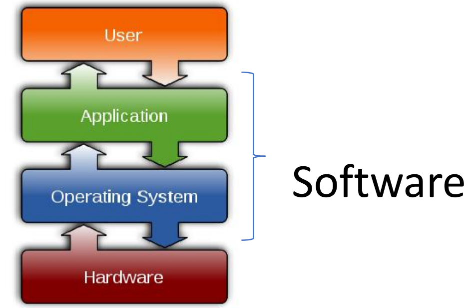
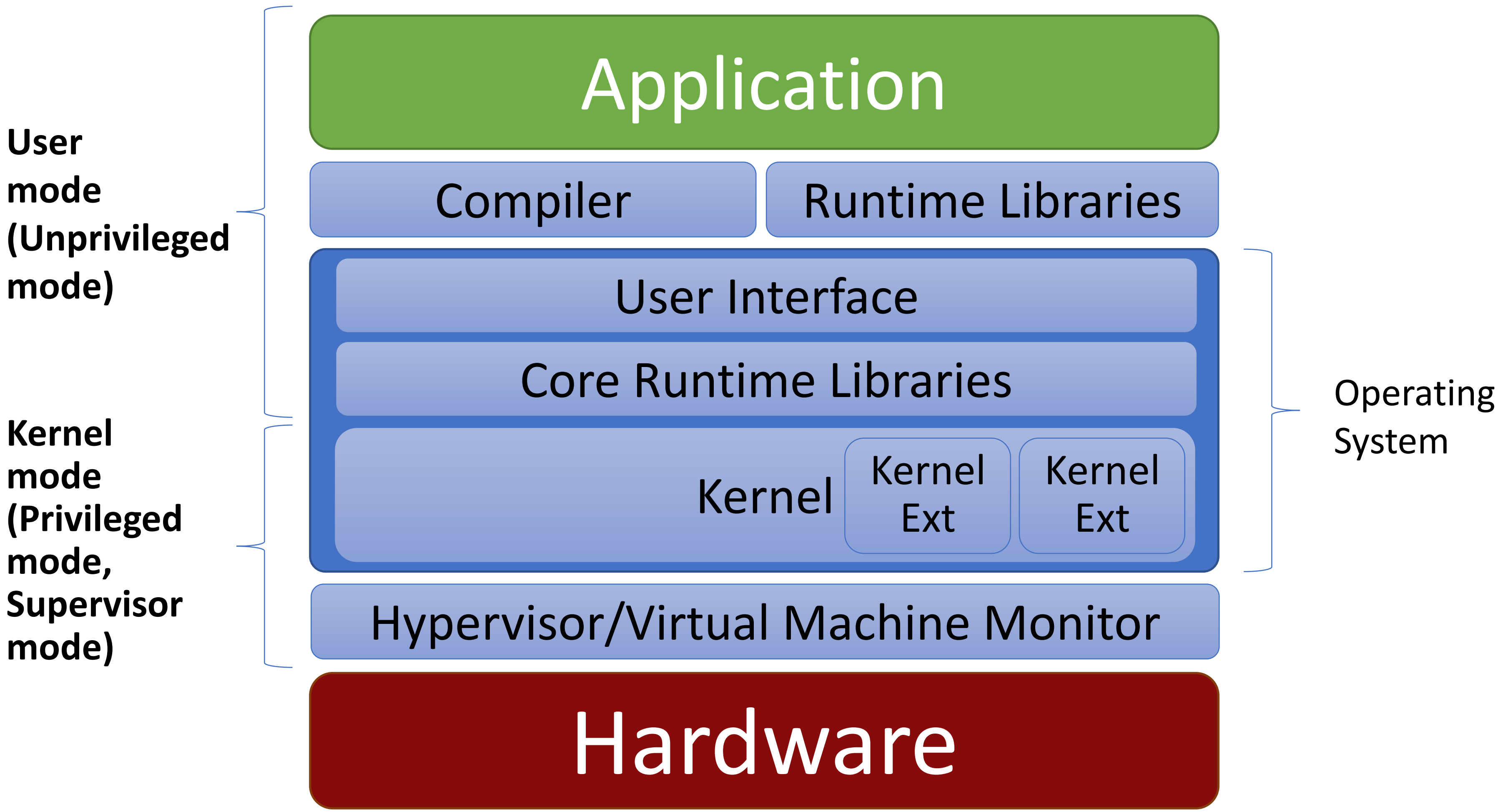
CS492 Spring 2019
Final Review

First Half

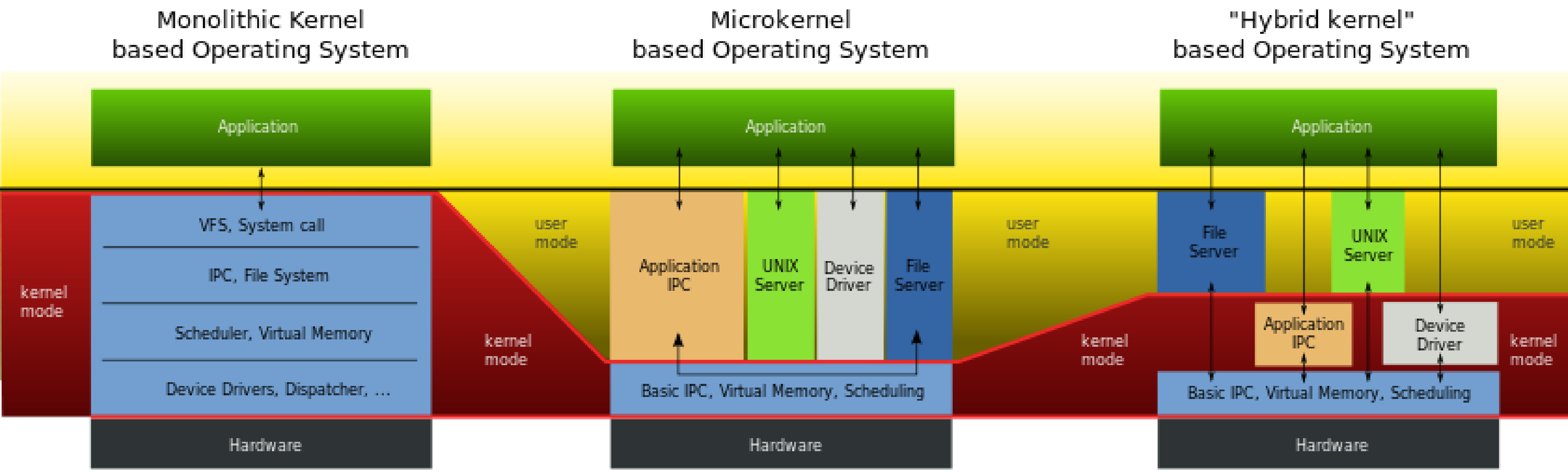
What is OS?





NOTE there exist OSes that do not use modes, there is hardware that doesn’t support modes

OS Kernel Designs



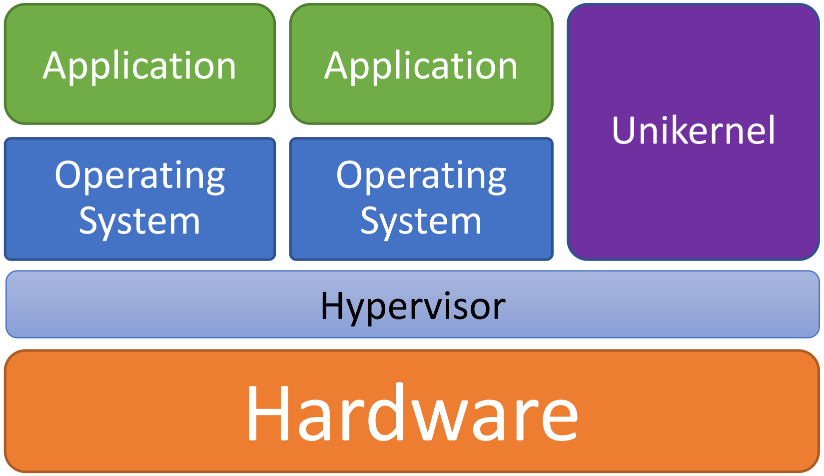
UNIX
Linux
Most BSD variants
…

MINIX
Mac
L4 family of kernels
…

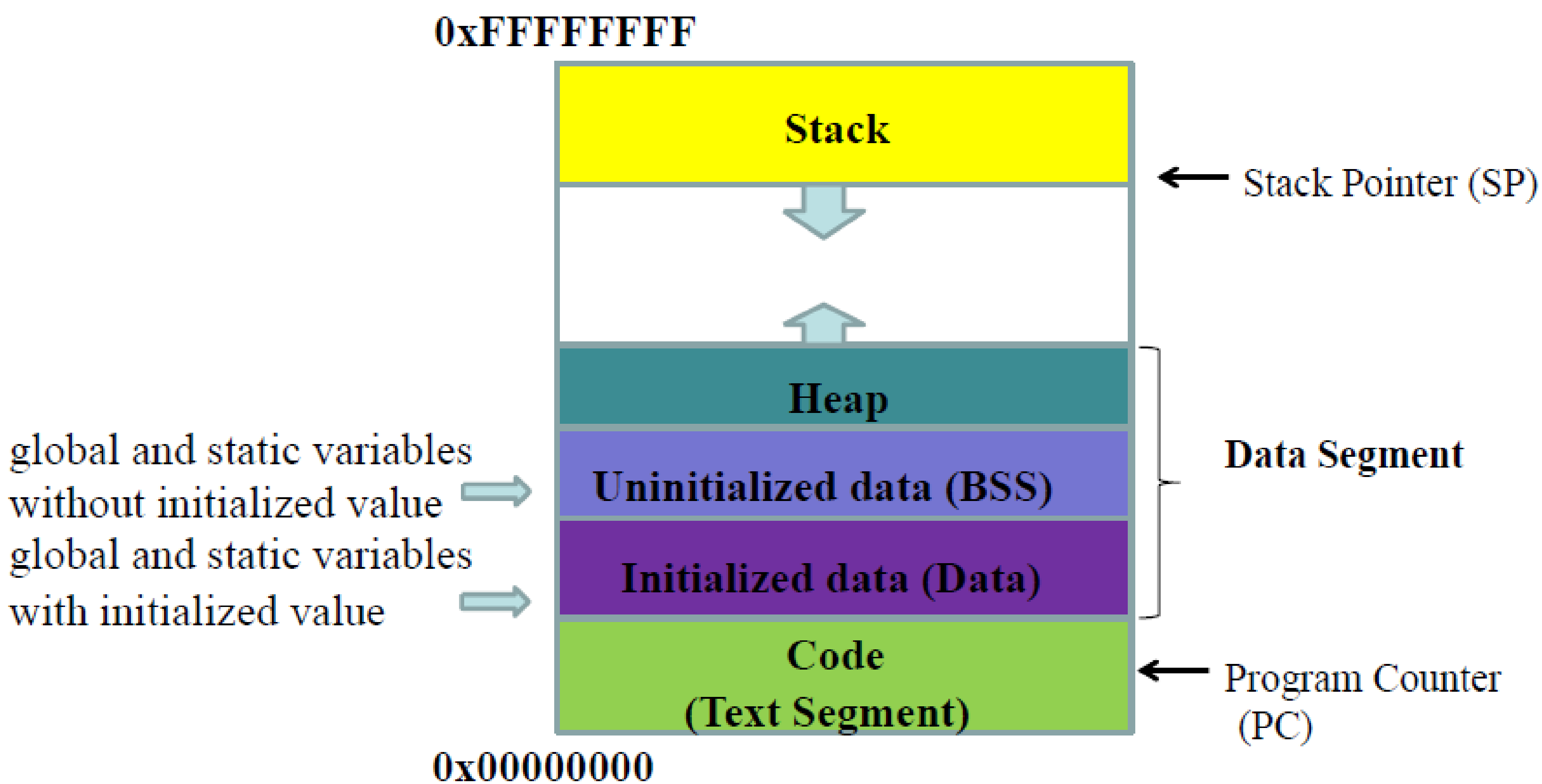
Windows
Xnu/Darwin
DragonFly BSD
…

What else?

- Virtual machines/hypervisors (e.g., VirtualBox, Xen)
  - Are not OSes
    - They interface with the hardware (below)
    - They provide an hardware interface (above)
  - Run OSes
    - Linux, Windows, BSD, etc.
    - **Unikernels ("libOS", cf. exokernel)**



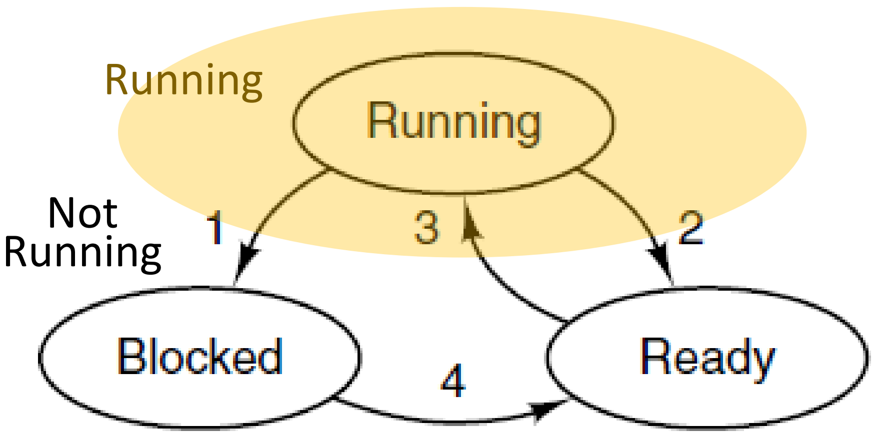
Address Space/Memory Layout



UNIX Process Creation Example

```
#include <stdio.h>
#include <unistd.h>
int main()
{
   int pid1 = getpid(); int pid2;
   int ret = fork();
   if (ret < 0) { /* error */
      printf(“error pid1: %d ret: %d\n”, pid1, ret);
   } else if (ret > 0) { /* parent */
      pid2 = getpid();
      printf(“parent pid2: %d pid1: %d ret: %d\n”, pid2, pid1, ret);
   } else { /* child */
      pid2 = getpid();
      printf(“child pid2: %d pid1: %d ret: %d\n”, pid2, pid1, ret);
   }
   return 0;
}
```

Process State and State Transitions

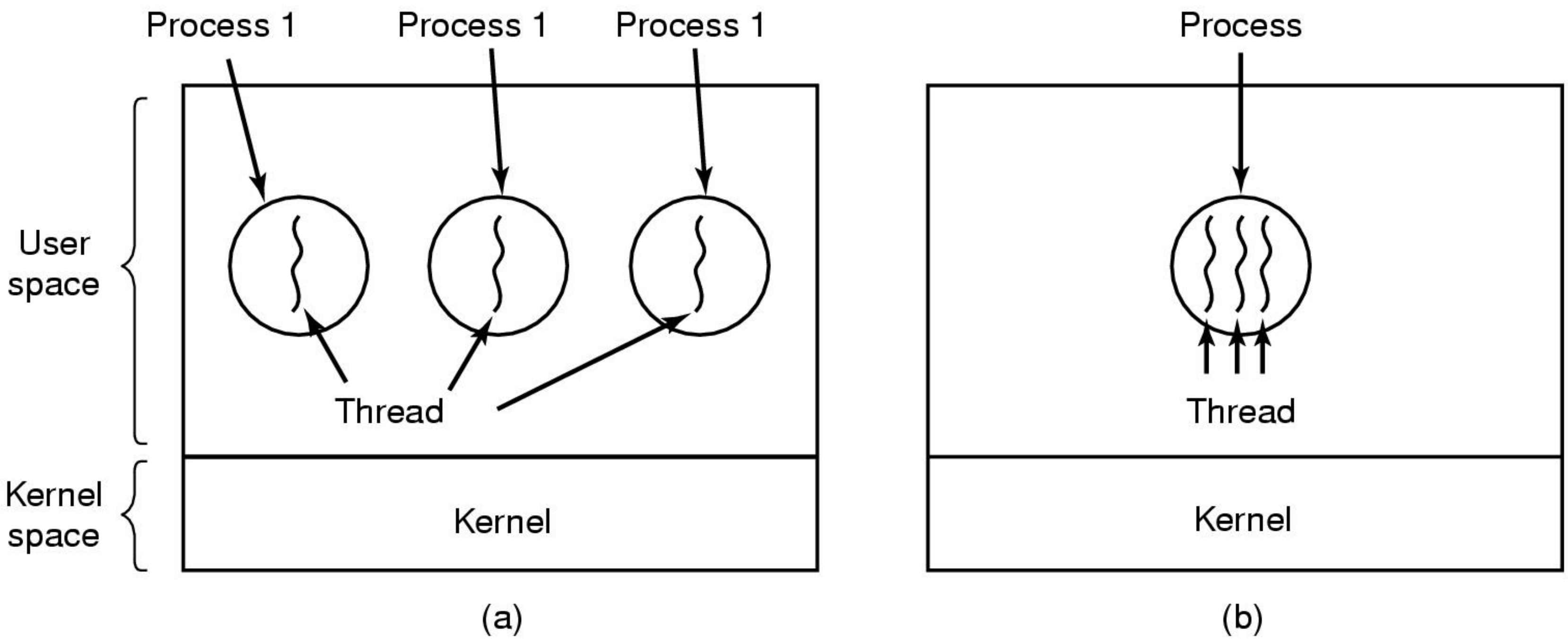


1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

A process can be in running, blocked, or ready state. Transitions between these states are as shown. (MOS Figure 2-2)

Why are there transitions missing?

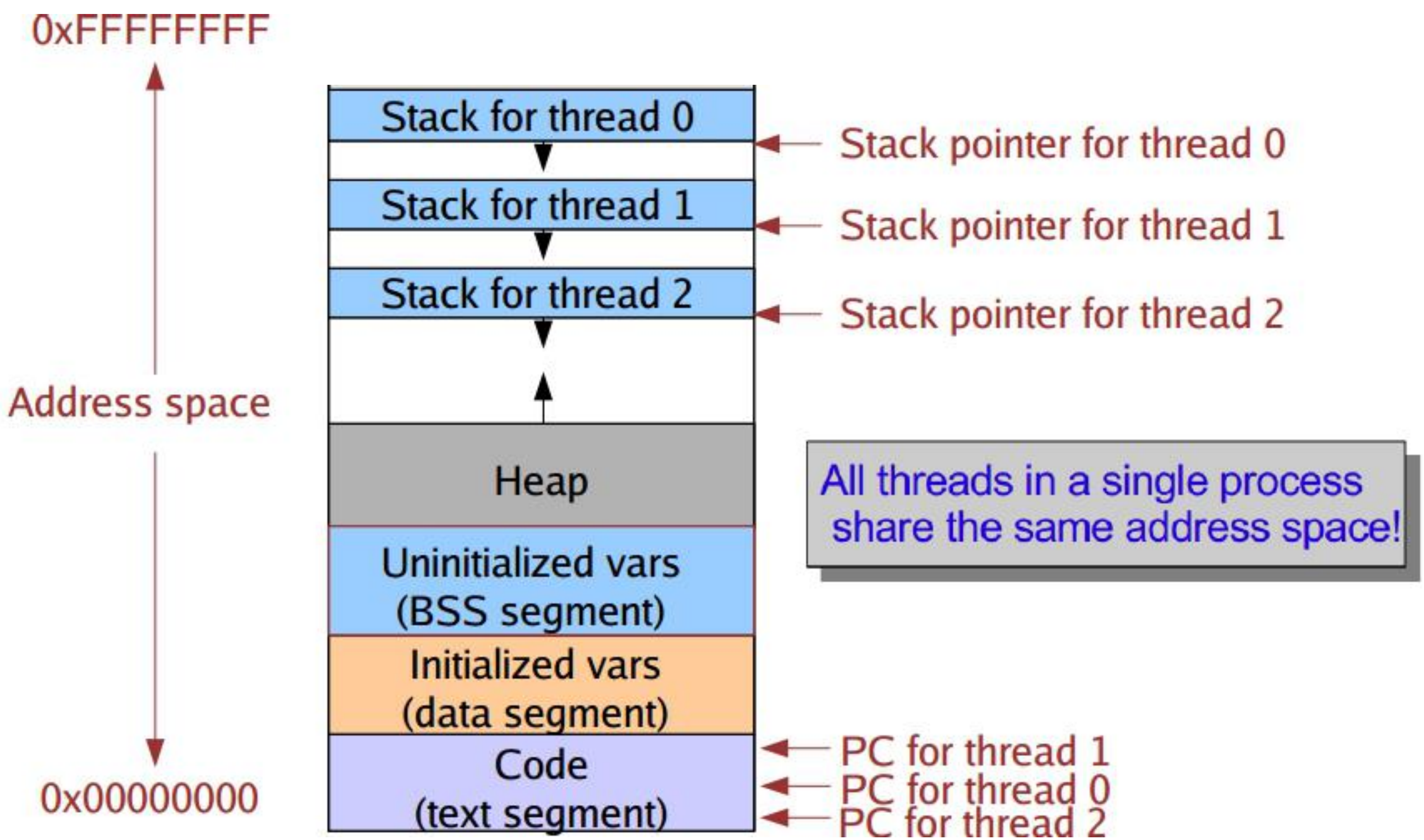
Multiprocessing vs Multithreading



Multiprocessing **Multithreading**

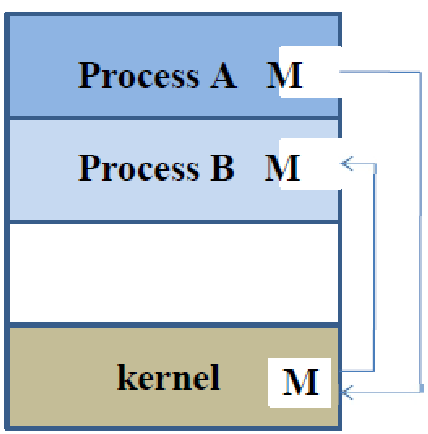
(a) Three processes each with one thread. (b) One process with three threads. (MOS Figure 2-11)

Address Space **with Threads**

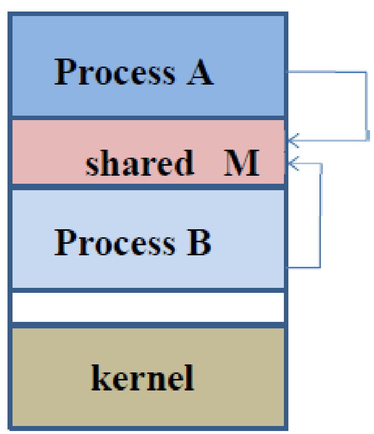


How One Process Can Pass Information to Another?

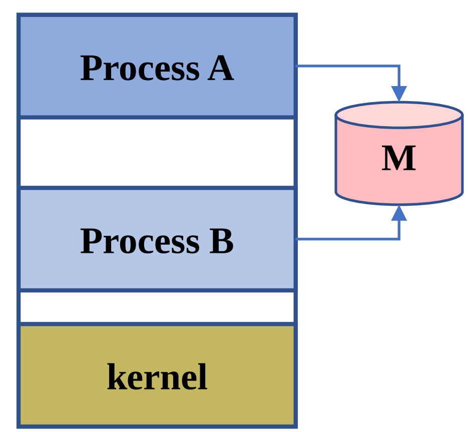
1. By passing messages through the kernel
2. By sharing memory
3. By sharing a file
4. Through asynchronous signals or alerts
5. …



Approach 1



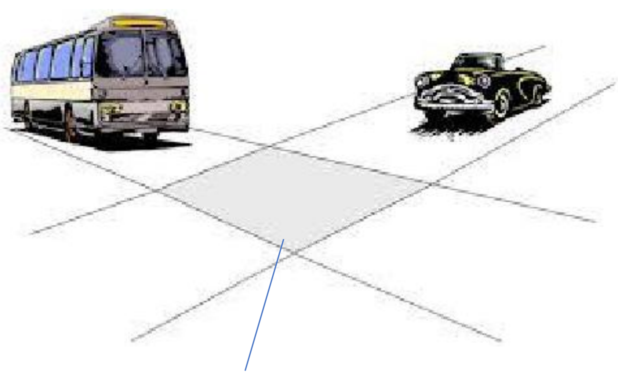
Approach 2



Approach 3

Race Conditions

- **Race condition**
  - Two or more processes reading or writing shared data
  - The final result depends on who runs precisely when
- **How to avoid race conditions?**
- **Critical Region or Section Modeling**
  - Part of the program where the shared data is accessed
    - Uncoordinated read/write of the data in critical section may lead to races



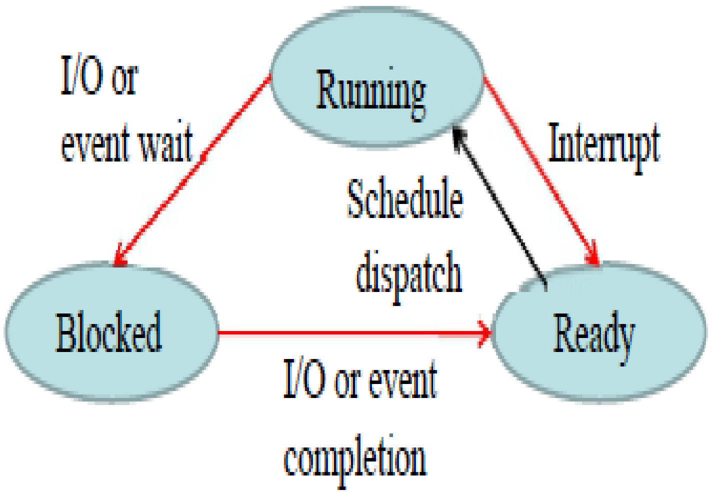
Mechanisms for Mutual Exclusion

- Hardware
  - Disabling interrupts
- Busy waiting
  - Lock variable
  - Strict alternation
  - Peterson's solution
  - Test-and-Set Lock (TSL) Instruction (hardware support)
- Sleep and wakeup
  - sleep() and wakeup()
  - Semaphores
  - Mutexes
  - Monitors

Why sleep and wakeup mechanisms are preferred vs busy waiting ones?

When to Schedule

- Scheduling decisions may take place when a **process/thread**
  - Is created
  - In running state exits
  - Blocks on IO, or an event
    - Switch from *Running* to *Blocked*
- Scheduling decisions may take place when an **interrupt occurs**
  - Clock interrupt
    - Switch from *Running* to *Ready*
  - IO interrupt, or (unblocking) syscall
    - Switch from *Blocked* to *Ready*



Basic Scheduling Algorithms

- Batch Systems
  - First-come First-served
  - Shortest Job First and Shortest Remaining Time Next
- Interactive Systems
  - Round-robin
  - Priority
  - Multiple Queues
  - Shorted Process Next, Guarantee, Lottery, Fair-share
- Real-time Systems
  - Fixed Priority
  - Dynamic Priority

Scheduling in Real-time Systems

- Categorization
  - **Hard real-time**
    - There are absolute deadlines that must be met
  - **Soft real-time**
    - Missing an occasional deadline is undesirable but tolerable
- Assumption(s)
  - **Processes behavior is predictable and known in advance**
    - **Release time (R_i), execution time (C_i), deadline (D_i)**
  - Periodic process
  - Sporadic process
  - Aperiodic process

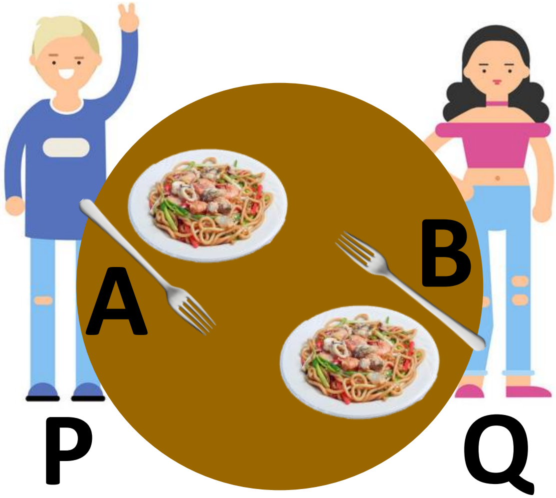
Scheduling Periodic Processes

- How many periodic processes are **schedulable**?
  - "can fit/run on a processor" – single CPU
  - All combination of processes that satisfy the formula

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

- ***m*** periodic events
- Event ***i*** occurs
  - with period P_i
  - requires C_i time on the CPU

Deadlock Example



Semaphore muxA = 1 /* protects resource A */

Semaphore muxB = 1 /* protects resource B */

Process P:

```
{
  /* initial compute */
  down(muxA)
  down(muxB)
  /* use both resources */
  up(muxA)
  up(muxB)
}
```

Process Q:

```
{
  /* initial compute */
  down(muxB)
  down(muxA)
  /* use both resources */
  up(muxB)
  up(muxA)
}
```

Strategies Dealing with Deadlocks

- **Allow** deadlock to happen
  a) Ostrich algorithm: ignore the problem altogether
  b) Deadlock ***detection and recovery***: allow deadlock, detect it, break it
- Ensure deadlock **never** occurs (Part 2)
  c) Deadlock ***avoidance***: careful resource allocation – each resource request is analyzed and denied if deadlock might result
  d) Deadlock ***prevention***: negate one of the four necessary conditions

Second Half

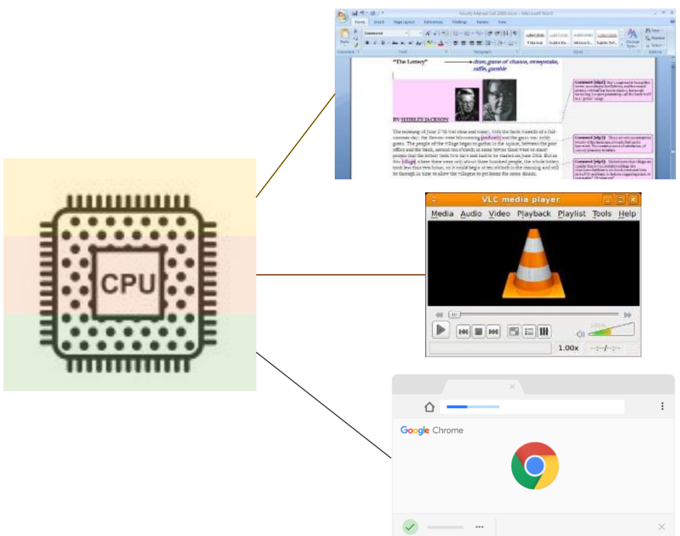
Problems With Programs on Physical Memory

- Protection
- Relocation
  - Base+Limit registers
- Fitting multiple programs
  - Swapping
- Memory fragmentation
  - Compaction/memory management **– very expensive**
- Application larger than memory
  - Overlay manager **– very expensive, difficult to program, reduced resource exploitation, no operating system involvement**

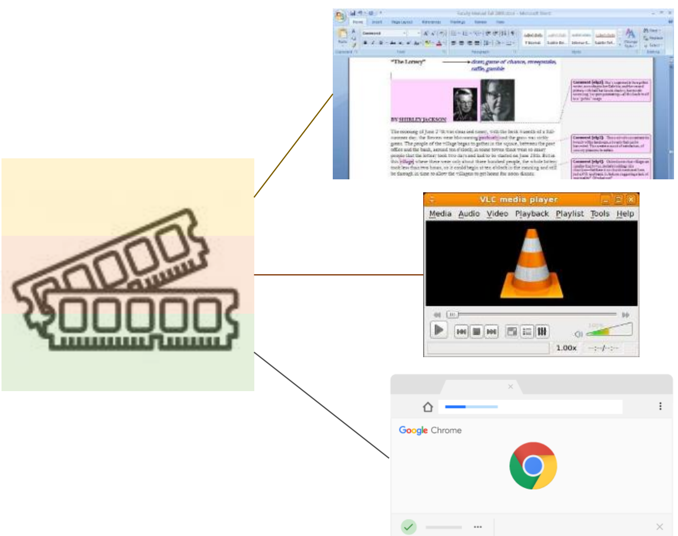
Memory Abstraction: Address Space

NOT all physical memory addresses!

- Address Space
  - Abstraction from physical memory space
  - Set of memory addresses that a process can use
    - Independently from other processes



Process, abstracts physical CPU

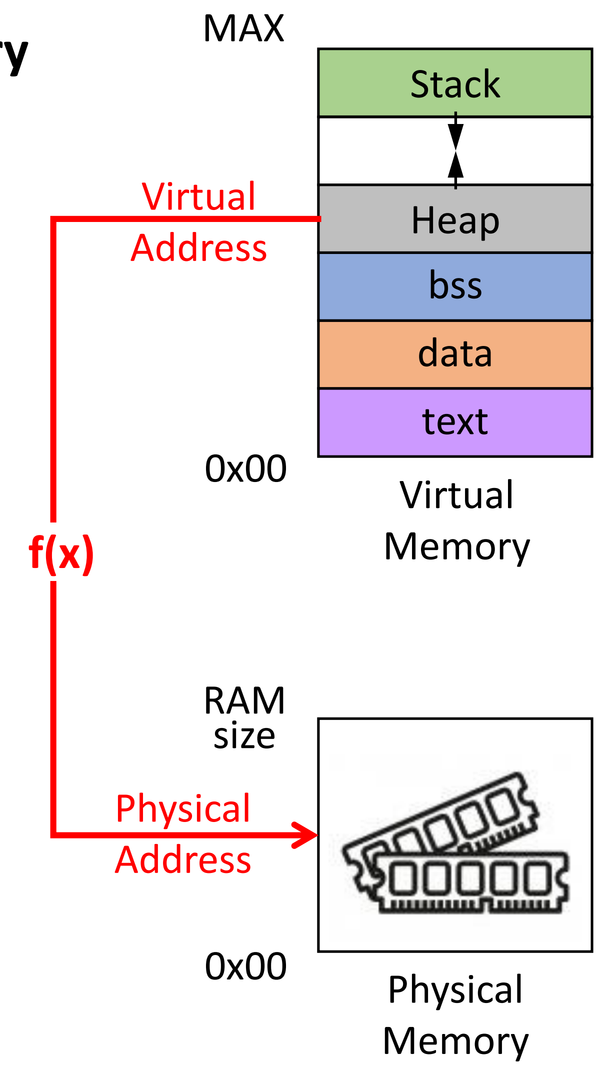


Address space, abstracts physical memory

Virtual Memory: Method

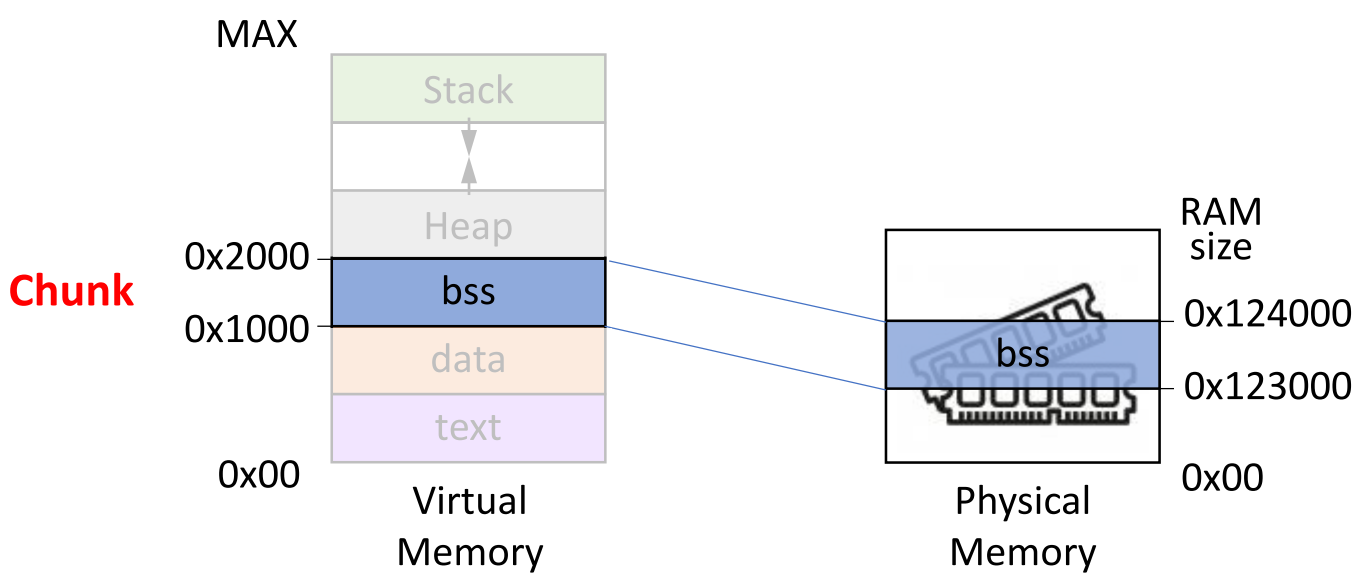
- **Decouples address space** from **physical memory**
- Illusion of large and private address space
  - Independently of the physical memory size

- Each process its own (virtual) address space
  - From address 0x00, to MAX
- Program generates **virtual addresses**
- Virtual address ≠ physical RAM address
  - Virtual address **translated** to physical address
  - Translation **transparent** to the program



Virtual Memory: Method (Continue)

- Virtual address space **broken into chunks**
  - Chunk is a contiguous range of addresses
    - Mapped onto a contiguous range of physical memory
  - Process always sees the entire virtual address space



f(x) is byte by byte, but monotonically continuous chunk by chunk

Virtual Memory: Method (Continue)

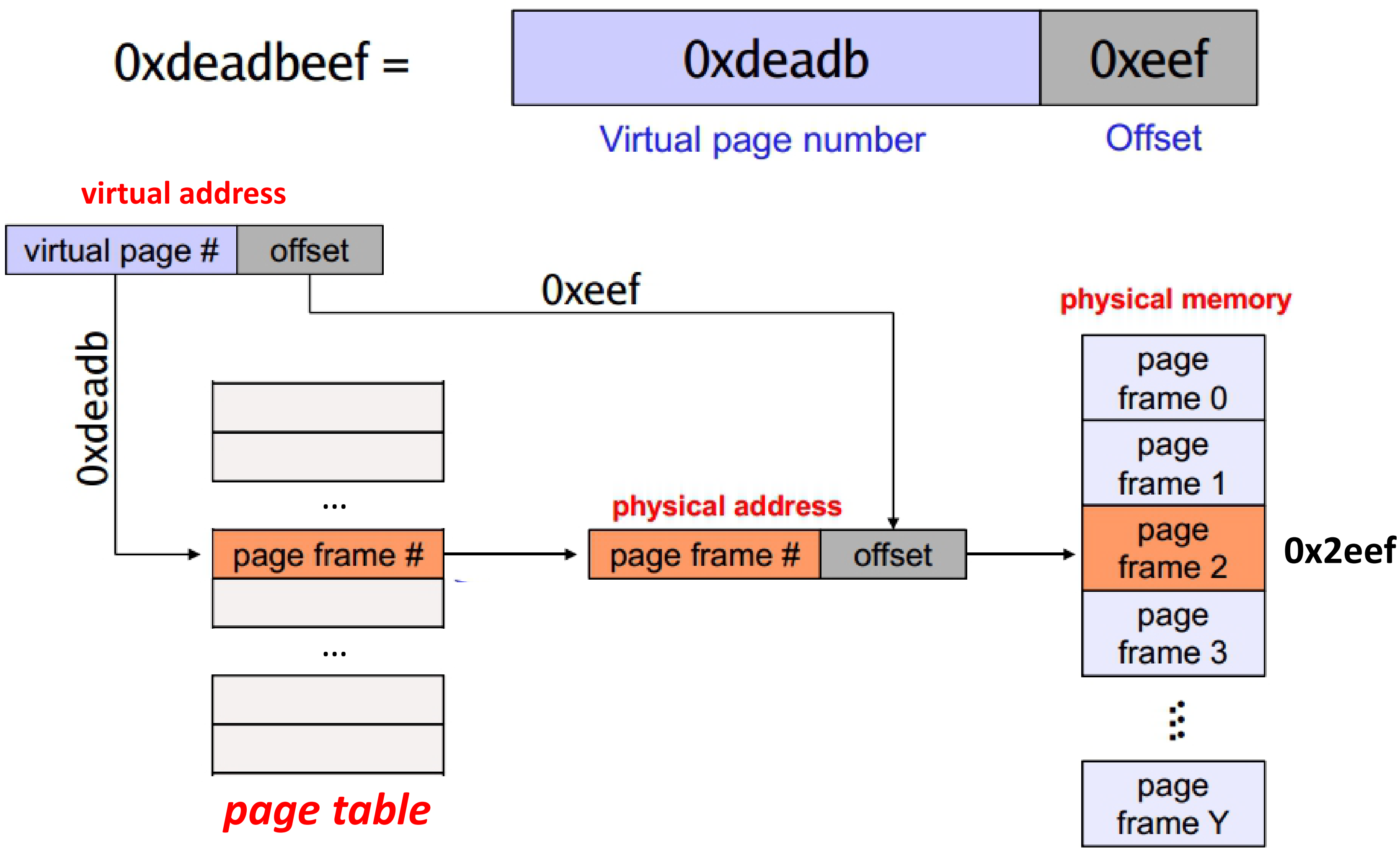
- Virtual address space **broken into chunks**
  - Chunk is a contiguous range of addresses
    - Mapped onto a contiguous range of physical memory
  - Process always sees the entire virtual address space

- Not all chunks should be in physical memory to run the program
  - Chunk switching hidden from processes
  - Program references a chunk in physical memory
    - **Hardware** performs the necessary mapping on the fly
  - Program references a chunk **not** in physical memory
    - **OS** gets it from disk and re-execute the memory instruction

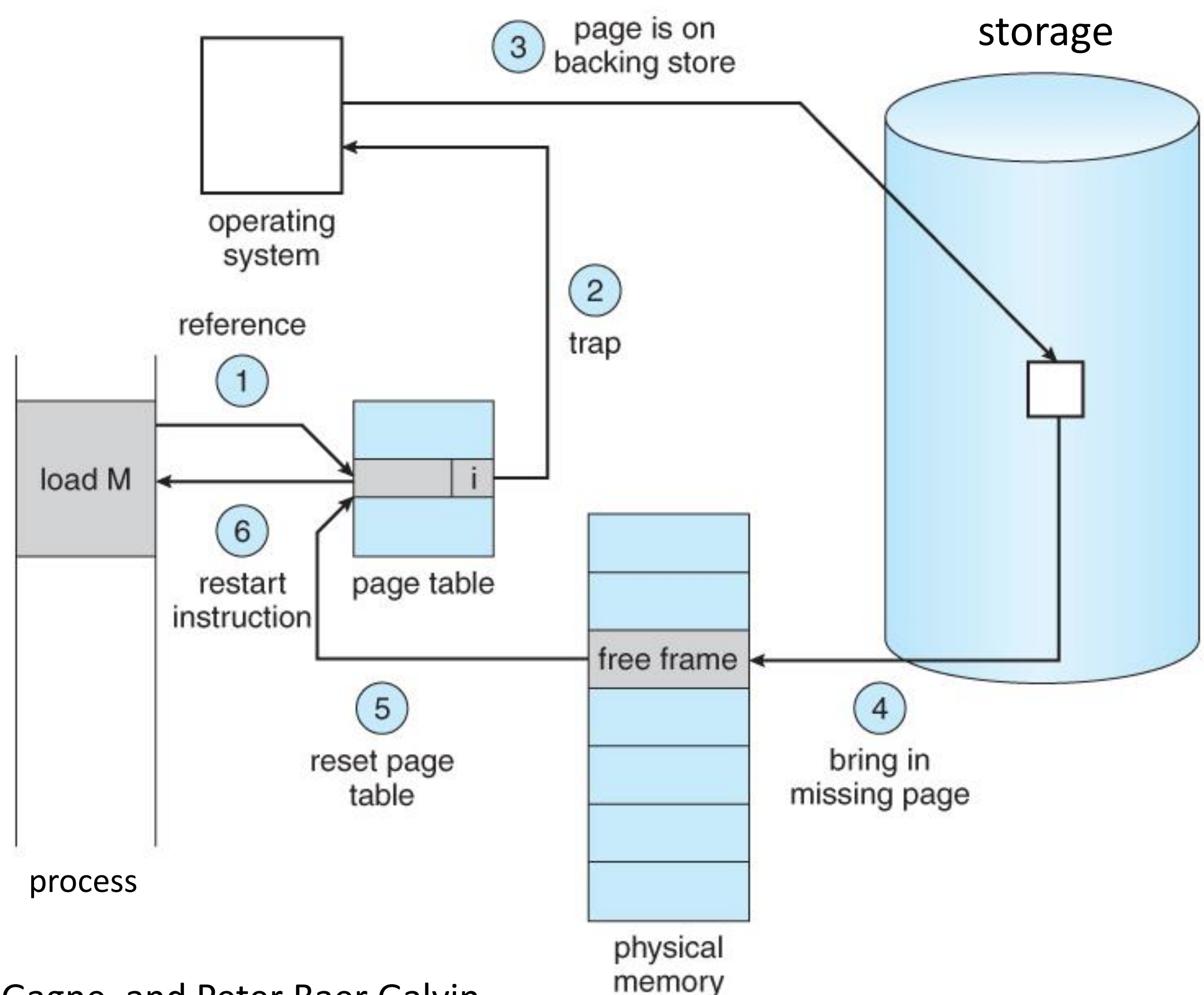
Paging

- Virtual address space consists of fixed-size units called **pages**
- Corresponding units in physical memory are called **page frames**
- Pages and page frames are of the **same size**
  - Usually 4kB
    - Page sizes from 512B to 1GB have been used
  - Example
    - 4kB page size with 64kB of virtual address space and 32kB of physical memory, gets 16 virtual pages and 8 page frames
- The hardware memory management unit (MMU) **maps pages to page frames**

Paging: Address Translation (3)



Page Fault



Abraham Silberschatz, Greg Gagne, and Peter Baer Galvin, "Operating System Concepts, Ninth Edition ", Chapter 9

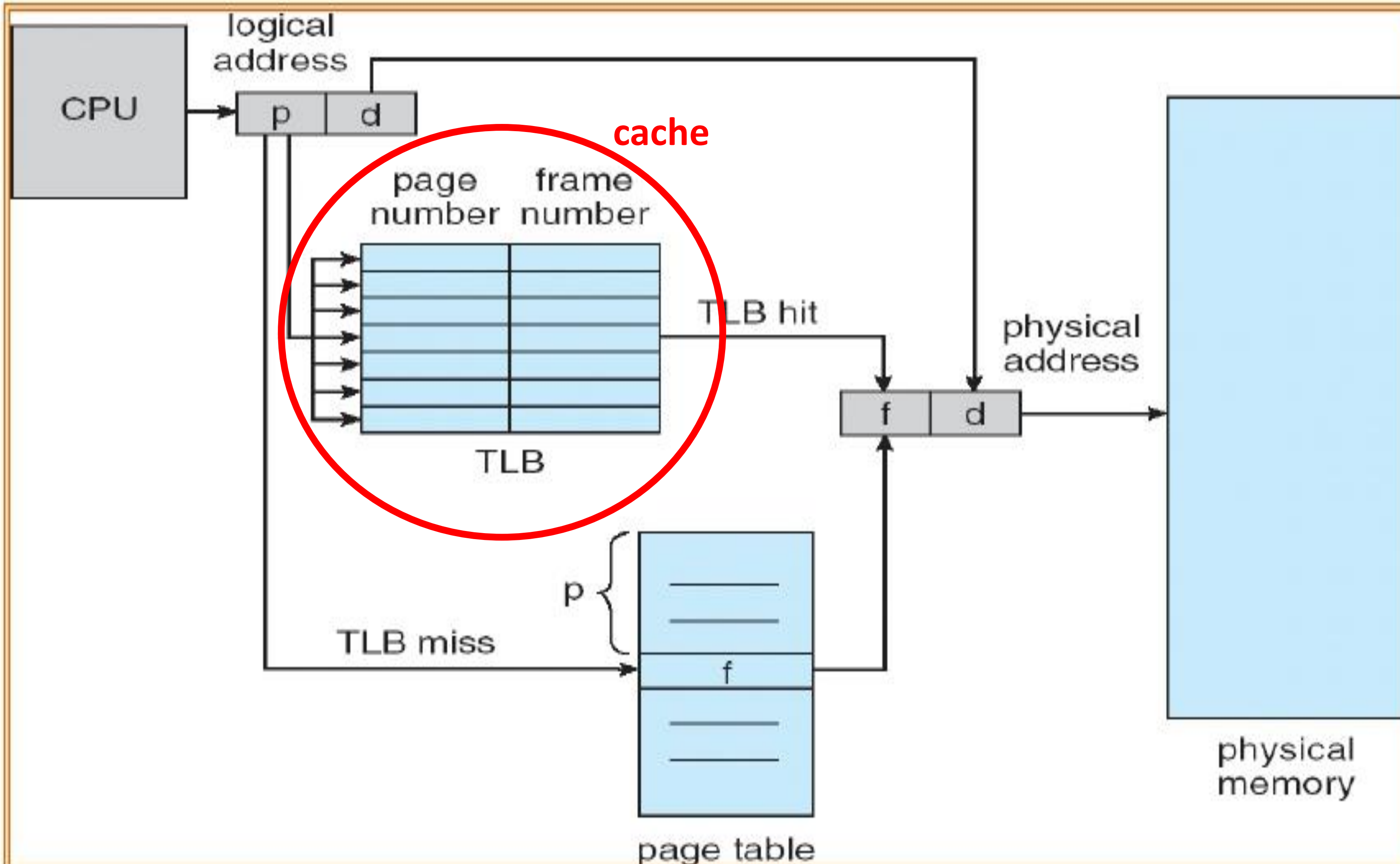
Exercise 1: Page Table Size

- Consider a 32-bit virtual address space. Each page consists of 4096 virtual addresses. Assume each page table entry (PTE) takes 4 bytes

- Questions
  - What is the size of the page table for one process?
  - What is the total size of the page tables for 100 running processes?

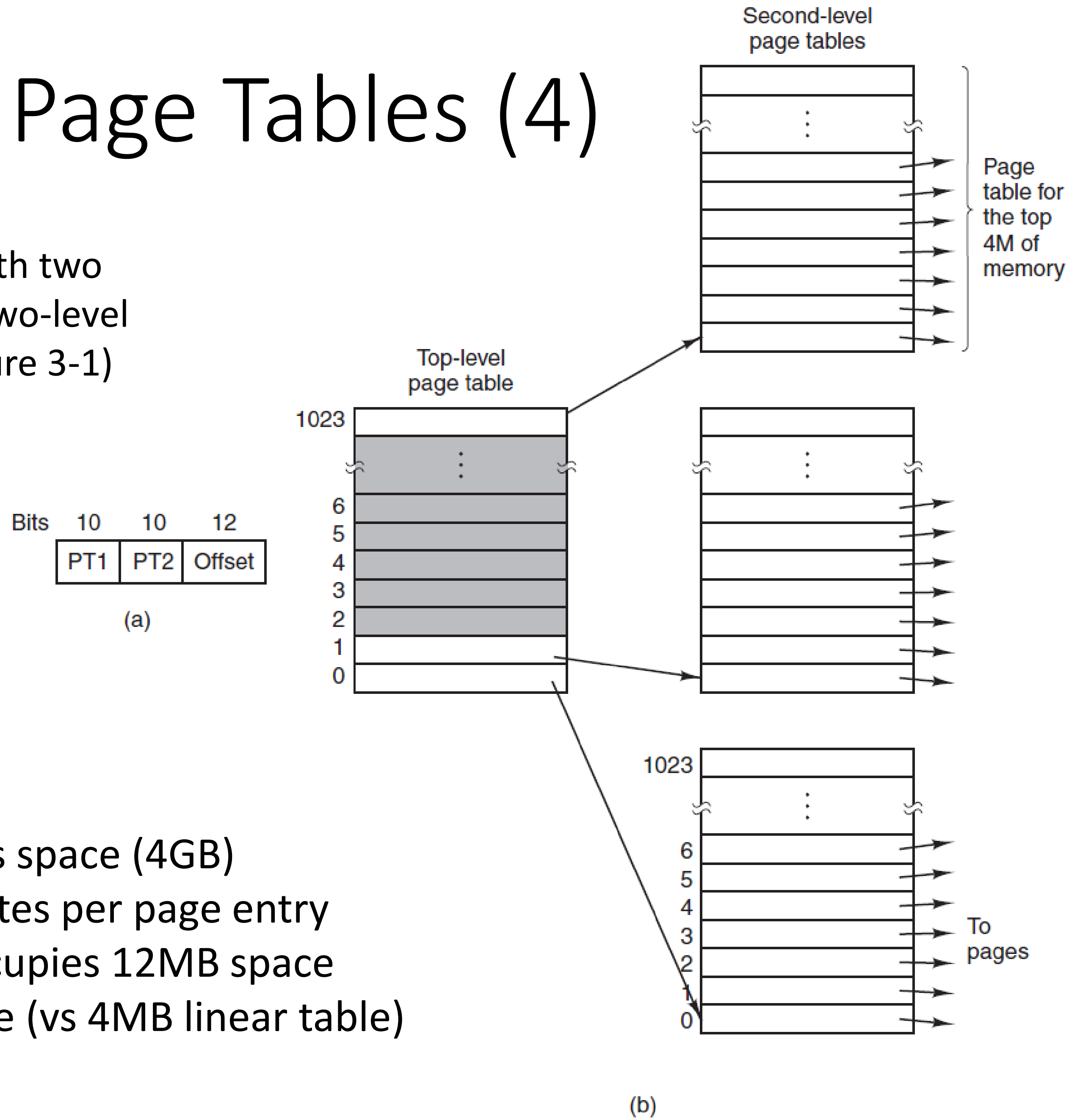
- Answers
  - Page table entries: 2^(32-12)=2^20. Size of table = 4*2^20 Bytes = 4MB
  - 100*4 = 400MB

Translation Lookaside Buffer (5)



Multilevel Page Tables (4)

(a) A 32-bit address with two page table fields. (b) Two-level page tables. (MOS Figure 3-1)



- Example
  - A 32bit address space (4GB)
  - 4kB pages, 4bytes per page entry
  - Application occupies 12MB space
  - 16kB page table (vs 4MB linear table)

How Big Is a Multilevel Page Table? (1)

- Consider a 32-bit virtual address space. Each page consists of 4K virtual addresses. Also assume each page table entry (PTE) takes 4 bytes.
  - Recall: A 1-level page table takes 4MB!
  - Question: assume the 32-bit address is allocated as following: 10 bits to the primary page, 10 bits to the secondary page, 12 bits to the page offset
    - What is the size of each 2-level page table?
    - What is the total size of the page tables (including 1-level and 2-level ones)?
    - How much memory is needed for one virtual address translation?

- -2^10*4=4KB
- -1025*4KB = 4MB and 4 KB
- - 8 KB

Page Replacement Algorithm(s)

- Question
  - What page to evict when memory is full and a new page is demanded?
- Goal
  - Lowest number of (future) page faults
- *Input of algorithm*
  - A particular string of memory references
    - (page) 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- *Output of algorithm*
  - The number of page faults on that string

Optimal Algorithm (OPT)

- What's the best we can possibly do?
  - Assume OS knows about the future
- Algorithm
  - Replace the page that will be used ***furthest*** in the future
- Estimate by
  - Logging page use on previous runs of process
  - Impractical
    - Depends on the application
    - Depends on its input

Nice, but not achievable in real systems!
(Need to know about the future)

Not Recently Used (NRU) (1)

- Idea
  - Use virtual memory hardware tracking bits
  - Determine what page was not recently accessed
- Observation
  - It is better to remove a modified page that has not been referenced lately than a clean page that is in heavy use
    - A modified page may have to be updated on disk

Not Recently Used (NRU) (2)

- Algorithm
  1) When a process is started up, R and M bits for all its pages are set to 0 by the OS
  2) 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
  3) When a page fault occurs, the operating system inspects all the pages and divides them into four categories
     - Class 0: not referenced, not modified
     - Class 1: not referenced, modified
     - Class 2: referenced, not modified
     - Class 3: referenced, modified
  4) The algorithm removes a page at random from the lowest-numbered nonempty class

However, pages that are frequently referenced might be removed

First In First Out (FIFO)

- FIFO
    - First in First Out

- Maintain a linked list of all pages
    - **In the order that they came into memory**

- A new page is added at the tail
- Page at the beginning of list is removed

FIFO: Example

- 3 physical page frames
- 5 virtual pages
- Reference string: 0, 1, 2, 3, 0, 1, 4, 0, 1, 2, 3, 4

0	1	2	3	0	1	4	0	1	2	3	4
0	1	2	3	0	1	4	4	4	2	3	3
	0	1	2	3	0	1	1	1	4	2	2
		0	1	2	3	0	0	0	1	4	4
P	P	P	P	P	P	P			P	P	

3 frames

Tail of List

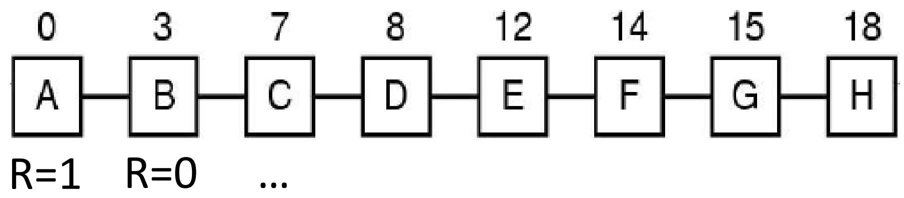
Beginning of List

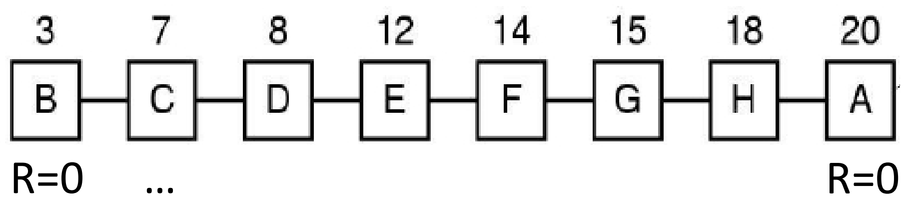
9 Page faults

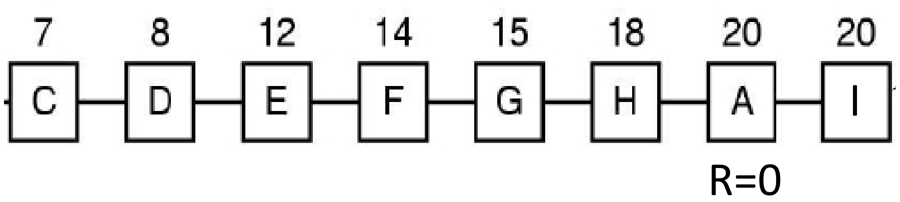
Second Chance

- FIFO variant
  - Adds the concept of usage (references)
- Examine pages in FIFO order starting from beginning of list but
  - Consider “reference bit” R
    a) IF R=0, remove page, go to **c)**
    b) IF R=1, set R=0 and place it at the end of FIFO list (hence, the second chance), go to **a)**
    c) Add new page at the end of FIFO (with R=0)
  - If not enough replaces on first pass, revert to pure FIFO on second pass

Second Chance: Example

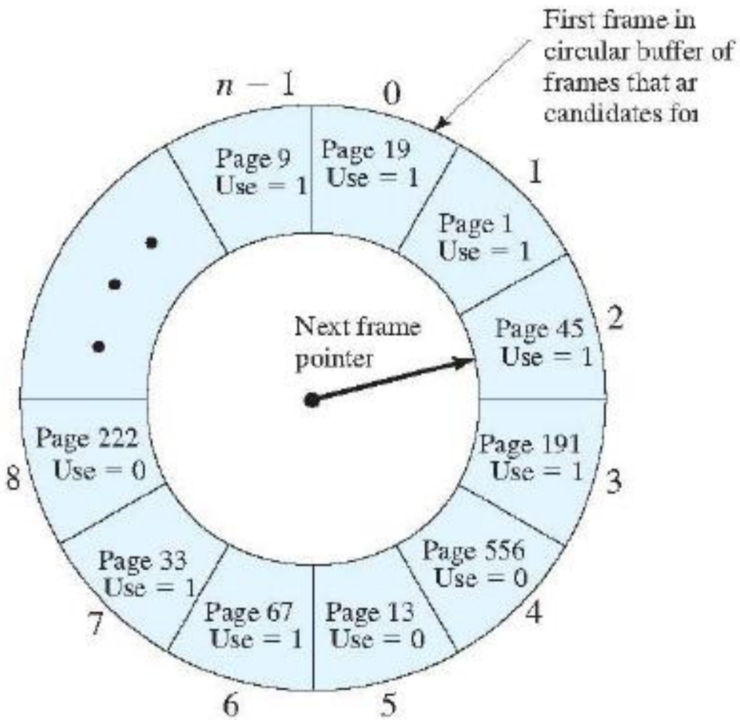




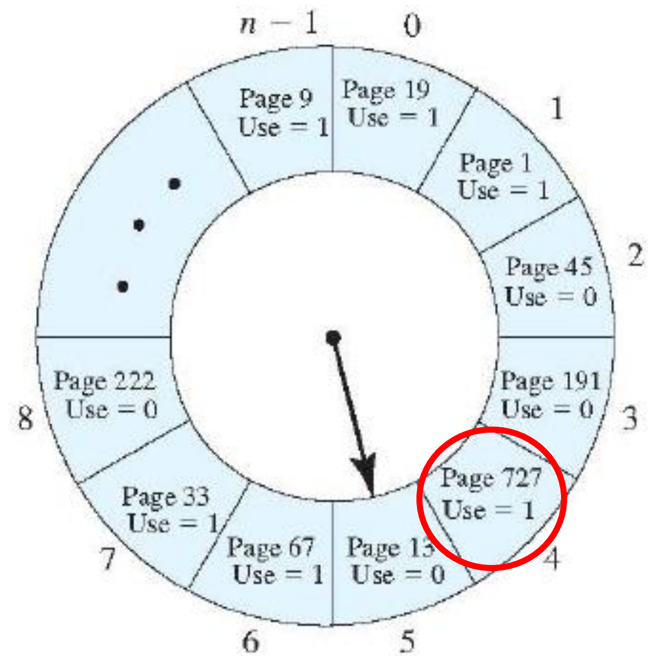


Clock

- Second chance variant
- Problem of the *second chance* mechanism
  - moving pages around on list is not efficient
- Clock is a more performant implementation



(a) State of buffer just prior to a page replacement



(b) State of buffer just after the next page replacement

Least Recently Used (LRU)

- From the Book

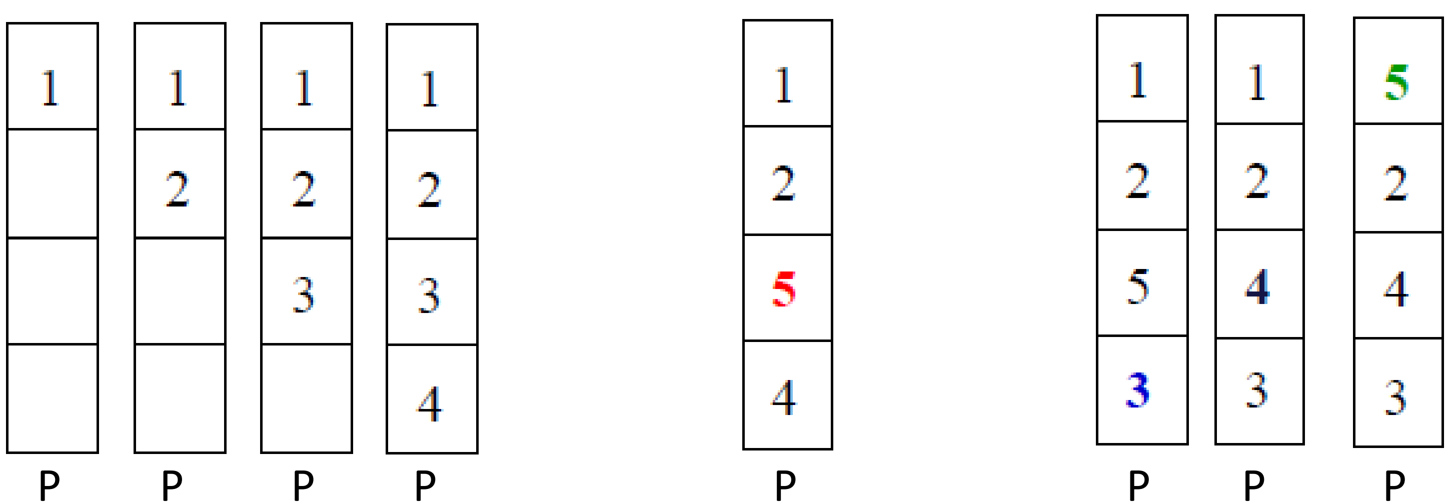
- *A **good** approximation of the optimal algorithm is based on the observation that*
  - *Pages that have been heavily used in the last few instructions will probably be heavily used again soon*
  - *Conversely, pages that have not been used for ages will probably remain unused for a long time*

- *This idea suggests a realizable algorithm, LRU*
  - *When a page fault occurs, throw out the page that has been unused for the longest time*

Least Recently Used: Example

- Reference string: 1, 2, 3, 4, 1, 2, **5**, 1, 2, **3**, **4**, **5**
- Optimal vs LRU
  - Optimal: Throw out pages the furthest in the ***future***
  - LRU: Throw out pages the furthest in the ***past***

1 2 3 4 1 2 **5** 1 2 **3** **4** **5**

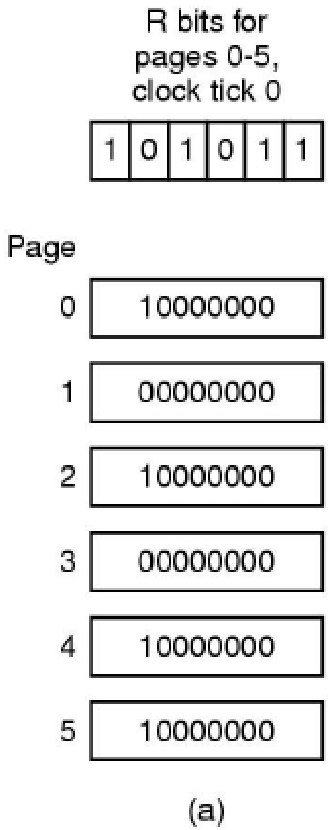


Aging

Not Frequently Used (NFU) never forgets, therefore …

- A N-bit counter per page
- Periodically
  - Shift counter to the right
    - Reduce counter values over time by dividing it by two
  - Add R to the leftmost bit
    - Recent R bit is added as most significant bit, therefore more weight given to more recent references!
- Page replacement
  - Replace page with the lowest counter

Aging: Example



The aging algorithm simulates LRU in software. Shown are six pages for five clock ticks. The five clock ticks are represented by (a) to (e). (MOS Figure 3-17)

Question

- A small computer on a smart card has four page frames. At the first clock tick, the R bits are 0111 (page 0 is 0, the rest are 1). At subsequent clock ticks, the values are 1011, 1010, 1101, 0010, 1010, 1100, and 0001. If the aging algorithm is used with an 8-bit counter, give the values of the four counters after the last tick

Question

- If FIFO page replacement is used with four page frames and eight pages, how many page faults will occur with the reference string 0172327103 if the four frames are initially empty? Now repeat this problem for LRU.

When to Move Pages into Memory?

- Action also called fetching

- Main techniques
  - Demand paging
    - Pages are loaded on demand, not in advance
  - Prepaging
    - Load group of pages at once
- Demand paging and prepaging may be used together

Working Set Based Page Replacement Algorithm

- A threshold T, and for every page saved
  - Time of last use
  - Reference bit R, and modified bit M (writeback check)
- If reference bit R=0, page is a candidate for removal
  - Calculate *age* = (current time – time of last use)
  - If age > threshold, page is replaced
  - If age < threshold, still in working set, but may be removed if it is oldest page in working set
- If R=1, set time of last use = current time, set R=0
  - Page was recently referenced, so in working set
- If no page has R=0, choose the oldest (one that requires no writeback) – when all same age pick random

WSClock variant of this algorithm

Question

Suppose that the WSClock page replacement algorithm uses a τ of two ticks, and the system state is the following:

Page	Time stamp	V	R	M
0	6	1	0	1
1	9	1	1	0
2	9	1	1	1
3	7	1	0	0
4	4	0	0	0

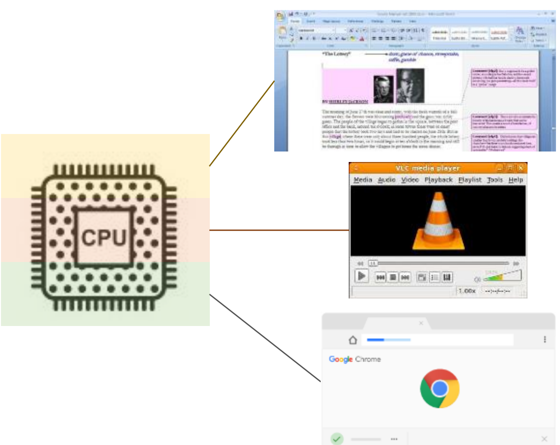
where the three flag bits V, R, and M stand for Valid, Referenced, and Modified, respectively.

(a) If a clock interrupt occurs at tick 10, show the contents of the new table entries. Explain. (You can omit entries that are unchanged.)

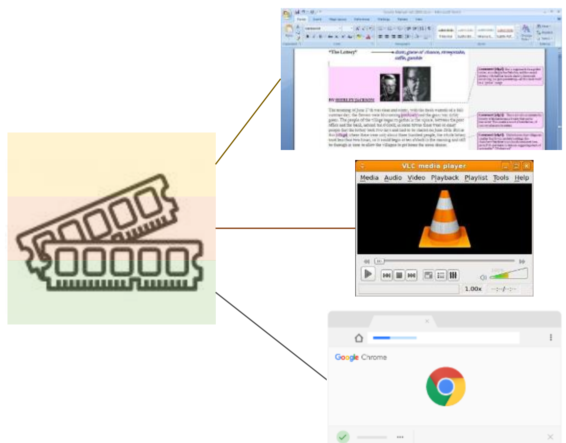
(b) Suppose that instead of a clock interrupt, a page fault occurs at tick 10 due to a read request to page 4. Show the contents of the new table entries. Explain. (You can omit entries that are unchanged.)

The File Abstraction

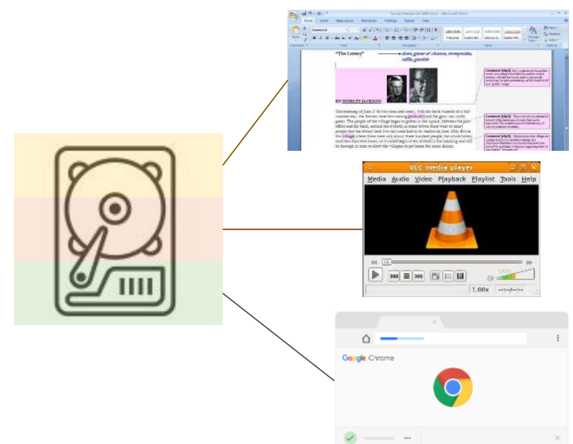
- A file is an abstraction
  - The OS abstracts away the concept of disk to offer files
  - Shield the user from the details about storage



Process, abstracts physical CPU

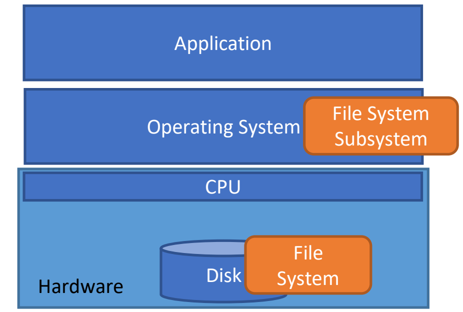


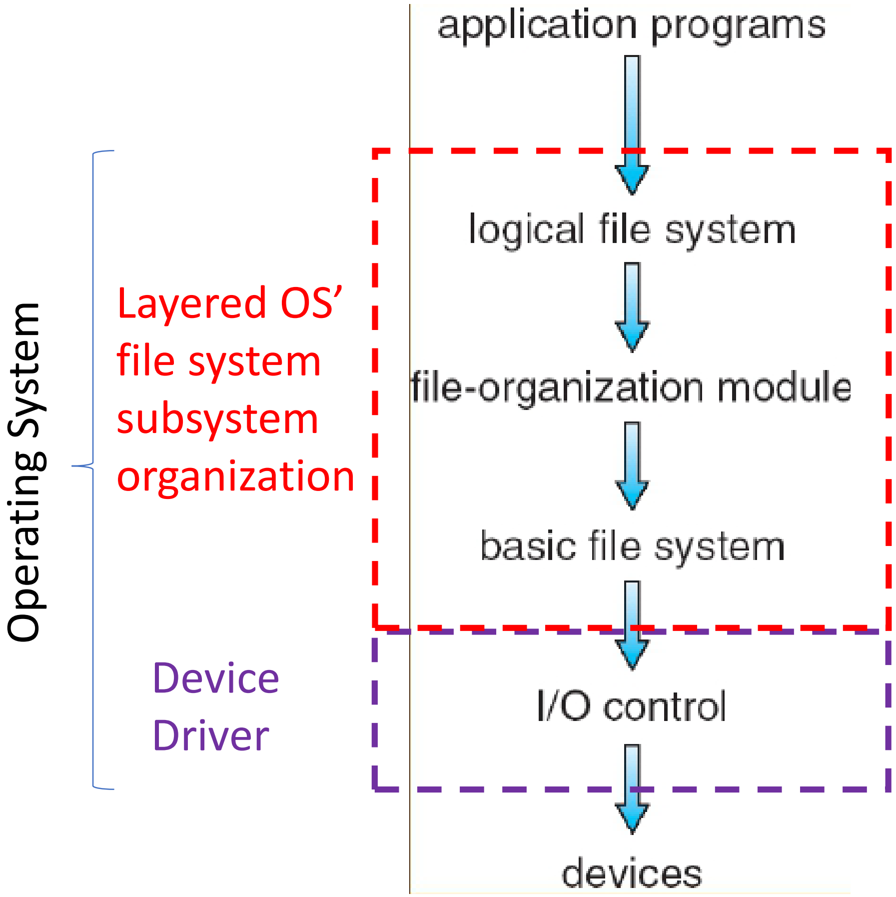
Address space, abstracts physical memory



File, abstracts disk

File System

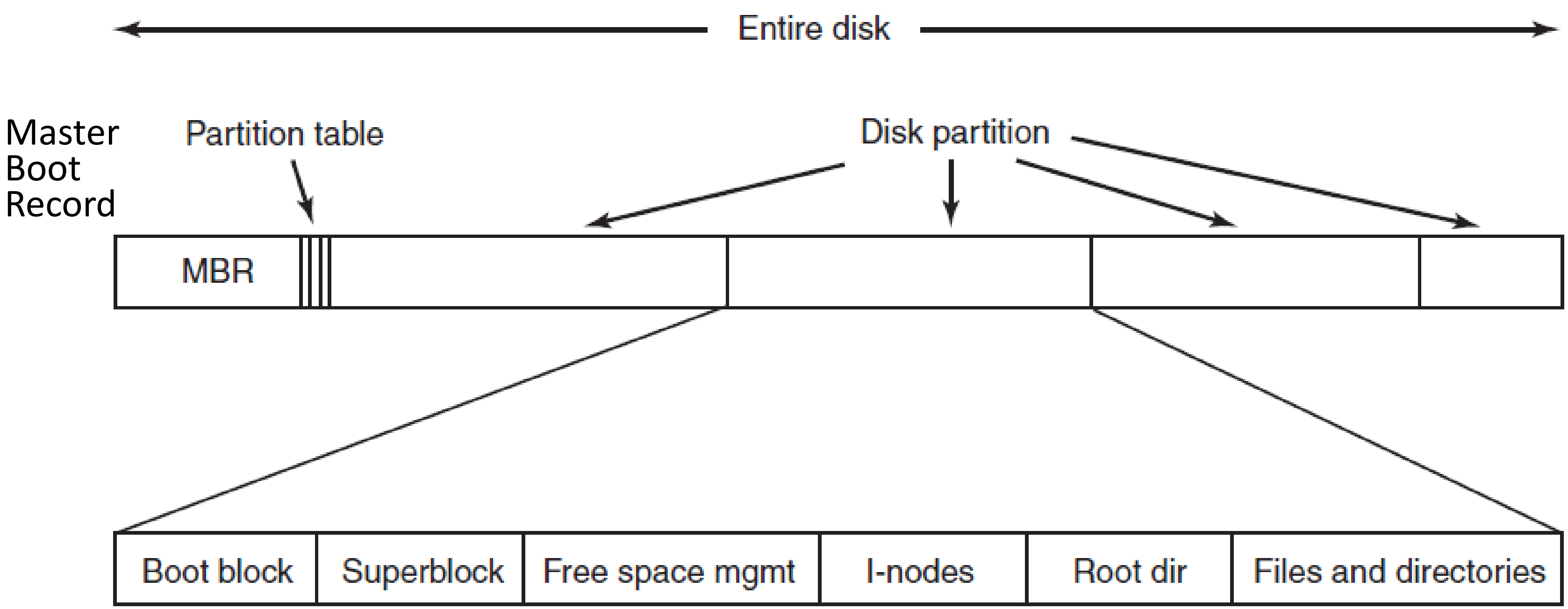




- **File** structure
  - A logical storage unit
  - Associated Metadata
  - Saved on secondary storage
- **File system** implements file structures
  - File system resides on secondary storage (disks)
  - OS' File system subsystem organized into layers

Disk and **File System** Layout

- A disk can be divided up into more partitions
  - Each with an independent file system



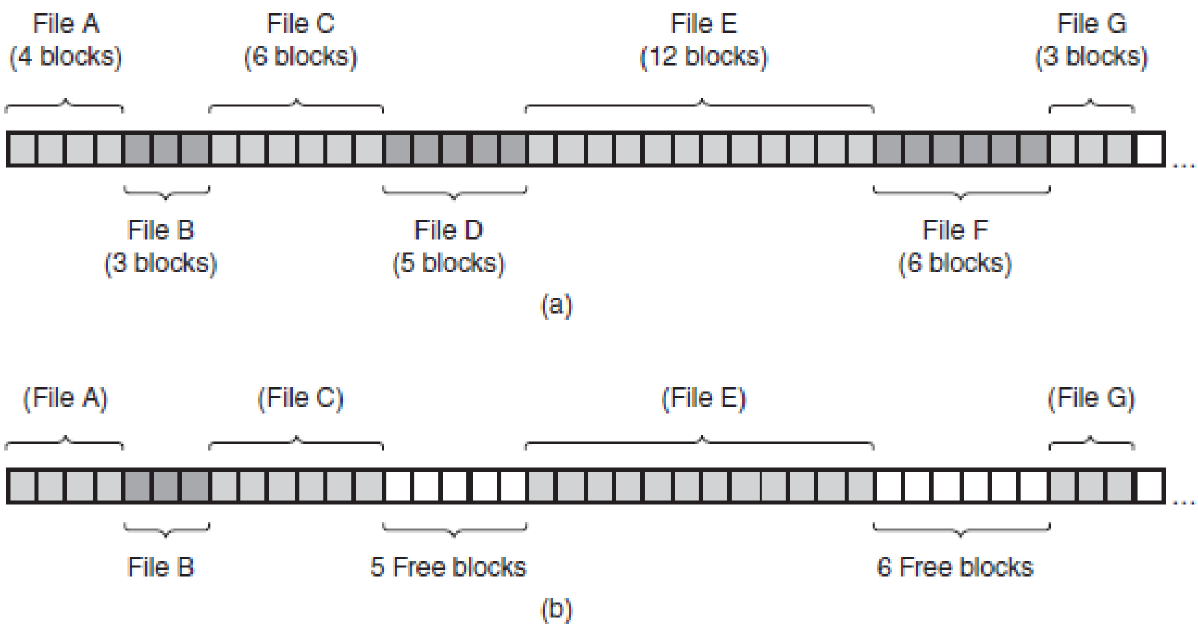
A possible disk and file system layout. (MOS Figure 4-9.)

Addressing

- Disk
  - Block is the minimal unit of allocation
  - Block = **logical disk address**/block size
- **Physical disk address**
  - In blocks
  - Absolute, from block zero
- **Logical file address** (LA)
  - In bytes
  - Relative to the beginning of the file (address 0)

Contiguous Allocation

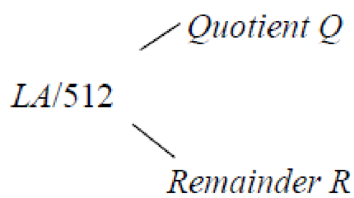
- Each file occupies a set of contiguous blocks
  - Entire blocks are used independently of the file size



(a) Contiguous allocation of disk space for seven files. (b) The state of the disk after files *D* and *F* have been removed.

Contiguous Allocation - Example

- Given a logical address *LA* of file A, how to map LA to its physical address (*B*,*D*) (B: block number; D: block offset)*?*
- Suppose the block size is 512 bytes



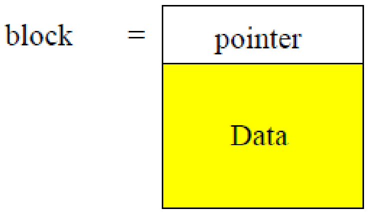
file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

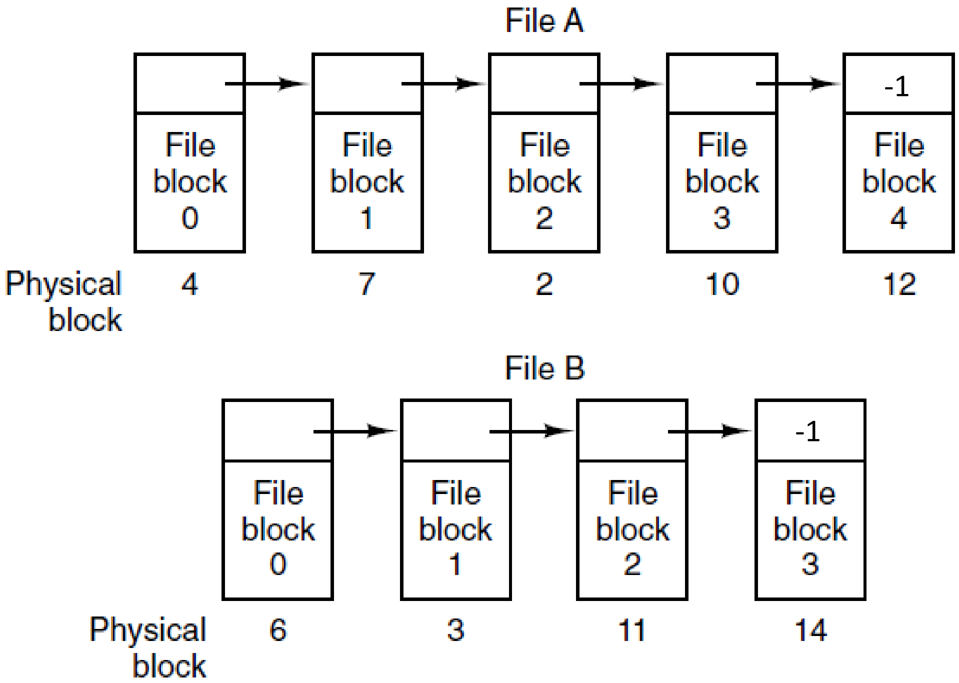
directory

- *Block number B = Q + starting address of file A in directory*
- *Block offset D = R*
- *Number of accesses to get the data at address LA*
  - *1 access for reading directory to get starting address of file A*
  - *1 access for reading data from block B at offset R*

Linked List Allocation

- Each file is a linked list of disk blocks
  - First bytes of the block contain a pointer to the next block
- Blocks may be scattered anywhere on the disk



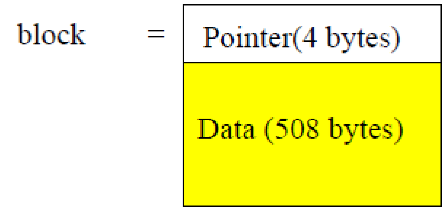


Storing a file as a linked list of disk blocks. (MOS Figure 4-11)

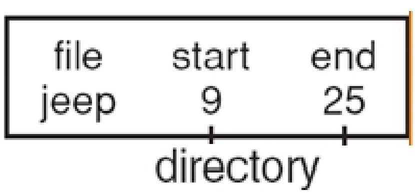
Linked List Allocation - Example

- Given a logical address *LA* of file A, how to map LA to its physical address (*B*,*D*) (B: block number; D: block offset)*?*
- Suppose block size is 512 bytes and each block contains 4 bytes reserved for pointer to next block

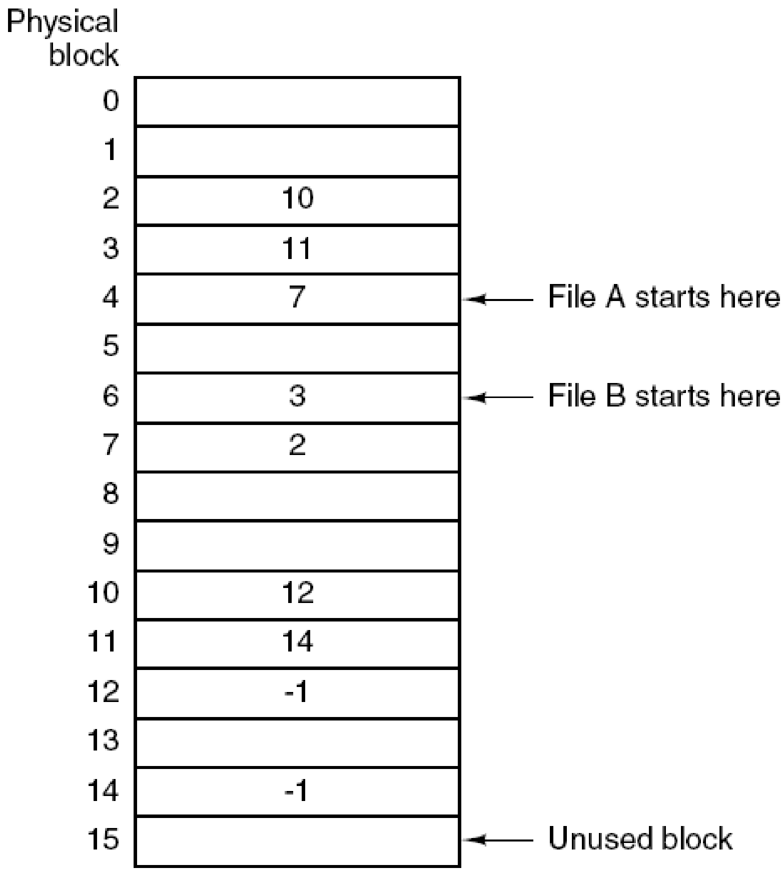
$LA/(512\text{-}4)$ → Quotient Q, Remainder R



- *Block number B* = *Q*th block in the linked chain of blocks, starting from the start block in directory
- *Block offset D* = *R* + 4
- *Number of accesses to get the data at address LA*
  - *1 access for reading directory to get starting block of file A*
  - *Q accesses for traversing Q blocks*



Linked-List Allocation with Table



Linked list allocation using a file allocation table in main memory (MOS Figure 4-12.)

- Variant of linked list allocation
  - File-Allocation Table (FAT)
- Keep a table in memory, each entry
  - corresponds to disk block number
  - contains a pointer to the next block or -1
- (Information about all files on the disk)

Linked List Allocation with Table-Example

- Given a logical address *LA* of file A, how to map LA to its physical address (*B*,*D*) (B: block number; D: block offset)*?*
- Suppose the block size is 512 bytes

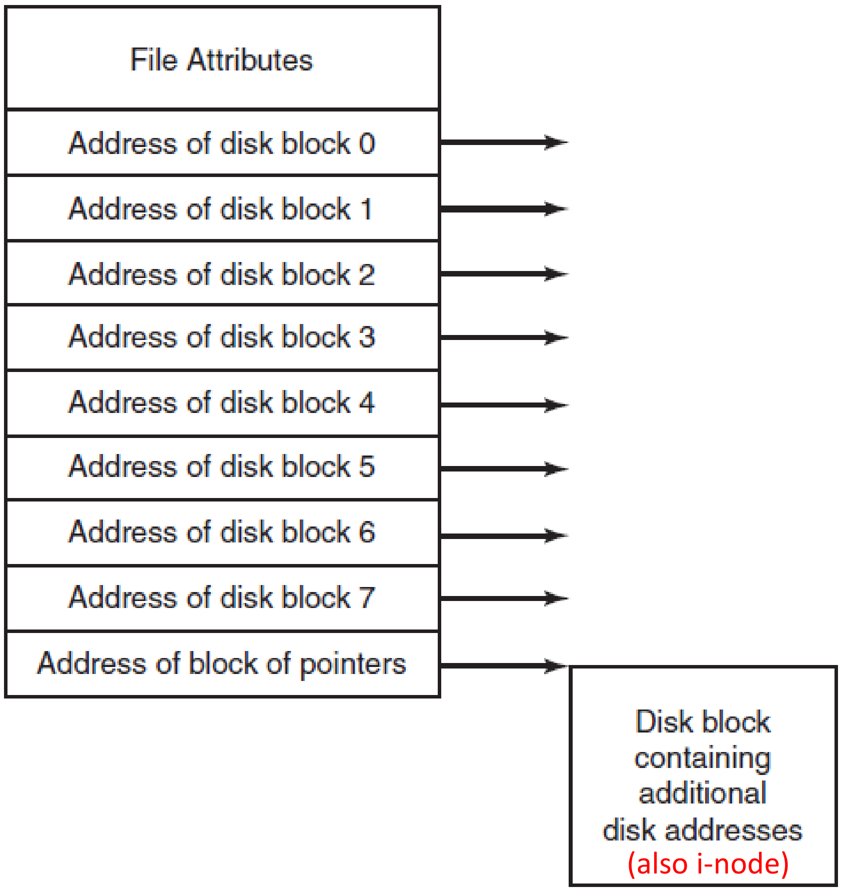
$$LA/512 \begin{cases} \text{Quotient } Q \\ \text{Remainder } R \end{cases}$$

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

directory

- *Block number B* = *Q*th block in the linked chain of blocks, starting from the start block in directory
- *Block offset D = R*
- *Number of accesses to get the data at address LA*
  - *1 access for reading directory to get starting address of file A*
  - *Q accesses for reading data from block B at offset R* ***IN MEMORY***
  - *1 access for reading data from block Qth at offset R*

Indexed Allocation (1)

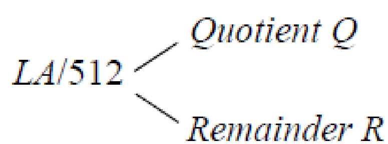


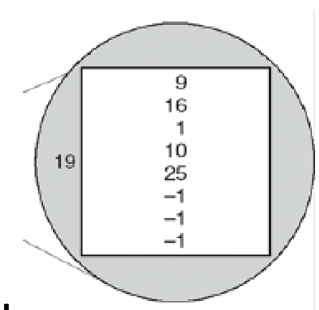
An example i-node. (MOS Figure 4-13.)

- Associate a data structure **(block size)** called index-node (i-node) to each file
  - Lists the attributes
  - Lists the disk's address of the file's blocks
- i-nodes may chain

Indexed Allocation - Example

- Given a logical address *LA* of file A, how to map LA to its physical address (*B*,*D*) (B: block number; D: block offset *)*?
- Assume the block size is 512 bytes

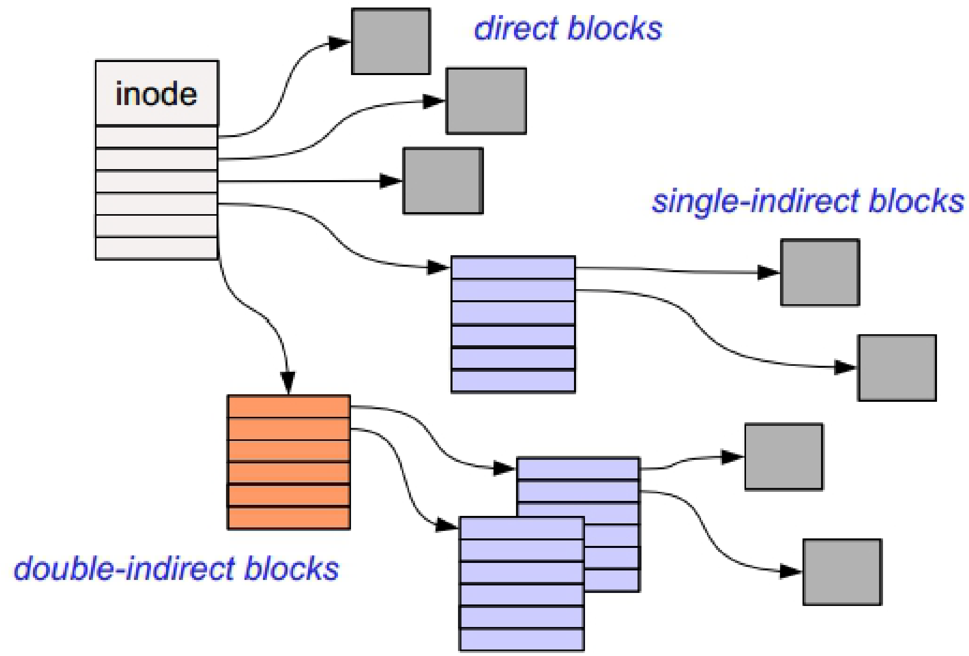




- Block number B: Look up the Q-th entry of the index table to obtain *B*
- *Block offset D = R*
- *Number of disk accesses to get the data at address LA*
  - *1 access for reading directory to get i-node address of file A*
  - *1 access for reading the index table*
  - *1 access to get data at block offset D*

Indirection in Indexed Allocation (1)

- i-node can contain a pointer to
  - direct block (data block)
  - single indirect
  - double indirect
  - triple indirect blocks
  - …
- Allows file to grow and to be incredibly large!



Exercise (1/4)

- Assume the disk blocks are of size 1 KB, and each block pointer is of 4 bytes. How large can a file be with...
  - (1) A single-level indirect node?

- **ANSWER:** How many block pointers can be stored in one block?
  - 1KB / 4 = 256 block pointers
  - File size: 256 * 1KB = 256 KB

Exercise (2/4)

- Assume the disk blocks are of size 1 KB, and each block pointer is of 4 bytes. How large can a file be with...

(2) A double-level indirect node?

- **ANSWER:**
  - 256 * 256 * 1KB = 65536 KB = 64 MB

Exercise (3/4)

- Assume the disk blocks are of size 1 KB, and each block pointer is of 4 bytes. How large can a file be with...

(3) A triple-level indirect node?

- **ANSWER:**
  - 256 * 256 * 256 * 1KB = 16 GB

Exercise (4/4)

- Assume the disk blocks are of size 1 KB, and each block pointer is of 4 bytes. How large can a file be with …

an i-node that points to 13 direct blocks + 1 single-level indirect node + 1 double level indirect node + 1 triple-level indirect node?

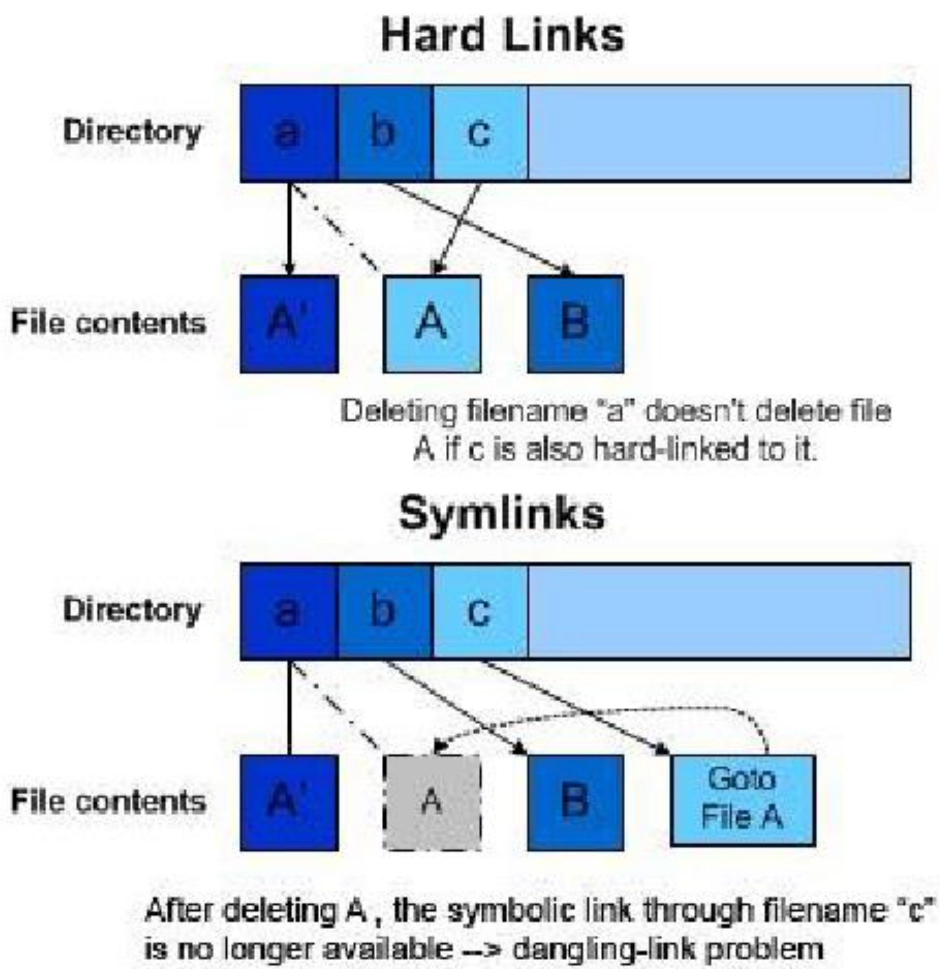
- **ANSWER:**
  - (13 * 1KB) + (256 * 1KB) + (256 * 256 *1KB) + (256 * 256 * 256 * 1KB) = 16.06 GB

Question

- **QUESTION:** Consider a file whose size varies between 4 kB and 4 MB during its lifetime. Which of the four allocation schemes (contiguous, linked-list, linked-list with table, indexed) will be most appropriate?

- **ANSWER:** Since the file size changes a lot, contiguous allocation will be inefficient requiring reallocation of disk space as the file grows in size and compaction of free blocks as the file shrinks in size. Both linked and table/indexed allocation will be efficient; between the two, table/indexed allocation will be more efficient for random-access scenarios.

Hard vs Soft Links



I/O Devices (1)

- Block devices
    - Stores information in fixed-size blocks
    - Each block has its own address
    - Transfers are in units of entire blocks
- Examples: hard disk, blu-ray disc, USB

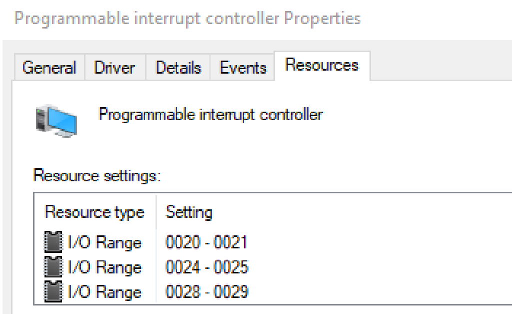
- Character devices
    - Delivers or accepts stream of characters, no block structure
    - Not strictly addressable, does not have any *seek* operation
- Examples: printers, network interfaces, serial line, mouse

- Some devices do not fit into this classification
- Examples: clocks, memory-mapped screens

Communication Mechanisms

- CPU initiated communication
  - Register and buffers
    - I/O Ports
    - Memory mapped I/O
    - Hybrid
- Offloaded communication
  - DMA
- I/O device notification
  - Interrupt

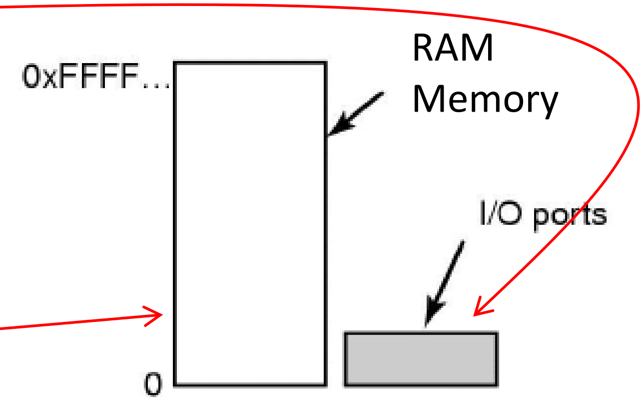
[Computer Architecture Review]



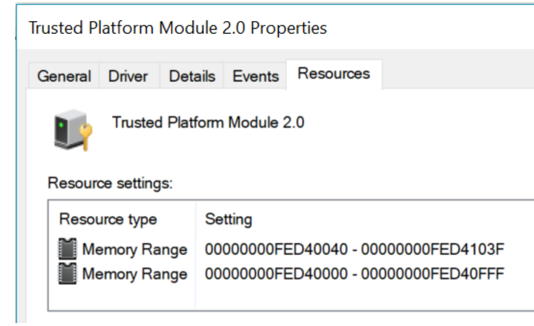
I/O Ports

- Each control register an I/O port number
- Special instructions to access the I/O port space
  - CPU reads in from device I/O PORT to CPU register
    - IN REG, PORT
  - CPU writes to device I/O PORT from CPU register
    - OUT PORT, REG
- Instruction are privileged (OS kernel only)
- Separate **I/O port space** and **memory space**
  - I/O instructions
    - *IN R0, 4*
    - *OUT 4, R0*
  - Similar memory access instruction
    - *MOV R0, 4*
    - *MOV 4, R0*

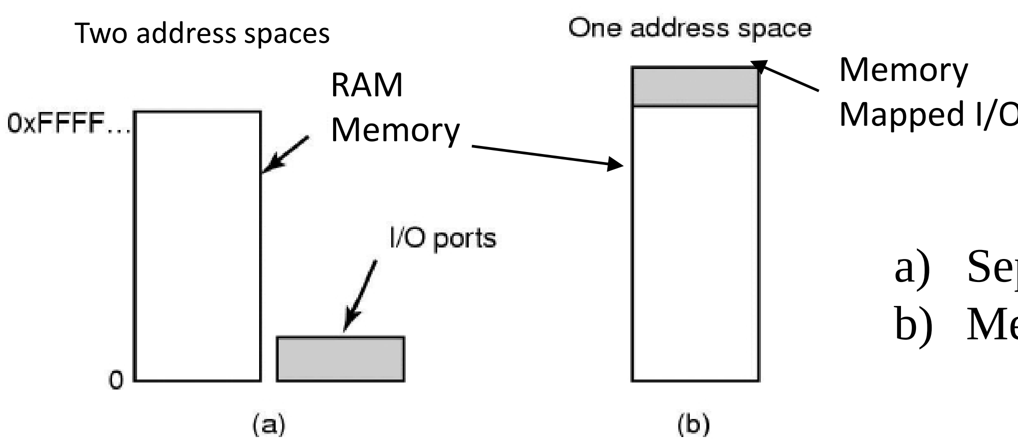
Two address spaces



Memory-mapped I/O

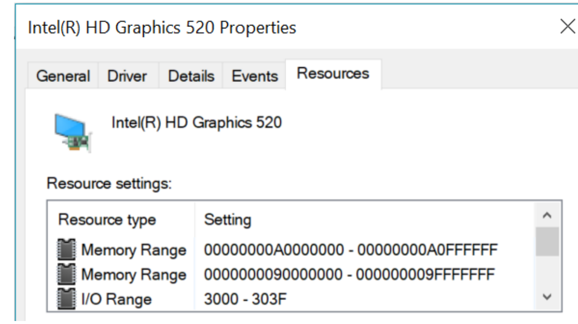


- All control registers and buffers into the memory space
- Each control register is assigned a unique memory address
  - There is no actual RAM memory for this address
- Such addresses may be at the top of the physical address space

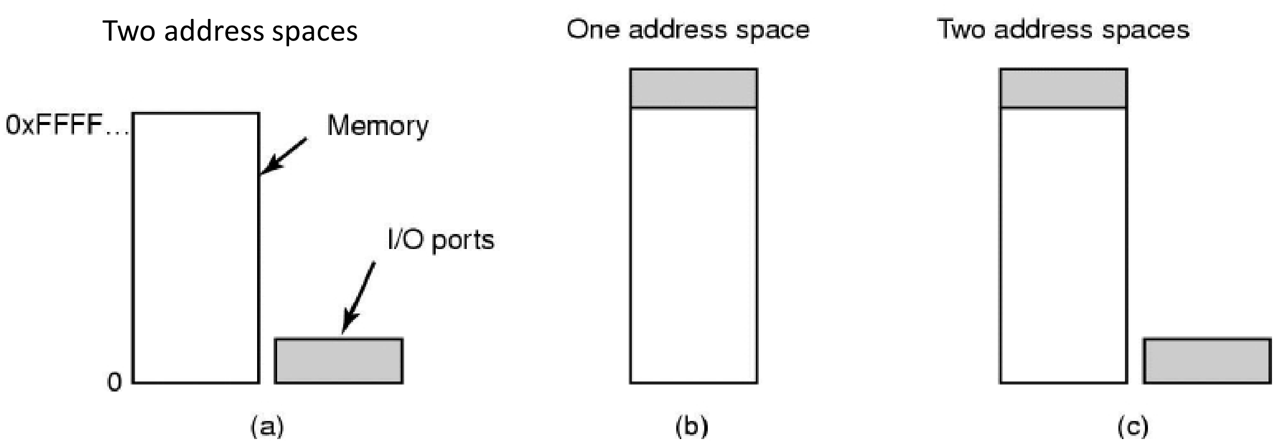


a) Separate I/O and memory space
b) Memory-mapped I/O

Hybrid



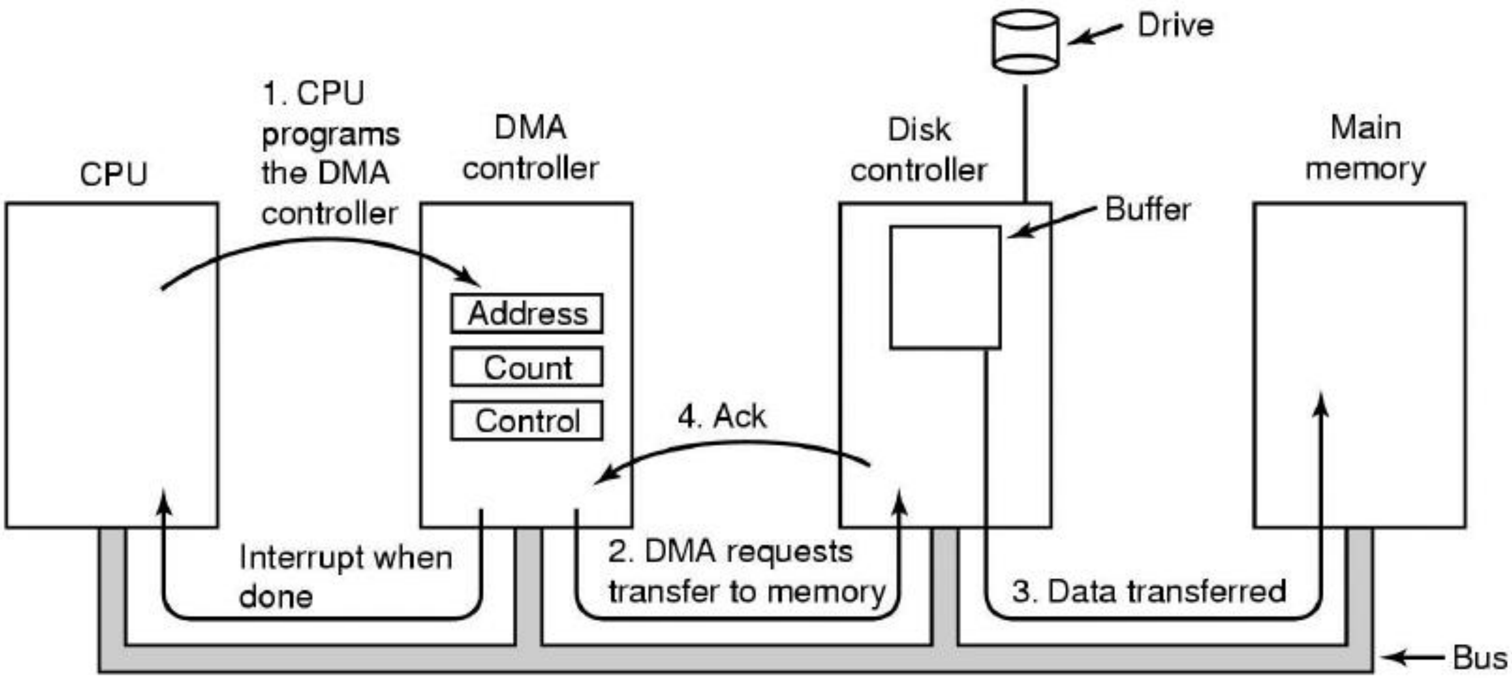
- I/O ports and memory-mapped IO
- Example
  - Memory-mapped I/O data buffers and separate I/O ports for the control registers
  - x86 CPUs, memory addresses 640K to 1M – 1 being reserved for device data buffers, in addition to I/O ports 0 to 64K – 1



a) Separate I/O and memory space
b) Memory-mapped I/O
c) Hybrid

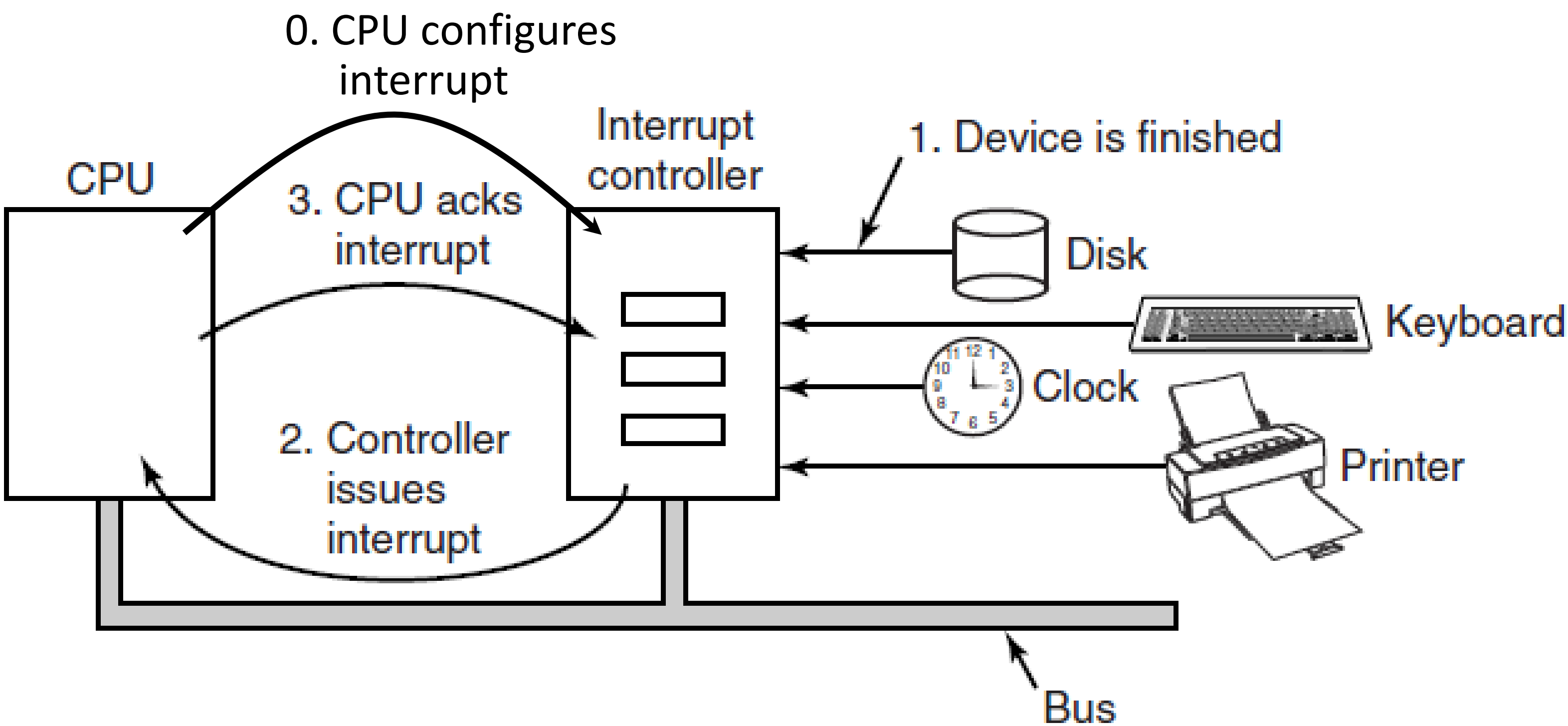
How Does DMA Work?

- CPU commands DMA controller by writing to control registers of DMA device



Example of a DMA transfer: transfer data from disk to memory (MOS Figure 5-4)

External Interrupts



How an interrupt happens. The connections between the devices and the interrupt controller actually use interrupt lines on the bus rather than dedicated wires. (MOS Figure 5-5.)

What are the differences between *internal* and *external* interrupts?

How Software Interacts with I/O?

- Programmed I/O
  - CPU does all the work

Uses
• I/O ports and/or memory-mapped I/O

- Interrupt-driven I/O
  - CPU does the work
  - But interrupts tell when

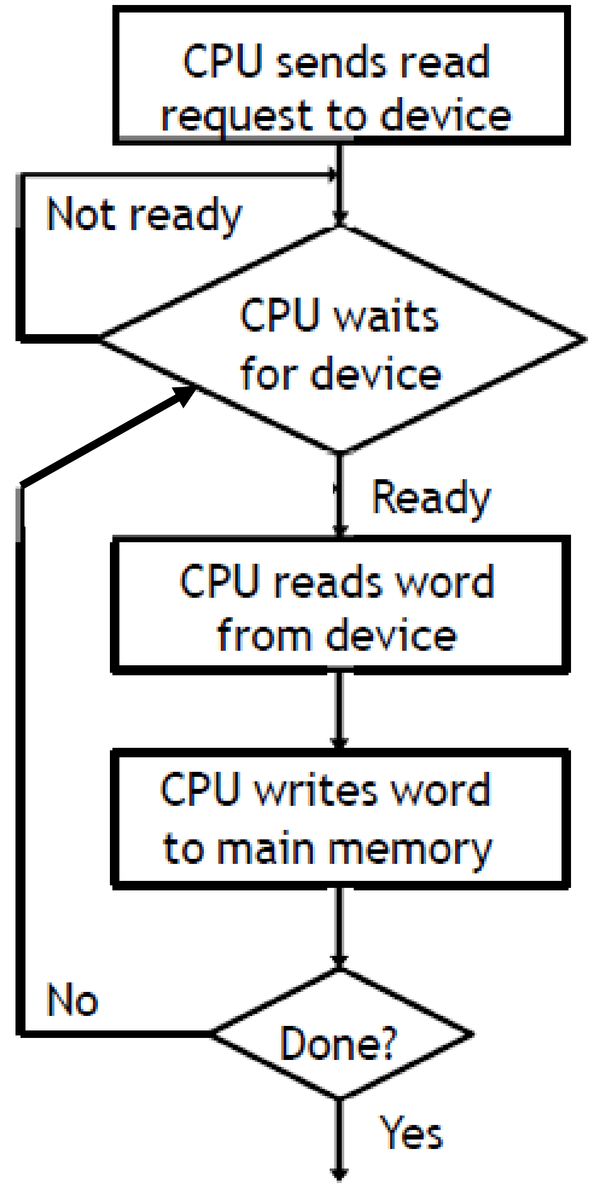
Uses
• I/O ports and/or memory-mapped I/O • Interrupts

- I/O using DMA
  - DMA controller does all the work
  - It uses interrupts for notification
  - But CPU needs to program the DMA controller

Uses
• I/O ports and/or memory-mapped I/O • Interrupts • DMA

Programmed I/O (1)

- CPU writes/reads a byte/word at a time
- from/to main memory to/from device
  - CPU makes a request, then waits for the device to become ready
  - Buses are only byte/word wide, so the last few steps are repeated for large transfers
- CPU time is wasted
  - If device is slow, the CPU may wait a long time



Applies to I/O Ports, Memory Mapped I/O, and Hybrid

Programmed I/O (2)

- CPU continuously polls the device to see if it is ready to accept another one
  - **Polling** or **busy waiting**

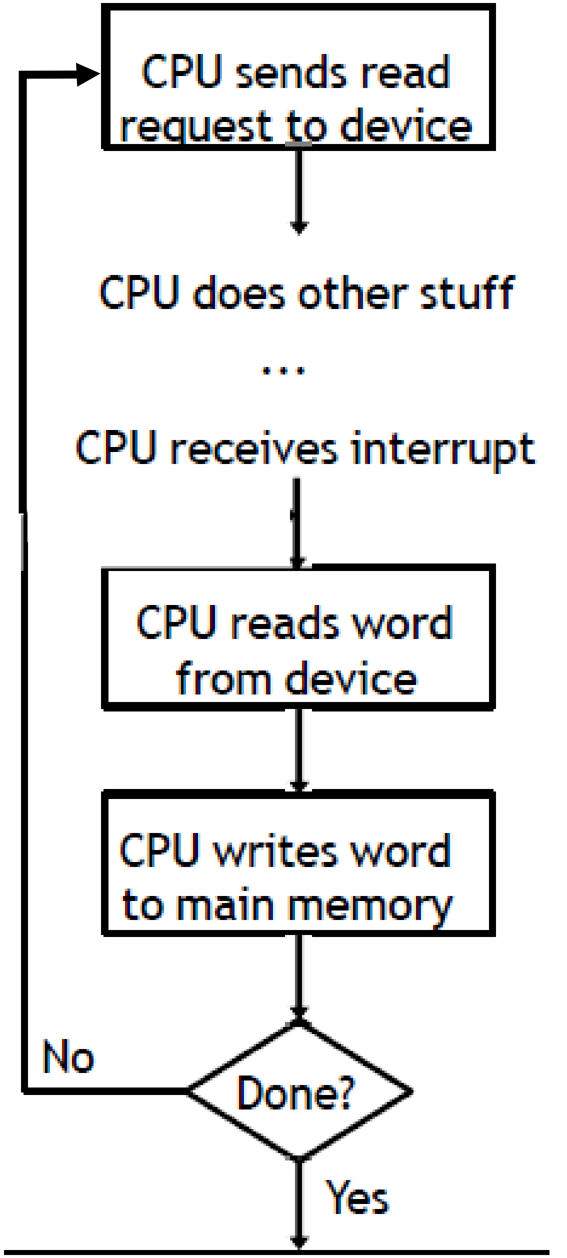
```
copy_from_user(buffer, p, count);             /* p is the kernel buffer */
for (i = 0; i < count; i++) {                 /* loop on every character */
        while (*printer_status_reg != READY) ; /* loop until ready */
        *printer_data_register = p[i];         /* output one character */
}
return_to_user( );
```

Figure 5-8. Writing a string to the printer using programmed I/O.

Is this using I/O port or memory-mapped I/O?

Interrupt-driven I/O (1)

- Why interrupts?
  - I/O devices are slower than memory, or CPU
  - Uncertainty of when device will be ready
- OS needs to know when
  - The I/O device has completed an operation
  - The I/O operation has encountered an error
- Instead of waiting
  - The CPU continues with other computations
  - The device interrupts the processor when
    - Operation completes
    - There is an error



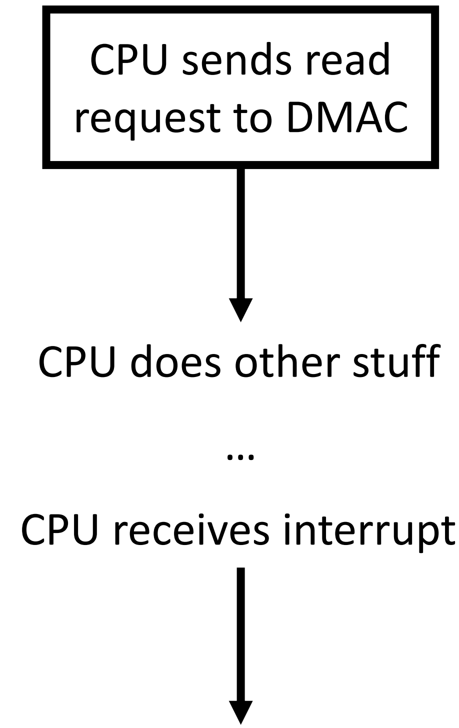
I/O Using DMA (1)

- Direct Memory Access (DMA)
  - Device read/write directly from/to memory
  - It has access to the **system bus** independent of the CPU

a) The CPU commands the operation

b) DMA does the transfer

c) When transfer is complete, DMAC notifies the CPU with an interrupt



Question

- A typical printed page of text contains 50 lines of 80 characters each. Imagine that a certain printer can print 6 pages per minute and that the time to write a character to the printer's output register is so short it can be ignored. Does it make sense to run this printer using interrupt-driven I/O if each character printed requires an interrupt that takes 50 μsec all-in to service?

Answer

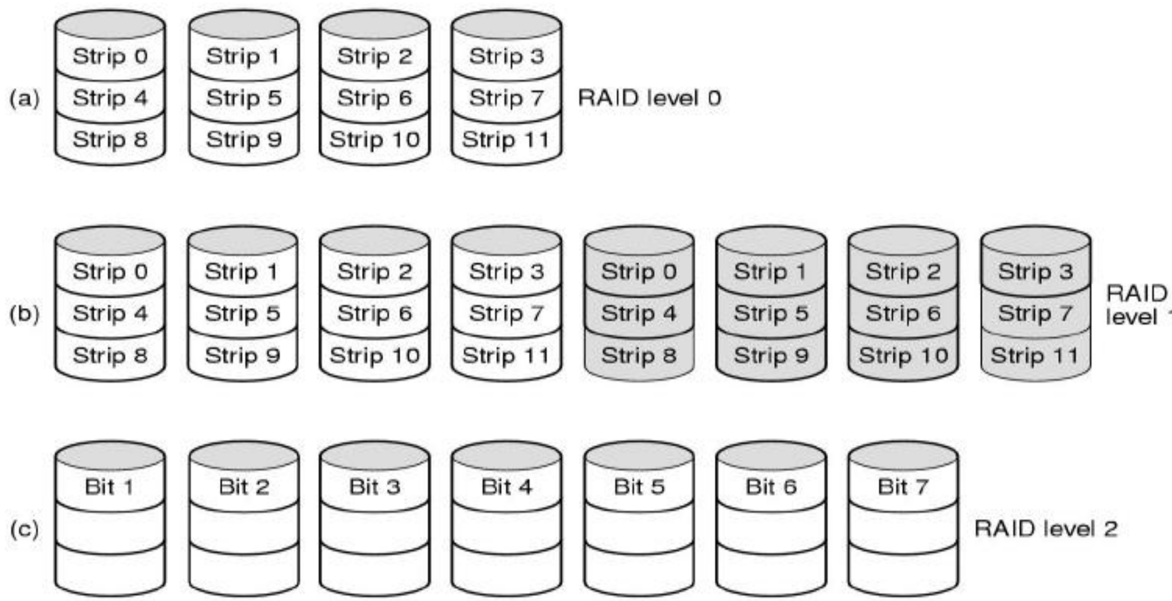
- The printer prints 50 x 80 x 6 = 24,000 characters/min, which is 400 characters/sec. Each character uses 50 µsec of CPU time for the interrupt, so collectively in each second the interrupt overhead is 20 msec. Using interrupt-driven I/O, the remaining 980 msec of time is available for other work. In other words, the interrupt overhead costs only 2% of the CPU, which will hardly affect the running program at all.

RAID

- **Redundant Array of Independent Disks**
    - **Improve performance (parallel access)**
    - **Larger size disk**
    - **Reliability, fail independently**

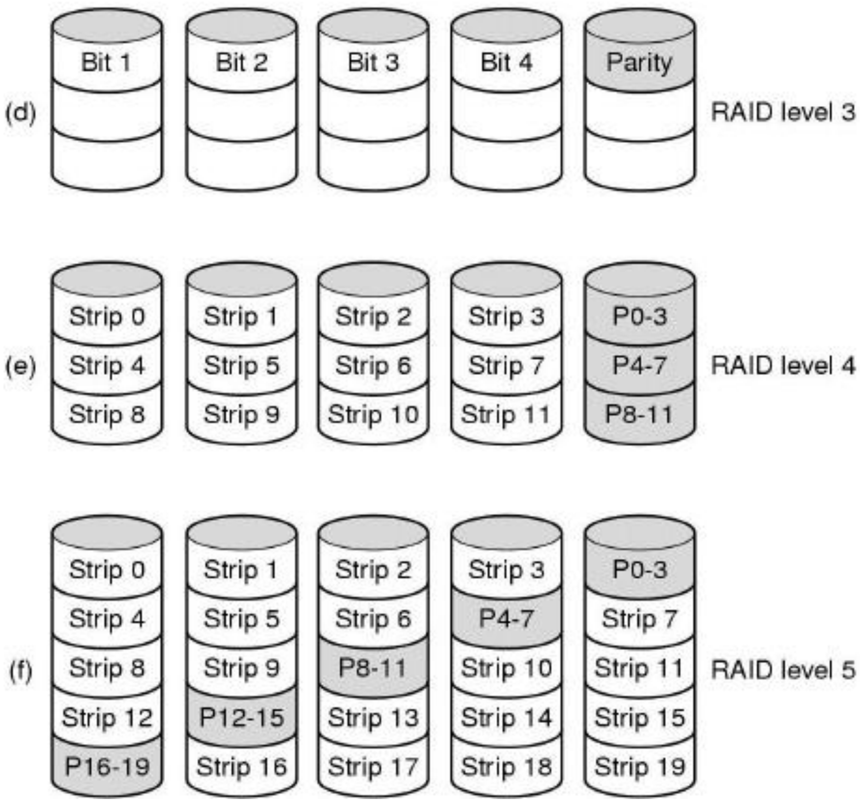
- Solutions
    - RAID 0 – Striping
    - RAID 1 – Mirroring
    - RAID 2 – Bit-level striping with Hamming-code
    - RAID 3 – Bit-level striping with parity
    - RAID 4 – Striping with parity
    - RAID 5 – Striping with distributed parity
    - RAID 6 – Striping with double distributed parity

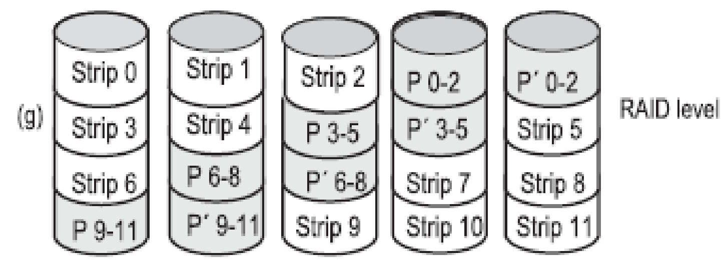
Summary: RAID 0, RAID 1, RAID 2



a) Strips of data blocks (sectors) distributed across disks
b) Mirror data in extra disks
c) Bits distributed across disks - Hamming(7,4)

Summary: RAID 3, RAID 4, RAID 5, RAID 6





d) Store parity bit for each data word: error detection/correction
e) Strip-to-strip parity
f) Distributing the parity strip uniformly over all drives in round-robin fashion
g) Double parity: multiple drives failure

Problem

- A RAID 5 can fail if two or more of its drives crash within a short time interval. Suppose that the probability of one drive crashing in a given hour is *p*. What is the probability of a *k* - drive RAID failing in a given hour?

A RAID can fail if two or more of its drives crash within a short time interval. Suppose that the probability of one drive crashing in a given hour is p. What is the probability of a k-drive RAID failing in a given hour?

The probability of 0 failures, P0, is (1 − p)^k. The probability of 1 failure, P1, is kp(1−p)^(k−1).
The probability of a RAID failure is then 1 − P0 − P1. This is 1−(1−p)^k-kp(1−p)^(k−1).