

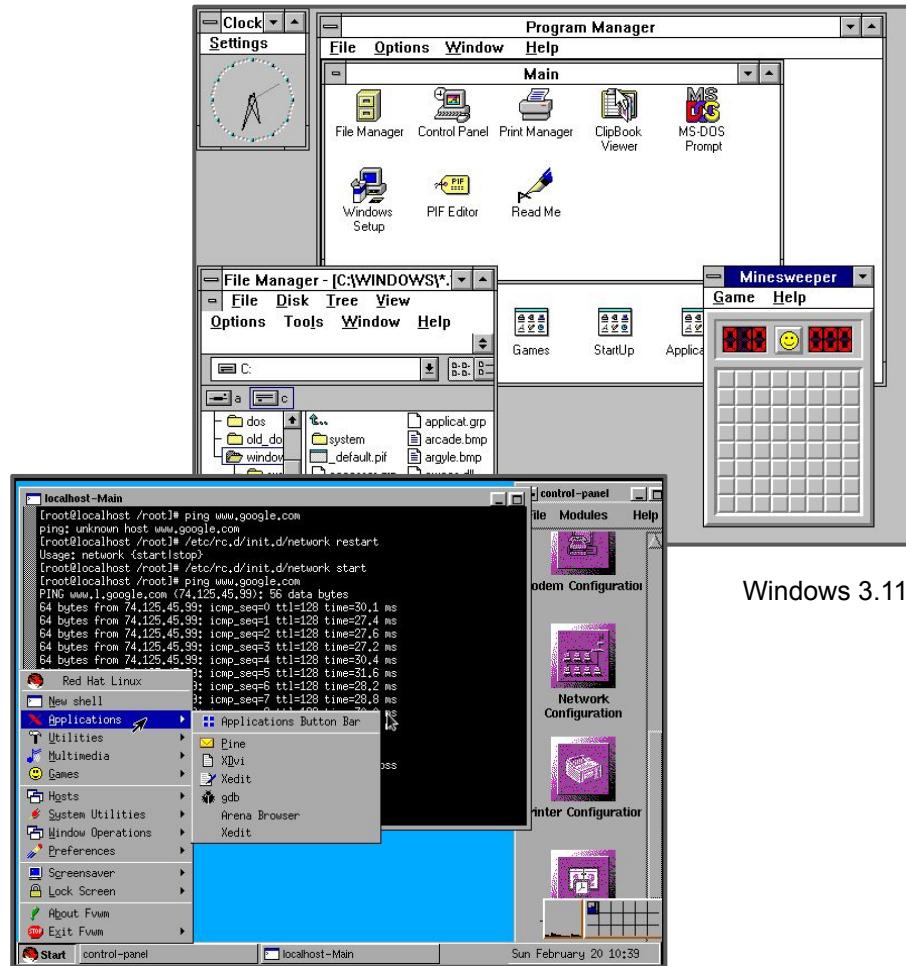
Operating Systems

CS-C3140, Lecture 1

Alexandru Paler

Contents

- Introduction to operating systems
- Practical arrangements of the course
 - Material (what to study)
 - Slides, Readings, Textbook, ...
 - Requirements (how to pass)
 - Exercises and exam
- Introduction to computer systems
 - Basics on computers
 - Processors, memory
 - Larger systems



Windows 3.11

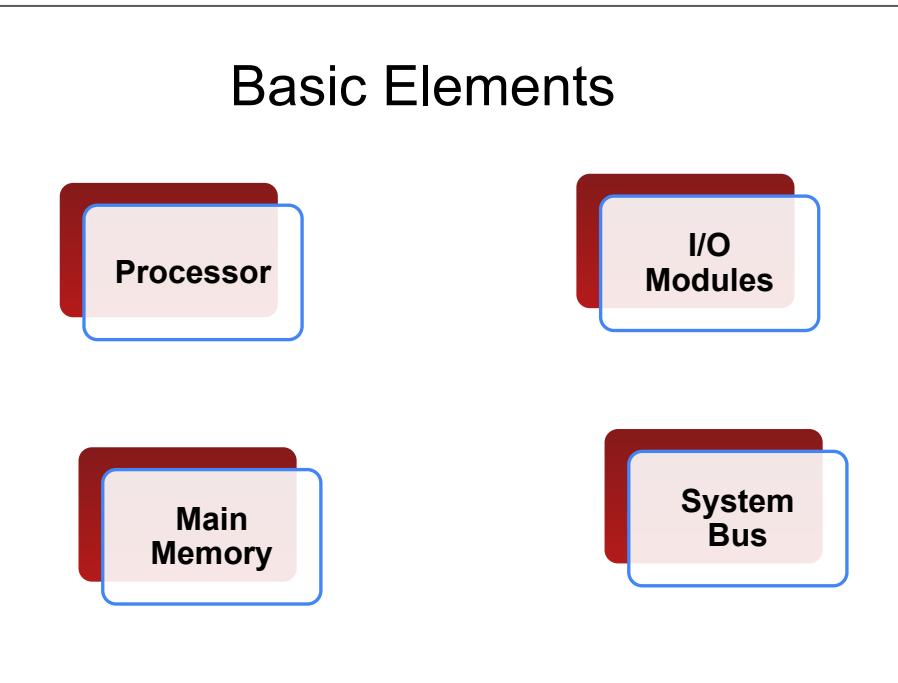
Learning objectives: ... understand the principles of operating system implementation

- Objectives
 - Systems in general (what do we "operate")
 - Software system structures
 - Processing, memory systems, storage systems
 - Concurrency and parallelism
 - Virtualization and distributed systems
- For CS students this is fundamental knowledge
 - For many others, this is essential
 - So, the course goes much beyond traditional issues

On systems and operating them ...

Operating System

- Exploits the **hardware resources** of one or more processors
- Provides a set of services to system **users**
- Manages secondary memory and I/O devices

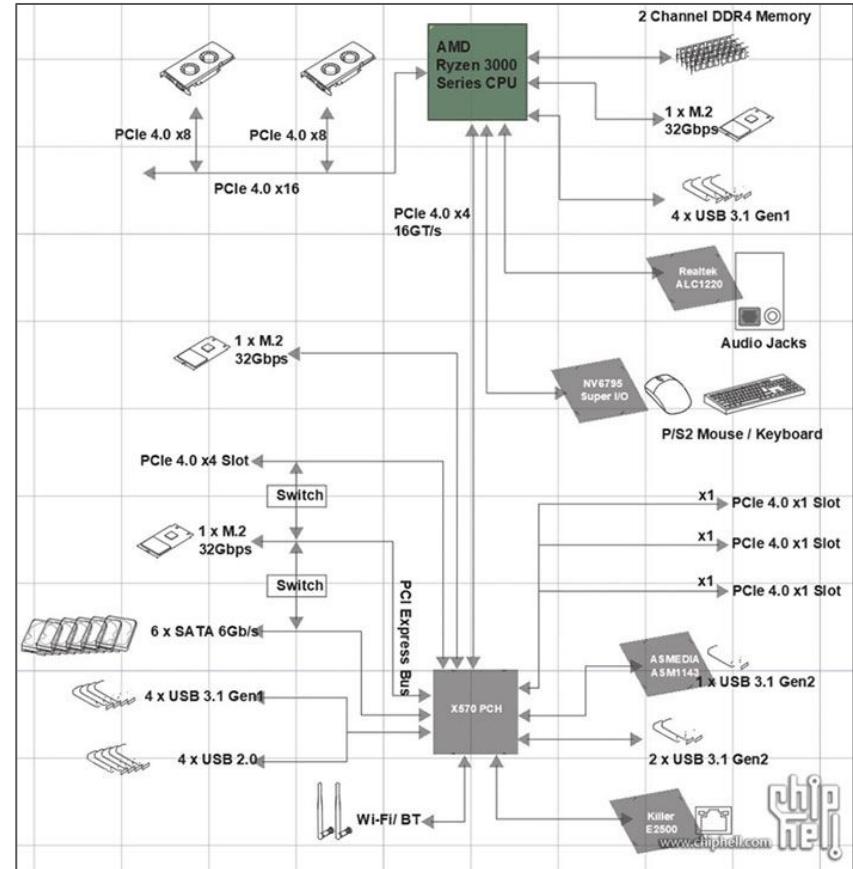


Operating Systems – What are “systems”?

- “A set of **interrelated components** working **together** toward some **common objective**”

- Structures
 - Delimitations
 - Subsystems / Components
- The services
 - Inputs and outputs
 - Interaction with environment

- In practice
 - Requirements, Interfaces between subsystems, Component properties
 - Not all “things” are systems



Operating Systems – What is “operating”?

- Systems are used in a shared way
 - Multiple users
 - Not necessarily “the end users”.
 - Tasks, processes, etc. that use the system resources
 - Multiple components (or “subsystems”)
 - System resources that are used in a shared way
- Typically, the systems are complex
 - Abstraction of the resources are needed
- Sharing resources is complex
 - Management of the shared resources is needed
 - In systems, the resources are typically inter-dependent
- But security, dependability, etc. is also needed...



1960's – IBM System/360

Course arrangements

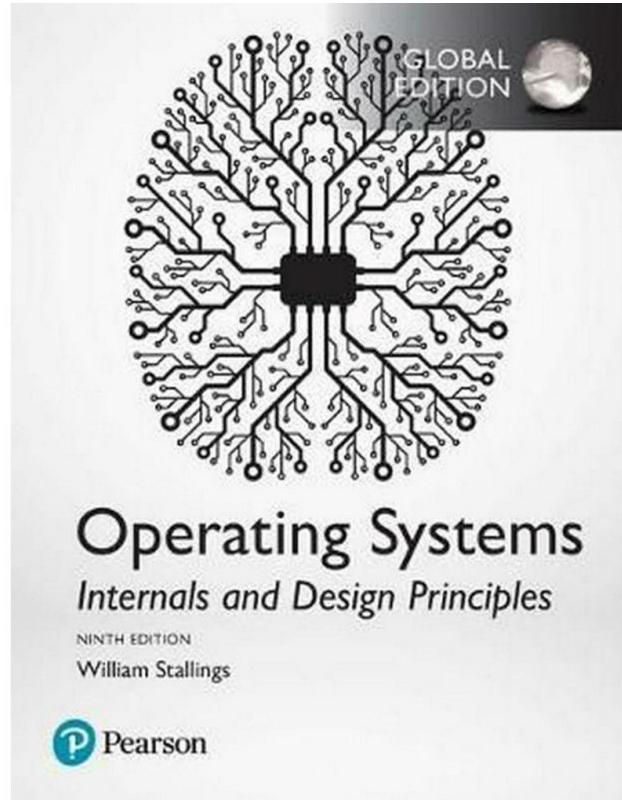
Nature of the course

- This is a standard course
 - A core course for SW system students
 - Similar course in any CS curricula
- Approach
 - The course is based on a textbook => exam
 - Note: the course is a “extended subset” of the book
 - The world is rapidly changing – and OSes are also
 - Exercises are supporting the basic contents
- Connected to CS history
 - Understand what is in a typical system
 - historical reasons



Material and Requirements

- Materials
 - William Stallings: Operating Systems Internals and Design Principles (any recent edition)
 - Check the material & weekly pages on MyCourses
- Requirements
 - Exercises (mandatory)
 - Testing your understanding and skills
 - Final exam (mandatory)
 - Testing your overall knowledge on the course topics



Arrangements

- Staff
 - *Alexandru Paler, Ioana Moflic, Sneha Saj, Alex Illov*
 - Discord server: <https://discord.gg/4rH8juDStu>
- Lectures
 - A broad view on operating system concepts
 - Weekly: Tuesday 8-10, T1 in CS building (this room)
 - Outline slides will appear on MyCourses
- Exercises
 - We will use A+ (instructions & readings there, published per round)
 - online each Friday at 22:00 Helsinki time
 - one week deadline: 100% points
 - one week extension (two weeks): 70% points
 - Weekly: Q&A session – Tuesday 10-12 (starting next week)
 - Weekly: Thursday 10-12 (starting next week)
 - Weekly: Friday 10-12 (starting next week)
 - The exercises are more on the practice (than the exam & lectures)

Grading

- The grade is based on points
 - The exercises: 8 (?) rounds and 20 points/round (= max. 160 total)
 - The exam points are scaled to max. 160 points total
- To pass the course you have to pass both the exam and the exercises
 - A minimum number of points for the exam (to be announced separately)
 - Per round limits for the exercises (you have to pass 6/8 rounds)

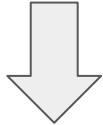
Introduction to computer architecture

Computers and operating systems

- A computer is a device consisting of
 - CPU
 - Memory
 - I/O peripherals: disk, display, network card
- A computer is executing programs
 - Perform arithmetic or logical computations
 - Have control
 - Do input and output (I/O)
- An operating system is a special program
 - Controls access to computer peripherals
 - Enables other programs to run
- How many different operating systems have you used?



Processor and Main Memory



Controls the operation of the computer

Performs the data processing functions

Referred to as the
Central Processing Unit (CPU)



- Stores data and programs
- Typically volatile
- Contents of the memory is lost when the computer is shut down
- Referred to as real memory or primary memory

I/O Modules and System Bus

Move data between the computer and its external environment

Secondary memory devices
(e.g. disks)

Communications equipment

Terminals

Provides for communication among processors, main memory, and I/O modules

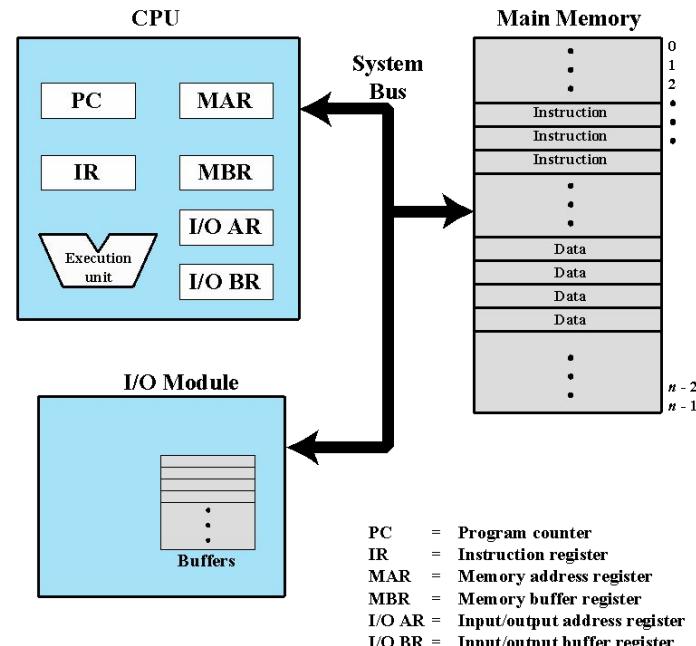
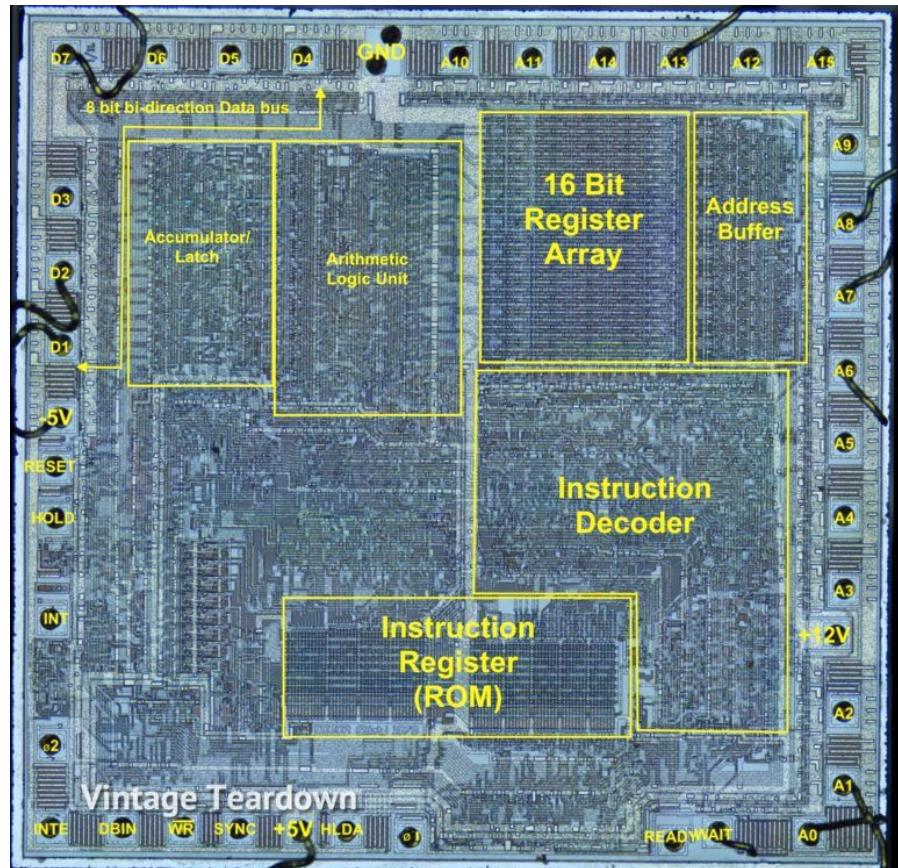


Figure 1.1 Computer Components: Top-Level View

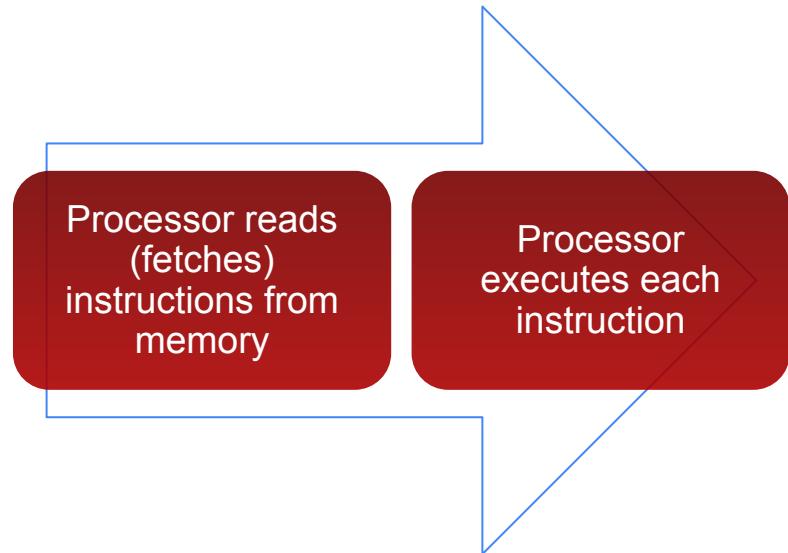
Processor Registers

- User-visible registers
 - available to all user programs or applications
 - data registers and address registers
 - some architectures save/restore all user-visible registers and others require user to do it
- Control and status registers
 - control the operation of the processor
 - usually visible only to the operating system
- Program Counter (PC): the address of the next instruction to execute
- Instruction register (IR): the mostly recently fetched instruction
- Program Status Word (PSW): condition codes, status information, interrupt enable/disable and user/kernel mode flags
- HW can be designed in a way that the OS support (e.g., memory protection/switching between tasks) can be implemented with less effort



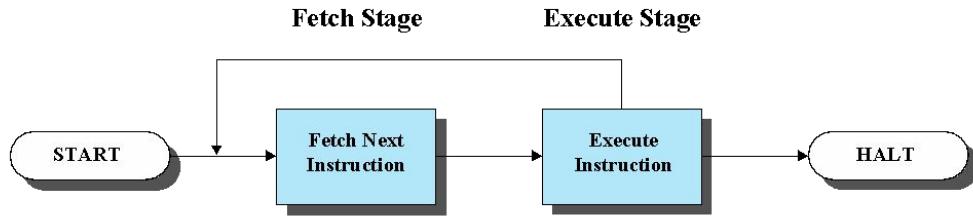
Instruction execution

- A program consists of a set of instructions
- A processor executes instructions in two-stage manner
 1. **Instruction fetch** from the memory
 2. **Instruction execution** which involves various CPU operations
- Program terminates
 - If an explicit instruction for that is issued
 - An unrecoverable error occurs
- Fetched instructions are loaded into Instruction Register IR and can perform
 - Processor-memory data transfer
 - Processor-I/O data transfer
 - Arithmetic or logic operation
 - Changing the program control (jumping to another address)



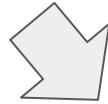
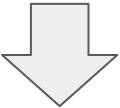
Instruction Fetch and Execute

- CPU fetches an instruction from memory
 - Program counter (PC) holds the address of the next instruction to be fetched
 - PC is incremented after each fetch
- Fetched instruction is loaded into Instruction Register (IR)
- Processor interprets the instruction and performs required action:
 - Processor-memory
 - Processor-I/O
 - Data processing
 - Control



Control Stack and Interrupts

Control stack and Interrupts



- Programs use memory to operate
 - Memory is usually divided into different areas
 - Used in a different way for different purposes
 - Text (program code)
 - Data (often there is a heap)
 - Stack (especially control, but also for data)
- The stack stores frames (data of a function call)
 - Frame pointer (FP) indicates the current frame
 - Access to data
 - Frames are often linked
 - Stack pointer (SP) indicates the top of the stack
 - In addition to call frames, there can be temporaries, etc.
- A method with which a peripheral (memory, disk, network card) can interrupt the CPU execution
 - Improves the processor utilization
 - I/O devices usually much slower than CPU
 - e.g. **accessing hard disk is several orders of magnitude slower than executing a CPU instruction**
 - without interrupts processor would have to wait until the device catches up
- With interrupts processor can execute something else while the I/O is being performed
 - when the external device becomes ready, it sends an interrupt request
 - CPU transfers control to an interrupt handler

Interrupts as events

- The terms exception, interrupt and trap are often used somewhat interchangeably
 - Note that there both HW and SW involved
 - Try not to get confused...
- Exception
 - An exception is a condition that occurs within the processor itself (e.g., division by zero)
- Interrupt
 - An interrupt is a signal from either an external HW
- Trap
 - Basically a software interrupt

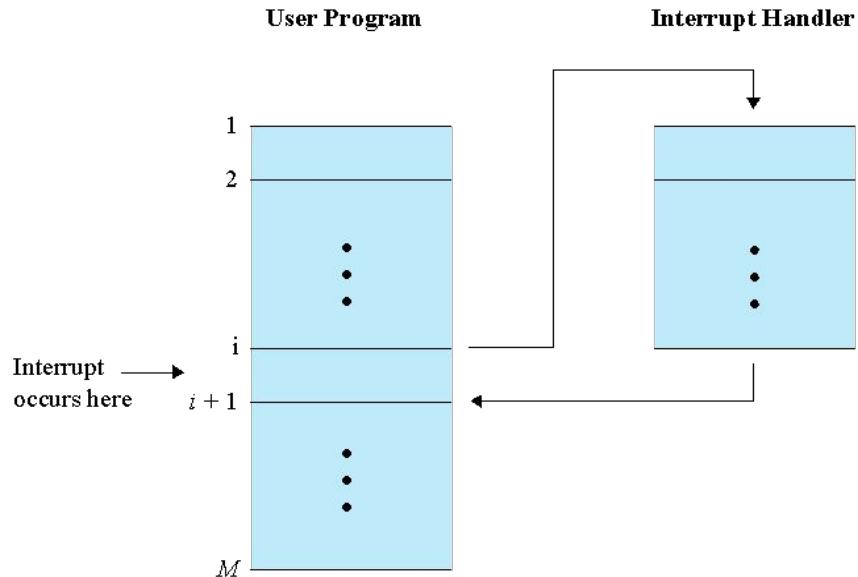


Figure 1.6 Transfer of Control via Interrupts

Multiple interrupts

- More than one device can generate interrupts simultaneously
 - E.g. printer and disk can complete their tasks at the same time
 - Disabling interrupts
 - CPU ignores further interrupts when in interrupt handler
 - Does not take into account relative priority or time-critical needs
- Prioritized interrupts
 - higher priority interrupt can interrupt lower priority interrupt handler
 - lower priority interrupt handler resumes once the higher interrupt has been dealt

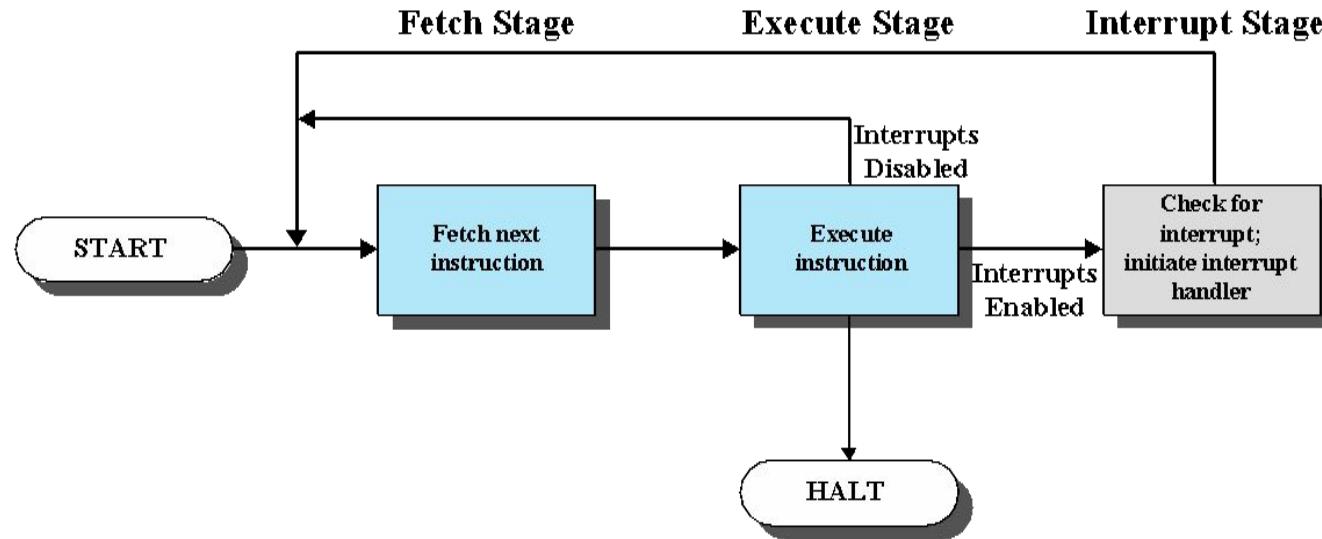


Figure 1.7 Instruction Cycle with Interrupts

Memory

Memory systems

- Trade-off between capacity, access time and cost
 - faster access time, greater cost per bit
 - greater capacity, smaller cost per bit
 - greater capacity, slower access speed
- The solution is to have a memory hierarchy
- When one goes down the hierarchy
 1. per bit cost decreases
 2. capacity increases
 3. access time increases
 4. frequency of access to the memory decreases
- Claims 1–3 apply due to our hardware
- Claim 4 is valid due to the *locality of reference*
 - the memory references of a program tend to cluster
 - applies to both code and data

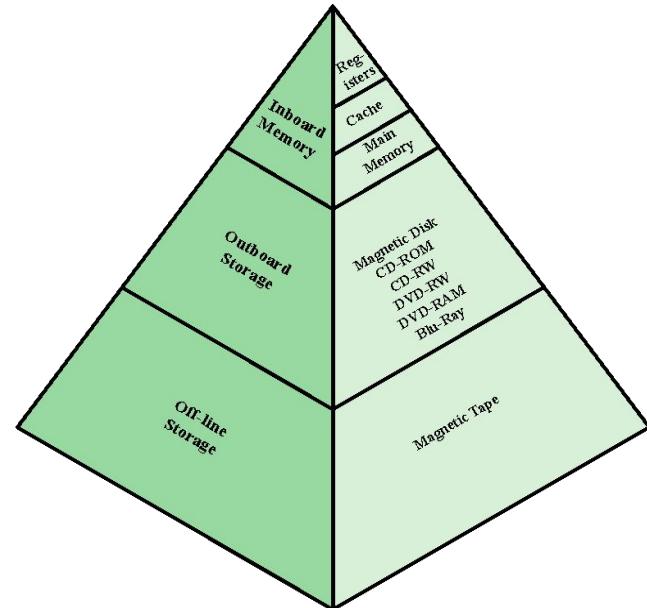
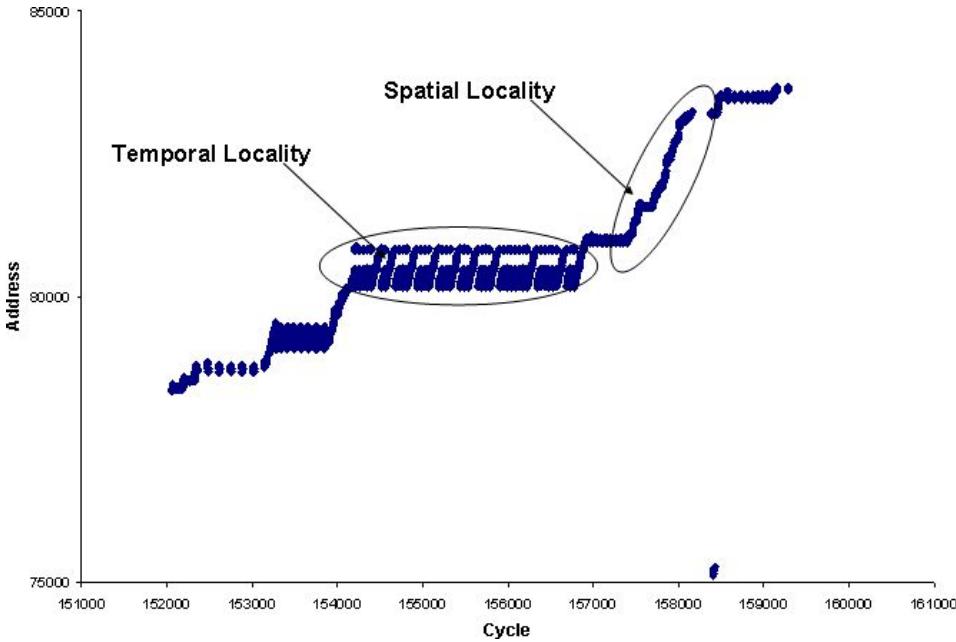


Figure 1.14 The Memory Hierarchy

Principle of Locality

- Memory references by the processor tend to cluster
- Data is organized so that the percentage of accesses to each successively lower level is substantially less than that of the level above
- Can be applied across more than two levels of memory



<https://stackoverflow.com/questions/7638932/what-is-locality-of-reference>

Memory hierarchy

- Typically there are several levels
 - Data path itself (latches, etc., not visible to programmer)
 - Registers (visible to the programmer/compiler)
 - Cache (not visible to the programmer/compiler, but OS handles)
 - L1 (often zero-wait, usually per core)
 - Often separate for code and data
 - L2 (shared)
 - L3 (can be found off-chip)
 - Main memory (usually DRAM)
 - Disk memory
- Differences in sizes and speeds are significant
- Note that the same data can reside in several places

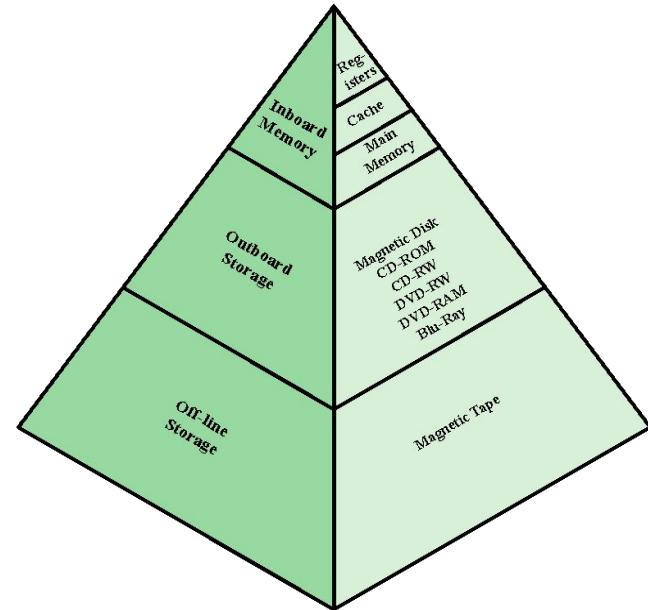
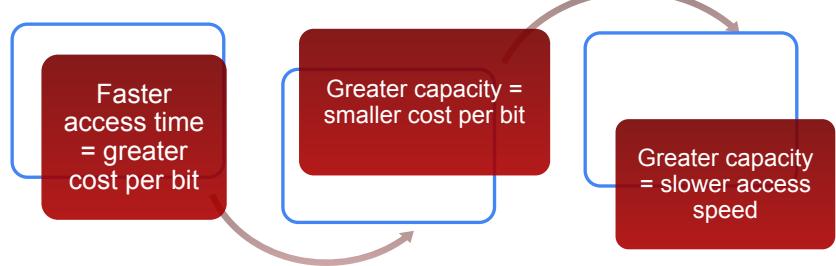


Figure 1.14 The Memory Hierarchy

Memory Relationships



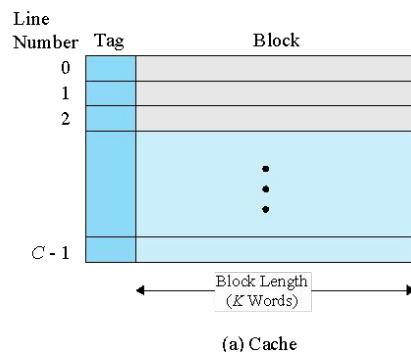
Secondary Memory

Also referred to as auxiliary memory

- External
- Nonvolatile
- Used to store program and data files

Basic cache operation

- Caches are small memories
 - Close to core
 - Invisible to the programmer
 - Storing frequently used data
- Cache organization
 - **Cache lines (small blocks, e.g., 64 bytes)**
 - Cache sets (direct-mapped are common, full assoc. rare)
- Associate memory operation
 - In addition to data, coding of the address
 - **Tag** and index (block bits not needed)
 - Bits for checks, e.g., validity and modifications (dirty bit)
 - Tags are the memory addresses
 - **Cache hits and misses**



(a) Cache

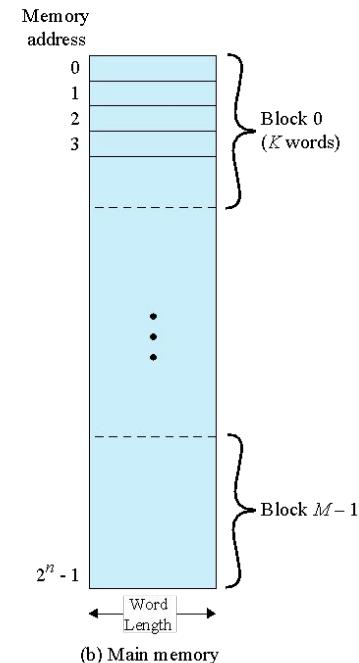
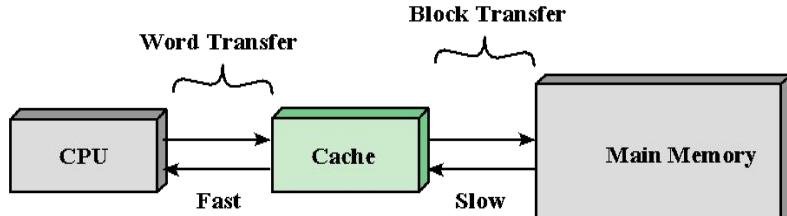


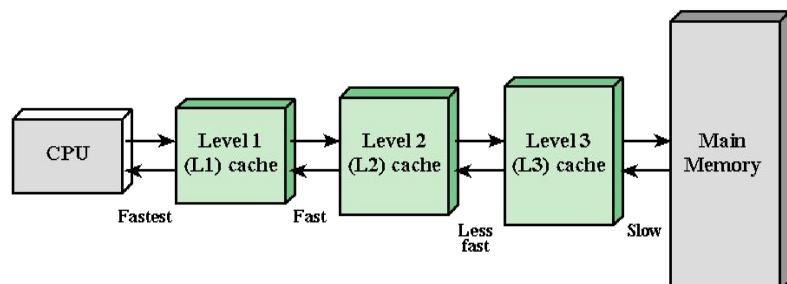
Figure 1.17 Cache/Main-Memory Structure

Cache Memory

- Invisible to the OS
- Interacts with other memory management hardware
- Processor must access memory at least once per instruction cycle
- Processor execution is limited by memory cycle time
- Exploit the principle of locality with a small, fast memory



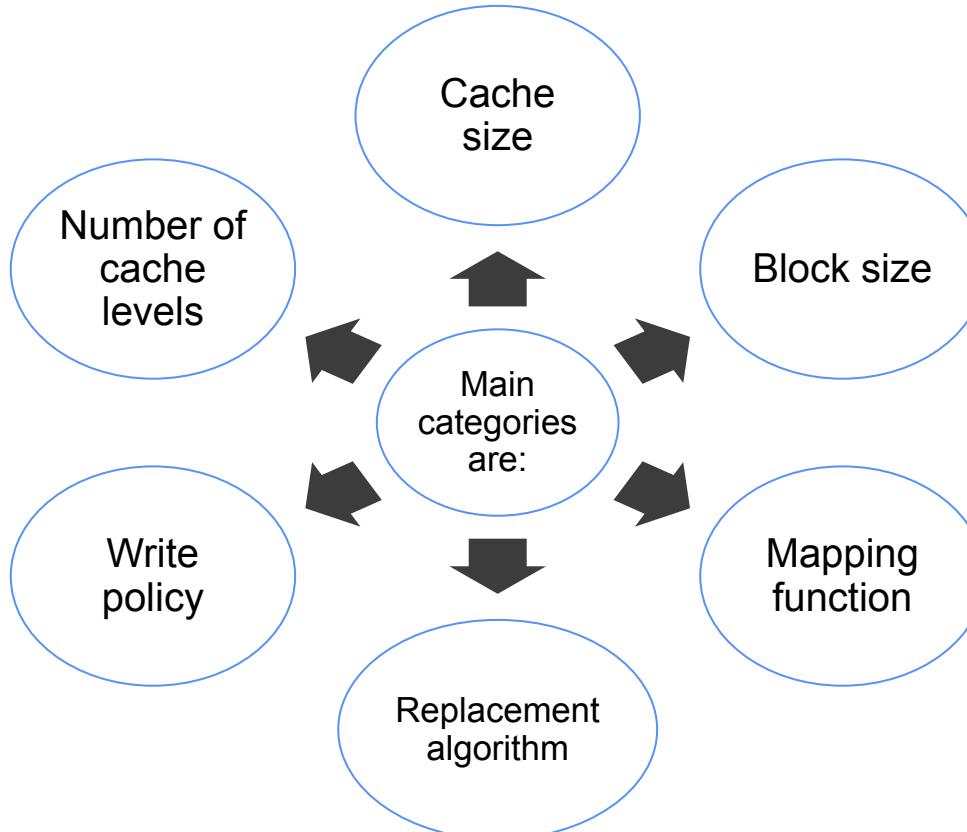
(a) Single cache



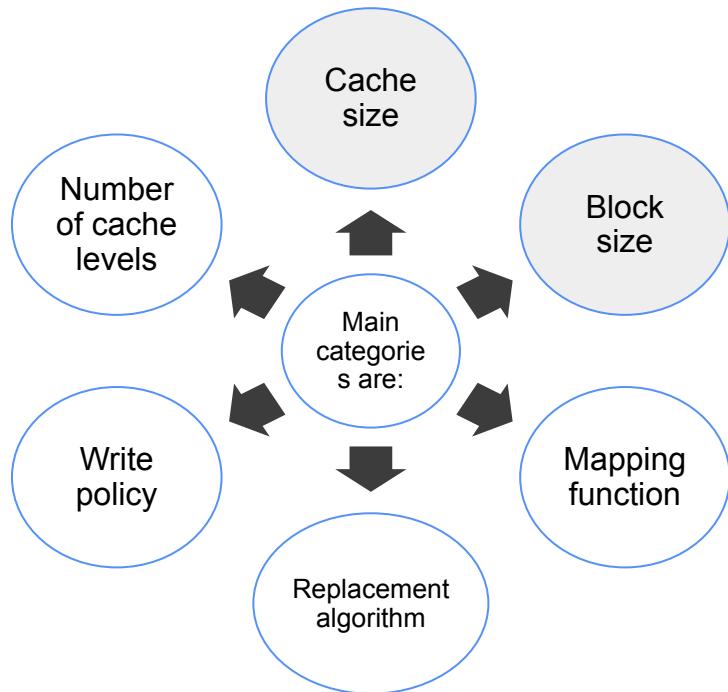
(b) Three-level cache organization

Figure 1.16 Cache and Main Memory

Cache Design



Cache Design



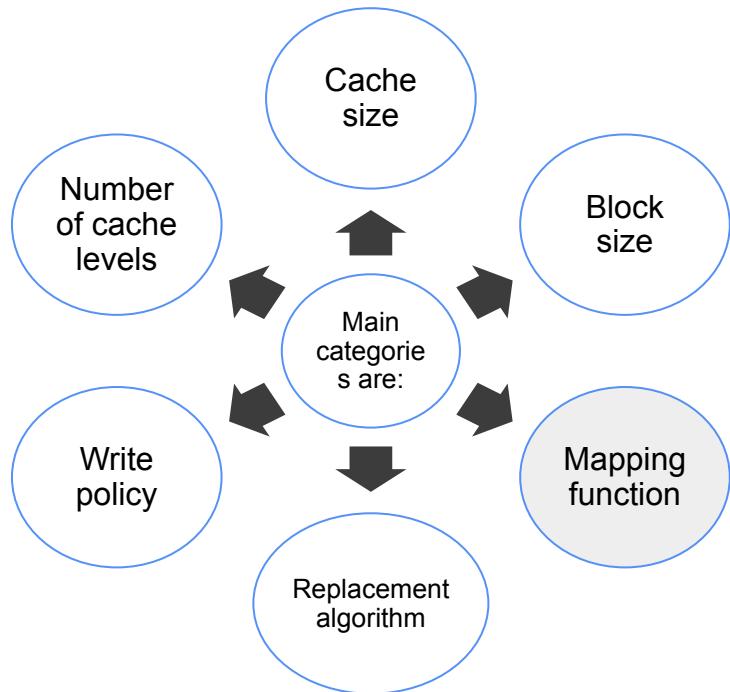
Cache Size

Small caches have significant impact on performance

Block Size

The unit of data exchanged between cache and main memory

Cache Design



Determines which cache location the block will occupy

Two constraints affect design:

When one block is read in, another may have to be replaced

The more flexible the mapping function, the more complex is the circuitry required to search the cache

Writes and allocation

- Basic operation
 - Replacement policy (random, LRU), must be fast!
 - Do not use page replacement algorithms here!
- Reads
 - Caches operate (mainly) on-demand
- Writes
 - Write-through and write-back
 - Datum at the missed-write location is loaded to cache, followed by a write-hit operation
 - Write around (typical with write-through)
 - not loaded to cache, direct writing to memory
- Allocation
 - Write allocate (typical with write-back)

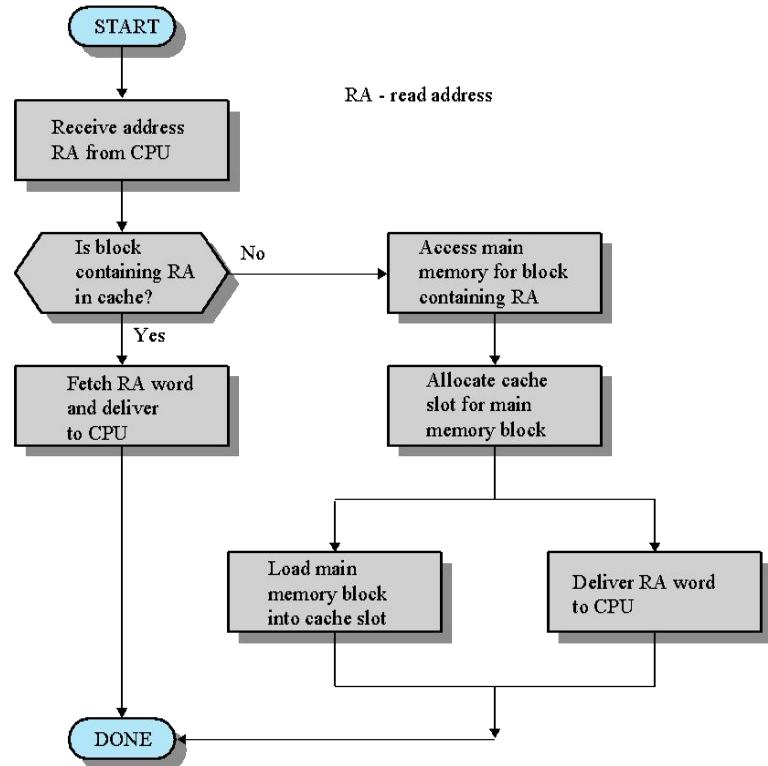


Figure 1.18 Cache Read Operation

Summary

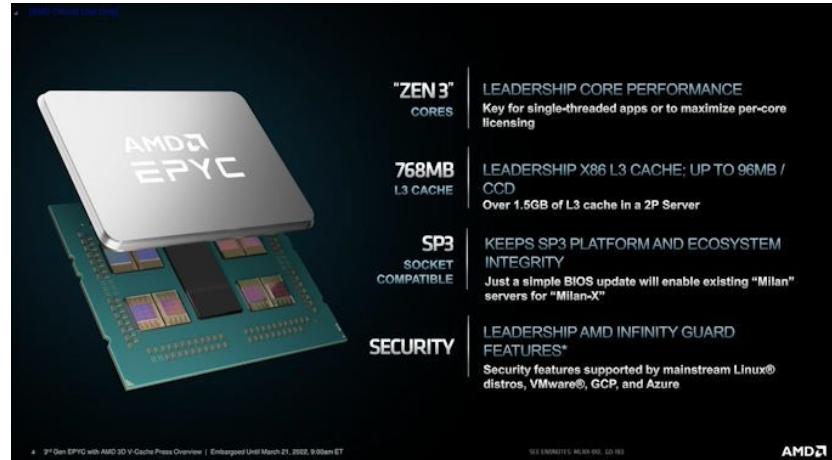
- Basic Elements
- Instruction execution
- Interrupts
 - Interrupts and the instruction cycle
 - Interrupt processing
 - Multiple interrupts
- The memory hierarchy
- Cache memory
 - Motivation
 - Cache principles
 - Cache design

Intel® Pentium® Pro Processor 200 MHz, 1M Cache, 66 MHz FSB	
Performance Specifications	
Total Cores	1
Processor Base Frequency	200 MHz
Cache	1 MB L2 Cache
Bus Speed	66 MHz
TDP	44 W
VID Voltage Range	3.3V±5%/3.2V±0.1V



Intel Pentium Pro, 1995

WD 640MB, 1995



AMD EPYC 7003, 2022

Operating Systems

CS-C3140, Lecture 2

Alexandru Paler

Grading

- The grade is based on points
 - The exercises: n (8?) rounds and 28 points/round -> $n \cdot 28$ points from exercises
 - The exam points are max. 400 - $n \cdot 28$
- To pass the course you have to pass both the exam and the exercises
 - A minimum number of points for the exam (to be announced separately)
 - Final grade = round $((\text{exercises} + \text{exam}) / 100)$
 - 449 -> 400 -> 4
 - 451 -> 500 -> 5
 - Per round limits for the exercises (you have to pass $(n-2)/n$ rounds)
 - 50% of max points for passing the assignments
 - 75% of max points if submitted one week later
 - 2 trials per exercise
 - Do not solve the exercises in the A+ interface
 - Use A+ only for submission
 - Solve in different software (e.g. Notepad)

Resource sharing is typical

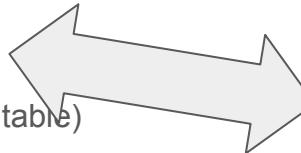
- The basic view of a computer
 - Processor, memory, peripheral devices
- Typically, there are several programs active
 - Even with a single processor (note the concept of concurrency)
 - They have to share the processor, the memory, and the devices
 - This calls for management resource sharing
- Further
 - Each program typically consists of several parts
 - How they can be active simultaneously?
 - The computer can be part of a distributed systems
 - Plenty of sharing of resources, plenty of concurrency



<https://www.computer-history.info/Page4.dir/pages/PDP.1.dir/>
<https://www.computerhistory.org/pdp-1/timesharing/>

Getting a program to run and Running it

- Source code is compiled into machine code
 - Compilers usually come with a runtime system (or similar)
 - Mechanisms for memory allocation are often there
- The compilation result has
 - Program code (text),
 - static data, meta data (e.g., symbol table)
 - other things (like debug info)
- Program parts linked together:
 - executable program
 - stored into a file, and loaded into memory
- Note that
 - There can be other phases
 - In modern systems, the phases are mixed (e.g., a compiler does some initial linking and a loader does some final compilation)



- Getting a program to run takes some time
 - Copying bytes, but also setting values, etc.
 - The CPU must be set
 - Especially the special registers like PSW, PC, FP, SP, etc.
 - This is typically rather fast (compared to memory)
- The major parts (segments) in memory are
 - Text (the program)
 - Data (what we are computing)
 - Stack (the control)
 - The stack together with the CPU state forms the context of the computation, which is essential for control
- Note that
 - Specific formats, padding, etc. is used
 - Typically, a small fraction of the total memory address space is used

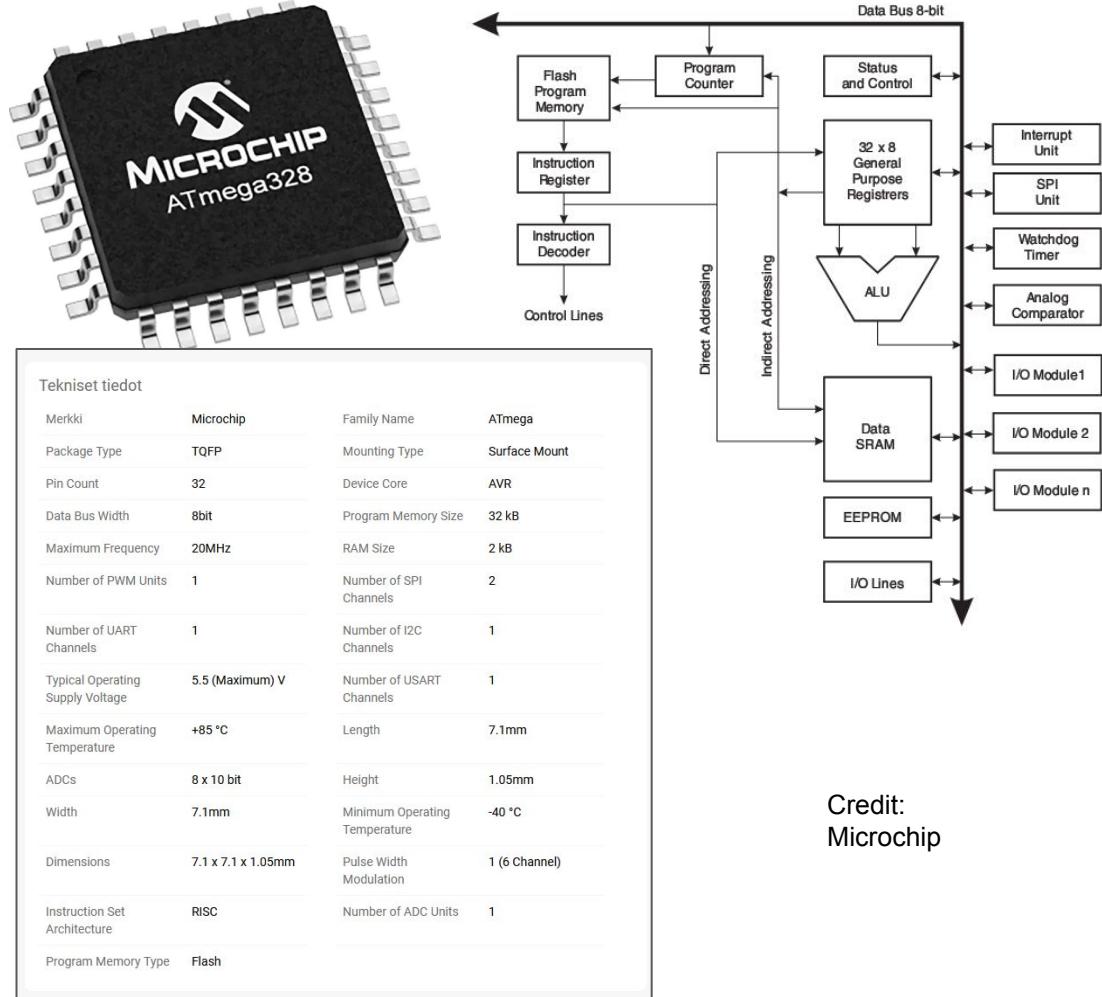
The control stack vs. execution context

- Programs typically have subroutines (or “methods”, etc.)
 - These can be called (or “invoked”, etc.)
 - Usually, the caller remains activated as the callee is executing
 - After the callee execution, the control returns to the caller
- This is usually implemented with a control stack
 - The stack contains frames (abstractly: subroutine activations)
 - The most recent is at the top of the stack
 - Note typically stacks grow downwards (i.e., the “top” is at low end)
 - The FP points to the topmost frame and the frames are linked
 - The FP is used for accessing data
 - The SP point to the stack top
 - The SP is used for allocating (or deallocating space)
- When code is executed, the execution context defines
 - The code and data bindings that are not in the executed code itself (e.g., variable bindings)
 - Is dependent on the source language, on the hardware, on all the layers participating the computation
- There are several levels of execution context
 - CPU state and a stack imply the context for single thread
 - Note that there can be several threads within a program
- In a wider view
 - There are open files, active network connections, etc.
 - Much of such context resides in the OS (or middleware layers)

System Types

Microcontrollers

- Single-chip computers
 - Designed for embedding
 - CPU
 - Memory
 - I/O subsystem
- CPU
 - For RT (predictable)
- Memory
 - Different types
- I/O subsystem
 - Analog/digital
 - Programmable



Credit:
Microchip

Boards

- Instead of buying a microcontroller, you typically buy a “single board computer”
 - With suitable capabilities (especially the interfaces)
 - Can be very cheap... or somewhat more expensive...
- Example: an ODROID system
 - An I/O shield at the top
 - Physical I/O (rich electric IFs)
 - Real-time processing (RT)
 - Slow but predictable
- ODROID-U3
 - Has processing power
 - Fast but unpredictable (not RT)
 - The cyber-side
 - Runs easily a full Linux
 - Network I/O

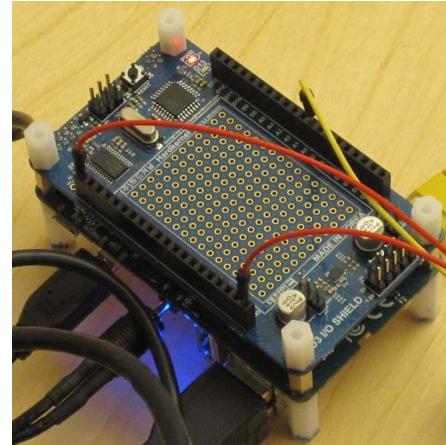
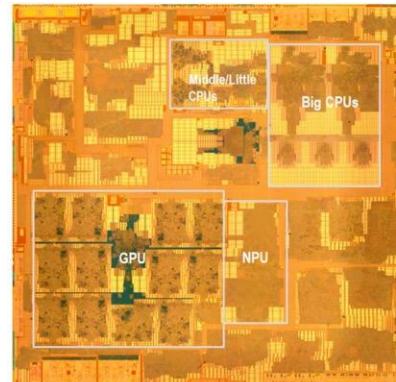
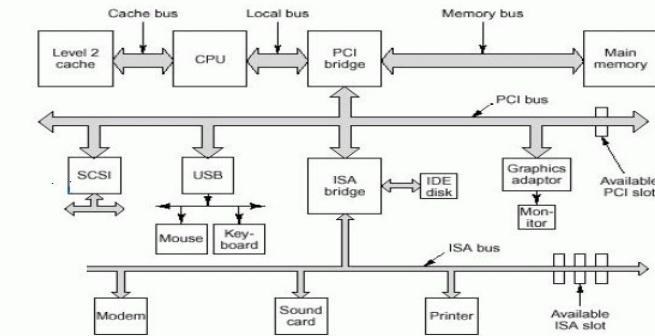


Photo: Vesa Hirvisalo

The classical PC (Personal Computer)

- Centered around the CPU-MEM axis
 - Busses, bridges, controllers
 - peripheral devices
- PCs have (more or less) evolved into laptops
 - Basically, more integration and compactness
 - resemble older PCs a lot
- Smartphones with touch screens dominate the outlook
 - share much with the PC, laptops, etc.



Embedded computers

- In embedded systems
 - A computer is embedded into a host
 - A host can be a classical physical system, i.e., a car
- A traditional computer (e.g. PC)
 - The focus on computation
 - The processor (CPU) and memory
 - The software computes something (of interest)
- An embedded computer
 - The focus is on interaction
 - Sensors and actuators attached
 - ... but there is a lot of variance!
- Industrially
 - An embedded computer is typical for a specific purpose
 - Very often “custom made” (ASICs, etc.)
 - Embedded computer = special purpose computer
 - Are expensive (not only HW manufacturing, but the testing, etc.)

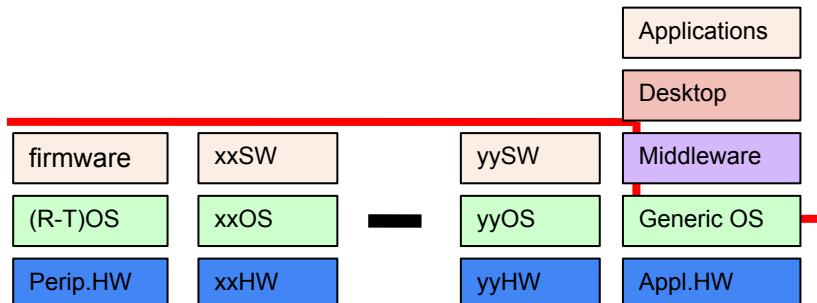
Much of software is embedded

- OS theory \sim “how to construct a SW system/stack”
- single process \rightarrow single stack \rightarrow a system

In a device we have

- many processors (inside peripheral devices)
- on them, many SW stacks and OSes
- the application stack is special

Distribution makes things very complex



Modern cars

- Power train & basic driving
 - Engine and transmission control
 - Steering etc. (ABS, ESP, ...)
 - Monitoring systems (TPMS, OBD-II, ...)
- Dashboard, infotainment, etc.
 - Lights, signals, doors, windows, locking, ...
 - Heating, ventilation, and air conditioning (HVAC)
 - Anti-theft systems, ...
- Actually, there are a lot of things
 - Tens of processors, huge number of sensors and actuators
 - Cars are connected to the cyber-world outside
 - Several communication links, at least one SIM, ...



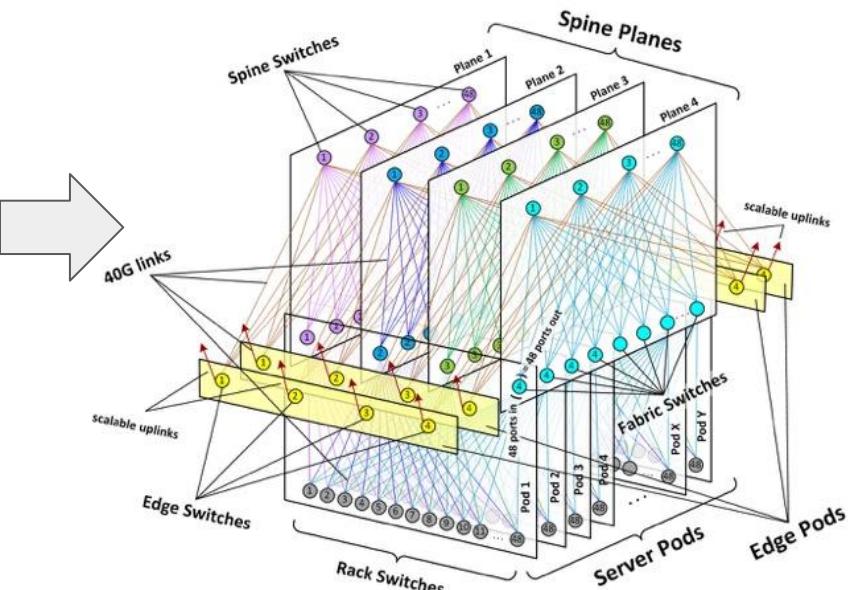
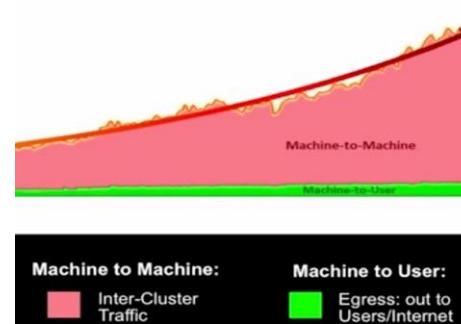
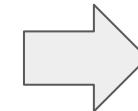
<https://www.carmagazine.co.uk/car-news/industry-news/rimac/mate-rimac-electric-cars/>

Clusters and Data centers

- A computing cluster is a set of connected computer working together for a joint purpose
 - Each node (computer) is usually set to perform the same task
 - There is a wide variety of computing clusters
 - This is because of the variety of reasons for having clusters
 - Better performance, dependability, efficiency, costs, etc. than what can be achieved with a single computer
 - Note that many modern “single computer systems” have merged plenty of cluster technology inside them
 - In practice, the mechanisms of distributed computing are used
 - HPC (High-Performance Computing) has been the main driving force in many respects
-
- Typical data centers are warehouse size computers
 - Built around a set of networks
 - Usually, there is
 - A set of networks
 - Technically, networking is the central thing for data centers
 - A hierarchy of memory and storage
 - Computing typically happens by altering their contents
 - The processing units
 - The processors sit on top of the complex networking and memory/storage systems
 - In addition
 - Plenty of issues on power, cooling, etc.
 - It really is a house!

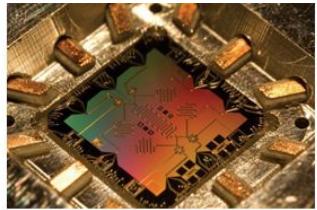
Data center architectures

- Datacenter to Internet traffic is huge
 - Traffic inside the data centers is several orders of magnitude larger
 - Datacenter technology is much about communication
- Traditional hierarchical cluster based designs do not scale with growing traffic
 - New approach is to make the entire data center building one high-performance network
- Automated tools for management
 - A large network with a complex topology and many devices and interconnects cannot be configured and operated in a manual way

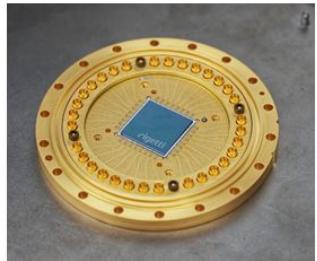


Source: Facebook

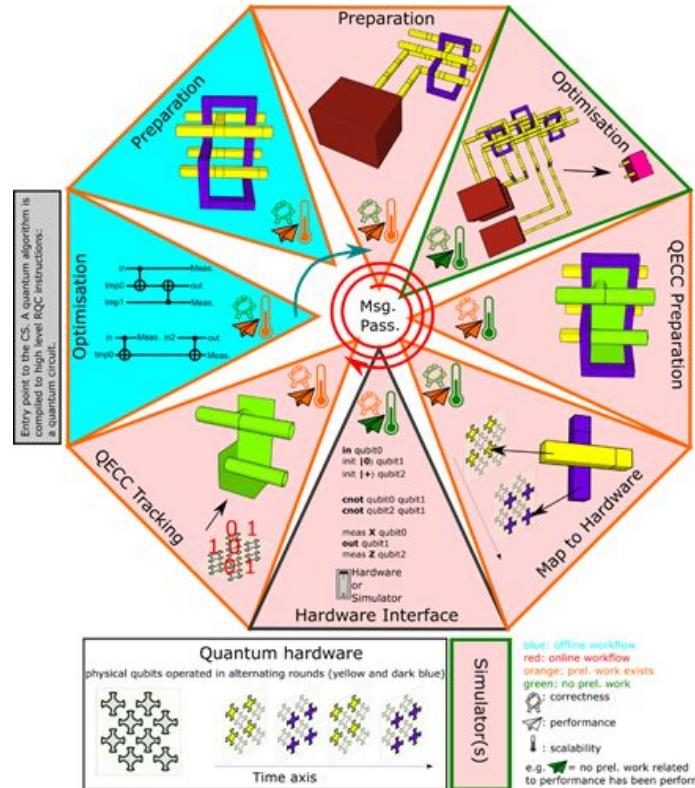
Quantum Computers



The era of NISQ?
Noisy Intermediate-Scale Quantum



Quantum Error-Correction
Quantum Algorithms
Design Automation
Computer Architecture
Machine Learning (?)



Operating Systems Overview

Computers and operating systems

- A computer is a device consisting of
 - CPU
 - Memory
 - I/O peripherals: disk, display, network card
- A computer is executing programs
 - Perform arithmetic or logical computations
 - Have control
 - Do input and output (I/O)
- An operating system is a special program
 - Controls access to computer peripherals
 - Enables other programs to run
- How many different operating systems have you used?



Operating System

- A program that controls the execution of application programs
- An interface between applications and hardware

Main objectives of an OS:

- Convenience
- Efficiency
- Ability to evolve

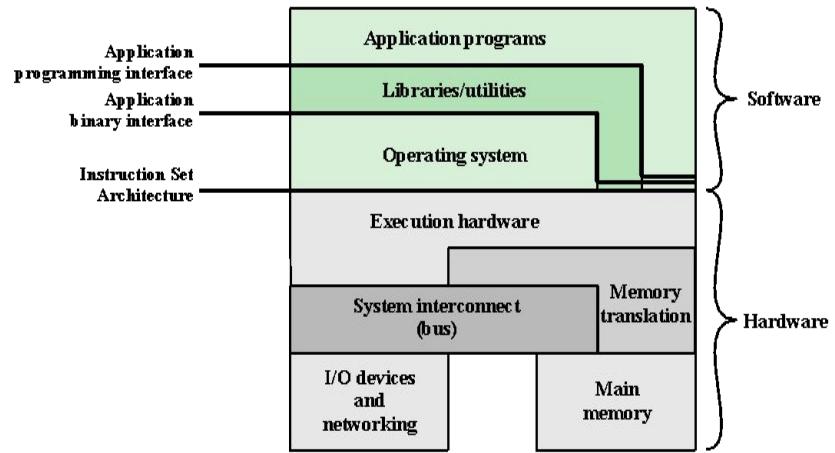


Figure 2.1 Computer Hardware and Software Structure

Operating System Services

- Program development
 - Program execution
 - Access I/O devices
 - Controlled access to files
 - System access
 - Error detection and response
 - Accounting
-
- Key Interfaces
 - Instruction set architecture (ISA)
 - Application binary interface (ABI)
 - Application programming interface (API)
 - **POSIX - Portable Operating System Interface**
 - system- and user-level application programming interfaces (API),
 - command line shells and utility interfaces, for software compatibility (portability) with variants of Unix and other operating systems
 - Certified: e.g macOS
 - Mostly compliant: e.g. Linux, OpenBSD

Operating System as Resource Manager

- The OS is responsible for controlling the use of a computer's resources, such as
 - I/O
 - main and secondary memory
 - processor execution time
- Functions in the same way as ordinary computer software
 - Program, or suite of programs, executed by the processor
 - Frequently relinquishes control and must depend on the processor to allow it to regain control

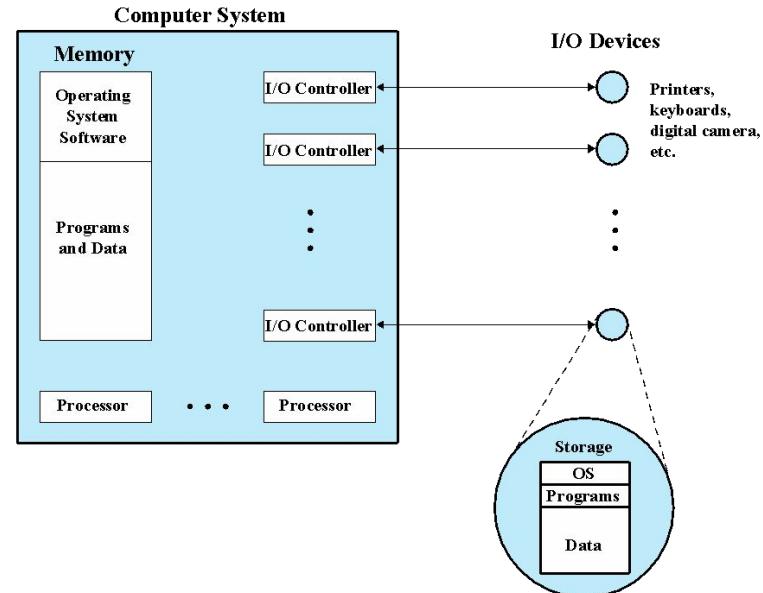


Figure 2.2 The Operating System as Resource Manager

Evolution of Operating Systems

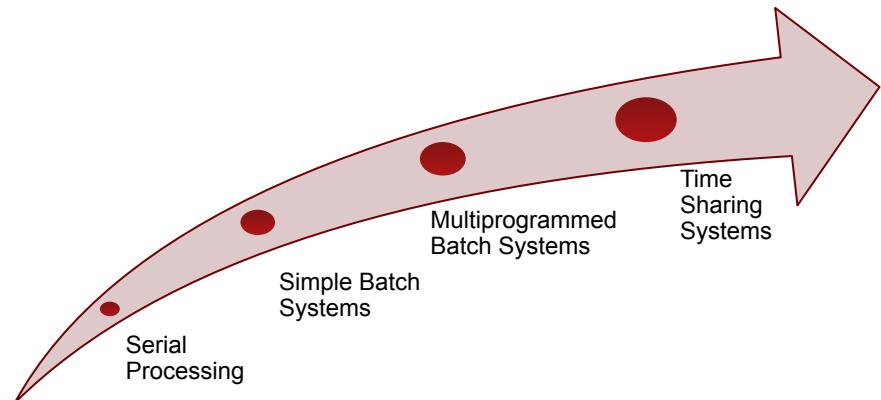
- A major OS will evolve over time for a number of reasons:

Hardware upgrades

New types of hardware

New services

Fixes



Serial Processing

Earliest Computers:

- No operating system
- Programmers interacted directly with the computer hardware
- Computers ran from a console with display lights, toggle switches, some form of input device, and a printer
- Users have access to the computer in “series”

- Scheduling

- Most installations used a hardcopy sign-up sheet to reserve computer time
- Time allocations could run short or long, resulting in wasted computer time

- Setup time

- A considerable amount of time was spent on setting up the program to run

Simple Batch Systems

- Early computers were very expensive
 - Important to maximize processor utilization
- Monitor
 - User no longer has direct access to processor
 - Monitor controls the sequence of events
 - Resident Monitor is software always in memory
 - Job is submitted to computer operator who batches them together and places them on an input device
 - Program branches back to the monitor when finished

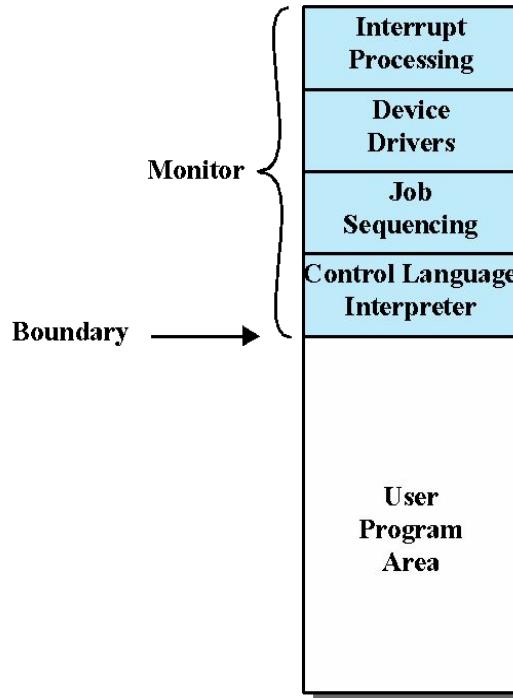


Figure 2.3 Memory Layout for a Resident Monitor

Modes of Operation

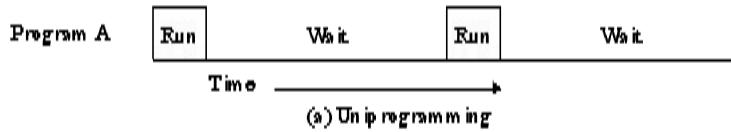
User Mode

- User program executes in user mode
- Certain areas of memory are protected from user access
- Certain instructions may not be executed

Kernel Mode

- Monitor executes in kernel mode
- Privileged instructions may be executed
- Protected areas of memory may be accessed

Uniprogramming



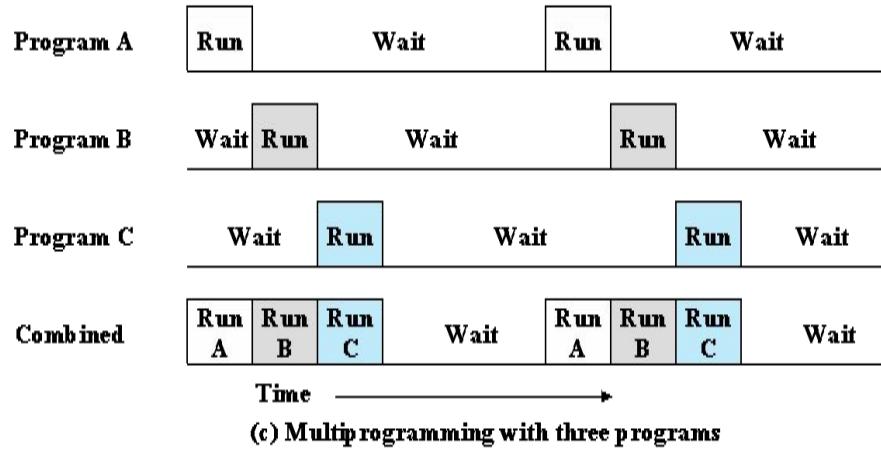
Read one record from file	15 μs
Execute 100 instructions	1 μs
Write one record to file	15 μs
TOTAL	31 μs

$$\text{Percent CPU Utilization} = \frac{1}{31} = 0.032 = 3.2\%$$

Figure 2.4 System Utilization Example

The processor spends a certain amount of time executing, until it reaches an I/O instruction; it must then wait until that I/O instruction concludes before proceeding

Multiprogramming aka multitasking



- There must be enough memory to hold the OS (resident monitor) and one user program
- When one job needs to wait for I/O, the processor can switch to the other job, which is likely not waiting for I/O
- Memory is expanded to hold three, four, or more programs and switch among all of them

Multiprogramming Example

	JOB1	JOB2	JOB3
Type of job	Heavy compute	Heavy I/O	Heavy I/O
Duration	5 min	15 min	10 min
Memory required	50 M	100 M	75 M
Need disk?	No	No	Yes
Need terminal?	No	Yes	No
Need printer?	No	No	Yes

	Uniprogramming	Multiprogramming
Processor use	20%	40%
Memory use	33%	67%
Disk use	33%	67%
Printer use	33%	67%
Elapsed time	30 min	15 min
Throughput	6 jobs/hr	12 jobs/hr
Mean response time	18 min	10 min

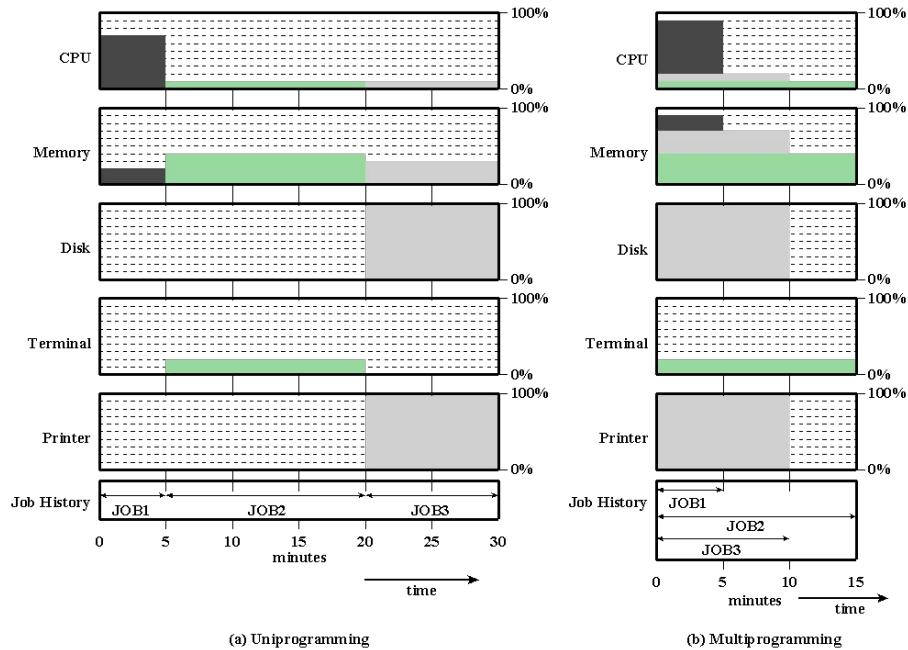


Figure 2.6 Utilization Histograms

Time-Sharing Systems

- Can be used to handle multiple interactive jobs
- Processor time is shared among multiple users
- Multiple users simultaneously access the system through terminals,
- The OS interleaves the execution of each user program in a short burst or quantum of computation

	Batch Multiprogramming	Time Sharing
Principal objective	Maximize processor use	Minimize response time
Source of directives to operating system	Job control language commands provided with the job	Commands entered at the terminal

Compatible Time-Sharing System (CTSS)

- One of the first time-sharing operating systems
 - Developed at MIT by a group known as Project MAC
 - The system was first developed for the IBM 709 in 1961
 - Ran on a computer with 32,000 36-bit words of main memory, with the resident monitor consuming 5000 of that
- Utilized a technique known as **time slicing**
 - System clock generated interrupts at a rate of approximately one every 0.2 seconds
 - At each clock interrupt the OS regained control and could assign the processor to another user
 - At regular time intervals the current user would be preempted and another user loaded in
- To preserve the old user program status for later resumption, the old user programs and data were written out to disk before the new user programs and data were read in
- Old user program code and data were restored in main memory when that program was next given a turn

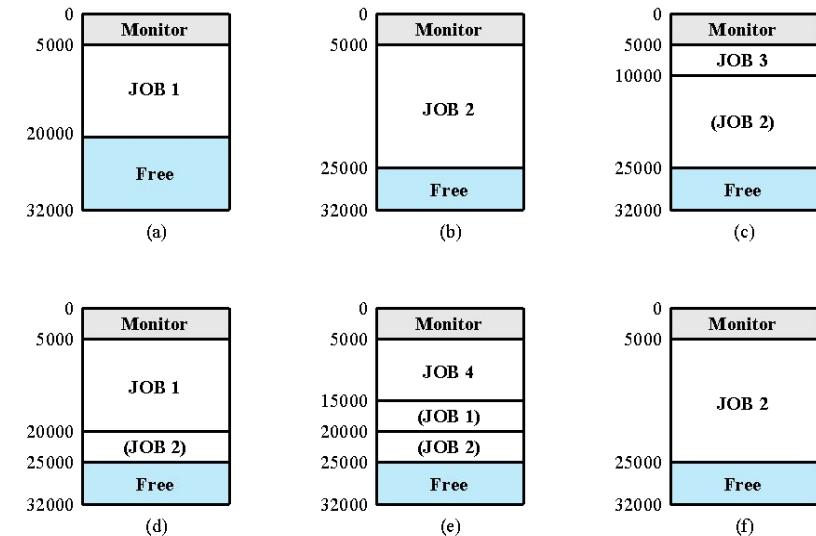


Figure 2.7 CTSS Operation

Major Achievements in OS

Major Achievements

- Operating Systems are among the most complex pieces of software ever developed
- Major advances in development include:
 - i. Processes
 - ii. Memory management
 - iii. Information protection and security
 - iv. Scheduling and resource management
 - v. System structure

1. Process

Fundamental to the structure of operating systems

A process can be defined as:

A program in execution

An instance of a running program

The entity that can be assigned to, and executed on, a processor

A unit of activity characterized by a single sequential thread of execution, a current state, and an associated set of system resources

Components of a Process

- A process contains three components:
 - An executable program
 - The associated data needed by the program (variables, work space, buffers, etc.)
 - The execution context

- The execution context is essential:
 - It is the internal data by which the OS is able to supervise and control the process
 - Includes the contents of the various process registers
 - Includes information such as the priority of the process and whether the process is waiting for the completion of a particular I/O event

Process Management

- The entire state of the process at any instant is contained in its context
- New features can be designed and incorporated into the OS by expanding the context to include any new information needed to support the feature

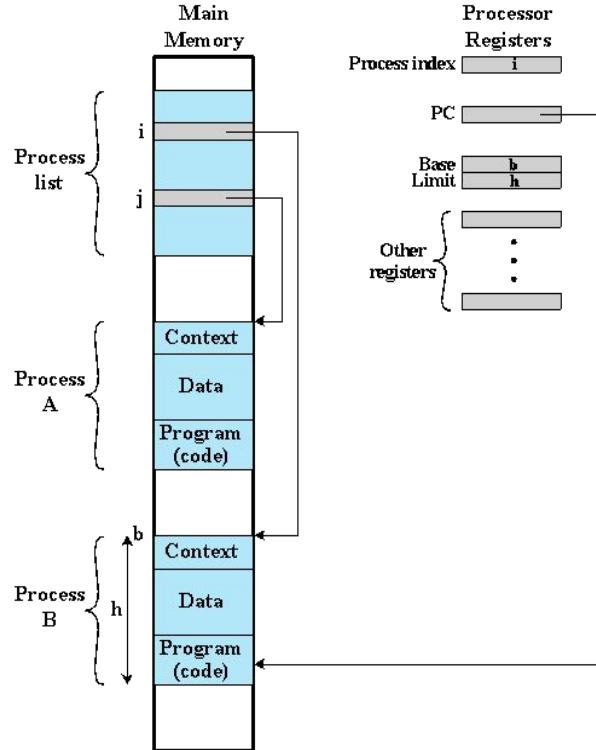


Figure 2.8 Typical Process Implementation

2. Memory Management

- The OS has five principal storage management responsibilities:

Process isolation

Automatic allocation
and management

Support of modular programming

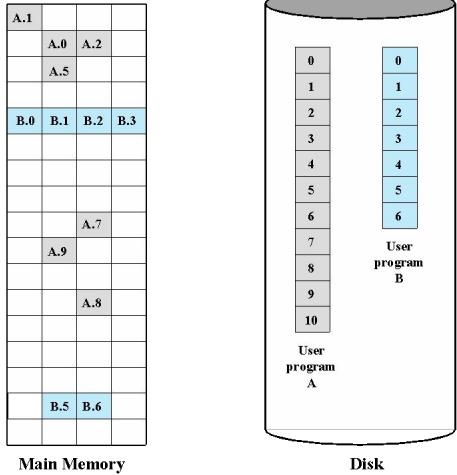
Protection and access control

Long-term storage

2. Virtual Memory and Paging

- A facility that allows programs to address memory from a logical point of view, without regard to the amount of main memory physically available
- Conceived to meet the requirement of having multiple user jobs reside in main memory concurrently
- Allows processes to be comprised of a number of fixed-size blocks, called pages
- Program references a word by means of a virtual address, consisting of a page number and an offset within the page
- Each page of a process may be located anywhere in main memory
- The paging system provides for a dynamic mapping between the virtual address used in the program and a real address (or physical address) in main memory

2. Virtual Memory and Paging



Main memory consists of a number of fixed-length frames, each equal to the size of a page. For a program to execute, some or all of its pages must be in main memory.

Secondary memory (disk) can hold many fixed-length pages. A user program consists of some number of pages. Pages for all programs plus the operating system are on disk, as are files.

Figure 2.9 Virtual Memory Concepts

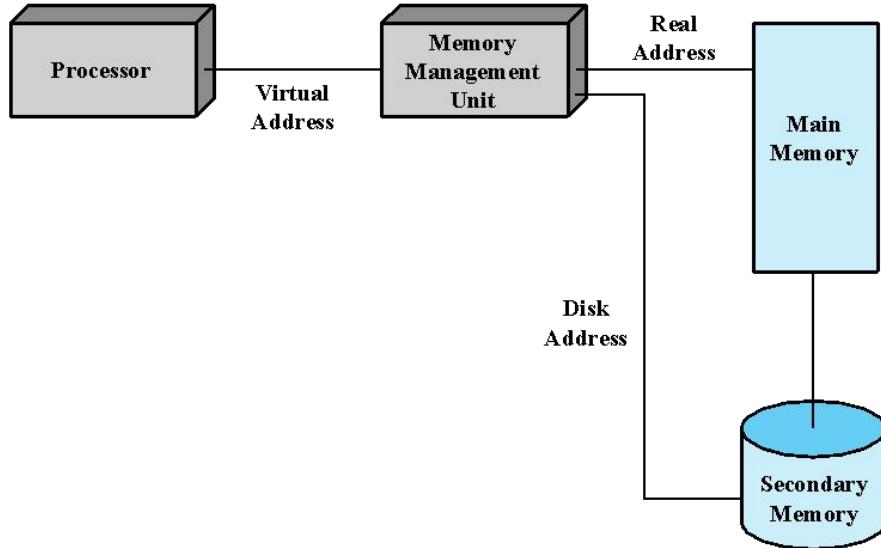
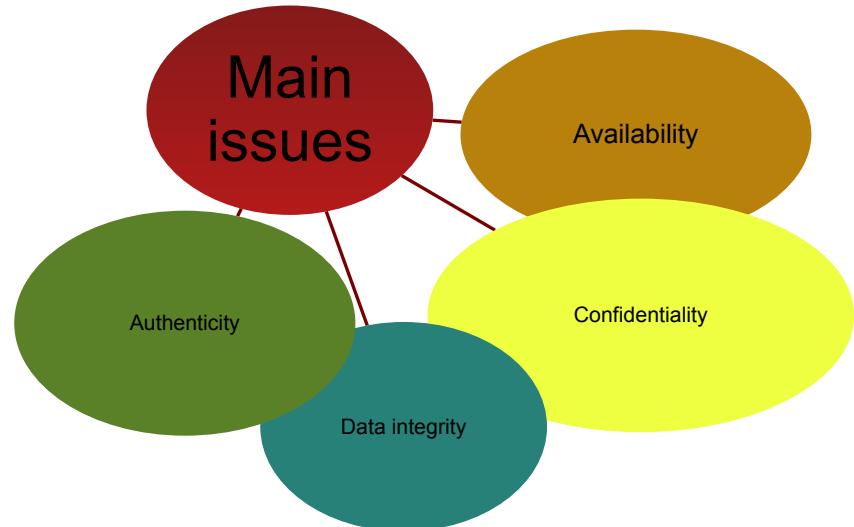


Figure 2.10 Virtual Memory Addressing

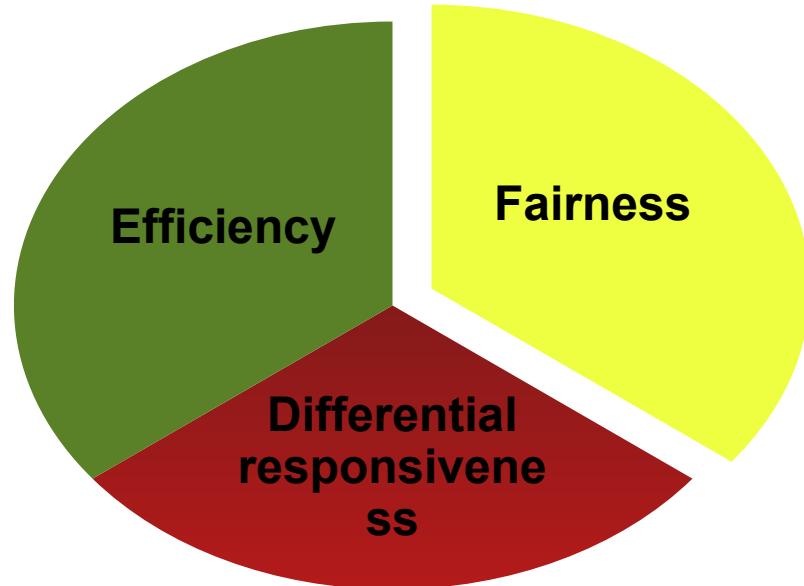
3. Information Protection and Security

- The nature of the threat that concerns an organization will vary greatly depending on the circumstances
- The problem involves controlling access to computer systems and the information stored in them



4. Scheduling and Resource Management

- Key responsibility of an OS is managing resources
- Resource allocation policies must consider:



5. Different Architectural Approaches

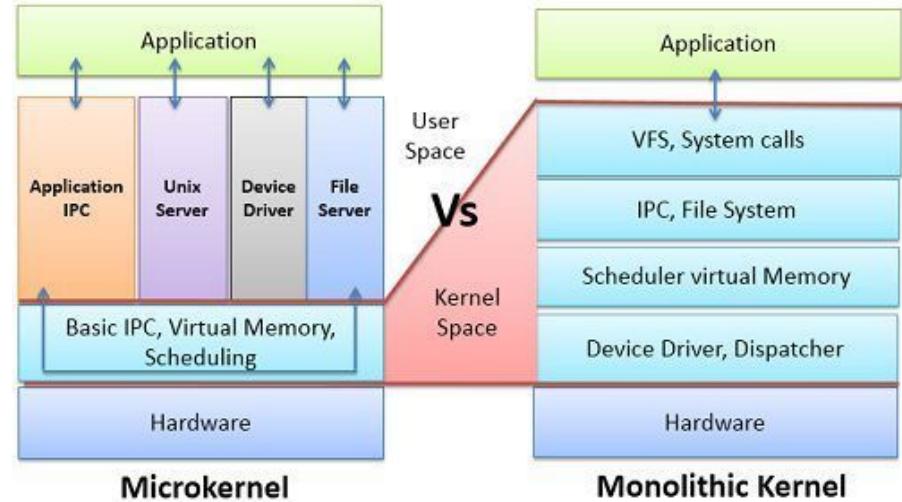
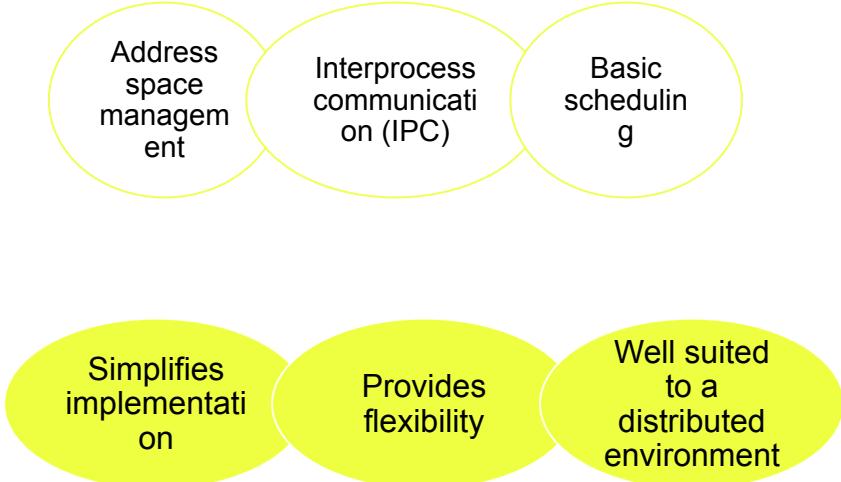
- Demands on operating systems require new ways of organizing the OS

Different approaches and design elements have been tried:

- Microkernel architecture
- Multithreading
- Symmetric multiprocessing
- Distributed operating systems
- Object-oriented design

Microkernel Architecture

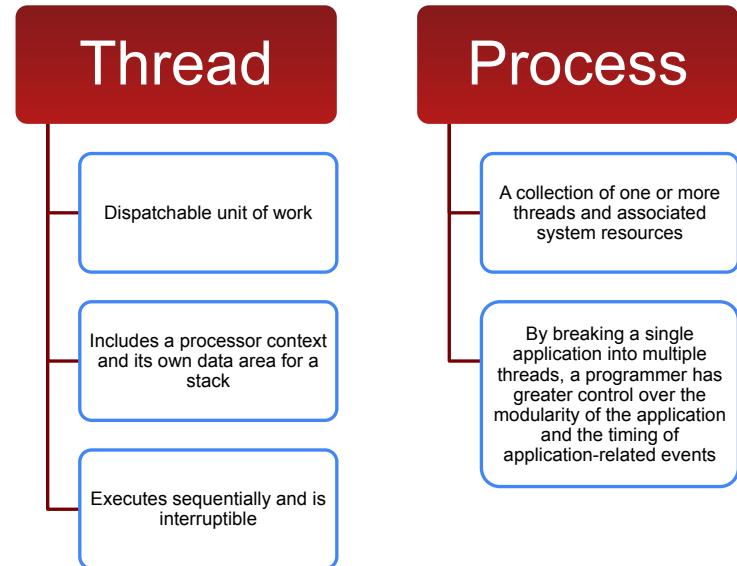
- Assigns only a few essential functions to the kernel:



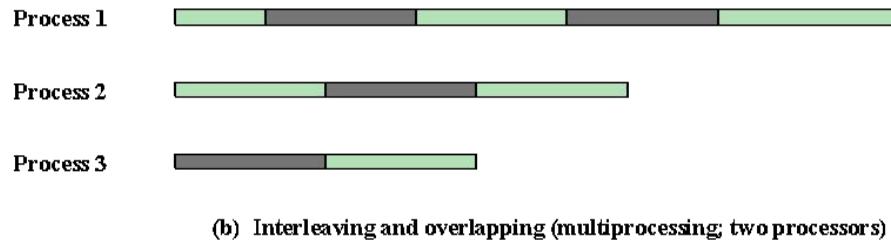
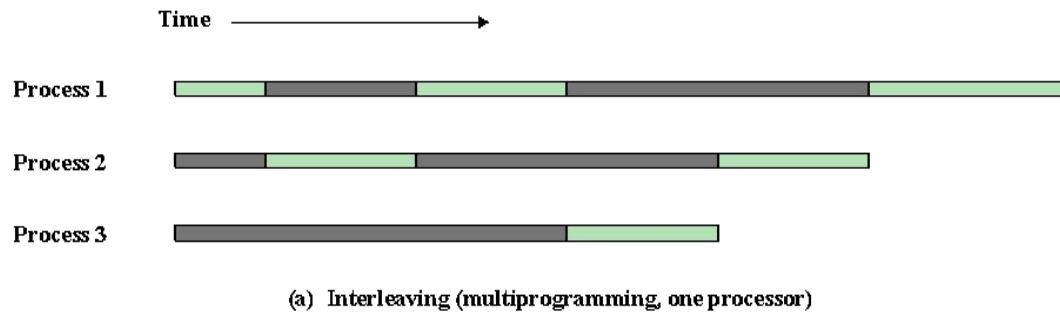
<https://techdifferences.com/difference-between-microkernel-and-monolithic-kernel.html>

Symmetric Multiprocessing and Multithreading

- Term that refers to a computer hardware architecture and also to the OS behavior that exploits that architecture
- The OS of an SMP schedules processes or threads across all of the processors
- The OS must provide tools and functions to exploit the parallelism in an SMP system
- Multithreading and SMP are often discussed together, but the two are independent facilities
- An attractive feature of an SMP is that the existence of multiple processors is transparent to the user
- Technique in which a process, executing an application, is divided into threads that can run concurrently



Multiprogramming and Multiprocessing



■ Blocked ■ Running

Operating Systems

CS-C3140, Lecture 3

Alexandru Paler

Fault Tolerance

- Refers to the ability of a system or component to continue normal operation despite the presence of hardware or software faults
 - involves some degree of redundancy
 - Intended to increase the reliability of a system
 - comes with a cost in financial terms or performance
- The extent adoption of fault tolerance measures must be determined by how critical the resource is

IEEE TRANSACTIONS ON DEPENDABLE AND SECURE COMPUTING, VOL. 1, NO. 1, JANUARY-MARCH 2004

1

Basic Concepts and Taxonomy of Dependable and Secure Computing

Algirdas Avizienis, *Fellow, IEEE*, Jean-Claude Laprie, Brian Randell, and Carl Landwehr

Abstract—This paper gives the main definitions relating to dependability, a generic concept including as special case such attributes as reliability, availability, safety, integrity, maintainability, etc. Security brings in concerns for confidentiality, in addition to availability and integrity. Basic definitions are given first. They are then commented upon, and supplemented by additional definitions, which address the threats to dependability and security (faults, errors, failures), their attributes, and the means for their achievement (fault prevention, fault tolerance, fault removal, fault forecasting). The aim is to explicate a set of general concepts, of relevance across a wide range of situations and, therefore, helping communication and cooperation among a number of scientific and technical communities, including ones that are concentrating on particular types of system, of system failures, or of causes of system failures.

Index Terms—Dependability, security, trust, faults, errors, failures, vulnerabilities, attacks, fault tolerance, fault removal, fault forecasting.

1 INTRODUCTION

THIS paper aims to give precise definitions characterizing the various concepts that come into play when addressing the dependability and security of computing and communication systems. Clarifying these concepts is surprisingly difficult when we discuss systems in which there are uncertainties about system boundaries. Furthermore, the very complexity of systems (and their specification) is often a major problem, the determination of possible causes or consequences of failure can be a very subtle process, and there are (fallible) provisions for preventing faults from causing failures.

Dependability is first introduced as a global concept that subsumes the usual attributes of reliability, availability, safety, integrity, maintainability, etc. The consideration of security brings in concerns for confidentiality, in addition to availability and integrity. The basic definitions are then commented upon and supplemented by additional definitions. **Boldface** characters are used when a term is defined, while *italic* characters are an invitation to focus the reader's attention.

This paper can be seen as an attempt to document a minimum consensus on concepts within various specialties in order to facilitate fruitful technical interactions; in addition, we hope that it will be suitable 1) for use by

the concepts: words are only of interest because they unequivocally label concepts and enable ideas and viewpoints to be shared. An important issue, for which we believe a consensus has not yet emerged, concerns the measures of dependability and security; this issue will necessitate further elaboration before being documented consistently with the other aspects of the taxonomy that is presented here.

The paper has no pretension of documenting the state-of-the-art. Thus, together with the focus on concepts, we do not address implementation issues such as can be found in standards, for example, in [30] for safety or [32] for security.

The dependability and security communities have followed distinct, but convergent paths: 1) dependability has realized that restriction to nonmalicious faults was addressing only a part of the problem, 2) security has realized that the main focus that was put in the past on confidentiality needed to be augmented with concerns for integrity and for availability (they have been always present in the definitions, but did not receive as much attention as confidentiality). The paper aims to bring together the common strands of dependability and security although, for reasons of space limitation, confidentiality is not given the attention it deserves.

From Failure to Disaster

- Computers are components in larger technical or societal systems
- Failure detection and manual back-up system can prevent disaster
 - Used routinely in safety-critical systems
 - Manual control/override in jetliners
 - Ground-based control for spacecraft
 - Manual bypass in nuclear reactors
- Catastrophic
 - Serious consequences
- Major
 - Incorrect operation
 - Possibly recoverable
- Minor
 - Inconvenience
- Not noticed



Space Shuttle Challenger Disaster

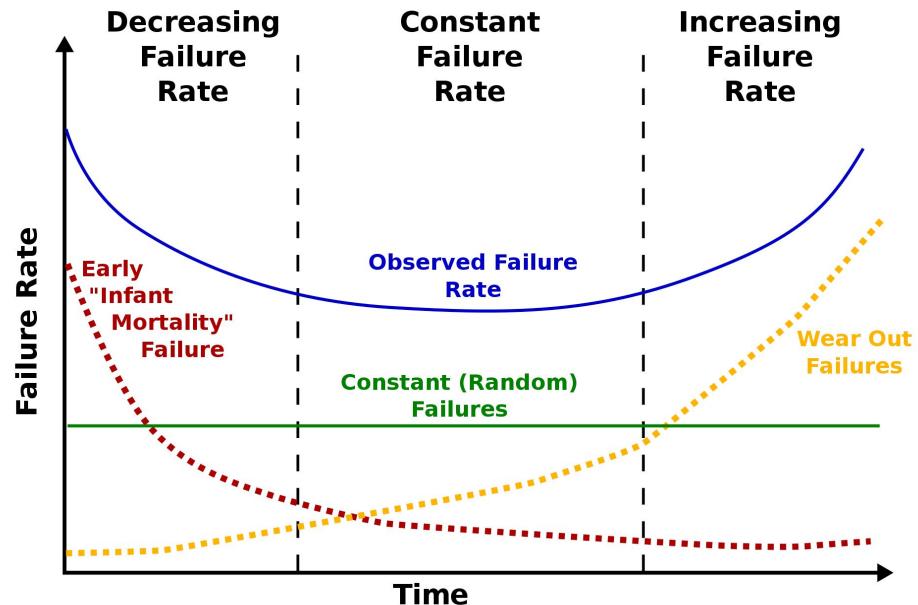
- Second shuttle after Columbia, first mission on April 4, 1983
- Exploded 73 seconds after launch on its 10th mission (January 28, 1986)
- Seven crew members killed
- Temperatures dipped below freezing on launch day
- Engineers determined that the cold temperatures caused a loss of flexibility in the O-rings that decreased their ability to seal the field joints, which allowed hot gas and soot to flow past the primary O-rings

Fault-Tolerance Steps

- Fault Detection
 - Determining that a fault has occurred
- Diagnosis
 - Determining what caused the fault, or exactly which subsystem or component is faulty
- Containment
 - Prevents the propagation of faults from their origin at one point in a system to a point where it can have an effect on the service to the user
- Masking
 - Insuring that only correct values get passed to the system boundary in spite of a failed component.
- Compensation
 - If a fault occurs and is confined to a subsystem, it may be necessary for the system to provide a response to compensate for output of the faulty subsystem.
- Repair
 - The process in which faults are removed from a system. In well-designed fault tolerant systems, faults are contained before they propagate to the extent that the delivery of system service is affected.
 - This leaves a portion of the system unusable because of residual faults.
 - If subsequent faults occur, the system may be unable to cope because of this loss of resources, unless these resources are reclaimed through a recovery process which insures that no faults remain in system resources or in the system state

Fundamental Concepts

- Reliability $R(t)$
 - continuity of correct service
 - The probability of its correct operation up to time t given that the system was operating correctly at time $t=0$
- Mean time to failure (MTTF)
 - Mean time to repair (MTTR) is the average time it takes to repair or replace a faulty element
- Availability
 - readiness for correct service
 - The fraction of time the system is available to service users' requests
- Safety: absence of catastrophic consequences on the user(s) and the environment
- Integrity: absence of improper system alterations.
- Maintainability: ability to undergo modifications and repairs



https://en.wikipedia.org/wiki/Bathtub_curve

Example and Classes

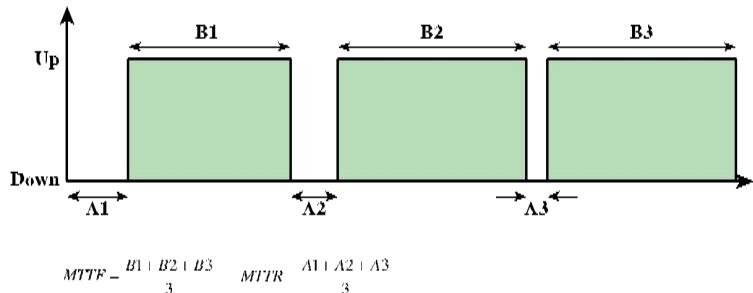


Figure 2.13 System Operational States

Class	Availability	Annual Downtime
Continuous	1.0	0
Fault Tolerant	0.99999	5 minutes
Fault Resilient	0.9999	53 minutes
High Availability	0.999	8.3 hours
Normal Availability	0.99 - 0.995	44-87 hours

Fault Categories

- Permanent
 - A fault that, after it occurs, is always present
 - The fault persists until the faulty component is replaced or repaired
- Temporary
 - A fault that is not present all the time for all operating conditions
 - Can be classified as
 - Transient – a fault that occurs only once
 - Intermittent – a fault that occurs at multiple, unpredictable times

IEEE TRANSACTIONS ON DEPENDABLE AND SECURE COMPUTING, VOL. 1, NO. 1, JANUARY-MARCH 2004

1

Basic Concepts and Taxonomy of Dependable and Secure Computing

Algirdas Avizienis, *Fellow, IEEE*, Jean-Claude Laprie, Brian Randell, and Carl Landwehr

Abstract—This paper gives the main definitions relating to dependability, a generic concept including as special case such attributes as reliability, availability, safety, integrity, maintainability, etc. Security brings in concerns for confidentiality, in addition to availability and integrity. Basic definitions are given first. They are then commented upon, and supplemented by additional definitions, which address the threats to dependability and security (faults, errors, failures), their attributes, and the means for their achievement (fault prevention, fault tolerance, fault removal, fault forecasting). The aim is to explicate a set of general concepts, of relevance across a wide range of situations and, therefore, helping communication and cooperation among a number of scientific and technical communities, including ones that are concentrating on particular types of system, of system failures, or of causes of system failures.

Index Terms—Dependability, security, trust, faults, errors, failures, vulnerabilities, attacks, fault tolerance, fault removal, fault forecasting.

1 INTRODUCTION

THIS paper aims to give precise definitions characterizing the various concepts that come into play when addressing the dependability and security of computing and communication systems. Clarifying these concepts is surprisingly difficult when we discuss systems in which there are uncertainties about system boundaries. Furthermore, the very complexity of systems (and their specification) is often a major problem, the determination of possible causes or consequences of failure can be a very subtle process, and there are (fallible) provisions for preventing faults from causing failures.

Dependability is first introduced as a global concept that subsumes the usual attributes of reliability, availability, safety, integrity, maintainability, etc. The consideration of security brings in concerns for confidentiality, in addition to availability and integrity. The basic definitions are then commented upon and supplemented by additional definitions. **Boldface** characters are used when a term is defined, while *italic* characters are an invitation to focus the reader's attention.

This paper can be seen as an attempt to document a minimum consensus on concepts within various specialties in order to facilitate fruitful technical interactions; in addition, we hope that it will be suitable 1) for use by

the concepts: words are only of interest because they unequivocally label concepts and enable ideas and viewpoints to be shared. An important issue, for which we believe a consensus has not yet emerged, concerns the measures of dependability and security; this issue will necessitate further elaboration before being documented consistently with the other aspects of the taxonomy that is presented here.

The paper has no pretension of documenting the state-of-the-art. Thus, together with the focus on concepts, we do not address implementation issues such as can be found in standards, for example, in [30] for safety or [32] for security.

The dependability and security communities have followed distinct, but convergent paths: 1) dependability has realized that restriction to nonmalicious faults was addressing only a part of the problem, 2) security has realized that the main focus that was put in the past on confidentiality needed to be augmented with concerns for integrity and for availability (they have been always present in the definitions, but did not receive as much attention as confidentiality). The paper aims to bring together the common strands of dependability and security although, for reasons of space limitation, confidentiality is not given the attention it deserves.

Causes of Errors

- Nondeterminate program operation
 - When **programs share memory**, and their execution is interleaved by the processor, they may interfere with each other by overwriting common memory areas in unpredictable ways
 - **The order in which programs are scheduled** may affect the outcome of any particular program
- Improper synchronization
 - It is often the case that **a routine must be suspended awaiting an event elsewhere in the system**
 - Improper design of the signaling mechanism can result in loss or duplication
- Failed mutual exclusion
 - More than one user or program attempts to **make use of a shared resource at the same time**
 - There must be some sort of mutual exclusion mechanism that permits only one routine at a time to perform an update against the file
- **Deadlocks**
 - It is possible for two or more programs to be hung up waiting for each other

Dependability and RAS (Reliability, Availability and Serviceability)

- A system should
 - be capable of detecting hardware errors, and, when possible correcting them in runtime
 - provide mechanisms to detect hardware degradation, in order to warn the system administrator to take the action of replacing a component before it causes data loss or system downtime
- **Dependability**
 - original definition: is the ability to deliver service that can justifiably be trusted
 - alternate definition: the ability to avoid service failures that are more frequent and more severe than is acceptable
- The dependence of system A on system B, thus, represents the extent to which system A's dependability is (or would be) affected by that of System B.
- The concept of dependence leads to that of trust, which can very conveniently be defined as accepted dependence.

Concept	Dependability	High Confidence	Survivability	Trustworthiness
Goal	1) ability to deliver service that can justifiably be trusted 2) ability of a system to avoid service failures that are more frequent or more severe than is acceptable	consequences of the system behavior are well understood and predictable	capability of a system to fulfill its mission in a timely manner	assurance that a system will perform as expected
Threats present	1) development faults (e.g., software flaws, hardware errata, malicious logic) 2) physical faults (e.g., production defects, physical deterioration) 3) interaction faults (e.g., physical interference, input mistakes, attacks, including viruses, worms, intrusions)	• internal and external threats • naturally occurring hazards and malicious attacks from a sophisticated and well-funded adversary	1) attacks (e.g., intrusions, probes, denials of service) 2) failures (internally generated events due to, e.g., software design errors, hardware degradation, human errors, corrupted data) 3) accidents (externally generated events such as natural disasters)	1) hostile attacks (from hackers or insiders) 2) environmental disruptions (accidental disruptions, either man-made or natural) 3) human and operator errors (e.g., software flaws, mistakes by human operators)
Reference	This paper	"Information Technology Frontiers for a New Millennium (Blue Book 2000)" (48)	"Survivable network systems" (16)	"Trust in cyberspace" (62)

Fig. 15. Dependability, high confidence, survivability, and trustworthiness.

Methods of Redundancy - No single point of failure

- Design the system with multiple instances of critical units in such a manner that the failure of some of these units does not directly fail the entire system.
- A number of techniques can be incorporated into OS software to support fault tolerance:
 - Process isolation
 - Concurrency controls
 - Virtual machines
 - Checkpoints and rollbacks

Spatial (physical) redundancy

Involves the use of multiple components that either perform the same function simultaneously or are configured so that one component is available as a backup in case of the failure of another component

Temporal redundancy

Involves repeating a function or operation when an error is detected

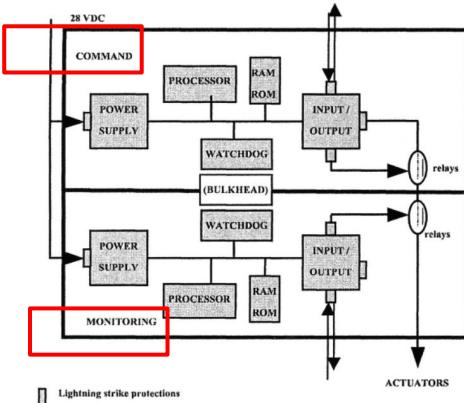
Is effective with temporary faults but not useful for permanent faults

Information redundancy

Provides fault tolerance by replicating or coding data in such a way that bit errors can be both detected and corrected

Example: Fly-by-wire and redundancy

- The first electrical flight control system (a.k.a. Fly-by-Wire) for a civil aircraft was designed by Aerospatiale and installed on Concorde.
- Airbus A320 entered operation in 1988
 - Other models include A340 and A380
 - Primary (P) and secondary (S) computers (different designs and suppliers)
 - Multiple redundant software modules



Two types of computers

- PRIM's (primary computers)
- SEC's (secondary computers)

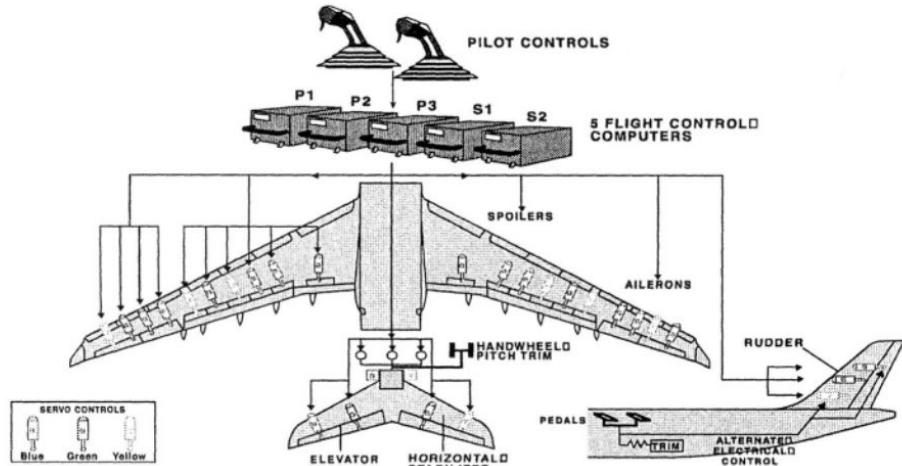
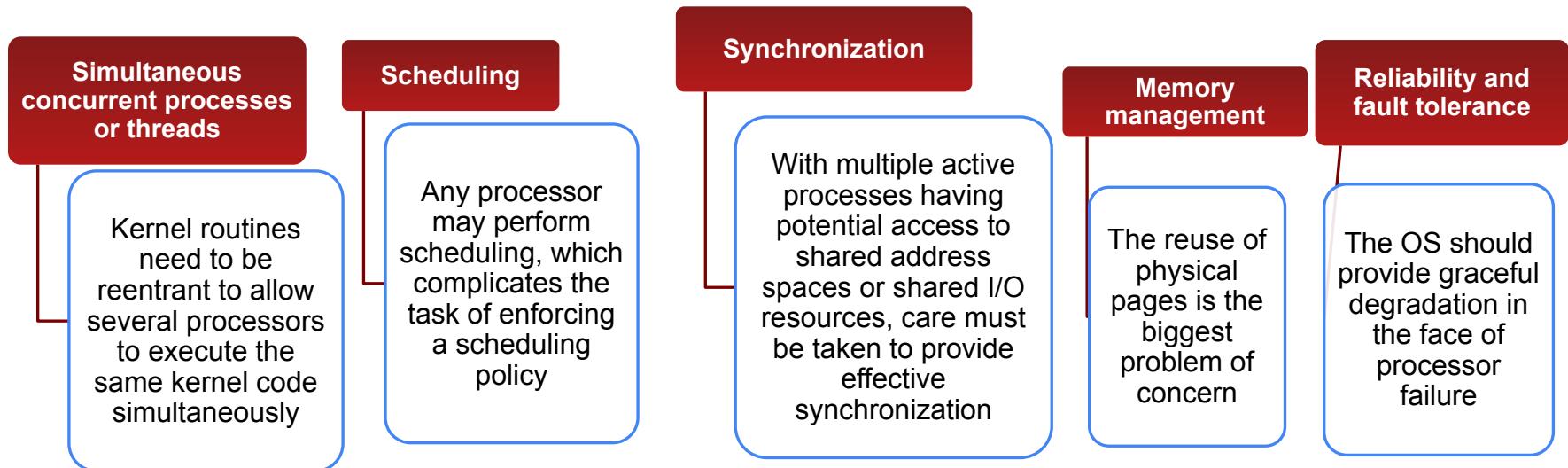


Figure 2: A340-600 system architecture

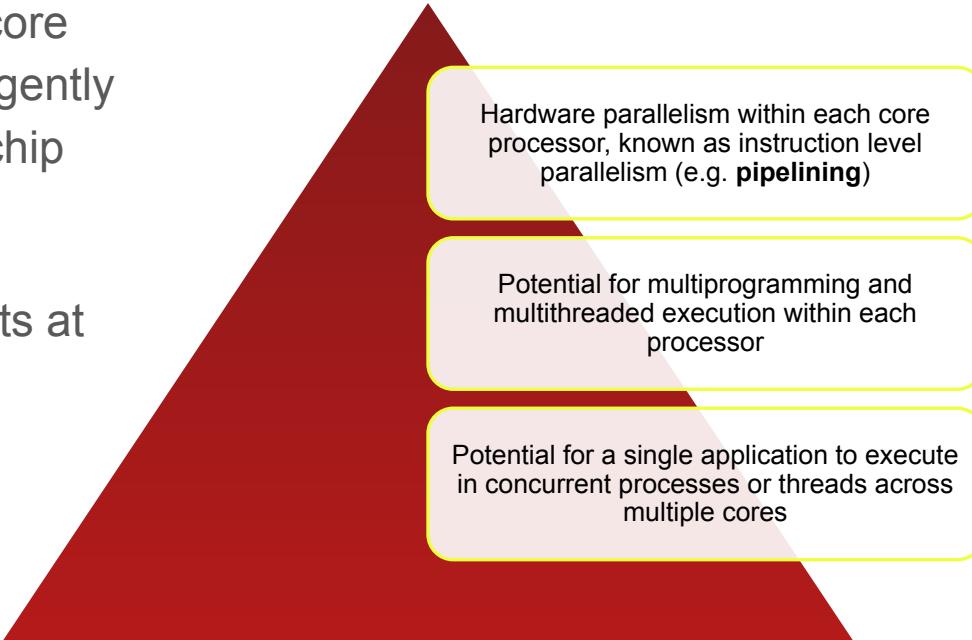
Symmetric Multiprocessor OS Considerations

- A multiprocessor OS must provide all the functionality of a multiprogramming system plus additional features to accommodate multiple processors
- **Key design issues:**



Parallelism and Multicore OS Considerations

- The design challenge for a many-core multicore system is to efficiently harness the multicore processing power and intelligently manage the substantial on-chip resources efficiently
- Potential for parallelism exists at three levels:



Instruction Level Parallelism (ILP): Pipelining

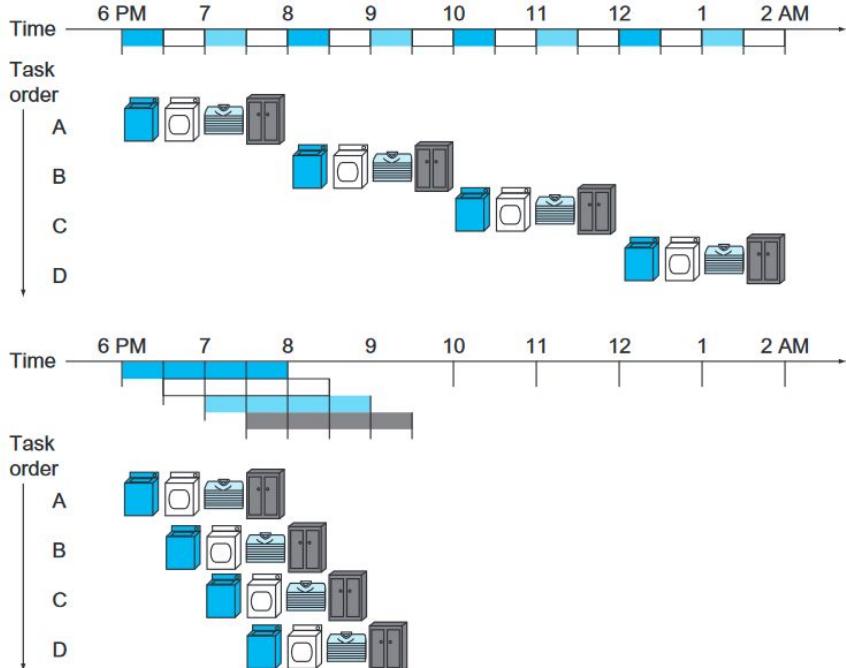


FIGURE 4.25 The laundry analogy for pipelining. Ann, Brian, Cathy, and Don each have dirty clothes to be washed, dried, folded, and put away. The washer, dryer, “folder,” and “storeroom” each take 30 minutes for their task. Sequential laundry takes 8 hours for 4 loads of wash, while pipelined laundry takes just 3.5 hours. We show the pipeline stage of different loads over time by showing copies of the four resources on this two-dimensional time line, but we really have just one of each resource.

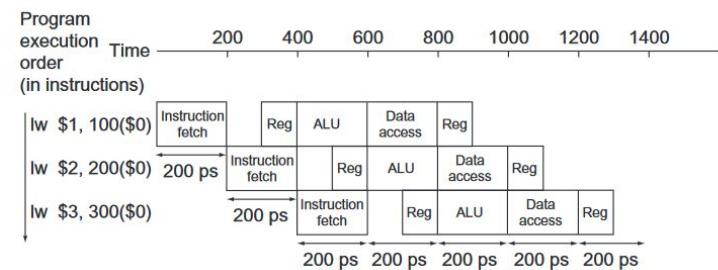
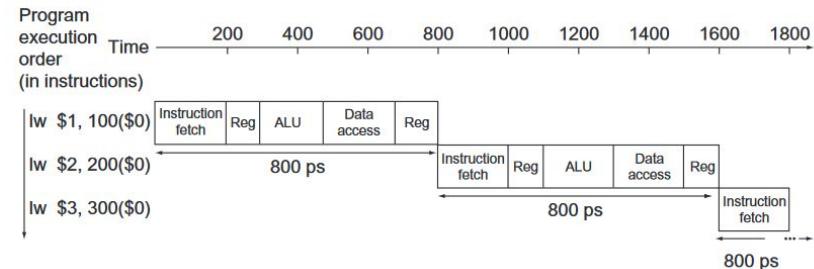


FIGURE 4.27 Single-cycle, nonpipelined execution in top versus pipelined execution in bottom. Both use the same hardware components, whose time is listed in Figure 4.26. In this case, we see a fourfold speed-up on average time between instructions, from 800 ps down to 200 ps. Compare this figure to Figure 4.25. For the laundry, we assumed all stages were equal. If the dryer were slowest, then the dryer stage would set the stage time. The pipeline stage times of a computer are also limited by the slowest resource, either the ALU operation or the memory access. We assume the write to the register file occurs in the first half of the clock cycle and the read from the register file occurs in the second half. We use this assumption throughout this chapter.

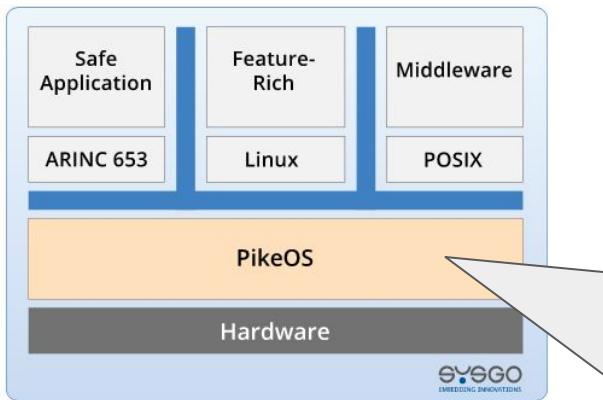
Virtualization and Clouds

- Traditional cloud computing is about
 - IaaS (Infrastructure as a Service)
 - Virtual machines, storage, communication and balancing, etc.
 - PaaS (Platform as a Service)
 - Runtimes, servers, etc. and the related development tools
 - SaaS (Software as a Service)
 - Basically, on-demand software
- Also, cluster computing uses layers
 - Technically many such setup are based on virtualization
- Virtualization: using virtual (abstract) resources instead of the concrete resources (on which they are based)
- Allows one or more cores to be dedicated to a particular process and then leave the processor alone to devote its efforts to that process
- Multicore OS could then act as a hypervisor that makes a high-level decision to allocate cores to applications but does little in the way of resource allocation beyond that

Virtualization - Hypervisor

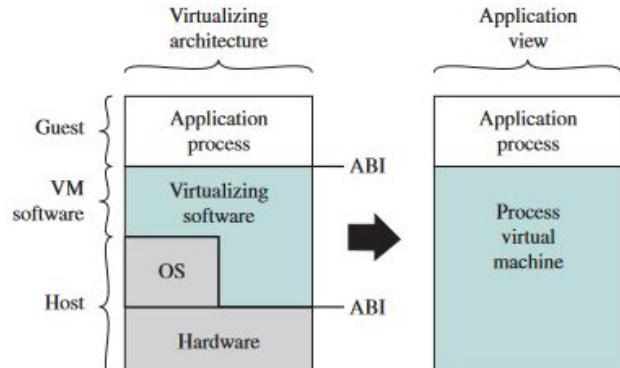
The virtual machine monitor (VMM), or **hypervisor**

- runs on top of (or is incorporated into) the host OS
- supports VMs, which are emulated hardware devices
- each VM runs a separate OS
- handles each operating system's communications with the processor, the storage medium, and the network
- hands off the processor control to a virtual OS on a VM

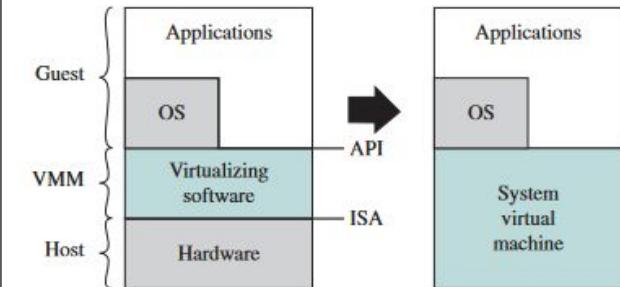


PikeOS is a combination of hypervisor and real-time operating system and is ARINC 653 standard compliant.

ARINC 653 standard defines virtualization in terms of static resources (memory, I/O) as well as processor time. It also defines an API suitable for Avionics concepts. The Airbus A380 and the Boeing A787 are examples of the IMA concept success.



(a) Process VM



(b) System VM

Figure 2.14 Process and System Virtual Machines

Traditional UNIX Systems

- Developed at Bell Labs and became operational on a PDP-7 in 1970
- The first notable milestone was porting the UNIX system from the PDP-7 to the PDP-11
- First showed that UNIX would be an OS for all computers
- **Next milestone was rewriting UNIX in the programming language C**
- Demonstrated the advantages of using a high-level language for system code
- Was described in a technical journal for the first time in 1974
- First widely available version outside Bell Labs was Version 6 in 1976
- Version 7, released in 1978, is the ancestor of most modern UNIX systems
- Most important of the non-AT&T systems was UNIX BSD (Berkeley Software Distribution), running first on PDP and then on VAX computers

The UNIX Time-Sharing System*

D. M. Ritchie and K. Thompson

ABSTRACT

Unix is a general-purpose, multi-user, interactive operating system for the larger Digital Equipment Corporation PDP-11 and the Interdata 8/32 computers. It offers a number of features seldom found even in larger operating systems, including

- i A hierarchical file system incorporating demountable volumes,
- ii Compatible file, device, and inter-process I/O,
- iii The ability to initiate asynchronous processes,
- iv System command language selectable on a per-user basis,
- v Over 100 subsystems including a dozen languages,
- vi High degree of portability.

This paper discusses the nature and implementation of the file system and of the user command interface.

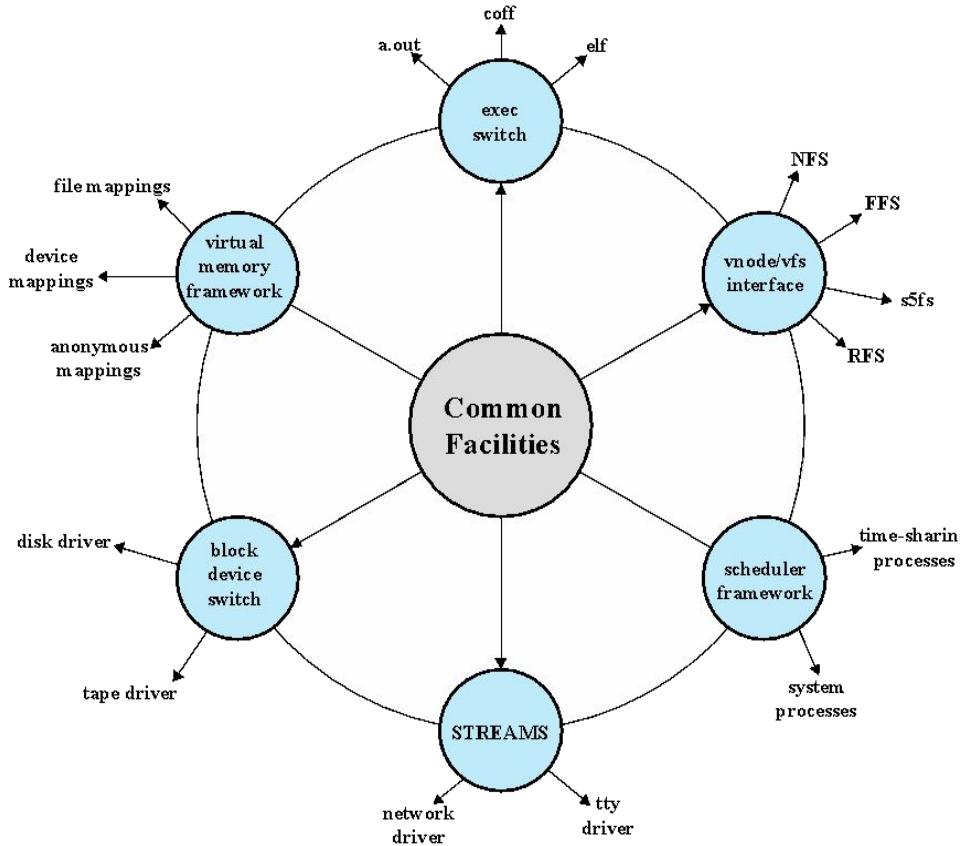
I. INTRODUCTION

There have been four versions of the Unix time-sharing system.

12 The earliest (circa 1969-70) ran on the Digital Equipment Corporation PDP-7 and -9 computers. The second version ran on the unprotected PDP-11/20 computer. The third incorporated multiprogramming and ran on the PDP-11/34, /40, /45, /60, and /70 computers; it is the one described in the previously published version of this paper, and is also the most widely used today. This paper describes only the fourth, current system that runs on the PDP-11/70 and the Interdata 8/32 computers. In fact, the differences among the various systems is rather small; most of the revisions made to the originally published version of this paper, aside from those concerned with style, had to do with details of the implementation of the file system.

<https://www.bell-labs.com/usr/dmr/www/cacm.pdf>

Modern Unix Kernel



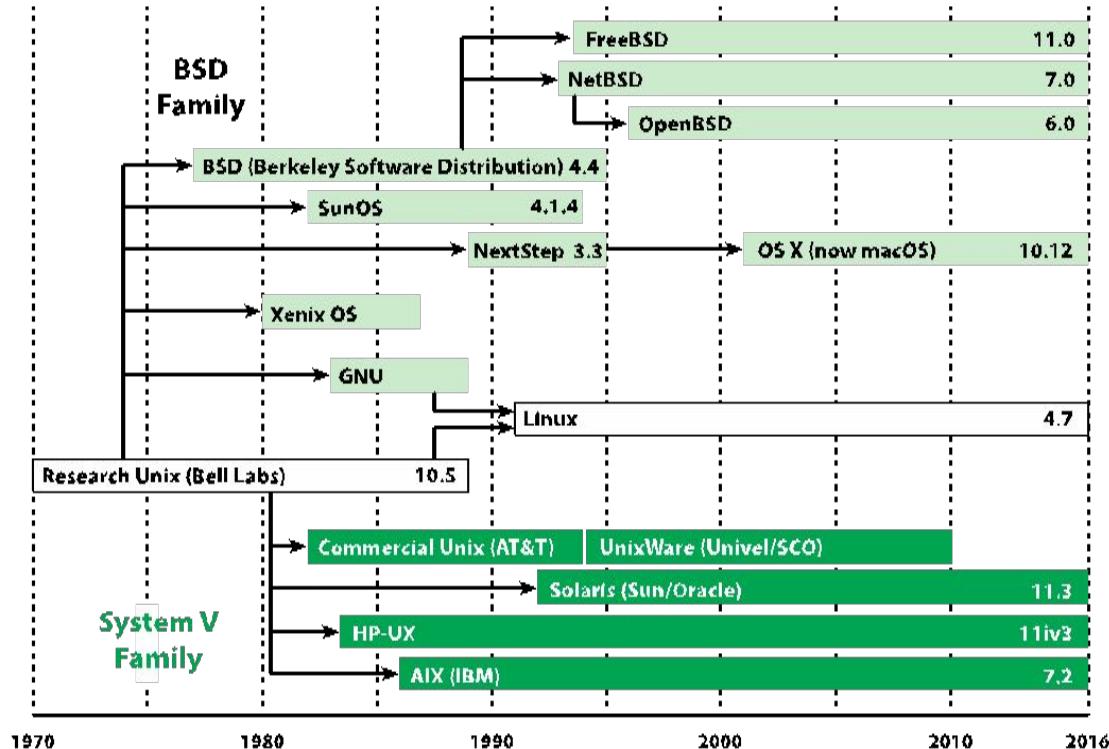
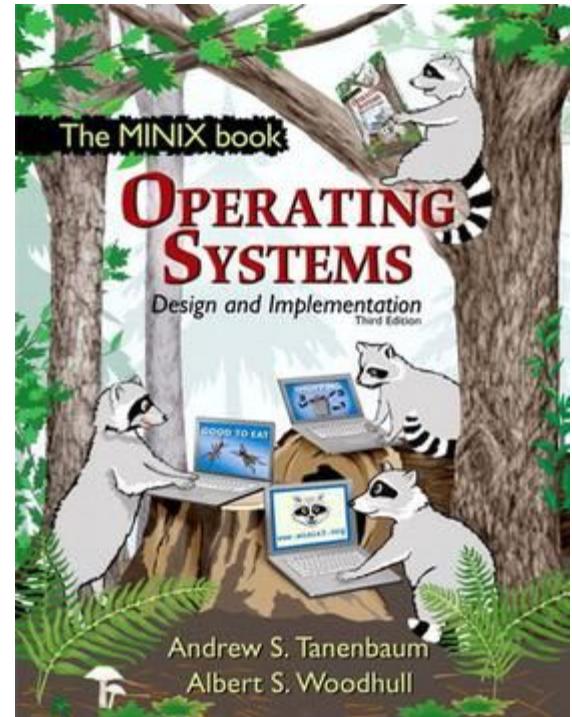


Figure 2.17 Unix Family Tree

LINUX Overview

- Started out as a UNIX variant for the IBM PC
- Linus Torvalds, a Finnish student of computer science, wrote the initial version
- Linux was first posted on the Internet in 1991
- Today it is a full-featured UNIX system that runs on virtually all platforms
- Is free and the source code is available
- Key to the success of Linux has been the availability of free software packages under the auspices of the Free Software Foundation (FSF)
- Highly modular and easily configured



Modular Structure – Loadable Modules

- Linux
 - does not use a microkernel approach,
 - it achieves many of the potential advantages of the approach by means of its particular modular architecture
- Linux is structured as a collection of modules, a number of which can be automatically loaded and unloaded on demand
- Relatively independent blocks
- A module is an object file whose code can be linked to and unlinked from the kernel at runtime
- A module is executed in kernel mode on behalf of the current process
- Have two important characteristics:
 - Dynamic linking
 - Stackable modules

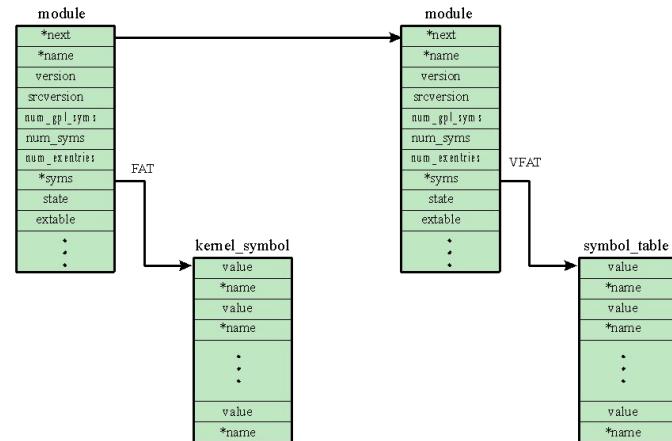


Figure 2.18 Example List of Linux Kernel Modules

Linux kernel components

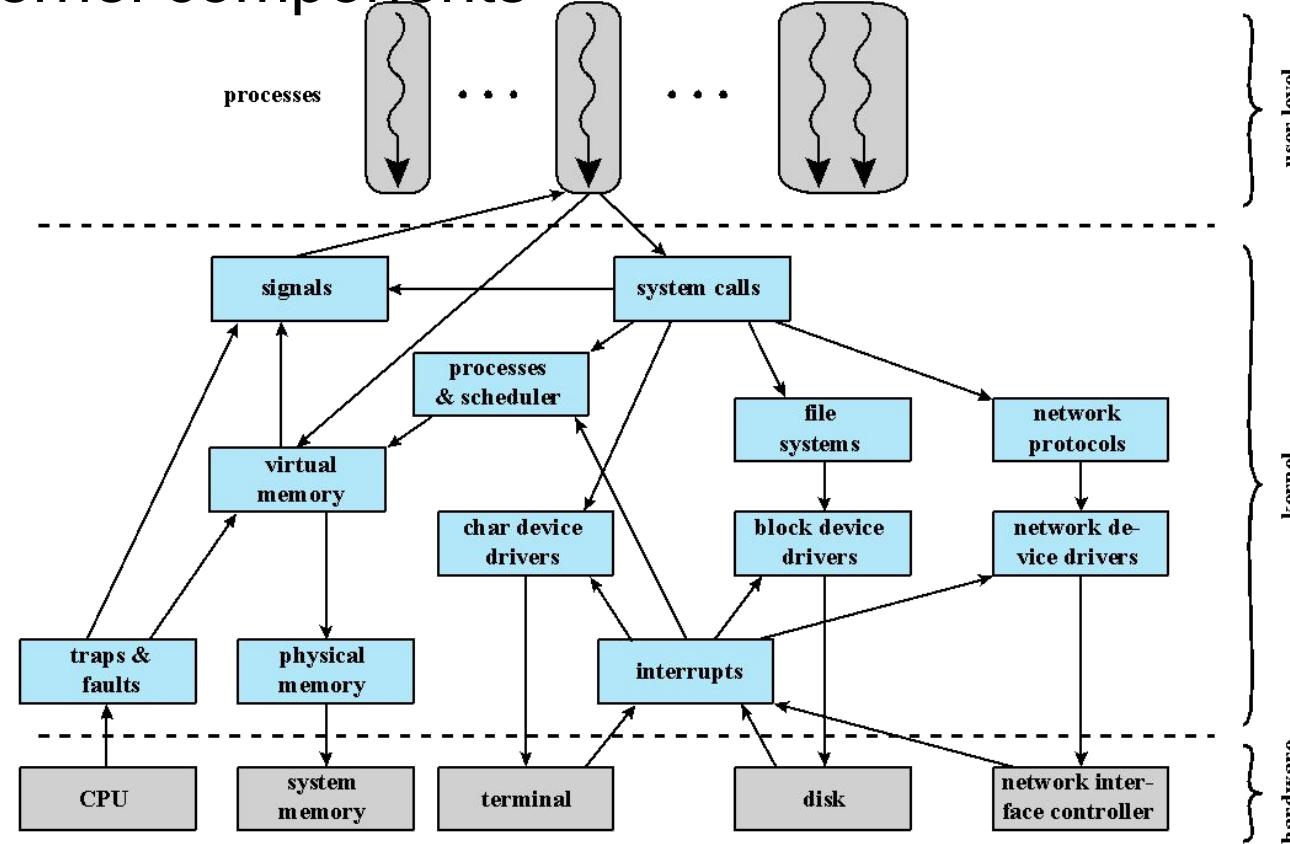


Figure 2.19 Linux Kernel Components

Linux Signals and System Calls

SIGHUP	Terminal hangup	SIGCONT	Continue
SIGQUIT	Keyboard quit	SIGTSTOP	Keyboard stop
SIGTRAP	Trace trap	SIGTTOU	Terminal write
SIGBUS	Bus error	SIGXCPU	CPU limit exceeded
SIGKILL	Kill signal	SIGVTALRM	Virtual alarm clock
SIGSEGV	Segmentation violation	SIGWINCH	Window size unchanged
SIGPIPE	Broken pipe	SIGPWR	Power failure
SIGTERM	Termination	SIGRTMIN	First real-time signal
SIGCHLD	Child status unchanged	SIGRTMAX	Last real-time signal

Filesystem related	
close	Close a file descriptor.
link	Make a new name for a file.
open	Open and possibly create a file or device.
read	Read from file descriptor.
write	Write to file descriptor
Process related	
execve	Execute program.
exit	Terminate the calling process.
getpid	Get process identification.
setuid	Set user identity of the current process.
ptrace	Provides a means by which a parent process may observe and control the execution of another process, and examine and change its core image and registers.
Scheduling related	
sched_getparam	Sets the scheduling parameters associated with the scheduling policy for the process identified by pid .
sched_get_priority_max	Returns the maximum priority value that can be used with the scheduling algorithm identified by policy .
sched_setscheduler	Sets both the scheduling policy (e.g., FIFO) and the associated parameters for the process pid .
sched_rr_get_interval	Writes into the timespec structure pointed to by the parameter tp the round robin time quantum for the process pid .
sched_yield	A process can relinquish the processor voluntarily without blocking via this system call. The process will then be moved to the end of the queue for its static priority and a new process gets to run.

Interprocess Communication (IPC) related	
msgrecv	A message buffer structure is allocated to receive a message. The system call then reads a message from the message queue specified by msqid into the newly created message buffer.
semctl	Performs the control operation specified by cmd on the semaphore set semid .
semop	Performs operations on selected members of the semaphore set semid .
shmat	Attaches the shared memory segment identified by shmid to the data segment of the calling process.
shmctl	Allows the user to receive information on a shared memory segment, set the owner, group, and permissions of a shared memory segment, or destroy a segment.
Socket (networking) related	
bind	Assigns the local IP address and port for a socket. Returns 0 for success and -1 for error.
connect	Establishes a connection between the given socket and the remote socket associated with sockaddr .
gethostname	Returns local host name.
send	Send the bytes contained in buffer pointed to by *msg over the given socket.
setsockopt	Sets the options on a socket
Miscellaneous	
fsync	Copies all in-core parts of a file to disk, and waits until the device reports that all parts are on stable storage.
time	Returns the time in seconds since January 1, 1970.
vhangup	Simulates a hangup on the current terminal. This call arranges for other users to have a "clean" tty at login time.

Summary

- Hardware types
- Operating system objectives and functions
 - User/computer interface
 - Resource manager
 - Evolution of operating systems
 - Serial processing
 - Simple/multiprogrammed/time-sharing batch systems
 - Major achievements
 - Fault tolerance
 - Fundamental concepts
 - Faults
 - OS mechanisms
- Traditional Unix systems
 - History/description
- Linux
 - History
 - Modular structure
 - Kernel components

Process Description and Control

Process Elements

- Resources are made available to multiple applications
 - The processor is switched among multiple applications so all will appear to be progressing
 - The processor and I/O devices can be used efficiently
- Two essential elements of a process are:
 - Code: when the processor begins to execute the program code, we refer to this executing entity as a process
 - Data: which may be shared with other processes that are executing the same program

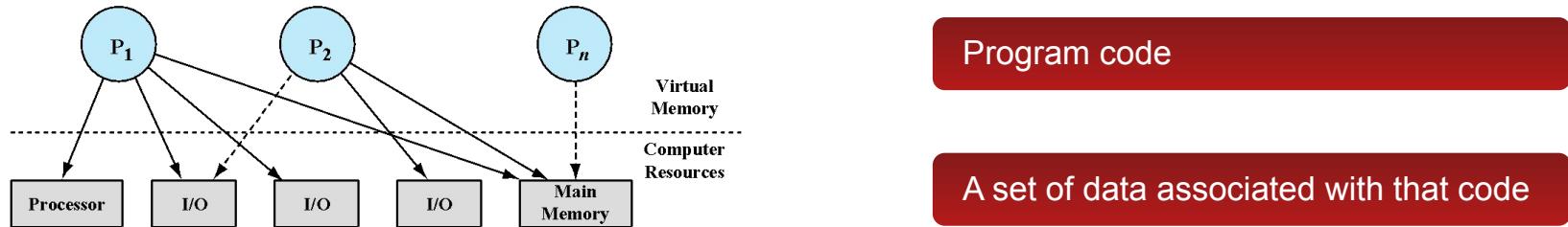


Figure 3.10 Processes and Resources (resource allocation at one snapshot in time)

Tables within the OS

- Memory tables:
 - keep track of both main (real) and secondary (virtual) memory
 - processes use some sort of virtual memory and/or simple swapping mechanism
- The memory tables must include the following information:
 - The allocation of main memory to processes
 - The allocation of secondary memory to processes
- I/O tables
 - used by the OS to manage the I/O devices and channels of the computer system.
 - At any given time, an I/O device may be available or assigned to a particular process.
 - If an I/O operation is in progress, the OS needs to know the status of the I/O operation and the location in main memory being used as the source or destination of the I/O transfer.
- File tables
 - provide information about the existence of files, their location on secondary memory, their current status, and other attributes
 - Much, if not all, of this information may be maintained by a file management system
- The OS must maintain process tables to manage processes

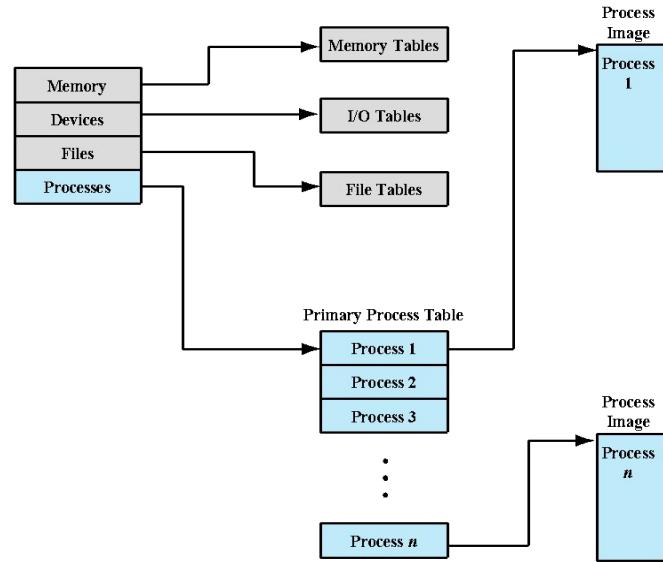
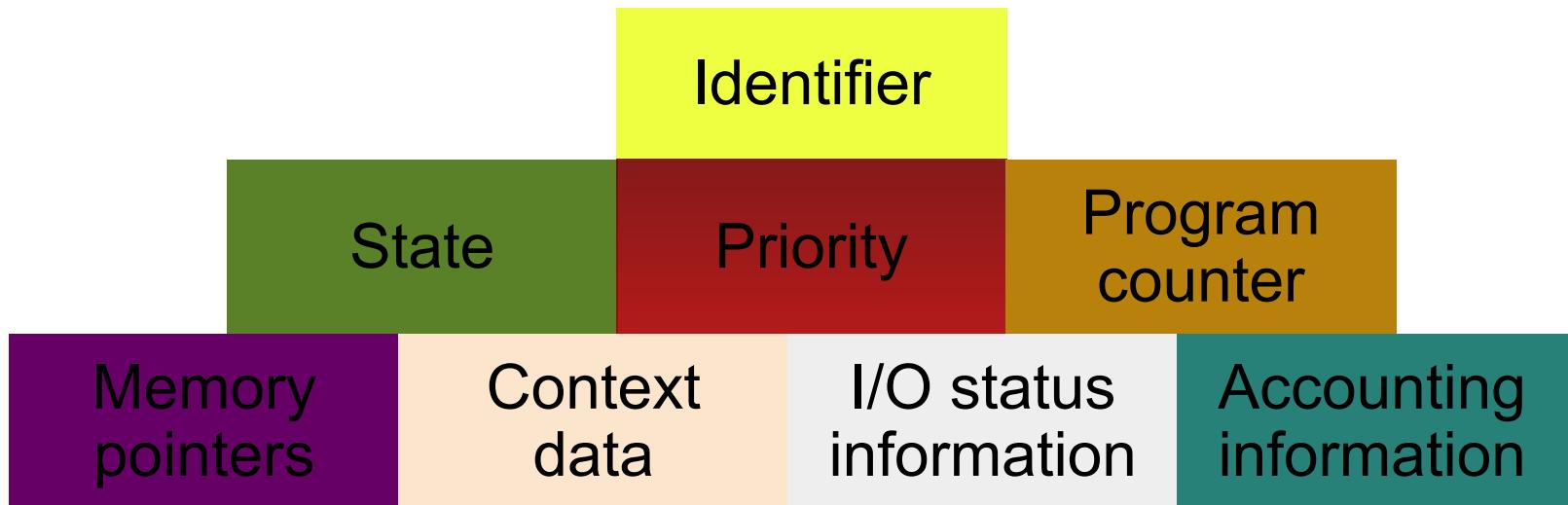


Figure 3.11 General Structure of Operating System Control Tables

Process Elements

- While the program is executing, a process can be uniquely characterized by a number of elements, including:



Process Control Block

- Contains the process elements
- It is possible to interrupt a running process and later resume execution as if the interruption had not occurred
- Created and managed by the operating system
- Key tool that allows support for multiple processes

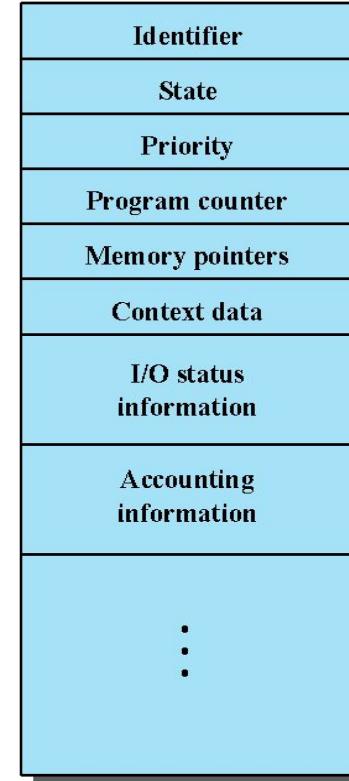


Figure 3.1 Simplified Process Control Block

Process States

Process States

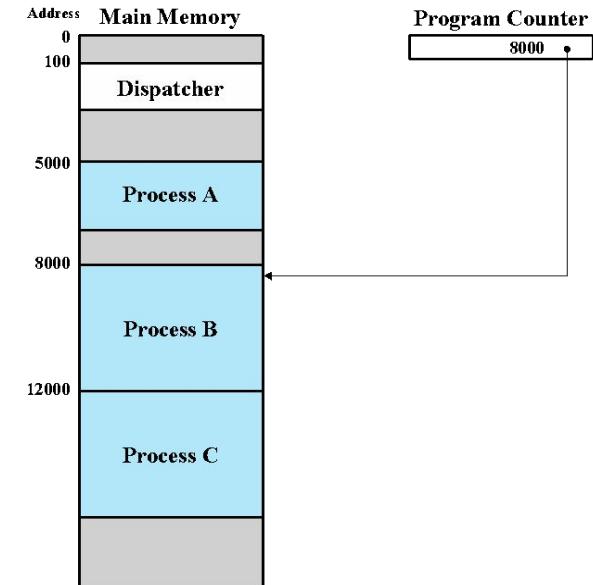
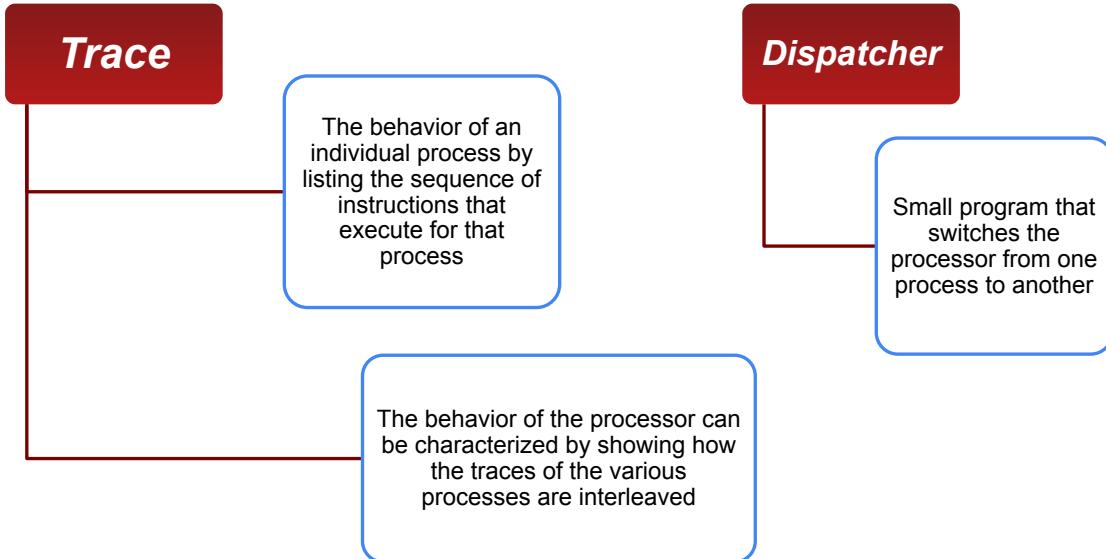
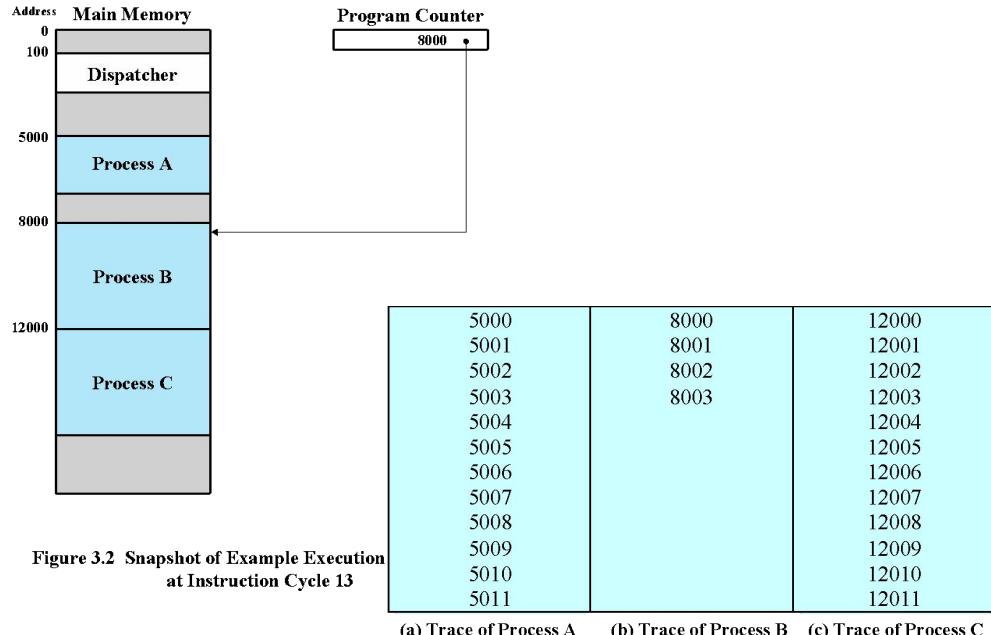


Figure 3.2 Snapshot of Example Execution (Figure 3.4) at Instruction Cycle 13

Process Execution



5000 = Starting address of program of Process A
 8000 = Starting address of program of Process B
 12000 = Starting address of program of Process C

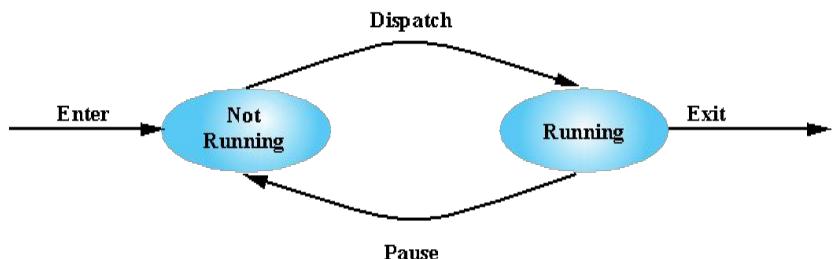
Figure 3.3 Traces of Processes of Figure 3.2

1	5000		
2	5001		
3	5002		
4	5003		
5	5004		
6	5005		
7	100		
8	101		
9	102		
10	103		
11	104		
12	105		
13	8000		
14	8001		
15	8002		
16	8003		
17	100		
18	101		
19	102		
20	103		
21	104		
22	105		
23	12000		
24	12001		
25	12002		
26	12003		
27	12004		
28	12005		
29	100		Timeout
30	101		
31	102		
32	103		
33	104		
34	105		
35	5006		
36	5007		
37	5008		
38	5009		
39	5010		
40	5011		
41	100		Timeout
42	101		
43	102		
44	103		
45	104		
46	105		
47	12006		
48	12007		
49	12008		
50	12009		
51	12010		
52	12011		

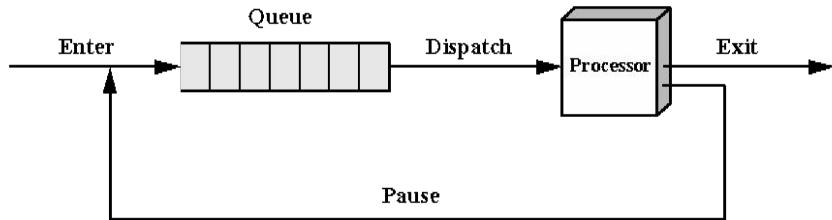
100 = Starting address of dispatcher program
 Shaded areas indicate execution of dispatcher process;
 first and third columns count instruction cycles;
 second and fourth columns show address of instruction being executed

Figure 3.4 Combined Trace of Processes of Figure 3.2

Two-State Process Model



(a) State transition diagram



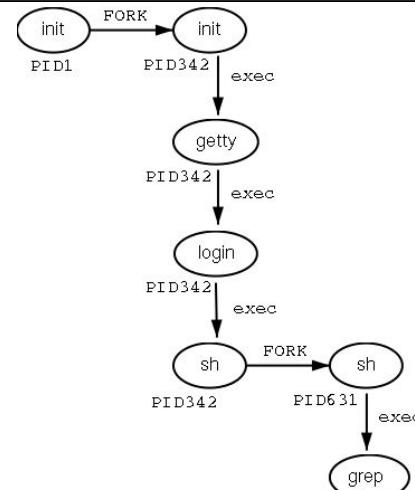
(b) Queuing diagram

Figure 3.5 Two-State Process Model

Process Creation

- Process spawning
 - When the OS creates a process at the explicit request of another process
 - Parent process: is the original, creating, process
 - Child process: is the new process
- Linux
 - An existing process makes an exact copy of itself
 - The child process has the same environment as its parent, only the process ID number is different. This procedure is called **forking**.
 - After the forking process, the **address space** of the child process is overwritten with the new process data. This is done through an **exec** call to the system.
 - Many programs, for instance, **daemonize** their child processes, so they can keep on running when the parent stops or is being stopped.

New batch job	The OS is provided with a batch job control stream, usually on tape or disk. When the OS is prepared to take on new work, it will read the next sequence of job control commands.
Interactive logon	A user at a terminal logs on to the system.
Created by OS to provide a service	The OS can create a process to perform a function on behalf of a user program, without the user having to wait (e.g., a process to control printing).
Spawned by existing process	For purposes of modularity or to exploit parallelism, a user program can dictate the creation of a number of processes.



Process Termination

- There must be a means for a process to indicate its completion
- A batch job should include a HALT instruction or an explicit OS service call for termination
- For an interactive application, the action of the user will indicate when the process is completed (e.g. log off, quitting an application)
- Linux:
 - Program returns its exit status to the parent
 - This exit status is a number providing the results of the program's execution
 - The system of returning information upon executing a job has its origin in the C programming language in which UNIX has been written.

Normal completion	The process executes an OS service call to indicate that it has completed running.
Time limit exceeded	The process has run longer than the specified total time limit. There are a number of possibilities for the type of time that is measured. These include total elapsed time ("wall clock time"), amount of time spent executing, and, in the case of an interactive process, the amount of time since the user last provided any input.
Memory unavailable	The process requires more memory than the system can provide.
Bounds violation	The process tries to access a memory location that it is not allowed to access.
Protection error	The process attempts to use a resource such as a file that it is not allowed to use, or it tries to use it in an improper fashion, such as writing to a read-only file.
Arithmetic error	The process tries a prohibited computation, such as division by zero, or tries to store numbers larger than the hardware can accommodate.
Time overrun	The process has waited longer than a specified maximum for a certain event to occur.
I/O failure	An error occurs during input or output, such as inability to find a file, failure to read or write after a specified maximum number of tries (when, for example, a defective area is encountered on a tape), or invalid operation (such as reading from the line printer).
Invalid instruction	The process attempts to execute a nonexistent instruction (often a result of branching into a data area and attempting to execute the data).
Privileged instruction	The process attempts to use an instruction reserved for the operating system.
Data misuse	A piece of data is of the wrong type or is not initialized.
Operator or OS intervention	For some reason, the operator or the operating system has terminated the process (e.g., if a deadlock exists).
Parent termination	When a parent terminates, the operating system may automatically terminate all of the offspring of that parent.
Parent request	A parent process typically has the authority to terminate any of its offspring.

Five-State Process Model

- **New:** A process that has just been created but has not yet been admitted to the pool of executable processes by the OS.
Typically, a new process has not yet been loaded into main memory, although its process control block has been created.
- **Ready:** A process that is prepared to execute when given the opportunity.
- **Running:** The process that is currently being executed.
- **Blocked/Waiting:** A process that cannot execute until some event occurs, such as the completion of an I/O operation.
- **Exit:** A process that has been released from the pool of executable processes by the OS, either because it halted or because it aborted for some reason.

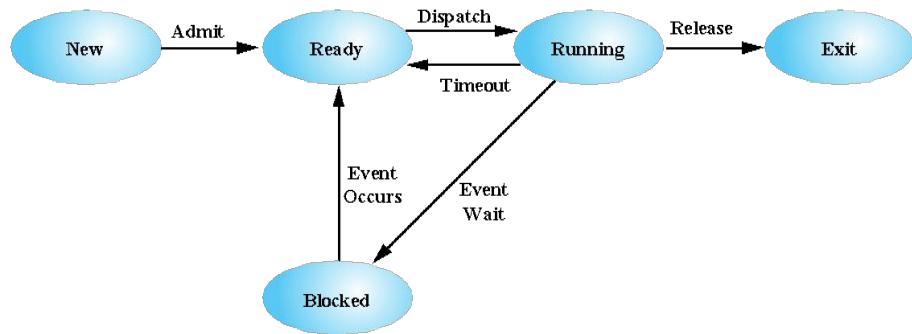


Figure 3.6 Five-State Process Model

Example: Trace of Process Execution

1	5000		27	12004
2	5001		28	12005
3	5002			----- Timeout
4	5003		29	100
5	5004		30	101
6	5005		31	102
			32	103
			33	104
7	100		34	105
8	101			----- Timeout
9	102		35	5006
10	103		36	5007
11	104		37	5008
12	105		38	5009
13	8000		39	5010
14	8001		40	5011
				----- Timeout
				I/O Request
17	100		41	100
18	101		42	101
19	102		43	102
20	103		44	103
21	104		45	104
22	105		46	105
23	12000		47	12006
24	12001		48	12007
25	12002		49	12008
26	12003		50	12009
			51	12010
			52	12011
				----- Timeout

100 = Starting address of dispatcher program

Shaded areas indicate execution of dispatcher process;
first and third columns count instruction cycles;
second and fourth columns show address of instruction being executed

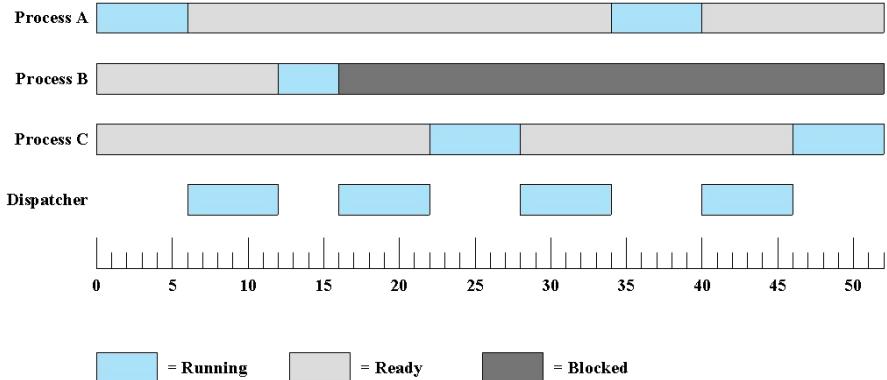
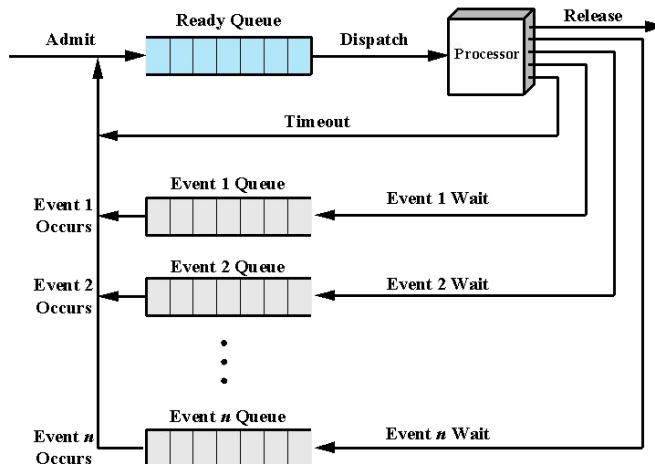
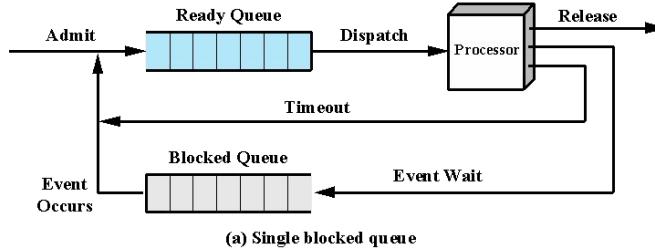


Figure 3.7 Process States for Trace of Figure 3.4

Faster/Efficient Scheduling of Processes

- Example, Linux Scheduler
 - separate run queue for each processor
 - each processor only selects processes from its own queue to run
 - it's possible for one processor to be idle while others have jobs waiting in their run queues
 - Periodically, the queues are rebalanced:
 - if one processor's run queue is too long
 - some processes are moved from it to another processor's queue
- 140 separate queues, one for each priority level
 - Number can be changed at a given site
 - Two sets, active and expired
 - Priorities 0-99 for real-time processes
 - Priorities 100-139 for normal processes

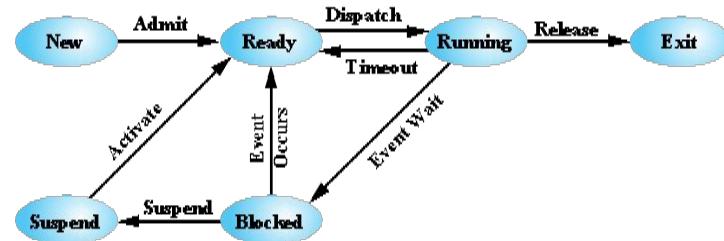


(b) Multiple blocked queues

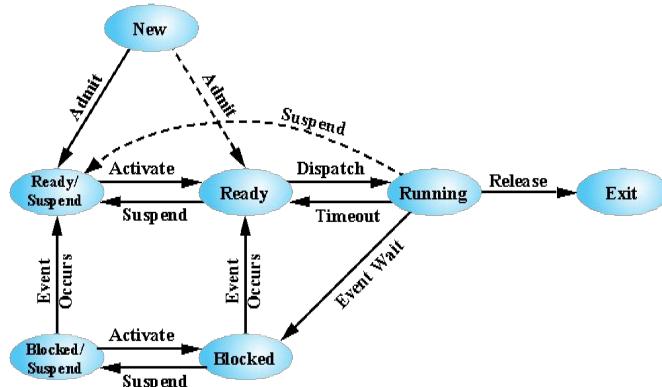
Figure 3.8 Queuing Model for Figure 3.6

Suspended Processes - Swapping

- Swapping
 - is an I/O operation and therefore there is the potential for making the problem worse, not better
 - disk I/O is generally the fastest I/O on a system swapping will usually enhance performance.
- Moving part of all of a process from main memory to disk
 - When none of the processes in main memory is in the Ready state, the OS swaps one of the blocked processes out on to disk into a suspend queue
 - This is a queue of existing processes that have been temporarily kicked out of main memory, or suspended
 - The OS then brings in another process from the suspend queue or it honors a new-process request
 - Execution then continues with the newly arrived process



(a) With One Suspend State



(b) With Two Suspend States

Figure 3.9 Process State Transition Diagram with Suspend States

Characteristics of a Suspended Process

- The process may or may not be waiting on an event
- The process is not immediately available for execution
- The process was placed in a suspended state by an agent: either itself, a parent process, or the OS, for the purpose of preventing its execution
- The process may not be removed from this state until the agent explicitly orders the removal

Swapping	The OS needs to release sufficient main memory to bring in a process that is ready to execute.
Other OS reason	The OS may suspend a background or utility process or a process that is suspected of causing a problem.
Interactive user request	A user may wish to suspend execution of a program for purposes of debugging or in connection with the use of a resource.
Timing	A process may be executed periodically (e.g., an accounting or system monitoring process) and may be suspended while waiting for the next time interval.
Parent process request	A parent process may wish to suspend execution of a descendent to examine or modify the suspended process, or to coordinate the activity of various descendants.

Process Control

Process Control Structures

- A process must include a program or set of programs to be executed
 - A process will consist of at least sufficient memory to hold the programs and data of that process
 - The execution of a program typically involves a stack that is used to keep track of procedure calls and parameter passing between procedures
- Each process has associated with it a number of attributes that are used by the OS for process control
 - The collection of program, data, stack, and attributes is referred to as the process image
 - Process image location will depend on the memory management scheme being used

To manage and control a process the OS must know:

- Where the process is located
- The attributes of the process that are necessary for its management

Typical Elements of a Process Image

User Data

The modifiable part of the user space. May include program data, a user stack area, and programs that may be modified.

User Program

The program to be executed.

Stack

Each process has one or more last-in-first-out (LIFO) stacks associated with it. A stack is used to store parameters and calling addresses for procedure and system calls.

Process Control Block

Data needed by the OS to control the process (see Table 3.5).

Process Control Information

Scheduling and State Information

This is information that is needed by the operating system to perform its scheduling function. Typical items of information:

- **Process state:** Defines the readiness of the process to be scheduled for execution (e.g., running, ready, waiting, halted).
- **Priority:** One or more fields may be used to describe the scheduling priority of the process. In some systems, several values are required (e.g., default, current, highest-allowable)
- **Scheduling-related information:** This will depend on the scheduling algorithm used. Examples are the amount of time that the process has been waiting and the amount of time that the process executed the last time it was running.
- **Event:** Identity of event the process is awaiting before it can be resumed.

Data Structuring

A process may be linked to other process in a queue, ring, or some other structure. For example, all processes in a waiting state for a particular priority level may be linked in a queue. A process may exhibit a parent-child (creator-created) relationship with another process. The process control block may contain pointers to other processes to support these structures.

Interprocess Communication

Various flags, signals, and messages may be associated with communication between two independent processes. Some or all of this information may be maintained in the process control block.

Process Privileges

Processes are granted privileges in terms of the memory that may be accessed and the types of instructions that may be executed. In addition, privileges may apply to the use of system utilities and services.

Memory Management

This section may include pointers to segment and/or page tables that describe the virtual memory assigned to this process.

Resource Ownership and Utilization

Resources controlled by the process may be indicated, such as opened files. A history of utilization of the processor or other resources may also be included; this information may be needed by the scheduler.

Process Identification

- Each process is assigned a unique numeric identifier
 - Otherwise there must be a mapping that allows the OS to locate the appropriate tables based on the process identifier
- Tables controlled by the OS may use process identifiers to cross-reference process tables
- Memory tables may be organized to provide a map of main memory with an indication of which process is assigned to each region
 - Similar references will appear in I/O and file tables
 - When processes communicate with one another, the process identifier informs the OS of the destination of a particular communication
- When processes are allowed to create other processes, identifiers indicate the parent and descendants of each process

Process Identification

Identifiers

Numeric identifiers that may be stored with the process control block include

- Identifier of this process
- Identifier of the process that created this process (parent process)
- User identifier

Processor State Information

User-Visible Registers

A user-visible register is one that may be referenced by means of the machine language that the processor executes while in user mode. Typically, there are from 8 to 32 of these registers, although some RISC implementations have over 100.

Control and Status Registers

These are a variety of processor registers that are employed to control the operation of the processor. These include

- **Program counter:** Contains the address of the next instruction to be fetched
- **Condition codes:** Result of the most recent arithmetic or logical operation (e.g., sign, zero, carry, equal, overflow)
- **Status information:** Includes interrupt enabled/disabled flags, execution mode

Stack Pointers

Each process has one or more last-in-first-out (LIFO) system stacks associated with it. A stack is used to store parameters and calling addresses for procedure and system calls. The stack pointer points to the top of the stack.

Operating Systems

CS-C3140, Lecture 4

Alexandru Paler

Announcements

- Exam: 13 December 2022, 17:00, U2
 - was 7 December 2022
 - due to overlapping project meetings
 - details about exact form will be announced asap
- Assignments will be opened each Sunday 23:59 -> deadline 7(14) days

Processor State Information

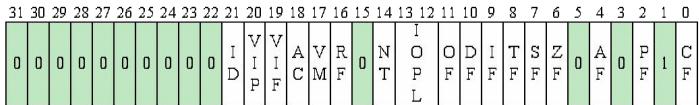
Consists of
the contents
of processor
registers

- User-visible registers
- Control and status registers
- Stack pointers

**Program
status
word
(PSW)**

- Contains condition codes plus other status information
- EFLAGS register is an example of a PSW used by any OS running on an x86 processor

x86 EFLAGS Register Bits



X ID = Identification flag
 X VIP = Virtual interrupt pending
 X VIF = Virtual interrupt flag
 X AC = Alignment check
 X VM = Virtual 8086 mode
 X RF = Resume flag
 X NT = Nested task flag
 X IOPL = I/O privilege level
 S OF = Overflow flag

S Indicates a Status Flag
 C Indicates a Control Flag
 X Indicates a System Flag
 Shaded bits are reserved

Figure 3.12 x86 EFLAGS Register

Status Flags (condition codes)	
AF (Auxiliary carry flag)	Represents carrying or borrowing between half-bytes of an 8-bit arithmetic or logic operation using the AL register.
CF (Carry flag)	Indicates carrying out or borrowing into the leftmost bit position following an arithmetic operation. Also modified by some of the shift and rotate operations.
OF (Overflow flag)	Indicates an arithmetic overflow after an addition or subtraction.
PF (Parity flag)	Parity of the result of an arithmetic or logic operation. 1 indicates even parity; 0 indicates odd parity.
SF (Sign flag)	Indicates the sign of the result of an arithmetic or logic operation.
ZF (Zero flag)	Indicates that the result of an arithmetic or logic operation is 0.
Control Flag	
DF (Direction flag)	Determines whether string processing instructions increment or decrement the 16-bit half-registers SI and DI (for 16-bit operations) or the 32-bit registers ESI and EDI (for 32-bit operations).
System Flags (should not be modified by application programs)	
AC (Alignment check)	Set if a word or doubleword is addressed on a nonword or nondoubleword boundary.
ID (Identification flag)	If this bit can be set and cleared, this processor supports the CPUID instruction. This instruction provides information about the vendor, family, and model.
RF (Resume flag)	Allows the programmer to disable debug exceptions so that the instruction can be restarted after a debug exception without immediately causing another debug exception.
IOPL (I/O privilege level)	When set, causes the processor to generate an exception on all accesses to I/O devices during protected mode operation.
IF (Interrupt enable flag)	When set, the processor will recognize external interrupts.
TF (Trap flag)	When set, causes an interrupt after the execution of each instruction. This is used for debugging.
NT (Nested task flag)	Indicates that the current task is nested within another task in protected mode operation.
VM (Virtual 8086 mode)	Allows the programmer to enable or disable virtual 8086 mode, which determines whether the processor runs as an 8086 machine.
VIP (Virtual interrupt pending)	Used in virtual 8086 mode to indicate that one or more interrupts are awaiting service.
VIF (Virtual interrupt flag)	Used in virtual 8086 mode instead of IF.

Process Control Information

- The most important data structure in an OS
 - Contains all of the information about a process that is needed by the OS
 - Blocks are read and/or modified by virtually every module in the OS
 - Defines the state of the OS
- **Difficulty is not access, but protection**
 - A bug in a single routine could damage process control blocks, which could destroy the system's ability to manage the affected processes
 - A design change in the structure or semantics of the process control block could affect a number of modules in the OS

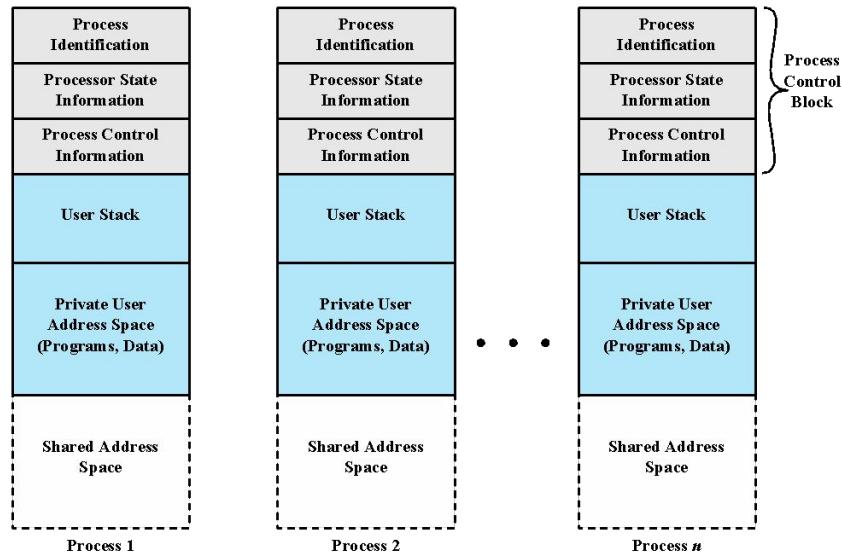
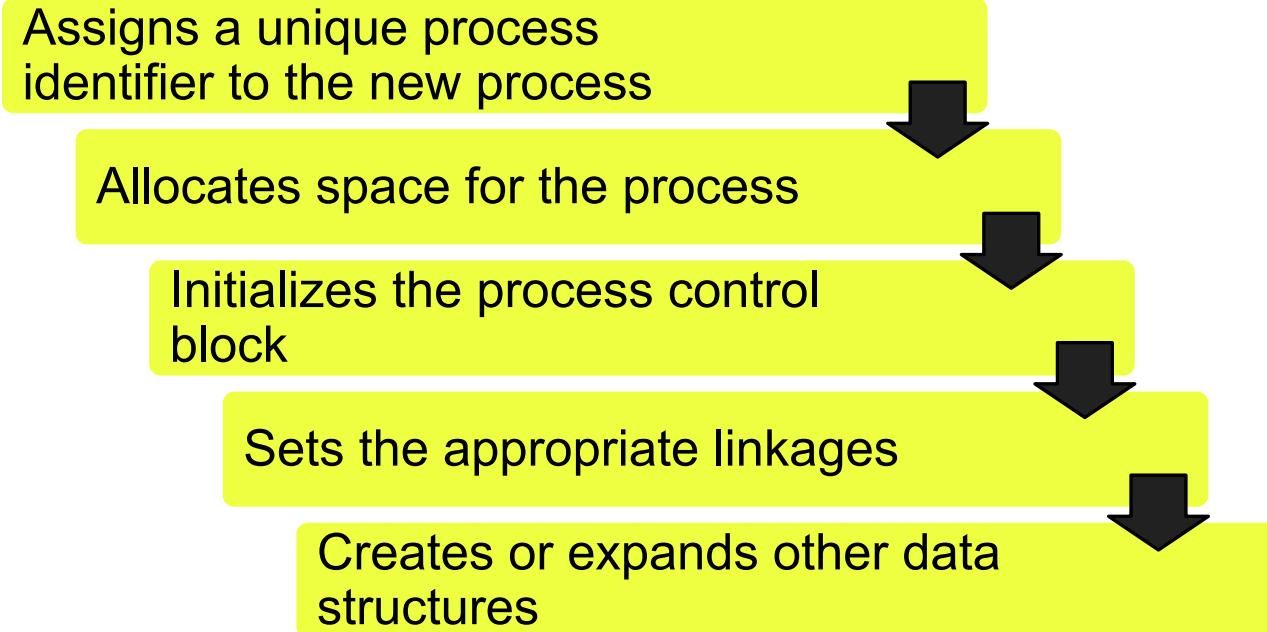


Figure 3.13 User Processes in Virtual Memory

Process Creation

- Once the OS decides to create a new process it:



Interrupting the Execution of a Process

- **Interrupt**
 - Due to some sort of event that is external to and independent of the currently running process
 - Clock interrupt
 - I/O interrupt
 - Memory fault
 - Time slice
 - The maximum amount of time that a process can execute before being interrupted
- **Trap**
 - An error or exception condition generated within the currently running process
 - OS determines if the condition is fatal
 - Moved to the Exit state and a process switch occurs
 - Action will depend on the nature of the error the design of the OS

Mechanism	Cause	Use
Interrupt	External to the execution of the current instruction	Reaction to an asynchronous external event
Trap	Associated with the execution of the current instruction	Handling of an error or an exception condition
Supervisor call	Explicit request	Call to an operating system function

Modes of Execution and Switching

- User Mode

- Less-privileged mode
- User programs typically execute in this mode

- System Mode

- More-privileged mode
- Also referred to as control mode or kernel mode
- Kernel of the operating system

If no interrupts are pending the processor:



Proceeds to the fetch stage and fetches the next instruction of the current program in the current process

If an interrupt is pending the processor:

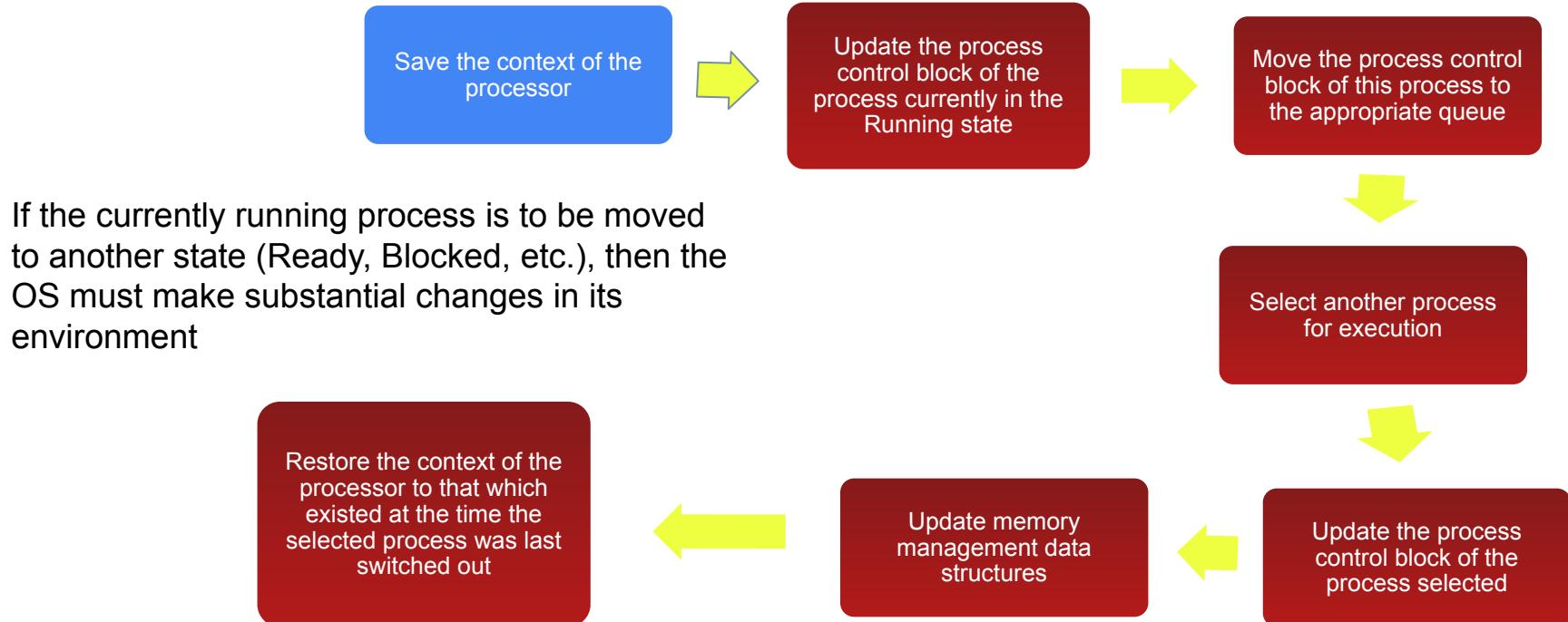


Sets the program counter to the starting address of an interrupt handler program

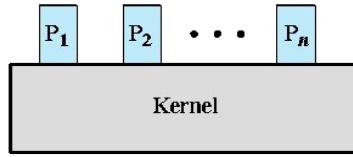


Switches from user mode to kernel mode so that the interrupt processing code may include privileged instructions

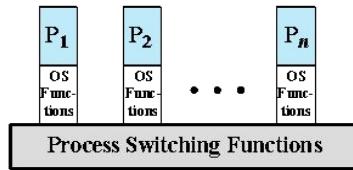
Change of Process State



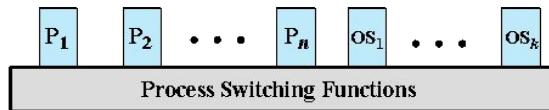
Execution of the Operating System



(a) Separate kernel



(b) OS functions execute within user processes



(c) OS functions execute as separate processes

Figure 3.15 Relationship Between Operating System and User Processes

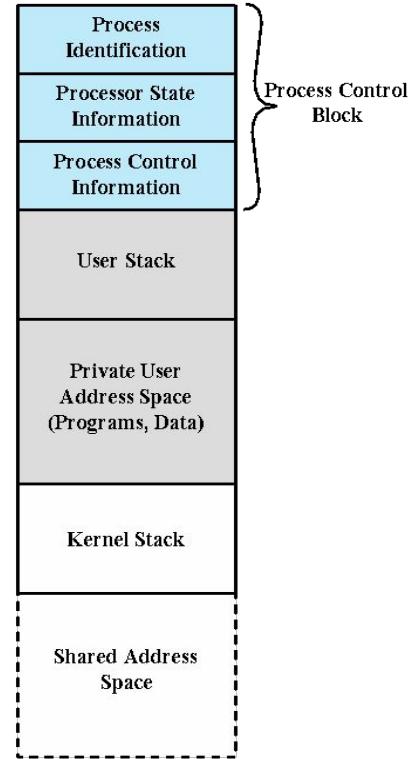


Figure 3.16 Process Image: Operating System Executes Within User Space

Summary

- What is a process?
 - Background
 - Processes and process control blocks
- Process states
 - Two-state process model
 - Creation and termination
 - Five-state model
 - Suspended processes
- Process description
 - Operating system control structures
 - Process control structures
- Process control
 - Modes of execution
 - Process creation
 - Process switching
- Execution of the operating system
 - Nonprocess kernel
 - Execution within user processes
 - Process-based operating system

Processes vs. Threads

Processes and Threads

- The unit of dispatching is referred to as a **thread or lightweight process**
- The unit of resource ownership is referred to as a **process or task**
- **Multithreading** - The ability of an OS to support multiple, concurrent paths of execution within a single process

Resource Ownership

- Process includes a virtual address space to hold the process image
- The OS performs a protection function to prevent unwanted interference between processes with respect to resources

Scheduling/Execution

- Follows an execution path that may be interleaved with other processes
- A process has an execution state (Running, Ready, etc.) and a dispatching priority, and is the entity that is scheduled and dispatched by the OS

Single- and Multi-Threaded Approaches

- A single thread of execution per process, in which the concept of a thread is not recognized, is referred to as a single-threaded approach
- MS-DOS: single process and single thread
- A Java run-time environment is an example of a system of one process with multiple threads

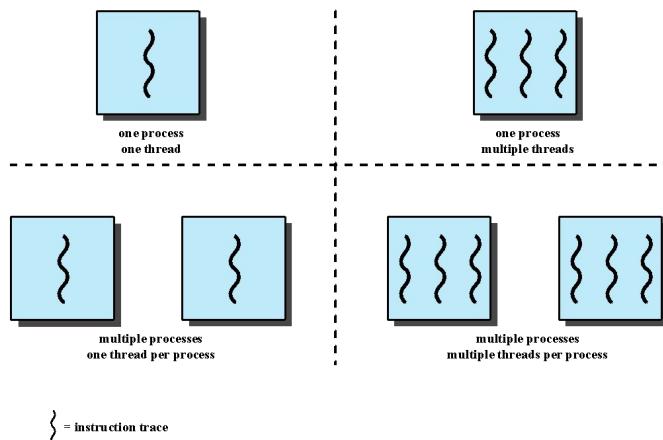


Figure 4.1 Threads and Processes

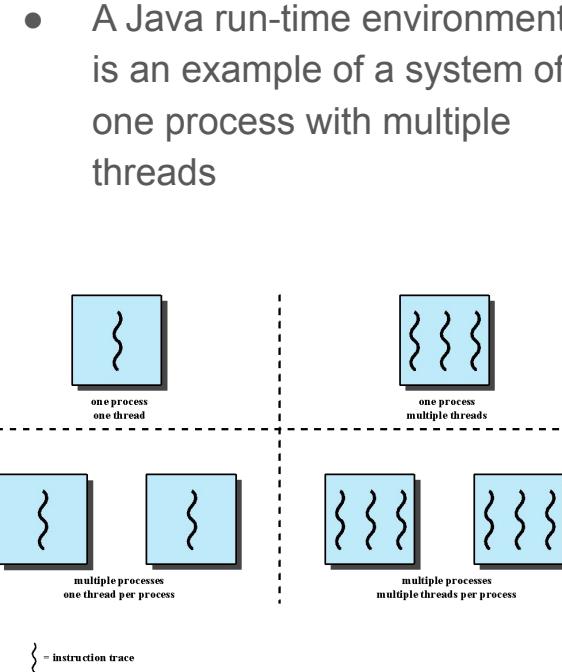


Figure 4.1 Threads and Processes

One or More Threads in a Process

Process - Defined in a multithreaded environment as “the unit of resource allocation and a unit of protection”:

- Associated with processes:
- A virtual address space that holds the process image
- Protected access to:
 - Processors
 - Other processes (for interprocess communication)
 - Files
 - I/O resources (devices and channels)

Each thread has:

- An execution state (Running, Ready, etc.)
- A saved thread context when not running
- An execution stack
- Some per-thread static storage for local variables
- Access to the memory and resources of its process, shared with all other threads in that process

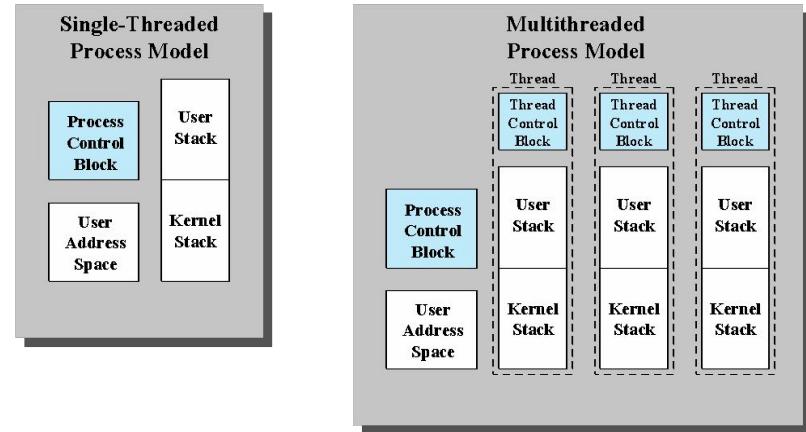
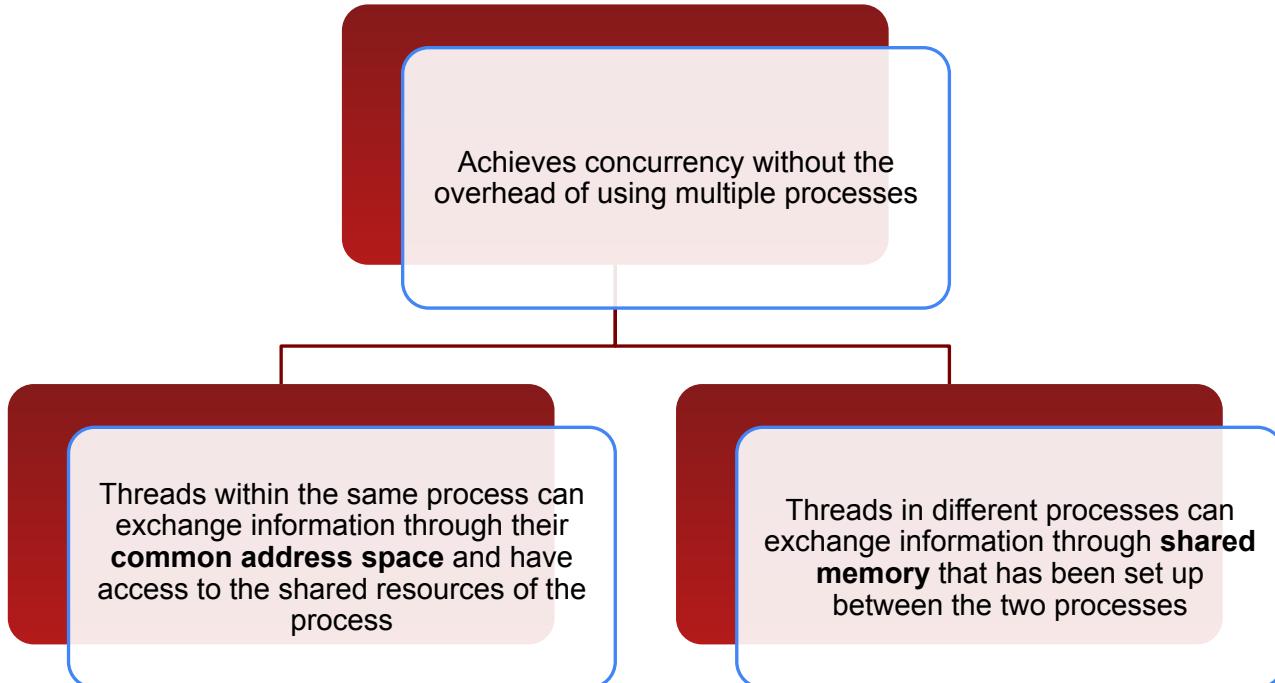
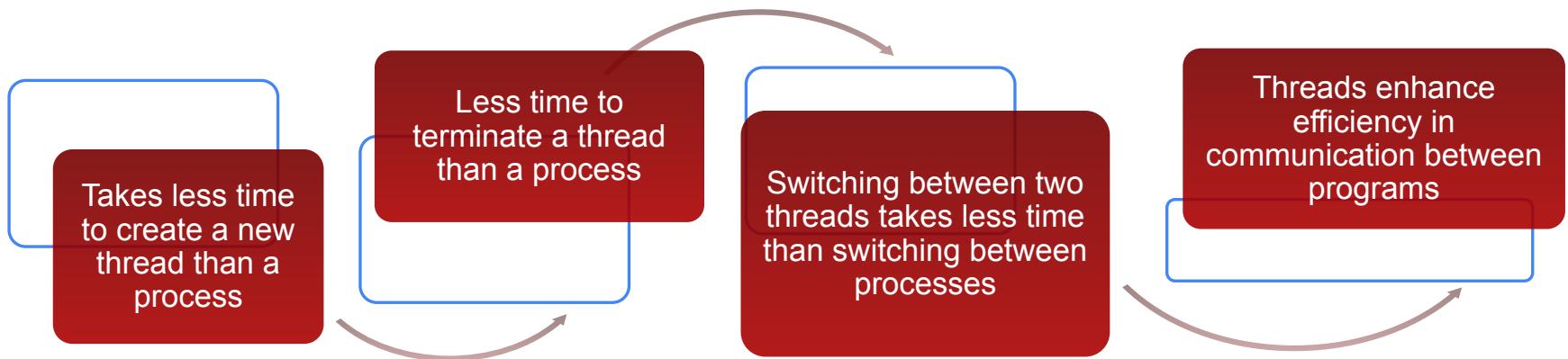


Figure 4.2 Single Threaded and Multithreaded Process Models

Multithreading

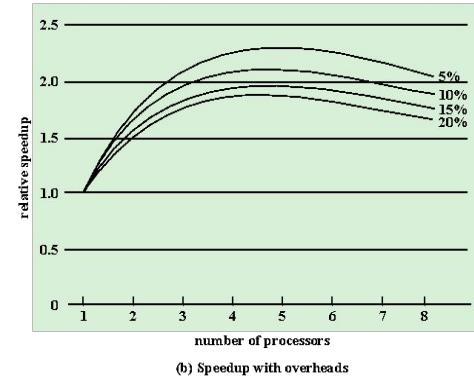
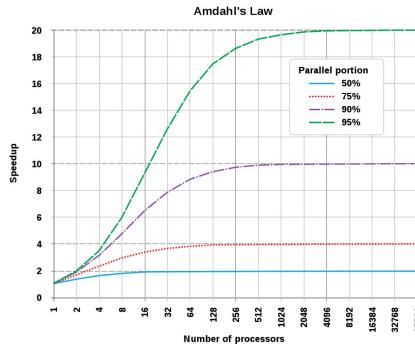
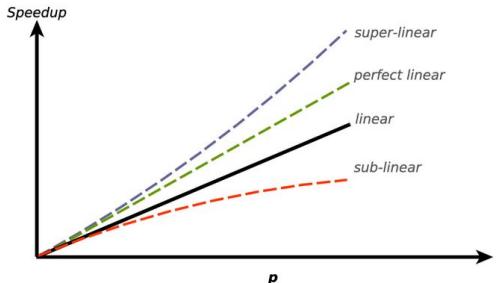


Key Benefits of Threads

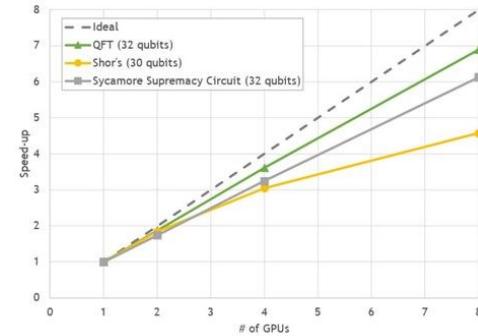
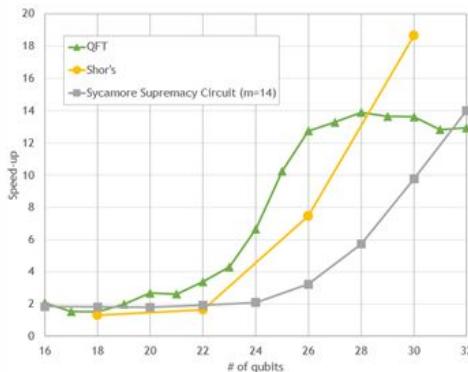


Applications That Benefit from Multi-Threading

- Multithreaded native applications - small number of highly threaded processes
- Multiprocess applications - have many single-threaded processes
- Examples:
 - Java applications benefit from multicore technology
 - Multi-instance applications: Multiple instances of the application in parallel



$$S_{\text{latency}}(s) = \frac{1}{(1-p) + \frac{p}{s}}$$



Relationship between Threads and Processes

Threads:Processes	Description	Example Systems
1:1	Each thread of execution is a unique process with its own address space and resources.	Traditional UNIX implementations
M:1	A process defines an address space and dynamic resource ownership. Multiple threads may be created and executed within that process.	Windows NT, Solaris, Linux, OS/2, OS/390, MACH
1:M	A thread may migrate from one process environment to another. This allows a thread to be easily moved among distinct systems.	Ra (Clouds), Emerald
M:N	Combines attributes of M:1 and 1:M cases.	TRIX

Threads and Their Use in a Single-User System

- Foreground and background work
- Asynchronous processing
- Speed of execution
- Modular program structure

- Thread synchronization
 - It is necessary to synchronize the activities of the various threads
 - All threads of a process share the same address space and other resources
 - Any alteration of a resource by one thread affects the other threads in the same process

- In an OS that supports threads, scheduling and dispatching is done on a thread basis
- Most of the state information dealing with execution is maintained in thread-level data structures
- Suspending a process involves suspending all threads of the process
- Termination of a process terminates all threads within the process

Thread Execution States

Thread operations associated with a change in thread state are:

- Spawn
- Block
- Unblock
- Finish

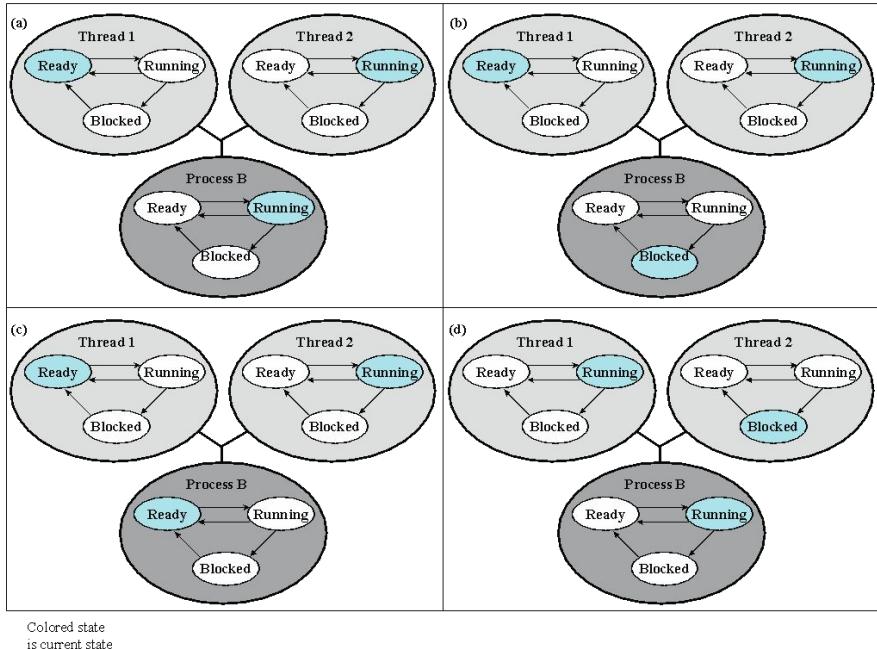
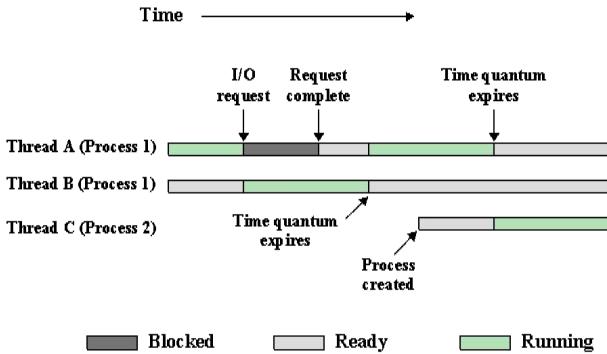
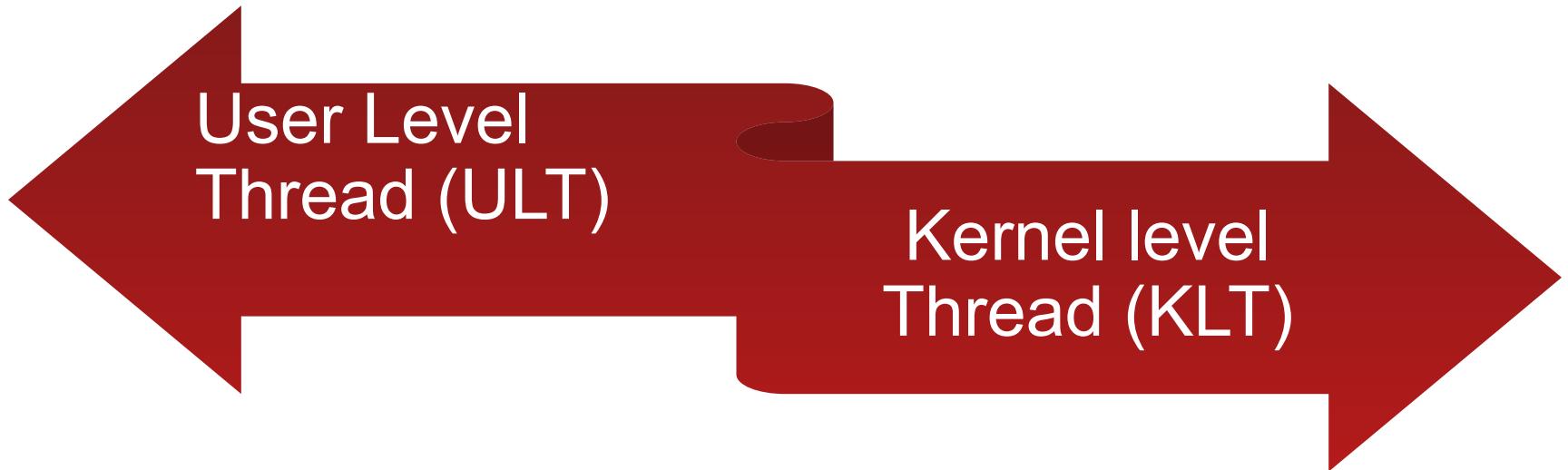


Figure 4.6 Examples of the Relationships Between User-Level Thread States and Process States

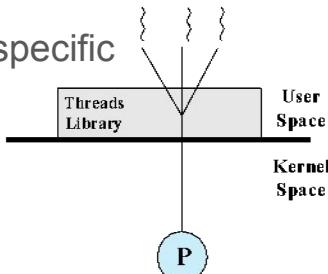
Types of Threads



ULT: Advantages and Disadvantages

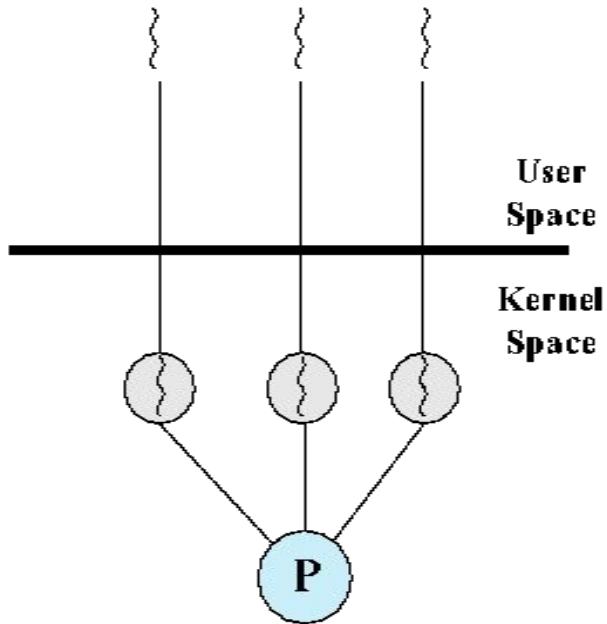
- All thread management is done by the application
- The kernel is not aware of the existence of threads
- Thread switching does not require kernel mode privileges
- Scheduling can be application specific
- ULTs can run on any OS

- In a typical OS many system calls are blocking
 - when a ULT executes a system call, not only is that thread blocked, but all of the threads within the process are blocked as well
 - Jacketing: convert a blocking system call into a non-blocking system call
- In a pure ULT strategy, a multithreaded application cannot take advantage of multiprocessing
 - A kernel assigns one process to only one processor at a time, therefore, only a single thread within a process can execute at a time



(a) Pure user-level

Kernel-Level Threads (KLTs)



(b) Pure kernel-level

- Thread management is done by the kernel
- There is no thread management code in the application level, simply an application programming interface (API) to the kernel thread facility
- **Windows is an example of this approach**

Advantages and Disadvantages of KLTs

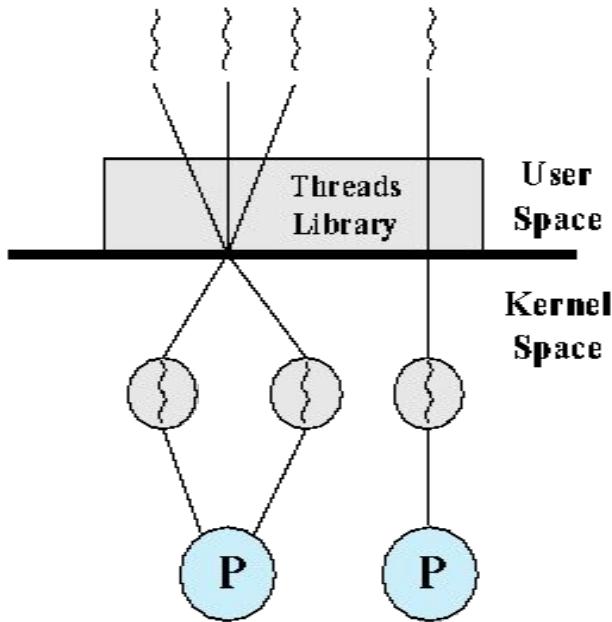
- The kernel can simultaneously schedule multiple threads from the same process on multiple processors
- If one thread in a process is blocked, the kernel can schedule another thread of the same process
- Kernel routines themselves can be multithreaded
- Combined Approach: Thread creation is done completely in the user space, as is the bulk of the scheduling and synchronization of threads within an application
- The transfer of control from one thread to another within the same process requires a mode switch to the kernel

Operation	User-Level Threads	Kernel-Level Threads	Processes
Null Fork	34	948	11,300
Signal Wait	37	441	1,840

Example: Thread and Process Operation Latencies (μs). Switching to Kernel-Level is more costly.

Combined Approaches

- Thread creation is done completely in the user space, as is the bulk of the scheduling and synchronization of threads within an application



(c) Combined

Linux: Processes and Threads

Linux Namespaces

- A namespace enables a process to have a different view of the system than other processes that have other associated namespaces
 - Widely by Linux Containers (LXC) projects
 - Different from Docker containers
- Namespaces
 - created by the **clone()** system call
 - gets as a parameter one of the six namespaces clone flags (CLONE_NEWNS, CLONE_NEWPID, CLONE_NEWNET, CLONE_NEWIPC, CLONE_NEWUTS, and CLONE_NEWUSER).
- A process can also create a namespace with the **unshare()** system call with the same flags;
 - as opposed to clone(), a new process is not created in such a case
 - a new namespace is created, and is attached to the calling process

Six Namespaces

- UTS Namespace
 - information about the current kernel, including nodename
- Mount Namespace
 - a specific view of the filesystem hierarchy
 - two processes with different mount namespaces see different filesystem hierarchies
- IPC Namespace
 - isolates certain interprocess communication (IPC) resources
 - semaphores, POSIX message queues etc
 - enable IPC among processes that share the same IPC namespace
- PID Namespace
 - processes can have the same PID
- Network Namespace
 - isolation of the system resources associated with networking
 - own network devices, IP addresses, IP routing tables etc.
 - allows processes that belong to the namespace to have the needed network access
- The Linux cgroup Subsystem
 - the basis of lightweight process virtualization
 - forms the basis of Linux containers

Linux Tasks

- Older versions of Linux offered no support for multithreaded applications.
- Execution flows of a multithreaded application
 - created, handled and scheduled in user mode
 - POSIX pthread library
- Linux uses lightweight processes (see later)
 - better support for multithreaded applications
 - two LWP may share resources
- **Thread group** is a set of LWPs that implement an application and act as a whole

A process, or task, in Linux is represented by a *task_struct* data structure

This structure contains information in a number of categories

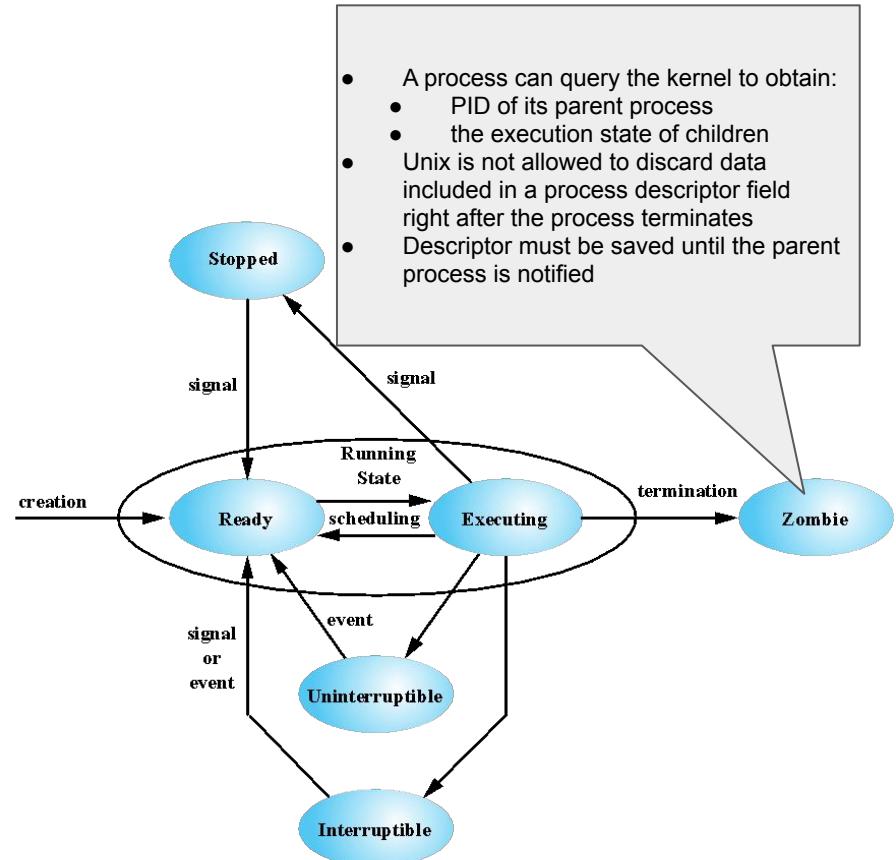


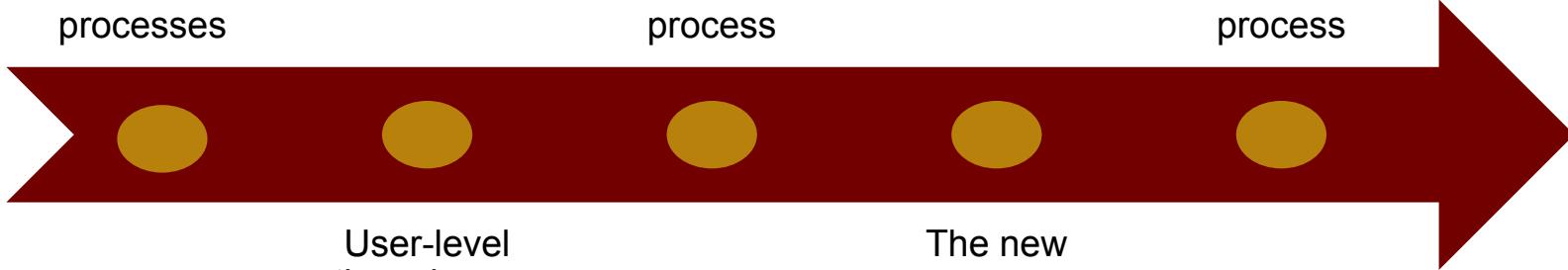
Figure 4.15 Linux Process/Thread Model

Linux Threads

Linux does not recognize a distinction between threads and processes

A new process is created by copying the attributes of the current process

The `clone()` call creates separate stack spaces for each process



User-level threads are mapped into kernel-level processes (see later)

The new process can be *cloned* so that it shares resources

Lightweight Process (LWP) Data Structure

- An LWP identifier
- The priority of this LWP and hence the kernel thread that supports it
- A signal mask that tells the kernel which signals will be accepted
- Saved values of user-level registers
- The kernel stack for this LWP
 - includes system call arguments, results,
 - error codes for each call level
- Resource usage and profiling data
- Pointer to the corresponding kernel thread
- Pointer to the process structure
- https://yajin.org/os2018fall/04_thread_b.pdf

Identifying a process

- Each execution context that can be independently scheduled must have its own process descriptor
 - 32-bit address of the **task_struct** structure
 - useful for the kernel to identify processes
- PIDs are numbered sequentially
 - getpid() and /proc/sys/kernel/pid_max

```
(base) alexandru@DESKTOP-THPML3T:~/os$ cat /proc/sys/kernel/pid_max
32768
```
 - recycling: pidmap_array bitmap denotes free and used PIDs
 - all threads of a multithreaded app have the same PID (POSIX 1003.1c standard)
- Process 1 init is the ancestor of all other processes

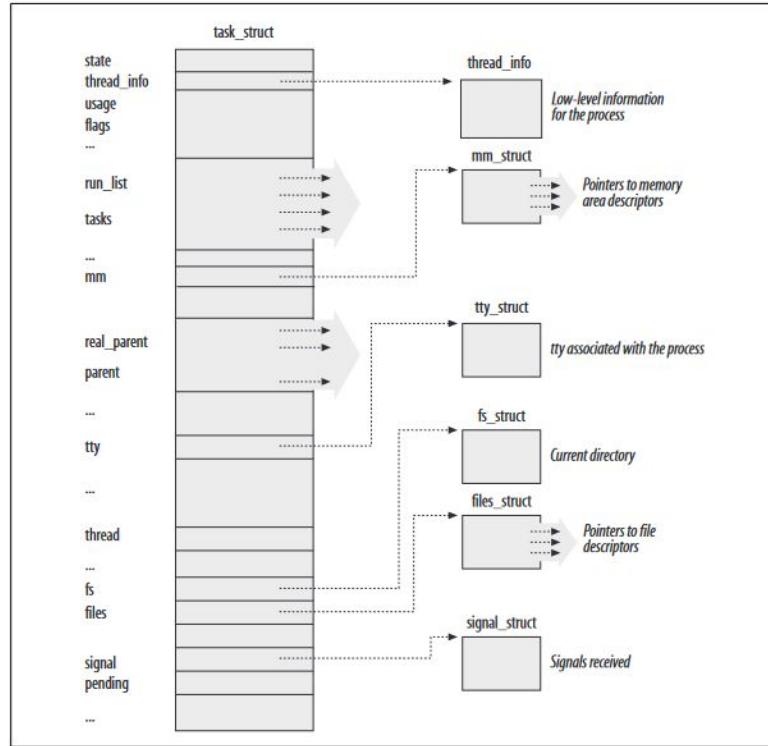


Figure 3-1. The Linux process descriptor

Process organization - Wait queues

Wait queues:

- have several uses in the kernel, particularly for interrupt handling, process synchronization, and timing.
- implement conditional waits on events: a process wishing to wait for a specific event places itself in the proper wait queue and relinquishes control
- implemented as doubly linked lists
 - elements include pointers to process descriptors
 - identified by a wait queue head, a data structure of type `wait_queue_head_t`

Sleeping processes:

- waiting for some event to occur
- **exclusive processes** (denoted by the value 1 in the flags field of the corresponding wait queue element) are selectively woken up by the kernel
- **nonexclusive processes** (denoted by the value 0 in the flags field) are always woken up by the kernel when the event occurs

The kernel awakens processes in the wait queues, putting them in the `TASK_RUNNING` state, by means of one of kernel code macros.

Creating Processes

Traditional Unix systems:

- treat all processes in the same way
- resources owned by the parent process are duplicated in the child process
 - process creation very slow and inefficient,
 - requires copying the entire address space of the parent process
- child process rarely needs to read or modify inherited resources
 - it issues an immediate `execve()`
 - wipes out the address space that was so carefully copied

```
EXECVE(2)                               Linux Programmer's Manual

EXECVE(2)

NAME
    execve - execute program

SYNOPSIS
    #include <unistd.h>

    int execve(const char *pathname, char *const argv[],
               char *const envp[]);

DESCRIPTION
    execve() executes the program referred to by pathname.
    This causes the program that is currently being run by the calling
    process to be replaced with a new program, with newly initialized
    stack, heap, and (initialized and uninitialized) data segments.
```

Modern Unix kernels:

- The Copy On Write technique allows both the parent and the child to read the same physical pages
 - When one tries to write on a physical page, the kernel copies its contents into a new physical page that is assigned to the writing process
- LWP allow both the parent and the child to share many per process kernel data structures
- The `vfork()` system call creates a process that shares the memory address space of its parent
 - prevents parent from overwriting data of child
 - the parent's execution is blocked until the child exits or executes a new program.

```
VFORK(2)                                Linux Programmer's Manual          VFORK(2)

NAME
    vfork - create a child process and block parent

SYNOPSIS
    #include <sys/types.h>
    #include <unistd.h>

    pid_t vfork(void);

Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

vfork():
    Since glibc 2.12:
        (_XOPEN_SOURCE >= 500) || (_POSIX_C_SOURCE >= 200809L)
        || /* Since glibc 2.19: */ __DEFAULT_SOURCE
        || /* Glibc versions <= 2.19: */ __BSD_SOURCE
    Before glibc 2.12:
        __BSD_SOURCE || _XOPEN_SOURCE >= 500

DESCRIPTION
    Standard description
    (From POSIX.1) The vfork() function has the same effect as fork(2), except that the behavior is undefined if the process created by vfork() either modifies any data other than a variable of type pid_t used to store the return value from vfork(), or returns from the function in which vfork() was called, or calls any other function before successfully calling _exit(2) or one of the exec(3) family of functions.
```

Kernel Threads

- Delegate critical tasks to intermittently running processes
 - flushing disk caches, swapping out unused pages, servicing network connections, and so on
 - not efficient to perform these tasks in strict linear fashion
 - better response if scheduled in the background.
- In Linux, kernel threads differ from regular processes in the following ways:
 - Kernel threads run only in Kernel Mode, while regular processes run alternatively in Kernel Mode and in User Mode.
 - kernel threads use only linear addresses greater than PAGE_OFFSET (Virtual start address of the first bank of RAM)

```
ps --ppid 2 -p 2 -o uname,pid,ppid,cmd,cls
```

Examples of kernel threads (parent pid is 2) are:

- **keventd** (also called events) Executes the functions in the keventd_wq workqueue
- **kapmd** Handles the events related to the Advanced Power Management (APM).
- **kswapd** Reclaims memory
- **pdflush** Flushes “dirty” buffers to disk to reclaim memory
- **kblockd** periodically activates the block device drivers
- **ksoftirqd** Runs the tasklets; one for each CPU in the system

<https://unix.stackexchange.com/questions/411159/linux-is-it-possible-to-see-only-kernel-space-threads-process>

/proc - process information pseudo-file system

/proc is a virtual filesystem:

- a control and information centre for the kernel
- read/change kernel parameters (sysctl) while the system is running
- files in this directory is the fact that all of them have a file size of 0, with the exception of kcore, mtrr and self
- sometimes referred to as a process information pseudo-file system
- Doesn't contain 'real' files but runtime system information

A lot of system utilities are simply calls to files in /proc:

- 'lsmod' is the same as 'cat /proc/modules'
- 'lspci' is a synonym for 'cat /proc/pci'

1	6430	dma	kpageflags	stat
1968	6437	driver	loadavg	swaps
1969	6498	execdomains	locks	sys
1971	6533	filesystems	mdstat	sysvipc
6361	6877	fs	meminfo	thread-self
6362	acpi	interrupts	misc	timer_list
6363	buddyinfo	iomem	modules	tty
6364	bus	ioports	mounts	uptime
6369	cgroups	irq	mtrr	version
6373	cmdline	kallsyms	net	vmallocinfo
6384	config.gz	kcore	pagetypeinfo	vmstat
6385	consoles	key-users	partitions	zoneinfo
6386	cpuinfo	keys	sched_debug	
6393	crypto	kmsg	schedstat	
6428	devices	kpagegroup	self	
6429	diskstats	kpagecount	softirqs	

Information about processes

- /proc/PID/cmdline Command line arguments.
- /proc/PID cwd Link to the current working directory.
- /proc/PID/environ Values of environment variables.
- /proc/PID/exe Link to the executable of this process.
- /proc/PID/fd Directory, which contains all file descriptors.
- /proc/PID/maps Memory maps to executables and libraries.
- /proc/PID/mem Memory held by this process.
- /proc/PID/root Link to the root directory of this process.
- /proc/PID/stat Process status.
- /proc/PID/statm Process memory status information.
- /proc/PID/status Process status in human readable form.

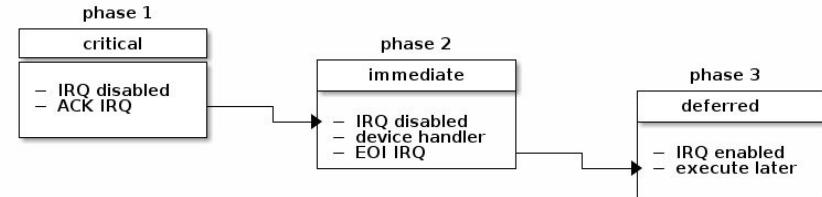
```
(base) alexandru@DESKTOP-TKPM3T:~/os$ sudo cat /proc/6373/cmdline
/home/alexandru/.vscode-server/bin/74b1f979648cc44d385a2286793c2
26e611f59e7/node/home/alexandru/.vscode-server/bin/74b1f979648cc
44d385a2286793c226e611f59e7/out/server-main.js--host=127.0.0.1--
port=80--connection-token=1891230501-3244511119-3775249901-366097
3238--use-host-proxy--without-browser-env-var--disable-websocket
--compression--accept-server-license-terms--telemetry-level=all
(base) alexandru@DESKTOP-TKPM3T:~/os$ |
```

```
Name: sh
Umask: 0022
State: S (sleeping)
Ppid: 6364
TracerPid: 0
Uid: 1000 1000 1000 1000
Gid: 1000 1000 1000 1000
FDSize: 64
Groups: 4 20 24 25 27 29 30 44 46 117 119 1000
NSgid: 6369
NSpid: 6369
NSpgid: 6363
NSsid: 6363
VmPeak: 2612 kB
VmSize: 2612 kB
VmLck: 0 kB
VmPin: 0 kB
VmHWM: 592 kB
VmRSS: 592 kB
RssAnon: 68 kB
RssFile: 524 kB
RssShmem: 0 kB
VmData: 184 kB
VmStk: 132 kB
VmExe: 76 kB
VmLib: 1648 kB
VmPTE: 44 kB
VmSwap: 0 kB
HugeTLBPages: 0 kB
CoreDumping: 0
TSD enabled: 1
Threads: 1
SigQ: 0/24/69
SigPnd: 0000000000000000
ShdPnd: 0000000000000000
SigBlk: 0000000000000000
SigIgn: 0000002000000000
SigCgt: 000000000010002
CapInh: 0000000000000000
CapPrm: 0000000000000000
CapEff: 0000000000000000
CapBnd: 000001ffffffffff
CapAmb: 0000000000000000
NoNewPrivs: 0
Seccomp: 0
Seccomp_filters: 0
Speculation_Store_Bypass: thread vulnerable
Cpus_allowed: ff
```

Interrupt handling in Linux

Three phases: critical, immediate and deferred.

- **critical**
 - the kernel will run the generic interrupt handler that determines the interrupt number, the interrupt handler for this particular interrupt and the interrupt controller
 - critical actions will also be performed (e.g. acknowledge the interrupt at the interrupt controller level).
 - Local processor interrupts are disabled
- **immediate**
 - device driver's handlers will be executed
 - the interrupt controller's "end of interrupt" method is called to allow the interrupt controller to reassert this interrupt
 - local processor interrupts are enabled at this point
- **deferred**
 - avoid doing too much work in the interrupt handler function
 - "bottom half" of the interrupt (the upper half being the part of the interrupt handling that runs with interrupts disabled).
 - interrupts are enabled on the local processor



Information about interrupts

/proc/interrupts

- Shows which interrupts are in use, and how many of each there have been
- Check which interrupts are currently in use and what they are used for
- smp_affinity
 - used to set IRQ to CPU affinity
 - "hook" an IRQ to only one CPU, or to exclude a CPU from handling IRQs.

	CPU0	CPU1	CPU2	CPU3	CPU4	CPU5	CPU6	CPU7	IO-APIC	8-edge	rtc0
8:	0	0	0	0	0	0	0	0	0	0	0
9:	76	0	0	0	0	0	0	0	0	0	0
NMI:	0	0	0	0	0	0	0	0	0	0	0
LOC:	0	0	0	0	0	0	0	0	0	0	0
SPU:	0	0	0	0	0	0	0	0	0	0	0
PMI:	0	0	0	0	0	0	0	0	0	0	0
0 IRQ work interrupts											
RTR:	0	0	0	0	0	0	0	0	0	0	0
RES:	18135156	5044246	10717458	4676029	10588434	4717322	10551013	4334762	APIC ICR read retries		
CAL:	496155	334095	159586	96848	135981	49658	127471	37324	Rescheduling interrupts		
TLB:	0	0	0	0	0	0	0	0	0	TLB shootdowns	
HYP:	547413	109800	2913	2286	1770	9523	11943	1699	Hypervisor callback interrupts		
HRE:	0	0	0	0	0	0	0	0	0	Hyper-V reenlightenment interrupts	
HVS:	5791138	2552633	5492557	2882265	5704215	2736784	5488315	2680139	Hyper-V timer0 interrupts		
ERR:	0										
MIS:	0										
PIN:	0	0	0	0	0	0	0	0	0	Posted-interrupt notification event	
NPI:	0	0	0	0	0	0	0	0	0	Nested posted-interrupt event	
PIW:	0	0	0	0	0	0	0	0	0	Posted-interrupt wakeup event	

```
(base) alexandru@DESKTOP-TKPML3T:~/os$ ls /proc/irq/  
0 1 10 11 12 13 14 15 2 3 4 5 6 7 8 9 default_smp_affinity  
(base) alexandru@DESKTOP-TKPML3T:~/os$ cat /proc/irq/10/smp_affinity  
ff
```

Operating Systems

CS-C3140, Lecture 5

Alexandru Paler

Announcements

- Exam: 7 December 2022, 9:00
 - was 7 December 2022, then became 13 December
 - details about exact form will be announced asap
- Discord is dead, long live Zulip

Operating System Software

The design of the memory management portion of an operating system depends on three fundamental areas of choice:

- 1) Whether or not to use virtual memory techniques
- 2) The use of paging or segmentation or both
- 3) Memory Management: the algorithms employed for various aspects

Frame	A fixed-length block of main memory.
Page	A fixed-length block of data that resides in secondary memory (such as disk). A page of data may temporarily be copied into a frame of main memory.
Segment	A variable-length block of data that resides in secondary memory. An entire segment may temporarily be copied into an available region of main memory (segmentation) or the segment may be divided into pages which can be individually copied into main memory (combined segmentation and paging).

Real and Virtual Memory

Real
memory

Virtual
memory

Main memory, the
actual RAM

Memory on disk

Allows for effective
multiprogramming and
relieves the user of tight
constraints of main memory

Memory Management Requirements

- Memory management is intended to satisfy the following requirements:
 - Relocation
 - Protection
 - Sharing
 - Logical organization
 - Physical organization
- Two characteristics fundamental to memory management:
 - All memory references are logical addresses that are dynamically translated into physical addresses at run time
 - A process may be broken up into a number of pieces that don't need to be contiguously located in main memory during execution
 - If these two characteristics are present, it is not necessary that all of the pages or segments of a process be in main memory during execution

Implications

- More processes may be maintained in main memory
 - Because only some of the pieces of any particular process are loaded, there is room for more processes
 - This leads to more efficient utilization of the processor because it is more likely that at least one of the more numerous processes will be in a Ready state at any particular time
- A process may be larger than all of main memory
 - If the program being written is too large, the programmer must devise ways to structure the program into pieces that can be loaded separately in some sort of overlay strategy
 - With virtual memory based on paging or segmentation, that job is left to the OS and the hardware
 - The OS automatically loads pieces of a process into main memory as required

Relocation

- Programmers typically do not know in advance which other programs will be resident in main memory at the time of execution of their program
- Active processes need to be able to be swapped in and out of main memory in order to maximize processor utilization
- Specifying that a process must be placed in the same memory region when it is swapped back in would be limiting
- May need to relocate the process to a different area of memory

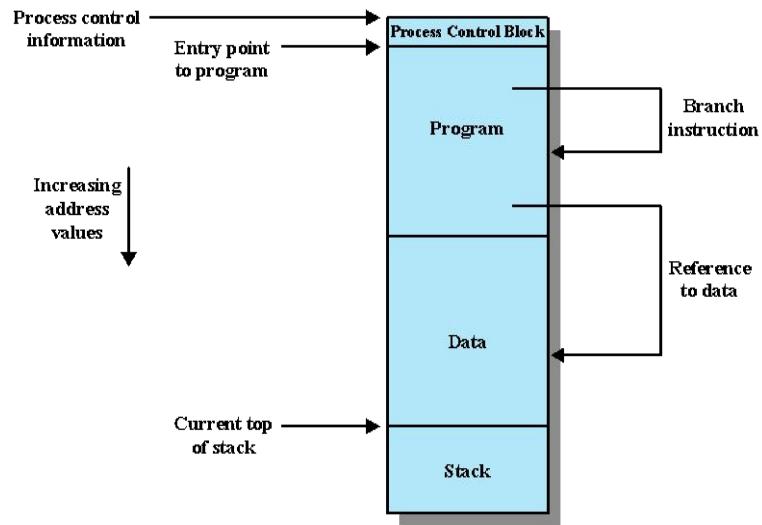


Figure 7.1 Addressing Requirements for a Process

Execution of a Process

- Resident set
 - Operating system brings into main memory a few pieces of the program
 - Portion of process that is in main memory
 - An interrupt is generated when an address is needed that is not in main memory
 - Operating system places the process in a blocking state
- Piece that contains the logical address is brought into main memory
 - operating system issues a disk I/O Read request
 - another process is dispatched to run while the disk I/O takes place
 - an interrupt is issued when disk I/O is complete,
 - place the affected process in the Ready state

Hardware Support for Relocation

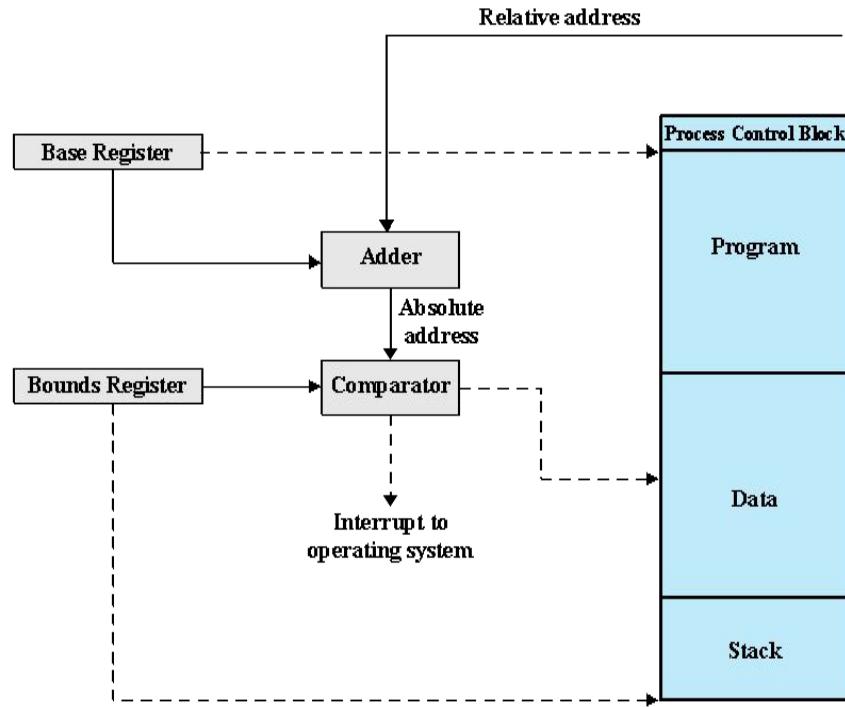
Process in Running state (loaded or swapped into memory)

- Base register indicates the starting address in main memory
- Bounds register indicates the ending location of the program

Memory references in the loaded process:

- are relative to the origin of the program
- hardware mechanism for translating relative addresses to physical main memory addresses at the time of execution of the instruction that contains the reference

Process is isolated by the contents of the base and bounds registers and safe from unwanted accesses by other processes.



Protection and Sharing

- Processes need to acquire permission to reference memory locations for reading or writing purposes
- Location of a program in main memory is unpredictable
- Memory references generated by a process must be checked at run time
- Mechanisms that support relocation also support protection
- Advantageous to allow each process access to the same copy of the program rather than have their own separate copy
- Memory management must allow controlled access to shared areas of memory without compromising protection
- Mechanisms used to support relocation support sharing capabilities

Logical and Physical Organization

- Memory is organized as linear
 - sequence of bytes or words
 - mirrors the actual machine hardware
 - does not correspond to the way in which programs are typically constructed
- Programs are written in modules
 - written and compiled independently
 - Different degrees of protection given to modules (read-only, execute-only)
- Sharing on a module level corresponds to the user's way of viewing the problem

Cannot leave the programmer with the responsibility to manage memory

Memory available for a program plus its data may be insufficient

Programmer does not know how much space will be available

Overlaying allows various modules to be assigned the same region of memory but is time consuming to program

Memory Management Techniques

Technique	Description	Strengths	Weaknesses
Fixed Partitioning	Main memory is divided into a number of static partitions at system generation time. A process may be loaded into a partition of equal or greater size.	Simple to implement; little operating system overhead.	Inefficient use of memory due to internal fragmentation; maximum number of active processes is fixed.
	Partitions are created dynamically, so that each process is loaded into a partition of exactly the same size as that process.	No internal fragmentation; more efficient use of main memory.	Inefficient use of processor due to the need for compaction to counter external fragmentation.
Simple Paging	Main memory is divided into a number of equal-size frames. Each process is divided into a number of equal-size pages of the same length as frames. A process is loaded by loading all of its pages into available, not necessarily contiguous, frames.	No external fragmentation.	A small amount of internal fragmentation.
	Each process is divided into a number of segments. A process is loaded by loading all of its segments into dynamic partitions that need not be contiguous.	No internal fragmentation; improved memory utilization and reduced overhead compared to dynamic partitioning.	External fragmentation
Virtual Memory Paging	As with simple paging, except that it is not necessary to load all of the pages of a process. Nonresident pages that are needed are brought in later automatically.	No external fragmentation; higher degree of multiprogramming; large virtual address space.	Overhead of complex memory management.
	As with simple segmentation, except that it is not necessary to load all of the segments of a process. Nonresident segments that are needed are brought in later automatically.	No internal fragmentation, higher degree of multiprogramming; large virtual address space; protection and sharing support.	Overhead of complex memory management.
Virtual Memory Segmentation			

Addresses

Logical

- Reference to a memory location independent of the current assignment of data to memory

Relative

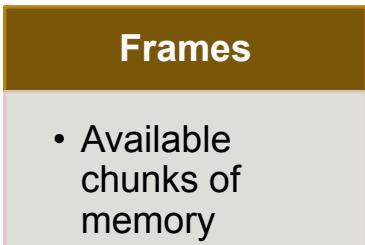
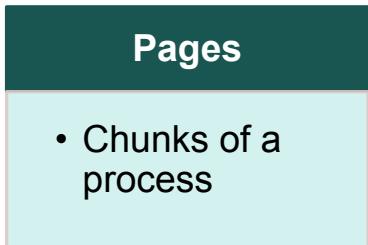
- A particular example of logical address, in which the address is expressed as a location relative to some known point

Physical or Absolute

- Actual location in main memory

Paging

- Partition memory into equal fixed-size chunks that are relatively small
- Process is also divided into small fixed-size chunks of the same size
- Each process has its own page table
 - Each page table entry (PTE) contains the frame number of the corresponding page in main memory
 - A **page table** is also needed for a virtual memory scheme based on paging



Frame number	Main memory
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	

(a) Fifteen Available Frames

Main memory
A.0
A.1
A.2
A.3

(b) Load Process A

Main memory
A.0
A.1
A.2
A.3
B.0
B.1
B.2

(c) Load Process B

Main memory
A.0
A.1
A.2
A.3
B.0
B.1
B.2
C.0
C.1
C.2
C.3

(d) Load Process C

Main memory
A.0
A.1
A.2
A.3
D.0
D.1
D.2
C.0
C.1
C.2
C.3

(e) Swap out B

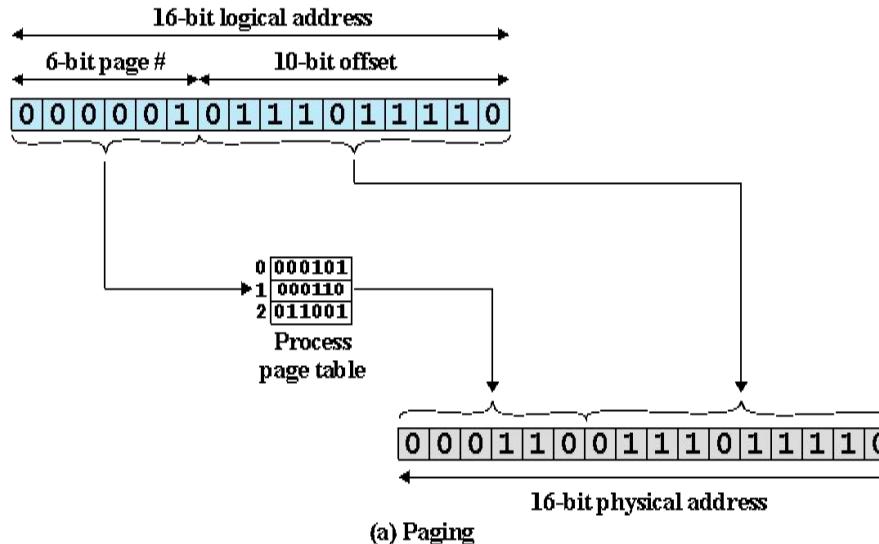
Main memory
A.0
A.1
A.2
A.3
D.3
D.4

(f) Load Process D

Figure 7.9 Assignment of Process Pages to Free Frames

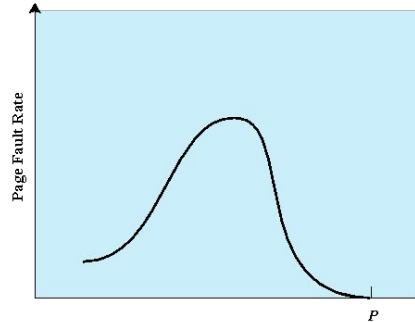
Page Table

- Maintained by operating system for each process
- Contains the frame location for each page in the process
- Processor must know how to access for the current process
- Used by processor to produce a physical address

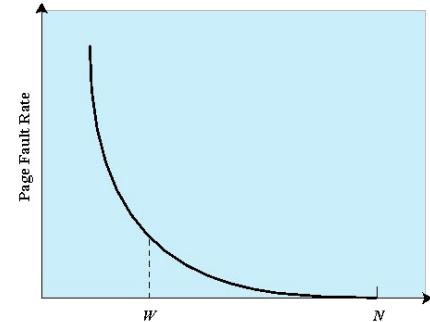


Page Size

- The smaller the page size, the lesser the amount of internal fragmentation
 - However, more pages are required per process
 - More pages per process means larger page tables
 - For large programs in a heavily multiprogrammed environment some portion of the page tables of active processes must be in virtual memory instead of main memory
 - The physical characteristics of most secondary-memory devices favor a larger page size for more efficient block transfer of data



(a) Page Size



(b) Number of Page Frames Allocated

P = size of entire process
 W = working set size
 N = total number of pages in process

The design issue of page size is related to the size of physical main memory and program size



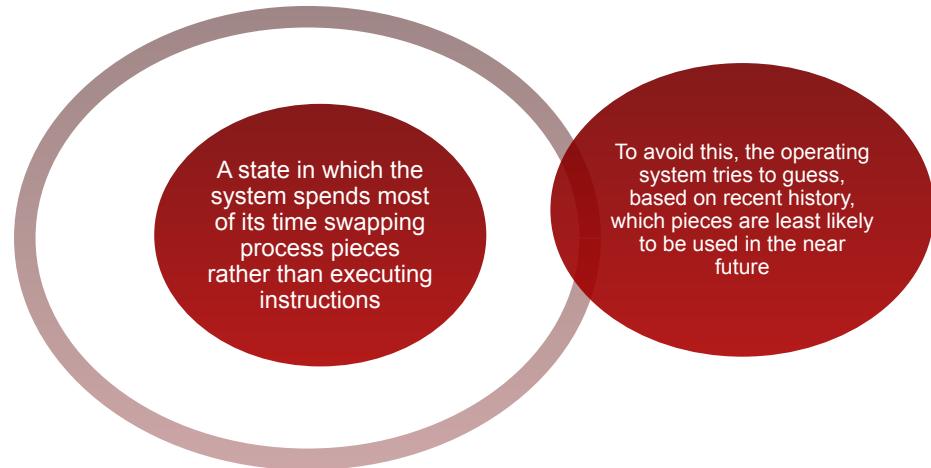
Main memory is getting larger and address space used by applications is also growing



Most obvious on personal computers where applications are becoming increasingly complex

Principle of Locality and Thrashing

- Program and data references within a process **tend to cluster**
- Only a few pieces of a process will be needed over a short period of time
- Therefore it is possible to make intelligent guesses about which pieces will be needed in the future
- Avoids thrashing



Segmentation: Protection and Sharing

- Segmentation lends itself to the implementation of protection and sharing policies
- Each entry has a base address and length so inadvertent memory access can be controlled
- Sharing can be achieved by segments referencing multiple processes

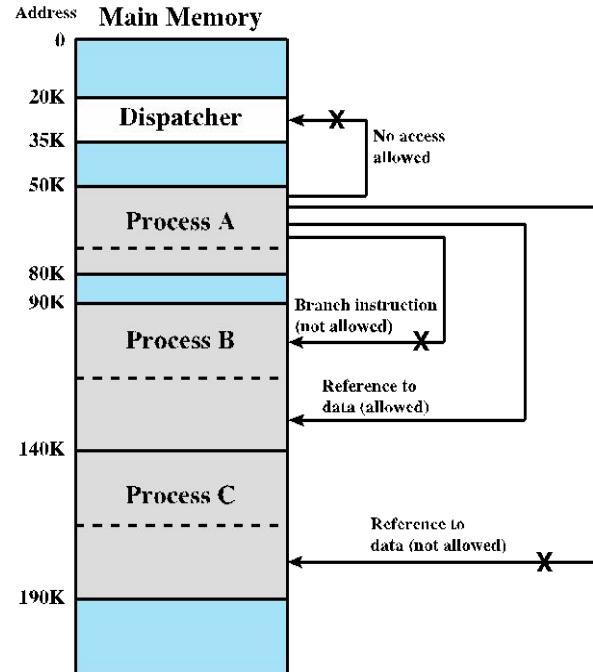


Figure 8.13 Protection Relationships Between Segments

Segment Organization

- Each segment table entry contains the starting address of the corresponding segment in main memory and the length of the segment
- A bit is needed to determine if the segment is already in main memory
- Another bit is needed to determine if the segment has been modified since it was loaded in main memory

Logical Addresses

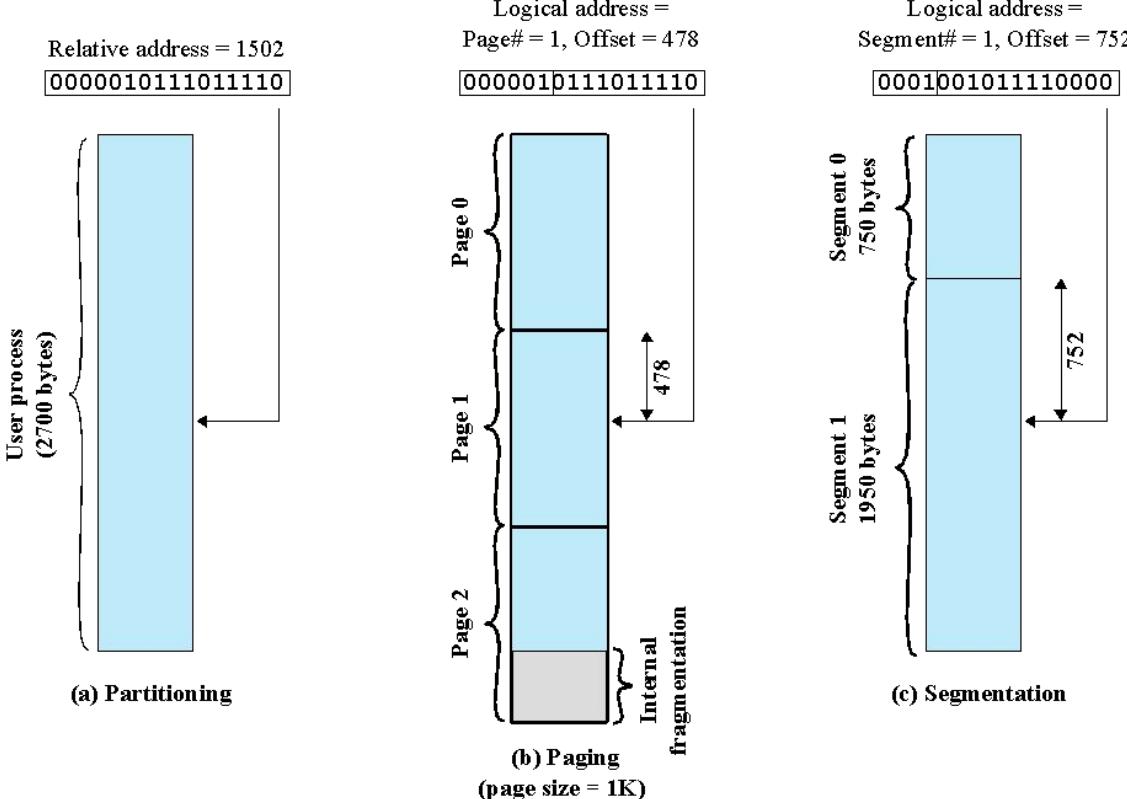


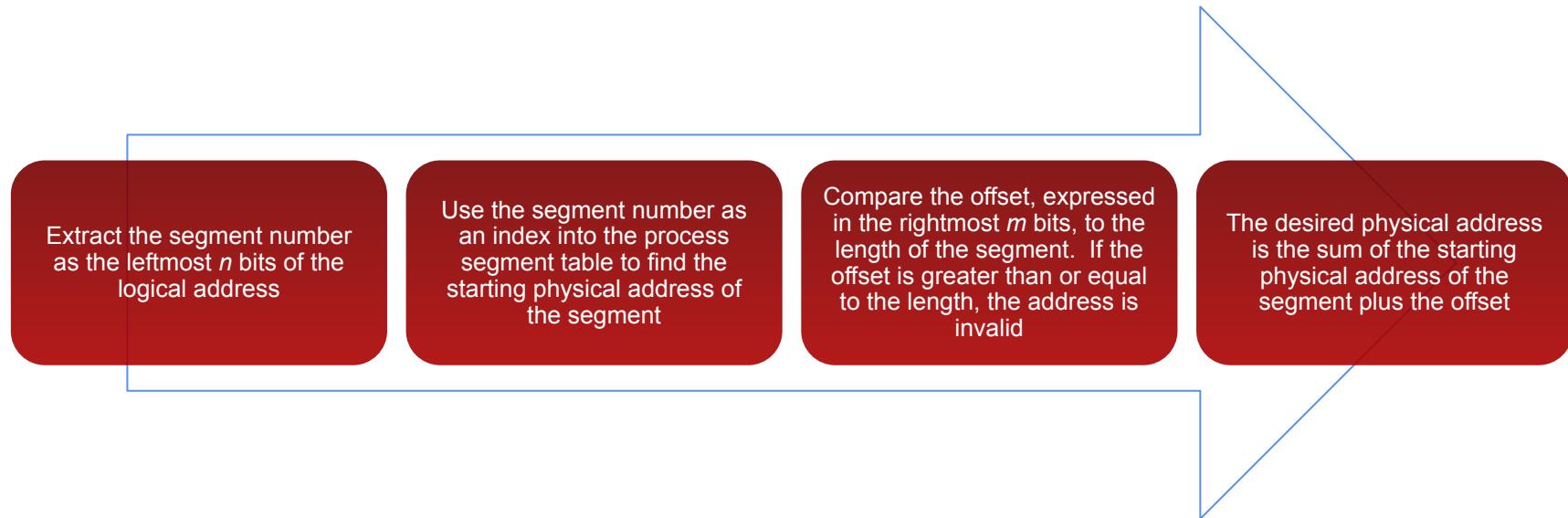
Figure 7.11 Logical Addresses

Segmentation

- A program can be subdivided into segments
 - May vary in length
 - There is a maximum length
- Addressing consists of two parts (similar to paging):
 - Segment number
 - An offset
- Eliminates internal fragmentation
- Typically the programmer will assign programs and data to different segments
 - modularity: the program or data may be further broken down into multiple segments
 - inconvenience: the programmer must be aware of the maximum segment size limitation
- A segment table entry contains
 - the starting address of the corresponding segment in main memory
 - the length of the segment
 - bit to determine if the segment is already in main memory
 - Another bit to determine if the segment has been modified

Address Translation - Steps

- Another consequence of unequal size segments is that there is no simple relationship between logical addresses and physical addresses



Combined Paging and Segmentation

In a combined paging/segmentation system a user's address space is broken up into a number of segments. Each segment is broken up into a number of fixed-sized pages which are equal in length to a main memory frame

Segmentation is visible to the programmer

Paging is transparent to the programmer

Virtual Memory

Virtual Memory Terminology

Virtual memory	A storage allocation scheme in which secondary memory can be addressed as though it were part of main memory. The addresses a program may use to reference memory are distinguished from the addresses the memory system uses to identify physical storage sites, and program-generated addresses are translated automatically to the corresponding machine addresses. The size of virtual storage is limited by the addressing scheme of the computer system and by the amount of secondary memory available and not by the actual number of main storage locations.
Virtual address	The address assigned to a location in virtual memory to allow that location to be accessed as though it were part of main memory.
Virtual address space	The virtual storage assigned to a process.
Address space	The range of memory addresses available to a process.
Real address	The address of a storage location in main memory.

Characteristics of Paging and Segmentation

Simple Paging	Virtual Memory Paging	Simple Segmentation	Virtual Memory Segmentation	
Main memory partitioned into small fixed-size chunks called frames		Main memory not partitioned		
Program broken into pages by the compiler or memory management system		Program segments specified by the programmer to the compiler (i.e., the decision is made by the programmer)		
Internal fragmentation within frames		No internal fragmentation		
No external fragmentation		External fragmentation		
Operating system must maintain a page table for each process showing which frame each page occupies		Operating system must maintain a segment table for each process showing the load address and length of each segment		
Operating system must maintain a free frame list		Operating system must maintain a list of free holes in main memory		
Processor uses page number, offset to calculate absolute address		Processor uses segment number, offset to calculate absolute address		
All the pages of a process must be in main memory for process to run, unless overlays are used	Not all pages of a process need be in main memory frames for the process to run. Pages may be read in as needed	All the segments of a process must be in main memory for process to run, unless overlays are used	Not all segments of a process need be in main memory for the process to run. Segments may be read in as needed	
	Reading a page into main memory may require writing a page out to disk		Reading a segment into main memory may require writing one or more segments out to disk	

Hardware Support and Issues

- Hardware must support paging and segmentation
- Operating system must include software for managing the movement of pages and/or segments between secondary memory and main memory

Issues to Address

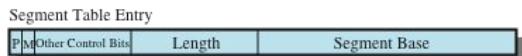
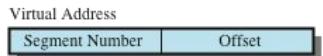
- Size of page-tables would be very large!
- For example, 32-bit virtual address spaces (4 GB) and a 4 KB page size would have ~1 M pages/entries in page-tables.
- A process does not access all of its address space at once!

Exploit this locality factor.

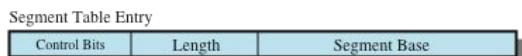
Memory Management Formats



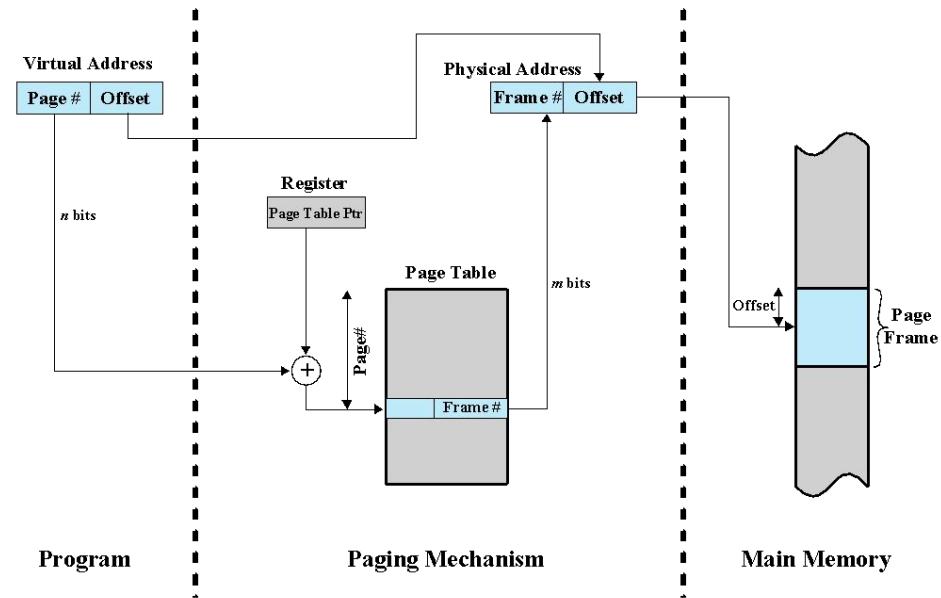
(a) Paging only



(b) Segmentation only

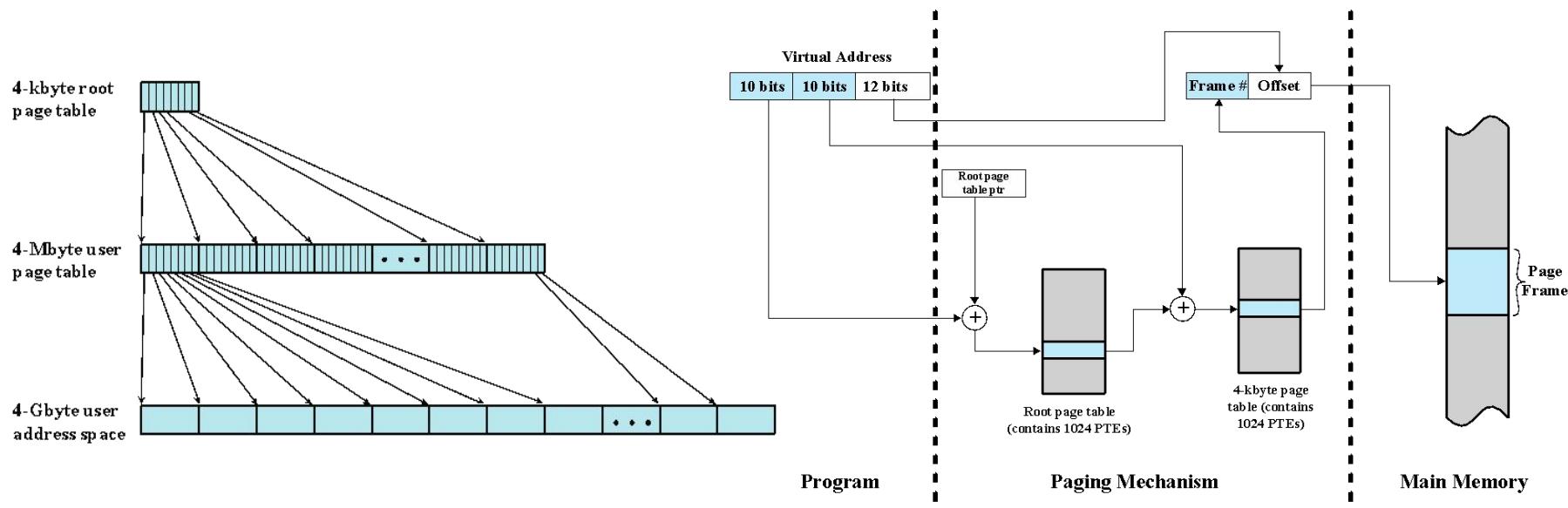


(c) Combined segmentation and paging



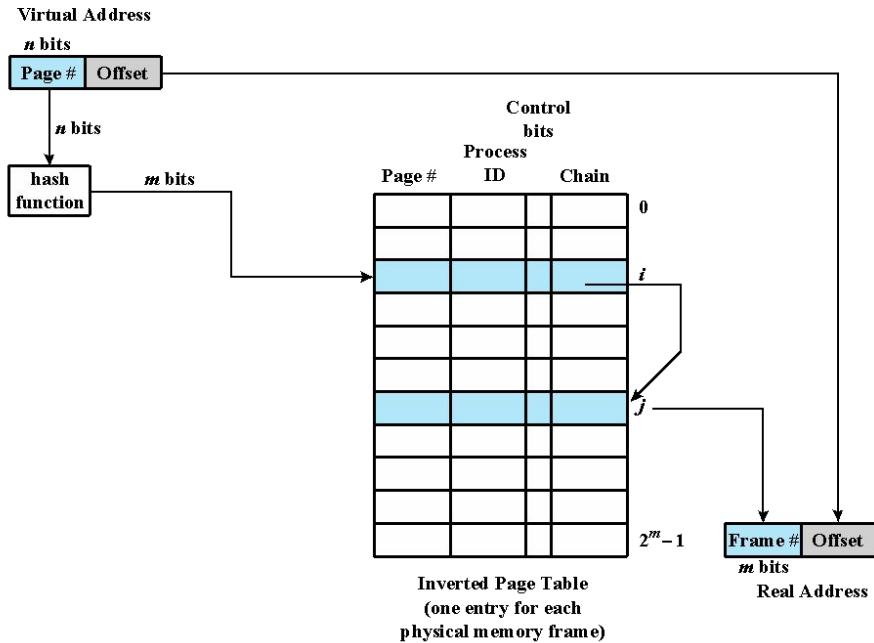
P= present bit
M = Modified bit

Two-Level Hierarchical Page Table



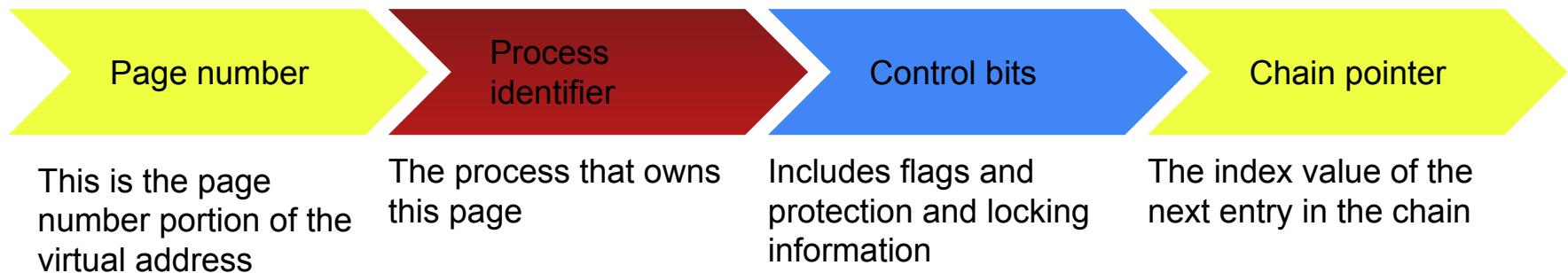
Inverted Page Table

- Used on the PowerPC, UltraSPARC, and the IA-64 architecture
- Page number portion of a virtual address is mapped into a hash value
- Hash value points to inverted page table
- Fixed proportion of real memory is required for the tables regardless of the number of processes or virtual pages supported
- Structure is called inverted because it indexes page table entries by frame number rather than by virtual page number



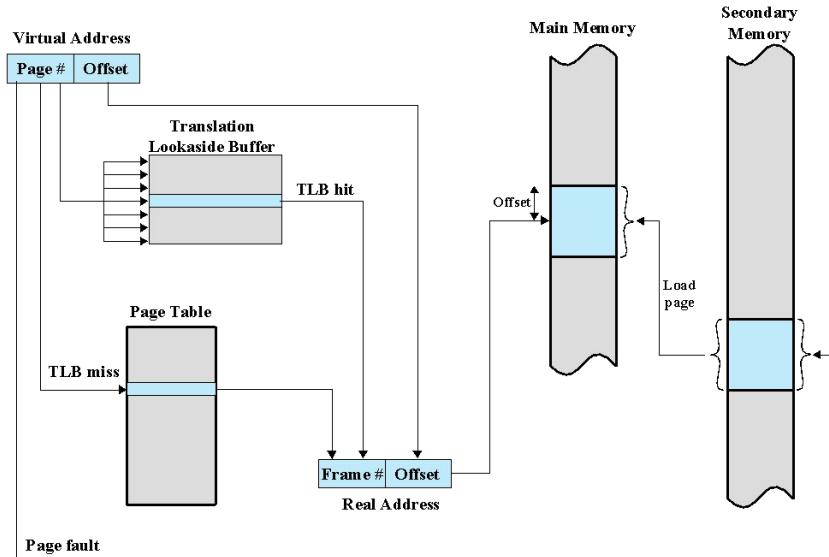
Inverted Page Table

Each entry in the page table includes:

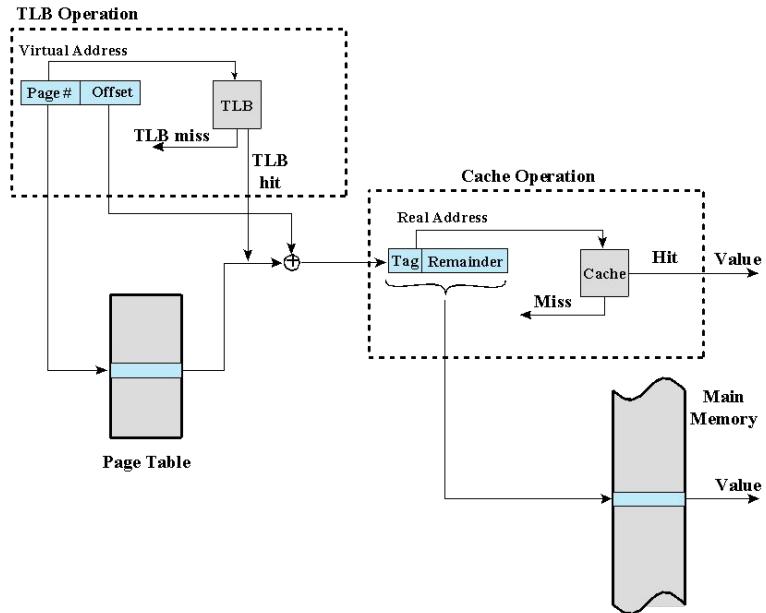
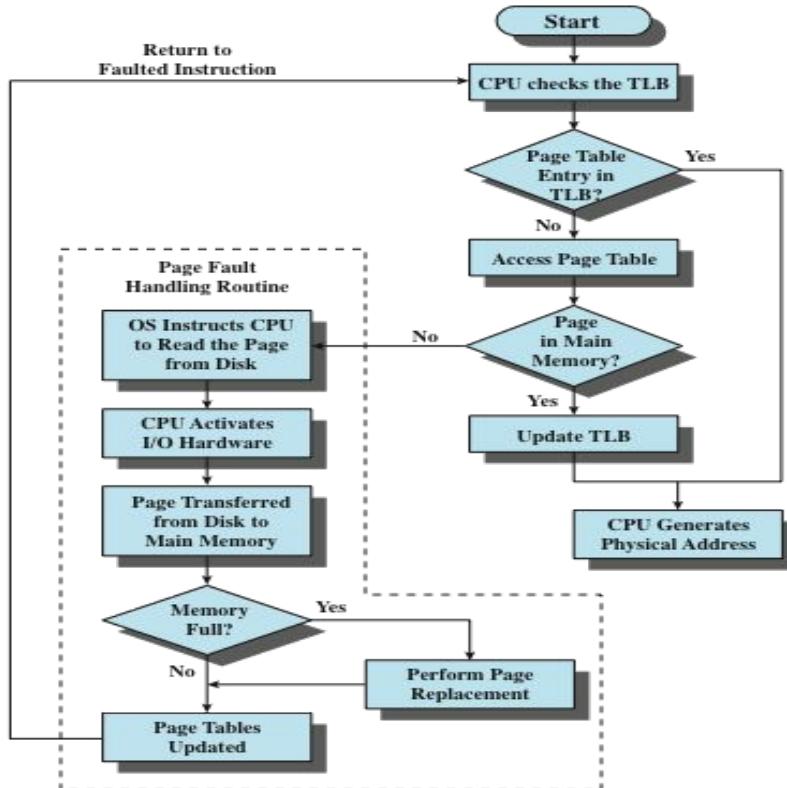


Translation Lookaside Buffer (TLB)

- To overcome the effect of doubling the memory access time, most virtual memory schemes make use of a special high-speed cache called a translation lookaside buffer (TLB)
- Each virtual memory reference can cause two physical memory accesses:
 - One to fetch the page table entry
 - One to fetch the data
- This cache functions in the same way as a memory cache and contains those page table entities that have been most recently used



Operation of Paging, TLB and Cache



Associative Mapping

- The TLB only contains some of the page table entries so we cannot simply index into the TLB based on page number
 - Each TLB entry must include the page number as well as the complete page table entry
- The processor is equipped with hardware that allows it to interrogate simultaneously a number of TLB entries to determine if there is a match on page number

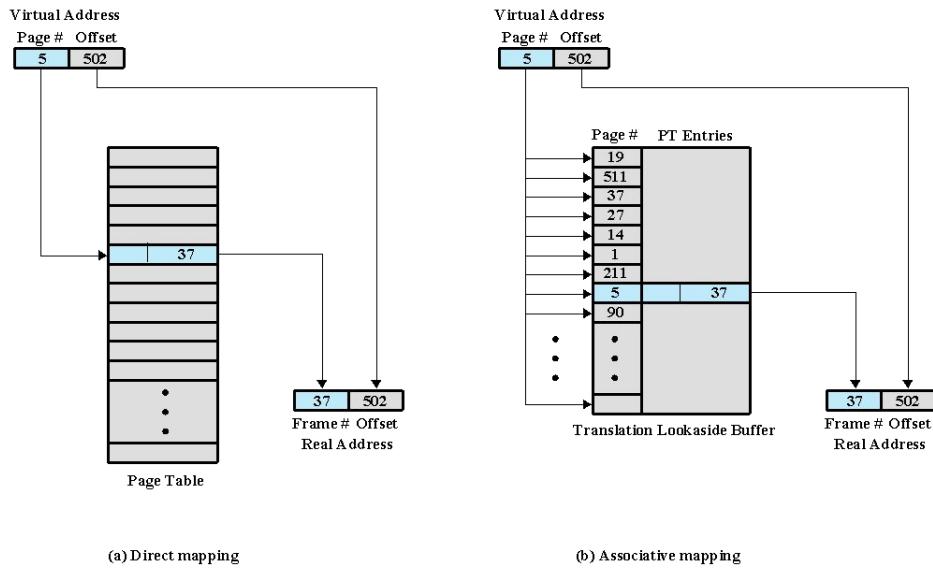


Figure 8.8 Direct Versus Associative Lookup for Page Table Entries

Warm-ups and cold starts

- A CPU runs fast when its caches and TLB are “warm”
 - Being “warm” means that the CPU has the cache and TLB contents that fit the memory reference patterns of its program
 - Warming up can take several milliseconds even for fast systems
- “Cold” when cache and TLB contents are unsuitable
 - Typically after a context switch (e.g., to a different thread)
- Note that there are several levels of coldness
 - A cold cache, a cold TLB, a cold memory
 - A cold memory is really costly, it is typical for process starts or after a process swap (everything must be fetched from secondary mem)
- The misses and faults are named accordingly
 - A cold cache miss, a cold TLB miss, a cold page fault
 - Note that the choice of replacement algorithms has no effect

Management of Pages

Process working sets and swapping

- Maintaining working sets in main memory
 - A program usually access data in the same area
 - Loops (etc.) in code iterating through data structures
 - Only currently used parts of a process need to be in memory
 - The OS tries to figure out the size of the working sets of processes and balance between them
- Processes can be swapped out
 - All the pages of a process can be in the secondary memory
 - Without swapping processes out, there can be too much competition over memory => performance degradation
 - Note: there are different meaning for the word “swapping”, e.g., process swapping and page swapping
- Resident set management
 - How many pages of a process to keep in memory?
 - Small resident set size
 - Larger number of processes in memory
- Scheduler can find ready processes to run
 - There is not enough of the process in the memory
 - Lots of page faults
- Large resident set size
 - Due to the **principle of locality**, there is a limit after which increasing resident set size does not help
- There are several policies for this also
 - Fixed allocation policy, variable allocation policy, etc.

Fetch policy

- When page should be brought to memory
 - Note that accessing main memory is typically much faster than accessing secondary memory (e.g. HDD/SDD)
- Demand paging
 - Bring the page once it is referenced
 - When a process is started, there is a bunch of page faults
 - After a while, page fault rate should drop to a low level
- Prepaging
 - Bring pages even if they are not referred to
 - Once we have to go to the disk, makes sense to read more than one block if data is suitably located on the disk

Two main types:

Demand Paging

Prepaging

Replacement policy

- Which page should be kicked out when a new page has been loaded
- Principle of locality
 - Does not make sense to throw out recently referenced pages
- Efficiency
 - The more complex the replacement policy the more overhead is caused
- Frame locking
 - Some frames of memory need to be locked
 - E.g., some I/O devices and parts of kernel code and data need to be placed in fixed physical addresses

Replacement algorithms

- Optimal
 - Replace the page for which the time to next reference is the longest
 - impossible to implement but can be calculated by recording the references and analyzing them afterwards
- First In First Out (FIFO)
 - Replaces the page that has been longest in memory
 - SW implementation is easy: maintain a queue buffer
 - may throw out busy pages ⇒ often not even near the optimal
- Least Recently Used (LRU)
 - Replace the page that has not been referenced the longest time
 - Tag the references or maintain a list
 - Hardware acceleration would be needed, but is complicated
 - accesses are frequent, replacements are (typically) rare

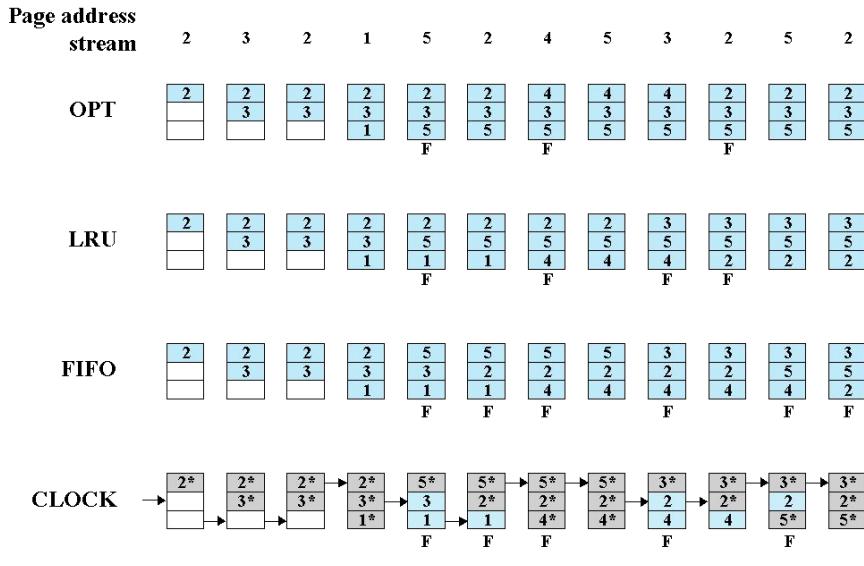


Figure 8.14 Behavior of Four Page-Replacement Algorithms

Clock algorithm

- Referenced bit
 - Use a bit per each frame to indicate if it is referenced
 - this can be done by MMU hardware (e.g., in TLB)
- Replacement
 - Frames are considered to be a circular buffer
 - If any of the frames have use bit zero \Rightarrow choose it for replacement
 - Each time use bit 1 is encountered \Rightarrow set it to zero
 - If all bits are 1 then a complete cycle has been done, and the first frame will be replaced
- Can be visualized as a clock diagram
 - Thus, we have the name, but there is no real clock!
- Can be improved by checking the dirty bit
 - This way algorithm prefers replacing pages that are clean

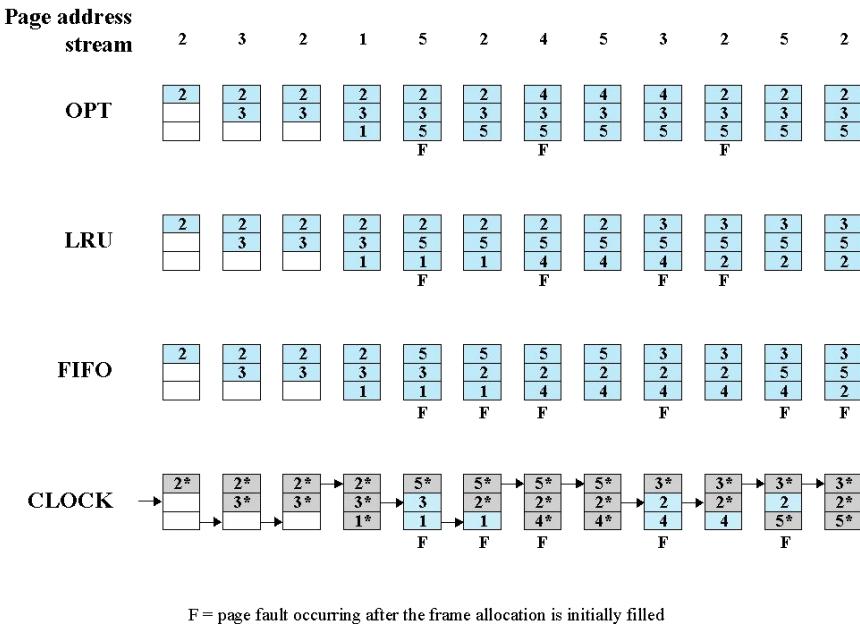
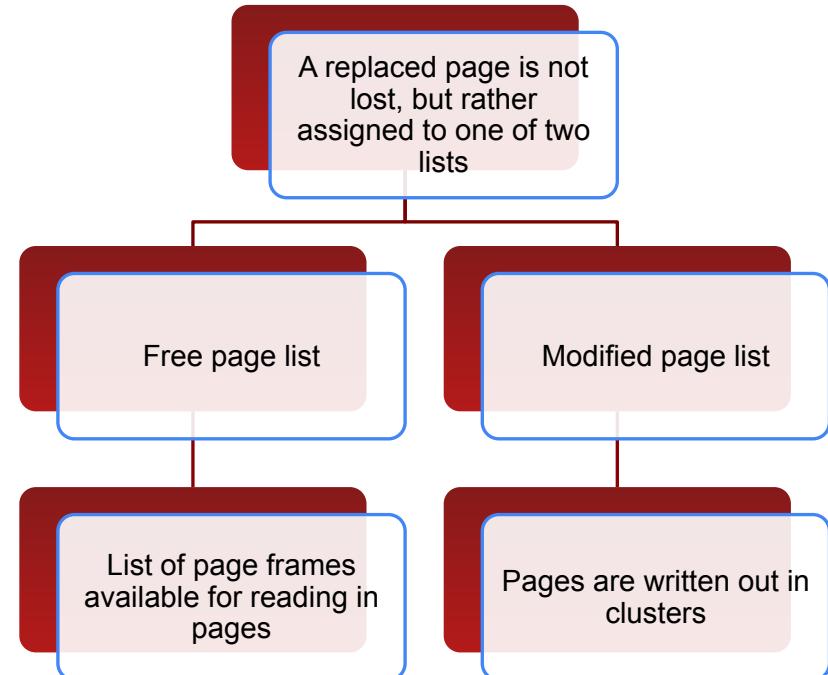


Figure 8.14 Behavior of Four Page-Replacement Algorithms

Cleaning policy

- Writing dirty pages to the secondary memory
 - Which pages? When?
- Demand cleaning
 - Write the page only if it is chosen for replacement
 - Page is only written if a new page is brought to memory
- Precleaning
 - Write modifies pages before the frames are needed
 - Allows bulk writing of page (but deciding which is hard...)
- Page buffering
 - A pool of free frames is maintained
 - When a page fault occurs, the desired page is read into a free frame from the pool. A victim frame is later swapped out if necessary and put into the free frames pool.



Operating Systems

CS-C3140, Lecture 6

Alexandru Paler

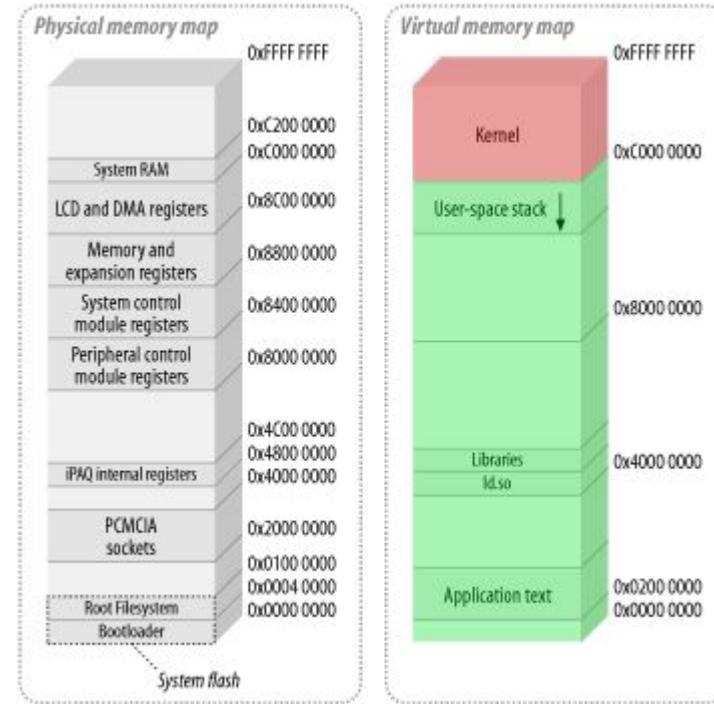
Grading

- The grade is based on points
 - The exercises: n (8?) rounds and 28 points/round -> $n \cdot 28$ points from exercises
 - The exam points are max. 400 - $n \cdot 28$
- To pass the course you have to pass both the exam and the exercises
 - A minimum number of points for the exam (to be announced separately)
 - Final grade = round $((\text{exercises} + \text{exam}) / 100)$
 - 449 -> 400 -> 4
 - 451 -> 500 -> 5
 - Per round limits for the exercises (you have to pass $(n-2)/n$ rounds)
 - 50% of max points for passing the assignments
 - 75% of max points if submitted one week later
 - 2 trials per exercise
 - Do not solve the exercises in the A+ interface
 - Use A+ only for submission
 - Solve in different software (e.g. Notepad)

Linux Memory Management

Linux Memory Management

- Linux memory
 - quite complex
 - shares many characteristics with UNIX
 - process virtual memory
 - kernel memory allocation
- It is traditional and good to have the **kernel mapped in every user process**
 - kernel at virtual addresses 0xC0000000 – 0xFFFFFFFF of every address space
 - the range 0x00000000 – 0xBFFFFFFF for user code, data, stacks, libraries, etc.
- Kernels that have such design are said to be "in the higher half" by opposition to kernels that use lowest virtual addresses for themselves, and leave higher addresses for the applications.



Address Types (Kernel)

- **User virtual addresses:** These are the regular addresses seen by user-space programs. User addresses are either 32 or 64 bits in length, depending on the underlying hardware architecture, and each process has its own virtual address space.
- **Physical addresses:** The addresses used between the processor and the system's memory. Physical addresses are 32- or 64-bit quantities; even 32-bit systems can use 64-bit physical addresses in some situations.
- Bus addresses: The addresses used between peripheral buses and memory. Often they are the same as the physical addresses used by the processor, but that is not necessarily the case.
- **Kernel logical addresses:** These make up the normal address space of the kernel. **These addresses map most or all of main memory, and are often treated as if they were physical addresses.** On most architectures, logical addresses and their associated physical addresses differ only by a constant offset. Logical addresses use the **hardware's native pointer size**, and thus may be unable to address all of physical memory on heavily equipped 32-bit systems.
- **Kernel virtual (linear) addresses:** These differ from logical addresses in that they **do not necessarily have a direct mapping to physical addresses.** All logical addresses are kernel virtual addresses; memory allocated by `vmalloc` also has a virtual address (but no direct physical mapping).

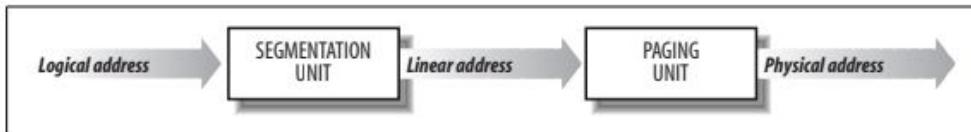
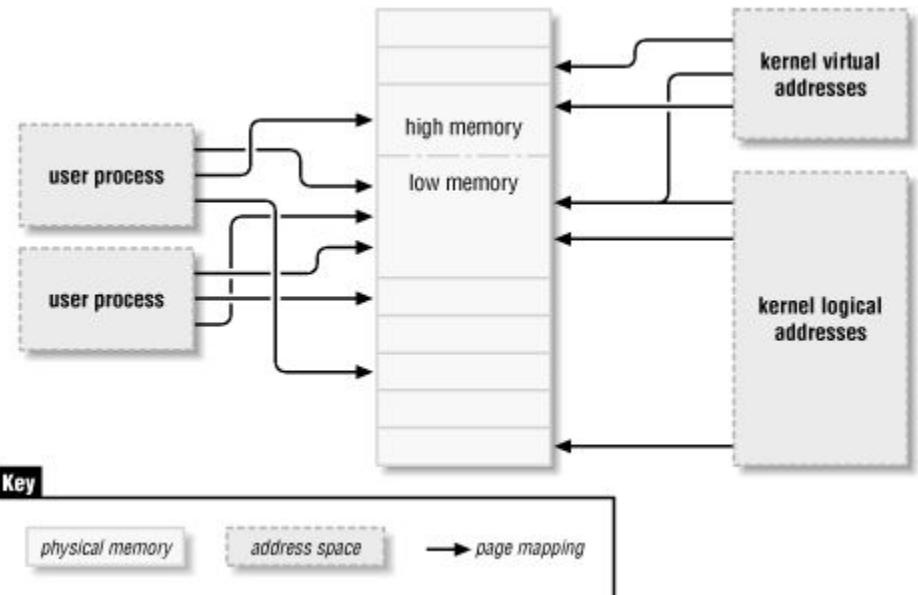
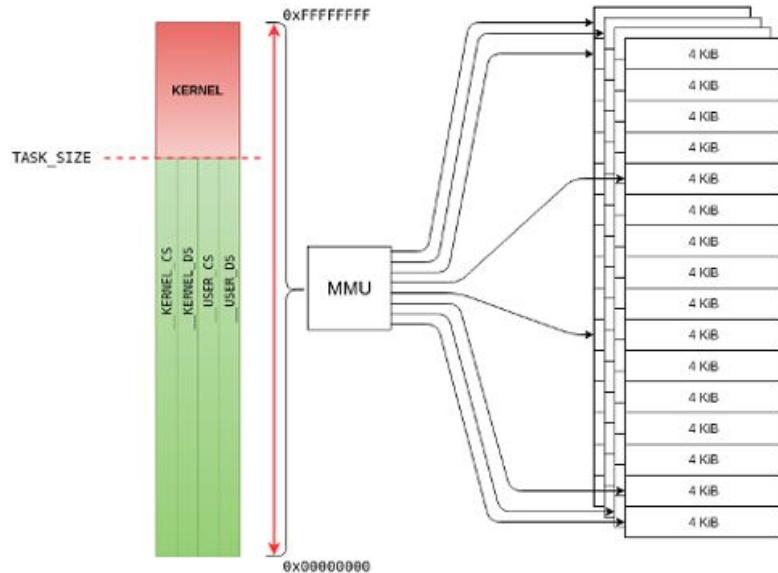


Figure 2-1. Logical address translation



Difference between logical and kernel virtual addresses

- With 32 bits, it is possible to address 4 GB of memory.
 - Linux was unable to handle more memory than it could set up logical addresses for
 - Linux needed directly mapped kernel addresses for all memory (kernel mapped in every user process – speed)
- Addresses in the range 0xC0000000 - 0xFFFFFFFF are kernel virtual addresses (red area).
 - The 896MiB range 0xC0000000 - 0xF7FFFFFF **directly maps kernel logical addresses 1:1 with kernel physical addresses** into the contiguous **lowmem** pages
 - The remaining 128MiB range 0xF8000000 - 0xFFFFFFFF is used to map virtual addresses for large buffer allocations (a window where parts of the extra RAM is mapped as needed), MMIO ports (Memory-Mapped I/O) and/or PAE memory into the not-contiguous **highmem** pages
 - lowmem is needed for per-process things including kernel stacks for every task (aka thread), and for page tables themselves
- Addresses in the range 0x00000000 - 0xBFFFFFFF
 - decision taken by Linus that user process virtual addresses start at 3GB (1GB virtual address for kernel was plenty many decades ago)
 - user virtual addresses (green area)
 - userland code, data and libraries reside
 - mapping can be in not-contiguous low-memory and high-memory pages.



Highmem and 64 bits

```
config NOHIGHMEM  
    bool "off"  
    help
```

Linux can use up to 64 Gigabytes of physical memory on x86 systems. However, the address space of 32-bit x86 processors is only 4 Gigabytes large. That means that, if you have a large amount of physical memory, not all of it can be "permanently mapped" by the kernel. The physical memory that's not permanently mapped is called "high memory".

If you are compiling a kernel which will never run on a machine with more than 1 Gigabyte total physical RAM, answer "off" here (default choice and suitable for most users). This will result in a "3GB/1GB" split: 3GB are mapped so that each process sees a 3GB virtual memory space and the remaining part of the 4GB virtual memory space is used by the kernel to permanently map as much physical memory as possible.

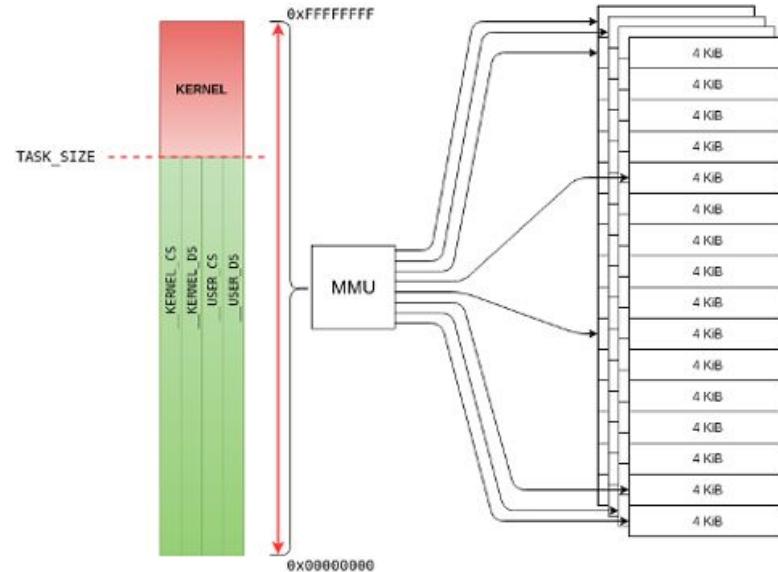
If the machine has between 1 and 4 Gigabytes physical RAM, then answer "4GB" here.

If more than 4 Gigabytes is used then answer "64GB" here. This selection turns Intel PAE (Physical Address Extension) mode on. PAE implements 3-level paging on IA32 processors. PAE is fully supported by Linux, PAE mode is implemented on all recent Intel processors (Pentium Pro and better). NOTE: If you say "64GB" here, then the kernel will not boot on CPUs that don't support PAE!

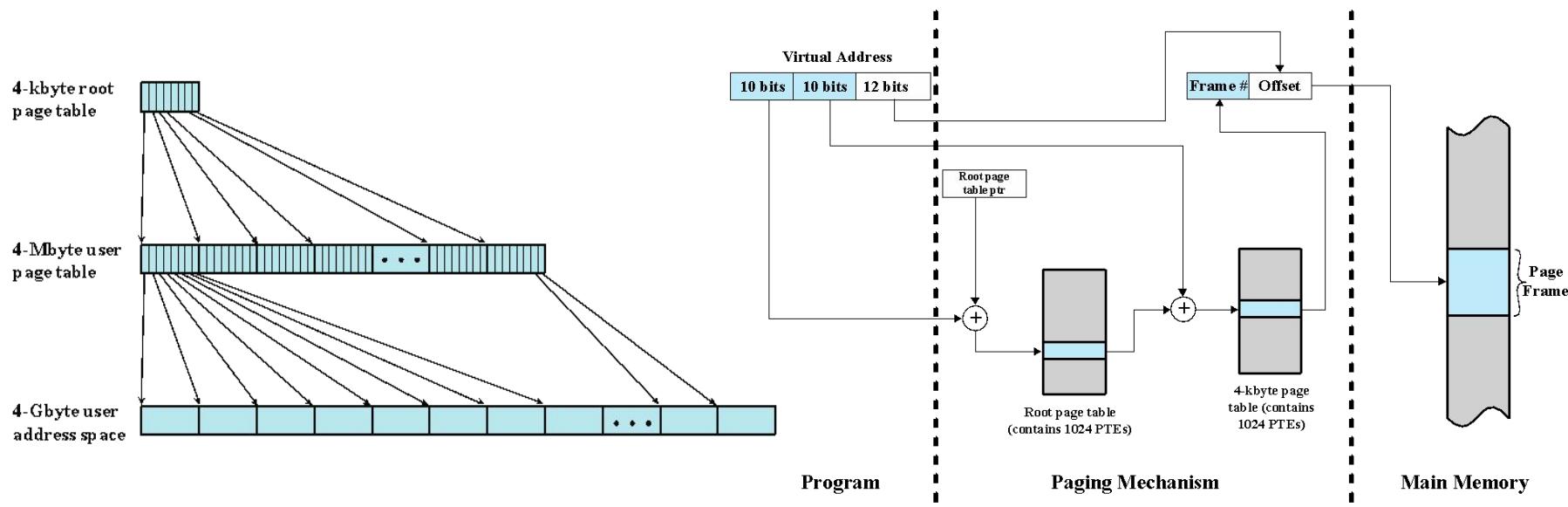
The actual amount of total physical memory will either be auto detected or can be forced by using a kernel command line option such as "mem=256M". (Try "man bootparam" or see the documentation of your boot loader (lilo or loadlin) about how to pass options to the kernel at boot time.)

If unsure, say "off".

<https://elixir.bootlin.com/linux/v5.8/source/arch/x86/Kconfig#L1363>



(Recap) Two-Level Hierarchical Page Table



Page Tables

Page Directory (PGD)

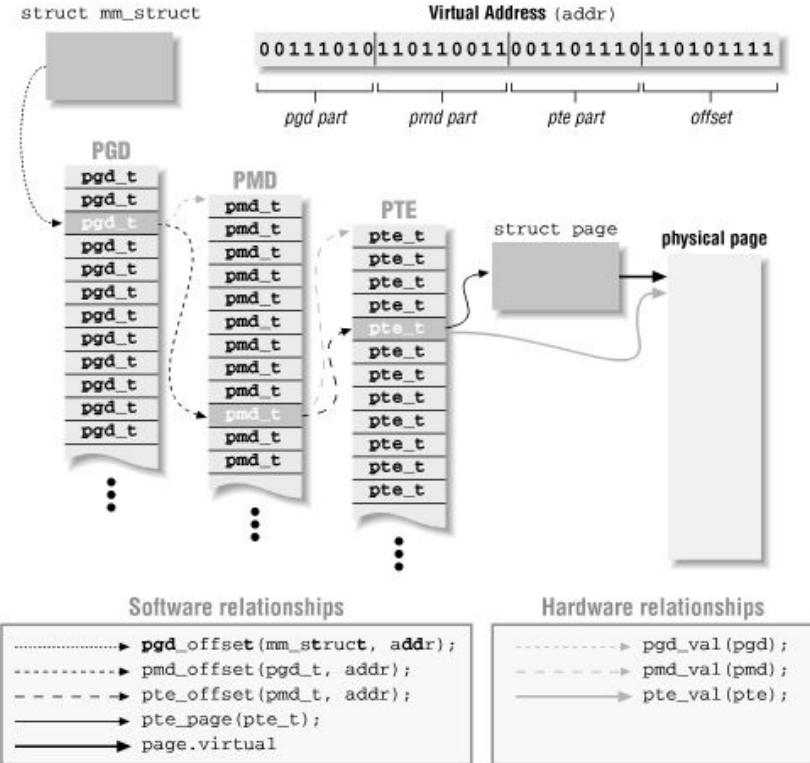
The top-level page table. The PGD is an array of pgd_t items, each of which points to a second-level page table. **Each process has its own page directory**, and there is one for kernel space as well. You can think of the page directory as a page-aligned array of pgd_ts.

Page mid-level Directory (PMD)

The second-level table. The PMD is a page-aligned array of pmd_t items. A pmd_t is a pointer to the third-level page table. Two-level processors have no physical PMD; they declare their PMD as an array with a single element, whose value is the PMD itself.

Page Table

A page-aligned array of items, each of which is called a Page Table Entry. The kernel uses the pte_t type for the items. A pte_t contains the physical address of the data page.

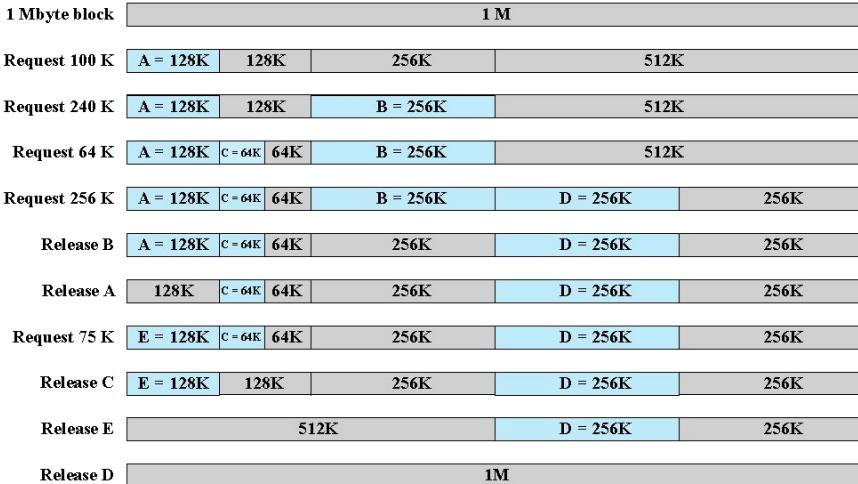


Kernel Memory Allocation

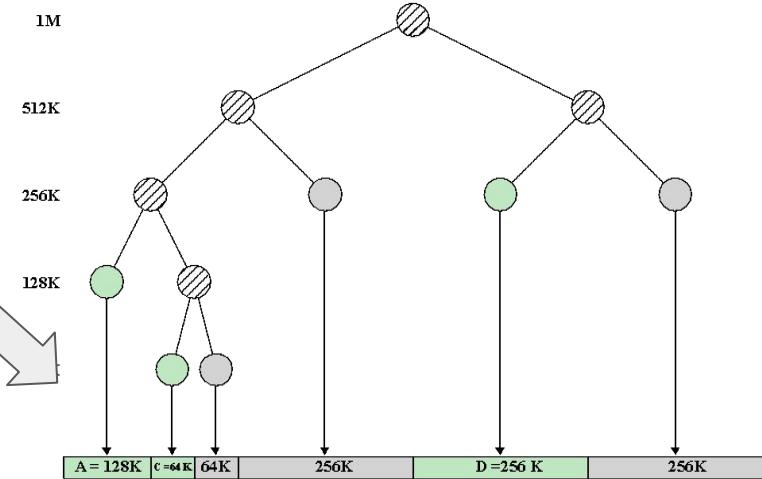
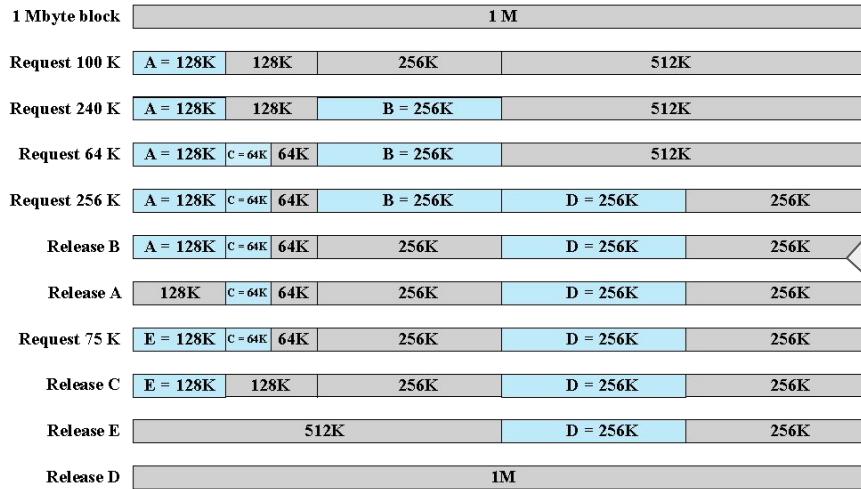
- Kernel memory capability manages physical main memory page frames
 - Primary function is to allocate and deallocate frames for particular uses
- Possible owners of a frame include:
- User-space processes
 - Dynamically allocated kernel data
 - Static kernel code
 - Page cache
- A buddy algorithm is used so that memory for the kernel can be allocated and deallocated in units of one or more pages
 - Page allocator alone would be inefficient because the kernel requires small short-term memory chunks in odd sizes
 - Slab allocation
 - On a x86 machine, the page size is 4 Kbytes, and chunks within a page may be allocated of sizes 32, 64, 128, 252, 508, 2,040, and 4,080 bytes.
 - Used by Linux to accommodate small chunks

Example: Buddy System

- Space available for allocation is treated as a single block
- Memory blocks are available of size 2^K words, $L \leq K \leq U$, where
 - 2^L = smallest size block that is allocated
 - 2^U = largest size block that is allocated; generally 2^U is the size of the entire memory available for allocation
- Fast algorithm for allocation and deallocation



Example of using a Buddy System



Leaf node for allocated block

Leaf node for unallocated block

Non-leaf node

Physical Page Allocation: Managing Free Blocks

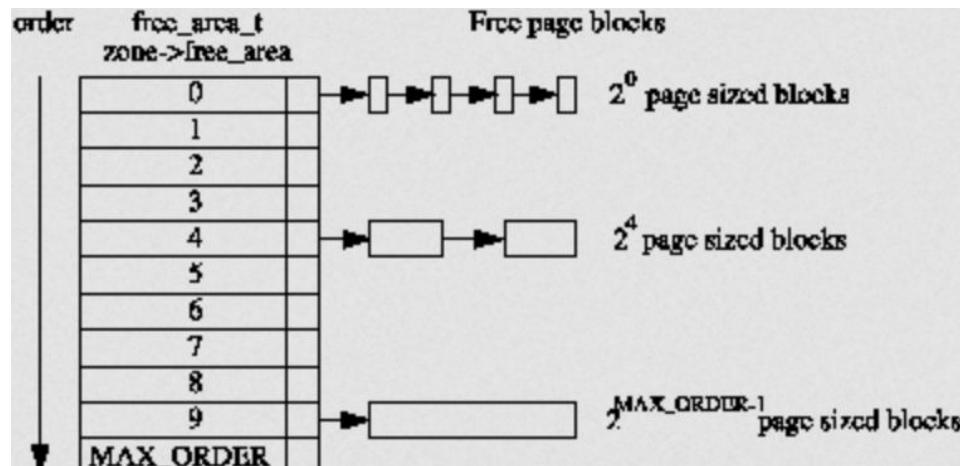
The allocator maintains blocks of free pages where each block is a power of two number of pages.

Eliminates the chance that a larger block will be split to satisfy a request where a smaller block would have sufficed.

The exponent for the power of two sized block is referred to as the *order*.

An array of `free_area_t` structs are maintained for each order that points to a linked list of **blocks of pages** that are free.

- the 0th element of the array will point to a list of free page blocks of size 2^0 or 1 page,
- the 1st element will be a list of 2^1
- up to $2^{MAX_ORDER-1}$ number of pages, where the `MAX_ORDER` is currently defined as 10



Physical Page Allocation

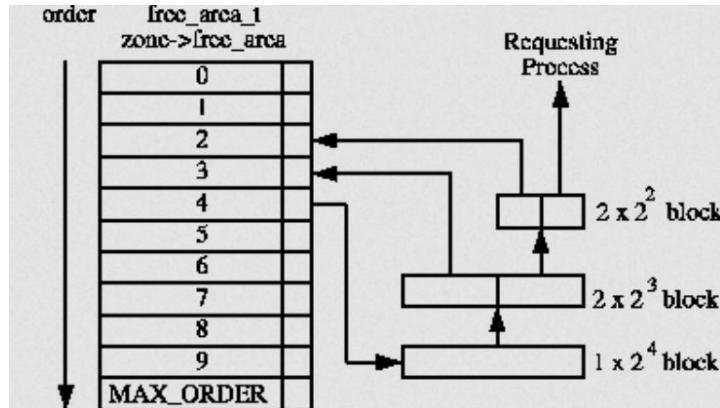
Allocations are always for a specified order

- 0 in the case where a single page is required
- If a free block cannot be found of the requested order, a higher order block is split into two buddies
- One is allocated and the other is placed on the free list for the lower order.

When the block is later freed

- the buddy will be checked
- If both are free, they are merged to form a higher order block and placed on the higher free list where its buddy is checked and so on.
- If the buddy is not free, the freed block is added to the free list at the current order

During these list manipulations, **interrupts have to be disabled** to prevent an interrupt handler manipulating the lists while a process has them in an inconsistent state.



A 2^4 block is split and the buddies are added to the free lists until a block for the process is available

Linux Page Replacement

- Prior to 2.6 based on the clock algorithm
 - The use bit is replaced with an 8-bit age variable
 - Incremented each time the page is accessed
- Periodically decrements the age bits
 - A page with an age of 0 is an “old” page that has not been referenced in some time and is the best candidate for replacement
- A form of least recently used (LRU) policy
 - **active_list** contains the working set of all processes
 - **inactive_list** contains reclaim candidates
 - replacement policy is global: reclaimable pages are contained in just two lists and pages belonging to any process may be reclaimed, rather than just those belonging to a faulting process

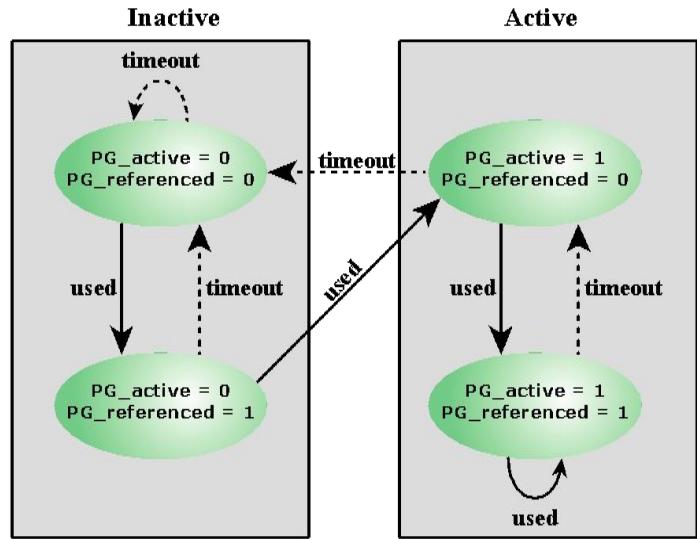


Figure 8.24 Linux Page Reclaiming

Page address stream	2	3	2	1	5	2	4	5	3	2	5	2
OPT	2	2	2	2	2	2	4	4	4	2	3	2
LRU	2	2	2	2	2	2	2	3	3	3	5	3

/proc/[pid]/maps

address	perms	offset	dev	inode	pathname
00400000-00452000	r-xp	00000000	08:02	173521	/usr/bin/dbus-daemon
00651000-00652000	r--p	00051000	08:02	173521	/usr/bin/dbus-daemon
00652000-00655000	rw-p	00052000	08:02	173521	/usr/bin/dbus-daemon
00e03000-00e24000	rw-p	00000000	00:00	0	[heap]
00e24000-011f7000	rw-p	00000000	00:00	0	[heap]
...					
35b1800000-35b1820000	r-xp	00000000	08:02	135522	/usr/lib64/ld-2.15.so
35b1a1f000-35b1a20000	r--p	0001f000	08:02	135522	/usr/lib64/ld-2.15.so
35b1a20000-35b1a21000	rw-p	00020000	08:02	135522	/usr/lib64/ld-2.15.so
35b1a21000-35b1a22000	rw-p	00000000	00:00	0	
35b1c00000-35b1dac000	r-xp	00000000	08:02	135870	/usr/lib64/libc-2.15.so
35b1dac000-35b1fac000	---p	001ac000	08:02	135870	/usr/lib64/libc-2.15.so
35b1fac000-35b1fb0000	r--p	001ac000	08:02	135870	/usr/lib64/libc-2.15.so
35b1fb0000-35b1fb2000	rw-p	001b0000	08:02	135870	/usr/lib64/libc-2.15.so
...					
f2c6ff8c000-7f2c7078c000	rw-p	00000000	00:00	0	[stack:986]
...					
7fff2c0d000-7fff2c2e000	rw-p	00000000	00:00	0	[stack]
7fff2d48000-7fff2d49000	r-xp	00000000	00:00	0	[vdso]

- The address field is the address space in the process that the mapping occupies.
- The perms field is a set of permissions:
 - r = read w = write x = execute
 - s = shared p = private (copy on write)
- The offset field is the offset into the file/whatever;
- dev is the device (major:minor);
- inode is the inode on that device. 0 indicates that no inode is associated with the memory region, as would be the case with BSS (uninitialized data).
- The pathname field will usually be the file that is backing the mapping.

Additional helpful pseudo-paths

- [stack] The initial process's (also known as the main thread's) stack.
- [heap] The process's heap.
- [stack:<tid>] (from Linux 3.4 to 4.4) A thread's stack (where the <tid> is a thread ID). Was removed in Linux 4.5, since providing this information for a process with large numbers of threads is expensive.
- [vdso] The virtual dynamically linked shared object.

C Tutorial

Stack and Heap

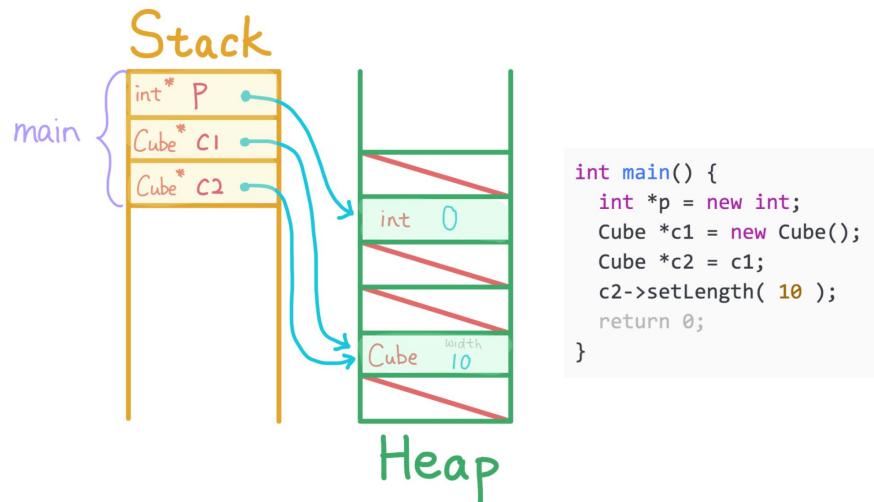
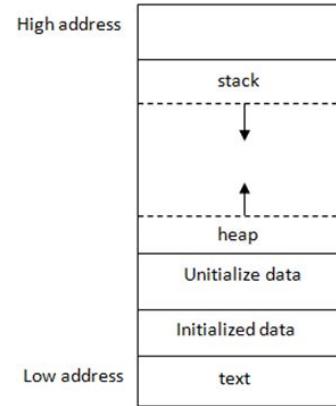
A running program takes up memory

- users are often not aware of allocated memory
- the program is allocating memory for each variable
- each byte of memory has an address

Each running program has its own memory layout, separated from other programs. The layout consists of a lot of segments, including

- stack: stores local variables
- heap: dynamic memory for programmer to allocate
- data: stores global variables, separated into initialized and uninitialized
- text: stores the code being executed

Heap segments have low address numbers, while the stack memory has higher addresses.



<https://web.stanford.edu/class/archive/cs/cs107/cs107.1212/lectures/4/Lecture4.pdf>

<https://web.stanford.edu/class/archive/cs/cs107/cs107.1212/lectures/6/Lecture6.pdf>

<https://web.stanford.edu/class/archive/cs/cs107/cs107.1212/lectures/7/Lecture7.pdf>

Operating Systems

CS-C3140, Lecture 7

Concurrency: Mutual Exclusion and Synchronization

Alexandru Paler

Multiple Processes

- Operating System design is concerned with the management of processes and threads
- Multiprogramming
 - The management of multiple processes within a uniprocessor system
- Multiprocessing
 - The management of multiple processes within a multiprocessor
- Distributed Processing
 - The management of multiple processes executing on multiple, distributed computer systems
 - The recent proliferation of clusters is a prime example of this type of system

Concurrency Arises in Three Different Contexts

Multiple Applications

Invented to allow processing time to be shared among active applications

Structured Applications

Extension of modular design and structured programming

Operating System Structure

OS themselves implemented as a set of processes or threads

Key Terms Related to Concurrency

atomic operation	A function or action implemented as a sequence of one or more instructions that appears to be indivisible; that is, no other process can see an intermediate state or interrupt the operation. The sequence of instruction is guaranteed to execute as a group, or not execute at all, having no visible effect on system state. Atomicity guarantees isolation from concurrent processes.
critical section	A section of code within a process that requires access to shared resources and that must not be executed while another process is in a corresponding section of code.
deadlock	A situation in which two or more processes are unable to proceed because each is waiting for one of the others to do something.
livelock	A situation in which two or more processes continuously change their states in response to changes in the other process(es) without doing any useful work.
mutual exclusion	The requirement that when one process is in a critical section that accesses shared resources, no other process may be in a critical section that accesses any of those shared resources.
race condition	A situation in which multiple threads or processes read and write a shared data item and the final result depends on the relative timing of their execution.
starvation	A situation in which a runnable process is overlooked indefinitely by the scheduler; although it is able to proceed, it is never chosen.

Mutual Exclusion: Software Approaches

- Software approaches can be implemented for concurrent processes that execute on a single-processor or a multiprocessor machine with shared main memory
- These approaches usually assume elementary mutual exclusion at the memory access level
 - Simultaneous accesses (reading and/or writing) to the same location in main memory are serialized by some sort of memory arbiter, although the order of access granting is not specified ahead of time
 - Beyond this, no support in the hardware, operating system, or programming language is assumed
- Dijkstra reported an algorithm for mutual exclusion for two processes, designed by the Dutch mathematician Dekker

Mutual Exclusion Attempts (1)

/* PROCESS 0 */ : while (turn != 0) /* do nothing */; /* critical section */; turn = 1; : ;	/* PROCESS 1 */ : while (turn != 1) /* do nothing */; /* critical section */; turn = 0; : ;
--	--

(a) First attempt

/* PROCESS 0 */ : while (flag[1]) /* do nothing */; flag[0] = true; /*critical section*/; flag[0] = false; : ;	/* PROCESS 1 */ : while (flag[0]) /* do nothing */; flag[1] = true; /* critical section */; flag[1] = false; : ;
--	--

(b) Second attempt

Figure 5.1 Mutual Exclusion Attempts (page 1)

P0 executes the while statement and finds flag[1] set to false
P1 executes the while statement and finds flag[0] set to false
P0 sets flag[0] to true and enters its critical section
P1 sets flag[1] to true and enters its critical section

Because both processes are now in their critical sections, the program is incorrect.
The problem is that the proposed solution is not independent of relative process execution speeds.

First Attempt

- Reserve a global memory location labeled turn.
- A process examines the contents of turn.
 - If the value of turn is equal to the number of the process, then the process may proceed to its critical section.
 - Otherwise, it is forced to wait.
 - The waiting process repeatedly reads the value of turn until it is allowed to enter its critical section.
- This procedure is known as busy waiting, or spin waiting, because the thwarted process can do nothing productive until it gets permission to enter its critical section.
- A serious problem is that if one process fails, the other process is permanently blocked. This is true whether a process fails in its critical section or outside of it.

Second Attempt

- We need state information about both processes.
- Each process should have its own key to the critical section so that if one fails, the other can still access its critical section.
- A Boolean vector flag is defined, with flag[0] corresponding to P0 and flag[1] corresponding to P1.
- Each process may examine the other's flag but may not alter it. When it leaves its critical section, it sets its flag to false.

Dekker's Algorithm

```
boolean flag [2];
int turn;
void P0()
{
    while (true) {
        flag [0] = true;
        while (flag [1]) {
            if (turn == 1) {
                flag [0] = false;
                while (turn == 1) /* do nothing
*/;
                flag [0] = true;
            }
        } /* critical section */;
        turn = 1;
        flag [0] = false;
        /* remainder */;
    }
}
void P1( )
{
    while (true) {
        flag [1] = true;
        while (flag [0]) {
            if (turn == 0) {
                flag [1] = false;
                while (turn == 0) /* do nothing
*/;
                flag [1] = true;
            }
        } /* critical section */;
        turn = 0;
        flag [1] = false;
        /* remainder */;
    }
}
void main ()
{
    flag [0] = false;
    flag [1] = false;
    turn = 1;
    parbegin (P0, P1);
}
```

Figure 5.2 Dekker's Algorithm

A solution

- the array variable **flag** is for observing the state of both processes
- the variable **turn** impose an order on the activities of the two processes in order to avoid the problem of "mutual courtesy" – which process has the right to insist on entering its critical region.

When P0 wants to enter its critical section

- it sets its flag to true.
- It then checks the flag of P1.
 - If that is false , P0 may immediately enter its critical section.
 - Otherwise, P0 consults turn.
 - If P0 finds that turn = 0 , then it knows that it is its turn to insist and periodically checks P1's flag.
 - P1 will at some point note that it is its turn to defer and set its to flag false , allowing P0 to proceed.
- After P0 has used its critical section, it sets its flag to false to free the critical section, and sets turn to 1 to transfer the right to insist to P1

Principles of Concurrency

- Interleaving and overlapping
 - Can be viewed as examples of concurrent processing
 - Both present the same problems
 - Uniprocessor – the relative speed of execution of processes cannot be predicted
 - Depends on activities of other processes
 - The way the OS handles interrupts
 - Scheduling policies of the OS

Difficulties of Concurrency

- The sharing of global resources is fraught with peril
 - two processes both make use of the same global variable and perform reads and writes on that variable
 - the order in which the various reads and writes are executed is critical
- It is difficult for the OS to manage the allocation of resources optimally
 - a process may request use of, and be granted control of, a particular I/O channel and then be suspended before using that channel
 - it may be undesirable for the OS simply to lock the channel and prevent its use by other processes
 - may lead to a deadlock condition
- Very difficult to locate a programming error because results are typically not deterministic and reproducible

All of the foregoing difficulties present themselves in a multiprocessor system, as well

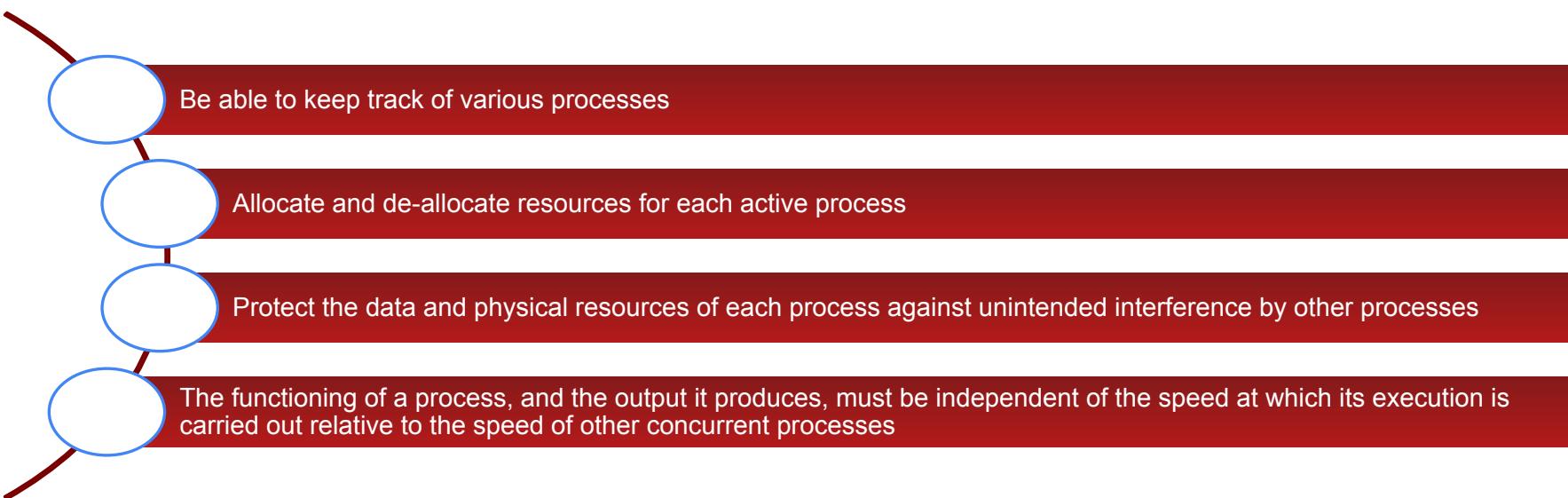
- the relative speed of execution of processes is unpredictable
- deal with problems arising from the simultaneous execution of multiple processes

Race Condition

- A race condition occurs when multiple processes or threads read and write data items so that the final result depends on the order of execution of instructions in the multiple processes. Let us consider two simple examples.
- First example
 - two processes, P1 and P2, share the global variable **a**
 - at some point in its execution, P1 updates **a** to the value 1
 - at some point in its execution, P2 updates **a** to the value 2
 - the two tasks are in a **race to write variable a**
 - the “loser” of the race (the process that updates last) determines the final value of **a**
- Second example
 - two process, P3 and P4, that share global variables **b** and **c** , with initial values $b = 1$ and $c = 2$
 - at some point in its execution, P3 executes the assignment $b = b + c$
 - at some point in its execution, P4 executes the assignment $c = b + c$
 - the final values of **b** and **c** depend on the order in which the two processes execute
 - if P3 executes its assignment statement first, then the final values are $b = 3$ and $c = 5$
 - if P4 executes its assignment statement first, then the final values are $b = 4$ and $c = 3$

Operating System Concerns

- Design and management issues raised by the existence of concurrency. The OS must:



Be able to keep track of various processes

Allocate and de-allocate resources for each active process

Protect the data and physical resources of each process against unintended interference by other processes

The functioning of a process, and the output it produces, must be independent of the speed at which its execution is carried out relative to the speed of other concurrent processes

Process Interaction

Degree of Awareness	Relationship	Influence that One Process Has on the Other	Potential Control Problems
Processes unaware of each other	Competition	<ul style="list-style-type: none">•Results of one process independent of the action of others•Timing of process may be affected	<ul style="list-style-type: none">•Mutual exclusion•Deadlock (renewable resource)•Starvation
Processes indirectly aware of each other (e.g., shared object)	Cooperation by sharing	<ul style="list-style-type: none">•Results of one process may depend on information obtained from others•Timing of process may be affected	<ul style="list-style-type: none">•Mutual exclusion•Deadlock (renewable resource)•Starvation•Data coherence
Processes directly aware of each other (have communication primitives available to them)	Cooperation by communication	<ul style="list-style-type: none">•Results of one process may depend on information obtained from others•Timing of process may be affected	<ul style="list-style-type: none">•Deadlock (consumable resource)•Starvation

1. Resource Competition

- Concurrent processes come into conflict when they are competing for use of the same resource
 - For example: I/O devices, memory, processor time, clock
 - In the case of competing processes three control problems must be faced:
 - The need for mutual exclusion
 - **Deadlock:**
 - For example, consider two processes, P1 and P2, and two resources, R1 and R2. Suppose that each process needs access to both resources to perform part of its function. Then it is possible to have the following situation: the OS assigns R1 to P2, and R2 to P1. Each process is waiting for one of the two resources. Neither will release the resource that it already owns until it has acquired the other resource and performed the function requiring both resources. The two processes are deadlocked.
 - **Starvation:**
 - Suppose that three processes (P1, P2, P3) each require periodic access to resource R. Consider the situation in which P1 is in possession of the resource, and both P2 and P3 are delayed, waiting for that resource. When P1 exits its critical section, either P2 or P3 should be allowed access to R. Assume that the OS grants access to P3 and that P1 again requires access before P3 completes its critical section. If the OS grants access to P1 after P3 has finished, and subsequently alternately grants access to P1 and P3, then P2 may indefinitely be denied access to the resource, even though there is no deadlock situation.

1. Resource Competition

The diagram shows three boxes representing processes. The first box is labeled 'PROCESS 1 /*' and contains code for process P1. The second box is labeled '/* PROCESS 2 /*' and contains code for process P2. The third box is labeled '/* PROCESS n /*' and contains code for process Pn. Each process has a similar structure: a while loop that runs true, followed by code preceding the critical section, then the entercritical function, then the critical section (indicated by /* critical section */), then the exitcritical function, and finally code following the critical section. An ellipsis between the second and third boxes indicates that there are more processes.

```
PROCESS 1 /*  
void P1  
{  
    while (true) {  
        /* preceding code */;  
        entercritical (Ra);  
        /* critical section */;  
        exitcritical (Ra);  
        /* following code */;  
    }  
}  
  
/* PROCESS 2 /*  
void P2  
{  
    while (true) {  
        /* preceding code */;  
        entercritical (Ra);  
        /* critical section */;  
        exitcritical (Ra);  
        /* following code */;  
    }  
}  
  
...  
  
/* PROCESS n /*  
void Pn  
{  
    while (true) {  
        /* preceding code */;  
        entercritical (Ra);  
        /* critical section */;  
        exitcritical (Ra);  
        /* following code */;  
    }  
}
```

Figure 5.4 Illustration of Mutual Exclusion

Control of competition

- processes need to be able to express the requirement for mutual exclusion in some fashion, such as **locking** a resource prior to its use
- involves the OS, such as the provision of the locking facility

There are n processes to be executed concurrently. Each process includes:

- a critical section that operates on some resource
- additional code preceding and following the critical section that does not involve access to the resource
- any process that attempts to enter its critical section while another process is in its critical section, for the same resource, is made to wait

To enforce mutual exclusion, two functions are provided: entercritical and exitcritical

2. Cooperation by Sharing

Covers processes that interact with other processes without being explicitly aware of them

Processes may use and update the shared data without reference to other processes, but know that other processes may have access to the same data

Thus the processes must cooperate to ensure that the data they share are properly managed

The control mechanisms must ensure the integrity of the shared data

Because data are held on resources (devices, memory), the control problems of mutual exclusion, deadlock, and starvation are again present

- The only difference is that data items may be accessed in two different modes, reading and writing, and only writing operations must be mutually exclusive

3. Cooperation by Communication

- The various processes participate in a common effort that links all of the processes
- The communication provides a way to synchronize, or coordinate, the various activities
- Communication can be characterized as consisting of messages of some sort
- Primitives for sending and receiving messages may be provided as part of the programming language or provided by the OS kernel
- Mutual exclusion is not a control requirement for this sort of cooperation
- The problems of deadlock and starvation are still present

Requirements for Mutual Exclusion

- Any facility or capability that is to provide support for mutual exclusion should meet the following requirements
 - Mutual exclusion must be enforced: only one process at a time is allowed into its critical section, among all processes that have critical sections for the same resource or shared object
 - A process that halts must do so without interfering with other processes
 - It must not be possible for a process requiring access to a critical section to be delayed indefinitely: **no deadlock or starvation**
 - When no process is in a critical section, any process that request entry to its critical section must be permitted to enter without delay
 - No assumptions are made about relative process speeds or number of processes
 - A process remains inside its critical section for a finite time only

Mutual Exclusion: Hardware Support (1)

Interrupt Disabling

- In a uniprocessor system
 - concurrent processes cannot have overlapped execution
 - can only be interleaved
- A process will continue to run until it invokes an OS service or until it is interrupted
- To guarantee mutual exclusion, it is sufficient to prevent a process from being interrupted
- This capability can be provided in the form of primitives defined by the OS kernel for disabling and enabling interrupts

Disadvantages:

- The efficiency of execution could be noticeably degraded because the processor is limited in its ability to interleave processes
- This approach will not work in a multiprocessor architecture

Mutual Exclusion: Hardware Support (2)

- Compare&Swap Instruction

- Also called a “compare and exchange instruction”
- A compare is made between a memory value and a test value
- If the values are the same a swap occurs
- Carried out atomically (not subject to interruption)

```
/* program mutual exclusion */
const int n = /* number of processes */;
int bolt;
void P(int i)
{
    while (true) {
        while (compare_and_swap(&bolt, 0, 1) == 1)
            /* do nothing */;
        /* critical section */;
        bolt = 0;
        /* remainder */;
    }
}
void main()
{
    bolt = 0;
    parbegin (P(1), P(2), . . . , P(n));
}
```

(a) Compare and swap instruction

```
/* program mutual exclusion */
int const n = /* number of processes*/;
int bolt;
void P(int i)
{
    while (true) {
        int keyi = 1;
        do exchange (&keyi, &bolt)
        while (keyi != 0);
        /* critical section */;
        bolt = 0;
        /* remainder */;
    }
}
void main()
{
    bolt = 0;
    parbegin (P(1), P(2), . . . , P(n));
}
```

(b) Exchange instruction

Figure 5.5 Hardware Support for Mutual Exclusion

Special Machine Instruction:

- **Advantages**
- Applicable to any number of processes on either a single processor or multiple processors sharing main memory
- Simple and easy to verify
- It can be used to support multiple critical sections; each critical section can be defined by its own variable
- **Disadvantages**
- Busy-waiting is employed
 - Thus while a process is waiting for access to a critical section it continues to consume processor time
- Deadlock is possible
- Starvation is possible
 - When a process leaves a critical section and more than one process is waiting, the selection of a waiting process is arbitrary; some process could indefinitely be denied access

Common Concurrency Mechanisms

Semaphore	An integer value used for signaling among processes. Only three operations may be performed on a semaphore, all of which are atomic: initialize, decrement, and increment. The decrement operation may result in the blocking of a process, and the increment operation may result in the unblocking of a process. Also known as a counting semaphore or a general semaphore
Binary Semaphore	A semaphore that takes on only the values 0 and 1.
Mutex	Similar to a binary semaphore. A key difference between the two is that the process that locks the mutex (sets the value to zero) must be the one to unlock it (sets the value to 1).
Condition Variable	A data type that is used to block a process or thread until a particular condition is true.
Monitor	A programming language construct that encapsulates variables, access procedures and initialization code within an abstract data type. The monitor's variable may only be accessed via its access procedures and only one process may be actively accessing the monitor at any one time. The access procedures are <i>critical sections</i> . A monitor may have a queue of processes that are waiting to access it.
Event Flags	A memory word used as a synchronization mechanism. Application code may associate a different event with each bit in a flag. A thread can wait for either a single event or a combination of events by checking one or multiple bits in the corresponding flag. The thread is blocked until all of the required bits are set (AND) or until at least one of the bits is set (OR).
Mailboxes/Messages	A means for two processes to exchange information and that may be used for synchronization.
Spinlocks	Mutual exclusion mechanism in which a process executes in an infinite loop waiting for the value of a lock variable to indicate availability.

Semaphore

A variable that has an integer value upon which only three operations are defined:

- There is no way to inspect or manipulate semaphores other than these three operations

- 1) A semaphore may be initialized to a nonnegative integer value
- 2) The semWait operation decrements the semaphore value
- 3) The semSignal operation increments the semaphore value

Consequences

There is no way to know before a process decrements a semaphore whether it will block or not

There is no way to know which process will continue immediately on a uniprocessor system when two processes are running concurrently

You don't know whether another process is waiting so the number of unblocked processes may be zero or one

General Semaphores

```
struct semaphore {
    int count;
    queueType queue;
};

void semWait(semaphore s)
{
    s.count--;
    if (s.count < 0) {
        /* place this process in s.queue */;
        /* block this process */;
    }
}

void semSignal(semaphore s)
{
    s.count++;
    if (s.count <= 0) {
        /* remove a process P from s.queue */;
        /* place process P on ready list */;
    }
}
```

Figure 5.6 A Definition of Semaphore Primitives

- A queue is used to hold processes waiting on the semaphore
- Strong semaphores
 - The process that has been blocked the longest is released from the queue first (FIFO)
- Weak semaphores
 - The order in which processes are removed from the queue is not specified

Binary Semaphore

```
struct binary semaphore {
    enum {zero, one} value;
    queueType queue;
};

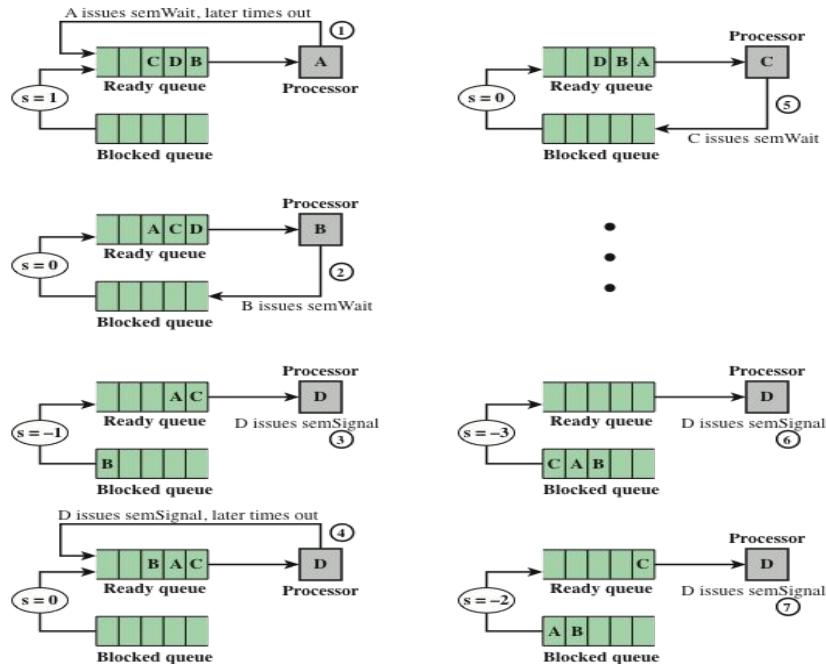
void semWaitB(binary semaphore s)
{
    if (s.value == one)
        s.value = zero;
    else {
        /* place this process in s.queue */;
        /* block this process */;
    }
}

void semSignalB(semaphore s)
{
    if (s.queue is empty())
        s.value = one;
    else {
        /* remove a process P from s.queue */;
        /* place process P on ready list */;
    }
}
```

Figure 5.7 A Definition of Binary Semaphore Primitives

- A binary semaphore may be initialized to 0 or 1.
- The semWaitB operation
 - checks the semaphore value
 - If the value is zero, then the process executing the semWaitB is blocked
 - If the value is one, then the value is changed to zero and the process continues execution.
- The semSignalB operation
 - checks to see if any processes are blocked on this semaphore (semaphore value equals 0). If so, then a process blocked by a semWaitB operation is unblocked. If no processes are blocked, then the value of the semaphore is set to one.
- Mutual exclusion lock (mutex)
 - concept related to the binary semaphore
 - a programming flag used to grab and release an object
 - set to lock (typically zero), which blocks other attempts to use it.
 - set to unlock when the data are no longer needed or the routine is finished
- A key difference between the a mutex and a binary semaphore
 - mutex: the process that locks (sets the value to zero) must be the one to unlock it (sets the value to 1)
 - semaphore: it is possible for one process to lock a binary semaphore and for another to unlock it

Example: Strong Semaphore



Processes A, B, and C depend on a result from process D.

(1), A is running; B, C, and D are ready; and the semaphore count is 1, indicating that one of D's results is available. When A issues a semWait instruction on semaphore s , the semaphore decrements to 0, and A can continue to execute; subsequently it rejoins the ready queue.

(2) Then B runs , eventually issues a semWait instruction, and is blocked, allowing D to run

(3). When D completes a new result, it issues a semSignal instruction, which allows B to move to the ready queue

(4) D rejoins the ready queue and C begins to run.

(5) but is blocked when it issues a semWait instruction. Similarly, A and B run and are blocked on the semaphore, allowing D to resume execution

(6). When D has a result, it issues a semSignal , which transfers C to the ready queue. Later cycles of D will release A and B from the Blocked state.

Figure 5.8 Example of Semaphore Mechanism

Semaphore for Mutual Exclusion

```
/* program mutual exclusion */
const int n = /* number of processes */;
semaphore s = 1;
void P(int i)
{
    while (true) {
        semWait(s);
        /* critical section */
        semSignal(s);
        /* remainder */
    }
}
void main()
{
    parbegin (P(1), P(2), . . . , P(n));
}
```

Figure 5.9 Mutual Exclusion Using Semaphores

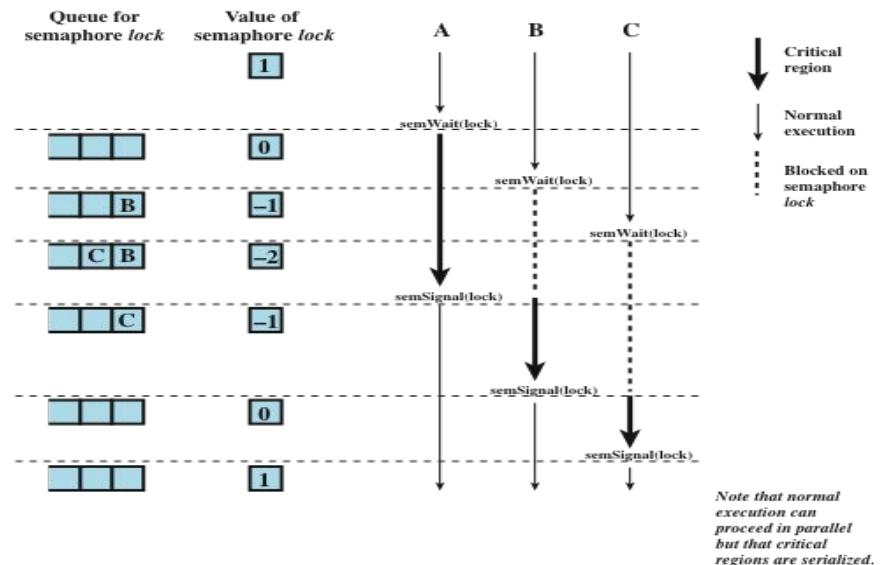


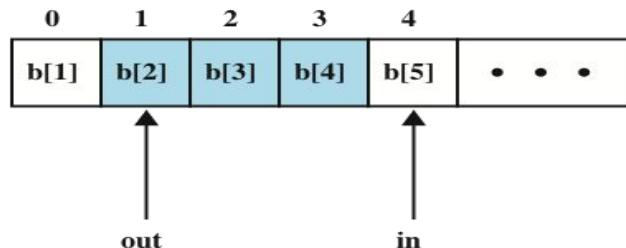
Figure 5.10 Processes Accessing Shared Data Protected by a Semaphore

Producer/Consumer Problem

General Statement:

- One or more producers are generating data and placing these in a buffer
- A single consumer is taking items out of the buffer one at a time
- Only one producer or consumer may access the buffer at any one time

The Problem: Ensure that the producer won't try to add data into the buffer if its full, and that the consumer won't try to remove data from an empty buffer



Note: shaded area indicates portion of buffer that is occupied

Figure 5.11 Infinite Buffer for the Producer/Consumer Problem

Incorrect Solution:

```
/* program producerconsumer */
int n;
binary_semaphore s = 1, delay = 0;
void producer()
{
    while (true) {
        produce();
        semWaitB(s);
        append();
        n++;
        if (n==1) semSignalB(delay);
        semSignalB(s);
    }
}
void consumer()
{
    semWaitB(delay);
    while (true) {
        semWaitB(s);
        take();
        n--;
        semSignalB(s);
        consume();
        if (n==0) semWaitB(delay);
    }
}
void main()
{
    n = 0;
    parbegin (producer, consumer);
}
```

Figure 5.12 An Incorrect Solution to the Infinite-Buffer Producer/Consumer Problem Using Binary Semaphores

	Producer	Consumer	s	n	Delay
1			1	0	0
2	semWaitB(s)		0	0	0
3	n++		0	1	0
4	if (n==1) (semSignalB(delay))		0	1	1
5	semSignalB(s)		1	1	1
6		semWaitB(delay)	1	1	0
7		semWaitB(s)	0	1	0
8		n--	0	0	0
9		semSignalB(s)	1	0	0
10	semWaitB(s)		0	0	0
11	n++		0	1	0
12	if (n==1) (semSignalB(delay))		0	1	1
13	semSignalB(s)		1	1	1
14		if (n==0) (semWaitB(delay))	1	1	1
15		semWaitB(s)	0	1	1
16		n--	0	0	1
17		semSignalB(s)	1	0	1
18		if (n==0) (semWaitB(delay))	1	0	0
19		semWaitB(s)	0	0	0
20		n--	0	-1	0
21		semSignalB(s)	1	-1	0

Implementation of Semaphores

- semWait and semSignal operations be implemented as atomic primitives
- Can be implemented in hardware or firmware
- Software schemes such as Dekker's or Peterson's algorithms can be used
- Alternative: use one of the hardware-supported schemes for mutual exclusion

```
semWait(s)
{
    while (compare_and_swap(s.flag, 0 , 1) == 1)
        /* do nothing */;
    s.count--;
    if (s.count < 0) {
        /* place this process in s.queue*/;
        /* block this process (must also set s.flag to 0) */
    }
    s.flag = 0;
}

semSignal(s)
{
    while (compare_and_swap(s.flag, 0 , 1) == 1)
        /* do nothing */;
    s.count++;
    if (s.count <= 0) {
        /* remove a process P from s.queue */;
        /* place process P on ready list */;
    }
    s.flag = 0;
}
```

(a) Compare and Swap Instruction

```
semWait(s)
{
    inhibit interrupts;
    s.count--;
    if (s.count < 0) {
        /* place this process in s.queue */;
        /* block this process and allow interrupts */;
    }
    else
        allow interrupts;
}

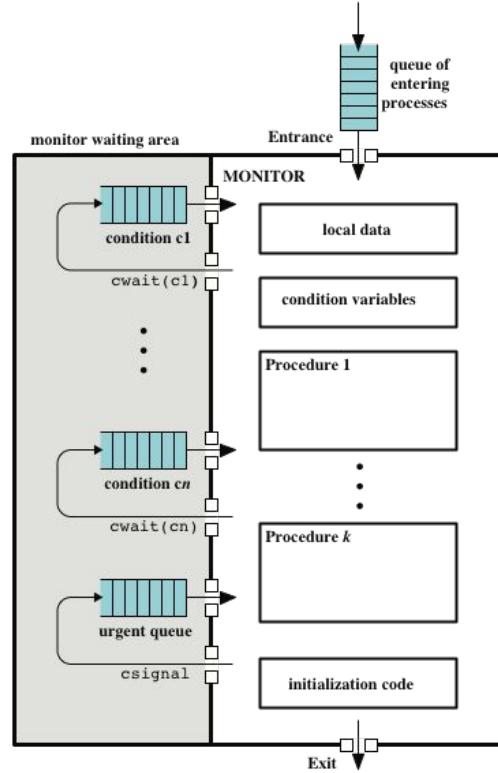
semSignal(s)
{
    inhibit interrupts;
    s.count++;
    if (s.count <= 0) {
        /* remove a process P from s.queue */;
        /* place process P on ready list */;
    }
    allow interrupts;
}
```

(b) Interrupts

Figure 5.17 Two Possible Implementations of Semaphores

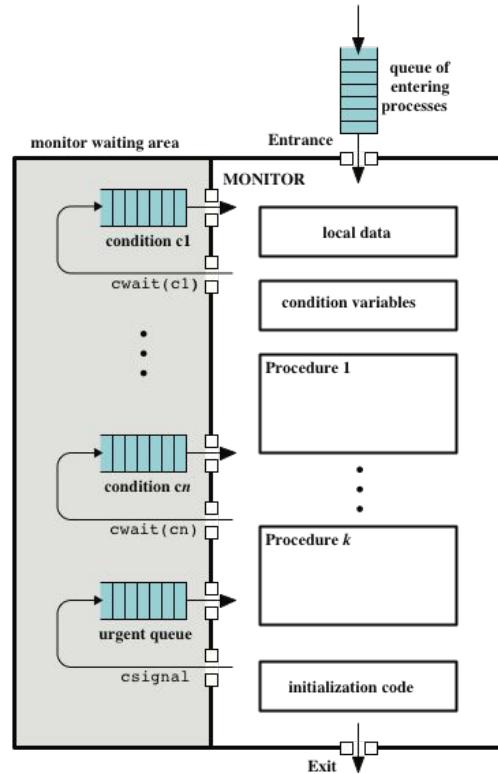
Monitors

- Programming language construct that provides equivalent functionality to that of semaphores and is easier to control
- Implemented in a number of programming languages
 - Concurrent Pascal, Pascal-Plus, Modula-2, Modula-3, Java
 - Has also been implemented as a program library
 - In particular, for something like a linked list, you may want to lock all linked lists with one lock, or have one lock for each list, or have one lock for each element of each list.
- A monitor is a software module consisting of one or more procedures, an initialization sequence, and local data.



Monitor Synchronization

- Local data variables are accessible only by the monitor's procedures and not by any external procedure
- Process enters monitor by invoking one of its procedures
- Only one process may be executing in the monitor at a time
- A monitor supports synchronization by the use of condition variables that are contained within the monitor and accessible only within the monitor
 - **Condition variables** are a special data type in monitors which are operated on by two functions:
 - **cwait(c)**: suspend execution of the calling process on condition c
 - **csignal(c)**: resume execution of some process blocked after a cwait on the same condition



Message Passing

- When processes interact with one another two fundamental requirements must be satisfied:

Synchronization

- To enforce mutual exclusion

Communication

- To exchange information

- Message passing is one approach to providing both of these functions
- Works with distributed systems and shared memory multiprocessor and uniprocessor systems

Message Passing

- The actual function is normally provided in the form of a pair of primitives:

send (destination, message)

receive (source, message)

- A process sends information in the form of a *message* to another process designated by a *destination*
- A process receives information by executing the receive primitive, indicating the *source* and the *message*

Synchronization

Communication of a message between two processes implies synchronization between the two

- The receiver cannot receive a message until it has been sent by another process

When a receive primitive is executed in a process there are two possibilities:

- If there is no waiting message the process is blocked until a message arrives or the process continues to execute, abandoning the attempt to receive
- If a message has previously been sent the message is received and execution continues

- **Blocking Send, Blocking Receive**
 - Both sender and receiver are blocked until the message is delivered
 - Sometimes referred to as a rendezvous
 - Allows for tight synchronization between processes
- **Nonblocking send, blocking receive**
 - Sender continues on but receiver is blocked until the requested message arrives
 - Most useful combination
 - Sends one or more messages to a variety of destinations as quickly as possible
 - Example -- a service process that exists to provide a service or resource to other processes
- **Nonblocking send, nonblocking receive**
 - Neither party is required to wait

Addressing in send and receive primitives

Direct Addressing

- Send primitive includes a specific identifier of the destination process
- Receive primitive can be handled in one of two ways:
 - Require that the process explicitly designate a sending process
 - Effective for cooperating concurrent processes
 - Implicit addressing
 - Source parameter of the receive primitive possesses a value returned when the receive operation has been performed

Indirect Addressing

- Messages are sent to a shared data structure consisting of queues that can temporarily hold messages
- Queues are referred to as mailboxes
- One process sends a message to the mailbox and the other process picks up the message from the mailbox
- Allows for greater flexibility in the use of messages

Indirect Process Communication

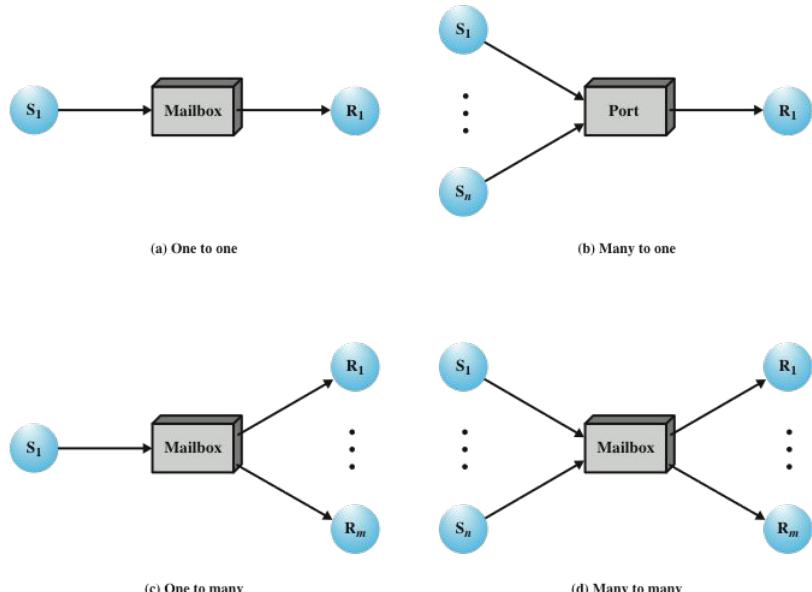


Figure 5.21 Indirect Process Communication

- A one-to-one relationship
 - allows a private communications link to be set up between two processes.
 - This insulates their interaction from erroneous interference from other processes.
- A many-to-one relationship is useful for client/server interaction
 - one process provides service to a number of other processes.
 - The mailbox is often referred to as a port
- A one-to-many relationship allows for one sender and multiple receivers
 - useful for applications where a message or some information is to be broadcast to a set of processes
- A many-to-many relationship
 - allows multiple server processes to provide concurrent service to multiple clients
- **Queueing Discipline**
 - Simplest queueing discipline is first-in-first-out
 - Message priority
 - Allow the receiver to inspect the message queue and select which message to receive next

Summary

- Mutual exclusion
 - Software approaches
 - Dekker's algorithm
- Principles of concurrency
 - Race condition
 - OS concerns
 - Process interaction
 - Requirements for mutual exclusion
- Mutual exclusion: hardware support
 - Interrupt disabling
 - Special machine instructions
- Semaphores
 - Mutual exclusion
 - Producer/consumer problem
 - Implementation of semaphores
- Message passing
 - Synchronization
 - Addressing
 - Message format
 - Queueing discipline
 - Mutual exclusion
- Readers/writers problem
 - Readers have priority
 - Writers have priority

Operating Systems

CS-C3140, Lecture 8

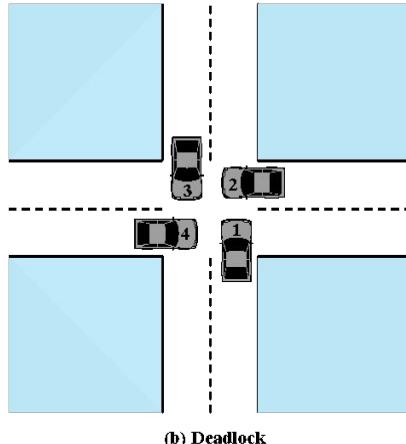
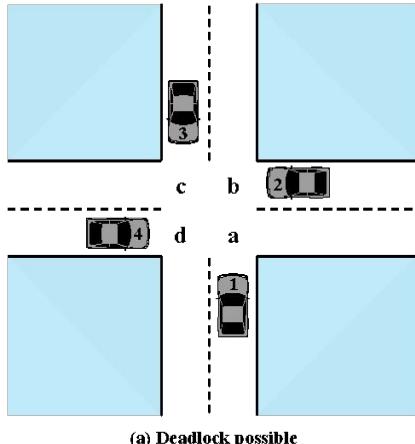
Deadlock and Starvation

Alexandru Paler

Announcements

- The exam will not include any questions from the C programming assignments
- In order to pass the assignments 50% of the total points of six (out of eight) assignments has to be passed
 - $8 \times 28 = 224$ points maximum
 - $6 \times 14 = 84$ points needed to pass
- There will be a bonus assignment
 - number nine
 - points are added to the regular ones

Deadlock and Everyday Examples



- The *permanent* blocking of a set of processes that either compete for system resources or communicate with each other
- A set of processes is deadlocked when each process in the set is blocked awaiting an event that can only be triggered by another blocked process in the set
- Permanent because none of the events is ever triggered
- No efficient solution in the general case

Joint Progress Diagrams

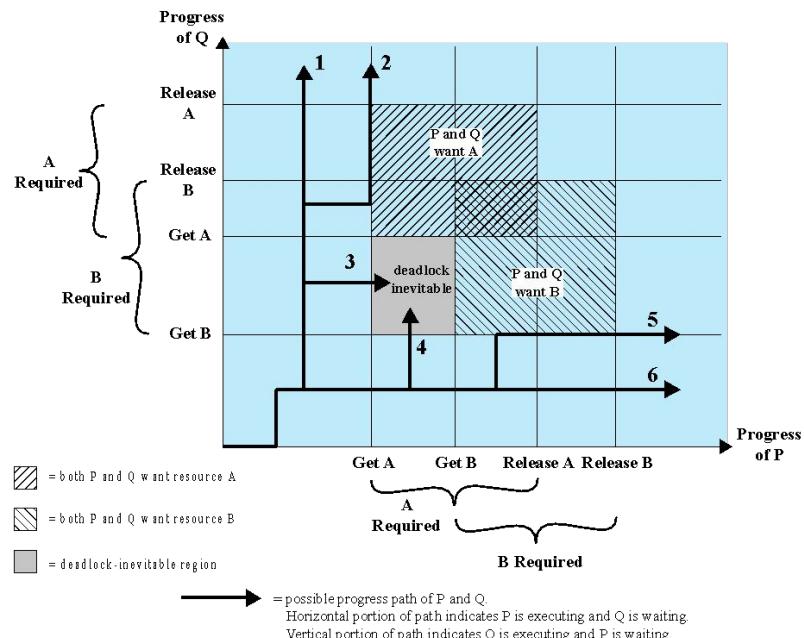


Figure 6.2 Example of Deadlock

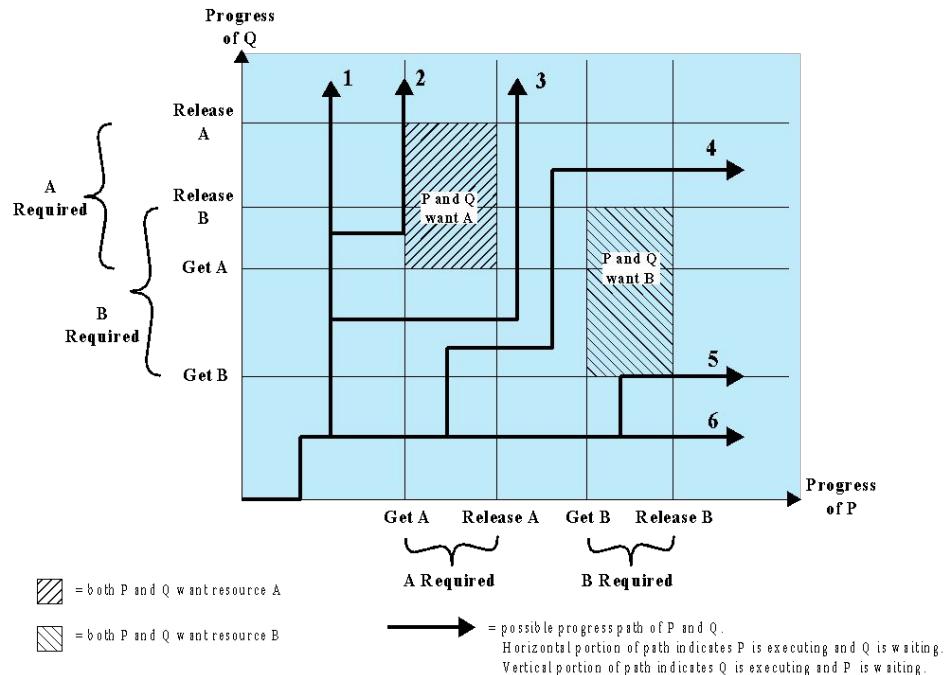


Figure 6.3 Example of No Deadlock

Resource Categories

Reusable

- Can be safely used by only one process at a time and is not depleted by that use
- Processors, I/O channels, main and secondary memory, devices, and data structures such as files, databases, and semaphores
- Example:
 - Space is available for allocation of 200Kbytes
 - Deadlock occurs if both processes progress to their second request

P1

...
Request 80 Kbytes
...
Request 60 Kbytes;

P2

...
Request 70 Kbytes
...
Request 80 Kbytes;

Consumable

- One that can be created (produced) and destroyed (consumed)
- Interrupts, signals, messages, and information, I/O buffers
- Example:
 - Deadlock occurs if the Receive is blocking (i.e., the receiving process is blocked until the message is received).

P1

...
Receive (P2);
...
Send (P2, M1);

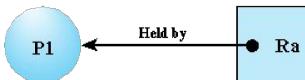
P2

...
Receive (P1);
...
Send (P1, M2);

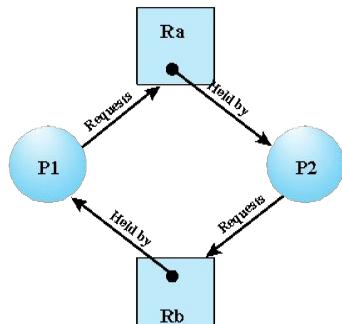
Resource Allocation Graphs



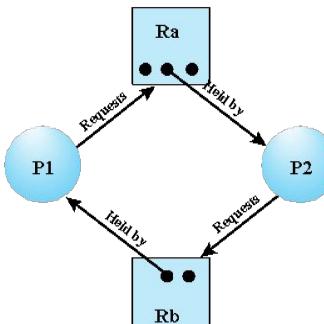
(a) Resource is requested



(b) Resource is held



(c) Circular wait



(d) No deadlock

A directed graph that depicts a state of the system of resources and processes, with each process and each resource represented by a node.

A graph edge directed from a process to a resource indicates a resource that has been requested by the process but not yet granted.

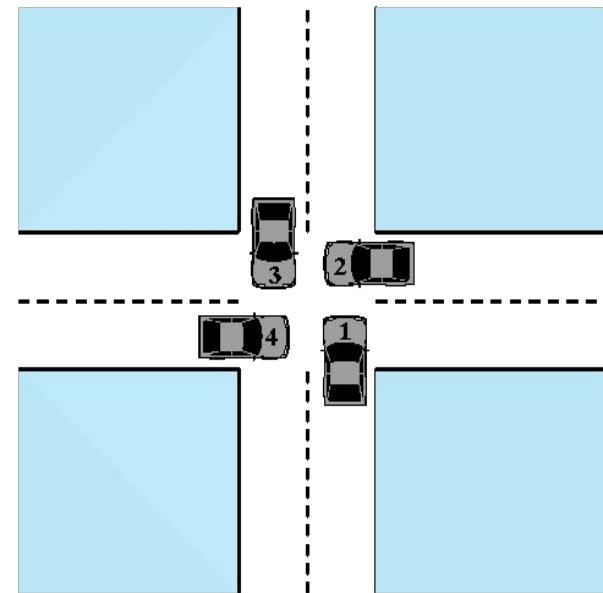
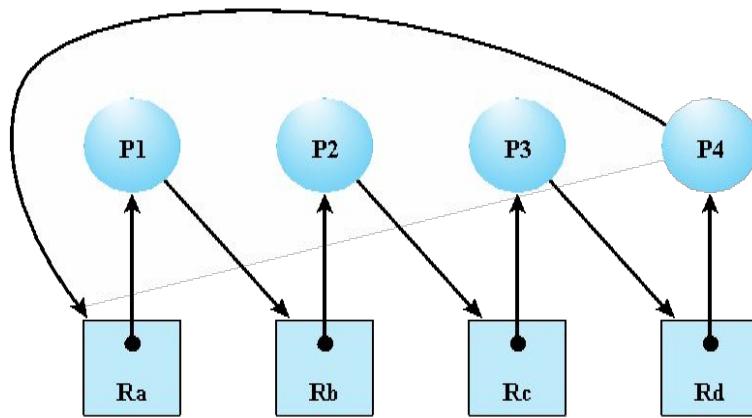
- Within a resource node, a dot is shown for each instance of that resource.
- A graph edge directed from a resource node dot to a process indicates a request that has been granted; that is, the **process has been assigned one unit** of that resource.

Figure 6.5c shows an example deadlock. There is only one unit each of resources Ra and Rb. Process P1 holds Rb and requests Ra, while P2 holds Ra but requests Rb.

Figure 6.5d has the same topology as Figure 6.5c , but there is no deadlock because multiple units of each resource are available.

Figure 6.5 Examples of Resource Allocation Graphs

Deadlock: Cycle in resource allocation graph



(b) Deadlock

Conditions for Deadlock

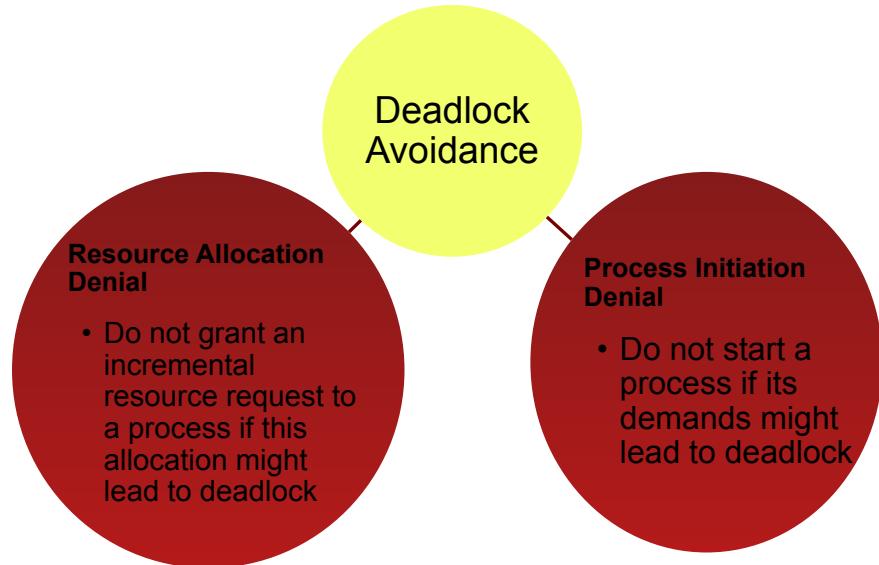
Mutual Exclusion	Hold-and-Wait	No Pre-emption	Circular Wait
<ul style="list-style-type: none">• Only one process may use a resource at a time• No process may access a resource until that has been allocated to another process	<ul style="list-style-type: none">• A process may hold allocated resources while awaiting assignment of other resources	<ul style="list-style-type: none">• No resource can be forcibly removed from a process holding it	<ul style="list-style-type: none">• A closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain

Three Approaches wrt Deadlocks

- There is no single effective strategy that can deal with all types of deadlock
- Deadlock avoidance
 - Do not grant a resource request if this allocation might lead to deadlock
- Deadlock prevention
 - Disallow one of the three necessary conditions for deadlock occurrence
 - Prevent circular wait condition from happening
- Deadlock detection
 - Grant resource requests when possible, but periodically check for the presence of deadlock and take action to recover
 - A check for deadlock can be made as frequently as each resource request or, less frequently, depending on how likely it is for a deadlock to occur
 - Advantage: It leads to early detection; The algorithm is relatively simple
 - Disadvantage: Frequent checks consume considerable processor time

Deadlock Avoidance

- Allows the three necessary conditions but makes judicious choices to assure that the deadlock point is never reached
- A decision is made dynamically whether the current resource allocation request will, if granted, potentially lead to a deadlock
- Allows the three necessary conditions but makes judicious choices to assure that the deadlock point is never reached
- Requires knowledge of future process requests
- Advantages
 - It is not necessary to preempt and rollback processes, as in deadlock detection
 - It is less restrictive than deadlock prevention



Deadlock Prevention Strategy

- Design a system in such a way that the possibility of deadlock is excluded
- Two main methods:
 - Indirect
 - Prevent the occurrence of one of the three necessary conditions: mutual exclusion, hold-and-wait, no pre-emption
 - Direct
 - Prevent the occurrence of a circular wait

Deadlock Condition Prevention

- Mutual exclusion
 - If access to a resource requires mutual exclusion, then mutual exclusion must be supported by the OS
 - Some resources, such as files, may allow multiple accesses for reads but only exclusive access for writes
 - Even in this case, deadlock can occur if more than one process requires write permission
- Hold and wait
 - Can be prevented by requiring that a process request all of its required resources at one time and blocking the process until all requests can be granted simultaneously
- No Preemption
 - If a process holding certain resources is denied a further request, that process must release its original resources and request them again
 - OS may preempt the second process and require it to release its resources
- Circular Wait
 - The circular wait condition can be prevented by defining a linear ordering of resource types

Resource Allocation Denial

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	1	0	0
P2	6	1	2
P3	2	1	1
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	2	2	2
P2	0	0	1
P3	1	0	3
P4	4	2	0

$C - A$

	R1	R2	R3
Resource vector R	9	3	6

	R1	R2	R3
Available vector V	0	1	1

(a) Initial state

- Referred to as the banker's algorithm
- **State** of the system reflects the current allocation of resources to processes
 - The state consists of the two vectors, Resource and Available, and the two matrices, Claim and Allocation
- **Safe state** is one in which there is at least one sequence of resource allocations to processes that does not result in a deadlock
- **Unsafe state** is a state that is not safe

The deadlock avoidance strategy

- does not predict deadlock with certainty
- it anticipates the possibility of deadlock
- assures that there is never such a possibility

Resource Allocation Denial: Determination of a Safe State

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix C

P1	R1	R2	R3
P2	6	1	2
P3	2	1	1
P4	0	0	2

Allocation matrix A

P1	R1	R2	R3
P2	0	0	1
P3	1	0	3
P4	4	2	0

$C - A$

	R1	R2	R3
	9	3	6

Resource vector R

	R1	R2	R3
	0	1	1

Available vector V

(a) Initial state

Four processes and three resources.

- The total amount of resources R1, R2, and R3 are 9, 3, and 6 units, respectively.
- In the current state allocations have been made to the four processes, leaving 1 unit of R2 and 1 unit of R3 available.

Is this a safe state?

- We ask an intermediate question
- Any of the four processes be run to completion with the resources available?**

Can the **difference between the maximum requirement and current allocation for any process be met with the available resources?** In terms of the matrices and vectors introduced earlier, the condition to be met for process i is:

$$C_{ij} - A_{ij} \leq V_j \text{ for all } j$$

Resource Allocation Denial: Determination of a Safe State

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	1	0	0
P2	6	1	2
P3	2	1	1
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	2	2	2
P2	0	0	1
P3	1	0	3
P4	4	2	0

C - A

	R1	R2	R3
	9	3	6

Resource vector R

	R1	R2	R3
	0	1	1

Available vector V

(a) Initial state

Not possible for P1

- has only 1 unit of R1
- requires 2 more units of R1, 2 units of R2, and 2 units of R3

By assigning one unit of R3 to process P2, P2 has its maximum required resources allocated and can run to completion.

When P2 completes, its resources can be returned to the pool of available resources(Figure 6.7b)

	R1	R2	R3
P1	3	2	2
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	1	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	2	2	2
P2	0	0	0
P3	1	0	3
P4	4	2	0

C - A

	R1	R2	R3
	9	3	6

Resource vector R

	R1	R2	R3
	6	2	3

Available vector V

(b) P2 runs to completion

Resource Allocation Denial: Determination of a Safe State

	R1	R2	R3
P1	3	2	2
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	1	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	2	2	2
P2	0	0	0
P3	1	0	3
P4	4	2	0

C - A

	R1	R2	R3
9	3	6	

Resource vector R

	R1	R2	R3
6	2	3	

Available vector V

(b) P2 runs to completion

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	1	0	3
P4	4	2	0

C - A

	R1	R2	R3
9	3	6	

Resource vector R

	R1	R2	R3
7	2	3	

Available vector V

(c) P1 runs to completion

Ask again if any of the remaining processes can be completed

- Each of the remaining processes could be completed.
- Suppose we choose P1, **allocate the required resources, complete P1, and return all of P1's resources to the available pool (Figure 6.7c)**

Resource Allocation Denial: Determination of a Safe State

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	1	0	3
P4	4	2	0

C - A

	R1	R2	R3
	9	3	6

Resource vector R

	R1	R2	R3
	7	2	3

Available vector V

(c) P1 runs to completion

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	0

C - A

	R1	R2	R3
	9	3	6

Resource vector R

	R1	R2	R3
	9	3	4

Available vector V

(d) P3 runs to completion

We can complete P3, resulting in the state of Figure 6.7d .

Finally, we can complete P4.

Conclusion: All of the processes have been run to completion. Thus, the state defined by Figure 6.7a is a safe state.

Deadlock avoidance strategy, which ensures that the system of processes and resources is always in a safe state:

1. When a process makes a request for a set of resources, assume that the request is granted, update the system state accordingly, and then determine if the result is a safe state.
2. If so, grant the request
3. If not, block the process until it is safe to grant the request

Resource Allocation Denial: Determ. of an Unsafe State

	R1	R2	R3	
P1	3	2	2	P1
P2	6	1	3	P2
P3	3	1	4	P3
P4	4	2	2	P4

	R1	R2	R3	
P1	1	0	0	P1
P2	5	1	1	P2
P3	2	1	1	P3
P4	0	0	2	P4

	R1	R2	R3	
P1	1	0	0	P1
P2	5	1	1	P2
P3	2	1	1	P3
P4	0	0	2	P4

	R1	R2	R3	
P1	1	2	2	P1
P2	1	0	2	P2
P3	1	0	3	P3
P4	4	2	0	P4

	R1	R2	R3	
	9	3	6	Resource vector R

	R1	R2	R3	
	1	1	2	Available vector V

(a) Initial state

	R1	R2	R3	
P1	3	2	2	P1
P2	6	1	3	P2
P3	3	1	4	P3
P4	4	2	2	P4

	R1	R2	R3	
P1	2	0	1	P1
P2	5	1	1	P2
P3	2	1	1	P3
P4	0	0	2	P4

	R1	R2	R3	
P1	1	2	1	P1
P2	1	0	2	P2
P3	1	0	3	P3
P4	4	2	0	P4

	R1	R2	R3	
	9	3	6	Resource vector R

	R1	R2	R3	
	0	1	1	Available vector V

(b) P1 requests one unit each of R1 and R3

Consider the state defined in Figure 6.8a

- suppose that P1 makes the request for one additional unit each of R1 and R3;
- Is this a safe state?

The answer is no, because **each process will need at least one additional unit of R1, and there are none available**. Thus, on the basis of deadlock avoidance, the request by P1 should be denied and P1 should be blocked.

It is important to point out that Figure 6.8b is not a deadlocked state. It merely has the potential for deadlock. It is possible, for example, that if P1 were run from this state it would subsequently release one unit of R1 and one unit of R3 prior to needing these resources again. If that happened, the system would return to a safe state. Thus, the deadlock avoidance strategy does not predict deadlock with certainty; it merely anticipates the possibility of deadlock and assures that there is never such a possibility.

Resource Allocation Denial: Determ. of an Unsafe State

	R1	R2	R3				
P1	3	2	2	P1	1	0	0
P2	6	1	3	P2	5	1	1
P3	3	1	4	P3	2	1	1
P4	4	2	2	P4	0	0	2

Claim matrix C Allocation matrix A

	R1	R2	R3	
	9	3	6	

Resource vector R

	R1	R2	R3	
		1	1	2

Available vector V

(a) Initial state

	R1	R2	R3				
P1	3	2	2	P1	2	0	1
P2	6	1	3	P2	5	1	1
P3	3	1	4	P3	2	1	1
P4	4	2	2	P4	0	0	2

Claim matrix C Allocation matrix A

	R1	R2	R3				
P1	1	2	1	P1	2	0	1
P2	1	0	2	P2	5	1	1
P3	1	0	3	P3	2	1	1
P4	4	2	0	P4	0	2	0

C - A

	R1	R2	R3	
	9	3	6	

	R1	R2	R3	
	0	1	1	

(b) P1 requests one unit each of R1 and R3

Consider the state defined in Figure 6.8a

- suppose that P1 makes the request for one additional unit each of R1 and R3;
- Is this a safe state?
 - Is not a deadlocked state
 - It merely has the potential for deadlock.

The answer is no, because **each process will need at least one additional unit of R1, and there are none available.**

On the basis of deadlock avoidance, the request by P1 should be denied and P1 should be blocked.

The deadlock avoidance strategy does not predict deadlock with certainty; it merely anticipates the possibility of deadlock and assures that there is never such a possibility.

Deadlock Avoidance Restrictions and Time Complexity

- Maximum resource requirement for each process must be stated in advance
- Processes under consideration must be independent and with no synchronization requirements
- There must be a fixed number of resources to allocate
- No process may exit while holding resources

```
BOOLEAN function SAFESTATE is -- Determines if current state is safe
{ NOCHANGE : boolean;
  WORK : array[1..m] of INTEGER = AVAILABLE;
  FINISH : array[1..n] of boolean = [false, ..., false];
  I : integer;

repeat
  NOCHANGE = TRUE;
  for I = 1 to N do
    if ((not FINISH[i]) and
        NEEDi <= WORK) then {
      WORK = WORK + ALLOCATION_i;
      FINISH[i] = true;
      NOCHANGE = false;
    }
  until NOCHANGE;
  return (FINISH == (true, ..., true));
}
```

Analysis

- n Processes, m Resources
- for loop nested in a repeat loop
 - n^*m
- the repeat loop can run, in worst case, n times
 - n
- Consequently n^*n^*m , polynomial

Deadlock Strategies

Deadlock prevention strategies are very conservative

- Limit access to resources by imposing restrictions on processes

Deadlock detection strategies do the opposite

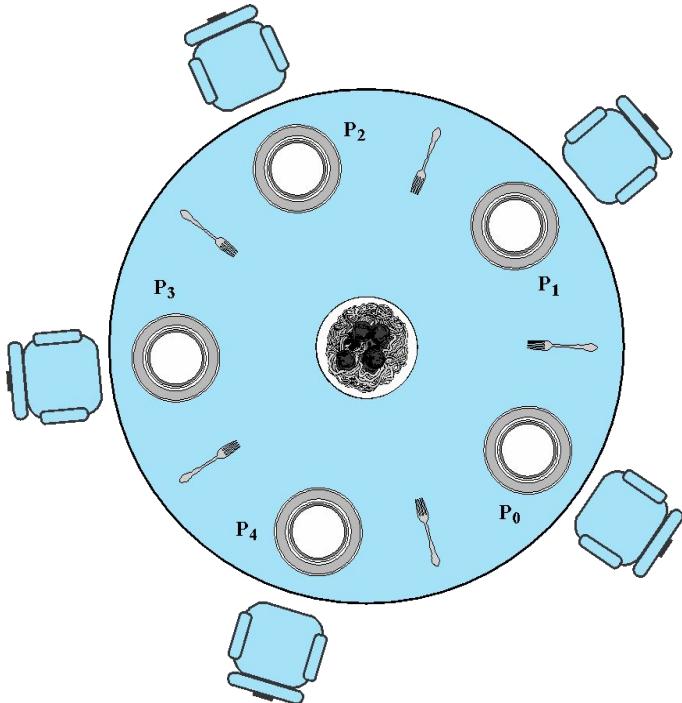
- Resource requests are granted whenever possible

Recovery Strategies

- Abort all deadlocked processes
- Back up each deadlocked process to some previously defined checkpoint and restart all processes
- Successively abort deadlocked processes until deadlock no longer exists
- Successively preempt resources until deadlock no longer exists

Dining Philosophers Problem

Dining Philosophers Problem



- Five philosophers live in a house, where a table is laid for them
- The life of each philosopher consists of thinking and eating
- Each philosopher requires two forks to eat spaghetti

A philosopher wishing to eat:

- goes to a place at the table
- using the two forks on either side of the plate, takes and eats some spaghetti

Devise a ritual (algorithm) that will allow the philosophers to eat. The algorithm must satisfy:

- mutual exclusion (no two philosophers can use the same fork at the same time)
- avoiding deadlock and starvation

The dining philosophers problem can be seen as representative of problems dealing with the coordination of shared resources. Problem illustrates basic problems in deadlock and starvation.

Attempts to develop solutions reveal many of the difficulties in concurrent programming.

First solution to the Dining Philosophers Problem

```
/* program      diningphilosophers */
semaphore fork [5] = {1};
int i;
void philosopher (int i)
{
    while (true) {
        think();
        wait (fork[i]);
        wait (fork [(i+1) mod 5]);
        eat();
        signal(fork [(i+1) mod 5]);
        signal(fork[i]);
    }
}
void main()
{
    parbegin (philosopher (0), philosopher (1), philosopher
(2),
              philosopher (3), philosopher (4));
}
```

Each philosopher:

- picks up first the fork on the left
- picks then the fork on the right
- finished eating and places the two forks on the table

If all of the philosophers are hungry at the same time:

- they all sit down
- they all pick up the fork on their left
- they all reach out for the other fork, which is not there
- Deadlock - All philosophers starve

To overcome the risk of deadlock:

- buy five additional forks
- teach the philosophers to eat spaghetti with one fork
- add an attendant who only allows four philosophers at a time into the dining room (next slide)

Second solution: free of deadlock and starvation

```
/* program diningphilosophers */
semaphore fork[5] = {1};
semaphore room = {4};
int i;
void philosopher (int i)
{
    while (true) {
        think();
        wait (room);
        wait (fork[i]);
        wait (fork [(i+1) mod 5]);
        eat();
        signal (fork [(i+1) mod 5]);
        signal (fork[i]);
        signal (room);
    }
}
void main()
{
    parbegin (philosopher (0), philosopher (1), philosopher (2),
              philosopher (3), philosopher (4));
}
```

<https://colab.research.google.com/drive/1VY9klr7wyZoZxyBFkOB0QxquFt3x2CyC?usp=sharing>

Linux Concurrency Mechanisms

UNIX Concurrency Mechanisms

- UNIX provides a variety of mechanisms for interprocessor communication and synchronization including:

Pipes

Messages

Shared
memory

Semaphores

Signals

Pipes

- Circular buffers allowing two processes to communicate on the producer-consumer model
- First-in-first-out queue, written by one process and read by another
- Named (a special file similar to a pipe but with a name on the filesystem)
- Unnamed

<https://man7.org/linux/man-pages/man2/pipe.2.html>

Program source

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

int
main(int argc, char *argv[])
{
    int pipefd[2];
    pid_t cpid;
    char buf;

    if (argc != 2) {
        fprintf(stderr, "Usage: %s <string>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    if (pipe(pipefd) == -1) {
        perror("pipe");
        exit(EXIT_FAILURE);
    }

    cpid = fork();
    if (cpid == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    }

    if (cpid == 0) { /* Child reads from pipe */
        close(pipefd[1]);           /* Close unused write end */

        while (read(pipefd[0], &buf, 1) > 0)
            write(STDOUT_FILENO, &buf, 1);

        write(STDOUT_FILENO, "\n", 1);
        close(pipefd[0]);
        _exit(EXIT_SUCCESS);
    } else { /* Parent writes argv[1] to pipe */
        close(pipefd[0]);           /* Close unused read end */
        write(pipefd[1], argv[1], strlen(argv[1]));
        close(pipefd[1]);           /* Reader will see EOF */
        wait(NULL);                /* Wait for child */
        exit(EXIT_SUCCESS);
    }
}
```

Messages

- A block of bytes with an accompanying type
 - UNIX provides msgsnd and msgrcv system calls for processes to engage in message passing
 - The message sender specifies the type of message with each message sent, and this can be used as a selection criterion by the receiver.
- A process
 - will block when trying to send a message to a full queue.
 - will also block when trying to read from an empty queue.
 - will not block if it attempts to read a message of a certain type and fails because no message of that type is present

mq_overview(7) — Linux manual page

[NAME](#) | [DESCRIPTION](#) | [NOTES](#) | [BUGS](#) | [EXAMPLES](#) | [SEE ALSO](#) | [COLOPHON](#)

[MQ_OVERVIEW\(7\)](#)

[Linux Programmer's Manual](#)

[MQ_OVERVIEW\(7\)](#)

NAME [top](#)

`mq_overview` - overview of POSIX message queues

DESCRIPTION [top](#)

POSIX message queues allow processes to exchange data in the form of messages. This API is distinct from that provided by System V message queues (`msgget(2)`, `msgsnd(2)`, `msgrcv(2)`, etc.), but provides similar functionality.

Message queues are created and opened using `mq_open(3)`; this function returns a *message queue descriptor* (`mqd_t`), which is used to refer to the open message queue in later calls. Each message queue is identified by a name of the form `/somename`; that is, a null-terminated string of up to `NAME_MAX` (i.e., 255) characters consisting of an initial slash, followed by one or more characters, none of which are slashes. Two processes can operate on the same queue by passing the same name to `mq_open(3)`.

Messages are transferred to and from a queue using `mq_send(3)` and `mq_receive(3)`. When a process has finished using the queue, it closes it using `mq_close(3)`, and when the queue is no longer required, it can be deleted using `mq_unlink(3)`. Queue attributes can be retrieved and (in some cases) modified using `mq_getattr(3)` and `mq_setattr(3)`. A process can request asynchronous notification of the arrival of a message on a previously empty queue using `mq_notify(3)`.

Shared Memory

- Fastest form of interprocess communication
- Common block of virtual memory shared by multiple processes
- Permission is read-only or read-write for a process
- Mutual exclusion constraints are not part of the shared-memory facility but must be provided by the processes using the shared memory

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>

void* create_shared_memory(size_t size) {
    // Our memory buffer will be readable and writable:
    int protection = PROT_READ | PROT_WRITE;

    // The buffer will be shared (meaning other processes can access it), but
    // anonymous (meaning third-party processes cannot obtain an address for it),
    // so only this process and its children will be able to use it:
    int visibility = MAP_SHARED | MAP_ANONYMOUS;

    // The remaining parameters to 'mmap()' are not important for this use case,
    // but the manpage for 'mmap' explains their purpose.
    return mmap(NULL, size, protection, visibility, -1, 0);
}
```

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

int main() {
    char parent_message[] = "hello"; // parent process will write this message
    char child_message[] = "goodbye"; // child process will then write this one

    void* shmem = create_shared_memory(128);

    memcpy(shmem, parent_message, sizeof(parent_message));

    int pid = fork();

    if (pid == 0) {
        printf("Child read: %s\n", shmem);
        memcpy(shmem, child_message, sizeof(child_message));
        printf("Child wrote: %s\n", shmem);
    } else {
        printf("Parent read: %s\n", shmem);
        sleep(1);
        printf("After 1s, parent read: %s\n", shmem);
    }
}
```

Signals

- A software mechanism that informs a process of the occurrence of asynchronous events
- Similar to a hardware interrupt, but does not employ priorities
- A signal is delivered by updating a field in the process table for the process to which the signal is being sent
- A process may respond to a signal by:
 - Performing some default action
 - Executing a signal-handler function
 - Ignoring the signal

Synopsis

```
kill [-s signal|-p] [--] pid...
kill -l [signal]
```

Description

The command **kill** sends the specified signal to the specified process or process group. If no signal is specified, the TERM signal is sent. The TERM signal will kill processes which do not catch this signal. For other processes, it may be necessary to use the KILL (9) signal, since this signal cannot be caught.

Most modern shells have a builtin kill function, with a usage rather similar to that of the command described here. The '-a' and '-p' options, and the possibility to specify pids by command name is a local extension.

If sig is 0, then no signal is sent, but error checking is still performed.

Value	Name	Description
01	SIGHUP	Hang up; sent to process when kernel assumes that the user of that process is doing no useful work
02	SIGINT	Interrupt
03	SIGQUIT	Quit; sent by user to induce halting of process and production of core dump
04	SIGILL	Illegal instruction
05	SIGTRAP	Trace trap; triggers the execution of code for process tracing
06	SIGIOT	IOT instruction
07	SIGEMT	EMT instruction
08	SIGFPE	Floating-point exception
09	SIGKILL	Kill; terminate process
10	SIGBUS	Bus error
11	SIGSEGV	Segmentation violation; process attempts to access location outside its virtual address space
12	SIGSYS	Bad argument to system call
13	SIGPIPE	Write on a pipe that has no readers attached to it
14	SIGALRM	Alarm clock; issued when a process wishes to receive a signal after a period of time
15	SIGTERM	Software termination
16	SIGUSR1	User-defined signal 1
17	SIGUSR2	User-defined signal 2
18	SIGCHLD	Death of a child
19	SIGPWR	Power failure

Atomic Operations and Spinlocks

- Atomic operations execute without interruption and without interference
 - Simplest of the approaches to kernel synchronization
 - Two types
-
- Most common technique for protecting a critical section in Linux
 - Can only be acquired by one thread at a time
 - Any other thread will keep trying (spinning) until it can acquire the lock
 - Built on an integer location in memory that is checked by each thread before it enters its critical section
 - Effective in situations where the wait time for acquiring a lock is expected to be very short
 - Disadvantage: Locked-out threads continue to execute in a busy-waiting mode
 - `spin_lock(&lock)`
 - `/* critical section */`
 - `spin_unlock(&lock)`

Integer Operations

Operate on an integer variable

Typically used to implement counters

Bitmap Operations

Operate on one of a sequence of bits at an arbitrary memory location indicated by a pointer variable

Memory Barriers

1. The barriers relate to machine instructions, namely loads and stores. Thus the higher-level language instruction $a = b$ involves both a load (read) from location b and a store (write) to location a .

2. The rmb, wmb, and mb operations dictate the behavior of both the compiler and the processor.

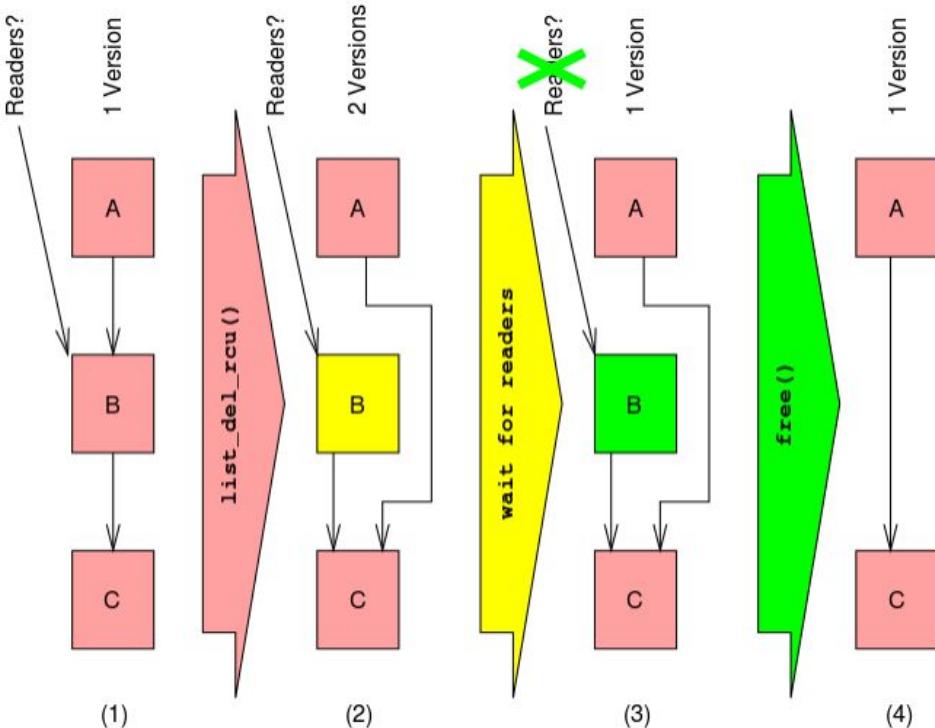
In the case of the compiler, the barrier operation dictates that the compiler not reorder instructions during the compile process.

In the case of the processor, the barrier operation dictates that any instructions pending in the pipeline before the barrier must be committed for execution before any instructions encountered after the barrier.

rmb()	Prevents loads from being reordered across the barrier
wmb()	Prevents stores from being reordered across the barrier
mb()	Prevents loads and stores from being reordered across the barrier
Barrier()	Prevents the compiler from reordering loads or stores across the barrier
smp_rmb()	On SMP, provides a rmb() and on UP provides a barrier()
smp_wmb()	On SMP, provides a wmb() and on UP provides a barrier()
smp_mb()	On SMP, provides a mb() and on UP provides a barrier()

Read-Copy-Update (RCU)

- Integrated into the Linux kernel in 2002
- The shared resources that the RCU mechanism protects must be accessed via a pointer
- The RCU mechanism provides access for multiple readers and writers to a shared resource
 - create a new structure,
 - copy the data from the old structure into the new one, and save a pointer to the old structure,
 - modify the new, copied, structure,
 - update the global pointer to refer to the new structure,
 - sleep until the operating system kernel determines that there are no readers left using the old structure, for example, in the Linux kernel, by using `synchronize_rcu()`,
 - once awakened by the kernel, deallocate the old structure.



Summary

- Principles of deadlock
 - Reusable/consumable resources
 - Resource allocation graphs
 - Conditions for deadlock
- Deadlock prevention
 - Mutual exclusion
 - Hold and wait
 - No preemption
 - Circular wait
- Deadlock avoidance
 - Process initiation denial
 - Resource allocation denial
- Deadlock detection
 - UNIX concurrency mechanisms
 - Pipes
 - Messages
 - Shared memory
 - Semaphores
 - Signals
 - Linux kernel concurrency mechanisms
 - Atomic operations
 - Spinlocks
 - Semaphores
 - Barriers

Operating Systems

CS-C3140, Lecture 9

Scheduling: Uniprocessor and Real-Time

Alexandru Paler

Processor Scheduling

- Assign processes to be executed by the processor in a way that meets system objectives:
 - response time
 - throughput
 - processor efficiency
- Broken down into three separate functions: long-, medium- and short-term scheduling

Long-term scheduling	The decision to add to the pool of processes to be executed
Medium-term scheduling	The decision to add to the number of processes that are partially or fully in main memory
Short-term scheduling	The decision as to which available process will be executed by the processor
I/O scheduling	The decision as to which process's pending I/O request shall be handled by an available I/O device

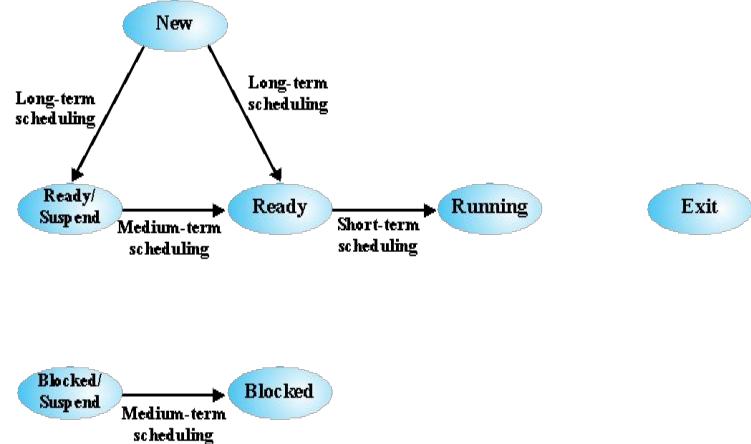
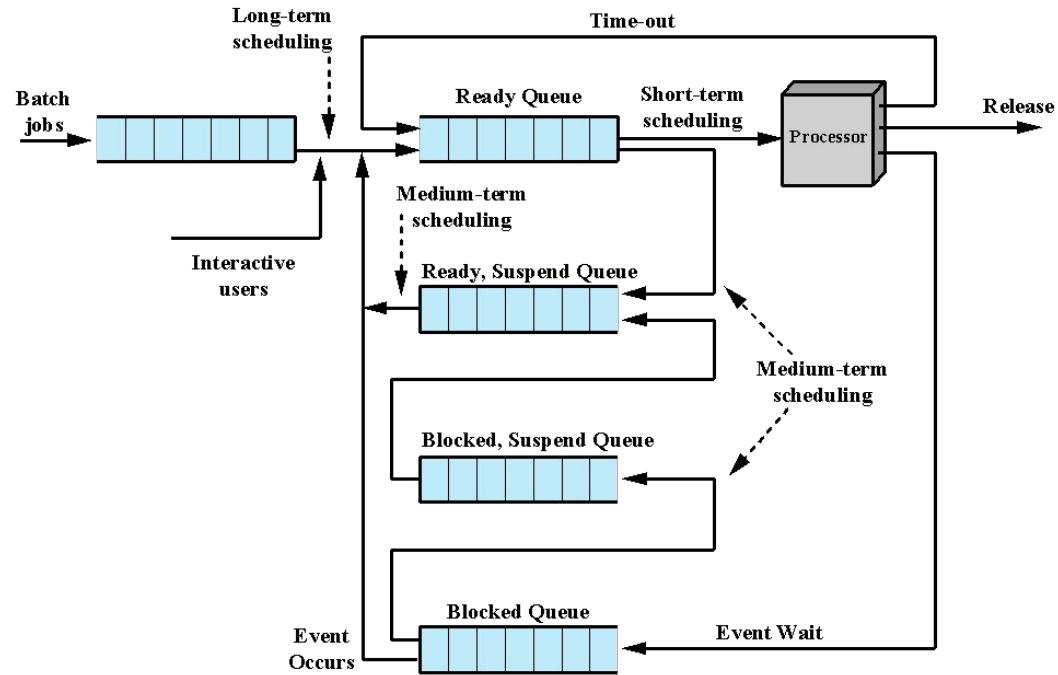
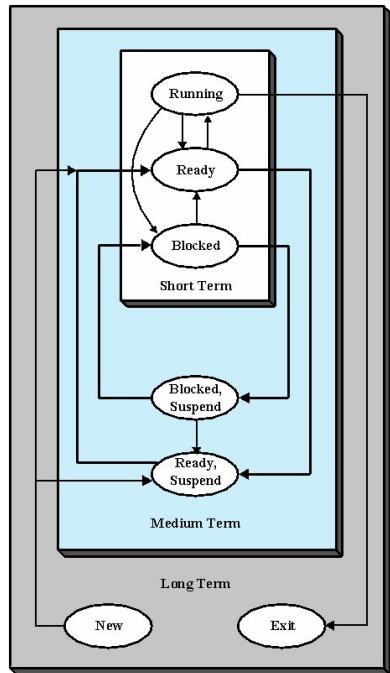
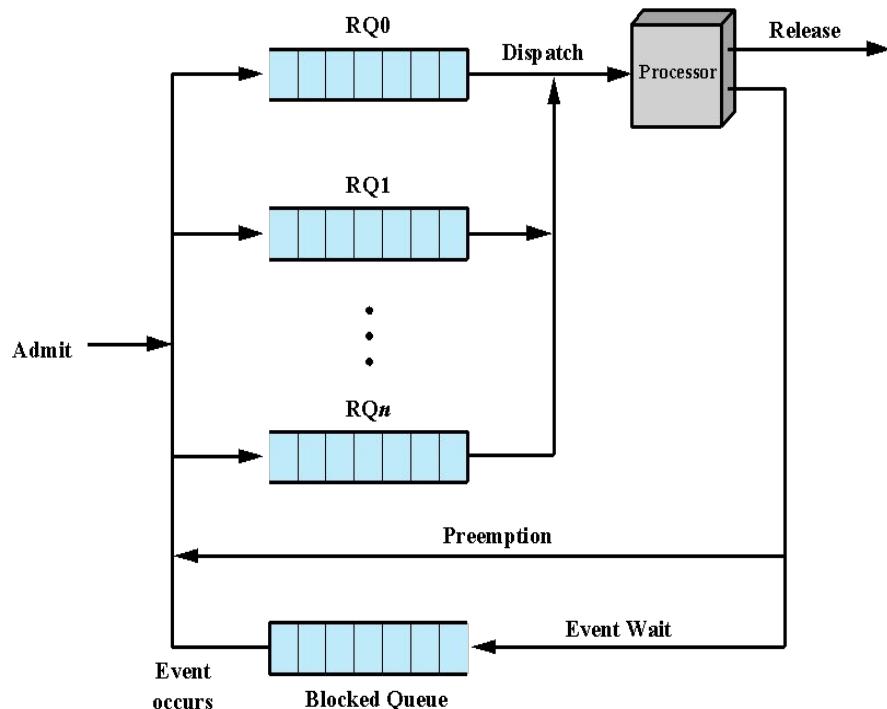


Figure 9.1 Scheduling and Process State Transitions

Scheduling : Managing queues to minimize queuing delay and to optimize performance in a queuing environment



Priority Queuing



- Each process is assigned a priority
 - A set of queues, in descending order of priority
 - RQ_0, RQ_1, \dots, RQ_n
 - $\text{priority}[RQ\ i] > \text{priority}[RQ\ j]$ for $i > j$
- The scheduler will always choose a process of higher priority over one of lower priority
 - When a scheduling selection is to be made, the scheduler will start at the highest-priority ready queue (RQ_0).
 - If there are one or more processes in the queue, a process is selected using some scheduling policy.
 - If RQ_0 is empty, then RQ_1 is examined, and so on.

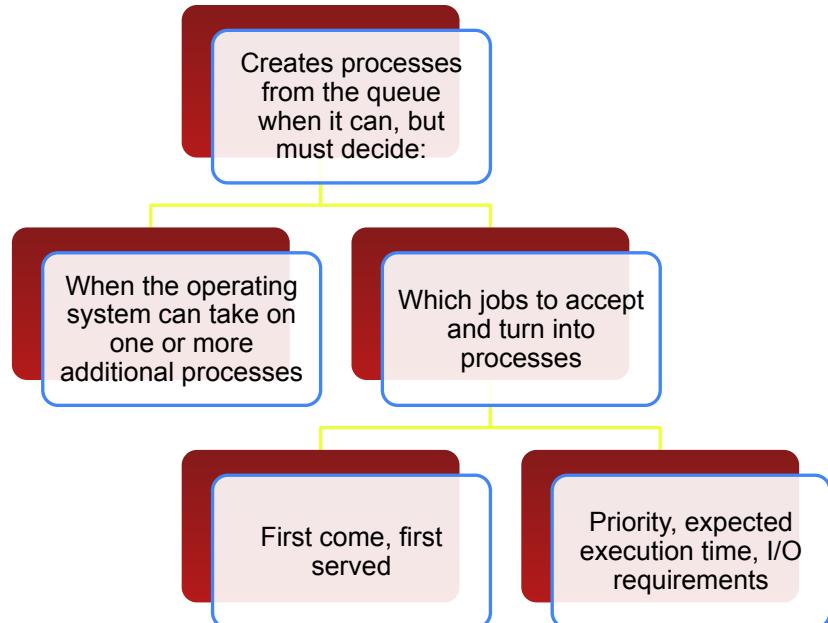
Disadvantages:

- lower-priority processes may suffer starvation
- Happens if there is always a steady supply of higher-priority ready processes
- If this behavior is not desirable, the priority of a process can change with its age or execution history.

For clarity, the queuing diagram is simplified, ignoring the existence of multiple blocked queues and of suspended states

Long-Term Scheduler

- Determines which programs are admitted to the system for processing
- Controls the degree of multiprogramming
 - The more processes that are created, the smaller the percentage of time that each process can be executed
 - May limit to provide satisfactory service to the current set of processes



Medium- and Short- Term Scheduling

- Medium-Term - Part of the swapping function
 - Makes swapping-in decisions
 - Considers the memory requirements of the swapped-out processes
- Short-Term - a.k.a. the dispatcher
 - Executes most frequently
 - Allocate processor time to optimize certain aspects of system behavior
 - Makes the fine-grained decision of which process to execute next
 - Invoked when an event occurs that may lead to the blocking of the current process or that may provide an opportunity to preempt a currently running process in favor of another
 - Clock interrupts
 - I/O interrupts
 - Operating system calls
 - Signals (e.g., semaphores)

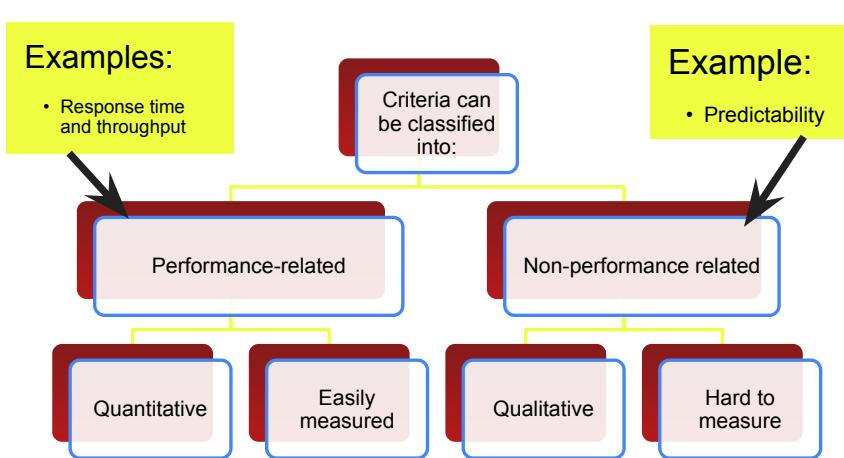
User-oriented criteria

- Relate to the behavior of the system as perceived by the individual user or process (such as response time in an interactive system)
- Important on virtually all systems

System-oriented criteria

- Focus is on **effective and efficient utilization** of the processor (rate at which processes are completed)
- Generally of minor importance on single-user systems

Performance and Scheduling Criteria



User Oriented, Performance Related
Turnaround time This is the interval of time between the submission of a process and its completion. Includes actual execution time plus time spent waiting for resources, including the processor. This is an appropriate measure for a batch job.
Response time For an interactive process, this is the time from the submission of a request until the response begins to be received. Often a process can begin producing some output to the user while continuing to process the request. Thus, this is a better measure than turnaround time from the user's point of view. The scheduling discipline should attempt to achieve low response time and to maximize the number of interactive users receiving acceptable response time.
Deadlines When process completion deadlines can be specified, the scheduling discipline should subordinate other goals to that of maximizing the percentage of deadlines met.
User Oriented, Other
Predictability A given job should run in about the same amount of time and at about the same cost regardless of the load on the system. A wide variation in response time or turnaround time is distracting to users. It may signal a wide swing in system workloads or the need for system tuning to cure instabilities.
System Oriented, Performance Related
Throughput The scheduling policy should attempt to maximize the number of processes completed per unit of time. This is a measure of how much work is being performed. This clearly depends on the average length of a process but is also influenced by the scheduling policy, which may affect utilization.
Processor utilization This is the percentage of time that the processor is busy. For an expensive shared system, this is a significant criterion. In single-user systems and in some other systems, such as real-time systems, this criterion is less important than some of the others.
System Oriented, Other
Fairness In the absence of guidance from the user or other system-supplied guidance, processes should be treated the same, and no process should suffer starvation.
Enforcing priorities When processes are assigned priorities, the scheduling policy should favor higher-priority processes.
Balancing resources The scheduling policy should keep the resources of the system busy. Processes that will underutilize stressed resources should be favored. This criterion also involves medium-term and long-term scheduling.

Selection Function and Decision Modes

- Determines which process, among ready processes, is selected next for execution
- May be based on priority, resource requirements, or the execution characteristics of the process
- If based on execution characteristics, then important quantities are:
 - w = time spent in system so far, waiting
 - e = time spent in execution so far
 - s = total service time required by the process, including e ; generally, this quantity must be estimated or supplied by the user

Mode: Specifies the instants in time at which the selection function is exercised

Nonpreemptive - once a process is in the Running state, it continues to execute until

- it terminates or
- it blocks itself to wait for I/O or to request some OS service.

Preemptive - Currently running process may be interrupted and moved to ready state by the OS

- Decision may be performed when
 - a new process arrives,
 - when an interrupt occurs that places a blocked process in the Ready state
 - periodically, based on a clock interrupt

Characteristics of Various Scheduling Policies

	FCFS	Round robin	SPN	SRT	HRRN	Feedback
Selection function	$\max[w]$	constant	$\min[s]$	$\min[s - e]$	$\max\left(\frac{W + S}{S}\right)$	(see text)
Decision mode	Non-preemptive	Preemptive (at time quantum)	Non-preemptive	Preemptive (at arrival)	Non-preemptive	Preemptive (at time quantum)
Through-Put	Not emphasized	May be low if quantum is too small	High	High	High	Not emphasized
Response time	May be high, especially if there is a large variance in process execution times	Provides good response time for short processes	Provides good response time for short processes	Provides good response time	Provides good response time	Not emphasized
Overhead	Minimum	Minimum	Can be high	Can be high	Can be high	Can be high
Effect on processes	Penalizes short processes; penalizes I/O bound processes	Fair treatment	Penalizes long processes	Penalizes long processes	Good balance	May favor I/O bound processes
Starvation	No	No	Possible	Possible	No	Possible

Number of processes completed per unit of time

Time from submission until first response

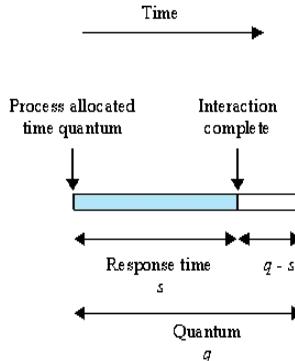
FCFS - First Come First Served; SPN - Shortest Process Next; SRT - Shortest Remaining Time; HRRN - Highest Response Ratio Next
 w = time spent in system so far, waiting; e = time spent in execution so far; s = total service time required by the process, including e ;

First-Come-First-Served (FCFS)

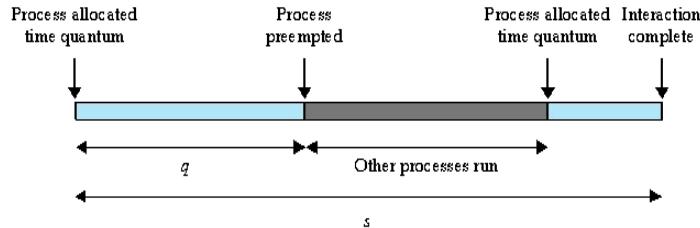
- Simplest scheduling policy
- Also known as first-in-first-out (FIFO) or a strict queuing scheme
- As each process becomes ready, it joins the ready queue
- When the currently running process ceases to execute, the process that has been in the ready queue the longest is selected for running
- Performs much better for long processes than short ones
- Tends to favor processor-bound processes over I/O-bound processes

Round Robin

- Time slicing
 - each process is given a slice of time before being preempted
 - Uses preemption based on a clock
 - Effective in a general-purpose time-sharing system or transaction processing system
- Principal design issue is the length of the time quantum, or slice
 - Time quantum should be slightly greater than the time required for a typical interaction or process function
 - if it is less, then most processes will require at least two time quanta
 - if it is longer than the longest running process, round robin degenerates to FCFS
- One drawback is its relative treatment of processor-bound and I/O-bound processes



(a) Time quantum greater than typical interaction



(b) Time quantum less than typical interaction

Shortest Process Next (SPN)

- Nonpreemptive policy
- The process with the shortest expected processing time is selected next
- A short process will jump to the head of the queue
- Disadvantages
 - Possibility of starvation for longer processes
 - estimate the required processing time
 - if the estimate is substantially under the actual running time, the system may abort the job

$$S_{n+1} = \frac{1}{n} \sum_{i=1}^n T_i \quad (9.1)$$

where

T_i = processor execution time for the i th instance of this process (total execution time for batch job; processor burst time for interactive job)

S_i = predicted value for the i th instance

S_1 = predicted value for first instance; not calculated

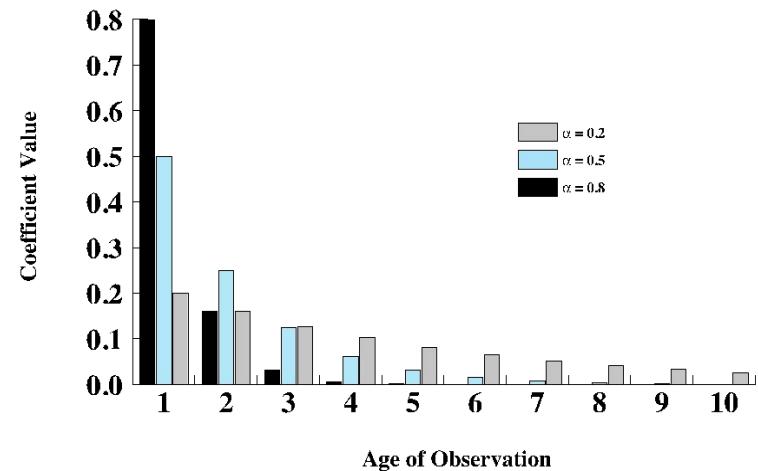


Figure 9.8 Exponential Smoothing Coefficients

$$S_{n+1} = \alpha T_n + (1 - \alpha)S_n \quad (9.3)$$

$$S_{n+1} = \alpha T_n + (1 - \alpha)\alpha T_{n-1} + \dots + (1 - \alpha)^i \alpha T_{n-i} + \dots + (1 - \alpha)^n S_1 \quad (9.4)$$

Exponential averaging: Reaction to the change in the observed values

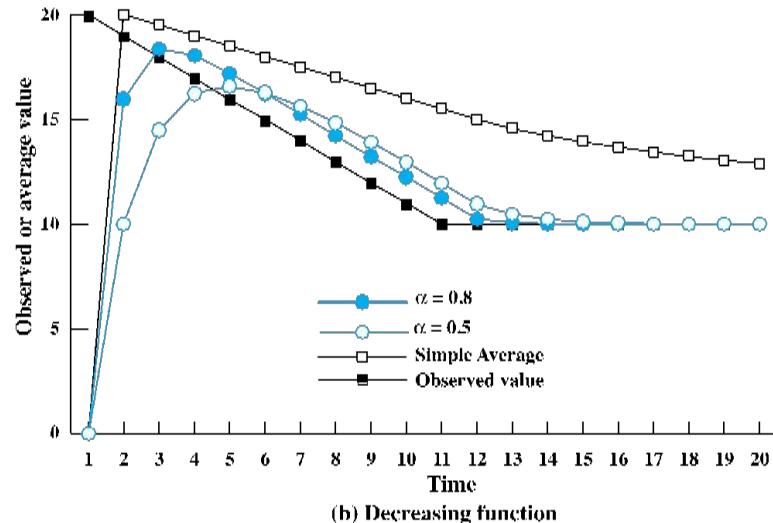
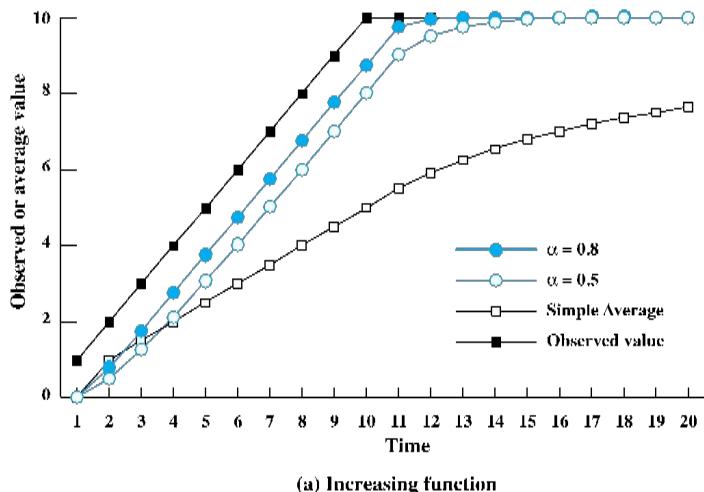


Figure 9.9 Use of Exponential Averaging

Shortest Remaining Time (SRT) & Highest Response Ratio Next (HRRN)

- Preemptive version of SPN
- Scheduler always chooses the process that has the shortest expected remaining processing time
- Risk of starvation of longer processes
 - Should give superior turnaround time performance to SPN because a short job is given immediate preference to a running longer job
- Thus, our scheduling rule becomes the following: when the current process completes or is blocked, choose the ready process with the greatest value of R .
- This approach is attractive because it accounts for the age of the process. While shorter jobs are favored (a smaller denominator yields a larger ratio), aging without service increases the ratio so that a longer process will eventually get past competing shorter jobs.
-
- Chooses next process with the greatest ratio
- Attractive because it accounts for the age of the process
- While shorter jobs are favored, aging without service increases the ratio so that a longer process will eventually get past competing shorter jobs
- $$Ratio = \frac{\text{time spent waiting} + \text{expected service time}}{\text{expected service time}}$$

Feedback Scheduling

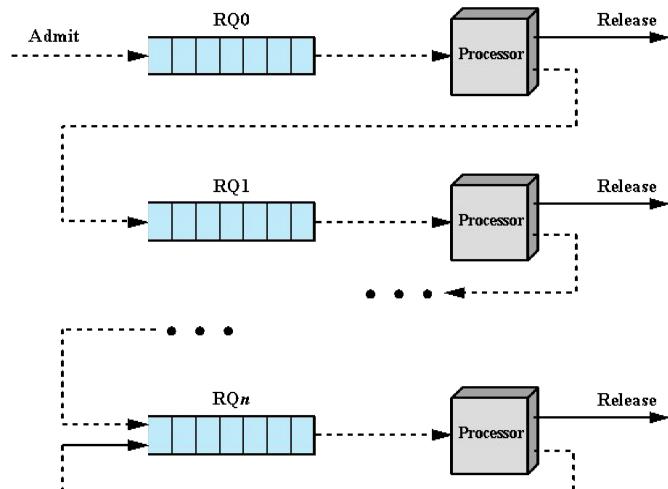


Figure 9.10 Feedback Scheduling

- Focus on the time spent in execution so far.
- This approach is known as multilevel feedback , meaning that the operating system allocates the processor to a process and, when the process blocks or is preempted, feeds it back into one of several priority queues.

Scheduling is done on a preemptive basis, and a dynamic priority mechanism is used

- When a process first enters the system, it is placed in **RQ0**.
- After its first preemption, when it returns to the Ready state, it is placed in **RQ1**
- Each subsequent time that it is preempted, it is demoted to the next lower-priority queue

Details:

- A short process will complete quickly, without migrating very far down the hierarchy of ready queues
- A longer process will gradually drift downward
- Newer, shorter processes are favored over older, longer processes
- Within each queue, except the lowest-priority queue, a simple FCFS mechanism is used.

Even with the allowance for greater time allocation at lower priority

- A longer process may still suffer starvation
- A possible remedy is to promote a process to a higher-priority queue after it spends a certain amount of time waiting for service in its current queue

Process Scheduling Example

Process	Arrival Time	Service Time
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

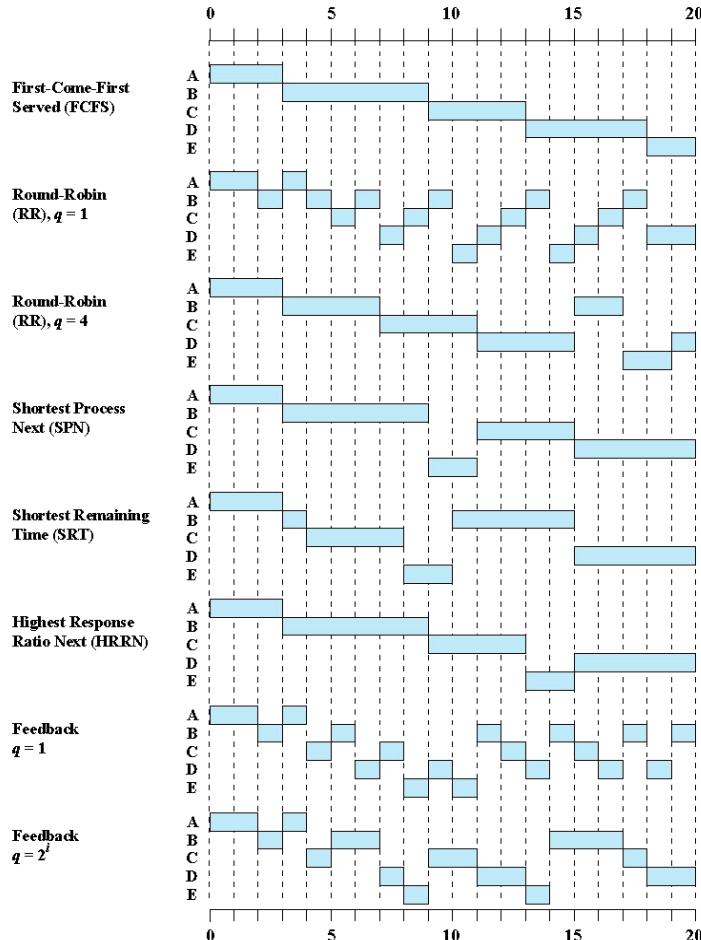


Figure 9.5 A Comparison of Scheduling Policies

Comparison of Scheduling Policies

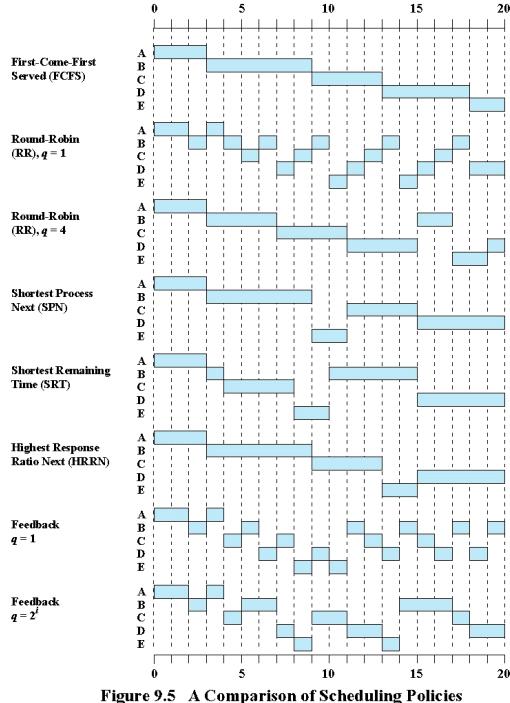


Figure 9.5 A Comparison of Scheduling Policies

- Turnaround time (TAT) is the residence time T_r , or total time that the item spends in the system (waiting time plus service time).
- Normalized turnaround time, which is the **ratio of turnaround time to service time**. (increasing values \rightarrow decreasing level of service)

Process	A	B	C	D	E	
Arrival Time	0	2	4	6	8	
Service Time (T_S)	3	6	4	5	2	Mean
						FCFS
Finish Time	3	9	13	18	20	
Turnaround Time (T_r)	3	7	9	12	12	8.60
T_r/T_S	1.00	1.17	2.25	2.40	6.00	2.56
						RR $q = 1$
Finish Time	4	18	17	20	15	
Turnaround Time (T_r)	4	16	13	14	7	10.80
T_r/T_S	1.33	2.67	3.25	2.80	3.50	2.71
						RR $q = 4$
Finish Time	3	17	11	20	19	
Turnaround Time (T_r)	3	15	7	14	11	10.00
T_r/T_S	1.00	2.5	1.75	2.80	5.50	2.71
						SPN
Finish Time	3	9	15	20	11	
Turnaround Time (T_r)	3	7	11	14	3	7.60
T_r/T_S	1.00	1.17	2.75	2.80	1.50	1.84
						SRT
Finish Time	3	15	8	20	10	
Turnaround Time (T_r)	3	13	4	14	2	7.20
T_r/T_S	1.00	2.17	1.00	2.80	1.00	1.59
						HRRN
Finish Time	3	9	13	20	15	
Turnaround Time (T_r)	3	7	9	14	7	8.00
T_r/T_S	1.00	1.17	2.25	2.80	3.5	2.14
						FB $q = 1$
Finish Time	4	20	16	19	11	
Turnaround Time (T_r)	4	18	12	13	3	10.00
T_r/T_S	1.33	3.00	3.00	2.60	1.5	2.29
						FB $q = 2t$
Finish Time	4	17	18	20	14	
Turnaround Time (T_r)	4	15	14	14	6	10.60
T_r/T_S	1.33	2.50	3.50	2.80	3.00	2.63

Traditional UNIX Scheduling

- Designed to provide good response time for interactive users while ensuring that low-priority background jobs do not starve
- Employs multilevel feedback using round robin within each of the priority queues
- Makes use of one-second preemption
- Priority
 - based on process type and execution history
 - recomputed once per second, at which time a new scheduling decision is made
- **base priority** divides all processes into fixed bands of priority levels.
- **CPU** and **nice** restrict a process from migrating out of its assigned band

$$CPU_j(i) = \frac{CPU_j(i - 1)}{2}$$

$$P_j(i) = Base_j + \frac{CPU_j(i)}{2} + nice_j$$

where

$CPU_j(i)$ = measure of processor utilization by process j through interval i

$P_j(i)$ = priority of process j at beginning of interval i ; lower values equal higher priorities

$Base_j$ = base priority of process j

$nice_j$ = user-controllable adjustment factor

Real-Time Systems

- The operating system, and in particular the scheduler, is perhaps the most important component
 - Control of laboratory experiments
 - Process control in industrial plants
 - Robotics
 - Air traffic control
 - Telecommunications
 - Military command and control systems
- Correctness of the system depends not only on the logical result of the computation but also on the time at which the results are produced
- Tasks or processes attempt to control or react to events that take place in the outside world
- These events occur in “real time” and tasks must be able to keep up with them

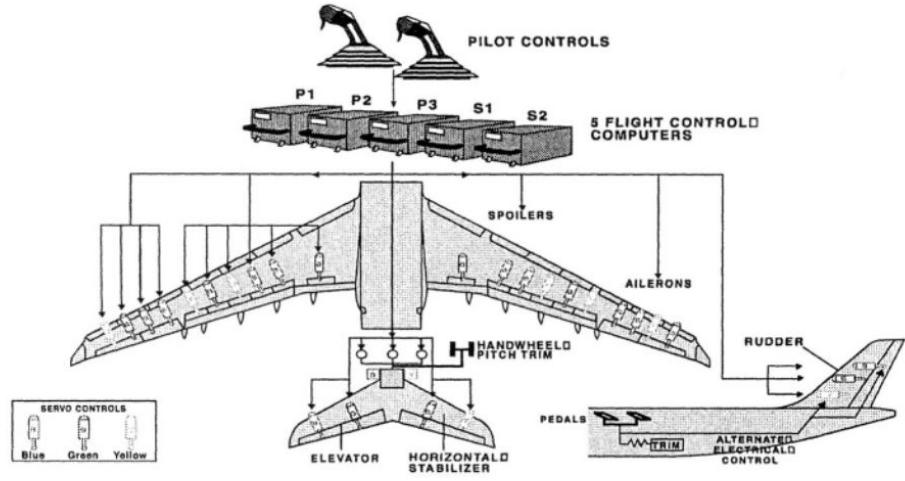


Figure 2: A340-600 system architecture

https://en.wikipedia.org/wiki/Comparison_of_real-time_operating_systems

Hard and Soft Real-Time Tasks

Hard real-time task

- One that must meet its deadline
- Otherwise it will cause unacceptable damage or a fatal error to the system

Soft real-time task

- Has an associated deadline that is desirable but not mandatory
- It still makes sense to schedule and complete the task even if it has passed its deadline

- Periodic tasks

- Requirement may be stated as:
 - Once per period T
 - Exactly T units apart

- Aperiodic tasks

- Has a deadline by which it must finish or start
- May have a constraint on both start and finish time

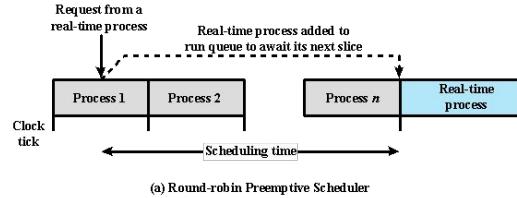
Characteristics of Real Time Systems (1)

- Determinism
 - Concerned with how long an operating system delays before acknowledging an interrupt
 - Operations are performed at fixed, predetermined times or within predetermined time intervals
 - When multiple processes are competing for resources and processor time, no system will be fully deterministic
- Responsiveness
 - Critical for real-time systems that must meet timing requirements imposed by individuals, devices, and data flows external to the system
 - Concerned with how long, after acknowledgment, it takes an operating system to service the interrupt
- User Control
 - It is essential to allow the user fine-grained control over task priority
 - User should be able to distinguish between hard and soft tasks and to specify relative priorities within each class
 - May allow user to specify such characteristics as:
 - Paging or process swapping
 - What processes must always be resident in main memory
 - What disk transfer algorithms are to be used
 - What rights the processes in various priority bands have

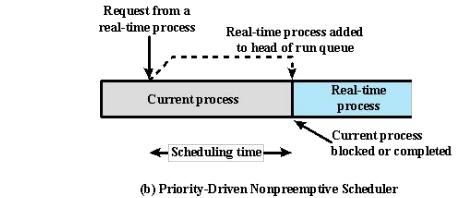
Characteristics of Real Time Systems (2)

- Reliability
 - Real-time systems respond to and control events in real time so loss or degradation of performance may have catastrophic consequences such as:
 - Financial loss
 - Major equipment damage
 - Loss of life
- Fail-Soft Operation
 - A characteristic that refers to the ability of a system to fail in such a way as to preserve as much capability and data as possible
 - Important aspect is stability
 - A real-time system is stable if the system will meet the deadlines of its most critical, highest-priority tasks even if some less critical task deadlines are not always met
- The following features are common to most real-time OSs
 - A stricter use of priorities than in an ordinary OS, with preemptive scheduling that is designed to meet real-time requirements
 - Interrupt latency is bounded and relatively short
 - More precise and predictable timing characteristics than general purpose OSs

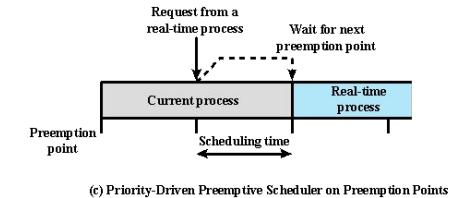
RT Scheduling Examples



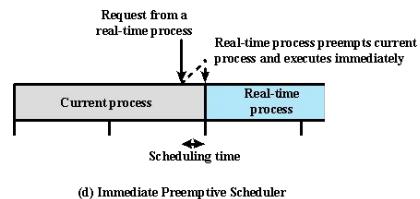
(a) Round-robin Preemptive Scheduler



(b) Priority-Driven Nonpreemptive Scheduler



(c) Priority-Driven Preemptive Scheduler on Preemption Points



(d) Immediate Preemptive Scheduler

A preemptive scheduler that uses simple round-robin scheduling. A real-time task is added to the ready queue to await its next time slice. Scheduling time will generally be unacceptable for real-time applications.

A nonpreemptive scheduler and priority scheduling. A real-time task that is ready would be scheduled as soon as the current process blocks or runs to completion. This could lead to a delay of several seconds if a low, low-priority task were executing at a critical time. Again, this approach is not acceptable.

Priorities with clock-based interrupts. Preemption points occur at regular intervals. When a preemption point occurs, the currently running task is preempted if a higher-priority task is waiting. This would include the preemption of tasks that are part of the operating system kernel. Such a delay may be on the order of several milliseconds.

Immediate preemption. Operating system responds to an interrupt almost immediately, unless the system is in a critical-code lockout section. Scheduling delays for a real-time task are very low.

Classes of Real-Time Scheduling Algorithms

Static table-driven approaches

- Performs a static analysis of feasible schedules of dispatching
- Result is a schedule that determines, at run time, when a task must begin execution

Static priority-driven preemptive approaches

- A static analysis is performed but no schedule is drawn up
- Analysis is used to assign priorities to tasks so that a traditional priority-driven preemptive scheduler can be used

Dynamic planning-based approaches

- Feasibility is determined at run time rather than offline prior to the start of execution
- One result of the analysis is a schedule or plan that is used to decide when to dispatch this task

Dynamic best effort approaches

- No feasibility analysis is performed
- System tries to meet all deadlines and aborts any started process whose deadline is missed

Static Scheduling Algorithms

Static table-driven scheduling

- Applicable to tasks that are periodic
- Input to the analysis consists of the periodic arrival time, execution time, periodic ending deadline, and relative priority of each task
- This is a predictable approach but one that is inflexible because any change to any task requirements requires that the schedule be redone
- Earliest-deadline-first (EDF) or other periodic deadline techniques are typical of this category

Static priority-driven preemptive scheduling

- Makes use of the priority-driven preemptive scheduling mechanism common to most non-real-time multiprogramming systems
- Priority assignment is related to the time constraints associated with each task
- Example: rate monotonic algorithm (RMS) which assigns static priorities to tasks based on the length of their periods

Dynamic Scheduling Algorithms

Dynamic planning-based scheduling

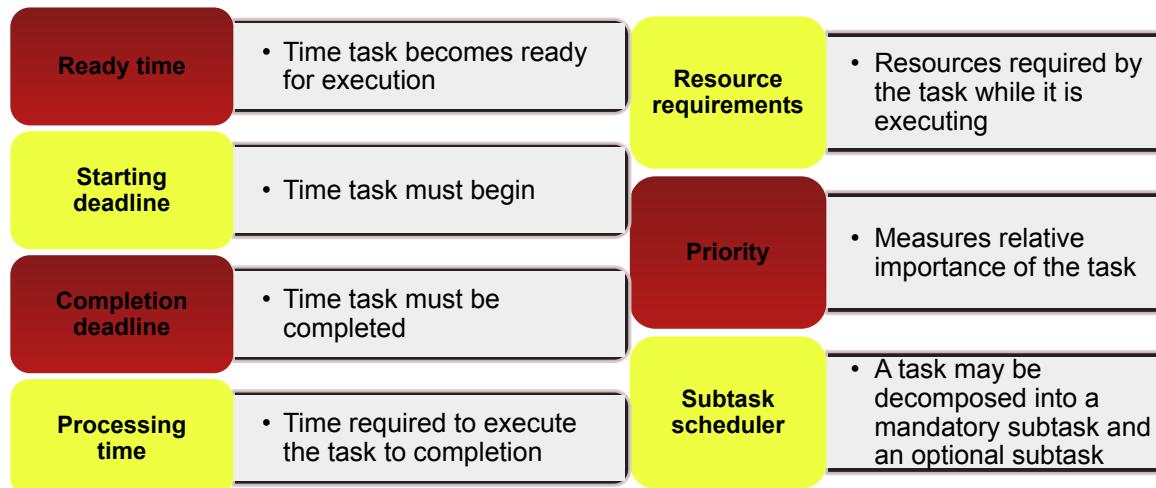
- After a task arrives, but before its execution begins, an attempt is made to create a schedule that contains the previously scheduled tasks as well as the new arrival
- If the new arrival can be scheduled in such a way that its deadlines are satisfied and that no currently scheduled task misses a deadline, then the schedule is revised to accommodate the new task

Dynamic best effort scheduling

- The approach used by many real-time systems that are currently commercially available
- **When a task arrives, the system assigns a priority based on the characteristics of the task**
- **Some form of deadline scheduling is typically used**
- Typically the tasks are aperiodic so no static scheduling analysis is possible
- The major disadvantage of this form of scheduling is, that until a deadline arrives or until the task completes, we do not know whether a timing constraint will be met
- Its advantage is that it is easy to implement

Deadline Scheduling

- Real-time operating systems
 - designed with the objective of starting real-time tasks as rapidly as possible and emphasize rapid interrupt handling and task dispatching
 - Real-time applications are generally not concerned with sheer speed but rather with completing (or starting) tasks at the most valuable times
- Priorities provide a crude tool and do not capture the requirement of completion (or initiation) at the most valuable time



Schedule Examples

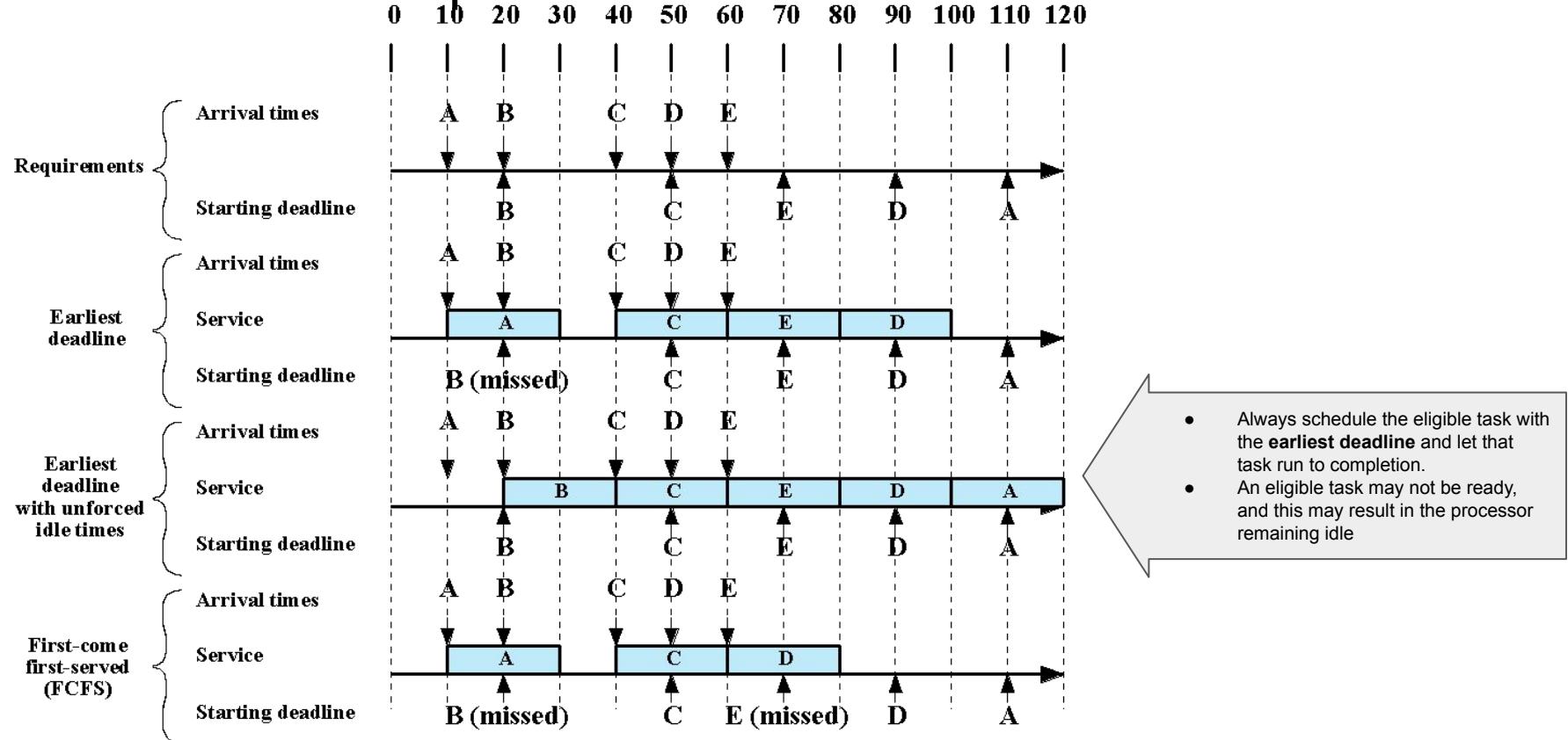
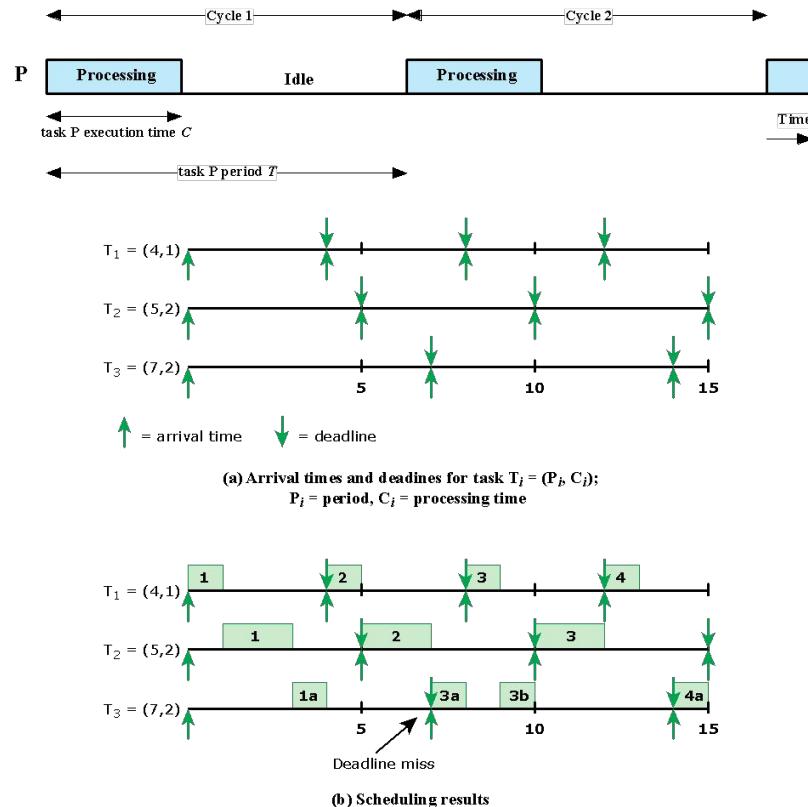


Figure 10.6 Scheduling of Aperiodic Real-time Tasks with Starting Deadlines

Rate Monotonic Scheduling



Assigns priorities to tasks on the basis of their periods

- the highest-priority task is the one with the shortest period
- the second highest-priority task is the one with the second shortest period, and so on.
- When more than one task is available for execution, the one with the shortest period is serviced first.

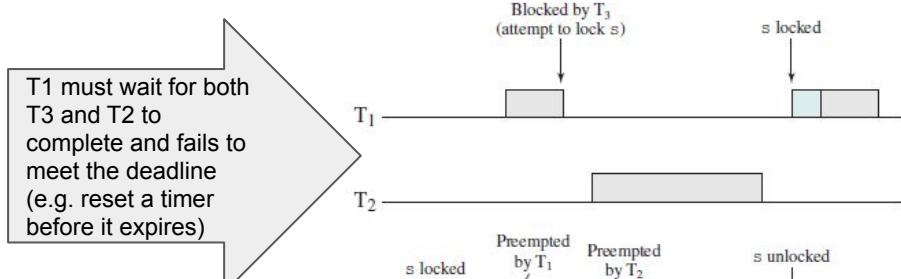
Plot the priority of tasks as a function of their rate, the result is a monotonically increasing function, hence the name, “rate monotonic scheduling.”

Example:

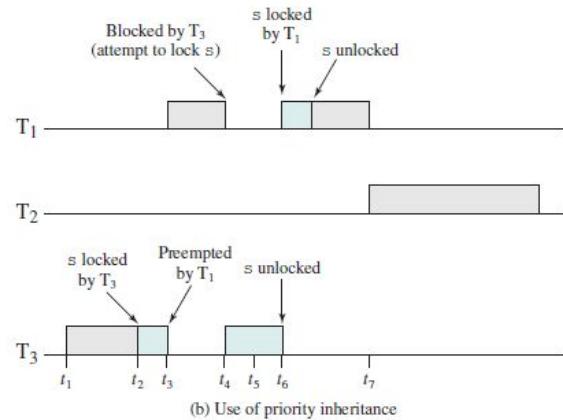
- Task instances are numbered sequentially within tasks.
- For task 3, the second instance is not executed because the deadline is missed. The third instance of task 3 experiences a preemption but is still able to complete before the deadline.

Priority Inversion

- Can occur in any priority-based preemptive scheduling scheme
- Occurs when circumstances within the system force a higher priority task to wait for a lower priority task
- **Unbounded Priority Inversion:** The duration of a priority inversion depends not only on the time required to handle a shared resource, but also on the unpredictable actions of other unrelated tasks
- **Priority inheritance**
 - a lower-priority task inherits the priority of any higher-priority task pending on a resource they share
 - change takes place as soon as the higher-priority task blocks on the resource; it should end when the resource is released by the lower-priority task.



(a) Unbounded priority inversion



Normal execution

Execution in critical section

Linux Scheduling

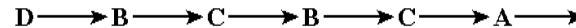
- The three primary Linux scheduling classes are:
 - SCHED_FIFO: First-in-first-out real-time threads
 - SCHED_RR: Round-robin real-time threads
 - SCHED_NORMAL: Other, non-real-time threads
- Within each class multiple priorities may be used, with priorities in the real-time classes higher than the priorities for the SCHED_NORMAL class

A	minimum
B	middle
C	middle
D	maximum

(a) Relative thread priorities



(b) Flow with FIFO scheduling



(c) Flow with RR scheduling

Figure 10.10 Example of Linux Real-Time Scheduling

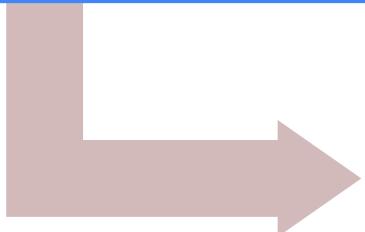
Non-Real-Time Scheduling

- The Linux 2.4 scheduler for the SCHED_OTHER class did not scale well with increasing number of processors and increasing number of processes
- Linux 2.6 uses a completely new priority scheduler known as the O(1) scheduler
 - designed so the time to select the appropriate process and assign it to a processor is constant regardless of the load on the system or number of processors
 - The O(1) scheduler proved to be unwieldy in the kernel because the amount of code is large and the algorithms are complex
- Completely Fair Scheduler (CFS)
 - Used as a result of the drawbacks of the O(1) scheduler
 - Models an ideal multitasking CPU on real hardware that provides fair access to all tasks
 - In order to achieve this goal, the CFS maintains a virtual runtime for each task
 - The virtual runtime is the amount of time spent executing so far, normalized by the number of runnable processes
 - The smaller a task's virtual runtime is, the higher is its need for the processor
 - Includes the concept of sleeper fairness to ensure that tasks that are not currently runnable receive a comparable share of the processor when they eventually need it

Red Black Tree

The CFS scheduler is based on using a Red Black tree, as opposed to other schedulers, which are typically based on run queues

- This scheme provides high efficiency in inserting, deleting, and searching tasks, due to its $O(\log N)$ complexity



A Red Black tree is a type of **self-balancing binary** search tree that obeys the following rules:

- A node is either red or black
- The root is black
- All leaves (NIL) are black
- If a node is red, then both its children are black
- Every path from a given node to any of its descendant NIL nodes contains the same number of black nodes

CFS Scheduler Red Black Tree

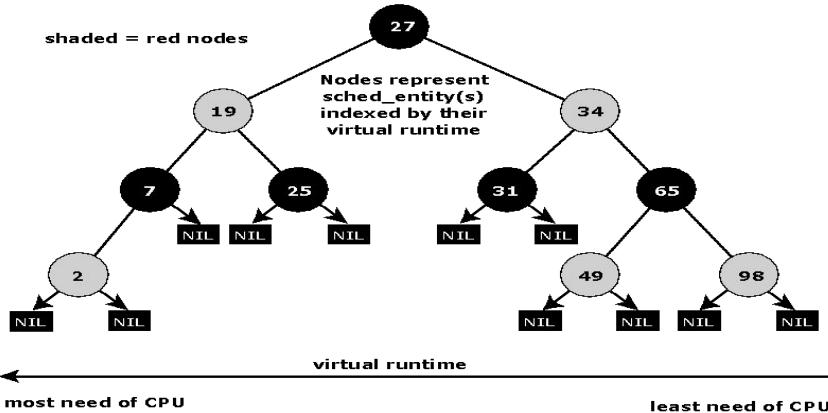


Figure 10.11 Example of Red Black Tree for CFS

The rb_node elements of the tree are embedded in *sched_entity* object.

The Red Black tree is ordered by *vruntime*

- the leftmost node of the tree represents the process that has the lowest *vruntime*, and that has the highest need for CPU
- this leftmost is the first one to be picked by the CPU
- when it runs, its *vruntime* is updated according to the time it consumed;
- when it is inserted back into the tree, very likely it will no longer be the one with the lowest *vruntime*,
- the tree will be rebalanced to reflect the current state

While an insert operation is being performed in RB tree, the leftmost child value is cached in *sched_entity* for fast lookup.

Operating Systems

CS-C3140, Lecture 10

Distributed Computing and Virtualization

Alexandru Paler

Announcements

- This is the last lecture. Next weeks no lecture.
- Next week the Q&A sessions are for asking questions from the lectures
- There will be a bonus assignment
- Exam will not include C programming exercises, but system calls and shell commands might be part of questions
-

Process Scheduling Example

Process	Arrival Time	Service Time
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

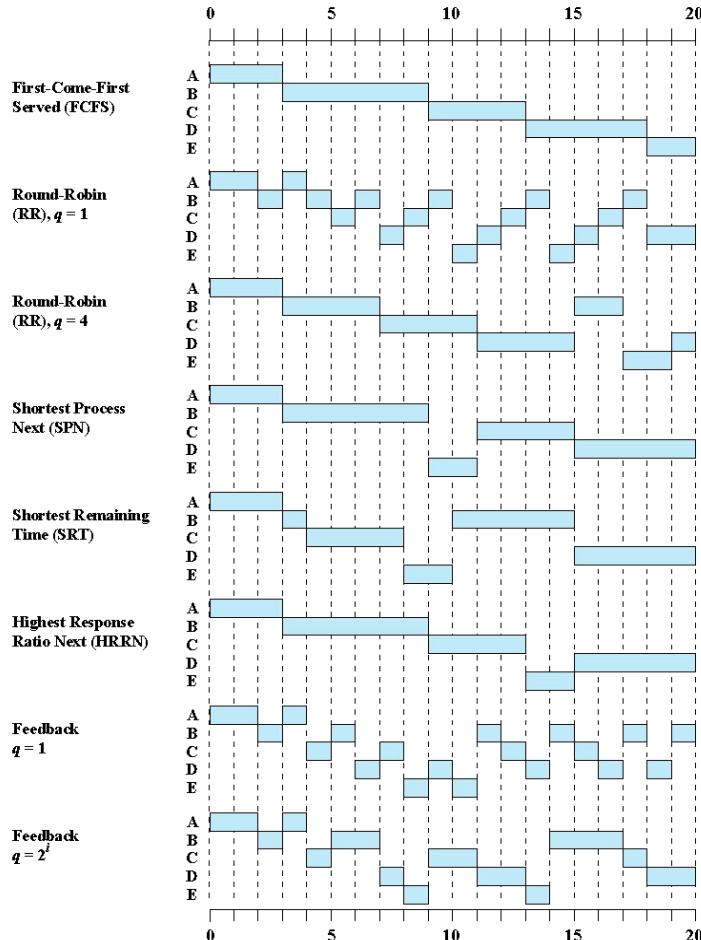
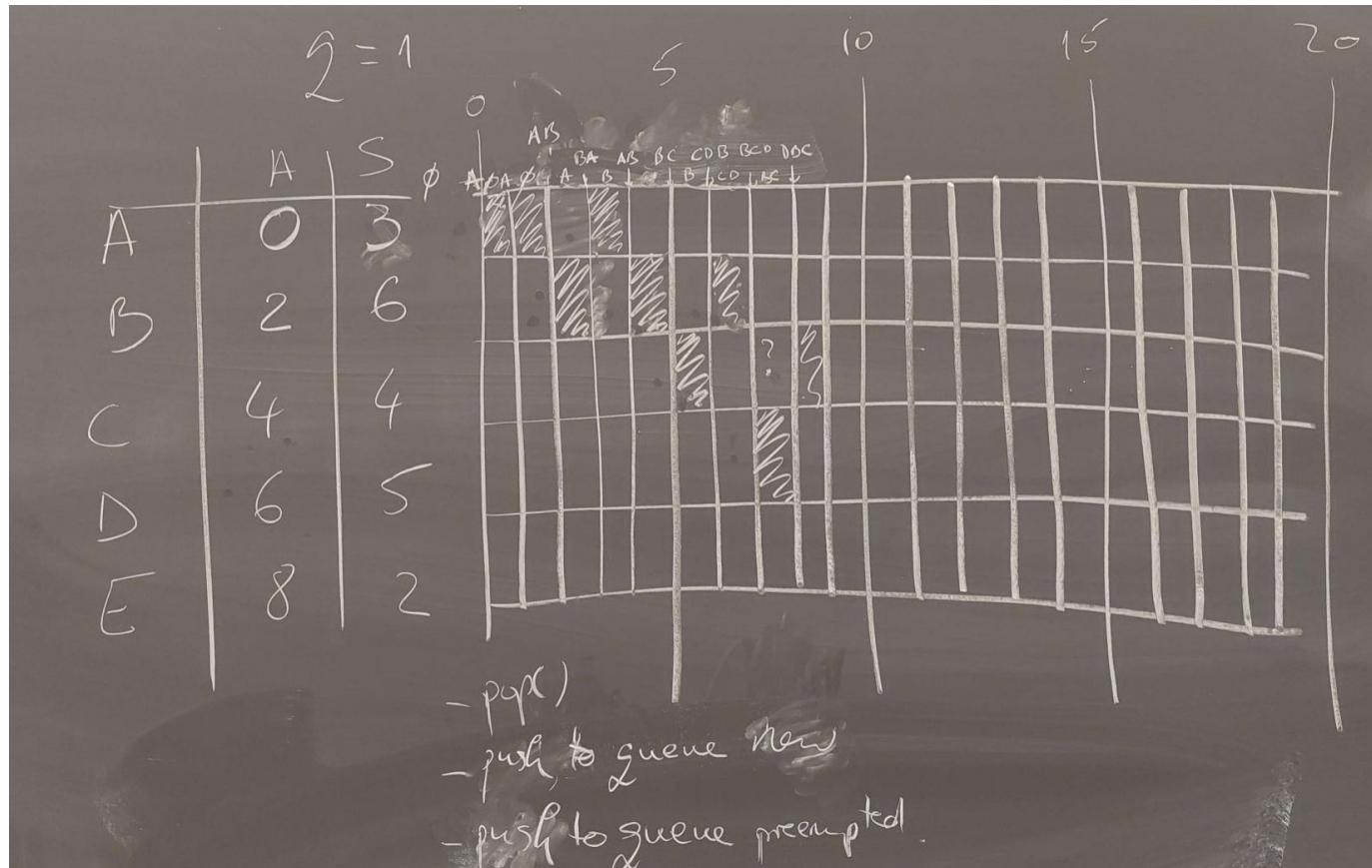


Figure 9.5 A Comparison of Scheduling Policies

Process Scheduling Example: Round Robin



Networks and Distributed Computing

What is the Internet?

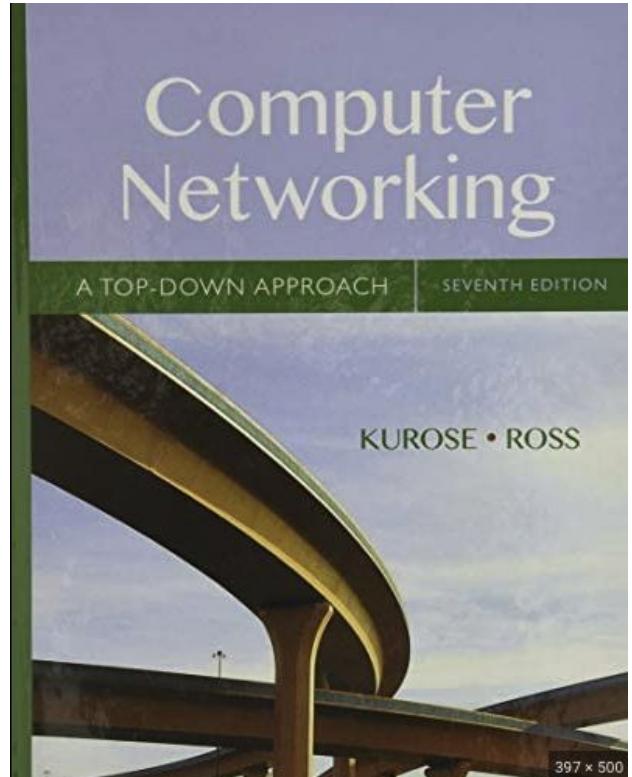
The Internet in a nutshell

- a global-scale communication infrastructure
- consisting of communication links and network elements
 - fiber optics, satellite radio, twisted-pair copper wire
 - routers, access points, switches
- senders and receivers exchange messages
 - route (or path): sequence of links / elements traversed by a message

The Internet as a black box

- the exact details on how messages traverse do not matter
- as long as the sender and the receiver can be identified

Source: James Kurose and Keith W. Ross, "Computer Networking: A Top-Down Approach", 6th edition, Pearson Education, 2013, Chapter 1



Networks applications and services

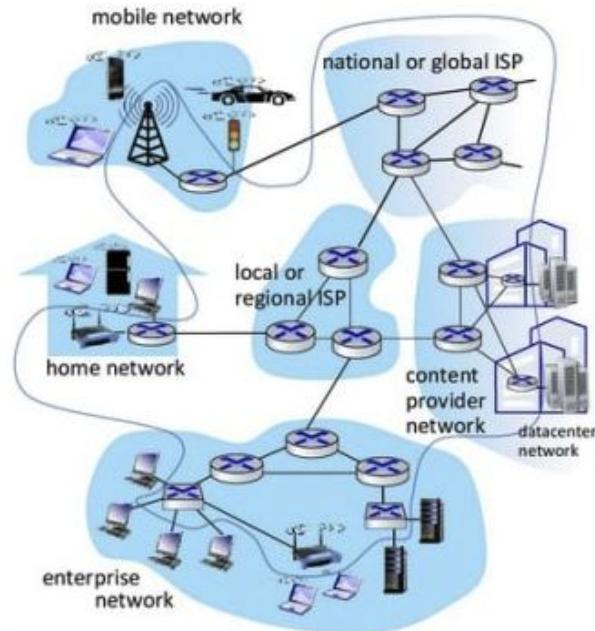
The Internet: a service view

- an infrastructure that provides services to applications
- applications are distributed
- interactions through Application Programming Interfaces (APIs)

Protocol

- specific messages and associated actions in response to them
- a more precise definition
 - a specification of the format and the order of messages exchanged between two or more communicating entities
 - the actions taken on the transmission and (or) receipt of a message or other event

Source: James Kurose and Keith W. Ross, "Computer Networking: A Top-Down Approach", 6th edition, Pearson Education, 2013, Chapter 1



Applications over the Internet

Network applications

- programs running on different systems
- communicating over the network

Client-server architecture

- client: resource-constrained device
 - desktop,laptop,personal/mobile device (e.g.,smartphone or tablet)
 - limited energy (i.e., battery) and computing capabilities
- server: very powerful hardware
 - machine with plenty of computing, bandwidth and storage resources
 - generally located in a data center
- collectively referred to as hosts or end systems

Source: James Kurose and Keith W. Ross, “Computer Networking: A Top-Down Approach”, 6th edition, Pearson Education, 2013, Chapter 2

Clients and servers

- operating system processes in a communication session
- client: initiates the communication
- server: waits to be contacted to start a session

Addressing processes

- an Internet host through an IP address (e.g., 130.233.192.1)
- or by a hostname
 - human-friendly name (e.g., www.aalto.fi)
 - Domain Name System (DNS): directory service translating hostnames to IP addresses
- the receiving process on that host through a port (e.g., 80)

Internet services for different applications

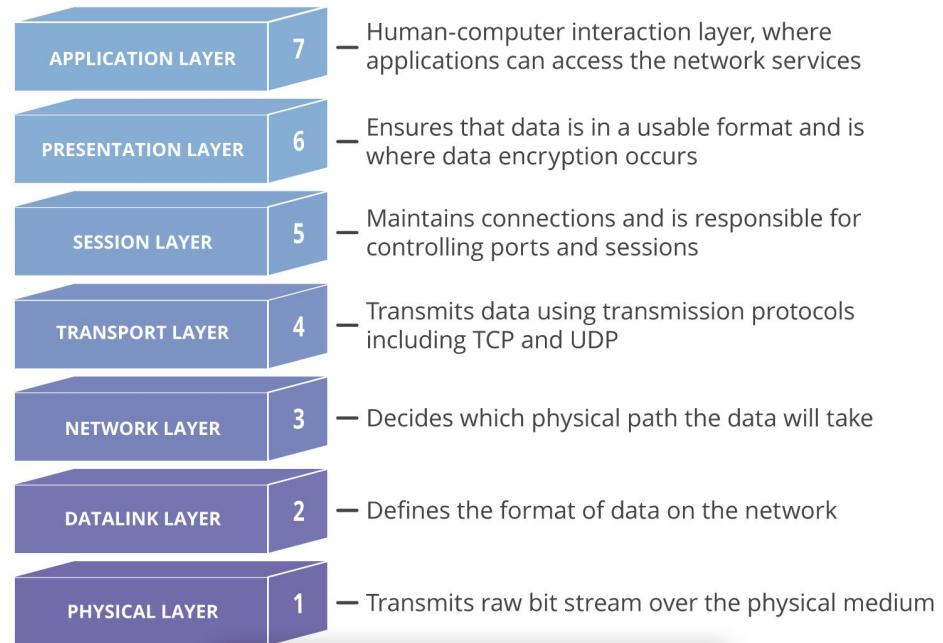
Transport-layer services

- logical communication between application processes running on different hosts
- abstraction from underlying physical infrastructure, as processes were on same machine
- realized by transport protocols

Major transport protocols

- Transmission Control Protocol (TCP)
- User Datagram Protocol (UDP)

Source: James Kurose and Keith W. Ross, “Computer Networking: A Top-Down Approach”, 6th edition, Pearson Education, 2013, Chapter 3, Section 3.1



<https://www.cloudflare.com/learning/ddos/glossary/open-systems-interconnection-model-osi/>

Transmission Control Protocol vs. User Datagram Protocol

TCP - Connection-oriented service

- client and servers need to exchange control information before sending and receiving actual application messages
- such handshaking process creates a TCP connection
- full-duplex channel: messages are sent / received over the connection at the same time

Reliable transport service

- recovers from errors and lost messages
- ensures correct message ordering and no duplicates
- adapts to available bandwidth between sender and receiver

Vinton Cerf and
Robert Kahn,
Turing prize 2004



UDP - Connectionless service

- very lightweight solution
- no handshaking

Unreliable transport service

- no guarantee that messages will be correctly delivered, if at all
 - errors and message loss are not handled
 - there could be duplicate messages
 - the order of messages may not be preserved
- does not consider network congestion
- application could still implement needed features on top of UDP
 - Domain Name System (DNS)
 - Streaming media applications such as movies
 - Online multiplayer games
 - Voice over IP (VoIP)
 - Trivial File Transfer Protocol (TFTP)

(Indirect) Communication between processes - lecture 7

Shared resources and synchronization

- locks, semaphores and monitors
- shared address space between processes
- communication through shared data

Message passing and data transfer

- use two communication primitives: send and receive
- possibly work over multiple machines, even with different operating systems
- implicit synchronization, no need for mutual exclusion

Pipe

- an output stream connected to an input stream through an in-memory queue
- uni-directional communication
- send is non-blocking, receive is blocking
- the operating system takes care of buffering

Mailbox

- allow many-to-one or one-to-many communication
- still uni-directional

Pipes in UNIX

Anonymous pipes

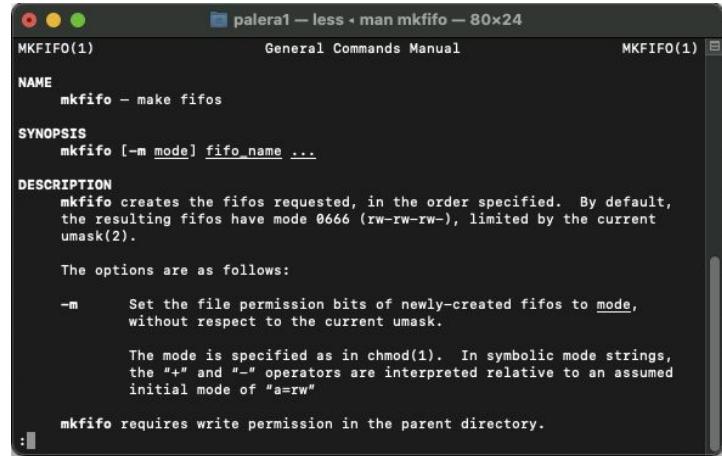
- output of one process becomes input of another process
- indicated with a vertical bar (i.e., |) in the command line
 - example:

```
echo "Operating systems" | sed "s/systems/machines/g"
```

Named pipes

- entities in a filesystem, subject to related permissions
- also called FIFOs as they use the First In First Out policy
- created with the mkfifo command / function

Sources: Computerphile, “Unix Pipeline (Brian Kernighan)”
and David A. Rusling, “The Linux Kernel”, 1999, Section 5.2



The screenshot shows a terminal window titled 'palera1 — less < man mkfifo — 80x24'. The window displays the 'mkfifo(1)' man page from the 'General Commands Manual'. The page includes sections for NAME, SYNOPSIS, DESCRIPTION, and options. The SYNOPSIS section shows the command: 'mkfifo [-m mode] fifo_name ...'. The DESCRIPTION section explains that mkfifo creates FIFOs with mode 0666 by default, limited by the current umask(2). It also describes the options: '-m' to set file permission bits and the symbolic mode strings for chmod(1). The terminal window has a dark background with light-colored text.

Sockets

Definition

- bi-directional mechanism
- for communication both locally and over a network
- same application programming interface

UNIX domain sockets

- similar to Internet sockets but limited to the same machine
- both named and unnamed
- generally preferred to named pipes to implement inter-process communication for important services

Source: David A. Wheeler, “Secure Programming HOWTO”, Version 3.72
(September 19, 2015) □ Section 3.4

Network sockets

Features

- transport services for different application requirements
- client-server paradigm

Socket API

- creation / deletion

```
sid = socket(AF_INET, SOCK_STREAM, 0); se = close(sid);
```

- connection setup

```
se = bind(sid, localaddr, addrlen);
listen(sid, length);
accept(sid, addr, length);
connect(sid, destaddr, addrlen);
```

- message passing

```
send(sid, buf, size, flags);
recv(sid, buf, size, flags);
```

```
main()
{
    char buf[100];
    socklen_t len;
    int k,sock_desc,temp_sock_desc;
    struct sockaddr_in client,server;
    memset(&client,0,sizeof(client));
    memset(&server,0,sizeof(server));
    sock_desc = socket(AF_INET,SOCK_STREAM,0);
    server.sin_family = AF_INET; server.sin_addr.s_addr = inet_addr("127.0.0.1");
    server.sin_port = 7777;
    k = bind(sock_desc,(struct sockaddr*)&server,sizeof(server));
    k = listen(sock_desc,20); len = sizeof(client);
    temp_sock_desc = accept(sock_desc,(struct sockaddr*)&client,&len);
    while(1)
    {
        k = recv(temp_sock_desc,buf,100,0);
        if(strcmp(buf,"exit")==0)
            break;
        if(k>0)
            printf("%s",buf);
    } close(sock_desc);
    close(temp_sock_desc);
    return 0;
}
```

```
main()
{
    char buf[100];
    struct sockaddr_in client;
    int sock_desc,k;
    sock_desc = socket(AF_INET,SOCK_STREAM,0);
    memset(&client,0,sizeof(client));
    client.sin_family = AF_INET;
    client.sin_addr.s_addr = inet_addr("127.0.0.1");
    client.sin_port = 7777;
    k = connect(sock_desc,(struct sockaddr*)&client,sizeof(client));
    while(1)
    {
        gets(buf);
        k = send(sock_desc,buf,100,0);
        if(strcmp(buf,"exit")==0)
            break;
    }
    close(sock_desc);
    return 0;
}
```

Virtualization

Virtualization

Definition

- the creation of a virtual version of a computing resource
- not physically existing as such but made by software to appear to do so

History

- fundamental research already in the early 70s
- only recently realized in practice due to availability of resources and advances in computer systems architecture

Source: Oxford Dictionaries (<https://en.oxforddictionaries.com/definition/virtualize>)

Key Reasons for Using Virtualization

- We can summarize the key reasons the organizations use virtualization as follows:
- Legacy hardware
 - Applications built for legacy hardware can still be run by virtualizing the legacy hardware, enabling the retirement of the old hardware
- Rapid deployment
 - A new VM may be deployed in a matter of minutes
- Versatility
 - Hardware usage can be optimized by maximizing the number of kinds of applications that a single computer can handle
- Consolidation
 - A large-capacity or high-speed resource can be used more efficiently by sharing the resource among multiple applications simultaneously
- Aggregating
 - Virtualization makes it easy to combine multiple resources into one virtual resource, such as in the case of storage virtualization
- Dynamics
 - Hardware resources can be easily allocated in a dynamic fashion, enhancing load balancing and fault tolerance
- Ease of management
 - Virtual machines facilitate deployment and testing of software
- Increased availability
 - Virtual machine hosts are clustered together to form pools of compute resources

Application virtual machine

Process virtual machine

- a runtime software encapsulates a process
- it appears to the operating system as a native process

High-level language virtual machines

- translate a certain programming language into machine-independent bytecode
- bytecode is converted into machine code for execution on a given hardware platform
 - at runtime by interpreters or just-in-time compilers
 - entirely beforehand by ahead-of-time compilers
- examples: Java VM and Android Runtime (ART)

Virtualization taxonomy

Platform virtualization

- also called system or server virtualization
- software layer encapsulating an operating system to replicate behavior of a physical machine

Operating system-level virtualization

- also called container-based virtualization
- shares the resources of the underlying operating system

Desktop virtualization

- also called virtual desktop infrastructure
- based on thin-client remote desktop solutions

Source: M. Pearce, S. Zeadally, and R. Hunt, “Virtualization: Issues, security threats, and solutions”, ACM Computing Surveys 45(2):17, February 2013

From workstations to data centers

Conventional computing systems

- process as primary computing abstraction
- operating systems allow process multiplexing and hardware management through a high-level interface
- privileged software (kernel space), regular programs only employ system calls (user space)
- one operating system per physical machine

Modern computing systems

- several cores, many threads
- very powerful specialized hardware
- may run multiple operating system at once

Data Center vs Cloud Computing

Definition

- a building with multiple interconnected and co-located servers
- due to different requirements, including physical security and ease of maintenance

What does a data center need?

- space
 - warehouse-scale computers: an extremely large number of servers, together with their software platforms
- power
- cooling

Source: Luiz André Barroso et al., *The Datacenter as a Computer*, Morgan & Claypool, 2013

Definition

- a model for enabling ubiquitous, [...] on-demand network access
- to a shared pool of configurable computing resources
 - e.g., networks, servers, storage, applications, and services
- that can be rapidly provisioned and released
- with minimal management effort or service provider interaction

Virtualization as key enabling technology

- isolation
 - applications in a virtual environment cannot access other applications running on the same physical machine
- resource control
 - applications in a virtual environment cannot use more than the resources they have been assigned

Source: P. Mell and T. Grance, "The NIST Definition of Cloud Computing", September 2011

Platform Virtualization

Platform Virtualization: Basics

Virtual machine monitor

- also called hypervisor
- provides an environment that looks like the physical system but decoupled from the hardware state

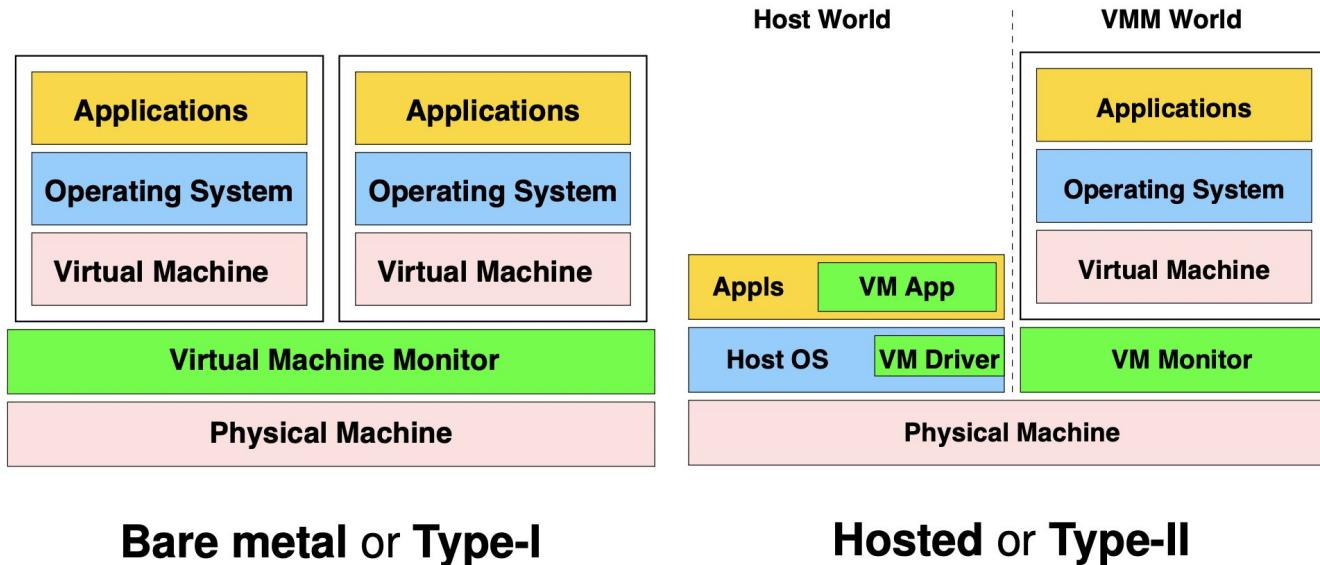
Virtual machine

- virtual environment in platform virtualization
- runs an operating system

Terminology

- host: the physical machine running the hypervisor
- guest: the operating system running within a VM

Types of virtual machine monitors



Key properties and benefits

Isolation

- virtual machines cannot interfere with each other
- not even aware that others are running on the same machine

Resource sharing

- most servers run at a low utilization (typically less than 15%)
- share one physical machine between different operating systems
- consolidate load on less active servers

Hardware independence

- virtual machines run on any cloud infrastructure
- easy to move them from one data center to another

Platform virtualization: requirements

Not all systems can be virtualized

- it depends on the specific hardware architecture
- general criteria exist to assess virtualization
- what if a certain architecture cannot be virtualized?
 - construct a hybrid virtual machine

Classical virtualization

- according to the criteria in [Popek and Goldberg, CACM 1974]
- widely used still today
- based on two classes of properties
 - for instructions
 - for the virtualized architecture

Formal Requirements for Virtualizable Third Generation Architectures

Gerald J. Popek

University of California, Los Angeles
and

Robert P. Goldberg

Honeywell Information Systems and
Harvard University

Virtual machine systems have been implemented on a limited number of third generation computer systems, e.g. CP-67 on the IBM 360/67. From previous empirical studies, it is known that certain third generation computer systems, e.g. the DEC PDP-10, cannot support a virtual machine system. In this paper, model of a third-generation-like computer system is developed. Formal techniques are used to derive precise sufficient conditions to test whether such an architecture can support virtual machines.

That is, an instruction is control sensitive if it attempts to change the amount of (memory) resources available, or affects the processor mode without going through the memory trap sequence.² The examples given of privileged instructions are also control sensitive. Another example of a control sensitive instruction is JRST 1, on the DEC PDP-10, which is a return to user mode.

Key Words and Phrases: operating system, third
sensitive instruction, formal
el, proof, virtual machine,
virtual machine monitor
5, 5.21, 5.22

Properties of classical virtualization

Instruction properties

- Privilege: a privileged instruction allows a user space process to run a kernel space function, for instance, system calls invoking a software interrupt (trap)
- Sensitivity: a sensitive instruction may interfere with the hypervisor

Virtualized architecture properties

- Efficiency: hypervisor runs non-sensitive instructions directly
- Control: guest cannot bypass the hypervisor to access or manipulate resources
- Equivalence: hypervisor behaves the same as a physical machine, except for timing and resource availability

There are three properties of interest when any arbitrary program is run while the control program is resident: efficiency, resource control, and equivalence.

The efficiency property. All innocuous instructions are executed by the hardware directly, with no intervention at all on the part of the control program.

The resource control property. It must be impossible for that arbitrary program to affect the system resources, i.e. memory, available to it; the allocator of the control program is to be invoked upon any attempt.

The equivalence property. Any program K executing with a control program resident, with two possible exceptions, performs in a manner indistinguishable from the case when the control program did not exist and K had whatever freedom of access to privileged instructions that the programmer had intended.

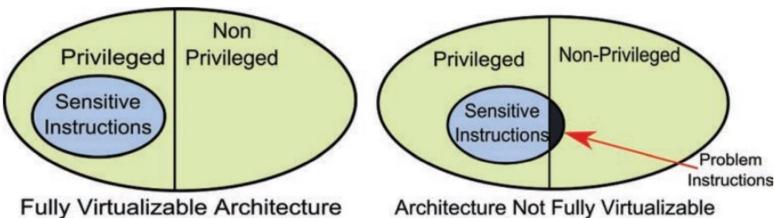
As mentioned earlier, the two exceptions result from timing and resource availability problems. Because of

THEOREM 1. *For any conventional third generation computer, a virtual machine monitor may be constructed if the set of sensitive instructions for that computer is a subset of the set of privileged instructions.*

Classically virtualizable architectures

Constraints for full virtualization

- “an architecture is fully virtualizable if the set of sensitive instructions [...] is a subset of the set of privileged instructions”
- the x86 architecture is not fully virtualizable



Source: M. Pearce, S. Zeadally, and R. Hunt, “Virtualization: Issues, security threats, and solutions”, ACM Computing Surveys 45(2):17, February 2013

Non-virtualizable architectures - Approach

- construct a hybrid virtual machine
- special handling of critical instructions (i.e., non-privileged but sensitive)

Comparison with emulation

- emulator handles all instructions
- hypervisor only handles sensitive instructions
- non-sensitive instructions run on the hardware
 - consequence of the efficiency property

Paravirtualization and Hardware-assisted virtualization

Main idea

- the virtual machine abstraction is similar to (but not the same as) the real hardware

Approach

- modify operating system
- replace critical instructions with hypervisor calls
- fast but requires access to the source code
- first realized by Xen

Architecture extensions

- special functions added to hardware platforms to support virtualization
- define the structure of a hardware-based hypervisor
- I/O Memory Management Unit (IOMMU)

Hardware virtualization technologies

- AMD Virtualization (AMD-V)
- Intel Virtualization Technologies
 - CPU virtualization (VT-x)
 - memory virtualization (VT-d)
- ARM Virtualization Extensions

Virtualization software

VirtualBox

- cross-platform virtualization application, desktop-oriented
- hosted hypervisor

XEN

- open-source bare-metal hypervisor
- initially provided para-virtualization only, currently supports hardware-assisted virtualization too

KVM

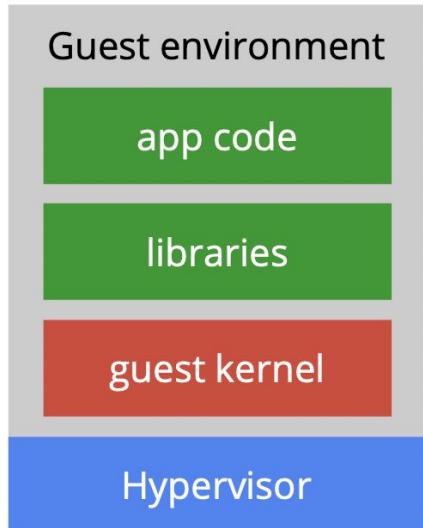
- full open-source virtualization solution for the Linux kernel
- hosted hypervisor
- hardware-assisted virtualization only

Containers and Docker

Issues with virtual machines (1/2)

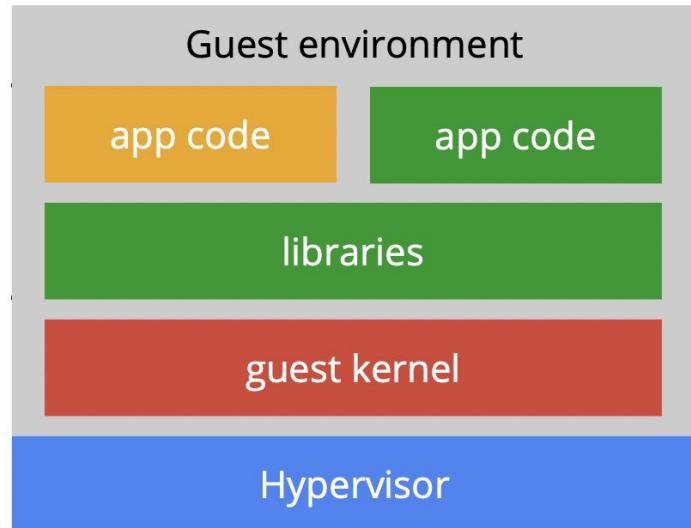
Portability

- platform-bound image



Isolation

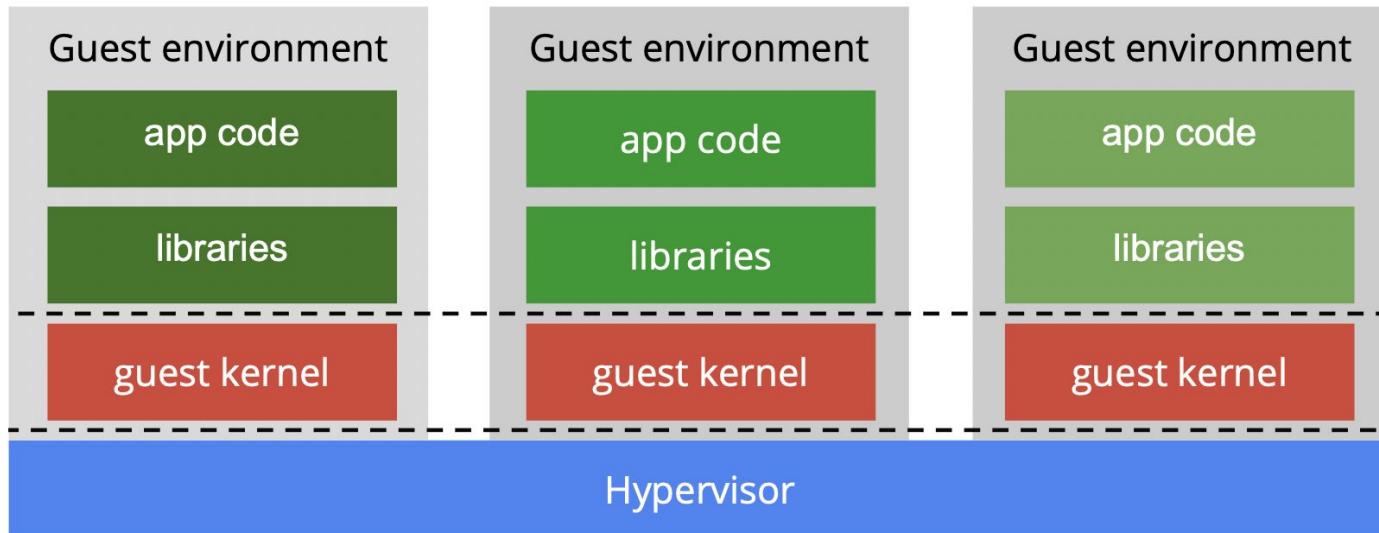
- dependencies



Issues with virtual machines (2/2)

Redundancy

- limited component reuse
- multiple unnecessary guest operating systems



Containers as virtualization technology

Operating system-level virtualization

- also called container-based virtualization
- shares the resources of the underlying operating system
- additional mechanisms for isolation between processes

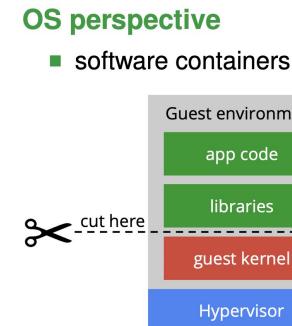
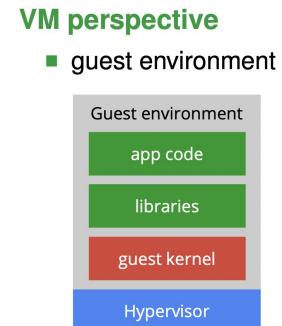
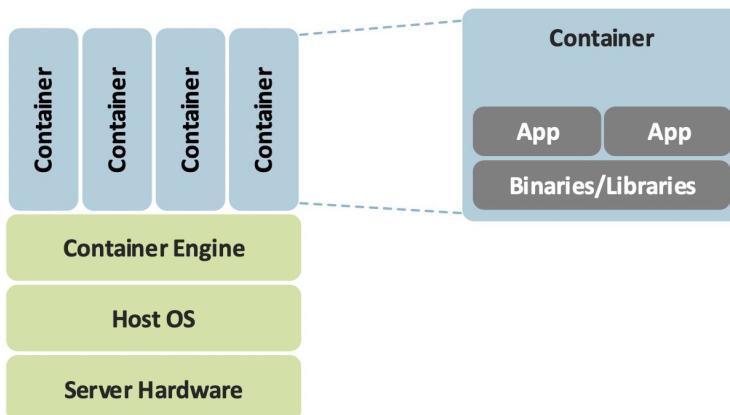


Image source: Asad Javed, “Container-based IoT Sensor Node on Raspberry Pi and the Kubernetes Cluster Framework”, Master thesis, Aalto University, 2016

Isolation in UNIX and FreeBSD, Linux

Change root

- chroot command
- changes the root of the filesystem
- does not attempt to enforce strict isolation

Jails

- also referred to as chroot jail
- facility to partition the operating system environment
- keeps the traditional UNIX security model based on root user
- processes in the jail see local resources only

Control groups

- also known as cgroups
- fine-grained resource allocation and limitation
- hierarchical structure

Namespaces

- system resources wrapped in abstraction layers
- appear to processes as isolated instances
- separate namespaces for different subsystems
 - mount (filesystems), UNIX Time-sharing Subsystem UTS (node and domain names), interprocess communication (POSIX queues), process IDs, network, users

Containers beyond virtualization

Packaging

- self-contained images
 - similar to an application package
- stored in a repository
 - most widely used: Docker Hub

Management

- automation
 - continuous integration and delivery
- container orchestration
 - examples: Docker Swarm, Kubernetes

There are several software platforms for creating, shipping, and running containers. Docker is so prevalent in the industry (and beyond) that Docker containers and software containers are commonly used as synonyms.

Separation of Concerns

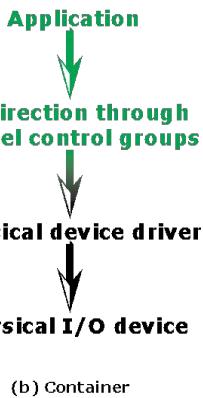
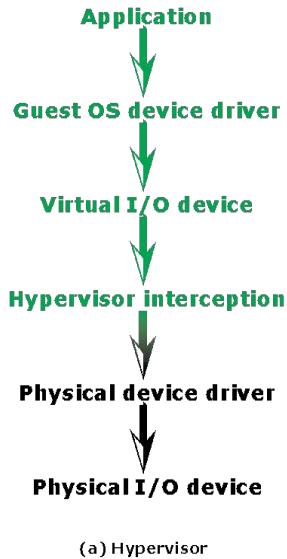
Developer view

- what is inside a container
- code, libraries, applications, data
- base environment: Linux server

System administrator view

- what is outside a container
- logging, monitoring, networking
- consistent interface for lifecycle management

Data Flow for I/O Operation via Hypervisor and Container



To compare virtual machines with containers, consider I/O operation in during an application with process P in virtualized environment.

- In classical system virtualization environment (with no hardware support), process P would be executed inside a guest virtual machine
 - I/O operation is routed through guest OS stack to emulated guest I/O device
 - I/O call is further intercepted by hypervisor that forward it through host OS stack to the physical device
- The container is primarily based on indirection mechanism provided by container framework extensions that have been incorporated into mainstream kernel
 - a single kernel is shared between multiple containers
 - in comparison with individual OS kernel in system virtual machines

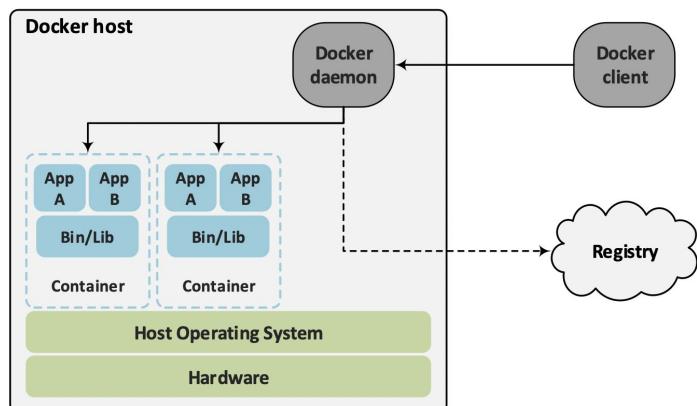
The two noteworthy characteristics of containers are the following:

1. There is no need for a guest OS in the container environment. Therefore, containers are lightweight and have less overhead compared to virtual machines.
2. Container management software simplifies the procedure for container creation and management.

Docker and its Components

Overview

- open-source engine for deployment of containerized applications
- client-server architecture with a few core components



Daemon

- server with an application programming interface
- manages containers

Client

- command-line interface to interact with the daemon

Host

- the machine running the containers
 - with an OS supporting container-based virtualization

Registry

- public or private image repository
 - examples: Docker Hub, Google Container Registry

Docker and containers

Management

- docker command-line application
- similar to standard tools for dealing with UNIX processes
 - docker ps, docker kill, docker exec

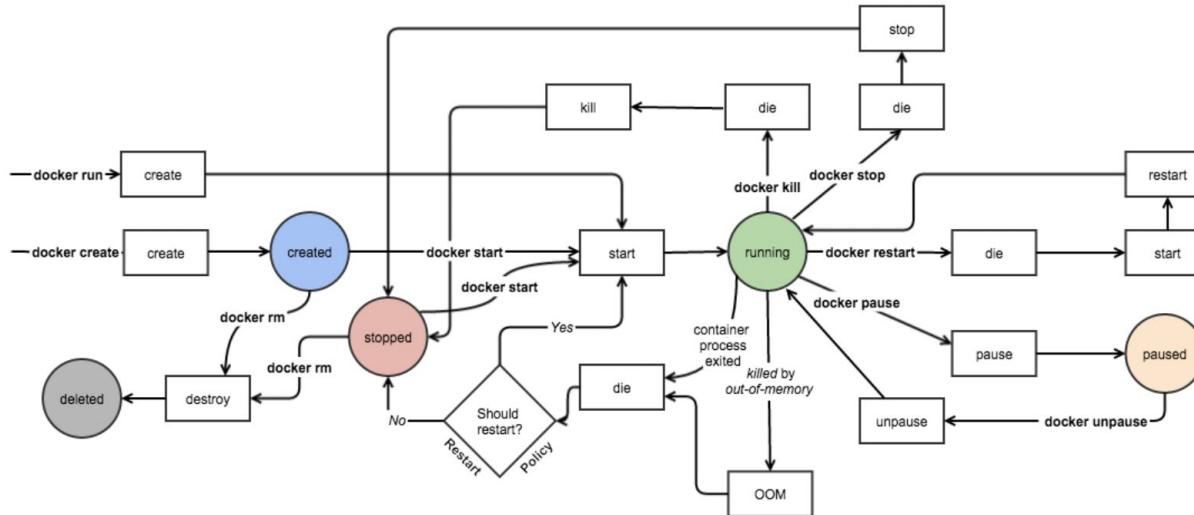


Image source: [Docker Saigon, "Docker Internals – A Deep Dive Into Docker For Engineers Interested In The Gritty Details"](#)

Virtualization

Source: J. P. Buzen and
U. O. Gagliardi, "[The evolution
of virtual machine architecture](#)",
National Computer Conference
and Exposition (AFIPS '73),
pp. 291–299, June 1973

