)
- Priority scheduling
- Round Robin (RR)

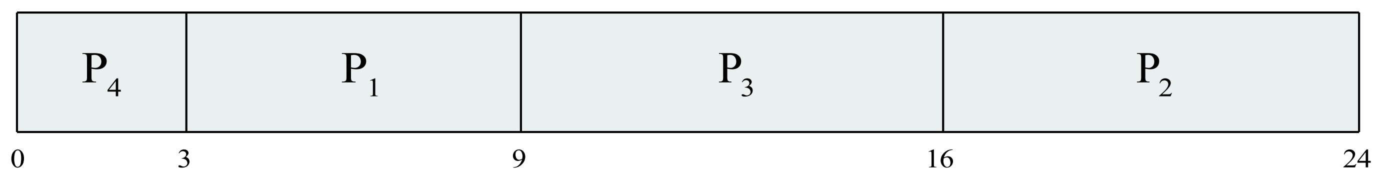
Shortest Job First (SJF)

- Associate length of next CPU burst with each process
- Schedule the process with the shortest next burst
- Optimality: SJF gives the least possible waiting time for a given set of processes
- But how can you know the length of the **next** CPU burst?
  - Ask the user?
  - Ask the developer?
  - Measure and predict?

Shortest Job First (SJF)

- Consider the following arrivals process and resulting Gantt chart:

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3



- Gives an average per-process waiting time of $\frac{3+16+9+0}{4} = 7$

Outline

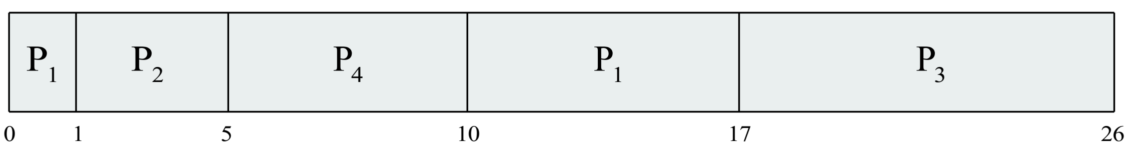
- First-Come First-Served (FCFS)
- Shortest Job First (SJF)
- Shortest Remaining Time First (SRTF)
  - Predicting the future
  - Exponential averaging
- Priority scheduling
- Round Robin (RR)

Shortest Remaining Time First (SRTF)

- Simply a pre-emptive version of SJF
  - Pre-empt current process if a new one arrives with a shorter predicted burst length than the remaining time of the current process
- Distinguish **arrival time** and **burst length**, e.g.,

Process	Arrival Time	Burst Length
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

- Gives Gantt chart



- Average waiting time is now $\frac{(10-1)+(1-1)+(17-2)+(5-3)}{4} = \frac{26}{4} = 6\,{}^1/_2$

Optimality in the future

- If SJF is optimal given a known set of processes (**demand**), then surely SRTF is optimal in the face of new runnable processes arriving?
- No! Why?
- Context switches are not free, so if short burst processes keep arriving the OS will start thrashing the CPU, so no useful work gets done

- More fundamentally,

how can we know the length of a **future** burst?

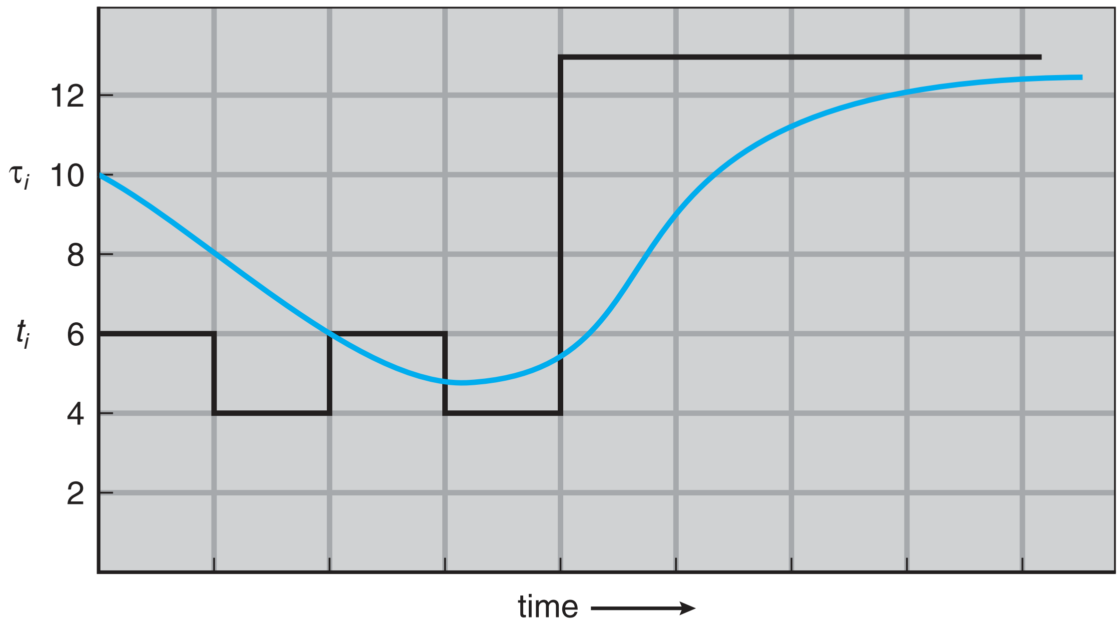
Predicting burst lengths

- Assume the next burst will not be too different from the previous
- Then
  - measure burst lengths as processes are scheduled,
  - predict next burst length, and
  - choose the process with the shortest predicted burst length
- E.g., exponential averaging on length of previous bursts
  - Set t_n to be the measured length of the n^{th} CPU burst
  - Define τ_{n+1}, predicted length of $n+1^{\text{th}}$ burst as $\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$

Examples of exponential averaging

- Expanding this formula gives
  $$\tau_{n+1} = \alpha t_n + \ \dots + (1-\alpha)^j \alpha t_{n-j} + \ \dots + (1-\alpha)^{n+1}\tau_0$$
  where τ_0 is some constant
- As both $\alpha, 1-\alpha \le 1$, each term has less weight than its predecessor
- Choose value of α according to our belief about the system, e.g,
  - If we believe past history irrelevant, choose $\alpha \approx 1$ and then get $\tau_{n+1} \approx t_n$
  - If we believe recent history irrelevant, choose $\alpha \approx 0$ and then get $\tau_{n+1} \approx \tau_n$
- Exponential averaging is often a good predictor if the variance is small
  - NB. Also should consider load, else (counter-intuitively) priorities increase with the load

Examples of exponential averaging



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

Outline

- First-Come First-Served (FCFS)
- Shortest Job First (SJF)
- Shortest Remaining Time First (SRTF)
- Priority scheduling
  - Dynamic priorities
  - Computed priorities
- Round Robin (RR)

Priority scheduling

- Associate integer priority with process, and schedule the highest priority (~ lowest number) process, e.g.,

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

- Average waiting time now

$$\frac{(1+5)+0+(1+5+10)+(1+5+10+2)+1}{5}=\frac{41}{5}=8\,{}^{1}/_{5}$$

- Consider: SJF as priority scheduling using inverse of predicted burst length

Dynamic priority scheduling

- **Starvation** can occur if low priority processes never execute
- Urban legend?
  - When the IBM 7074 at MIT was shut down in 1973, low-priority processes were found that had been submitted in 1967 and had not yet been run...
- This is the biggest problem with static priority systems!
  - A low priority process is not guaranteed to run — ever!
- Solve by making priorities **dynamic**
  - E.g., **aging** increases priority starting from a static base as time passes without process being scheduled

Computed Priority

- E.g., traditional UNIX scheduler
  - Priorities 0–127; user processes ≥ PUSER = 50
  - Round robin within priorities, quantum e.g. 100ms
- Priority of process j at start of interval i is based on
  - *nice* level, a user controllable parameter between -20 and 20, and
  - $load_j$ the sampled average length of the run queue for process j

$$P_j(i) = \mathrm{Base_j} + \frac{\mathrm{CPU}_j(i-1)}{4} + 2\times\mathrm{nice_j}$$

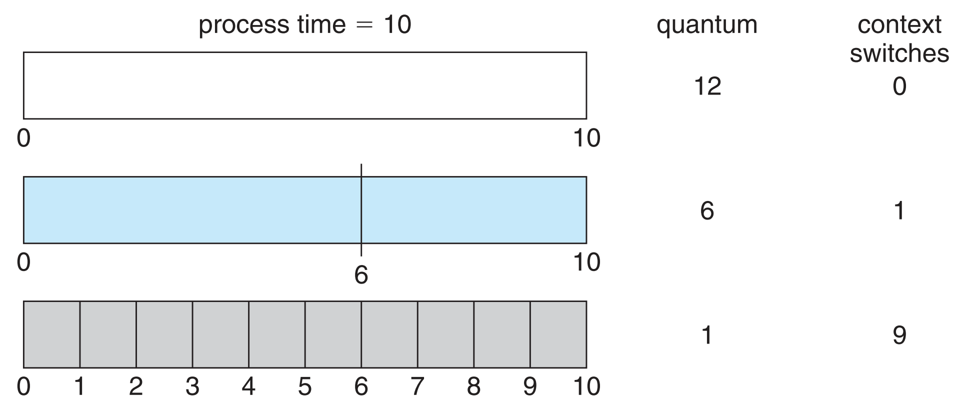
$$\mathrm{CPU}_j(i) = \frac{2\times\mathrm{load_j}}{(2\times\mathrm{load_j})+1}\mathrm{CPU}_j(i-1) + \mathrm{nice_j}$$

Outline

- First-Come First-Served (FCFS)
- Shortest Job First (SJF)
- Shortest Remaining Time First (SRTF)
- Priority scheduling
- Round Robin (RR)
  - Multilevel queues
  - Multilevel feedback queues

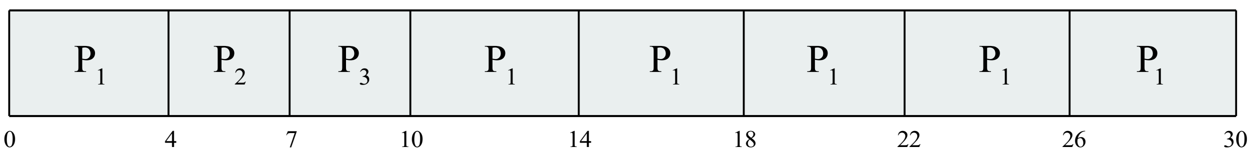
Round Robin

- A pre-emptive scheduling scheme for time-sharing systems
  - Give each process a **quantum** (or time-slice) of CPU time e.g., 10—100 milliseconds
  - Once quantum elapsed, process is pre-empted and appended to the ready queue
  - Timer interrupts every quantum to schedule next process
- Can be tricky to choose q correctly
  - q too large degenerates into a FIFO queue (∼ FCFS)
  - q too small makes the context switch overhead too great
- q usually 10ms to 100ms, while context switch < 10 μsec



Round Robin

- Consider the first example again

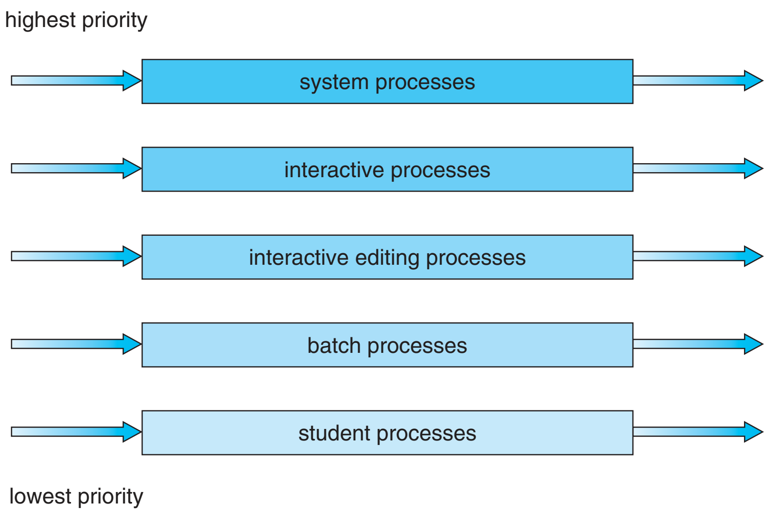


Process	Burst Time
P_1	24
P_2	3
P_3	3

- For quantum q and n processes ready,
  - **Fair**: each process gets $^1/_n$ CPU time in chunks of at most q time units, and
  - **Live**: no process ever waits more than $(n-1)q$ time units
- Typically
  - higher average turnaround time than SRTF, but
  - better average response time

Multilevel Queues

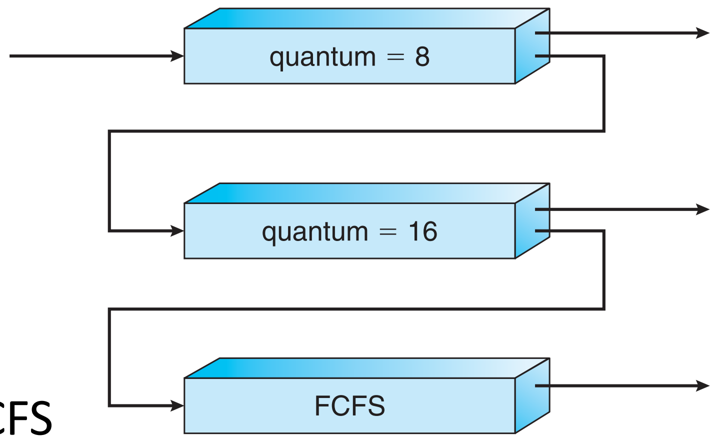
- Partition Ready queue into many queues for different types of process, e.g.,
  - Foreground/interactive processes
  - Background/batch processes
- Each process is permanently assigned a given queue
- Each queue runs its own scheduling algorithm, e.g.,
  - Foreground runs Round Robin
  - Background runs First-Come First-Served



Multilevel Feedback Queues

- Now scheduling must be done between the queues:
  - **Fixed priority**, e.g., serve all from foreground then from background, permits starvation
  - **Time slice**, each queue gets a certain amount of CPU time which it can schedule amongst its processes, e.g., 80% to foreground in RR, 20% to background in FCFS
- A process can move between the various queues
  - Aging can be implemented this way
- Multilevel-feedback-queue scheduler defined by the following parameters:
  - number of queues
  - scheduling algorithms for each queue
  - method used to determine when to upgrade a process
  - method used to determine when to demote a process
  - method used to determine which queue a process will enter when that process needs service

Multilevel Feedback Queues



- Three queues:
  - Q_0 – RR with time quantum 8 milliseconds
  - Q_1 – RR time quantum 16 milliseconds
  - Q_2 – FCFS
- Scheduling
  - A new job enters queue Q_0 which is served FCFS
  - When it gains CPU, job receives 8 milliseconds
  - If it does not finish in 8 milliseconds, job is moved to queue Q_1
  - At Q_1 job is again served FCFS and receives 16 additional milliseconds
  - If it still does not complete, it is pre-empted and moved to queue Q_2

Summary

- First-Come First-Served (FCFS)
  - Convoy effect
- Shortest Job First (SJF)
- Shortest Remaining Time First (SRTF)
  - Predicting the future
  - Exponential averaging
- Priority scheduling
  - Dynamic priorities
  - Computed priorities
- Round Robin (RR)
  - Multilevel queues
  - Multilevel feedback queues

06. Memory Management

9th ed: Ch. 8, 9

10th ed: Ch. 9, 10

Objectives

- To describe the hardware required for memory protection
- To introduce the concepts of logical and physical addresses
- To discuss the problem of address binding
- To introduce the concept of segmentation
- To understand the problem of fragmentation

Outline

- Memory protection
- Memory allocation

Outline

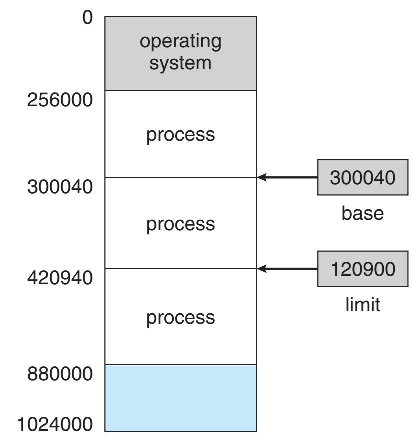
- Memory protection
  - Address binding
  - Logical and physical addresses
  - Memory Management Unit (MMU)
  - Linking and loading
- Memory allocation

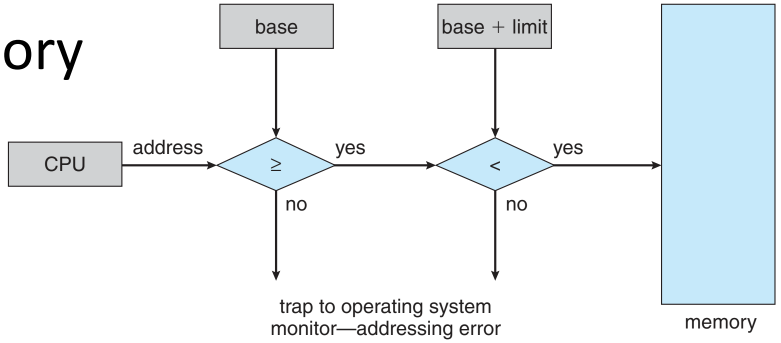
Memory management

- Will have many programs in memory simultaneously
  - Program code loaded from storage
- The CPU can only access registers and main memory directly
  - Register access in a single cycle, but memory access takes many cycles
  - Multiple levels of cache attempt to hide main memory latency (L1, L2, L3)
- Memory unit sees only a stream of
  - Address plus read request
  - Address plus data plus write request
- Need to protect memory accesses to prevent malicious or just buggy user programs corrupting other programs, including the kernel

Hardware address protection

- **Base** and **limit** registers define the logical address space
  - Base is the smallest legal address, e.g., 300040
  - Limit is the size of the range, e.g., 120900
  - Thus program can access addresses in the range $[300040, 420940)$
- CPU must check every user-mode memory access to ensure it is in that range
  - Exception raised to OS if not





Address binding

- Programs on disk is brought into memory to create running processes – but where in memory to put them?
  - Multi-programming means they can't all be put at 0x0000, the default location
- Program code will refer to memory locations – but how?
  - Consider a simple program and the assembly code it might generate
  - [Rx] means the contents of memory at address Rx

```
int x, y;
x = 5;
y = x + 3;
```

```
str #5, [Rx]     ; store 5 into x
ldr R1, [Rx]     ; load value of x from memory
add R2, R1, #3   ; and add 3 to it
str R2, [Ry]     ; and store result in y
```

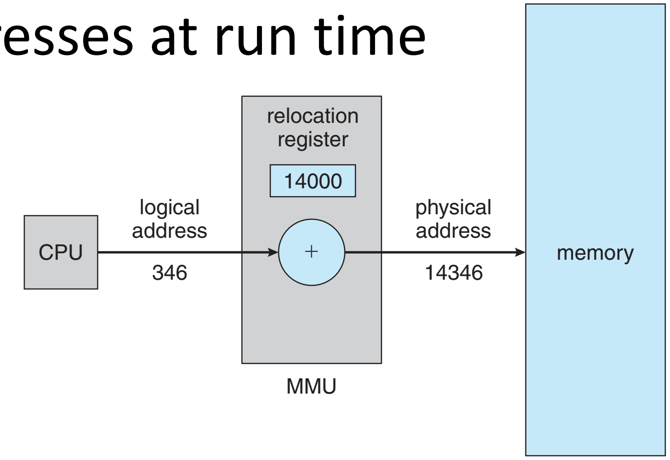
- Address binding happens at three different points
  - **Compile time**: If memory location known *a priori*, absolute code can be generated; requires recompilation if base location changes
  - **Load time**: Need to generate relocatable code if memory location is not known at compile time
  - **Execution time**: Binding delayed until run time if the process can be moved during its execution from one memory segment to another
- Bindings map one address space to another – requires hardware support

Logical vs physical addresses

- The concept of a logical address space that is bound to a separate physical address space is central to proper memory management
  - **Logical** (**virtual**) **address** – as generated by the CPU
  - **Physical address** – address seen by the memory unit
  - Identical in compile-time and load-time address-binding schemes
  - Differ in execution-time address-binding schemes
- The logical/physical address space is the set of all logical/physical addresses generated by a program
- Need hardware support to perform the mapping from logical to physical addresses at run time

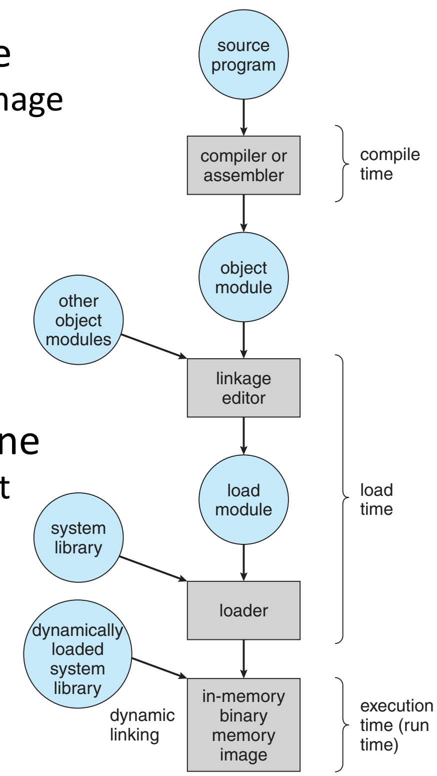
Memory Management Unit (MMU)

- Hardware that maps logical to physical addresses at run time
- Conceptually simple scheme: replace base register with **relocation register**
- Add the value in the relocation register to every address generated by a user process at the time it is sent to memory
  - User programs deal with logical addresses, never seeing physical addresses
- Execution-time binding occurs when reference is made to location in memory
  - Logical address is bound to physical addresses by the MMU



Dynamic linking and loading

- Linking combines different object code modules to create a program's code
  - **Static linking** – all libraries and program code combined into the binary program image
  - **Dynamic linking** – postpone linking to execution time
- Dynamic linking is particularly useful for libraries
  - **System** or **shared** libraries
  - May need to track versions
- Calls replaced with a **stub**
  - A small piece of code to locate the appropriate in-memory routine
- Stub replaces itself with the address of the routine, and executes the routine
  - Operating system checks if routine is in processes' memory address, adding it if not
- Dynamic loading avoids loading routines until they're called
  - Better memory usage as unused routines are never loaded
  - Requires they be compiled with relocatable addresses
  - Useful when large amounts of code are needed infrequently
- OS can help by providing libraries to implement dynamic loading

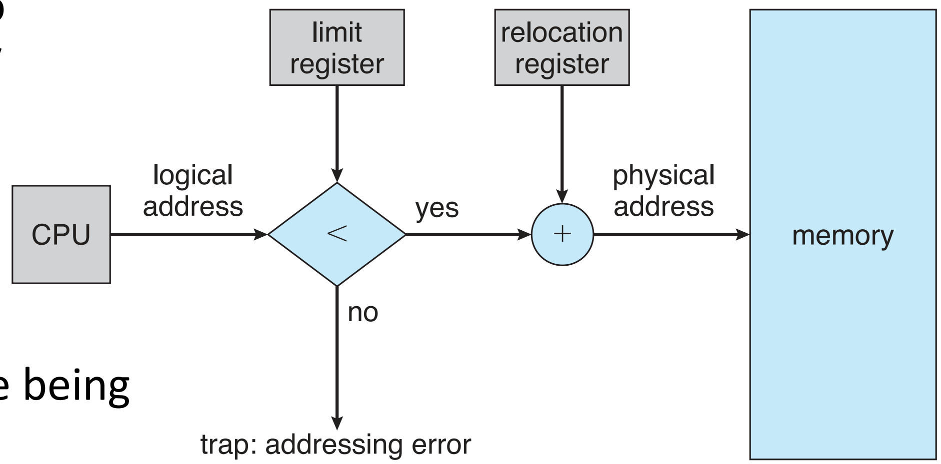


Outline

- Memory protection
- Memory allocation
  - Swapping
  - Dynamic allocation
  - Fragmentation
  - Compaction
  - Segmentation

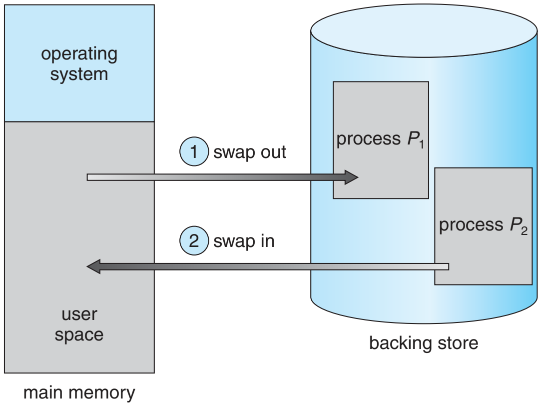
Memory allocation

- Main memory must support both kernel and user processes
  - Limited resource, must allocate efficiently
  - **Contiguous allocation** is early method putting each process in one chunk of memory
- How to determine chunks?
  - Multiple fixed-sized partitions limits the degree of multiprogramming; prefer **variable partitioning**
- Main memory usually partitioned into two
  - Resident kernel, usually held in low memory alongside interrupt vectors
  - User processes then held in high memory, each in a single contiguous section
- Relocation registers used to protect
  - User processes from each other, and
  - OS code and data from being modified
- Can then allow actions such as kernel code being transient and kernel changing size



Swapping

- When physical memory requested exceeds physical memory in machine, temporarily swap processes out
  - Move processes from main memory to storage
- Significant performance impact
  - Time to transfer process to/from storage directly proportional to the amount of memory swapped
  - Context switches can thus become very expensive
  - E.g., 100MB process with storage transfer rate of 50MB/s
- Swapping default disabled
  - Enabled only while allocated memory exceeds threshold
  - Plus consider pending I/O to / from process memory space
  - System maintains a ready queue of ready-to-run processes with memory images on disk
- Must swapped out processes be swapped in to the same physical addresses?
  - Depends on address binding method



Multiple variable-partition allocation

- **Holes**, blocks of available memory of various size are scattered throughout memory
  - When a process arrives, it is allocated memory from a hole large enough to accommodate it
  - Process exiting frees its partition, adjacent free partitions combined
- OS maintains information about:
  - allocated partitions and
  - free partitions (holes)

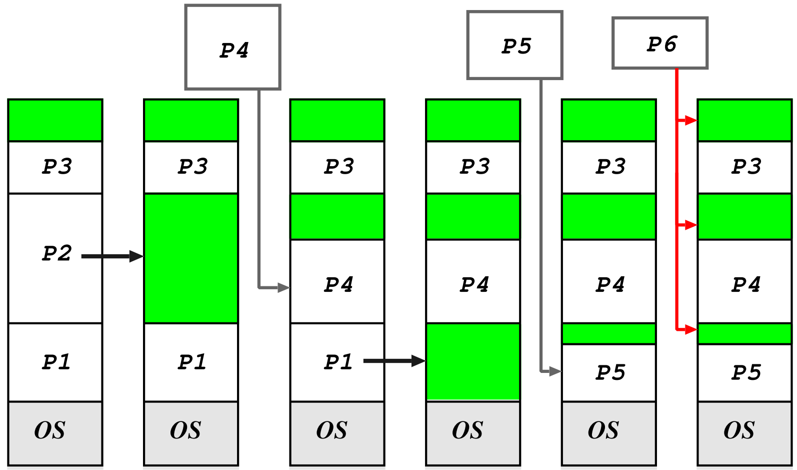
Dynamic allocation problem

- How to satisfy a request of size n from a list of free holes?

- **First-fit**, allocate the first hole that is big enough
- **Best-fit**, allocate the smallest hole that is big enough
  - Requires searching entire list, unless maintained ordered by size
  - Produces the smallest leftover hole
- **Worst-fit**, allocate the largest hole
  - Also requires searching entire list, producing the largest leftover hole
- First-fit and best-fit better than worst-fit in terms of speed and storage utilization

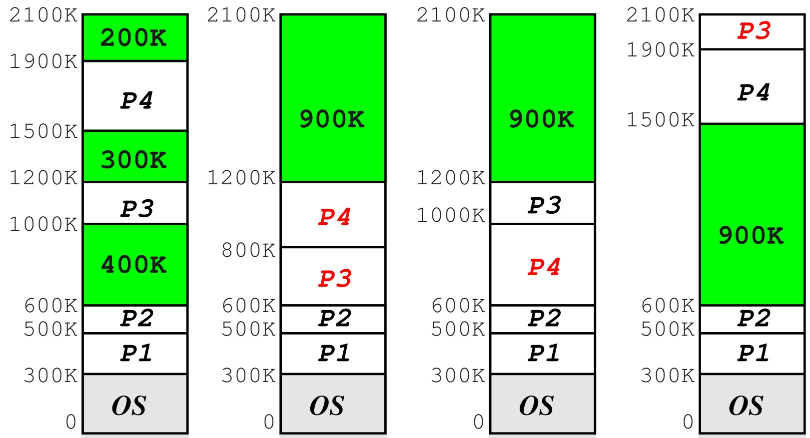
Fragmentation

- Fragmentation results in memory being unused and unusable
- **External Fragmentation**
  - Occurs when free memory exists to satisfy a request but it is not contiguous
  - Can eventually result in blocking as insufficient contiguous memory to swap any process in
- **Internal Fragmentation**
  - Occurs when allocated memory is slightly larger than requested memory
  - Memory internal to a partition, but unused
- Analysis of first-fit indicates that for *N* blocks allocated, 0.5 *N* blocks lost to fragmentation



Compaction

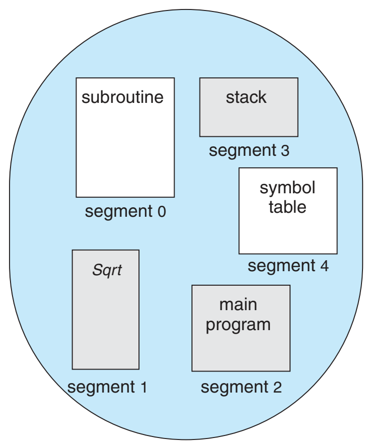
- Reduce external fragmentation by **compaction**
  - Shuffle memory contents to place all free memory together in one large block
- Compaction is possible only if
  - relocation is dynamic, and
  - done at execution time
- I/O problem
  - Pin job in memory while involved in I/O
  - Do I/O only into OS buffers



- Now consider that backing store has same fragmentation problems

Segmentation

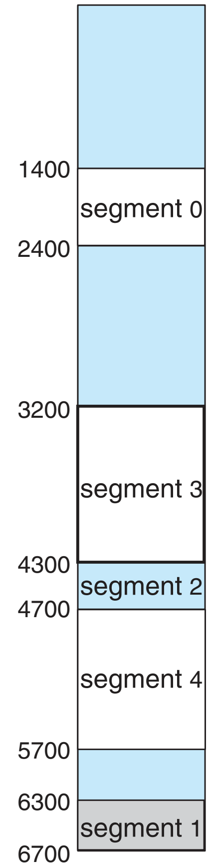
- Memory-management scheme supporting user view of memory
  - View a program as a collection of **segments**, logical program units such as the program, a procedure, an object, an array, etc
- Accessing memory requires user program to specify
  - **Segment name** (**number**) and
  - **Offset** within segment



logical address space

	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

segment table

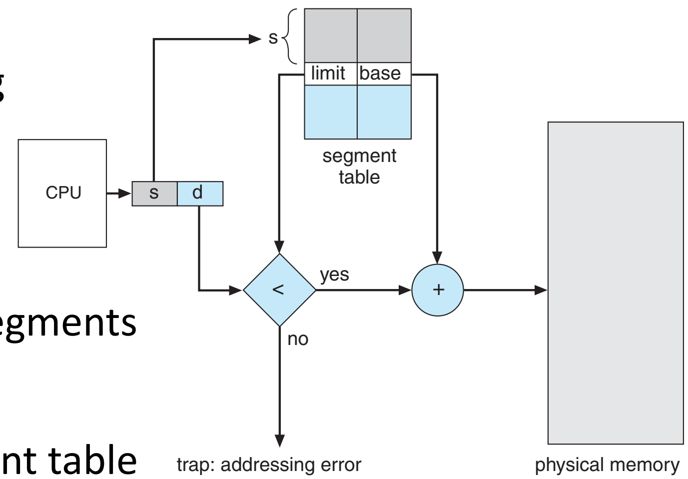


physical memory

Segmentation hardware

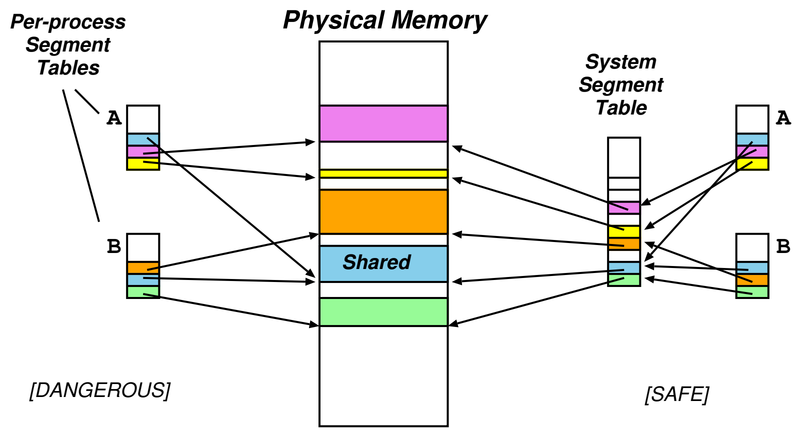
- Logical address is now a pair < *segment-number*, *offset* >
- **Segment table** maps to physical addresses via entries having
  - **Base**, the starting physical address where the segments reside
  - **Limit**, specifying the length of the segment
- **Segment-table base register** (**STBR**) points to the segment table's location in memory
- **Segment-table length register** (**STLR**) indicates number of segments used by a program;

  Segment number s is legal if $s < STLR$
- Protection provided by associating with each entry in segment table
  - Validation bit indicating legal / illegal segment
  - Read/Write/Execute privileges
  - Associated with segments so code sharing occurs at segment level
- Segments vary in length so memory allocation is a dynamic storage-allocation problem



Sharing segments is subtle

- Consider jumps within shared code
  - Specified as a condition and a transfer address < *segment-number*, *offset* >
  - *segment-number* is (of course) this one
- So all programs sharing this segment must use the same number to refer to it
  - The difficulty of finding a common shared segment number grows as the number of users sharing a segment
  - Thus, specify branches as PC-relative or relative to a register containing the current segment number
  - Read only segments containing no pointers may be shared with different segment numbers
- Wasteful to store common information on shared segment in each process segment table
  - Also dangerous as can get out of sync between processes
- Assign each segment a unique **System Segment Number (SSN)**
  - **Process Segment Table** then maps from a **Process Segment Number (PSN)** to SSN



Summary

- Memory protection
  - Address binding
  - Logical and physical addresses
  - Memory Management Unit (MMU)
  - Linking and loading
- Memory allocation
  - Swapping
  - Dynamic allocation
  - Fragmentation
  - Compaction
  - Segmentation

07. Paging

9th ed: Ch. 8, 9

10th ed: Ch. 9, 10

Objectives

- To discuss the purpose of paging
- To understand how paging is implemented
- To know some different ways that page tables are structured
- To be aware of the performance impact of the translation lookaside buffer
- To discuss how paging interacts with segmentation

Outline

- Non-contiguous allocation
- Paging implementation
- Page table structure

Outline

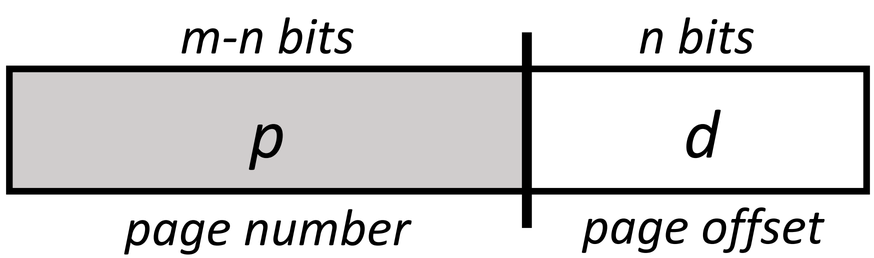
- Non-contiguous allocation
  - Address translation
  - Paging model
- Paging implementation
- Page table structure

Non-contiguous allocation

- How can we enable the physical address space of a process to be non-contiguous?
  - Allows physical memory to be allocated whenever available
  - Avoids external fragmentation and the problem of varying sized memory chunks
  - Still have internal fragmentation though
- Paging
  - Divide physical memory into **frames**, fixed-size (power of two) blocks from 512 bytes to 1GB
  - Divide logical memory into **pages**, blocks of the same fixed size
  - Build a **page table** to map between pages and frames
- Running a program that needs *N* pages then requires
  - Find *N* free frames,
  - Create entries in page table to map each page to a frame,
  - Load the program

Address translation

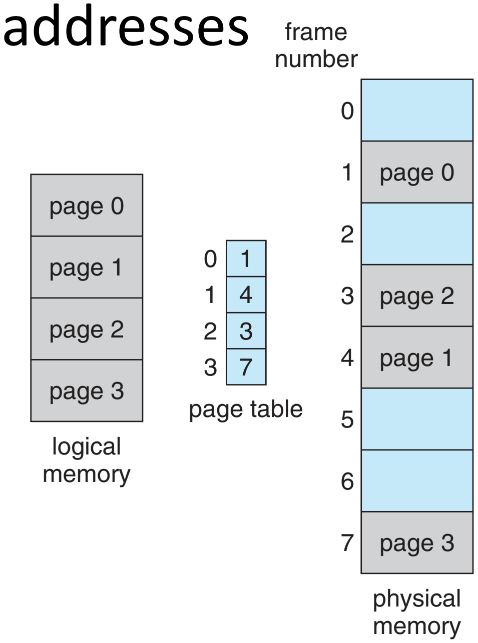
- Divide each logical address generated by the CPU into:
  - **Page number** (p) used as an index into a page table which contains base address of each page in physical memory
  - **Page offset** (d) is combined with base address to define the physical memory address that is sent to the memory unit



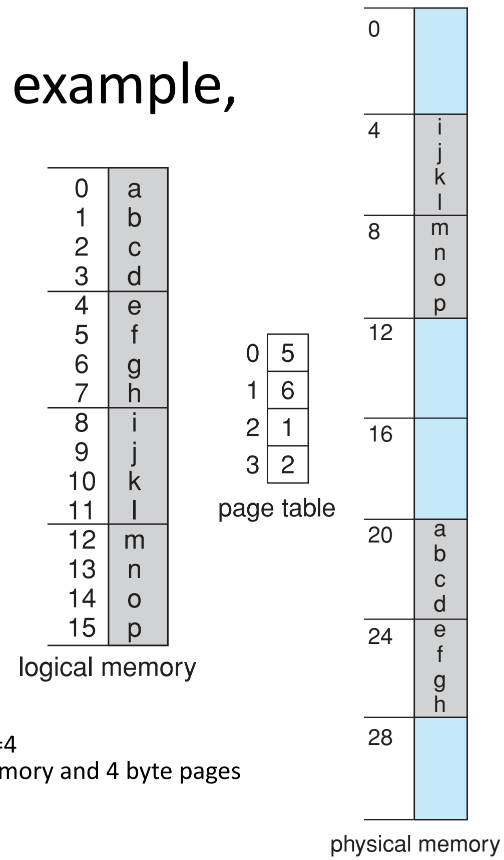
- For given logical address space 2^m and page size 2^n

Paging model

- **Page Table** stores **Page Table Entries** (**PTEs**) that map between logical and physical addresses



- For example,



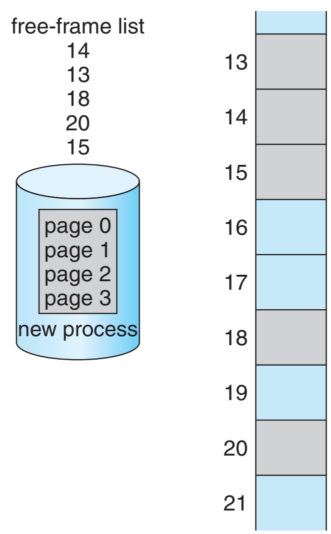
n=2 and m=4
32 byte memory and 4 byte pages

Pros and cons

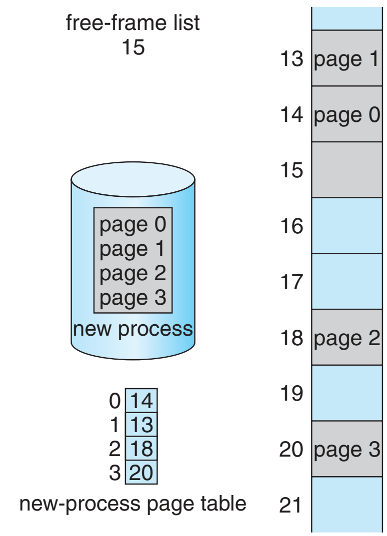
- No external fragmentation but still have internal fragmentation, e.g.,
  - Page size 2048 bytes, process size 72,766 bytes, so process requires 35 pages plus 1086 bytes, so internal fragmentation is $2048 - 1086 = 962$ bytes
- On average, fragmentation is ½ frame per process
  - So small frame sizes desirable to waste less
  - But each page table entry takes memory to track so page table grows
- Process view and physical memory now very different
  - OS controls the mapping so user process can only access its own memory
  - OS must track the free frames
  - OS must remap the page table on every context switch – adds overhead

Free frames

- Before allocation, OS has several frames on the free frame list



- After allocation, page table entries created and frames no longer in free frame list

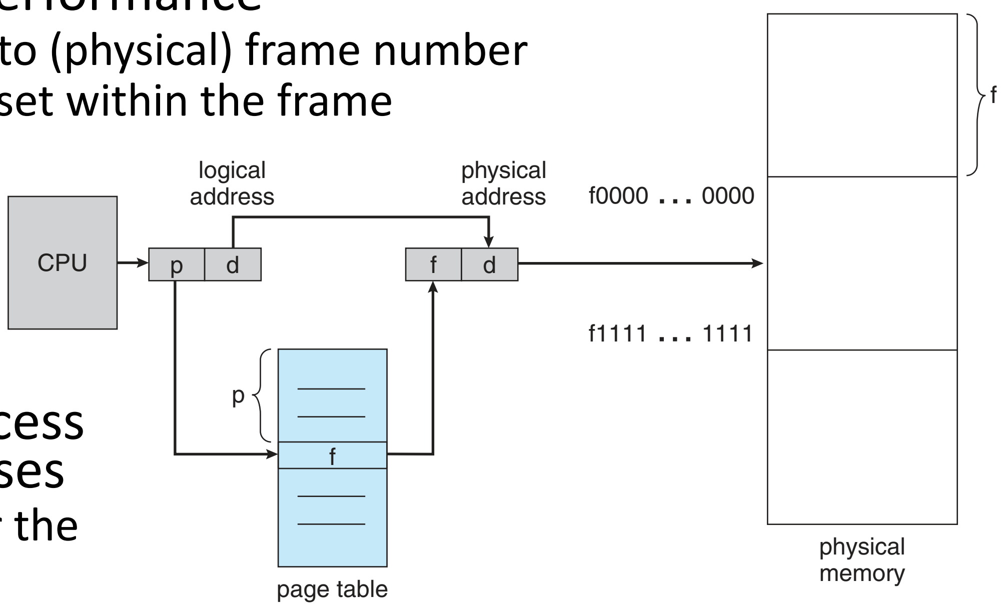


Outline

- Non-contiguous allocation
- Paging implementation
  - Free frames
  - Translation Lookaside Buffer
  - Protection
  - Sharing
- Page table structure

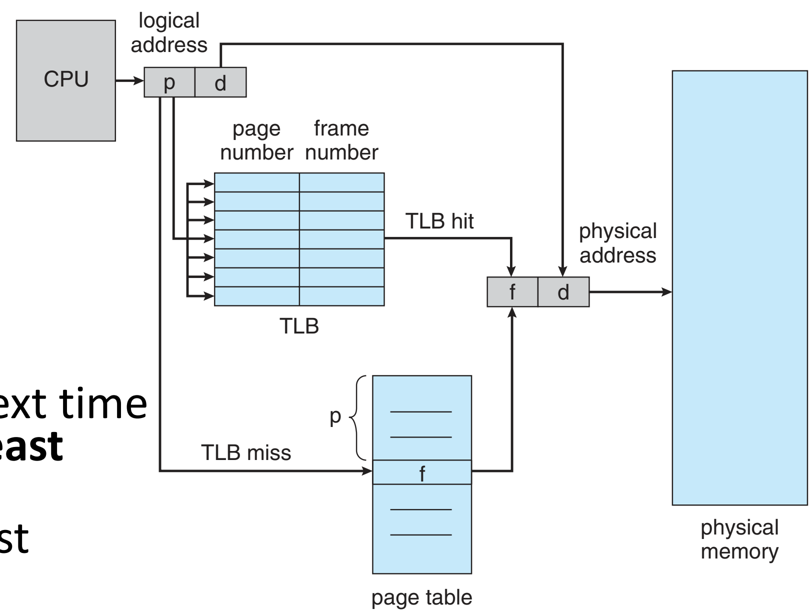
Page table implementation

- Hardware support required for performance
  - Translates (logical) page number into (physical) frame number
  - Offset within a page is then the offset within the frame
- Page table sits in main memory
  - **Page-table base register** (**PTBR**) points to the page table
  - **Page-table length register** (**PTLR**) indicates size of the page table
- Means every data/instruction access now requires two memory accesses
  - One for the page table plus one for the data/instruction
  - Dramatically reduces performance



Translation Lookaside Buffer (TLB)

- Resolves the performance issue of two memory accesses
  - Effectively a special hardware cache using associative memory
  - Typically fairly small, 64—1024 entries
- Operation
  - If translation is in the TLB, use it
  - Else we have a **TLB miss** so do the slow two-memory-access lookup in the page table
  - Also add entry to the TLB for faster access next time subject to replacement policies – typically **Least Recently Used** (**LRU**)
  - Can sometimes pin entries for permanent fast access



TLB performance

- Performance is measured in terms of **hit ratio**, the proportion of time a PTE is found in TLB, e.g., assume
  - TLB search time of 20ns, memory access time of 100ns, hit ratio of 80%
- If one memory reference is required for lookup, what is the **effective memory access time**?
  - $0.8 \times 120\mathrm{ns} + 0.2 \times 220\mathrm{ns} = 140\mathrm{ns}$
- If the hit ratio increases to 98%, what is the new effective access time?
  - $0.98 \times 120\mathrm{ns} + 0.2 \times 220\mathrm{ns} = 122\mathrm{ns}$
  - That is, it only gives a 13% improvement
  - (Intel 80486 had 32 registers and claimed a 98% hit ratio)
- TLB also adds context switch overhead as need to flush the TLB each time
  - Can store address-space identifiers (ASIDs) in each entry to avoid this

Protection

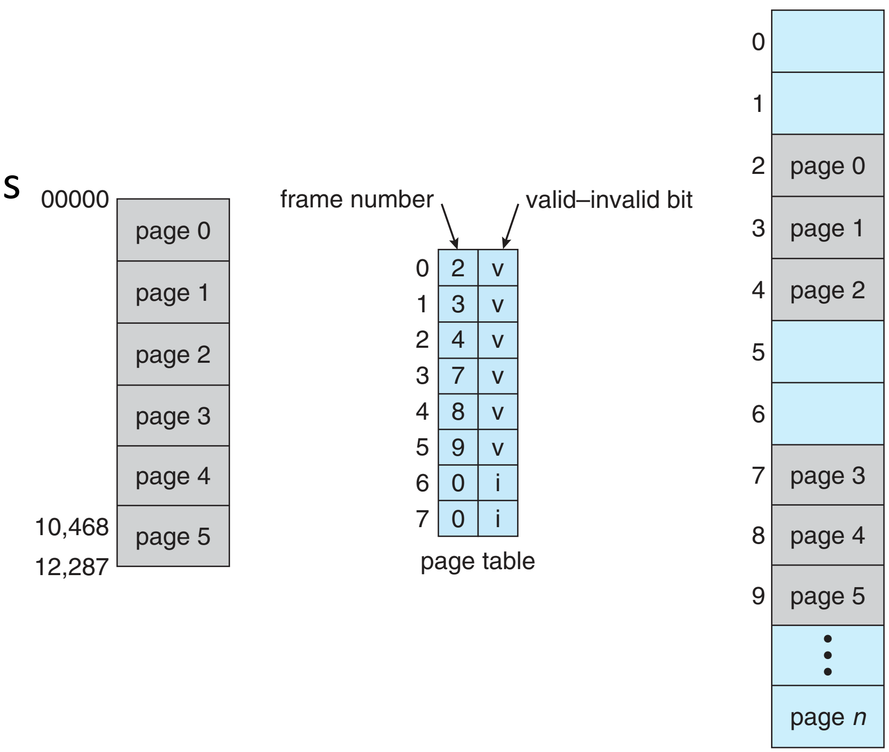
- Associate **protection bits** with each page, in the Page Table Entry (PTE), e.g.,
  - Accessible in kernel mode only, or user mode
  - Read/Write/Execute to page permitted
  - Valid/Invalid

Frame Number	K	R	W	X	V

- As the address goes through the page hardware, protection bits are checked
  - Note this only gives page granularity protection, not byte granularity protection
- Attempts to violate protection cause a hardware trap to the OS
  - TLB entry has the valid/invalid bit indicating whether the page is mapped
  - If invalid, trap to the OS handler to map the page
- Can do lots of interesting things here, particularly with regard to sharing and virtualization

Sharing pages

- Shared code
  - Keep just one copy of read-only (reentrant) code shared among processes
  - Similar to multiple threads sharing the same process space
  - Can also be useful for IPC if read-write pages can be shared
- Private code and data
  - Each process keeps its own copy of private code and data
  - Pages for which can appear anywhere in the logical address space

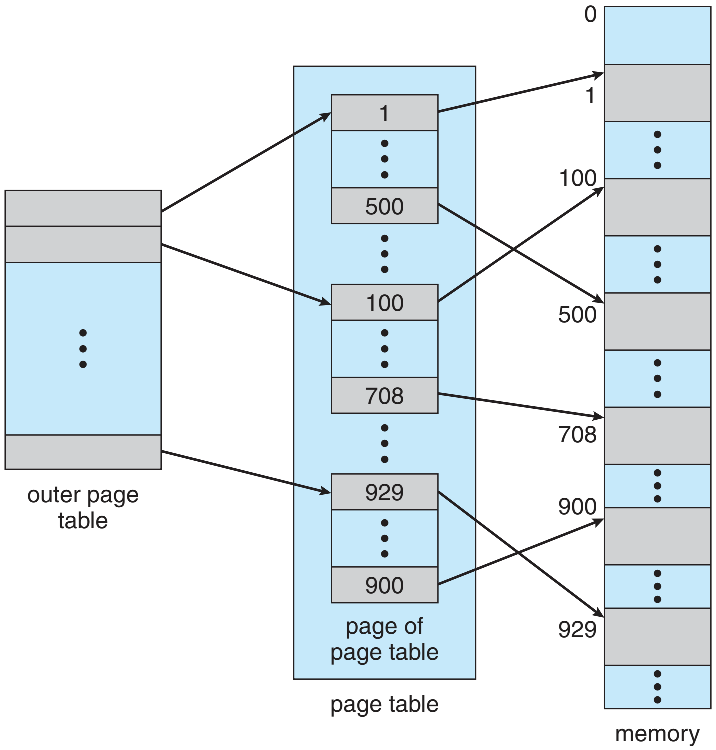


Outline

- Non-contiguous allocation
- Paging implementation
- Page table structure
  - Two-level page table
  - Larger address spaces
  - Examples: IA-32, x86-64, ARM

Page table structure

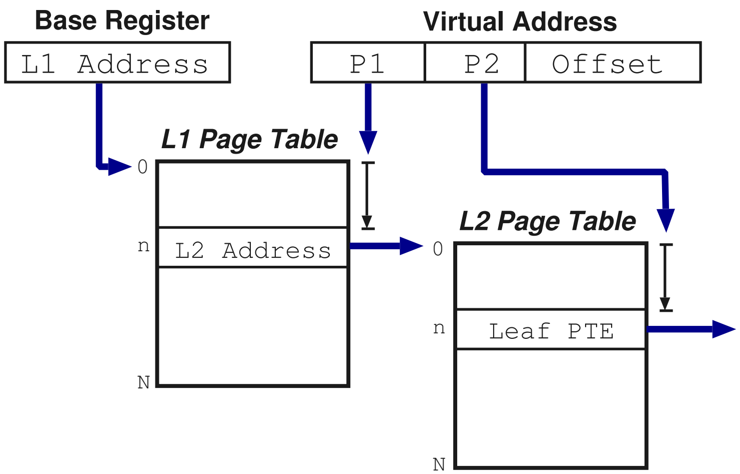
- Page tables can get huge using straight-forward methods
  - E.g., for a 32-bit logical address space and page size of 4 KB (2^{12}), page table would have 1 million entries ($2^{32}/2^{12} = 2^{20}$)
  - If each entry is 4 bytes that means 4 MB of physical memory for page table – don't want to contiguously allocate that
- Instead, split the page table into multiple levels and page out all but the outermost level



Two-level paging

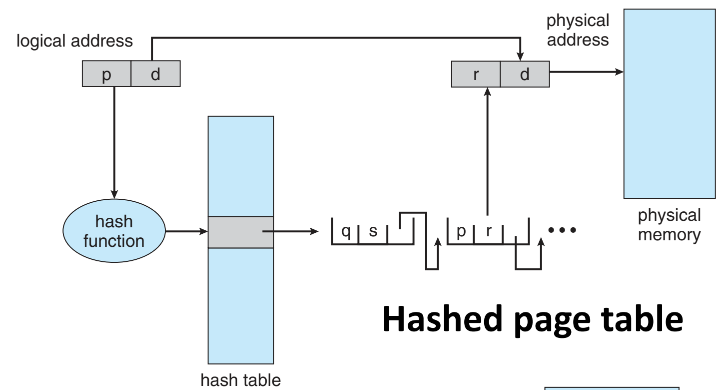
- For example, given a 20 bit page number and a 12 bit page offset, split the page number into two equal sized parts of 10 bits each
  - NB. A 12 bit offset implies $2^{12} = 4096$ byte pages
  - There is no requirement that the two (or more) parts be equal sized
- The PTBR then points to the address of the outermost L1 page table and lookup proceeds by
  - The 10 bit p_1 value indexes into the L1 page table to obtain the address of the relevant page of the L2 page table
  - The 10 bit p_2 value then indexes into the L2 page table to obtain the address of the mapped frame
  - Finally the page offset d then indexes into the frame to obtain the intended byte
- This is a **forward mapped** page table

page number		page offset
p_1	p_2	d
10	10	12

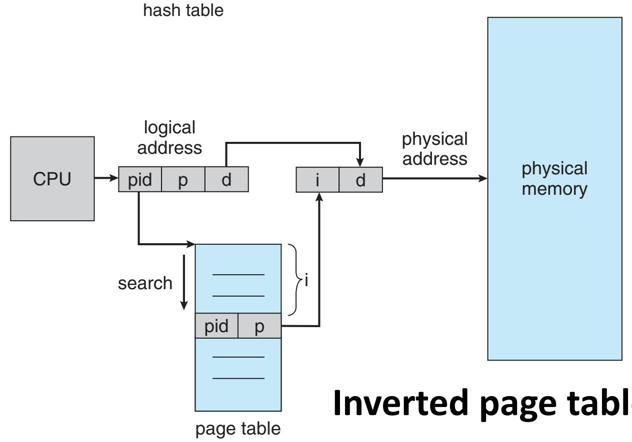


Larger address spaces

- For large address spaces – e.g., 64 bit – simple hierarchy is impractical
  - Either one or more layers remains too large,
  - Or the number of accesses to get to the target address becomes too large
- **Non-examinable** alternatives include
  - **Hashed page tables**, where the page number is hashed into a table and the chain followed until the specific entry is found
  - **Inverted page tables**, with an entry for each frame and a hash-table used to limit the search to one or a few entries, trading size for lookup latency
- Three **non-examinable** practical examples follow: Intel IA-32, Intel x86-64, and ARM



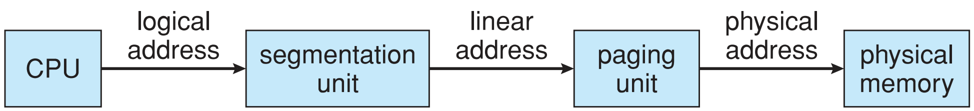
Hashed page table



Inverted page table

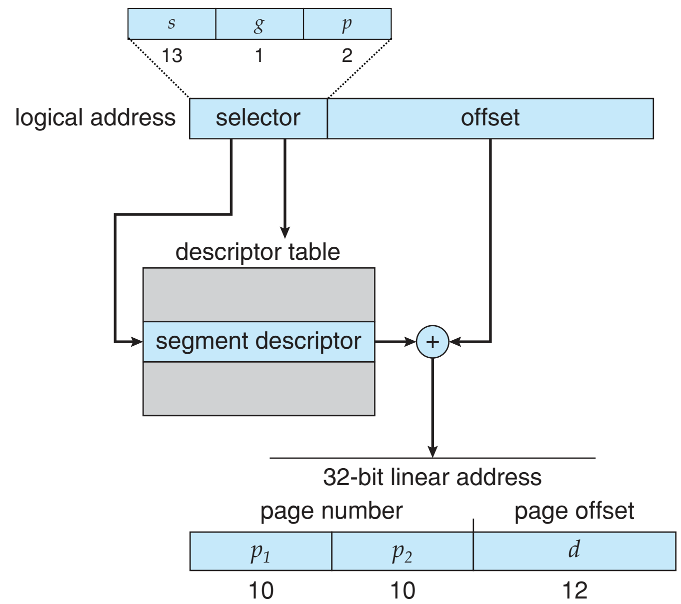
Example: Intel IA-32 architecture

- Hybrid using **segmentation with paging**
  - Each segment up to 4GB, and up to 16,384 segments per process split into two equal partitions
  - First partition's segments are private to the process, kept in the **Local Descriptor Table** (**LDT**)
  - Second partition's segments are shared among all processes, kept in the **Global Descriptor Table** (**GDT**)
  - LDT and GDT entries are 8 bytes with info about a given segment including its base location and limit
- CPU generates a **logical address** which the segmentation unit translates to a **linear address** which the paging unit translates to a **physical address**



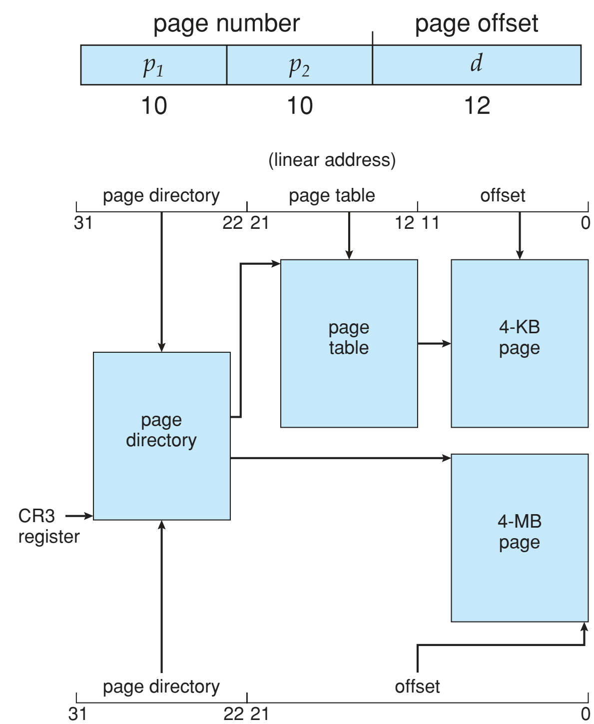
Example: Intel IA-32 architecture

- **Logical address** is a pair < *selector*, *offset* > where
  - the **selector** is a 16 bit number indicating segment number *s*, global/local indicator *g*, and protection bits *p*, and
  - the **offset** is a 32 bit number indicating the byte in the selected segment
- Generate **linear address** by
  - Six segment registers so can address six segments at any given time, and further six 8 bit microprogram registers hold the LDT/GDT descriptors
  - Segment register points to entry in LDT/GDT
  - Limit information validates the offset
  - If valid, offset is added to base giving linear address



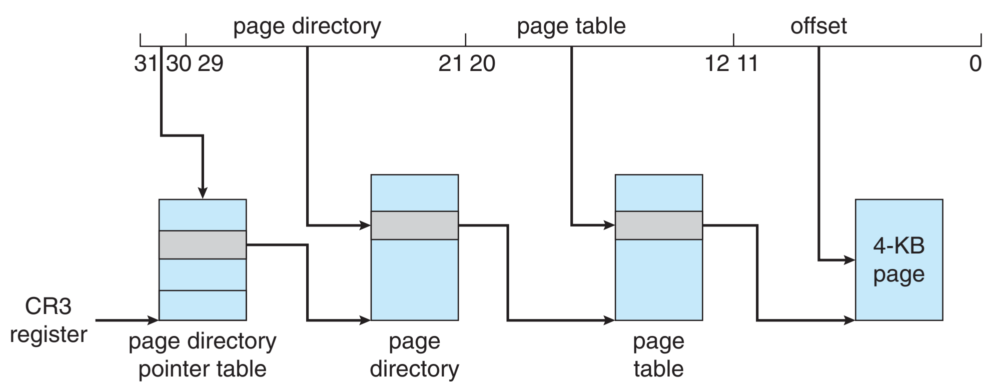
Example: Intel IA-32 architecture

- **Linear address** is then resolved
  - If the *page_size* flag is not set, then standard 4kB pages are used with a two level lookup, with Intel referring to the (outermost) L1 table as the **page directory** and the L2 table as the **page table**
  - Otherwise 4MB pages and frames are used with the page directory pointing directly to the 4MB frame, bypassing the inner page table completely
- In the former case, a valid/invalid bit in the page directory entry indicates whether the inner page table is itself swapped out or not
  - If it is, the other 31 bits indicate the disk address from which to swap it in



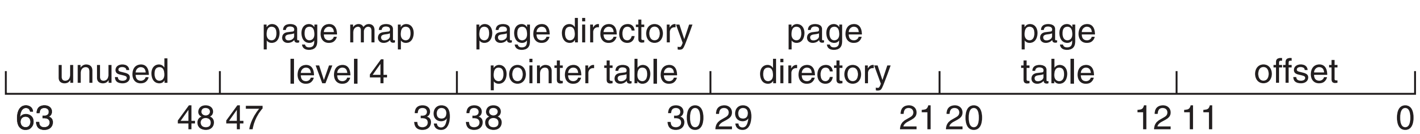
Example: Intel Page Address Extensions (PAE)

- 32 bit address limits led Intel to create **Page Address Extension** (**PAE**) allowing 36 bit addresses ~ access to 64GB physical memory
  - Paging went to a 3-level scheme
  - Top two bits refer to a page directory pointer table
  - Page-directory and page-table entries moved to 64-bits in size



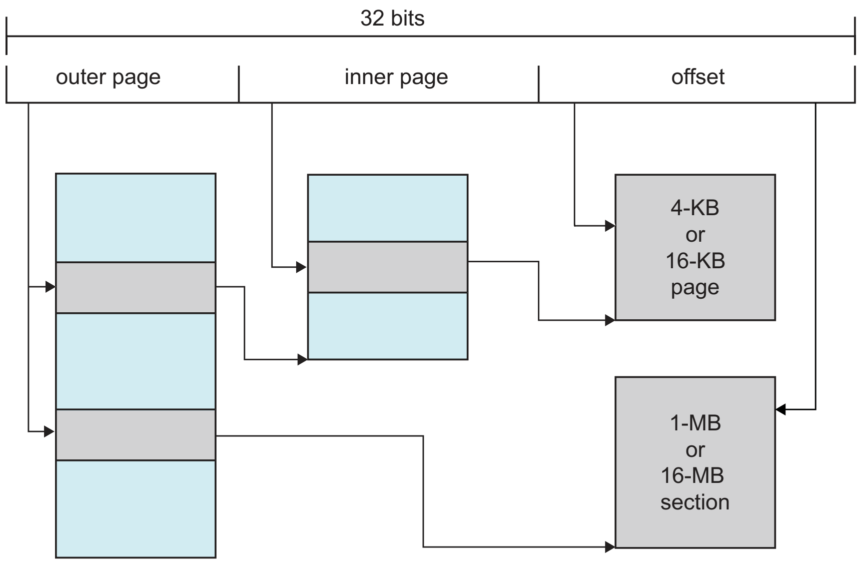
Example: Intel x86-64

- Current generation Intel x86 architecture
  - Developed by AMD, adopted by Intel
  - 64 bits is enormous – more than 16 exabytes
- In practice only implement 48 bit addressing
  - Page sizes of 4kB, 2MB, 1GB
  - Four levels of paging hierarchy
- Can also use PAE so virtual addresses are 48 bits but physical addresses are 52 bits



Example: ARM

- Modern, energy efficient, 32-bit CPU
  - Dominant mobile platform chip
  - E.g., Apple iOS and Google Android devices
- Paging structures
  - 4 kB and 16 kB pages
  - 1 MB and 16 MB pages called **sections**
  - One-level paging for sections, two-level for smaller pages
- TLB support in two levels
  - Outer level has two micro TLBs: one for data, one for instructions
  - Micro TLBs support ASIDs
  - Inner is single main TLB
- Lookup proceeds by
  - First check inner TLB
  - On miss, check outers
  - On miss, CPU performs page table walk



Summary

- Non-contiguous allocation
  - Address translation
  - Paging model
- Paging implementation
  - Free frames
  - Translation Lookaside Buffer
  - Protection
  - Sharing
- Page table structure
  - Two-level page table
  - Larger address spaces
  - Examples: IA-32, x86-64, ARM

08. Virtual Memory

9th ed: Ch. 8, 9

10th ed: Ch. 9, 10

Objectives

- To describe the benefits of a virtual memory system
- To explain the concepts of demand paging and the working set model
- To understand some page-replacement and allocation algorithms
- To be aware of problems of thrashing and Belady's anomaly

Outline

- Virtual memory
- Page faults
- Page replacement
- Frame allocation

Outline

- Virtual memory
  - Virtual memory benefits
  - Virtual address space
- Page faults
- Page replacement
- Frame allocation

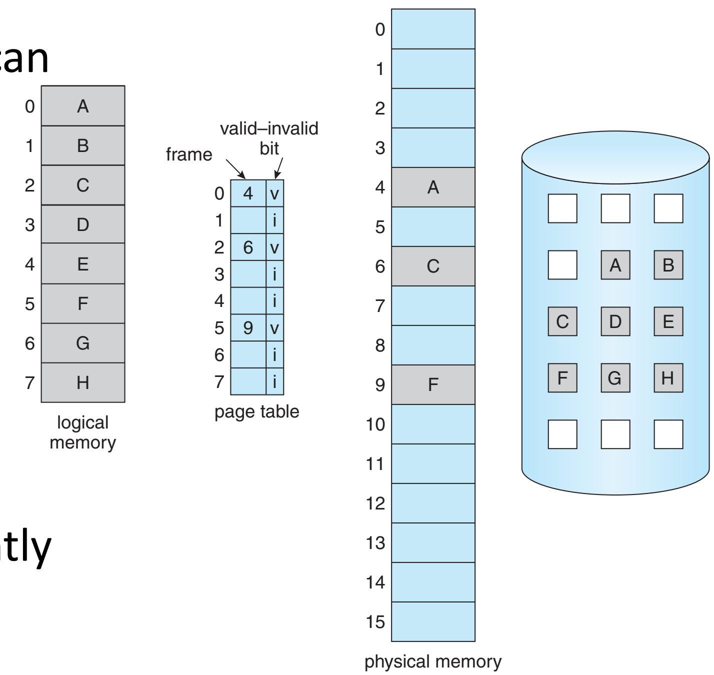
Virtual memory

- Virtual addressing allows us to introduce the idea of virtual memory
- Already have valid or invalid page translations; introduce “non-resident” designation and put such pages on a non-volatile backing store
- Processes access non-resident memory just as if it were “the real thing”
- Separates program logical memory from physical memory, allowing logical address space to be much larger than physical address space
- Implemented via **demand paging** and **demand segmentation**

Virtual memory benefits

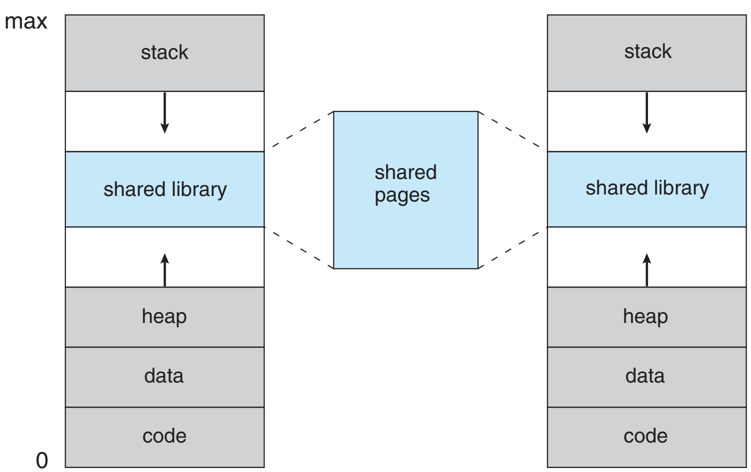
- Portability

  Programs work regardless of how much physical memory, can be larger than physical memory, and can start executing before fully loaded
- Convenience
  - Less of the program needs to be in memory at once, thus potentially more efficient multi-programming, less IO loading/swapping program into memory, large sparse data-structures easily supported
- Efficiency
  - No need to waste (real) memory on code or data which isn't used (e.g., error handling or infrequently called routines)



Virtual address space

- Virtual address space gives the logical view of how process is stored in memory
  - Usually start at address 0, contiguous addresses until end of space
- Physical memory organized in page frames
  - MMU must map logical to physical
- Usually stack starts at maximum logical address and grows “down” while heap grows “up”
  - Maximizes address space use
  - Unused address space between stack and heap is the hole
- No physical memory needed until heap or stack grows to a new page
  - Enables sparse address spaces with holes left for growth, dynamically linked libraries, etc
- System libraries shared via mapping into virtual address space
  - Shared memory by mapping pages read-write into virtual address space
  - Pages can be shared during fork(), speeding process creation

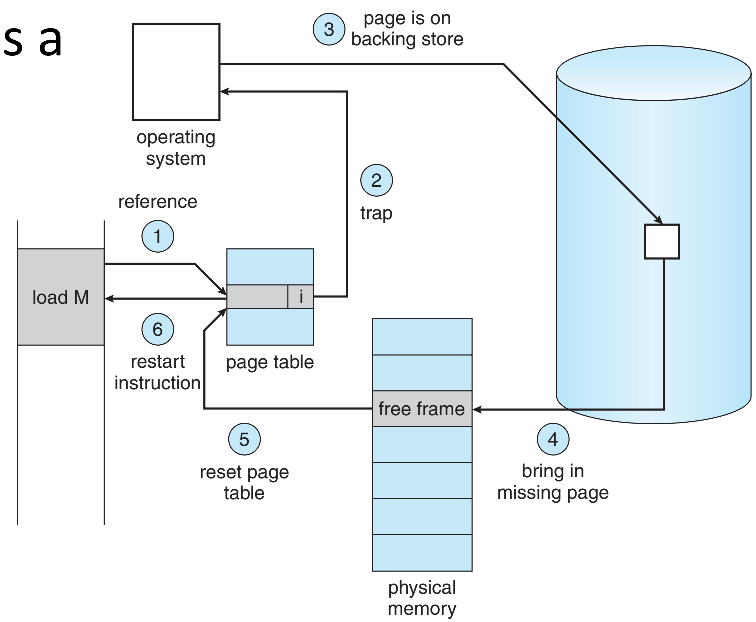


Outline

- Virtual memory
- Page faults
    - Instruction restart
    - Locality of reference
    - Demand paging
    - Optimisations
- Page replacement
- Frame allocation

Page faults

- When an invalid page is referenced, it causes a trap to the OS – a **page fault**
  - E.g., when referenced for the first time
- OS handles the trap by examining another table
  - If invalid memory reference, then abort
  - If valid but not resident, find a free frame and swap the page in
  - Entry is now marked valid as page is in memory
- After handing the fault, restart the instruction that caused the fault

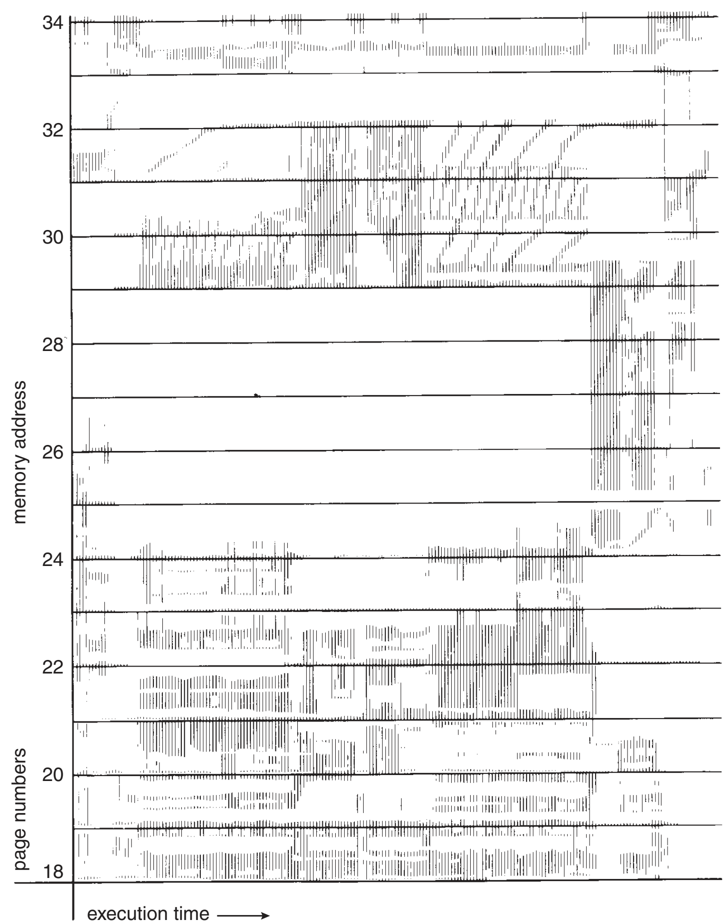


Instruction restart

- E.g., fetch and add two numbers from memory, and store the result back
  - Fetch and decode instruction (*add*), then fetch operands *A* and *B*, perform the addition, and store result to *C*
  - If store to *C* faults, need to handle the fault and then restart from the beginning (fetch and decode instruction, etc)
  - Locality of reference helps: unlikely to have multiple faults per instruction
- More complex: an instruction that could access several different locations
  - E.g., move a block of memory where source and destination can overlap, and either source or destination (or both) straddle a page boundary
  - As the instruction executes, the source might be modified – so it can't be restarted from scratch
  - Handle by, e.g., microcode for instruction strides across block, touching every page to ensure valid so no fault can occur
- **Double fault**: if the page fault handler itself triggers a fault – just give up…

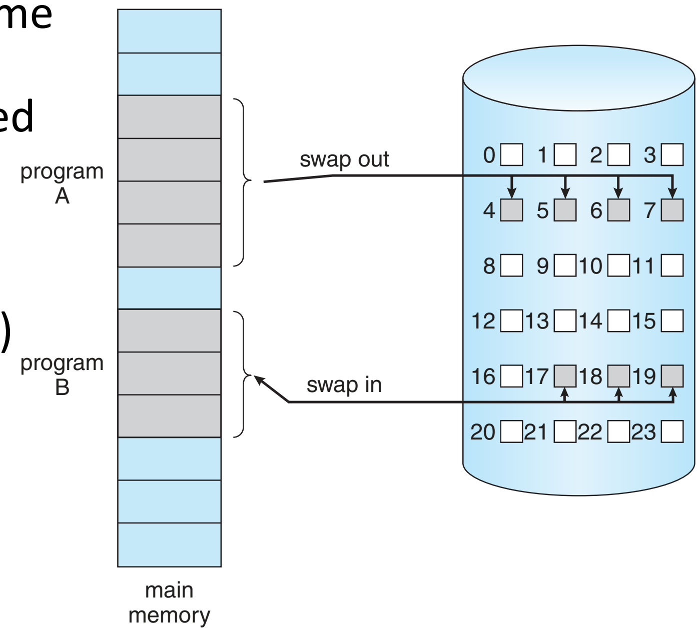
Locality of reference

- In a short time interval, the locations referenced by a process tend to group into a few regions of its address space
- E.g.,
  - Procedure being executed
  - Sub-procedures
  - Data access
  - Stack variables



Demand paging

- Could bring entire process into memory at load time, or bring pages into memory as needed
  - Reduces I/O and memory needed and response time
  - Supports more running processes
  - **Pure demand paging** starts with every page marked invalid
- Hardware support required
  - Page table with valid / invalid bit
  - Secondary memory (swap device with swap space)
  - Ability to restart instructions
- **Lazy swapper** (or **pager**) never swaps a page into memory unless page will be needed
  - But what to swap in and out?



Demand paging performance – worst case

1. Trap to the OS
2. Save the user registers and process state
3. Determine that the interrupt was a page fault
4. Check the page reference was legal and find the page on disk
5. Issue a read from the disk into 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. Reallocate CPU to another program
7. Receive an interrupt when disk I/O completes
8. Save the registers and process state for the other program
9. Determine that the interrupt was from the disk
10. Correct 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

Demand paging performance

- Assume memory access time is 200ns, average page-fault service time 8ms, and page fault rate p
  - $0 \leq p \leq 1$: if $p = 0$, no page faults; if $p = 1$, every reference causes a fault
- **Effective Access Time (EAT)**

$$= (1 - p) \times 200\text{ns} + p \times 8 \text{ ms}$$
$$= (1 - p) \times 200 + p \times 8{,}000{,}000 = 200 + 7{,}999{,}800\, p$$

- If one access in 1,000 causes a page fault, EAT $= 8.2\mu\text{secs}$ — a 40x slowdown!
- For performance degradation below 10% require

$$220 \geq \text{EAT} = 200 + 7{,}999{,}800\, p$$

- Solving for p gives $p < 0.0000025$, i.e., less than one page fault per 400,000 accesses

Demand paging optimisations

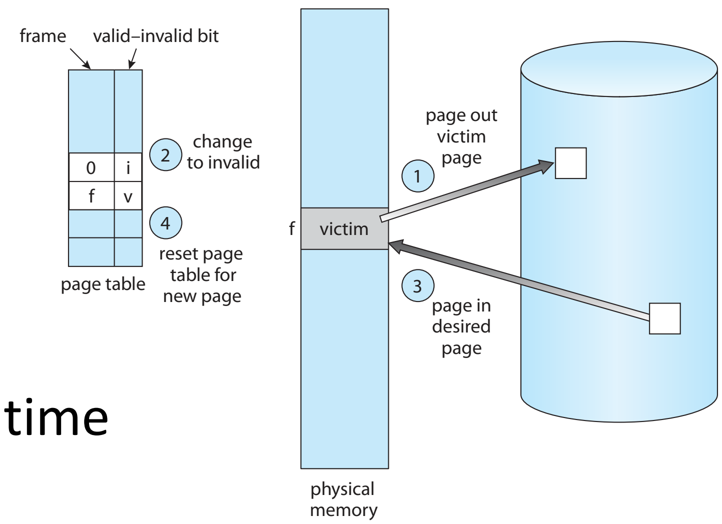
- Swap space I/O can be faster than file system I/O even on the same device
  - Allocate swap in larger chunks requiring less management than file system
  - Copy entire process image to swap space at process load time and then page in/out of swap space
- Demand page program from binary on disk – discard when freeing unmodified frame
- **Copy-on-Write** (**COW**)
  - Both parent and child processes initially share the same pages in memory
  - Only when a process actually modifies a shared page is the page copied
  - COW allows more efficient process creation as only modified pages are copied
- Allocate free pages from a pool of zero-fill-on-demand pages
  - Pool should always have free frames for fast demand page execution
  - Don't want to have to free a frame as well as other processing on page fault
- *vfork* variation of *fork* has child created as copy-on-write address space of parent
  - Very efficient when the child just calls *exec*

Outline

- Virtual memory
- Page faults
- Page replacement
  - Algorithms
  - OPT, LRU
  - Counting algorithms
  - Page buffering algorithms
  - Performance
- Frame allocation

Page replacement

- Paging in from disk requires a free frame — but physical memory is limited
  - Either discard unused pages if total demand for pages exceeds physical memory size
  - Or swap out an entire process to free some frames
- Page fault handler must
  1. Locate the desired replacement page on disk
  2. Select a free frame for the incoming page:
     1. If there is a free frame use it, else select a victim page to free
     2. Write the victim page back to disk
     3. Mark it as invalid in its process' page tables
  3. Read desired page into the now free frame
  4. Restart the faulting process
- No free frames ~ doubles page fault service time

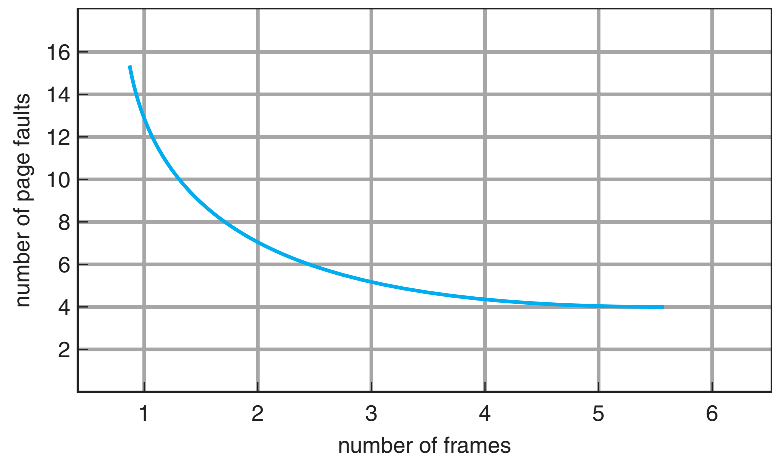


Page replacement algorithms

- Want the lowest page fault on both first and subsequent accesses
  - Evaluate using a sequence of page numbers, noting repeated access to same page does not trigger a fault

7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1

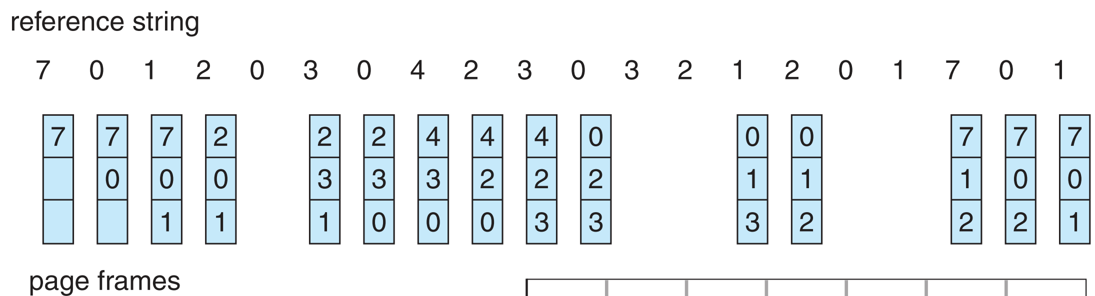
  - Assume three frames available



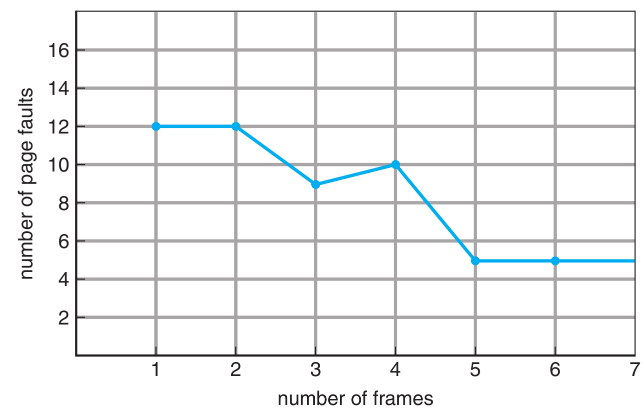
- Will look at three algorithms
  - First-In First-Out (FIFO)
  - Optimal (OPT)
  - Least Recently Used (LRU)

Page replacement algorithm: FIFO

- Simple FIFO queue for replacement gives 15 page faults

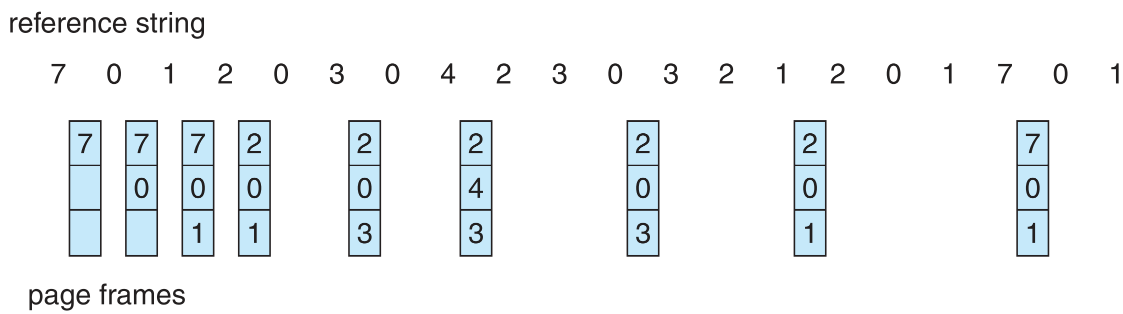


- Note that FIFO exhibits **Belady's Anomaly**
- As the number of frames **increases so can the number of page faults!**



Page replacement algorithm: OPT

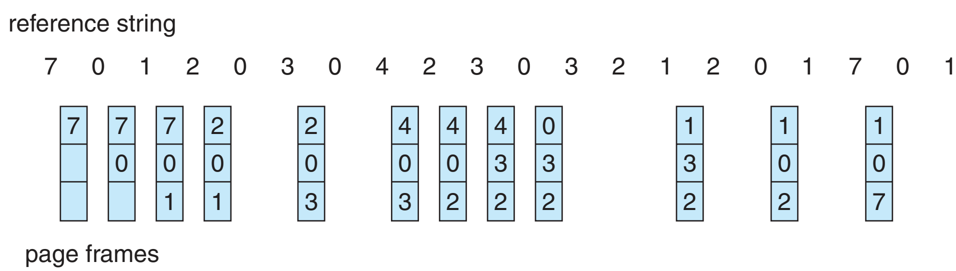
- Obvious: replace page that will not be used for the longest time



  - In this case, 9 is the best we can do
- Not obvious: how to build the oracle that knows the future
- Useful as a benchmark to measure how well your algorithm performs

Page replacement algorithm: LRU

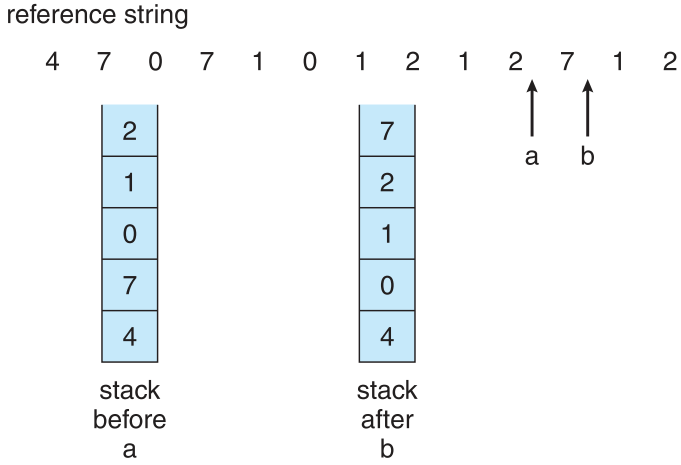
- Approximate OPT
  - Assume that the (recent) past is a good predictor of the future
  - Replace the page not used for the longest time



- Gives 12 faults – better than FIFO but worse than OPT
  - Generally good, frequently used – but how to implement?
  - Note both LRU and OPT are **stack algorithms** so don't have Belady's Anomaly

LRU implementation

- Counter implementation
  - Each PTE holds clock value, updated when page referenced through this PTE
  - Replace page with smallest counter value
  - Requires search through table, as well as memory write on every access
- Stack implementation
  - Maintain doubly-linked stack of page numbers
  - When page is referenced, move it to the top
  - Requires up to six pointers to be changed
  - Tail always points at the replacement



Approximating LRU

- Use a **reference bit** in the PTE, initially 0 and set to 1 when page touched
- **Not Recently Used** replacement
  - Periodically (every 20ms) clear reference bits
  - Victimise pages according to reference (and dirty) bits
  - Better: use an 8 bit value, shift bit in from the left
  - Maintains history for last 8 clock sweeps

Referenced?	Dirty?	Comment
no	no	best type of page to evict
no	yes	next best (needs writeback)
yes	no	probably code in use
yes	yes	bad choice of victim

- **Second-chance** (**Clock**) algorithm
  - Store pages in queue as per FIFO, often with a circular queue and a current pointer
  - Discard current if reference bit is 0 else reset reference bit (second chance) and increment current
  - Guaranteed to terminate after at most one cycle; devolves into a FIFO if all pages are referenced
- Can emulate reference bit (and dirty bit) if no hardware support
  - Mark page *no access* to clear reference bit
  - Reference causes a trap – update PTE, and resume
  - Check permissions to check if referenced

Counting algorithms

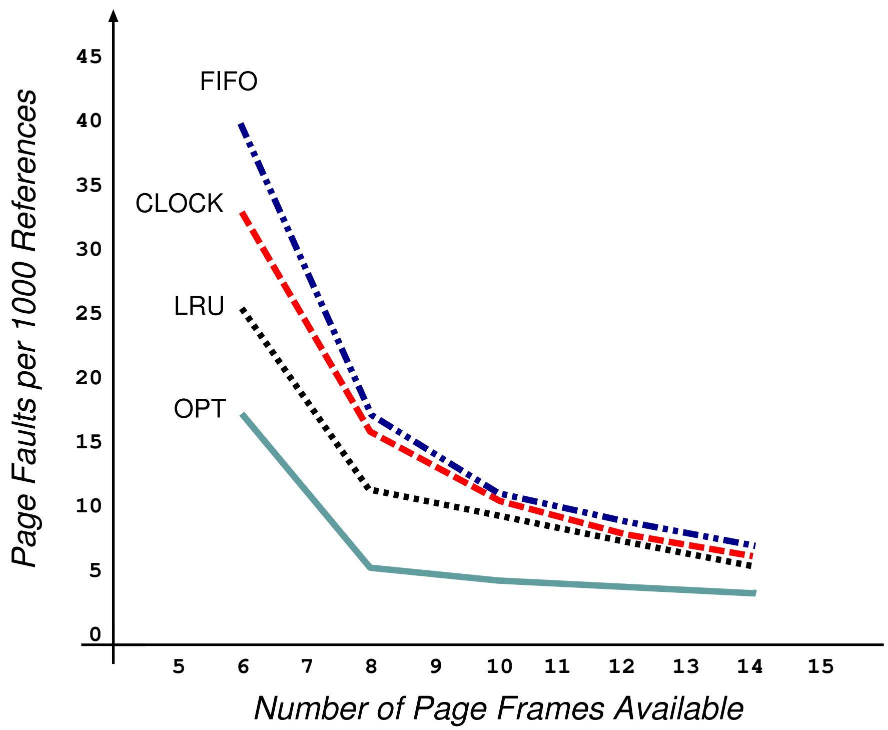
- Keep a count of the number of references to each page
- **Least Frequently Used** (**LFU**)
  - Replace page with smallest count
  - Takes no time information into account
  - Page can stick in memory from initialisation
  - Need to periodically decrement counts
- **Most Frequently Used** (**MFU**)
  - Replace highest count page
  - Low count indicates recently brought in
- Neither is common: expensive and don’t emulate OPT well

Page buffering algorithms

- Keep a minimum sized pool of free frames, always available
  - Read page into free frame before selecting victim and adding to free pool
  - When convenient, evict victim
- Possibly, keep list of modified pages
  - When backing store otherwise idle, write pages there and set to non-dirty
- Possibly, keep free frame contents intact and note what is in them
  - If referenced again before reused, no need to load contents again from disk
  - Generally useful to reduce penalty if wrong victim frame selected
- Alternatively, stop having the OS guess about future page access
  - Applications may have better knowledge, e.g., databases
  - OS can give raw access to the disk, getting out of the way of the applications

Page replacement performance comparison

- Compare page-fault rate against number of physical frames
  - Pseudo-local reference string
  - Note offset x origin
- Seek to minimise area under curve
  - Getting the frame allocation right has major impact
  - Much more than which page replacement algorithm you use!



Outline

- Virtual memory
- Page faults
- Page replacement
- Frame allocation
  - Global vs local
  - Thrashing
  - Working set

Frame allocation

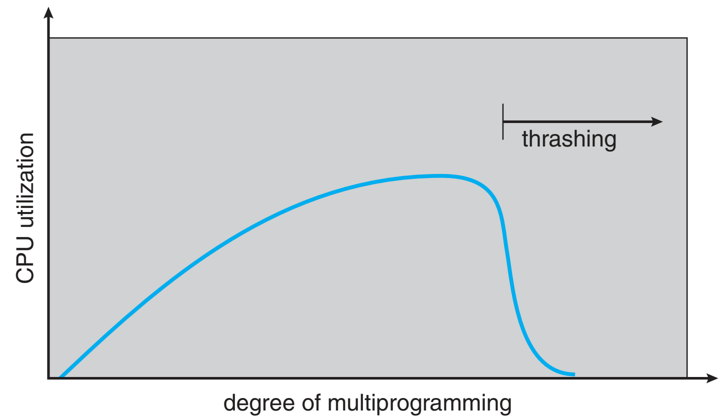
- Need an allocation policy to determine how to distribute frames
  - After reserving a fraction of physical memory per-process and for OS code/data
- Objective: Fairness (or proportional fairness)?
  - E.g. divide m frames between n processes as m/n, remainder in free pool
  - E.g. divide frames in proportion to size of process (i.e. number of pages used)
- Objective: Minimize system-wide page-fault rate?
  - E.g. allocate all memory to few processes
- Objective: Maximize level of multiprogramming?
  - E.g. allocate minimum memory to many processes

Global / Local allocation

- Most replacement schemes are **global**: any page could be a victim
  - Process execution time can vary greatly but greater throughput so more common
  - Allocation policy implicitly enforced during page-in: allocation only succeeds if policy agrees
  - Process cannot control its own page fault rate: performance can depend entirely on what other processes do
- E.g., given 64 frames and 5 processes, each gets 12 with four left over
  - When a process next faults after another process has died, it will allocate a frame
  - Eventually all will be allocated and a newly arriving process will need to steal some pages back from the existing allocations
- Alternatively, **local** replacement
  - Each process selects from only its own set of allocated frames
  - More consistent per-process performance but possibly underutilised memory

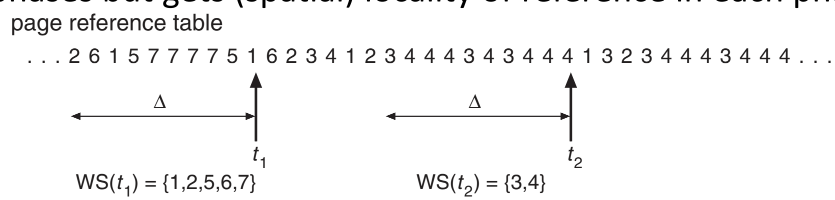
Thrashing

- A process without “enough” pages has high page-fault rate
  - Page fault to get page, replacing existing frame
  - But quickly need replaced frame back
- Cascading failure
  - Time wasted handling page faults leads to low CPU utilisation
  - Low CPU utilisation triggers OS think to increase degree of multiprogramming
  - This adds another process added to the system, increasing memory pressure
  - Collapse
- Why does demand paging work? Locality
  - Process migrates from one locality to another
  - Localities may overlap
- Thrashing occurs when size of locality > total memory
  - Limit effects by using local or priority page replacement



Working set

- Avoid thrashing by considering the **working set**
  - Those pages required at the same time for a process to make progress
  - Varies between processes and during execution
  - Assume process shifts phases but gets (spatial) locality of reference in each phase



- E.g., consider a window Δ of a fixed number of page references, say 10,000 instructions
  - Working set of process P_i is WSS_i, total number of pages referenced in the most recent window
  - Δ too small will not encompass entire locality
  - Δ too large will encompass several localities (entire program)
- Demand, $D = \sum_i WSS_i$, approximation of locality
  - Thrashing occurs if $D > m$, number of frames, in which case suspend/swap out a process
  - Approximate with interval timer and a reference bit: page in working set if one reference bit is set
  - **Pre-paging**: bring in working set pages when (re-)starting a process

Summary

- Virtual memory
  - Virtual memory benefits
  - Virtual address space
- Page faults
  - Instruction restart
  - Locality of reference
  - Demand paging
  - Optimisations
- Page replacement
  - Algorithms
  - OPT, LRU
  - Counting algorithms
  - Page buffering algorithms
  - Performance
- Frame allocation
  - Global vs local
  - Thrashing
  - Working set

09. I/O Systems

9th ed: Ch. 13

10th ed: Ch. 12

Objectives

- To understand the general structure of the I/O subsystem
- To know different ways of performing I/O including polling, interrupts, and direct memory access
- To know of different types of device
- To be aware of other issues including caching, scheduling, and performance

Outline

- I/O subsystem
- I/O devices
- Kernel data structures

Outline

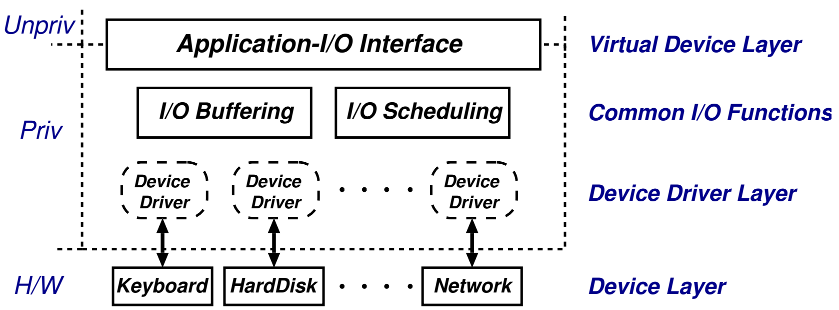
- I/O subsystem
  - Polling
  - Interrupts
  - Interrupt handling
  - Direct Memory Access (DMA)
- I/O devices
- Kernel data structures

Computers and computation rely on I/O

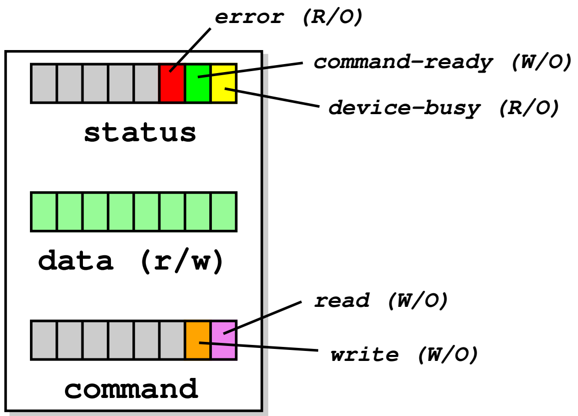
- Need input data to process, and means to output results
- There is a huge range of I/O devices
  - **Human readable**: graphical displays, keyboard, mouse, printers
  - **Machine readable**: disks, tapes, CD, sensors
  - **Communications**: modems, network interfaces, radios
- All differ significantly from one another in several ways:
  - **Data rate**: orders of magnitude different between keyboard and network
  - **Control complexity**: printers much simpler than disks
  - **Transfer unit and direction**: blocks vs characters vs frame stores
  - **Data representation**
  - **Error handling**
- I/O management is therefore a major component of an OS
  - New devices come along frequently
  - I/O performance is critical to system performance
  - Also wish to present a homogenous API

I/O subsystem

- Incredible variety of I/O devices but there are commonalities
  - Signals from I/O devices interface with computer
  - A device has at least one connection point, or **port**
  - Devices interconnect via a **bus**, either daisy-chained or shared direct access
  - Devices have integrated or separate controllers (host adapters) containing processor, microcode, private memory, etc that operate the device, handle bus connections, any ports
- Typically device will have registers to hold commands, addresses, data
  - E.g., Data-in register, data-out register, status register, control register
- Devices have addresses and are used by either
  - Direct I/O instructions, usually privileged, or
  - Memory-mapped I/O, where device registers are mapped into processor address space, especially when large (e.g., graphics cards)



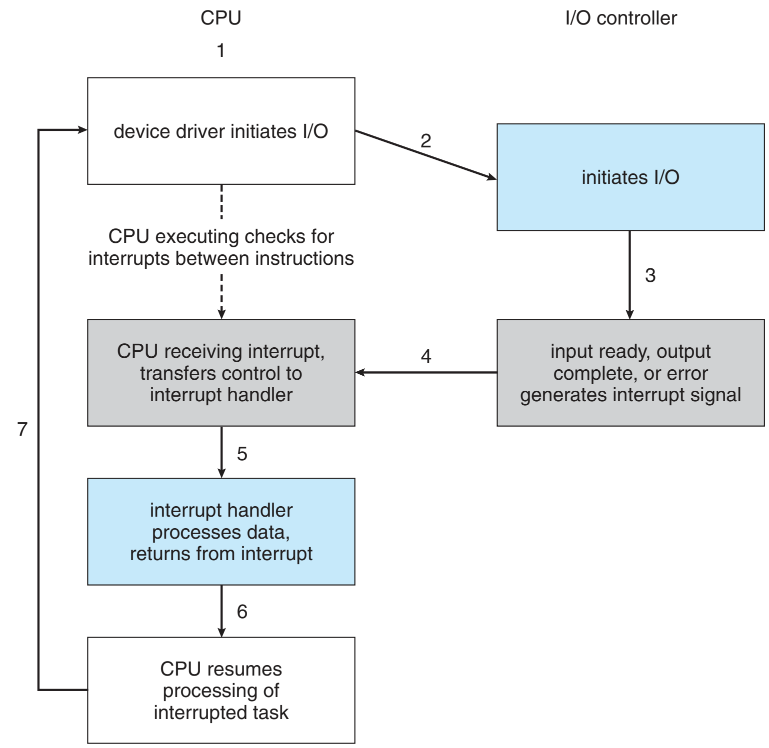
Polling



- Consider a simple device
  - Three registers: status, data and command
  - Host can read and write registers via the bus
- Polled mode operation is as follows, for every byte:
  - Host repeatedly reads *device-busy* until clear
  - Host sets *read* or *write* bit in command register, and puts data into data register
  - Host sets *command-ready* bit in status register
  - Device sees *command-ready* and sets *device-busy*
  - Device performs requested operation, executing transfer
  - Device clears *command-ready* and any *error* bit, and then clears *device-busy*
- Step 1 is polling – a **busy-wait** cycle, waiting for some I/O from device
  - This is ok if the device is fast but very inefficient if not
  - If the CPU switches to another task it risks missing a cycle leading to data being overwritten or lost

Interrupts

- More efficient than polling when device is relatively infrequently accessed
- Device triggers **interrupt-request line**
  - Checked by the CPU after each instruction
  - Aligns interrupts with instruction boundaries
- **Interrupt handler** receives the interrupt unless masked
- **Interrupt vector** dispatches interrupt to correct handler
  - Context switch required before and after
  - Priorities applied, and some interrupts may be **non-maskable**



Intel Pentium interrupt vectors

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

Handling interrupts

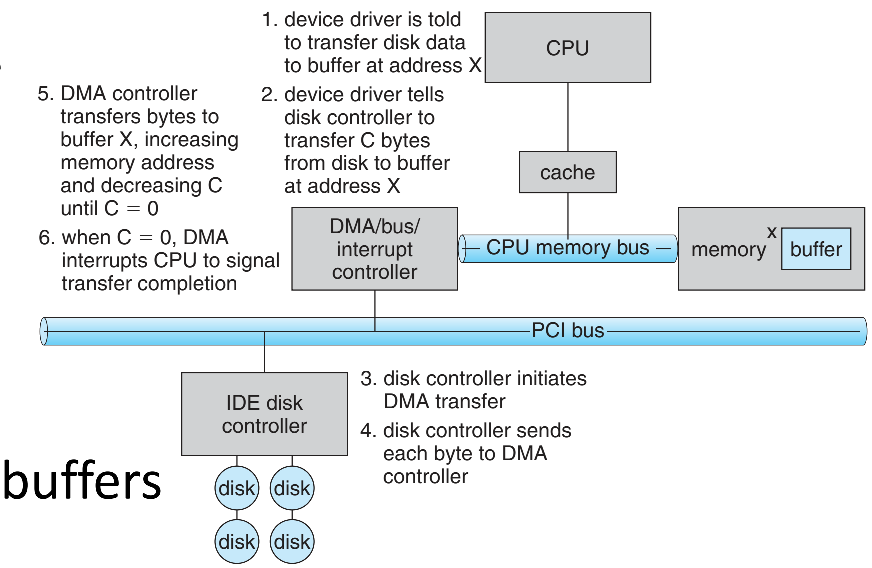
- Split the implementation into two parts:
  - Bottom half, the **interrupt handler**
  - Top half, **interrupt service routines** (ISR; per-device)
- Processor-dependent interrupt handler may:
  - Save more registers and establish a language environment
  - Demultiplex interrupt in software and invoke relevant ISR
- Device- (not processor-) dependent interrupt service routine will:
  - For programmed IO device: transfer data and clear interrupt
  - For DMA devices: acknowledge transfer; request any more pending; signal any waiting processes; and finally enter the scheduler or return
- But who is scheduling whom? Consider, e.g., network **livelock**

Direct Memory Access (DMA)

- Used for high-speed I/O devices able to transmit information at close to memory speeds
  - Interrupts good but (e.g.) **livelock** a problem
  - Better if devices can read and write processor memory **directly** – Direct Memory Access (DMA)
- Device controller transfers blocks of data from buffer storage directly to main memory without CPU intervention with generic DMA "command" include, e.g.,
  - Source address plus increment / decrement / do nothing
  - Sink address plus increment / decrement / do nothing
  - Transfer size

Direct Memory Access (DMA)

- Only generate one interrupt per block rather than one per byte
- DMA channels may be provided by dedicated DMA controller, or by devices themselves
  - E.g. disk controller passes disk address, memory address and size, and read/write
- All that's required is that a device can become a bus master
  - Requires ability for arbitration as not just CPU driving the bus
  - Involves **cycle stealing** as taking the bus away from the CPU
- Scatter/Gather DMA chains multiple requests, e.g., of disk reads into set of buffers



Outline

- I/O subsystem
- I/O devices
  - Device characteristics
  - Blocking, non-blocking, asynchronous I/O
  - I/O structure
- Kernel data structures

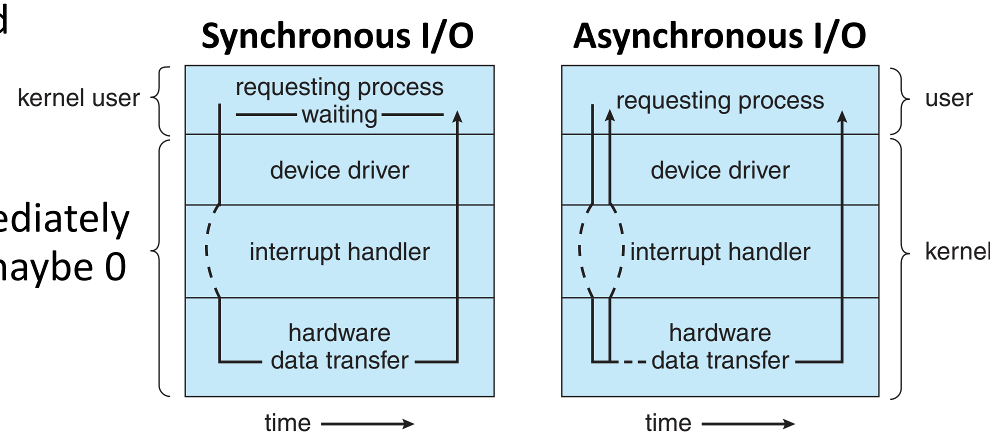
I/O device characteristics

- **Block devices**, e.g. disk drives, CD
  - Commands include *read*, *write*, *seek*
  - Can have *raw* access or via (e.g.) filesystem ("cooked") or *memory-mapped*
- **Character devices**, e.g. keyboards, mice, serial
  - Commands include *get*, *put*
  - Layer libraries on top for line editing, etc
- **Network Devices**
  - Vary enough from block and character devices to get their own interface
  - Unix and Windows NT use the Berkeley Socket interface
- **Miscellaneous**
  - Current time, elapsed time, timers, clocks
  - On Unix, *ioctl* covers other odd aspects of I/O

aspect	variation	example
data-transfer mode	character block	terminal disk
access method	sequential random	modem CD-ROM
transfer schedule	synchronous asynchronous	tape keyboard
sharing	dedicated sharable	tape keyboard
device speed	latency seek time transfer rate delay between operations	
I/O direction	read only write only read–write	CD-ROM graphics controller disk

Blocking, non-blocking, asynchronous I/O

- From programmer perspective, I/O system calls exhibit one of three behaviours
- **Blocking**
  - Process suspended until I/O completed
  - Easy to use and understand but insufficient for some needs
- **Nonblocking**
  - I/O call returns all available data, immediately
  - Returns count of bytes read/written, maybe 0
  - *select* following *read/write*
  - Relies on multi-threading



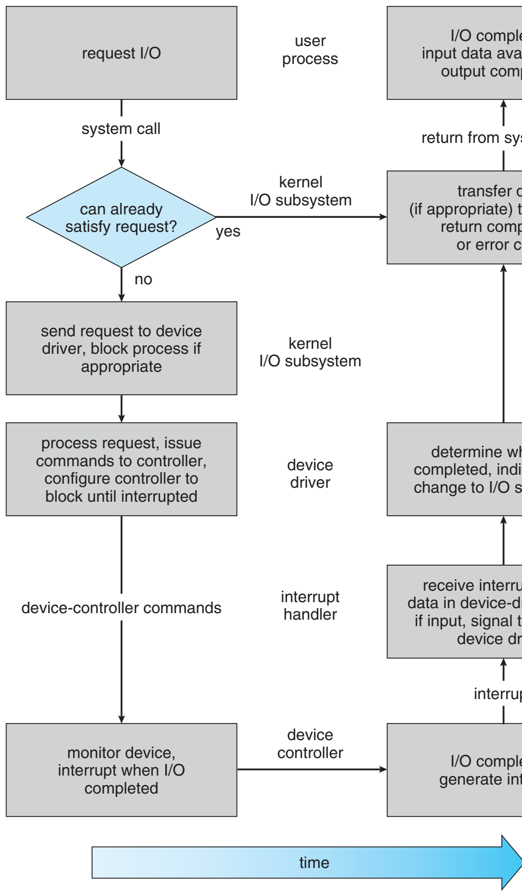
- **Asynchronous**
  - Process continues running while I/O executes with I/O subsystem explicitly signalling I/O completion
  - Most flexible and potentially most efficient, but also most complex to use

I/O structure

- **Synchronous**
  - After I/O starts, control returns to user program only upon I/O completion
  - Wait instruction idles the CPU until the next interrupt
  - Wait loop (contention for memory access)
  - At most one I/O request is outstanding at a time, no simultaneous I/O processing
- **Asynchronous**
  - After I/O starts, control returns to user program without waiting for I/O completion
  - **System call** allows application to request to the OS to allow user to wait for I/O completion
  - **Device-status table** contains entry for each I/O device indicating its type, address, and state
  - OS indexes into I/O device table to determine device status and to modify table entry to include interrupt

I/O request lifecycle

- Consider process reading a file from disk:
  - Determine device holding file
  - Translate name to device representation
  - Physically read data from disk into buffer
  - Make data available to requesting process
  - Return control to process



Outline

- I/O subsystem
- I/O devices
- Kernel data structures
  - Vectored I/O
  - Buffering
  - Other issues

Kernel data structures

- To manage all this, the OS kernel must maintain state for IO components:
  - Open file tables
  - Network connections
  - Character device states
- Results in many complex and performance critical data structures to track buffers, memory allocation, “dirty” blocks
- Consider reading a file from disk for a process:
  - Determine device holding file
  - Translate name to device representation
  - Physically read data from disk into buffer
  - Make data available to requesting process
  - Return control to process

Vectored I/O

- Enable one system call to perform multiple I/O operations
  - E.g., Unix *readve* accepts a vector of multiple buffers to read into or write from
- This **scatter-gather** method better than multiple individual I/O calls
  - Decreases context switching and system call overhead
- Some versions provide atomicity
  - Avoids, e.g., worry about multiple threads changing data while I/O occurring

Buffering

- So OS can deal with mismatches between devices, e.g., speed, transfer size), different buffering strategies can be used
  - **Single buffering**: OS assigns a system buffer to the user request
  - **Double buffering**: process consumes from one buffer while system fills the next
  - **Circular buffering**: most useful for bursty IO
  - Details often dictated by device type: character devices buffer by line; network devices are very bursty; block devices often the major user of IO buffer memory
- Can smooth peaks/troughs in data rate but can't help if on average:
  - Process demand > data rate – the process will spend time waiting, or
  - Data rate > capability of the system – the buffers will all fill and data will spill
- However, buffering can introduce jitter which is bad for real-time or multimedia applications

Other issues

- **Caching**: fast memory holding copy of data for both reads and writes; critical to IO performance
- **Scheduling**: order IO requests in per-device queues; some OSs may even attempt to be fair
- **Spooling**: queue output for a device, useful if device is "single user" (e.g., printer), i.e. can serve only one request at a time
- **Device reservation**: system calls for acquiring or releasing exclusive access to a device (care required)
- **Error handling**: generally get some form of error number or code when request fails, logged into system error log (e.g., transient write failed, disk full, device unavailable, ...)
- **Protection**: process might attempt to disrupt normal operation via illegal I/O operations so all such instructions must be privileged and memory-mapped and I/O port memory locations protected, with I/O performed via system calls
- **Performance**: I/O really affects performance through demands on CPU to execute device driver, kernel I/O code, context switches due to interrupts, data copying

Summary

- I/O subsystem
  - Polling
  - Interrupts
  - Interrupt handling
  - Direct Memory Access (DMA)
- I/O devices
  - Device haracteristics
  - Blocking, non-blocking, asynchronous I/O
  - I/O structure
- Kernel data structures
  - Vectored I/O
  - Buffering
  - Other issues

10. Storage & File Management

9th ed: Ch. (10,) 11, 12

10th ed: Ch. (11,) 13, 14, 15

Objectives

- To understand the nature of mass storage
- To be aware of the challenges of (disk) storage management
- To understand concepts of files, directories and directory namespaces, directory structures, hard- and soft-links
- To know of basic file operations and access control mechanisms
- To be aware of the relationship between paging and block storage in the buffer cache

Outline

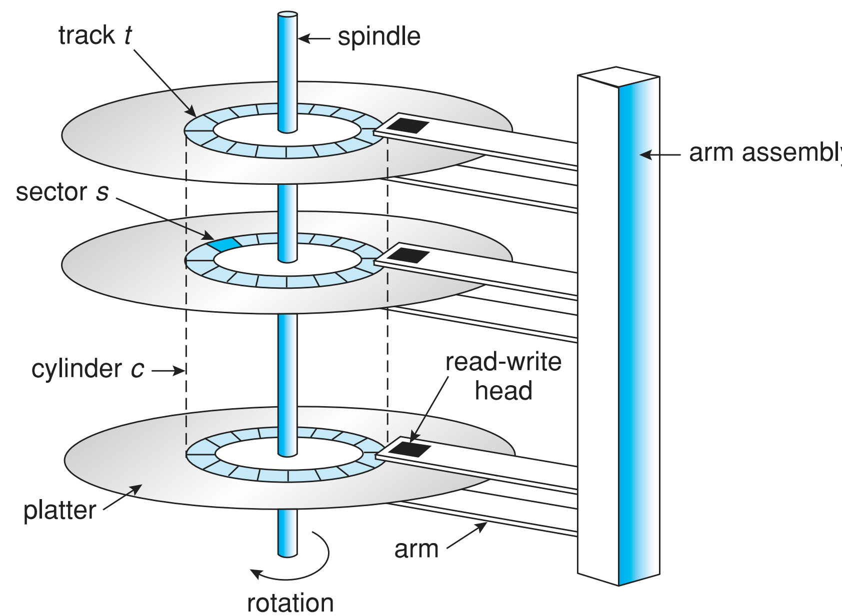
- Mass storage
- Disk scheduling
- Disk management
- Files
- Directories
- Other issues

Outline

- Mass storage
  - Hard disks
  - Solid state disks
- Disk scheduling
- Disk management
- Files
- Directories
- Other issues

Mass storage: Hard disks (HDs)

- Stack of platters
  - Historically 0.85” to 14”
  - Commonly 3.5”, 2.5”, 1.8”
  - Capacity continually increases but perhaps 30GB – 3TB
- Performance
  - Transfer Rate – theoretical – 6 Gb/sec
  - Effective Transfer Rate – real – 1Gb/sec
  - Seek time 3–12ms with around 9ms common
  - Rotation typically 7200 or 15,000 RPM



Hard disk performance

- **Average latency** [secs]
  $= \frac{1}{2}$ Latency $= {}^{1}/_{2} \times {}^{1}/_{60} / \left({}^{\text{rotations}}/_{\text{minute}}\right) = 30/\text{RPM}$
- **Access latency** [secs] = Average seek time + average latency
- **Average I/O time** [secs]
  $= \text{Access latency} + \left({}^{\text{amount to transfer}}/_{\text{transfer rate}}\right) + \text{controller overhead}$
- E.g., 4kB block, 7200 RPM, 5ms average seek time, 1Gb/sec transfer rate, 0.1ms controller overhead
  - Average latency $= 30/7200 = 4.17\text{ms}$
  - Transfer time $= (4096\text{ B} \times 8\, {}^{b}/_{B})/1024^3\text{ b/s} = 0.031\text{ms}$
  - Average I/O time $= 5\text{ms} + 4.17\text{ms} + 0.031\text{ms} + 0.1\text{ms} = 9.301\text{ms}$

Mass storage: Solid state disks (SSDs)

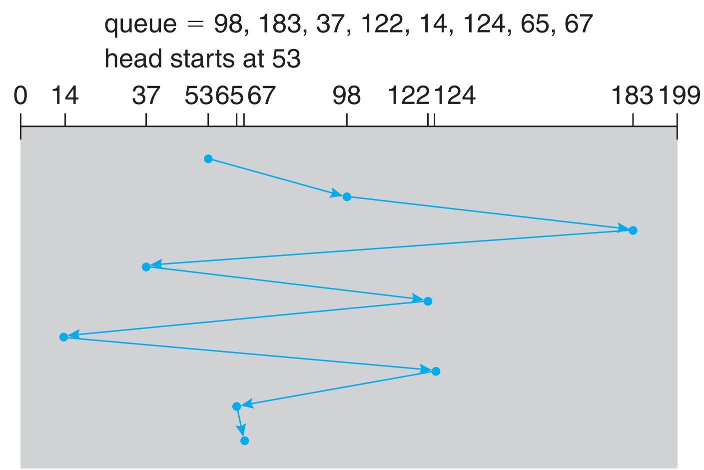
- Non-volatile memory used like a hard drive; many variations
- Pros
  - Can be more reliable than HDDs
  - No moving parts, so no seek time or rotational latency
  - Much faster
- Cons
  - Reads/writes wear out cells leading to unreliability and potentially shorter
  - More expensive per MB
  - Lower capacity

Outline

- Mass storage
- Disk scheduling
    - First-Come First-Served
    - Shortest Seek Time First
    - SCAN, C-SCAN
- Disk management
- Files
- Directories
- Other issues

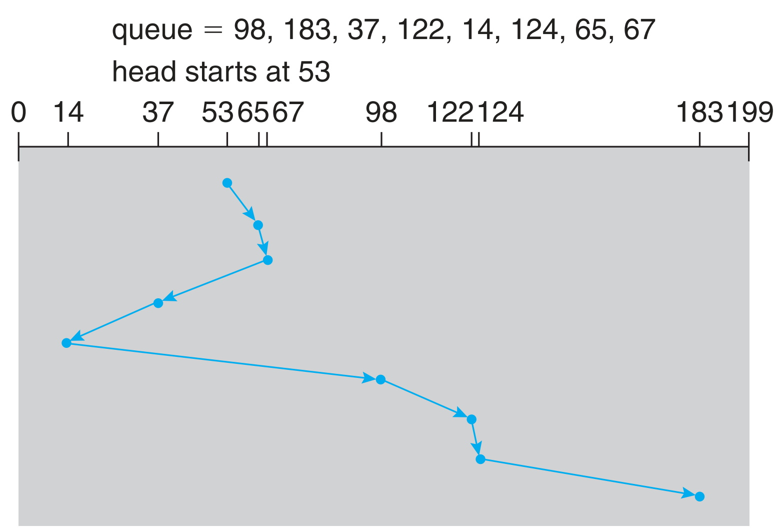
Disk scheduling

- The disk controller receives a sequence of read/write requests from the OS that it must schedule
  - How best to order reads and writes to achieve policy aim?
  - Analogous to CPU scheduling but with very different mechanisms, constraints, and policy aims
  - Many algorithms exist
- Simplest: First-come First-served (FCFS)
  - Intrinsically fair but inefficient
  - E.g., requests for blocks on cylinders are 98, 183, 37, 122, 14, 124, 65, 67



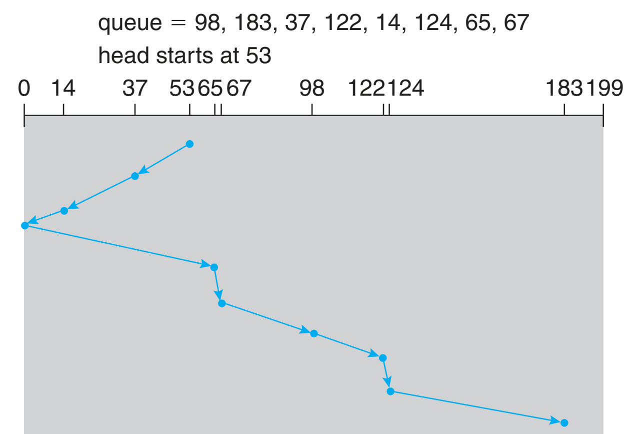
Shortest Seek-Time First (SSTF)

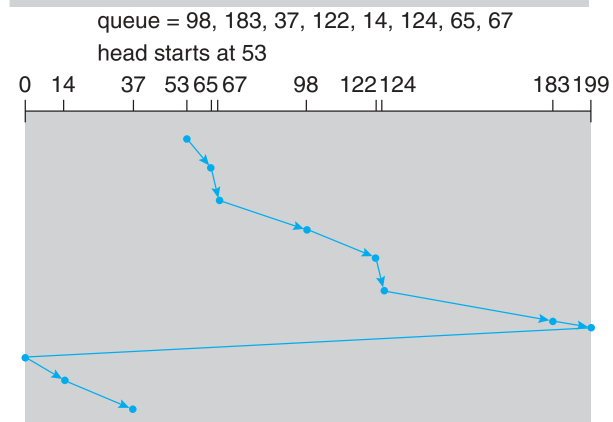
- Service requests based on distance to current head position
  - Next request in queue is that with the shortest seek time
- For this example, involves movement of just 236 cylinders
  - $^{1}/_{3}$ of that required by FCFS
- Somewhat analogous to SJF
  - A big improvement but allows starvation
  - Not optimal: from 53 move to 37 then 14 and then 65 etc – gives movement of 208 cylinders



SCAN and C-SCAN

- **SCAN** or **elevator** algorithm
  - Start at one end of the disk and move to the other end
  - Service everything on the way
- Consider density of requests when changing direction
  - Have just serviced (almost) everything in that vicinity
  - Those furthest away have waited longest so…
- **Circular-SCAN**
  - Return back to the start when reaching the end
  - Cylinders treated as a circular list, wrapping when reaching the end





Outline

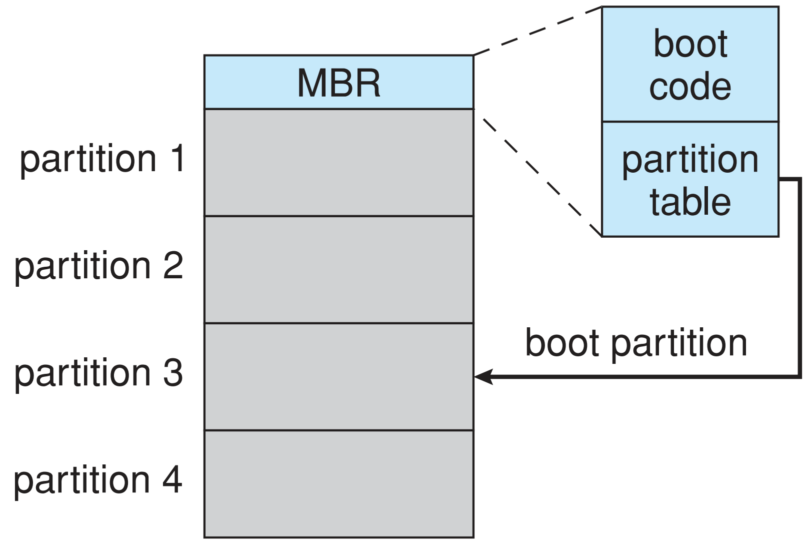
- Mass storage
- Disk scheduling
- Disk management
  - Booting from disk
- Files
- Directories
- Other issues

Disk management

- **Low-level** or **physical formatting**
  - Divides a disk into sectors that the disk controller can read and write
  - Each sector can hold header information, plus data, plus error correction code (ECC)
  - Usually 512 bytes of data but can be selectable
- **Logical formatting** to make a file system required before disk can hold files
  - OS needs to record its own data structures on the disk so it can find files
  - Partition the disk into one or more groups of cylinders, each treated as a logical disk
  - To increase efficiency most file systems group blocks into clusters
- **Disk** I/O done in **blocks**
- **File** I/O done in **clusters**
  - Some applications, e.g., databases, will prefer “raw” block access

Booting from disk

- OS needs to know where to start looking
  - BIOS (or similar) is "firm-coded" to e.g., read first block of first disk
- First block contains bootloader program, which is executed
- Bootloader knows enough to start reading in the right blocks to read the filesystem starting with the **partition table**
  - Sometimes need to **chain-load** to get enough code to parse more complex filesystems
- Allows for handling of bad blocks
  - E.g., by **sector sparing** where spare good blocks logically substitute for bad ones



Outline

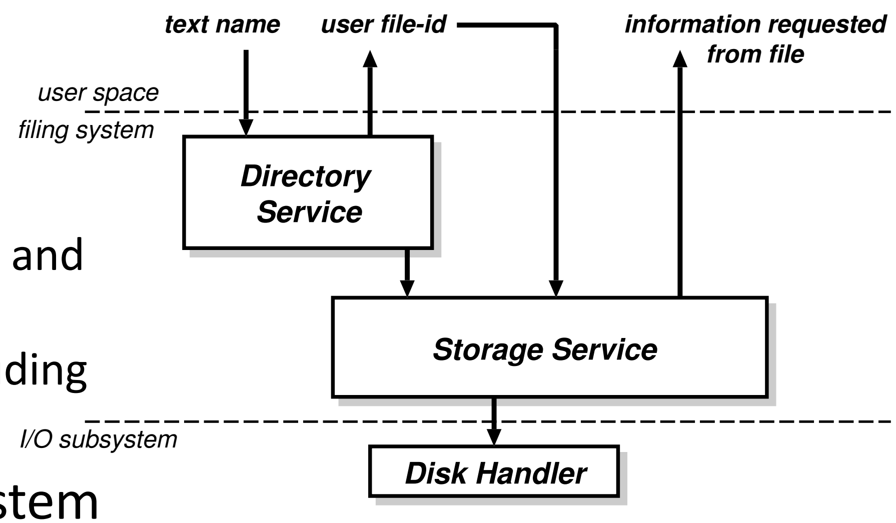
- Mass storage
- Disk scheduling
- Disk management
- Files
  - File systems
  - File metadata
  - File and directory operations
- Directories
- Other issues

Files

- The basic abstraction for non-volatile storage:
  - Can be a user or an OS abstraction (convenience vs flexibility)
  - Typically comprises a single contiguous logical address space
- Many different types
  - Data: numeric, character, binary (text vs binary split quite common)
  - Program: source, object, executable
  - “Documents”
- Can have varied internal structure:
  - None: a simple sequence of words or bytes
  - Simple record structures: lines, fixed length, variable length
  - Complex internal structure: formatted document, relocatable object file

File system

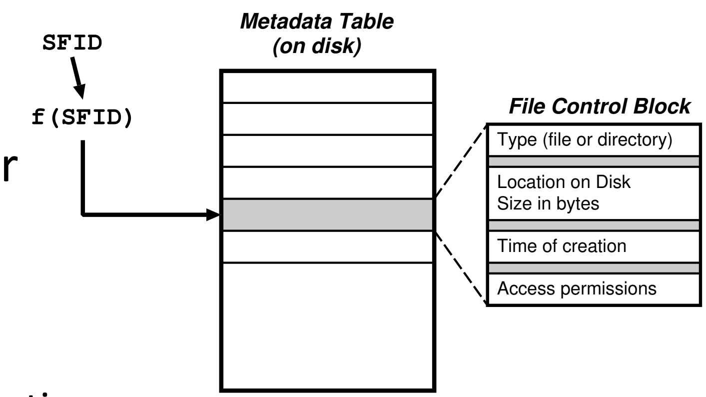
- Consider only simple file systems
  - **Directory service** maps names to file identifiers and metadata, handles access and existence control
  - **Storage service** stores data on disk, including storing directories
- Each partition formatted with a filesystem
  - Logically, a **directory** and some **files**
  - Directory maps human name (*hello.java*) to **System File ID** (typically an integer)
  - Different filesystems implement using different structures



Name	SFID
hello.java	12353
Makefile	23812
README	9742

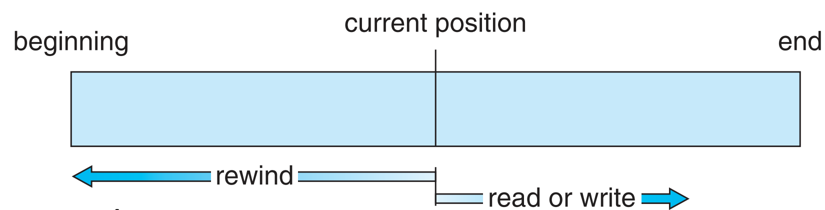
File metadata

- The mapping from SFID to File Control Block (FCB) is filesystem specific
- Files typically have a number of other attributes or metadata stored in directory
  - **Type** – file or directory
  - **Location** – pointer to file location on device
  - **Size** – current file size
  - **Protection** – controls who can do reading, writing, executing
  - **Time**, **date**, and **user identification** – data for protection, security, and usage monitoring
- OS must also track open files in an **open-file table** containing
  - **File pointer** or **cursor**: last read/written location per process with the file open
  - **File-open count**: how often is each file open, so as to remove it from open-file table when last process closes it
  - **On-disk location**: a cache of data access information
  - **Access rights**: per-process access mode information



File and directory operations

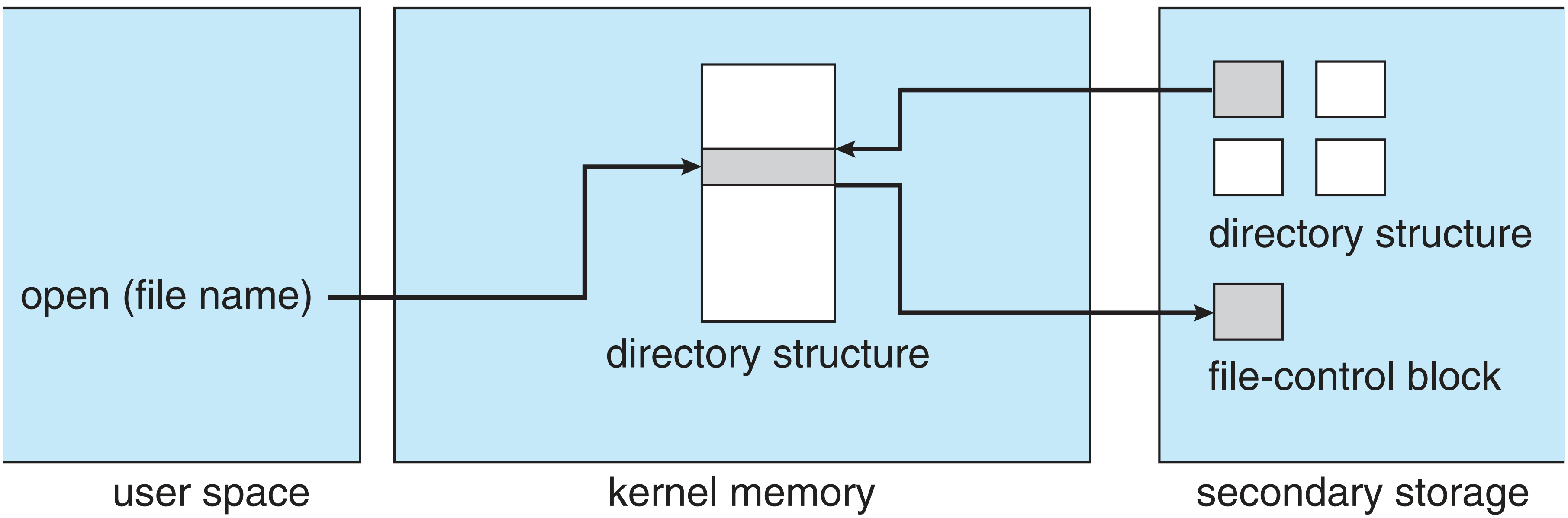
- A file as an **abstract data type** (**ADT**) over some (possibly structured) bytes
- **Directory operations** to manage lifetime of a file
  - **Create** allocates blocks to back the file
  - **Open/Close** handle to the file, typically including OS maintained current position (**cursor**)
  - **Delete** returns allocated blocks to the free list
  - **Stat** retrieves file status including existence ~ reads and returns file metadata
- **File operations** to interact with file
  - **Write** provided data at cursor location
  - **Read** data at cursor location into provided memory
  - **Truncate** clips length of file to end at current cursor value



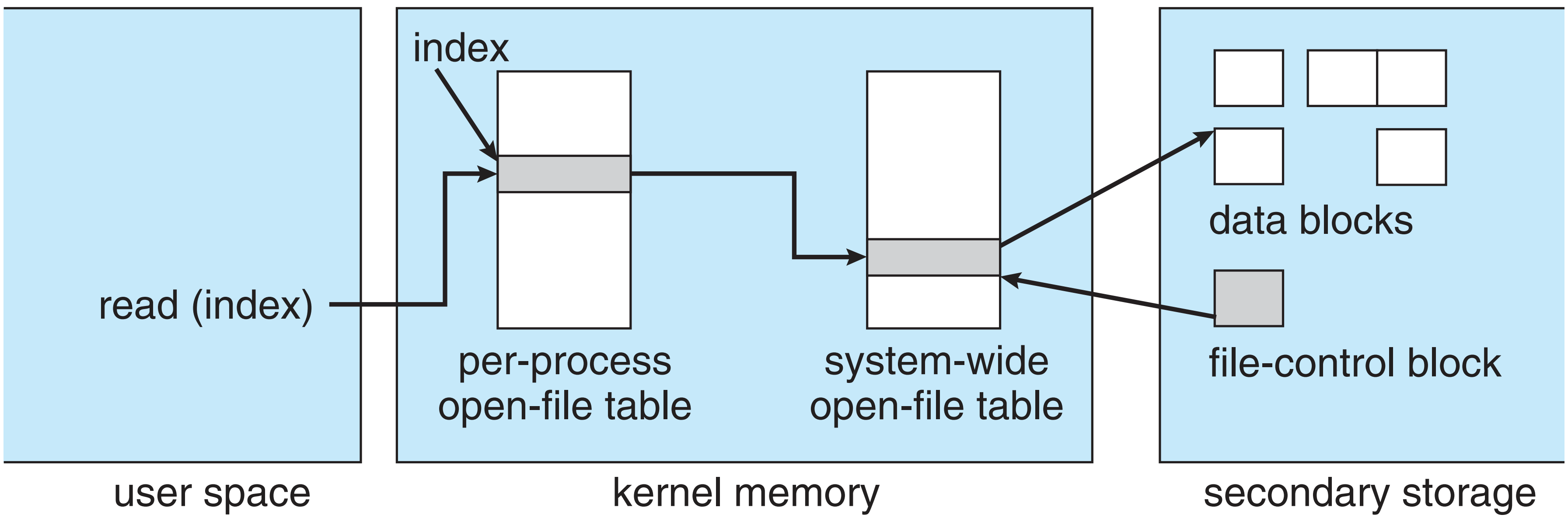
- Access pattern:
  - **Random access** permits **seek** to move cursor without reading or writing
  - **Sequential access** permits only **rewind** to move cursor back to beginning

Opening a file

- In-memory directory structure previously read from disk resolves file name to a file control block



Reading a file



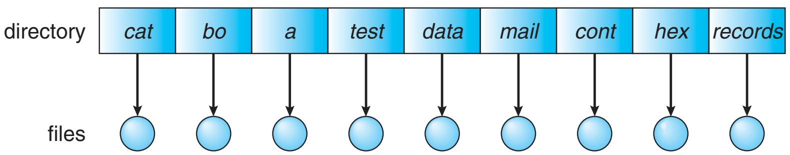
- Using per-process open-file table, index (file handle or file descriptor) resolves to system-wide open-file table containing file-control block which resolves to actual data blocks on disk

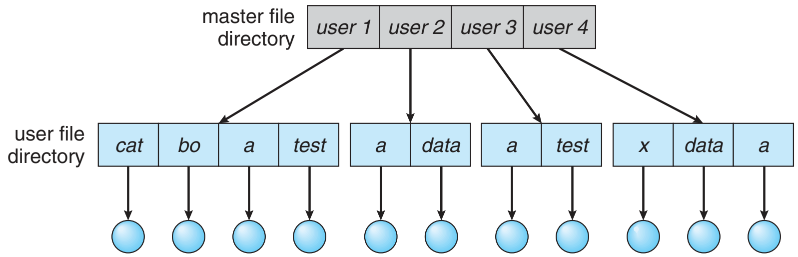
Outline

- Mass storage
- Disk scheduling
- Disk management
- Files
- Directories
  - Tree-structured
  - Acyclic-graph structured
  - File system mounting
- Other issues

Directories

- Implementations must provide
  - **Grouping**, to enable related files to be kept together
  - **Naming**, for user convenience so different files can have the same name and one file can have many names
  - **Efficiency**, to find files quickly
- **Single-level directory** is simplest
  - Naming and grouping problems though
- **Two-level directory** is next (FAT)
  - Same names for different users via paths
  - Efficient searching but no grouping





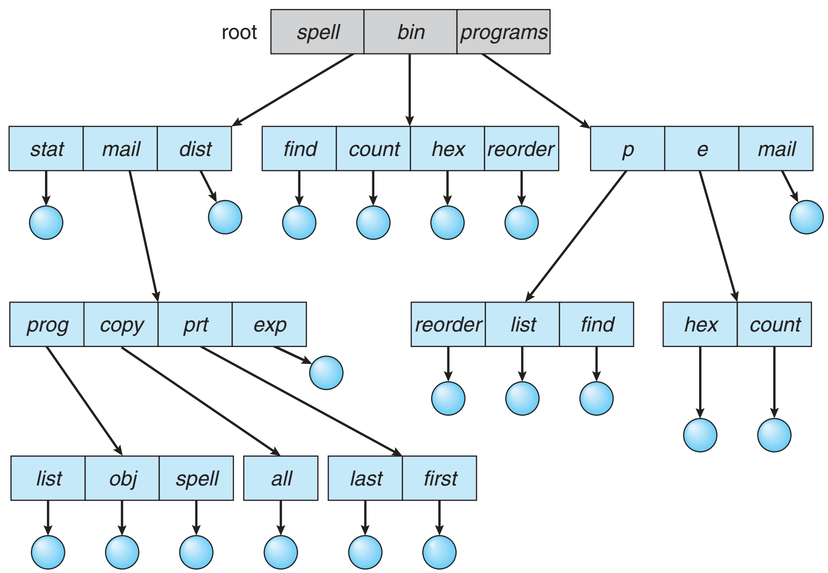
Tree-structured directories

- Provide naming convenience, efficient search, and grouping
- Introduce notion of **current working directory** (**CWD**)

  cd /spell/mail/prog

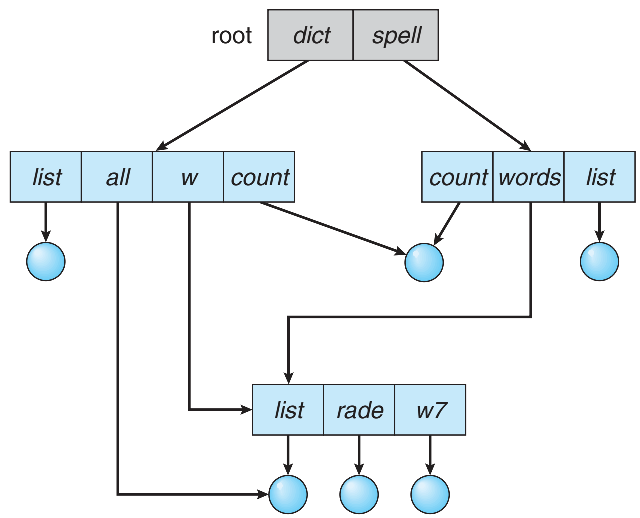
  type list

- Gives rise to **absolute** or **relative** path names
  - Name is resolved with respect to the CWD
- Other operations also typically carried out relative to CWD



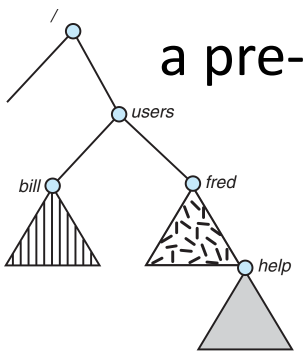
Acyclic-graph structured directories

- Generalise to a DAG so can share subdirectories and files
  - Allows files to have two different absolute names (**aliasing**)
- Need to know when to actually delete a file
  - Use back-references or reference counting
  - Compare soft- and hard-links in Unix
- Need to know how to account storage
  - Which user “owns” the storage backing the file
  - For deletion and generally for permissions
- Need to avoid creating cycles
  - Forbid links to subdirectories

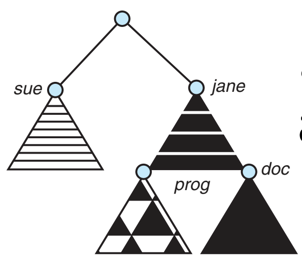


File-system mounting

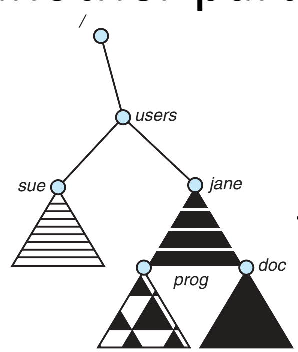
- Filesystems must be **mounted** at a **mount-point** before access
- E.g.,



a pre-existing file-system



…an unmounted filesystem in another partition



…is mounted, overlaying the *users* subdirectory

Outline

- Mass storage
- Disk scheduling
- Disk management
- Files
- Directories
- Other issues
  - Consistency
  - Efficiency
  - Buffer cache

Consistency issues

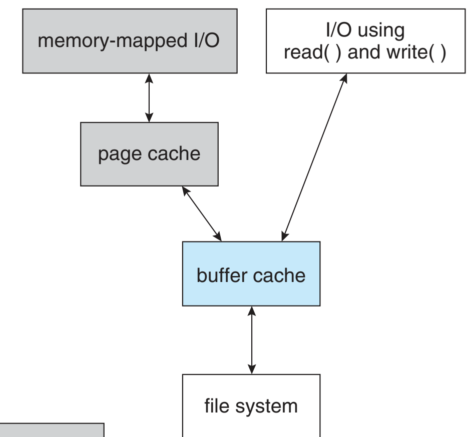
- Arise without multiple threads!
- E.g., Deleting a file uses the *unlink* system call
  - Invoked from the shell as *rm <filename>*
- Implementation must
  - Check if user has sufficient permissions on the file (write access)
  - Check if user has sufficient permissions on the directory (write access)
  - If ok, remove entry from directory
  - Decrement reference count on inode
  - If reference count is now zero, free data blocks and inode
- If the system crashes, must check the entire filesystem (*fsck*)
  - Check if any block is unreferenced, and mark free
  - Check if any block double referenced, and update reference counts

Efficiency and performance

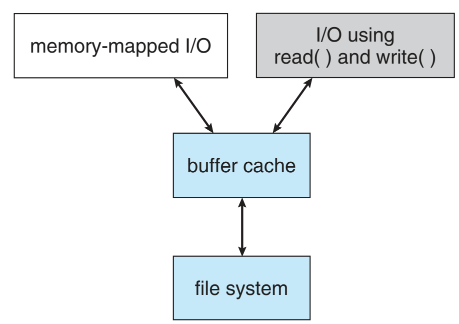
- Efficiency depends on, e.g,
  - Disk allocation and directory algorithms
    - Similar challenges to memory of allocation, fragmentation, compaction
  - Types of metadata in directory entries
    - E.g., file creation time vs last written time vs last accessed time
  - Pre-allocation or as-needed allocation of metadata structures
    - Fixed-size or varying-size data structures
- Performance measures include
  - Keep data and metadata close together
  - Create a buffer cache, a separate part of memory for often used blocks
    - Synchronous writes sometimes requested by apps or needed by OS
    - Require no buffering / caching – writes must hit the disk before acknowledgement
    - Asynchronous writes more common, can be buffered, are faster

Buffer caches

- Not unified
  - **Page cache** caches pages not disk blocks, using virtual memory techniques and addresses
  - Memory-mapped I/O uses a page cache while routine I/O through the file system uses the **buffer** (**disk**) **cache**



- Unified
  - A single **buffer cache** uses a single page cache for both memory-mapped I/O and normal disk I/O



Summary

- Mass storage
  - Hard disks
  - Solid state disks
- Disk scheduling
  - First-Come First-Served
  - Shortest Seek Time First
  - SCAN, C-SCAN
- Disk management
  - Booting from disk
- Files
  - File systems
  - File metadata
  - File and directory operations
- Directories
  - Tree-structured
  - Acyclic-graph structured
  - File system mounting
- Other issues
  - Consistency
  - Efficiency
  - Buffer cache

11. Case Study I UNIX (Linux)

9th ed: Ch. 18

10th ed: Ch. 20

Objectives

- To know a little of the history of UNIX from which Linux is derived
- To understand some principles upon which Linux's design is based
- To examine the Linux process model and lifecycle
- To describe how Linux schedules processes, provides kernel synchronization, and provides inter-process communication

Outline

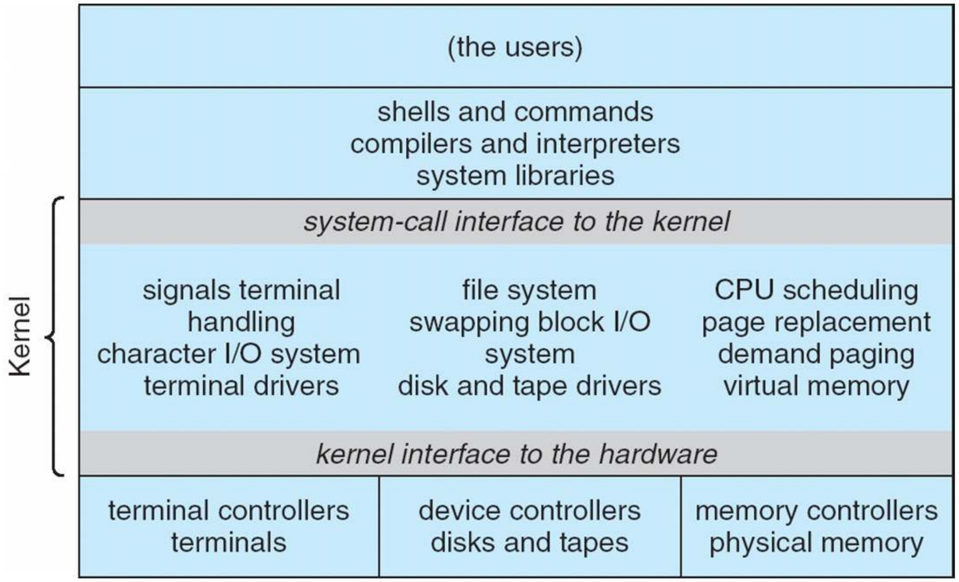
- UNIX / Linux
- Processes
- Tasks

Outline

- UNIX / Linux
  - History
  - Components
  - Kernel modules
- Processes
- Tasks

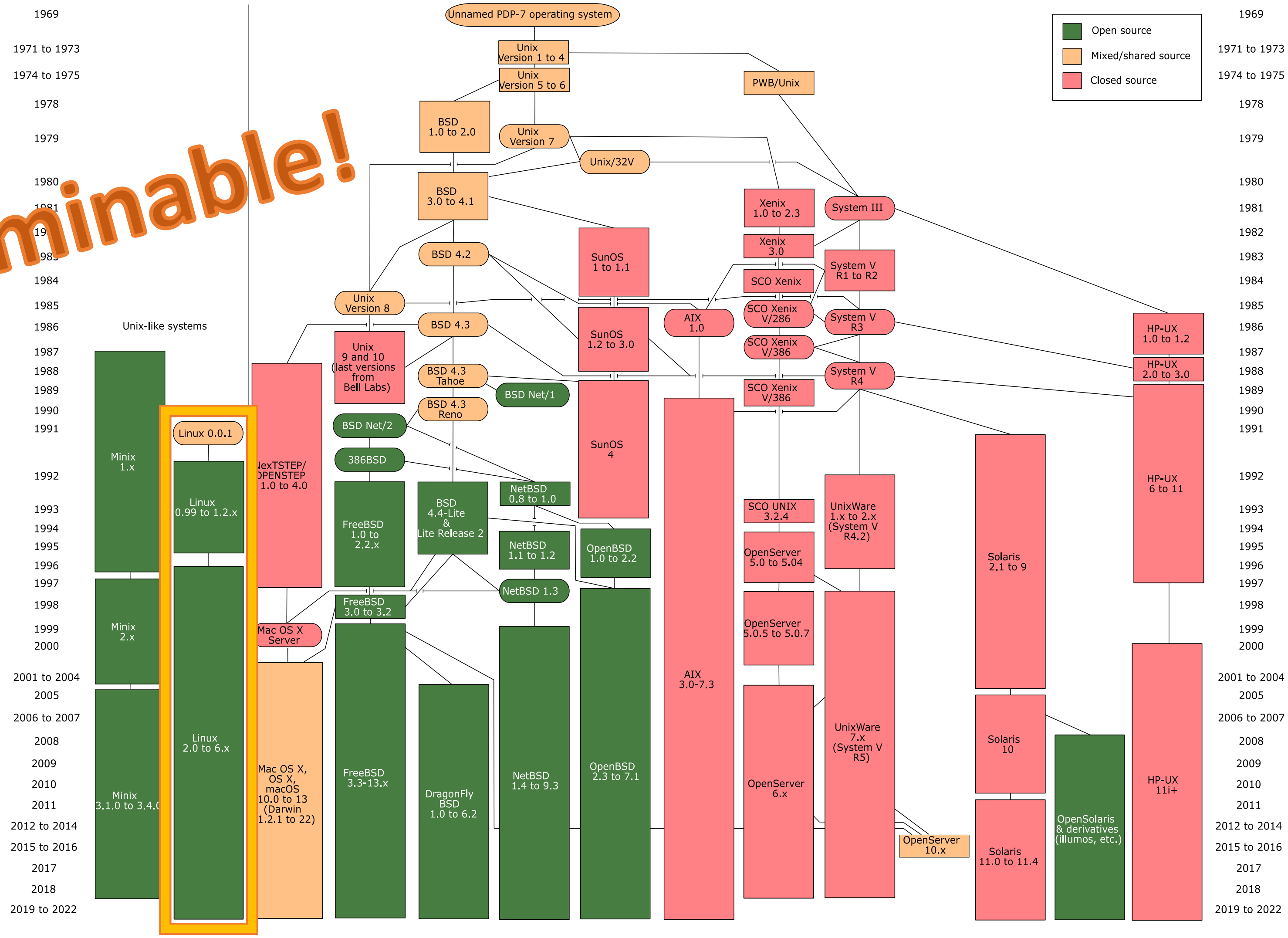
UNIX key feature

- Separation of kernel from user space
  - Only essential features inside the OS – editors, compilers etc are just applications
- Processes are the units of scheduling and protection
  - Command interpreter (shell) just another process
- All I/O looks like file operations
  - In UNIX, everything is a file



UNIX

Non-examinable!



UNIX history

- Developed in 1969 by Thompson & Ritchie at Bell Labs
  - A reaction to Multics which was rather bloated
  - Focus on (relative) ease-of-use due to e.g., interactive shell
  - In 1973 re-written from ASM to (portable) C even though performance critical
- Development continued through 1970s, 1980s
  - Notably, 1976 release of 6th edition (“V6”) included source code, so features could easily be added from other OSs
- From 1978 two main families
  - **System V** from AT&T and **BSD** from University of California at Berkeley
  - Introduction of POSIX standard, attempting to re-unify
  - Addition over time of, e.g., virtual memory, networking
  - Notably, 4.2BSD in 1983 included TCP/IP stack funded by DARPA
- Most common UNIX today is Linux

Linux history

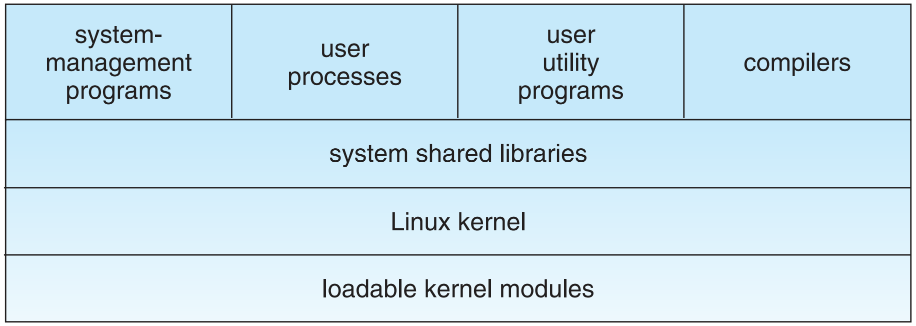
- A modern free OS based on UNIX standards
    - Originally a small self-contained kernel in 1991 by Linus Torvalds, release open-source
    - Designed for efficiency on common PC hardware but now runs on a huge range of platforms
    - Kernel entirely original but compatibility gives an entire UNIX-compatible OS, for free
    - Different distributions provide package management, support, configurations, tools, etc
    - Odd-number kernels are development kernels, even numbered are production
- Version 0.01, May 1991
    - No networking, Intel 80386-compatible processors and PC hardware only, extremely limited device-drive support, supported only the Minix file system
- Version 1.0, March 1994
    - TCP/IP plus BSD-compatible socket interface and device-driver support for IP on Ethernet
    - Enhanced file system and SCSI controller support for high-performance disk access
    - Linux 1.2, March 1995, was the final PC-only Linux kernel
- Development continues at pace

Linux design principles

- Multiuser, multitasking system with a full set of UNIX-compatible tools
  - File system adheres to traditional UNIX semantics
  - Fully implements the standard UNIX networking model
  - Designed to be POSIX compliant, achieved by at least two distributions
- Main design goals are speed, efficiency, and standardization
  - Constant tension between efficiency and security
- Supports Pthreads and a subset of POSIX real-time process control
- Linux programming interface has SVR4 UNIX semantics, not BSD

Components of a Linux system

- As most UNIX implementations, there are three main pieces
  - Most important distinction is between kernel and the rest
- The **kernel** is responsible for maintaining the important abstractions of the operating system
  - Executes in kernel mode with full access to all the physical resources of the computer
  - All kernel code and data structures share the same single address space
- **System libraries** define standard functions apps use to interact with the kernel
  - Implement much OS functionality that does not need kernel privileges
- **System utilities** perform individual specialized management tasks
  - Rich and varied user-mode programs



Kernel modules

- Sections of kernel code that can be compiled, loaded, and unloaded independently
  - Implement, e.g., device drivers, file systems, or networking protocols
  - Interface enables third parties to write and distribute non-GPL components
  - Enable a Linux system to be set up with a standard, minimal kernel, without extra device drivers compiled in
- Dynamic loading/unloading requires **conflict resolution**
  - Kernel must manage modules trying to access same hardware
  - E.g., reservation requests via kernel before granting access

Outline

- UNIX / Linux
- Processes
  - Management
  - Properties
  - Context
  - Threads
- Tasks

Process management

- UNIX process management separates the creation of processes and the running of a new program into two distinct operations.
  - The *fork* system call creates a new process before *exec* runs a new program
  - Under UNIX, a process encompasses all the information that the OS must maintain to track the context of a single execution of a single program
- Under Linux, process properties fall into three groups:
  - Identity
  - Environment
  - Context

Process properties

- Identity
  - **Process ID** (PID) uniquely identifies and is used to specify the process
  - Process **credentials** in the form of a User ID and one or more Group IDs
  - Support for emulation gives **personality** – not traditional but allows slightly modified semantics of system calls
  - **Namespace** gives specific view of file system hierarchy – typically shared but can be unique
- Environment, inherited from parent as two null-terminated vectors
  - **Argument vector** listing command-line arguments used to invoke the running program
  - **Environment vector** lists *NAME=VALUE* pairs associating named variables with arbitrary values
  - Flexible way to pass information between user-mode components, giving per-process customisation
- Context
  - The (constantly changing) state of a running program at any point in time

Process context

- Most important part is the **scheduling context**
  - Required for the scheduler to suspend and restart the process
  - Also includes accounting information about current and past resources consumed
- An array of pointers into kernel file structures called the **file table**
  - I/O system calls use indexes into this table, the **file descriptor** (**fd**)
- Separately, **file-system context** applies to requests to open new files
  - Current root and default directories for new file searches are stored here
- **Signal-handler table** defines per-process per-signal signal handling routine
- **Virtual-memory context** describes full contents of process' private address space

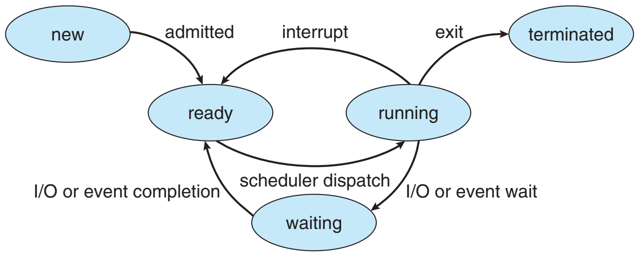
Processes and threads

- The same internal representation
  - A thread is just a new process that shares its parent's address space
- Both called **tasks** by Linux, distinguished only when created via *clone*
  - *fork* creates a new task with an entirely new task context
  - *clone* creates a new task with its own identity, but sharing parent's data structures
- *clone* gives control over exactly what is shared between two threads
  - File system, memory space, signal handlers, open files

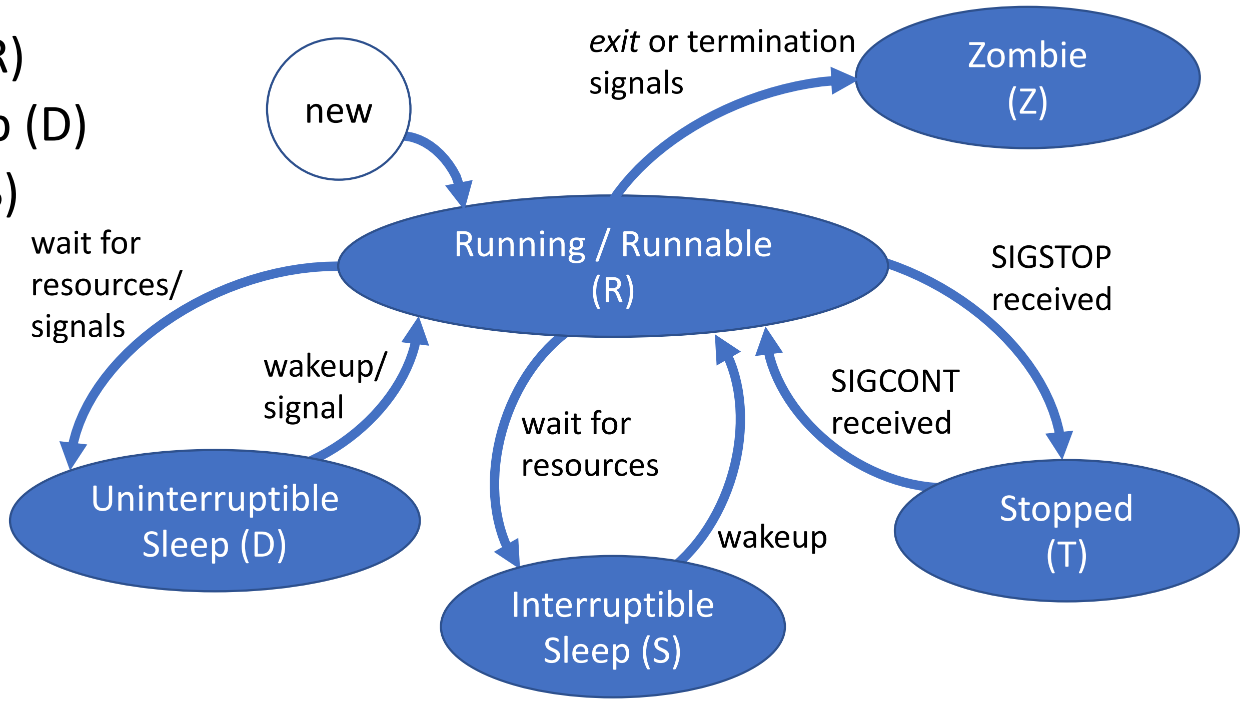
Outline

- UNIX / Linux
- Processes
- Tasks
  - Lifecycle
  - Scheduling
  - Synchronisation
  - Interrupt handlers
  - IPC

Task lifecycle



- Five states:
  - Running/Runnable (R)
  - Uninterruptible Sleep (D)
  - Interruptible Sleep (S)
  - Stopped (T)
  - Zombie (Z)



Task scheduling

- Allocation of CPU time to different tasks
  - As well as processes, in Linux this includes various kernel tasks
  - Those requested by a running process and those executed for a device driver
- Traditional UNIX scheduling uses fixed time slices and priorities to boost/penalise
  - Quantum 100ms, round-robin within priority levels
  - Priority set from process' base priority, average length of process' run queue, and *nice* value
- Worked ok for early time-sharing systems but did not scale or provide good interactive performance for current systems

Completely Fair Scheduler (CFS)

- Since 2.6.23 – no more time slices
  - Start by assuming every task should have 1/N of the CPU
  - Adjust based on **nice** value from -20 to +19: smaller is higher priority giving higher weighting
  - Run task j for a time slice $t_j \propto w_j / \sum_i w_i$
- Actual length of time given a task is the **target latency**
  - Interval during which time every runnable task should run at least once
  - E.g., target latency is 10ms, two runnable tasks of equal priority, each will run for 5ms
  - If ten runnable tasks, each runs for 1ms – but what if 1000 runnable tasks?
  - To avoid excessive switching overheads, **minimum granularity** is the minimum length of time for which a process will be scheduled
- CFS scheduler maintains per-task virtual run time in variable *vruntime*
  - Scheduler picks task with lowest *vruntime*, in default case, the same as actual run time
  - Lower priority means higher decay rate of *vruntime*
  - Implemented as red-black tree with left-most bottom-most value (lowest *vruntime*) cached

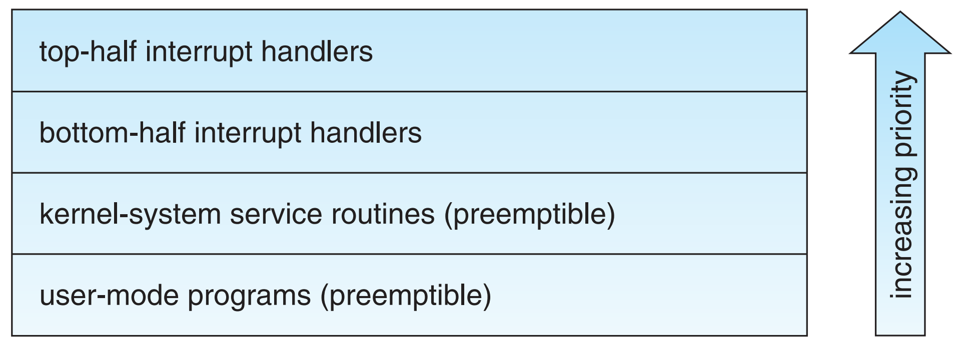
Kernel synchronisation

- Kernel-mode execution requested in two ways:
  - Process requests an OS service, explicitly via a system call or implicitly e.g. when a page fault occurs
  - A device driver delivers a hardware interrupt causing the CPU to start executing a kernel-defined handler for that interrupt
- Need guarantees that kernel's critical sections run without interruption by another critical section
  - Before 2.6, kernel code is **non-preemptible** so timer interrupt sets *need_resched*
  - After 2.6, either **spin locks** or **enable/disable pre-emption**

single processor	multiple processors
Disable kernel preemption.	Acquire spin lock.
Enable kernel preemption.	Release spin lock.

Interrupt handlers, top and bottom

- Want long critical sections to be able to run without disabling interrupts for long periods of time
- Split interrupt service routines into a **top half** and a **bottom half**
  - **Top half** is a normal interrupt service routine, run with recursive interrupts disabled
  - **Bottom half** is run, with all interrupts enabled, by a miniature scheduler that ensures bottom halves never self-interrupt



- This architecture is completed by a mechanism for disabling selected bottom halves while executing normal, foreground kernel code

Inter-Process Communication

- **Signals**
  - Process-to-process
  - Limited number, carry no information other than which signal has occurred
- **Wait queues**
  - Used inside the kernel
  - Process puts itself on wait queue for an event, and informs scheduler that it is no longer eligible for execution
  - All waiting processes are woken when the event completes
- **Pipes**
  - Just another type of *inode* in the VFS
  - Each pipe has a pair of wait queues for reader and writer to synchronise
- **Shared memory**
  - Fast but no synchronisation mechanism – need to be provided
  - Persistent object, like a small independent address space

Summary

- UNIX / Linux
  - History
  - Components
  - Kernel modules
- Processes
  - Management
  - Properties
  - Context
  - Threads
- Tasks
  - Lifecycle
  - Scheduling
  - Synchronisation
  - Interrupt handlers
  - IPC

12. Case Study II UNIX (Linux)

9th ed: Ch. 6, 18

10th ed: Ch. 5, 20

Objectives

- To examine memory management in Linux
- To explore how Linux implements file systems
- To understand how Linux manages I/O devices
- To understand how a shell works

Outline

- Physical memory
- Virtual memory
- File systems
- I/O
- Start of day

Outline

- Physical memory
  - Page allocation
  - Slab allocation
- Virtual memory
- File systems
- I/O
- Start of day

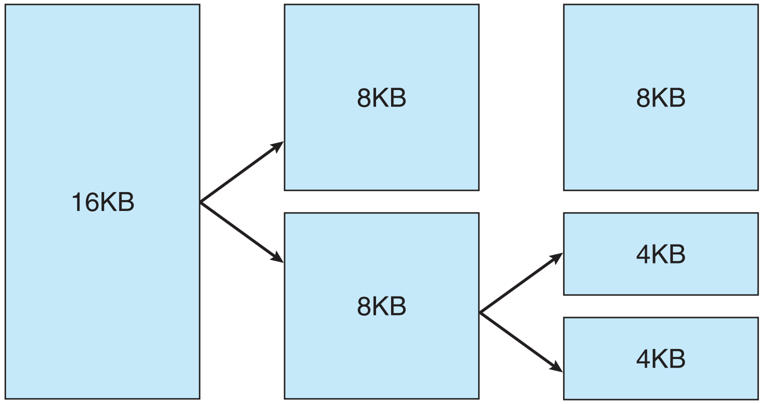
Physical memory management

- Deals with allocation/freeing of pages, groups of pages, small blocks of memory
  - Additional mechanisms for handling **virtual memory**, memory mapped into the address space of running processes
- Splits memory into zones based on hardware characteristics
  - DMA, DMA32, NORMAL, HIGHMEM
- Architecture specific; e.g., x86_32
  - Some devices only address lower 16MB, so DMA must take place there
  - HIGHMEM is memory not mapped into kernel space, all else is NORMAL
- Other systems have different constraints
  - E.g., some devices can only access first 4GB (even with 64 bit addresses)
  - x86-64 has (small) 16MB DMA zone for legacy devices, and the rest is ZONE_NORMAL

zone	physical memory
ZONE_DMA	< 16 MB
ZONE_NORMAL	16 .. 896 MB
ZONE_HIGHMEM	> 896 MB

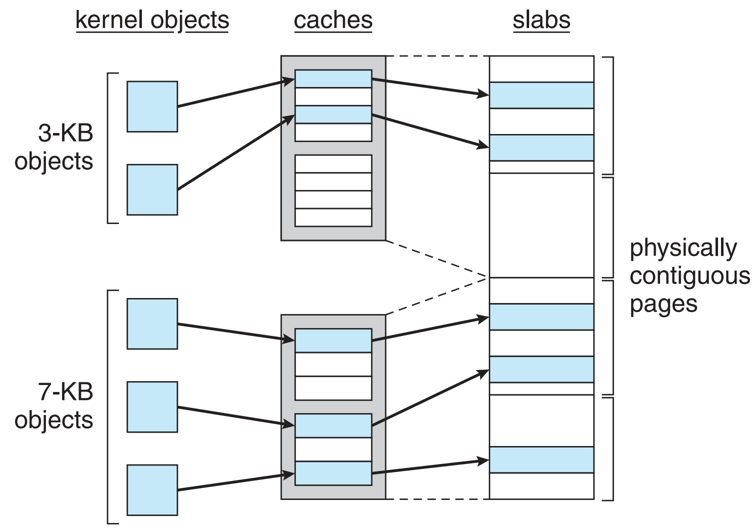
Physical page allocation

- **Page allocator** allocates and frees all physical pages
  - Can allocate ranges of physically-contiguous pages on request
- Uses a **buddy-heap algorithm** to track available physical pages
  - Each allocatable memory region is paired with an adjacent partner
  - Two allocated partner regions freed together are combined into a larger region
  - If no small free region exists to satisfy a small memory request, subdivide a larger free region into two pieces to satisfy the request



Slab allocation

- Allocation in the kernel occurs either
  - **Statically**, drivers reserve contiguous memory during system boot, or
  - **Dynamically**, via the page allocator
- Uses a **slab allocator** for kernel memory
- Using **page cache**, virtual memory system also manages physical memory
  - Kernel's main cache for files
  - Main mechanism for I/O to block devices
  - Stores entire pages of file contents for local and network file I/O



Outline

- Physical memory
- Virtual memory
  - Creation
  - Running a program
- File systems
- I/O
- Start of day

Virtual memory

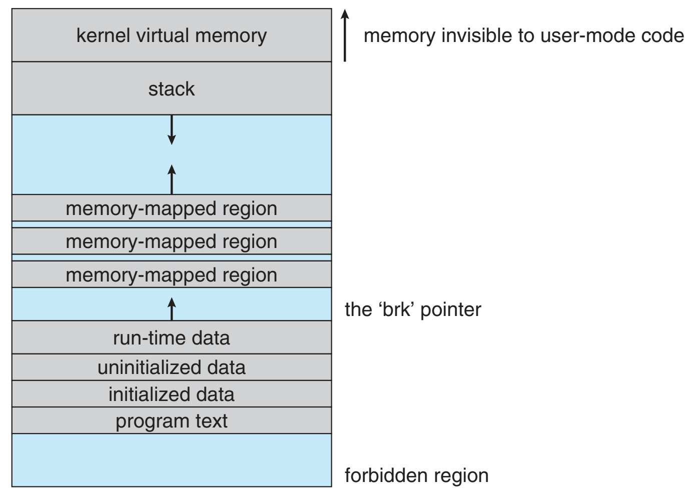
- Virtual memory system maintains each process' address space
  - Creates pages of virtual memory on demand
  - Manages loading of those pages from disk or swapping back out as required
- VM manager maintains two views of a process's address space
  - **Logical view** describes the layout of the address space, a set of non-overlapping regions, each representing a continuous, page-aligned subset of the address space
  - **Physical view** stored in the process' hardware page tables
- Virtual memory regions are characterized by
  - The **backing store**, which describes from where the pages for a region come; regions are usually backed by a file or by nothing (demand-zero memory)
  - The region's **reaction to writes**, either **page sharing** or **copy-on-write**
- **Paging system** uses **page-out policy** to decide which pages to move to and from backing store using the **paging mechanism**

Virtual memory creation

- The kernel creates a new virtual address space for two reasons
- **A process runs a new program via *exec***
  - The existing process is given a new, completely empty virtual-address space
  - Program-loading routines populate the address space with virtual-memory regions
- **A process creates a new process via *fork***
  - New process is given a complete copy of the parent's virtual address space
  - Kernel copies parent's VMA descriptors and creates a new set of page tables for the child
  - Then copies parent's page tables into the child's, incrementing the reference count of each page covered
  - Thus parent and child address spaces initially share the same physical pages of memory
- Kernel reserves a constant (architecture-dependent) area of two regions
  - **Static region** has page table references to every available physical page to ease logical-physical translation in kernel
  - Remainder is unreserved and PTEs can be pointed to any other area of memory

Running a program

- Kernel has function table for program **loading**
  - Supports multiple binary formats, commonly ELF
- ELF-format program has a header plus several page-aligned sections
  - Pages initially mapped into virtual memory, and then faulted in to physical memory
  - ELF loader reads header and maps sections of the file into separate VM regions
- Unless **statically** linked there will be symbols defined elsewhere
  - Calling dynamic linker stubs trigger mapping of the link library into memory, resolving references
  - Shared libraries typically compiled to **position-independent code** (**PIC**) so can be loaded anywhere



Outline

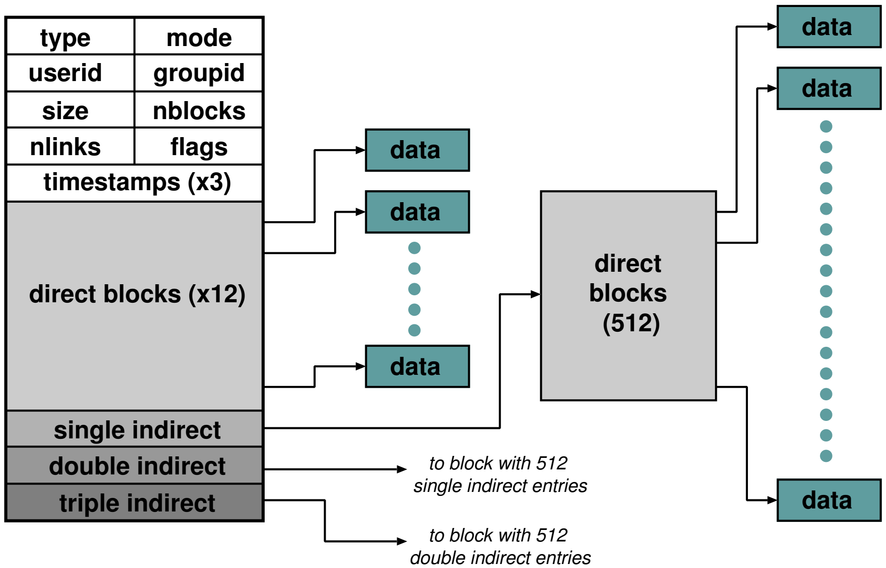
- Physical memory
- Virtual memory
- File systems
    - Implementation
    - Directories and links
    - Access control
- I/O
- Start of day

File systems

- To the user, Linux's file system appears as a hierarchical directory tree obeying UNIX semantics
  - Devices are represented by special files
  - **proc file system** doesn't store data but computes it on demand using inode number to identify the operation
- Kernel hides details, managing different file systems via the **virtual file system** (**VFS**), an abstraction layer with four components
  - The **inode object** structure represent an individual file
  - The **file object** represents an open file
  - The **superblock object** represents an entire file system
  - A **dentry object** represents an individual directory entry
- Then manipulate those objects via a set of operations on the objects, e.g., for files include
  - int (***open**) (struct inode *, struct file *);
  - ssize_t (***read**) (struct file *, char __user *, size_t, loff_t *);
  - ssize_t (***write**) (struct file *, const char __user *, size_t, loff_t *);
  - int (***mmap**) (struct file *, struct vm_area_struct *);

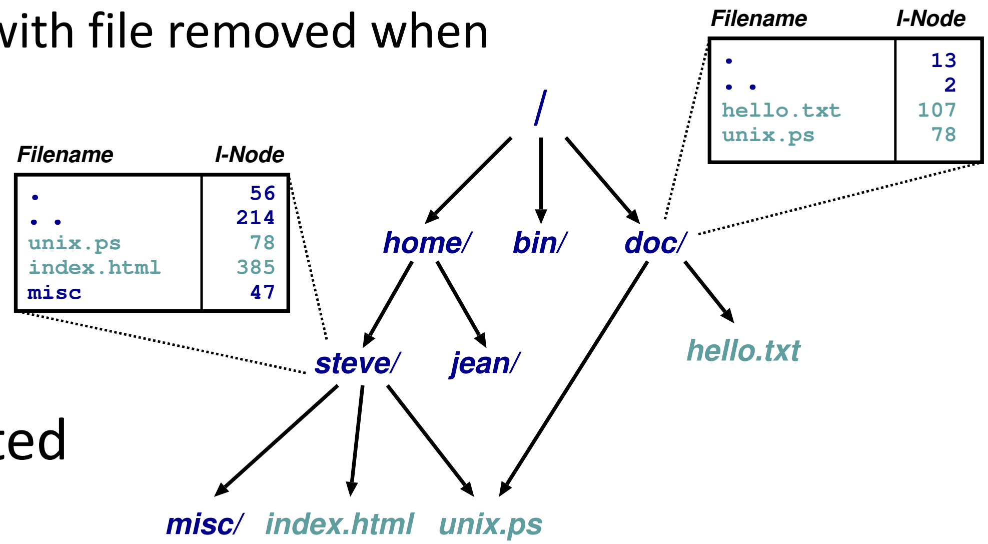
File system implementation

- UNIX file systems use **inodes** (index nodes) as FCBs
  - A **combined scheme**: the inode contains pointers to blocks, and pointers to pointers to blocks, and so on
- Alternatives include **linked schemes** where an index block points to blocks and ends with either a *null* or a pointer to the next index block



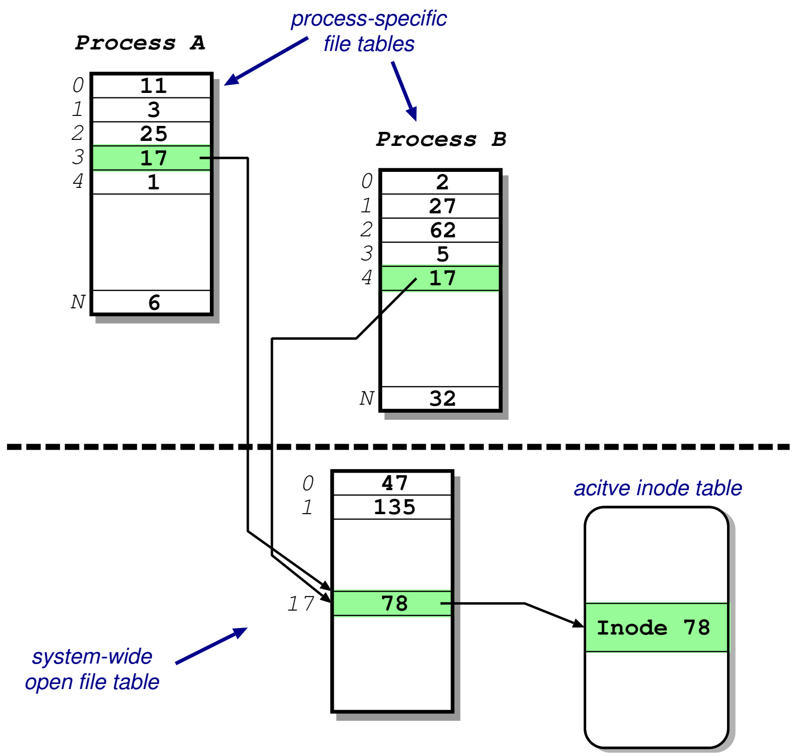
Directories and links

- Directory is just a file, itself pointed to by an inode, mapping **filenames** to **inodes**
- An instance of a file in a directory is a **hardlink**
  - Reference counted in the inode with file removed when reference count becomes zero
  - Directories cannot have more than one hardlink otherwise cycles might be created
- Alternatively, a **softlink** or **symbolic-link** is a normal file containing a filename, interpreted by the filesystem



In-memory tables

- Each process sees files as **file descriptors**
    - Index into a process-specific **open file table**
- Table entries point into a **system-wide open file table**
    - Multiple processes might operate on the same file, including deleting it
- System-wide table entries then point to in-memory **inode table**

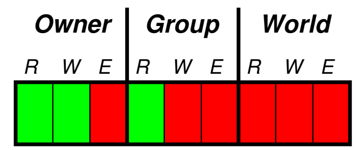


Access control

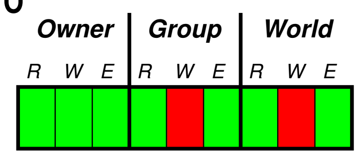
- Every object uses same mechanism: unique numeric identifiers
  - **User ID** (UID) identifies single user (set of rights)
  - **Group ID** (GID) identifies a group (rights held by one or more users)
- Processes have a single UID but one or more GIDs
  - Process UID matches object UID, then process has **user/owner rights**
  - Else if a process GID matches an object GID, then process has **group rights**
  - Else process has **world rights**
- Object has **protection mask** indicating R/W/X for user/group/world
  - Root UID process has automatic rights to everything
- Rights can be passed by forwarding fds down a local network socket
  - E.g., Print server is passed a descriptor for the file to be printed, avoiding the need for it to have rights to read any other of the user's files

File access control

- Access control information held in each inode
  - Three bits for each of owner, group and world
  - For files, read, write execute
  - For directories, read entry, write entry, traverse directory



= 0640



= 0755

- Also have *setuid* and *setgid* bits:
  - Normally processes inherit permissions of invoking user
  - *setuid*/*setgid* allow user to "become" someone else when running a given program
- E.g. an assessment application might have
  - A *sit-exam* application owned by the examiner with permissions 0711 plus *setuid*
  - A *test-scores* file also owned by the examiner but with permissions 0600

Outline

- Physical memory
- Virtual memory
- File systems
- I/O
  - Buffer cache
  - Device types
- Start of day

Input/Output

- Device-oriented file system accesses disk storage via two caches:
  - The **page cache** caches data, unified with the virtual memory system
  - The **buffer cache** caches metadata separately, indexed by physical disk block
- Three classes of device:
  - **Block devices** allow random access to independent, fixed size blocks of data
  - **Character devices** include most other devices, not needing the functionality of regular files
  - **Network devices** are interfaced via the kernel's networking subsystem

Buffer cache

- Maintain copies of some parts of disk in memory for speed
- Reading then involves
  - Locate relevant blocks from inode
  - Check if in buffer cache
  - If not, read from disk into buffer cache memory
  - Return data from buffer cache
- Writing is the same except final step updates the version **in the cache**
  - “Typically” prevents majority (around 85%) of implied disk transfers
  - But at risk of losing data while the update is only in the buffer cache
- Must periodically (30 seconds) flush dirty buffers to disk
  - Can cache metadata too but what problems can that cause?

Device types

- **Block devices** provide the main interface to system's disk devices
  - Block buffer cache acts as a pool of buffers for active I/O and as a cache for completed I/O
  - Request manager handles reading/writing of buffer contents to/from block device driver using **Completely Fair Queueing** (**CFQ**)
- **Character devices** do not offer random access, with driver just passing on request directly
  - Main exception are **terminal devices** where **line discipline** is responsible for interpreting information from device
  - Eg., **tty discipline** glues *stdin*/*stdout* onto terminal data/output streams
- **Network structure** complex with socket interface, protocol drivers, network device drivers
  - Also firewall management, filtering, marking etc

Outline

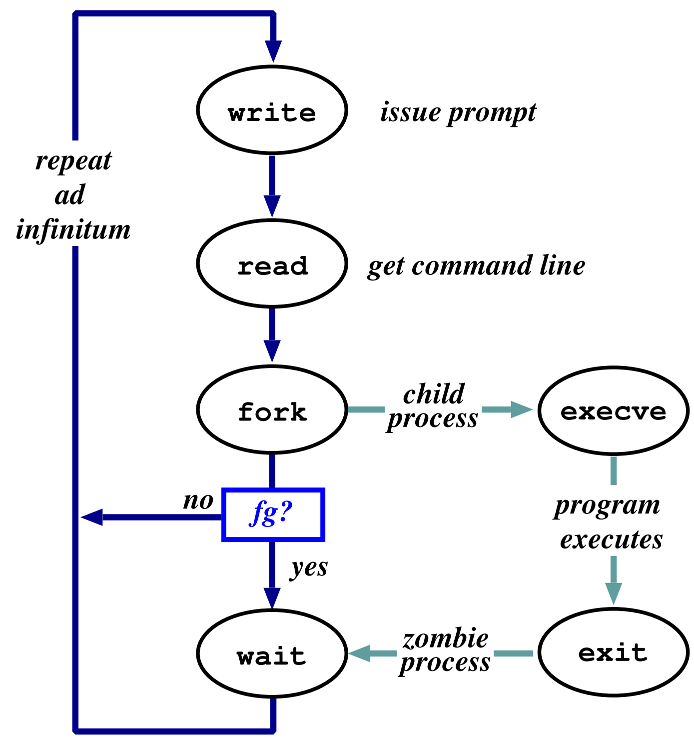
- Physical memory
- Virtual memory
- File systems
- I/O
- Start of day
    - Shell operation
    - Standard I/O

UNIX start of day

- Kernel (*/vmunix*) loaded from disk and executed, mounting root filesystem
    - Bootloader required to read from the disk
    - First process (PID=1), traditionally */etc/init*, is hand-crafted
- Proceeds by reading */etc/inittab* and, for each entry:
    - Opens terminal special file, e.g. */dev/tty0*, duplicates the resulting fd twice, and forks an */etc/tty* process
- Each tty process then:
    - Initialises the terminal, outputs the string **login:** & waits for input
    - On receiving input, *execve /bin/login*
- */bin/login* then
    - Outputs the string **password:** & waits for input
    - On receiving input, hash it and check against entyr in */etc/passwd*
    - If match, set the UID & GID, and *execve* the indicated shell
- When the shell exits, the parent *init* resurrects the */etc/tty* process which goes again

Shell operation

- Just another process – needn't understand commands, just files
  - Using CWD avoids need for fully qualified pathnames
- Command line parsing can be complex
  - Wildcard expansion (**globbing**)
  - Tilde (~) processing
  - Conventionally trailing & backgrounds forked process



Standard I/O

- Every process has three fds on creation:
  - **stdin** from which to read input
  - **stdout** to which output is sent
  - **stderr** to which diagnostics are sent
- Inherited from parent but can be **redirected** to/from a file, e.g.,

  ls >listing.txt ls >&listing.txt sh <commands.sh
- Consider: *ls >temp.txt; wc <temp.txt >results*
  - Pipeline is better, e.g. *ls | wc >results*
- Unix command lines can become very complex e.g., with many filters
  - Redirection can cause some buffering subtleties

Summary

- Physical memory
  - Page allocation
  - Slab allocation
- Virtual memory
  - Creation
  - Running a program
- File systems
  - Implementation
  - Directories and links
  - Access control
- I/O
  - Buffer cache
  - Device types
- Start of day
  - Shell operation
  - Standard I/O