22 I/O Management (2)

I/O Design, I/O Subsystem, I/O-Handler

Device Driver, Buffering, Disks, RAID

January 28 2009

WT 2008/09

Roadmap

- Motivation
- Repetition: I/O-Devices
  - Device Categories
  - I/O-Functionality
  - Data Transfer
- I/O-Subsystem
  - Design Parameters
  - I/O Layering
  - I/O-Buffering
- Disk I/O Management
  - Disk, CD-Rom, ...
  - Disk Layouts and Formats
  - Disk Scheduling
  - RAID
  - Disk Caching
- Clocks and Timer

Operating System Design Issues

Discuss these trade-offs very very carefully!!!

Efficiency versus **Generality**

Power versus Performance

System Design Objectives (1)

Analysis(1): *Efficiency?*

- Most I/O-devices slow compared to RAM & CPU ⇒ potential bottleneck of system
  - Use of multiprogramming allows for some tasks/processes to be waiting on I/O while another task/process is running
  - Often I/O cannot keep up with processor speed, but some devices are faster at least than RAM (Gigabit-Network)
  - Swapping and Paging may be used to improve multi-programming degree ⇒ more additional I/O operations

⇒ **Optimize I/0-Efficiency**
(especially disk & network) is
the **important issue** (← Liedtke)

System Design Objectives (2)

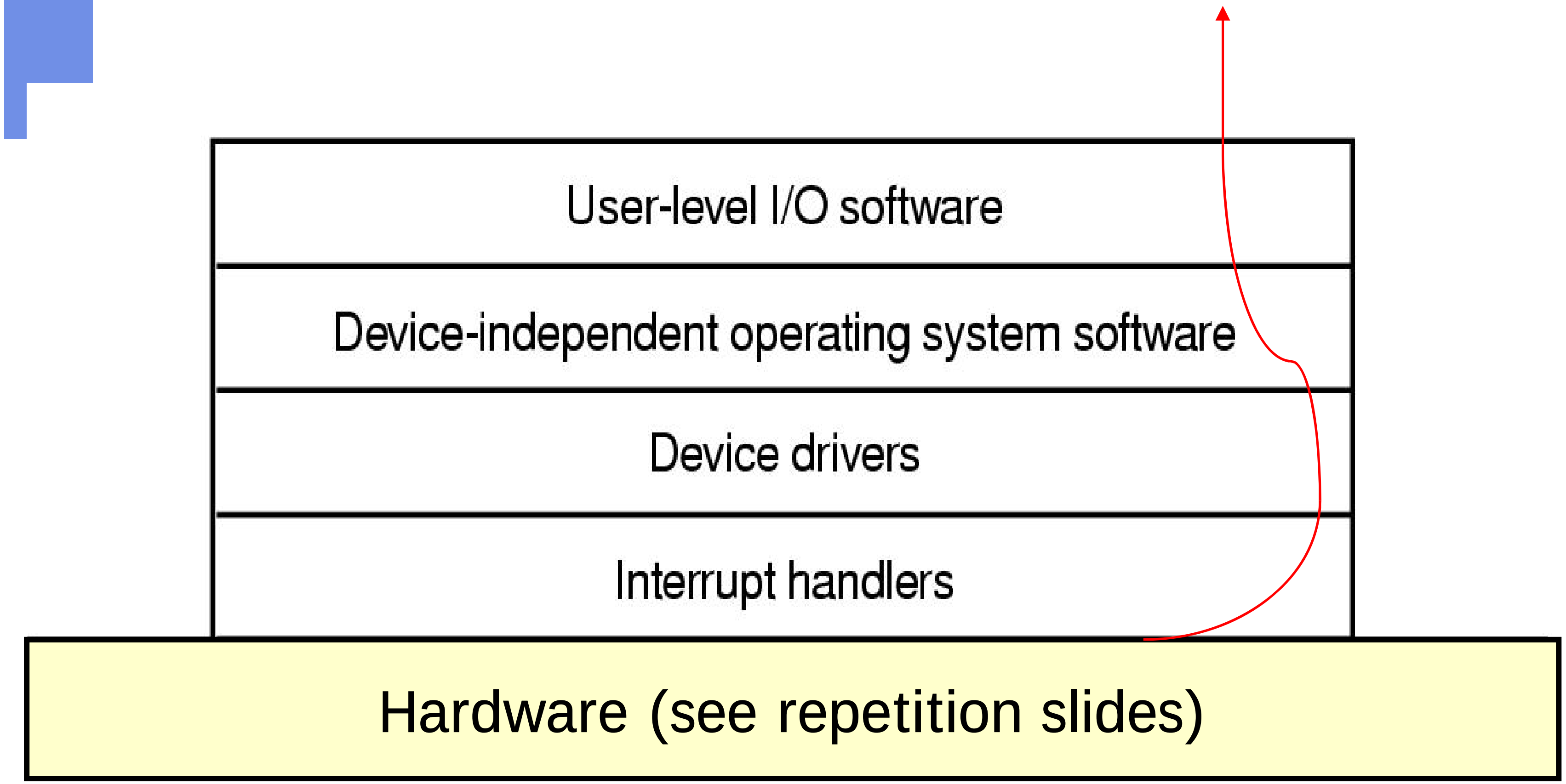
Analysis (2): *How about generality/uniformity?*

- Ideally, handle all I/O devices in the same way
  - Both in OS (kernel land) and in applications (user land)

- Problem = Diversity of devices
  - Access methods (random ~ versus stream based access)

- Hide details of device I/O in low-level routines so that tasks/processes and upper level I/O functions can see devices in general terms such as files
  - read and write or
  - open and close or
  - lock and unlock ...

Layers of I/O Software System



User-level I/O software

Device-independent operating system software

Device drivers

Interrupt handlers

Hardware (see repetition slides)

Interrupt Handler (1)

- Interrupt handlers are best hidden
  - Can be executed at almost any time
    - Raise (complex) concurrency issues in the kernel
    - Have similar problems within applications if interrupts are propagated to user-level code (via signals, upcalls)
  - Generally, a driver having started an I/O blocks, until the “completion interrupt” notifies the waiting driver
  - Interrupt handler does its work related with the I/O-device and then unblocks driver that has started the finished I/O

- The following steps must be performed in software after an interrupt has occurred, ...

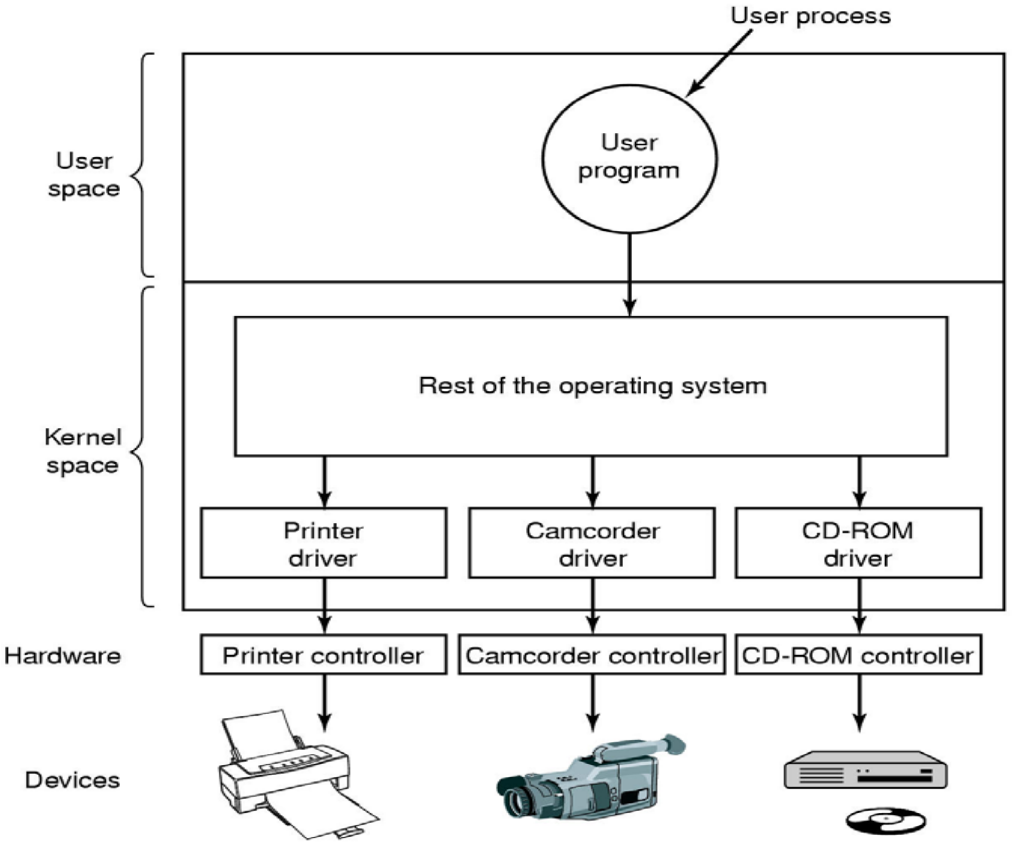
Interrupt Handler (2)

1. Save registers not already saved by HW-interrupt mechanism
2. Set up context (address space) for interrupt service procedure
    - Typically, handler runs in the context of the currently running process/task ⇒ not that expensive context switch
3. Set up stack for interrupt service procedure
    - Handler usually runs on the kernel stack of the current process/kernel-level thread
    - Handler cannot block, otherwise the unlucky interrupted process/kernel-thread would also be blocked, might lead to starvation or even to a deadlock
4. Acknowledge/mask interrupt controller, thus re-enable other interrupts

Interrupt Handler (2)

5. Run interrupt service procedure
   - Acknowledges interrupt at device level
   - Figures out what caused the interrupt, e.g.
     - Received a network packet
     - Disk read has properly finished, ...
   - If needed, it signals the blocked device driver
6. In some cases, we have to wake up a higher priority process/kernel level thread
   - Potentially schedule another process/kernel-level thread
   - Set up MMU context for process to run next
7. Load new/original process' registers
8. Return from Interrupt, start running new/original process

Device Driver



Communication between drivers and device controllers is done via the bus

Device Driver

- Drivers classified into similar categories
  - Block devices and
  - Character (stream of data) devices
- OS defines standard (internal) interface to the different classes of devices
  - Device drivers job
    - Translate user request through device-independent standard interface, e.g. open, read, ..., close) into appropriate sequence of device or controller commands (register manipulation)
    - Initialize HW at boot time
    - Shut down HW

Device Driver

- After issue the command to the device, device either
    - completes immediately and the driver simply returns to the caller or it
    - processes request and the driver usually blocks waiting for an I/O (complete) interrupt signal
- Drivers are reentrant as they can be called by another process while a process is already blocked in the driver
    - Reentrant: code that can be executed by more than one thread (or CPU) at the same time
        - Manages concurrency using synch primitives

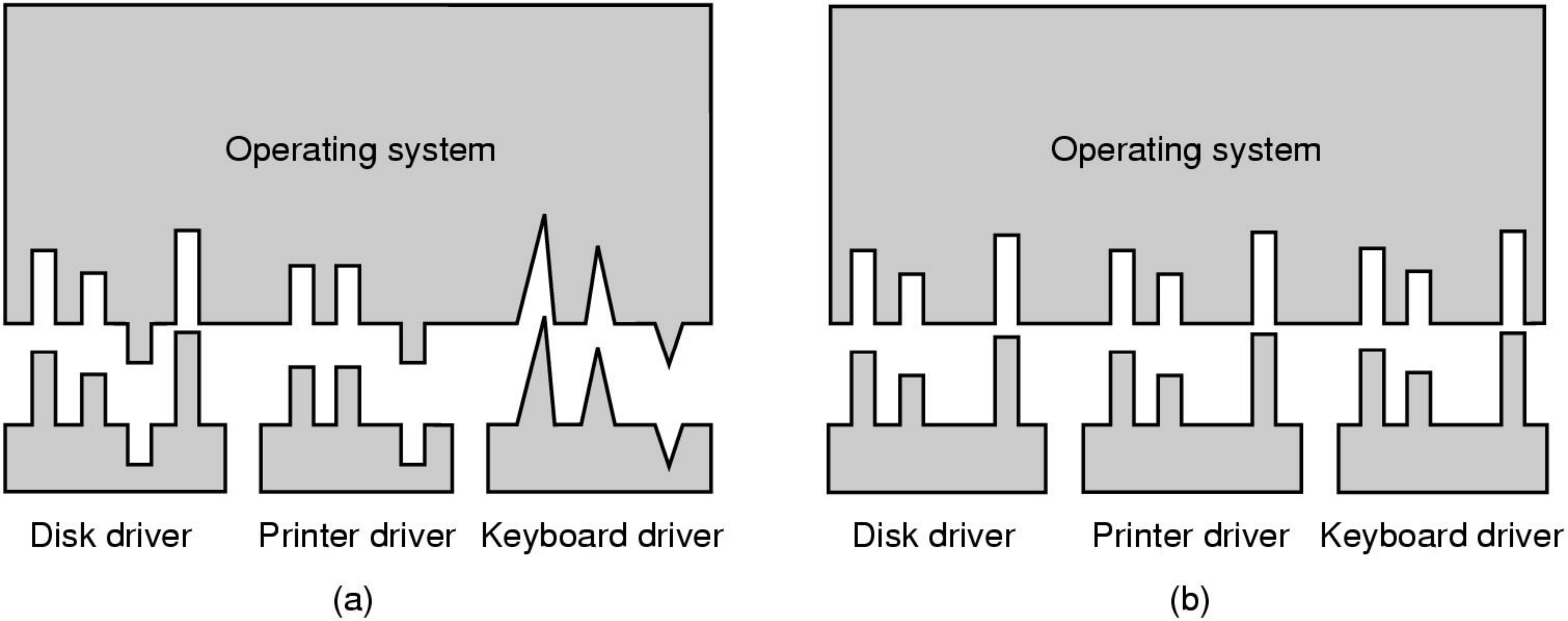
Device Drivers upon Micro-Kernels

- Single threaded
  - Accepting user request
  - Preparing device (controller)
  - Reacting on interrupt
- Multi-threaded
  - Repeated single-threaded
  - Pipe-lining

Device-Independent I/O Software (1)

- There is some commonality between drivers of similar classes ⇒
  - Divide I/O software into device-dependent and device independent I/O software, e.g.
    - Buffer or buffer-cache management, i.e. provide a device-independent block size
    - Allocating and releasing dedicate devices
    - Error reporting to upper levels, i.e. all errors the driver cannot resolve

Device-Independent I/O Software (2)



(a) Without a standard driver interface

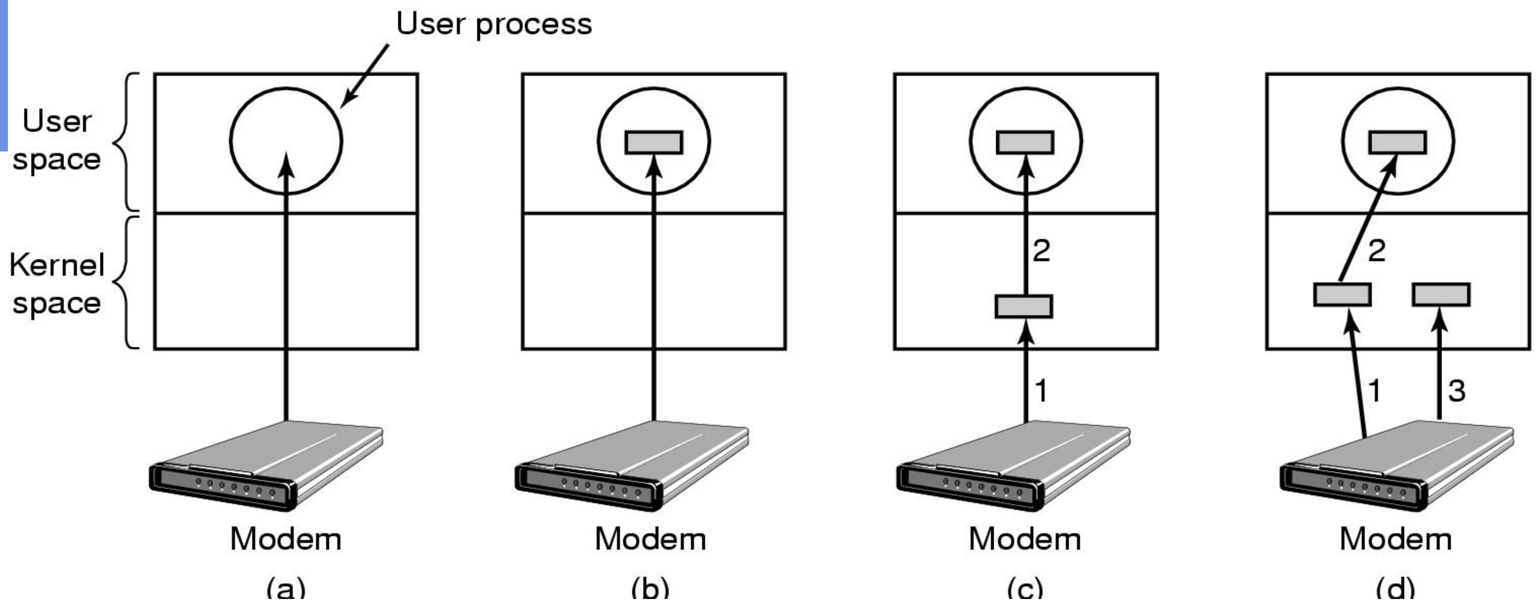
(b) With a standard driver interface

Device-Independent I/O Software (3)

Driver ⇔ Kernel Interface

- Uniform interface to devices and kernel
  - Uniform device interface for kernel code
    - Allows different devices to be used in the same way, e.g. no need to rewrite your file-system when you are switching from IDE to SCSI or even to RAM disks
    - Allows internal changes of drivers without fearing of breaking kernel code
  - Uniform kernel interface for device code
    - Drivers use a defined interface to kernel service, e.g. kmalloc, install IRQ handler, etc.
    - Allows kernels to evolve without breaking device drivers

Device Independent Software (4)



(a) Unbuffered input

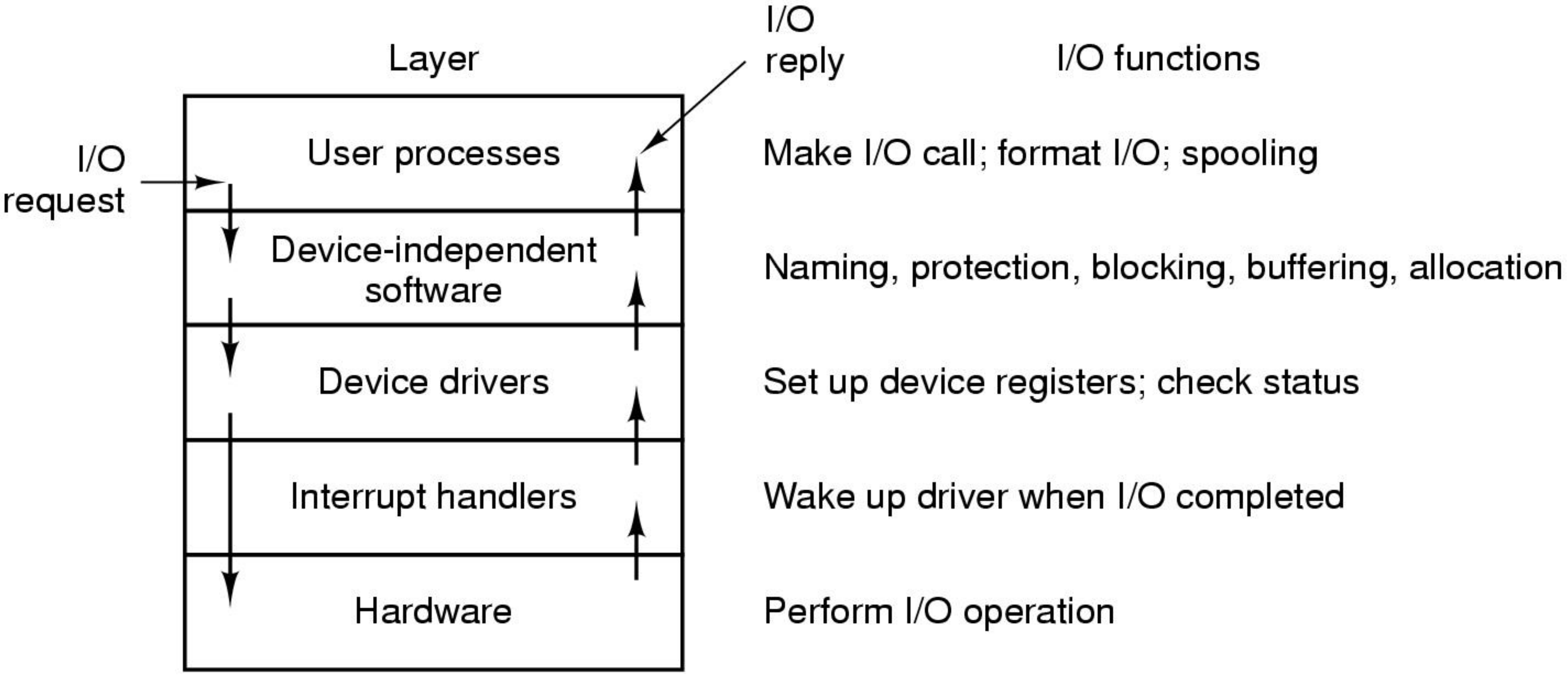
(b) Buffering in user space

(c) Buffering in the kernel followed by copying to user space

(d) Double buffering in the kernel

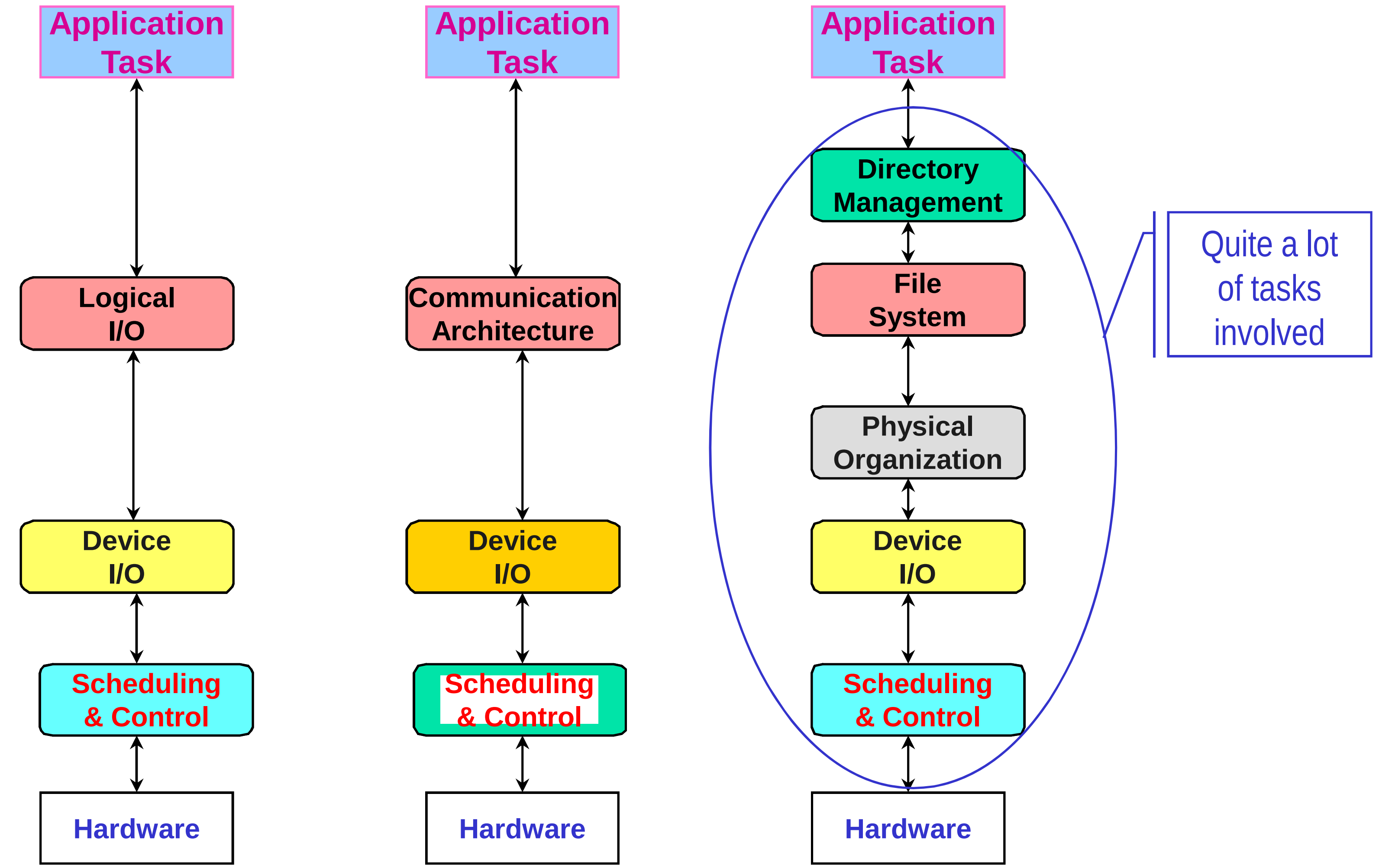
Buffering on later Slides

I/O Software Summary



Layers of I/O system and main functions of each layer

Examples of I/O-Organization



Local peripheral device **Communications port** **File System**

I/O Buffering*

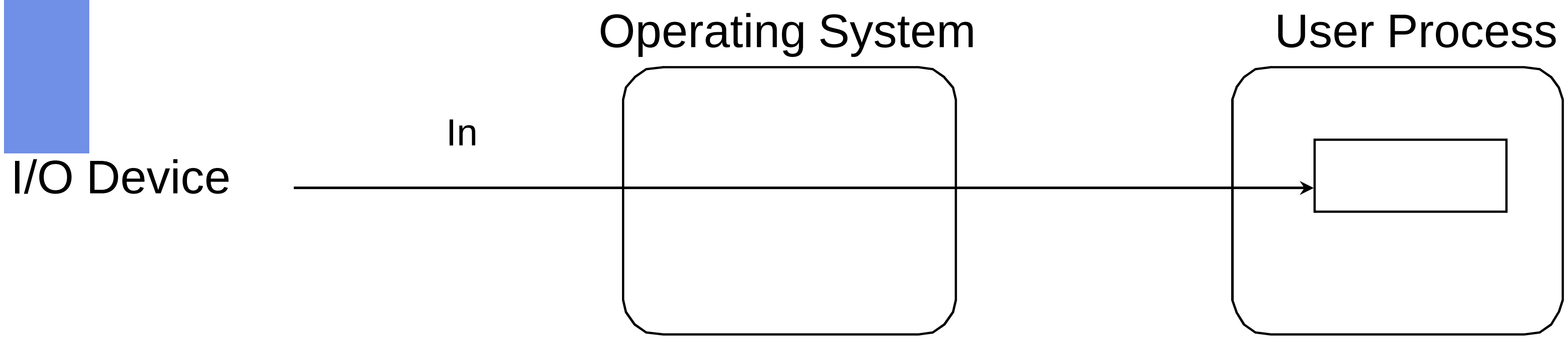
- Reasons for buffering
  - Otherwise threads must wait for I/O to complete before proceeding
  - Pages must remain in main memory during physical I/O
- Block-oriented
  - information is stored in fixed sized blocks
  - transfers are made a block at a time
  - used for disks and tapes
- Stream-oriented
  - transfer information as a stream of bytes
  - used for terminals, printers, communication ports, mouse, and most other devices that are not secondary storage

*Principle of buffering was invented because of I/O

No Buffering

- Process reads/writes a device a byte/word at a time
  - Each individual system call adds significant overhead
  - Process must wait until every I/O is complete
  - Blocking/interrupt handling/deblocking adds to overhead
  - Many short CPU phases are inefficient, because
    - overhead induced by thread_switch (or even worse address_space_switch)
    - poor cache and TLB usage

User Level Buffering



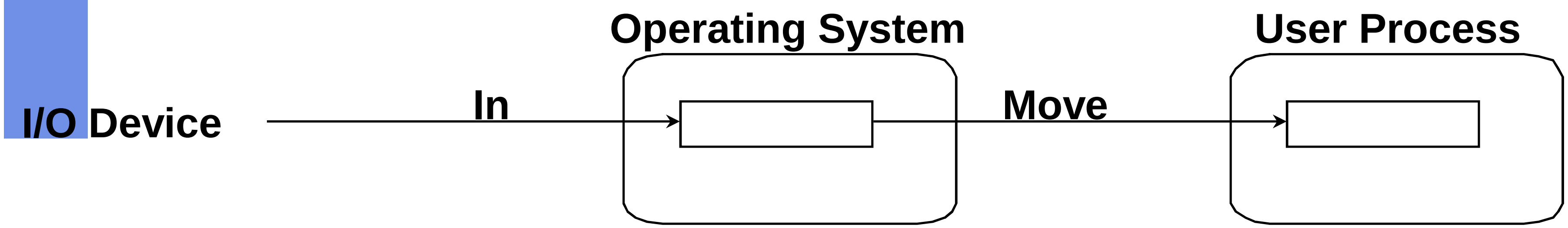
No buffering in OS

- Task specifies a memory buffer that incoming data is placed in until it fills
    - Filling can be done by interrupt service routine
    - Only one system_call and block/deblock per data buffer
        - More efficient than “NO BUFFERING”

User Level Buffering

- Issues
    - *What happens if buffer is currently paged out to disk?*
        - You may loose data while buffer is paged in
        - You could lock/pin this buffer (needed for DMA), however, you have to trust the application programmer, that sheThe is not starting a denial of service attack
    - *Additional problems with writing?*
        - *When is the buffer available for re-use?*

Single Buffer



Single buffering

- User Process can process one block of data while next block is read in
- Swapping can occur since input is taking place in system memory, not user memory
- OS keeps track of assignment of system buffers to user processes

Single Buffer

- Stream-oriented
  - Buffer is an input line at time with carriage return signaling the end of the line
- Block-oriented
  - Input transfers made to system buffer
  - Buffer moved to user space when needed
  - Another block is read into system buffer

Single Buffer Speed Up

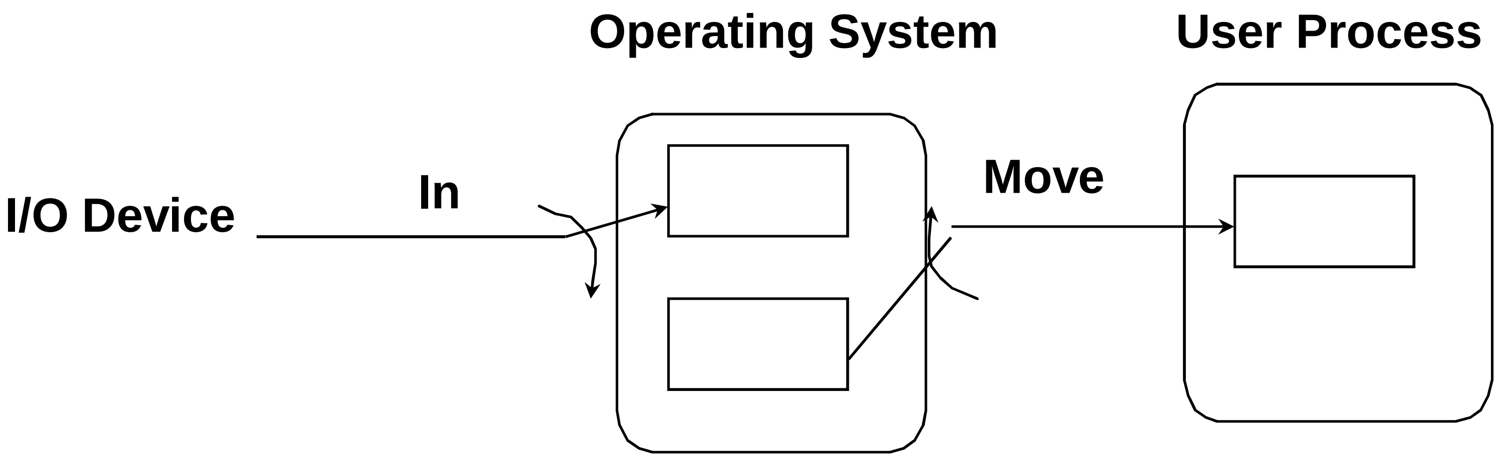
- Assumption:
  - T = transfer time from device
  - C = copying time from system- to user-buffer
  - P = processing time of complete buffer content
  - Processing and transfer can be done in parallel
  - Potential speed up with single buffering:

$$\frac{T + P}{\max\{T, P\} + C}$$

Single Buffer Problem

- *What happens if system buffer is full, user buffer is swapped out, and more data is received?*
  - *Loose characters or drop network packets*

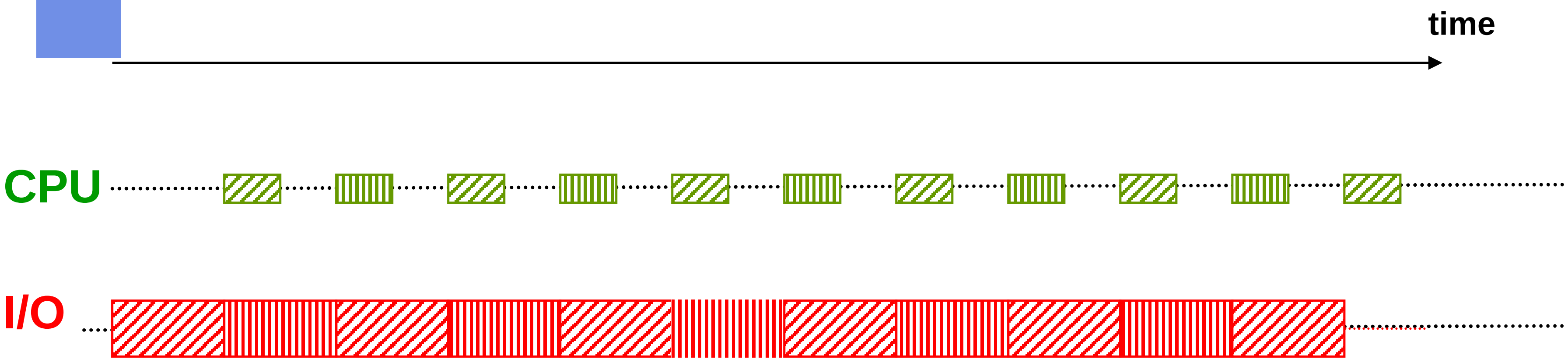
Double Buffer



Double buffering

- Use 2 system buffers instead of 1 (per user process)
- User process can write to or read from one buffer while the OS empties or fills the other buffer

Timing Diagram for Double Buffering



Analysis: The slower I/O-device is busy the whole input-period, thus additional buffers are not needed (in this case).

Double Buffer Speed Up

- Processing and memory copying in parallel with data transfer ⇒
- Speed up with double buffering:

$$\frac{T + P}{\max\{T, P+C\}}$$

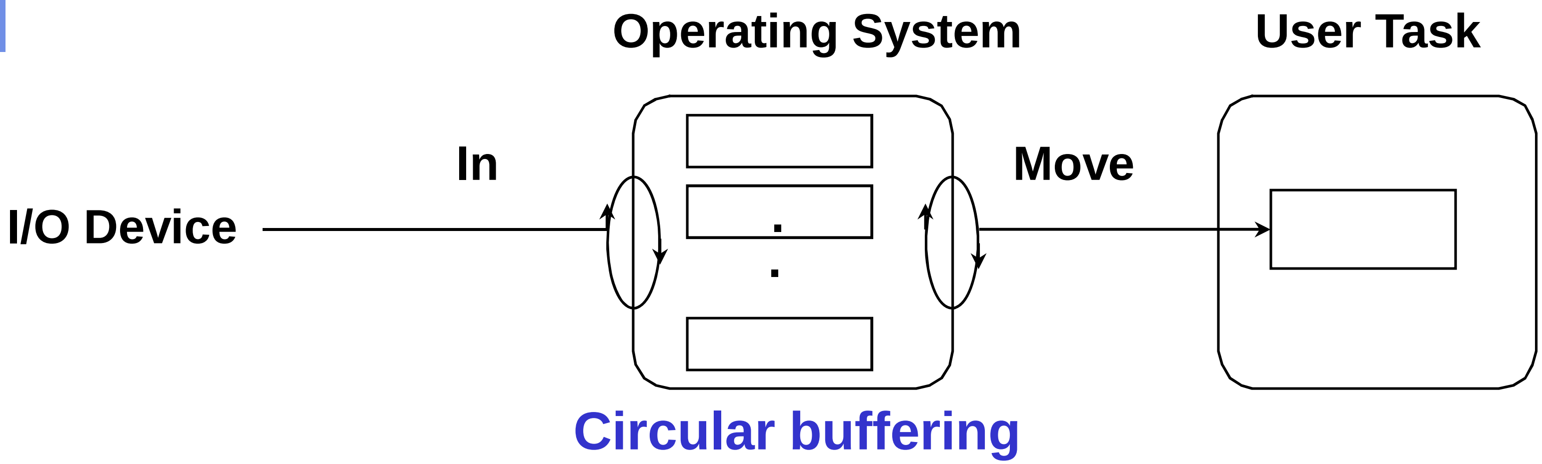
- Usually C << than

Circular Buffering

- Double buffering may be insufficient for really bursty traffic situations:
  - Many writes between long periods of computations
  - Long periods of computations while receiving data
  - Might want to read ahead more than just a single block from disk

⇒ *Circular buffering with n>1 system buffers*

Circular Buffering



Circular buffering

- More than two buffers are used to face I/O-bursts
- Each individual buffer is one unit in a circular buffer

How to implement Buffering?

- Remember:

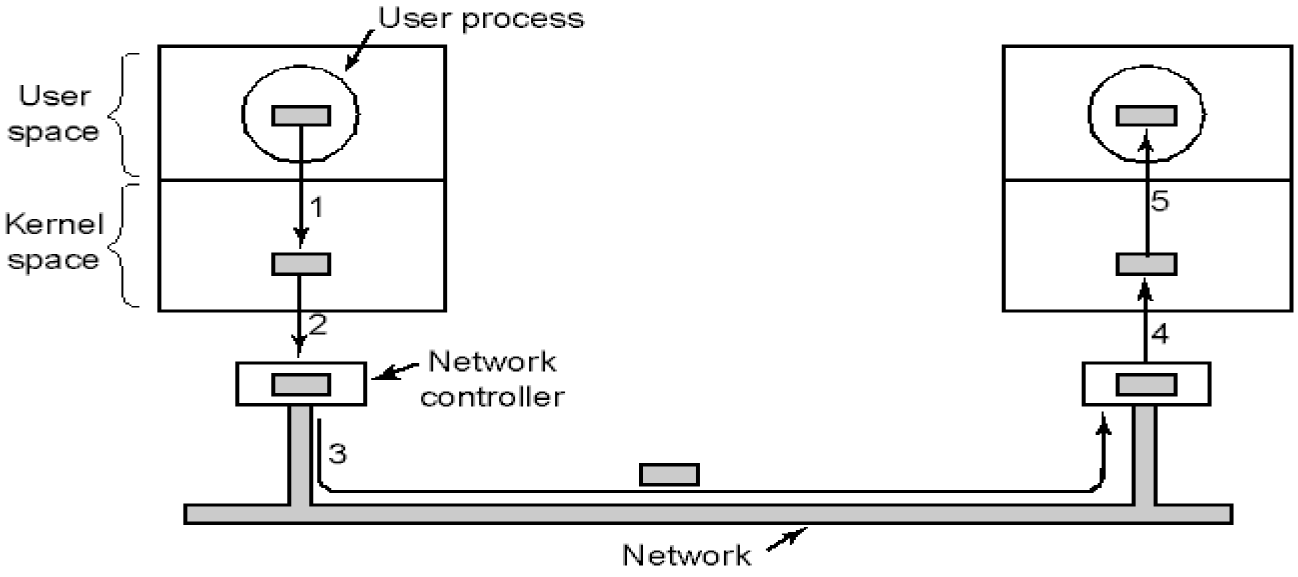
  Single-, double-,and circular-buffering are all

  Bounded-Buffer

  Producer-/Consumer Problems

- *Is buffering always a good idea?*
- Analyze carefully

Buffering in Fast networks

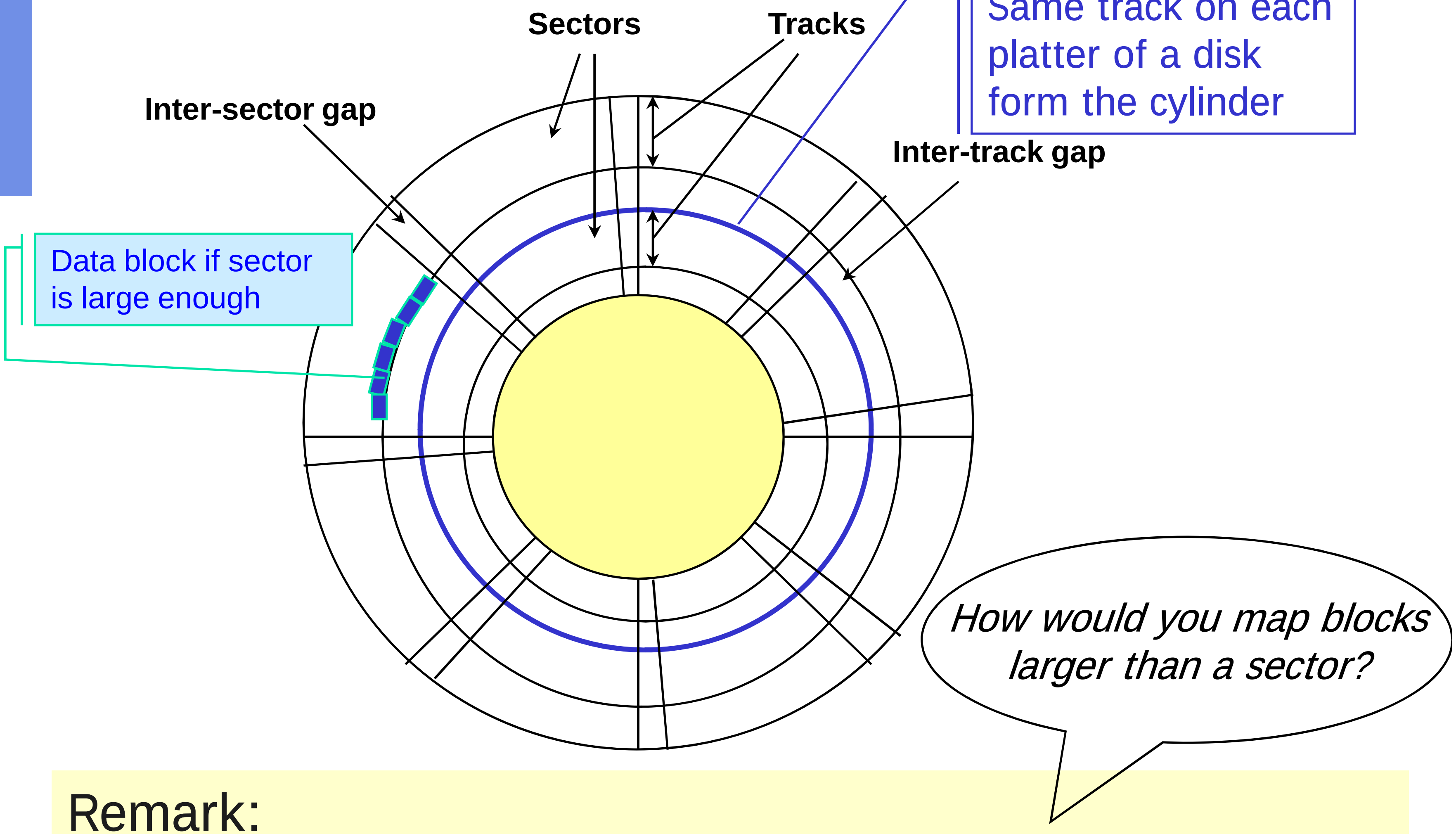


- Networking may involve many copies
- Copying reduces overall performance
- Super-fast networks put significant effort into achieving zero-copying
- Buffering may also increase latency

Disk Management

- Management of disk accesses is important
    - Huge speed gap between main memory and disk
    - Disk throughput is sensitive to
        - Request order ⇒ Disk Scheduling
        - Placement of data on the disk ⇒
            - File System Design and Implementation
            - Swap Area Design
        - Disk scheduler must be aware of disk geometry

Disk Data Layout



Same track on each platter of a disk form the cylinder

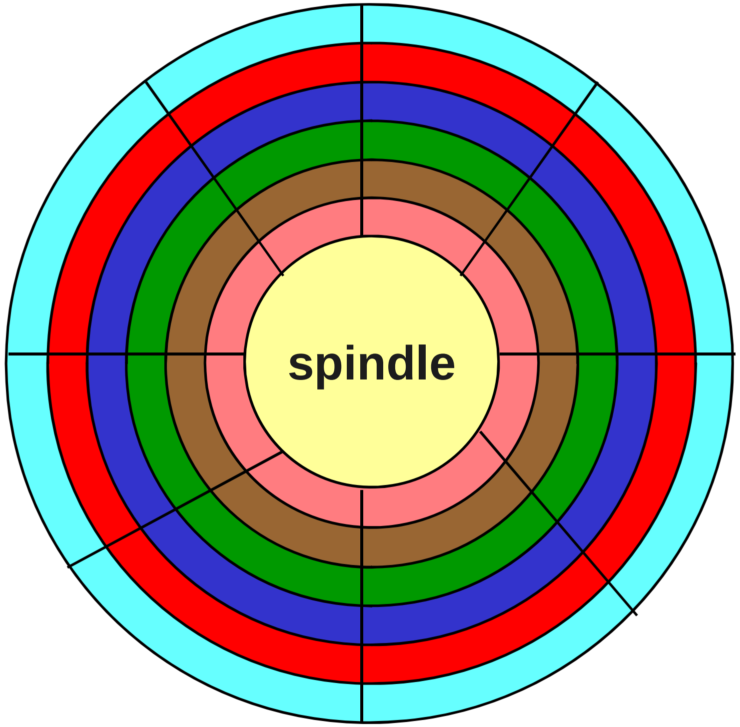
Data block if sector is large enough

How would you map blocks larger than a sector?

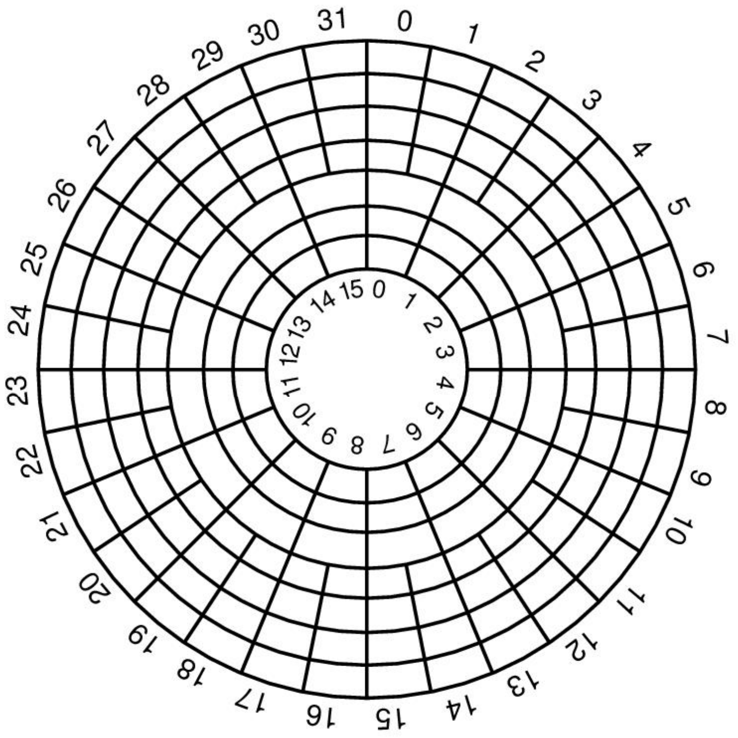
Remark:
Typical sector size 0.5 KB, typical block size $\in$[0.5, 8] KB

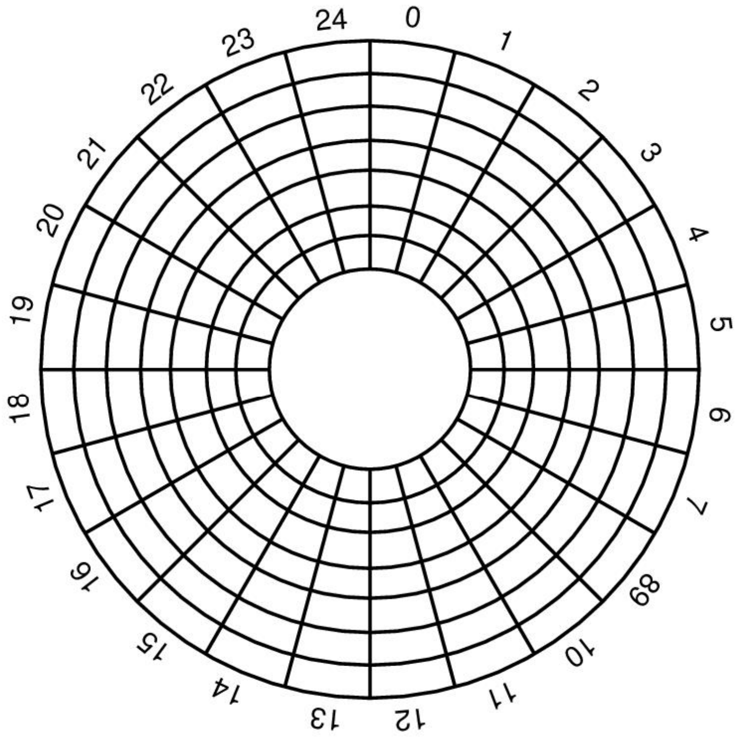
Partitioning a Disk

- Set of consecutive cylinders form a “disk partition”
- FFS divides a partition into **c** cylinder groups: Storing “related data” into one cylinder group may help to minimize head movements
- Contiguous blocks of a file are located within a cylinder-group using interleaving



Modern Disk Geometry





- Physical geometry of a disk with two zones
- A possible virtual geometry for this disk

Disk Hardware

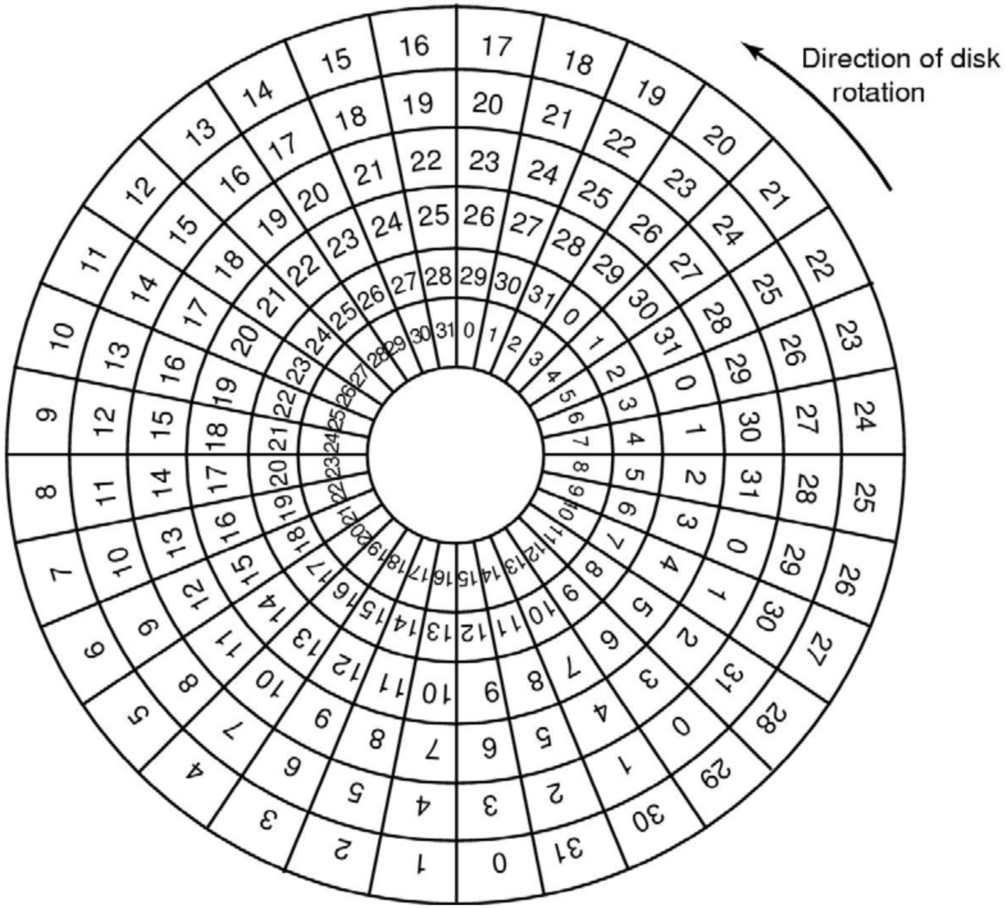
Parameter	IBM 360-KB floppy disk	WD 18300 hard disk
Number of cylinders	40	10601
Tracks per cylinder	2	12
Sectors per track	9	281 (avg)
Sectors per disk	720	35742000
Bytes per sector	512	512
Disk capacity	360 KB	18.3 GB
Seek time (adjacent cylinders)	6 msec	0.8 msec
Seek time (average case)	77 msec	6.9 msec
Rotation time	200 msec	8.33 msec
Motor stop/start time	250 msec	20 sec
Time to transfer 1 sector	22 msec	17 μsec

Disk parameters for the original IBM PC floppy disk
and a Western Digital WD 18300 hard disk

Low Level Disk Formatting (1)

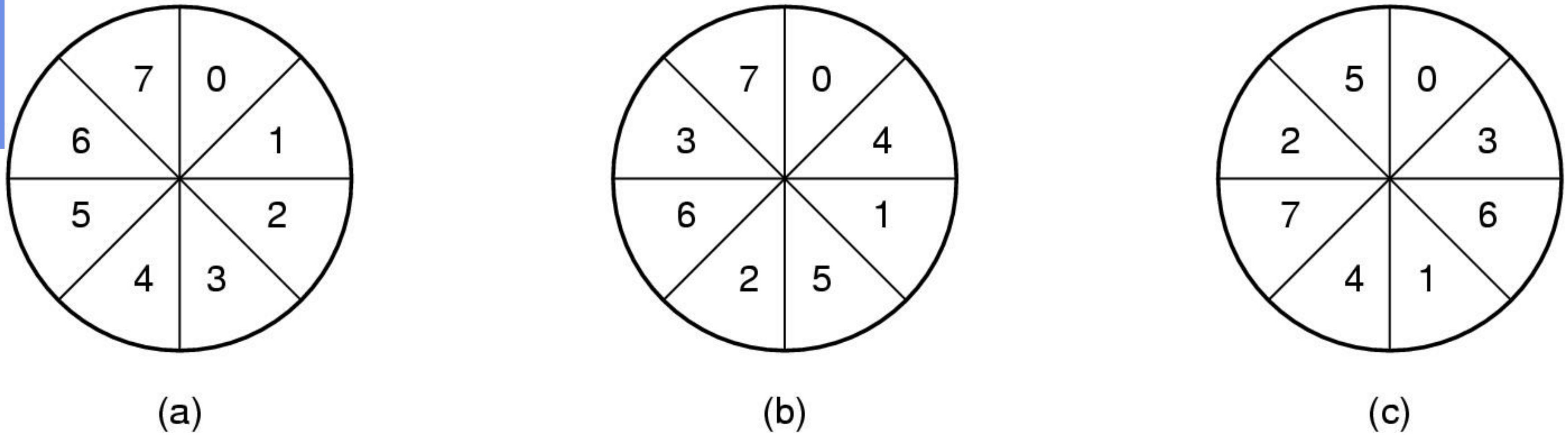
A disk sector

Low Level Disk Formatting (2)



An illustration of *cylinder skew*

Low Level Disk Formatting (3)



- No interleaving
- Single interleaving
- Double interleaving
- Modern drives overcome interleaving by simply reading the entire track into the on-disk-controllers cache

Disk Performance Parameters (1)

- To read or write from or to a disk, the disk head must be positioned at the desired track (and at the beginning of the desired sector)
- Seek time
  - time it takes to position head at the desired track
- Rotational delay or rotational latency
  - time its takes until the desired sector has been rotated to line up with read/write-head

Disk Performance Parameters (2)

- Access time
  - sum of seek time and rotational delay
  - the time it takes to get in position to read or write
- Data transfer occurs as the sector moves under the head
- Data transfer for an entire file is faster when the file is stored in the same cylinder and in adjacent sectors

Performance Charactersitic of Disks

- Time required to read or write a disk block determined by 3 factors
  - Seek time
  - Rotational delay
  - Actual transfer time
- Seek time dominates
- For a single disk, there will be a number of disk-I/O requests $\Rightarrow$ processing them in random order leads to worst possible disk performance
- Error checking is done by controllers

Disk Scheduling

No longer needed

Most of Disk scheduling is done

by the Disk Controller

Overview: Disk Scheduling Policies

- Random (no real policy at all)
- First come, first served (FCFS)
- Priority (???)
- SCAN
- C-SCAN
- N-Step SCAN
- Minimal Seek Time First (Stalling’s Shortest Service Time First!)
- Antcipatory Disk Scheduling
- Shortest Service Time First
- Proportional-share scheduling

Seek time
reducing
disk schedulers

First come, first served (FCFS)

- Manage disk requests as they come
- Fair to all “disk clients” (⇒ no starvation)
- Good for just a few concurrent processes/tasks with clustered requests
- Performs ~ random scheduling”

  if there are many concurrent “disk clients”

Remark: Already a single “*copy file*” may lead to a “*ping-pong effect*” on the disk surface

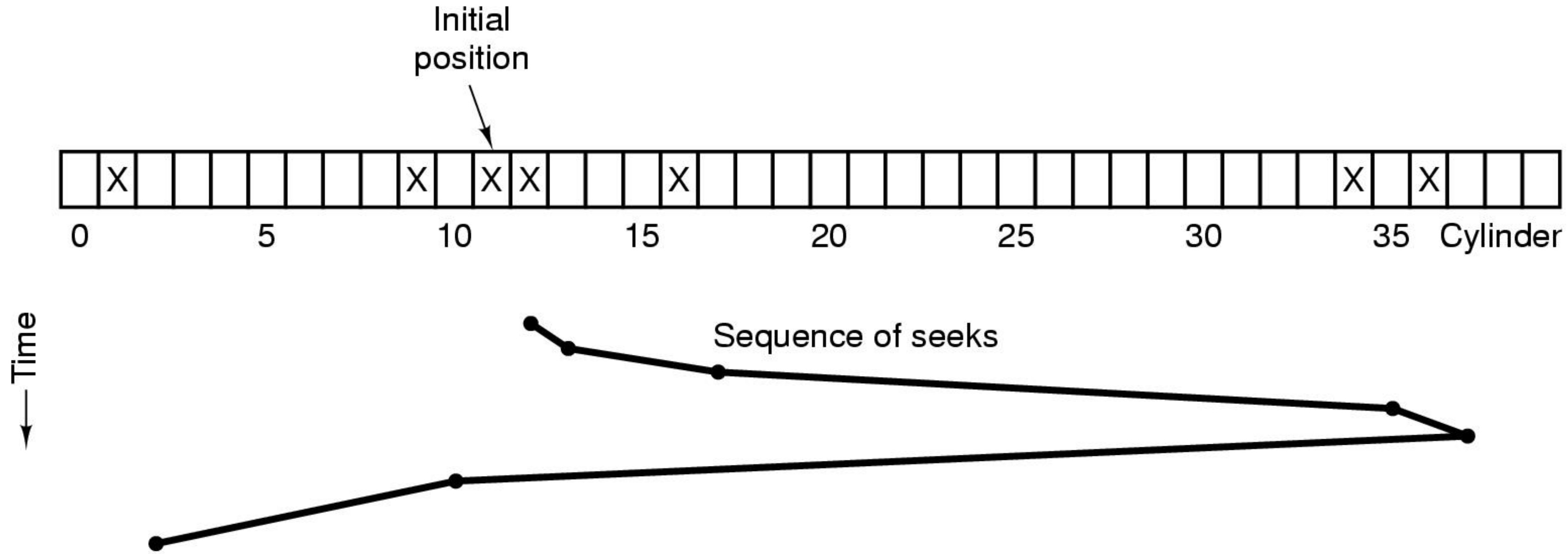
Priority

- Goal is not to optimize disk usage, but to meet other objectives, e.g. favor special applications
- Short batch jobs may have higher priority
- May improve turnaround times of these high priority jobs, but ....??

SCAN (~Elevator)

- Disk arm moves in one direction
  - satisfying all pending requests until it reaches the last track in that direction
  - Direction of arm movement is reversed afterwards, ...
- Better than FCFS, usually worse than SSTF
- Makes poor use of sequential reads on down-scan

Example: SCAN



C(ircular)-SCAN

- Like elevator, but restricts scanning to one direction only
  - when last track has been visited, move arm at full speed to first track
- Better locality on sequential reads
- Better use of read ahead cache on controller
- Reduce maximal delay to read a particular sector

N-step-SCAN

What‘s the optimal N?
How to initialize?

- segments the disk request queue into sub-queues of length **N**
- sub-queues are processed one at a time, using SCAN
- new requests added to another queue

FSCAN

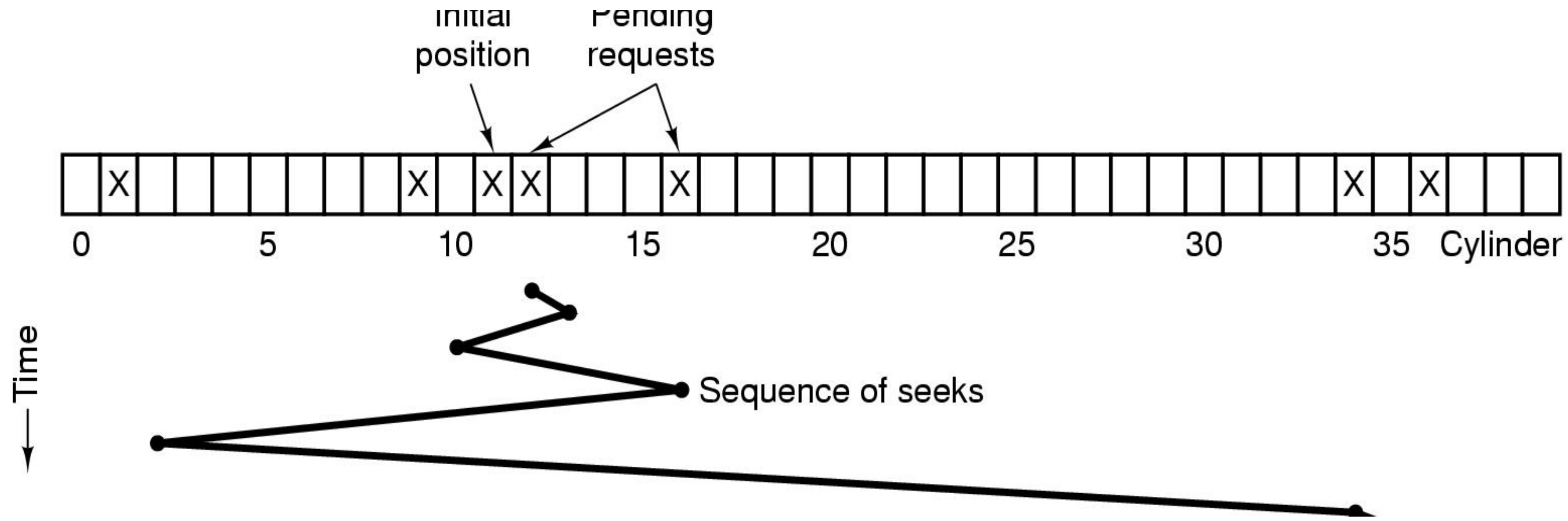
- (no limit on queue-length)
    - two queues
    - one queue is empty for new request

Shortest Seek Time First (SSTF)

- Select the disk I/O request that requires the least movement of the disk arm from its current position
- Each request on the most neighbored track is serviced regardless of its potential delay due to rotational time

Remark:
Requests on the most outer/inner tracks may starve, if we have huge traffic in the midst or at the opposite side of the disk

Example: SSTF



Shortest Service Time First (SSvTF)

- Select disk I/O request that is serviced with minimal sum of seek and rotational time

Analysis: Algorithmic drawback (comparable to chess novice)

Just looking for 1 minimal request,

don‘t reflecting a sequence of requests!!

Counterargument:

Too much overhead and possible changes due to new arrivals of disk requests.

Proportional-Share Scheduler

- Offers a usage ratio to the current active competing tasks
- Enables to give quality of service guarantees to disk-clients

Anticipatory Disk Scheduling*

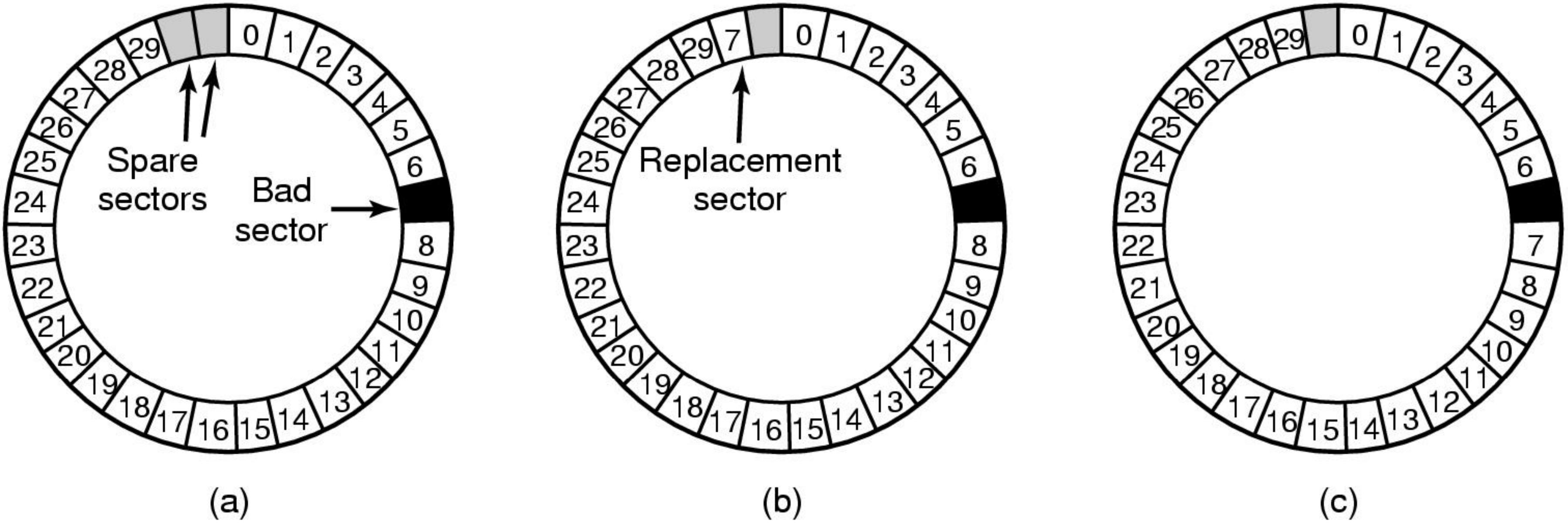
- See slides of “HotSystem WT 200172002” and http://cs.nmu.edu/~randy/research/speaches/1 on topic: Dusk Scheduling in Linux
- Idea:

  Even though there is another request, wait a bit, may be a better one will arrive soon
- Having waited long enough, use SCAN
- Goal:

  Having at least two different request sources, i.e. different application- or system-pocesses/tasks, next request = nearby

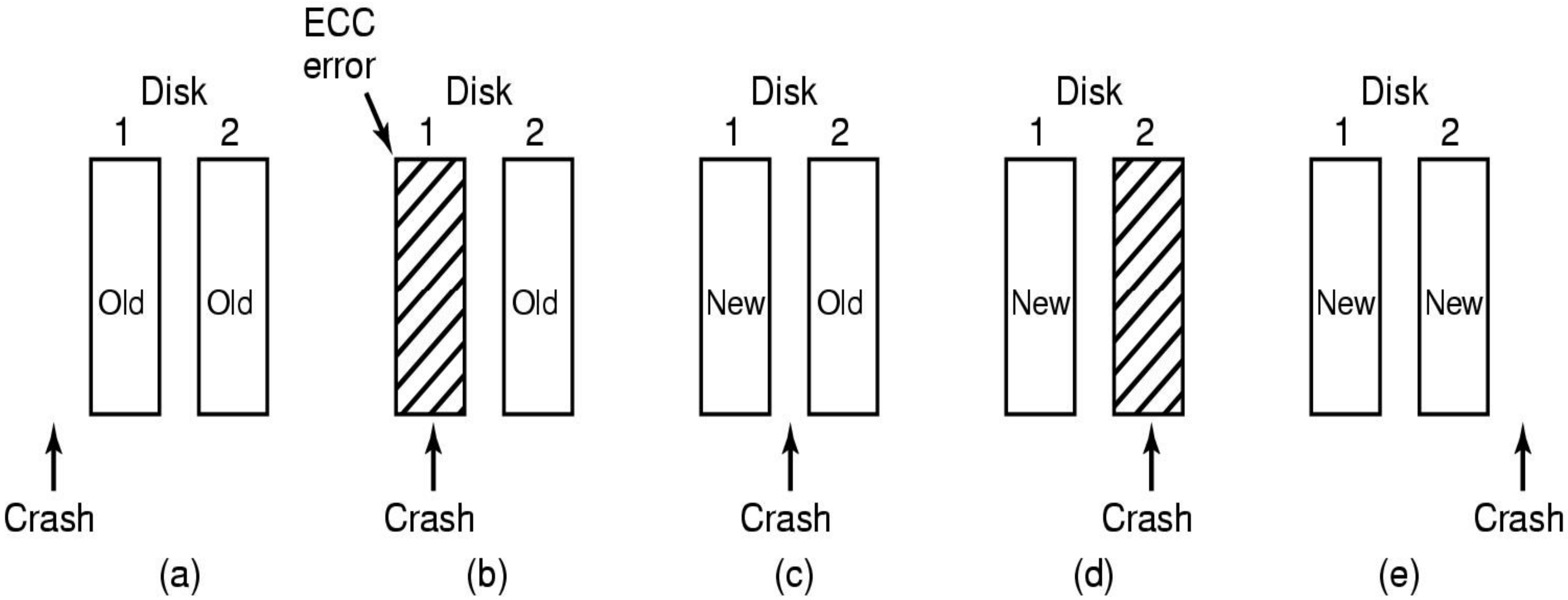
*Another famous proposal by P. Druschel’s team at Rice

Error Handling



- A disk track with a bad sector (and 2 spares)
- Substituting a spare for the bad sector
- Shifting all the sectors to bypass the bad one
- Bad sectors are handled transparently by on-disk-controller

Stable Storage



- Use 2 disks to implement stable storage
  - Problem is when a write(update) corrupts old version, without completing write of new version
  - Solution: First write to disk 1, then write to disk 2
  - Analysis of the influences of crashes on stable writes

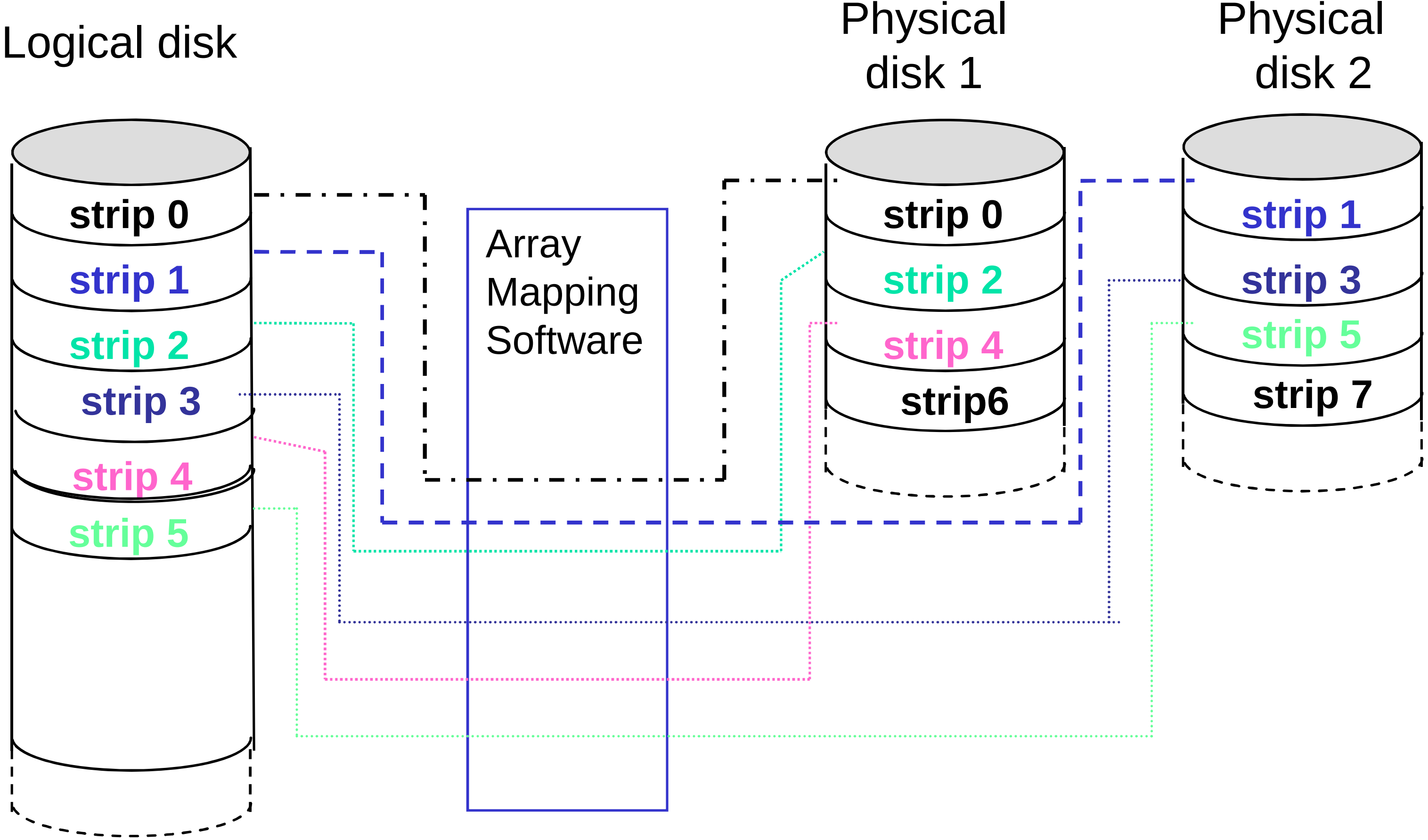
RAID Technology

Further Improvements for Disk-I/O

- Analysis:
  data rate of a disk << data rate of CPU or RAM
- Idea:
  - Use multiple disks to parallelize disk-I/O
  - provide a better disk availability
  - Instead of 1 single large expensive disk (SLED) use

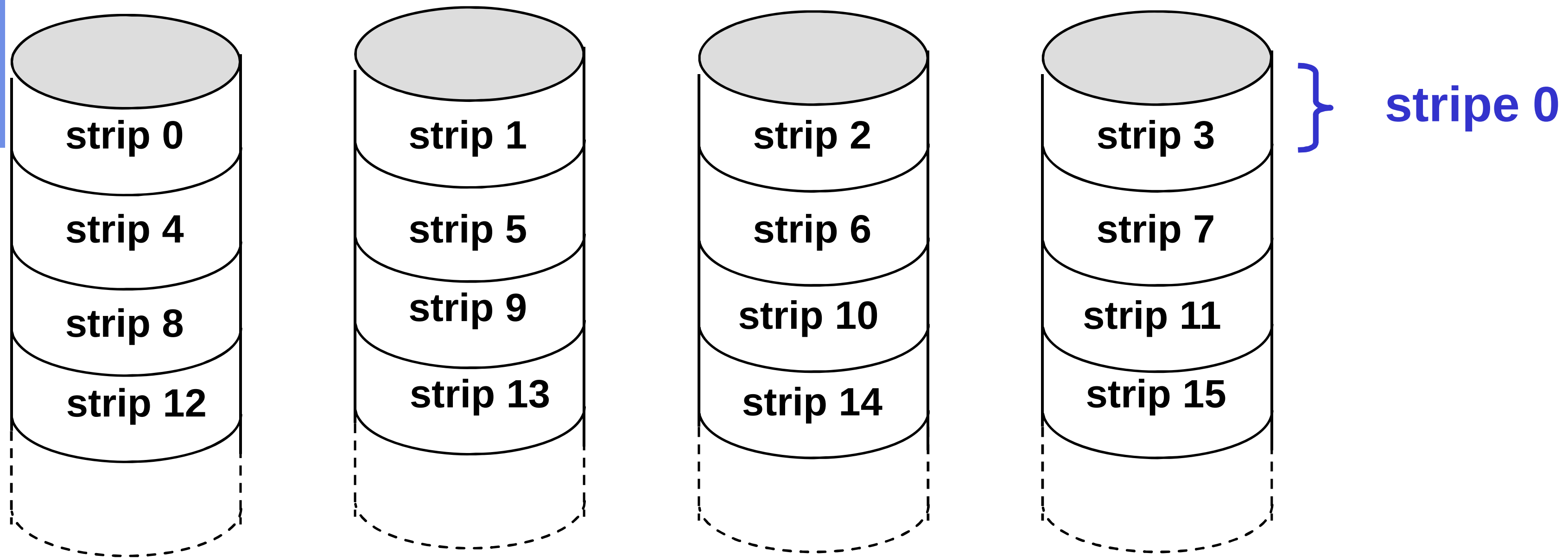
⇒ RAID = redundant array of independent disks
(originally: redundant array of *inexpensive* disks)

RAID Levels: Mapping Logical Disk(s) to Phsyical Disk(s)



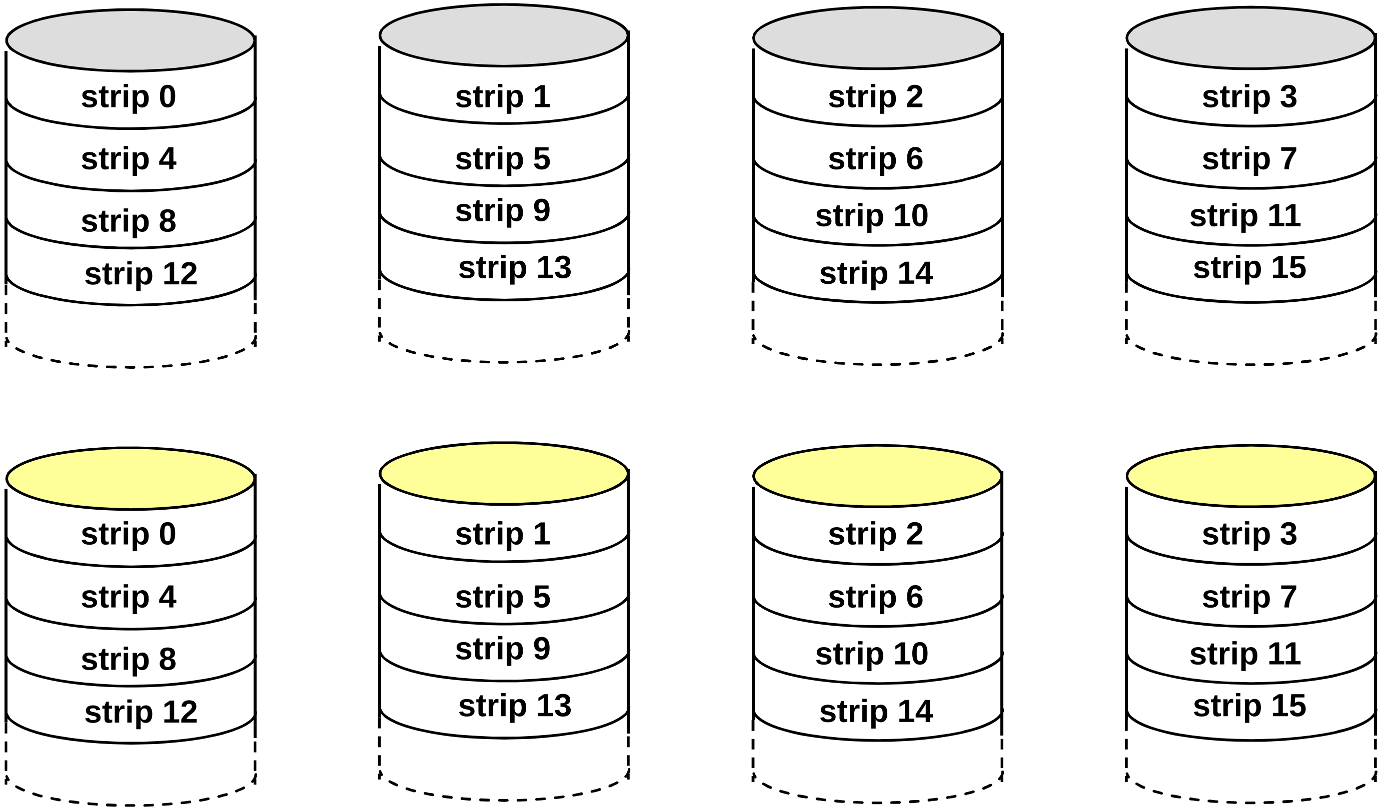
<u>Remark</u>:
A strip is either a physical block, e.g. a sector or a multiple of it

RAID 0 (without any redundancy)



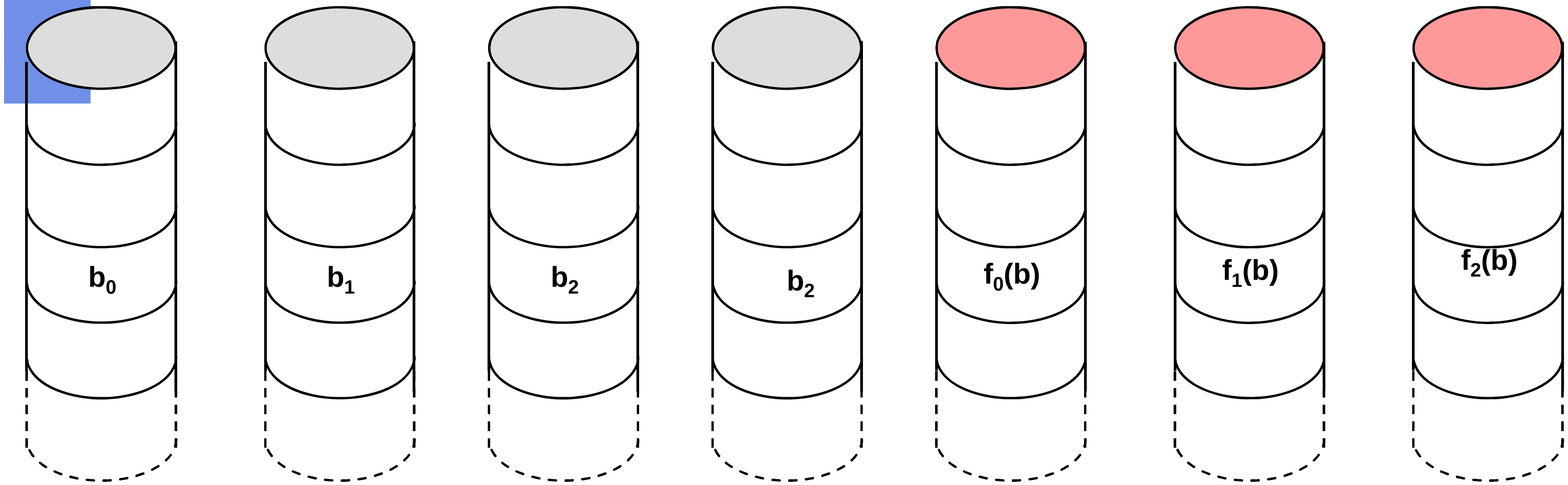
- Decreased availability compared to the SLED
- Increased bandwidth to/from logical disk
- Analyze applications which may profit from RAID0

RAID 1 (just mirrored)



Remark: Discuss the pros and cons of RAID 1. *How to start with?*

RAID 2 (redundancy through Hamming code)

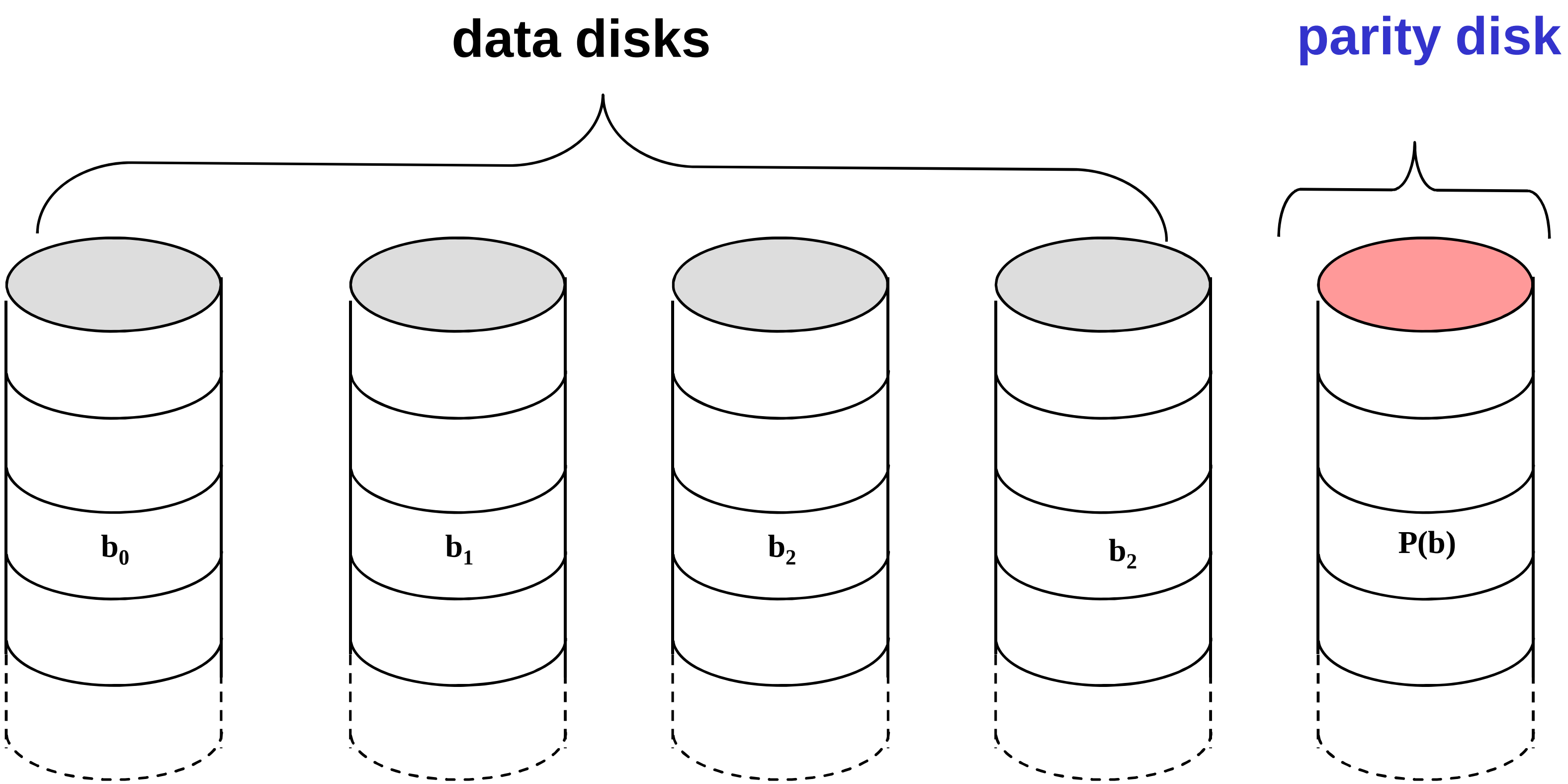


<u>Rough analysis:</u>

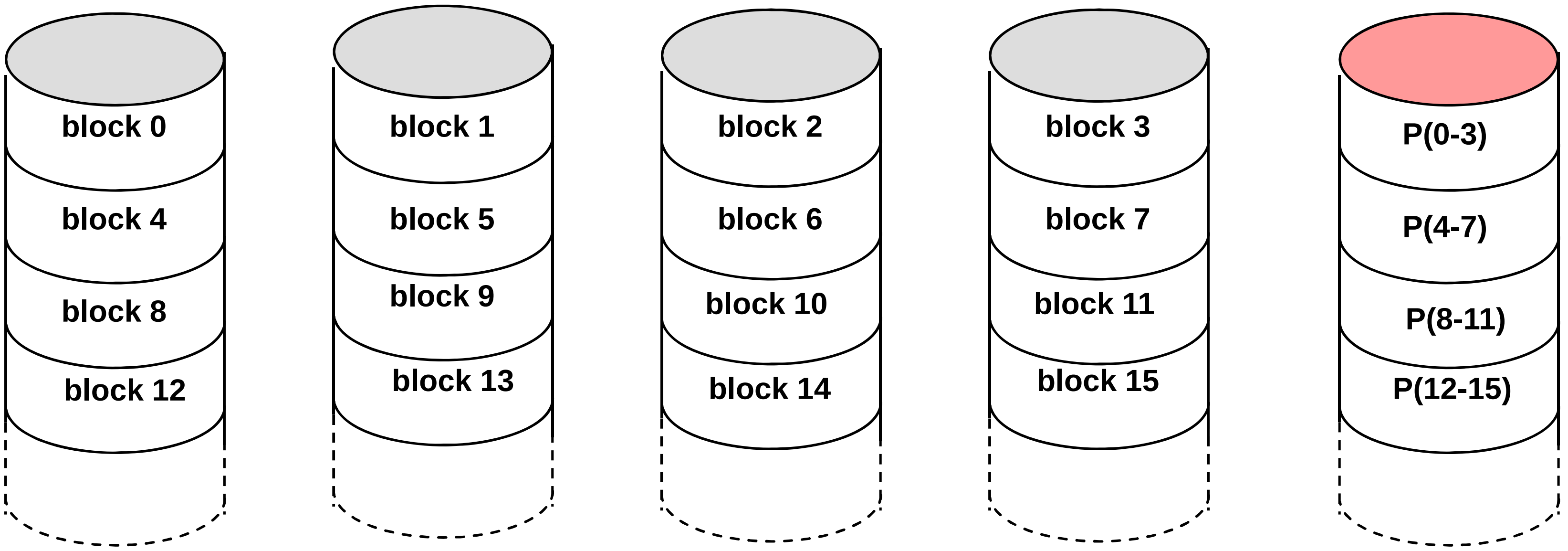
RAID 2 is an overkill and never implemented

Hamming code used for f(b), b are very small strips, still a remarkable disk overhead compared to RAID 0

RAID 3 (bit-interleaved parity)



RAID 4 (block-level parity)



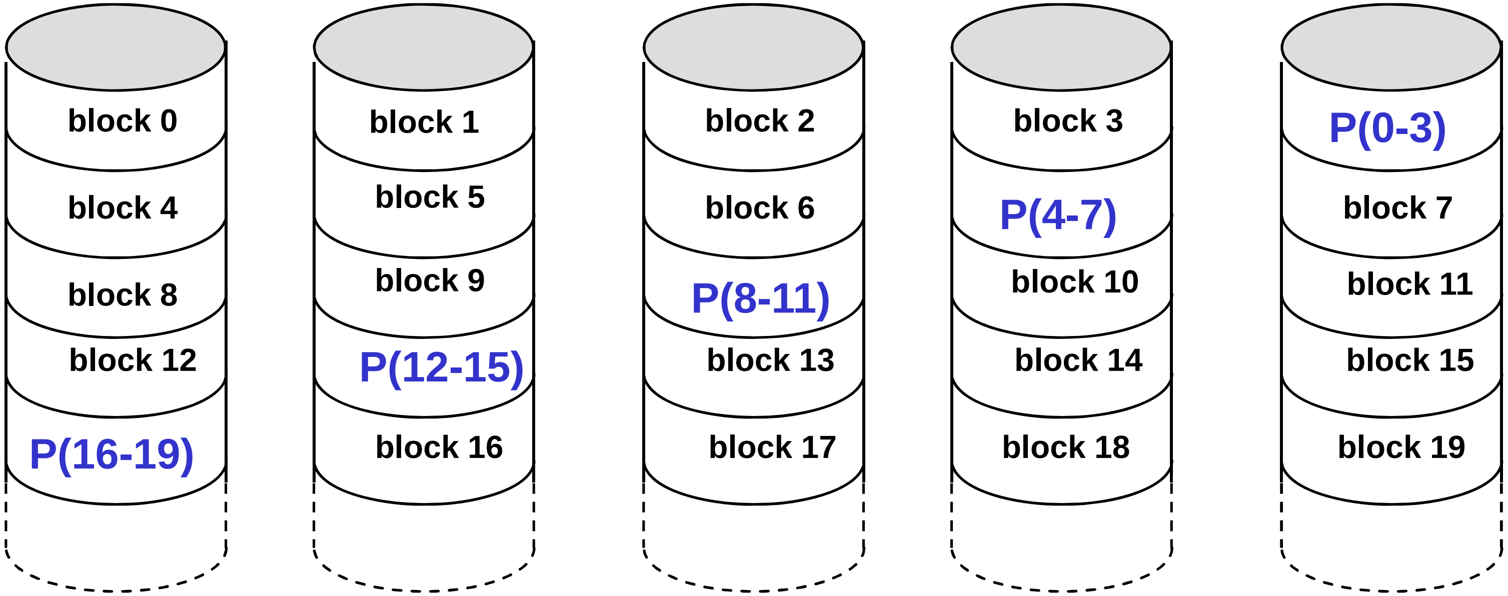
Parity computation: P(0..3) = block0 ⊗ block1 ⊗ block2 ⊗block3

Result:

Small updates require 2 reads (old block + parity) and 2 writes (new block + parity) to update a single disk block

Parity disk may be a bottleneck

RAID 5 (block-level distributed parity)



block 0	block 1	block 2	block 3	P(0-3)
block 4	block 5	block 6	P(4-7)	block 7
block 8	block 9	P(8-11)	block 10	block 11
block 12	P(12-15)	block 13	block 14	block 15
P(16-19)	block 16	block 17	block 18	block 19

- Like RAID4, but we distribute parity block on all disks ⇒no longer a “bottleneck disk”
- Update performance still less than on a SLED
- Reconstruction after a failure is a bit tricky

Raid Summary

- RAID0 provides performance improvements, but no additional availability
- RAID1 provides performance and availability improvements, but expensive to implement
- RAID5 is cheaper (only 1 single additional disk compared to RAID0), but has a poor write update performance
- Others are not used

Example: HP AutoRAID

- Uses RAID1 and RAID5 at the same time
- Hot data uses RAID1 for good performance
- When disk space is tight, it transparently migrates some of the data into RAID5
- Goal is to provide best of both approaches:
    - Good performance
    - Compact, available stable storage

Disk Caches

- Buffer in main memory for disk sectors (blocks)
- Contains a copy of some sector on the disk
- From time to time “cache contents” have to be “swapped out to” disk to keep the memory blocks consistent with the disk blocks
- If cache is full buffers have to be replaced according to some replacement policy (see paging)

Least Recently Used

- Block that has been in the cache the longest with no reference in the very past will be used for replacement
- Cache consists of a “stack of blocks”
- Most recently referenced block is on the top of the stack
- Whenever a block is referenced or brought into cache, it is placed on top of LRU-stack

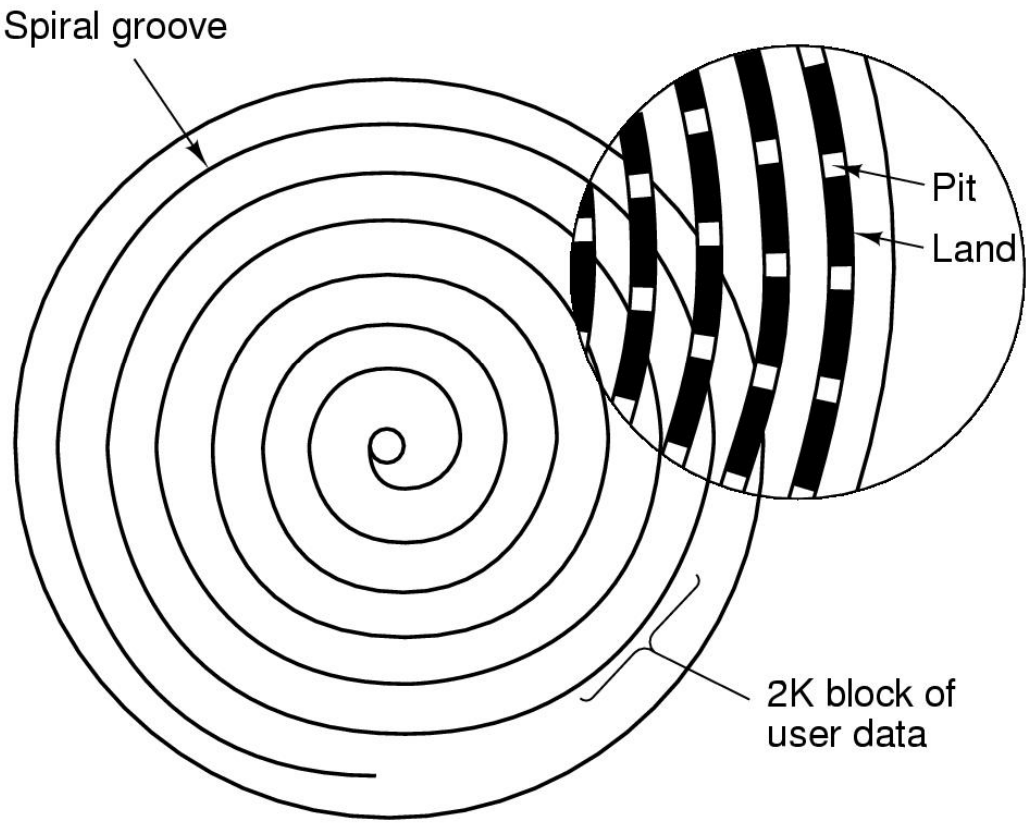
Least Recently Used

- The block on the bottom of the stack is removed when the cache is full, if a new block has to be swapped in
- Blocks don’t actually move around in main memory
- Pointers within some block-headers are used to establish the LRU-stack

Least Frequently Used

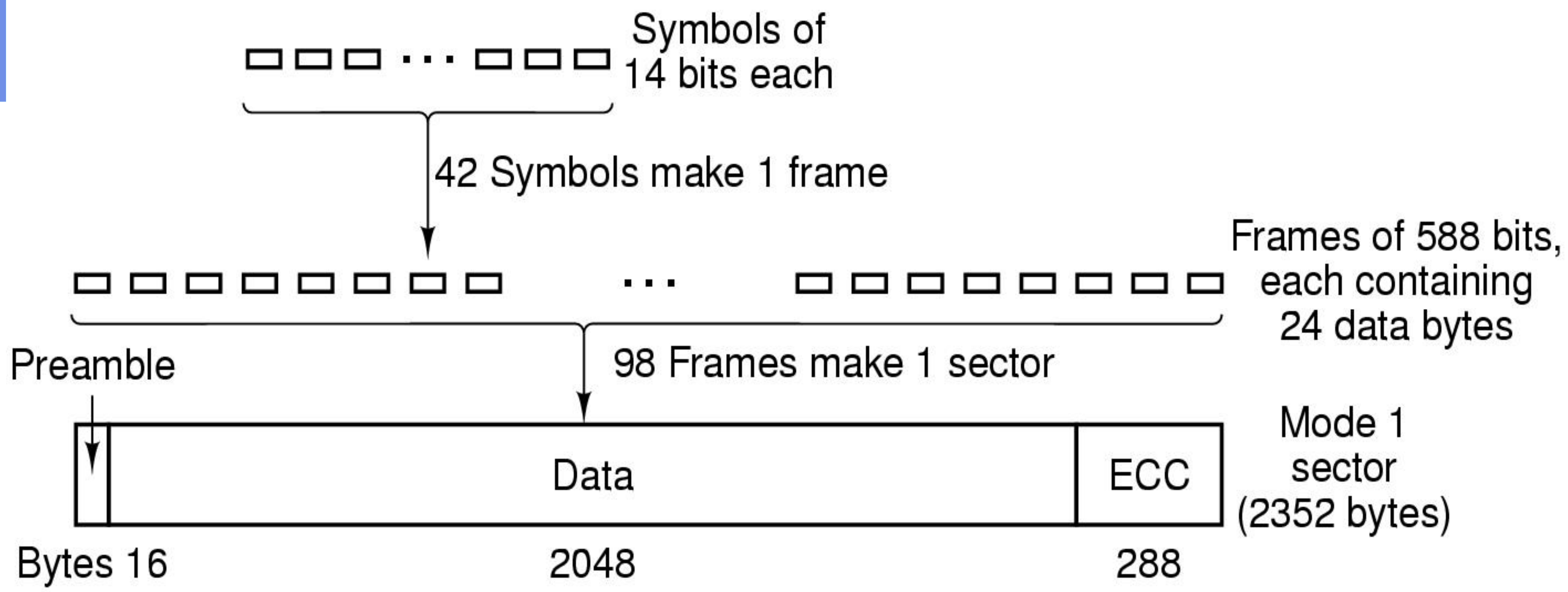
- The block that has experienced the fewest references is replaced
- A counter is associated with each block
- Counter is incremented each time block accessed
- Some blocks may be referenced many times in a short period of time and then not needed any more

CD-ROM



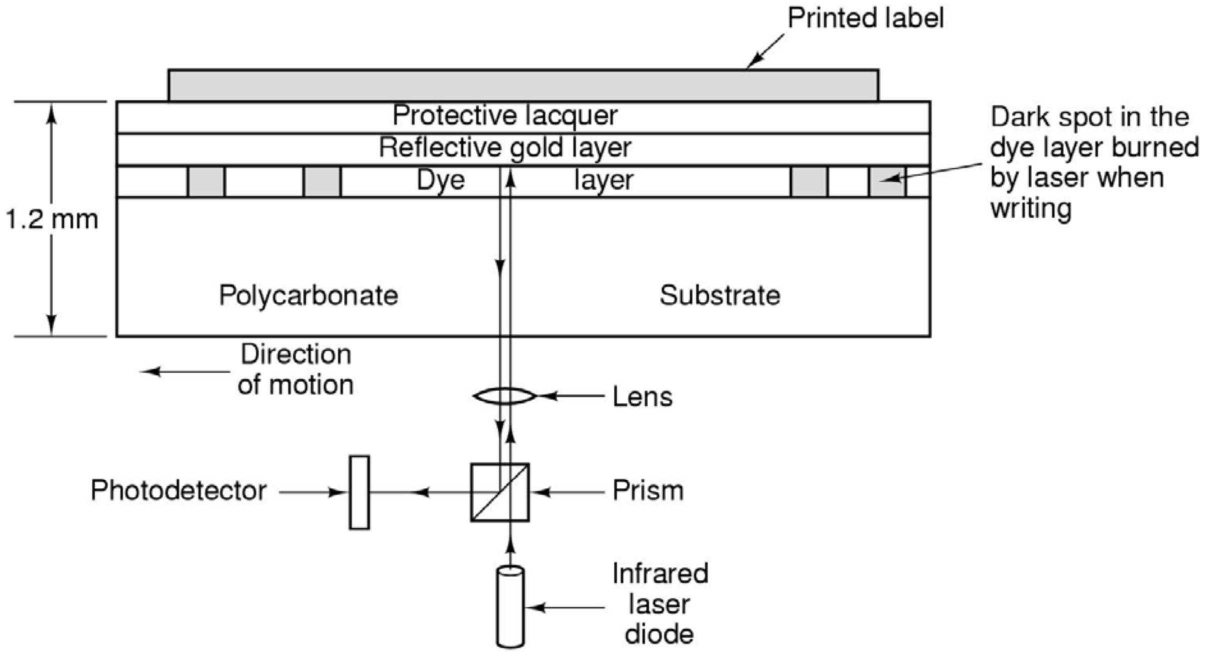
Recording structure of a CD or CD-ROM

CD-Rom (2)



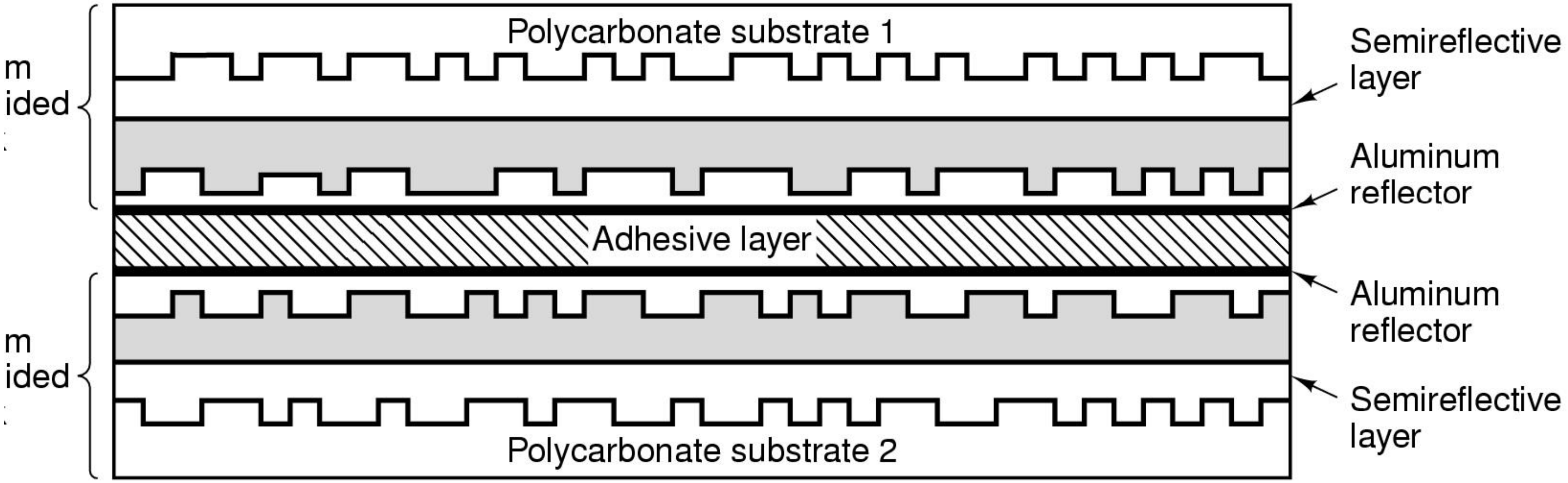
Logical data layout on a CD-ROM

CD-ROM (3)



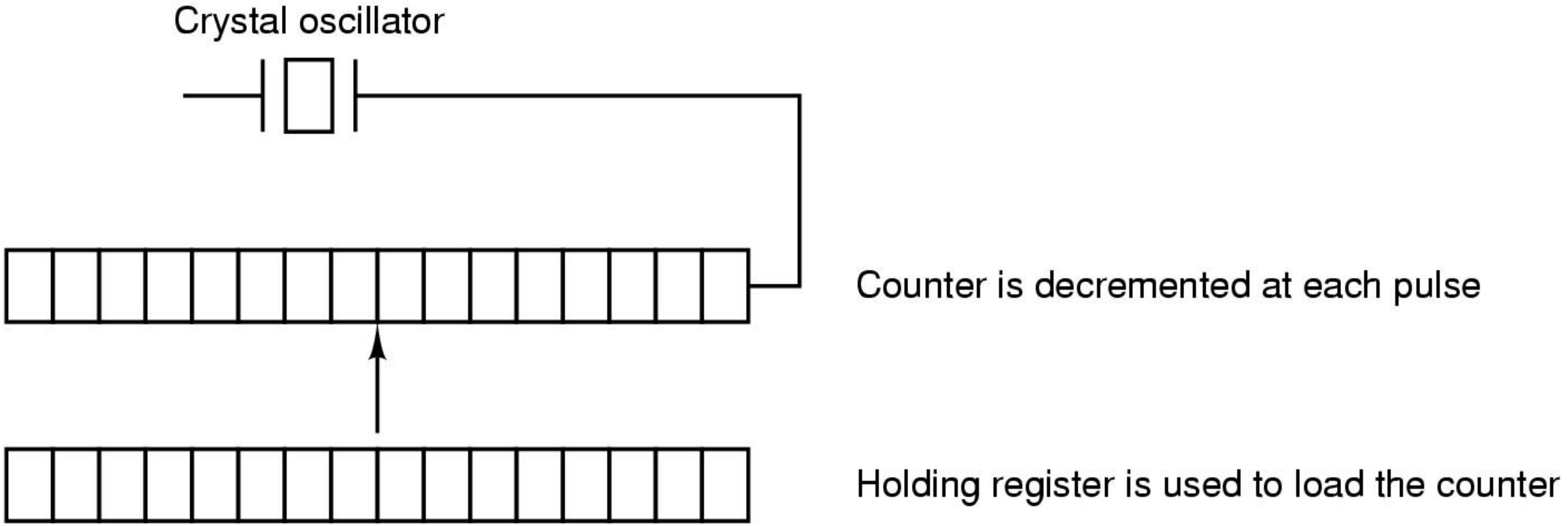
- Cross section of a CD-R disk and laser (not to scale)
- Silver CD-ROM has similar structure
  - without dye layer
  - with pitted aluminum layer instead of gold

DVD-Disk



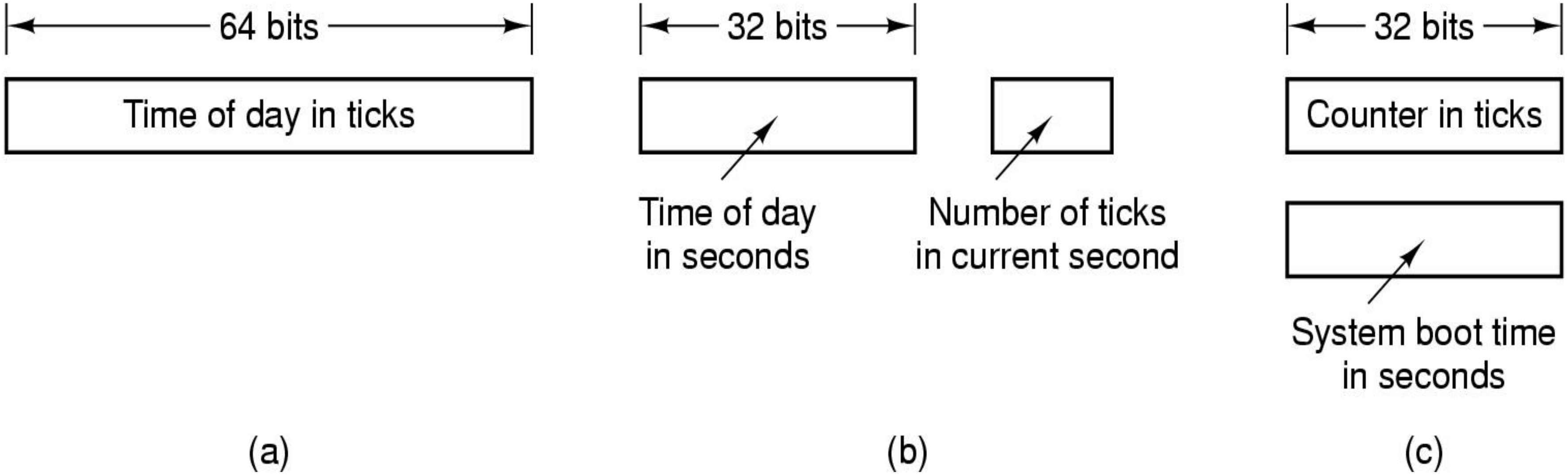
A double sided, dual layer DVD disk

Clocks



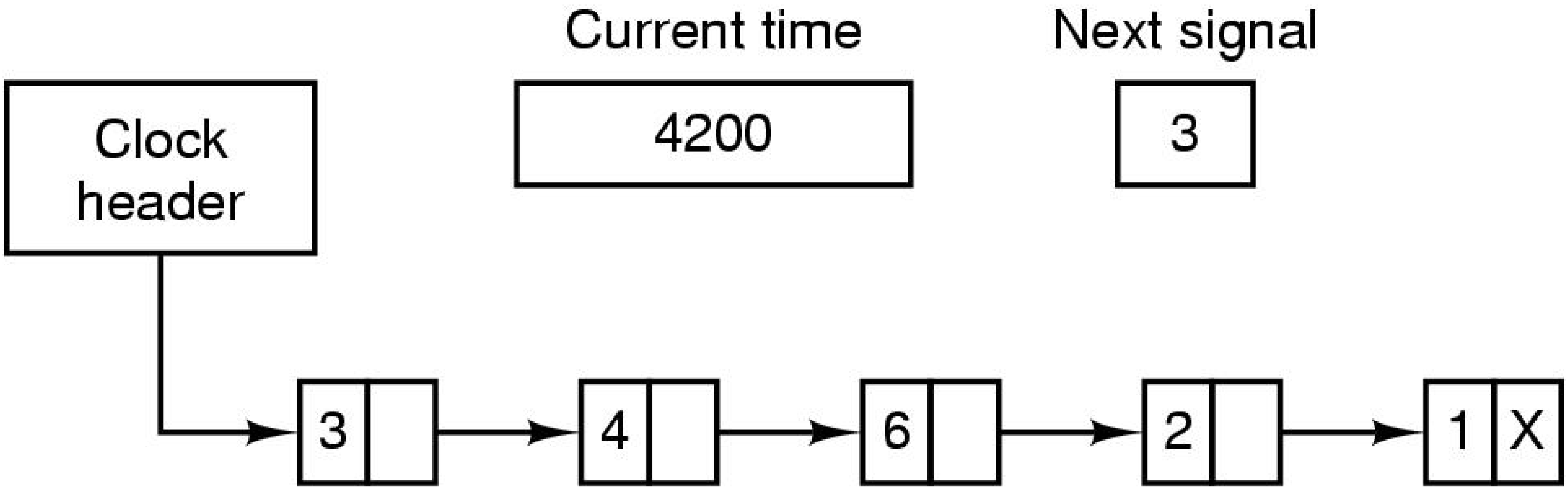
A programmable clock

Clock Software (1)



Three ways to maintain the time of day

Clock Software (2)



Simulating multiple timers with a single clock

Soft Timers

- A second clock available for timer interrupts
  - specified by applications
  - no problems if interrupt frequency is low

- Soft timers avoid interrupts
  - kernel always checks for soft timer expiration before kernel exits to user mode
  - how well this works depends on rate of kernel entries

Recommended Reading

Alessandro Rubini: Linux Device Drivers, O'Reilly 2001

P. Chen et al.: RAID: High Performance,
Reliable Secondary Storage,
ACM Computing Surveys, 1994

D. Patterson et al.: Computer Organization and Design,
Morgan Kaufmann, 1998