Final Advice

- Think about what was covered by projects, homeworks, and the midterm
  - This slide deck should give you some focus for the final, but isn't in and of itself sufficient for adequate study
- I will not expect you to write code on the exam
- I will not expect you to recall obscure facts
- I will expect that you have:
  - an intuition about pros/cons of various system design choices, from processes on up
  - reasonable recall of fundamental concepts
  - the ability to perform simple “back of the envelope” calculations (no calculators)

What Was Covered

- Lectures cover material found in both:

  MOS

  - 1.5, 1.6, 2, 3, 4, 5

    and (alternatively)

  OSTEP

  - 4-10, 13-18, 20, 26-28, 30, 31, 36-44

Review

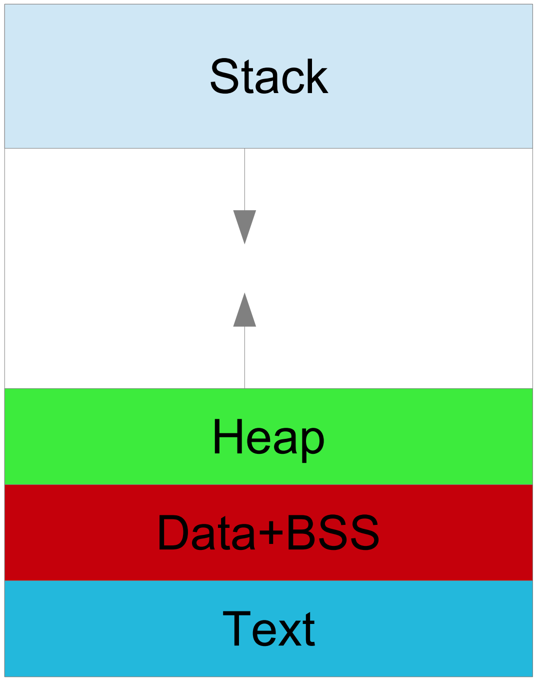
- Processes
- Threads
- Synchronization
- Scheduling
- Memory/VM
- Paging
- Segmentation
- Files/Directories
- File Systems
- I/O
- Interrupts/DMA
- Disk drives

Processes

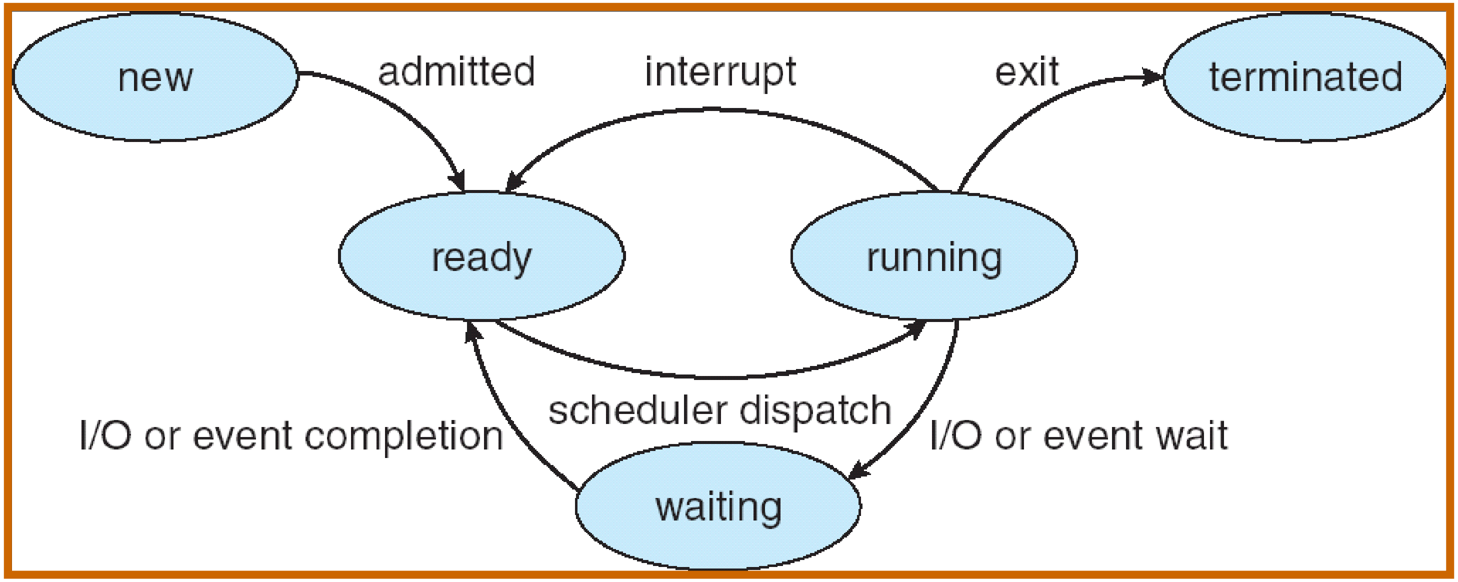
- What is a process?
  - Program in execution
- What does it look like?
  - Think about arrangement in memory
- How are they managed?
  - Creation (unix style): fork() and exec()
  - States: new, ready, running, waiting, terminated
  - Synchronization and scheduling

What is a process?

- Process is a program in execution
  - Program = code and static data in a file
  - Process = a program's execution context
    - Each process has own address space
    - Memory map:
      - Text: compiled program
      - Data: initialized static data
      - BSS: uninitialized static data
      - Heap: dynamically allocated memory
      - Stack: call stack
    - Process context
      - Program counter (PC)
      - CPU registers



Process States

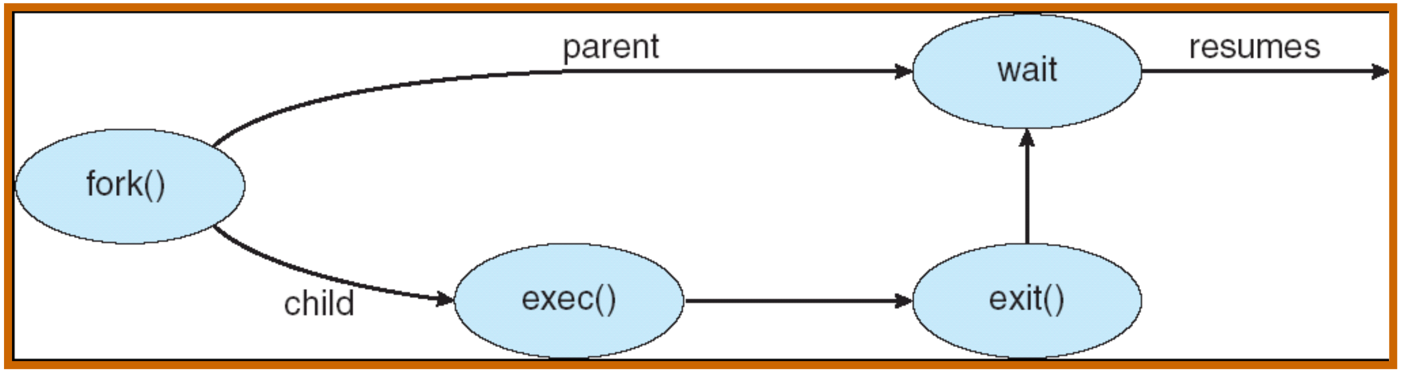


fork() and exec()

- In UNIX, only fork() can create a new process
- What does it do?
  - Copies all of the calling process' memory
  - Copies all but one register (process id is 0 if child)
  - Copies all file descriptors and references to open file state
- exec() replaces current process image with a new one
- Creating new processes is done by a fork() followed by exec()

Now we're forking...

```
#include <stdio.h>
main(int argc, char** argv){
    int pid;
    switch (pid = fork()){
    case 0: printf("this is the child\n"); break;
    default: printf("this is the parent of %d\n", pid); break;
    }}
```



Shell: fork() and exec()

```
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 */
    }
}
```

Some Important Topics

- Process state
- Process creation
  - Copy on write
- PCB
- Context switching
- CPU utilization/multiprogramming

Review

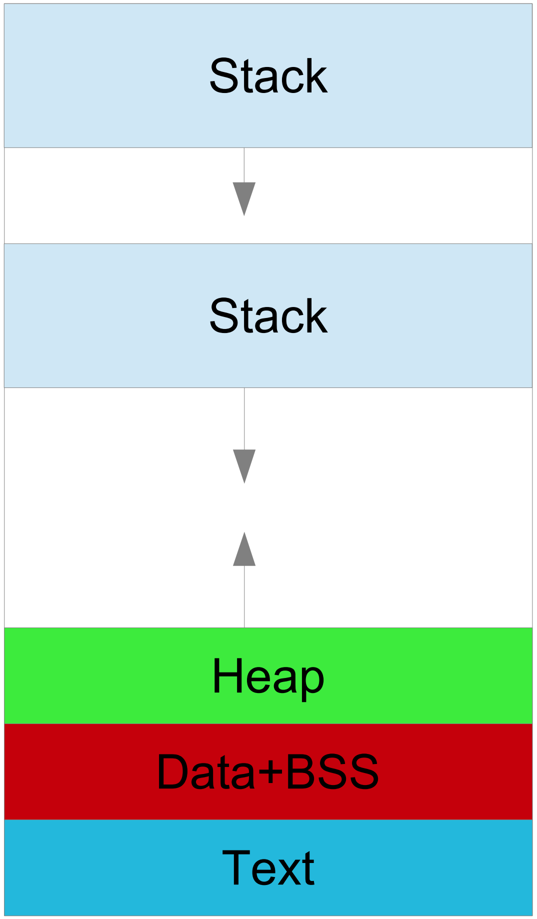
- Processes
- Threads
- Synchronization
- Scheduling
- Memory/VM
- Paging
- Segmentation
- Files/Directories
- File Systems
- I/O
- Interrupts/DMA
- Disk drives

Threads

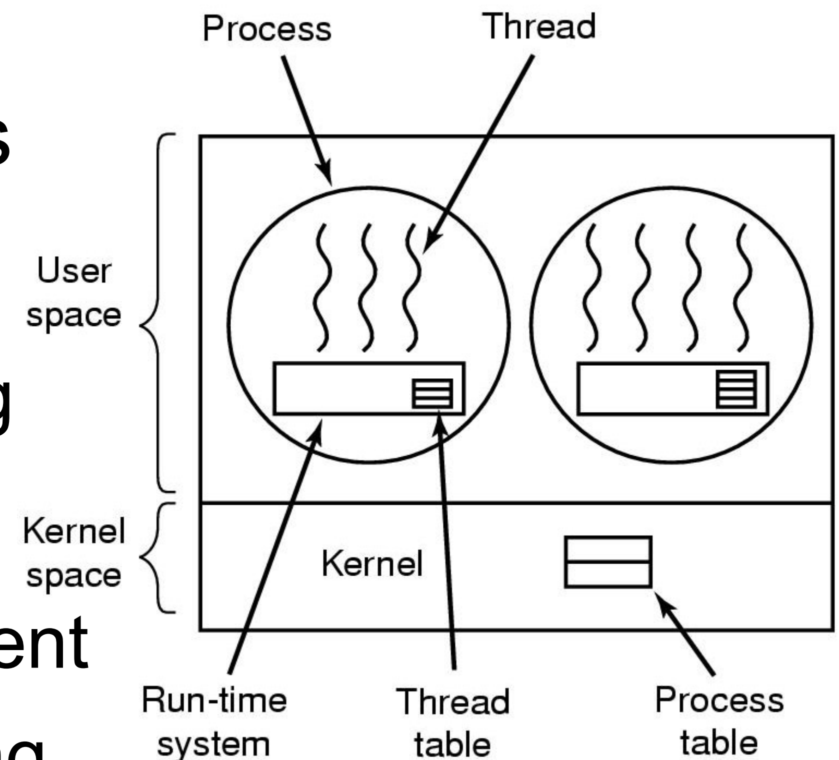
- What are threads?
  - Lightweight processes
- What kinds of threads can we have?
  - User threads
  - Kernel threads
  - Pros? Cons?

Threads

- Memory map shows process with 2 threads
- Shared: Text, data, BSS, heap,
  working dir, signals,
  open fd's, user/group ids
- Unique: Thread ID, stack, signal
  mask, priority, registers,
  stack pointer, instruction
  pointer

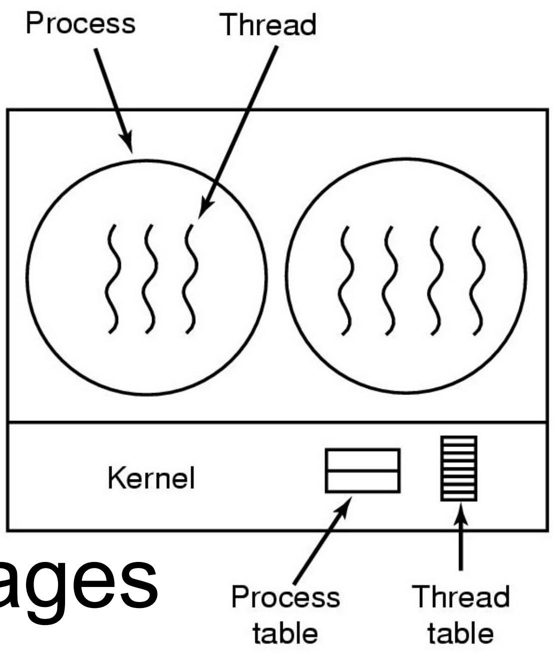


Threads in User Space



- Kernel not aware of threads
- Thread library
  - Scheduling and bookkeeping
- Benefits:
  - Fast creation and management
  - Allows customized scheduling
- Problem:
  - Blocking system calls
    - If one thread blocks, then whole process blocks

Threads in the Kernel



- Kernel creates, schedules and manages threads
  - Scheduling on a per-thread basis
- Blocking sys calls do not block entire process
- Slower than user-level
  - Must call kernel

Some Important Topics

- User vs Kernel Threads
  - Pros/cons, etc.
  - Scheduling
  - Implementation choices

Review

- Processes
- Threads
- Synchronization
- Scheduling
- Memory/VM
- Paging
- Segmentation
- Files/Directories
- File Systems
- I/O
- Interrupts/DMA
- Disk drives

Process Synchronization

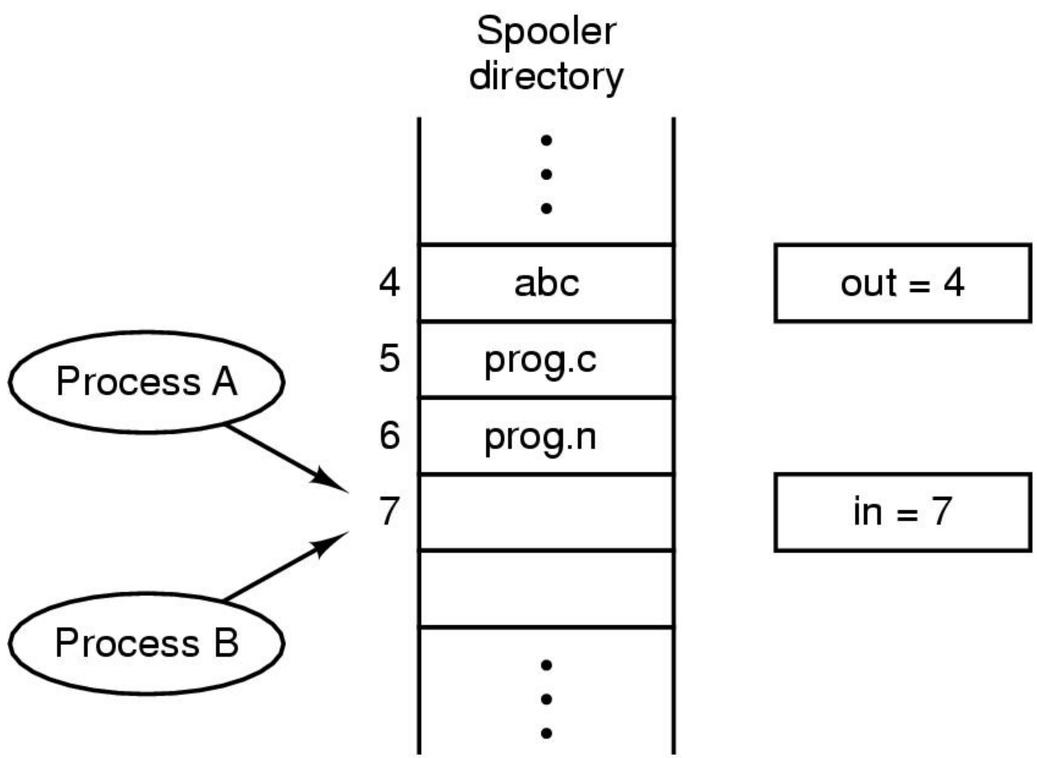
- Cooperating processes may share data via:
  - Shared address space (threads)
  - Shared memory objects
  - Shared files
- What about processes accessing same data simultaneously?

Thread Example

Thread 1	Thread 2
a = 0	a = 0
b = 0	b = 0
b++	b++
a = b	a = b

- Final values of a?
  - a == 0 ?   a == 1?   a == 2?   a == 3?   a == b?

Race Conditions



- Two processes access same memory at same time
  - Unpredictable behavior

Semaphores

- Synchronization variables
  - Take on positive values
- Two atomic operations
  - Down (wait): waits for semaphore to become greater than zero, decrements it by 1
  - Up (signal): increments semaphore by 1

Semaphores (2)

- Waiting queue for each semaphore stores:
  - Process id
  - Pointer to next record in queue
- Operations
  - Block: places process invoking the operation on the wait queue
  - Wakeup: removes one process from wait queue, places it in ready queue

Semaphores (3)

```
down (*value){
        value--;
         if (value < 0) { //add this process to waiting queue
             block();  }
}

up (*value){
         value++;
          if (value <= 0) {  //remove process from waiting queue
              wakeup(P);  }
}
```

Some Important Topics

- Race conditions
- Critical regions
- Mutual exclusion
  - Choices? Disable interrupts, locks, alternation, etc.
- Producer-consumer
- Semaphores, locks, monitors

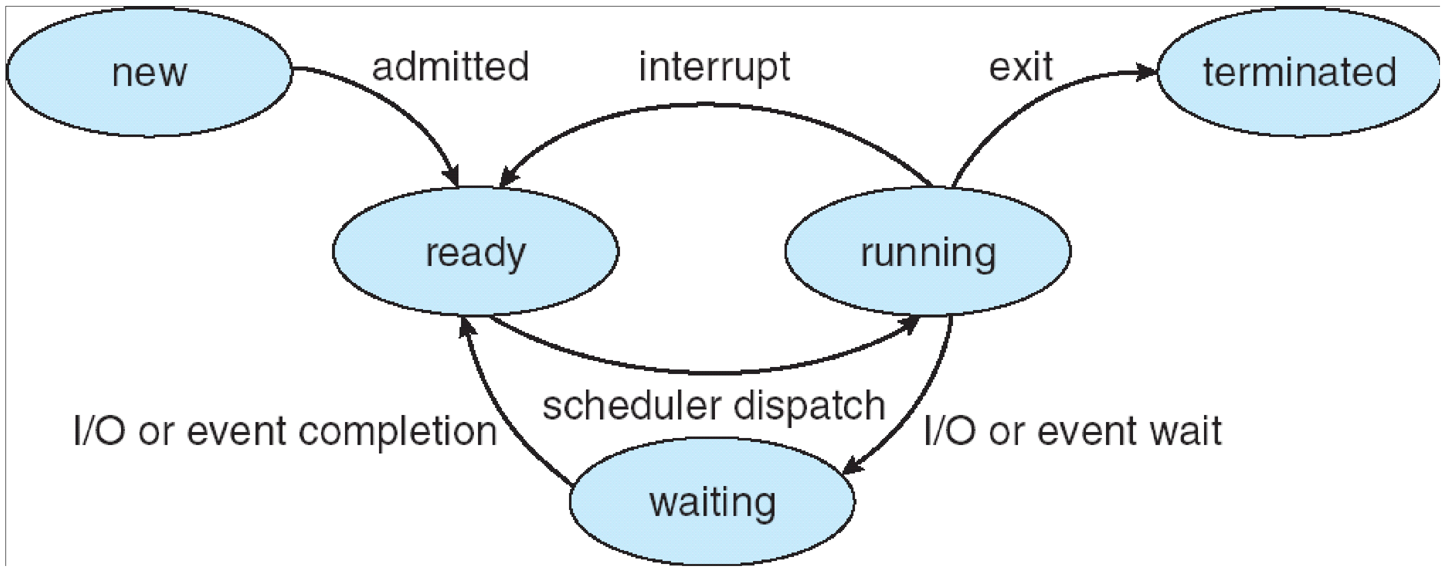
Review

- Processes
- Threads
- Synchronization
- Scheduling
- Memory/VM
- Paging
- Segmentation
- Files/Directories
- File Systems
- I/O
- Interrupts/DMA
- Disk drives

When?

- New process creation
  - Run parent or child?
- Process exit
  - Run idle process if none are available
- Process blocked
- Preemptive
  - Process made to relinquish control
- Nonpreemptive
  - Process runs until blocked or until it yields

Here's a Picture



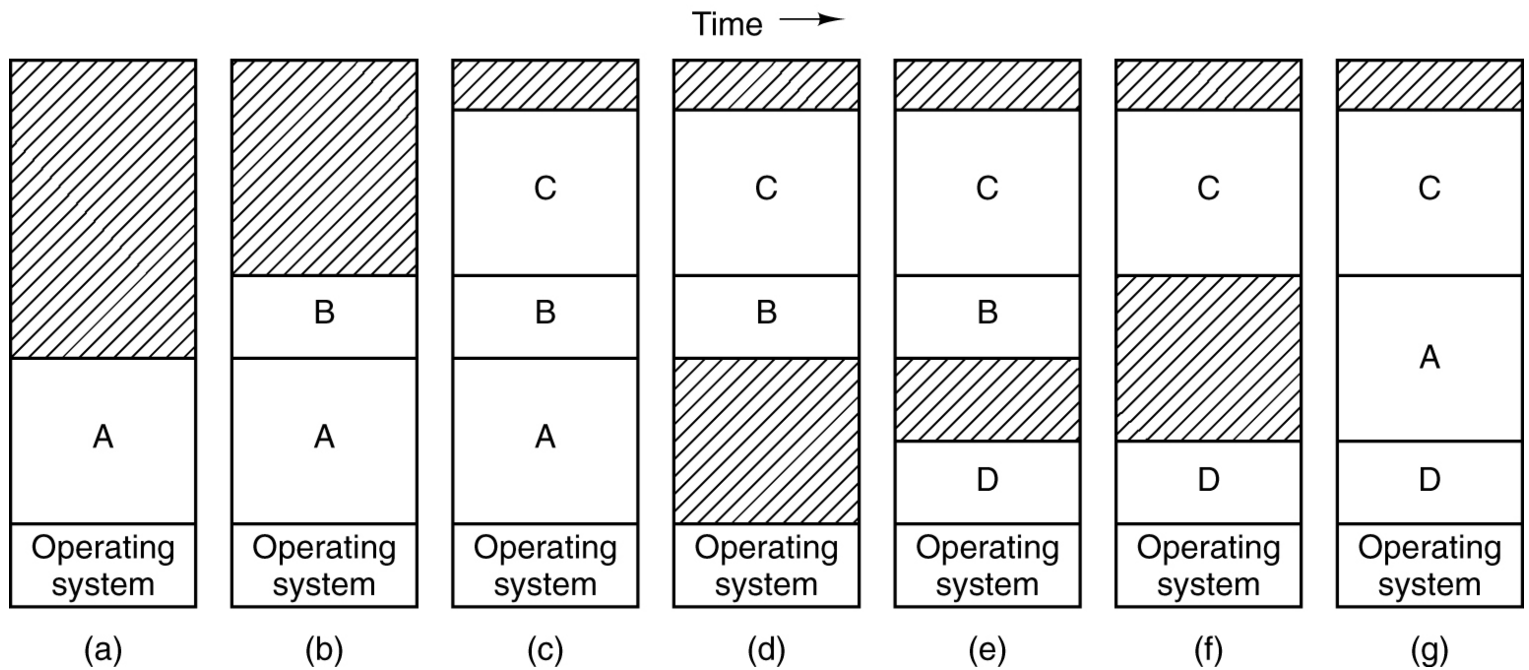
Some Important Topics

- Batch
  - First come first served
  - Shortest job first
  - Shortest remaining time
- Interactive
  - Round robin
  - Priority
  - Multilevel queue
  - Lottery
  - Fair share
- Real-time
  - Hard
  - Soft

Review

- Processes
- Threads
- Synchronization
- Scheduling
- Memory/VM
- Paging
- Segmentation
- Files/Directories
- File Systems
- I/O
- Interrupts/DMA
- Disk drives

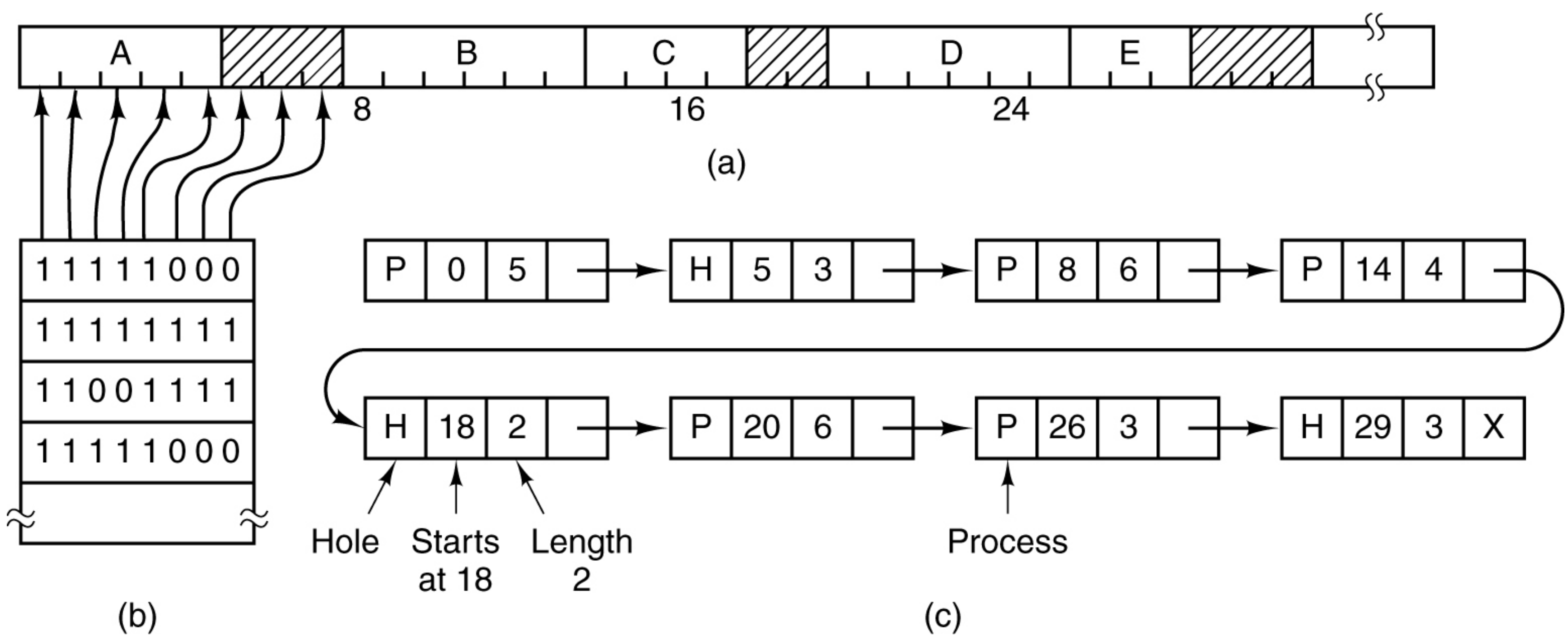
Contiguous Allocation



- Multiple-partition allocation
  - Hole: block of available memory, somewhere
  - New process placed in large-enough hole
  - OS keeps track of allocated partitions and holes
  - Fragmentation can be managed by compaction

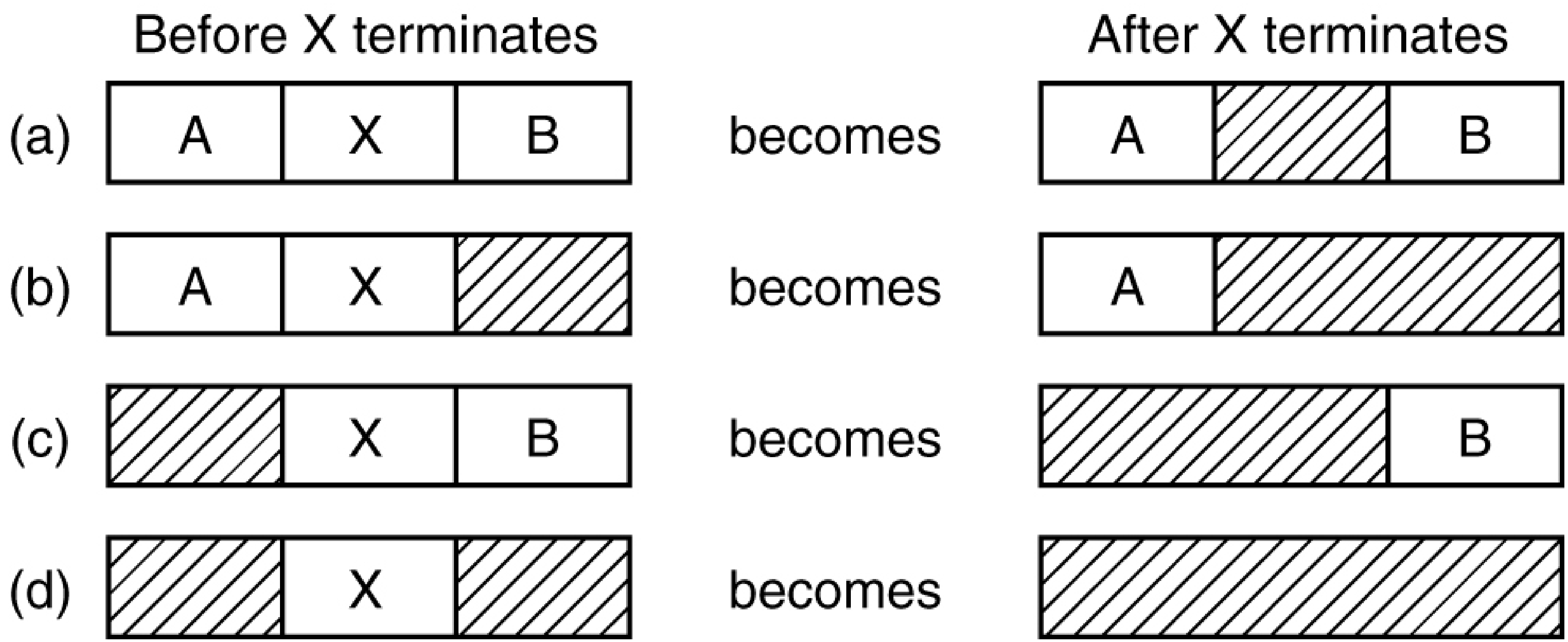
Managing Free Memory

- Bitmaps
  - Large allocation unit = small bitmap, and vice versa



Managing Free Memory

- Linked lists
  - Easy and efficient



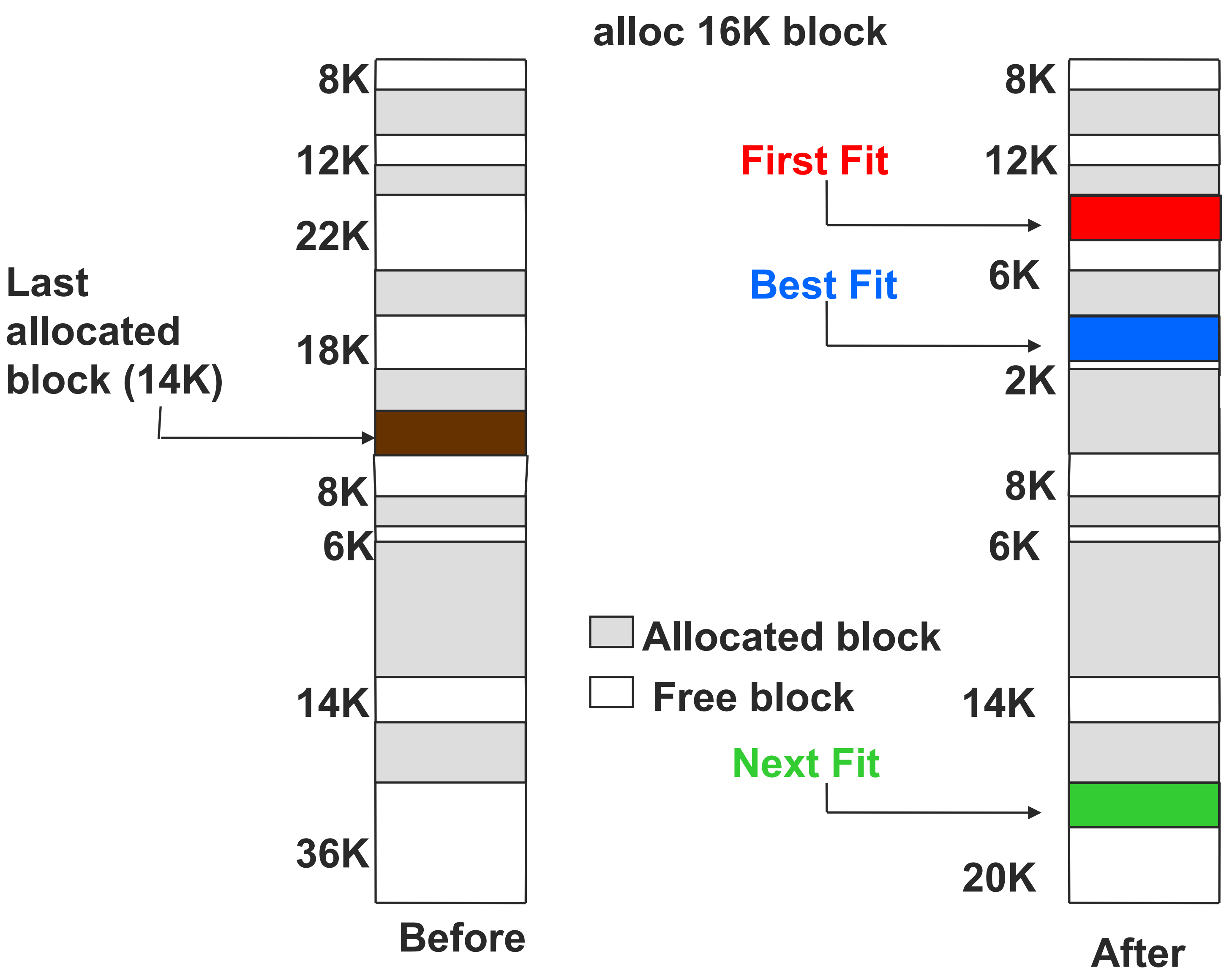
Placement Algorithms

- First-fit
  - Scan list of free memory from beginning
  - Choose first available block that is large enough
  - Fast
- Next-fit
  - Same as first fit, but next scan starts where last one made the placement
  - More often allocates a block of memory from end of memory, where largest free block is
  - Compaction required to obtain large free block at end of memory

Placement Algorithms(2)

- Best-fit
  - Searches entire list, from beginning to end
  - Chooses smallest hole that is adequate
  - Slow
  - Results in most fragmentation, tiny useless holes everywhere
- Worst-fit?
  - Take largest available hole
  - Leaves large new hole, on average

Example


alloc 16K block
8K
12K
22K
Last allocated block (14K)
18K
8K
6K
14K
36K
Before
First Fit
Best Fit
Allocated block
Free block
Next Fit
8K
12K
6K
2K
8K
6K
14K
20K
After

Fragmentation Summary

- External fragmentation
  - Total memory space exists to satisfy an allocation request, but it is not contiguous
- Internal fragmentation
  - Allocated memory is larger than requested memory
  - Size difference is internal to partition
    - This memory is wasted for the life of the process
- Compaction reduces external frag.
  - Move memory contents to create large free blocks
  - Possible only with dynamic relocation, done at execution time

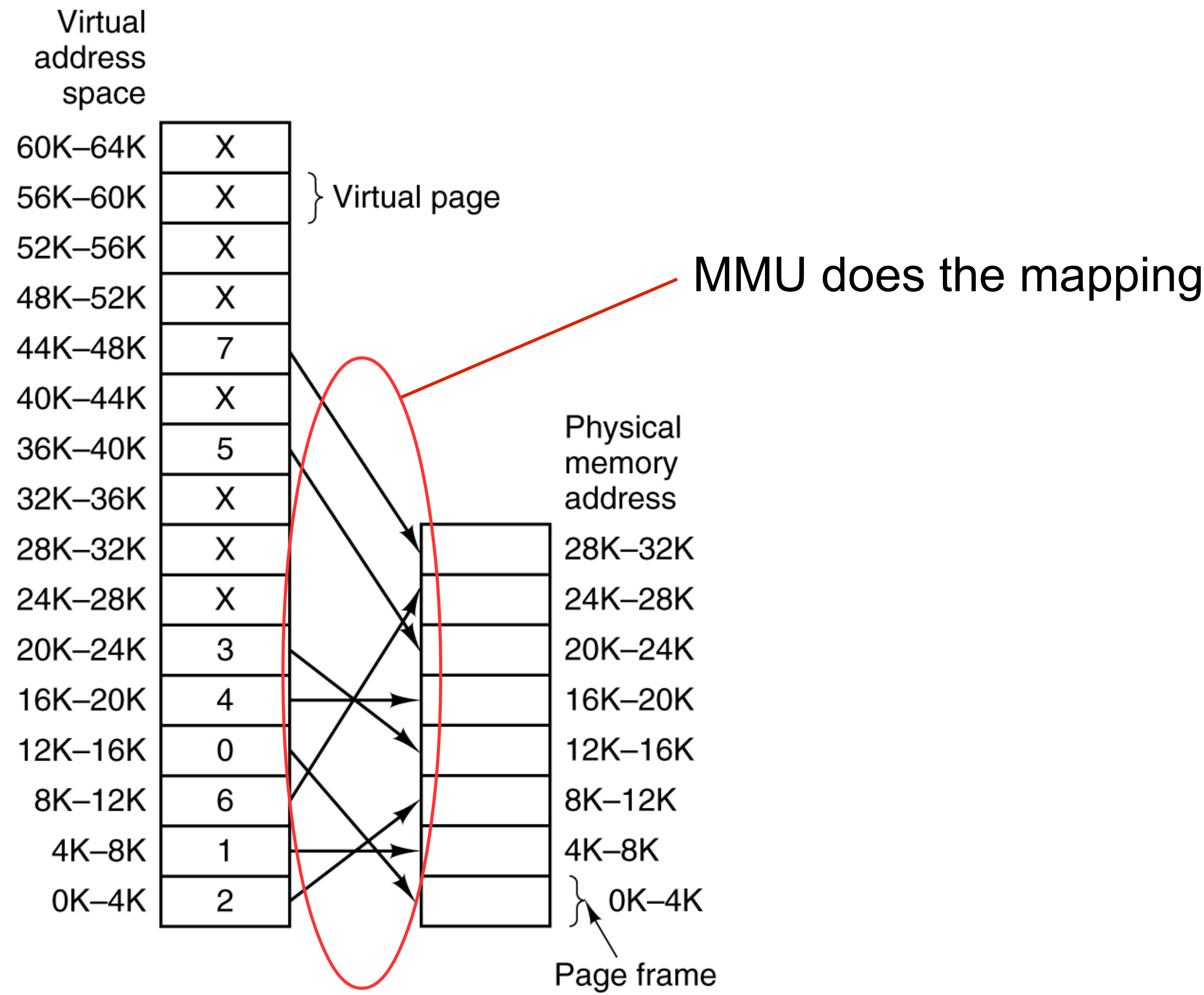
Some Important Topics

- Allocation/compaction
- Free memory management
  - Bitmaps vs linked lists
  - First fit, next fit, best fit, worst fit, quick fit

Review

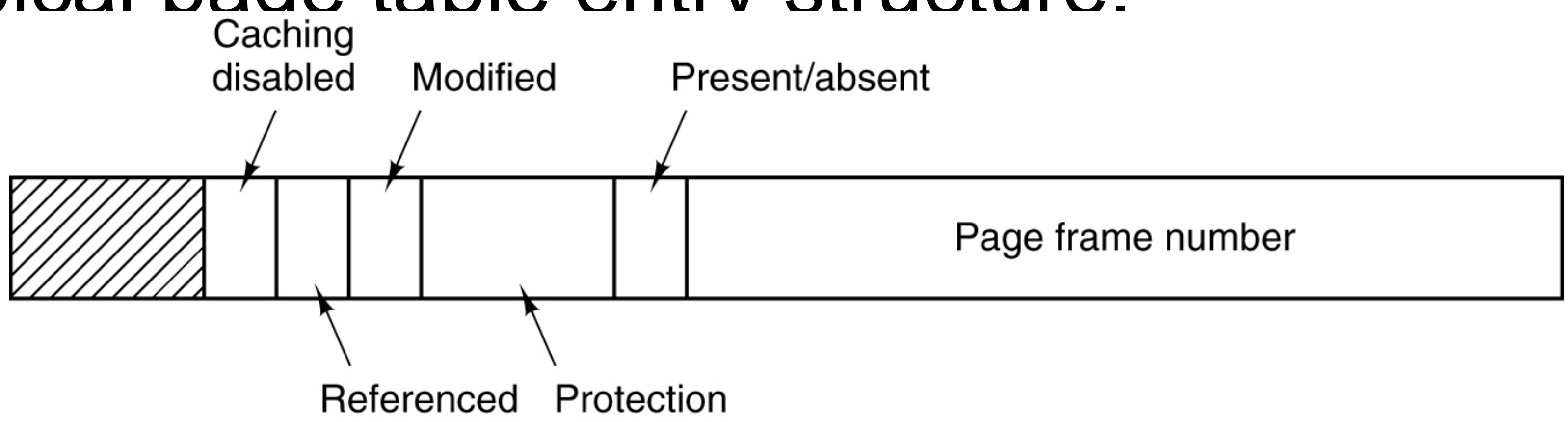
- Processes
- Threads
- Synchronization
- Scheduling
- Memory/VM
- Paging
- Segmentation
- Files/Directories
- File Systems
- I/O
- Interrupts/DMA
- Disk drives

Virtual → Memory Example

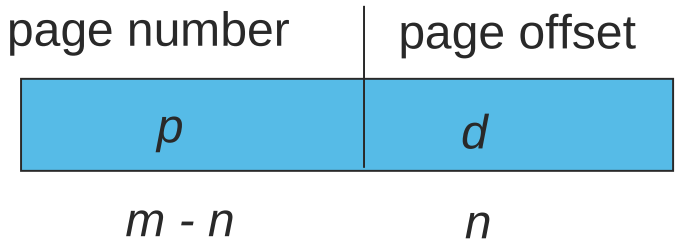


Address Translation

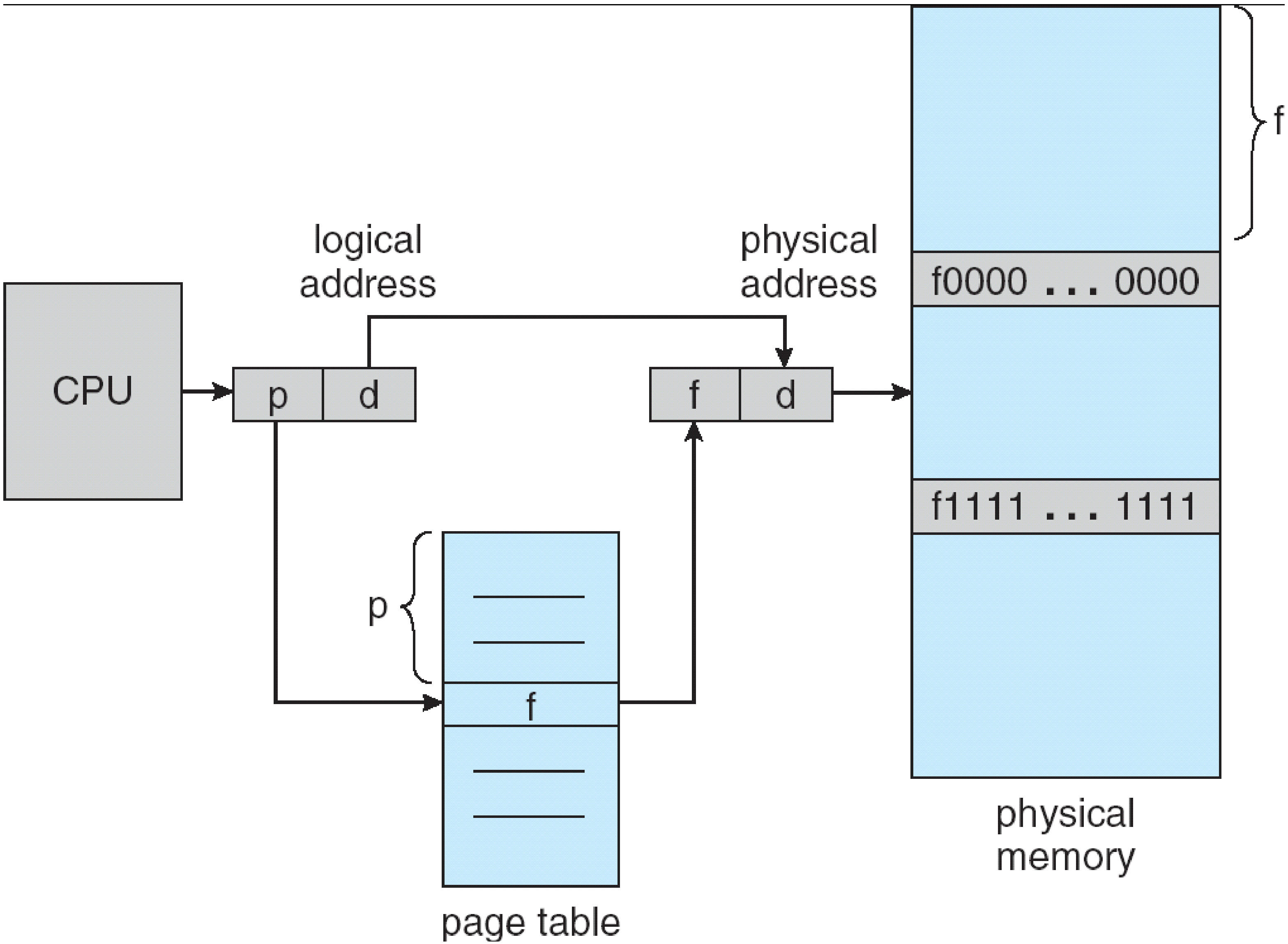
- Virtual address split into:
  - Page number
  - Offset within page
- Typical page table entry structure:



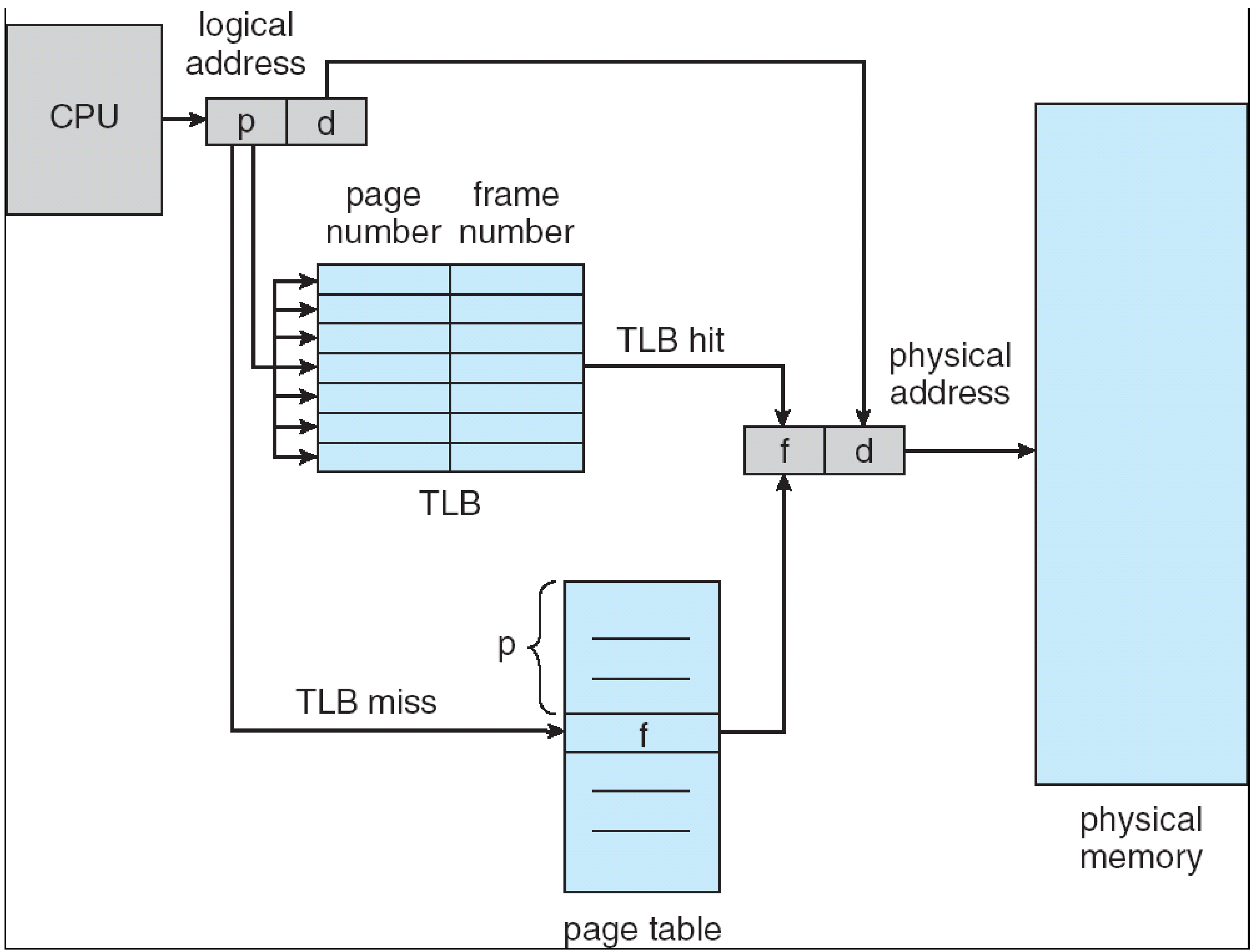
- Simplified:



Address Translation Example



Paging with TLB example



Second Issue: Page Table Storage

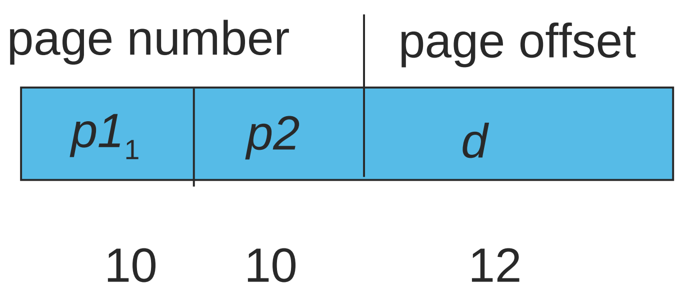
- Page tables are created per process
- If we have a 32 bit system... not a big deal
  - 2^32 addressable bytes (4GB)
  - 4KB page=2^12 bits per page, so need 12 bits for page offset
  - Remaining 20 bits used to index into page table
  - So, 4 bytes total per page table entry
  - 2^20 entries in page table per process
  - 4MB page table per process

What About Large Address Spaces?

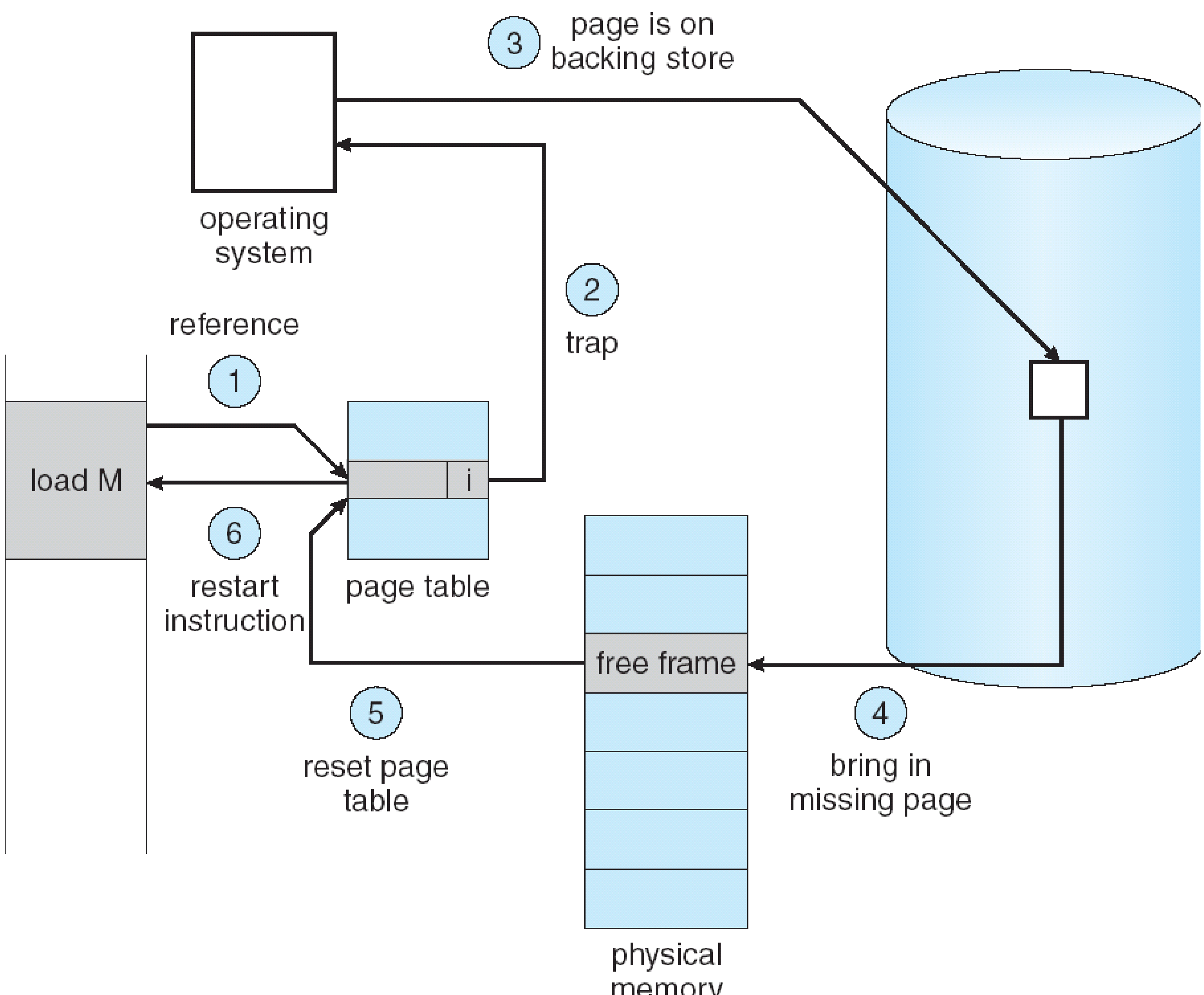
- Why are large address spaces a problem?
  - 64 bit computer = 2^64 addressable bytes
  - Assuming 4KB pages
    - 2^64/2^12 = 2^52 page table entries per process
  - 2^52 * 4 bytes... that's a lot of bytes per process
- Fortunately most entries are unused
- Solutions?
  - Hierarchical paging
  - Hashed page tables
  - Inverted page tables

Two-Level Paging Example

- On a 32-bit system, divide logical address into:
  - d: page offset, 12 bits
  - p1: 10 bit offset into outer table
  - p2: 10 bit offset into inner table



Illustrated Page Fault



Measuring Performance

- Page fault rate $0 < p < 1.0$
  - 0 → No page faults
  - 1 → Every reference is a fault
- Effective Access Time (EAT):

  EAT= (1-p) * (memory access time)

  + p * (page fault overhead)

  - Page fault overhead = page swap in

    + page swap out

    + restart instruction

Example

- Memory access time = 100 nanoseconds
- Page fault overhead = 25 milliseconds
- Page fault rate = 1/1000

EAT=(1-p)*100 + p*25,000,000

=100 + 24,999,900 * p

=100 + 24,999,900 * 1/1000 = 25 microsec.

Page Replacement

- What if we have a page fault but no free frames
  - Terminate the user process (not desirable)
  - Swap out process (reduces degree of multiprogramming, so likewise not desirable)
  - Replace some other page with the needed one
- Page replacement
  - If there exists a free frame, use it
  - Otherwise:
    - Find victim frame
    - Write page to disk, update tables, read in new page
    - Restart process

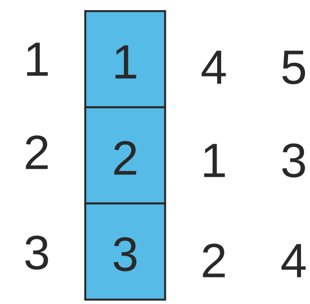
Not Recently Used (NRU)

- Use R and M bits (Referenced and Modified)
- At start of process execution
  – Set R and M bits to 0
- Periodically (at clock interrupt)
  – Set R bit to 0
- At page fault divide pages by category
  – Class 0: R=0, M=0 Class 2: R=1, M=0
  – Class 1: R=0, M=1 Class 3: R=1, M=1
- Remove random page from lowest numbered non-empty class

FIFO: First-In-First-Out

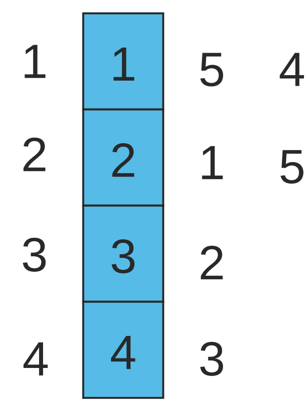
- 1,2,3,4,1,2,5,1,2,3,4,5

3 frames per process



9 page faults

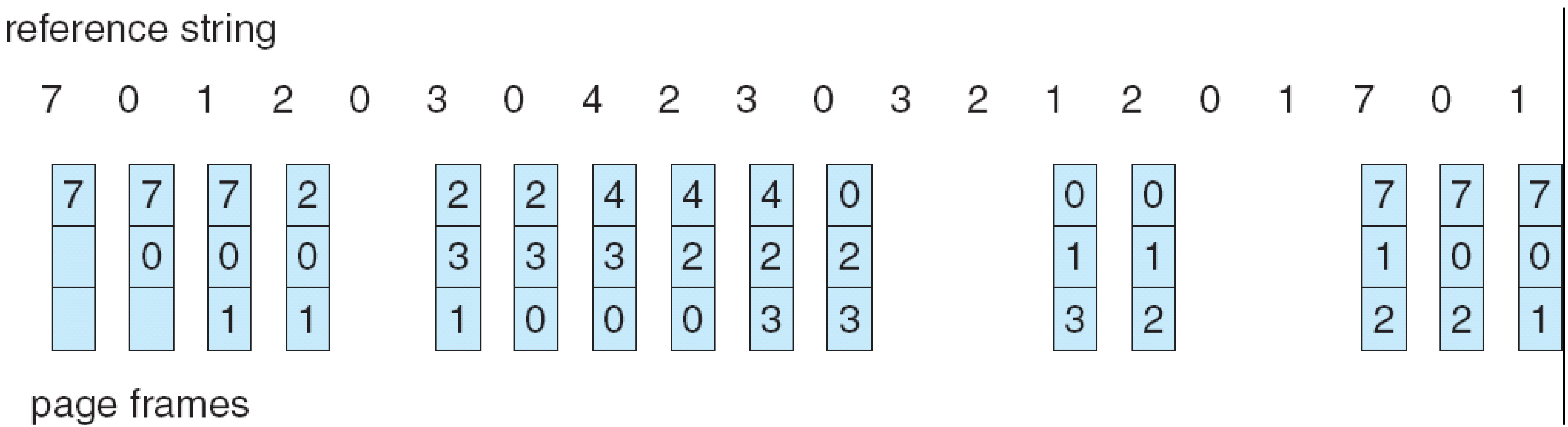
4 frames per process



10 page faults

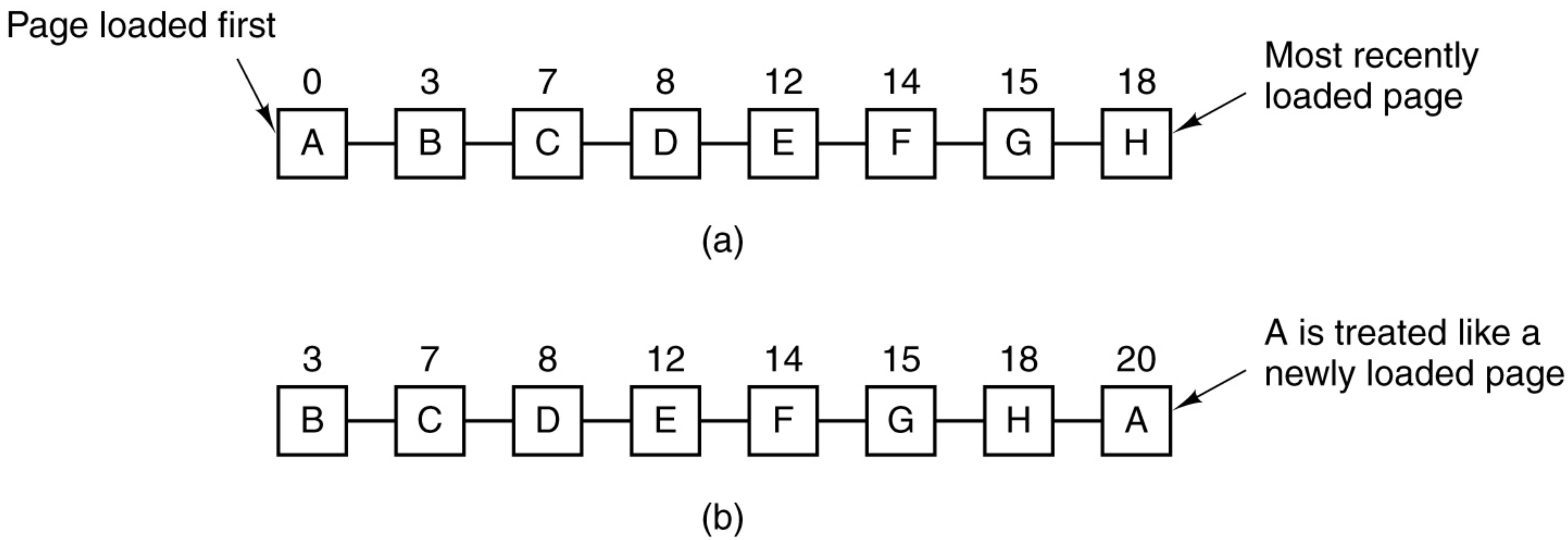
- This is known as Belady's Anomaly

FIFO: First-In-First-Out



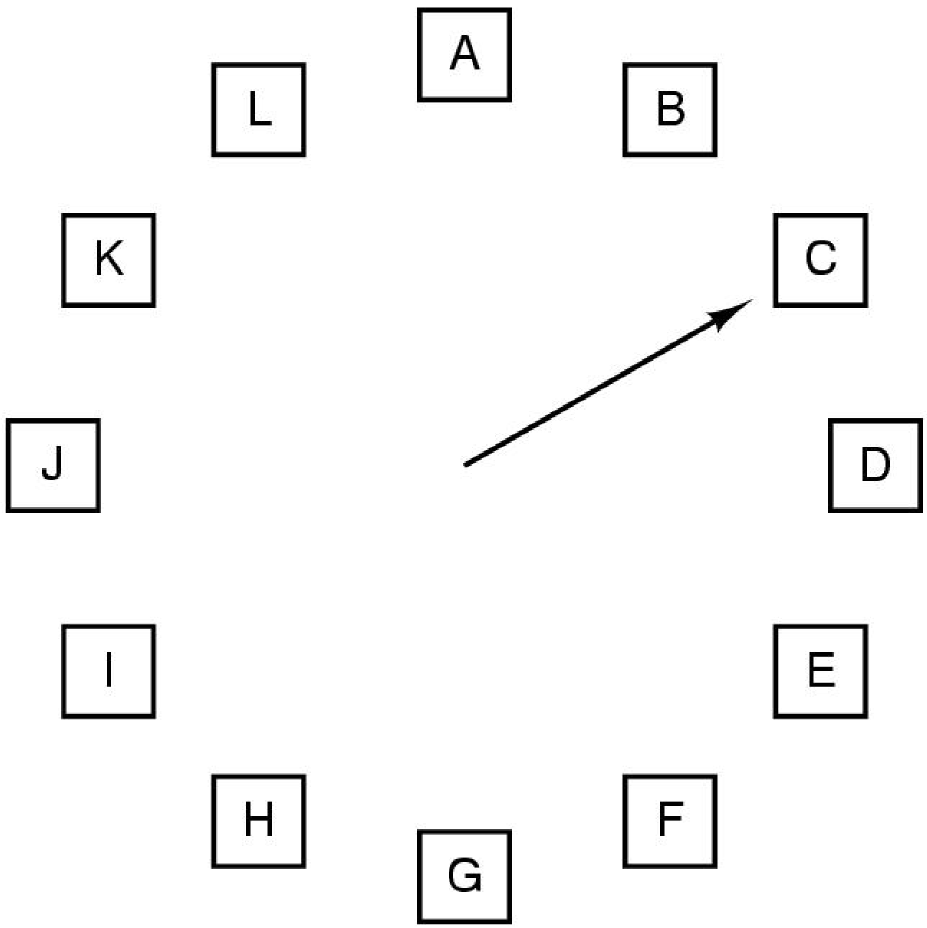
- Problem? Page in memory the longest may be frequently used
- Solution? Use a referenced bit.
  - If a page is marked as reference, unmark and place at end of FIFO list.
  - This is known as Second Chance

Second Chance



- So, if a page is referenced enough, it is never replaced
- Can degenerate to FIFO
- Inefficient due to moving pages around in the list (solved by next algorithm)

Clock Replacement Algorithm

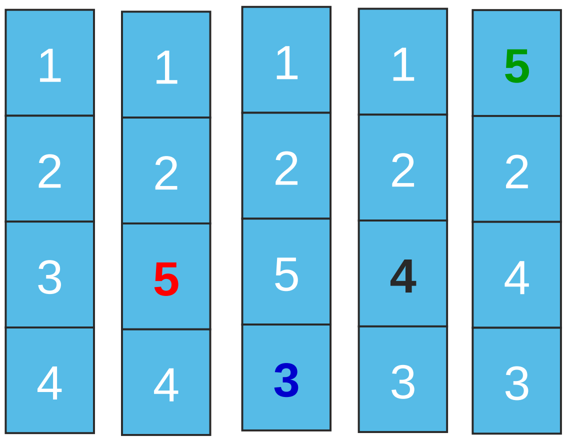


When a page fault occurs, the page the hand is pointing to is inspected. The action taken depends on the R bit:

- R = 0: Evict the page
- R = 1: Clear R and advance hand

Least Recently Used (LRU)

- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5



- Counter implementation
  - Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter
  - When a page needs to be changed, look at the counters to determine which are the oldest

Counting Algorithms

- Keep count of references made to each page
- LFU: Least Frequently Used
  - Replace page with smallest count
  - May leave pages that are initially hot but never used again
- MFU: Most Frequently Used (bad idea)
  - Replace page with highest count (pages with smallest count may have just been brought in)
  - Replaces popular pages
- Counting algorithms perform poorly in general

Thrashing

Thrashing ⇨ process is busy swapping pages in and out

- Suppose there are many users, processes are making frequent references to 50 pages, memory has 49
- Each time one page is brought in, another page, whose contents will soon be referenced, is thrown out
- What is the average memory access time?
- The system is spending most of its time paging!
- The progress of programs makes it look like memory access is as slow as disk, rather than disk being as fast as memory

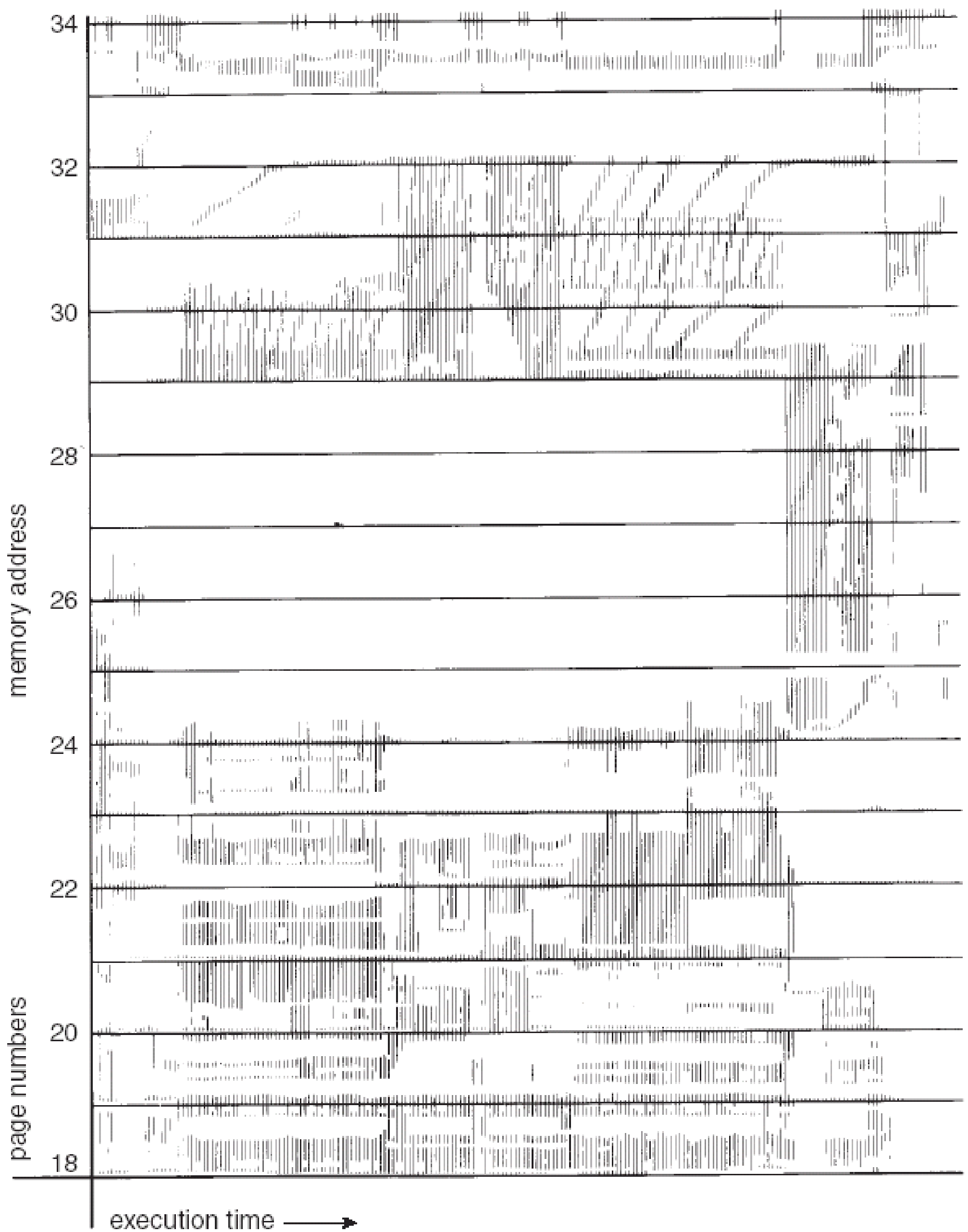
Example: Looping Reference

- Application repeatedly scanning 5 pages
  - 4 Frames, 5 pages
  - Reference pattern: 1, 2, 3, 4, 5

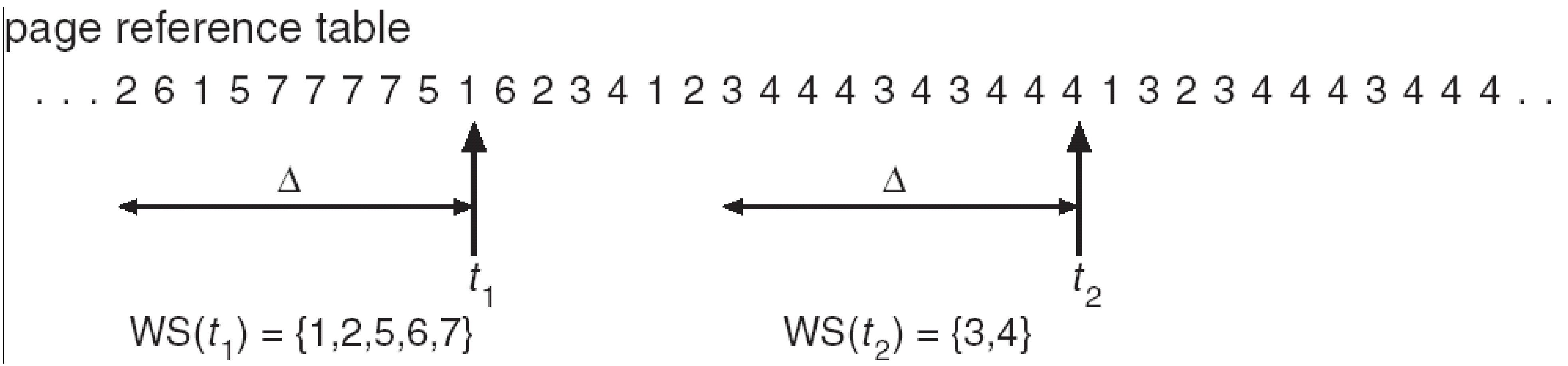
- What happens if LRU is used
  - Thrashing

- Btw, what is the optimal strategy here?
  - MRU ⇨replace the most recent one

Locality of Memory References



Working-set model



Some Important Topics

- Fragmentation types
- Page size effects
- Page faults
- Page tables
  - TLB
- Replacement algorithms
- Segmentation

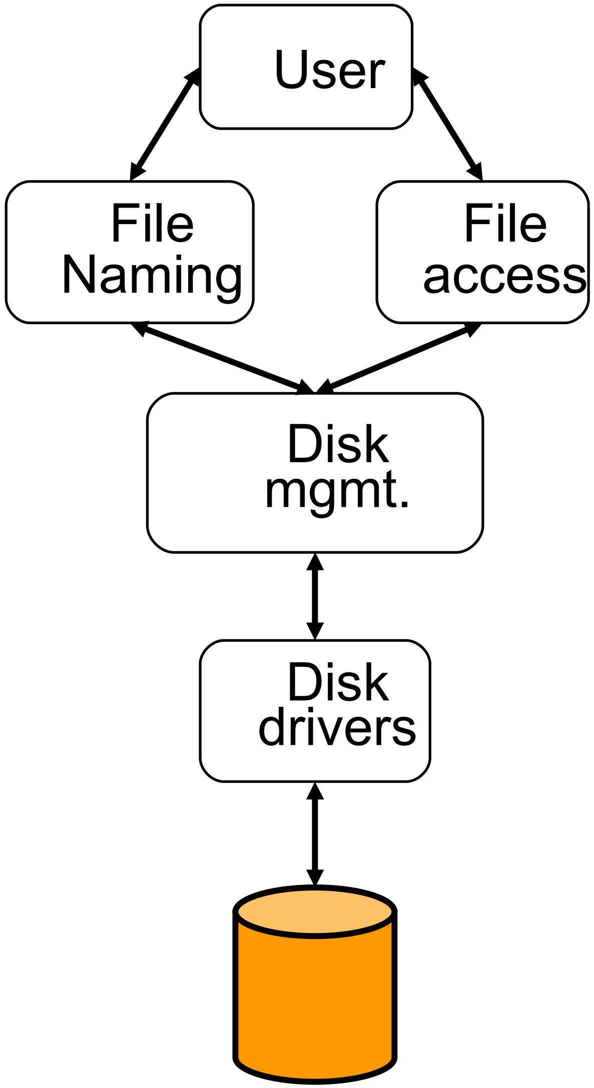
Review

- Processes
- Threads
- Synchronization
- Scheduling
- Memory/VM
- Paging
- Segmentation
- Files/Directories
- File Systems
- I/O
- Interrupts/DMA
- Disk drives

File System Components

- Disk management
  - Arrange collection of disk blocks into files
- Naming
  - User gives file name, not track or sector number, to locate data

- Security
  - Keep information secure
- Reliability/durability
  - When system crashes, lose stuff in memory, but want files to be durable



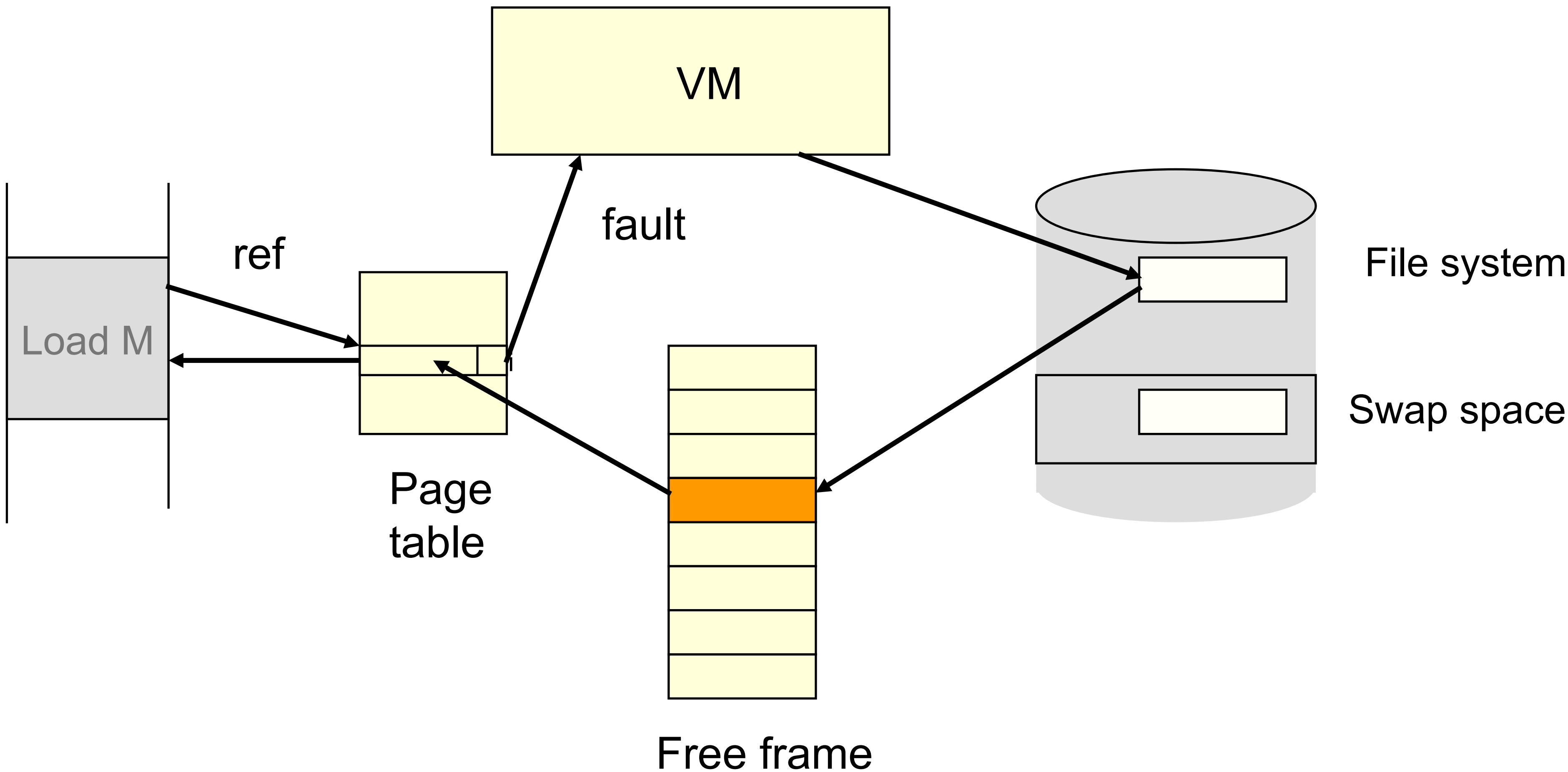
What is a File?

- File: a named collection of bytes stored on disk
  - Contiguous logical address space

- From OS's standpoint
  - A file consists of a bunch of blocks stored on the device

- From programmer's view
  - A collection of records
  - But this does not matter to the file system, always
    - Pack bytes into disk blocks on writing
    - Unpack them again on reading

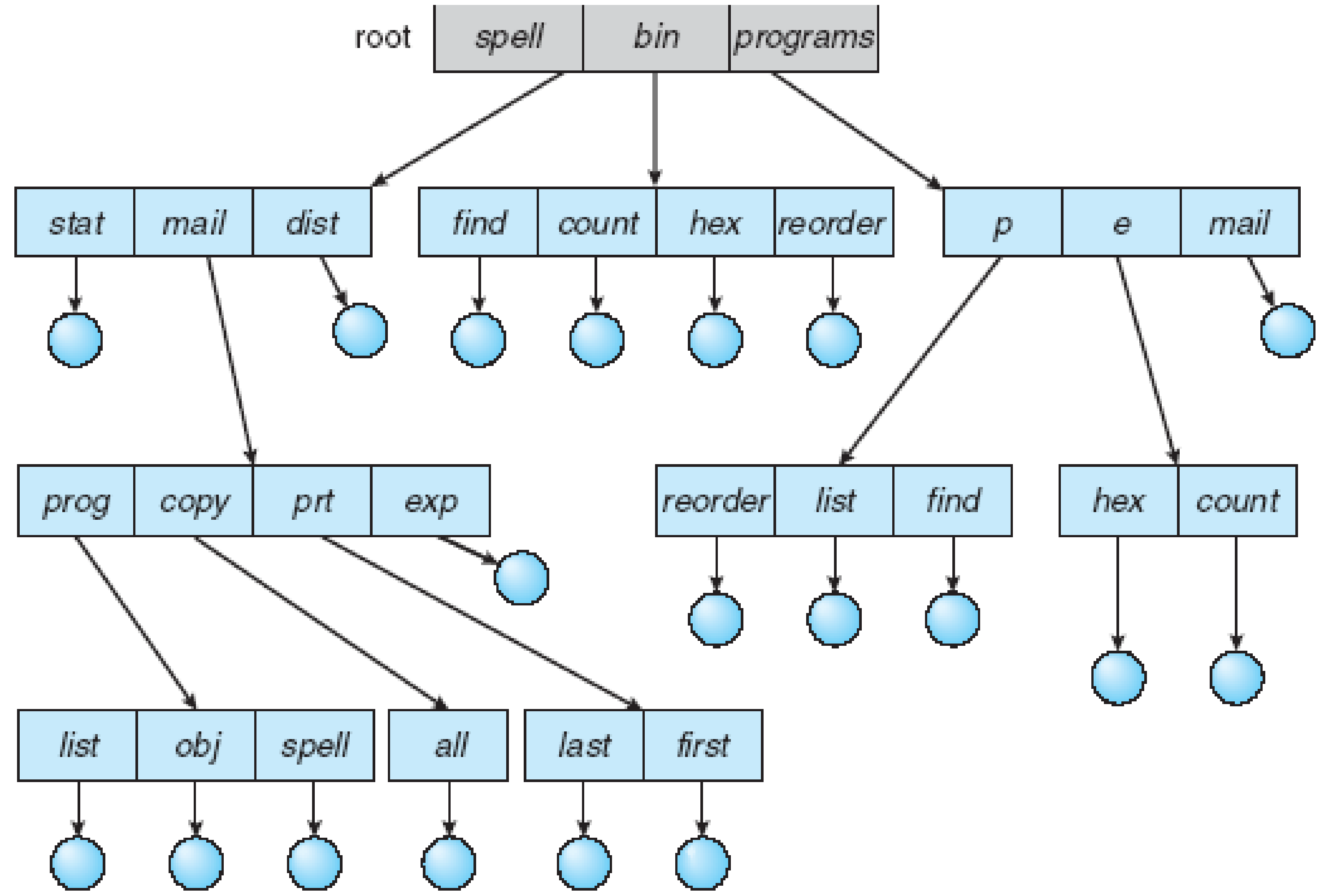
Memory-Mapped Files



So What’s a Directory?

- In Linux, just a specially-formatted file
  - Yes, you can read it just like a file
  - It’s a directory because “we” treat it that way
- Directory contains names of files
  - <name, fd index> pairs in no particular order
  - The file pointed to by index may be another dir
  - A special dir, called root, has no name

Tree-Structured Directories

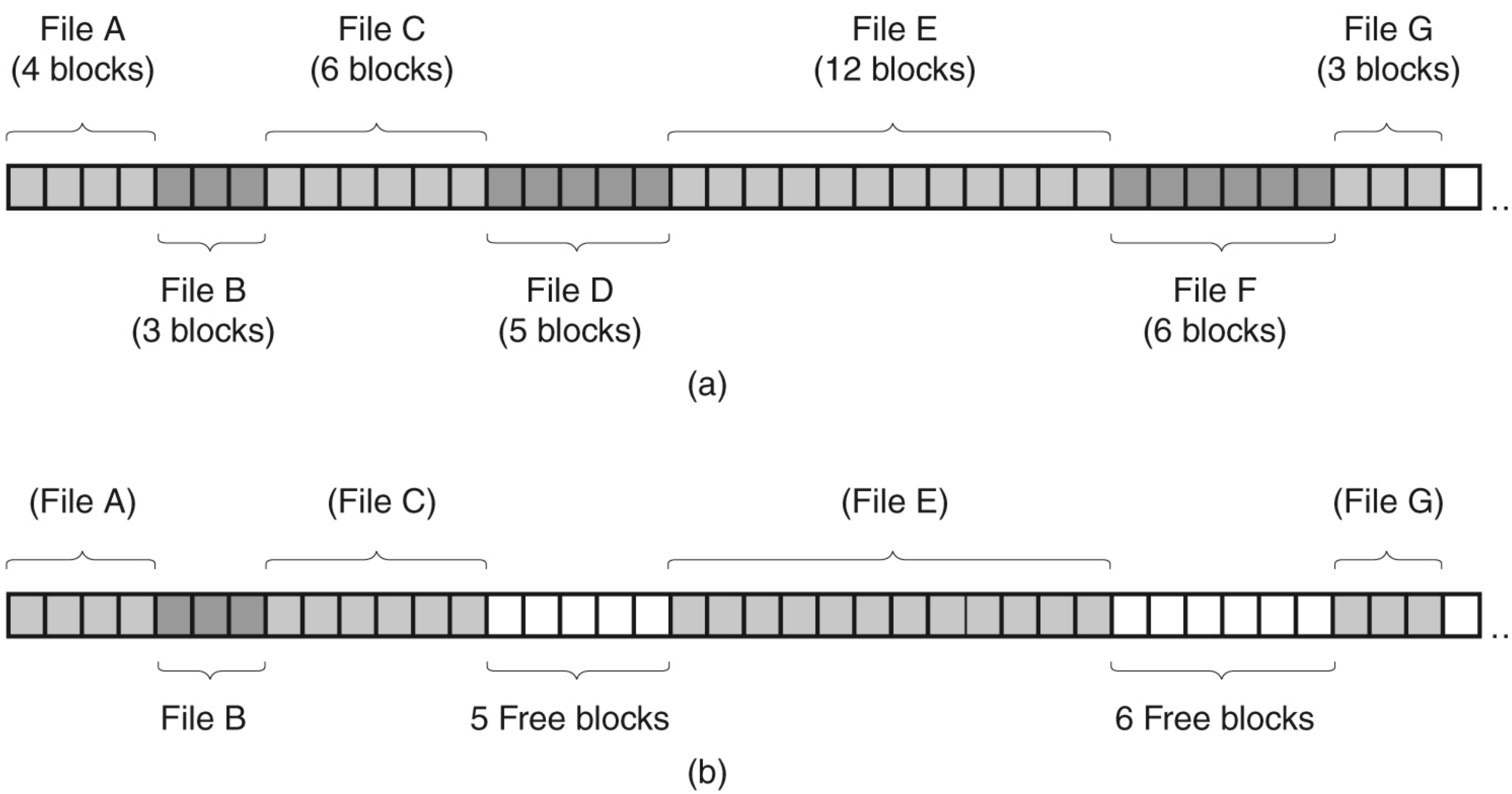


How Does This Work in Reality?

- Finding /dir1/dr2/dir3/readme.txt

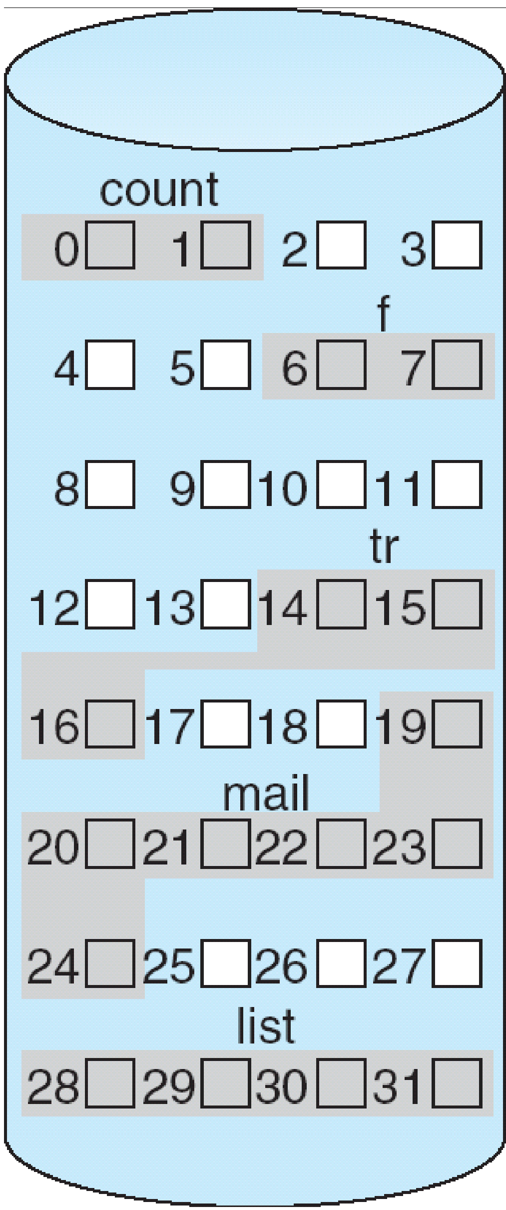
- Fetch root inode
  - Start loading root directory data blocks
  - Walk directory data until you find dir1
  - Get inode # for dir1 from directory file
- Fetch dir1's inode
  - Start loading dir1's directory data blocks
  - Walk directory data until you find dir2
  - Get inode # for dir2 from directory file
- …
- Repeat process until you have inode # for file

Files (Contiguous Allocation)



- Files begin at start of new block
- Simple: keep starting block and # of blocks
- Read performance excellent (few seeks)
- Fragmentation, have to know length of files

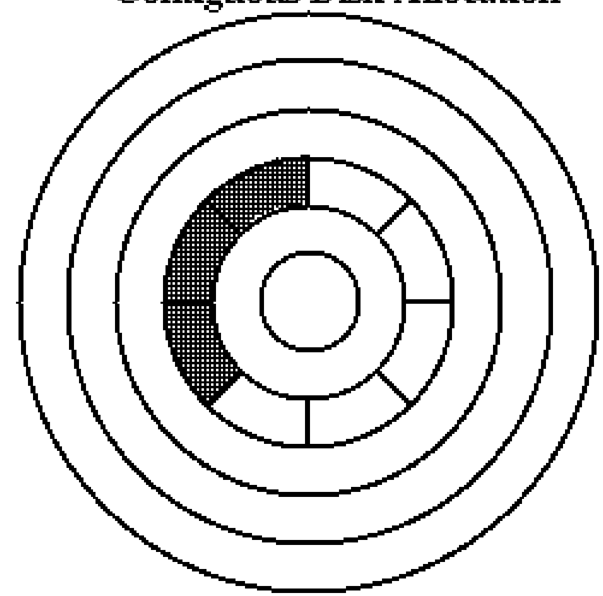
Contiguous Allocation of Disk Space



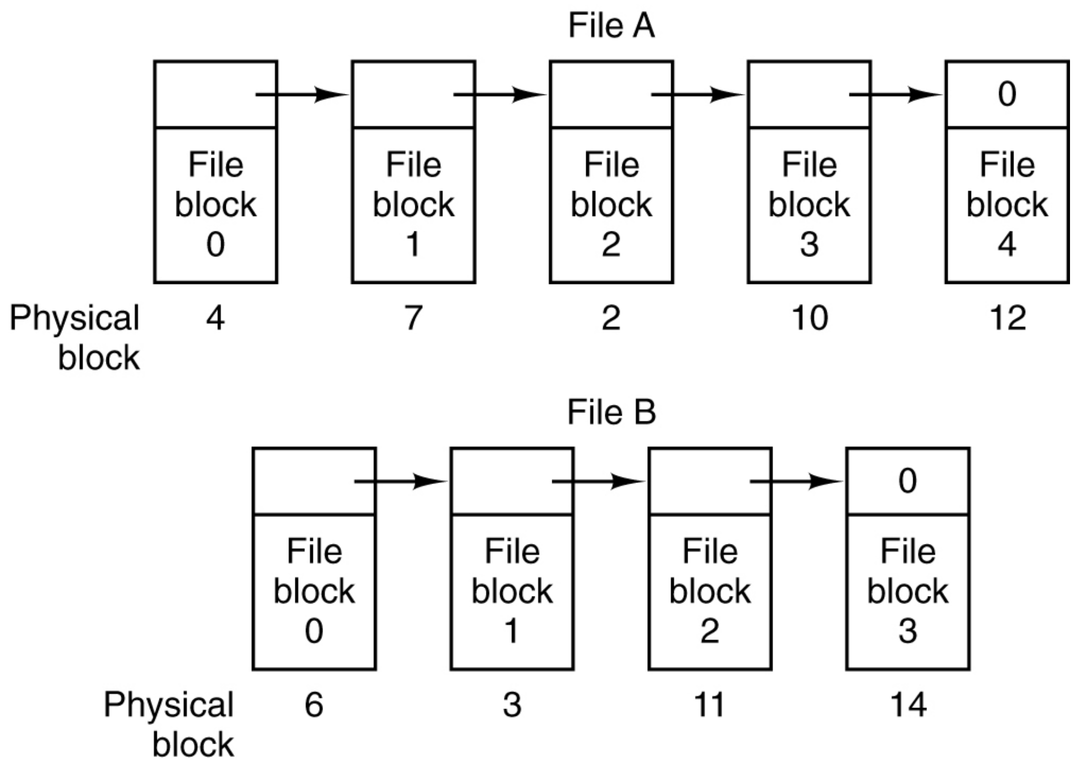
directory

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

Contiguous Disk Allocation

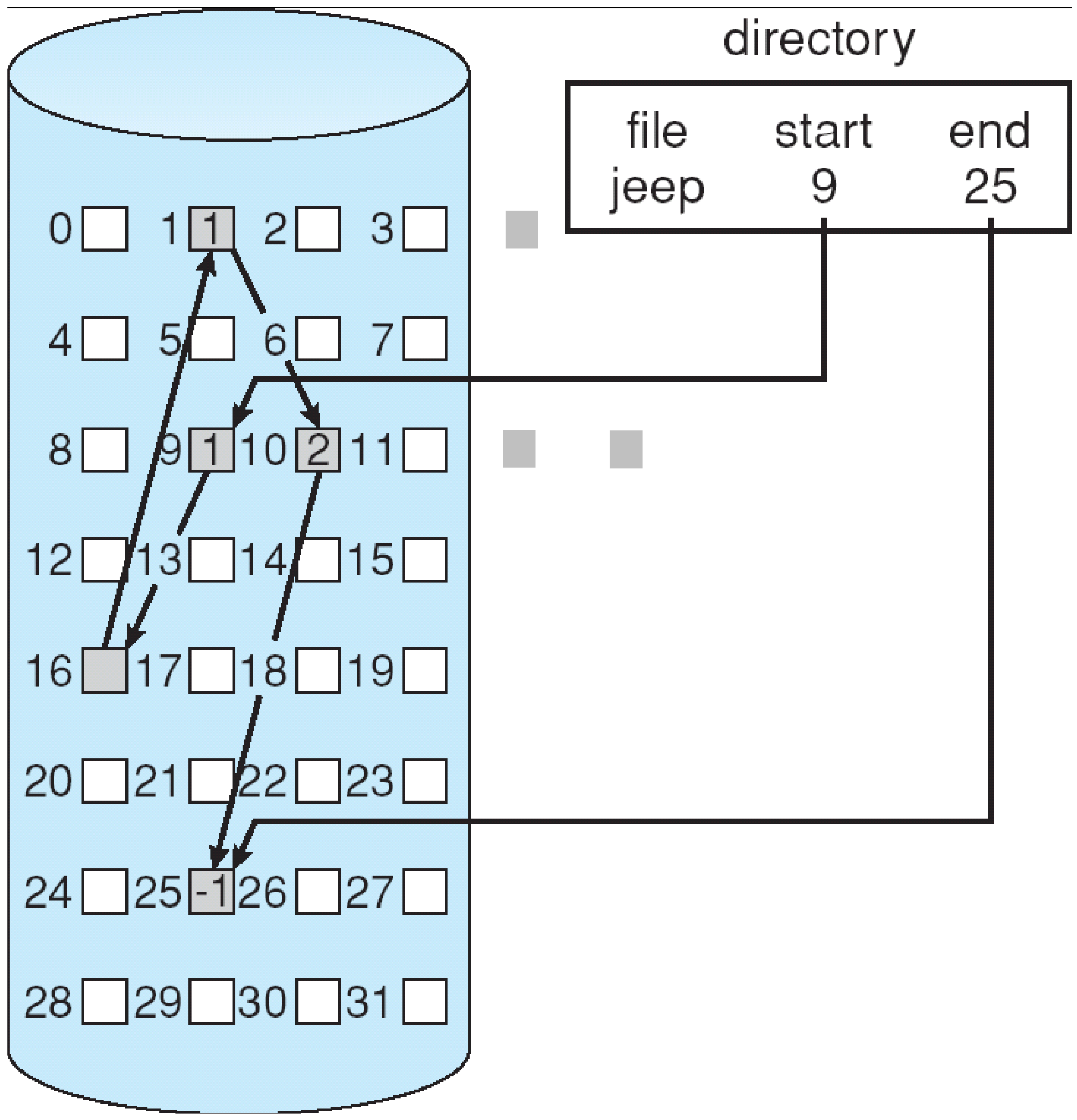


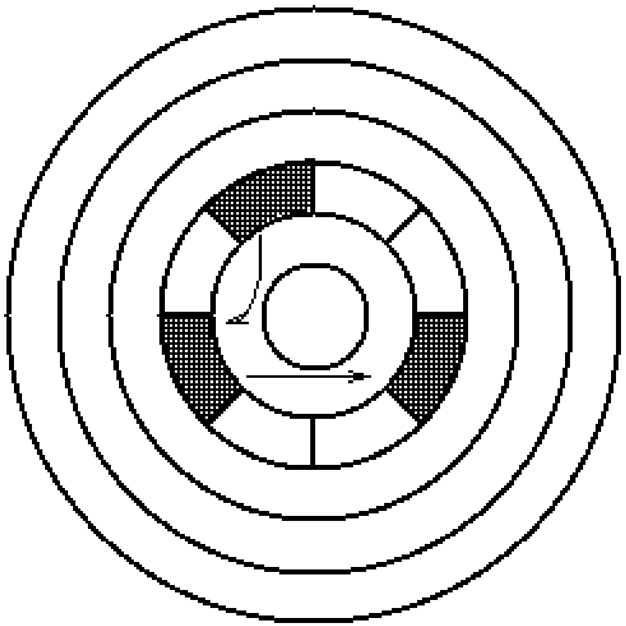
Files (Linked-List Allocation)



- First word in block points to next block
  - No external fragmentation
  - Random access is slow, lots of seeks
  - Reading block size of data requires 2 blocks

Linked Allocation of Disk Space





Performance

- How do users access files?
  - Sequential: bytes read/written in order
  - Random: read/write blocks in middle of file
  - May access whole or partial file
- How are files used?
  - Most are small
  - Large files take up most of disk space
  - Large files account for most bytes transferred
- Everything needs to be efficient

Performance

- Hard drive
  - Disk allocation and directory algorithms
  - Types of data kept in file's directory entry
  - Data/metadata layout

- Operating system
  - Disk cache – separate section of main memory for frequently used blocks
  - Delayed-writes – aggregation / higher priority to reads
  - Read-ahead – Prefetching of sequential blocks
  - Section of memory as virtual disk (RAM disk)

File Caching

- Locality of reference in file accesses
  - Yet another application of the principle of locality
  - What were the earlier instances?
- Keep a number of disk blocks in “the much faster” memory
  - when accessing disk, check the cache first!

Some Important Topics

- File vs directory
- File: types, operations, identification
- Directories: hierarchies, paths
- Hard vs soft link
- Allocation choices
- Inodes
- Disk space management, block size effects, backups, consistency, etc.

Review

- Processes
- Threads
- Synchronization
- Scheduling
- Memory/VM
- Paging
- Segmentation
- Files/Directories
- File Systems
- I/O
- Interrupts/DMA
- Disk drives

I/O Devices

- There's more to a computer than CPU and memory
- I/O Devices:
  - Store information
  - Communicate to the outside world
- Role of the OS → Control I/O devices
  - Range of types and speeds
  - OS concern is with the interface between the hardware and the user

Types of Devices (Block)

- Stores information in blocks
  - Blocks addressed/accessed individually
  - 512 – 32k (common sizes)
  - Block is a unit of transfer
- HDD, CD-ROM, flash drives
- Typically have a seek operation
  - Random access

Types of Devices (Character)

- Delivers or accepts a stream of characters
  - No block structure
  - Not addressable
  - No seek
- Printer, network interface, mouse...

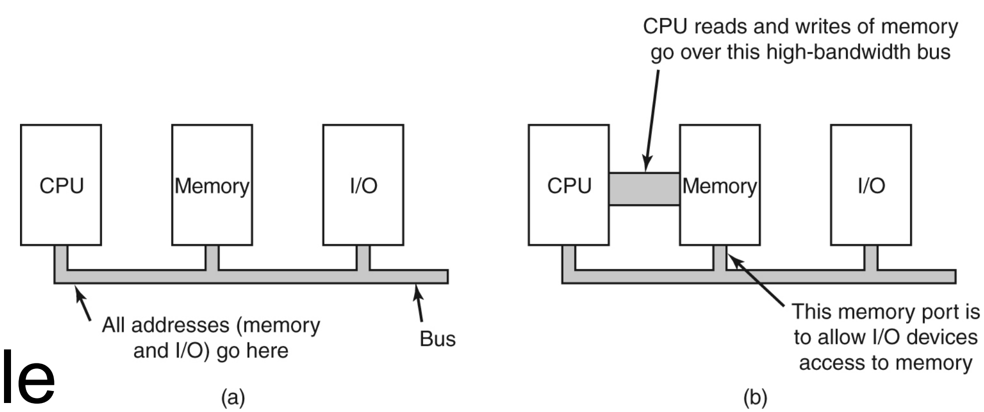
- Not all devices are block or character
  - Timers, clocks

Memory-Mapped I/O

- Each device gets a portion of real address space
  - Access limited to kernel
  - Reads/writes to this memory interpreted as commands by I/O device
- Device control addressable by high-level language (C/C++)
- No special protection needed
  - Don't map device memory to user processes
- Existing instructions repurposed for I/O control

Memory-Mapped I/O (2)

- Potential problem with page caching
  - e.g. while (status!=0)
    - What if status is cached?
  - Have to have ability to disable caching per page
- All controllers must examine all memory accesses (problem?)
  - Intel uses PCI bridge to filter addresses that fall w/in non-addressable range



Direct Memory Access (DMA)

- External to the CPU
- DMA controller is a bus master
- Advantages
  - Bypasses CPU, transferring data between memory and device
  - Programs not flooded by interrupts while processing data
- Disadvantages
  - May not be as fast
  - Overhead: set up DMA engine

Is DMA Always Better?

- Main CPU is much faster than DMA controller... so, no
  - If CPU has nothing to do, waiting for DMA is wasteful
- Excluding DMA controllers reduces HW cost

Implementing I/O

- Synchronous
  - Programmed I/O
    - Polling
    - Busy waiting
- Asynchronous
  - Interrupt-driven I/O
  - I/O via DMA

Some Important Topics

- Block vs character
- Memory-mapped I/O
- DMA
- Interrupts

Review

- Processes
- Threads
- Synchronization
- Scheduling
- Memory/VM
- Paging
- Segmentation
- Files/Directories
- File Systems
- I/O
- Interrupts/DMA
- Disk drives

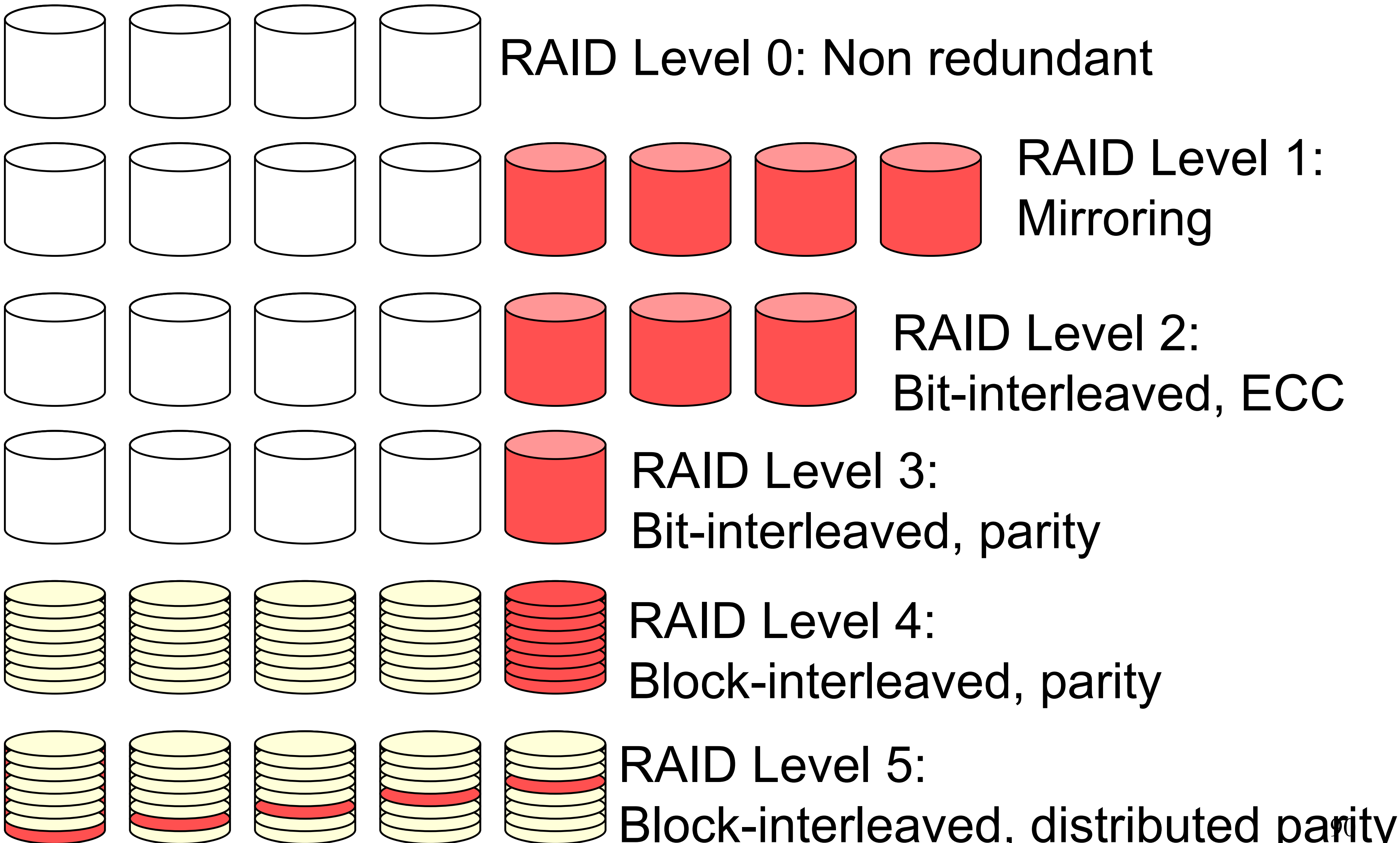
Disk Performance

- Seek
  - Position heads over cylinder, typically 5.3 - 8 ms
- Rotational delay
  - Wait for a sector to rotate underneath the heads
  - Typically 8.3 - 6.0 ms (7,200 – 10,000RPM) or ½ rotation takes 4.15-3ms
- Transfer rate
  - Average transfer bandwidth (15-37 Mbytes/sec)

Disk Performance

- Performance of transfer 512 bytes (1 sector)
  - Seek (5.3 ms) + half rotational delay (3ms) + transfer (0.02 ms)
  - Total time is 8.32ms or 60 Kbytes/sec!
- Oh man, disks are slow
  - But wait! Disk transfer rates are tens of MBytes/sec

Synopsis of RAID Levels



Some Important Topics

- Block addressing
- Raid

Programmable Clock

- Assume 500 MHz crystal
  - Counter is pulsed every 2 nsec
- Assume unsigned 32-bit register
  - Interrupts can occur at intervals
    - 2 nsec to 8.6 sec
- Battery powered backup clock keeps current time between powered down periods

Uses

- Prevent processes from running too long
  - Initialize counter to the value of a process quantum
- Account for CPU usage
  - Start timer when process starts
  - Check timer when process is stopped
- Handle alarm system calls
  - Processes may require timed warnings
    - e.g. retransmission of packets

Uses

- Provide watchdog timers
  - System timer that triggers a reset or corrective action
  - Regular heartbeat signal expected to reset watchdog, else it triggers a processor reset or non-maskable interrupt
    - Aka: service pulse, kicking the dog, feeding the watchdog
- Profiling, monitoring, and statistic gathering
  - Use system clock to keep track of execution time of various program components (see gprof and bprof)

Some Important Topics

- Programming the clock
- Clock interrupts
  - Scheduler quanta