CHAPTER 1 INTRODUCTION TO OPERATING SYSTEMS

Class outline

- Introduction, concepts, review & historical perspective
- Processes
  - Synchronization
  - Scheduling
  - Deadlock
- Memory management, address translation, and virtual memory
- Operating system management of I/O
- File systems
- Security & protection

Overview: Chapter 1

- What ***is*** an operating system, anyway?
- Operating systems history
- The zoo of modern operating systems
- Review of computer hardware
- Operating system concepts
- Operating system structure
  - User interface to the operating system
  - Anatomy of a system call

What *is* an operating system?

- A program that runs on the "raw" hardware and supports
  - Resource Abstraction
  - Resource Sharing
- Abstracts and standardizes the interface to the user across different types of hardware
  - Virtual machine hides the messy details which must be performed
- Manages the hardware resources
  - Each program gets time with the resource
  - Each program gets space on the resource
- May have potentially conflicting goals:
  - Use hardware efficiently
  - Give maximum performance to each user

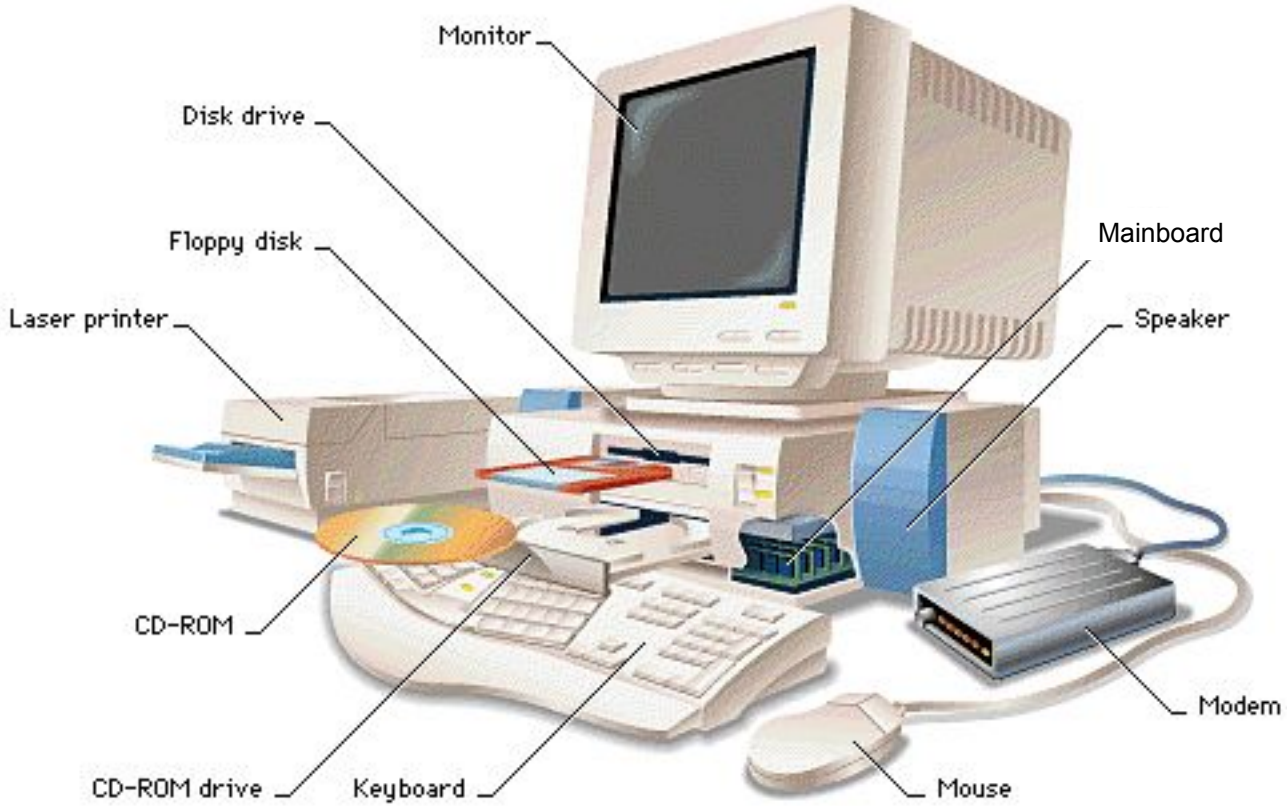
1.1 General Definition

- An OS is a program which acts as an *interface* between computer system users and the computer hardware.
- It provides a user-friendly environment in which a user may easily develop and execute programs.
- Otherwise, hardware knowledge would be mandatory for computer programming.
- So, it can be said that an OS hides the complexity of hardware from uninterested users.

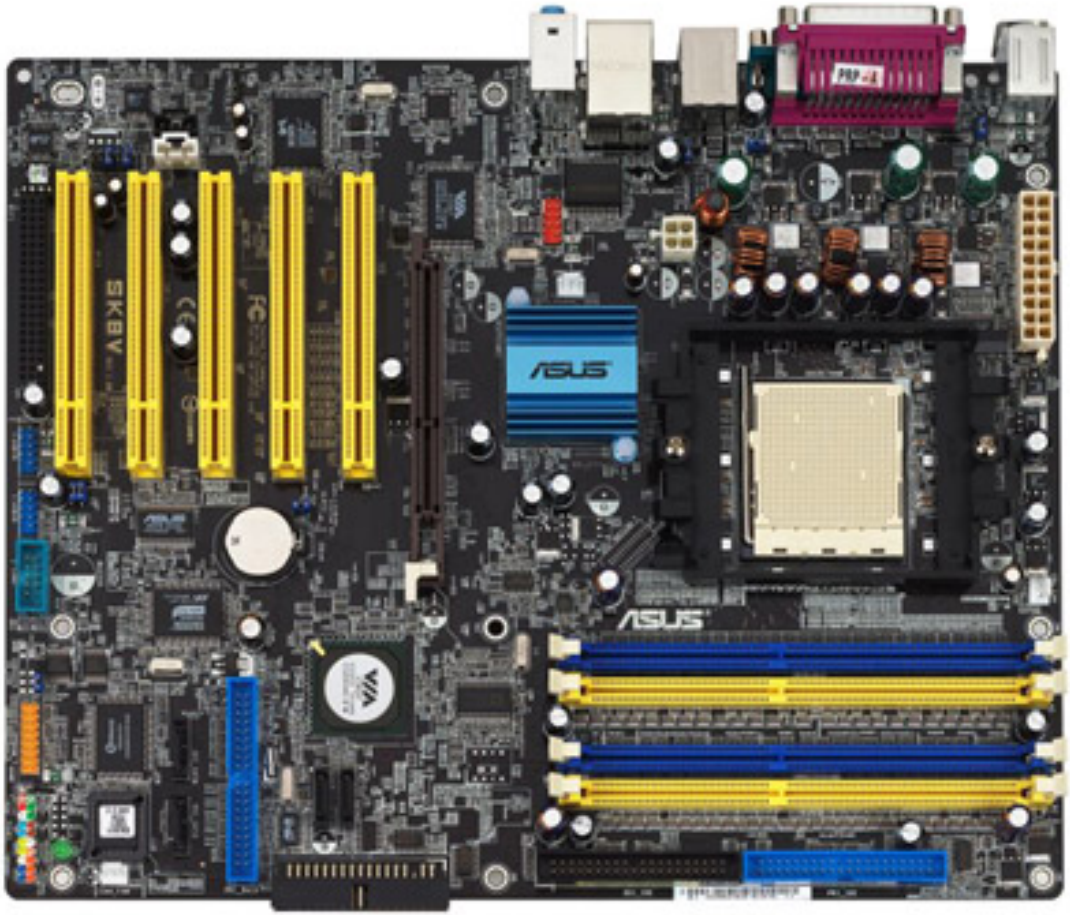
1.1 General Definition

- In general, a computer system has some resources which may be utilized to solve a problem. They are
  - Memory
  - Processor(s)
  - I/O
  - File System
  - etc.

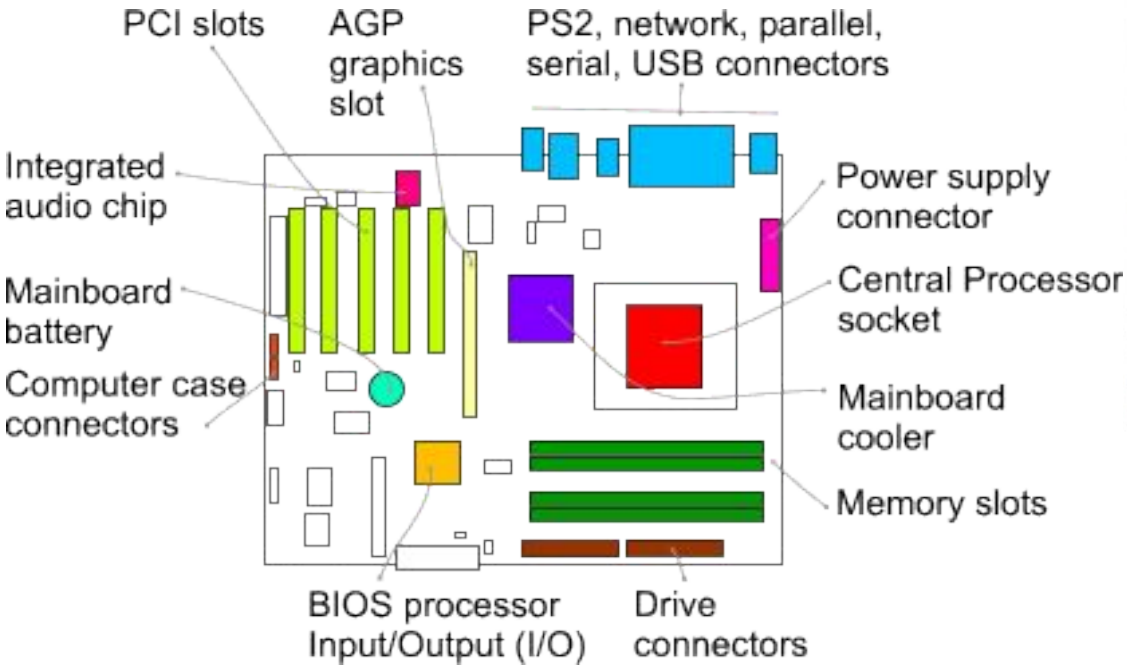
1.1 General Definition

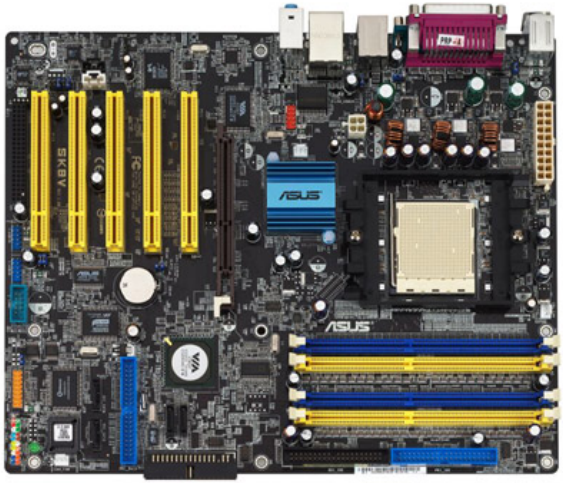


1.1 General Definition



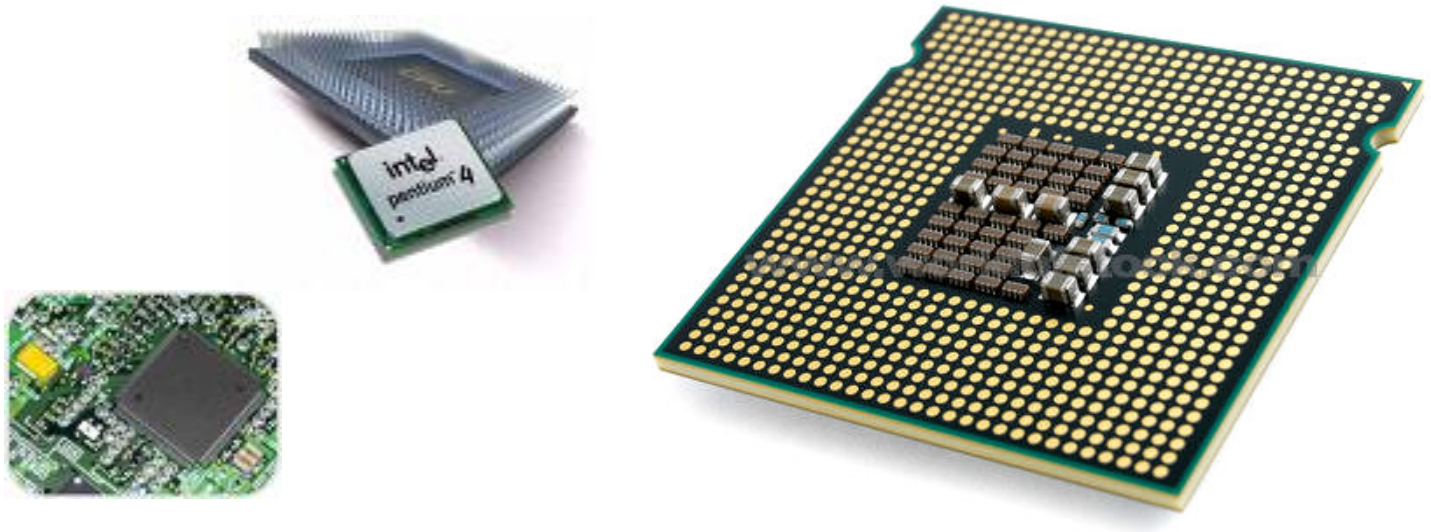
1.1 General Definition





mainboard

1.1 General Definition



processor

1.1 General Definition

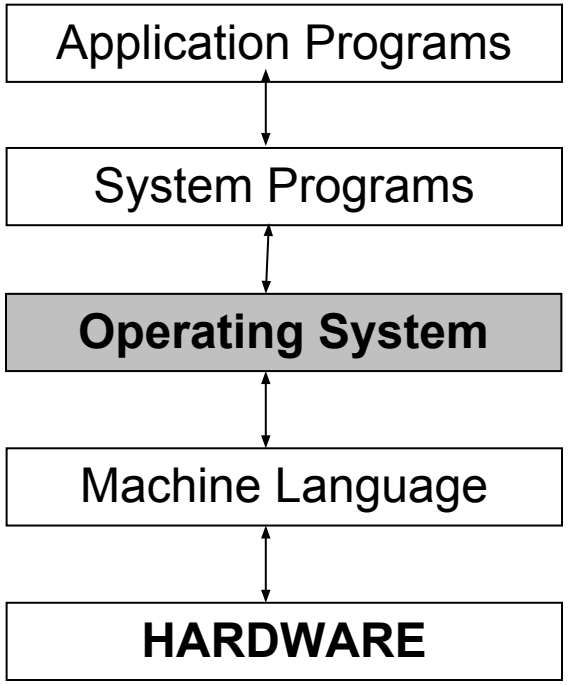
RAM

1.1 General Definition

- The OS manages these resources and allocates them to specific programs and users.
- With the management of the OS, a programmer is rid of difficult hardware considerations.
- An OS provides services for
  - Processor Management
  - Memory Management
  - File Management
  - Device Management
  - Concurrency Control

1.1 General Definition

- Another aspect for the usage of OS is that; it is used as a *predefined library* for hardware-software interaction.
- This is why, system programs apply to the installed OS since they cannot reach hardware directly.



1.1 General Definition

- Since we have an already written library, namely the OS, to add two numbers we simply write the following line to our program:

  c = a + b ;

1.1 General Definition

- in a system where there is no OS installed, we should consider some hardware work as:

  (Assuming an MC 6800 computer hardware)

LDAA $80 □ Loading the number at memory location 80
LDAB $81 □ Loading the number at memory location 81
ADDB □ Adding these two numbers
STAA $55 □ Storing the sum to memory location 55

- As seen, we considered memory locations and used our hardware knowledge of the system.

1.1 General Definition

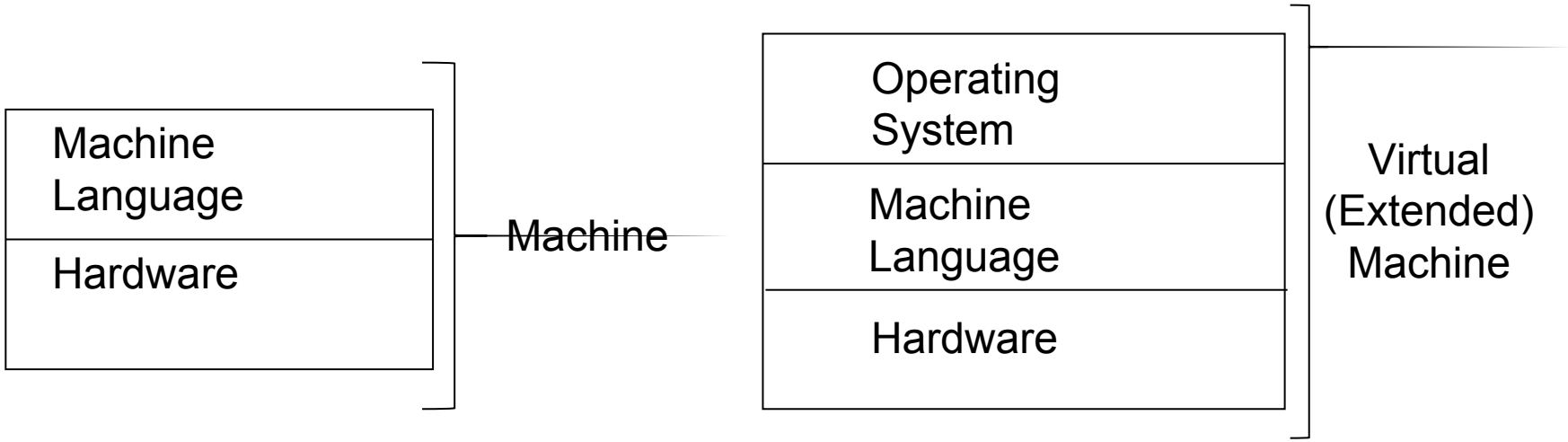
- In an OS installed machine, since we have an intermediate layer, our programs obtain *some advantage of mobility* by not dealing with hardware.
- For example, the above program segment would not work for an 8086 machine, where as the

  “c = a + b ;”

  syntax will be suitable for both.

1.1 General Definition

- With the advantage of easier programming provided by the OS, the hardware, its machine language and the OS constitutes a new combination called as a **virtual (extended) machine**.



1.1 General Definition

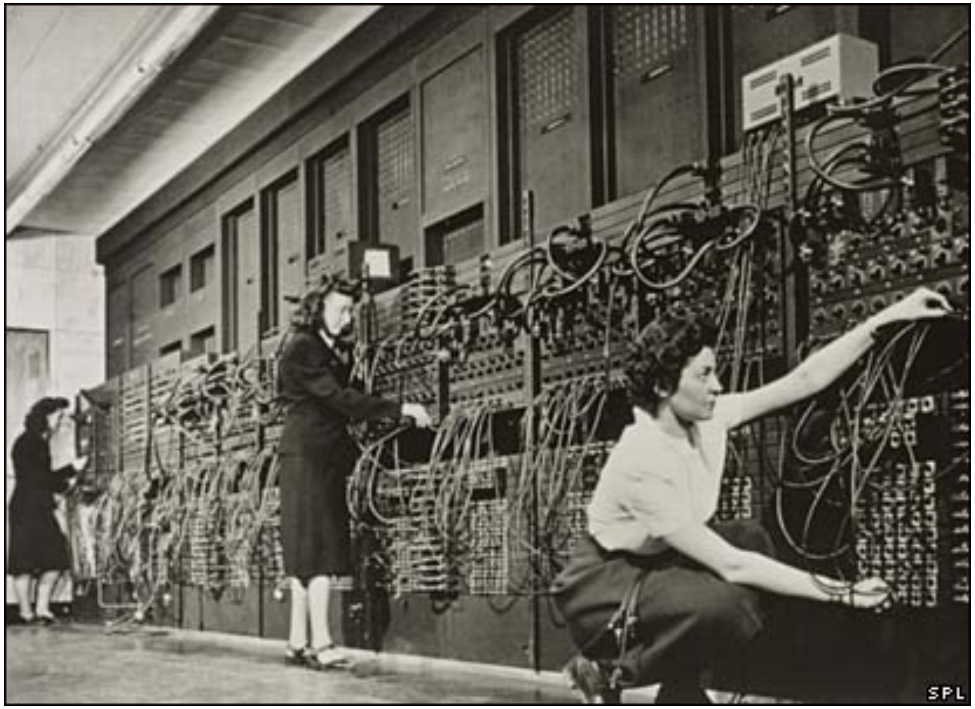
- In a more simplistic approach, in fact, OS itself is a program.
- But it has a priority which application programs don't have.
- OS uses the **kernel mode** of the microprocessor, whereas other programs use the **user mode**.
- The difference between two is that; all hardware instructions are valid in kernel mode, where some of them cannot be used in the user mode.

Operating system timeline

- First generation: 1945 – 1955
  - Vacuum tubes
  - Plug boards
- Second generation: 1955 – 1965
  - Transistors
  - Batch systems
- Third generation: 1965 – 1980
  - Integrated circuits
  - Multiprogramming
- Fourth generation: 1980 – present
  - Large scale integration
  - Personal computers
- Next generation: ???
  - Systems connected by high-speed networks?
  - Wide area resource management?

1.2 History of Operating Systems

- It all started with computer hardware in about 1940s.



ENIAC 1943

1.2 History of Operating Systems

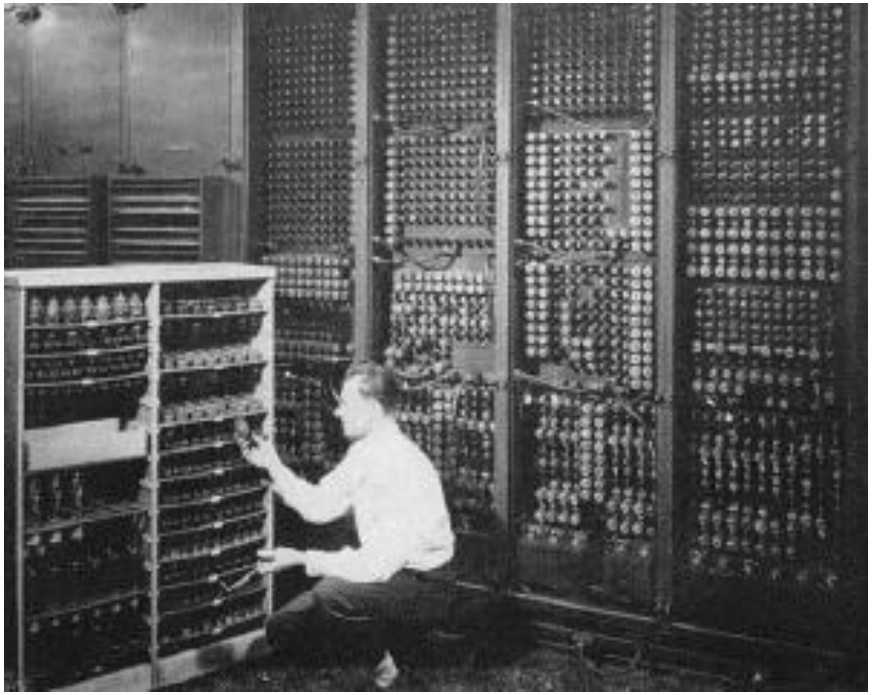
- ENIAC (Electronic Numerical Integrator and Computer), at the U.S. Army's Aberdeen Proving Ground in Maryland.
  - built in the 1940s,
  - weighed 30 tons,
  - was eight feet high, three feet deep, and 100 feet long
  - contained over 18,000 vacuum tubes that were cooled by 80 air blowers.

1.2 History of Operating Systems

- Computers were using vacuum tube technology.

ENIAC's vacuum tubes

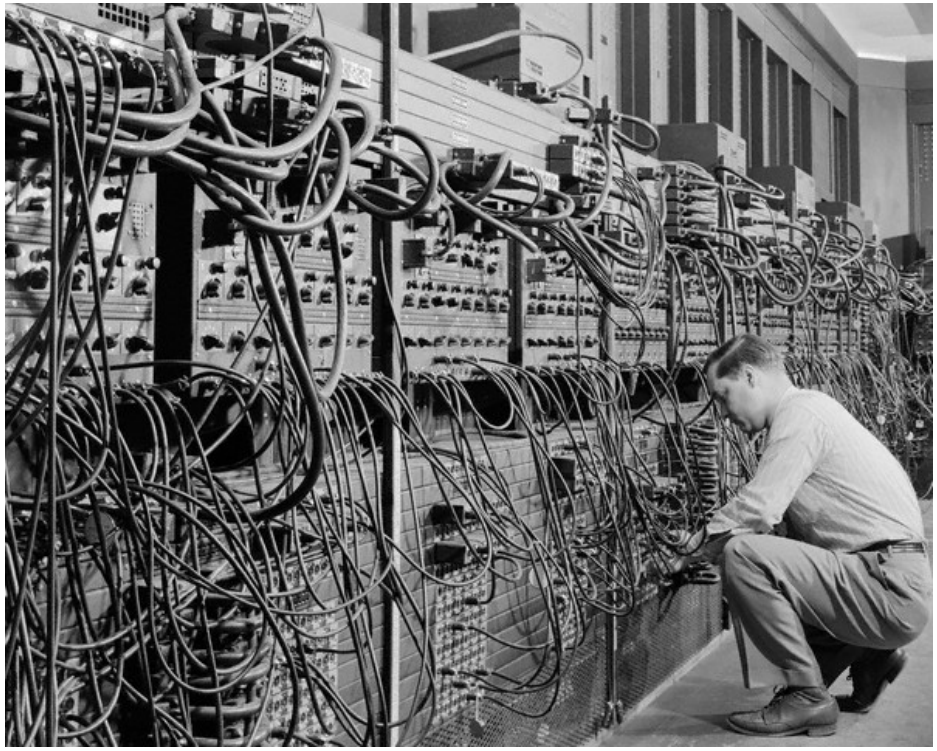
1.2 History of Operating Systems



ENIAC's backside

1.2 History of Operating Systems

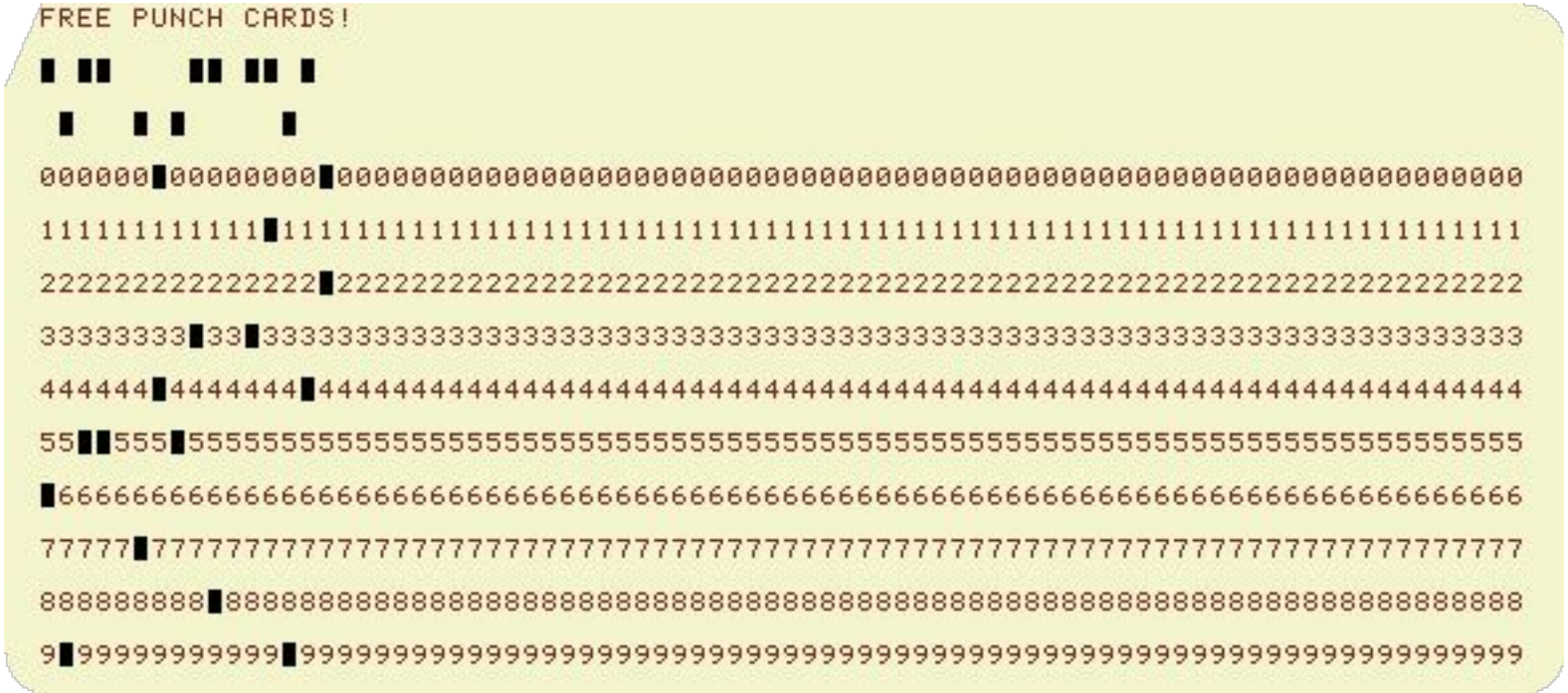
Programs were loaded into memory manually using switches, punched cards, or paper tapes.



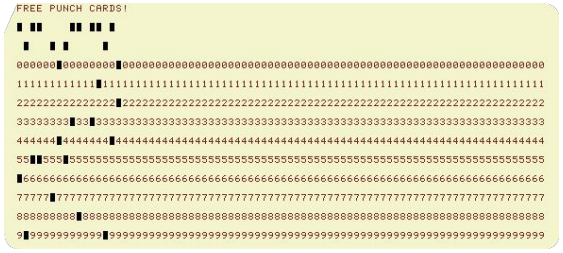
ENIAC : coding by cable connections

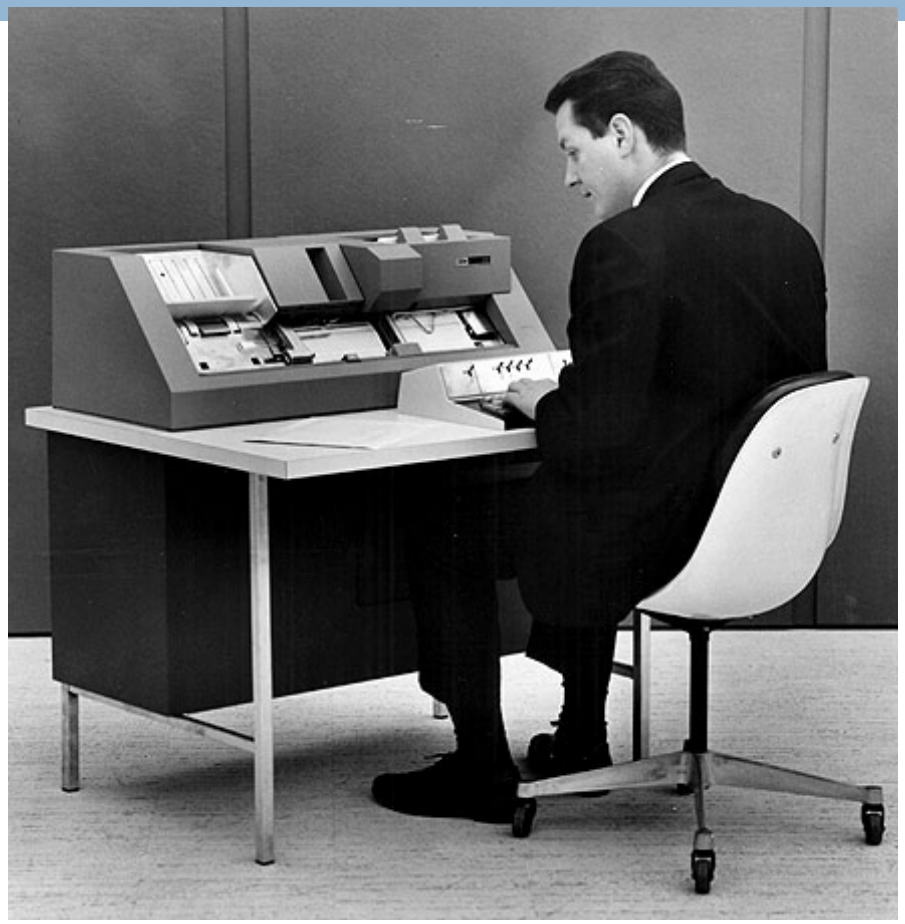
1.2 History of Operating Systems

punch card

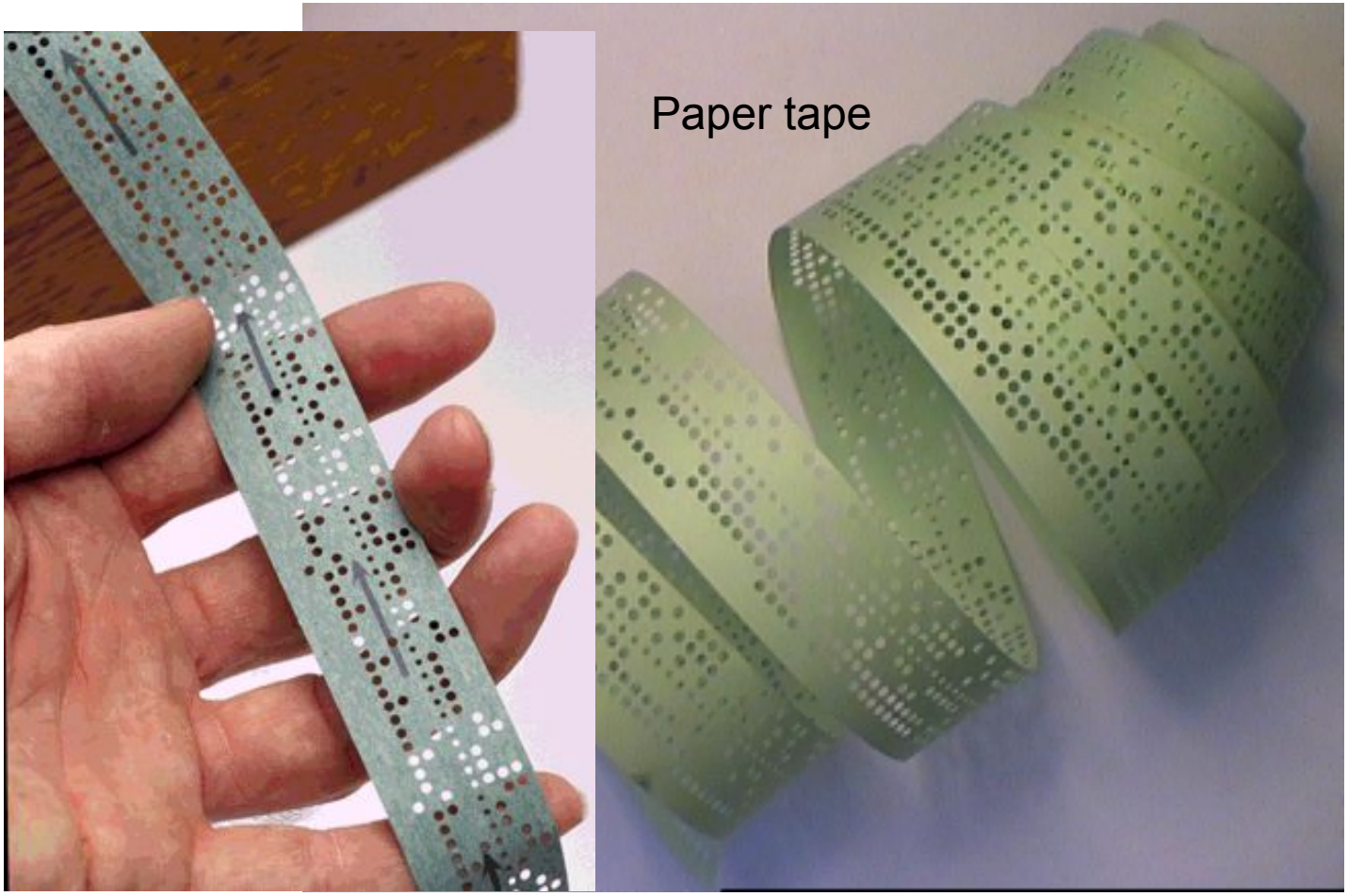


1.2 History of Operating Systems

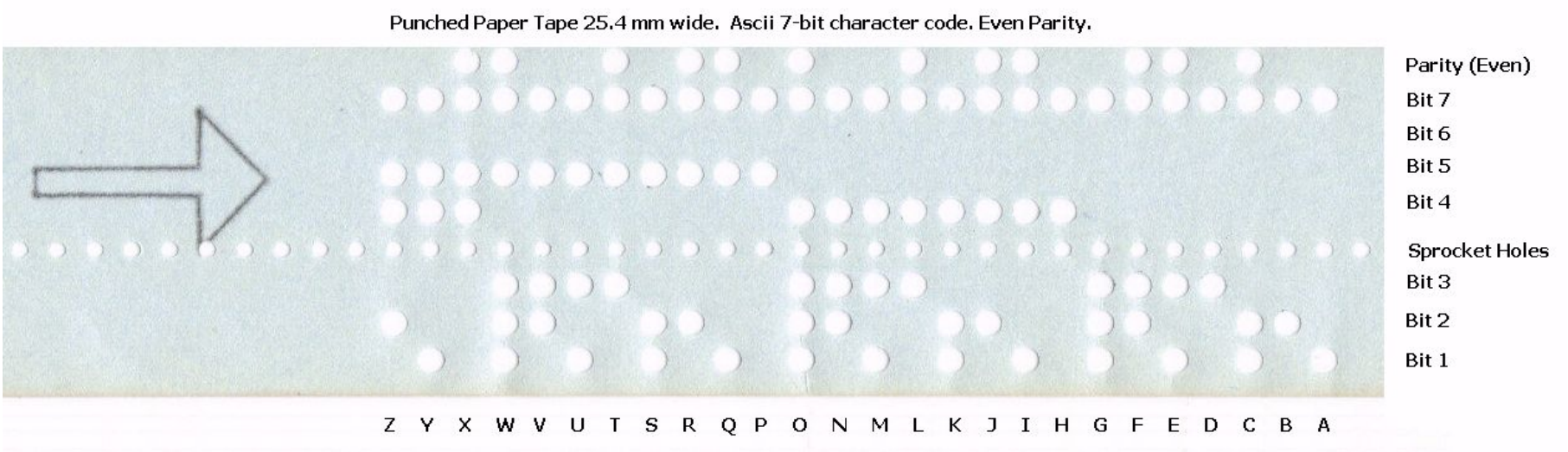




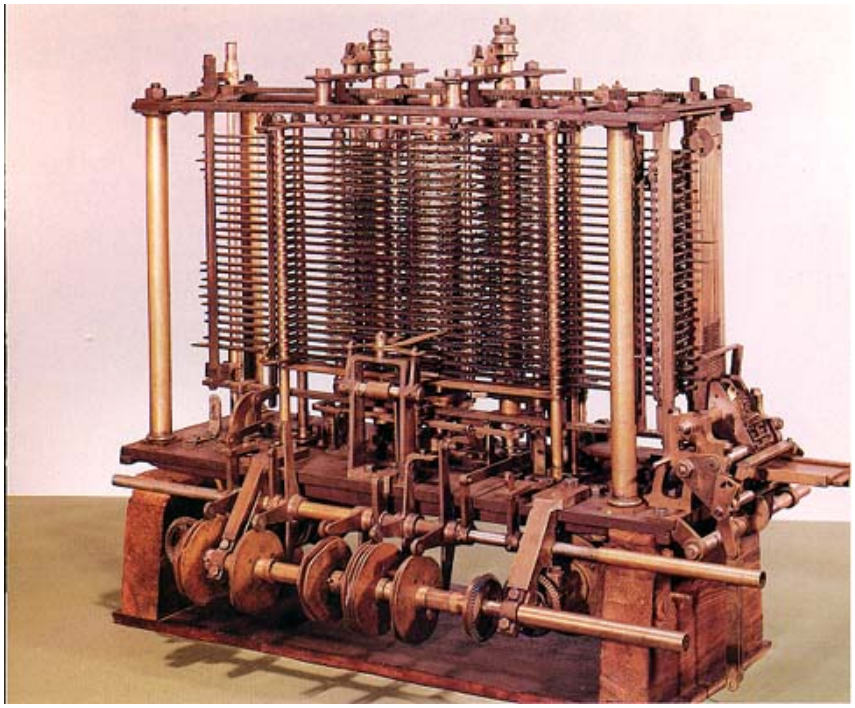
1.2 History of Operating Systems



1.2 History of Operating Systems



1.2 History of Operating Systems

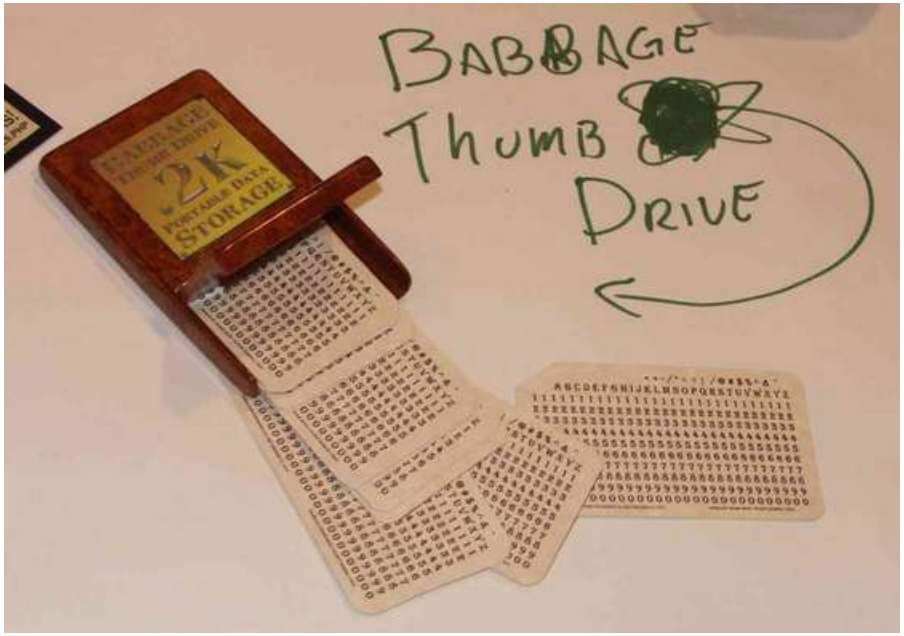


Babbage's analytical engine
(designed in 1840's by Charles Babbage, but could not be constructed by him.
An earlier and simpler version is constructed in 2002, in London)

http://www.computerhistory.org/babbage/

1.2 History of Operating Systems

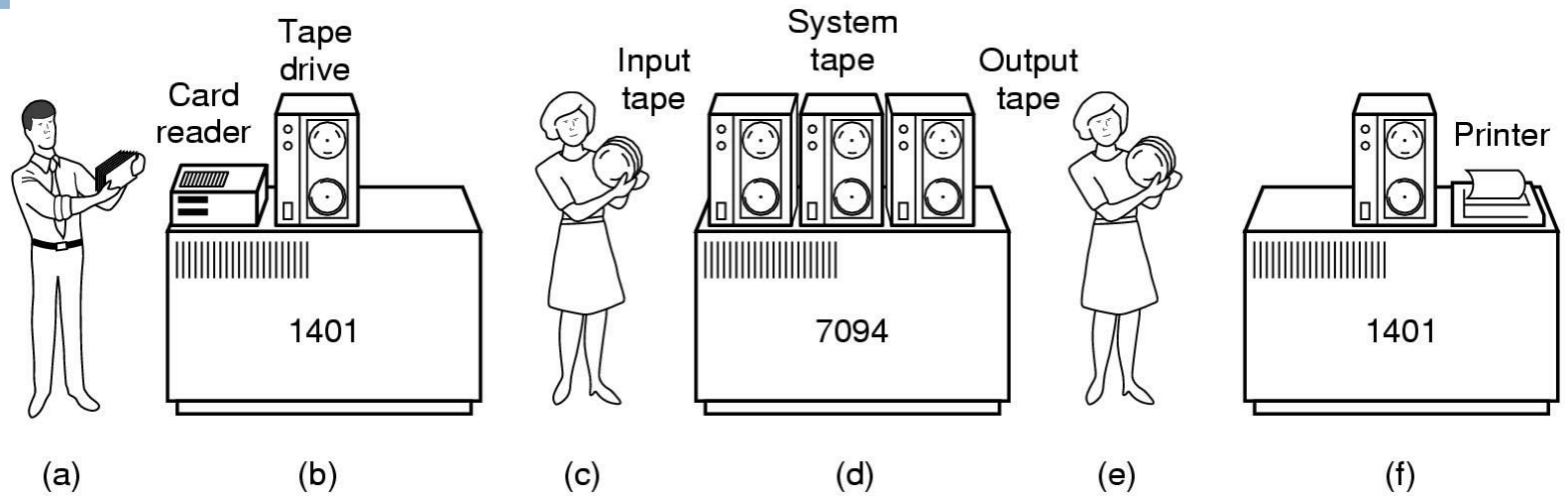
- Ada Lovalence (at time of Charles Babbage) wrote code for analytical engine to compute Bernulli Numbers



First generation: direct input

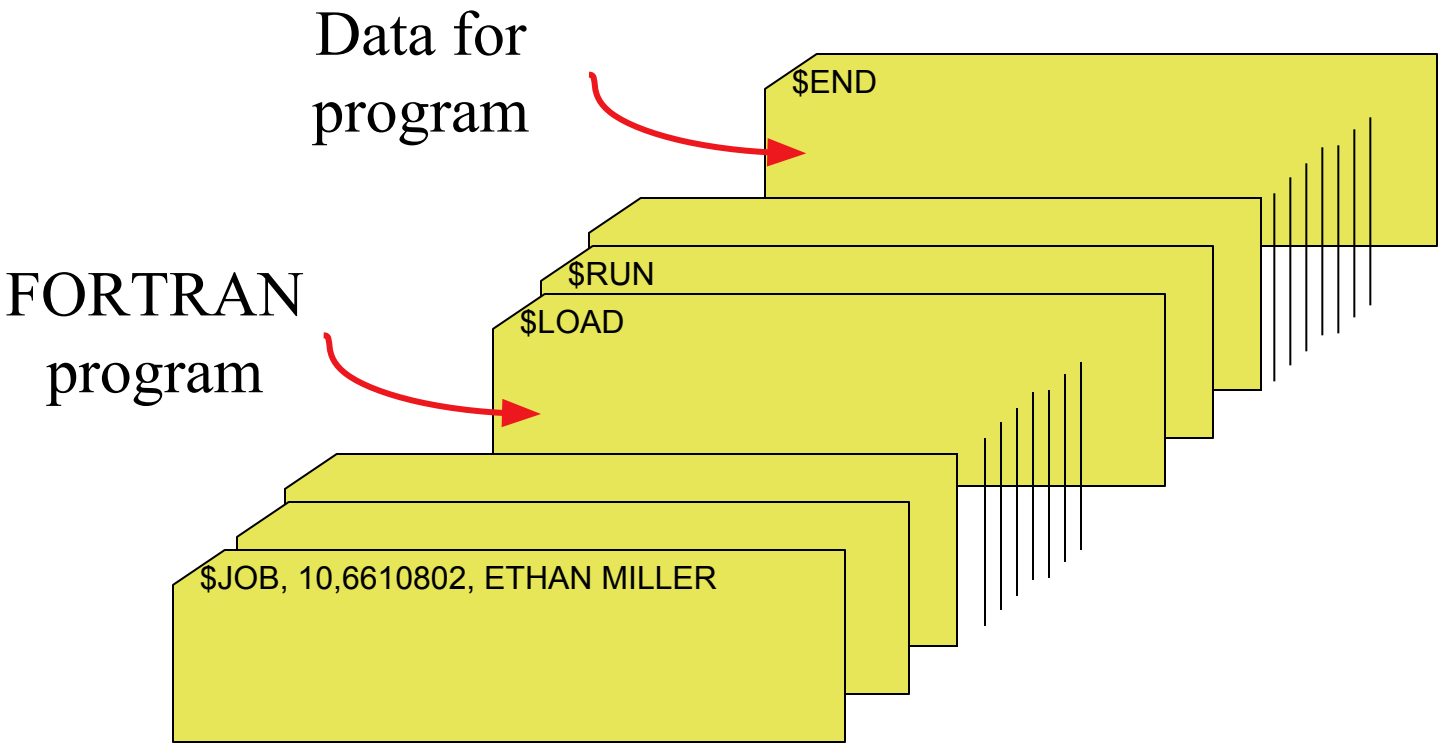
- Run one job at a time
  - Enter it into the computer (might require rewiring!)
  - Run it
  - Record the results
- Problem: lots of wasted computer time!
  - Computer was idle during first and last steps
  - Computers were ***very*** expensive!
- Goal: make better use of an expensive commodity: computer time

Second generation: batch systems

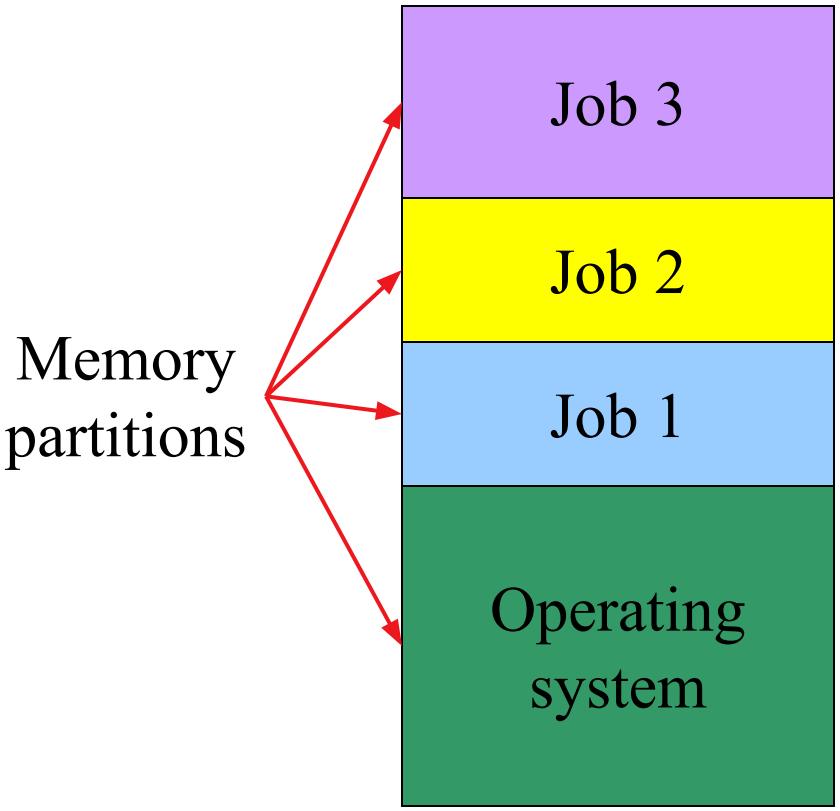


- Bring cards to 1401
- Read cards onto input tape
- Put input tape on 7094
- Perform the computation, writing results to output tape
- Put output tape on 1401, which prints output

Structure of a typical 2nd generation job



Third generation: multiprogramming



- Multiple jobs in memory
  - Protected from one another
- Operating system protected from each job as well
- Resources (time, hardware) split between jobs
- Still not interactive
  - User submits job
  - Computer runs it
  - User gets results minutes (hours, days) later

Spooling

- Original batch systems used tape drives
- Later batch systems used disks for buffering
  - Operator read cards onto disk attached to the computer
  - Computer read jobs from disk
  - Computer wrote job results to disk
  - Operator directed that job results be printed from disk
- Disks enabled simultaneous peripheral operation on-line (spooling)
  - Computer overlapped I/O of one job with execution of another
  - Better utilization of the expensive CPU
  - Still only one job active at any given time

Spooling

- Stands for Simultaneous Peripheral Operation On-Line
- Takes advantage of disk technology (new at this point)
- Allows for overlap of IO from one job with the computation of another job
- While executing current job
  - Read next job from card reader to disk
  - Print previous job to printer
- Disk is relegated to the role of a partitioned buffer

Job pool

- Advent of disk allows for random access
  - (Tape and card are sequential)
- Several jobs can be waiting on the disk
- The job pool is a data structure that contains info and points to the jobs on the disk
- We can now have job scheduling to determine the order in which the jobs run so that CPU utilization can increase.

Multitasking (Time-sharing)

- Extension of Multiprogramming
  - Need for user interactivity
  - Instead of switching jobs when waiting for IO, a timer causes jobs to switch
- User interacts with computer via CRT and keyboard
  - Systems have to balance CPU utilization against response time
  - Better device management
- Need for file system to allow user to access data and code
- Need to provide user with an “interaction environment”

Virtual Memory

- Programs can be larger than memory
  - Program loaded into memory as needed
  - Active program and data “swapped” to a disk until needed
- Memory space treated uniformly

Key Events 3rd Generation

- 1964-1966 IBM/360 and OS/360
- 1964 Dartmouth Time Sharing System
- 1965 DEC PDP-8
- 1965 MIT – Multics Time sharing System
- 1969 – Beginnings of ARPANet
- 1969 - Unix
- 1971 IBM 4001 – Processor on a chip
- 1973 – Ethernet concept Bob Metcalf @ Xerox Parc
- 1974 - Gary Kildall – CP/M OS
- 1974 Zilog Z80 Processor

Key Events (cont)

- 1974 - Edward Roberts, William Yates and Jim Bybee
  - MITS Altair 8800.
    - $375
    - contained 256 bytes of memory
    - no keyboard, no display, and no aux storage device.
- 1976 Steve Jobs and Steve Wozniak
  - Apple II
- 1977 Commodore PET, Radio Shack TRS_80
- 1979 Unix 3BSD

Fourth Generation : (1980 – 1990)

- Personal Computers
- Computer dedicated to a single user
- IO Devices now consist of keyboards, mice, CGA-VGA displays, small printers
- User convenience and responsiveness
- Can adopt lessons from larger operating systems
- No need for some of the advanced options at the personal level

Key Events 4th Generation

- 1981 IBM PC (8086)
- 1981 Osborne 1
- 1981 Vic 20
- 1981 Xerox Star Workstation
- 1984 Apple macintosh
- 1984 SunOS
- 1985 C++
- 1985 MSWindows
- 1986 – 386 Chip

Key Events 4th Generation (cont)

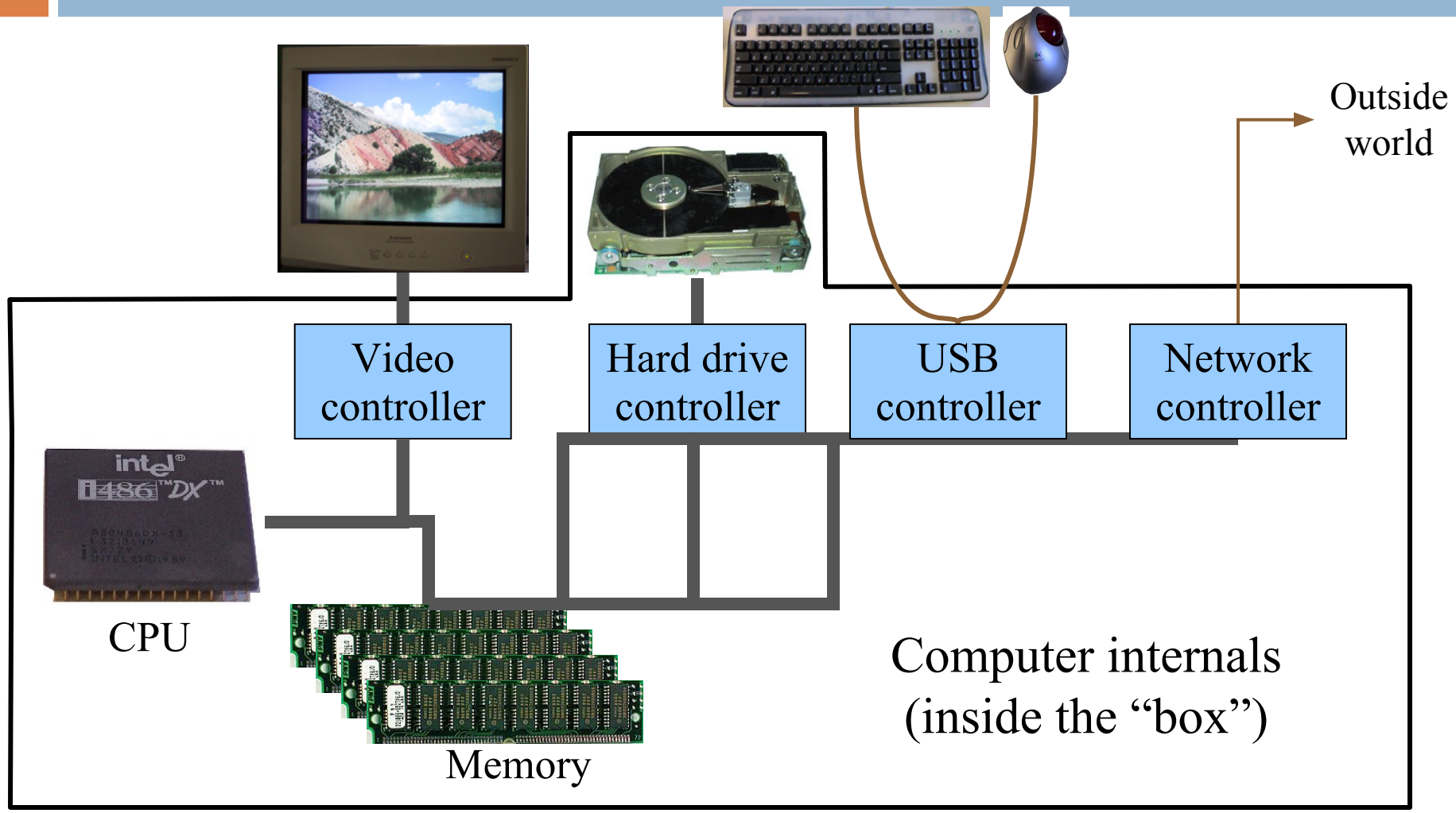
- 1987 OS/2
- 1988 Next Unix Workstations
- 1989 Motif
- 1990 Windows 3,
- 1990 Berners-Lee Prototype for the web

Types of modern operating systems

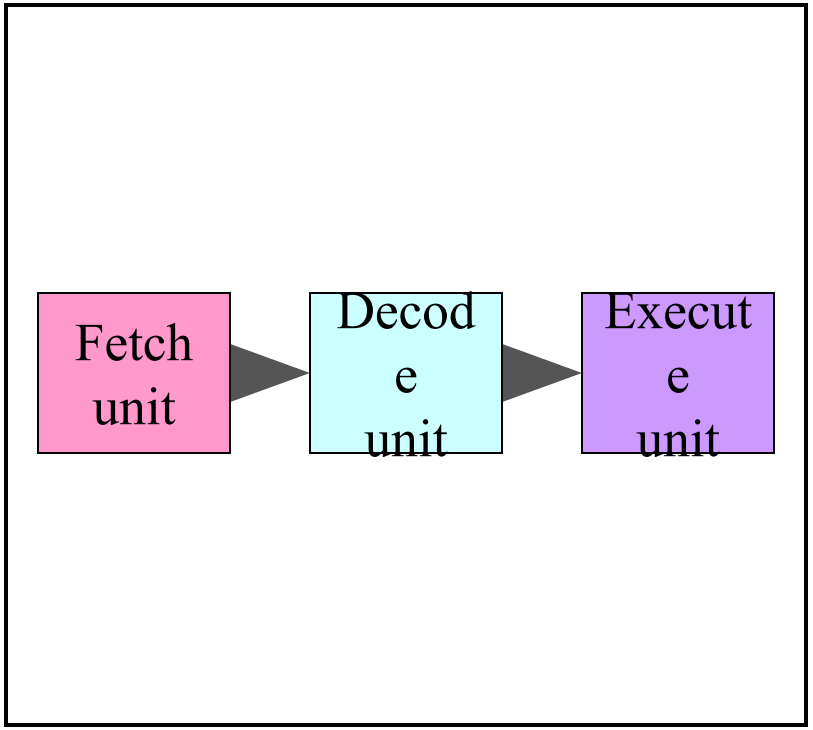
- Mainframe operating systems: MVS
- Server operating systems: FreeBSD, Solaris
- Multiprocessor operating systems: Cellular IRIX
- Personal computer operating systems: Windows, Unix
- Real-time operating systems: VxWorks
- Embedded operating systems
- Smart card operating systems

⇒ Some operating systems can fit into more than one category

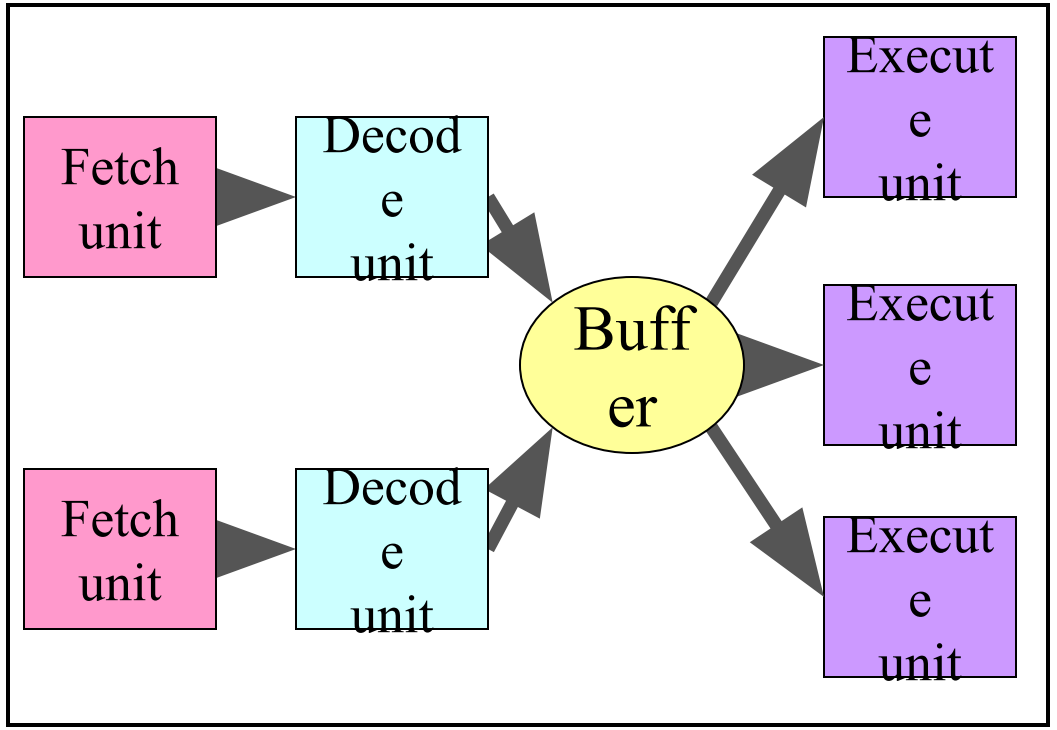
Components of a simple PC



CPU internals

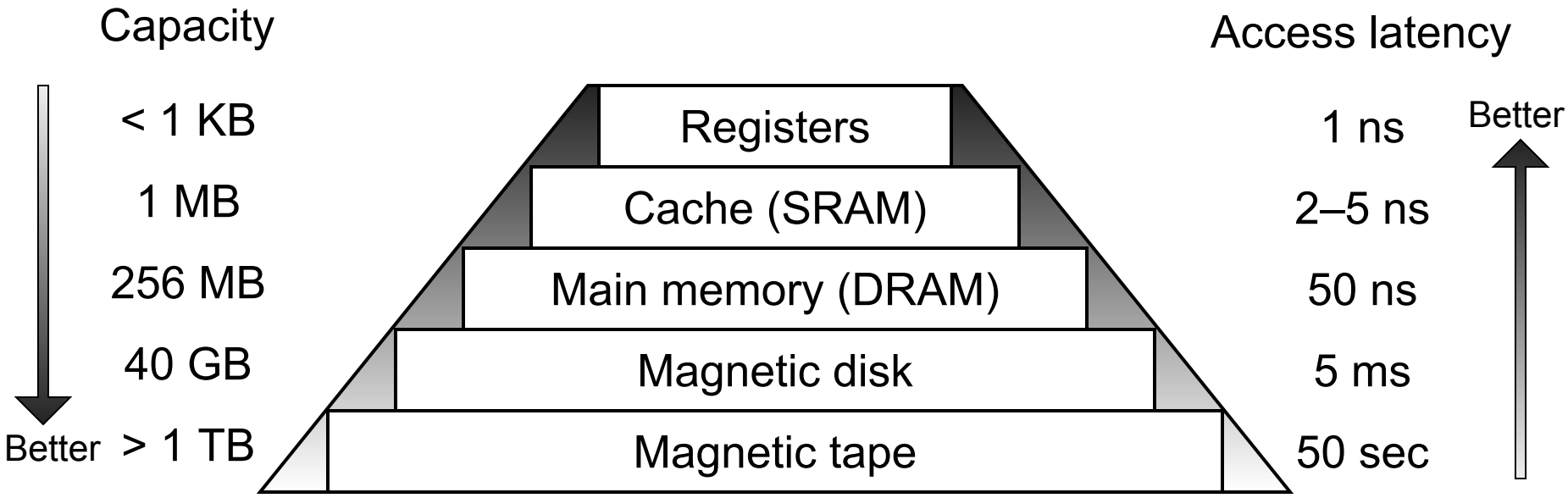


Pipelined CPU



Superscalar CPU

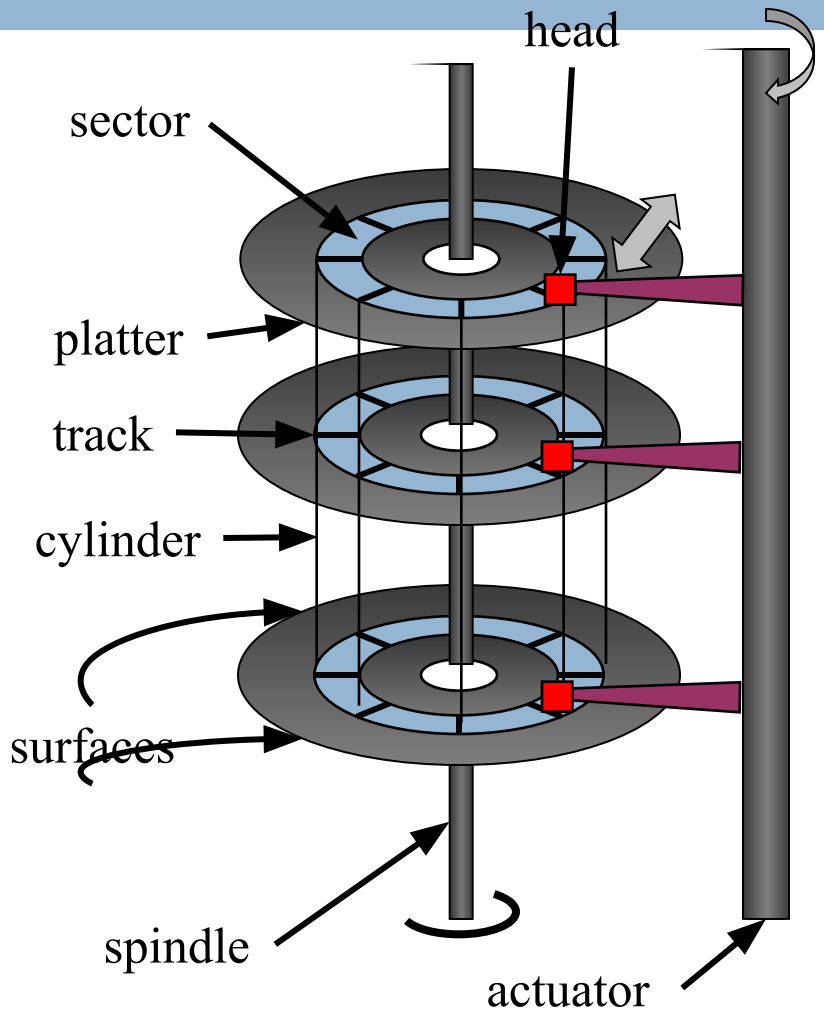
Storage pyramid



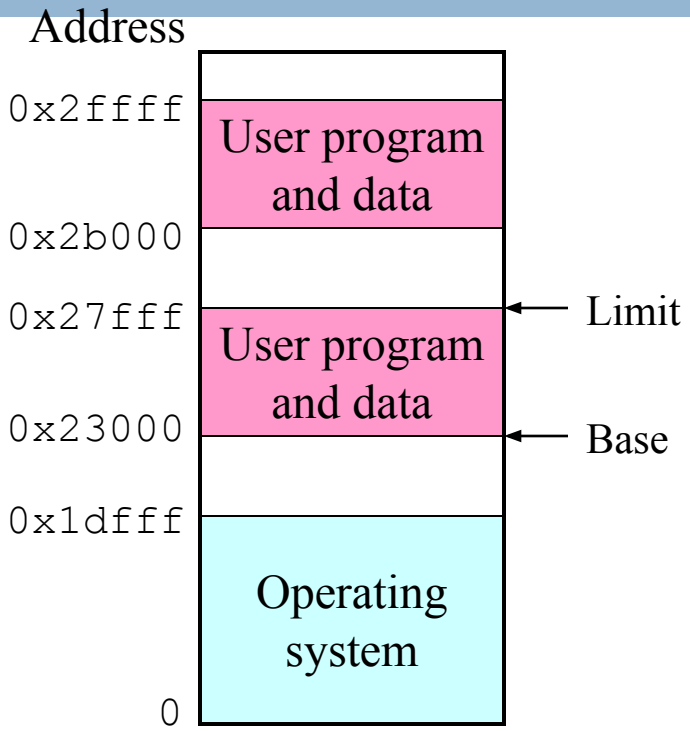
- Goal: really large memory with very low latency
  - Latencies are smaller at the top of the hierarchy
  - Capacities are larger at the bottom of the hierarchy
- Solution: move data between levels to create illusion of large memory with low latency

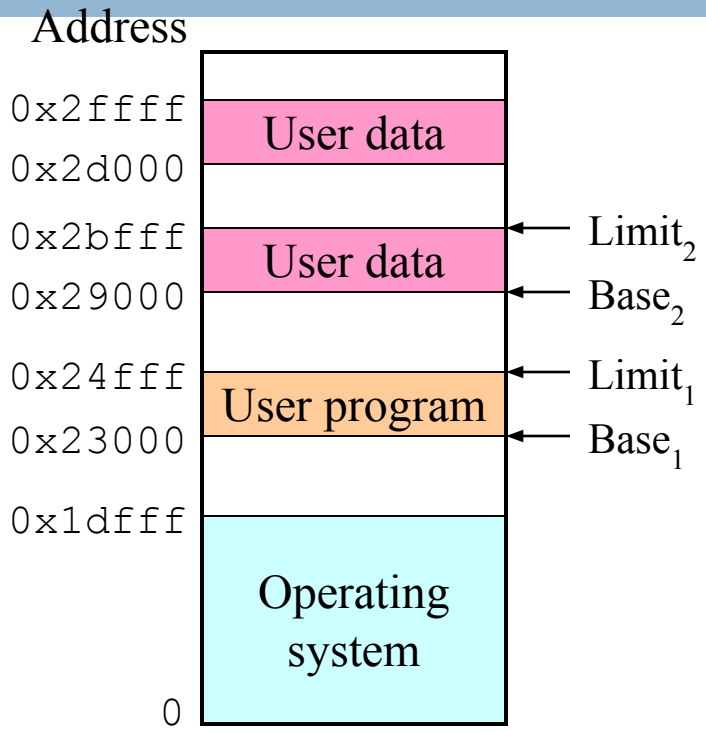
Disk drive structure

- Data stored on surfaces
  - Up to two surfaces per platter
  - One or more platters per disk
- Data in concentric tracks
  - Tracks broken into sectors
    - 256B-1KB per sector
  - Cylinder: corresponding tracks on all surfaces
- Data read and written by heads
  - Actuator moves heads
  - Heads move in unison



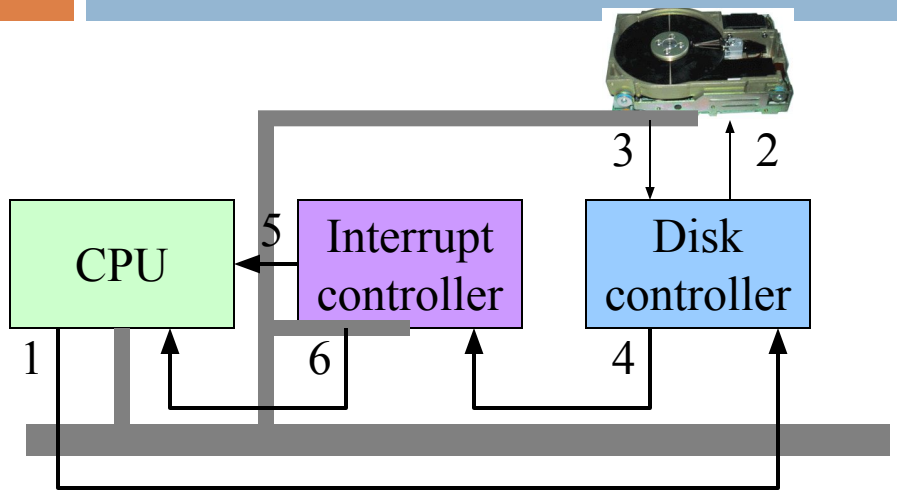
Memory

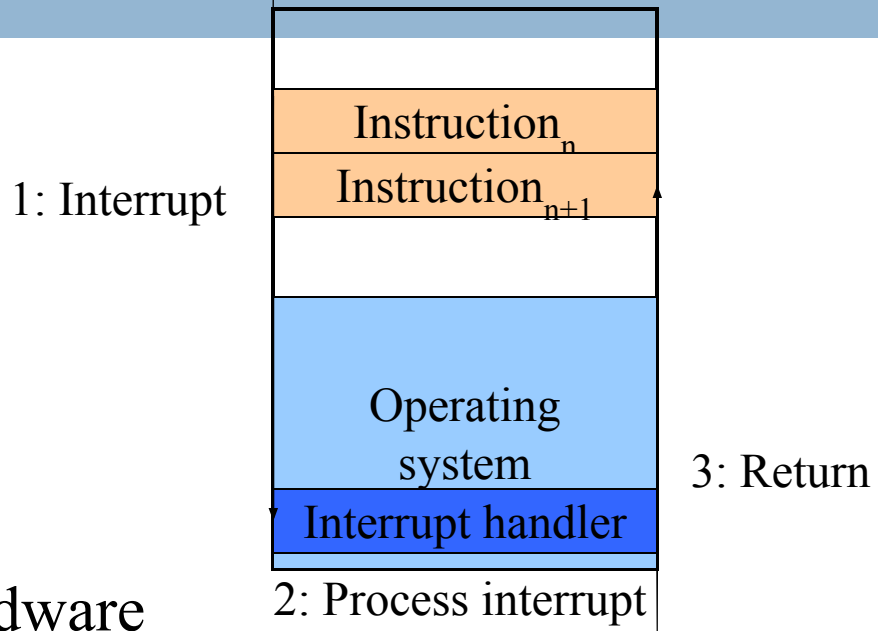




- Single base/limit pair: set for each process
- Two base/limit registers: one for program, one for data

Anatomy of a device request



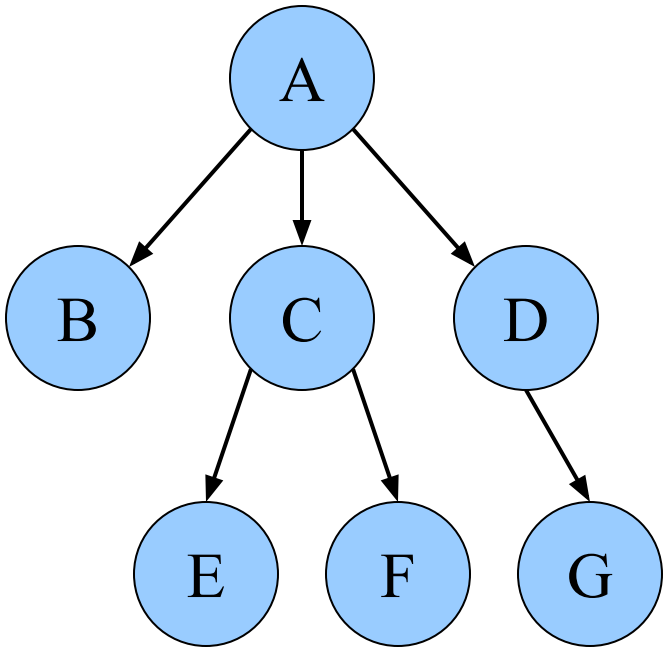


- Left: sequence as seen by hardware
  - Request sent to controller, then to disk
  - Disk responds, signals disk controller which tells interrupt controller
  - Interrupt controller notifies CPU
- Right: interrupt handling (software point of view)

Operating systems concepts

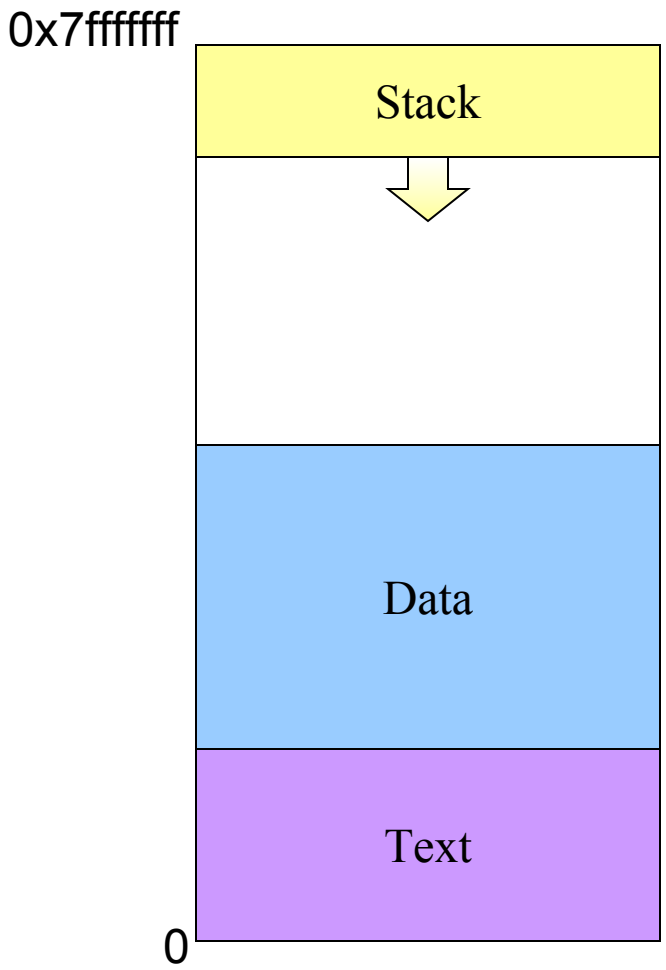
- Many of these should be familiar to Unix users…
- Processes (and trees of processes)
- Deadlock
- File systems & directory trees
- Pipes
- We'll cover all of these in more depth later on, but it's useful to have some basic definitions now

Processes



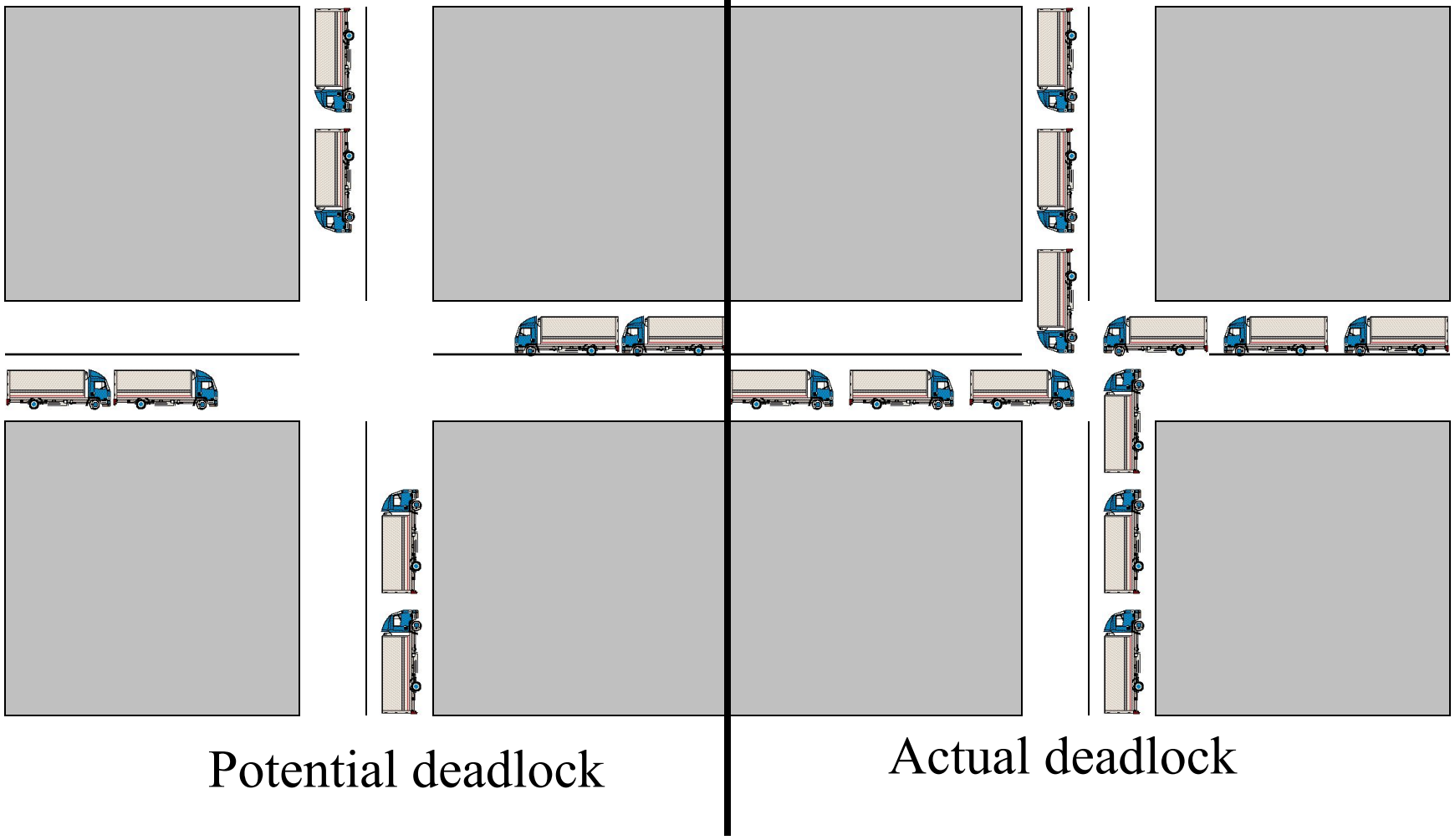
- Process: program in execution
  - Address space (memory) the program can use
  - State (registers, including program counter & stack pointer)
- OS keeps track of all processes in a *process table*
- Processes can create other processes
  - Process tree tracks these relationships
  - A is the *root* of the tree
  - A created three child processes: B, C, and D
  - C created two child processes: E and F
  - D created one child process: G

Inside a (Unix) process

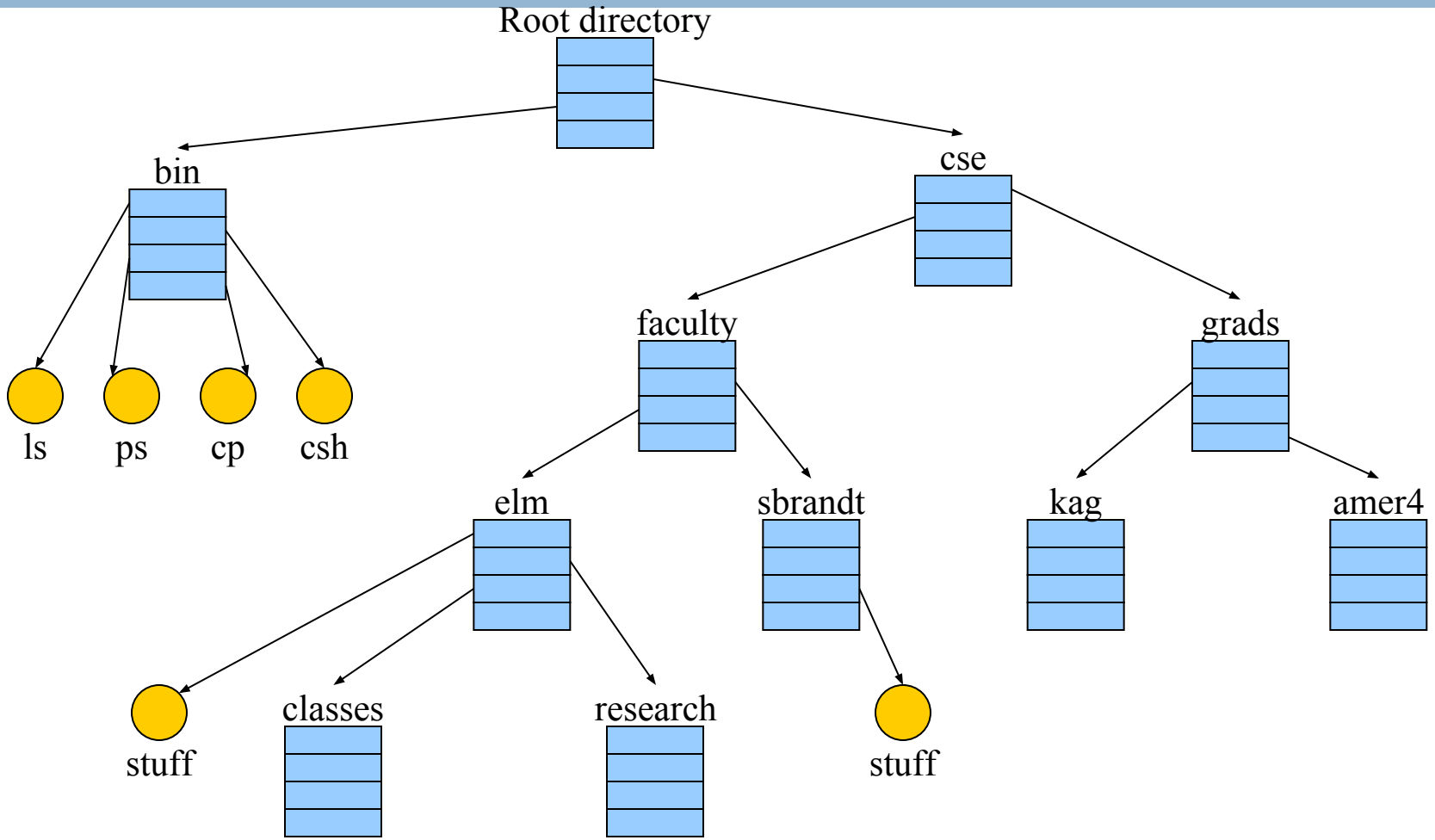


- Processes have three segments
  - Text: program code
  - Data: program data
    - Statically declared variables
    - Areas allocated by malloc() or new
  - Stack
    - Automatic variables
    - Procedure call information
- Address space growth
  - Text: doesn't grow
  - Data: grows "up"
  - Stack: grows "down"

Deadlock

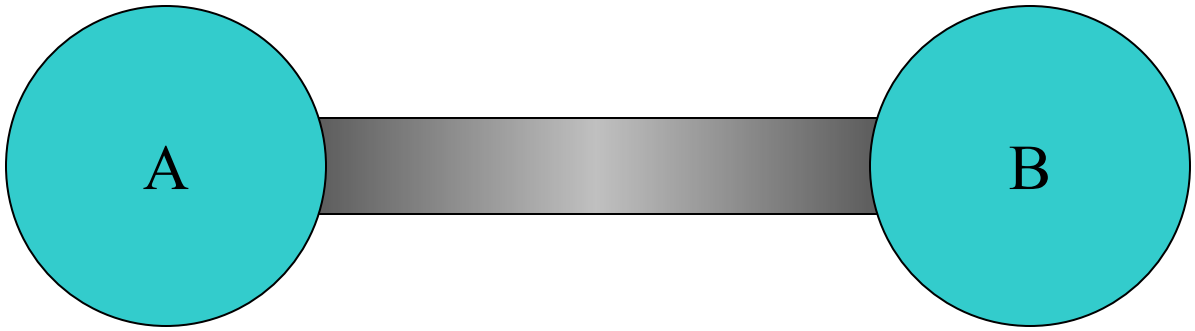


Hierarchical file systems



Interprocess communication

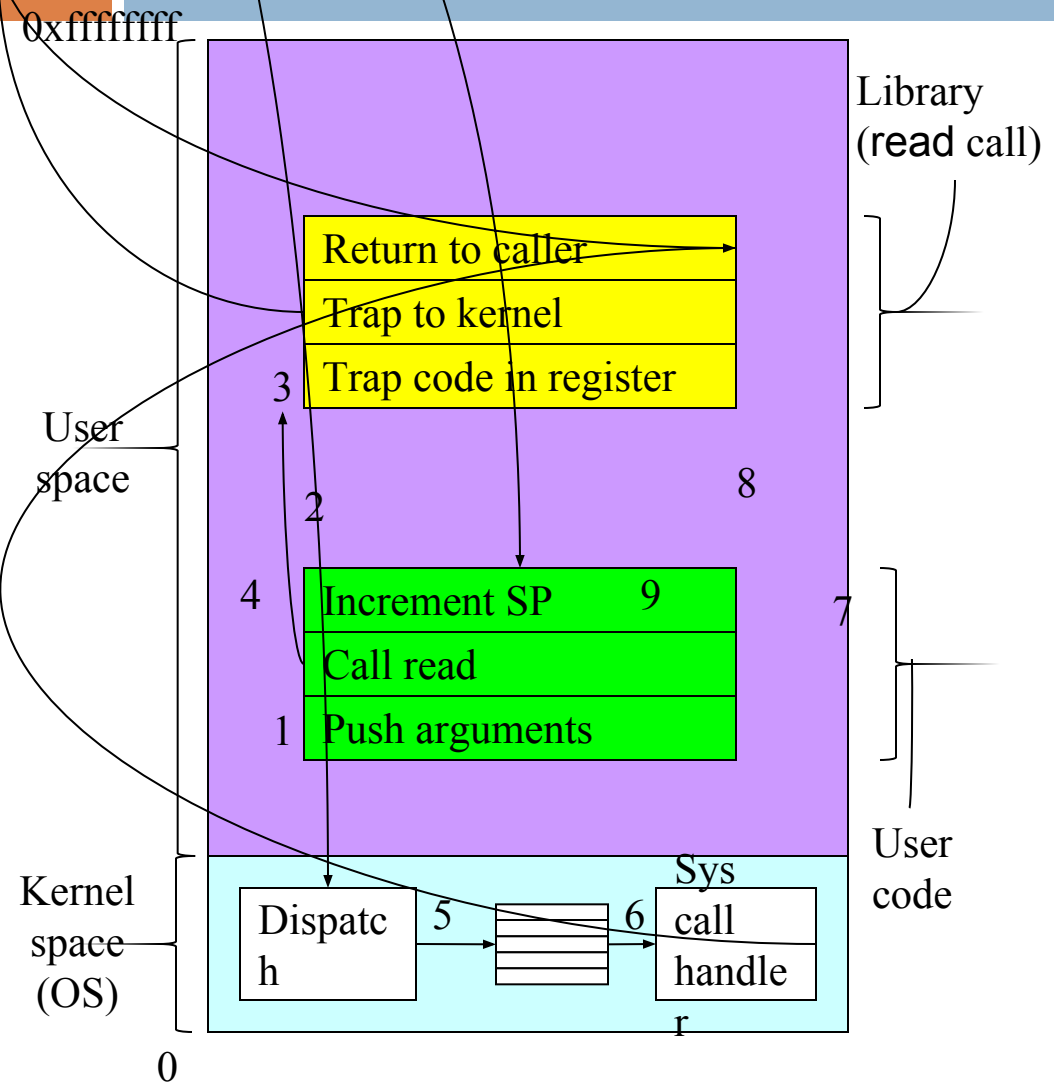
- Processes want to exchange information with each other
- Many ways to do this, including
  - Network
  - Pipe (special file): A writes into pipe, and B reads from it



System calls

- Programs want the OS to perform a service
  - Access a file
  - Create a process
  - Others…
- Accomplished by system call
  - Program passes relevant information to OS
  - OS performs the service if
    - The OS is able to do so
    - The service is permitted for this program at this time
  - OS checks information passed to make sure it's OK
    - Don't want programs reading data into other programs' memory!

Making a system call



- System call:
  read(fd,buffer,length)
- Program pushes arguments, calls library
- Library sets up trap, calls OS
- OS handles system call
- Control returns to library
- Library returns to user program

System calls for files & directories

Call	Description
fd = open(name,how)	Open a file for reading and/or writing
s = close(fd)	Close an open file
n = read(fd,buffer,size)	Read data from a file into a buffer
n = write(fd,buffer,size)	Write data from a buffer into a file
s = lseek(fd,offset,whence)	Move the "current" pointer for a file
s = stat(name,&buffer)	Get a file's status information (in *buffer*)
s = mkdir(name,mode)	Create a new directory
s = rmdir(name)	Remove a directory (must be empty)
s = link(name1,name2)	Create a new entry (*name2*) that points to the same object as *name1*
s = unlink(name)	Remove *name* as a link to an object (deletes the object if *name* was the only link to it)

More system calls

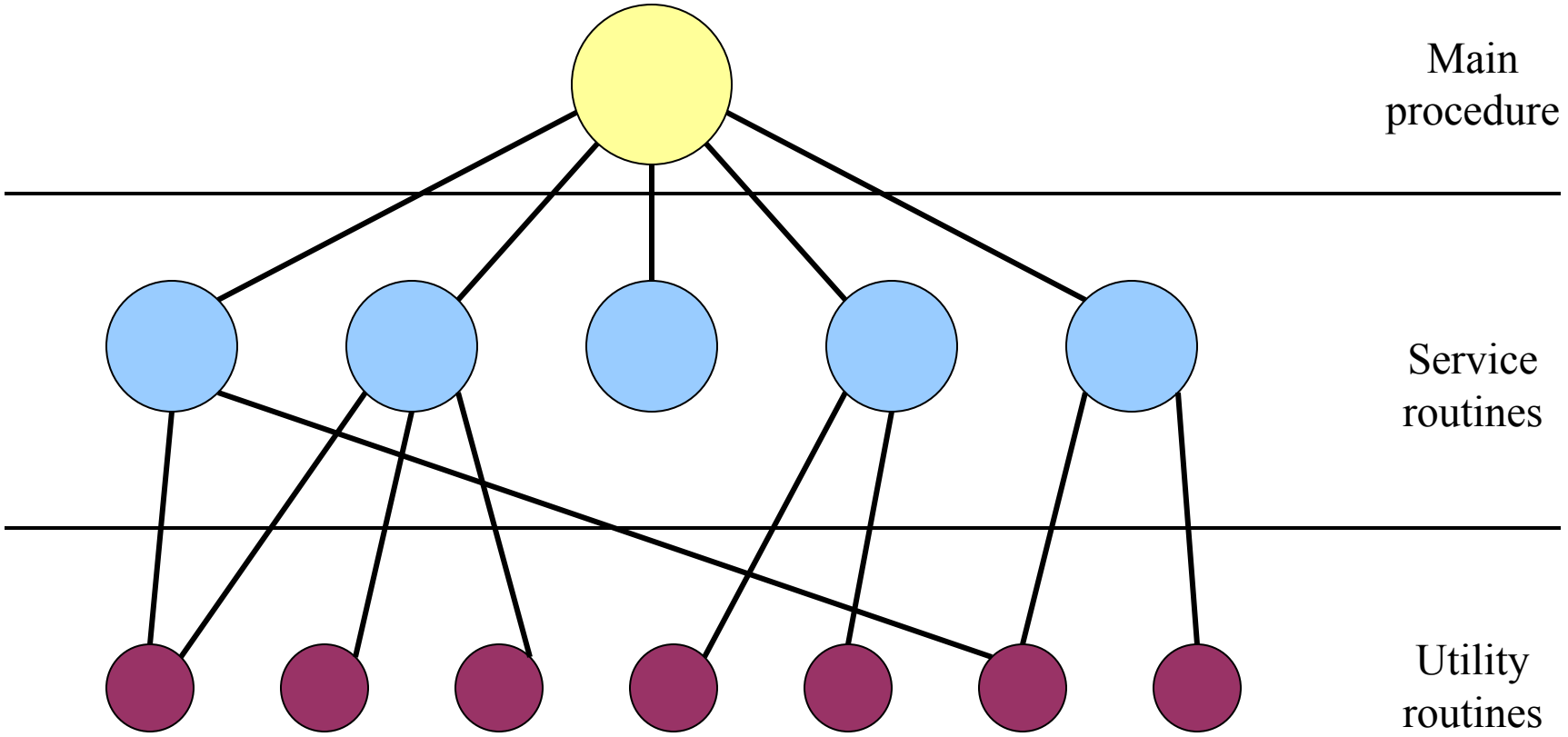
Call	Description
pid = fork()	Create a child process identical to the parent
pid=waitpid(pid,&statloc,options)	Wait for a child to terminate
s = execve(name,argv,environp)	Replace a process' core image
exit(status)	Terminate process execution and return status
s = chdir(dirname)	Change the working directory
s = chmod(name,mode)	Change a file's protection bits
s = kill(pid,signal)	Send a signal to a process
seconds = time(&seconds)	Get the elapsed time since 1 Jan 1970

A simple shell

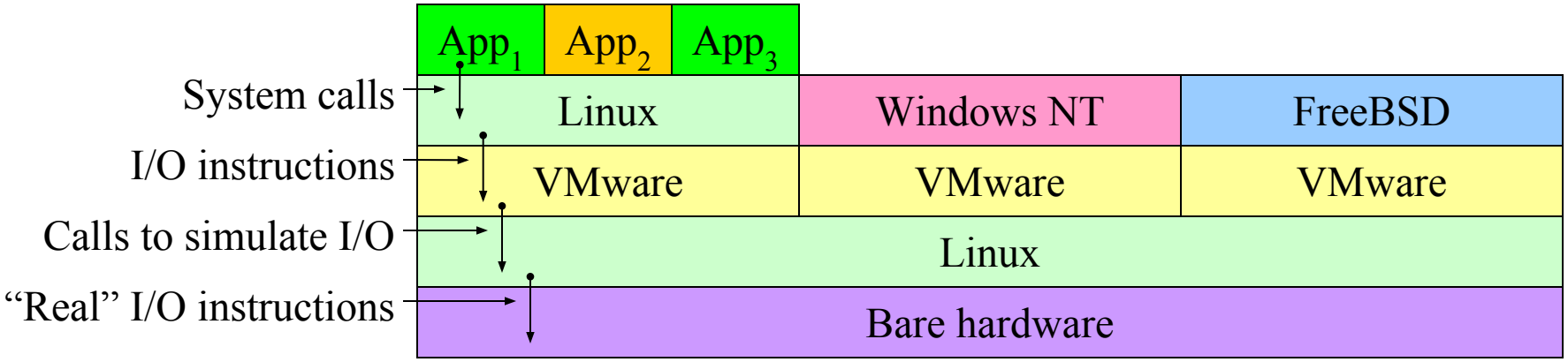
```
while (TRUE) {                          /* repeat forever */
  type_prompt( );                       /* display prompt */
  read_command (command, parameters)      /* input from terminal */

if (fork() != 0) {          /* fork off child process */
  /* Parent code */
  waitpid( -1, &status, 0);  /* wait for child to exit */
} else {
  /* Child code */
  execve (command, parameters, 0);       /* execute command */
 }
}
```

Monolithic OS structure

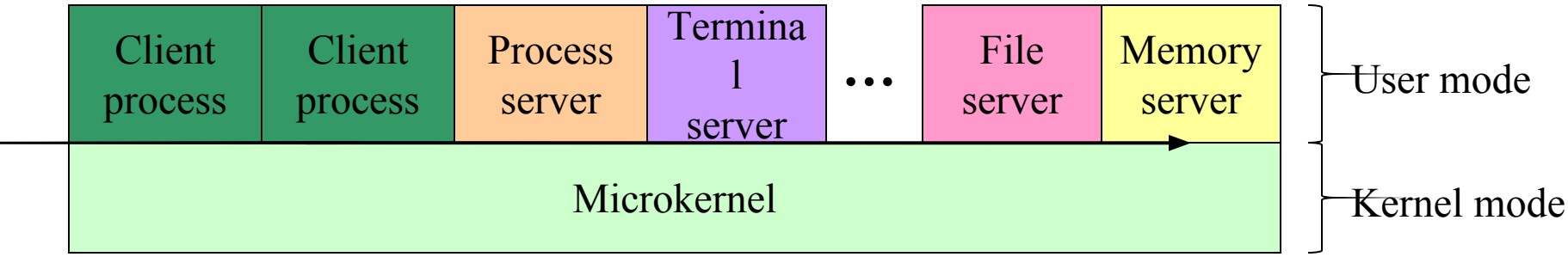


Virtual machines



- First widely used in VM/370 with CMS
- Available today in VMware
  - Allows users to run any x86-based OS on top of Linux or NT
- "Guest" OS can crash without harming underlying OS
  - Only virtual machine fails—rest of underlying OS is fine
- "Guest" OS can even use raw hardware
  - Virtual machine keeps things separated

Microkernels (client-server)



- Processes (clients and OS servers) don't share memory
  - Communication via message-passing
  - Separation reduces risk of "byzantine" failures
- Examples include Mach

Memory Management

Memory management

- Basic memory management
- Swapping
- Virtual memory
- Page replacement algorithms
- Modeling page replacement algorithms
- Design issues for paging systems
- Implementation issues
- Segmentation

In an ideal world…

- The ideal world has memory that is
  - Very large
  - Very fast
  - Non-volatile (doesn't go away when power is turned off)
- The real world has memory that is:
  - Very large
  - Very fast
  - Affordable!

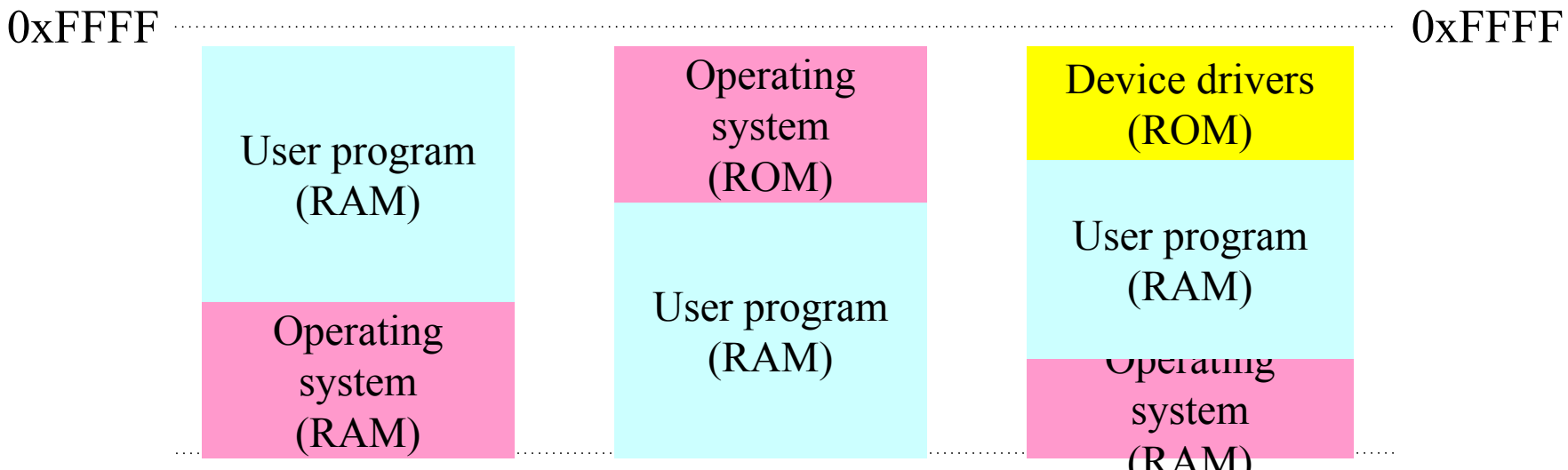
  ⇒ Pick any two…
- Memory management goal: make the real world look as much like the ideal world as possible

Memory hierarchy

- What is the memory hierarchy?
  - Different levels of memory
  - Some are small & fast
  - Others are large & slow
- What levels are usually included?
  - Cache: small amount of fast, expensive memory
    - L1 (level 1) cache: usually on the CPU chip
    - L2 & L3 cache: off-chip, made of SRAM
  - Main memory: medium-speed, medium price memory (DRAM)
  - Disk: many gigabytes of slow, cheap, non-volatile storage
- Memory manager handles the memory hierarchy

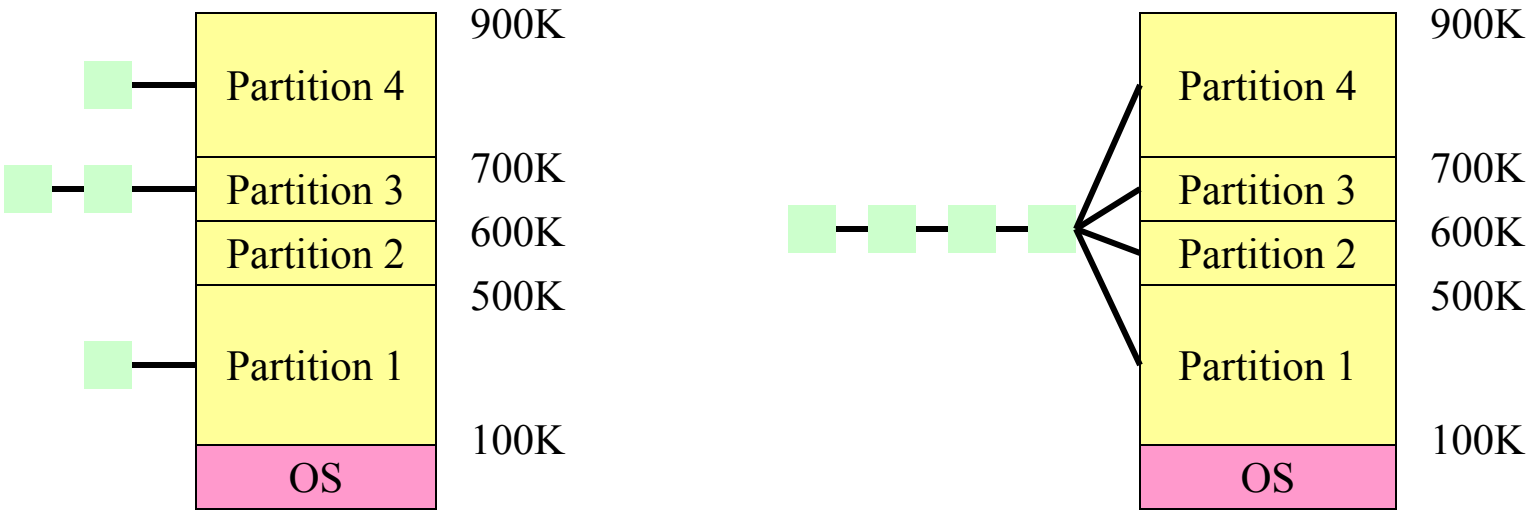
Basic memory management

- Components include
  - Operating system (perhaps with device drivers)
  - Single process
- Goal: lay these out in memory
  - Memory protection may not be an issue (only one program)
  - Flexibility may still be useful (allow OS changes, etc.)
- No swapping or paging



Fixed partitions: multiple programs

- Fixed memory partitions
  - Divide memory into fixed spaces
  - Assign a process to a space when it's free
- Mechanisms
  - Separate input queues for each partition
  - Single input queue: better ability to optimize CPU usage



How many programs is enough?

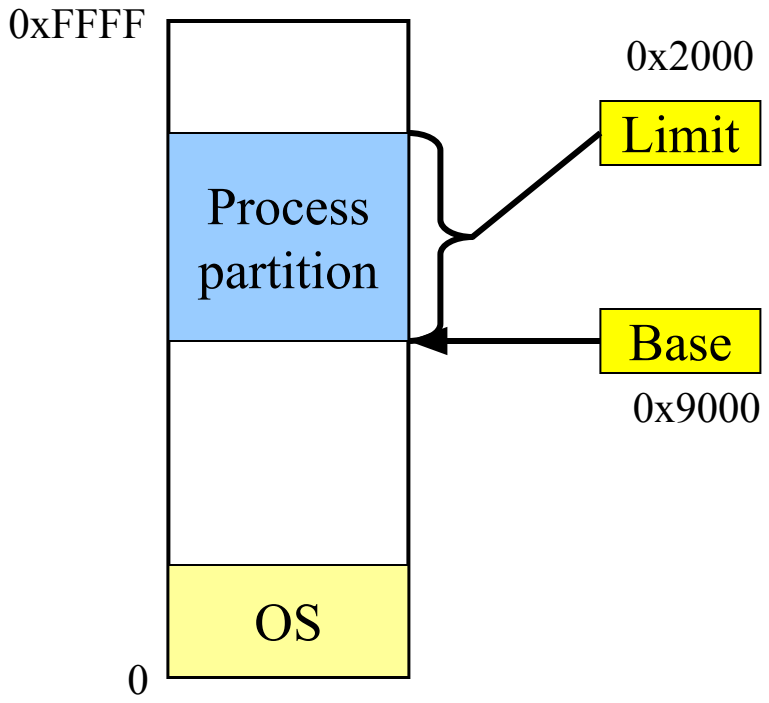
- Several memory partitions (fixed or variable size)
- Lots of processes wanting to use the CPU
- Tradeoff
  - More processes utilize the CPU better
  - Fewer processes use less memory (cheaper!)
- How many processes do we need to keep the CPU fully utilized?
  - This will help determine how much memory we need
  - Is this still relevant with memory costing less than $1/GB?

Memory and multiprogramming

- Memory needs two things for multiprogramming
  - Relocation
  - Protection
- The OS cannot be certain where a program will be loaded in memory
  - Variables and procedures can’t use absolute locations in memory
  - Several ways to guarantee this
- The OS must keep processes’ memory separate
  - Protect a process from other processes reading or modifying its own memory
  - Protect a process from modifying its own memory in undesirable ways (such as writing to program code)

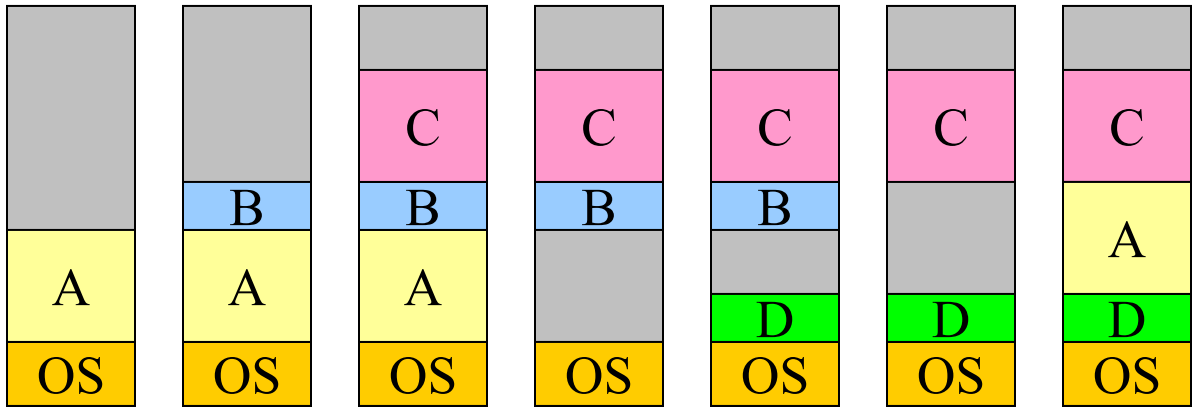
Base and limit registers

- Special CPU registers: base & limit
  - Access to the registers limited to system mode
  - Registers contain
    - Base: start of the process's memory partition
    - Limit: length of the process's memory partition
- Address generation
  - Physical address: location in actual memory
  - Logical address: location from the process's point of view
  - Physical address = base + logical address
  - Logical address larger than limit => error



Logical address: 0x1204
Physical address:
0x1204+0x9000 = 0xa204

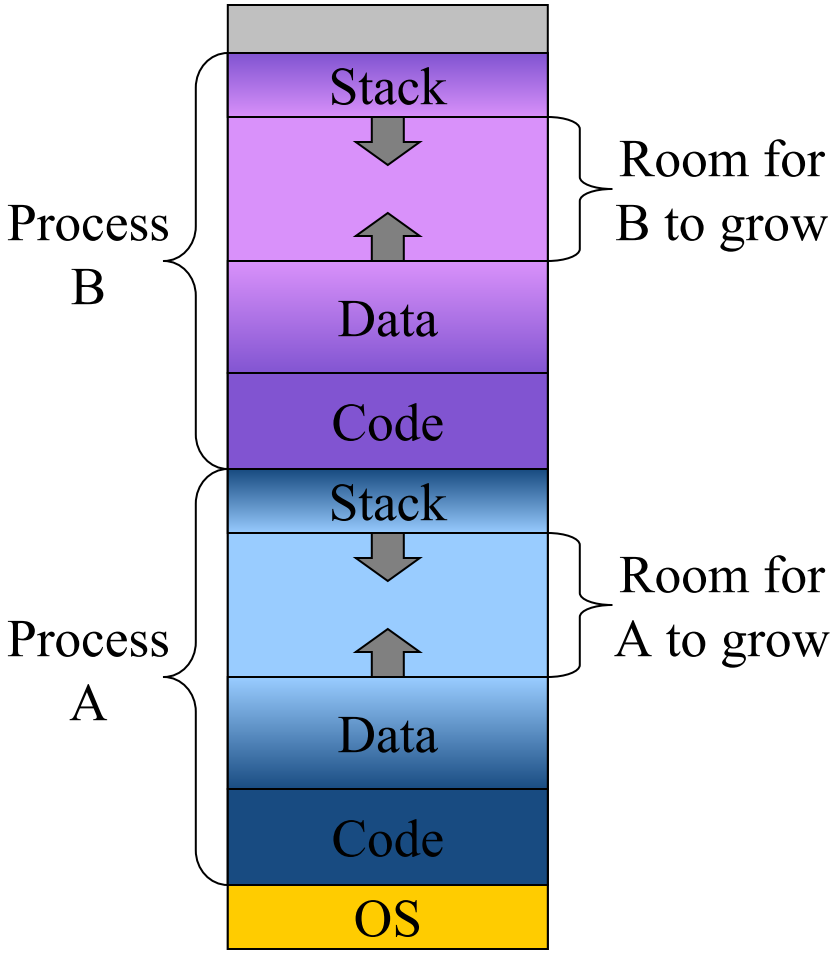
Swapping



- Memory allocation changes as
  - Processes come into memory
  - Processes leave memory
    - Swapped to disk
    - Complete execution
- Gray regions are unused memory

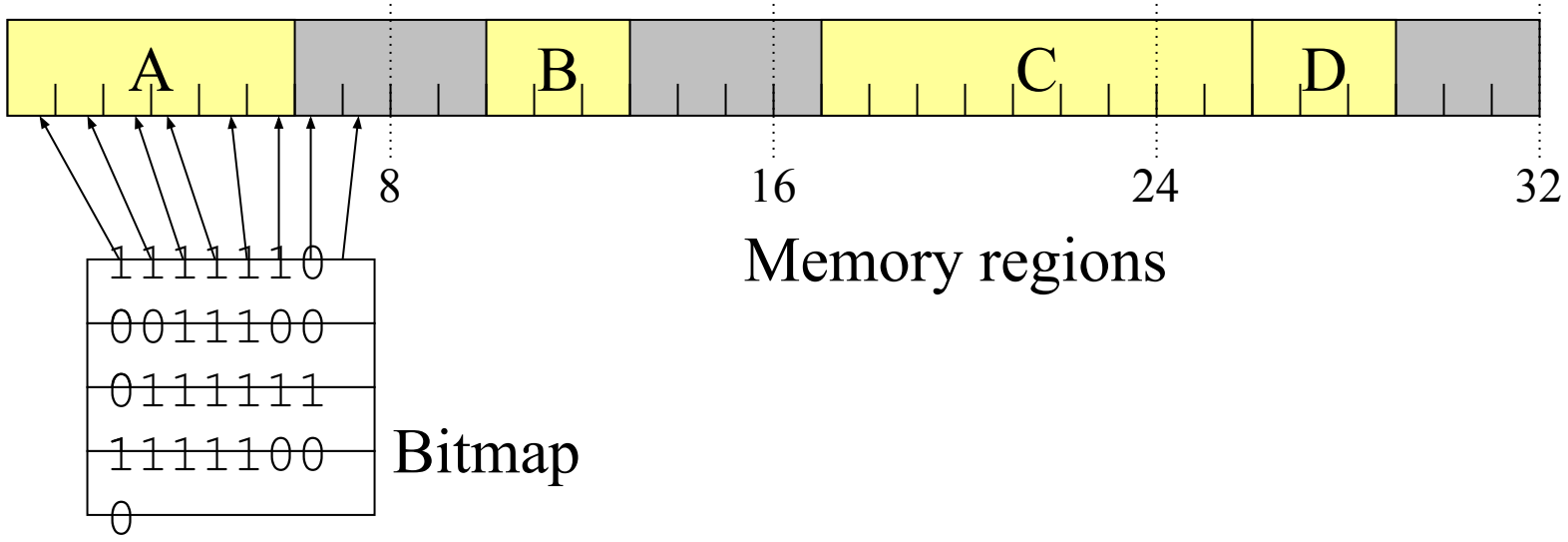
Swapping: leaving room to grow

- Need to allow for programs to grow
  - Allocate more memory for data
  - Larger stack
- Handled by allocating more space than is necessary at the start
  - Inefficient: wastes memory that's not currently in use
  - What if the process requests too much memory?



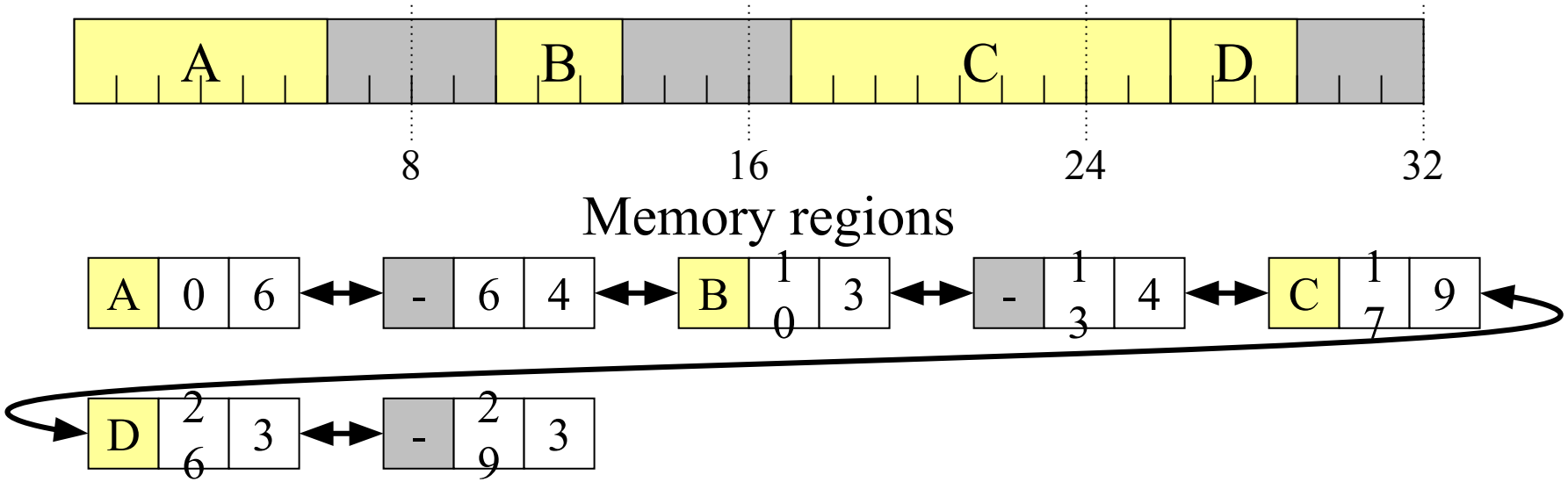
Tracking memory usage: bitmaps

- Keep track of free / allocated memory regions with a bitmap
  - One bit in map corresponds to a fixed-size region of memory
  - Bitmap is a constant size for a given amount of memory regardless of how much is allocated at a particular time
- Chunk size determines efficiency
  - At 1 bit per 4KB chunk, we need just 256 bits (32 bytes) per MB of memory
  - For smaller chunks, we need more memory for the bitmap
  - Can be difficult to find large contiguous free areas in bitmap



Tracking memory usage: linked lists

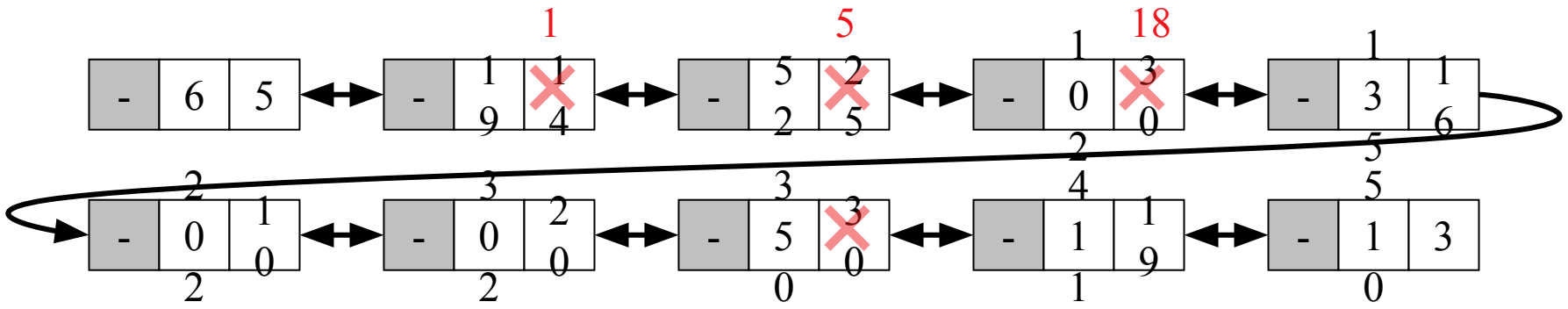
- Keep track of free / allocated memory regions with a linked list
  - Each entry in the list corresponds to a contiguous region of memory
  - Entry can indicate either allocated or free (and, optionally, owning process)
  - May have separate lists for free and allocated areas
- Efficient if chunks are large
  - Fixed-size representation for each region
  - More regions => more space needed for free lists



Allocating memory

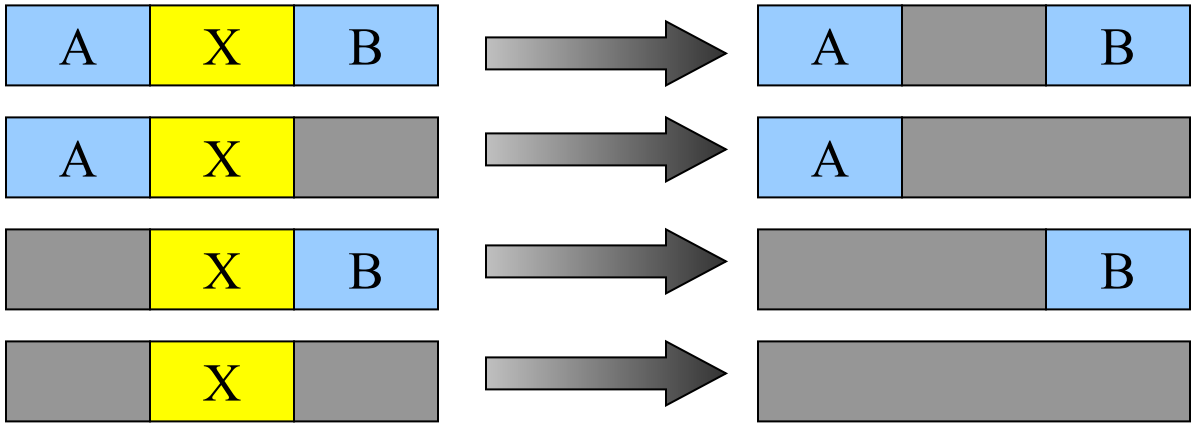
- Search through region list to find a large enough space
- Suppose there are several choices: which one to use?
  - First fit: the first suitable hole on the list
  - Next fit: the first suitable after the previously allocated hole
  - Best fit: the smallest hole that is larger than the desired region (wastes least space?)
  - Worst fit: the largest available hole (leaves largest fragment)
- Option: maintain separate queues for different-size holes

Allocate 20 blocks first fit
Allocate 12 blocks next fit
Allocate 13 blocks best fit
Allocate 15 blocks worst fit



Freeing memory

- Allocation structures must be updated when memory is freed
- Easy with bitmaps: just set the appropriate bits in the bitmap
- Linked lists: modify adjacent elements as needed
  - Merge adjacent free regions into a single region
  - May involve merging two regions with the just-freed area



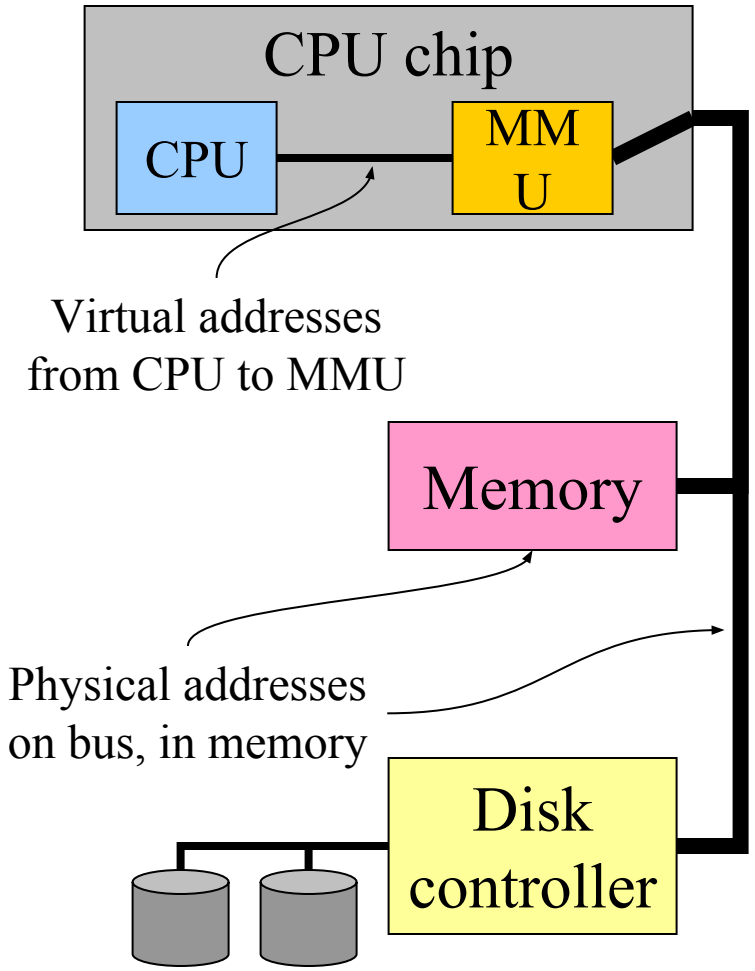
Limitations of swapping

- Problems with swapping
  - Process must fit into physical memory (impossible to run larger processes)
  - Memory becomes fragmented
    - External fragmentation: lots of small free areas
    - Compaction needed to reassemble larger free areas
  - Processes are either in memory or on disk: half and half doesn’t do any good
- Overlays solved the first problem
  - Bring in pieces of the process over time (typically data)
  - Still doesn’t solve the problem of fragmentation or partially resident processes

Virtual memory

- Basic idea: allow the OS to hand out more memory than exists on the system
- Keep recently used stuff in physical memory
- Move less recently used stuff to disk
- Keep all of this hidden from processes
  - Processes still see an address space from 0 – max address
  - Movement of information to and from disk handled by the OS without process help
- Virtual memory (VM) especially helpful in multiprogrammed system
  - CPU schedules process B while process A waits for its memory to be retrieved from disk

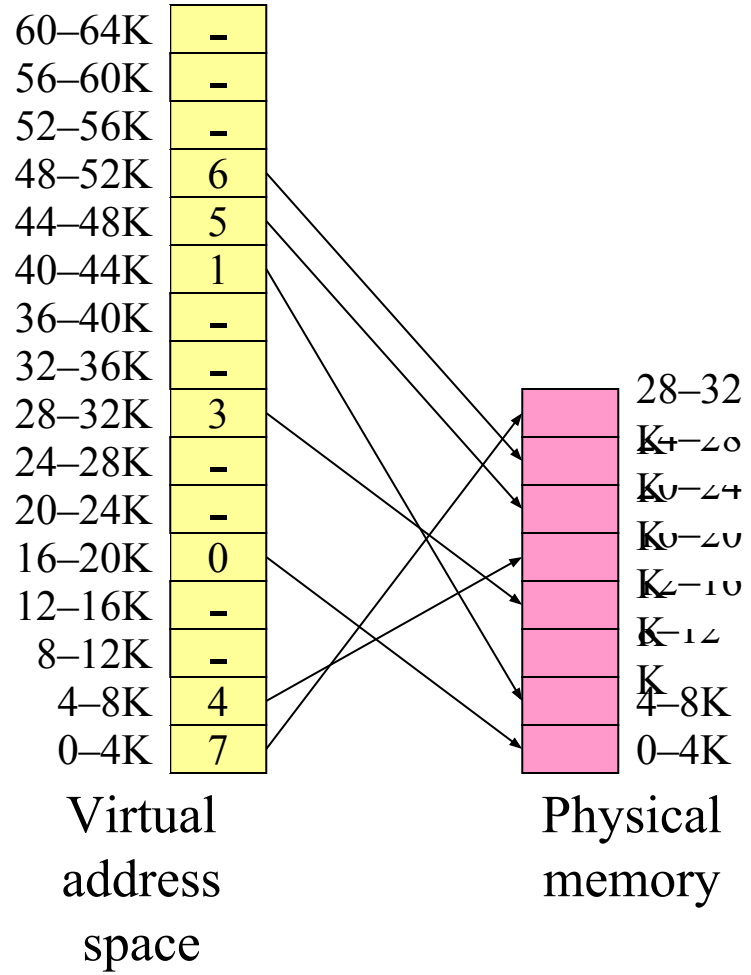
Virtual and physical addresses



- Program uses *virtual addresses*
  - Addresses local to the process
  - Hardware translates virtual address to *physical address*
- Translation done by the ***M**emory **M**anagement **U**nit*
  - Usually on the same chip as the CPU
  - Only physical addresses leave the CPU/MMU chip
- Physical memory indexed by physical addresses

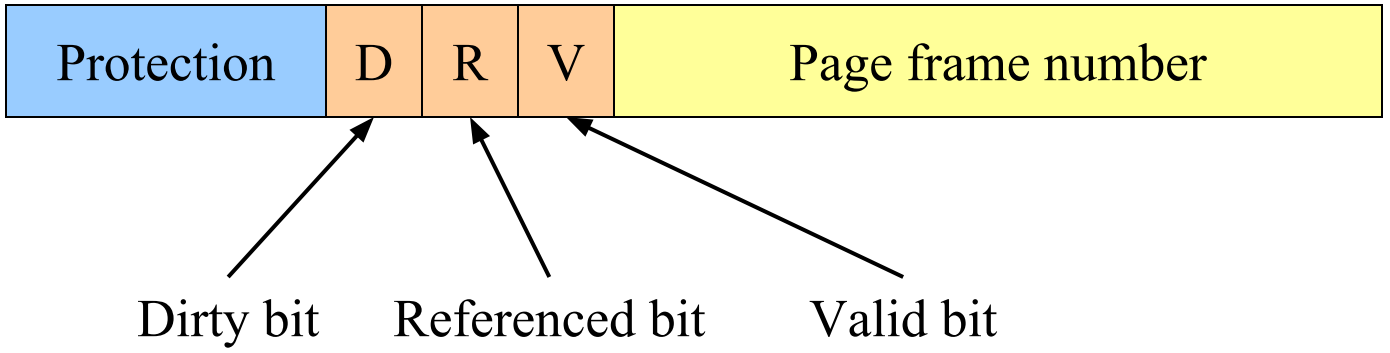
Paging and page tables

- Virtual addresses mapped to physical addresses
  - Unit of mapping is called a *page*
  - All addresses in the same virtual page are in the same physical page
  - *Page table entry* (PTE) contains translation for a single page
- Table translates virtual page number to physical page number
  - Not all virtual memory has a physical page
  - Not every physical page need be used
- Example:
  - 64 KB virtual memory
  - 32 KB physical memory



What's in a page table entry?

- Each entry in the page table contains
  - Valid bit: set if this logical page number has a corresponding physical frame in memory
    - If not valid, remainder of PTE is irrelevant
  - Page frame number: page in physical memory
  - Referenced bit: set if data on the page has been accessed
  - Dirty (modified) bit :set if data on the page has been modified
  - Protection information

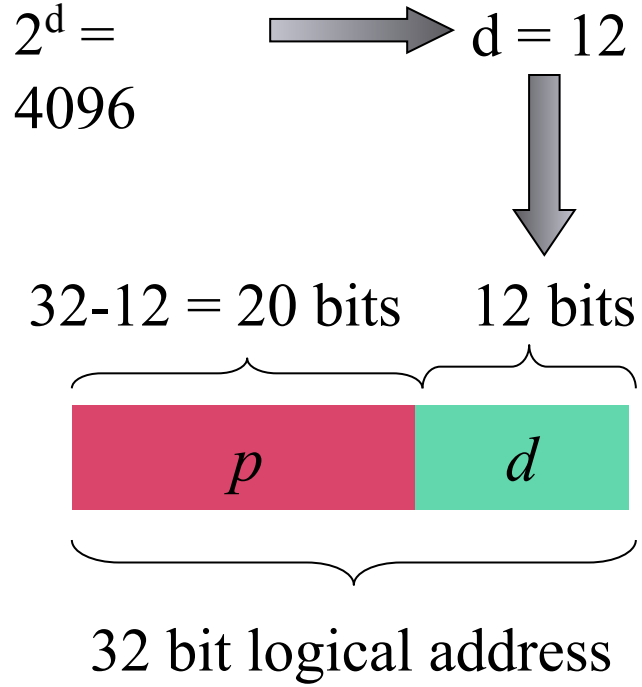


Mapping logical => physical address

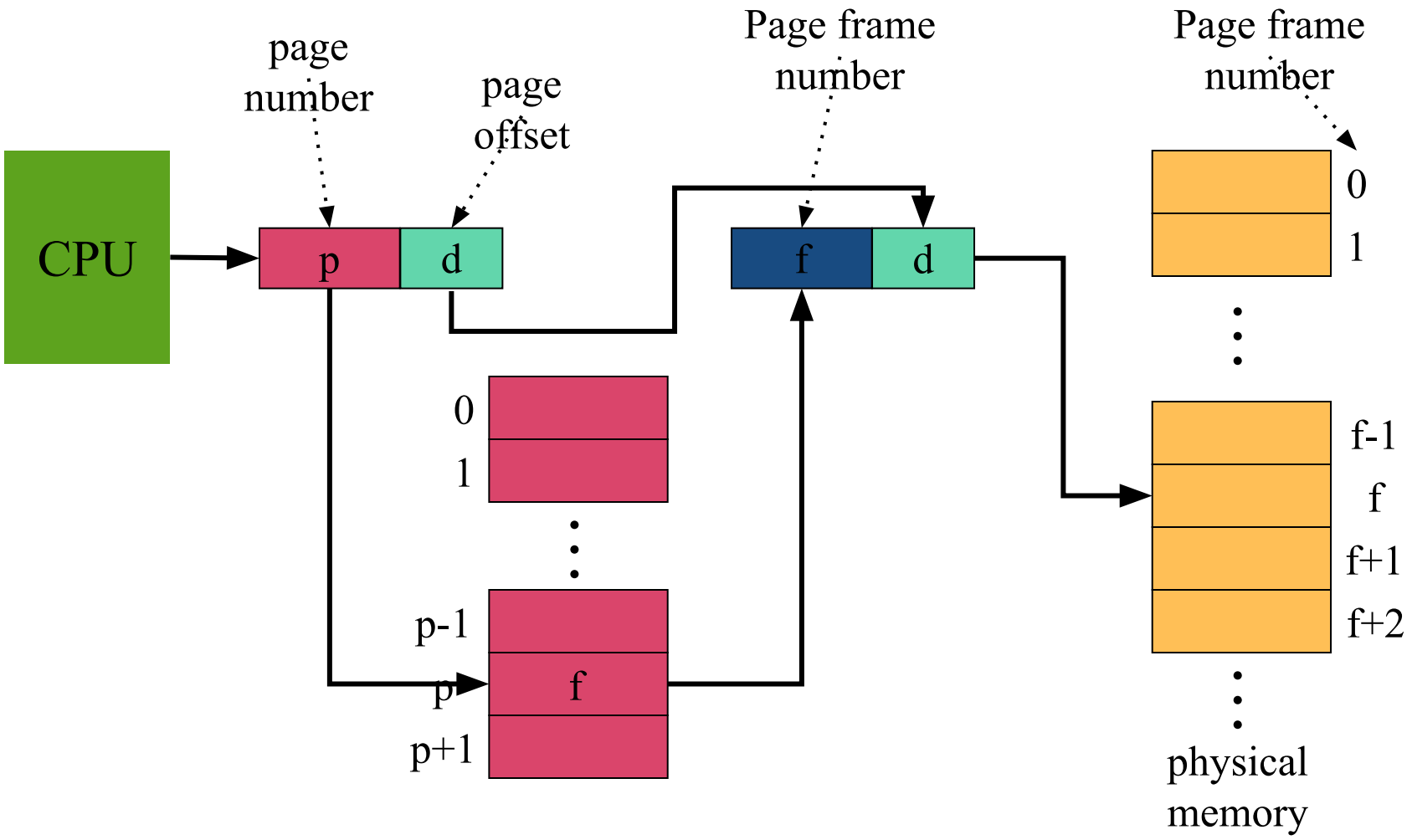
- Split address from CPU into two pieces
  - Page number (*p*)
  - Page offset (*d*)
- Page number
  - Index into page table
  - Page table contains base address of page in physical memory
- Page offset
  - Added to base address to get actual physical memory address
- Page size = 2^d bytes

Example:

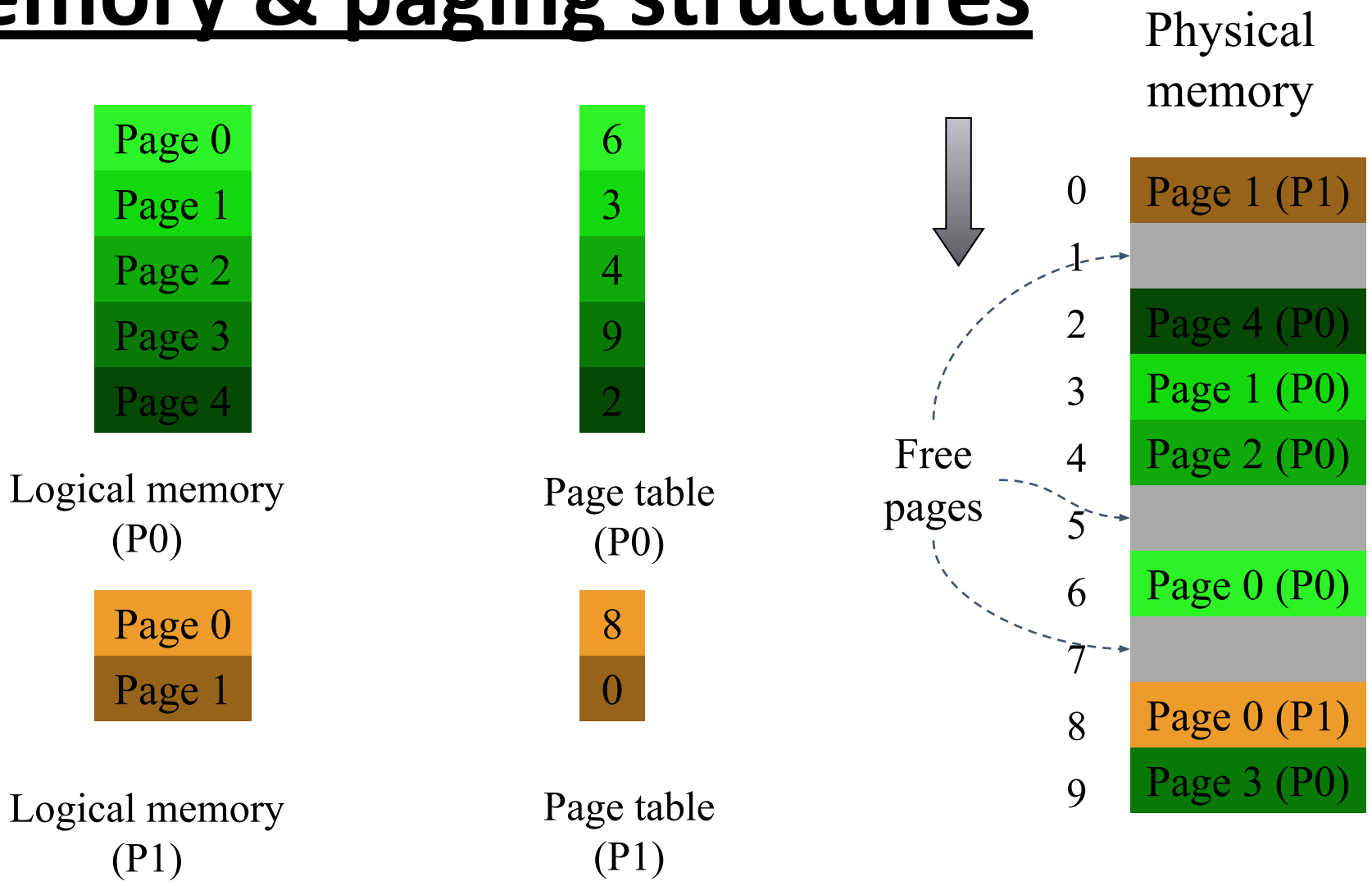
- 4 KB (=4096 byte) pages
- 32 bit logical addresses



Address translation architecture

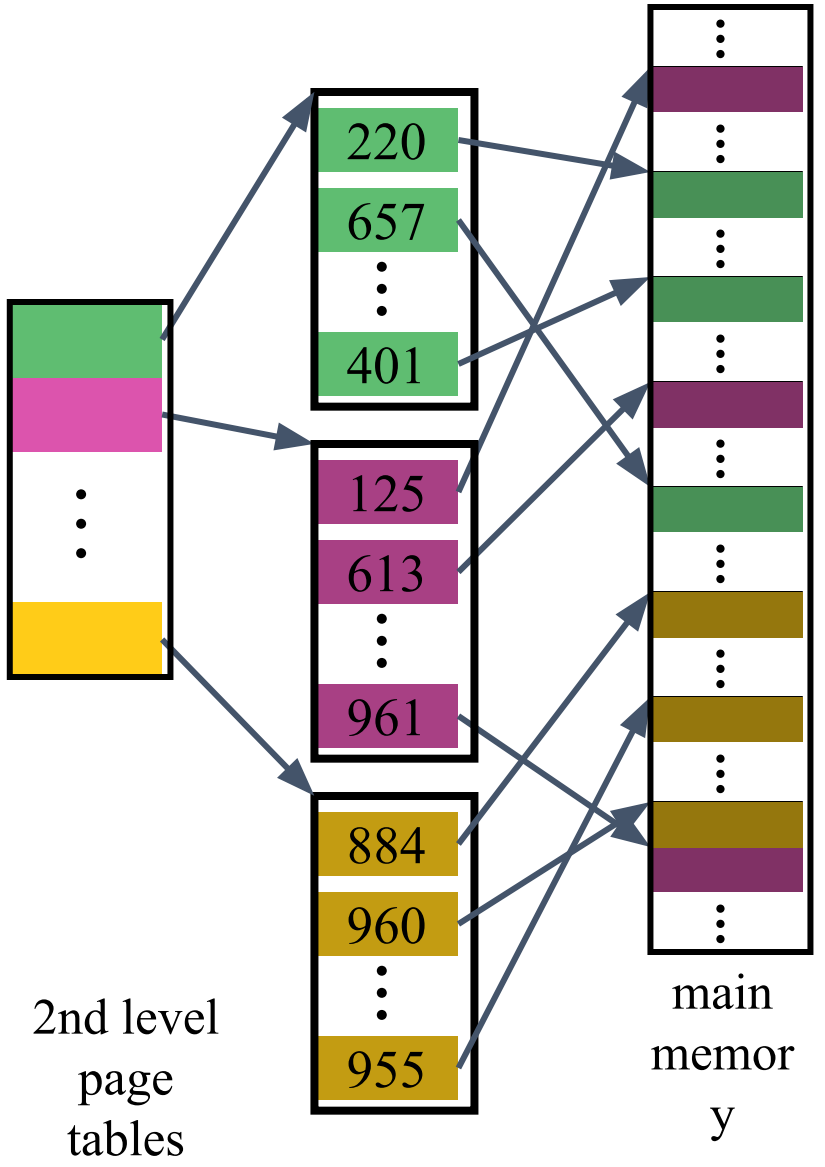


Memory & paging structures



Two-level page tables

- Problem: page tables can be too large
  - 2^{32} bytes in 4KB pages need 1 million PTEs
- Solution: use multi-level page tables
  - "Page size" in first page table is large (megabytes)
  - PTE marked invalid in first page table needs no 2nd level page table
- 1st level page table has pointers to 2nd level page tables
- 2nd level page table has actual physical page numbers in it

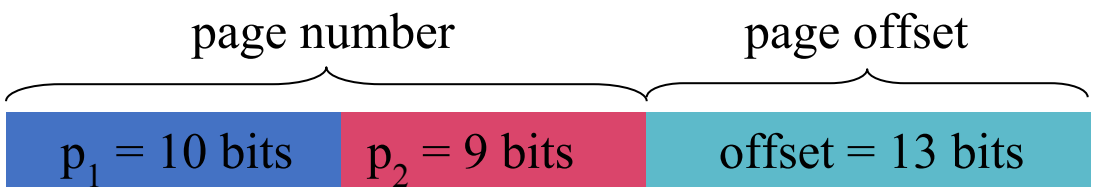


More on two-level page tables

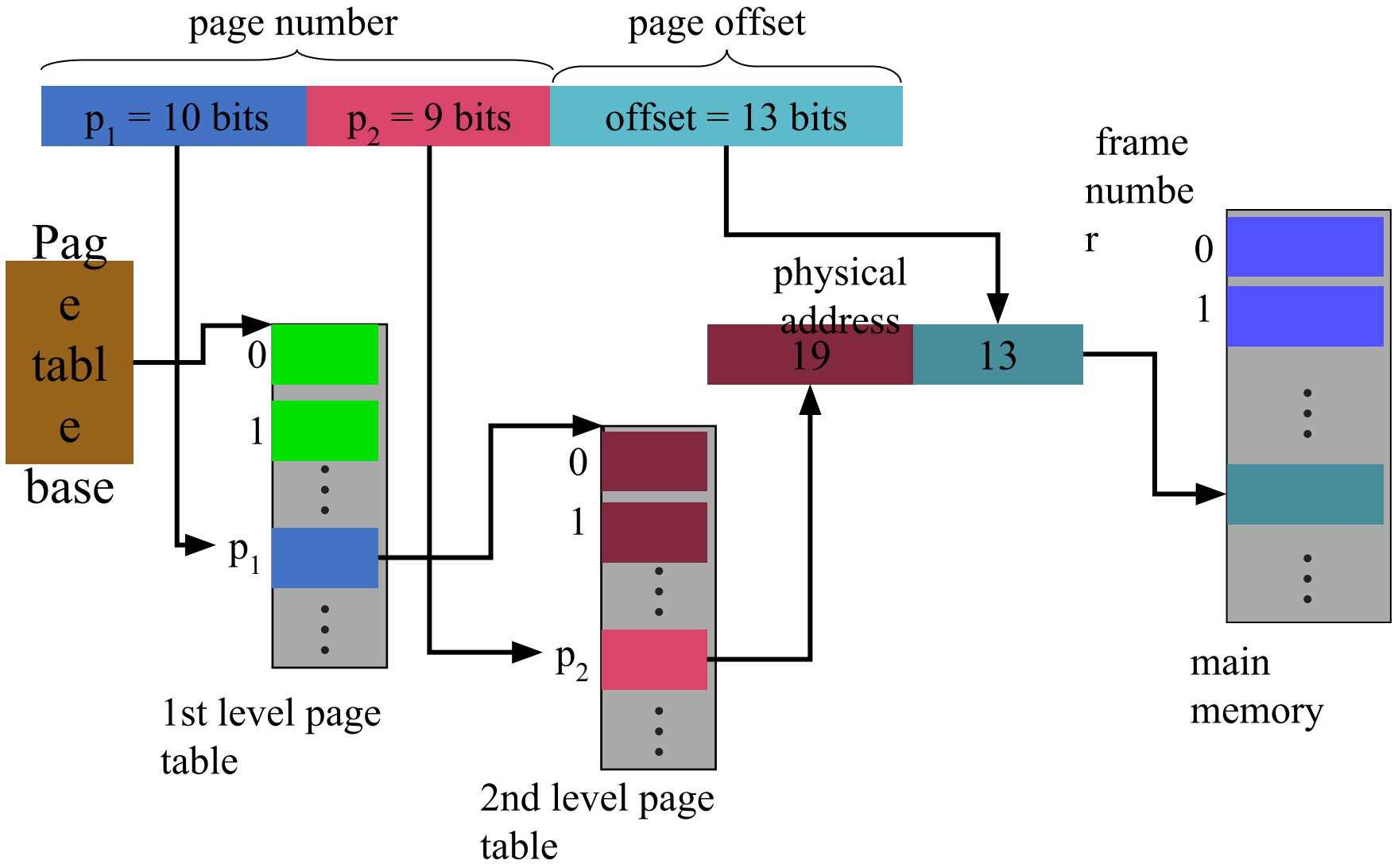
- Tradeoffs between 1st and 2nd level page table sizes
  - Total number of bits indexing 1st and 2nd level table is constant for a given page size and logical address length
  - Tradeoff between number of bits indexing 1st and number indexing 2nd level tables
    - More bits in 1st level: fine granularity at 2nd level
    - Fewer bits in 1st level: maybe less wasted space?
- All addresses in table are physical addresses
- Protection bits kept in 2nd level table

Two-level paging: example

- System characteristics
  - 8 KB pages
  - 32-bit logical address divided into 13 bit page offset, 19 bit page number
- Page number divided into:
  - 10 bit page number
  - 9 bit page offset
- Logical address looks like this:
  - p_1 is an index into the 1st level page table
  - p_2 is an index into the 2nd level page table pointed to by p_1



2-level address translation example

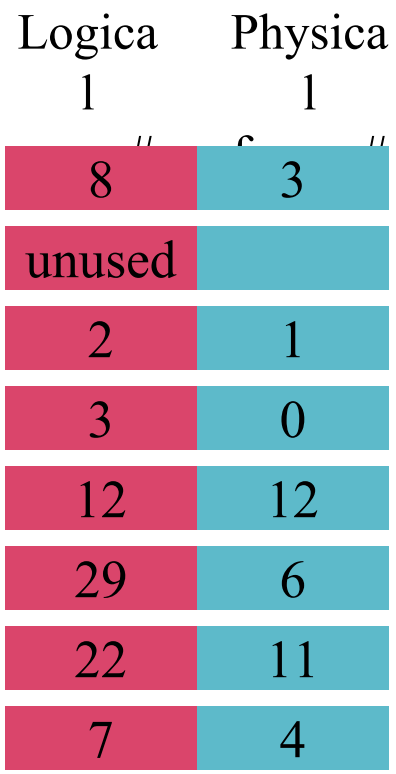


Implementing page tables in hardware

- Page table resides in main (physical) memory
- CPU uses special registers for paging
  - Page table base register (PTBR) points to the page table
  - Page table length register (PTLR) contains length of page table: restricts maximum legal logical address
- Translating an address requires two memory accesses
  - First access reads page table entry (PTE)
  - Second access reads the data / instruction from memory
- Reduce number of memory accesses
  - Can’t avoid second access (we need the value from memory)
  - Eliminate first access by keeping a hardware cache (called a *translation lookaside buffer* or TLB) of recently used page table entries

Translation Lookaside Buffer (TLB)

- Search the TLB for the desired logical page number
  - Search entries in parallel
  - Use standard cache techniques
- If desired logical page number is found, get frame number from TLB
- If desired logical page number isn't found
  - Get frame number from page table in memory
  - Replace an entry in the TLB with the logical & physical page numbers from this reference



Example TLB

Handling TLB misses

- If PTE isn't found in TLB, OS needs to do the lookup in the page table
- Lookup can be done in hardware or software
- Hardware TLB replacement
  - CPU hardware does page table lookup
  - Can be faster than software
  - Less flexible than software, and more complex hardware
- Software TLB replacement
  - OS gets TLB exception
  - Exception handler does page table lookup & places the result into the TLB
  - Program continues after return from exception
  - Larger TLB (lower miss rate) can make this feasible

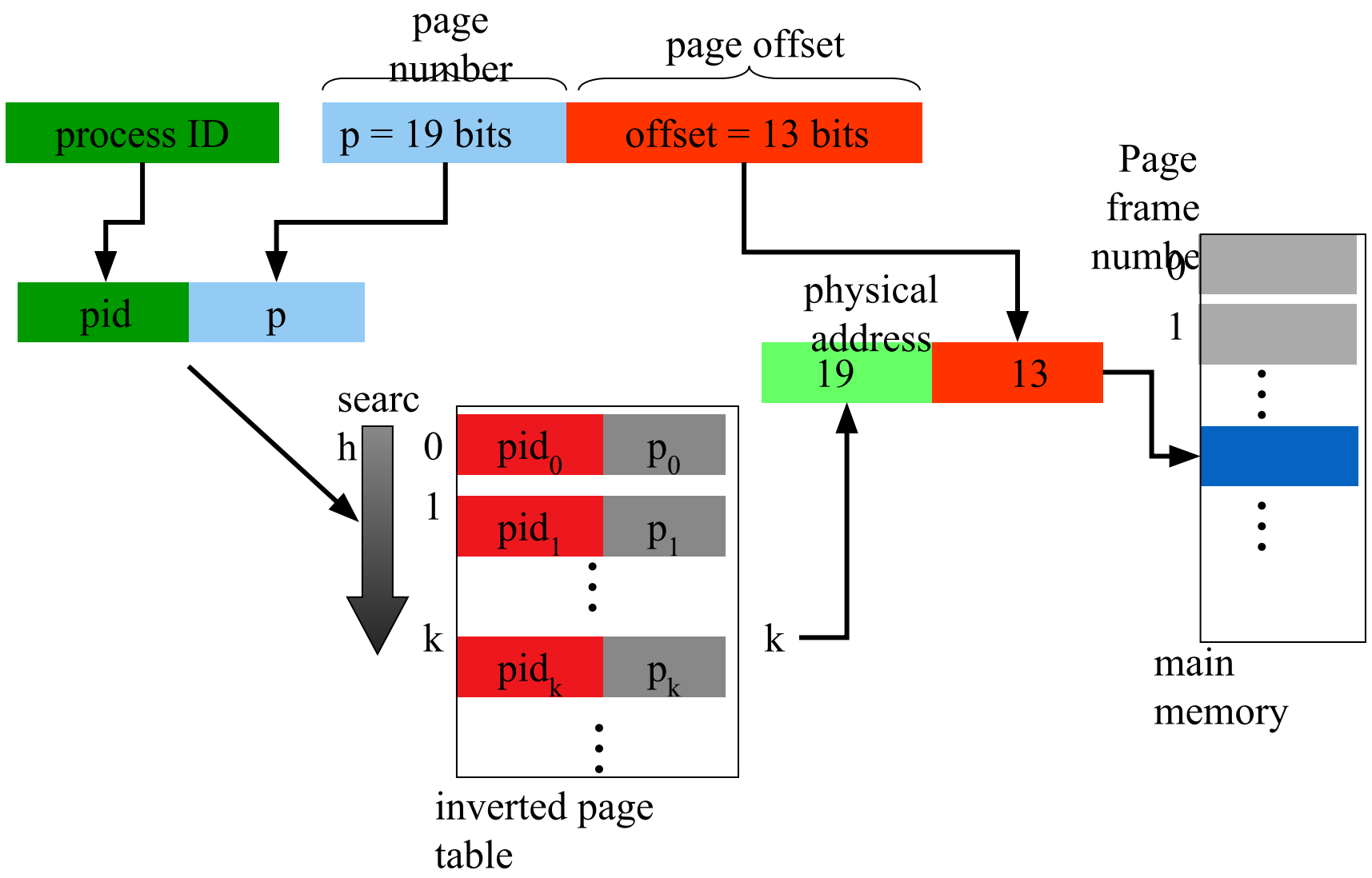
How long do memory accesses take?

- Assume the following times:
  - TLB lookup time = a (often zero - overlapped in CPU)
  - Memory access time = m
- Hit ratio (h) is percentage of time that a logical page number is found in the TLB
  - Larger TLB usually means higher h
  - TLB structure can affect h as well
- Effective access time (an average) is calculated as:
  - EAT = (m + a)h + (m + m + a)(1-h)
  - EAT =a + (2-h)m
- Interpretation
  - Reference always requires TLB lookup, 1 memory access
  - TLB misses also require an additional memory reference

Inverted page table

- Reduce page table size further: keep one entry for each frame in memory
- PTE contains
  - Virtual address pointing to this frame
  - Information about the process that owns this page
- Search page table by
  - Hashing the virtual page number and process ID
  - Starting at the entry corresponding to the hash result
  - Search until either the entry is found or a limit is reached
- Page frame number is index of PTE
- Improve performance by using more advanced hashing algorithms

Inverted page table architecture



PAGE REPLACEMENT ALGORITHMS

VIRTUAL MEMORY

- BASE AND LIMIT REGISTERS CAN BE USED TO CREATE THE ABSTRACTION OF ADDRESS SPACES, THERE IS ANOTHER PROBLEM THAT HAS TO BE SOLVED: MANAGING BLOATWARE
- MEMORY SIZES ARE INCREASING RAPIDLY, SOFTWARE SIZES ARE INCREASING MUCH FASTER
- AS A CONSEQUENCE OF THESE DEVELOPMENTS, THERE IS A NEED TO RUN PROGRAMS THAT ARE TOO LARGE TO FIT IN MEMORY, AND THERE IS CERTAINLY A NEED TO HAVE SYSTEMS THAT CAN SUPPORT MULTIPLE PROGRAMS RUNNING SIMULTANEOUSLY, EACH OF WHICH FITS IN MEMORY BUT ALL OF WHICH COLLECTIVELY EXCEED MEMORY
- A SOLUTION ADOPTED IN THE 1960S WAS TO SPLIT PROGRAMS INTO LITTLE PIECES, CALLED OVERLAYS. WHEN A PROGRAM STARTED, ALL THAT WAS LOADED INTO MEMORY WAS THE OVERLAY MANAGER, WHICH IMMEDIATELY LOADED AND RAN OVERLAY 0. WHEN IT WAS DONE, IT WOULD TELL THE OVERLAY MANAGER TO LOAD OVERLAY 1, EITHER ABOVE OVERLAY 0 IN MEM ORY (IF THERE WAS SPACE FOR IT) OR ON TOP OF OVERLAY 0 (IF THERE WAS NO SPACE).
- THE METHOD THAT WAS DEVISED (FOTHERINGHAM, 1961) HAS COME TO BE KNOWN AS VIRTUAL MEMORY. THE BASIC IDEA BEHIND VIRTUAL MEMORY IS THAT EACH PROGRAM HAS ITS OWN ADDRESS SPACE, WHICH IS BROKEN UP INTO CHUNKS CALLED PAGES. EACH PAGE IS A CONTIGUOUS RANGE OF ADDRESSES. THESE PAGES ARE MAPPED ONTO PHYSICAL MEMORY, BUT NOT ALL PAGES HAVE TO BE IN PHYSICAL MEMORY AT THE SAME TIME TO RUN THE PRO GRAM.

PAGING

- THESE PROGRAM-GENERATED ADDRESSES ARE CALLED VIRTUAL ADDRESSES AND FORM THE VIRTUAL ADDRESS SPACE. ON COMPUTERS WITHOUT VIRTUAL MEMORY, THE VIRTUAL ADDRESS IS PUT DIRECTLY ONTO THE MEMORY BUS AND CAUSES THE PHYSICAL MEMORY WORD WITH THE SAME ADDRESS TO BE READ OR WRITTEN.
- WHEN VIRTUAL MEMORY IS USED, THE VIRTUAL ADDRESSES DO NOT GO DIRECTLY TO THE MEMORY BUS. INSTEAD, THEY GO TO AN MMU (MEMORY MANAGEMENT UNIT) THAT MAPS THE VIRTUAL ADDRESSES ONTO THE PHYSICAL MEMORY ADDRESSES

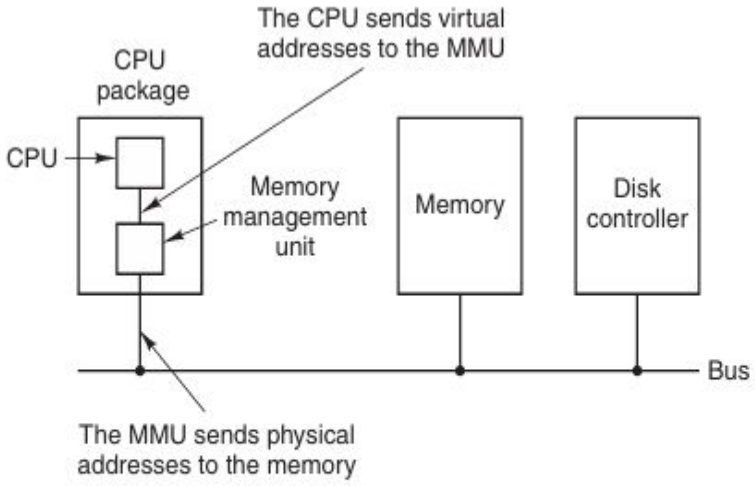


Figure 3-8. The position and function of the MMU. Here the MMU is shown as being a part of the CPU chip because it commonly is nowadays. However, logically it could be a separate chip and was years ago.

PAGING

- WE HAVE A COMPUTER THAT GENERATES 16-BIT ADDRESSES, FROM 0 UP TO 64K − 1. THESE ARE THE VIRTUAL ADDRESSES. THIS COMPUTER, HOWEV ER, HAS ONLY 32 KB OF PHYSICAL MEMORY. SO ALTHOUGH 64-KB PROGRAMS CAN BE WRITTEN, THEY CANNOT BE LOADED INTO MEMORY IN THEIR ENTIRETY AND RUN. A COMPLETE COPY OF A PROGRAM'S CORE IMAGE, UP TO 64 KB, MUST BE PRESENT ON THE DISK
- THE VIRTUAL ADDRESS SPACE CONSISTS OF FIXED-SIZE UNITS CALLED PAGES. THE CORRESPONDING UNITS IN THE PHYSICAL MEMORY ARE CALLED PAGE FRAMES. THE PAGES AND PAGE FRAMES ARE GENERALLY THE SAME SIZE. IN THIS EXAMPLE THEY ARE 4 KB, WITH 64 KB OF VIRTUAL ADDRESS SPACE AND 32 KB OF PHYSICAL MEMORY, WE GET 16 VIRTUAL PAGES AND 8 PAGE FRAMES. TRANSFERS BETWEEN RAM AND DISK ARE ALWAYS IN WHOLE PAGES.
- BY ITSELF, THIS ABILITY TO MAP THE 16 VIRTUAL PAGES ONTO ANY OF THE EIGHT PAGE FRAMES BY SETTING THE MMU'S MAP APPROPRIATELY DOES NOT SOLVE THE PROBLEM THAT THE VIRTUAL ADDRESS SPACE IS LARGER THAN THE PHYSICAL MEMORY. SINCE WE HAVE ONLY EIGHT PHYSICAL PAGE FRAMES, ONLY EIGHT OF THE VIRTUAL PAGES ARE MAPPED ONTO PHYSICAL MEMORY. THE OTHERS, SHOWN AS A CROSS IN THE FIGURE, ARE NOT MAPPED. IN THE ACTUAL HARDWARE, A PRESENT/ABSENT BIT KEEPS TRACK OF WHICH PAGES ARE PHYSICALLY PRESENT IN MEMORY

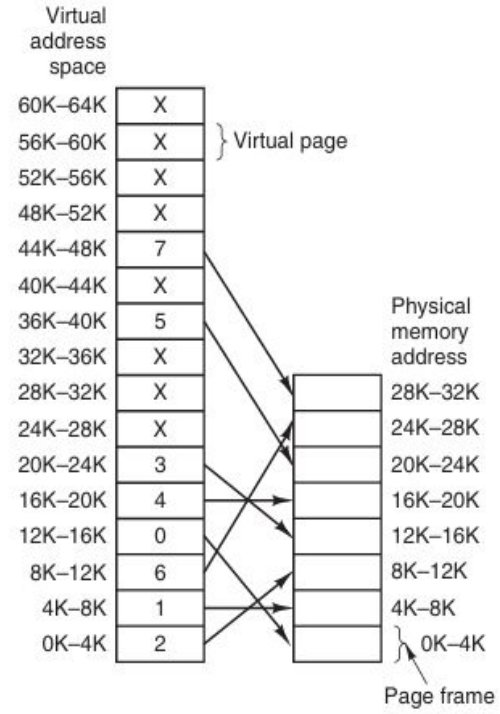


Figure 3-9. The relation between virtual addresses and physical memory addresses is given by the **page table**. Every page begins on a multiple of 4096 and ends 4095 addresses higher, so 4K–8K really means 4096–8191 and 8K to 12K means 8192–12287.

WHAT HAPPENS IF THE PROGRAM REFERENCES AN UNMAPPED ADDRESS, THE MMU NOTICES THAT THE PAGE IS UNMAPPED (INDICATED BY A CROSS IN THE FIGURE) AND CAUSES THE CPU TRAP TO THE OPERATING SYSTEM. THIS TRAP IS CALLED A PAGE FAULT.

PAGE REPLACEMENT ALGORITHMS

- WHEN A PAGE FAULT OCCURS, THE OPERATING SYSTEM HAS TO CHOOSE A PAGE TO EVICT (REMOVE FROM MEMORY) TO MAKE ROOM FOR THE INCOMING PAGE.

- IF THE PAGE TO BE RE MOVED HAS BEEN MODIFIED WHILE IN MEMORY, IT MUST BE REWRITTEN TO THE DISK TO BRING THE DISK COPY UP TO DATE.

- IF, HOWEVER, THE PAGE HAS NOT BEEN CHANGED (E.G., IT CONTAINS PROGRAM TEXT), THE DISK COPY IS ALREADY UP TO DATE, SO NO REWRITE IS NEEDED. THE PAGE TO BE READ IN JUST OVERWRITES THE PAGE BEING EVICTED.

THE OPTIMAL PAGE REPLACEMENT ALGORITHM

- AT THE MOMENT THAT A PAGE FAULT OCCURS, SOME SET OF PAGES IS IN MEMORY. ONE OF THESE PAGES WILL BE REFERENCED ON THE VERY NEXT INSTRUCTION (THE PAGE CONTAINING THAT INSTRUCTION). OTHER PAGES MAY NOT BE REFERENCED UNTIL 10, 100, OR PERHAPS 1000 INSTRUCTIONS LATER. EACH PAGE CAN BE LABELED WITH THE NUMBER OF INSTRUCTIONS THAT WILL BE EXECUTED BEFORE THAT PAGE IS FIRST REFERENCED.
- THE OPTIMAL PAGE REPLACEMENT ALGORITHM SAYS THAT THE PAGE WITH THE HIGHEST LABEL SHOULD BE REMOVED. IF ONE PAGE WILL NOT BE USED FOR 8 MILLION INSTRUCTIONS AND ANOTHER PAGE WILL NOT BE USED FOR 6 MILLION INSTRUCTIONS, REMOVING THE FORMER PUSHES THE PAGE FAULT THAT WILL FETCH IT BACK AS FAR INTO THE FUTURE AS POSSIBLE.
- THE ONLY PROBLEM WITH THIS ALGORITHM IS THAT IT IS UNREALIZABLE. AT THE TIME OF THE PAGE FAULT, THE OPERATING SYSTEM HAS NO WAY OF KNOWING WHEN EACH OF THE PAGES WILL BE REFERENCED NEXT.

THE NOT RECENTLY USED PAGE REPLACEMENT ALGORITHM

- IN ORDER TO ALLOW THE OPERATING SYSTEM TO COLLECT USEFUL PAGE USAGE STATISTICS, MOST COMPUTERS WITH VIRTUAL MEMORY HAVE TWO STATUS BITS, R AND M, ASSOCIATED WITH EACH PAGE. R IS SET WHENEVER THE PAGE IS REFERENCED (READ OR WRITTEN). M IS SET WHEN THE PAGE IS WRITTEN TO (I.E., MODIFIED).
- WHEN A PROCESS IS STARTED UP, ALL OF ITS PAGE TABLE ENTRIES ARE MARKED AS NOT IN MEMORY. AS SOON AS ANY PAGE IS REFERENCED, A PAGE FAULT WILL OCCUR. THE OPERATING SYSTEM THEN SETS THE R BIT (IN ITS INTERNAL TABLES), CHANGES THE PAGE TABLE ENTRY TO POINT TO THE CORRECT PAGE, WITH MODE READ ONLY, AND RESTARTS THE INSTRUCTION. IF THE PAGE IS SUBSEQUENTLY MODIFIED, ANOTHER PAGE FAULT WILL OCCUR, ALLOWING THE OPERATING SYSTEM TO SET THE M BIT AND CHANGE THE PAGE'S MODE TO READ/WRITE
- THE R AND M BITS CAN BE USED TO BUILD A SIMPLE PAGING ALGORITHM AS FOLLOWS. WHEN A PROCESS IS STARTED UP, BOTH PAGE BITS FOR ALL ITS PAGES ARE SET TO 0 BY THE OPERATING SYSTEM. PERIODICALLY (E.G., ON EACH CLOCK INTERRUPT), THE R BIT IS CLEARED, TO DISTINGUISH PAGES THAT HAVE NOT BEEN REFERENCED RECENTLY FROM THOSE THAT HAVE BEEN.

THE NOT RECENTLY USED PAGE REPLACEMENT ALGORITHM

- WHEN A PAGE FAULT OCCURS, THE OPERATING SYSTEM INSPECTS ALL THE PAGES AND DIVIDES THEM INTO FOUR CATEGORIES BASED ON THE CURRENT VALUES OF THEIR R AND M BITS:
  - CLASS 0: NOT REFERENCED, NOT MODIFIED.
  - CLASS 1: NOT REFERENCED, MODIFIED.
  - CLASS 2: REFERENCED, NOT MODIFIED.
  - CLASS 3: REFERENCED, MODIFIED.

- THE NRU (NOT RECENTLY USED) ALGORITHM REMOVES A PAGE AT RANDOM FROM THE LOWEST-NUMBERED NONEMPTY CLASS. IMPLICIT IN THIS ALGORITHM IS THE IDEA THAT IT IS BETTER TO REMOVE A MODIFIED PAGE THAT HAS NOT BEEN REFERENCED IN AT LEAST ONE CLOCK TICK (TYPICALLY ABOUT 20 MSEC) THAN A CLEAN PAGE THAT IS IN HEAVY USE. THE MAIN ATTRACTION OF NRU IS THAT IT IS EASY TO UNDERSTAND, MODERATELY EFFICIENT TO IMPLEMENT, AND GIVES A PERFORMANCE THAT, WHILE CERTAINLY NOT OPTIMAL, MAY BE ADEQUATE.

THE FIRST-IN, FIRST-OUT (FIFO) PAGE REPLACEMENT ALGORITHM

- THE OPERATING SYSTEM MAINTAINS A LIST OF ALL PAGES CURRENTLY IN MEMORY, WITH THE MOST RECENT ARRIVAL AT THE TAIL AND THE LEAST RECENT ARRIVAL AT THE HEAD.

- ON A PAGE FAULT, THE PAGE AT THE HEAD IS REMOVED AND THE NEW PAGE ADDED TO THE TAIL OF THE LIST.

- PROBLEM : THE OLDEST PAGE MAY STILL BE USEFUL. FOR THIS REASON, FIFO IN ITS PURE FORM IS RARELY USED

THE SECOND-CHANCE PAGE REPLACEMENT ALGORITHM

- ASIMPLE MODIFICATION TO FIFO THAT AVOIDS THE PROBLEM OF THROWING OUT A HEAVILY USED PAGE IS TO INSPECT THE R BIT OF THE OLDEST PAGE. IF IT IS 0, THE PAGE IS BOTH OLD AND UNUSED, SO IT IS REPLACED IMMEDIATELY. IF THE R BIT IS 1, THE BIT IS CLEARED, THE PAGE IS PUT ONTO THE END OF THE LIST OF PAGES, AND ITS LOAD TIME IS UPDATED AS THOUGH IT HAD JUST ARRIVED IN MEMORY. THEN THE SEARCH CONTINUES.
- THE OPERATION OF THIS ALGORITHM, CALLED *SECOND CHANCE* WE SEE PAGES A THROUGH H KEPT ON A LINKED LIST AND SORTED BY THE TIME THEY ARRIVED IN MEMORY.

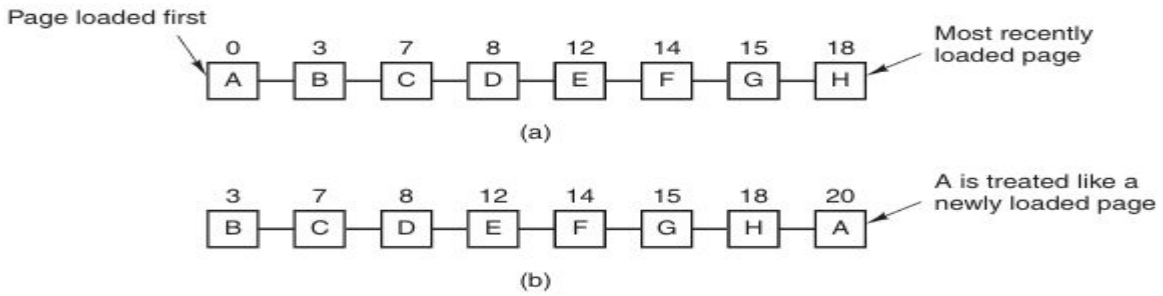


Figure 3-15. Operation of second chance. (a) Pages sorted in FIFO order. (b) Page list if a page fault occurs at time 20 and *A* has its *R* bit set. The numbers above the pages are their load times.

THE CLOCK PAGE REPLACEMENT ALGORITHM

- SECOND CHANCE IS A REASONABLE ALGORITHM, IT IS UNNECESSARILY INEFFICIENT BECAUSE IT IS CONSTANTLY MOVING PAGES AROUND ON ITS LIST
- KEEP ALL THE PAGE FRAMES ON A CIRCULAR LIST IN THE FORM OF A CLOCK. THE HAND POINTS TO THE OLDEST PAGE
- WHEN A PAGE FAULT OCCURS, THE PAGE BEING POINTED TO BY THE HAND IS INSPECTED. IF ITS R BIT IS 0, THE PAGE IS EVICTED, THE NEW PAGE IS INSERTED INTO THE CLOCK IN ITS PLACE, AND THE HAND IS ADVANCED ONE POSITION. IF R IS 1, IT IS CLEARED AND THE HAND IS ADVANCED TO THE NEXT PAGE. THIS PROCESS IS REPEATED UNTIL A PAGE IS FOUND WITH R =0.

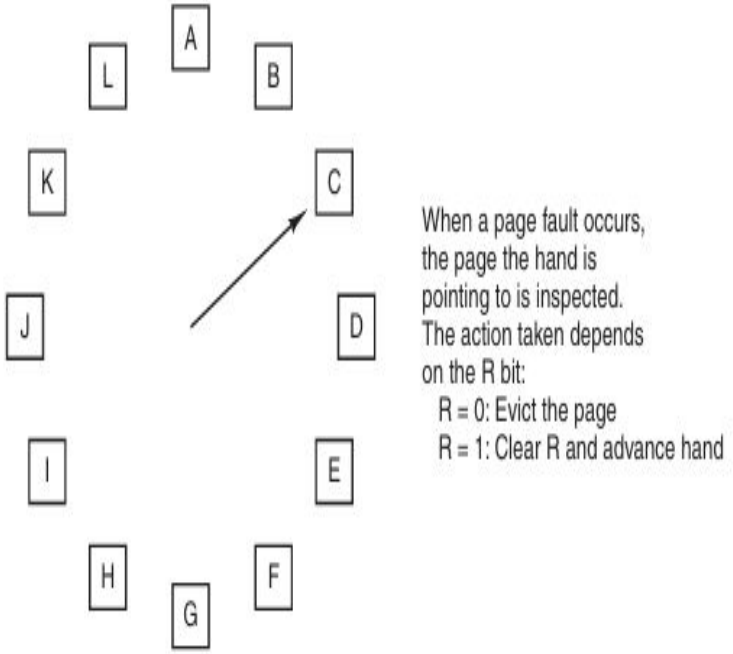


Figure 3-16. The clock page replacement algorithm.

THE LEAST RECENTLY USED (LRU) PAGE REPLACEMENT ALGORITHM

- WHEN A PAGE FAULT OCCURS, THROW OUT THE PAGE THAT HAS BEEN UNUSED FOR THE LONGEST TIME. THIS STRATEGY IS CALLED LRU (LEAST RECENTLY USED) PAGING.
- TO FULLY IMPLEMENT LRU, IT IS NECESSARY TO MAINTAIN A LINKED LIST OF ALL PAGES IN MEM ORY, WITH THE MOST RECENTLY USED PAGE AT THE FRONT AND THE LEAST RECENTLY USED PAGE AT THE REAR. THE DIFFICULTY IS THAT THE LIST MUST BE UPDATED ON EVERY MEMORY REFER ENCE. FINDING A PAGE IN THE LIST, DELETING IT, AND THEN MOVING IT TO THE FRONT IS A VERY TIME CONSUMING OPERATION, EVEN IN HARDWARE
- IMPLEMENT LRU WITH SPECIAL HARDWARE. LET US CONSIDER THE SIMPLEST WAY FIRST. THIS METHOD REQUIRES EQUIPPING THE HARDWARE WITH A 64-BIT COUNTER, C, THAT IS AUTOMATICALLY INCREMENTED AFTER EACH INSTRUCTION. FURTHERMORE, EACH PAGE TABLE ENTRY MUST ALSO HAVE A FIELD LARGE ENOUGH TO CONTAIN THE COUNTER. AFTER EACH MEMORY REFERENCE, THE CURRENT VALUE OF C IS STORED IN PAGE TABLE ENTRY FOR THE PAGE JUST REFERENCED. WHEN A PAGE FAULT OCCURS, THE OPERATING SYSTEM EXAMINES ALL THE COUNTERS IN THE PAGE TABLE TO FIND THE LOWEST ONE. THAT PAGE IS THE LEAST RECENTLY USED.

THE WORKING SET PAGE REPLACEMENT ALGORITHM

- IN THE PUREST FORM OF PAGING, PROCESSES ARE STARTED UP WITH NONE OF THEIR PAGES IN MEMORY. AS SOON AS THE CPU TRIES TO FETCH THE FIRST INSTRUCTION, IT GETS A PAGE FAULT, CAUSING THE OPERATING SYSTEM TO BRING IN THE PAGE CONTAINING THE FIRST INSTRUC TION. OTHER PAGE FAULTS FOR GLOBAL VARIABLES AND THE STACK USUALLY FOLLOW QUICKLY. AFTER A WHILE, THE PROCESS HAS MOST OF THE PAGES IT NEEDS AND SETTLES DOWN TO RUN WITH RELATIVELY FEW PAGE FAULTS. THIS STRATEGY IS CALLED DEMAND PAGING BECAUSE PAGES ARE LOADED ONLY ON DEMAND, NOT IN ADVANCE.
- MOST PROCESSES EXHIBIT A LOCALITY OF REFERENCE, MEANING THAT DURING ANY PHASE OF EXECUTION, THE PROCESS REFERENCES ONLY A RELATIVELY SMALL FRACTION OF ITS PAGES.
- THE SET OF PAGES THAT A PROCESS IS CURRENTLY USING IS ITS *WORKING SET* .IF THE ENTIRE WORKING SET IS IN MEMORY, THE PROCESS WILL RUN WITHOUT CAUSING MANY FAULTS UNTIL IT MOVES INTO ANOTHER EXECUTION PHASE
- A PROGRAM CAUSING PAGE FAULTS EVERY FEW INSTRUCTIONS IS SAID TO BE *THRASHING*

THE WORKING SET PAGE REPLACEMENT ALGORITHM

- THEREFORE, MANY PAGING SYSTEMS TRY TO KEEP TRACK OF EACH PROCESS' WORKING SET AND MAKE SURE THAT IT IS IN MEMORY BEFORE LETTING THE PROCESS RUN. THIS APPROACH IS CALLED THE WORKING SET MODEL .
- IT IS DESIGNED TO GREATLY REDUCE THE PAGE FAULT RATE. LOADING THE PAGES BEFORE LETTING PROCESSES RUN IS ALSO CALLED PREPAGING. NOTE THAT THE WORKING SET CHANGES OVER TIME.
- TO IMPLEMENT THE WORKING SET MODEL, IT IS NECESSARY FOR THE OPERATING SYSTEM TO KEEP TRACK OF WHICH PAGES ARE IN THE WORKING SET. HAVING THIS INFORMATION ALSO IMMEDIATELY LEADS TO A POSSIBLE PAGE REPLACEMENT ALGORITHM: WHEN A PAGE FAULT OC CURS, FIND A PAGE NOT IN THE WORKING SET AND EVICT IT.
- TO IMPLEMENT SUCH AN ALGORITHM, WE NEED A PRECISE WAY OF DETERMINING WHICH PAGES ARE IN THE WORKING SET. BY DEFINITION, THE WORKING SET IS THE SET OF PAGES USED IN THE K MOST RECENT MEMORY REFERENCES.
- TO IMPLEMENT ANY WORKING SET ALGORITHM, SOME VALUE OF K MUST BE CHOSEN IN ADVANCE. THEN, AFTER EVERY MEMORY REFERENCE, THE SET OF PAGES USED BY THE MOST RECENT K MEMORY REFERENCES IS UNIQUELY DETERMINED.

THE WORKING SET PAGE REPLACEMENT ALGORITHM

- THE BASIC IDEA IS TO FIND A PAGE THAT IS NOT IN THE WORKING SET AND EVICT IT. IN FIG. WE SEE A PORTION OF A PAGE TABLE FOR SOME MACHINE. BECAUSE ONLY PAGES LOCATED IN MEMORY ARE CONSIDERED AS CANDIDATES FOR EVICTION, PAGES THAT ARE ABSENT FROM MEMORY ARE IGNORED BY THIS ALGORITHM. EACH ENTRY CONTAINS (AT LEAST) TWO KEY ITEMS OF INFORMATION: THE (APPROXIMATE) TIME THE PAGE WAS LAST USED AND THE R (REFER ENCED) BIT. AN EMPTY WHITE RECTANGLE SYMBOLIZES THE OTHER FIELDS NOT NEEDED FOR THIS ALGORITHM, SUCH AS THE PAGE FRAME NUMBER, THE PROTECTION BITS, AND THE M (MODIFIED) BIT.
- ON EVERY PAGE FAULT, THE PAGE TABLE IS SCANNED TO LOOK FOR A SUITABLE PAGE TO EVICT. AS EACH ENTRY IS PROCESSED, THE R BIT IS EXAMINED. IF IT IS 1, THE CURRENT VIRTUAL TIME IS WRITTEN INTO THE TIME OF LAST USE FIELD IN THE PAGE TABLE, INDICATING THAT THE PAGE WAS IN USE AT THE TIME THE FAULT OCCURRED. SINCE THE PAGE HAS BEEN REFERENCED DURING THE CURRENT CLOCK TICK, IT IS CLEARLY IN THE WORKING SET AND IS NOT A CANDIDATE FOR REMOVAL (T IS ASSUMED TO SPAN MULTIPLE CLOCK TICKS).
- IF R IS 0, THE PAGE HAS NOT BEEN REFERENCED DURING THE CURRENT CLOCK TICK AND MAY BE A CANDIDATE FOR REMOVAL. TO SEE WHETHER OR NOT IT SHOULD BE REMOVED, ITS AGE (THE CURRENT VIRTUAL TIME MINUS ITS TIME OF LAST USE) IS COMPUTED AND COMPARED TO T. IF THE AGE IS GREATER THAN T, THE PAGE IS NO LONGER IN THE WORKING SET AND THE NEW PAGE REPLACES IT. THE SCAN CONTINUES UPDATING THE REMAINING ENTRIES.
- IN THE WORST CASE, ALL PAGES HAVE BEEN REFERENCED DURING THE CURRENT CLOCK TICK (AND THUS ALL HAVE R = 1), SO ONE IS CHOSEN AT RANDOM FOR REMOVAL, PREFER ABLY A CLEAN PAGE, IF ONE EXISTS.

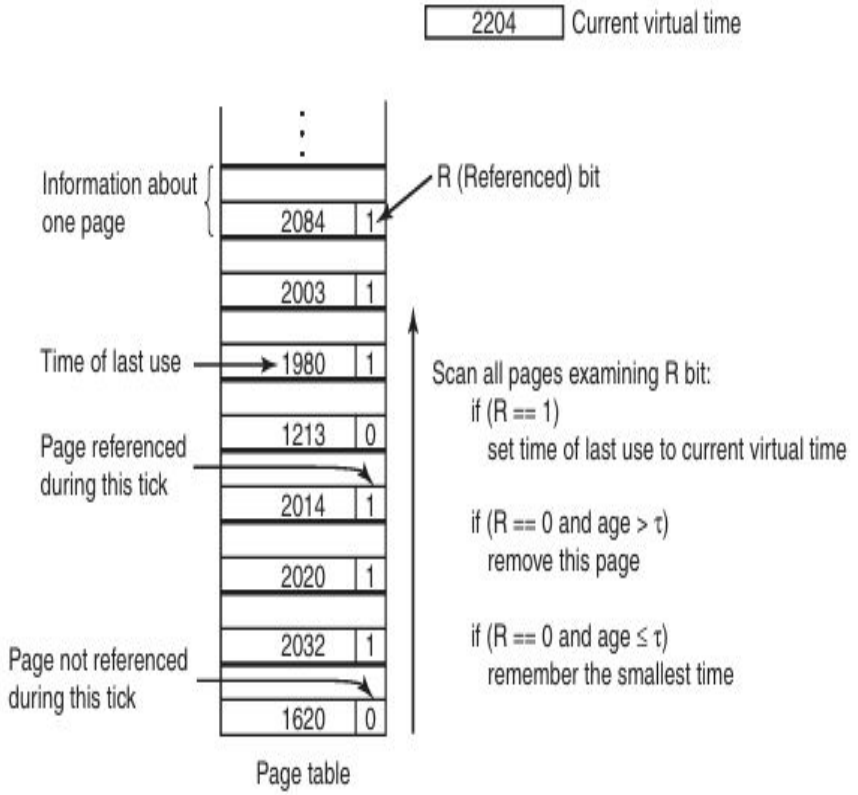


Figure 3-19. The working set algorithm.

THE WSCLOCK PAGE REPLACEMENT ALGORITHM

- The basic working set algorithm is cumbersome, since the entire page table has to be scanned at each page fault until a suitable candidate is located.
- An improved algorithm, which is based on the clock algorithm but also uses the working set information, is called WSClock
- The data structure needed is a circular list of page frames, as in the clock algo rithm, and as shown in Fig. 3-20(a). Initially, this list is empty. When the first page is loaded, it is added to the list. As more pages are added, they go into the list to form a ring. Each entry contains the Time of last use field from the basic working set algorithm, as well as the R bit (shown) and the M bit (not shown).

THE WSCLOCK PAGE REPLACEMENT ALGORITHM

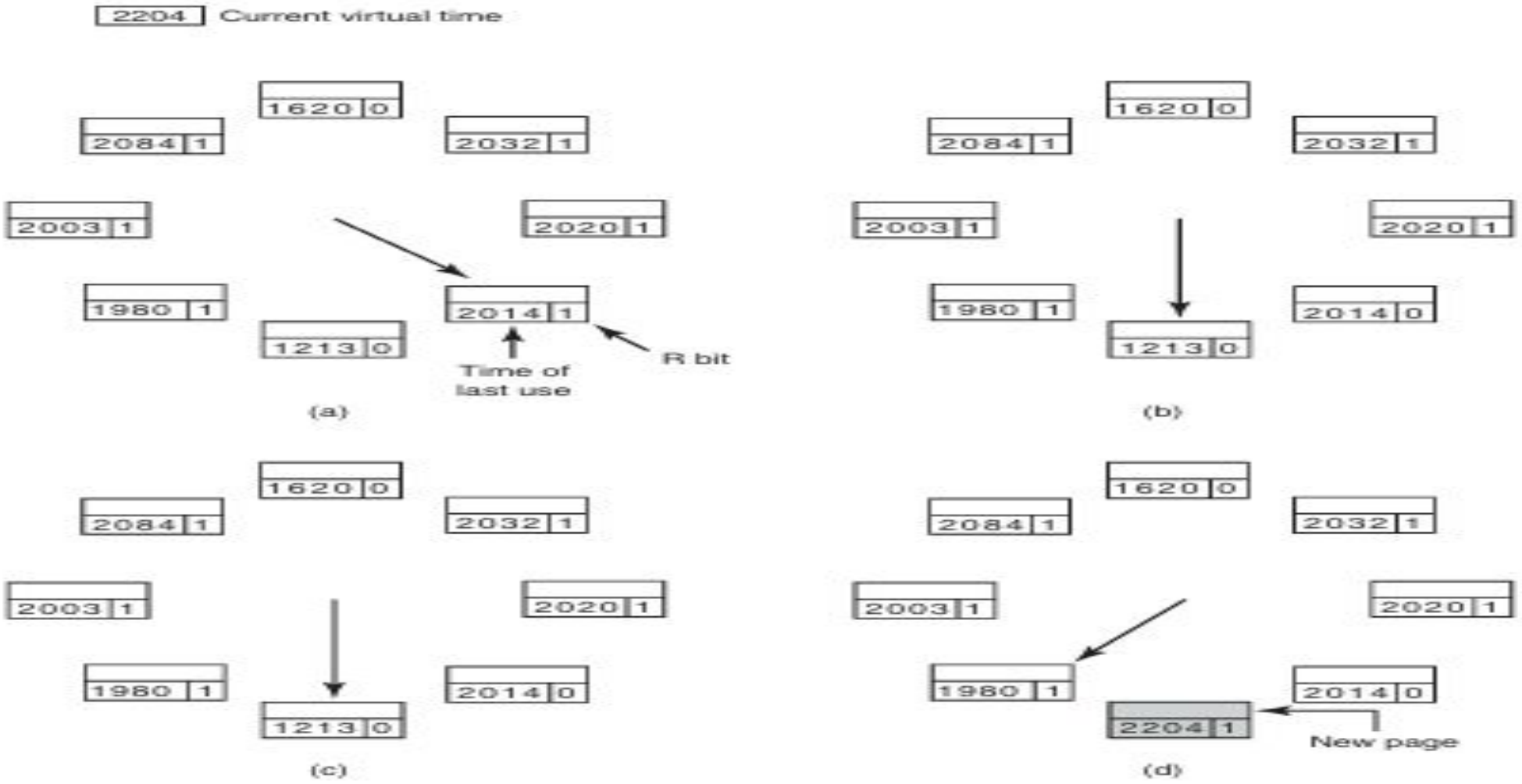


Figure 3-20. Operation of the WSClock algorithm. (a) and (b) give an example of what happens when $R = 1$. (c) and (d) give an example of $R = 0$.

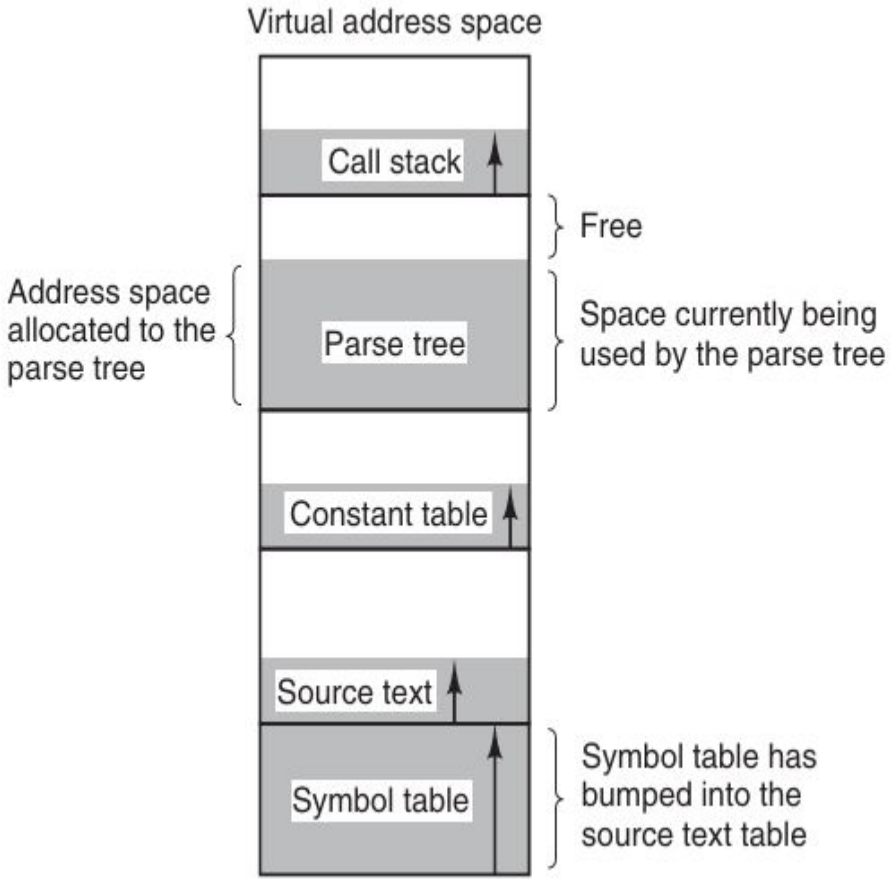
SUMMARY

Algorithm	Comment
Optimal	Not implementable, but useful as a benchmark
NRU (Not Recently Used)	Very crude approximation of LRU
FIFO (First-In, First-Out)	Might throw out important pages
Second chance	Big improvement over FIFO
Clock	Realistic
LRU (Least Recently Used)	Excellent, but difficult to implement exactly
NFU (Not Frequently Used)	Fairly crude approximation to LRU
Aging	Efficient algorithm that approximates LRU well
Working set	Somewhat expensive to implement
WSClock	Good efficient algorithm

SEGMENTATION

- The virtual memory discussed so far is one-dimensional because the virtual ad dresses go from 0 to some maximum address, one address after another. For many problems, having two or more separate virtual address spaces may be much better than having only one.
- **EXAMPLE**

1. The source text being saved for the printed listing (on batch systems).
2. The symbol table, containing the names and attributes of variables
3. The table containing all the integer and floating-point constants used.
4. The parse tree, containing the syntactic analysis of the program.
5. The stack used for procedure calls within the compiler.



SEGMENTATION

- provide the machine with many completely independent address spaces, which are called segments
- Each segment consists of a linear sequence of addresses, starting at 0 and going up to some maximum value. The length of each segment may be anything from 0 to the maximum address allowed.
- Because each segment constitutes a separate address space, different segments can grow or shrink independently without affecting each other.
- To specify an address in this segmented or two-dimensional memory, the program must supply a two-part address, a segment number, and an address within the segment.

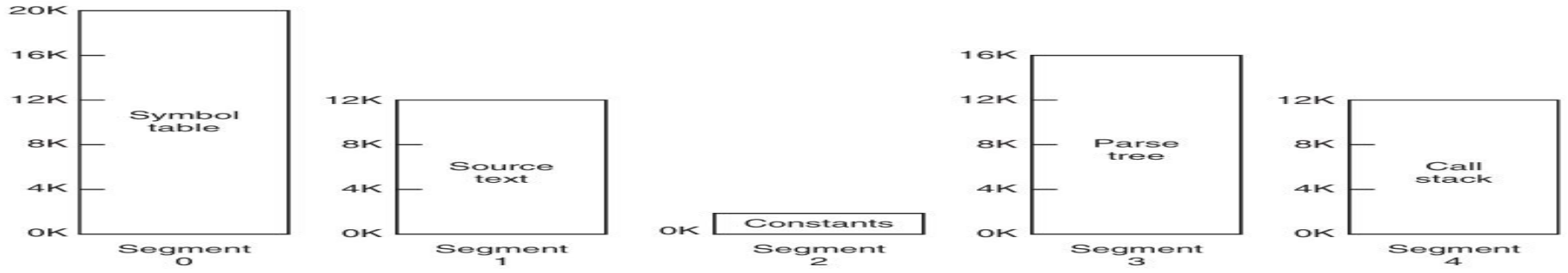


Figure 3-31. A segmented memory allows each table to grow or shrink independently of the other tables.

ADVANTAGES OF SEGMENTATION

- SIMPLIFYING THE HANDLING OF DATA STRUCTURES THAT ARE GROWING OR SHRINKING. IF EACH PROCEDURE OCCUPIES A SEPA RATE SEGMENT, WITH ADDRESS 0 AS ITS STARTING ADDRESS, THE LINKING OF PROCEDURES COM PILED SEPARATELY IS GREATLY SIMPLIFIED. AFTER ALL THE PROCEDURES THAT CONSTITUTE A PRO GRAM HAVE BEEN COMPILED AND LINKED UP, A PROCEDURE CALL TO THE PROCEDURE IN SEG MENT N WILL USE THE TWO-PART ADDRESS (N, 0) TO ADDRESS WORD 0 (THE ENTRY POINT).

- SEGMENTATION ALSO FACILITATES SHARING PROCEDURES OR DATA BETWEEN SEVERAL PROC ESSES. A COMMON EXAMPLE IS THE SHARED LIBRARY. MODERN WORKSTATIONS THAT RUN AD VANCED WINDOW SYSTEMS OFTEN HAVE EXTREMELY LARGE GRAPHICAL LIBRARIES COMPILED INTO NEARLY EVERY PROGRAM. IN A SEGMENTED SYSTEM, THE GRAPHICAL LIBRARY CAN BE PUT IN A SEGMENT AND SHARED BY MULTIPLE PROCESSES, ELIMINATING THE NEED FOR HAVING IT IN EV ERY PROCESS' ADDRESS SPACE.

- EACH SEGMENT FORMS A LOGICAL ENTITY THAT PROGRAMMERS KNOW ABOUT, SUCH AS A PROCEDURE, OR AN ARRAY, DIFFERENT SEGMENTS CAN HAVE DIFFERENT KINDS OF PROTEC TION. A PROCEDURE SEGMENT CAN BE SPECIFIED AS EXECUTE ONLY, PROHIBITING ATTEMPTS TO READ FROM OR STORE INTO IT. A FLOATING-POINT ARRAY CAN BE SPECIFIED AS READ/WRITE BUT NOT EXECUTE, AND ATTEMPTS TO JUMP TO IT WILL BE CAUGHT. SUCH PROTECTION IS HELP FUL IN CATCHING BUGS.

PAGING VS SEGMENTATION

Consideration	Paging	Segmentation
Need the programmer be aware that this technique is being used?	No	Yes
How many linear address spaces are there?	1	Many
Can the total address space exceed the size of physical memory?	Yes	Yes
Can procedures and data be distinguished and separately protected?	No	Yes
Can tables whose size fluctuates be accommodated easily?	No	Yes
Is sharing of procedures between users facilitated?	No	Yes
Why was this technique invented?	To get a large linear address space without having to buy more physical memory	To allow programs and data to be broken up into logically independent address spaces and to aid sharing and protection

Week: 3-4

Process Scheduling

Topics

- Job and Process Scheduling
- Scheduling Levels
- Objectives and Criterias
- Scheduling Algorithm

Books:

https://drive.google.com/file/d/1GlVwWVPVl2RSm24bcEGzc2GqywGEzZQL/view?usp=sharing

Scheduling

- If two or more process are in Ready State and only one processor is available then a choice has to be made which process to run next.
- The part of the operating system that makes the choice is called the scheduler, and the algorithm it uses is called the scheduling algorithm.
- There are two levels of scheduling: Process Scheduling and Thread Scheduling.
- Though there are some common issues between them.
- Thread Scheduling does not care about or little care about of the process to which the thread belongs.

Process Behavior

2 Types of Process:

- CPU-bound Process
- I/O-bound Process

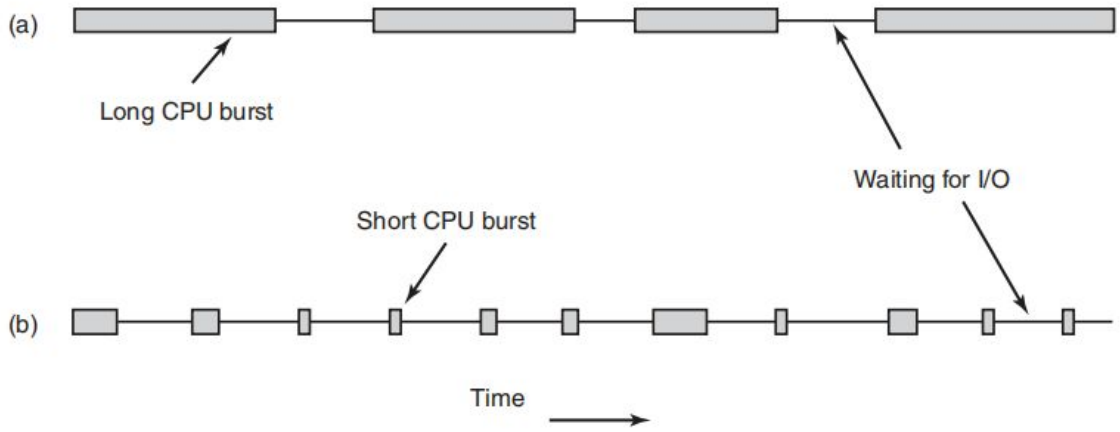


Figure 2-39. Bursts of CPU usage alternate with periods of waiting for I/O. (a) A CPU-bound process. (b) An I/O-bound process.

When to Schedule

- When a new process is created
- When a process exits
- When a process blocks on I/O, on a semaphore, or for some other reason
- When an I/O interrupt occurs

Types of Scheduling Respect to Clock Interrupt

- Scheduling algorithms can be divided into two categories with respect to how they deal with clock interrupts
- A **nonpreemptive** scheduling algorithm picks a process to run and then just lets it run until it blocks (either on I/O or waiting for another process) or voluntarily releases the CPU.
- A **preemptive** scheduling algorithm picks a process and lets it run for a maximum of some fixed time. A **clock interrupt** is required for preemptive scheduling.

Categories of Scheduling Algorithms

In different environments different scheduling algorithms are needed. What the scheduler should optimize for a system is not the same in all systems. There are three common environments:

- Batch
- Interactive
- Real time

Categories of Scheduling Algorithms

- Batch
  - Mainly use in business world
  - There are no users impatiently waiting at their terminals for a quick response to a short request.
  - Non Preemptive algorithms, or preemptive algorithms with long time periods for each process, are often acceptable.
- Interactive
  - Server with lots of requests which needs to be response quickly.
  - Personal Computer with many application.
  - Preemption is essential to keep one process from hogging the CPU and denying service to the others.
- Interactive
  - Sometimes preemption is not needed because the processes know that they may not run for long periods of time and usually do their work and block quickly.
  - Interactive system are general purpose and may run arbitrary application where Real time are for specific purpose.

Scheduling Algorithm Goals

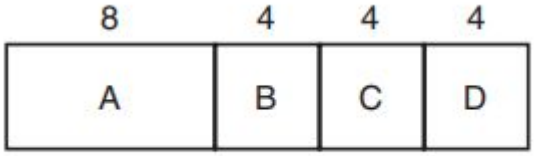
- All systems
  - Fairness - giving each process a fair share of the CPU
  - Policy enforcement - seeing that stated policy is carried out
  - Balance - keeping all parts of the system busy
- Batch systems
  - Throughput - maximize jobs per hour
  - **Turnaround time** - minimize time between submission and termination
    - Throughput vs Turnaround time
  - CPU utilization - keep the CPU busy all the time
- Interactive systems
  - Response time - respond to requests quickly
  - Proportionality - meet users' expectations
- Real-time systems
  - Meeting deadlines - avoid losing data
  - Predictability - avoid quality degradation in multimedia systems

Scheduling Algorithm: Batch System

- First-Come, First-Served (nonpreemptive)
  - Processes are assigned the CPU in the order they request it.
  - There is a single queue of ready processes.
  - A single linked list keeps track of all ready processes
- Problem Scenario:
  - 1 CPU-Bound 1 sec at a time, Many I/O Bound each of them need 1000 disk reads. Need 1000*1 sec = 1000 sec to complete the I/O Bound Process. But with preemption after 10 msec it decreases to 10 sec.

Scheduling Algorithm: Batch System

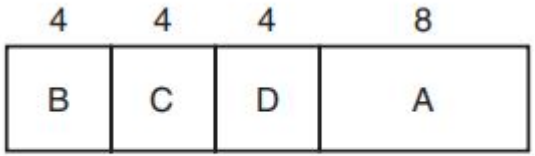
- Shortest Job First (nonpreemptive)
  - **Assumes** the run times are known in advance.
  - When several equally important jobs are sitting in the input queue waiting to be started, the scheduler picks the shortest job first.
  - The mean turnaround time is (4a + 3b + 2c + d)/4 where a is the shortest job and d is the longest job and a ≤ b ≤ c ≤d.



(a)

Fig: Running four jobs in the original order

- Example:
  - Four jobs A, B, C, and D with run times of 8, 4, 4, and 4 minutes, respectively.
  - What will be **turnaround** time and waiting for each of the job for both cases?



(b)

Fig: Running them in shortest job first order

Scheduling Algorithm: Batch System

Disadvantage:

- Shortest job first is optimal only when all the jobs are available simultaneously.
- Consider five jobs, A through E, with run times of 2, 4, 1, 1, and 1, respectively. Their arrival times are 0, 0, 3, 3, and 3. Calculate the turnaround time and wait time.
- However, running them in the order B, C, D, E, A has an average wait of 4.4.

Solution:

- A preemptive version of shortest job first is **shortest remaining time next**. When a new job arrives, its total time is compared to the current process' remaining time. And chooses the less one.

Scheduling Algorithm: Interactive System

- Round-Robin Scheduling
  - Each process is assigned a time interval, called its quantum, during which it is allowed to run.
  - After that he CPU is preempted and given to another process.
  - Setting the quantum too short causes too many process switches and lowers the **CPU efficiency**, but setting it too long may cause **poor response** to short interactive requests.

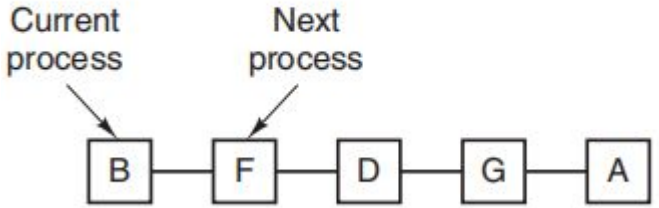


Fig: The list of runnable processes

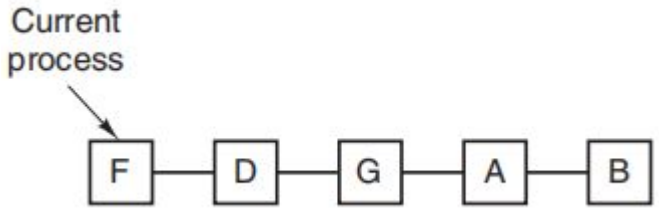


Fig: The list of runnable processes After the quantum

Scheduling Algorithm: Interactive System

- Priority Scheduling
  - Each process is assigned a priority, and the runnable process with the highest priority is allowed to run.
  - Allow for both multi user process or multiple process of same user.
  - To prevent high-priority processes from running indefinitely, the scheduler may decrease the **priority** of the currently running process at each clock tick (i.e., at each clock interrupt).
  - **Priorities** can also be assigned **dynamically** by the system to achieve certain system goals.
  - Give **I/O Bound process more priority** so that they can not occupy memory for an unnecessarily long time. **The I/O request can proceed with parallel with other process.**
  - A simple **algorithm** for giving good service to **I/O-bound** processes is to set the **priority** to **1/f**, where **f is the fraction of the last quantum that a process used**.

Scheduling Algorithm: Interactive System

- Priority Scheduling with Group Processes
  - Use Priority scheduling among the **priority classes** but **round-robin scheduling** within each class.
  - **If priorities are not adjusted occasionally, lower-priority classes may all starve to death.**

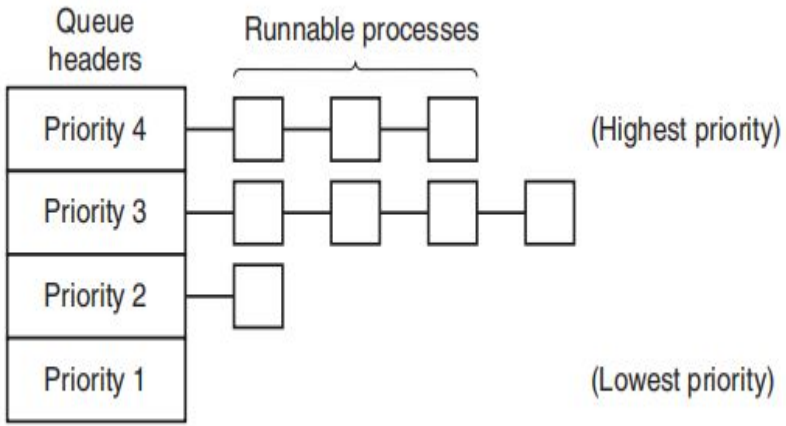


Figure 2-43. A scheduling algorithm with four priority classes.

Scheduling Algorithm: Interactive System

- It is more efficient to give **CPU-bound processes a large quantum** once in a while, rather than giving them small quantum frequently (**to reduce swapping**).
- We already know that “Giving all processes a large quantum would mean poor response time”.

Soln:

- **Multiple Queues [Changing Quantum]**
  - Processes on highest class get the lowest quantum and Processes on lowest class get the highest quantum.
  - Whenever a process used up all the quanta allocated to it, it was moved down one class.
  - Example: CTSS

Scheduling Algorithm: Interactive System

- Shortest Process Next
  - The problem is figuring out which of the currently runnable processes is the shortest one.
  - Make estimates based on past behavior and run the process with the shortest estimated running time.
  - If the estimated time per command for some process is T0 and its next run is measured to be T1.

Then **next estimation** will be: ***aT0 + (1 − a)T1*** [Weighted Sum of *T0* & *T1*]

  - With a = 1/2, successive estimates are:

$$T0,\ T0/2 + T1/2,\ T0/4 + T1/4 + T2/2,\ T0/8 + T1/8 + T2/4 + T3/2$$

  - This technique is called **aging**.
  - Aging is especially easy to implement when a = 1/2. All that is needed is to add the new value to the current estimate and divide the sum by 2 (by shifting it right 1 bit).

Scheduling Algorithm: Interactive System

- Guaranteed Scheduling [Every Process are equal]
  - System with n processes running, all things being equal, each one should get **1/n** of the CPU cycles.
  - System must keep track of how much CPU each process has **had** since its creation.
  - It then computes the amount of CPU each one is **entitled** to, namely the time since creation divided by n.
  - Compute the ratio of actual CPU time consumed to CPU time entitled.
  - Run the process with the lowest ratio until its ratio has moved above that of **its closest competitor**. Then that one is chosen to run next.

Scheduling Algorithm: Interactive System

- Lottery Scheduling
  - The basic idea is to give processes lottery tickets for various system resources, such as CPU time.
  - Whenever a scheduling decision has to be made, a lottery ticket is chosen at random, and the process holding that ticket gets the resource.
  - More important processes can be given extra tickets, to increase their odds of winning.
  - In contrast to a priority scheduler, where it is very hard to state what having a priority of 40 actually means, here the rule is clear: a process holding a fraction f of the tickets will get about a fraction f of the resource in question.
  - Lottery scheduling is highly responsive.
  - Cooperating processes may exchange tickets if they wish.
  - Example: A video server in which several processes are feeding video streams to their clients, but at different frame rates. Suppose that the processes need frames at 10, 20, and 25 frames/sec. By allocating these processes 10, 20, and 25 tickets, respectively, they will automatically divide the CPU in approximately the correct proportion, that is, 10 : 20 : 25.

Scheduling Algorithm: Interactive System

- Fair-Share Scheduling
  - Each process is scheduled on its own, without regard to who its owner is.
  - Each user is allocated some fraction of the CPU and the scheduler picks processes in such a way as to enforce it.
  - As an example, consider a system with two users, each of which has been promised 50% of the CPU. User 1 has four processes, A, B, C, and D, and user 2 has only one process, E. If round-robin scheduling is used, a possible scheduling sequence that meets all the constraints is this one:

    A E B E C E D E A E B E C E D E

  - If user 1 is entitled to twice as much CPU time as user 2:

    A B E C D E A B E C D E

Scheduling Algorithm: Real-Time System

- One or more physical devices external to the computer generate stimuli, and the computer must react appropriately to them within a fixed amount of time.
- When an external event is detected, it is the job of the scheduler to schedule the processes in such a way that all deadlines are met.
- The computer in a compact disc player gets the bits as they come off the drive and must convert them into music within a very tight time interval.
- Having the right answer but having it too late is often just as bad as not having it at all.
- Categorized into 2 types. **Hard Real Time** where deadlines must meet & **Soft Real TIme** where missing deadline undesirable.

Scheduling Algorithm: Real-Time System

For, **Periodic Real Time System**, if there are ***m*** periodic events and event ***i*** occurs with period P_i and requires C_i sec of CPU time to handle each event, then the load can be handled only if:

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$$

A real-time system that meets this criterion is said to be schedulable. This means it can actually be implemented. A process that fails to meet this test cannot be scheduled because the total amount of CPU time the processes want collectively is more than the CPU can deliver.

Scheduling Algorithm: Real-Time System

If there are ***m*** periodic events and event ***i*** occurs with period P_i and requires C_i sec of CPU time to handle each event, then the load can be handled only if:

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$$

Tasks	*1*	*2*	*3*
Period Time (P_i)	*100* msec	*200* msec	*500* msec
Require Time (C_i)	*50* msec	*30* msec	*100* msec
C_i/P_i	*.5*	*.15*	*.2*

Is the system schedulable or not? If a fourth event with a period of 1 sec is added, for which maximum processing time the system will remain schedulable?

Policy & Mechanism

Two Different Separated Idea:

- **Scheduling Mechanism**
  - What we have studied in early lectures. Mainly done by kernel.
- **Scheduling Policy**
  - Parent (User Process) can set the priorities.

The kernel uses a priority-scheduling algorithm but provides a **system call** by which a process can set (and change) **the priorities** of its children. In this way, the parent can control how its children are scheduled, even though **it itself does not do the scheduling**. Here the mechanism is in the kernel but policy is set by a user process.

Thread Scheduling

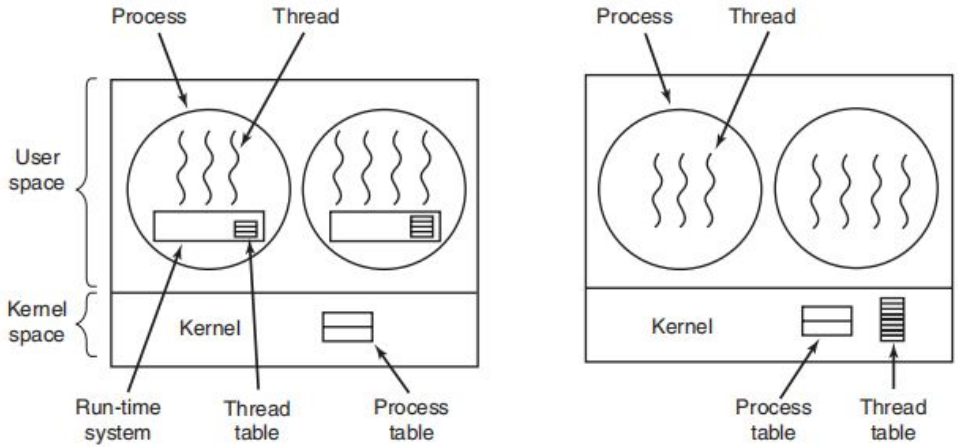


Figure 2-16. (a) A user-level threads package. (b) A threads package managed by the kernel.

- In User Level, everything is done by **libraries** (run time system needed).
- Thread switching is **faster** than trapping to the kernel (**like procedure call only**). The procedure that saves the thread's state and the scheduler are just local **procedures**, so invoking them is much more efficient than making a kernel call. **Among other issues, no trap is needed, no context switch is needed, the memory cache need not be flushed, and so on. This makes thread scheduling very fast**.
- **Blocking system call will block all the threads.**
- No clock interrupts for threads.
- Kernel only sees single process no thread.
- Application-specific thread schedulers can tune an application better than the kernel can.

- In Kernel Level everything is done by **Kernel.**
- Kernel threads do not require any new, non blocking system calls.
- Clock interrupt for threads.

Thread Scheduling

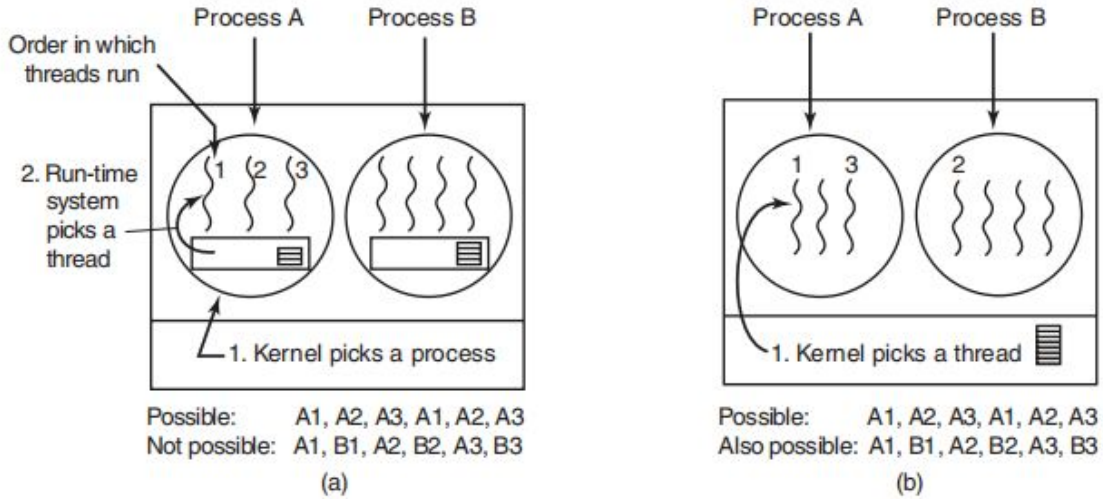


Figure 2-44. (a) Possible scheduling of user-level threads with a 50-msec process quantum and threads that run 5 msec per CPU burst. (b) Possible scheduling of kernel-level threads with the same characteristics as (a).

Exercise-01

Measurements of a certain system have shown that the average process runs for a time *T* before blocking on I/O. A process switch requires a time *S*, which is effectively wasted (**overhead**). For round-robin scheduling with quantum *Q*, give a formula for the CPU efficiency for each of the following:

(a) $Q = \infty$

(b) $Q > T$

(c) $S < Q < T$

(d) $Q = S < T$

(e) Q nearly 0

Exercise-02

Five batch jobs. A through E, arrive at a computer center at almost the same time. They have estimated running times of 10, 6, 2, 4, and 8 minutes. Their (externally determined) priorities are 3, 5, 2, 1, and 4, respectively, with 5 being the highest priority. For each of the following scheduling algorithms, determine the **mean process turnaround time**. Ignore process switching overhead.

(a) Round robin.

(b) Priority scheduling.

(c) First-come, first-served (run in order A, B, C, D & E).

(d) Shortest job first.

For (a), assume that the system is multiprogrammed, and that each job gets **its fair share of the CPU**. For (b) through (d), **assume that only one job at a time runs, until it finishes**. All jobs are completely CPU bound.

Exercise-03 and 04

Five jobs are waiting to be run. Their expected run times are 9, 6, 3, 5, and X. In what order should they be run to minimize average turnaround time or waiting time or response time? (Your answer will depend on X).

A process running on CTSS needs 30 quanta to complete. How many times must it be swapped in, including the very first time (before it has run at all)?

Exercise-05

A soft real-time system has four periodic events with periods of 50, 100, 200, and 250 msec each. Suppose that the four events require 35, 20, 10, and x msec of CPU time, respectively. What is the largest value of x for which the system is schedulable?

Exercise: DIY

Problem: 40 to 50


thank
you

PROCESS SCHEDULING

FIRST COME FIRST SERVE-EXAMPLE

- Let's consider three processes with the following details:
- **Process P1**: Arrival Time = 0, Burst Time = 4
- **Process P2**: Arrival Time = 1, Burst Time = 3
- **Process P3**: Arrival Time = 2, Burst Time = 1

- **Execution Order**

FCFS executes processes in the order they arrive.
P1 arrives at time 0 and starts immediately.
P2 arrives at time 1, but must wait for P1 to finish.
P3 arrives at time 2, but must wait for both P1 and P2 to finish.

Calculate Completion Time (CT)

Completion Time is the time at which a process finishes execution.

- **P1**: Arrives at time 0 and starts immediately.
  - Completion Time = Arrival Time + Burst Time = 0 + 4 = 4
- **P2**: Arrives at time 1 but starts after P1 finishes.
  - Start Time = 4 (P1 finishes at time 4)
  - Completion Time = Start Time + Burst Time = 4 + 3 = 7
- **P3**: Arrives at time 2 but starts after P2 finishes.
  - Start Time = 7 (P2 finishes at time 7)
  - Completion Time = Start Time + Burst Time = 7 + 1 = 8

Calculate Turnaround Time (TAT)

- Turnaround Time is the total time taken from arrival to completion.

TAT = CT-AT

- **P1**: TAT = Completion Time - Arrival Time = 4 - 0 = 4
- **P2**: TAT = Completion Time - Arrival Time = 7 - 1 = 6
- **P3**: TAT = Completion Time - Arrival Time = 8 - 2 = 6

Calculate Waiting Time (WT)

- Waiting Time is the total time a process spends waiting in the ready queue before it gets the CPU.

$$WT = TAT - BT$$

- **P1**: WT = Turnaround Time - Burst Time = 4 - 4 = 0
- **P2**: WT = Turnaround Time - Burst Time = 6 - 3 = 3
- **P3**: WT = Turnaround Time - Burst Time = 6 - 1 = 5

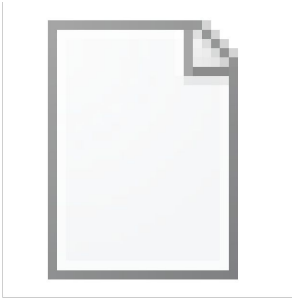
AVERAGE WAITING TIME AND TURNAROUND TIME

- **Average Waiting Time**:

Average WT=(0+3+5) / 3=8/3≈2.67 units

- **Average Turnaround Time**:
- Average TAT=(4+6+6)/3=16/3≈5.33 units

FCFS Implementation



FCFS.sh

SHORTEST JOB FIRST (SJF)

- Shortest Job First (SJF) scheduling algorithm selects the process with the smallest burst time for execution next

- **Example Scenario**
- Consider four processes with the following details:
- **Process P1**: Arrival Time = 0, Burst Time = 7
- **Process P2**: Arrival Time = 2, Burst Time = 4
- **Process P3**: Arrival Time = 4, Burst Time = 1
- **Process P4**: Arrival Time = 5, Burst Time = 4

Determine the Execution Order

The processes are scheduled based on the shortest burst time available at the current time. This will be a non-preemptive SJF example.

1. **P1** arrives at time 0 and starts executing because it's the only process available.
2. **P2** and **P3** arrive while **P1** is running. At time 7 (when **P1** completes), **P3** has the shortest burst time and is selected next.
3. After **P3** completes, **P2** and **P4** are considered. **P2** has the shortest burst time among them.
4. Finally, **P4** runs as it is the remaining process.

Calculate Completion Time (CT), Turnaround Time (TAT), and Waiting Time (WT)

Process	Arrival Time (AT)	Burst Time (BT)	Completion Time (CT)	Turnaround Time (TAT)	Waiting Time (WT)
P1	0	7	7	7	0
P2	2	4	12	10	6
P3	4	1	8	4	3
P4	5	4	16	11	7

SJF IMPLEMENTATION

SJS.sh

ROUND ROBIN (RR) SCHEDULING

- Round Robin (RR) scheduling is a preemptive scheduling algorithm where each process is assigned a fixed time slice (quantum). When a process's time slice expires, it is moved to the back of the queue, and the next process is executed.

- **Example Scenario**
- Consider four processes with the following details:
- **Process P1**: Arrival Time = 0, Burst Time = 7
- **Process P2**: Arrival Time = 2, Burst Time = 4
- **Process P3**: Arrival Time = 4, Burst Time = 1
- **Process P4**: Arrival Time = 5, Burst Time = 4
- Let's use a time quantum of 2 units.

EXECUTION

- We will use a Gantt chart to visualize the execution order of the processes.

1. **Time 0-2**: P1 (remaining burst time: 5)
2. **Time 2-4**: P2 (remaining burst time: 2)
3. **Time 4-5**: P3 (remaining burst time: 0, complete)
4. **Time 5-7**: P4 (remaining burst time: 2)
5. **Time 7-9**: P1 (remaining burst time: 3)
6. **Time 9-11**: P2 (remaining burst time: 0, complete)
7. **Time 11-13**: P4 (remaining burst time: 0, complete)
8. **Time 13-15**: P1 (remaining burst time: 1)
9. **Time 15-16**: P1 (remaining burst time: 0, complete)

Calculate Completion Time (CT), Turnaround Time (TAT), and Waiting Time (WT)

- **Turnaround Time (TAT)**: The total time taken from arrival to completion (TAT = CT - AT).
- **Waiting Time (WT)**: The total time a process spends waiting in the ready queue (WT = TAT - BT).

Process	Arrival Time (AT)	Burst Time (BT)	Completion Time (CT)	Turnaround Time (TAT)	Waiting Time (WT)
P1	0	7	16	16	9
P2	2	4	11	9	5
P3	4	1	5	1	0
P4	5	4	13	8	4

RR IMPLEMENTATION

RR.sh

Shortest Remaining Time First (SRTF)

- Shortest Remaining Time First (SRTF) is a preemptive version of the Shortest Job First (SJF) scheduling algorithm.
- In SRTF, the process with the smallest amount of remaining time is selected for execution.
- If a new process arrives with a burst time smaller than the remaining time of the currently running process, the current process is preempted, and the new process is scheduled

- **Example Scenario**
- Consider four processes with the following arrival and burst times:
- **Process P1**: Arrival Time = 0, Burst Time = 8
- **Process P2**: Arrival Time = 1, Burst Time = 4
- **Process P3**: Arrival Time = 2, Burst Time = 9
- **Process P4**: Arrival Time = 3, Burst Time = 5

Step-by-Step Calculation

1. **Time 0-1**: P1 starts execution (remaining time: 7)
2. **Time 1**: P2 arrives, P2 is scheduled because it has a smaller burst time (remaining time: 3)
3. **Time 2**: P3 arrives, P2 continues because it has less remaining time than P3 (remaining time: 2)
4. **Time 3**: P4 arrives, P2 continues because it has less remaining time than P4 (remaining time: 1)
5. **Time 4**: P2 completes, P4 is scheduled (remaining time: 4)
6. **Time 5-8**: P4 continues (remaining time: 3 to 0)
7. **Time 8**: P4 completes, P1 is scheduled (remaining time: 6)
8. **Time 9-13**: P1 continues (remaining time: 2)
9. **Time 14-17**: P1 completes, P3 is scheduled (remaining time: 6)
10. **Time 18-23**: P3 continues (remaining time: 0)

Completion Time (CT), Turnaround Time (TAT), and Waiting Time (WT)

Process	Arrival Time (AT)	Burst Time (BT)	Completion Time (CT)	Turnaround Time (TAT)	Waiting Time (WT)
P1	0	8	17	17 - 0 = 17	17 - 8 = 9
P2	1	4	5	5 - 1 = 4	4 - 4 = 0
P3	2	9	23	23 - 2 = 21	21 - 9 = 12
P4	3	5	8	8 - 3 = 5	5 - 5 = 0

SRTF Implementation

SRTF.sh

Priority scheduling

- Priority scheduling is a scheduling algorithm where each process is assigned a priority, and the process with the highest priority is selected for execution.

- If two processes have the same priority, they can be scheduled based on another criterion, such as First-Come, First-Served (FCFS).

Example

- **Example Scenario**
- Consider the following processes with their arrival times, burst times, and priorities:
- **Process P1**: Arrival Time = 0, Burst Time = 10, Priority = 3
- **Process P2**: Arrival Time = 2, Burst Time = 1, Priority = 1
- **Process P3**: Arrival Time = 1, Burst Time = 2, Priority = 4
- **Process P4**: Arrival Time = 3, Burst Time = 1, Priority = 5

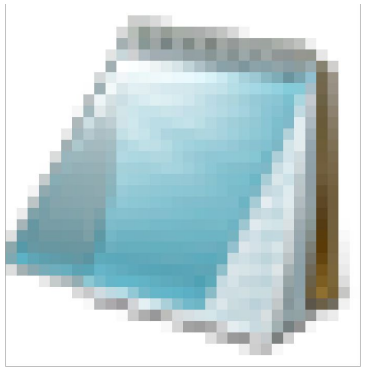
Step-by-Step Calculation

1. **Time 0**: P1 starts (P1 has the highest priority and is the only process available)
2. **Time 1**: P1 continues (P3 arrives but P1 has higher priority)
3. **Time 2**: P2 arrives with the highest priority (priority = 1) and preempts P1
4. **Time 3**: P2 completes, P3 with the next highest priority starts
5. **Time 4**: P3 continues
6. **Time 5**: P3 completes, P4 starts
7. **Time 6**: P4 completes, P1 continues
8. **Time 7-14**: P1 continues and completes

Completion Time (CT), Turnaround Time (TAT), and Waiting Time (WT)

Process	Arrival Time (AT)	Burst Time (BT)	Priority	Completion Time (CT)	Turnaround Time (TAT)	Waiting Time (WT)
P1	0	10	3	14	14 - 0 = 14	14 - 10 = 4
P2	2	1	1	3	3 - 2 = 1	1 - 1 = 0
P3	1	2	4	5	5 - 1 = 4	4 - 2 = 2
P4	3	1	5	6	6 - 3 = 3	3 - 1 = 2

PS IMPLEMENTATION



PS.sh

Week: 7
Deadlocks

Topics

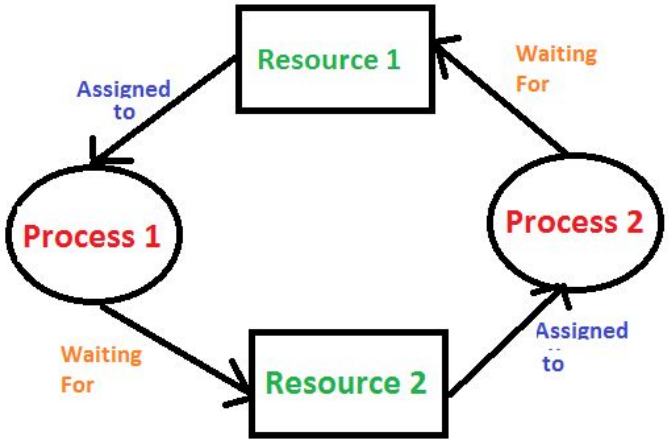
- Deadlock
  - Deadlock Modeling
  - Resource Graph
  - Detecting Deadlock
- Livelock
- Starvation

Book Link: https://drive.google.com/file/d/1GlVwWVPVl2RSm24bcEGzc2GqywGEzZQL/view

Chapter: 06

Introduction

- A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause.




```
void* fn1(void* args){
    pthread_mutex_lock(&m1);
    cout<<"Using a"<<endl;
    Sleep(5);
    cout<<"Waiting for b"<<endl;
    pthread_mutex_lock(&m2);
    cout<<"Using b"<<endl;
    Sleep(5);

    pthread_mutex_unlock(&m1);
    pthread_mutex_unlock(&m2);
}

void* fn2(void* args){
    pthread_mutex_lock(&m2);
    cout<<"Using b"<<endl;
    Sleep(5);
    cout<<"Waiting for a"<<endl;
    pthread_mutex_lock(&m1);
    cout<<"Using a"<<endl;
    Sleep(5);

    pthread_mutex_unlock(&m1);
    pthread_mutex_unlock(&m2);
}
```

```
"E:\MIST_L\1. MIST_2022\Fall 2
Using aUsing b

Waiting for b
Waiting for a
```

Conditions for Resource Deadlocks

- **Mutual exclusion:** Each resource is either currently assigned to exactly one process or is available.
- **Hold-and-wait:** Processes currently holding resources that were granted earlier can request new resources.
- **No-preemption:** Resources previously granted cannot be forcibly taken away from a process. They must be explicitly released by the process holding them.
- **Circular wait:** There must be a circular list of two or more processes, **each of which is waiting for a resource held by the next member of the chain**.

Deadlock Modeling

- Here A, B, C & D are process.
- And R, S, T, U are resources.
- Process are denoted using
- Resources are denoting using
- The arrow direction is different for both holding and requesting.

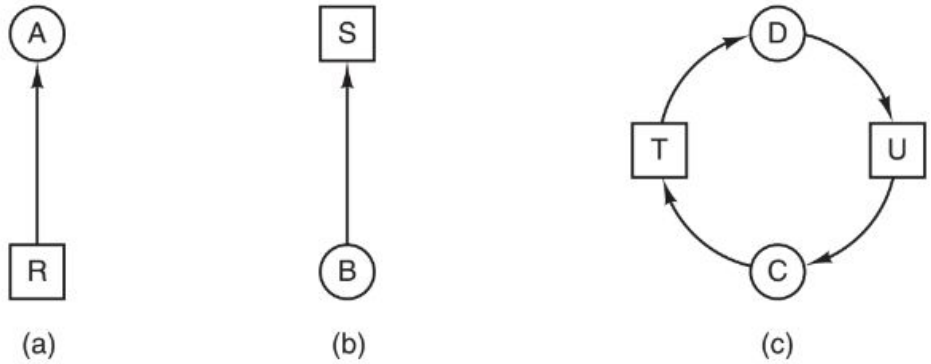


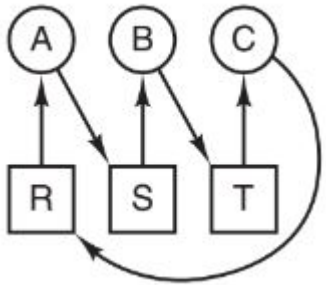
Figure 6-3. Resource allocation graphs. (a) Holding a resource. (b) Requesting a resource. (c) Deadlock.

Resource Graph

- **Resource graph** is a tool that lets us see if a given request/release sequence leads to deadlock.
  - Carry out the requests and releases step by step, and after every step we check the graph to see if it contains any cycles.
  - If **cycle**, then there is a **deadlock**.

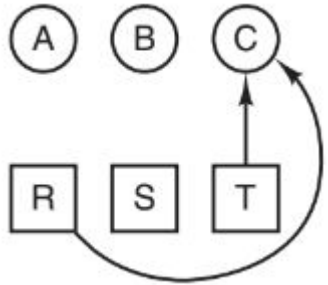
1. A requests R
2. B requests S
3. C requests T
4. A requests S
5. B requests T
6. C requests R

deadlock



1. A requests R
2. C requests T
3. A requests S
4. C requests R
5. A releases R
6. A releases S

no deadlock



Livelock

Politely release the lock so that other threads can use.

But others also doing the same thing.

```
void process_A(void) {
        acquire_lock(&resource_1);
        while (try_lock(&resource_2) == FAIL) {
                release_lock(&resource_1);
                wait_fixed_time();
                acquire_lock(&resource_1);
        }
        use_both_resources( );
        release_lock(&resource_2);
        release_lock(&resource_1);
}

void process_A(void) {
        acquire_lock(&resource_2);
        while (try_lock(&resource_1) == FAIL) {
                release_lock(&resource_2);
                wait_fixed_time();
                acquire_lock(&resource_2);
        }
        use_both_resources( );
        release_lock(&resource_1);
        release_lock(&resource_2);
}
```

Starvation

Some policy is needed to make a decision about who gets which resource when.

This policy, although seemingly reasonable, may lead to some processes never getting service even though they are not deadlocked.


thank
you

DEADLOCKS

FOUR STRATEGIES ARE USED FOR DEALING WITH DEADLOCKS

- Just ignore the problem. Maybe if you ignore it, it will ignore you.
- Detection and recovery. Let them occur, detect them, and take action.
- Dynamic avoidance by careful resource allocation.
- Prevention, by structurally negating one of the four conditions.

OSTRICH ALGORITHM

- The simplest approach is the ostrich algorithm: stick your head in the sand and pretend there is no problem.† People react to this strategy in different ways.
- Mathematicians find it unacceptable and say that deadlocks must be prevented at all costs.
- Engineers ask how often the problem is expected, how often the system crashes for other reasons, and how serious a deadlock is.
- If deadlocks occur on the average once every five years, but system crashes due to hardware failures and operating system bugs occur once a week, most engineers would not be willing to pay a large penalty in performance or convenience to eliminate deadlocks

DEADLOCK DETECTION AND RECOVERY

Deadlock Detection with One Resource of Each Type

- Process A holds R and wants S.
- Process B holds nothing but wants T.
- Process C holds nothing but wants S.
- Process D holds U and wants S and T.
- Process E holds T and wants V.
- Process F holds W and wants S.
- Process G holds V and wants U

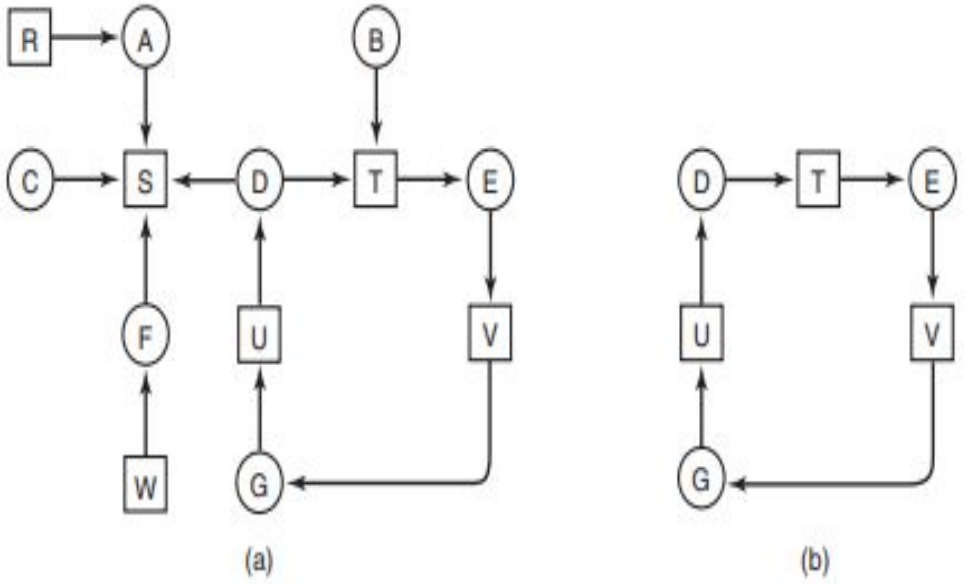


Figure 6-5. (a) A resource graph. (b) A cycle extracted from (a).

ALGORITHM FOR CYCLE DETECTION

The algorithm operates by carrying out the following steps as specified:

1. For each node, N, in the graph, perform the following fiv e steps with N as the starting node.
2. Initialize L to the empty list, and designate all the arcs as unmarked.
3. Add the current node to the end of L and check to see if the node now appears in L two times. If it does, the graph contains a cycle (listed in L) and the algorithm terminates.
4. From the given node, see if there are any unmarked outgoing arcs. If so, go to step 5; if not, go to step 6.
5. Pick an unmarked outgoing arc at random and mark it. Then follow it to the new current node and go to step 3.
6. If this node is the initial node, the graph does not contain any cycles and the algorithm terminates. Otherwise, we have now reached a dead end. Remove it and go back to the previous node, that is, the one that was current just before this one, make that one the current node, and go to step 3.

EXAMPLE

- We start at R and initialize L to the empty list.
- Then we add R to the list and move to the only possibility, A, and add it to L, giving L = [R, A].
- From A we go to S, giving L = [R, A, S]. S has no outgoing arcs, so it is a dead end, forcing us to backtrack to A. Since A has no unmarked outgoing arcs, we backtrack to R, completing our inspection of R.
- Now we restart the algorithm starting at A, resetting L to the empty list. This search, too, quickly stops, so we start again at B.
- From B we continue to follow outgoing arcs until we get to D, at which time L = [B, T, E, V, G, U, D]. Now we must make a (random) choice. If we pick S we come to a dead end and backtrack to D. The second time we pick T and update L to be [B, T, E, V, G, U, D, T], at which point we discover the cycle and stop the algorithm

DEADLOCK DETECTION WITH MULTIPLE RESOURCES OF EACH TYPE

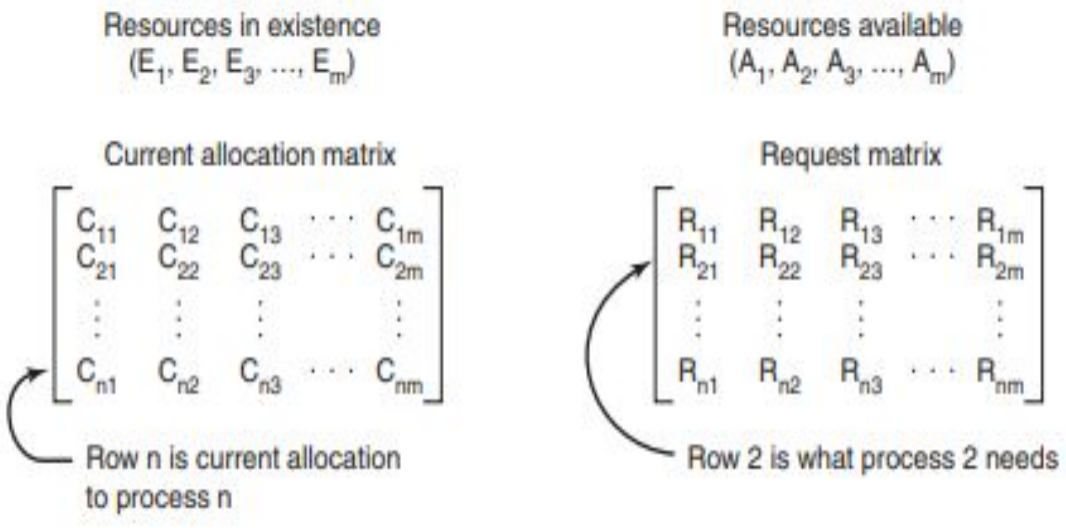


Figure 6-6. The four data structures needed by the deadlock detection algorithm.

The deadlock detection algorithm can now be given as follows.

1. Look for an unmarked process, Pi, for which the ith row of R is less than or equal to A.
2. If such a process is found, add the ith row of C to A, mark the process, and go back to step 1.
3. If no such process exists, the algorithm terminates.

EXAMPLE

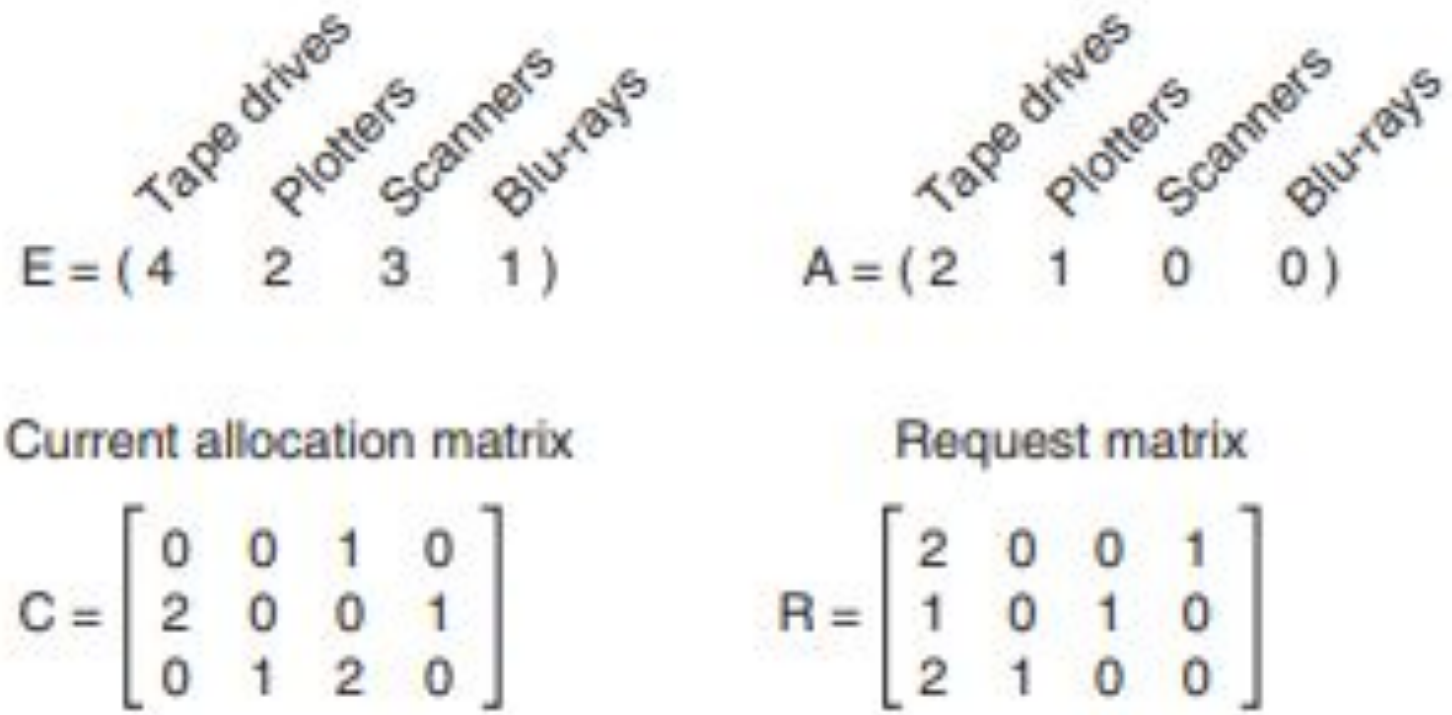


Figure 6-7. An example for the deadlock detection algorithm.

RECOVERY FROM DEADLOCK

- Recovery through Preemption
- Recovery through Rollback
- Recovery through Killing Processes

RECOVERY THROUGH PREEMPTION

- In some cases it may be possible to temporarily take a resource away from its current owner and give it to another process.
- In many cases, manual intervention may be required, especially in batch-processing operating systems running on mainframes.
- For example, to take a laser printer away from its owner, the operator can collect all the sheets already printed and put them in a pile. Then the process can be suspended (marked as not runnable).
- At this point the printer can be assigned to another process. When that process finishes, the pile of printed sheets can be put back in the printer's output tray and the original process restarted.
- The ability to take a resource away from a process, have another process use it, and then give it back without the process noticing it is highly dependent on the nature of the resource. Recovering this way is frequently difficult or impossible. Choosing the process to suspend depends largely on which ones have resources that can easily be taken back.

RECOVERY THROUGH ROLLBACK

- If the system designers and machine operators know that deadlocks are likely, they can arrange to have processes checkpointed periodically.
- Checkpointing a process means that its state is written to a file so that it can be restarted later.
- The checkpoint contains not only the memory image, but also the resource state, in other words, which resources are currently assigned to the process.
- To be most effective, new checkpoints should not overwrite old ones but should be written to new files, so as the process executes, a whole sequence accumulates. When a deadlock is detected, it is easy to see which resources are needed.
- To do the recovery, a process that owns a needed resource is rolled back to a point in time before it acquired that resource by starting at one of its earlier checkpoints. All the work done since the checkpoint is lost (e.g., output printed since the checkpoint must be discarded, since it will be printed again).
- In effect, the process is reset to an earlier moment when it did not have the resource, which is now assigned to one of the deadlocked processes. If the restarted process tries to acquire the resource again, it will have to wait until it becomes available.

RECOVERY THROUGH KILLING PROCESSES

- The crudest but simplest way to break a deadlock is to kill one or more processes. One possibility is to kill a process in the cycle. With a little luck, the other processes will be able to continue. If this does not help, it can be repeated until the cycle is broken.
- Where possible, it is best to kill a process that can be rerun from the beginning with no ill effects. For example, a compilation can always be rerun because all it does is read a source file and produce an object file. If it is killed partway through, the first run has no influence on the second run.
- On the other hand, a process that updates a database cannot always be run a second time safely. If the process adds 1 to some field of a table in the database, running it once, killing it, and then running it again will add 2 to the field, which is incorrect.

DEADLOCK AVOIDANCE

- In the discussion of deadlock detection, we tacitly assumed that when a process asks for resources, it asks for them all at once.
- In most systems, however, resources are requested one at a time. The system must be able to decide whether granting a resource is safe or not and make the allocation only when it is safe.
- Thus, the question arises: Is there an algorithm that can always avoid deadlock by making the right choice all the time? The answer is a qualified yes—we can avoid deadlocks, but only if certain information is available in advance

SAFE AND UNSAFE STATES

- At any instant of time, there is a current state consisting of E, A, C, and R. A state is said to be safe if there is some scheduling order in which every process can run to completion even if all of them suddenly request their maximum number of resources immediately.
- we have a state in which A has three instances of the resource but may need as many as nine eventually. B currently has two and may need four altogether, later. Similarly, C also has two but may need an additional fiv e. A total of 10 instances of the resource exist, so with seven resources already allocated, three there are still free

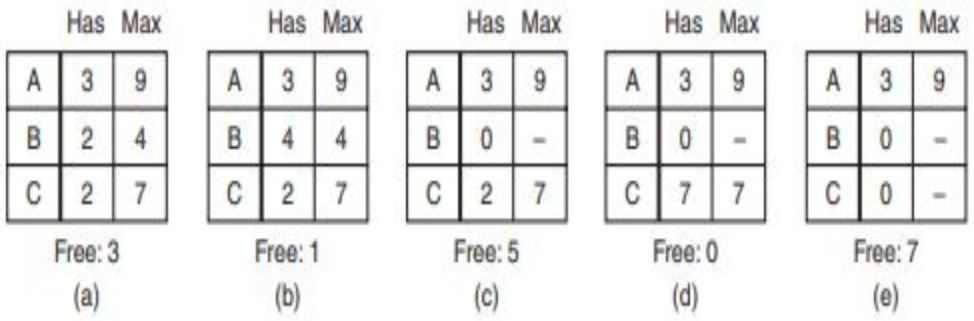


Figure 6-9. Demonstration that the state in (a) is safe.

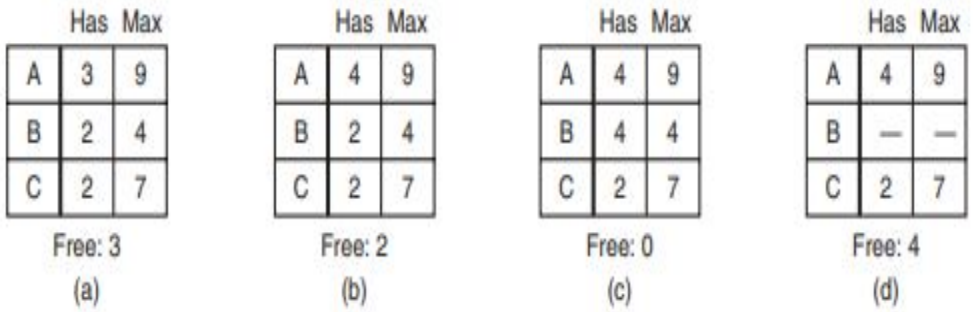
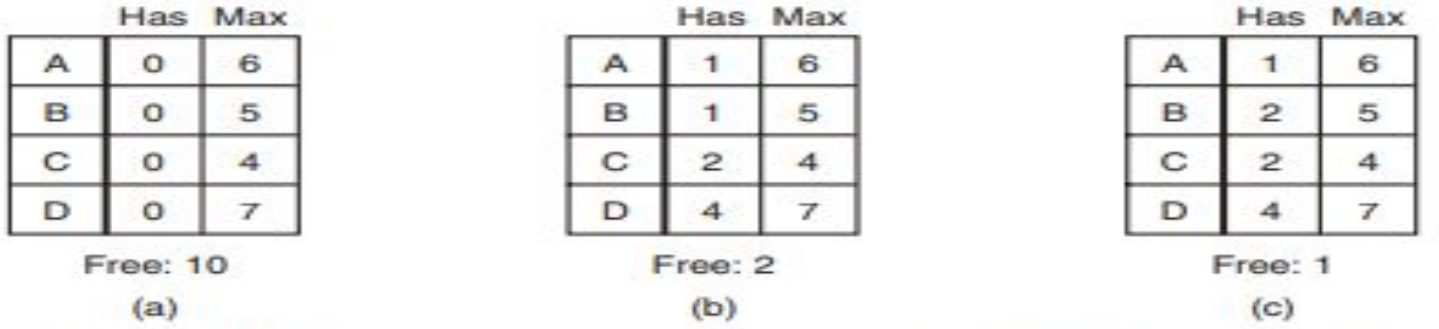


Figure 6-10. Demonstration that the state in (b) is not safe.

THE BANKER'S ALGORITHM FOR A SINGLE RESOURCE

- The banker's algorithm considers each request as it occurs, seeing whether granting it leads to a safe state.
- If it does, the request is granted; otherwise, it is postponed until later. To see if a state is safe, the banker checks to see if he has enough resources to satisfy some customer.
- If so, those loans are assumed to be repaid, and the customer now closest to the limit is checked, and so on. If all loans can eventually be repaid, the state is safe and the initial request can be granted.



	Has	Max
A	0	6
B	0	5
C	0	4
D	0	7

Free: 10

(a)

	Has	Max
A	1	6
B	1	5
C	2	4
D	4	7

Free: 2

(b)

	Has	Max
A	1	6
B	2	5
C	2	4
D	4	7

Free: 1

(c)

Figure 6-11. Three resource allocation states: (a) Safe. (b) Safe. (c) Unsafe.

THE BANKER'S ALGORITHM FOR MULTIPLE RESOURCES

The algorithm for checking to see if a state is safe can now be stated.

1. Look for a row, R, whose unmet resource needs are all smaller than or equal to A. If no such row exists, the system will eventually deadlock since no process can run to completion (assuming processes keep all resources until they exit).
2. Assume the process of the chosen row requests all the resources it needs (which is guaranteed to be possible) and finishes. Mark that process as terminated and add all of its resources to the A vector.
3. Repeat steps 1 and 2 until either all processes are marked terminated (in which case the initial state was safe) or no process is left whose resource needs can be met (in which case the system was not safe).

EXAMPLE

Process	Tape drives	Plotters	Printers	Blu-rays
A	3	0	1	1
B	0	1	0	0
C	1	1	1	0
D	1	1	0	1
E	0	0	0	0

Resources assigned

Process	Tape drives	Plotters	Printers	Blu-rays
A	1	1	0	0
B	0	1	1	2
C	3	1	0	0
D	0	0	1	0
E	2	1	1	0

Resources still assigned

E = (6342)
P = (5322)
A = (1020)

Figure 6-12. The banker's algorithm with multiple resources.

DEADLOCK PREVENTION

- Having seen that deadlock avoidance is essentially impossible, because it requires information about future requests, which is not known, how do real systems avoid deadlock?
- The answer is to go back to the four conditions to see if they can provide a clue.
- If we can ensure that at least one of these conditions is never satisfied, then deadlocks will be structurally impossible.

ATTACKING THE MUTUAL-EXCLUSION CONDITION

- First let us attack the mutual exclusion condition. If no resource were ever assigned exclusively to a single process, we would never have deadlocks.
- By spooling printer output, several processes can generate output at the same time. In this model, the only process that actually requests the physical printer is the printer daemon. Since the daemon never requests any other resources, we can eliminate deadlock for the printer.
- If the daemon is programmed to begin printing even before all the output is spooled, the printer might lie idle if an output process decides to wait several hours after the first burst of output. For this reason, daemons are normally programmed to print only after the complete output file is available. However, this decision itself could lead to deadlock. What would happen if two processes each filled up one half of the available spooling space with output and neither was finished producing its full output? In this case, we would have two processes that had each finished part, but not all, of their output, and could not continue. Neither process will ever finish, so we would have a deadlock on the disk.

ATTACKING THE HOLD-AND-WAIT CONDITION

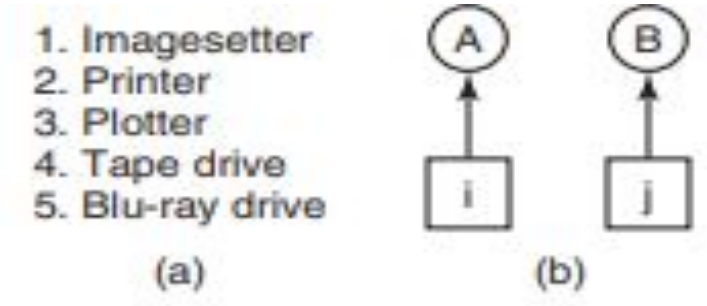
- If we can prevent processes that hold resources from waiting for more resources, we can eliminate deadlocks. One way to achieve this goal is to require all processes to request all their resources before starting execution. If everything is available, the process will be allocated whatever it needs and can run to completion. If one or more resources are busy, nothing will be allocated and the process will just wait.
- An immediate problem with this approach is that many processes do not know how many resources they will need until they have started running. In fact, if they knew, the banker's algorithm could be used.
- Another problem is that resources will not be used optimally with this approach. Take, as an example, a process that reads data from an input tape, analyzes it for an hour, and then writes an output tape as well as plotting the results. If all resources must be requested in advance, the process will tie up the output tape drive and the plotter for an hour.
- Nevertheless, some mainframe batch systems require the user to list all the resources on the first line of each job. The system then pre-allocates all resources immediately and does not release them until they are no longer needed by the job (or in the simplest case, until the job finishes). While this method puts a burden on the programmer and wastes resources, it does prevent deadlocks. A slightly different way to break the hold-and-wait condition is to require a process requesting a resource to first temporarily release all the resources it currently holds. Then it tries to get everything it needs all at once

ATTACKING THE NO-PREEMPTION CONDITION

- Attacking the third condition (no preemption) is also a possibility. If a process has been assigned the printer and is in the middle of printing its output, forcibly taking away the printer because a needed plotter is not available is tricky at best and impossible at worst.
- However, some resources can be virtualized to avoid this situation. Spooling printer output to the disk and allowing only the printer daemon access to the real printer eliminates deadlocks involving the printer, although it creates a potential for deadlock over disk space.
- With large disks though, running out of disk space is unlikely. However, not all resources can be virtualized like this. For example, records in databases or tables inside the operating system must be locked to be used and therein lies the potential for deadlock

ATTACKING THE CIRCULAR WAIT CONDITION

- One way is simply to have a rule saying that a process is entitled only to a single resource at any moment. If it needs a second one, it must release the first one. For a process that needs to copy a huge file from a tape to a printer, this restriction is unacceptable.
- Another way to avoid the circular wait is to provide a global numbering of all the resources.Now the rule is this: processes can request 6 resources whenever they want to, but all requests must be made in numerical order. A process may request first a printer and then a tape drive, but it may not request first a plotter and then a printer.



DEADLOCK PREVENTION

Condition	Approach
Mutual exclusion	Spool everything
Hold and wait	Request all resources initially
No preemption	Take resources away
Circular wait	Order resources numerically

Figure 6-14. Summary of approaches to deadlock prevention.

TWO-PHASE LOCKING

- Although both avoidance and prevention are not terribly promising in the general case, for specific applications, many excellent special-purpose algorithms are known.
- As an example, in many database systems, an operation that occurs frequently is requesting locks on several records and then updating all the locked records. When multiple processes are running at the same time, there is a real danger of deadlock
- The approach often used is called two-phase locking. In the first phase, the process tries to lock all the records it needs, one at a time. If it succeeds, it begins the second phase, performing its updates and releasing the locks. No real work is done in the first phase.
- If during the first phase, some record is needed that is already locked, the process just releases all its locks and starts the first phase all over. In a certain sense, this approach is similar to requesting all the resources needed in advance, or at least before anything irreversible is done. In some versions of two-phase locking, there is no release and restart if a locked record is encountered during the first phase. In these versions, deadlock can occur.

COMMUNICATION DEADLOCKS

- Another kind of deadlock can occur in communication systems (e.g., networks), in which two or more processes communicate by sending messages.
- A common arrangement is that process A sends a request message to process B, and then blocks until B sends back a reply message. Suppose that the request message gets lost.
- A is blocked waiting for the reply. B is blocked waiting for a request asking it to do something. We have a deadlock.
- This, though, is not the classical resource deadlock. A does not have possession of some resource B wants, and vice versa. In fact, there are no resources at all in sight. But it is a deadlock according to our formal definition since we have a set of (two) processes, each blocked waiting for an event only the other one can cause. This situation is called a communication deadlock.

COMMUNICATION DEADLOCKS

- Technique that can usually be employed to break communication deadlocks: timeouts.
- In most network communication systems, whenever a message is sent to which a reply is expected, a timer is started. If the timer goes off before the reply arrives, the sender of the message assumes that the message has been lost and sends it again (and again and again if needed).
- In this way, the deadlock is broken. Phrased differently, the timeout serves as a heuristic to detect deadlocks and enables recovery. This heuristic is applicable to resource deadlock also and is relied upon by users with temperamental or buggy device drivers that can deadlock and freeze the system.
- Of course, if the original message was not lost but the reply was simply delayed, the intended recipient may get the message two or more times, possibly with undesirable consequences.
- Think about an electronic banking system in which the message contains instructions to make a payment. Clearly, that should not be repeated (and executed) multiple times just because the network is slow or the timeout too short.

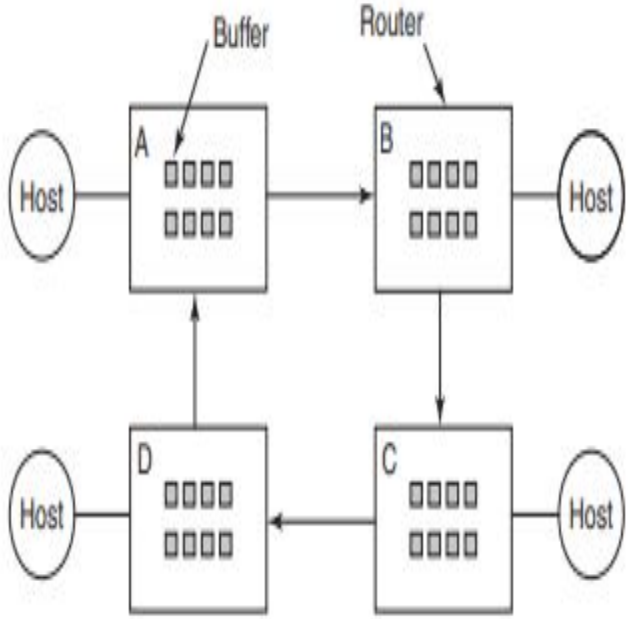
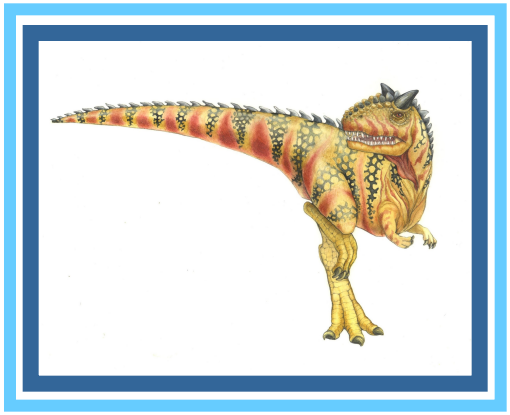


Figure 6-15. A resource deadlock in a network.

Chapter 16: Virtual Machines



Chapter 16: Virtual Machines

- Overview
- History
- Benefits and Features
- Building Blocks
- Types of Virtual Machines and Their Implementations
- Virtualization and Operating-System Components
- Examples

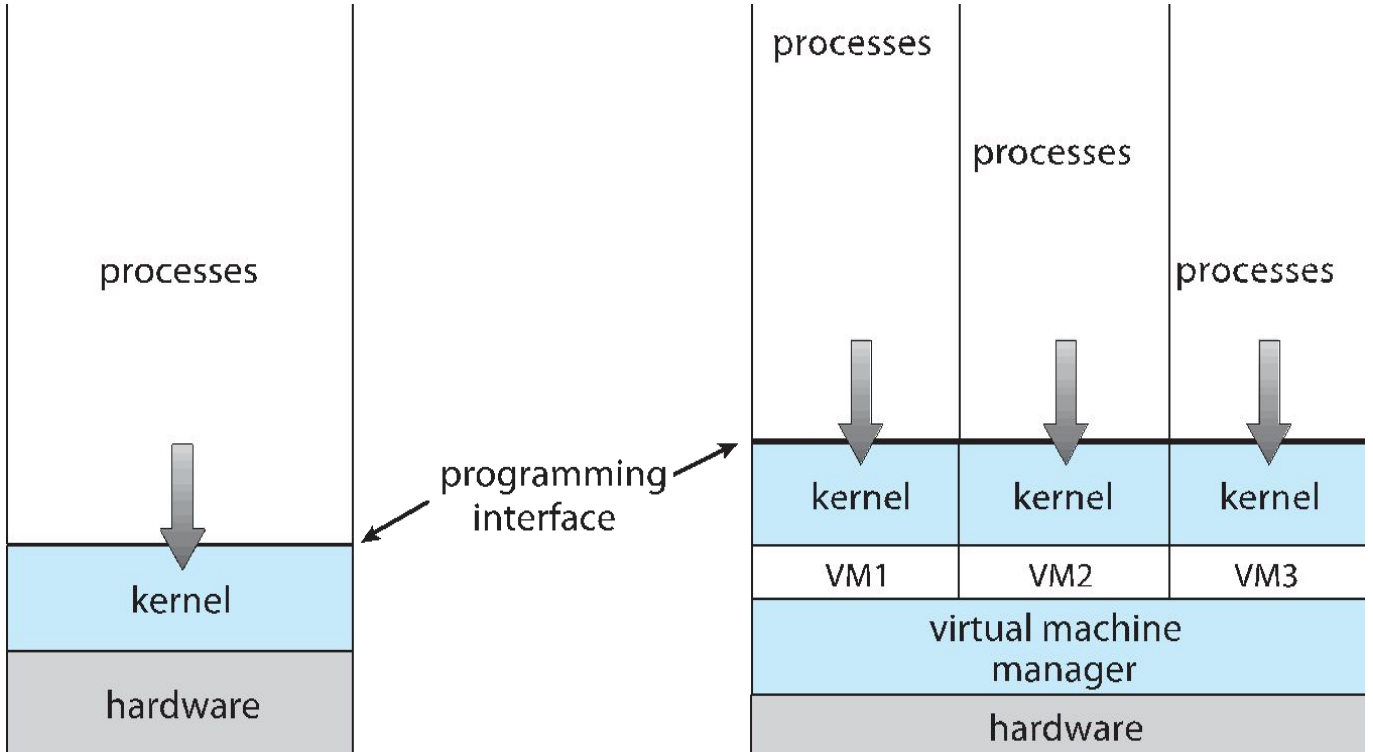
Chapter Objectives

- To explore the history and benefits of virtual machines
- To discuss the various virtual machine technologies
- To describe the methods used to implement virtualization
- To show the most common hardware features that support virtualization and explain how they are used by operating-system modules

Overview

- Fundamental idea – abstract hardware of a single computer into several different execution environments
  - Similar to layered approach
  - But layer creates virtual system (**virtual machine**, or **VM**) on which operation systems or applications can run
- Several components
  - **Host** – underlying hardware system
  - **Virtual machine manager** (**VMM**) or **hypervisor** – creates and runs virtual machines by providing interface that is *identical* to the host
    - (Except in the case of paravirtualization)
  - **Guest** – process provided with virtual copy of the host
    - Usually an operating system
- Single physical machine can run multiple operating systems concurrently, each in its own virtual machine

System Models



Non-virtual machine

Virtual machine

Implementation of VMMs

- Vary greatly, with options including:
  - **Type 0 hypervisors** - Hardware-based solutions that provide support for virtual machine creation and management via firmware
    - IBM LPARs and Oracle LDOMs are examples
  - **Type 1 hypervisors** - Operating-system-like software built to provide virtualization
    - Including VMware ESX, Joyent SmartOS, and Citrix XenServer
  - **Type 1 hypervisors** – Also includes general-purpose operating systems that provide standard functions as well as VMM functions
    - Including Microsoft Windows Server with HyperV and RedHat Linux with KVM
  - **Type 2 hypervisors** - Applications that run on standard operating systems but provide VMM features to guest operating systems
    - Includeing VMware Workstation and Fusion, Parallels Desktop, and Oracle VirtualBox

Implementation of VMMs

Criteria	Type 1 hypervisor	Type 2 hypervisor
AKA	Bare-metal or Native	Hosted
Definition	Runs directly on the system with VMs running on them	Runs on a conventional Operating System
Virtualization	Hardware Virtualization	OS Virtualization
Operation	Guest OS and applications run on the hypervisor	Runs as an application on the host OS
Scalability	Better Scalability	Not so much, because of its reliance on the underlying OS.
Setup/Installation	Simple, as long as you have the necessary hardware support	Lot simpler setup, as you already have an Operating System.
System Independence	Has direct access to hardware along with virtual machines it hosts	Are not allowed to directly access the host hardware and its resources
Speed	Faster	Slower because of the system's dependency
Performance	Higher-performance as there's no middle layer	Comparatively has reduced performance rate as it runs with extra overhead
Security	More Secure	Less Secure, as any problem in the base operating system affects the entire system including the protected Hypervisor
Examples	• VMware ESXi • Microsoft Hyper-V • Citrix XenServer	• VMware Workstation Player • Microsoft Virtual PC • Sun's VirtualBox

Implementation of VMMs

Type 0 Hypervisor

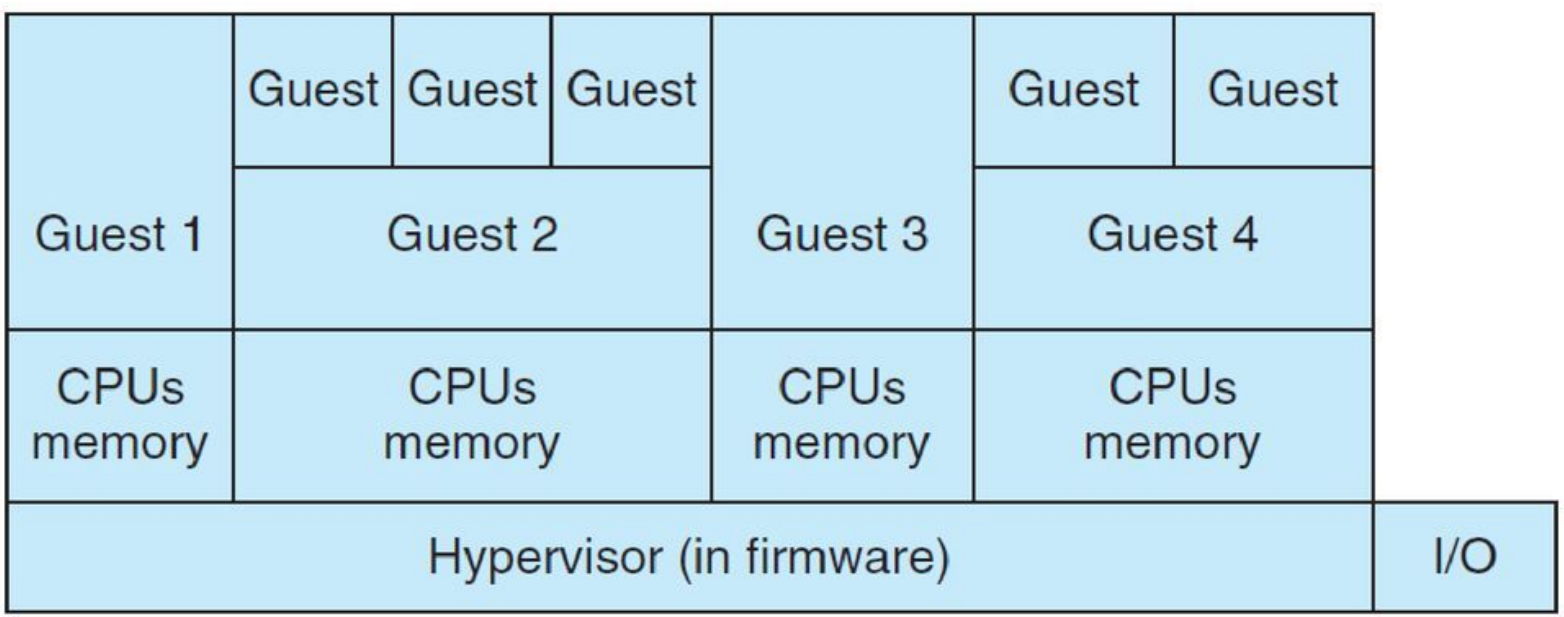
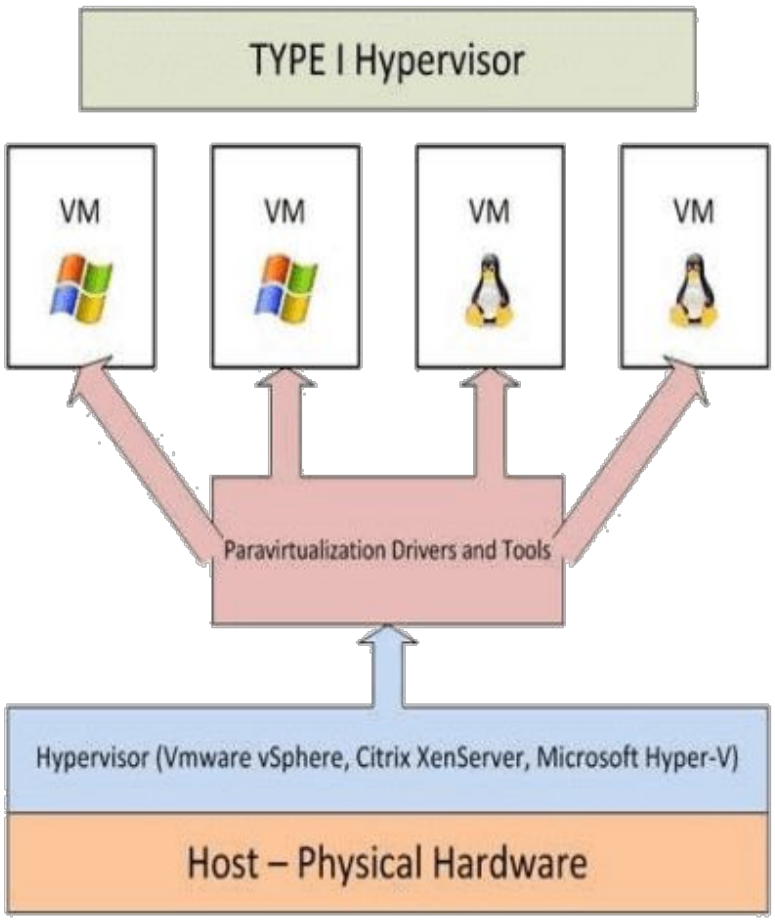


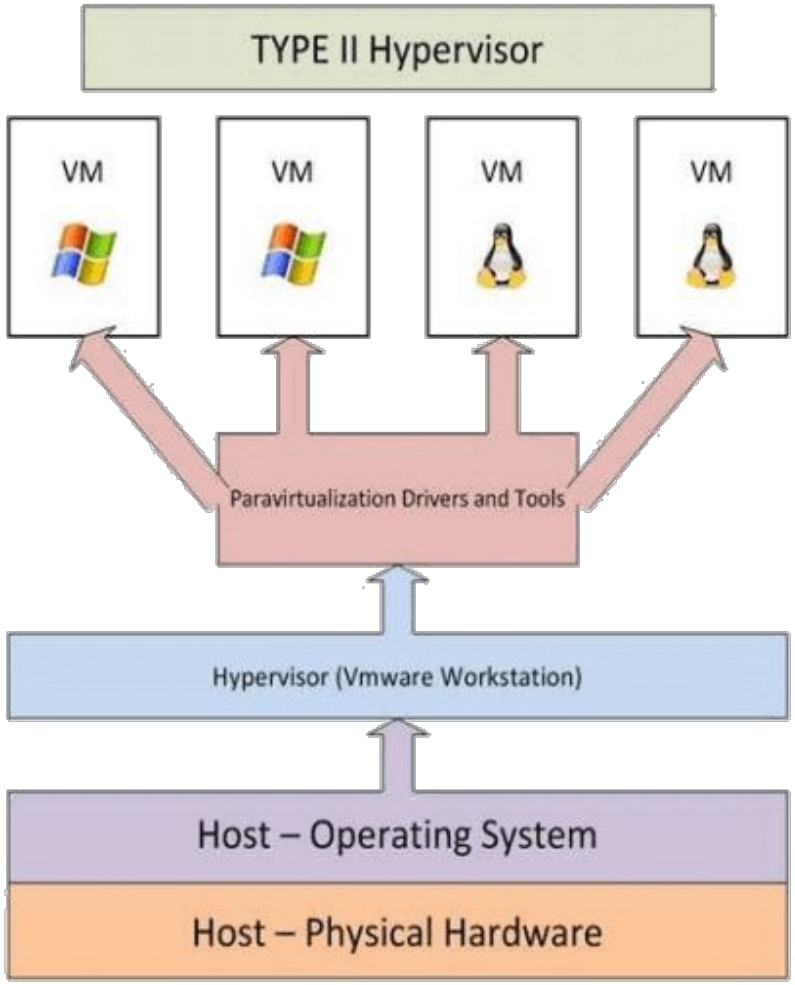
Figure 16.5 Type 0 hypervisor.

Other types of hypervisors usually cannot provide this virtualization-within-virtualization functionality.

Implementation of VMMs

Hypervisor





Implementation of VMMs (cont.)

- Other variations include:
  - **Paravirtualization** - Technique in which the guest operating system is modified to work in cooperation with the VMM to optimize performance
  - **Programming-environment virtualization** - VMMs do not virtualize real hardware but instead create an optimized virtual system
    - Used by Oracle Java and Microsoft.Net
  - **Emulators** – Allow applications written for one hardware environment to run on a very different hardware environment, such as a different type of CPU
  - **Application containment** - Not virtualization at all but rather provides virtualization-like features by segregating applications from the operating system, making them more secure, manageable
    - Including Oracle Solaris Zones, BSD Jails, and IBM AIX WPARs
- Much variation due to breadth, depth and importance of virtualization in modern computing

History

- First appeared in IBM mainframes in 1972
- Allowed multiple users to share a batch-oriented system
- Formal definition of virtualization helped move it beyond IBM
  1. A VMM provides an environment for programs that is essentially identical to the original machine
  2. Programs running within that environment show only minor performance decreases
  3. The VMM is in complete control of system resources
- In late 1990s Intel CPUs fast enough for researchers to try virtualizing on general purpose PCs
  - **Xen** and **VMware** created technologies, still used today
  - Virtualization has expanded to many OSes, CPUs, VMMs

Benefits and Features

- Host system protected from VMs, VMs protected from each other
  - I.e. A virus less likely to spread
  - Sharing is provided though via shared file system volume, network communication
- Freeze, **suspend**, running VM
  - Then can move or copy somewhere else and **resume**
  - Snapshot of a given state, able to restore back to that state
    - Some VMMs allow multiple snapshots per VM
  - **Clone** by creating copy and running both original and copy
- Great for OS research, better system development efficiency
- Run multiple, different OSes on a single machine
  - **Consolidation**, app dev, …

A VMware snapshot is a copy of the virtual machine's disk file (VMDK) at a given point in time. Snapshots provide a change log for the virtual disk and are used to restore a VM to a particular point in time when a failure or system error occurs.

Benefits and Features (cont.)

- **Templating** – create an OS + application VM, provide it to customers, use it to create multiple instances of that combination
- **Live migration** – move a running VM from one host to another!
  - No interruption of user access
- All those features taken together -> **cloud computing**
  - Using APIs, programs tell cloud infrastructure (servers, networking, storage) to create new guests, VMs, virtual desktops

Trap-and-Emulate

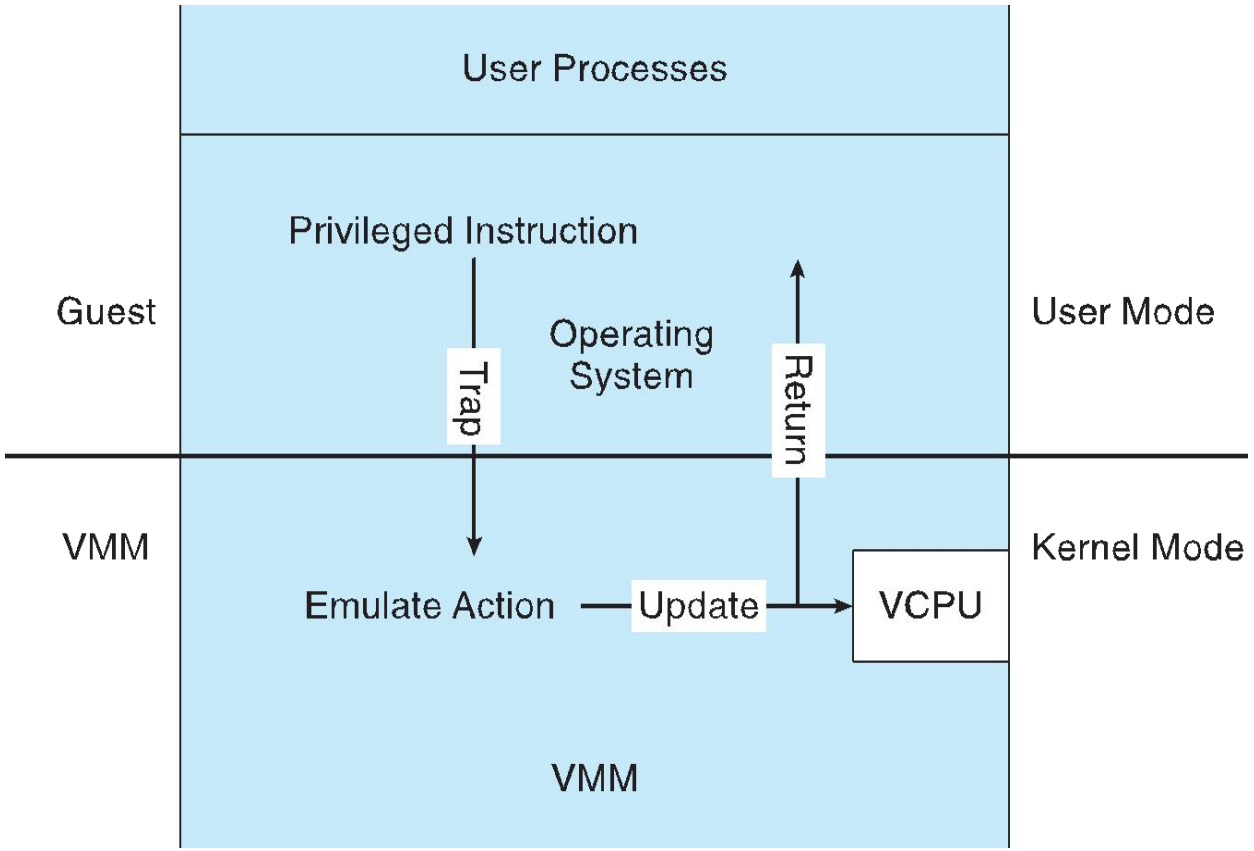
Trap and emulate is a technique that takes the basic of the emulation but improves performance by using interpretation selectively.

In this method also, both the user applications and guest operating system of virtual machines run in the user mode and the hypervisor runs in the privileged mode.

Trap-and-Emulate (cont.)

- How does switch from virtual user mode to virtual kernel mode occur?
  - Attempting a privileged instruction in user mode causes an error -> trap
  - VMM gains control, analyzes error, executes operation as attempted by guest
  - Returns control to guest in user mode
  - Known as **trap-and-emulate**
  - Most virtualization products use this at least in part
- User mode code in guest runs at same speed as if not a guest
- But kernel mode privilege mode code runs slower due to trap-and-emulate
  - Especially a problem when multiple guests running, each needing trap-and-emulate
- CPUs adding hardware support, mode CPU modes to improve virtualization performance

Trap-and-Emulate Virtualization Implementation

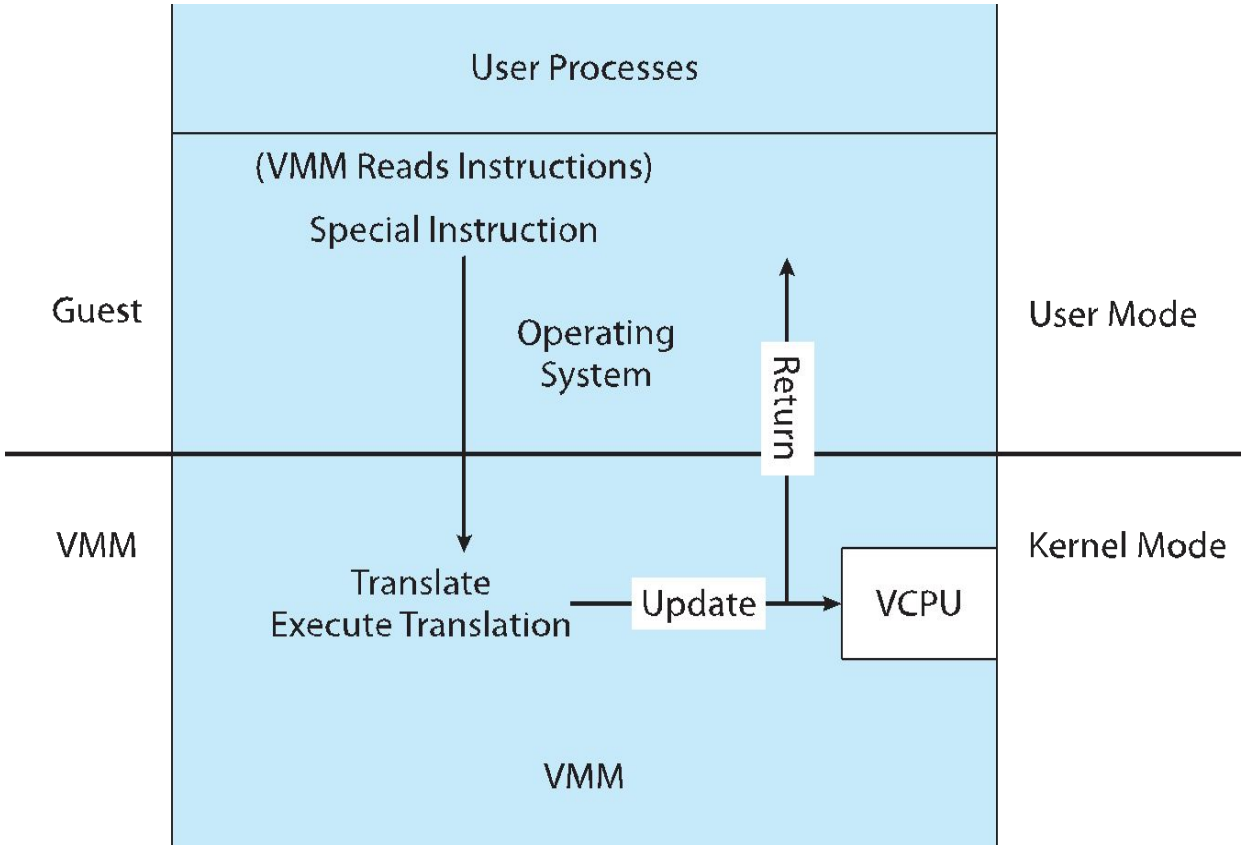


Binary Translation (cont.)

- Binary translation solves the problem
  - Basics are simple, but implementation very complex
  1. If guest VCPU is in user mode, guest can run instructions natively
  2. If guest VCPU in kernel mode (guest believes it is in kernel mode)
     1. VMM examines every instruction guest is about to execute by reading a few instructions ahead of program counter
     2. Non-special-instructions run natively
     3. Special instructions translated into new set of instructions that perform equivalent task (for example changing the flags in the VCPU)

Full virtualization does not need to modify the host OS. It relies on binary translation to trap and to virtualize the execution of certain sensitive, nonvirtualizable instructions. The guest OSes and their applications consist of noncritical and critical instructions.

Binary Translation Virtualization Implementation



Types of Virtual Machines and Implementations

- Many variations as well as HW details
  - Assume VMMs take advantage of HW features
    - 4 HW features can simplify implementation, improve performance
- Whatever the type, a VM has a lifecycle
  - Created by VMM
  - Resources assigned to it (number of cores, amount of memory, networking details, storage details)
  - In type 0 hypervisor, resources usually dedicated
  - Other types dedicate or share resources, or a mix
  - When no longer needed, VM can be deleted, freeing resouces
- Steps simpler, faster than with a physical machine install
  - Can lead to **virtual machine sprawl** with lots of VMs, history and state difficult to track

Types of VMs – Type 0 Hypervisor

- Old idea, under many names by HW manufacturers
  - "partitions", "domains"
  - A HW feature implemented by firmware
  - OS need to nothing special, VMM is in firmware
  - Smaller feature set than other types
  - Each guest has dedicated HW
- I/O a challenge as difficult to have enough devices, controllers to dedicate to each guest
- Sometimes VMM implements a **control partition** running daemons that other guests communicate with for shared I/O
- Can provide virtualization-within-virtualization (guest itself can be a VMM with guests
  - Other types have difficulty doing this

Types of VMs – Type 1 Hypervisor

- Commonly found in company datacenters
  - In a sense becoming "datacenter operating systems"
    - Datacenter managers control and manage OSes in new, sophisticated ways by controlling the Type 1 hypervisor
    - Consolidation of multiple OSes and apps onto less HW
    - Move guests between systems to balance performance
    - Snapshots and cloning
- Special purpose operating systems that run natively on HW
  - Rather than providing system call interface, create run and manage guest OSes
  - Can run on Type 0 hypervisors but not on other Type 1s
  - Run in kernel mode
  - Guests generally don't know they are running in a VM
  - Implement device drivers for host HW because no other component can
  - Also provide other traditional OS services like CPU and memory management

Types of VMs – Type 1 Hypervisor (cont.)

- Another variation is a general purpose OS that also provides VMM functionality
  - RedHat Enterprise Linux with KVM, Windows with Hyper-V, Oracle Solaris
  - Perform normal duties as well as VMM duties
  - Typically less feature rich than dedicated Type 1 hypervisors
- In many ways, treat guests OSes as just another process
  - Albeit with special handling when guest tries to execute special instructions

Types of VMs – Type 2 Hypervisor

- Less interesting from an OS perspective
  - Very little OS involvement in virtualization
  - VMM is simply another process, run and managed by host
    - Even the host doesn't know they are a VMM running guests
  - Tend to have poorer overall performance because can't take advantage of some HW features
  - But also a benefit because require no changes to host OS
    - Student could have Type 2 hypervisor on native host, run multiple guests, all on standard host OS such as Windows, Linux, MacOS

Types of VMs – Paravirtualization

- Does not fit the definition of virtualization – VMM not presenting an exact duplication of underlying hardware
  - But still useful!
  - VMM provides services that guest must be modified to use
  - Leads to increased performance
  - Less needed as hardware support for VMs grows
- Xen, leader in paravirtualized space, adds several techniques
  - For example, clean and simple device abstractions
    - Efficient I/O
    - Good communication between guest and VMM about device I/O
    - Each device has circular buffer shared by guest and VMM via shared memory

Types of VMs – Paravirtualization (cont.)

- Xen, leader in paravirtualized space, adds several techniques (Cont.)
  - Memory management does not include nested page tables
    - Each guest has own read-only tables
    - Guest uses **hypercall** (call to hypervisor) when page-table changes needed
- Paravirtualization allowed virtualization of older x86 CPUs (and others) without binary translation
- Guest had to be modified to use run on paravirtualized VMM
- But on modern CPUs Xen no longer requires guest modification -> no longer paravirtualization

Types of VMs – Programming Environment Virtualization

- Also not-really-virtualization but using same techniques, providing similar features
- Programming language is designed to run within custom-built virtualized environment
  - For example Oracle Java has many features that depend on running in **Java Virtual Machine** (**JVM**)
- In this case virtualization is defined as providing APIs that define a set of features made available to a language and programs written in that language to provide an improved execution environment
- JVM compiled to run on many systems (including some smart phones even)
- Programs written in Java run in the JVM no matter the underlying system
- Similar to **interpreted languages**

Types of VMs – Emulation

- Another (older) way for running one operating system on a different operating system
  - Virtualization requires underlying CPU to be same as guest was compiled for
  - Emulation allows guest to run on different CPU
- Necessary to translate all guest instructions from guest CPU to native CPU
  - Emulation, not virtualization
- Useful when host system has one architecture, guest compiled for other architecture
  - Company replacing outdated servers with new servers containing different CPU architecture, but still want to run old applications
- Performance challenge – order of magnitude slower than native code
  - New machines faster than older machines so can reduce slowdown
- Very popular – especially in gaming where old consoles emulated on new

Types of VMs – Application Containment

- Some goals of virtualization are segregation of apps, performance and resource management, easy start, stop, move, and management of them
- Can do those things without full-fledged virtualization
  - If applications compiled for the host operating system, don't need full virtualization to meet these goals
- Oracle **containers** / **zones** for example create virtual layer between OS and apps
  - Only one kernel running – host OS
  - OS and devices are virtualized, providing resources within zone with impression that they are only processes on system
  - Each zone has its own applications; networking stack, addresses, and ports; user accounts, etc
  - CPU and memory resources divided between zones
    4. Zone can have its own scheduler to use those resources

OS Component – CPU Scheduling

- Even single-CPU systems act like multiprocessor ones when virtualized
  - One or more virtual CPUs per guest
- Generally VMM has one or more physical CPUs and number of threads to run on them
  - Guests configured with certain number of VCPUs
    - Can be adjusted throughout life of VM
  - When enough CPUs for all guests -> VMM can allocate dedicated CPUs, each guest much like native operating system managing its CPUs
  - Usually not enough CPUs -> CPU **overcommitment**
    - VMM can use standard scheduling algorithms to put threads on CPUs
    - Some add fairness aspect

OS Component – CPU Scheduling (cont.)

- Cycle stealing by VMM and oversubscription of CPUs means guests don't get CPU cycles they expect
  - Consider timesharing scheduler in a guest trying to schedule 100ms time slices -> each may take 100ms, 1 second, or longer
    - Poor response times for users of guest
    - Time-of-day clocks incorrect
  - Some VMMs provide application to run in each guest to fix time-of-day and provide other integration features

OS Component – Memory Management

- Also suffers from oversubscription -> requires extra management efficiency from VMM
- For example, VMware ESX guests have a configured amount of physical memory, then ESX uses 3 methods of memory management
  1. Double-paging, in which the guest page table indicates a page is in a physical frame but the VMM moves some of those pages to backing store
  2. Install a **pseudo-device driver** in each guest (it looks like a device driver to the guest kernel but really just adds kernel-mode code to the guest)
     - **Balloon** memory manager communicates with VMM and is told to allocate or deallocate memory to decrease or increase physical memory use of guest, causing guest OS to free or have more memory available
  3. Deduplication by VMM determining if same page loaded more than once, memory mapping the same page into multiple guests

OS Component – I/O

- Easier for VMMs to integrate with guests because I/O has lots of variation
  - Already somewhat segregated / flexible via device drivers
  - VMM can provide new devices and device drivers
- But overall I/O is complicated for VMMs
  - Many short paths for I/O in standard OSes for improved performance
  - Less hypervisor needs to do for I/O for guests, the better
  - Possibilities include direct device access, DMA pass-through, direct interrupt delivery
    - Again, HW support needed for these
- Networking also complex as VMM and guests all need network access
  - VMM can **bridge** guest to network (allowing direct access)
  - And / or provide **network address translation** (**NAT**)
    - NAT address local to machine on which guest is running, VMM provides address translation to guest to hide its address

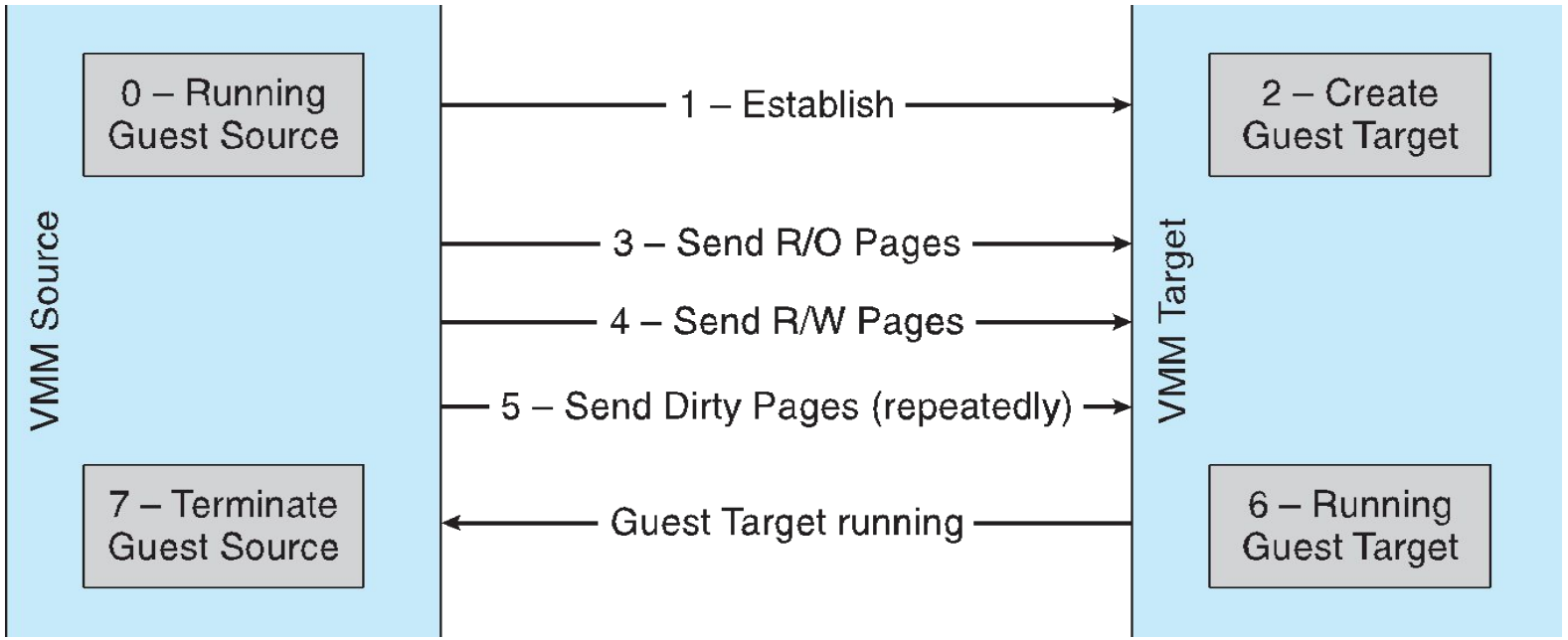
OS Component – Storage Management

- Both boot disk and general data access need be provided by VMM
- Need to support potentially dozens of guests per VMM (so standard disk partitioning not sufficient)
- Type 1 – storage guest root disks and config information within file system provided by VMM as a **disk image**
- Type 2 – store as files in file system provided by host OS
- Duplicate file -> create new guest
- Move file to another system -> move guest
- **Physical-to-virtual** (**P-to-V**) convert native disk blocks into VMM format
- **Virtual-to-physical** (**V-to-P**) convert from virtual format to native or disk format
- VMM also needs to provide access to network attached storage (just networking) and other disk images, disk partitions, disks, etc

OS Component – Live Migration

- Taking advantage of VMM features leads to new functionality not found on general operating systems such as live migration
- Running guest can be moved between systems, without interrupting user access to the guest or its apps
- Very useful for resource management, maintenance downtime windows, etc
  1. The source VMM establishes a connection with the target VMM
  2. The target creates a new guest by creating a new VCPU, etc
  3. The source sends all read-only guest memory pages to the target
  4. The source sends all read-write pages to the target, marking them as clean
  5. The source repeats step 4, as during that step some pages were probably modified by the guest and are now dirty
  6. When cycle of steps 4 and 5 becomes very short, source VMM freezes guest, sends VCPU's final state, sends other state details, sends final dirty pages, and tells target to start running the guest
     4 Once target acknowledges that guest running, source terminates guest

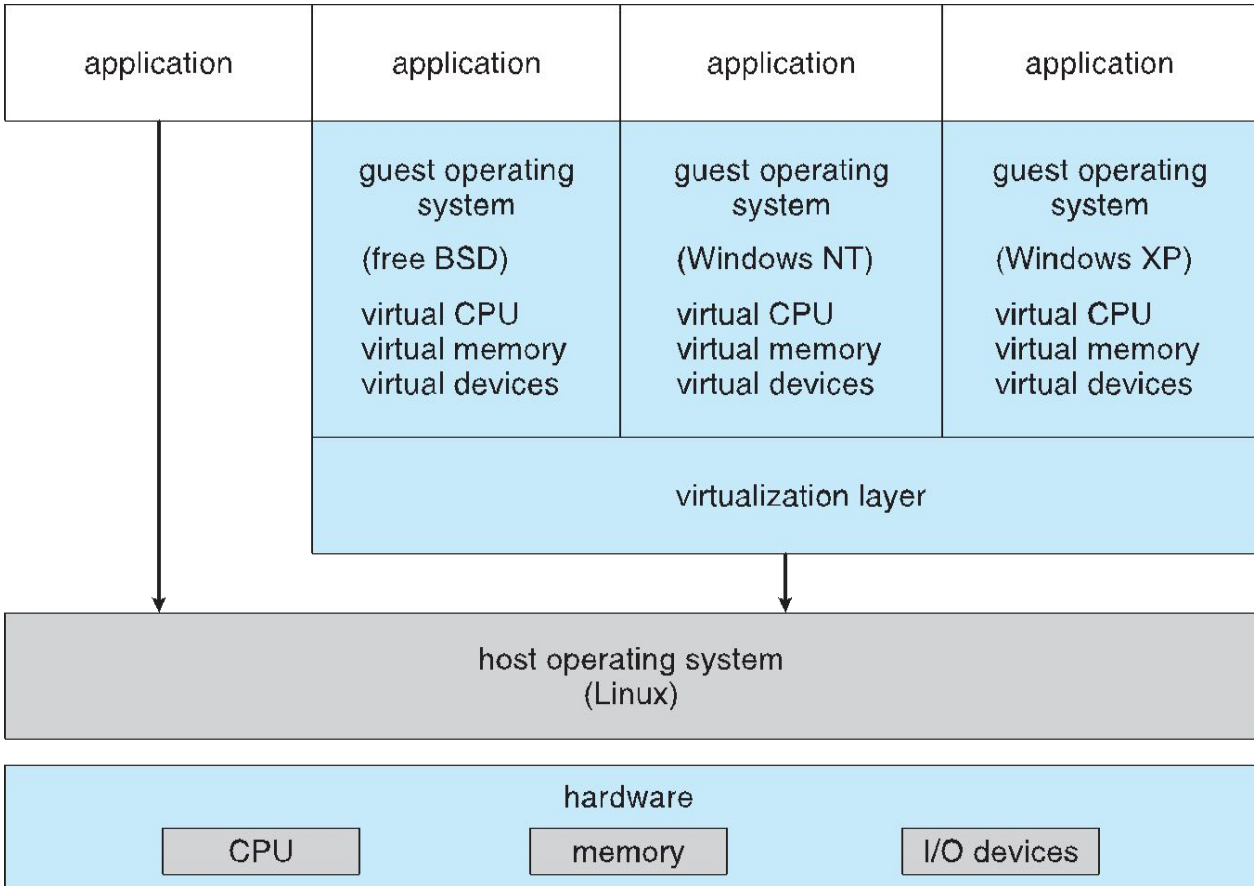
Live Migration of Guest Between Servers



Examples - VMware

- VMware Workstation runs on x86, provides VMM for guests
- Runs as application on other native, installed host operating system -> Type 2
- Lots of guests possible, including Windows, Linux, etc all runnable concurrently (as resources allow)
- Virtualization layer abstracts underlying HW, providing guest with is own virtual CPUs, memory, disk drives, network interfaces, etc
- Physical disks can be provided to guests, or virtual physical disks (just files within host file system)

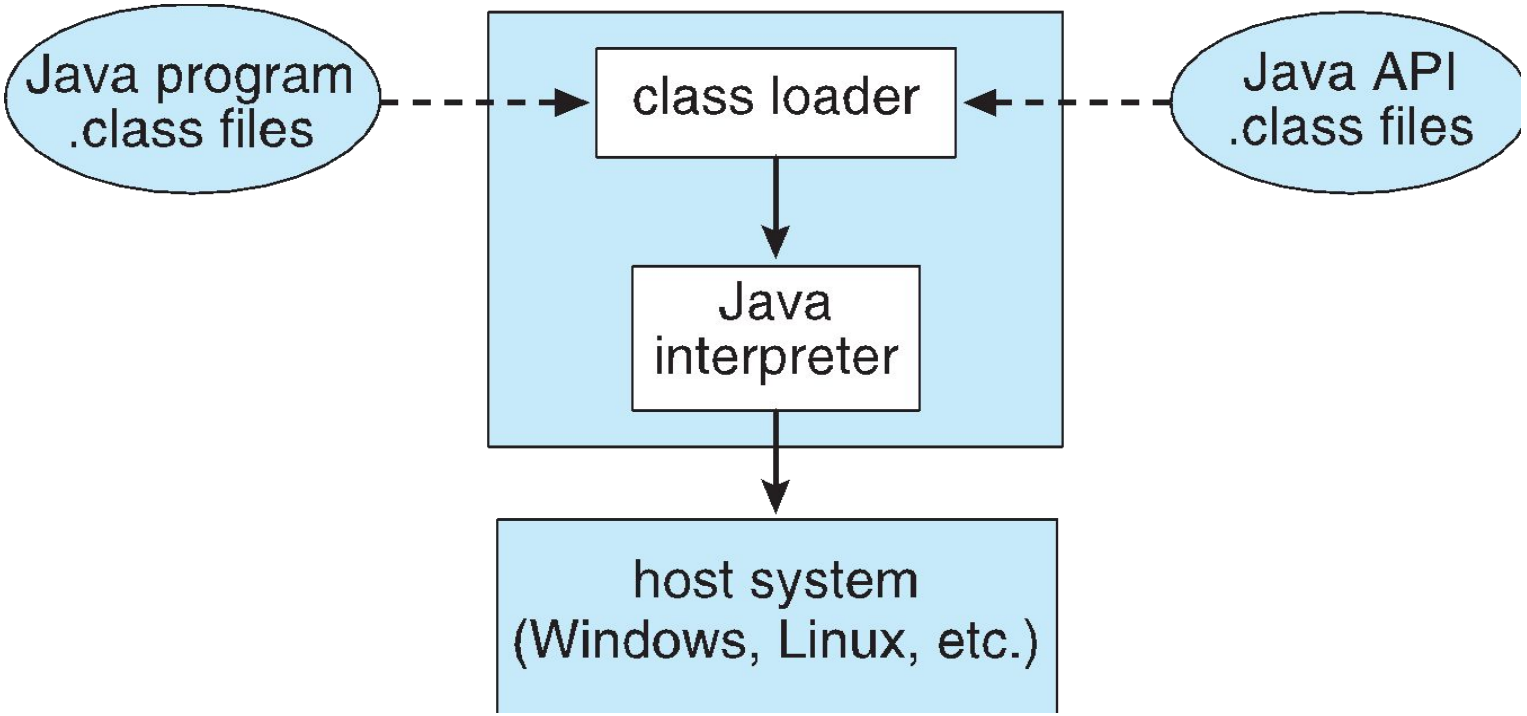
VMware Workstation Architecture



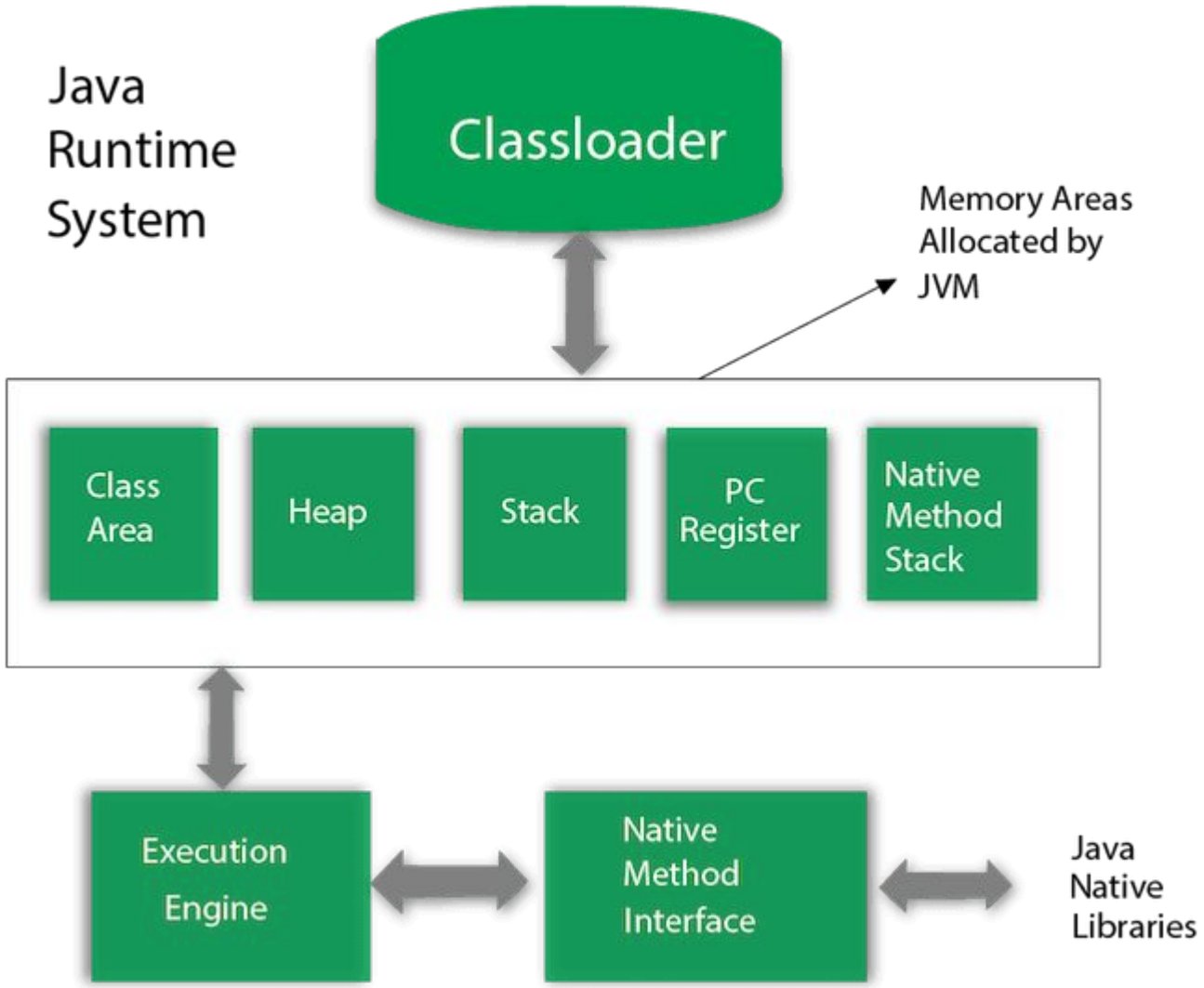
Examples – Java Virtual Machine

- Example of programming-environment virtualization
- Very popular language / application environment invented by Sun Microsystems in 1995
- Write once, run anywhere
- Includes language specification (Java), API library, Java virtual machine (JVM)
- Java objects specified by class construct, Java program is one or more objects
- Each Java object compiled into architecture-neutral **bytecode** output (`.class`) which JVM **class loader** loads
- JVM compiled per architecture, reads bytecode and executes
- Includes **garbage collection** to reclaim memory no longer in use
- Made faster by **just-in-time** (**JIT**) compiler that turns bytecodes into native code and caches them

The Java Virtual Machine



The Java Virtual Machine



End of Chapter 16