



Networking

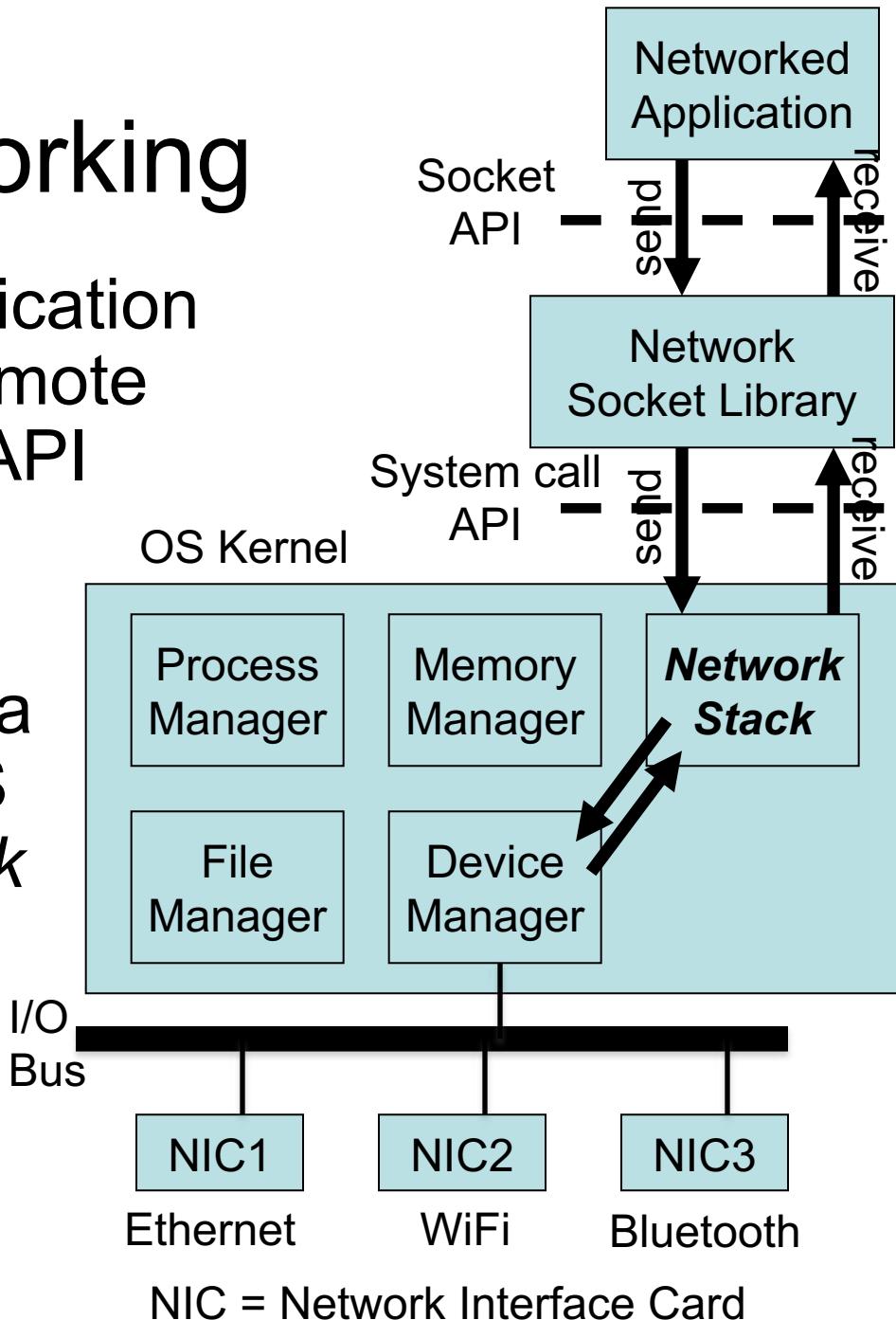
Networking

- Applications today leverage the Internet to send and receive data
 - Web browser requests pages from a Web server, e.g. Web search
 - P2P systems
 - Streaming video
 - Social networks
 - Mobile apps



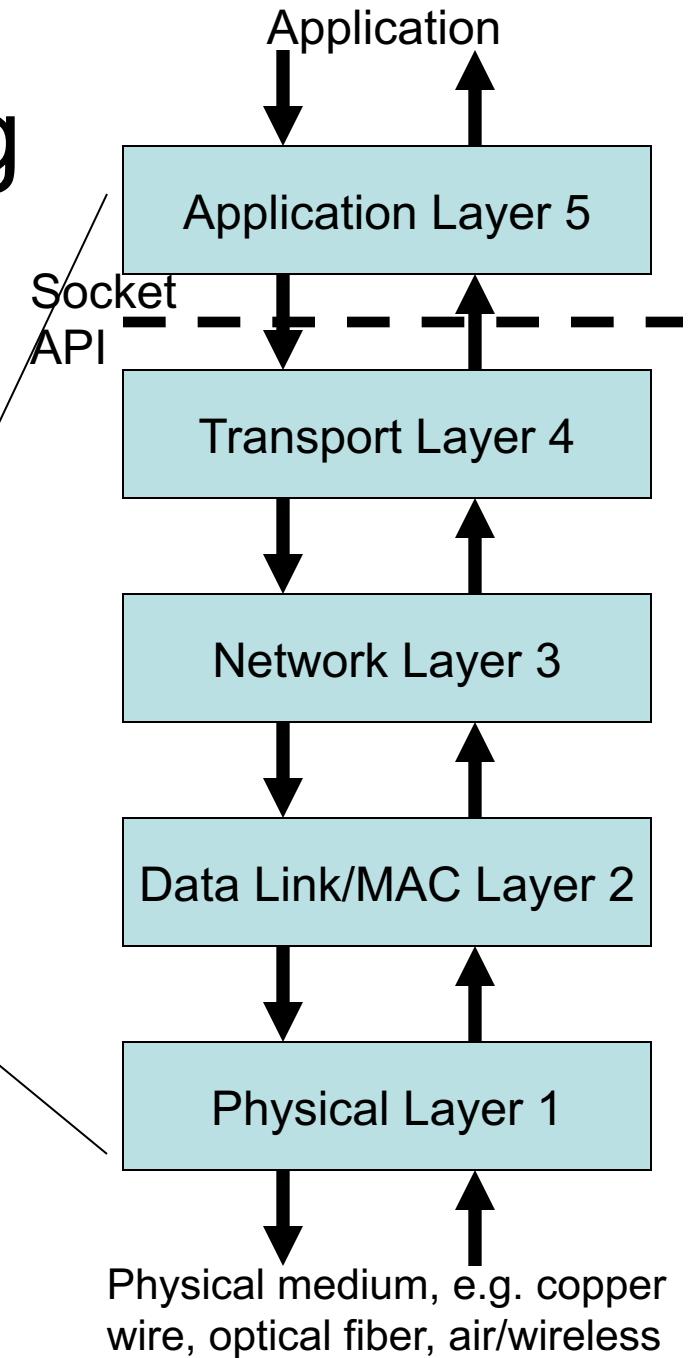
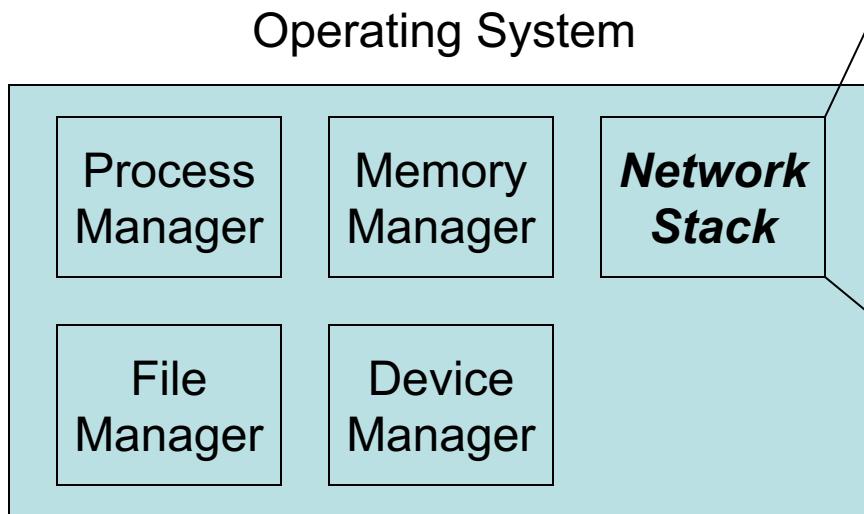
Networking

- Every networked application communicates to a remote process via a socket API
 - Send(message)
 - Receive(message)
- Socket library talks via system call API to OS kernel's *network stack*
 - Send(message)
 - Receive(message)



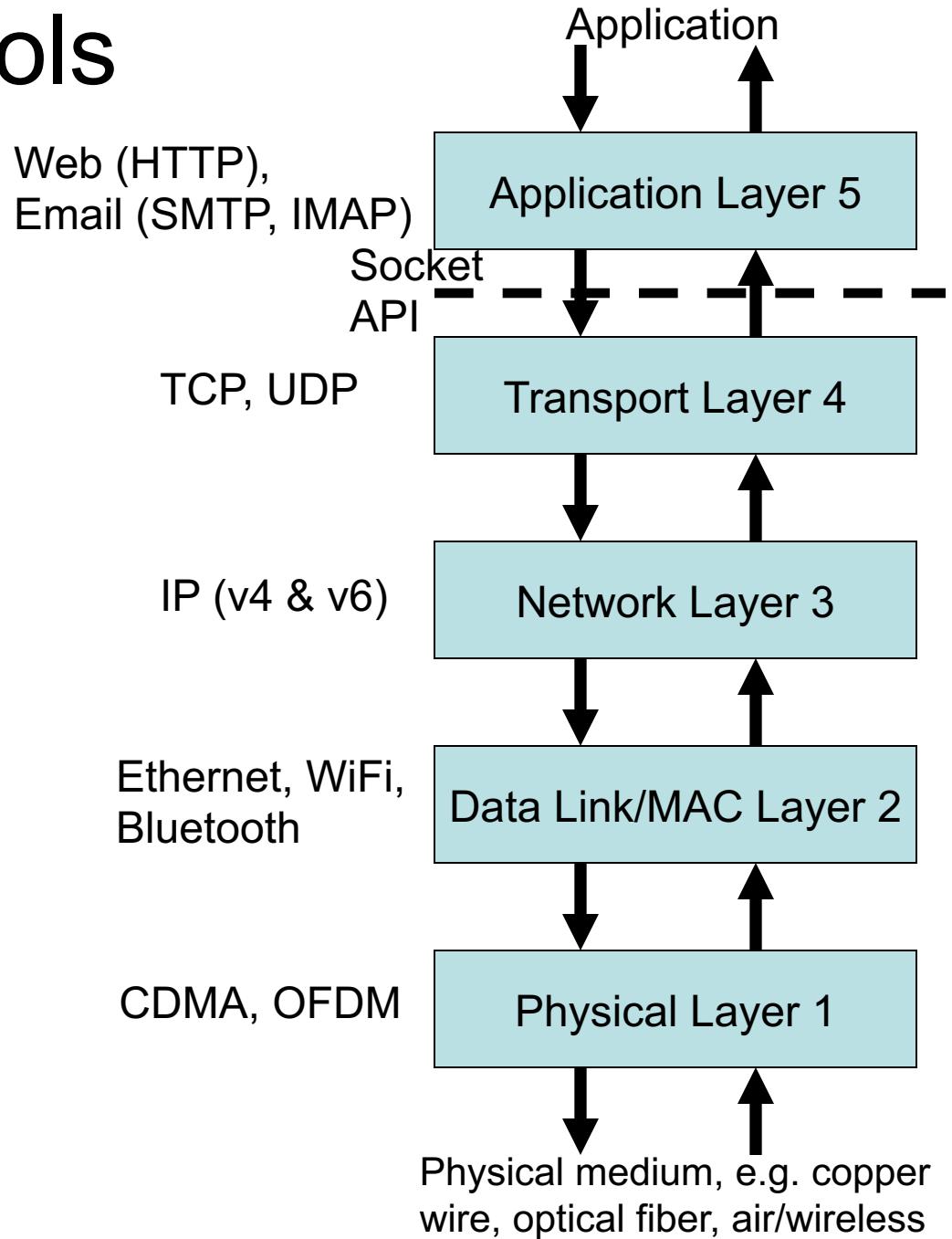
Networking

- The network stack's architecture is organized into multiple layers of protocols
 - Each protocol performs a specific set of duties



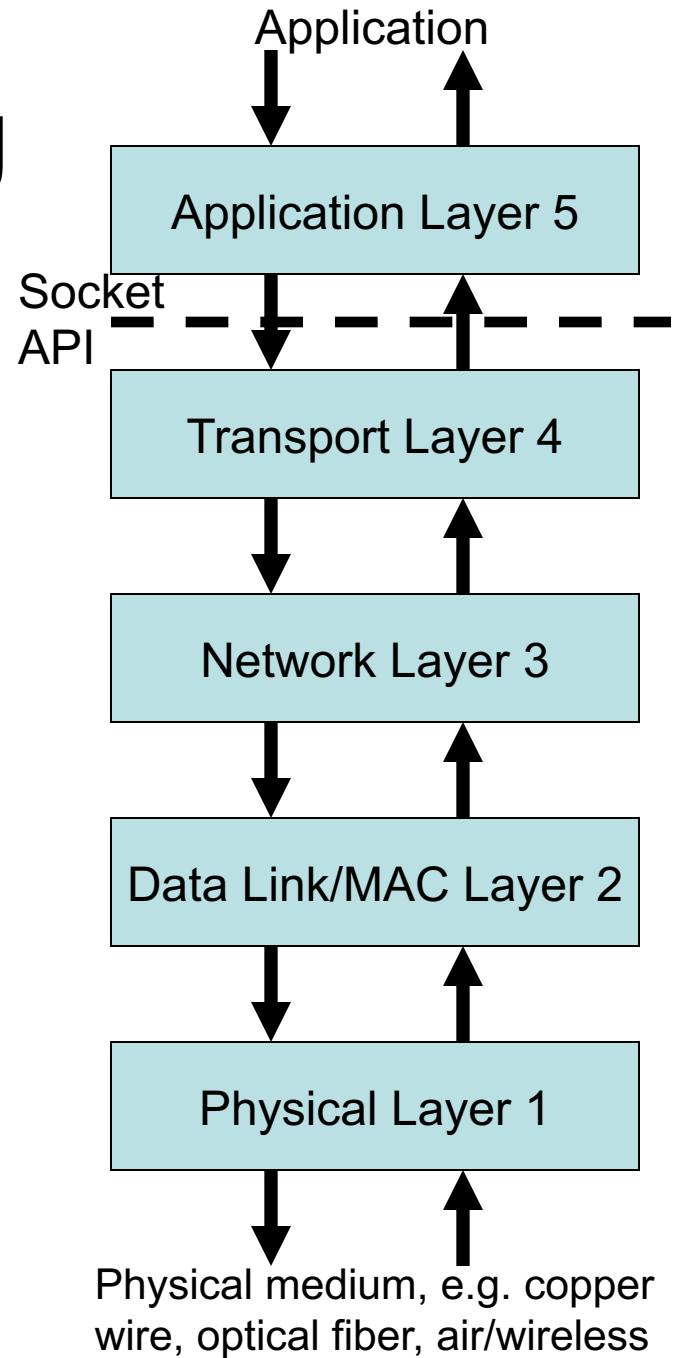
Network Protocols

- Examples of standard network protocols and where they reside in the network stack:



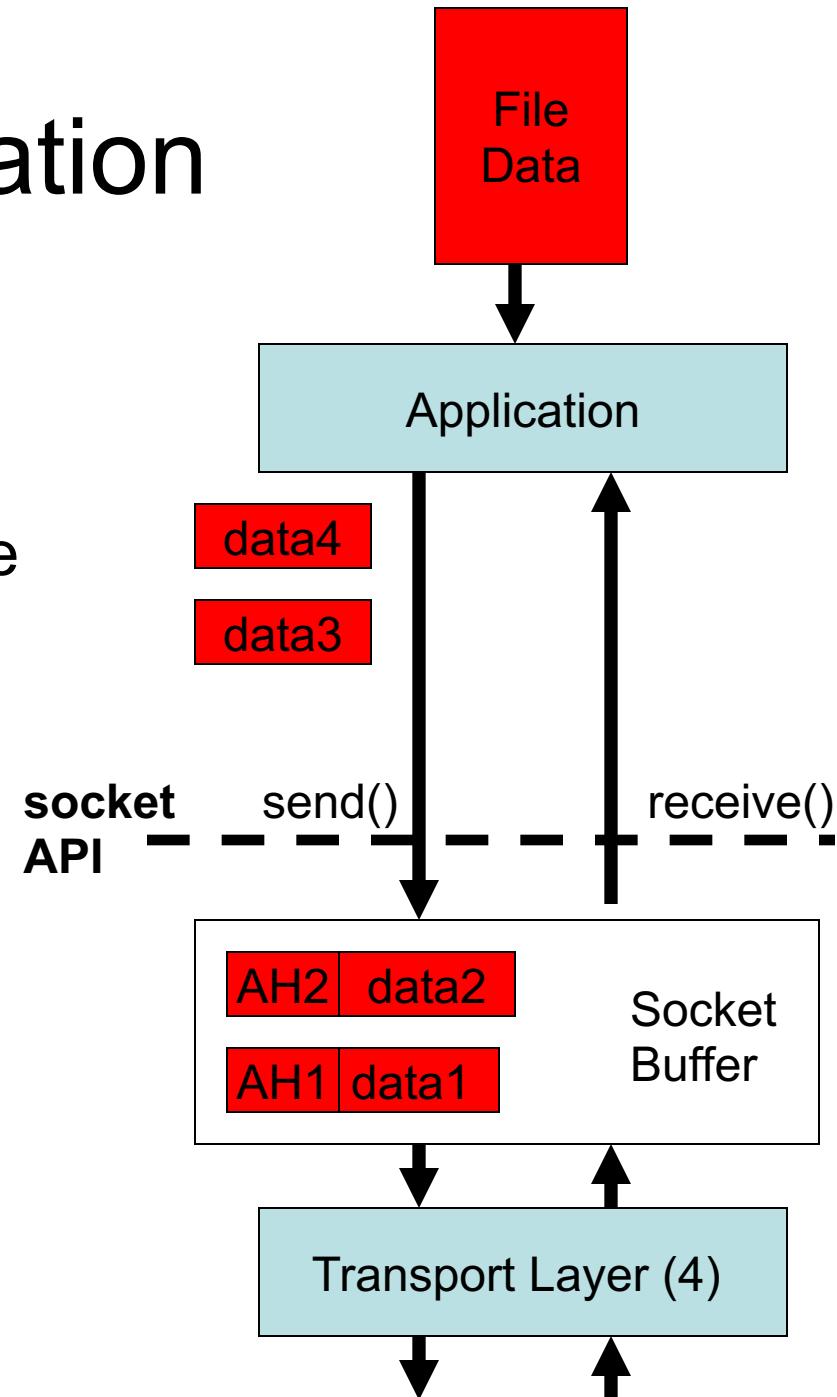
Networking

- to send a packet of data to a remote destination,
 - each layer first passes a packet of data down the stack to the next lowest layer
- to receive a packet of data,
 - each layer retrieves a packet of data from the layer below
 - and after processing the packet sends the packet to the layer above



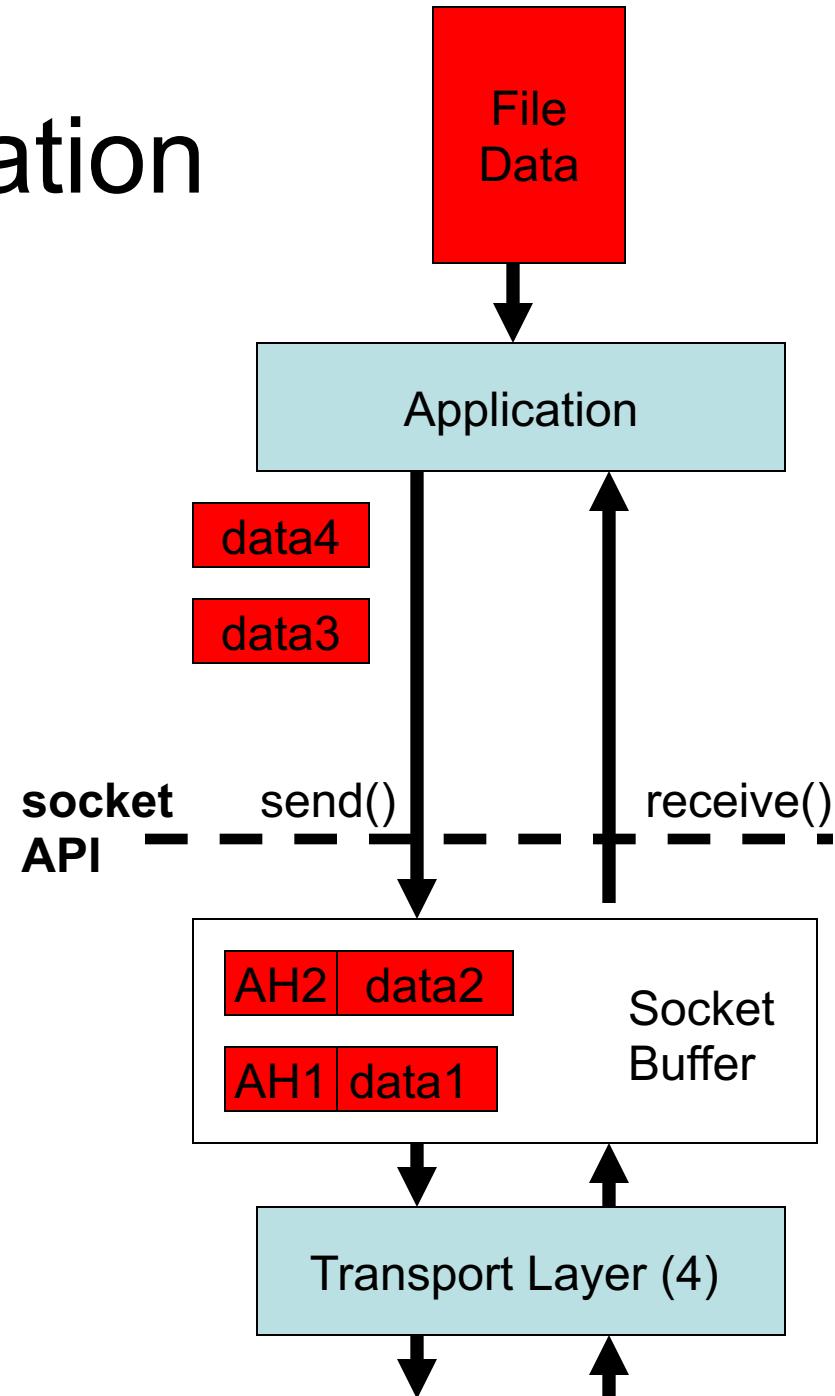
Packetization

- To send, an application calls socket API's send()
 - gives a pointer to the user space buffer containing the data to send
- If the file is large, the application segments the file into smaller packets
 - e.g. a 1 GB file is chopped into 1 KB packets



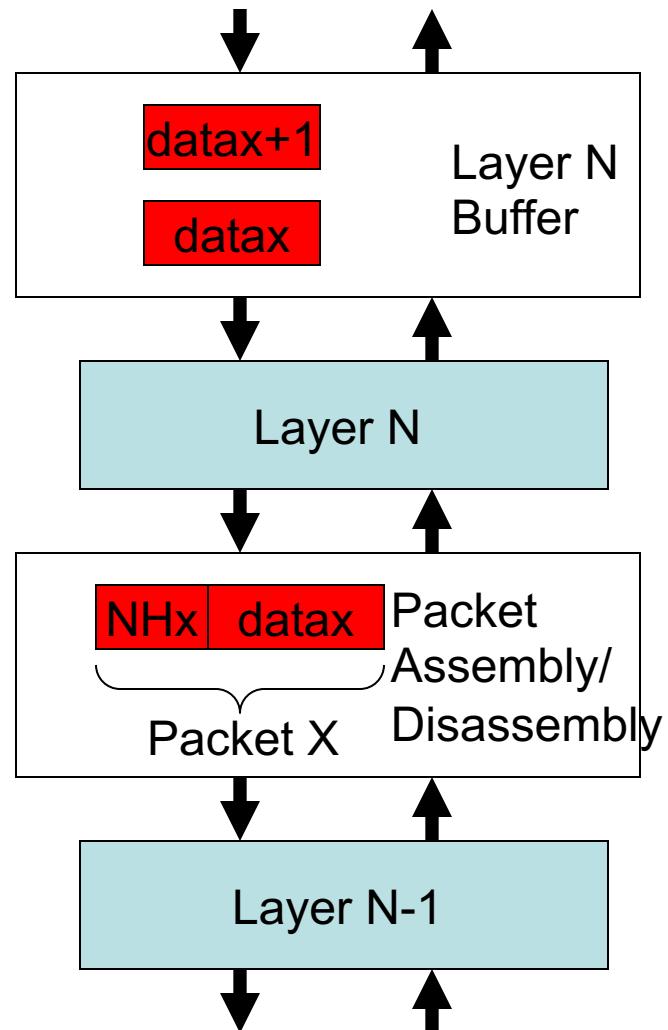
Packetization

- Application layer prepends a layer 5 header to the user data, forming a packet
 - Prepend the header AH1 to data1, forming packet 1
 - Header info is useful at the remote receiver to decode the packet
- Here, packets 1 & 2 are sent down to transport layer 4

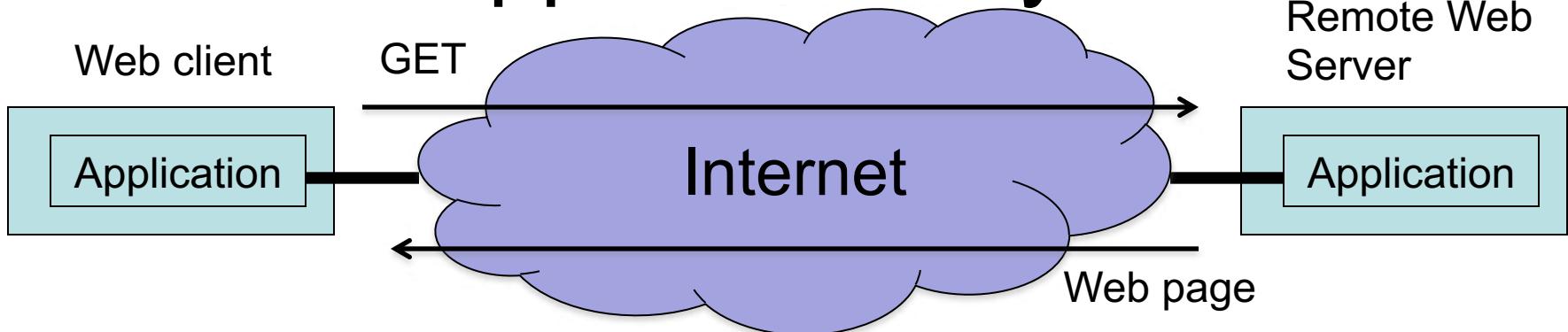


Packetization

- When sending a packet
 - In general, at each layer N, a packet header NH_x is prepended to data x and then sent to a lower layer N-1
 - Packet grows as it descends the network layered stack
- When receiving a packet
 - At each layer N, strip off the layer N header
 - Packet shrinks as it moves up the stack

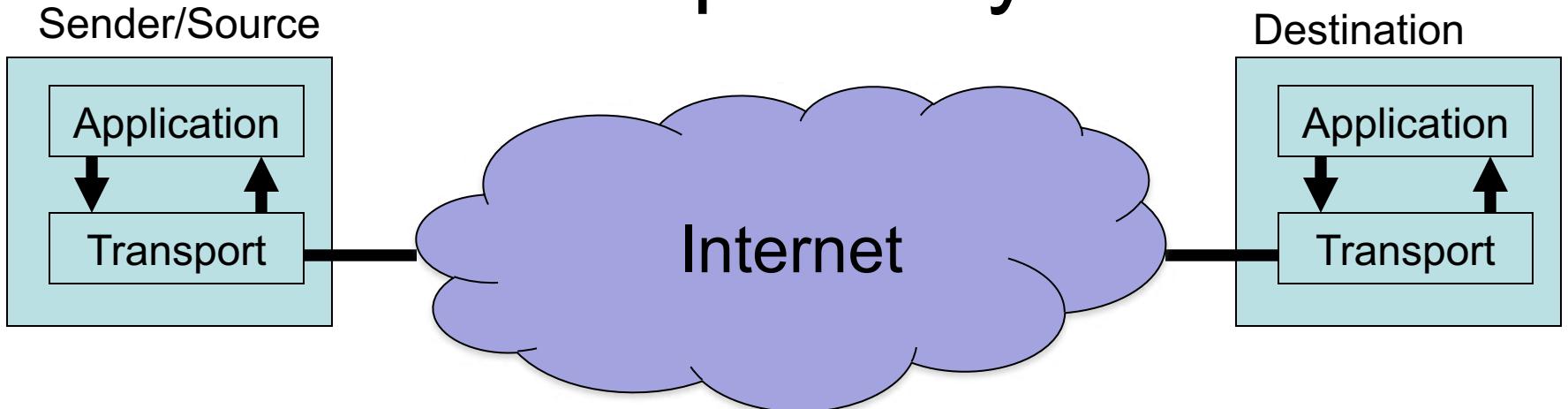


Application Layer



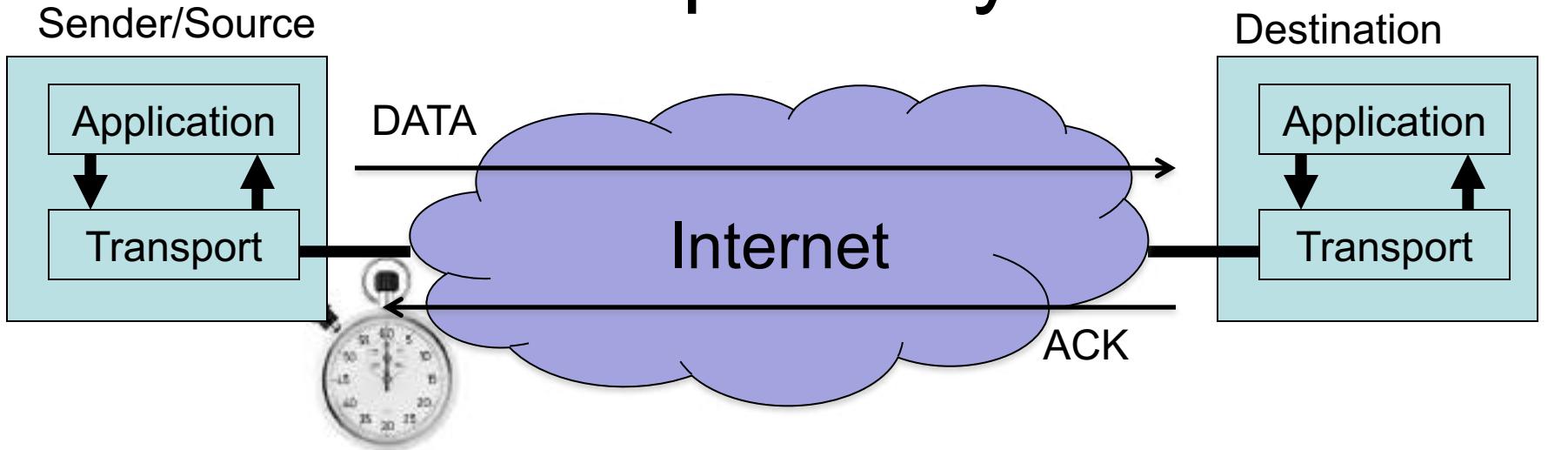
- Let us ignore the lower layers temporarily & focus only on layer 5
- Application layer 5 sender communicates *application-specific* information with its peer layer 5 receiver
 - e.g. Web (HTTP) client sends a GET request (at layer 5) to fetch a Web page from the remote Web server

Transport Layer



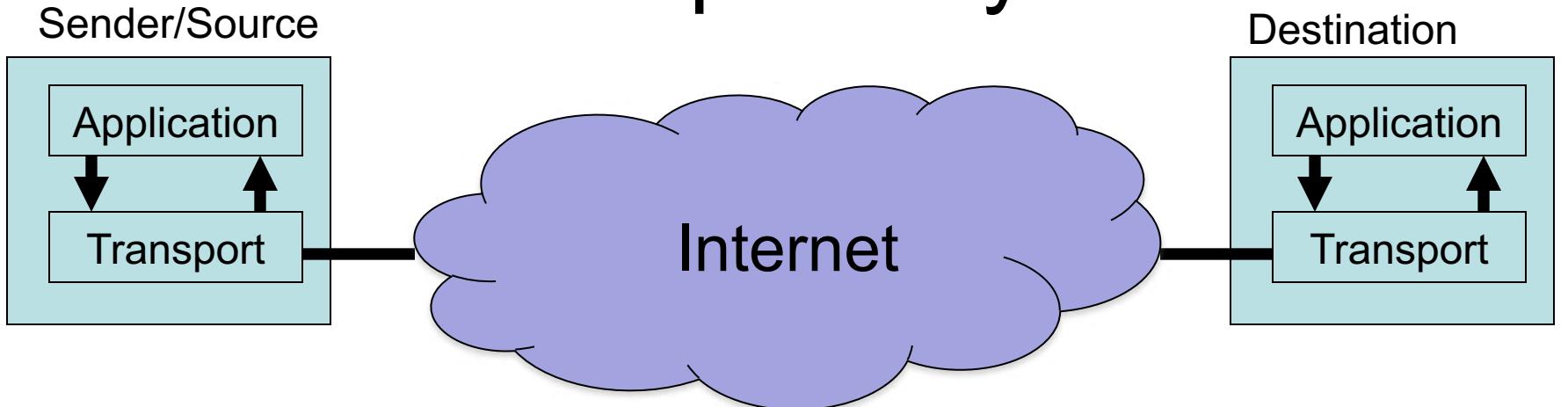
- The Internet can lose the application's message!
- The transport layer's job is end-to-end error recovery, if desired.
- How to recover from a lost packet?
 - *Retransmit* lost packets! This is TCP, the Transmission Control Protocol

Transport Layer



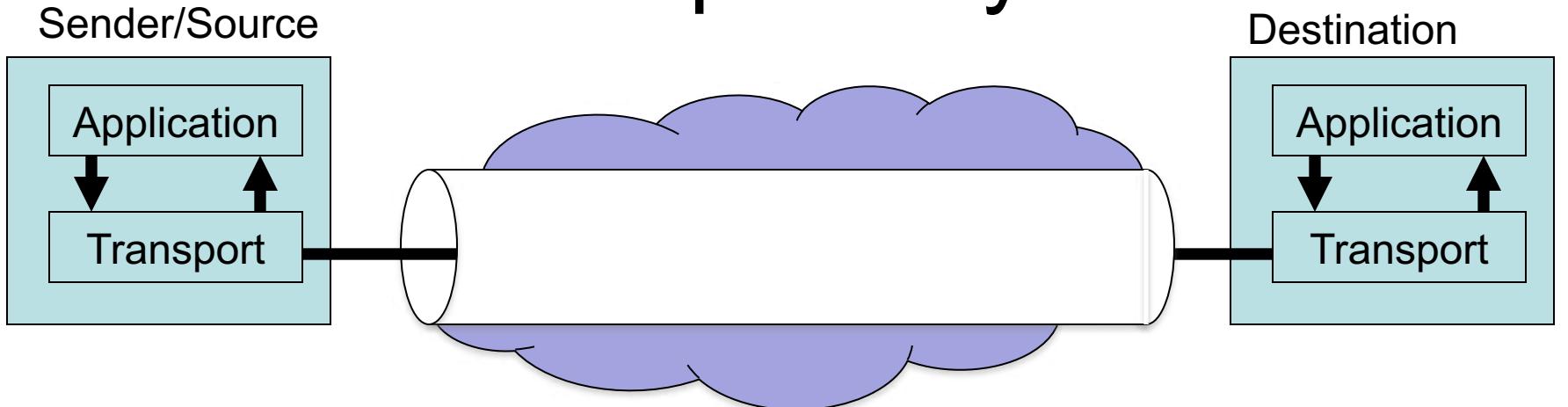
- How does the sender know if a packet was received correctly?
 - Receiver sends an *Acknowledgment* (ACK) packet back to sender
- When does sender know when to retransmit?
 - Sets a *timer*. If it *times out* before ACK received, then retransmit

Transport Layer



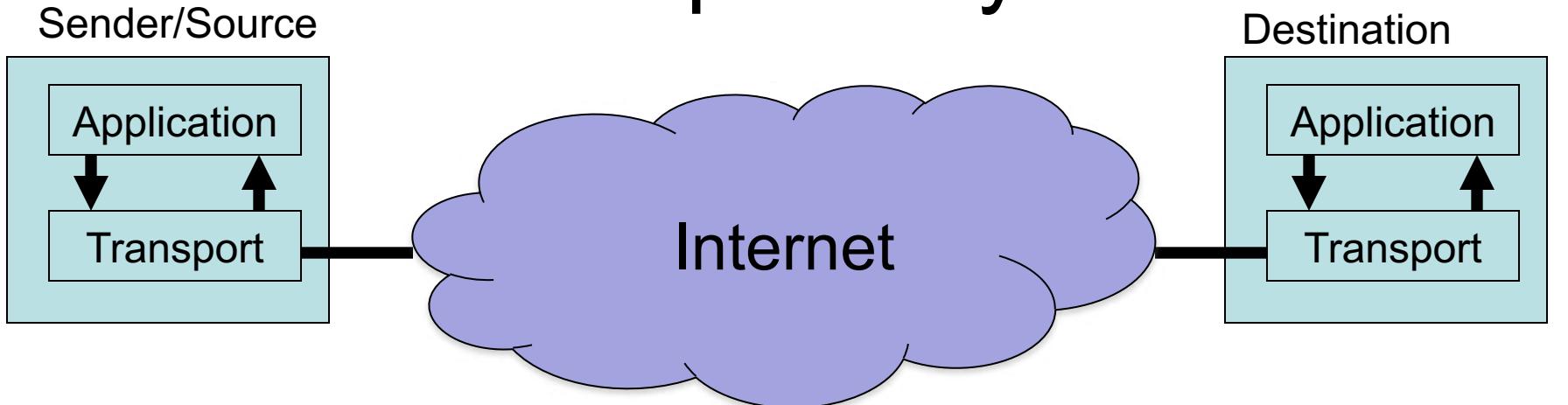
- TCP also ensures in-order delivery
- Many apps require TCP's reliable & in-order packet delivery service
 - Web, email, etc. - can't render a Web page or read email if there are holes in the Web page or email
 - Changing order of Web/email text also makes it unreadable

Transport Layer



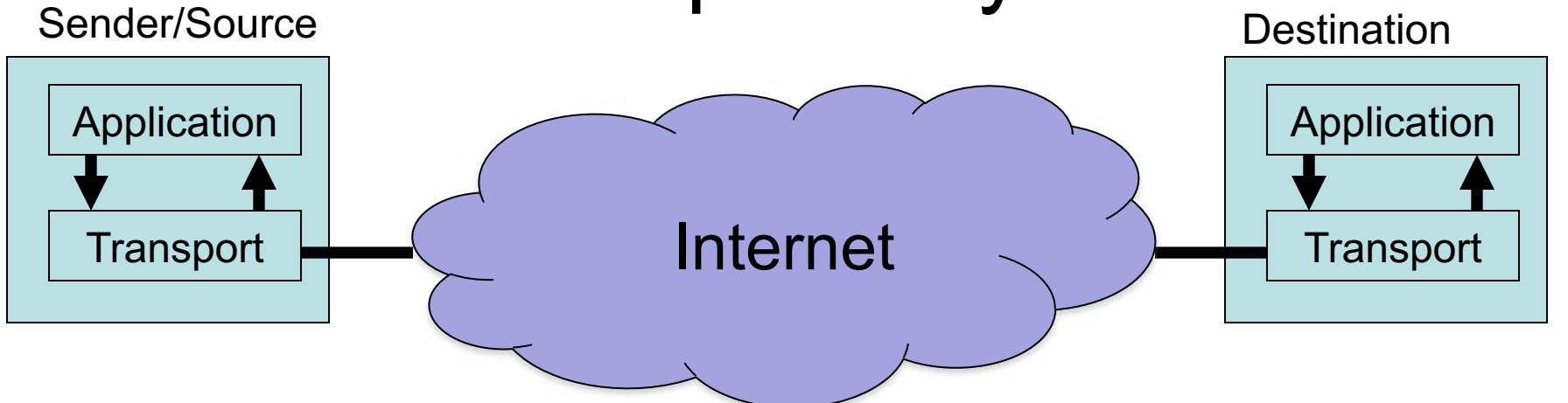
- Apps that use TCP can view the network connection as a pipe abstraction
 - Any data sent into the pipe appears at the other end, hence it is reliable, i.e. pipes don't lose data
 - A pipe preserves the order of the data sent into it at the output of the pipe – no reordering is possible

Transport Layer



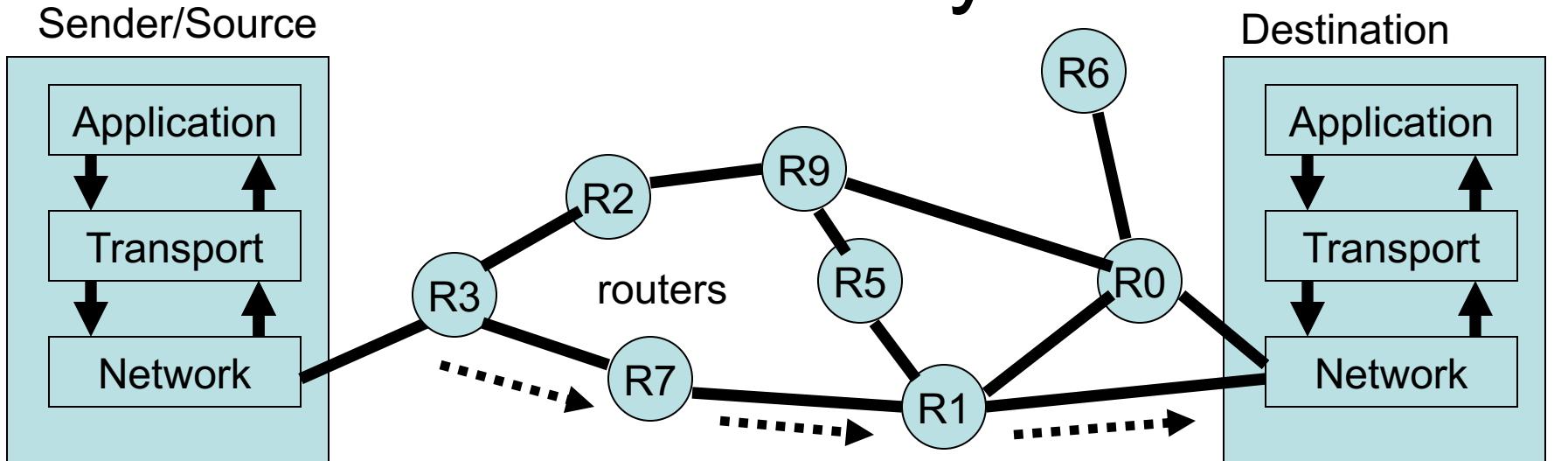
- Reliability comes at the cost of delay due to retransmissions
- Not all apps need/want TCP's reliability
 - Interactive real-time apps like Skype audio/video conferencing can't wait for TCP's retransmissions
 - Must get packet delivered in real time, e.g. within 30 ms

Transport Layer



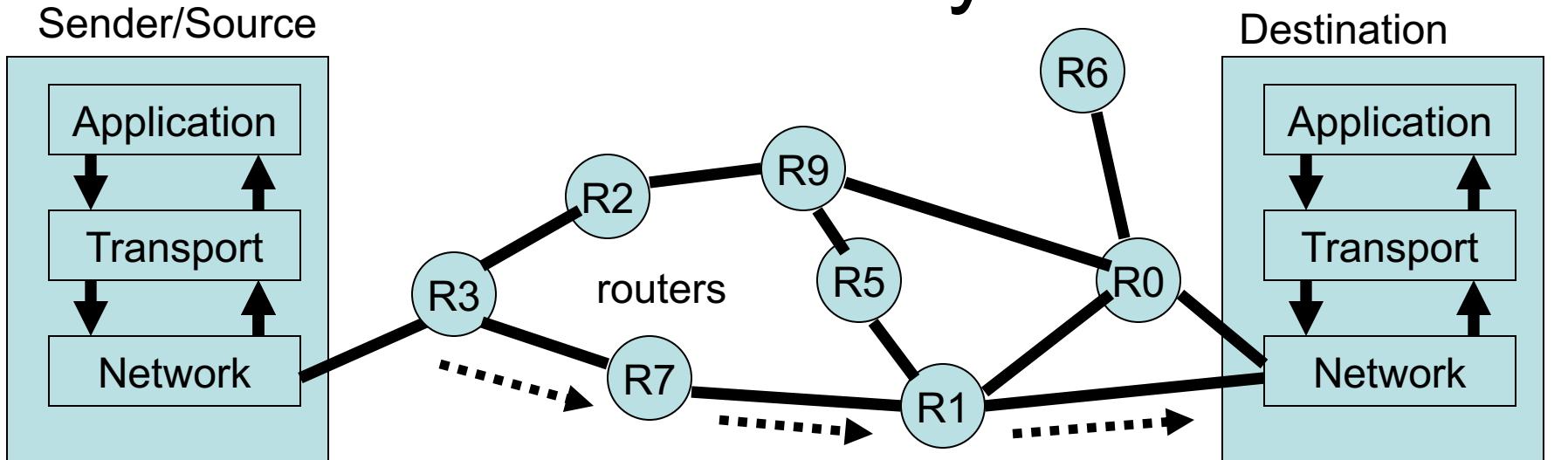
- Real-time Voice-over-IP (VOIP) apps like Skype & FaceTime can tolerate packet loss
 - may lose audio temporarily, but it's OK
- Such apps are built on top of unreliable UDP (User Datagram Protocol) at layer 4, not TCP

Network Layer



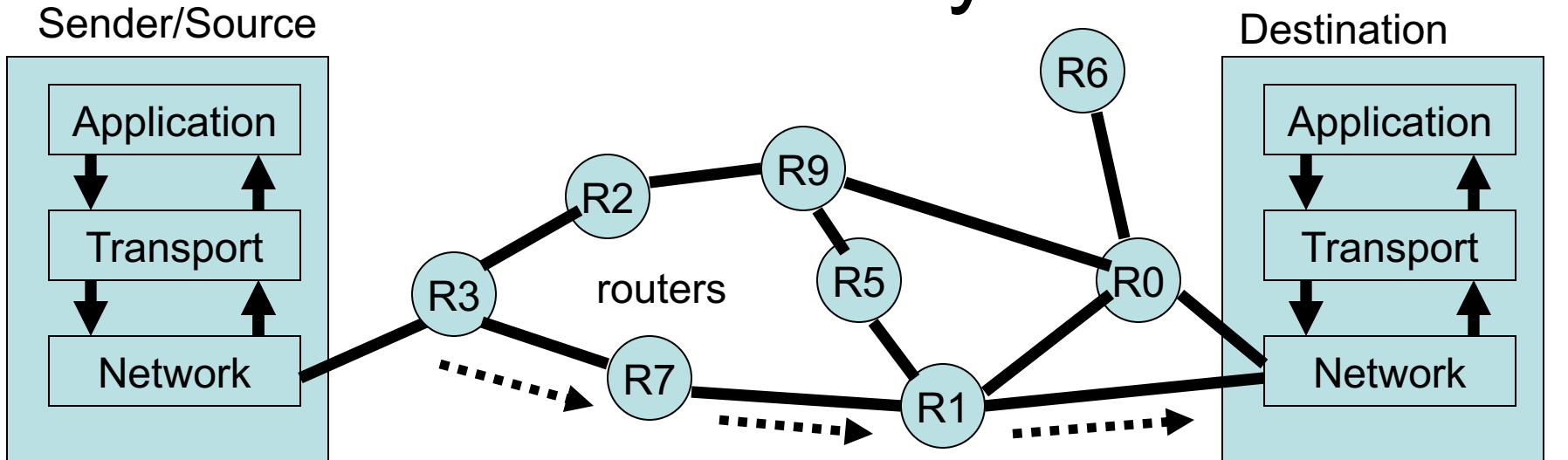
- The Internet consists of many routers that connect together to form a network graph
- The Internet Protocol (IP) network layer must route the IP packet to the correct destination
 - But there are many routes! Which one is best?

Network Layer



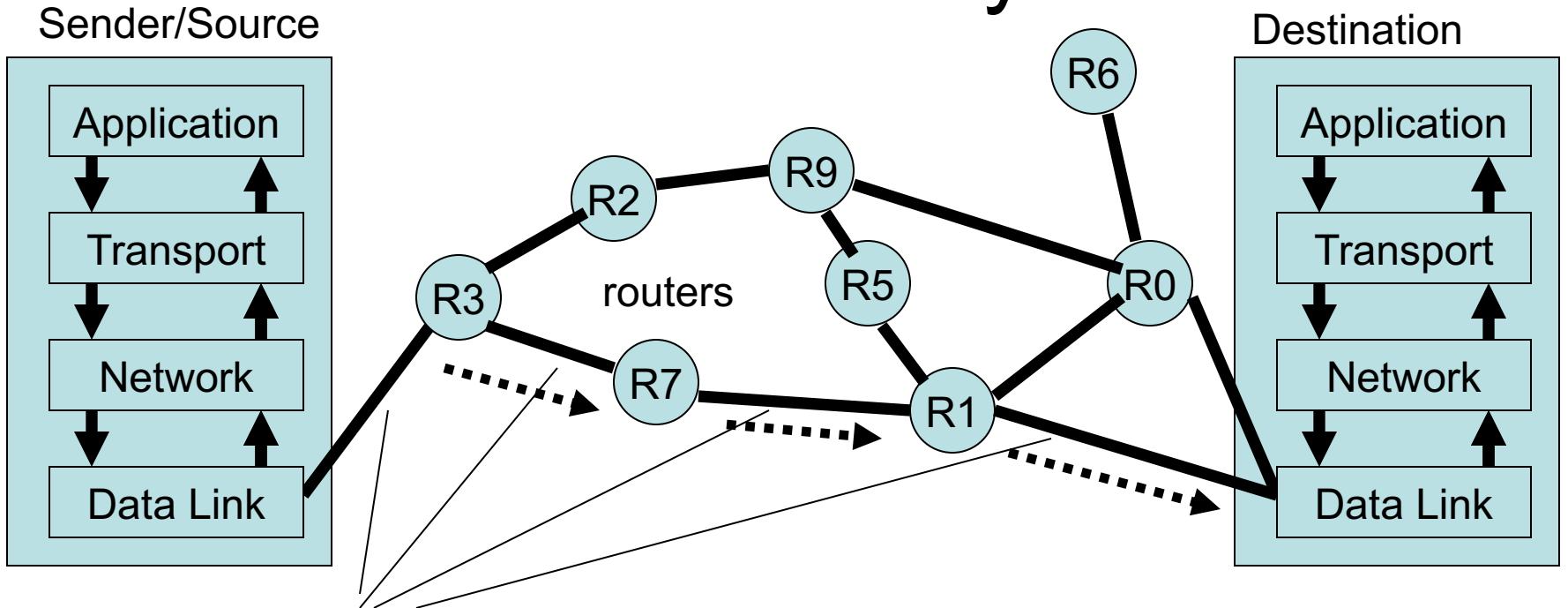
- The network layer tries to find the “shortest path” route, e.g. using Dijkstra’s algorithm
 - The metric for shortest path may be minimum # of hops, shortest physical distance, lowest delay, minimum cost, etc.

Network Layer



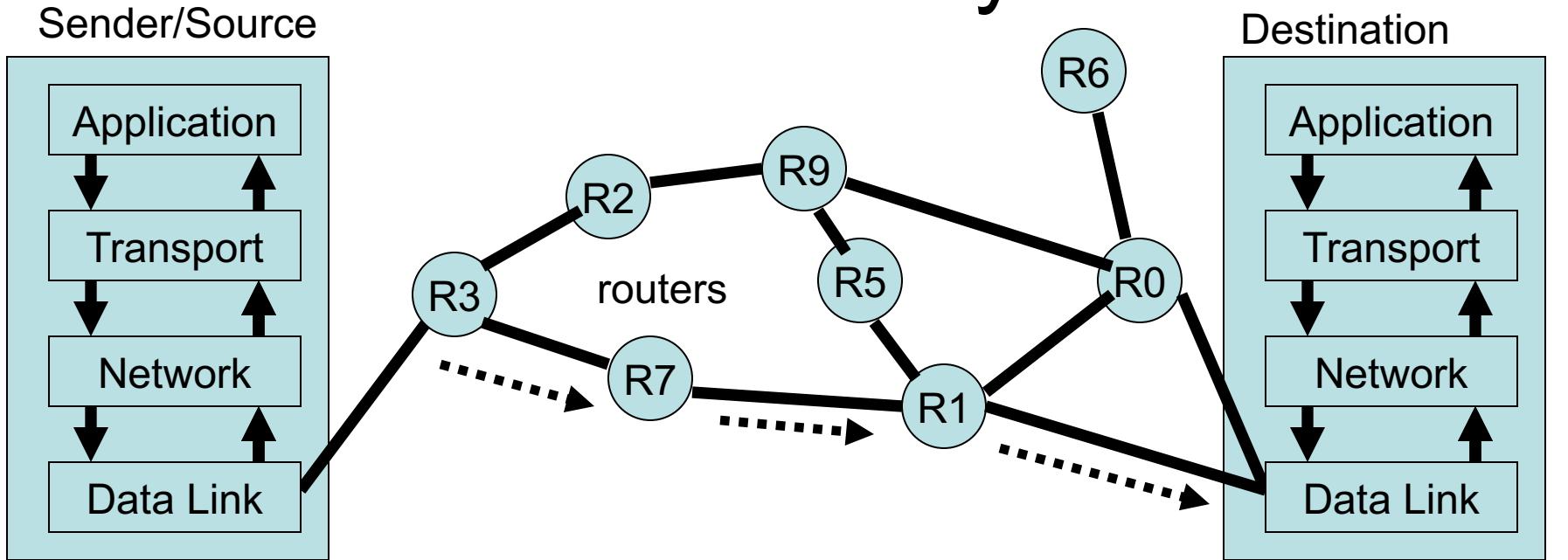
- Each router implements the network layer
- IP routing may lose packets!
 - Any router or link may fail at any time. Also congested router buffers may overflow.
 - That's OK, as long as TCP can retransmit them!

Data Link Layer



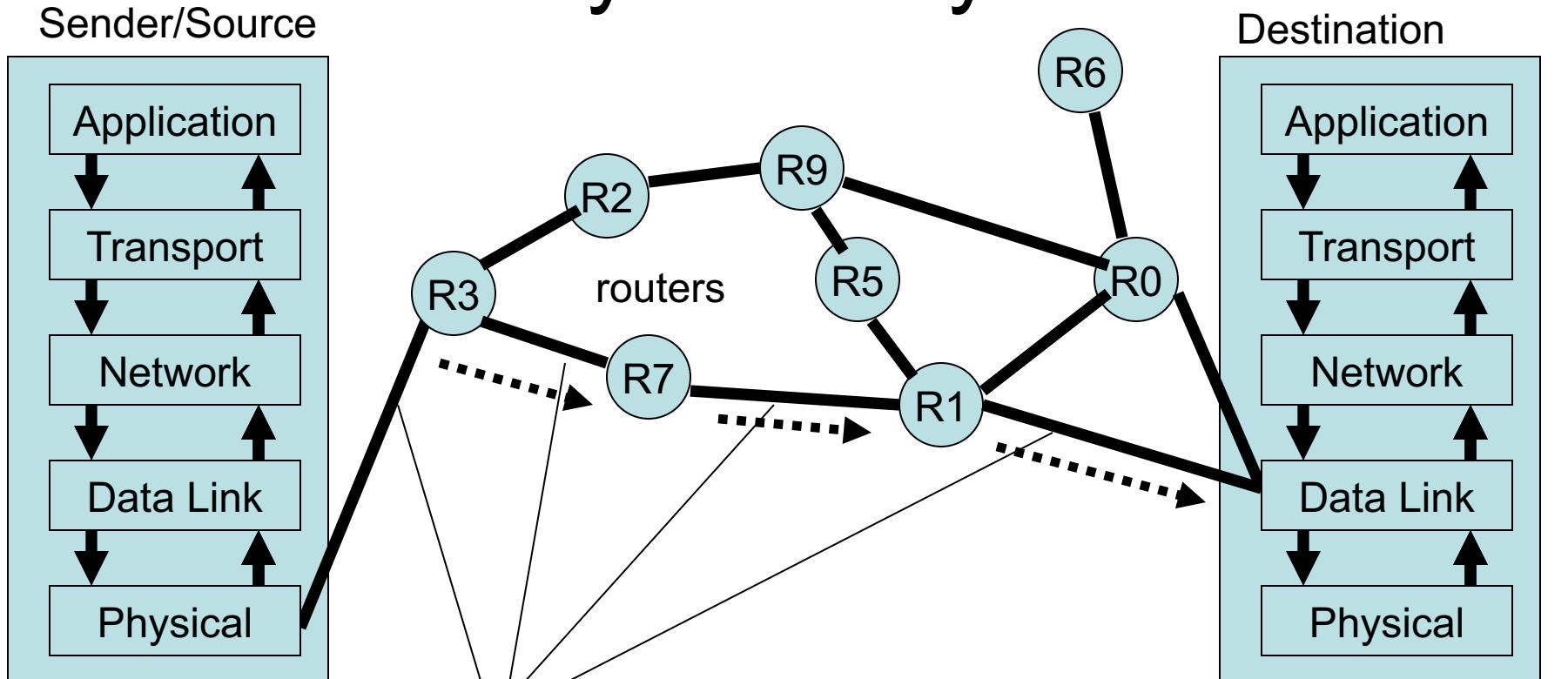
- Each link between any two routers (also endpoints) must be able to transmit packets
 - Data link layer is responsible for transmitting packets between any 2 neighboring nodes in the network

Data Link Layer



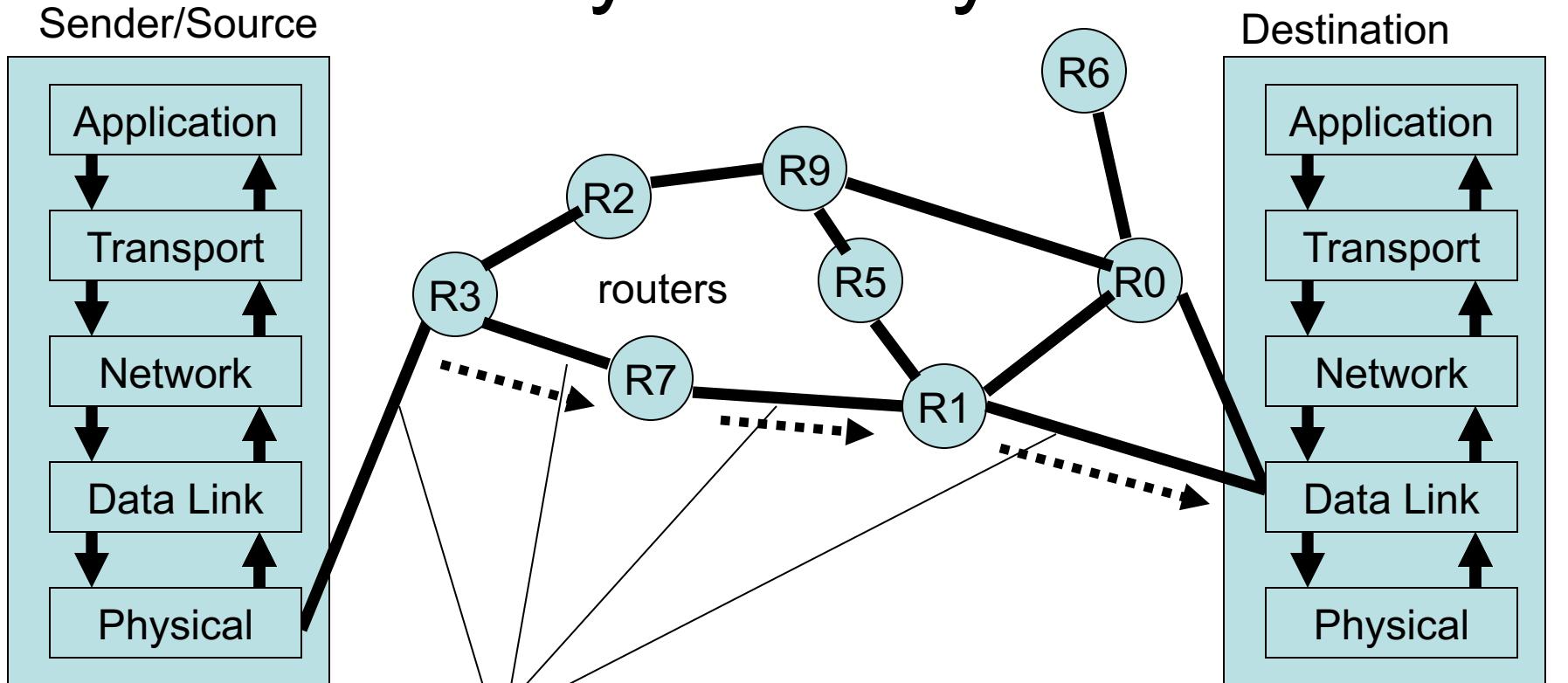
- This layer must define the beginning and end of packets, i.e. packet framing
- Packets may be lost, so this layer may also retransmit locally
- Examples: Ethernet, WiFi, Bluetooth, ...

Physical Layer



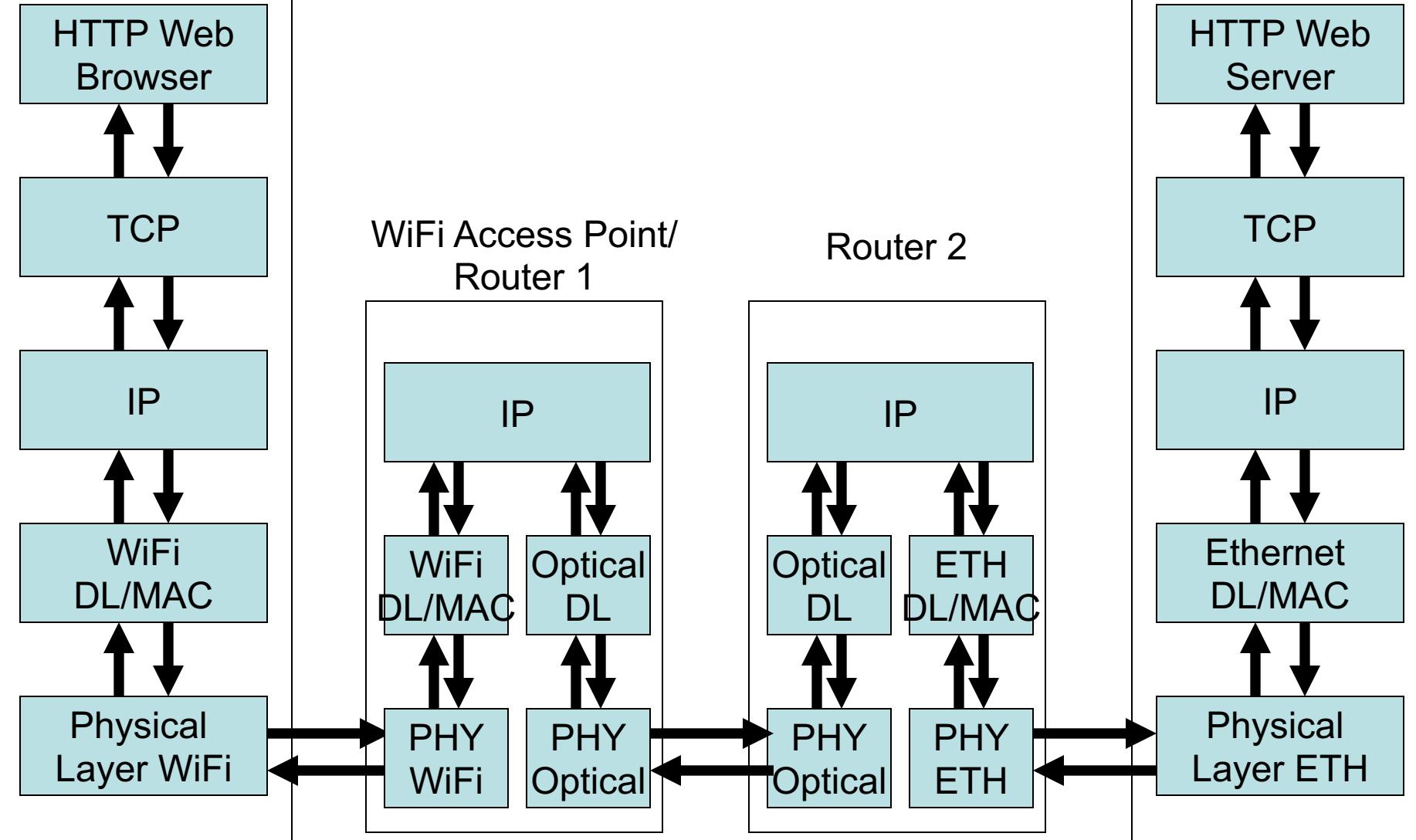
- Along each link, the physical layer determines how 1's and 0's, i.e. digital bits, are transmitted

Physical Layer



- Example: a '1' may be +5 volts, and a '0' may be 0 volts. Or 1s & 0s may correspond to different frequencies.

An Example Network

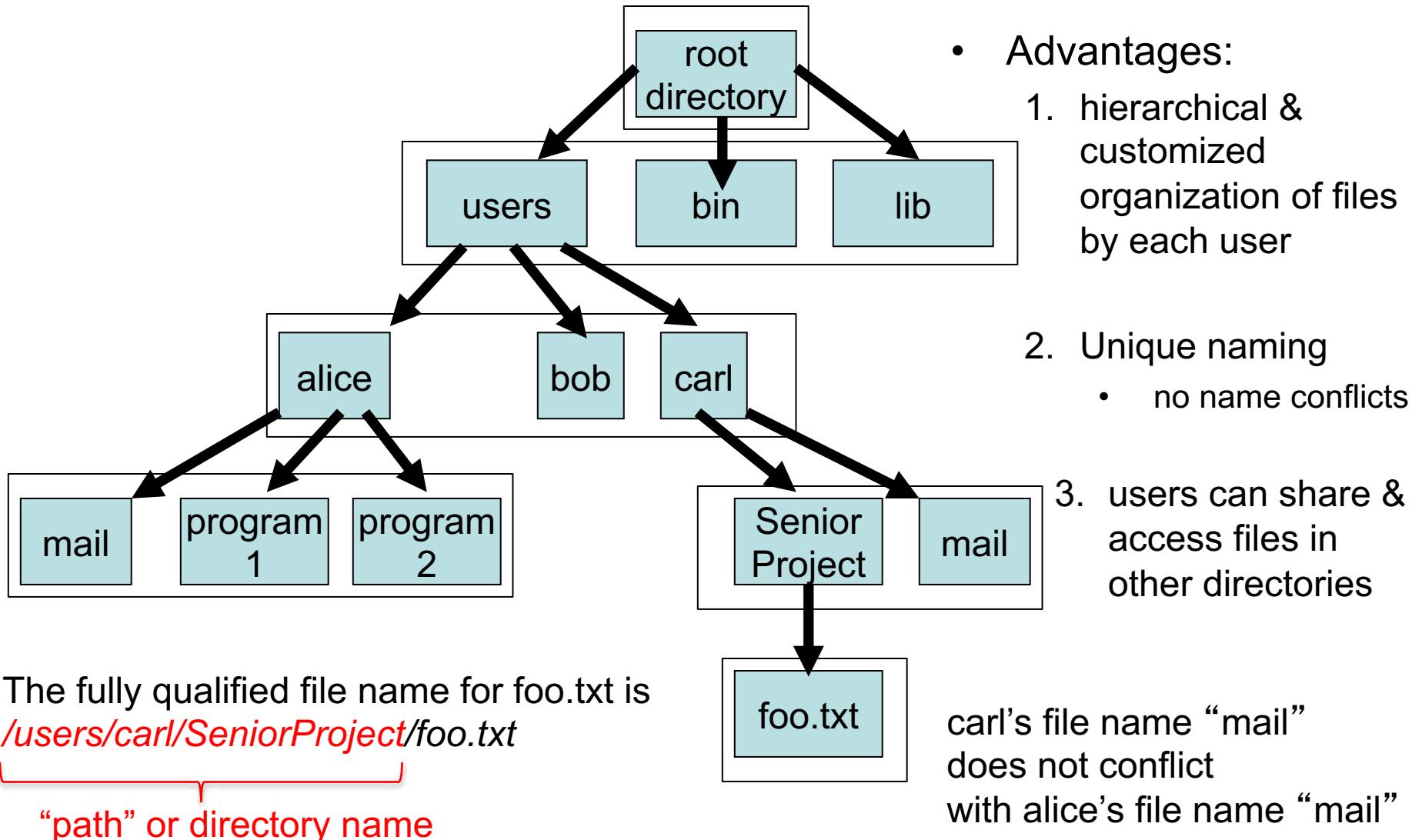




Lecture 26

File System Implementation

Tree-structured Directory



File System Implementation

- File system elements are stored on *both*:
 - Disk/flash – persistent storage
 - Main memory/RAM – volatile storage
- On *disk/flash*, **the entire file system is stored**, including 5 main elements:
 1. its entire directory tree structure
 2. each file's attributes are in a File Control Block →
 3. each file's data
 4. a *boot block*, typically the first block of a volume, that contains info needed to boot an operating system from this volume. Empty if no OS to boot.
 5. a *volume control block* that contains volume or partition details,
e.g. tracks free blocks on disk,
the number of blocks in a partition,
size of a block, etc.

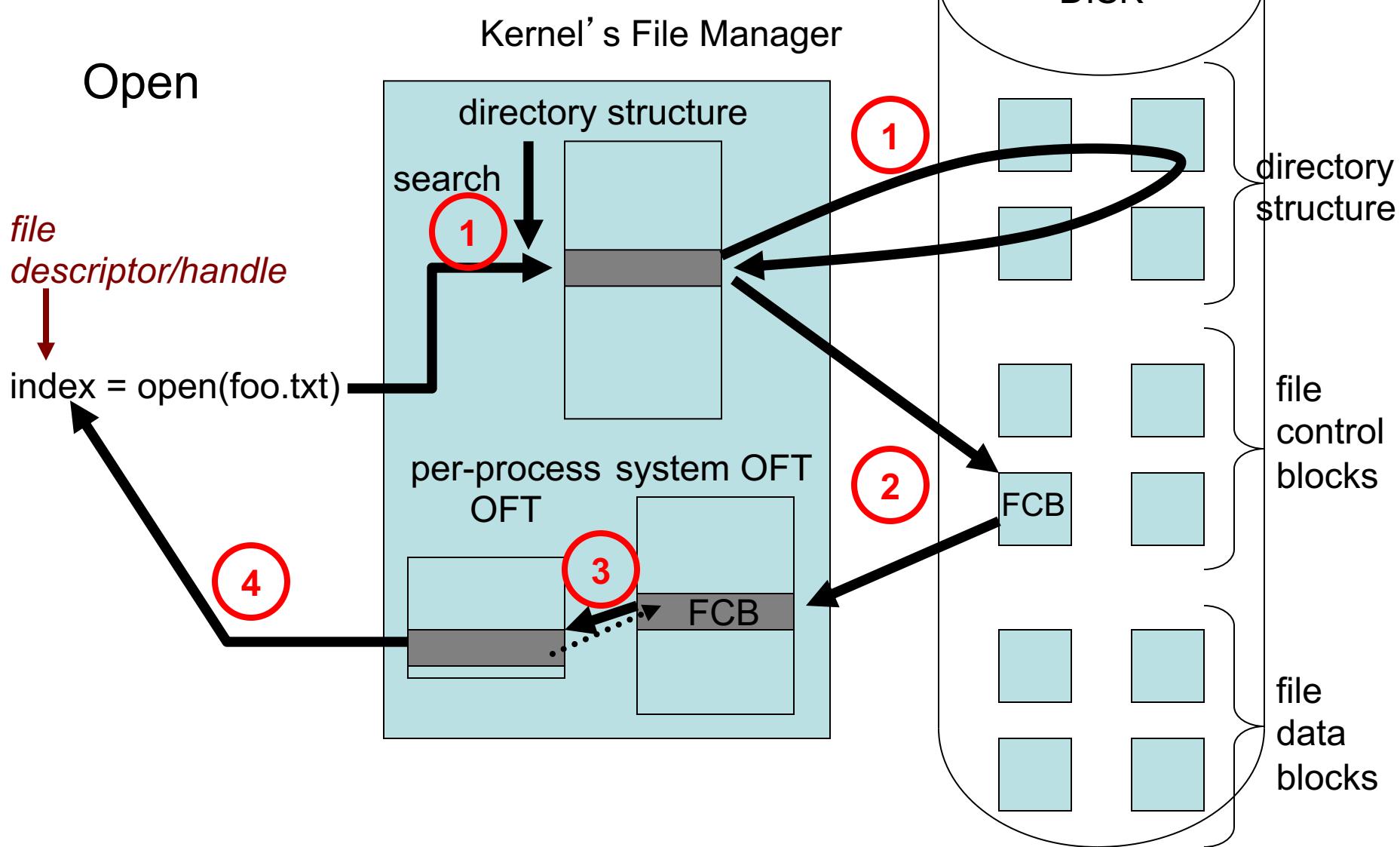
example FCB

name
unique ID
file permissions
dates (created,...)
size
location on disk

The four main file system components in *memory* are:

1. Recently accessed parts of the directory structure tree are stored in memory – for faster look up
2. OS also maintains a **system-wide open file table (OFT)** that tracks process-independent info of open files
 - the file header containing attributes about the open file is stored here
 - an open count of the number of processes that have files open is stored here
3. OS also maintains a **per-process OFT** - tracks all files that have been opened by a particular process, may store access rights, etc.
 - Also keeps a *current-file-position pointer*, i.e. where in the file the process is currently reading/writing
4. OS keeps a **mount table of devices with file systems** that have been mounted as volumes

File System Open



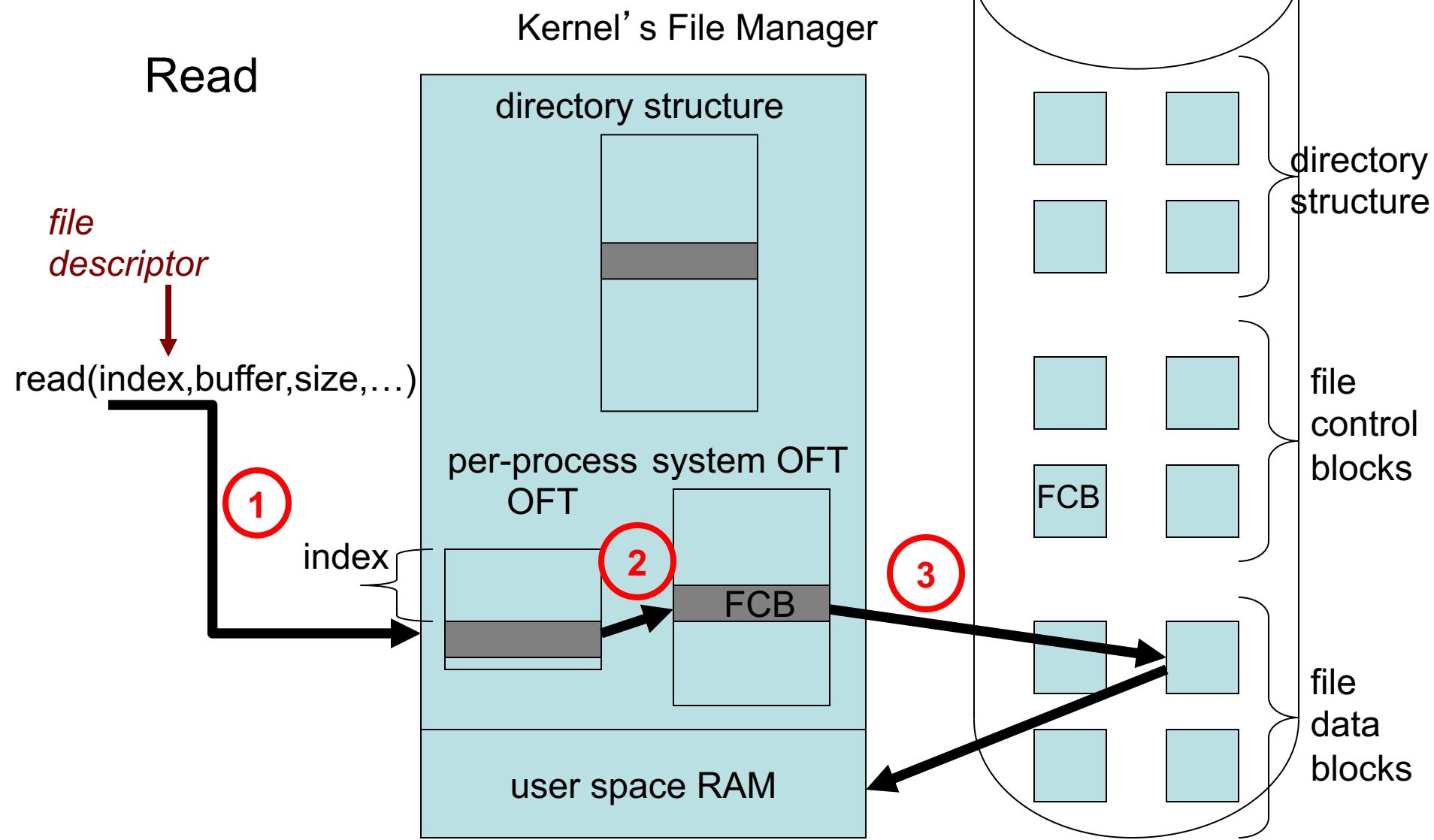
File System Open (steps)

- When a process calls `open(foo.txt)` to set up access to a file, the following procedural steps are followed:
 1. the directory structure is searched for the file name `foo.txt`
 - if the directory entries are in memory, then the search is fast
 - otherwise, directories and directory entries have to be retrieved from disk and cached for later accesses
 2. once the file name is found, the directory entry contains a pointer to the FCB on disk
 - retrieve the FCB from disk
 - copy the FCB into the system OFT. This acts as a cache for future file opens.
 - Increment the open file counter for this file in the system OFT
 3. add an entry to the per-process OFT that points to the file's FCB in the system OFT
 4. return a file descriptor or handle to the process that called `open()`

File System Open

- Some OS's employ a mandatory lock on an open file
 - Only one process at a time can use an open file
 - Windows policy
- Other OS's allow optional or advisory locks
 - UNIX policy
 - It's up to users to synchronize access to files

File System Read



File System Close

- on a close(),
 1. remove the entry from the per-process OFT
 2. decrement the open file counter for this file in the system OFT
 3. if counter = 0, then write back to disk any metadata changes to the FCB, e.g. its modification date
 - Note: there may be a temporary inconsistency between the FCB stored in memory and the FCB on disk – designers of file systems need to be aware of this. A similar inconsistency occurred for modified memory-mapped file data in RAM that had not yet been written to disk.



File Allocation

File Allocation

Approaches:

1. Contiguous file allocation

- a file is laid out contiguously, i.e. if a file is n blocks long, then a starting address b is selected and the file is allocated blocks $b, b+1, b+2, \dots, b+n-1$

2. Linked Allocation

- each file is a linked list of disk blocks

3. File Allocation Table (FAT) is an important variation of linked lists

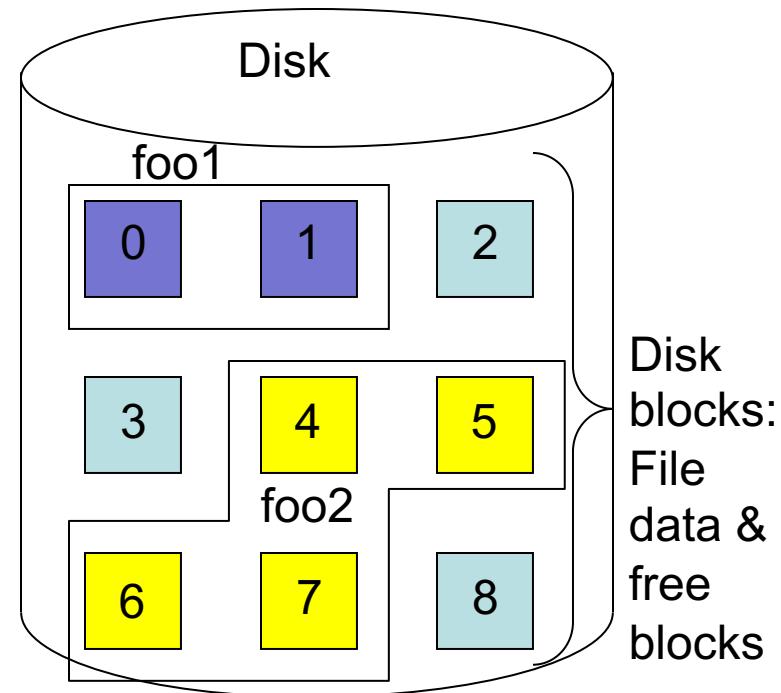
- Don't embed the pointers of the linked list with the file data blocks themselves
- Instead, separate the pointers out and put them in a special table – the file allocation table (FAT)
- The FAT is located at a section of disk at the beginning of a volume

4. Indexed Allocation

- collect all pointers into a list or table called an *index block*
- the index j into the list or index block retrieves a pointer to the j 'th block on disk

Approach #1: Contiguous File Allocation

File headers		
file	start	length
foo1	0	2
foo2	4	4



- Advantage: fast performance (low seek times because the blocks are all allocated near each other on disk)

Approach #1: Contiguous File Allocation

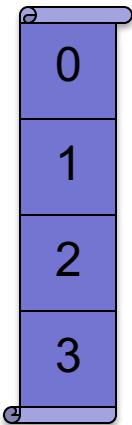
- Disadvantages:
 - Problem 1: external fragmentation (same problem as trying to contiguously fit processes into RAM)
 - same solutions apply: first fit, best fit, etc.
 - also compact memory/defragment disk
 - can be performed in the background late at night, etc.
 - Problem 2: May not know size of file in advance
 - allocate a larger size than estimated
 - if file exceeds allocation, have to copy file to a larger free “hole” between allocated files
 - Problem 3: Over-allocation of a “slow growth” file
 - A file may eventually need 1 million bytes of space
 - But initially, the file doesn’t need much, and it may be growing at a very slow rate, e.g. 1 byte/sec
 - So for much of the lifetime of the file, allocating 1 MB wastes allocation

General File Allocation

- Page table solved external fragmentation problem for process allocation
- Apply a similar concept to file allocation
 - Divide disk into fixed-sized blocks, just as main memory was divided into fixed-sized physical frames
 - Allow a file's data blocks to be spread across any collection of disk blocks, not necessarily contiguous
 - *Need a data structure to keep track of what block of a file is stored on which block in disk*

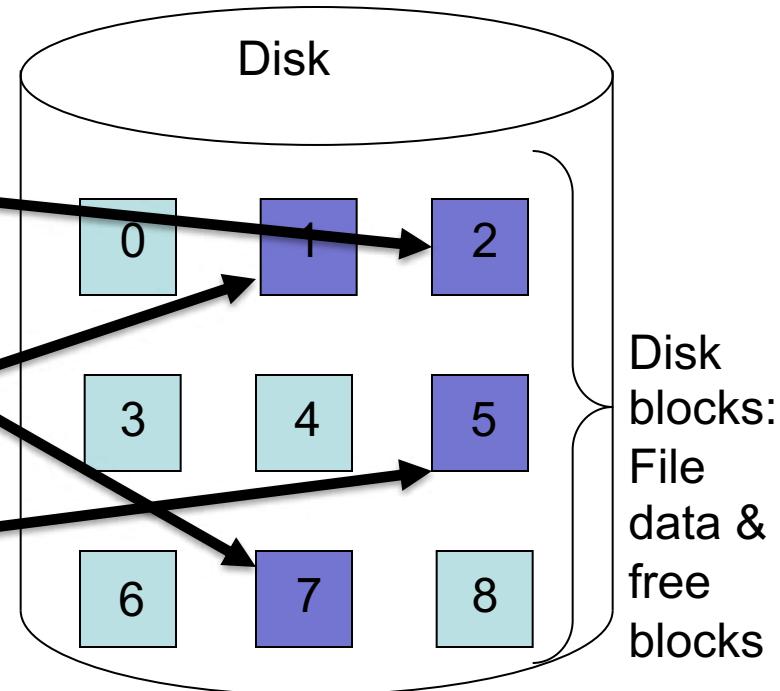
General File Allocation

File "foo1.txt"



Generic Data Structure

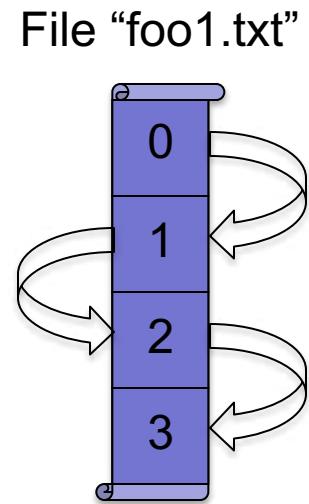
File block 0 is stored at disk block 2,
Next file block 1 is at disk block 7,
Next file block 2 is at disk block 1,
Next file block 3 is at disk block 5



- Generic data structure can be:
 - A Linked list and variants
 - Indexed allocation (somewhat resembles a page table) and variants

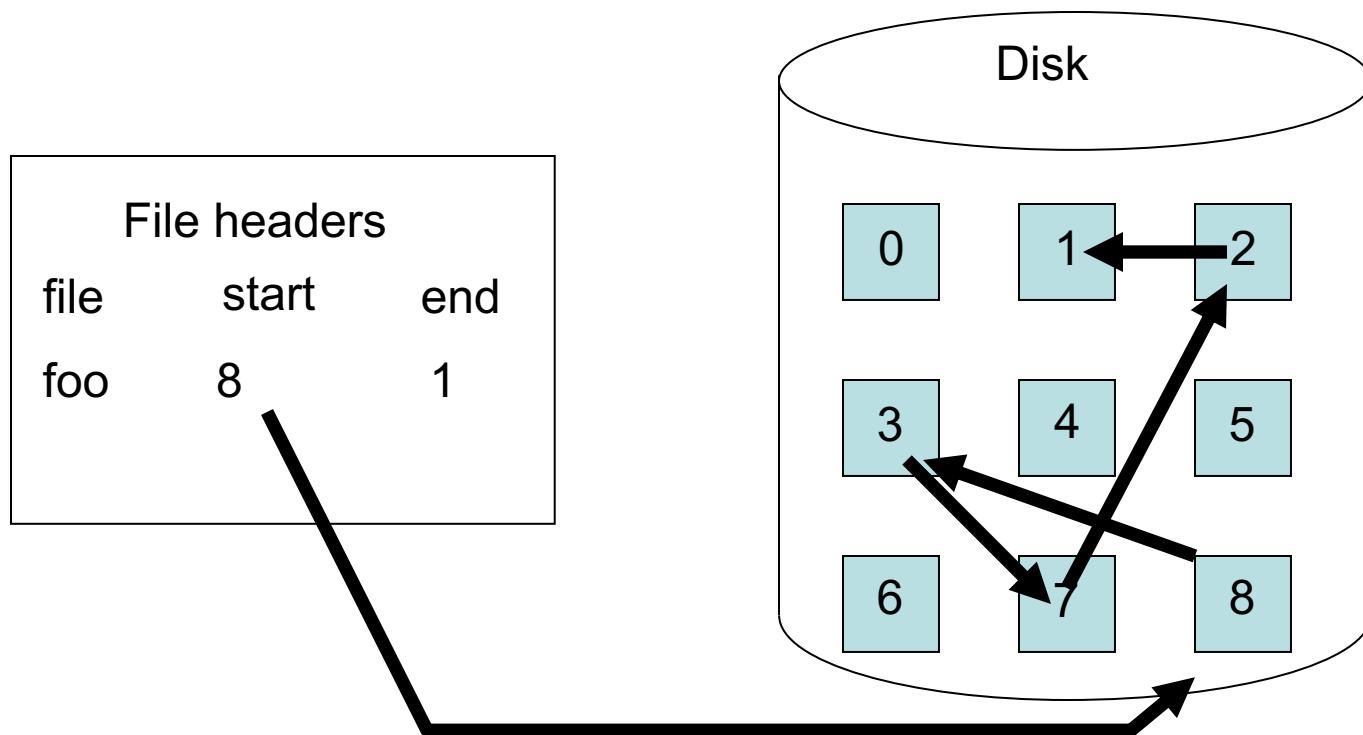
Approach #2: Linked File Allocation

- Linked Allocation
 - each file is a linked list of disk blocks
 - to add to a file, just modify the linked list either in the middle or at the tail, depending on where you wish to add a block
 - to read from a file, traverse linked list until reaching the desired data block



Linked File Allocation

- Linked Allocation
 - each file is a linked list of disk blocks



Linked File Allocation

- Advantages:
 - solves problems of contiguous allocation
 - no external fragmentation
 - don't need to know size of a file a priori
 - Minimal bookkeeping overhead in file header – just a pointer to start of file on disk
 - (-) Compromise is that all the pointer overhead is stored in each disk block
 - Good for sequential read/write data access
 - Easy to insert data into middle of linked list

Linked File Allocation

- Disadvantages:
 - performance of random (direct) data access is extremely slow for reads/writes
 - because you have to traverse the linked list until indexing into the correct disk block
 - Space is required for pointers on disk in every disk block
 - reliability is fragile
 - if one pointer is in error or corrupted, then lose the rest of the file after that pointer

File Allocation

Approaches:

1. Contiguous file allocation

- a file is laid out contiguously, i.e. if a file is n blocks long, then a starting address b is selected and the file is allocated blocks $b, b+1, b+2, \dots, b+n-1$

2. Linked Allocation

- each file is a linked list of disk blocks

3. File Allocation Table (FAT) is an important variation of linked lists

- Don't embed the pointers of the linked list with the file data blocks themselves
- Instead, separate the pointers out and put them in a special table – the file allocation table (FAT)
- The FAT is located at a section of disk at the beginning of a volume

4. Indexed Allocation

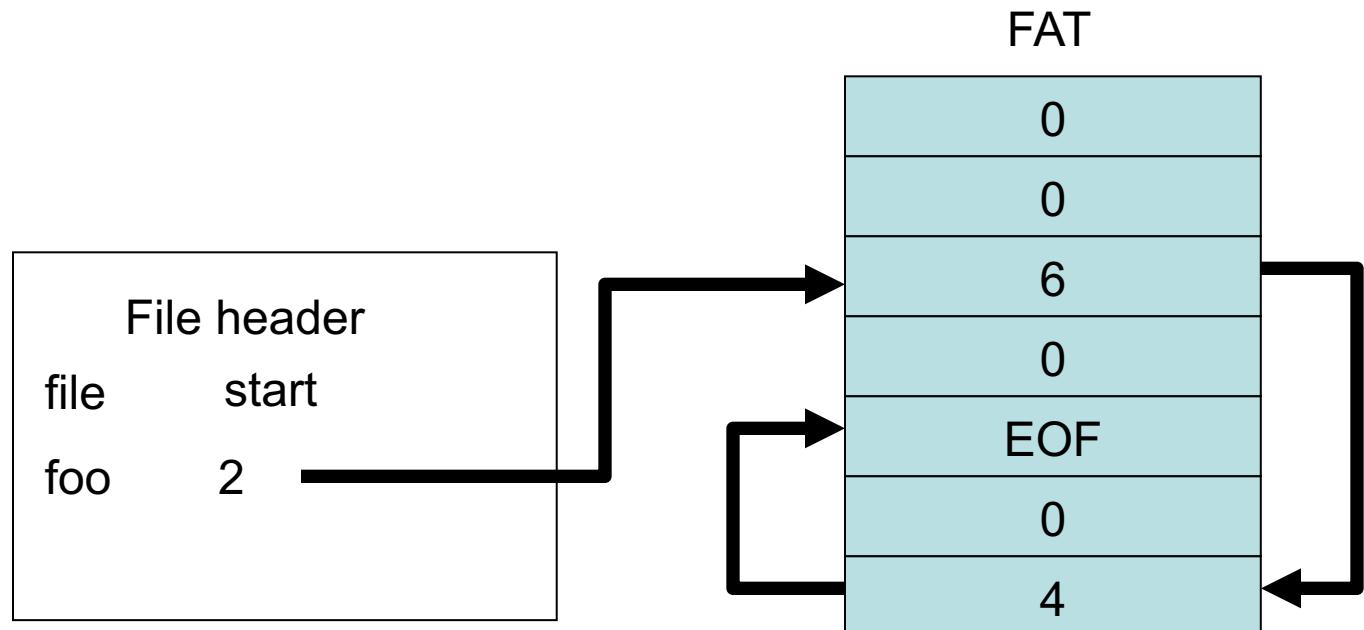
- collect all pointers into a list or table called an *index block*
- the index j into the list or index block retrieves a pointer to the j 'th block on disk

Approach #3: File Allocation Table (FAT)

- the File Allocation Table (FAT) is an important variation of linked lists
 - Don't embed the pointers of the linked list within the file data blocks themselves
 - Instead, separate the pointers out and put them in a special table – the file allocation table (FAT)
 - The FAT is located at a section of disk at the beginning of a volume

File Allocation Table

- entries in the FAT point to other entries in the FAT as a linked list, but their values are interpreted as the disk block number
- unused blocks in FAT initialized to 0



File Allocation Table

- FAT file systems used in MS-DOS and Win95/98
 - Bill Gates designed/coded original FAT file system
 - replaced by NTFS (basis of Windows file systems from WinNT through Windows Vista/7)
 - Variants include FAT16, FAT32, etc. FAT16 and FAT32 refer to the size of the address used in the FAT.

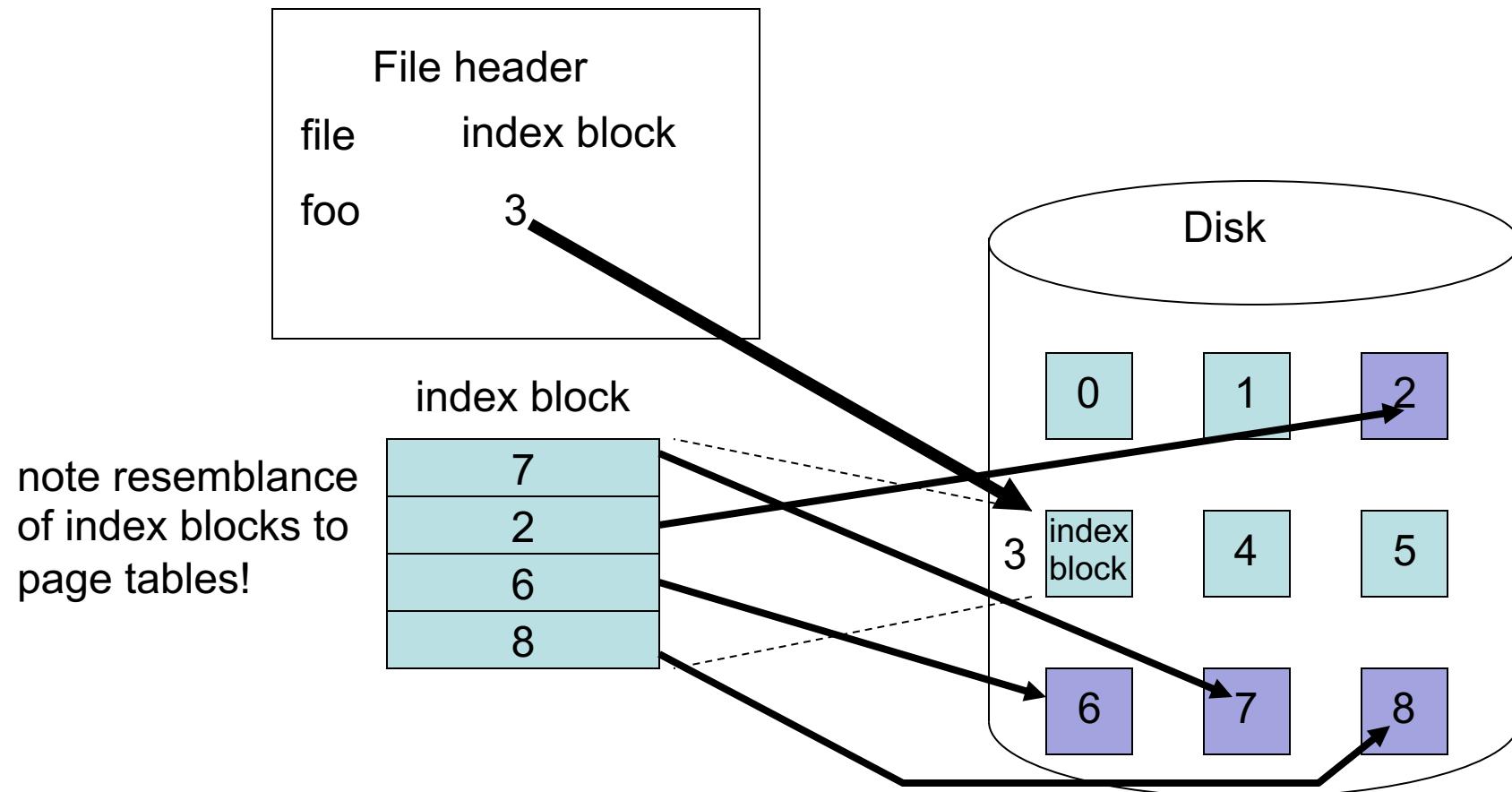
File Allocation Table

- Linked list for a file is terminated by a special end-of-file EOF value
- (+) Allocating a new block is simple - find the first 0-valued block
- (+) Random Reads/Writes faster than pure linked list
 - the pointers are all colocated in the FAT near each other at the beginning of disk volume - low disk seek time
- (-) still have to traverse the linked list though to find location of data – this is still a slow operation

Approach #4: Indexed Allocation

- Conceptually, collect all pointers into a list or table called an *index block*
 - the index j into the list or index block retrieves a pointer to the j 'th block on disk
 - Looks kind of like a page table, except it's extensible
- Unlike the FAT, the index block can be stored in any block on disk, not just in a special section at the beginning of disk
- Unlike the FAT, the index is just a linear list of pointers

Indexed Allocation

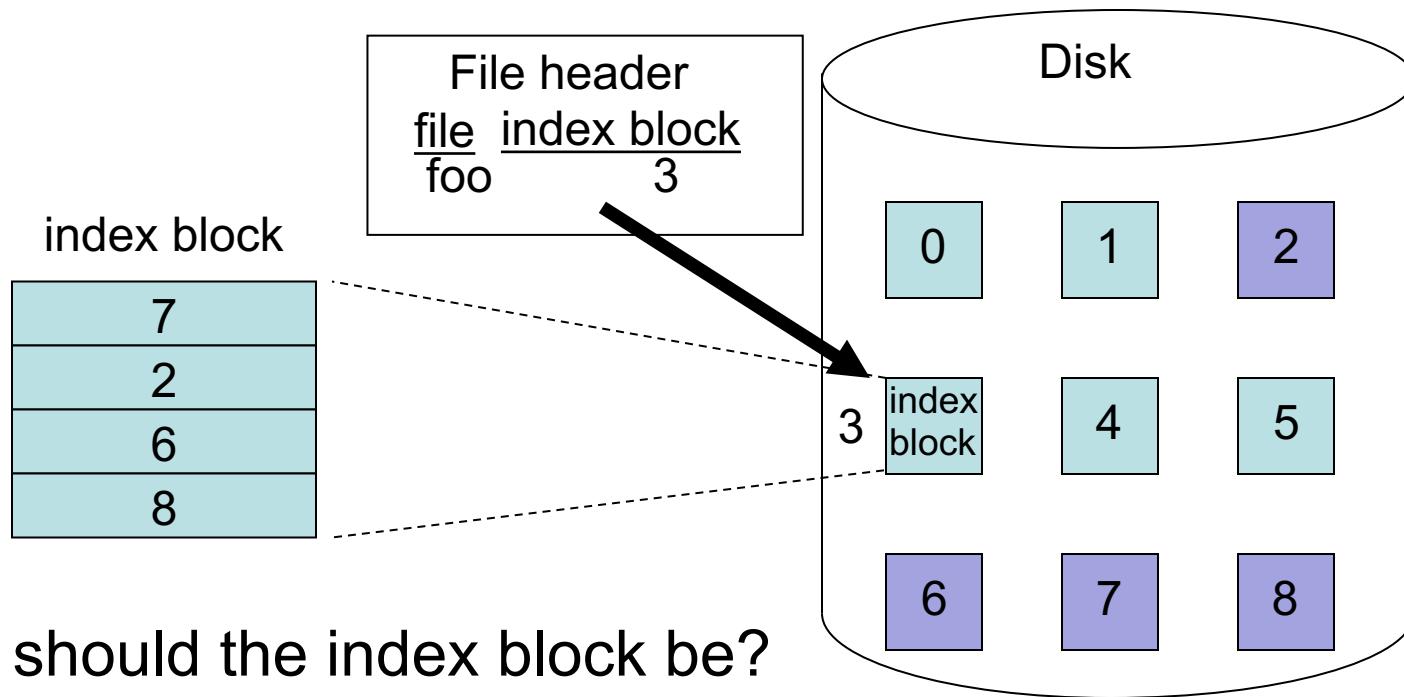


Indexed Allocation

- Solves many problems of contiguous and linked list allocation:
 - no external fragmentation
 - size of file not required a priori
 - don't have to traverse linked list for random/direct reads/writes
 - just index quickly into the index block

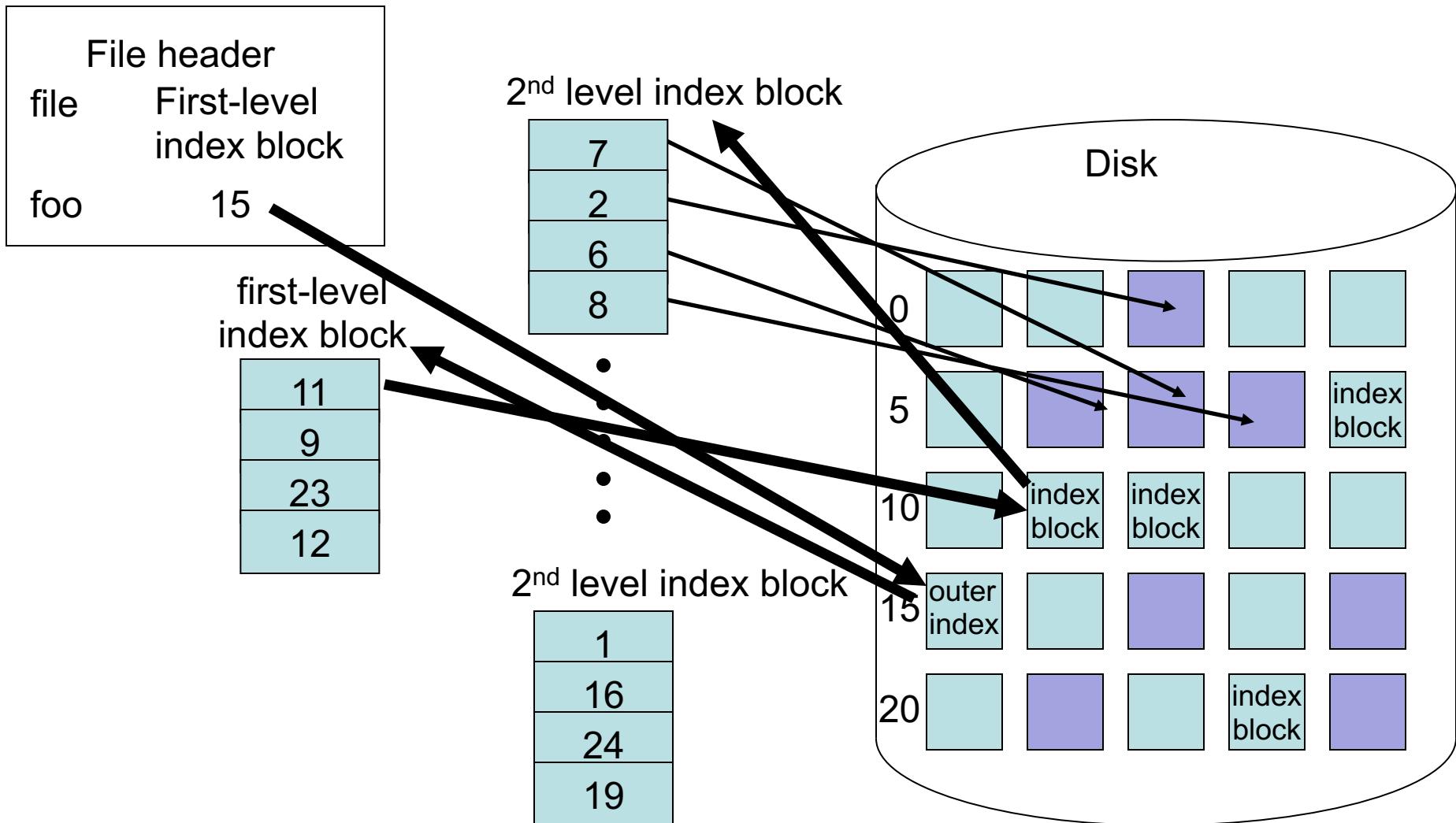
Indexed Allocation

- Solutions:
 1. link together index blocks –
 - each index block has link to next index block
 2. *multilevel index* (like hierarchical page tables!)
 - First level is list of all index blocks for file
 - Second level is list of all data blocks in that section of the file



How big should the index block be?

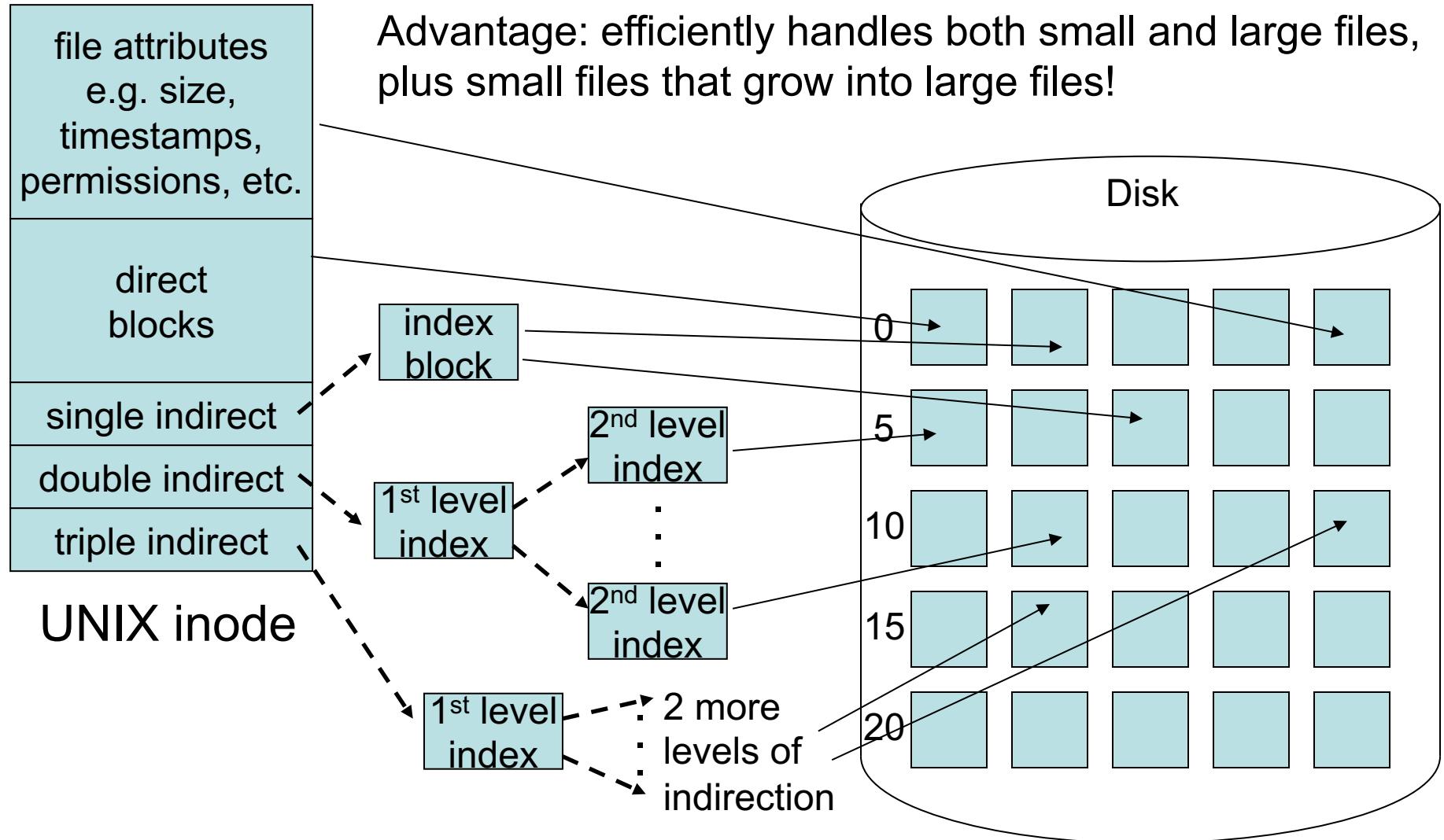
Approach #5: Multilevel Indexed Allocation



Multilevel Indexed Allocation

- Problem with multi-level indexing:
 - accessing small files might take just as long as large files
 - have to go through the same # of levels of indexing, hence same # of disk operations
 - accessing the data of a 100 byte file requires at least 4 block reads

UNIX Multilevel Indexed Allocation



UNIX Multilevel Indexed Allocation

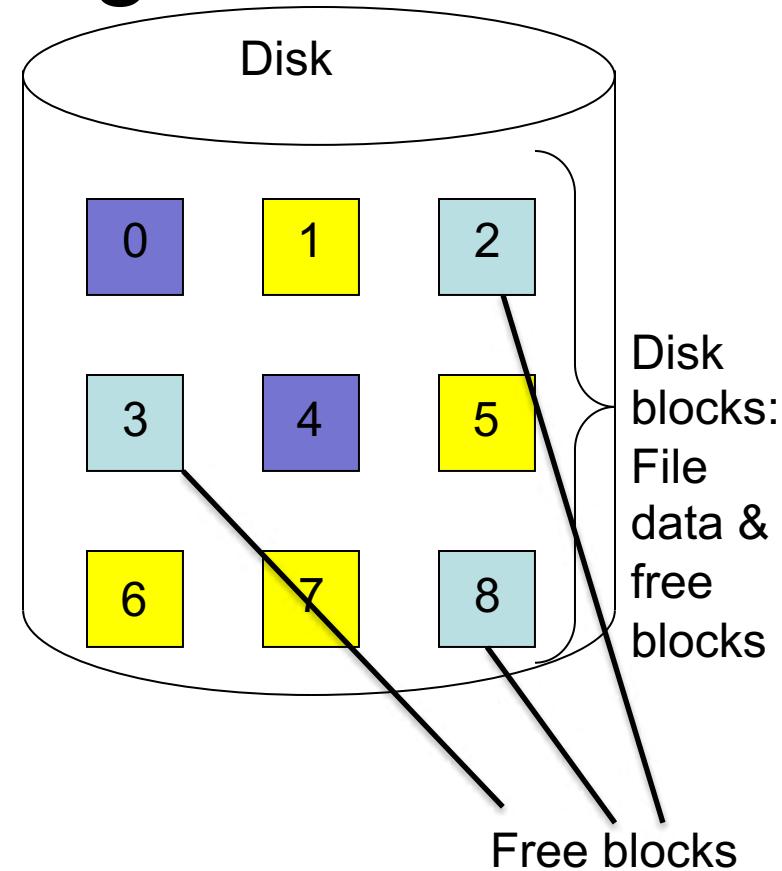
- for small files
 - only uses a small index block of 15 entries, so there is very little wasted memory
- for large files
 - the indirect pointers allow expansion of the index block to span a large number of disk blocks

Comparing File Allocation with Process Allocation

- In both cases, mapping an entity to storage
 - Process address space allocated frames in RAM via page tables
 - File data is allocated to disk/flash
- Differences:
 - Address spaces are fixed in size and known in advance,
 - Files grow/contract over time – files need a mapping/allocation system that is more flexible than page tables, which can't grow
 - Address spaces can be sparse and mostly unused, while file data is all “used”

Free Space Management

- Another aspect of managing a file system is managing free space
 - the file system needs to keep track of what blocks of disk are free/unallocated
 - keeps a free-space “list”
 - In this example, need to keep track that disk blocks 2, 3 and 8 are free/unallocated



Free Space Management Approaches

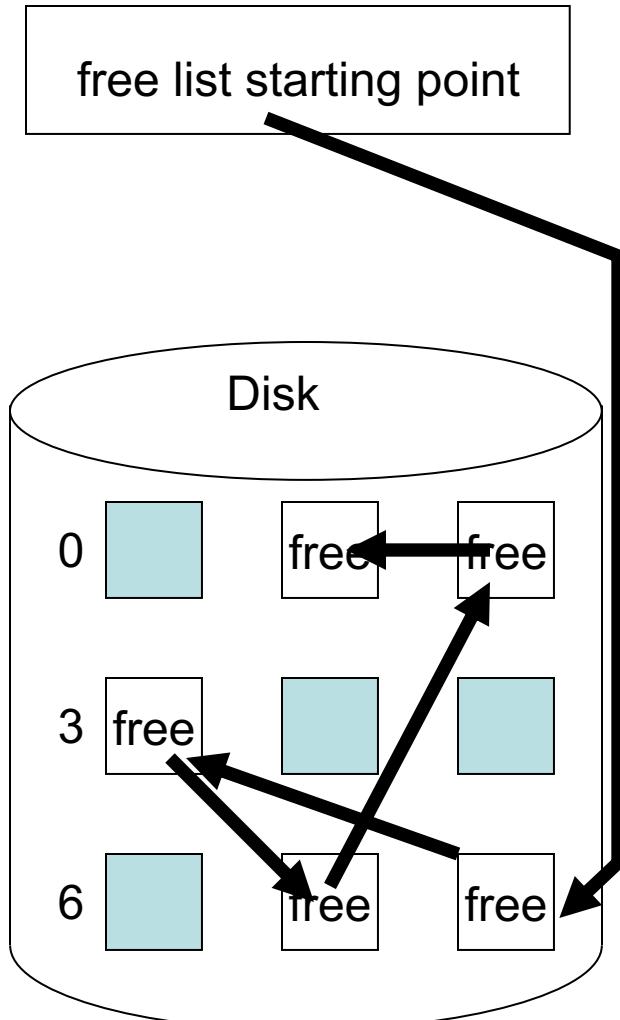
1. Bit Vector or Bit Map

- each block is represented by a bit.
- Concatenate all such bits into an array of bits, namely a bit vector.
 - The j'th bit indicates whether the j'th block has been allocated.
 - if bit = 1, then a block is free, else if bit = 0, then block is allocated

Free Space Management Approaches

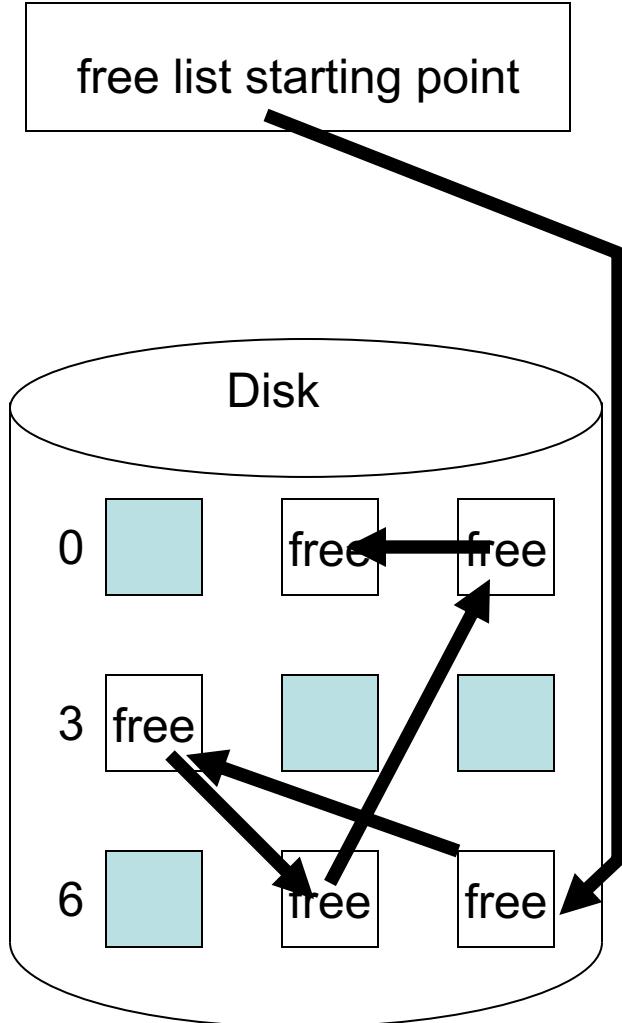
2. Linked List

- link together all free blocks
- efficient - keeps track of only the free blocks.
 - bitmap has the overhead of tracking both free and allocated blocks - this is wasteful if memory is mostly allocated
- Faster than bitmap – find 1st free block immediately



Free Space Management Approaches

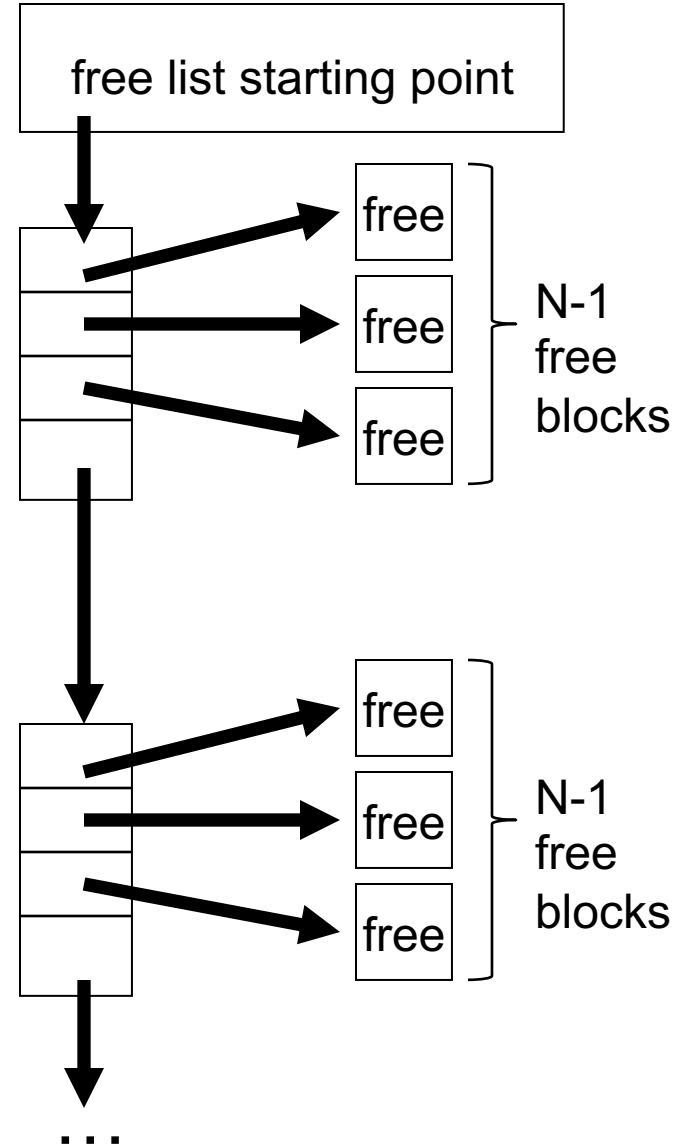
- Problem with Linked List free space management:
 - traversing the free list is slow if you want to allocate a large number of free blocks all at once
 - hopefully this occurs infrequently



Free Space Management Approaches

3. Grouping

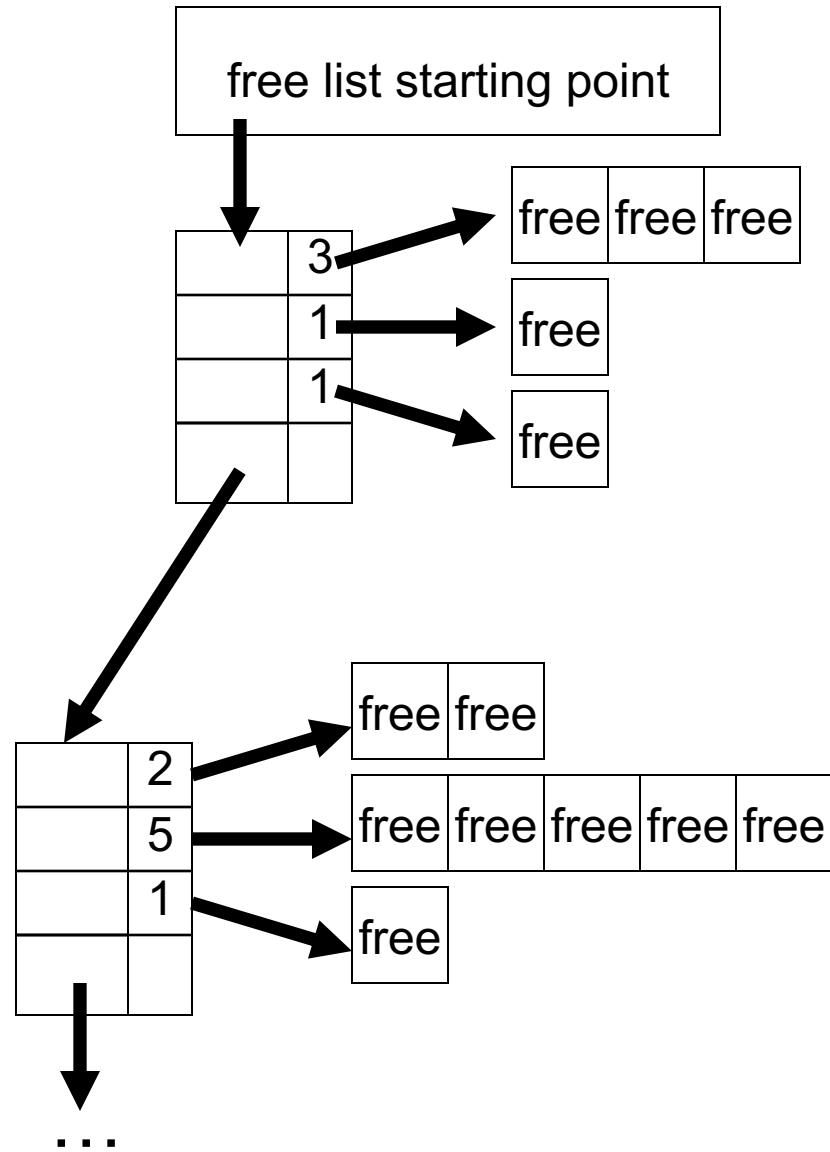
- linked list, except store $n-1$ pointers to free blocks in each list block
- the last block points to the next list block containing more free pointers
- allows faster allocation of larger numbers of free blocks all at once



Free Space Management Approaches

4. Counting -

- grouped linked list, but also add a field to each pointer entry that indicates the number of free blocks immediately after the block pointed to
- even faster allocation of large #'s of free blocks





File System Performance, Reliability, and Fault Recovery

File System Performance

- Approaches to improve performance in a file system:
 - In memory:
 - file header: caching FCB information about open files in memory improves performance (faster access)
 - directory:
 - caching directory entries in memory improves access speed.
 - And hash the directory tree to quickly find an entry and see if it's in memory.

File System Performance

- On disk:
 - file data: indexed allocation is generally faster than traversing linked list allocation
 - free block list: counting, grouped, linked list allows fast allocation of large # of files

File System Performance

- Other potential optimizations:
 - the disk controller can also have its own cache that stores file data/FCBs/etc. for fast access
 - Cache file data in memory
 - Smarter layout on disk: keep an inode/FCB near file data to reduce disk seeks, and/or file data blocks near each other
 - *read ahead*:
if the OS knows this is sequential access, then read the requested page and several subsequent pages into main memory cache in anticipation of future reads

File System Performance

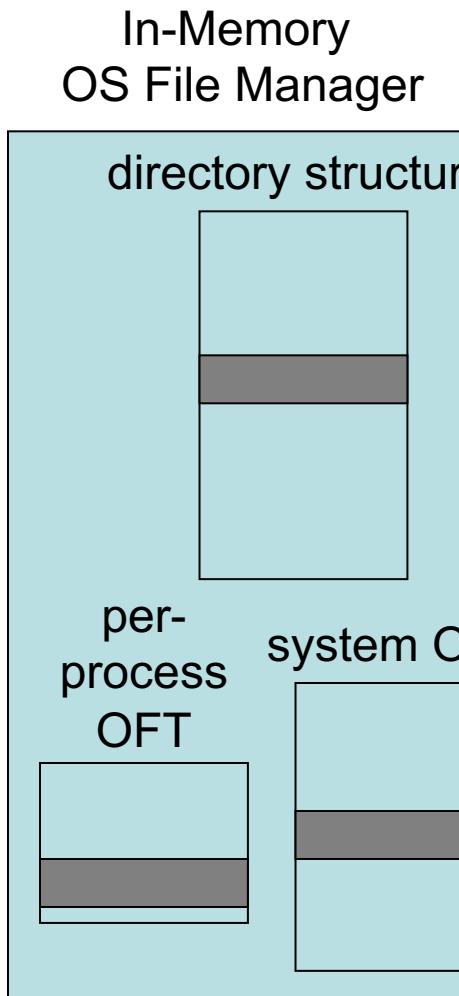
- Some other potential optimizations:
 - *asynchronous writes*: delay writing of file data until sometime later.
 - Advantages:
 - removes disk I/O wait time from the critical path of execution, e.g. a write(X) to a file can return quickly rather than waiting for completion of disk I/O, thereby allowing the program to move forward in its execution
 - This allows a disk to schedule writes efficiently, grouping nearby writes together
 - May avoid a disk write if the data has been changed again soon
 - note that in certain cases, you may prefer to enforce synchronous writes, e.g. when modifying file metadata in the FCB on an open() call

File System Reliability and Fault Recovery

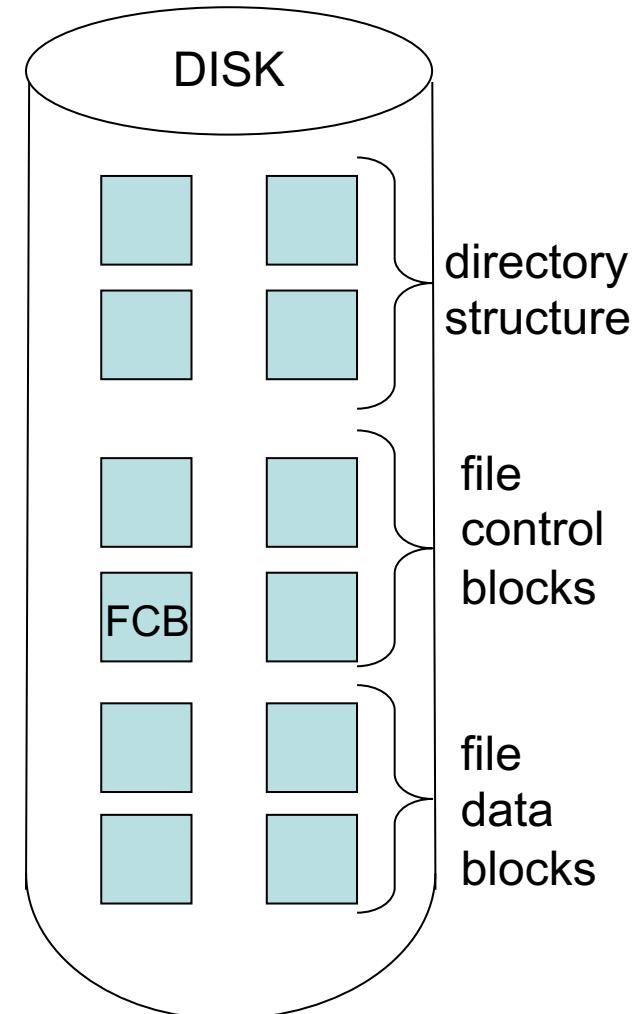
File System Reliability/ Fault Recovery

- **In general, OS should gracefully recover from hardware or software failure**
 - The file system needs to be engineered to ensure reliability/fault recovery
- **Problem: File system is quite fragile to system crashes**
 - There is a portion of the file system that is cached in memory
 - This portion may be inconsistent with the complete file system stored on disk

File System Fragility



- All in-memory file system data are lost on a **power loss**:
 - Directories, file metadata, and file data may all be cached in memory
 - They may all be modified
 - These modifications are lost if they weren't saved to disk



File System Reliability/ Fault Recovery

- Problem #1: **asynchronous writes produce inconsistency between in-memory and on-disk file system**
- Example: promised writes of filed data that were delayed by asynchrony may be lost
- Example: asynchronous writes of directory metadata can create inconsistency between the file system on disk and the writes cached in RAM
 - Directory information in RAM can be more up to date than disk
 - if there is a system failure, e.g. power loss, then the cached writes may be lost
 - in this case, the promised writes will not be executed

File System Reliability & Fault Recovery

- Solution: **To address asynchronous write inconsistency,**
 - UNIX caches directory entries for reads
 - But UNIX does not cache any data write that changes metadata or free space allocation

These changes to critical metadata are written synchronously (immediately) to disk, before the data blocks are written
- Problem #2: **Even if all writes are synchronous, there is still a consistency problem:**
 - any of the individual synchronous/asynchronous writes to disk can fail halfway through the operation, leaving a half-written directory entry, FCB, or file data block.

File System Reliability & Fault Recovery

- Problem #3: **Complex operations can create inconsistency while waiting for them to complete**
 - e.g. a file create() involves many operations, and may be interrupted at any time in mid-execution
 - file create() updates the directory, FCB, file data blocks, and free space management
 - if there is a failure after creating the FCB, then the file system is in an inconsistent state because the file data has not yet been saved on disk,
 - i.e. the directory says there is a file and points to the FCB, but the FCB is incomplete because its index block hasn't been fully allocated

Reliability/Fault Recovery Solutions

- **Approach:** file systems can run a consistency checker like fsck in UNIX or chkdsk in MSDOS
 - in linked allocation, would check each linked list and all FCB's to see if they are consistent with the directory structure.
 - similar checks for indexed allocation
 - Check each allocated file data block to see that its checksum is valid
- Disadvantages:
 - This is computational intensive and takes a long time to check the entire file system.
 - This can detect an error, but doesn't ensure recovery or correction

Reliability/Fault Recovery Solutions

- **Approach:** *log-based recovery* is a solution that helps OS recover from file system failures:
 - OS maintains a log or journal on disk of each operation on the file system
 - called log-based or journaling file systems,
- The **log on disk is consulted after a failure** to reconstruct the file system
 - In a **journaling file system**, the log is seen as a separate entity from the file data.
 - In a **log-structured system**, the log *is* the file system, and there are no separate structures for storing file data and metadata – it's all in the log.

Log-Based Recovery

- Each operation on the file system is written as a record to the log on disk *before* the operation is actually performed on data on disk
 - this is called *write-ahead logging*
- The file system has a sequence of records of operations in the log about what was intended in case of a crash.
 - The log contains a sequence of statements like **“I’m about to write this directory entry/file header/file data block”**,
 - and **“I just finished writing this directory/FH/data”**.

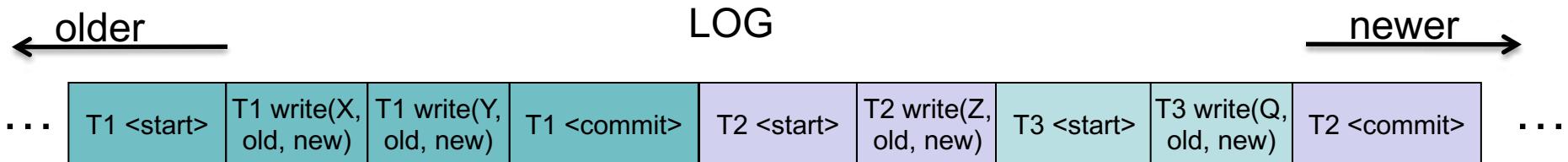
Log-Based Recovery

- operations are grouped in sets called ***transactions***
 - e.g. a file create() has many steps.
 - You either want the entire file created, or not at all if it fails at any step along the way.
 - Need the set of steps into a single logical unit
 - It is performed in its entirety or not at all.
- Transactions are viewed as atomic
 - Either succeed in their entirety or not at all

Log-Based Recovery

- A transaction T_i looks like the following:
 - begins with $< T_i \text{ starts} >$
 - followed by a sequence of records like $\text{write}(X)$, $\text{read}(Y)$, ... needed to complete the transaction, e.g. a file $\text{create}()$
 - ends with $< T_i \text{ commits} >$
- Write each of these operations to the log

Log-Based Recovery



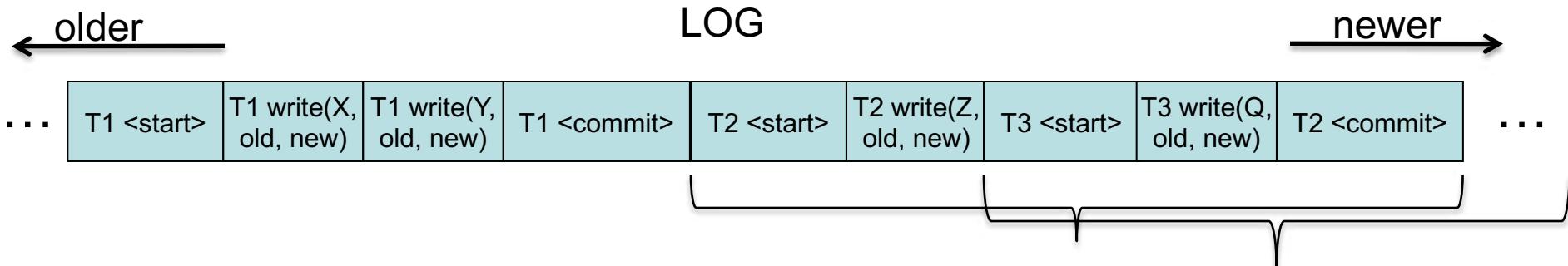
- Each log record of an operation within a transaction consists of:
 - transaction name T_i
 - data item name, e.g. X
 - old value
 - new value
- Both the old and new values must be saved in order for the system to recover from crashes in mid-transaction

Log-Based Recovery



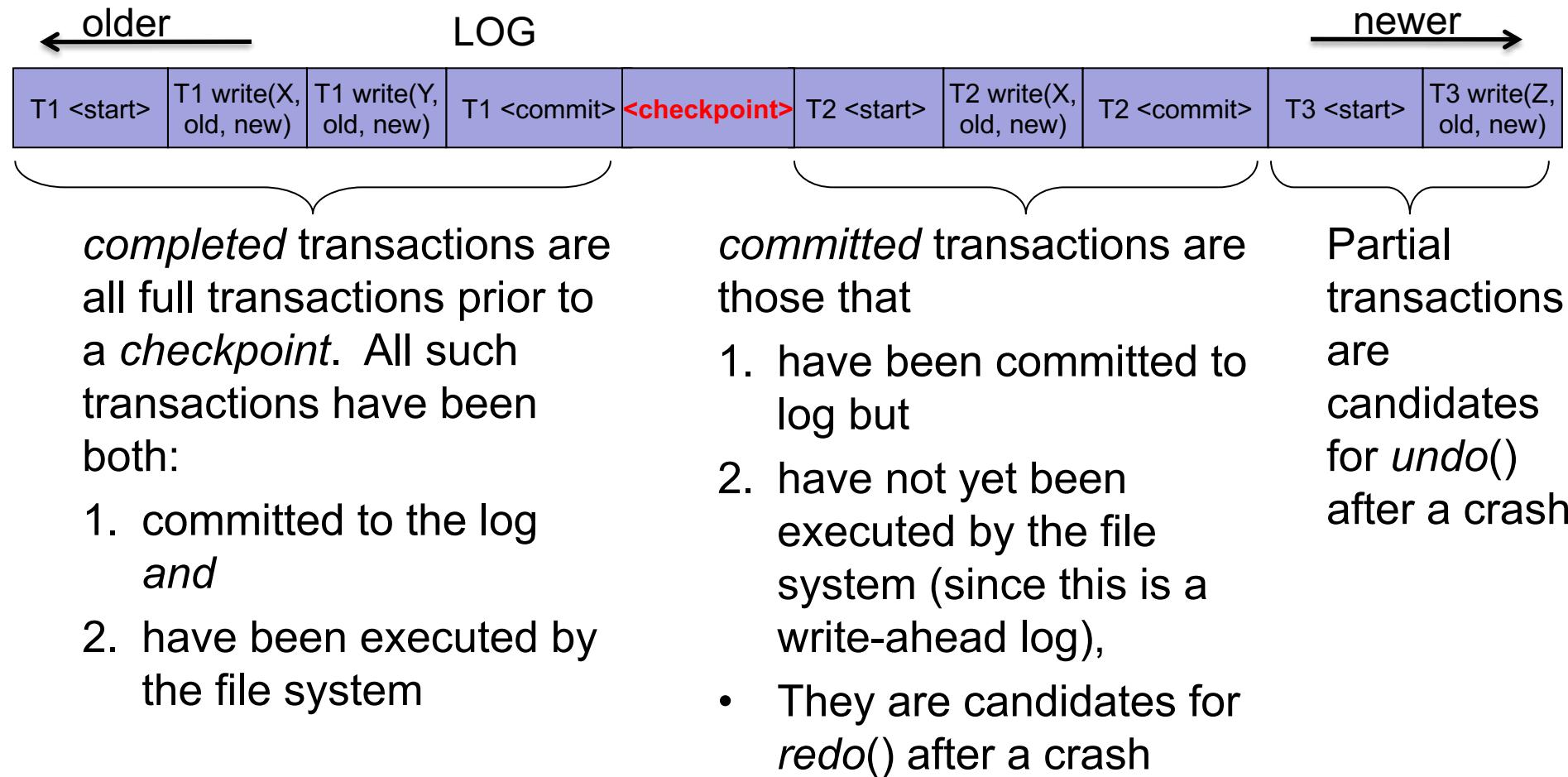
- **A transaction is not considered complete until it is committed in the log**
 - once the <commit> appears in the log, then even if the system crashes after this point, there is enough information in the log to fully execute the transaction upon recovery
 - therefore, once the <commit> appears in the log, it is OK to return from the system call that called file create() or file write()

Log-Based Recovery



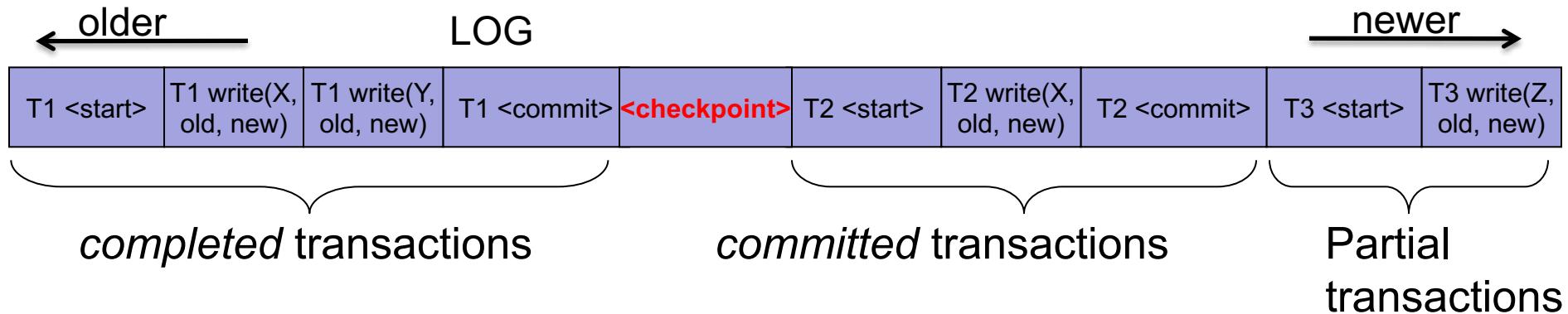
- Operations in different transactions can overlap in the log
- For asynchronous writes, the actual $\text{write}(X)$ to disk may occur much later than the entry written to the log

Log-Based Recovery



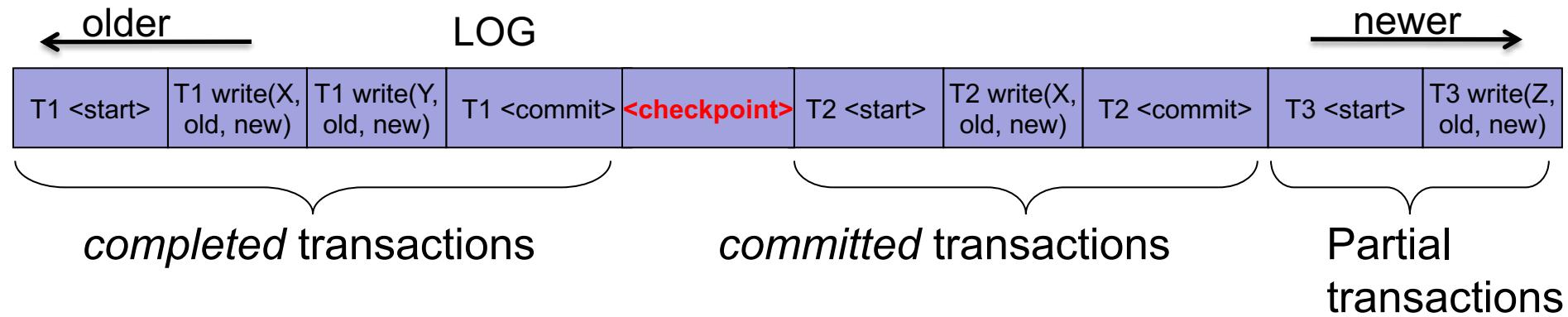
Checkpoint: confirm the state of the system: All trans are committed & executed

Log-Based Recovery



- The checkpoint indicates that all full transactions (those with a <start> and <commit>) prior to the checkpoint (to the left) have been written to *both* the disk *and* log
- Committed transactions are full transactions in the log that are to the right of the most recent checkpoint, and thus have not been written to disk yet

Log-Based Recovery



- In normal operation, the file system will
 - periodically *replay* committed transactions in the log onto disk,
 - Then add a new checkpoint to the log,
 - thus all committed transactions to the left of the newly added checkpoint are converted into completed transactions
 - completed transactions can be removed from the log, or just written over if it's a circular log

Log-Based Recovery

- On a failure, the OS looks for the latest checkpoint in the log, and redo()'s committed transactions and undo()'s partial transactions from that point on
 - redo() transaction T_i if the log contains both $< T_i \text{ starts}>$ and $< T_i \text{ commits}>$ and these transactions appear after a checkpoint
 - undo() transaction T_k if the log contains $< T_k \text{ starts}>$ but not $< T_k \text{ commits}>$
 - this is called an aborted transaction
 - during recovery, such a transaction is rolled back to its former state

Log-structured File Systems

- Assumption:
 - Write is more expensive than read since read can be served very quickly from cache
- Treat the storage as a circular log – Write sequentially to the head of the log
- Advantages:
 - Improve write performance through low seek time (batched write)
 - Allow time-travel or snapshotting
 - Recovery from crash is simpler based on the previous checkpoint.
- Disadvantage:
 - Might make read to be much slower since it fragments files (Optic and Magnetic disk)

Journaling File Systems

- Some file systems like NTFS only write changes to the metadata of a filesystem to the log
 - file headers and directory entries only
 - NOT any changes to file data
 - The journal is separate from the main file system
- Copy-on-write file systems (such as ZFS and Btrfs)
 - Avoid in-place changes to file data by writing out the data in newly allocated blocks
 - Followed by updated metadata that would point to the new data and disown the old
 - Followed by metadata pointing to updated metadata repeatedly to the root of the file system hierarchy
 - Has the same correctness-preserving properties as a journal, without the write-twice overhead
 - Add metadata (checksums) to insure data integrity

Journaling File Systems

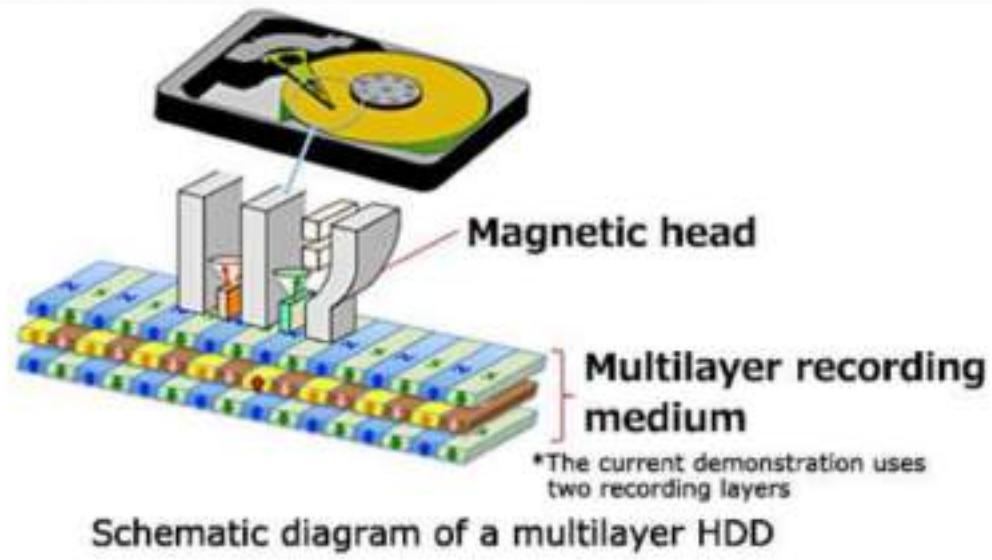
- Linux's ext4fs can be parameterized to operate in 3 modes (cont):
 1. *Journal mode*: both metadata and file data are logged. This is the safest mode, but there is the latency cost of two disk writes for every write.
 2. *Ordered mode*: only metadata is logged, not file data, and it's guaranteed that file contents are written to disk before associated metadata is marked as committed in the journal.
 - This is the default on many Linux distributions.
 3. *Write-back mode*: only metadata is logged, not file data, and no guarantee file data written before metadata, so files can become corrupted.
 - This is riskiest mode/least reliable but fastest.



Lecture 20

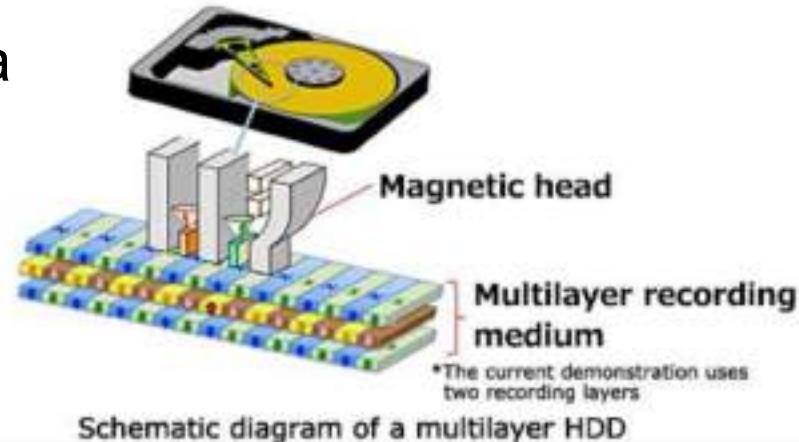
Storage Management

Magnetic Media

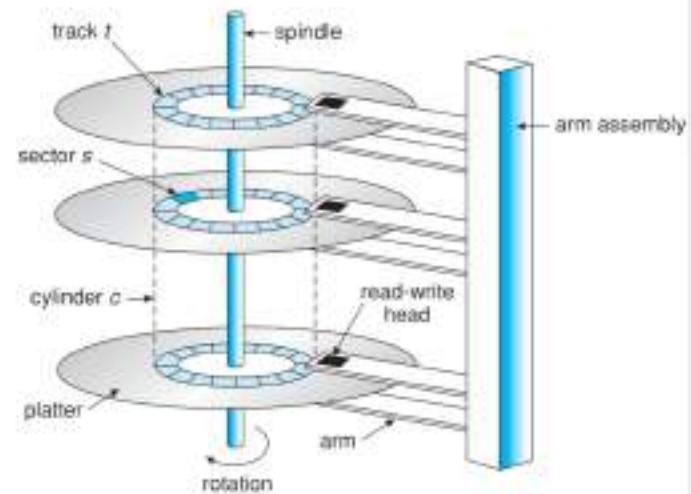


Magnetic Media

- Can set a bit to 1 or 0 by applying a strong enough magnetic field to a small region of a magnetic film
 - region retains its magnetic sense even when power is removed
- To read a sequence of bits,
 - Either move the magnetic head over the magnetic media,
 - Moving (or rotating) the media under the (static) magnetic head – this is the easier approach
- Supports both random and sequential access
- Array of magnetic disk platters increases bit storage at marginal cost in space and new disk heads

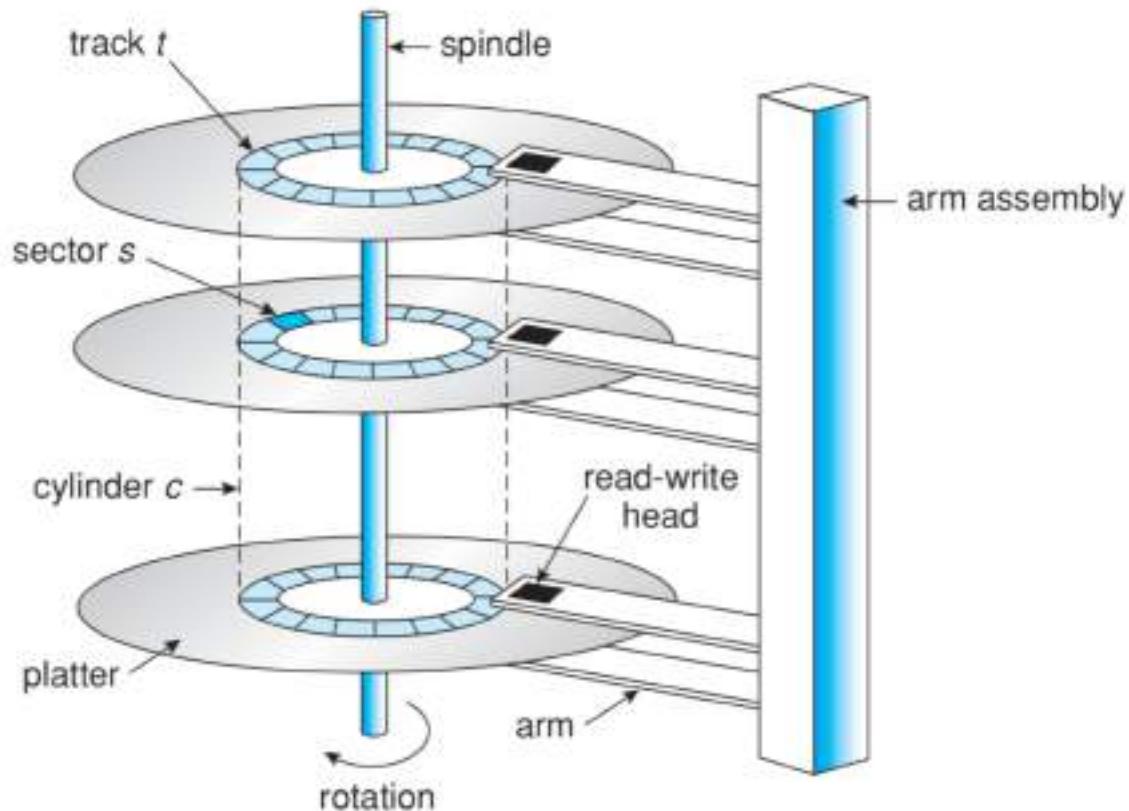


Schematic diagram of a multilayer HDD



Magnetic Disk

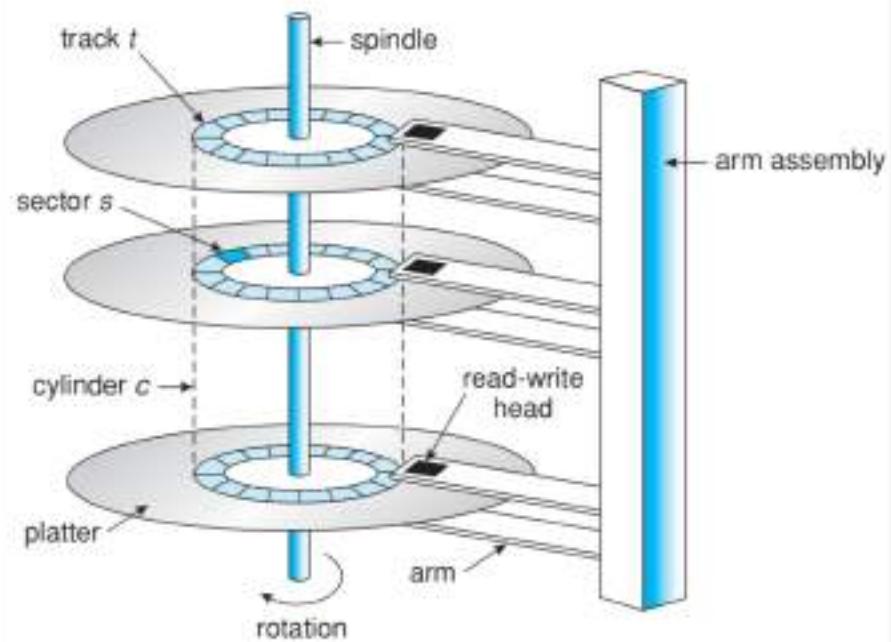
- Platter
- R/W head
- Arm
- Track
- Sector
- Cylinder



Magnetic Disk

Time to access data

- Move head into position to read correct track
(Seek time)
- Wait for sector to rotate under head
(rotational latency)
- Read data from sector on platter
(transfer into memory)



Disk Access Latency

$$\text{total delay to read/write from/to disk} = \text{seek time} + \text{rotational latency} + \text{transfer time}$$

typically 5-10 ms

typically 1-5 ms,
typical RPM is \sim 10000 revolutions per minute, so if on average it takes half a revolution to rotate to the right sector, then $0.5/(10000/60) \approx 3$ ms

typically in 10-100 μ s,
typical 1 Gb/s data transfer rate, so retrieving 10 KB of file data = $80000/(1 \text{ Gb}) = .08 \text{ ms}$

- Total disk access delay is often dominated by seek times and rotational latency



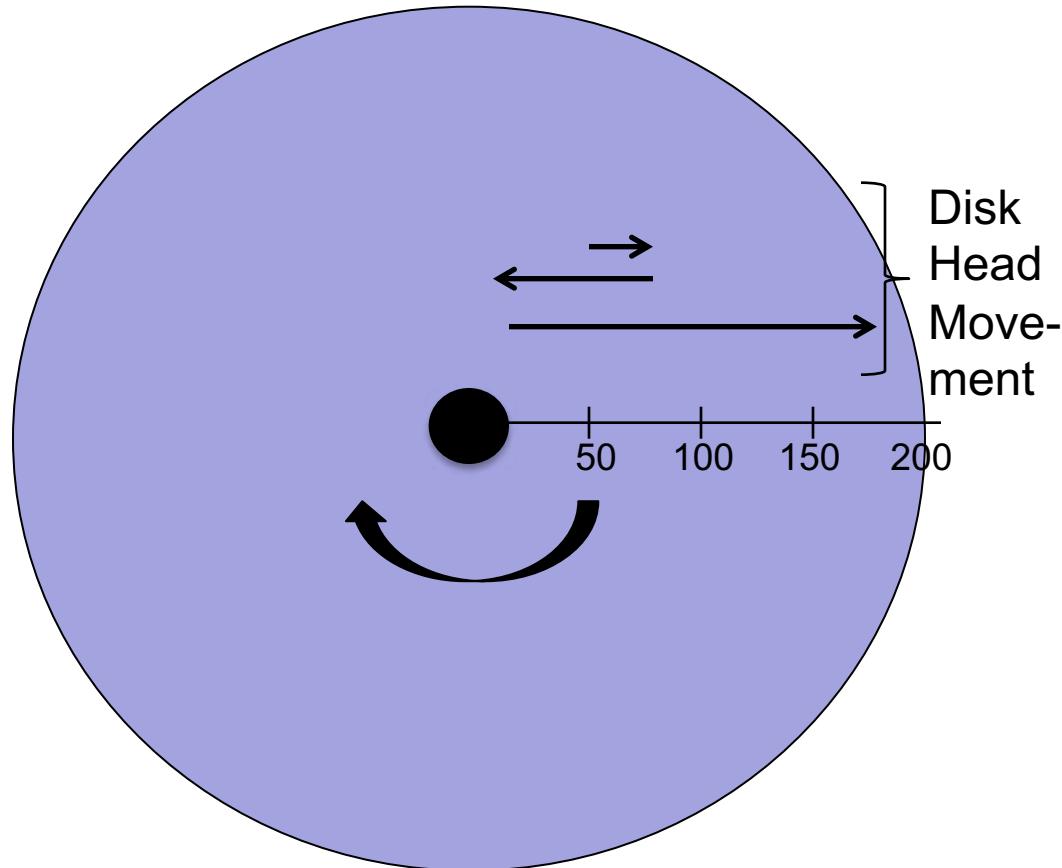
Disk Scheduling

Disk Scheduling

- Any technique that can **reduce seek times and rotational latency** is a big win in terms of improving disk access times
- **Disk scheduling** is designed to reduce seek times and rotational latency
- Disk operations take time to be processed
 - At any given time there will be multiple requests to access data from different places on the disk
 - These requests are stored in the queue
- The OS can choose to intelligently serve or schedule these requests so as to minimize latency
- **Our goal: find an algorithm that minimizes seek time ...**

Disk Scheduling

Rotating Disk



Disk Scheduling

- Our goal: find an algorithm that minimizes seek time ...
 - suppose we are given a series of disk I/O requests for different disk blocks that reside on the following different cylinder/tracks in the following order:
 - 98, 183, 37, 122, 14, 124, 65, 67
(Initial track location is 53)
 - The **total number of tracks traversed** is used as an evaluation metric of the algorithms.

Disk Scheduling

- FCFS Disk Scheduling
 - Requests would be scheduled in the same order as the order of arrival
 - Total number of cylinders/tracks traversed by the disk head under FCFS algorithm would be
$$|53-98| + |98-183| + |183-37| + |37-122| + \dots = 640 \text{ cylinders}$$
 - Observation:
 - disk R/W head would move less if 37 and 14 could be serviced together,
 - similarly for 122 and 124
 - easy to implement – but slow

98, 183, 37, 122, 14, 124, 65, 67
(Initial track location is 53)

Disk Scheduling

- SSTF Disk Scheduling
 - Shortest-Seek-Time-First (SSTF) scheduling selects the next request with the minimum seek time from the current R/W head position
 - SSTF services requests in this order
 - 65, 67, 37 (closer than 98), 14, 98, 122, 124, 183 => ???
 - Locally optimal, but why not globally optimal?
 - Better to move disk head from 53 to 37 then 14 then 65 (this would result in ??? cylinders total)
 - Another drawback: starvation
 - While the disk head is positioned at 37, a series of new nearby requests may come into queue for 39, 35, 38, ...

98, 183, 37, 122, 14, 124, 65, 67
(Initial track location is 53)

Disk Scheduling

- OPT Disk Scheduling
 - OPT looks to minimize the back and forth traversing
 - Move the disk head from the current position to the closest edge track, and **sweep through** once (either innermost to outermost, or outermost to innermost)
 - This should minimize the overlap of back and forth and give the minimum # of tracks covered
 - OPT services requests as follows:
 - 53→37→14→65→67→98→122→124→183
 - OPT ignores the initial direction and changes direction
 - total # cylinders traversed = 39 down + 169 up = 208 cylinders
 - **The problem with OPT is that:**
 - you have to recalculate every time there's a new request for a disk track
 - Best approximation to OPT is probably to just scan the disk in one direction, and then change the direction of scanning

98, 183, 37, 122, 14, 124, 65, 67
(Initial track location is 53)

Scheduling Algorithm	# Cylinders
OPT	208
FCFS	640
SSTF	236
SCAN	333

Disk Scheduling

- **SCAN Disk Scheduling**

- Move the read/write head in one direction from innermost to outermost track
- Then reserve direction
 - Sweeping across the disk in alternate directions
- SCAN services requests as follows:
 - 53 → 65 → 67 → 98 → 122 → 124 → 183 → 200 → 37 → 14
- Advantages:
 - Approximates OPT
 - Starvation free solution
 - handles dynamic arrival of new requests
 - simple to implement

98, 183, 37, 122, 14, 124, 65, 67
 (Initial track location is 53)

Disk Scheduling

- SCAN Disk Scheduling
 - Disadvantages:
 - Problem 1: SCAN only approximates OPT
 - There is significantly more overlap compared to the OPT service sequence
 - Problem 2: unnecessarily goes to extreme edges of disk even if no requests there
 - Problem 3: unfair for tracks on the edges of the disk (both inside and outside)
 - Writes to the edge tracks take the longest since after SCAN has reversed directions at one edge, the writes on the other edge always wait the longest to be served
 - After two full scans back and forth, middle tracks get serviced twice, edge tracks only once

98, 183, 37, 122, 14, 124, 65, 67
(Initial track location is 53)

Scheduling Algorithm	# Cylinders
OPT	208
FCFS	640
SSTF	236
SCAN	333
C-SCAN	384

Disk Scheduling

- C-SCAN Disk Scheduling
 - Treat disk as a queue of tracks
 - When reaching one edge, move the R/W head all the way back to the other edge
 - Continue scanning in the same direction rather than reserving direction
 - Circular SCAN, or C-SCAN
 - After reaching an edge, writes at the opposite edge get served first
 - This is more fair in service times for all tracks
 - C-SCAN services requests as follows:
 - 53→65→67→98→122→124→183→200→1→14→37
 - For a total distance of 384 tracks
 - Note how no scanning is done as the disk head is repositioned from 200 all the way down to 1 to ensure circularity

98, 183, 37, 122, 14, 124, 65, 67
 (Initial track location is 53)

Scheduling Algorithm	# Cylinders
OPT	208
FCFS	640
SSTF	236
SCAN	333
C-SCAN	384
LOOK	299

Disk Scheduling

- **LOOK Disk Scheduling**
 - don't travel to the extreme edge if no requests there
 - look at the furthest request in the current direction
 - only move the disk head that far
 - then reverse direction
 - No starvation in this algorithm either
 - LOOK would service requests in the following order:
 - 53 → 65 → 67 → 98 → 122 → 124 → 183 → 37 → 14
 - total # cylinders traversed = 130 up + 169 down = 299 cylinders

98, 183, 37, 122, 14, 124, 65, 67
 (Initial track location is 53)

Disk Scheduling

- C-LOOK Disk Scheduling
 - Improve LOOK by making it more fair
 - Circular LOOK
 - After moving disk head to furthest request in current direction
 - Move disk head directly all the way back to the further request in the other direction
 - Continue scanning in the same direction
 - C-LOOK would service requests in the following order:
 - 53→65→67→98→122→124→183→14→37
 - total # cylinders traversed = 130 up + 169 down + 23 up = 322 cylinders

Scheduling Algorithm	# Cylinders
OPT	208
FCFS	640
SSTF	236
SCAN	333
C-SCAN	384
LOOK	299
C-LOOK	322

98, 183, 37, 122, 14, 124, 65, 67
(Initial track location is 53)

Disk Scheduling

- Disk Scheduling in Practice
 - In the previous examples, the # of cylinders traversed by SCAN, LOOK, and C-LOOK are much lower if the initial direction was down
 - We can't keep changing the direction
 - Can cause starvation/unfairness.
 - Preserving directionality of the scan is important for fairness, but we sacrifice some optimality/performance
 - Either SSTF or LOOK are reasonable choices as default disk scheduling algorithms
 - These algorithms for reducing seek times are typically embedded in the disk controller today
 - There are other algorithms for reducing rotational latency, also embedded in the disk controller

Disk Scheduling vs. File Layout

- File Layout and Disk Scheduling
 - Block allocation schemes can effect access times
 - When a file's data is spread across the disk, it increasing seek times.
 - Contiguous allocation minimizes seek times
 - Select free blocks near adjacent file data
 - Opening a file for the first time requires searching the disk for the directory, file description header, and file data
 - These are all spread out across the disk
 - Seek times are large
 - Could store directories in the middle tracks, so at most half the disk is traversed to find the file header and data
 - Could store the file header near the file data, or near the directory



Flash Memory



What is Flash Memory?

- Flash memory is a type of solid state storage
 - bits are stored in an electronic chip
 - not on a magnetic disk
 - Programmed and erased electrically
 - no mechanical parts
 - Non-volatile storage
 - i.e. “permanent” with caveats
- Primarily used in solid state drives (SSDs), memory cards, USB flash drives, mobile smartphones, digital cameras, etc.



What is Flash Memory?

- Flash memory is a form of EEPROM
 - Electrically Erasable Programmable Read Only Memory
 - bits can be rewritten again and again using ordinary electrical signals
 - non-volatile
- Compared to:



RAM (Random Access Memory)
– main memory, volatile

ROM (Read Only Memory) – code is burned in at the factory and can only be read thereafter, no writes

PROM (Programmable ROM) – code can be written once using a special machine

EPROM (Erasable PROM) – code can be written multiple times with a special machine

What is Flash Memory?

- Advantages vs. disk:
 - much faster access (lower latency),
 - more resistant to kinetic shock (& intense pressure, water immersion etc.),
 - more compact,
 - lighter,
 - lower power (typically 1/3-1/2 of disks), ...

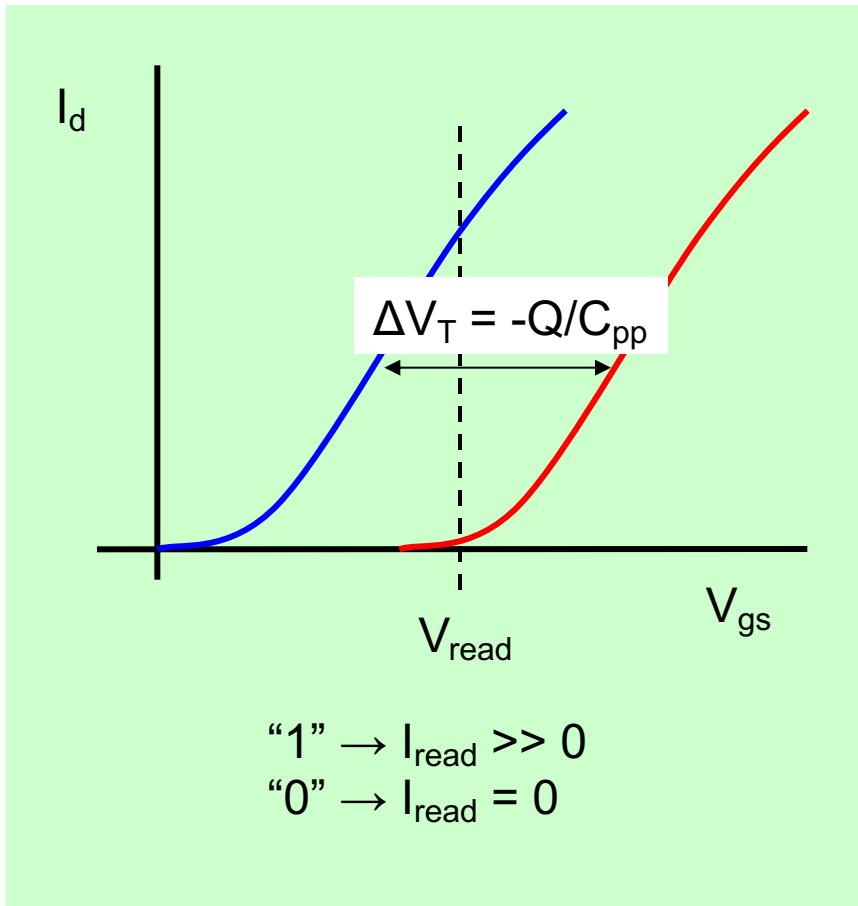


What is Flash Memory?

- Disadvantages vs. disk:
 - more costly per byte,
 - limited lifetime,
 - erases are costly, ...
- Flash's name:
 - can inexpensively erase a large amount of solid state memory quickly, like a “flash” of light



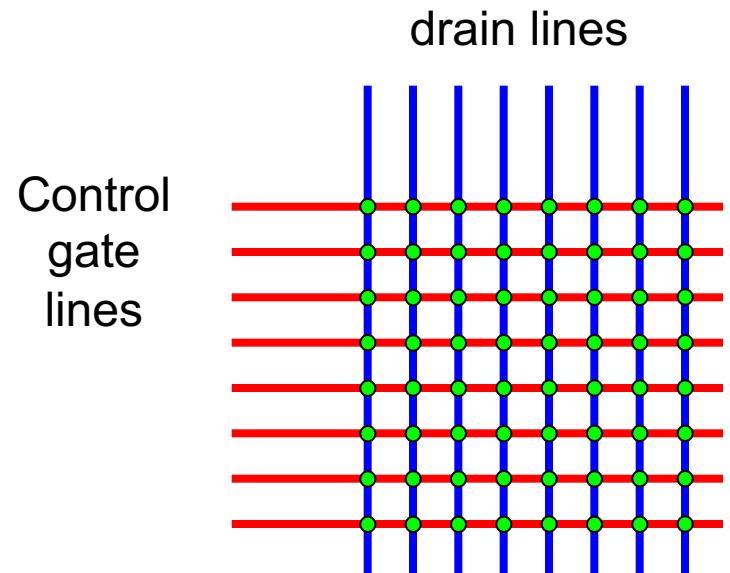
Logic “0” and “1”



Reading a bit means:

1. Apply V_{read} on the control gate
2. Measure drain current I_d of the floating-gate transistor

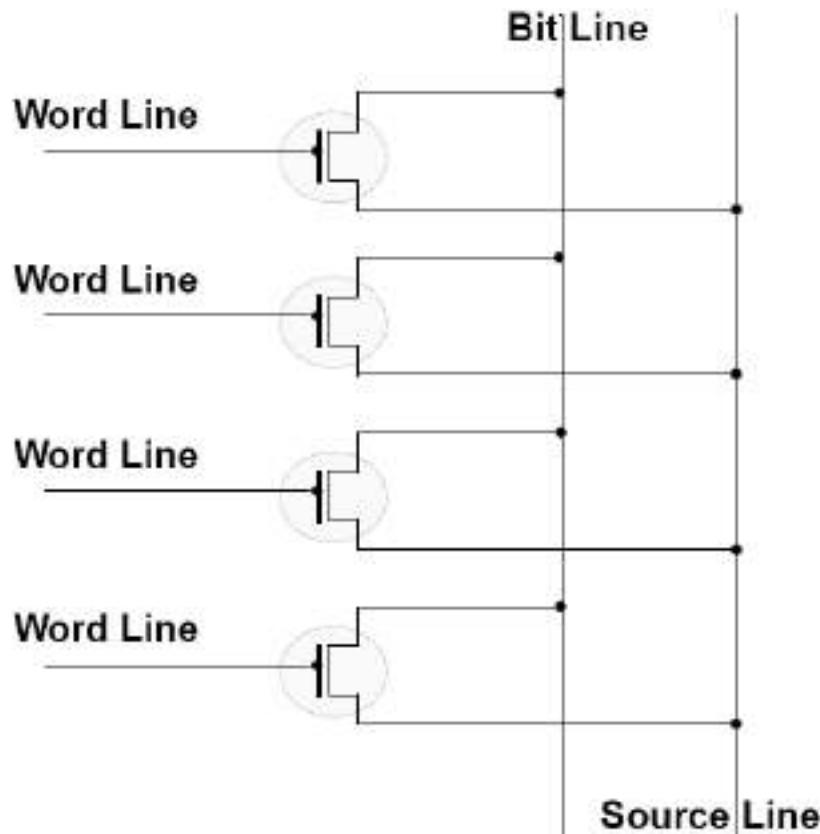
When cells are placed in a matrix:



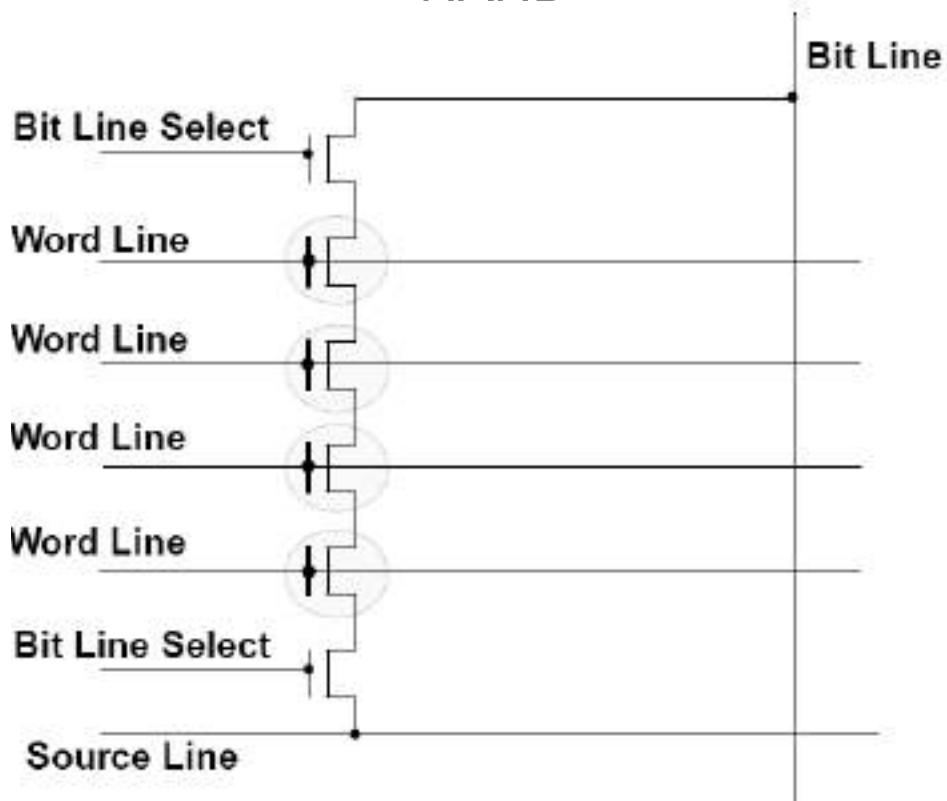
NOR or NAND addressing

'Word' = control gate; 'bit' = drain

- NOR



- NAND



less contacts → more compact

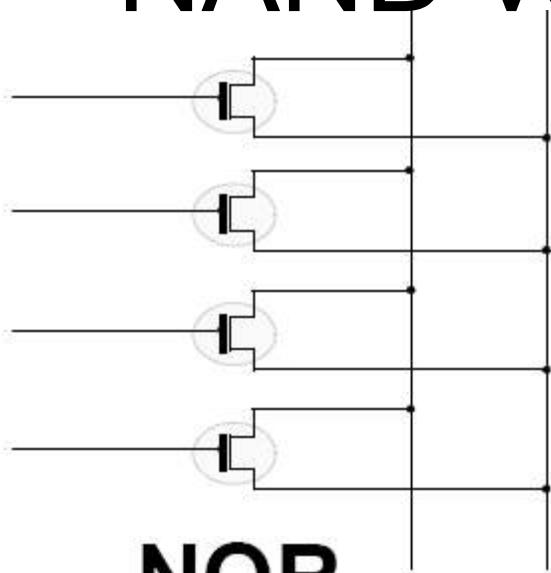
NAND Flash

- Smaller memory cell area (less expensive)
- Word accessible
 - large granularity, 100-1000 bits/word,
 - good for secondary storage where files don't mind being read out in large chunks/words)
- Slower random byte access (have to read a word)
- Short erasing (~2 ms) and programming times (~5 MB/sec sustained write speed)
- Longer write lifetime

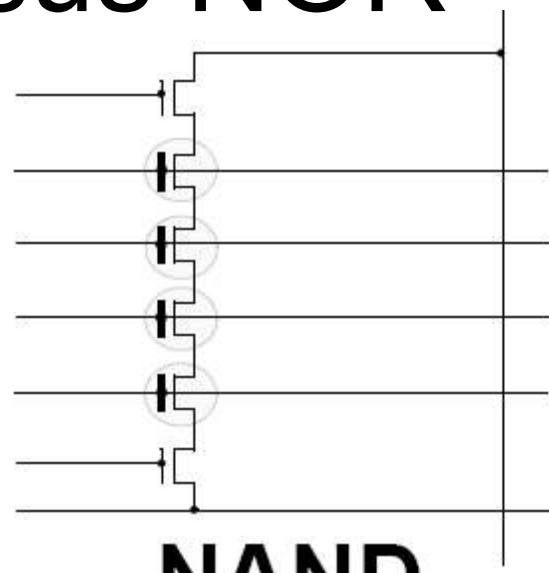
NOR Flash

- Larger memory cell area
- Byte accessible (good for ROM-like program storage, where instructions need to be read out on a byte-size granularity)
- Faster random byte access
- Longer erasing (~750 ms) and programming times (~.2 MB/sec sustained write speed)
- Shorter write lifetime
- iPhone uses both multi-GB NAND flash for multimedia file storage and multi-MB NOR flash for code/app storage

NAND versus NOR



NOR



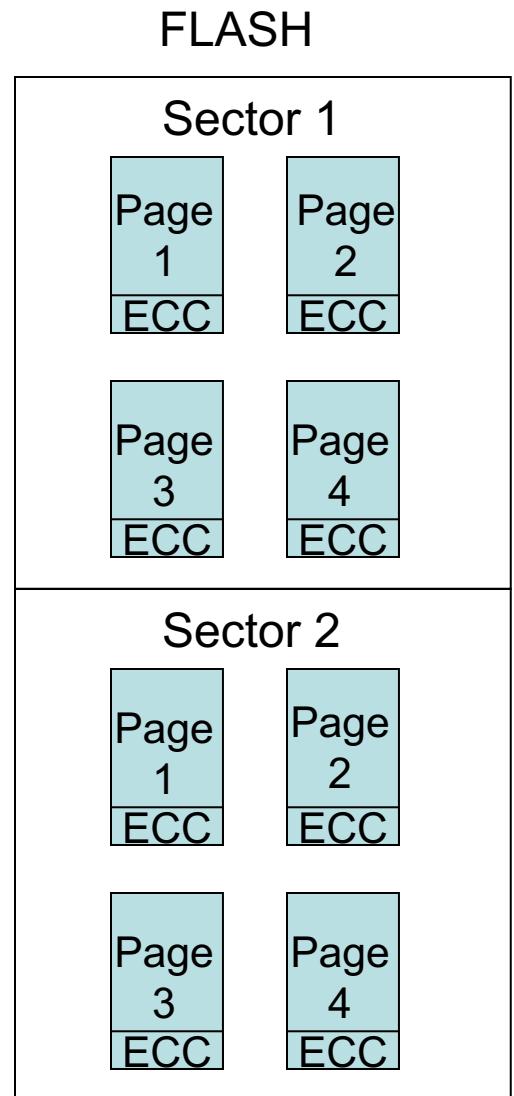
NAND

- 10x better endurance
- Fast read (~100 ns)
- Slow write (~10 µs)
- Used for Code

- Smaller cell size
- Slow read (~1 µs)
- Faster write (~1 µs)
- Used for Data

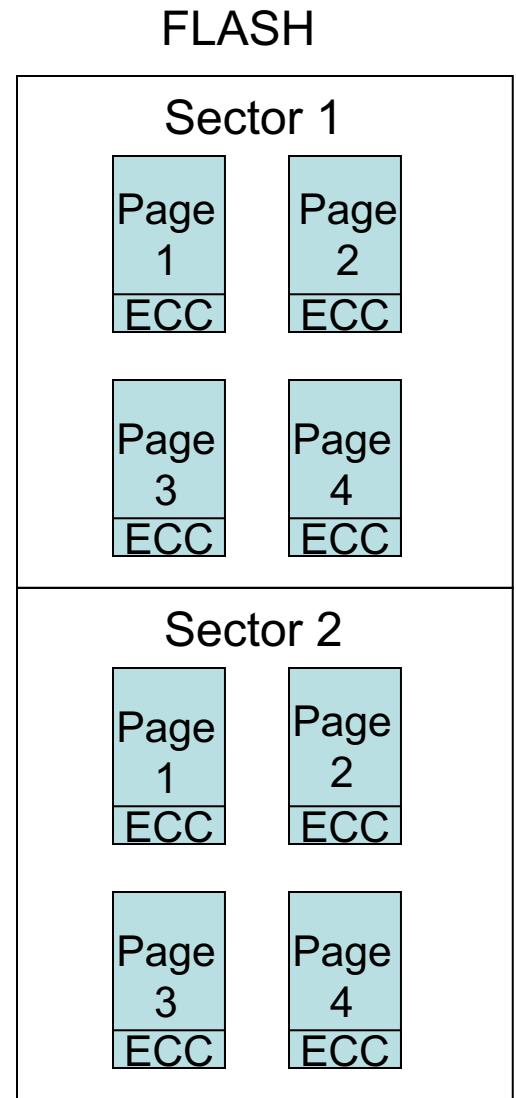
Organizing Flash Memory

- Flash memory is typically divided into blocks or sectors
 - There can be many *pages* per block/sector
 - e.g. 64 pages per block, 16 blocks per flash, and 4 KB per block => 4 MB of flash
 - The last 12-16 bytes of each page is reserved for error detection/correction (ECC) and bookkeeping
 - the flash file system can put information in this space



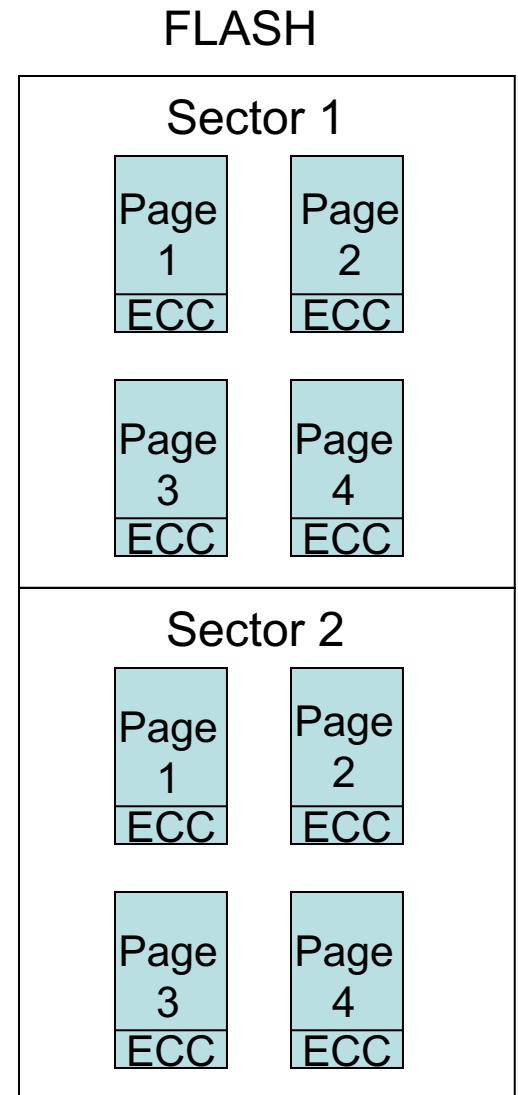
Operations on Flash Memory

- Reads and writes occur on a sub-page level granularity
 - Can read a byte (NOR) or word (NAND) by giving **page + offset**
- However, writes are tricky
 - Writes of a byte (NOR) or word (NAND) only proceed immediately if that memory has **been cleanly reset/erased and not yet written to**
 - “Rewrites” to memory that has already been written to **require an erase first before the write**



Rewrites Are Costly

- Unfortunately, erasing in flash can **only occur on a sector granularity!**
 - This is the price we pay for cheap “flash” technology
- Hence, to write just one byte into a memory location that has already been written to
 - First you have to erase the entire sector containing that byte address
 - Then write the byte!



Rewrites Are Costly

- Rewrites happen all the time.
 - Example: deleting a file
 - De-allocate the flash pages containing the file data
 - If we want to allocate these free flash pages to other files, we have to first erase the entire sectors containing these pages
- In comparison, the cost of rewrites is the same as writes for magnetic disk:
 - Can simply have magnetic head reset of the orientation of the bits on magnetic media
 - No need to erase before writing

Rewrites Are Costly

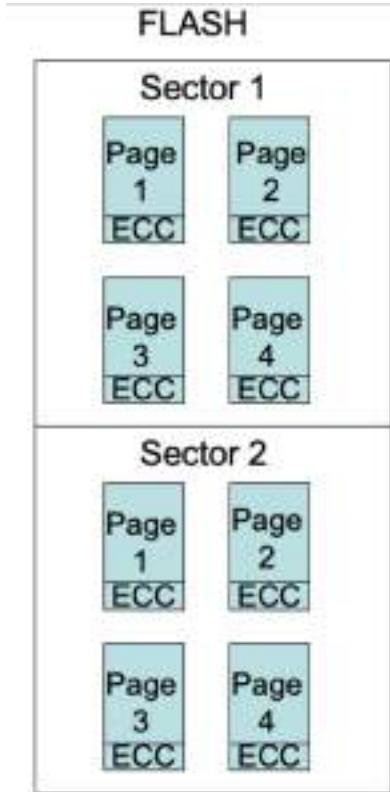
- To rewrite over a page of flash memory *while preserving* the other pages in the same sector:
 - Either:
 1. Copy the entire sector into RAM
 2. In RAM, write the changes to the page that you want to modify
 3. Erase the sector on flash
 4. Write all the sector's pages currently in RAM to the newly erased sector, including the modified page
 - Or:
 - Copy the entire sector into a new clean sector, except for the page to be modified
 - Write the one page to the correct location in the new sector (page is clean)

Limited Lifetime of Flash Memory

- Memory wear - Flash memory has a limited write lifetime
 - NOR flash memory can withstand 100,000 write-erase cycles before becoming unreliable
 - NAND flash memory can withstand 1,000,000 write-erase cycles
 - Reason: The strong electric field needed to set and reset bits eventually breaks down the silicon transistor
- Eventually, your mobile devices can no longer save data!

Wear Leveling Solution

- To extend the life of flash memory chips, write and erase operations are **spread out over the entire flash memory**
 - Keep a count in 12-16 byte trailer of each page of how many writes have been made to that page, and choose the free page with the lowest write count
 - Randomly scatter data over the entire flash memory span, since it doesn't take any longer to extract data from nearby pages as distant pages
 - Many of today's flash chips implement wear leveling in the hardware's device controller



Problems Applying Standard Disk File Systems to Flash Memory

1. A typical disk file system stores data/metadata in fixed locations
 - the directory structure, file headers, allocation structures (e.g. FAT) and free space structures (e.g. bitmap) and even file data are in relatively fixed locations
 - Repeated modifications to these structures in static locations on flash would result in rapidly exhausting the write lifetimes of those locations, hence all of flash

Problems Applying Standard Disk File Systems to Flash Memory

2. Disk file systems are not optimized for rewrites
 - Disk file system rewrites are viewed essentially the same as writes, and repeated rapid rewrites can be quickly done because there is little disk seek time
 - Flash rewrites incur a much greater penalty, are much longer than writes to clean flash pages, and repeated rapid rewrites in flash would cause disastrously long latency, not to mention reduced lifetime

Flash File Systems

- Must be designed to deal with 2 major problems of flash memory:
 - Rewrites requires erases – hold on to writes
 - Limited lifetime
- **Log-structured file systems are well-suited to flash memory!**
 - Writes to a file are added/appended to the end of the log, as are rewrites. The log is the file system.
 - The log grows sequentially and doesn't rewrite over the same location
 - Hence less erasing & also longer lifetime

Flash File Systems

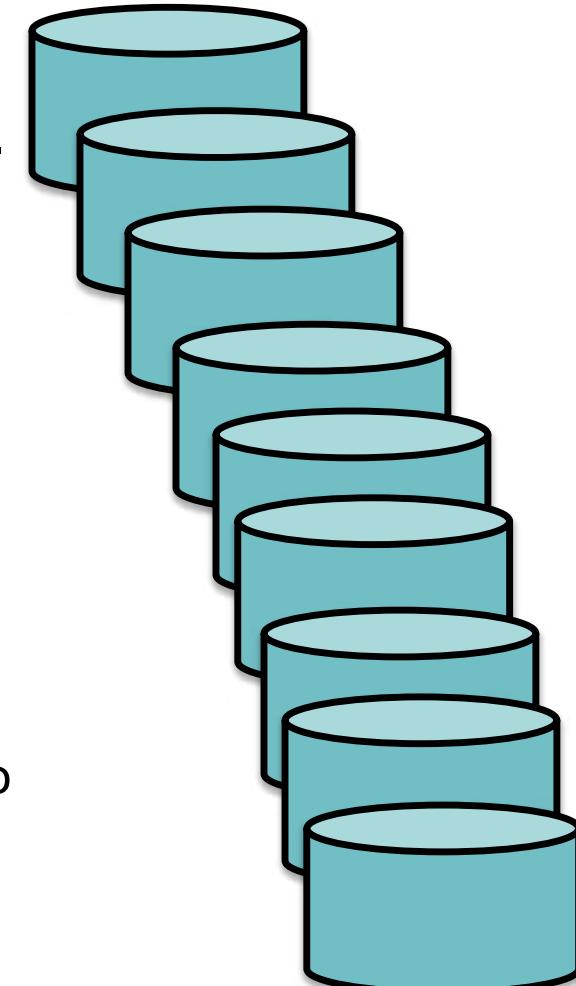
- Journaling Flash File System (JFFS) and JFFS2
 - Is a Log-structured file system similar to log-based recovery seen earlier for file system reliability
 - The file system is the log, and is written sequentially over flash pages,
 - The entries in the log contain all the data we need to reconstruct the file (recall the X & Y old and new values)
 - The log may be circular, eventually wrapping around once all free flash pages have been consumed.
 - Free space = distance between head and tail of log
 - As free space decreases, garbage collection is performed to free up some more space.



Redundant Arrays of Inexpensive Disks (RAID)

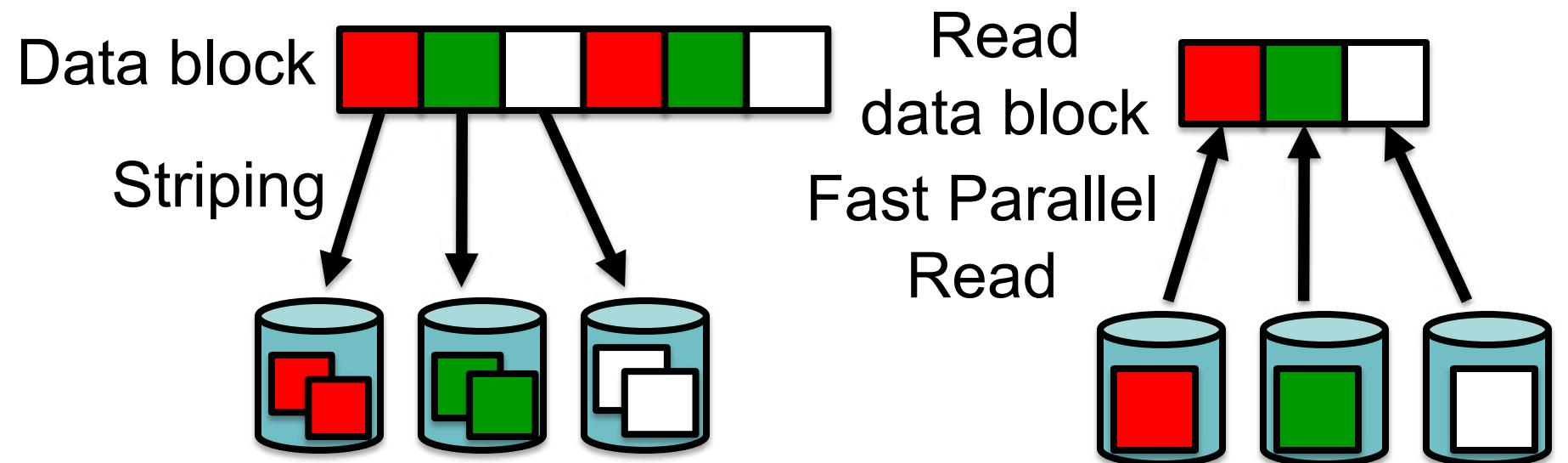
Redundant Arrays of Inexpensive Disks (RAID)

- Magnetic disks are cheap these days.
- Attaching an array of magnetic disks to a computer brings several advantages compared to a single disk:
 - Faster read/write access to data by having multiple reads/writes in parallel.
 - Data is striped across different disks, e.g. each byte of an 8-byte word is striped onto a different disk
 - Better fault tolerance/reliability
 - if one disk fails, a copy of the data could be stored on another disk – redundancy to the rescue!



RAID0

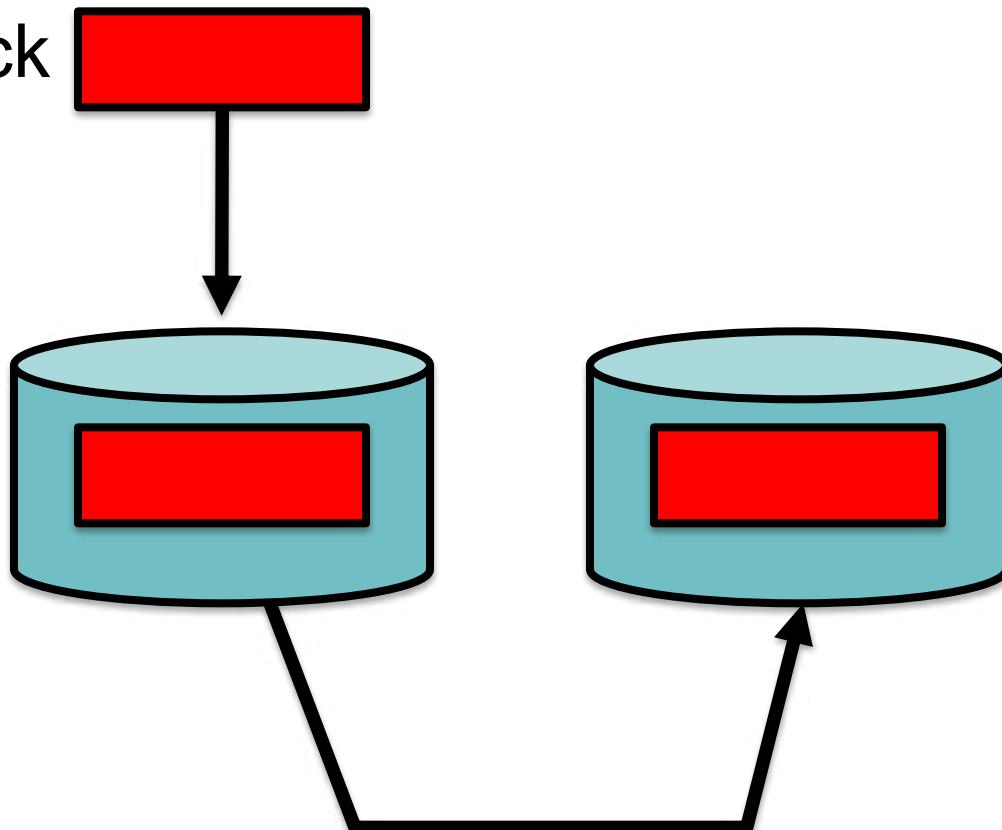
- RAID has different levels with increasing redundancy
 - RAID0 = data striping with no redundancy



RAID1

– RAID1 = mirror each disk

Data block



Mirroring

Get redundancy,
but not parallel
performance
speedup of
RAID0.

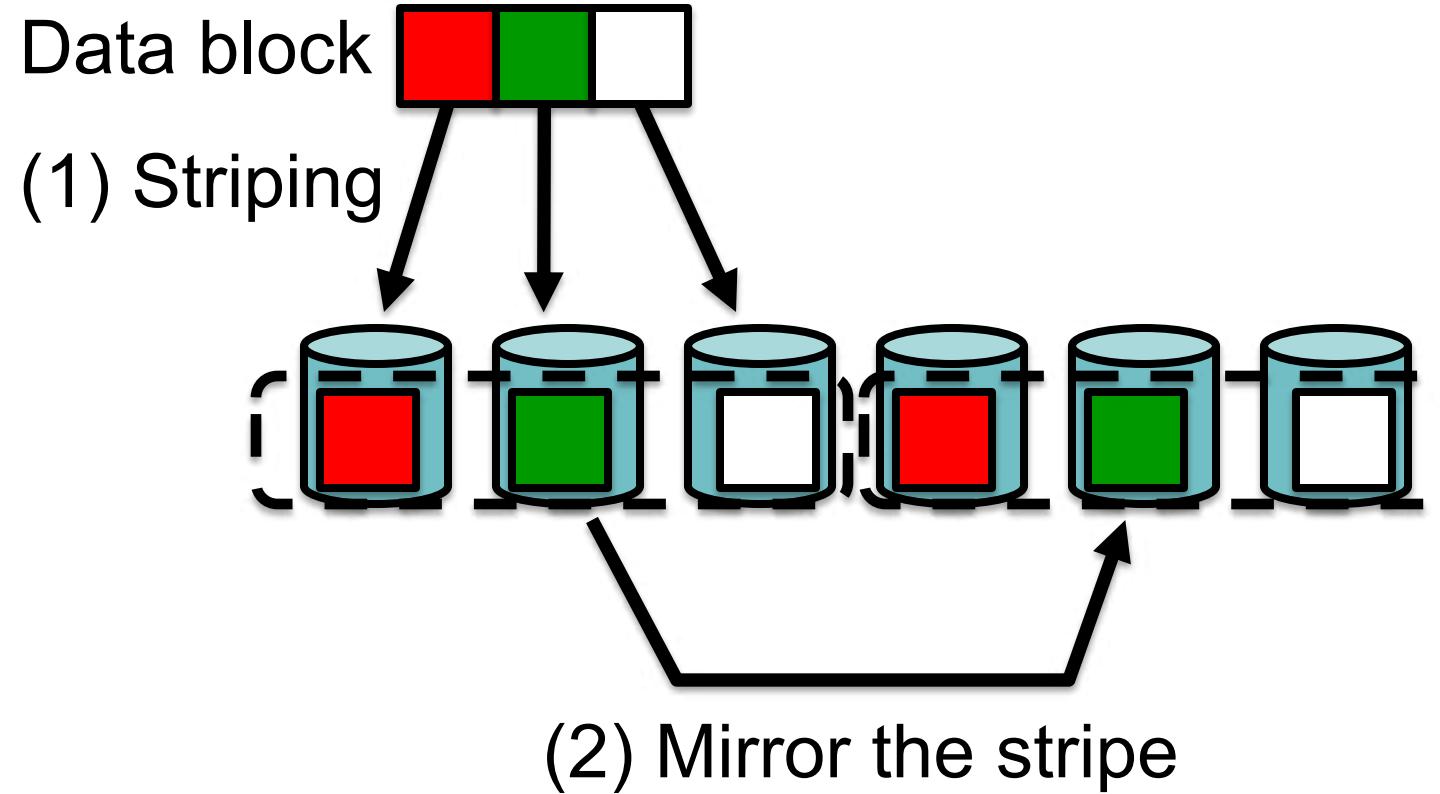
[Combine RAID0
with RAID1](#)

Can get limited
read speedup by
submitting the
read to all mirrors,
and taking the 1st
data result

Writes delay x 2
on synchronous
write , but OS can
mask this by
delaying writes

RAID0+1

– RAID0+1



RAID 0+1 = RAID 0 data striping for performance + RAID 1 mirroring of stripes for redundancy

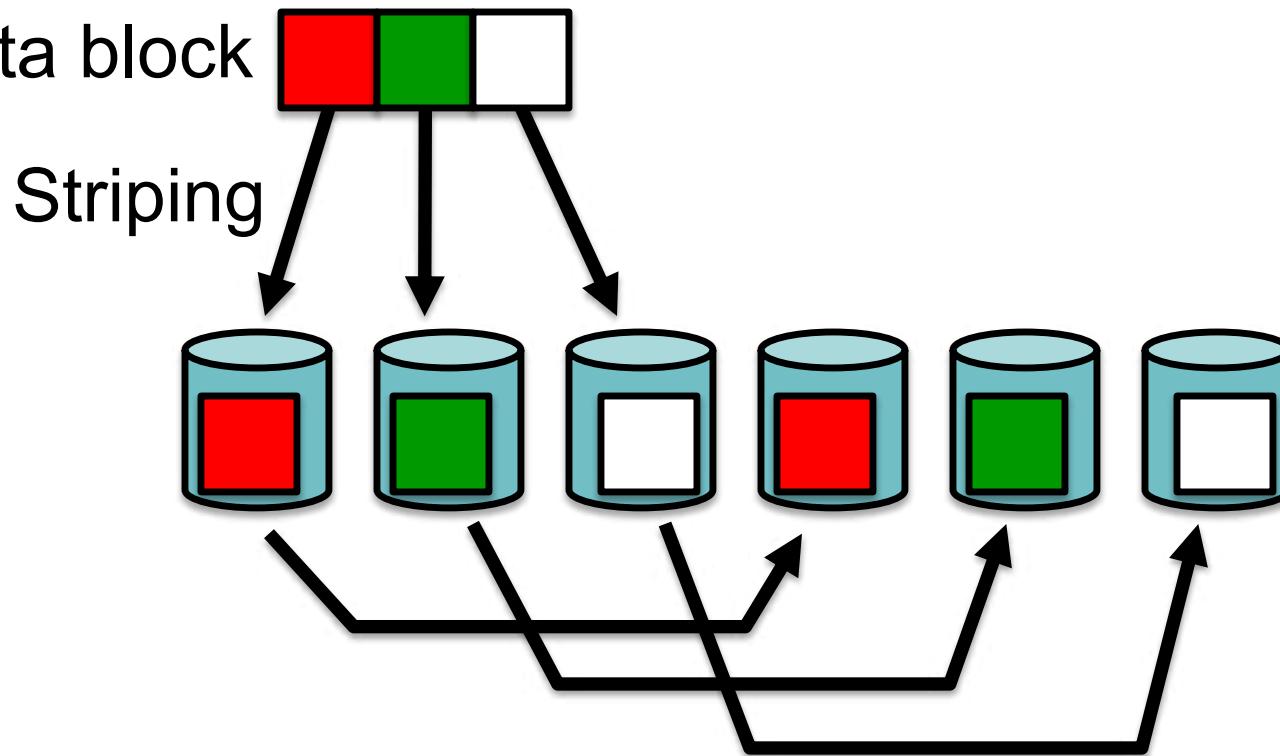
Note: if both primary stripe and second stripe fail then the entire data block is lost

RAID1+0

– RAID1+0, aka “RAID 10”

RAID 1+0 = mirror each disk then stripe.

Data block



Striping

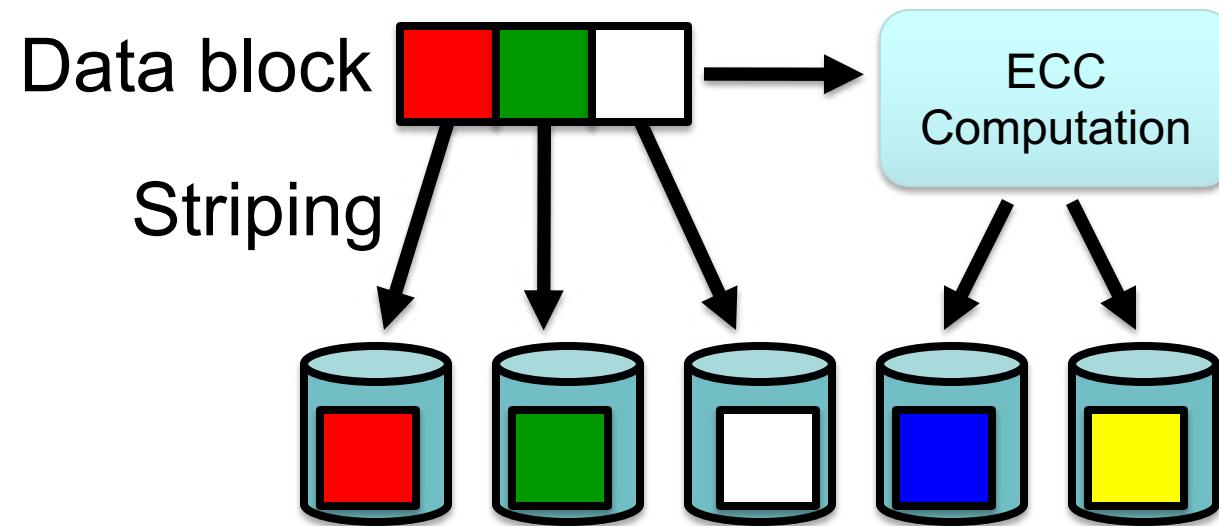
Any two disks may fail, and the data can still be retrieved, unless the two disks mirror the same data is failed

Thus, RAID 1+0 is more robust than RAID 0+1.

(1) Mirroring each disk

RAID2

- RAID2 = put Error Correction Code (ECC) bits on other disks

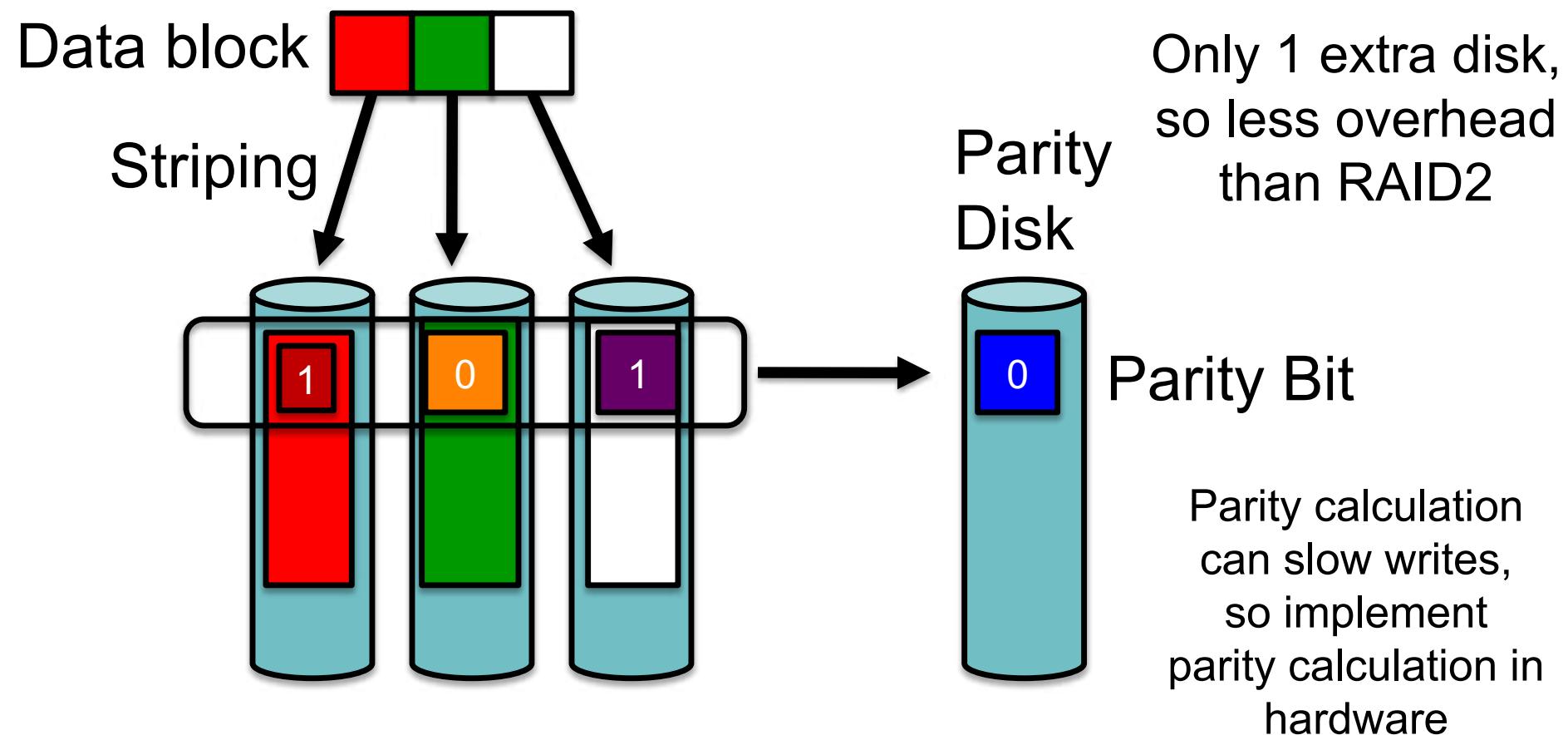


Error correction codes are more compact than just copying the data, and provide strong statistical guarantees against error

e.g. a crashed disk's lost data can be corrected with the redundant data stored in the ECC disks

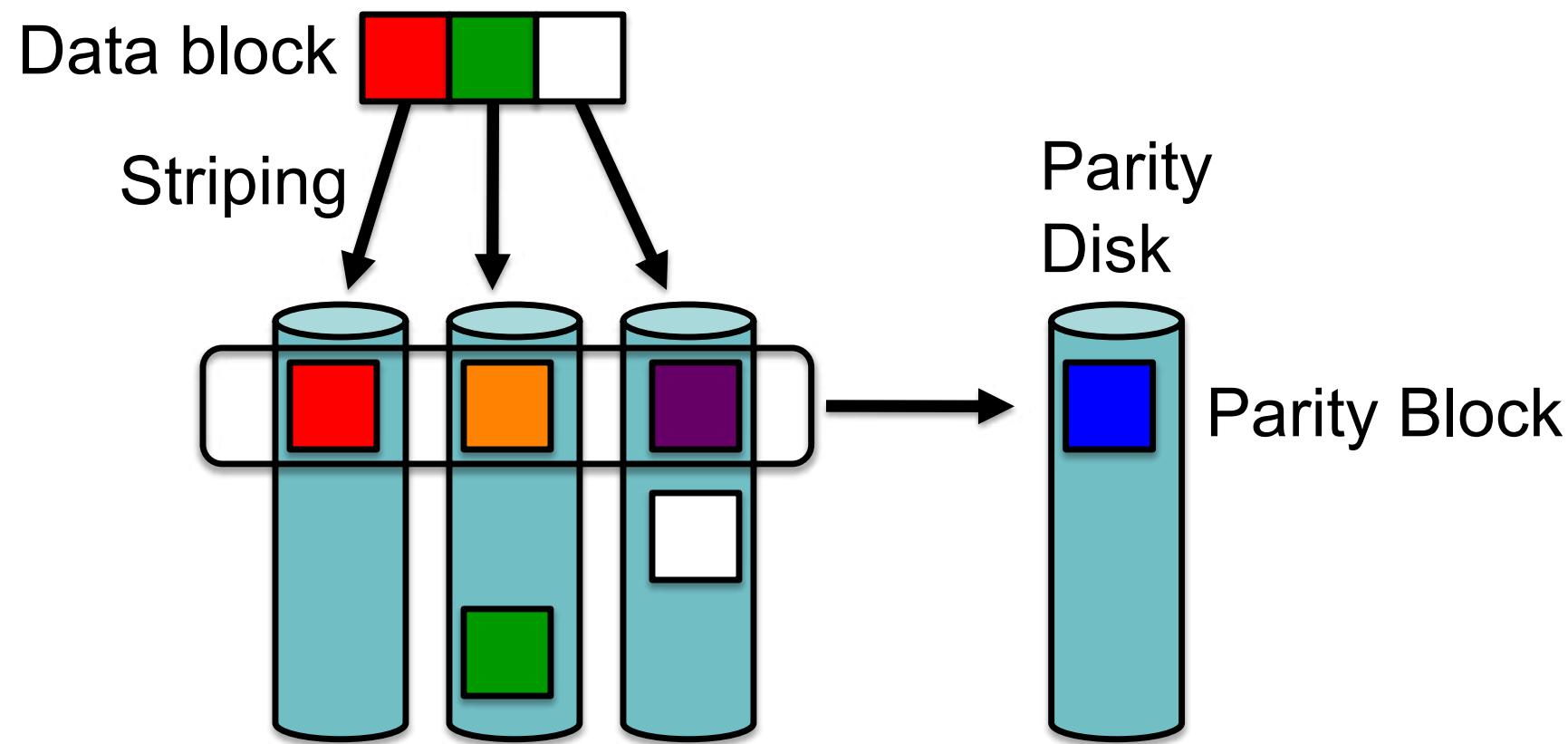
RAID3

- RAID3 = bit-interleaved parity: for each bit at the same location on different disks, compute a parity bit, that is stored on a parity disk



RAID4

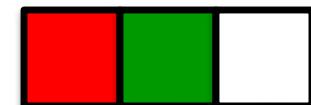
- RAID4 = block-interleaved parity: for each block at the same location on different disks, compute a parity block, that is stored on a parity disk



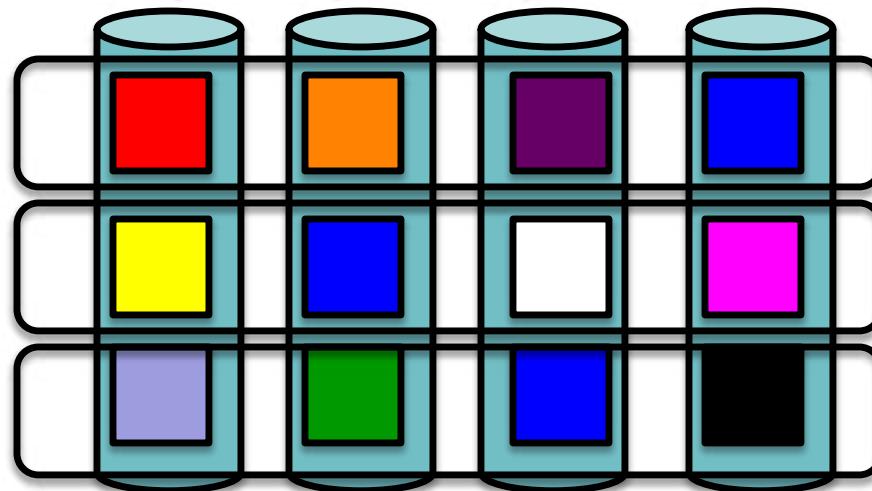
RAID5

- RAID5 = block-interleaved *distributed* parity: spread the parity blocks to different disks, so every disk has both data and parity blocks

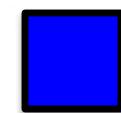
Data block



Striping



Avoids overuse of RAID4's parity disk, and is the most common parity RAID system

 = Parity Blocks

RAID6 replaces parity blocks with stronger Reed-Solomon ECC for multiple disk failures

RAID Implementation

- Three methods:
 - Software Based,
 - Firmware Based,
 - Hardware Based
- 1. Software Based: Plug in disks to your computer bus and implement RAID management in software in OS kernel
 - The RAID software layer sits above the device drivers
 - Cost effective and easy to implement, but not as efficient
 - In Linux, use the mdadm package to manage RAID arrays.
 - Cannot boot from RAID

RAID Implementation

2. Firmware/Driver Based: Plug in disks to your computer's I/O bus, and **an intelligent hardware RAID controller** recognizes them and integrates them into a RAID system automatically
 - Standard disk controller chips with special firmware and drivers
 - RAID instructions are stored in the firmware of the device
 - Can boot from RAID: During startup/boot, the RAID is essentially kick-started by the firmware
 - CPU still need to handle it => Costly

RAID Implementation

3. Hardware Based: Plug in disks to a RAID array, which is a stand-alone hardware unit with a RAID controller that looks like one physical device, e.g. SCSI, to your computer bus
 - File system doesn't have to know about RAID to use and benefit from this RAID disk array
 - Expensive but highly efficient

Summary

- Storage management
 - Magnetic
 - Disk Scheduling
 - Flash
 - Page/Sector management
 - RAID
 - Reliability vs. access time.



Lecture 19

Paging Policies



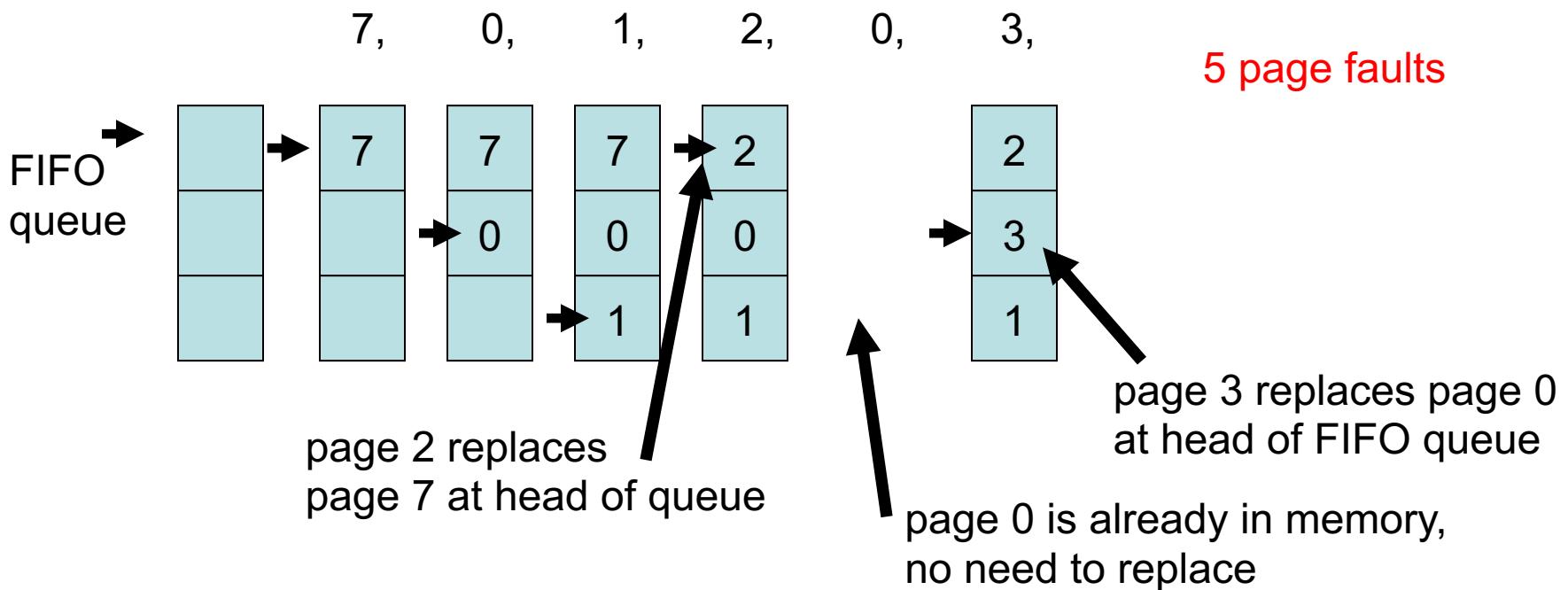
University of Colorado
Boulder

Page Replacement Policies

- Evaluation
 - Variables: algorithm, page reference sequence, # of memory frames
 - algorithm with lowest # of page faults is most desirable

FIFO Page Replacement

- FIFO - create a FIFO queue of all pages in memory
 - example reference string: 7, 0, 1, 2, 0, 3, ...
 - assume also that there are 3 frames of memory total

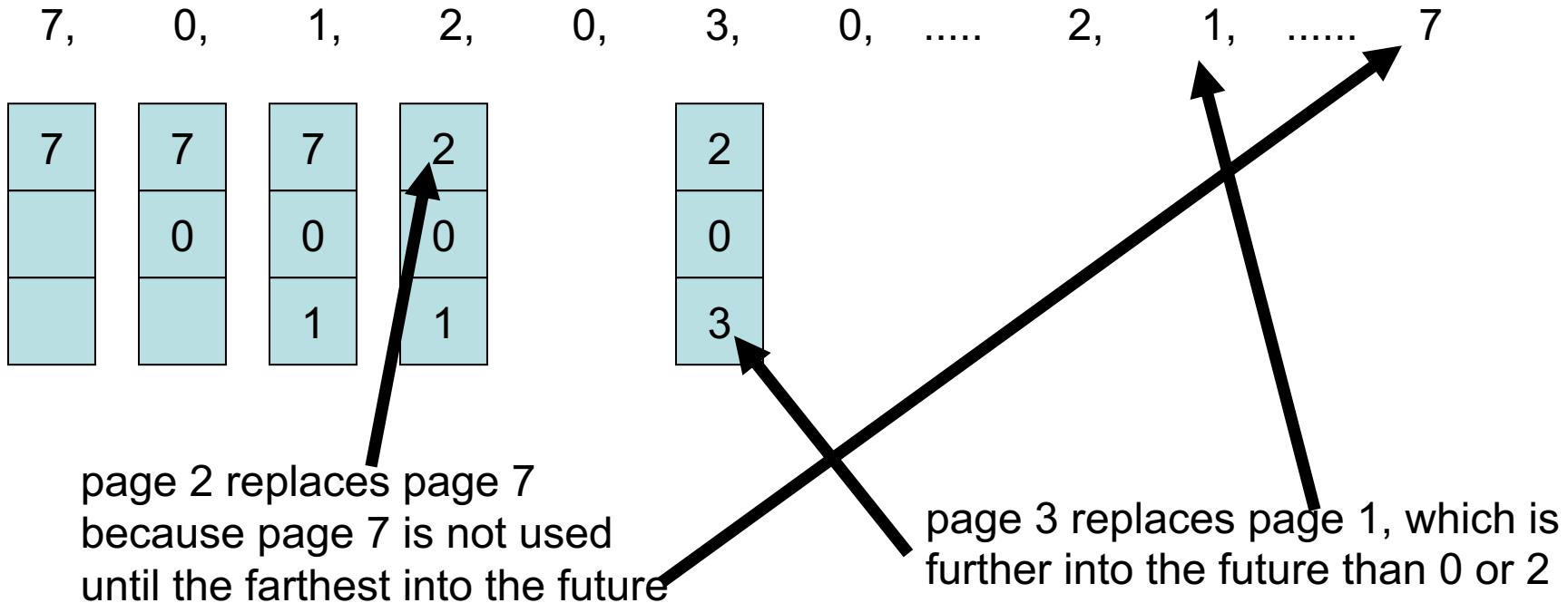


FIFO Page Replacement

- FIFO is easy to understand and implement
- Performance can be poor
 - Suppose page 7 that was replaced was a very active page that was frequently referenced, then page 7 will be referenced again very soon, causing a page fault because it's not in memory any more
 - In the worst case, each page that is paged out could be the one that is referenced next, leading to a high page fault rate
 - Ideally, keep around the pages that are about to be used next – this is the basis of the OPT algorithm in the next slide

OPT Page Replacement

- OPT = Optimal
 - Replace the page that will not be referenced for the longest time
 - Guarantees the lowest page-fault rate
 - Problem: requires future knowledge

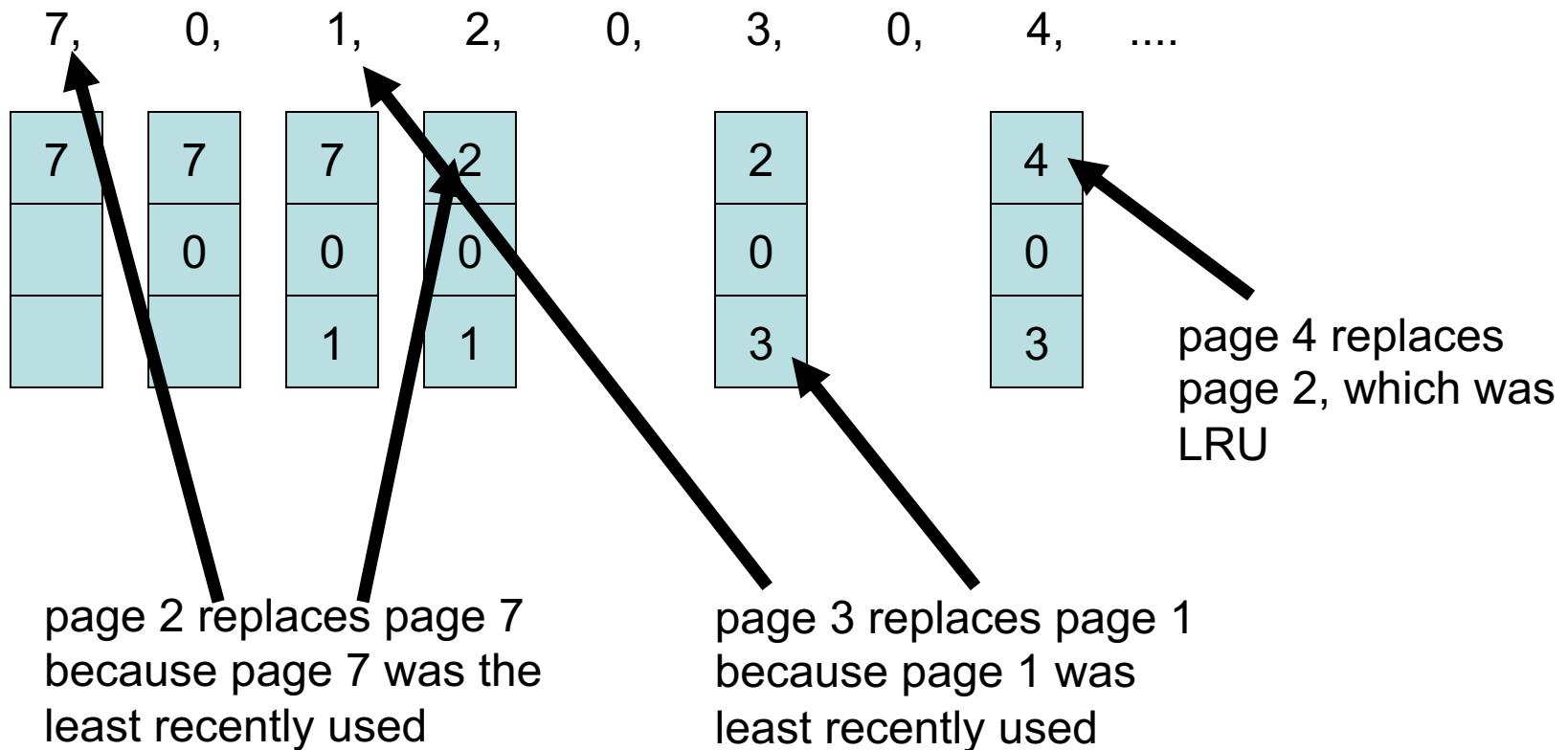


LRU Page Replacement

- LRU = Least Recently Used
 - Use the past to predict the future
 - if a page wasn't used recently, then it is unlikely to be used again in the near future
 - if a page was used recently, then it is likely to be used again in the near future
 - so select a victim that was least recently used
 - Approximation of OPT
 - page fault rate LRU > OPT, but LRU < FIFO
 - Variations of LRU are popular

LRU Page Replacement

- LRU example



LRU Implementation Options

- Keep a history of past page accesses
 - the entire history (lots of memory)
 - a sliding window
 - Complicated and slow
- Variations of LRU are popular

LRU Implementation Options

- Timers
 - keep an actual time stamp for each page as to when it was last used
 - Problem: expensive in delay (consult system clock on each page reference), storage (at least 64 bits per absolute time stamp), and search (find the page with the oldest time stamp)

LRU Implementation Options

- Counters
 - Approximate time stamp in the form of a counter that is incremented with any page reference, i.e. each page's counter must be incremented on each page reference
 - Counter is stored with that entry in the page table.
 - Counter is reset to 0 on a reference to a page.
 - Problem: expensive
 - Must update each page's counter (on each reference)
 - Must search list

LRU Implementation Options

- Linked List
 - whenever a page is referenced, put it on the end of the linked list, removing it from within the linked list if already present in list
 - Front of linked list is LRU
 - Problem: managing a (doubly) linked list and rearranging pointers becomes expensive
- Similar problems with a Stack data structure

LRU approximation algorithms

- Add an extra HW bit called a *reference bit*
 - This is set any time a page is referenced (read or write)
 - Allows OS to see what pages have been used, though not fine-grained detail on the order of use
 - Reference-bit based algorithms only *approximate* LRU, i.e. they do not seek to exactly implement LRU
 - 3 types of reference-bit LRU approximation algorithms:
 - Additional Reference-Bits Algorithm
 - Second-Chance (Clock) Algorithm
 - Enhanced Clock Algorithm with Dirty/Modify Bit

LRU approximation algorithms

- Additional Reference-Bits Algorithm
 - Record the last 8 reference bits for each page
 - Periodically a timer interrupt shifts right the bits in the record and puts the reference bit into the MSB
 - Example to compare times:
 - $11000100 > 01110111$
 - So LRU = lowest valued record

	bits							
page	0	1	2	3	4	5	6	7
0	1							
1								
2	1							

	bits							
page	0	1	2	3	4	5	6	7
0	1	1						
1	1							
2	1	1						

Periodically shift all the bit to the right

Reference-bit based LRU approximation algorithms

- Second-Chance Algorithm

- In-memory pages + reference bits conceptually form a *circular queue*; a hand points to a page.

- If page pointed to has $R = 0$, it is replaced and the hand moves forward.

- Otherwise, set $R = 0$; hand moves forward and checks the next page.

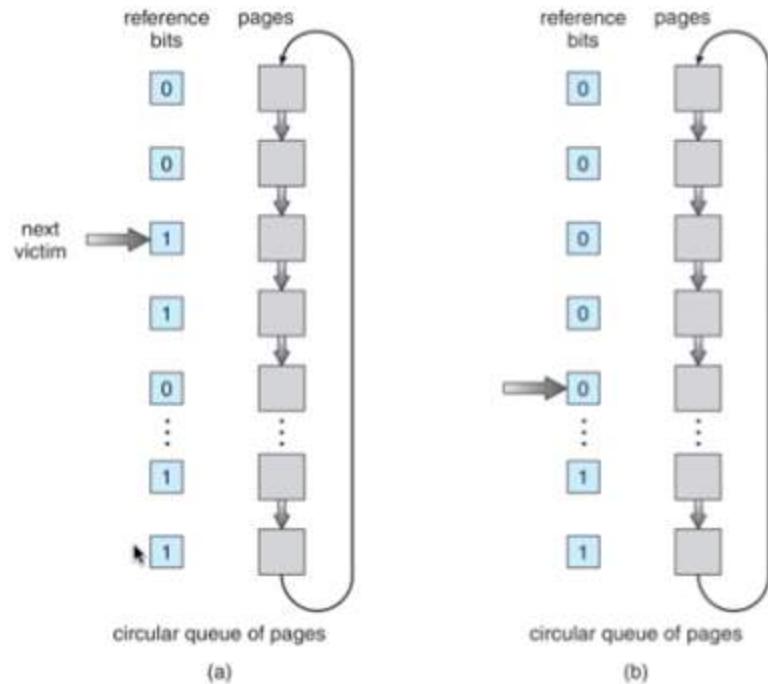


Figure 9.17 Second-chance (clock) page-replacement algorithm.

Second-Chance (Clock) Algorithm

- Advantages:
 - simple to implement (one pointer for the clock hand + 1 ref bit/page.
 - Note the circular buffer is actually just the page table with entries where the valid bit is set, so no new circular queue has to be constructed),
 - fast to check reference bit
 - usually fast to find first page with a 0 reference bit
 - approximates LRU
- Disadvantages:
 - in the worst case, have to rotate through the entire circular buffer once before finding the first victim frame

Counting-Based Page Replacement

- Keep a counter of number of page accesses for each page since its introduction, which is an activity or popularity index
- **Most Frequently Used**
 - Replace page with highest count
 - Assumes the smallest counts are most recently loaded
- **Least Frequently Used**
 - Replace page with lowest count
 - What if a page was heavily used in the beginning, but not recently?
 - Age the count by shifting its value right by 1 bit periodically - this is exponential decay of the count.
- **These kinds of policies are not commonly used**

Approximation of OPT Algorithm

- We use the past to approximate the future
- We can be smarter about looking at the past
 - Not just a sequence, but a pattern of use
 - Use this to create a better approximation of page use
 - Take advantage of the locality
 - Create a *working set* of pages



Belady's Anomaly

FIFO Page Replacement: Another example

Let page reference stream, $R = 012301401234$

Frame	0	1	2	3	0	1	4	0	1	2	3	4
0	<u>0</u>	0	0	<u>3</u>	3	3	<u>4</u>	4	4	4	4	4
1		<u>1</u>	1	<u>1</u>	<u>0</u>	0	<u>0</u>	0	0	<u>2</u>	2	2
2			<u>2</u>	2	<u>2</u>	<u>1</u>	1	1	1	1	<u>3</u>	3

- FIFO with $m = 3$ has 9 faults
- Goal: To reduce the number of page fault
 - Increase the size of memory

FIFO Page Replacement: Another example

Let page reference stream, $R = 012301401234$

Frame	0	1	2	3	0	1	4	0	1	2	3	4
0	<u>0</u>	0	0	0	0	<u>4</u>	4	4	4	<u>3</u>	3	3
1		<u>1</u>	1	1	1	1	<u>1</u>	<u>0</u>	0	0	<u>0</u>	<u>4</u>
2			<u>2</u>	2	2	2	2	<u>2</u>	<u>1</u>	1	1	1
3				<u>3</u>	3	3	3	3	<u>3</u>	<u>2</u>	2	2

- FIFO with $m = 4$ has 10 faults

Belady's anomaly: Increasing the size of memory may result in increasing the number of page faults for some programs.

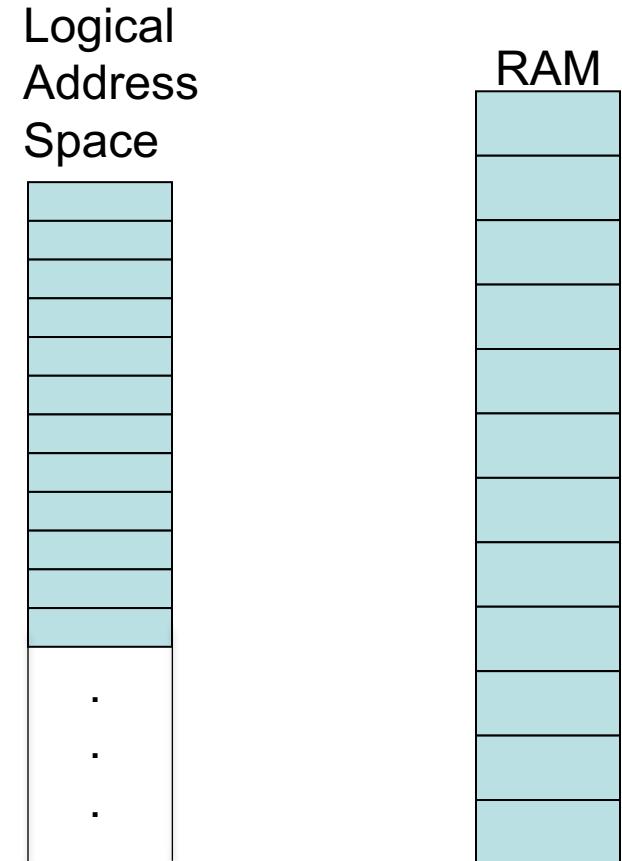
Stack Algorithms

- Stack algorithms are a class of page replacement algorithms that do not suffer from Belady's anomaly
- **Key property:**
 - Set of pages in memory for n frames is always a subset of the set of pages that would be in memory with $n+1$ frames, irrespective of the page reference string
- **OPT and LRU are stack algorithms, while FIFO is not a stack algorithm**
 - The OPT and LRU algorithms will always keep a subset of the pages used in a larger number of frames
 - FIFO may end up with a different set of pages depending on the number of frames



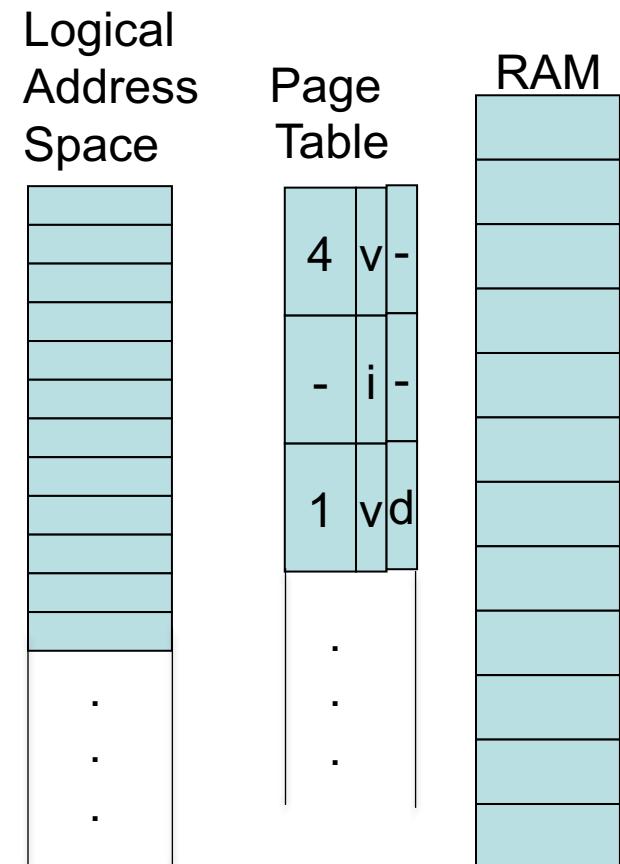
Frame Allocation

Techniques to Improve Page Replacement Performance



Techniques to Improve Page Replacement Performance

1. Use a **dirty/modify bit** to reduce disk writes
2. Choose a **smart page replacement algorithm**
 - keeps the most important pages in memory and evicts the least important
3. Make the **search for the least important page be fast**



Techniques to Improve Page Replacement Performance

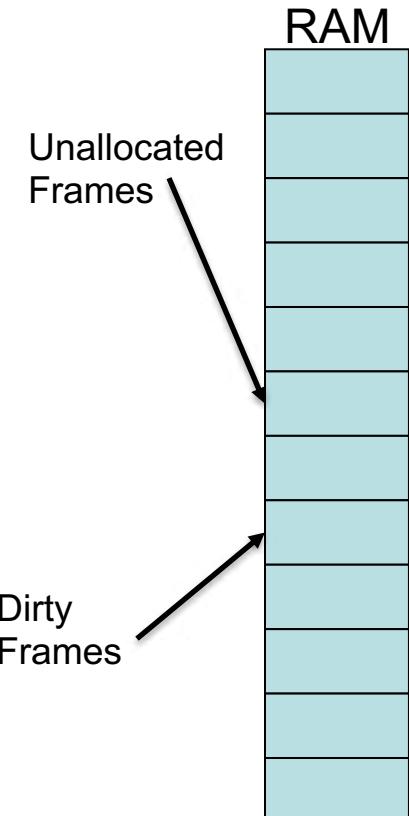
4. Page-buffering

- read in a frame first and start executing, then select a victim and write it out at a later time - faster perceived performance
- periodically write out all modified frames to disk when no other activity. Thus most frames are clean so few disk writes on a page fault.

5. Keep a pool of free frames and remember their content.

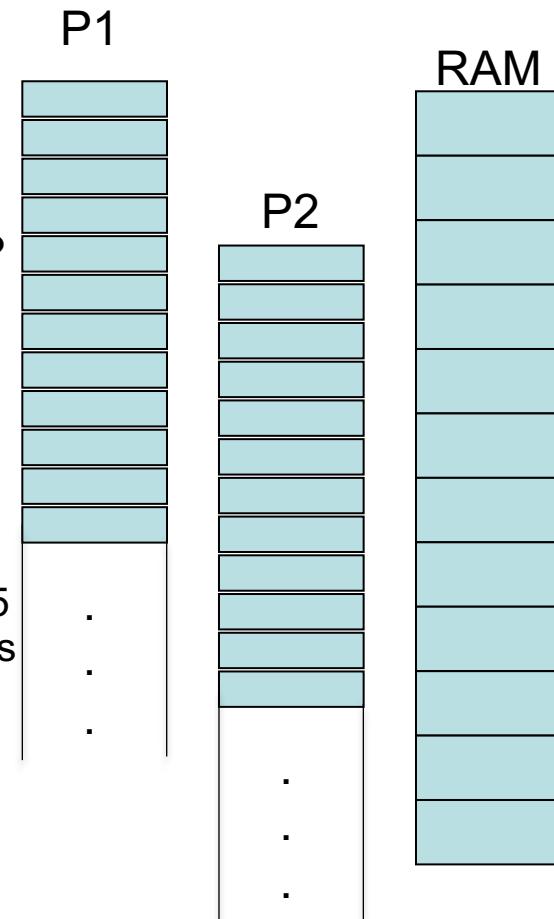
- Reuse this free frame if the same contents are needed again. Reduces disk reads.

6. Allocate the appropriate # of frames so a process avoids thrashing – we'll see this next



Allocation of Memory Frames

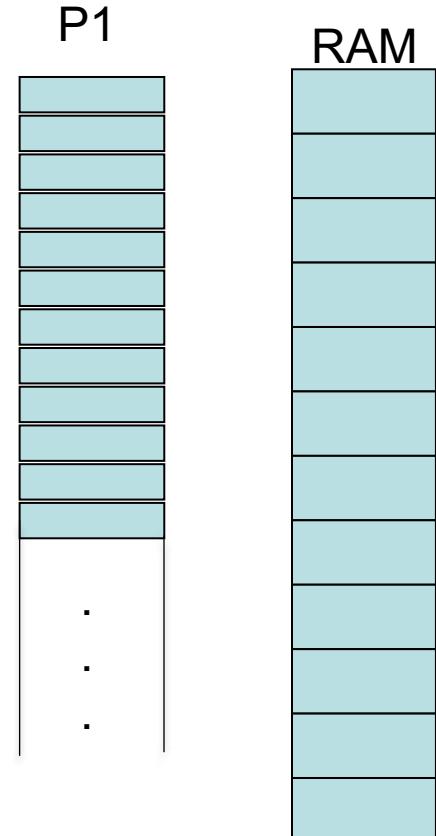
- Given that the OS employs paging for memory management, then physical memory is divided into fixed-sized frames or pages.
- How many frames does each process get allocated? How many frames does the OS get allocated versus user processes?
 - Variety of policies:
 - based on number of frames
 - based on whether frames are allocated locally or globally
 - Example: given the OS, and processes P1 and P2, and 15 frames of physical memory, how do we allocate the frames among these three entities?



Memory Allocation Policies

1. Minimum # frames:

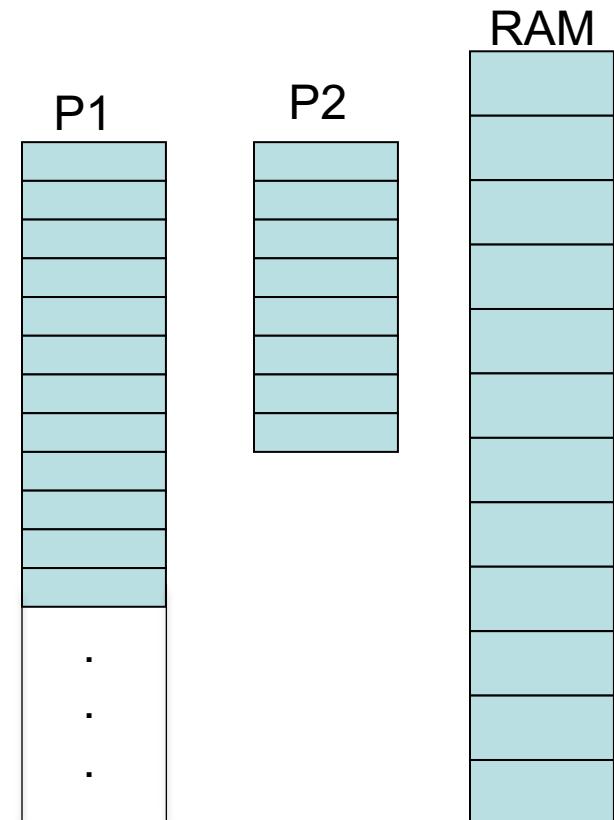
- determine the minimum # of frames that allow a process to allocate. Ideally, this is just one page, i.e. the page in which the program counter is currently executing.
Or one for code, one for data
- In practice, some CPU's support complex instructions.
 - multi-address instructions. Each address could belong to a different page.
- Also, there can be multiple levels of indirection in the addressing, i.e. pointers.
 - Each such level of indirection could result in a different page being accessed in order to execute the current instruction. Up to N levels of indirection may be supported, which means may need up to N pages as the minimum.



Memory Allocation Policies

2. Equal allocation:

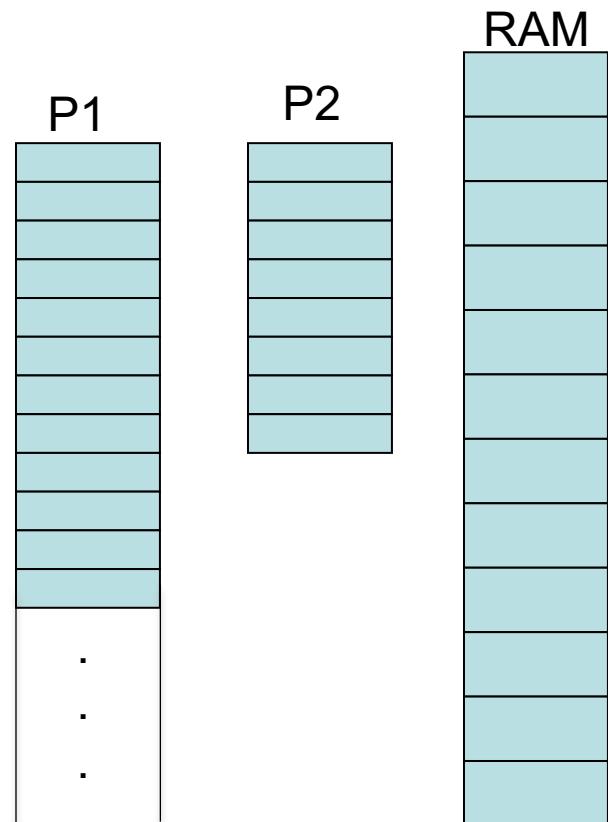
- split m frames equally among n process
- m/n frames per process
- problem: doesn't account for size of processes, e.g. a large database process versus a small client process whose size is $\ll m/n$
- needs to be adaptive as new process enter and the value of n fluctuates



Memory Allocation Policies

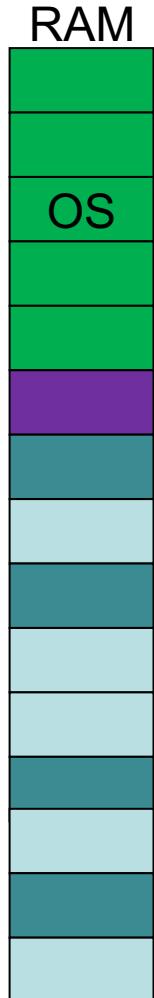
3. Proportional allocation

- allocate the number of frames relative to the size of each process
- Let S_i = size of process P_i
- $S = \sum S_i$
- Allocate $a_i = (S_i / S) * m$ frames to process P_i
- proportion a_i can vary as new processes start and existing processes finish
- Also, if size is based on the code size, or address space size, then that is not necessarily the number of pages that will be used by a process



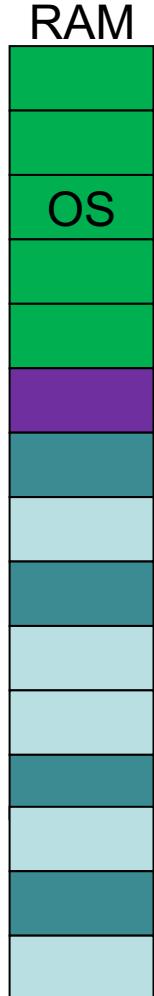
Local vs Global Allocation/Replacement

- In local allocation/page replacement, a process is assigned a fixed set of N memory pages for the lifetime of the process
 - When a page needs to be replaced, it is chosen only from this set of N pages
 - Easy to manage
 - Processes isolated from each other (not fully true)
 - Windows NT follows this model



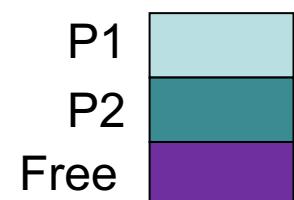
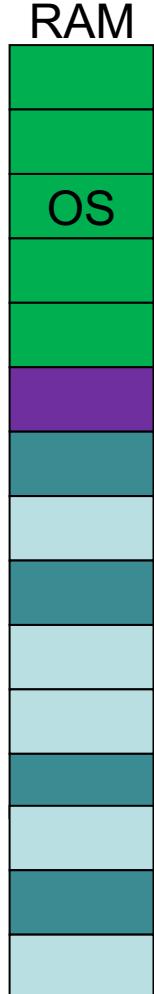
Local vs Global Allocation/Replacement

- Problems with local allocation:
 1. the behavior of processes may change as they execute
 - Sometimes they'll need more memory, sometimes less
 - local replacement doesn't allow a process to take advantage of unused pages in another process
 - Want a more adaptive allocation strategy that would allow a page fault to trigger the page replacement algorithm to increase its page allocation



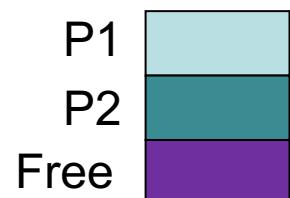
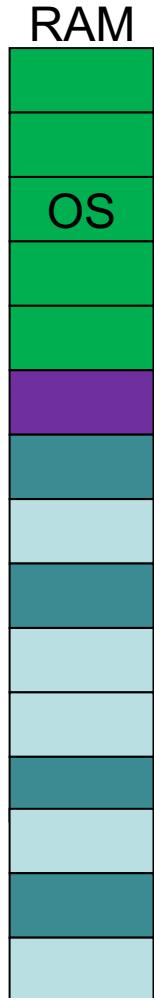
Local vs Global Allocation/Replacement

- Problems with local allocation:
 2. Local replacement would seem to isolate a process's paging behavior from other processes
 - Amount of memory allocated to each process is fixed
 - However, isolation is not perfect:
 - If a process P1 page faults frequently, then P1 will queue up many requests to read/write from/to disk.
 - If another process P2 needs to access the disk, e.g. to page fault, then even if P2 page faults less frequently, P2 will still be slowed down by P1's many queued up reads and writes to disk



Local vs Global Allocation/Replacement

- Global allocation and page replacement
 - Pool all pages from all processes together
 - When a page needs to be replaced/evicted, choose it from the global pool of pages
 - Linux follows this model
 - Global allocation and page replacement allows the # pages allocated to a process to fluctuate dramatically
 - Good: system adapts to let a process utilize “unused” pages of another process, leading to better memory utilization overall and better system throughput
 - Bad: a process cannot control its own page fault rate under global replacement, because other processes will take its allocated frames, potentially increasing its own page fault rate





Thrashing



University of Colorado
Boulder

Memory Management Thrashing

FIFO Page
Replacement
Example

Let page reference stream, $R = 012301401234$

Frame	0	1	2	3	0	1	2	3	0	1	2	3
0	<u>0</u>	0	0	<u>3</u>	3	3	<u>2</u>	2	2	<u>1</u>	1	1
1		<u>1</u>	1	1	<u>0</u>	0	<u>0</u>	<u>3</u>	3	<u>3</u>	<u>2</u>	2
2			<u>2</u>	2	2	<u>1</u>	1	1	<u>0</u>	0	0	<u>3</u>

- FIFO with $m = 3$ has 12 faults
- Every request is causing a page fault
- This is an extreme example of page thrashing

Thrashing

- **Describes situation of repeated page faulting**
 - this significantly slows down performance of the entire system, and is to be avoided
- **occurs when a process' allocated # of frames < size of its recently accessed set of frames**
 - each page access causes a page fault
 - must replace a page, load a new page from disk
 - but then the next page access also is not in memory, causing another page fault, resulting in a domino effect of page faults - this is called *thrashing*
 - a process spends more time page faulting to disk than executing – is I/O bound, causing a severe performance penalty

Thrashing

- Thrashing under global replacement
 - a process needs more frames, page faults and takes frames away from other processes, which then take frames from others, ... - domino effect
 - as processes queue up for the disk, CPU utilization drops
 - The process isolation is not perfect. A process's disk reads and writes will slow down other processes that also have disk I/Os
 - OS can add fuel to the fire:
 - suppose OS has the policy that if it notices CPU utilization dropping, then it thinks that the CPU is free, so it restarts some processes that have been frozen in swap space
 - these new processes page fault more, causing CPU utilization to drop even further

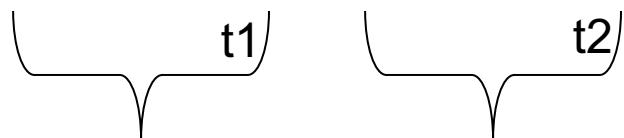
Thrashing

- **Solution 1: Build a *working set* model.**
 - Programs tend to exhibit *locality* of behavior, i.e. they tend to access/reuse same code/data pages
 - Find a set of recently accessed pages that captures this locality
 - To measure locality, define a window size Δ of the Δ most recent page references
 - Within the set of pages in Δ , the working set is the set of *unique* pages
 - pages recently referenced in working set will likely be referenced again
 - Then allocate to a process the size of the working set

Thrashing

- working set solution (continued):
 - Example: Assume $\Delta = 4$ and we have a page reference sequence of

2, 7, 3, 2, 0, 1, 2, 1, 2, 1, 0, 5, ...



working set at t1,
WS1 = {0,2,3,7}
working set size = 4

working set at t2, WS2 = {1,2}
working set size = 2

- at time t1, process only needs to be allocated 4 frames; at time t2, process only needs 2 frames
- Is $\Delta = 4$ the right size window to capture the locality?

Thrashing

- working set solution (cont.)
 - Choose window Δ carefully
 - if Δ is too small, Δ won't capture the locality of a process
 - if Δ is too large, then Δ will capture too many frames that aren't really relevant to the local behavior of the process
 - which will result in too many frames being allocated to the process

Thrashing

- working set solution (cont.)
 - Once Δ is selected, here is the working set solution:
 1. Periodically compute a working set and working set size WSS_i for each process P_i
 2. Allocate at least WSS_i frames for each process
 3. Let demand $D = \sum WSS_i$. If $D > m$ total # of free frames, then there will be thrashing. So swap out a process, and reallocate its frames to other processes.
 - This working set strategy limits thrashing

Thrashing

- **Approximate working set with a timer & a reference bit**
 - set a timer to expire periodically
 - any time a page is accessed, set its reference bit
 - at each expiration of the timer, shift the reference bit into the most significant bit of a record kept with each page,
 - a byte records references in the last 8 timer epochs
 - If any reference bit is set during the last N timer intervals, then the page is considered part of the working set, i.e. if record > 0
 - re-allocate frames based on the newly calculated working sets. Thus, it is not necessary to recalculate the working set on every page reference.

Thrashing

- **Solution 2: Instead of using a working set model, just directly measure the page fault frequency (PFF)**
 - When PFF > upper threshold, then increase the # frames allocated to this process
 - When PFF < lower threshold, then decrease the # frames allocated to this process (it doesn't need so many frames)
 - Windows NT used a version of this approach



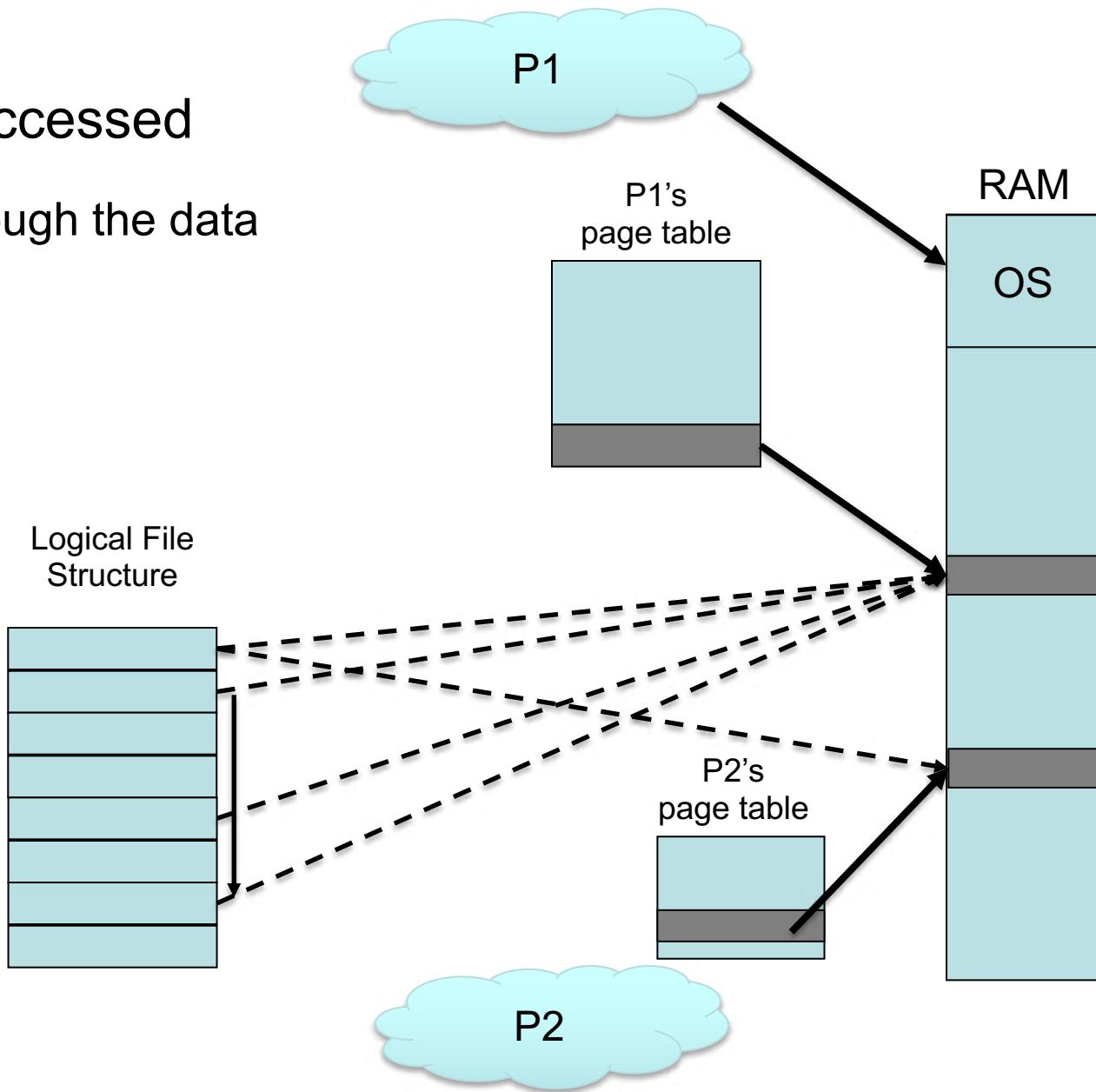
Memory Mapped Files



University of Colorado
Boulder

Memory-Mapped Files

- File data is accessed
 - Linearly through the data
 - Randomly



Memory-Mapped Files

- Map some parts of a file on disk to pages of virtual memory
 - normally, each read/write from/to a file requires a system call plus file manager involvement plus reading/writing from/to disk
 - programmer can improve performance by copying part of or entire file into a local buffer, manipulating it, then writing it back to disk
 - this requires manual action on the part of the programmer
 - instead, it would be faster and simpler if the file could be loaded into memory (almost) transparently so that reads/writes take place from/to RAM
 - Use the virtual memory mechanism to map (parts of) files on disk to pages in the logical address space

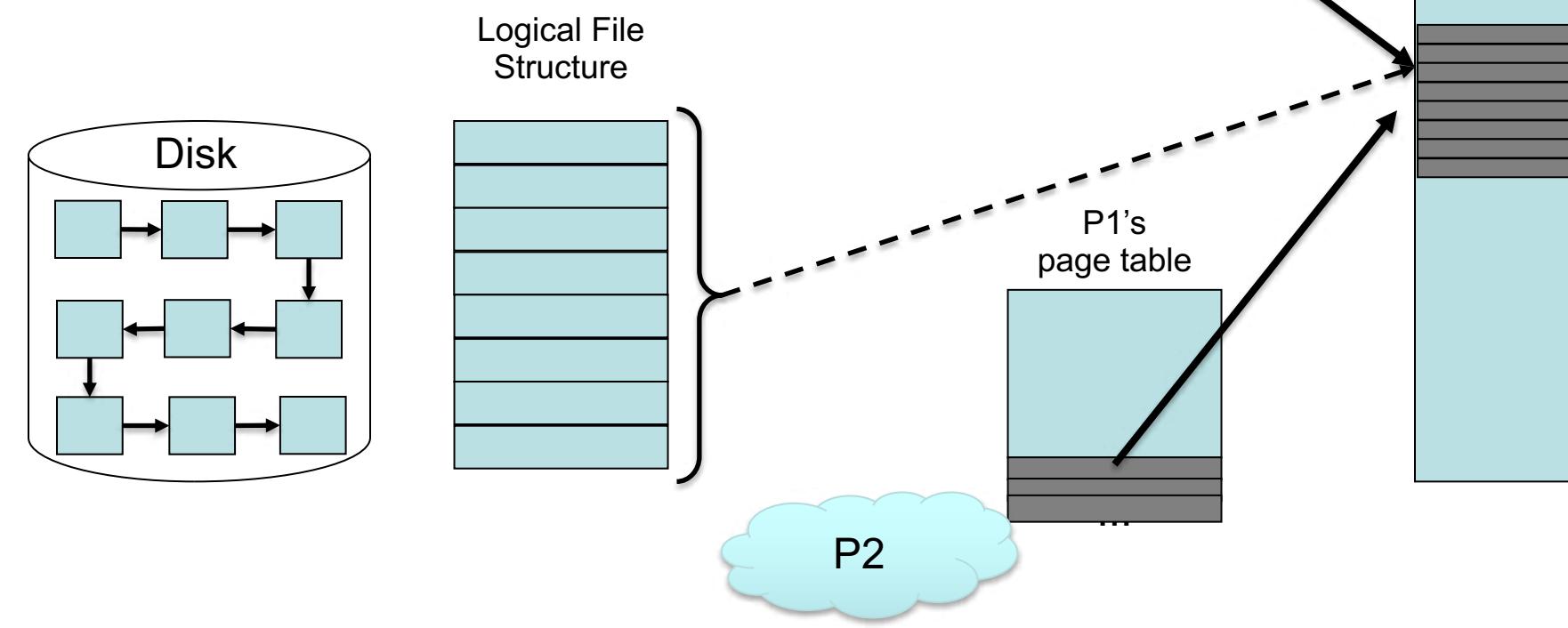
Steps for memory-mapping a file

Steps:

1. Obtain a handle to a file by creating or opening it
2. Reserve virtual address space for the file in your logical address space
3. Declare a (portion of a) file to be memory mapped by establishing a mapping between the file and your virtual address space
 - Use an OS function like *mmap()*
 - *void * mmap(void *start, size_t length, int prot, int flags, int fd, off_t offset)*
 - map length bytes beginning at offset into file fd, preferably at address start (hint only), prot = R/W/X/no access, flags = maps_fixed, map_shared, map_private
 - returns pointer to mmap'ed area

Memory-Mapped Files

- File data is accessed
 - Linearly through the data
 - Randomly



Memory-Mapped Files

Steps for memory-mapping a file: (cont.)

4. When file is first accessed, it's demand paged into physical memory
5. Subsequent read/write accesses to (that portion of) the file are served by physical memory

Memory-Mapped Files

- Advantages of memory-mapping files:
 1. after the first accesses, all subsequent reads/writes from/to a file (in memory) are fast
 - No longer use file system to read/write. Instead use MMU
 2. multiple processes can map the same file concurrently and
 - share efficiently, as shown in the previous figure
 - In Windows, this mapping mechanism is also used to create shared memory between processes and is the preferred memory for sharing information among address spaces
 - on Linux, have separate `mmap()` and shared memory calls, e.g. `shmget()` and `shmat()`

Memory-Mapped Files

3. Remember that the entire file **need not** be mapped into memory, just a portion or “view” of that file
 - in previous figure, F could represent just a portion of some larger file
 - F can also be subdivided into pages, each of which is mapped to a separate page in memory by the page table
4. File system has to be consulted less often
 - Step 4 to determine where on disk a view of a file is located
 - File Open is an expensive operation

Memory-Mapped Files

- 1) Problem 1 - Writes to a file in memory can result in momentary inconsistency between the file cached in memory and the file on disk
 - could write changes immediately (synchronously) to disk
 - Or delay and cache writes (asynchronous policy)
 - wait until OS periodically checks if dirty/modify bit has been set
 - wait until activity is less, then write altogether so that individual writes don't delay execution of process
 - this also allows the OS to group writes to the same part of disk to optimize performance. (we'll see this later)

MM I/O vs. MMFiles

- Similar behavior to memory-mapped files
- Recall from pre-midterm lecture slides that memory-mapped I/O maps device registers (instead of file pages) to memory locations
 - reads/writes from/to these memory addresses are easy and are automatically caught by the MMU (just as for memory-mapped files), causing the data to be automatically sent from/to the I/O devices
 - e.g. writing to a display's memory-mapped frame buffer, or reading from a memory-mapped serial port
 - Devices use an API to a virtual file

Summary

- Virtual memory
 - Making logical space independent of physical memory space
 - Advantages vs. disadvantages
- Page replacement policies
- Belady's Anomaly
- Frame allocation
- Thrashing
- Memory Mapped Files



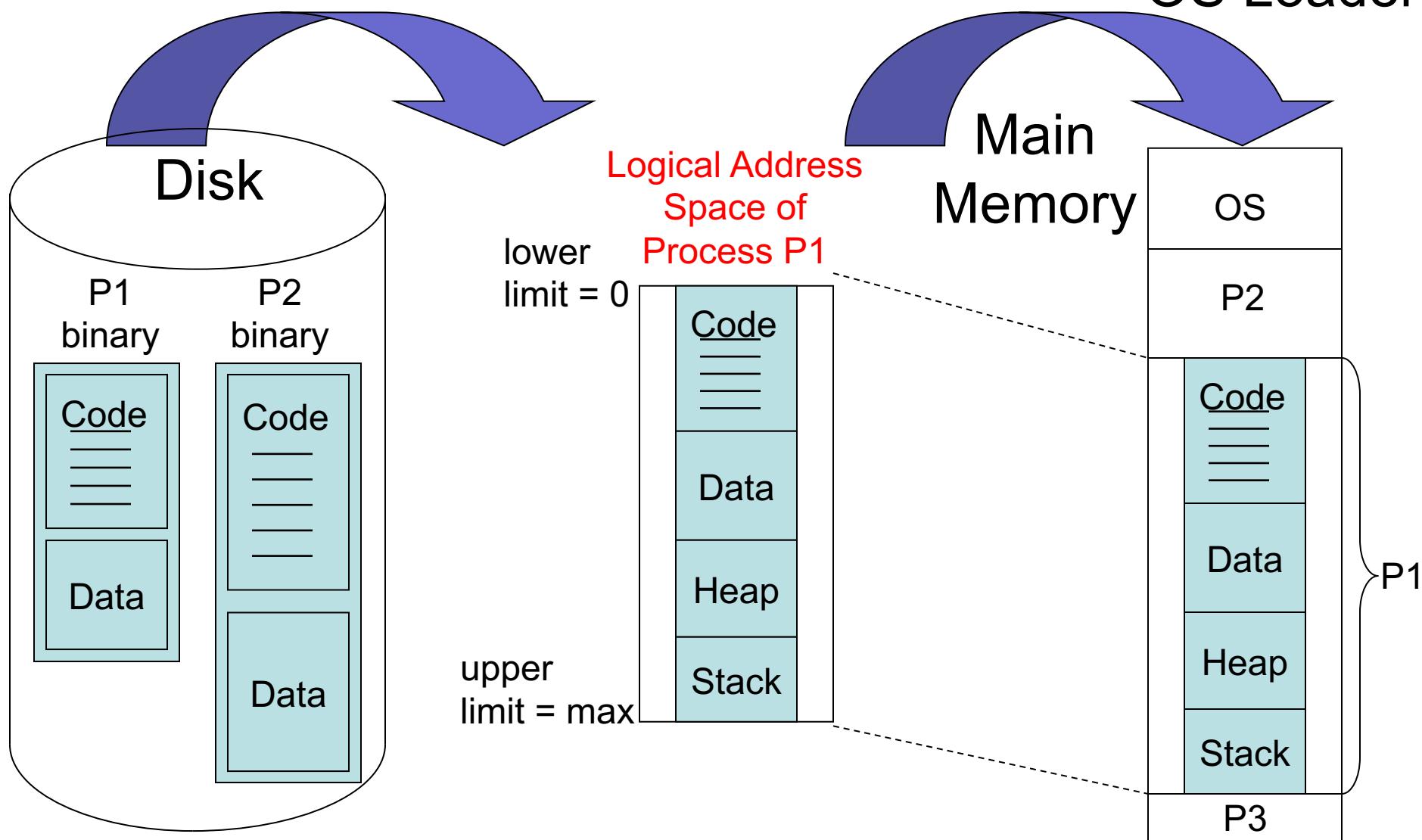
Lecture 18

Virtual Memory



University of Colorado
Boulder

Memory Management Virtual Memory OS Loader



Logical Address Space

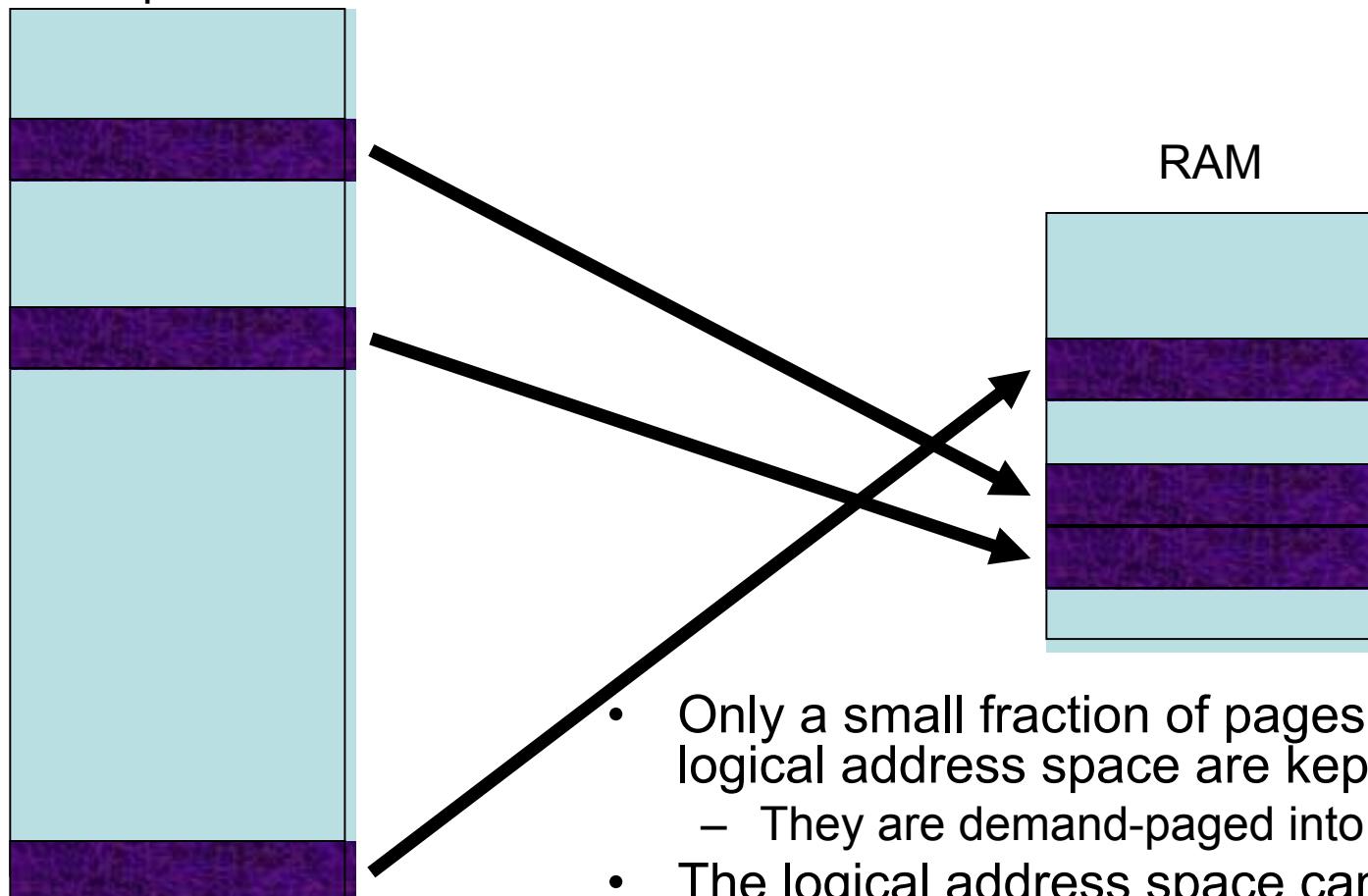
Code

Stck

Data

Virtual Memory

RAM



- Only a small fraction of pages from the logical address space are kept in RAM
 - They are demand-paged into RAM
- The logical address space can be much larger than physical RAM
 - Swap in the pages needed, when needed

On-Demand Paging

- Page tables may be large, consuming much RAM
- Key observation: not all pages in a logical address space need to be kept in memory
 - in the simplest case, just keep the current page where the PC is executing
 - all other pages could be on disk
 - when another page is needed, retrieve the page from disk and place in memory before executing
 - this would be costly and slow, because it would happen every time that a page different from the current one is needed

On-Demand Paging

- Instead of just keeping one page, keep a *subset* of a process's pages in memory
 - Load just what you need, not the entire address space
 - use memory as a cache of frequently or most recently used pages
 - rely on a program's behavior to exhibit locality of reference
 - if an instruction or data reference in a page was previously invoked, then that page is likely to be referenced again in the near future

On-Demand Paging

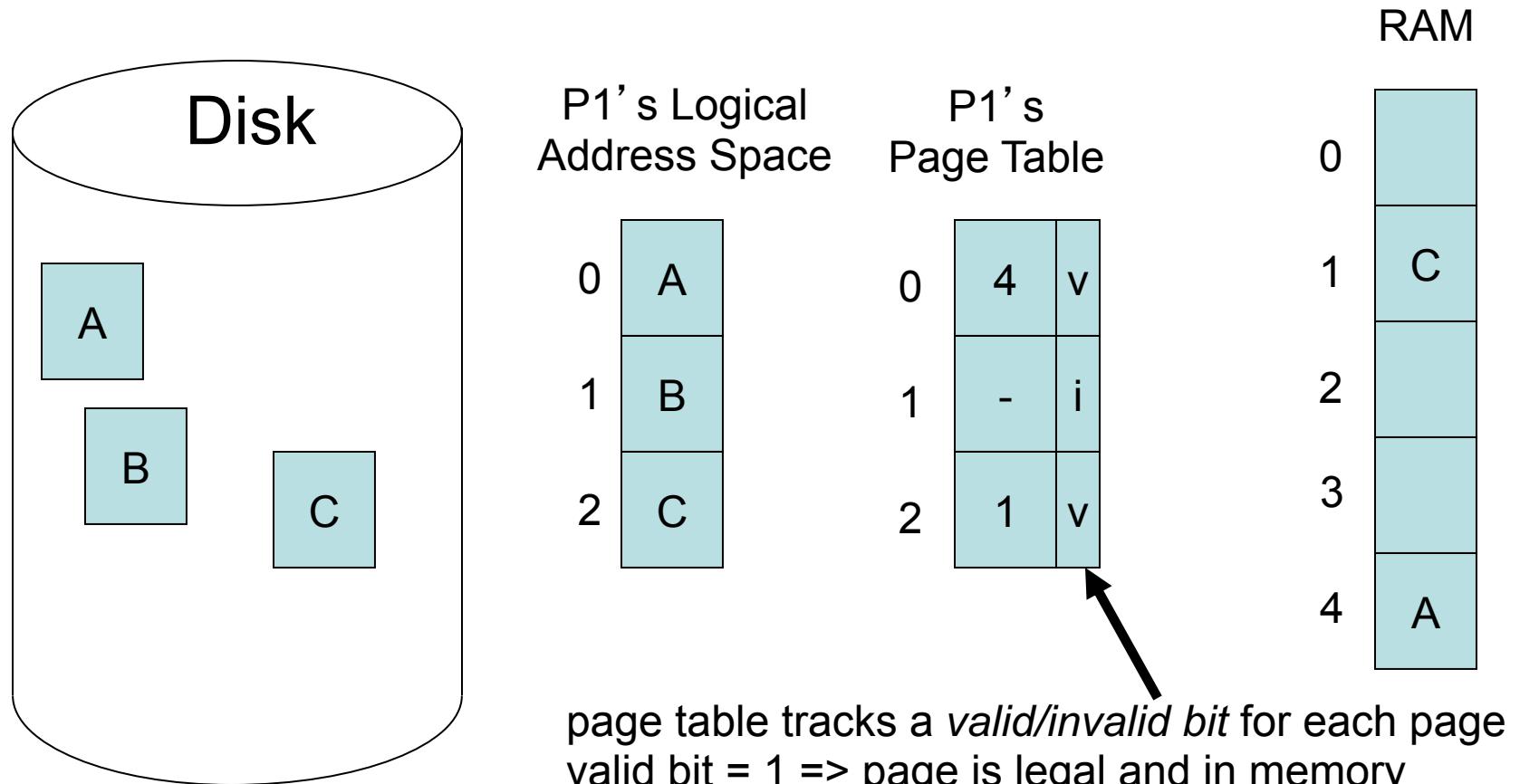
- Most programs exhibit some form of **locality**
 - looping locally through the same set of instructions
 - branching through the same code
 - executing linearly, the next instruction is typically the next one immediately after the previous instruction, rather than some random jump
- Thus process execution revisits pages already stored in memory
 - so you don't have to go to disk each time the program counter (PC) jumps to a different page

On-Demand Paging

- On-demand paging is used to page in new pages from disk to RAM
 - only when a page is needed is it loaded into memory from disk
- Can page in an **entire process on demand**:
 - starting with “zero” pages
 - the reference to the first instruction causes the first page of the process to be loaded on demand into RAM.
 - Subsequent pages are loaded on demand into RAM as the process executes.

On-Demand Paging

- On-demand paging loads a page from disk into RAM only when needed
 - in the example below, pages A and C are in memory, but page B is not

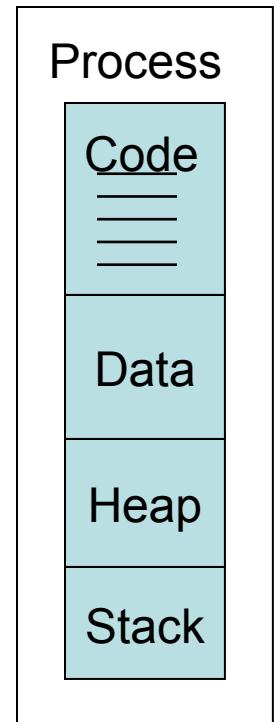


Virtual Memory Advantages

1. Virtual address space can now exceed physical RAM!
 - Only a subset (most demanded) of pages are kept in RAM
 - We have decoupled virtual memory from physical memory.
2. Fit many more processes in memory

Virtual Memory Advantages

3. Decreases swap time
 - there is less to swap
4. Can have large sparse address spaces
 - most of the address space is unused
 - Does not take up physical RAM
 - a large heap and stack that is mostly unused/empty won't take up any actual RAM until needed

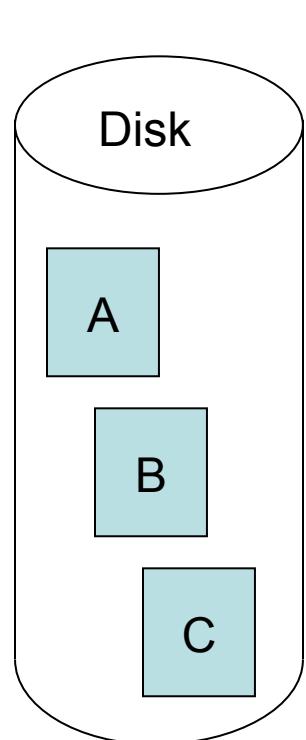


On-Demand Paging

- Open questions to be answered:
 - how is a needed page loaded into memory from disk?
 - how many pages in memory should be allocated to a process?
 - if the # of pages allocated to a process is exceeded, i.e. the cache is full, then how do you choose which page to replace?

Virtual Memory

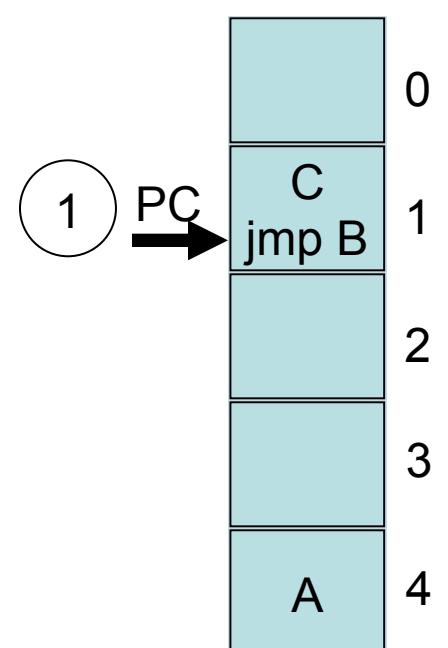
On-demand paging causes a page fault, which goes through the following steps



P1's
Page Table

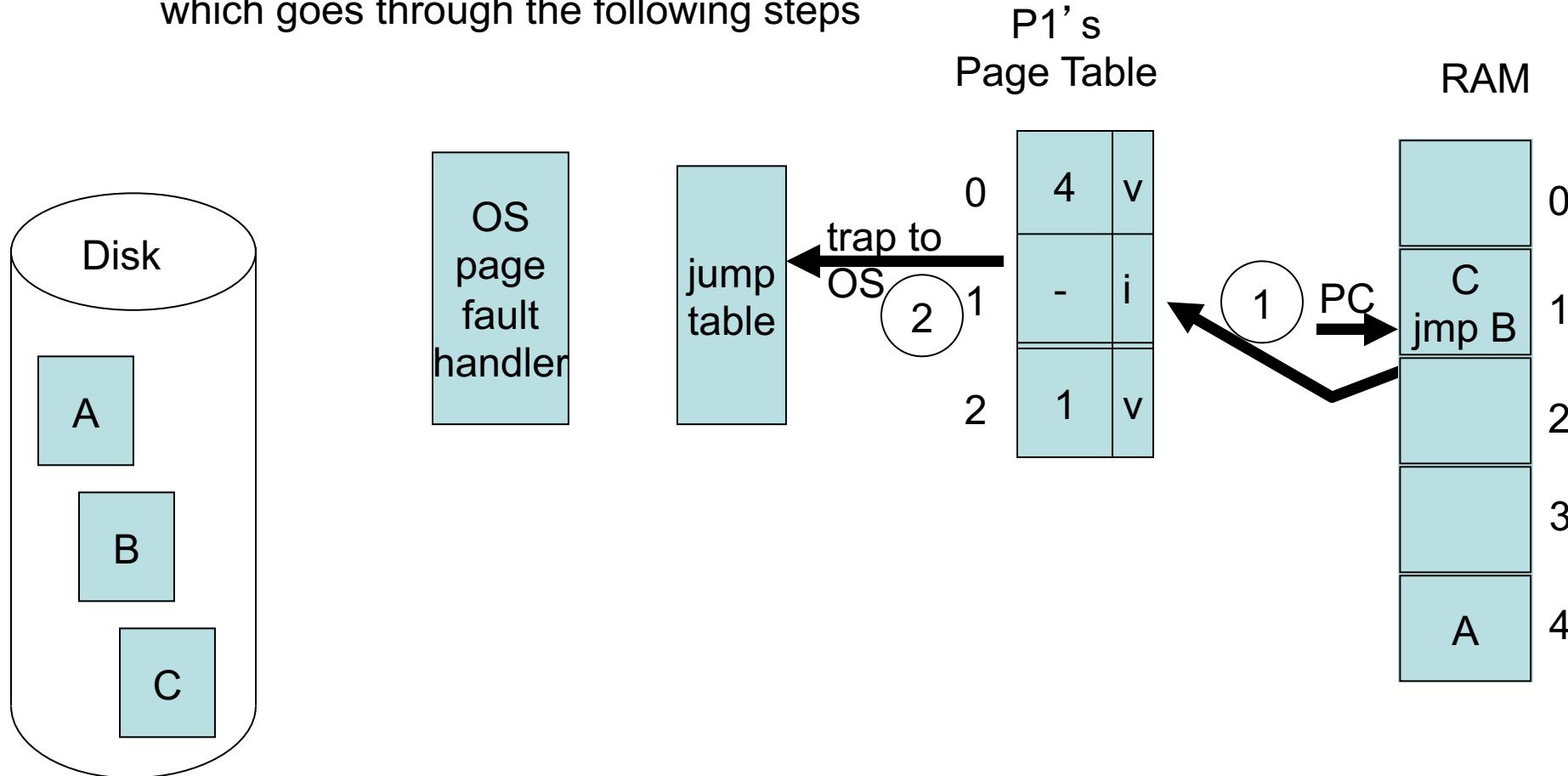
0	4	v
1	-	i
2	1	v

RAM



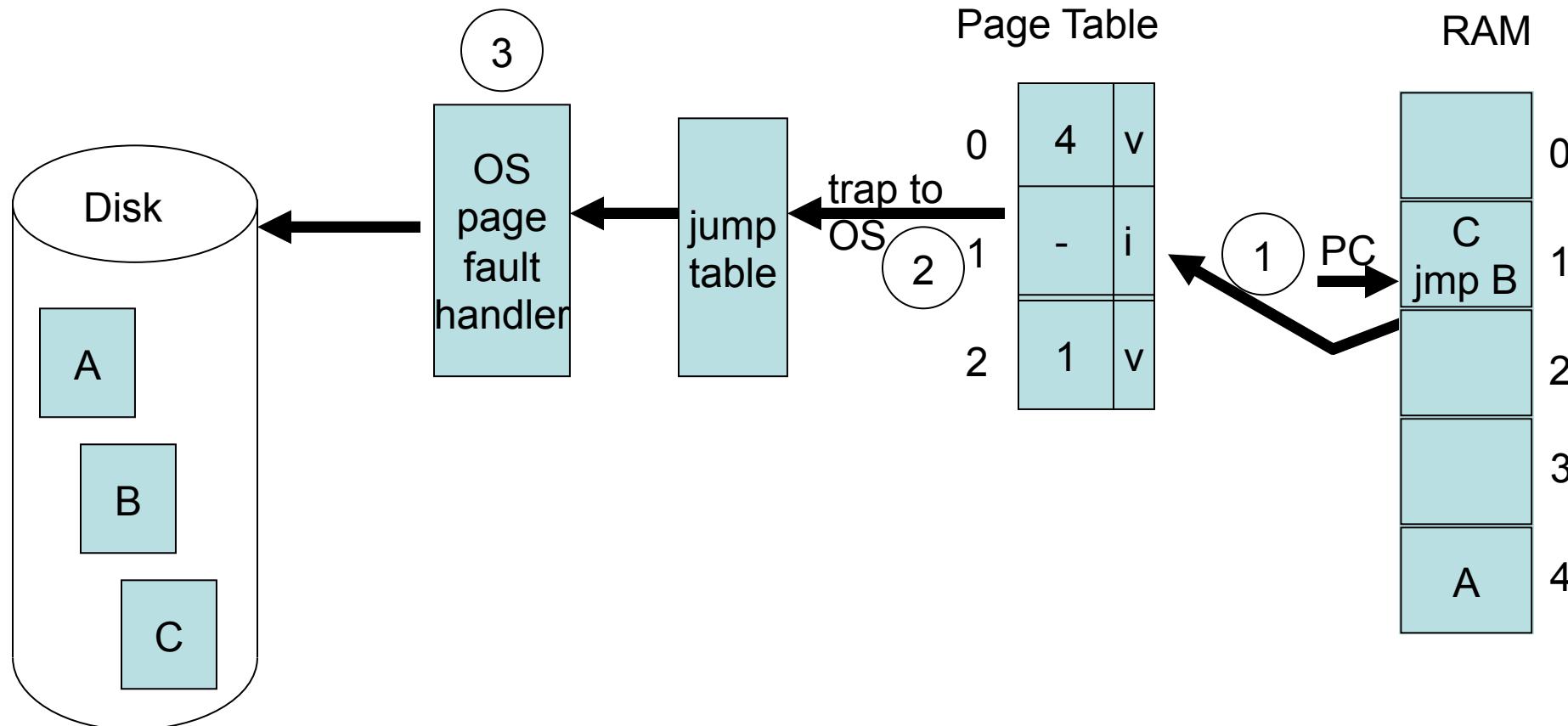
Virtual Memory

On-demand paging causes a page fault, which goes through the following steps



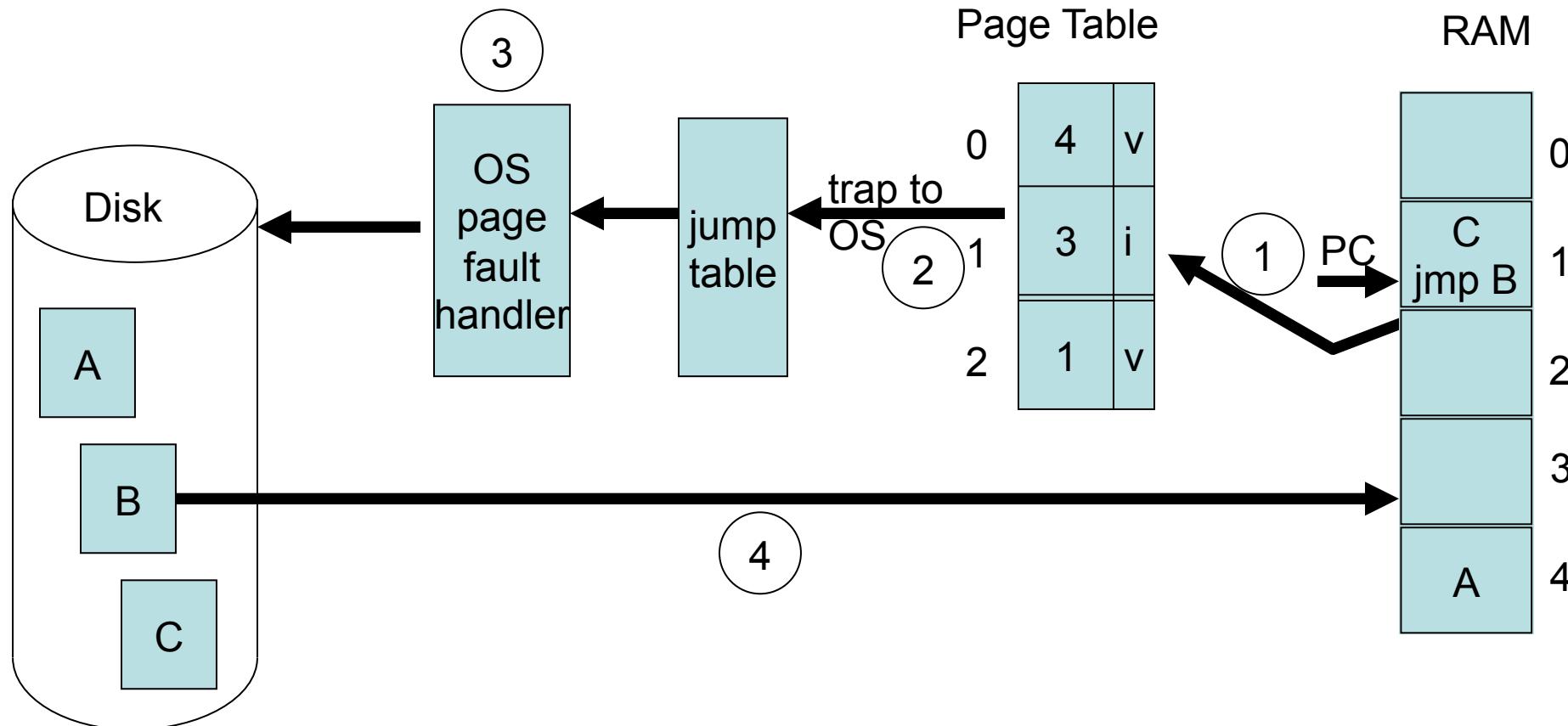
Virtual Memory

On-demand paging causes a page fault, which goes through the following steps



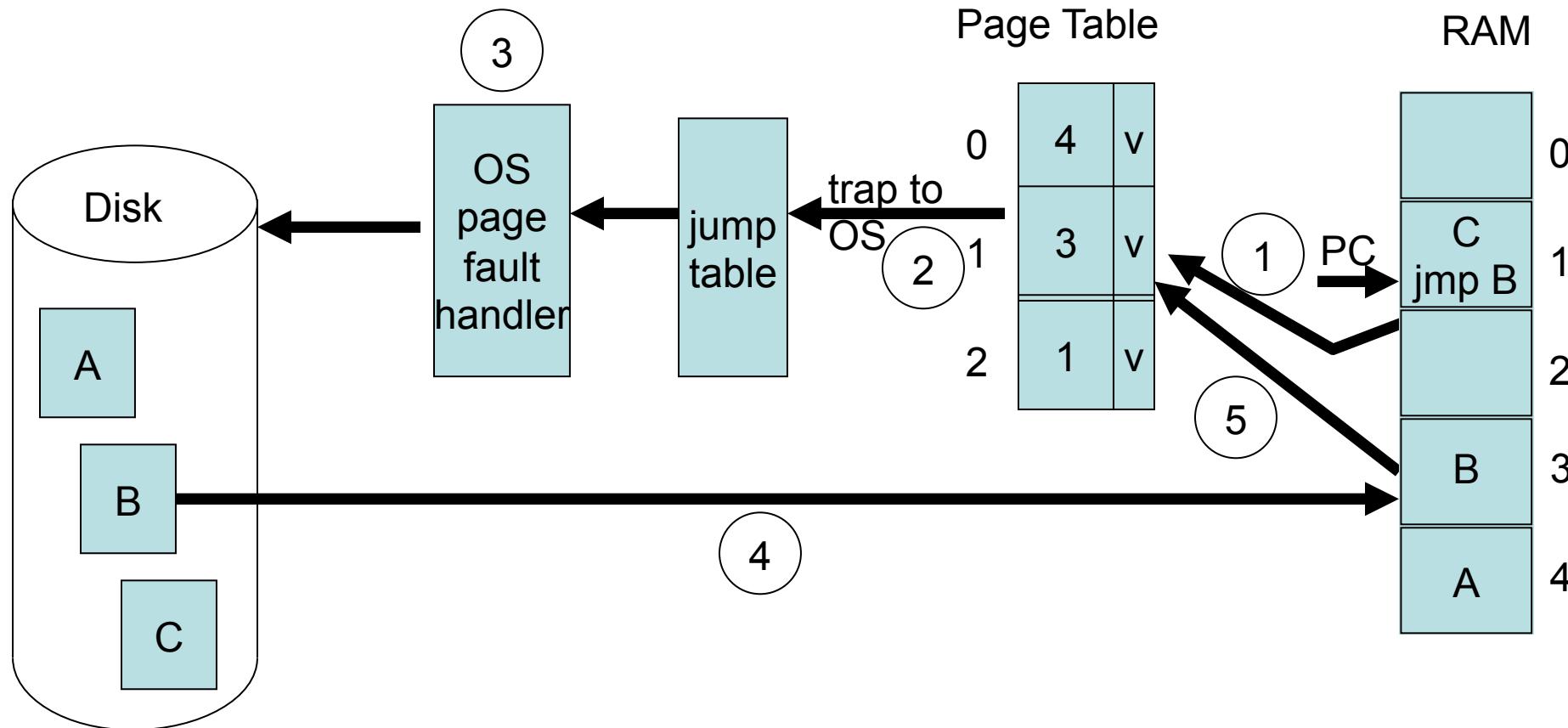
Virtual Memory

On-demand paging causes a page fault, which goes through the following steps



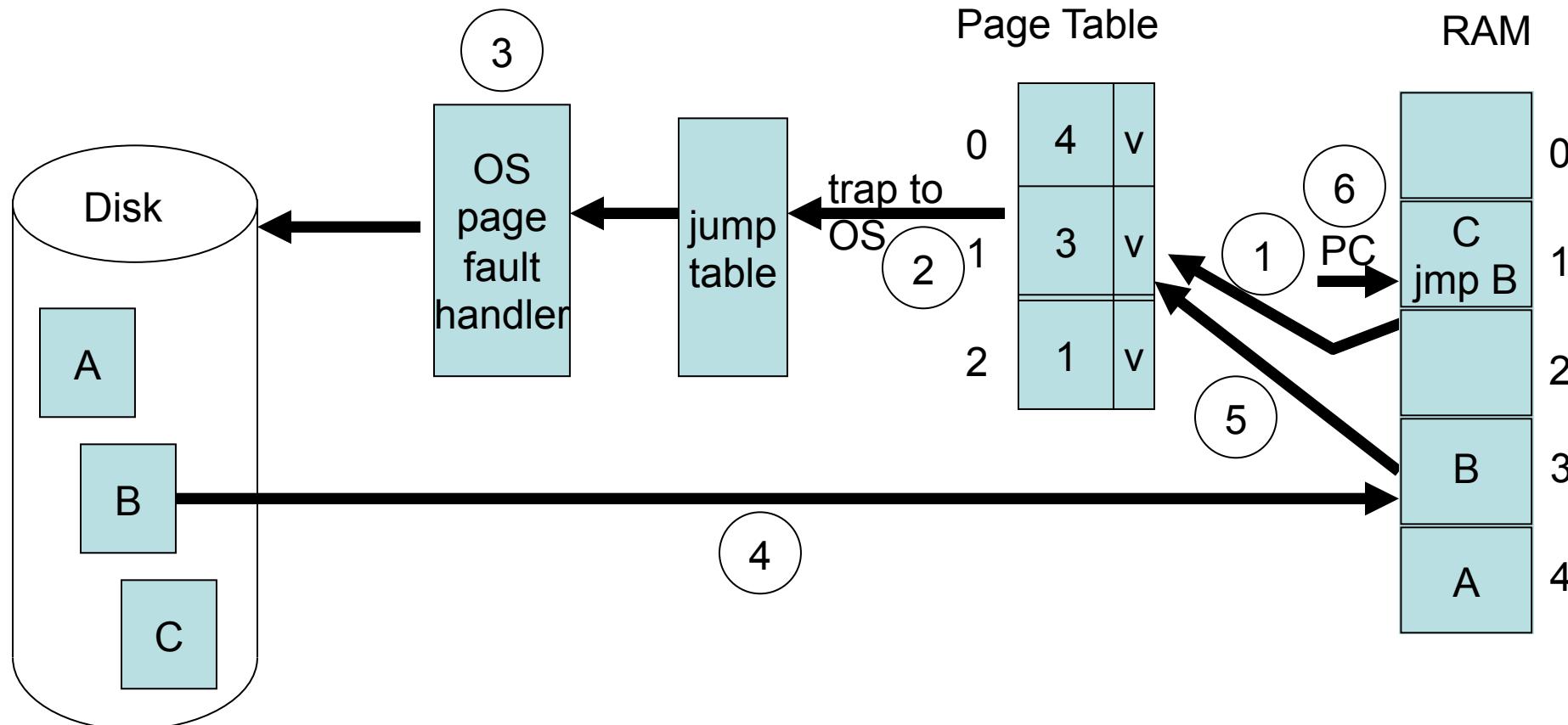
Virtual Memory

On-demand paging causes a page fault, which goes through the following steps



Virtual Memory

On-demand paging causes a page fault, which goes through the following steps



Virtual Memory

- Page-fault steps to load a page into RAM:
 1. MMU detects a page is not in memory (invalid bit set) which causes a *page-fault trap* to OS
 2. OS saves registers and process state. Determines that the fault was due to demand paging and jumps to page fault handler
 3. *Page fault handler*
 - a) If reference to page not in logical A.S. then seg fault.
 - b) Else if reference to page in logical A.S., but not in RAM, then load page
 - c) OS finds a free frame
 - d) OS schedules a disk read. Other processes may run in meantime.
 4. Disk returns with interrupt when done reading desired page. OS writes desired page into free frame
 5. OS updates page table, sets valid bit of page and its physical location
 6. Restart interrupted instruction that caused the page fault

Virtual Memory

- OS can retrieve the desired page either from the disk's
 - file system, or
 - swap space/backing store
 - faster, avoids overhead of file system lookup
- pages can be in swap space because:
 - the entire executable file was copied into swap space when the program was first started.
 - Avoids file system, but also allows the copied executable to be laid out contiguously on disk's swap space, for faster access to pages (no seek time)

Virtual Memory

- pages can be in swap space because:
 - as pages have to be replaced in RAM, they are written to swap space instead of the file system's portion of disk.
 - The next time they're needed, they're retrieved quickly from swap space, avoiding a file system lookup.

Performance of On-Demand Paging

- Want to limit the number/frequency of page faults, which cause a read from disk, which slows performance
 - disk read is about 10 ms
 - memory read is about 10 ns
- What is the average memory access time?
 - average access time = $p \cdot 10 \text{ ms} + (1-p) \cdot 10 \text{ ns}$, where p = probability of a page fault.
 - if $p=.001$, then average access time = $10 \mu\text{s} \gg 10 \text{ ns}$ (1000 X greater!)
 - to keep average access time within 10% of 10 ns, would need a page fault rate lower than $p < 10^{-7}$
 - Reducing page fault frequency improves performance in a big way



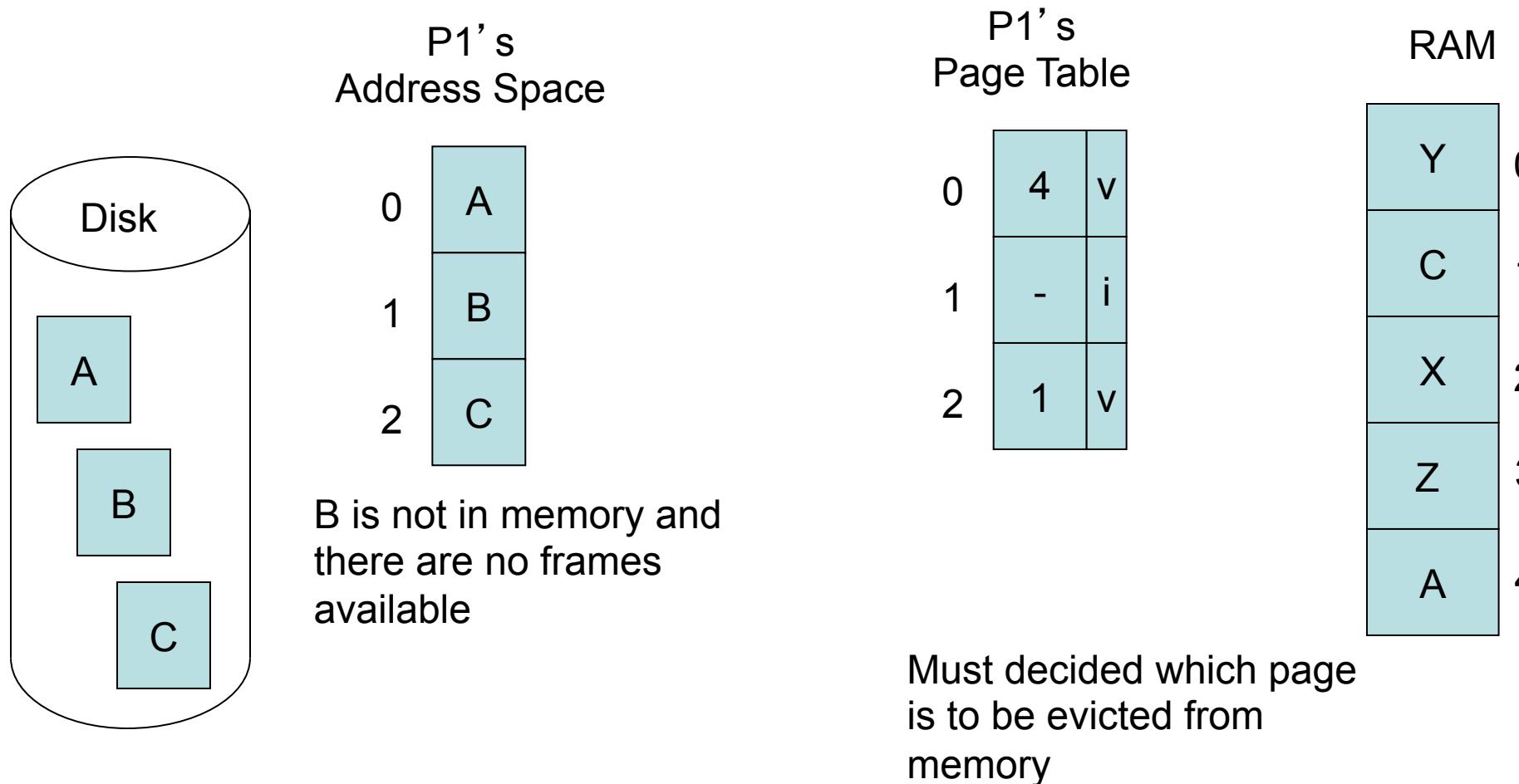
Page Replacement Policies



University of Colorado
Boulder

Page Replacement Policies

- Demand paging loads a page from disk into RAM only when needed
 - in the example below, pages A and C in memory, but page B is not
 - How can we access B if all other frames are already used

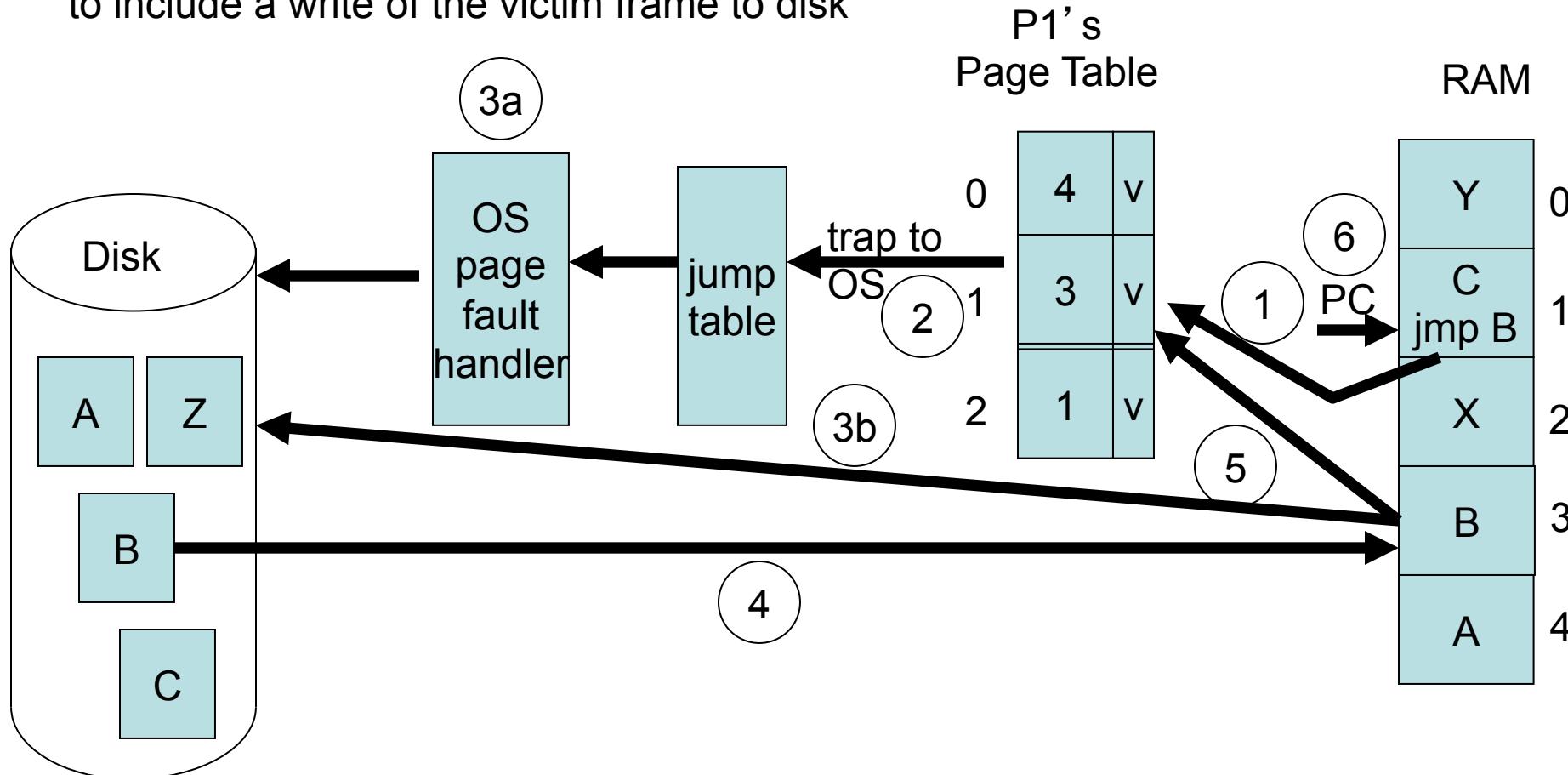


Page Replacement Policies

- As processes execute and bring in more pages on demand into memory, eventually the system runs out of free frames
 - need a *page replacement policy*
 - *Steps:*
 1. select a victim frame that is not currently being used
 2. save or write the victim frame to disk, update the page table (page now invalid)
 3. load in the new desired page from disk
 - If out of free frames, each page fault causes 2 disk operations, one to write the victim, and one to read the desired page
 - this is a big performance penalty

Page Replacement Policies

In Step 3b, we modify traditional on-demand paging to include a write of the victim frame to disk



Page Replacement Policies

- To reduce the performance penalty of 2 disk operations, systems can employ a *dirty/modify bit*
 - modify bit = 0 initially
 - when a page in memory is written to, set the bit = 1
 - when a victim page is needed, select a page that has not been modified (dirty bit = 0)
 - such an unmodified page need not be written to disk, because its mirror image is already on disk!
 - this saves on disk I/O - reduces to only 1 disk operation (read of desired page)

Page Table Status Bits

- Each entry in the page table can conceptually store several extra bits of metadata information along with the physical frame # f
 - *Valid/invalid bits* - for memory protection, accessing an invalid page causes a page fault

		Page Table			
		phys		fr #	
0	1	2	1	0	1
		8	0	1	0
		4	0	0	0
		7	1	1	0

Diagram illustrating the layout of status bits in a page table entry:

- The page table has 4 entries (rows) indexed 0 to 3.
- Each entry contains 4 fields (columns):
 - Physical Frame Number (phys fr #)
 - R/W or Read only
 - Valid/ Invalid
 - Dirty/ Modified

Page Table Status Bits

- *dirty bits* - has the page been modified for page replacement?
- *R/W or Read-only bits* - for memory protection, writing to a read-only page causes a fault and a trap to the OS
- *Reference bit* – useful for Clock page replacement algorithm

		Page Table			
		phys	fr #		
0	1	2		1	0
		8	0	1	0
2	3	4	0	0	0
3	4	7	1	1	0

R/W or Read only *Dirty/ Modified*
Valid/ Invalid

Recap ...

- Virtual memory
 - Keep only a few pages in memory, rest on disk
 - On-demand paging: retrieve a page when needed
 - Page fault
 - A referenced page is not loaded in memory
 - OS blocks the process and retrieves the referenced page
 - Significant performance overhead – need to keep page fault frequency low, e.g. less than 1 in 10^7 for overhead <10%
 - Page replacement algorithm
 - Principle of locality of reference
 - Dirty bit: choose a clean page before a dirty page



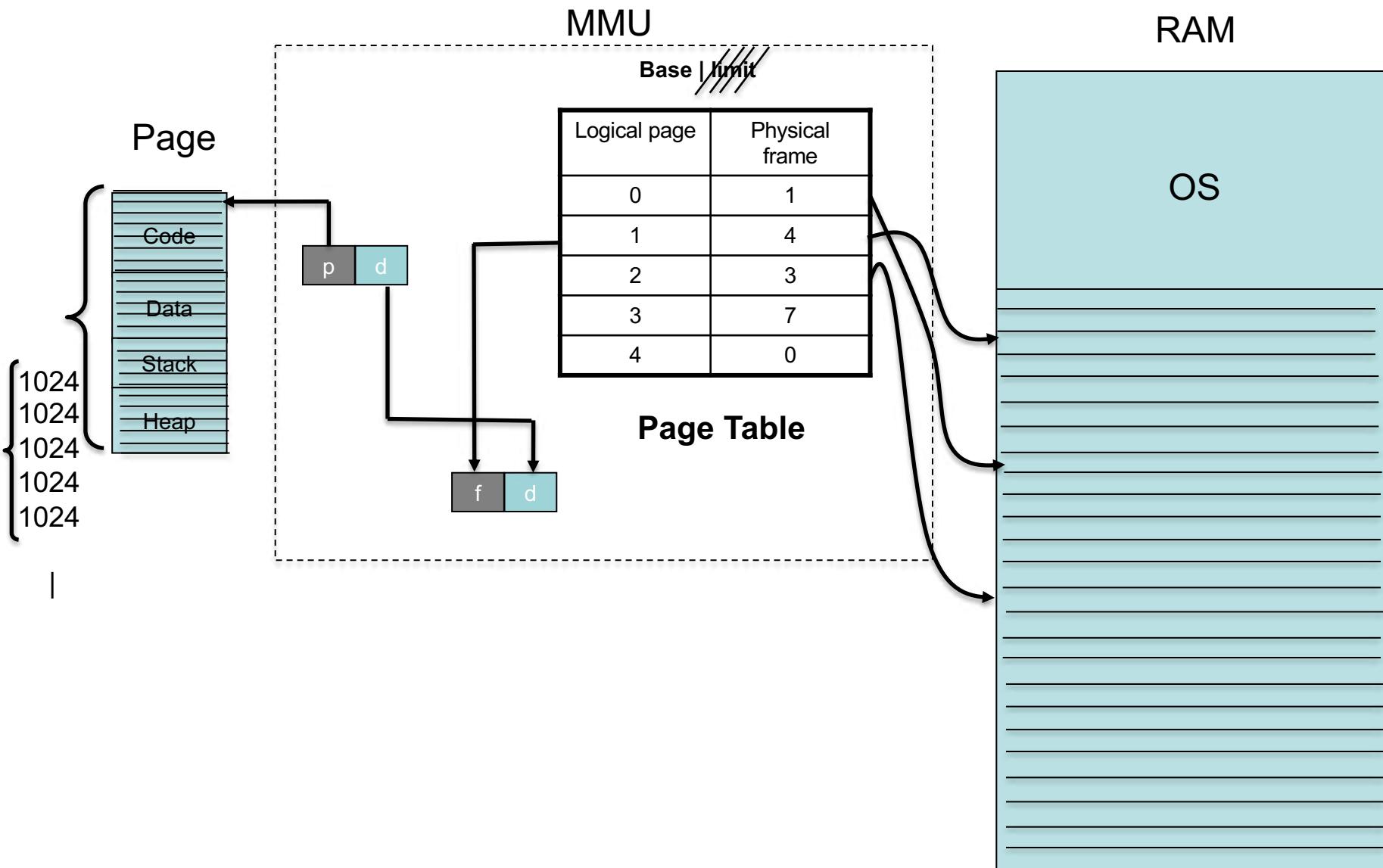
Lecture 17

Paging



University of Colorado
Boulder

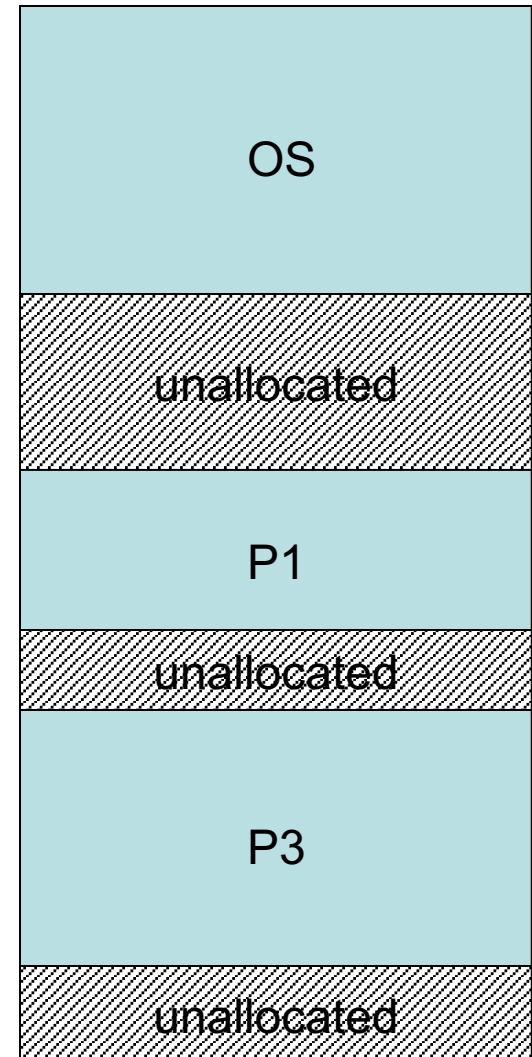
Better Solution to Reduce Fragmentation



Avoiding External Fragmentation

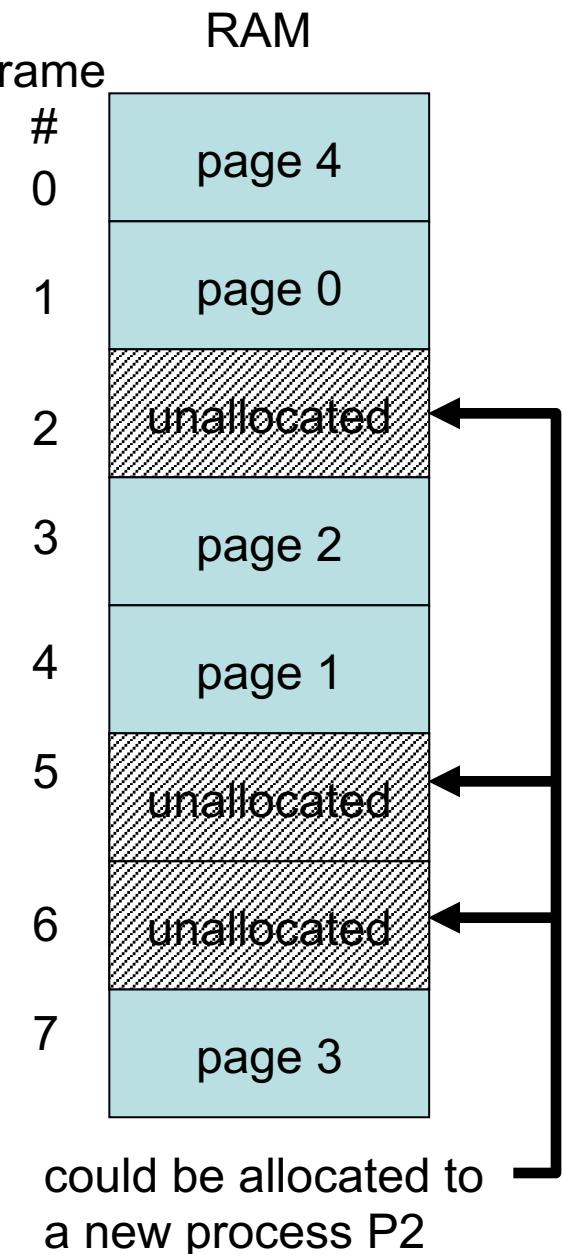
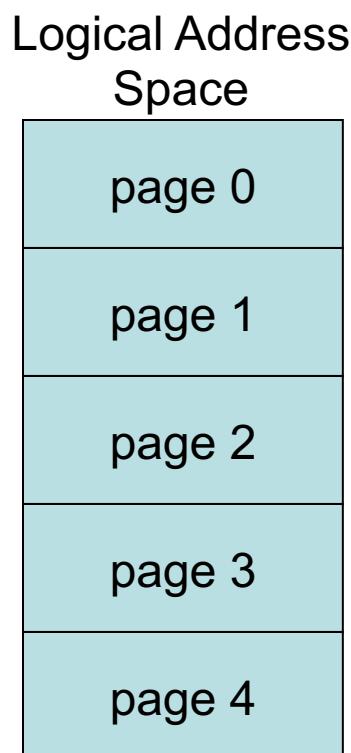
RAM

- Over time, repeated allocation and deallocation will cause many small chunks of non-contiguous unallocated memory to form between allocated processes in memory
- Resulting in external fragmentation
- OS must swap out current processes to create a large enough contiguous unallocated memory
- Could use de-fragmentation, but it is costly to move memory.



Paging

- A better solution to external fragmentation is to divide the logical address space into fixed-size pages
- We can also break main memory into fixed-sized frames
- Each page can be located in any frame



Paging

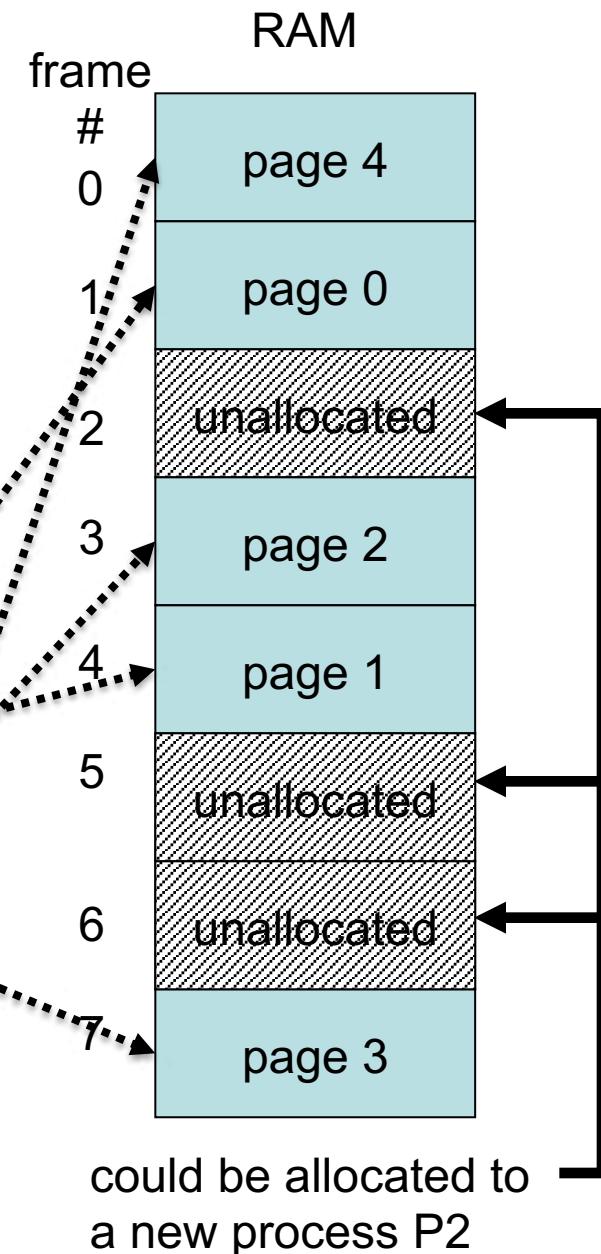
- OS maintains a page table for each process
- Given a logical address, MMU finds its logical page, then looks up physical frame in page table.

Logical Address Space

page 0
page 1
page 2
page 3
page 4

Page Table

Logical page	Physical frame
0	1
1	4
2	3
3	7
4	0



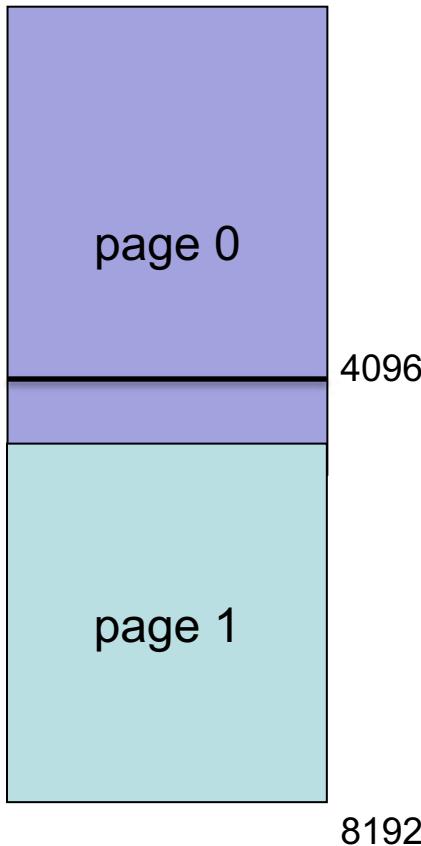
Paging

Logical Address Space

page 0
page 1
page 2
page 3
page 4

- User's view of memory is still as one contiguous block of logical address space
 - MMU performs run-time mapping of each logical address to a physical address using the page table
- Typical page size is 4-8 KB
 - Example: a 4 GB 32-bit address space with 4 KB/page (2^{12})
 $=> 2^{32}/2^{12} = 1 \text{ million entries in page table}$
 - Your page table would need to be ≥ 20 bits/ table entry (~1 MB per process)

Address Space



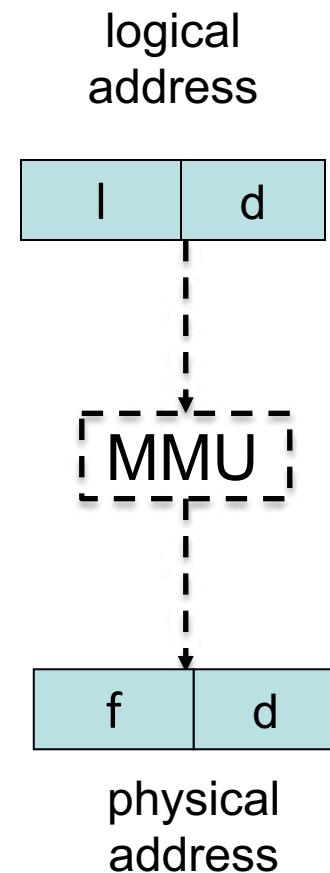
Paging

- No external fragmentation
- But we do get some ***internal fragmentation***
 - example: suppose my process is size 4001 B, and each page size is 4 KB (4096 B)
 - then I have to allocate two pages = 8 KB,
 - 3999 B of 2nd page is wasted due to fragmentation that is internal to a page
- OS also has to maintain a frame table/pool that keeps track of what physical memory frames are free

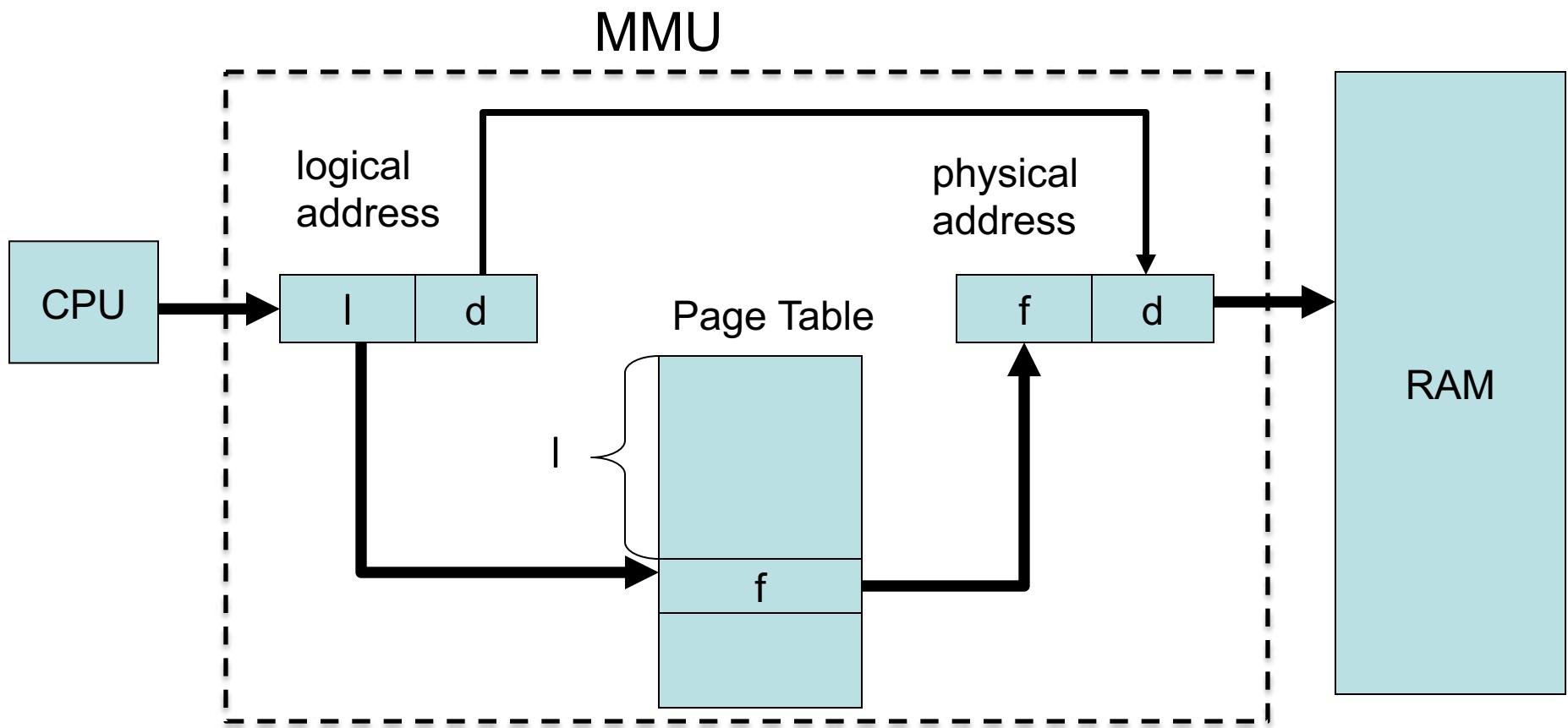
Paging

- Conceptually, every logical/virtual address can now be divided into two parts:

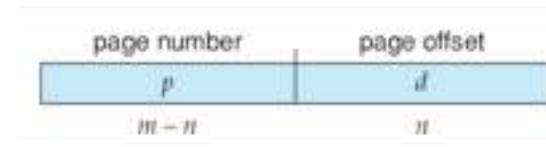
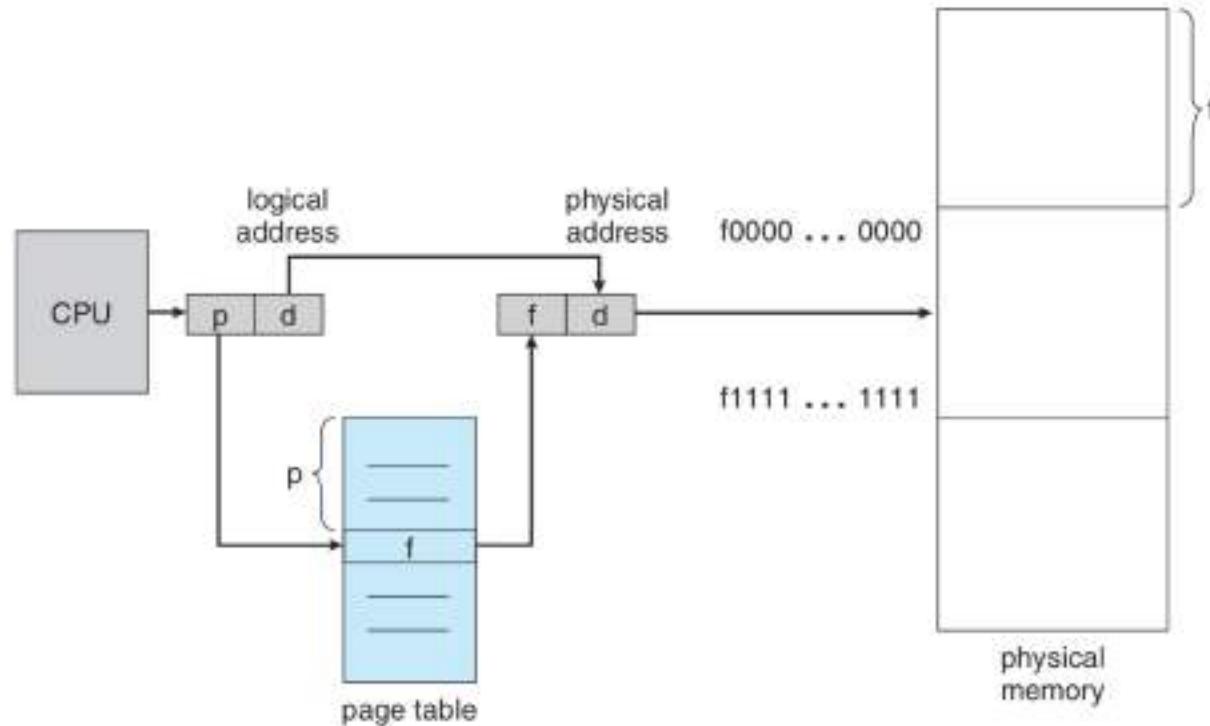
- most significant bits = logical page # l ,
 - Equals the virtual address / page size
 - used to index into page table to retrieve the corresponding physical frame f
 - least significant bits = page offset d



Paging



Paging

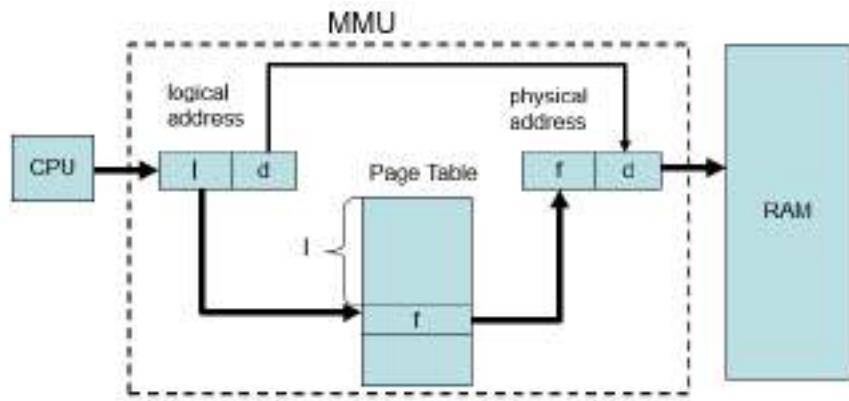


Size of the logical address space is 2^m

Size of each page is 2^n

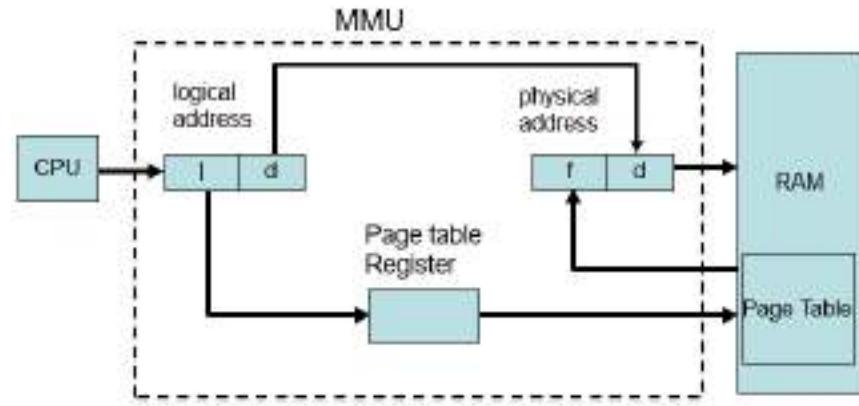
Paging

- Implementing a page table:



- option #1: use dedicated bank of hardware registers or memory to store the page table
 - fast per-instruction translation
 - slow per context switch - entire page table has to be reloaded for the new process
 - limited by cost (expensive hardware) to being too small - some page tables can be large, e.g. 1 million entries – too expensive

Implementing a page table



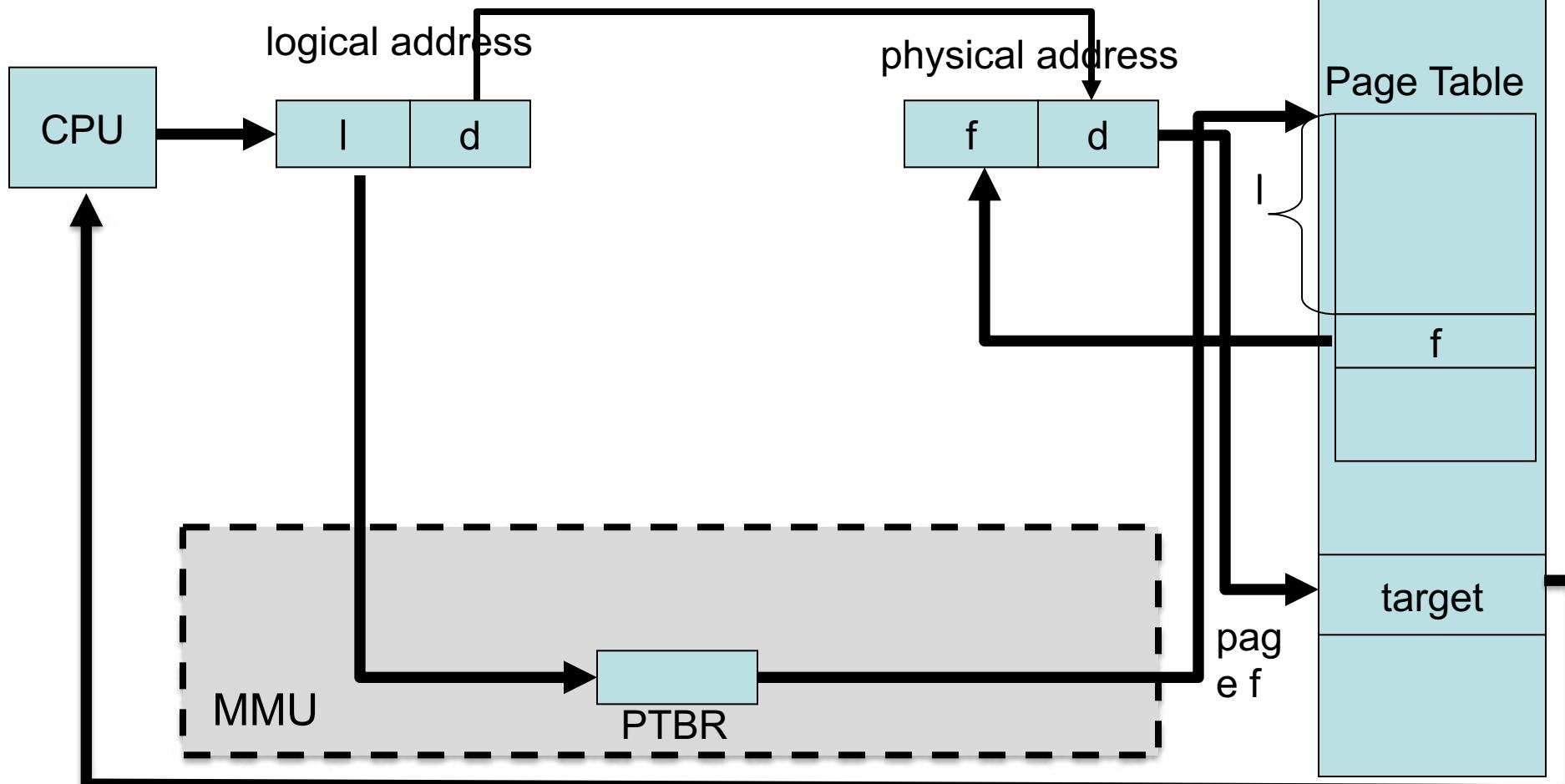
- option #2: store the page table in main memory
 - keep a pointer to the page table in a special CPU register called the Page Table Base Register (PTBR)
 - can accommodate fairly large page tables
 - fast context switch - only reload the PTBR!
 - slow per-instruction translation, because each instruction fetch requires *two steps memory access*:
 1. finding the page table in memory and indexing to the appropriate spot to retrieve the physical frame # f
 2. retrieving the instruction from physical memory frame f

• Paging with PTBR

RAM

- Slow per-instruction translation, because each instruction fetch requires two steps:

1. Finding the page table in memory and indexing to the appropriate spot to retrieve the physical frame # f
2. Retrieving the instruction from physical memory frame f

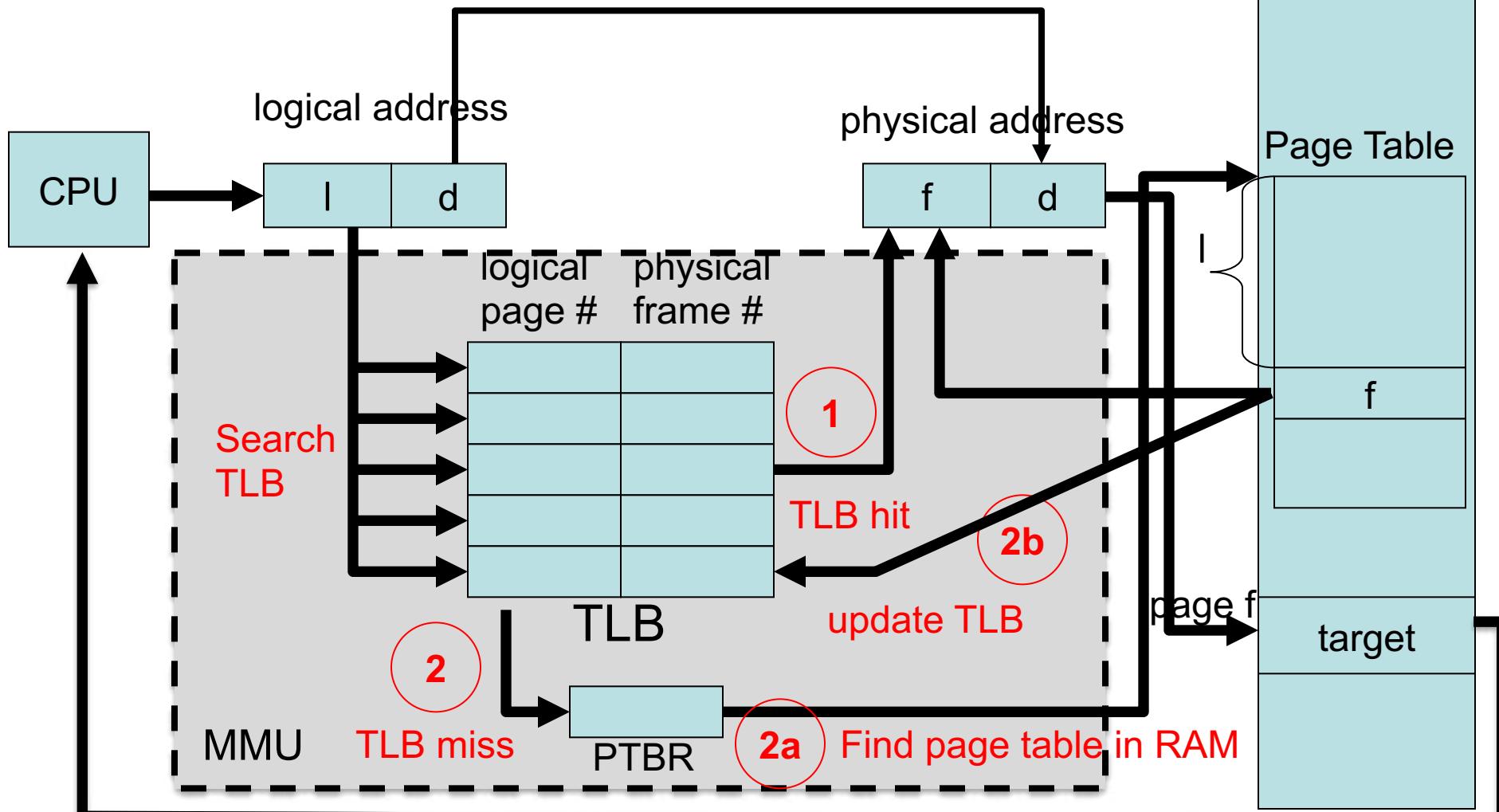


Implementing a page table

- Option #3: cache a subset of page table mappings/entries in a small set of CPU buffers called Translation-Look-aside Buffers (TLBs)
 - Fast solution to option #2's slow two-step memory access
 - Several TLB caching policies:
 - Cache the most popular or Most frequently referenced pages in TLB
 - Cache the most recently used pages

Paging

- Paging with TLB and PTBR



Return fetched code/data @ offset d inside page f

Paging and TLB Caching

- Summarize steps depicted in the graph on the last slide
- MMU in CPU first looks in TLB's to find a match for a given logical address
 1. if match found, then quickly call main memory with physical address frame f (plus offset d)
 - this is called a *TLB hit*
 - TLB as implemented in hardware does a fast parallel match of the input page to all stored values in the cache - about 10% overhead in speed

Paging and TLB Caching

(continued from previous slide)

2. if no match found, then this is a *TLB miss*
 - a) go through regular two-step lookup procedure: go to main memory to find page table and index into it to retrieve frame #f, then retrieve what's stored at address $\langle f, d \rangle$ in physical memory
 - b) Update TLB cache with the new entry from the page table
 - if cache full, then implement a cache replacement strategy, e.g. Least Recently Used (LRU) - we'll see this later
- Goal is to maximize TLB hits and minimize TLB misses

Paging and TLB Caching

- On a context switch, the TLB entries would typically have to all be invalidated/completely flushed
 - since different processes have different page tables
 - E.g. x86 behavior behaves like this
- An alternative is to include process IDs in TLB
 - at the additional cost of hardware and an additional comparison per lookup
 - Only TLB entries with a process ID matching the current task are considered valid
 - E.g. DEC RISC Alpha CPU

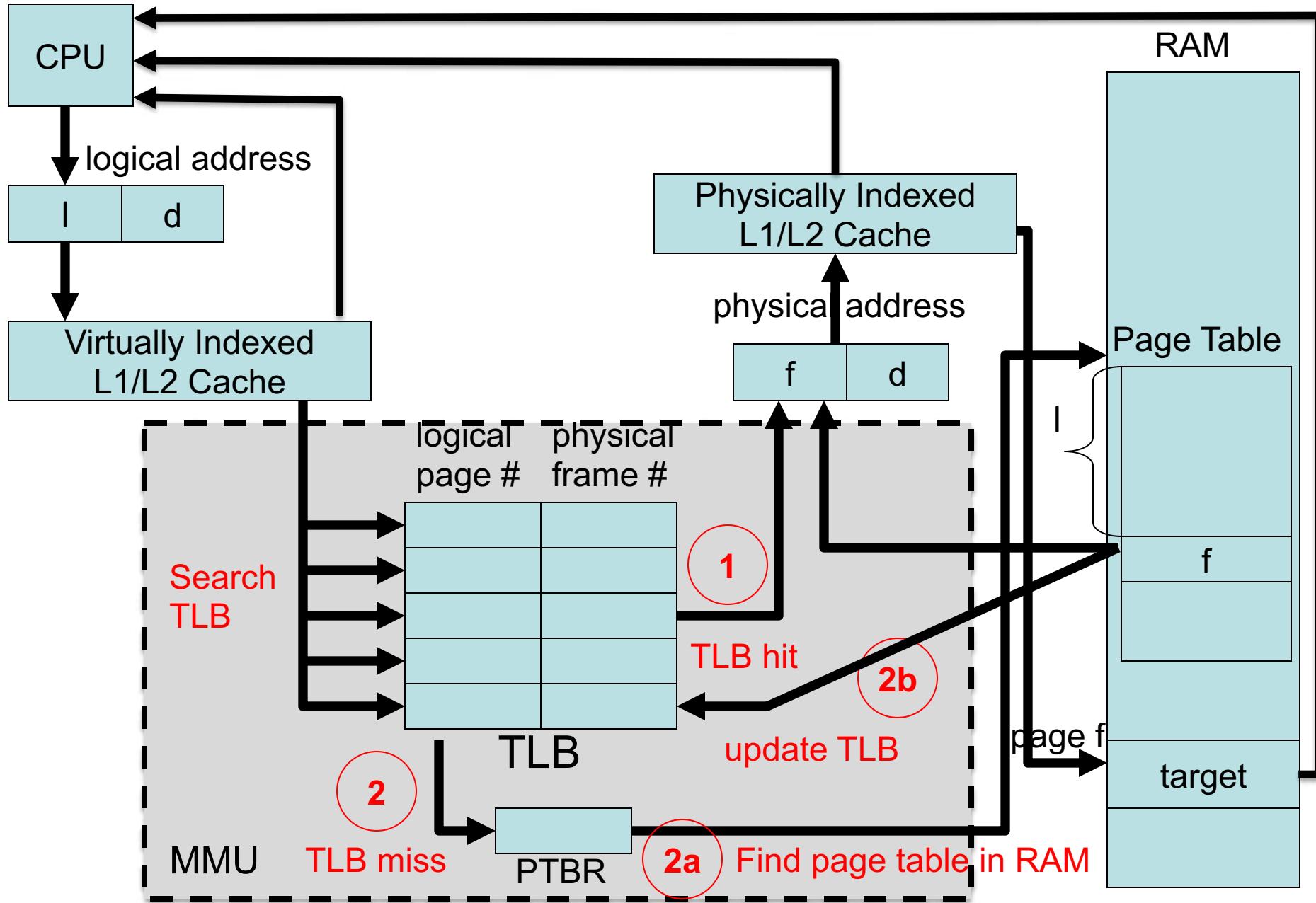
Paging and TLB Caching

- Another option: prevent frequently used pages from being automatically invalidated in the TLBs on a task switch
 - In Intel Pentium Pro, use the page global enable (PGE) flag in the register CR4 and the global (G) flag of a page-directory or page-table entry
- ARM allows flushing of individual entries from the TLB indexed by virtual address

Paging and L1/L2 Caching

- How does MMU interact with L1 or L2 data or instruction caches?
 - It depends on whether the items in a cache are indexed as virtual or physical (for look up purposes)
- L1/L2 data/instruction caches can store their information and be indexed by either virtual or physical addresses
 - If physical, then MMU must first convert virtual to physical, before the cache can be consulted – this is slow, but each entry is uniquely identifiable by its physical address
 - If virtual, then cache can be consulted quickly to see if there's a hit without invoking the MMU (if miss, then MMU must still be invoked...)

• Paging with TLB and L1/L2 Caching



Paging and L1/L2 Caching

- A virtually indexed L1/L2 data/instruction caches introduces some problems:
 - Homonym problem: when a new process is switched in, it may use the same virtual address V as the previous process.
 - The cache that indexes just by virtual address V will return the wrong information (cached information from the prior process).

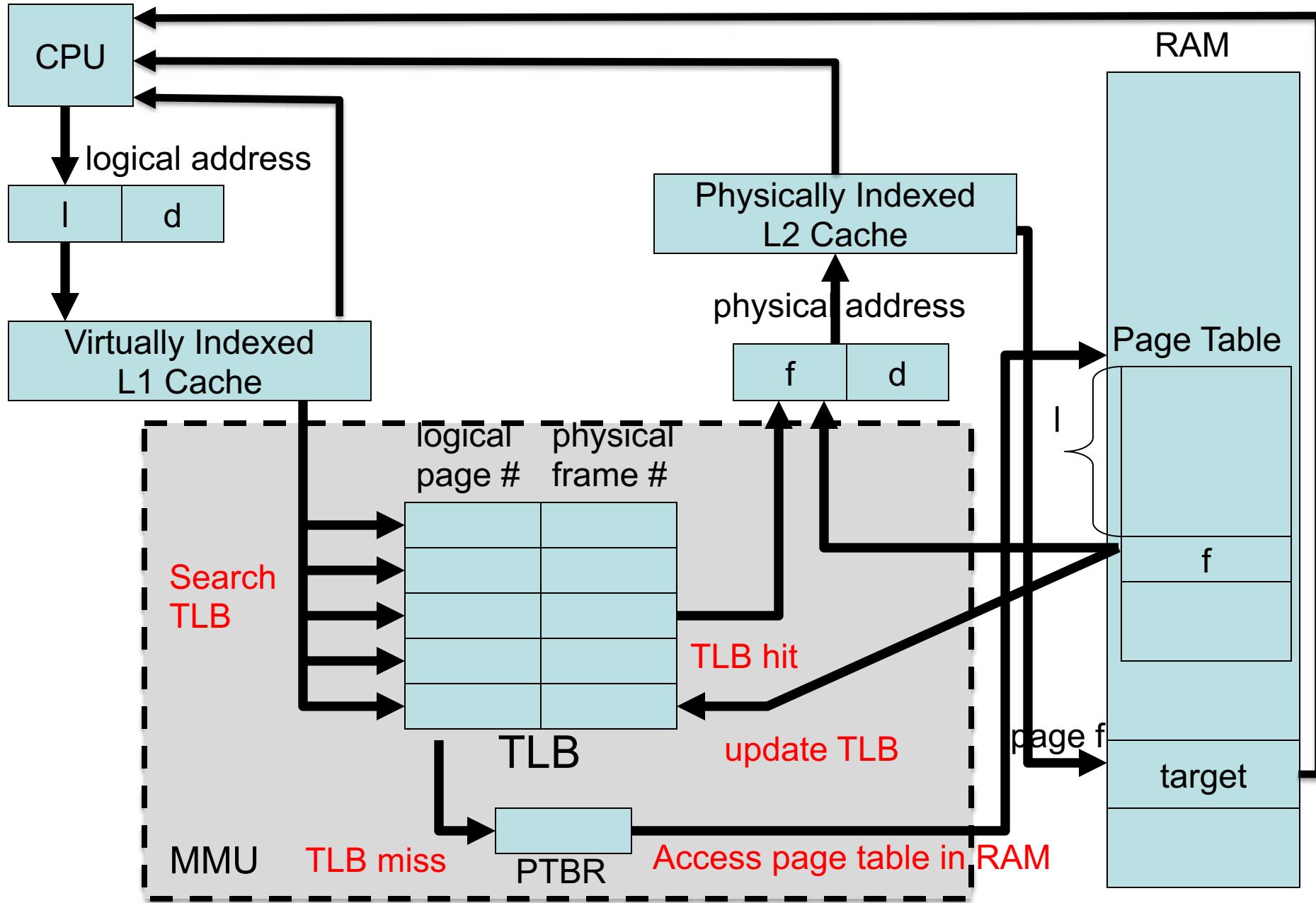
Paging and L1/L2 Caching

- Some solutions to the homonym problem:
 - Flush the cache on each context switch
 - process gets entire cache to itself, but have to rebuild cache
 - Add an address space id (process id) to each entry of the cache
 - so only data/instructions for the right process are returned for a given virtual address V
 - requires hardware support and an extra comparison
 - reduces available cache space for each process, since it has to be shared
 - Each process uses non-overlapping virtual addresses in its address space
 - unlikely, violates model that each process is compiled & executes independently in its own address space $[0, \text{MAX}]$

Paging and L1/L2 Caching

- In practice,
 - Most L1 caches are virtually indexed – fast
 - Most L2 caches are physically indexed
 - Each entry is unique
 - No collisions
 - This is good for code/data from shared library pages, i.e. if multiple processes share the same code/data, then it just has to be stored once in cache
 - The virtually indexed cache is essentially a small L1 cache, and the physically indexed cache is a much larger L2 cache.

• Paging with TLB and L1/L2 Caching



Shared Pages

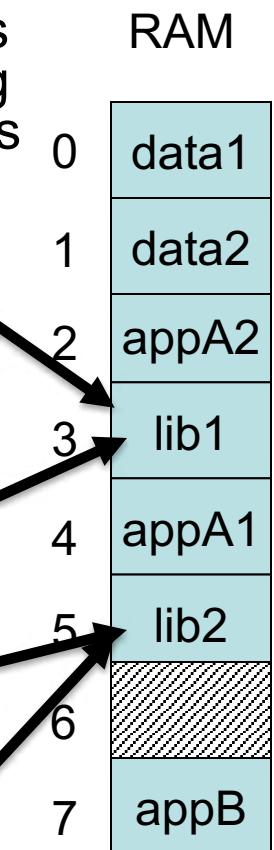
- Page tables can point to the *same* memory frames
 - e.g.: consider 2 app's A and B, running in 2 processes P1 and P2, sharing some C library functions consisting of 2 pages of code, lib1 and lib2, and each process has its own data

P1's logical address space P1's page table

0	appA1
1	appA2
2	lib1
3	lib2
4	data1

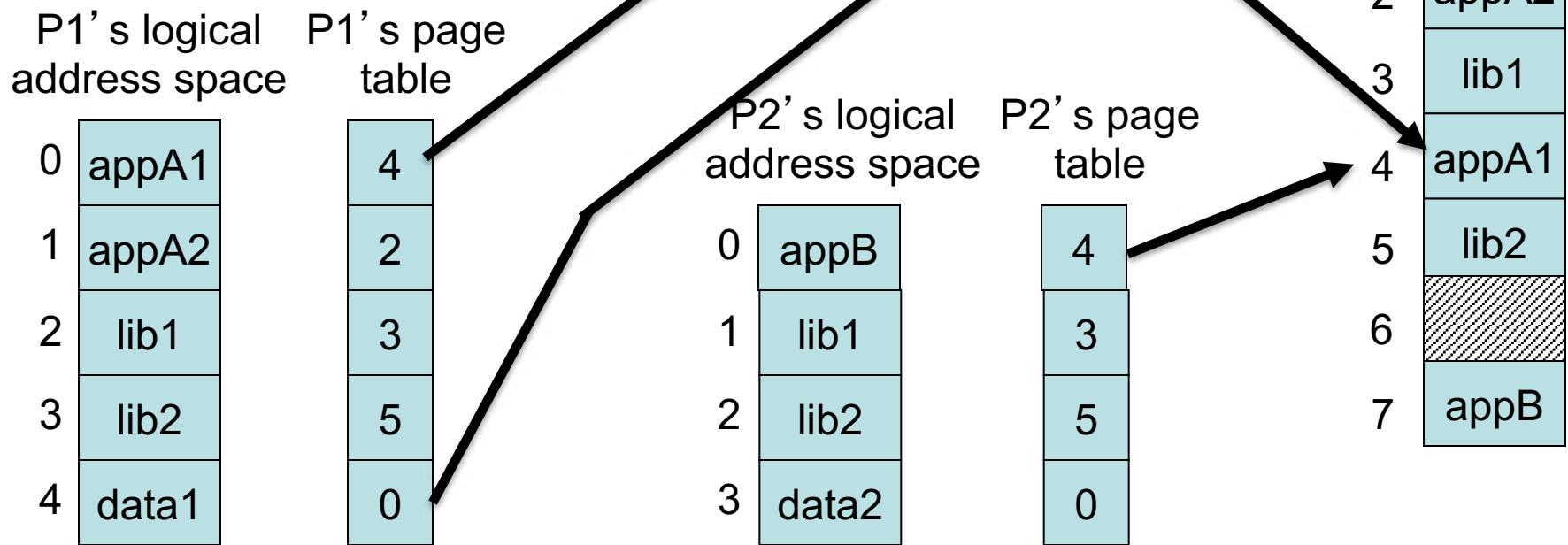
P2's logical address space P2's page table

0	appB
1	lib1
2	lib2
3	data2



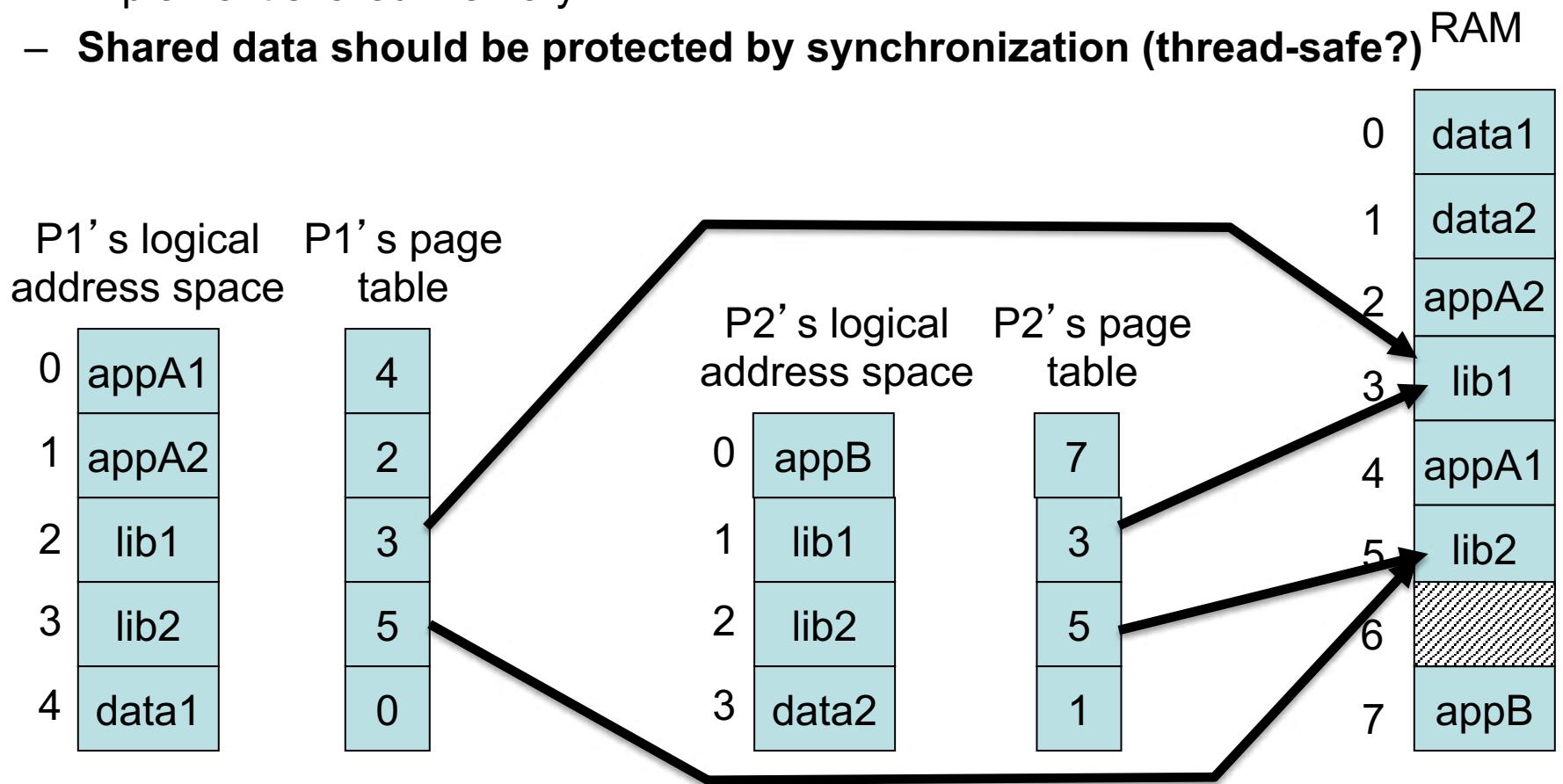
Shared Pages

- Can share code & data pages between processes by having entries in different process' page tables point to the same physical page/frame(s)



Shared Pages

- Sharing data:
 - Two or more processes may want to share memory between them, so pointing multiple page tables to the same data pages is a way to implement shared memory
 - **Shared data should be protected by synchronization (thread-safe?)** RAM



Shared Pages

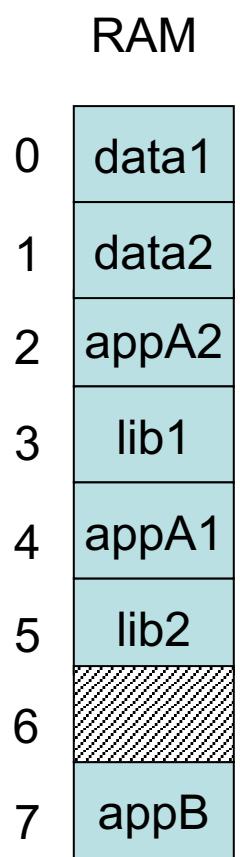
- Fork()'ing a child process causes the child to have a copy of the entire address space of the parent, including code
 - Rather than duplicating all such code pages, can simply map the child's page to the point to the same set of code pages as the parent
 - This is a way to implement copy-on-write

P1's logical address space P1's page table

0	appA1	4
1	appA2	2
2	lib1	3
3	lib2	5
4	data1	0

P2's logical address space P2's page table

0	appA1	4
1	appA2	2
2	lib1	3
3	lib2	5
4	data1	0



Shared Pages

- Fork()'ing a child process causes the child to have a copy of the entire address space of the parent, including code
 - Rather than duplicating all such code pages, can simply map the child's page to the point to the same set of code pages as the parent
 - This is a way to implement copy-on-write

P1's logical address space P1's page table

0	appA1	4
1	appA2	2
2	lib1	3
3	lib2	5
4	data1	0

P2's logical address space P2's page table

0	appA1	4
1	appA2	2
2	lib1	3
3	lib2	5
4	data2	1



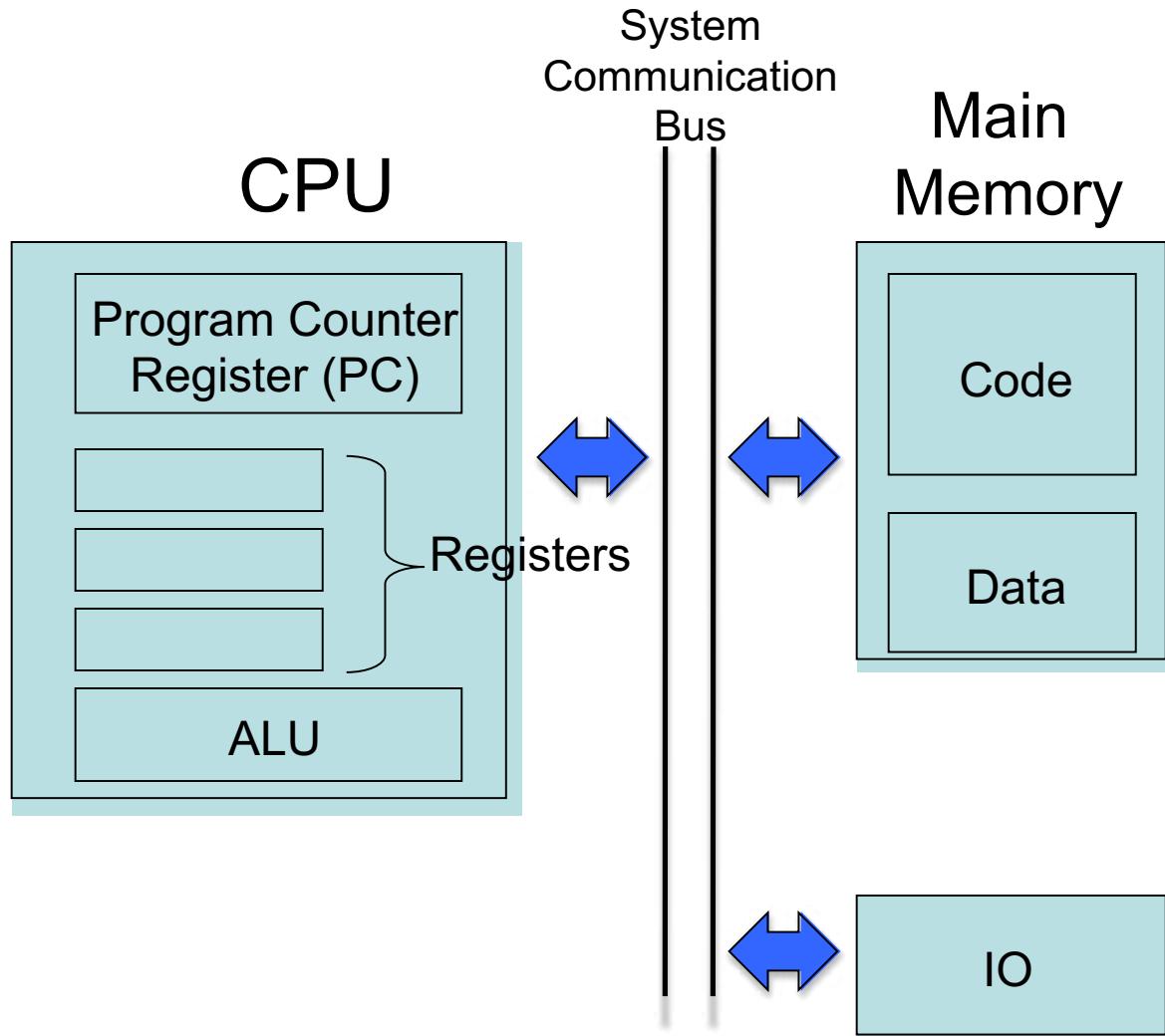


Lecture 16

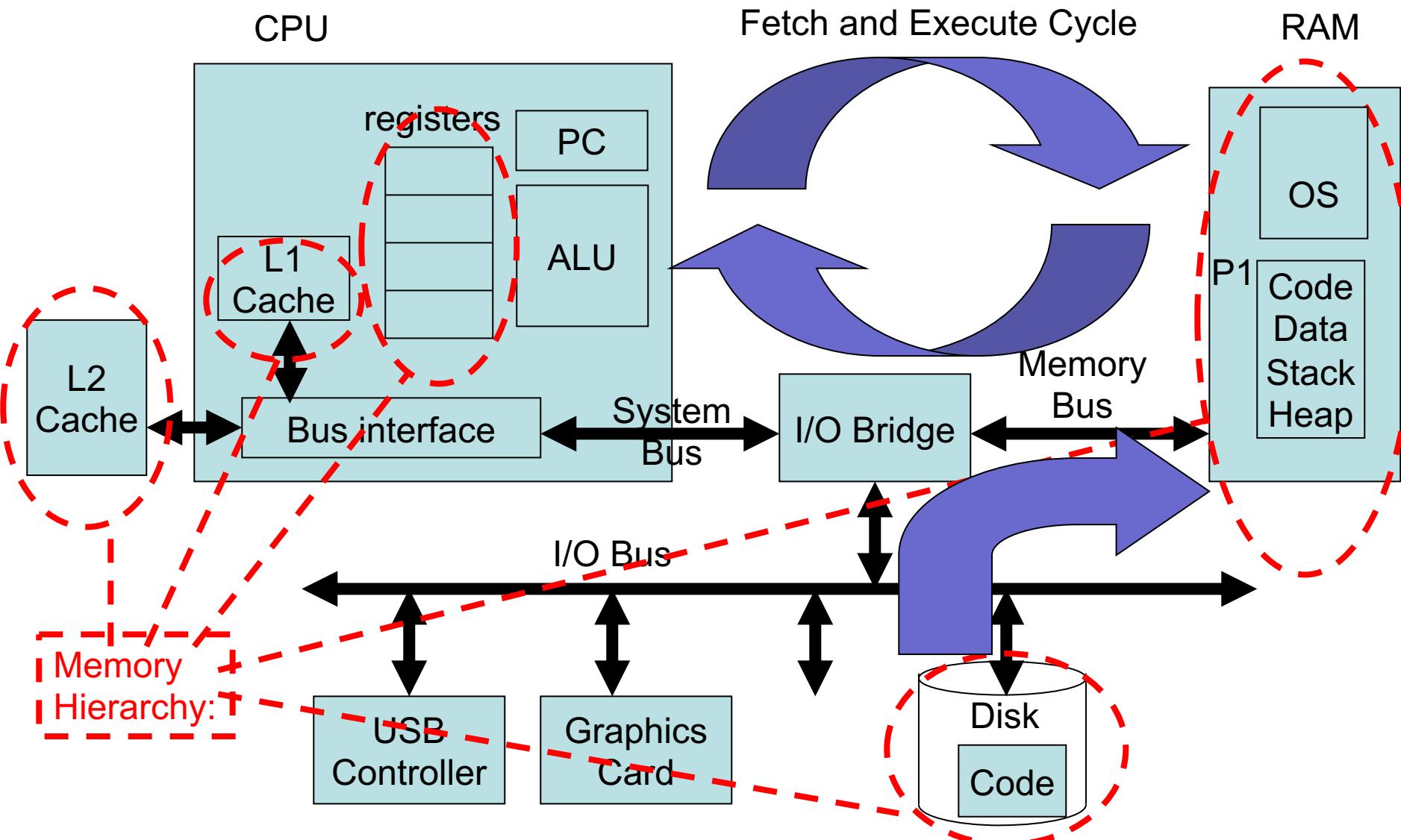
Memory Management



University of Colorado
Boulder



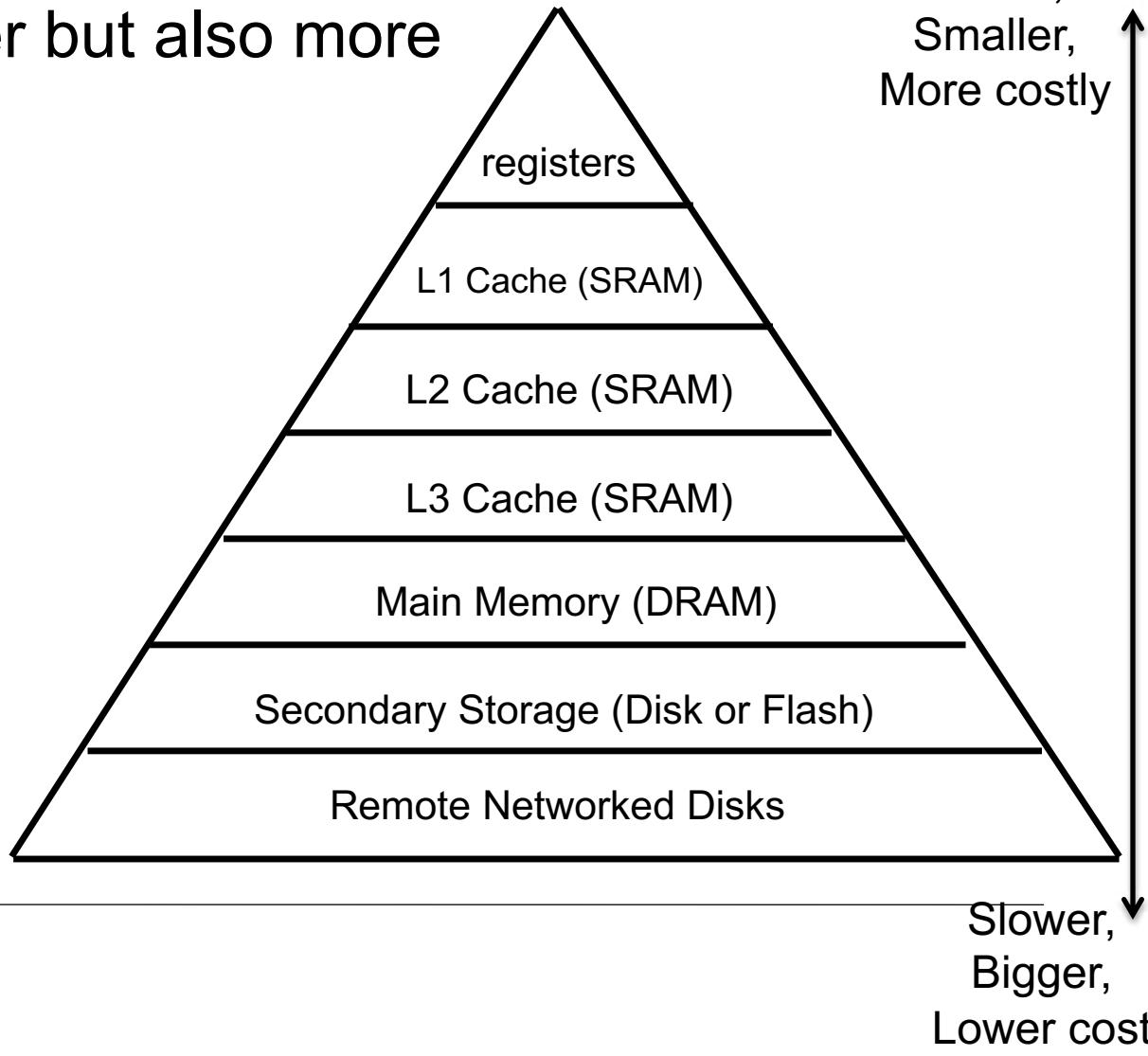
Memory Management



Memory Hierarchy

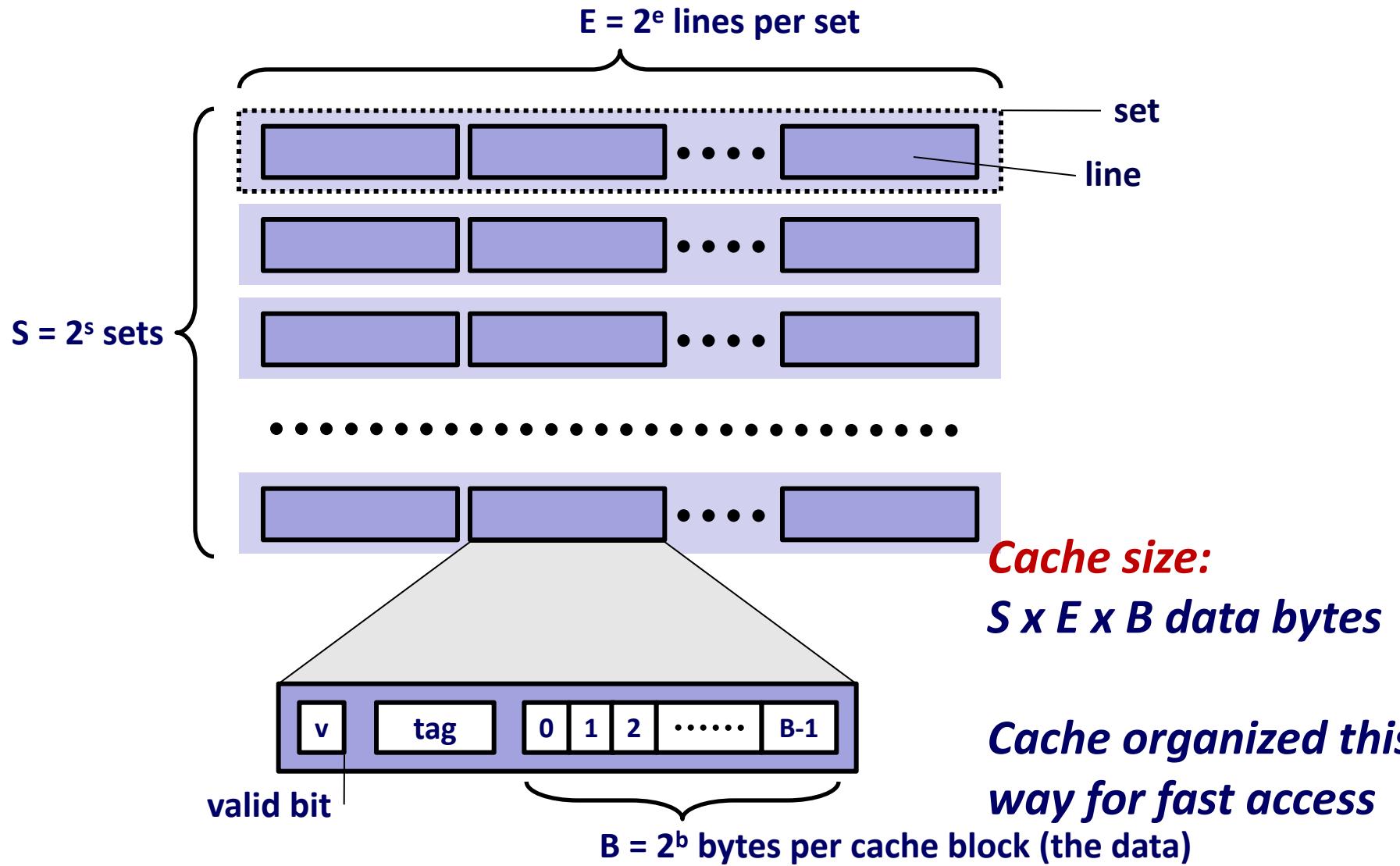
- cache frequently accessed instructions and/or data in local memory that is faster but also more expensive

- Register < 1 cycle (16 B)
- L1 = 1 cycle (~32 kB)
- L2 = 3 cycles (~512 kB)
- L3 = 10 cycles (2MB)
- RAM = 200 cycles (GB)
- Permanent storage:
 - Flash = 100k cycles (GB)
 - Disk = 1M cycles (TB)
 - Network = 1B cycles (PB)

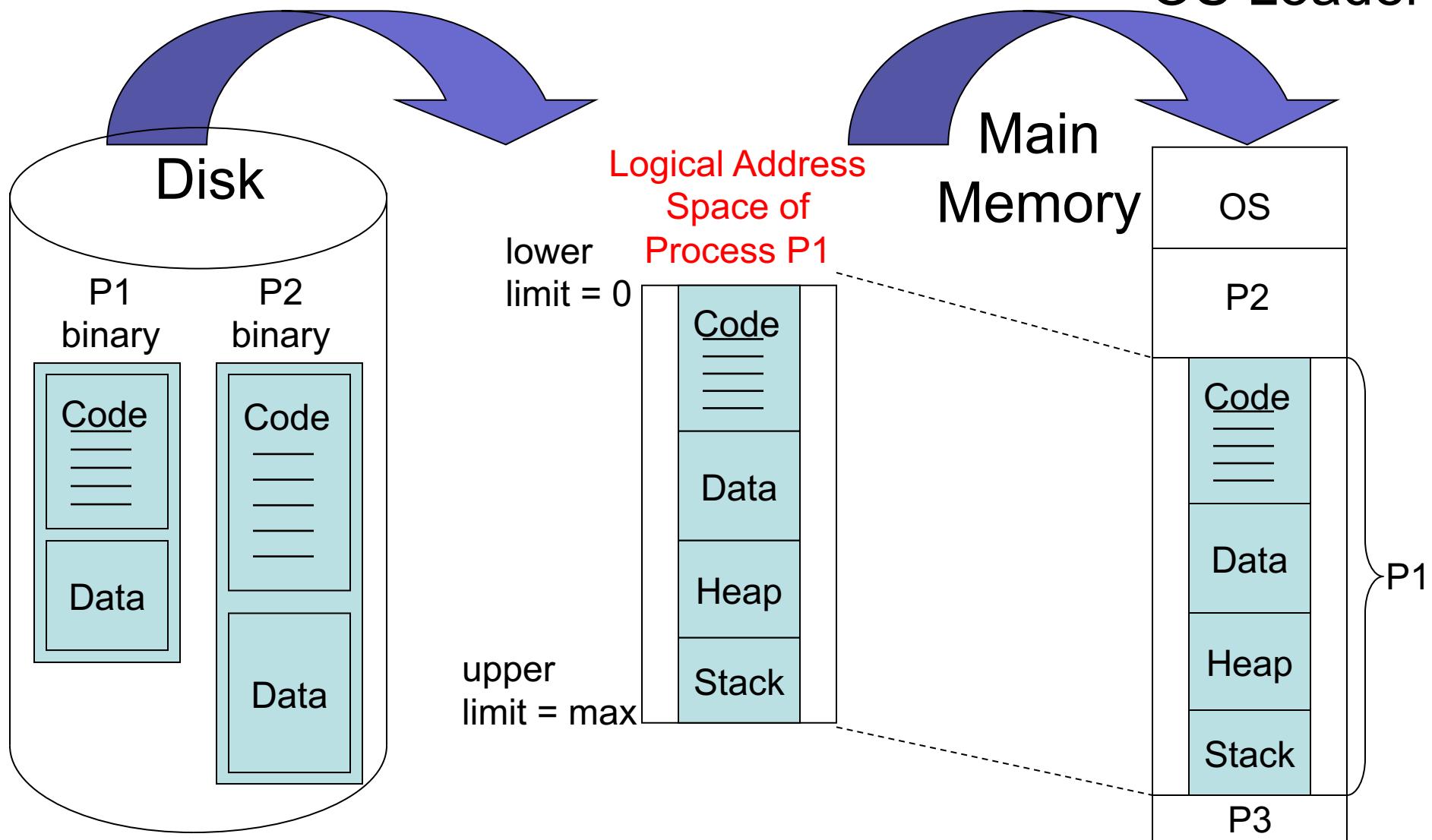


General Cache Organization (S, E, B)

For L1 hardware caches, access must be fast, so organize as follows:



Memory Management



Memory Management

- In the previous figure, want newly active process P1 to execute in its own *logical address space* ranging from 0 to max
 - It shouldn't have to know exactly where in physical memory its code and data are located
 - This decouples the compiler from run-time execution
 - There needs to be a mapping from logical addresses to physical addresses at run time
 - memory management unit (MMU) takes care of this.

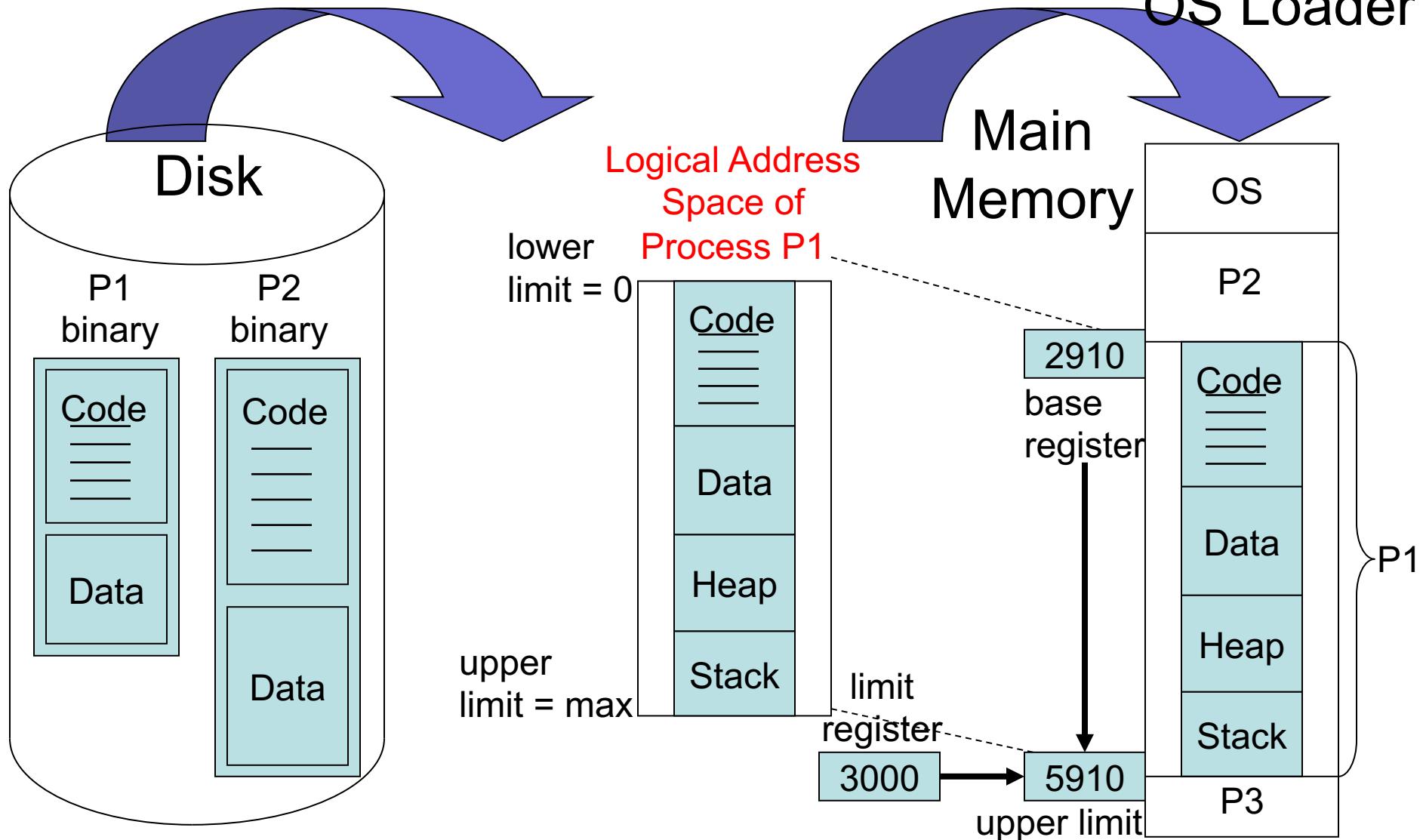
Memory Management

- MMU must do:
 1. Address translation: translate logical addresses into physical addresses, i.e. map the logical address space into a *physical address space*
 2. Bounds checking: check if the requested memory address is within the upper and lower limits of the address space

One approach is:

- *base register* in hardware keeps track of lower limit of the physical address space
- *limit register* keeps track of size of logical address space
- upper limit of physical address space = base register + limit register

Memory Management

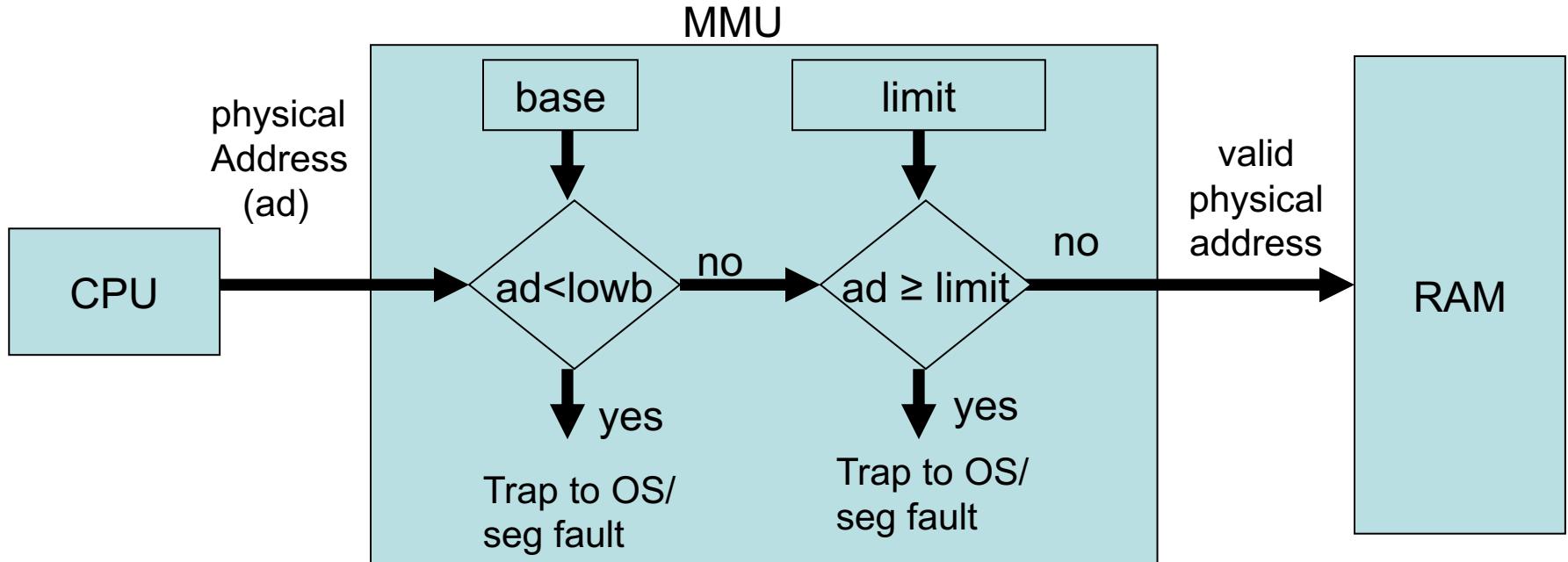


Memory Management

- base and limit registers provide hardware support for a simple MMU (Memory Management Unit)
 - memory access should not go out of bounds.
 - If out of bounds, then this is a segmentation fault so trap to the OS.
 - MMU will detect out-of-bounds memory access and notify OS by throwing an exception
- Only the OS can load the base and limit registers while in kernel/supervisor mode
 - these registers would be loaded as part of a context switch

Memory Management

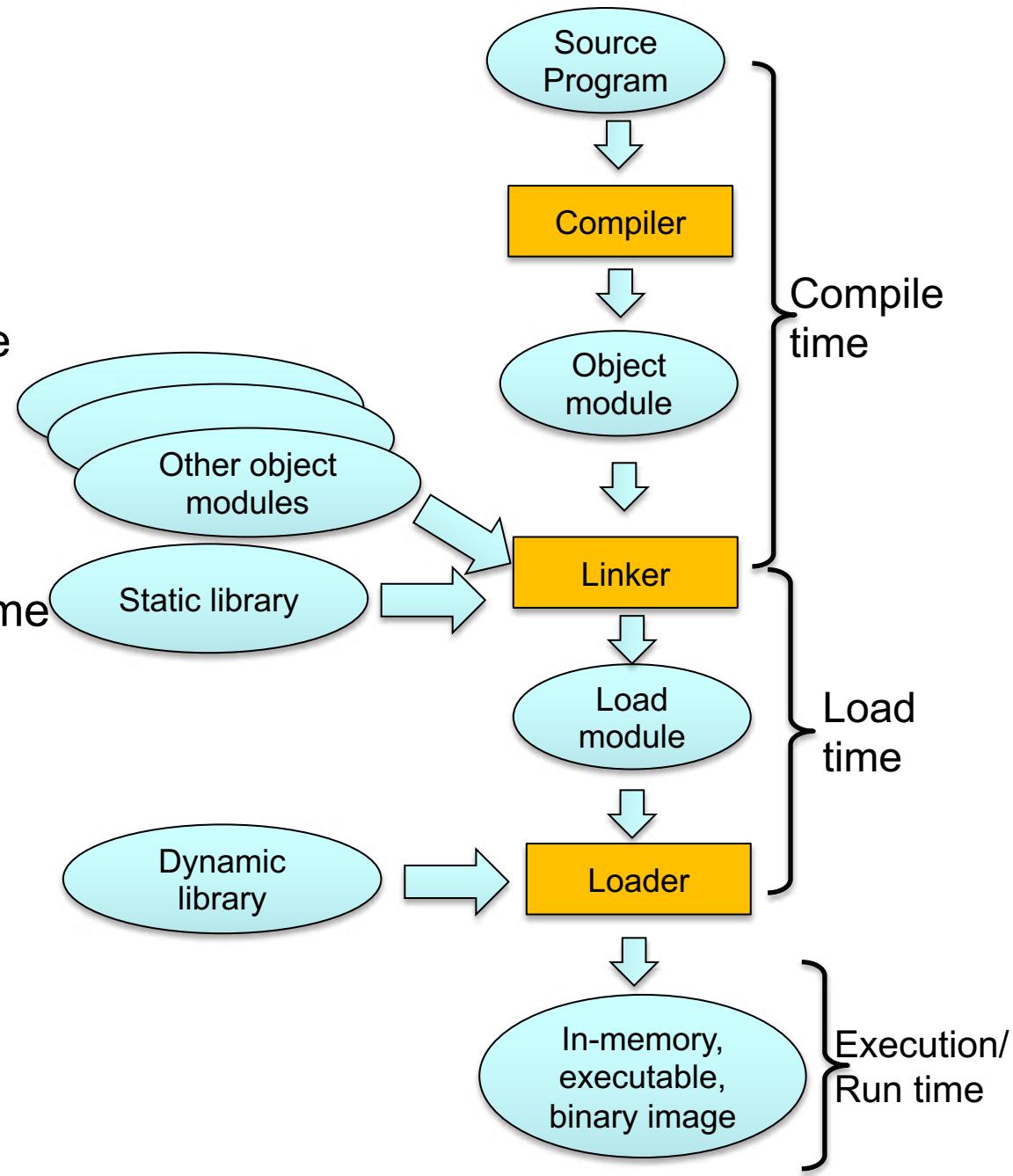
- MMU needs to check if physical memory access is out of bounds



Loading Tasks into Memory

- Using contiguous memory to hold task
- Task is stored as Code and Data on disk
- Task contains Code, Data, Stack, and Heap in memory
- Multiple tasks can be loaded in memory at the same time
- Kernel is responsible for protecting task memory from other tasks
 - Memory Management Unit(MMU) is used to validate all addresses.
 - Each task needs a base and limit register
 - Context switch must save/restore these address registers

- How does it know the addresses we want to access?
 - Binding at compile time
 - Binding at load time
 - Binding at execution time



Memory Management

- Address Binding at Compile Time:
 - If you know in advance where in physical memory a process will be placed, then compile your code with absolute physical addresses
 - Example: LOAD MEM_ADDR_X, reg1
STORE MEM_ADDR_Y, reg2

MEM_ADDR_X and MEM_ADDR_Y are hardwired by the compiler as absolute physical addresses

Memory Management

- Address Binding at Load Time
 - Code is first compiled in relocatable format.
 - **Replace logical addresses in code with physical addresses during loading**
 - Example: LOAD MEM_ADDR_X, reg1
STORE MEM_ADDR_Y, reg2

At load time, the loader replaces all occurrences in the code of MEM_ADDR_X and MEM_ADDR_Y with (base+MEM_ADDR_X) and (base+MEM_ADDR_Y).

- Once the binary has been thus changed, it is not really portable to any other area of memory, hence load time bound processes are not suitable for swapping (see later slides)

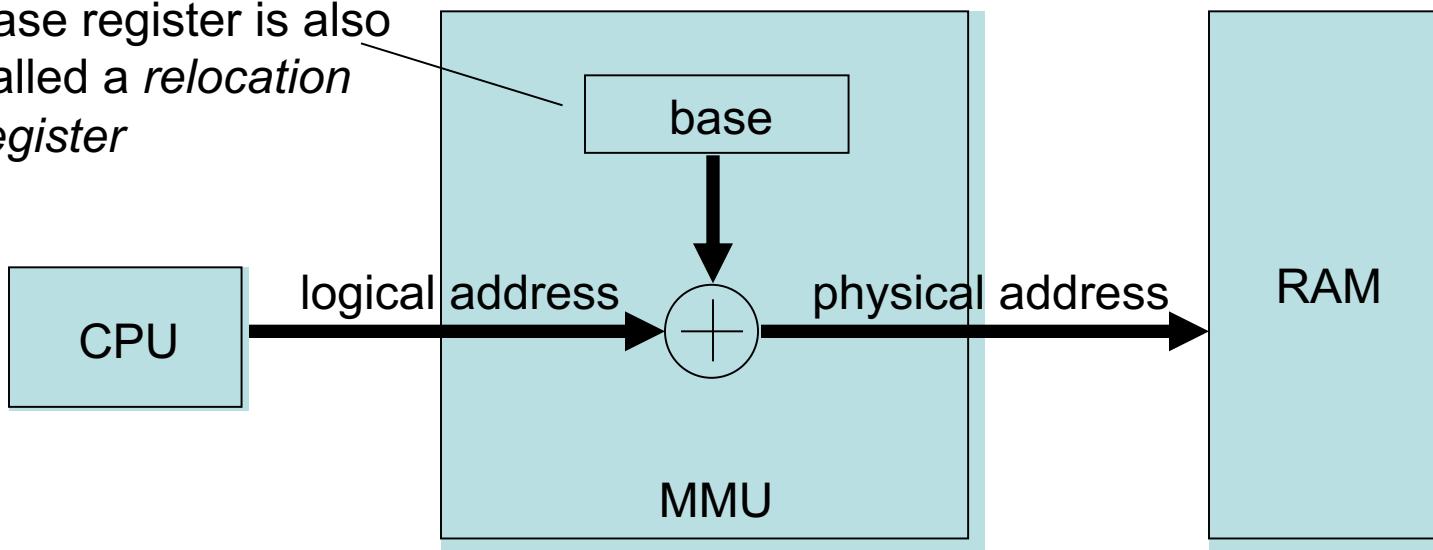


Memory Management

- Address Binding at Run Time (most modern OS's do this)
 - Code is first compiled in relocatable format as if executing in its own logical/virtual address space.
 - As each instruction is executed, i.e. at run time, the MMU relocates the logical address to a physical address using hardware support such as base/relocation registers.
 - Example: LOAD MEM_ADDR_X, reg1
MEM_ADDR_X is compiled as a logical address, and implicitly the **hardware MMU** will translate it to base+MEM_ADDR_X when the instruction executes

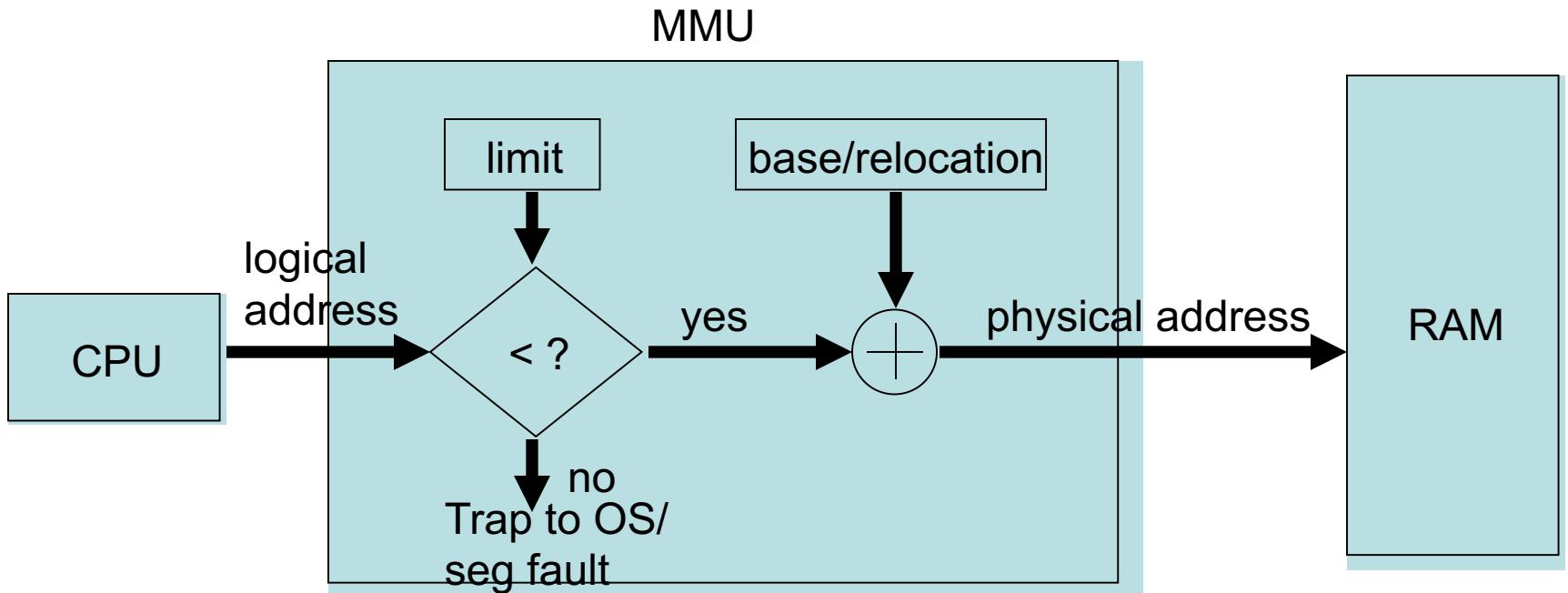
- MMU needs to perform run-time *mapping* of logical/virtual addresses to physical addresses
 - For run-time address binding,
 - each logical address is *relocated or translated* by MMU to a physical address that is used to access main memory/RAM
 - thus the application program never sees the actual physical memory - it just presents a logical address to MMU

base register is also
called a *relocation*
register



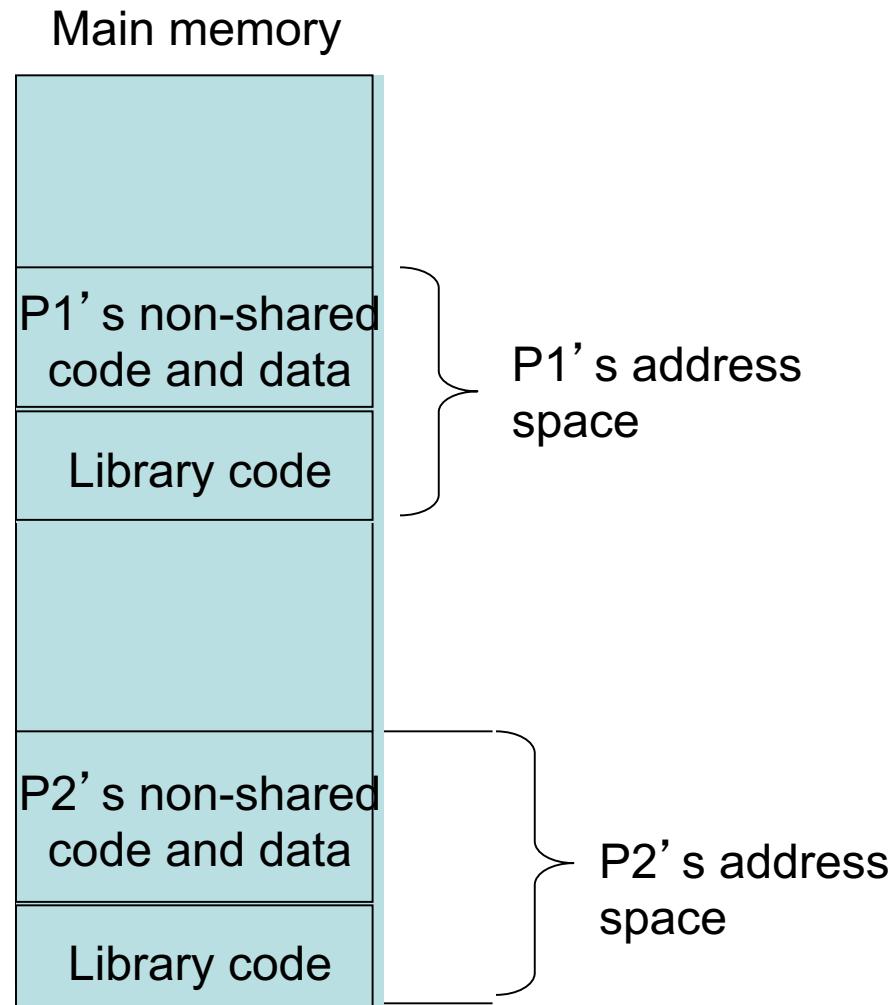
Memory Management

- Let's combine the MMU's two tasks (bounds checking, and memory mapping) into one figure
 - since logical addresses can't be negative, then lower bound check is unnecessary - just check the upper bound by comparing to the limit register
 - Also, by checking the limit first, no need to do relocation if out of bounds



Run Time Binding with Static Linking

- Advantages of static linking:
 - Applications have access to the same library code even if it has changed recently
 - Can move to machines without the library
 - Self contained processes

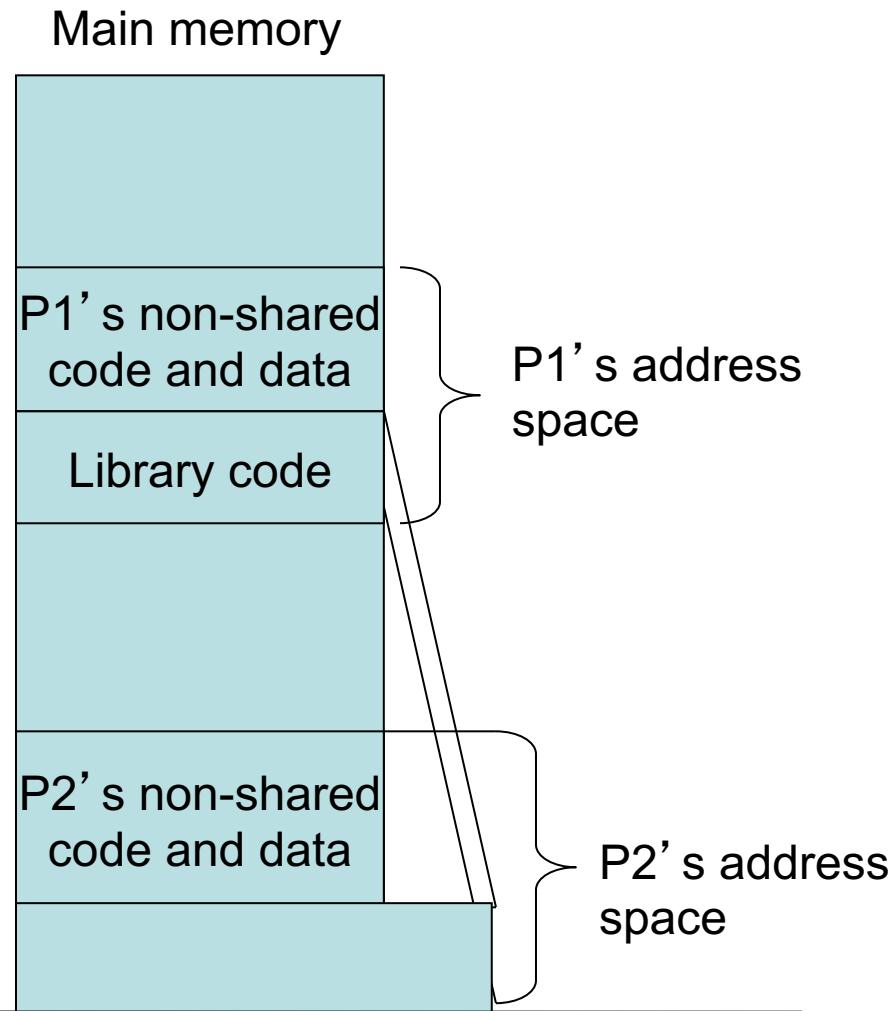


Run Time Binding

- Statically linked executable
 - Logical addresses are translated instruction by instruction into physical addresses at run time, *and the entire executable has all the code it needs at compile time through static linking*
 - Once a function is statically linked, it is embedded in the code and can't be changed except through recompilation
 - Your code can contain outdated functions that don't have the latest bug fixes, performance optimizations, and/or feature enhancements

Run Time Binding with Dynamic Linking

- Advantages of dynamic linking:
 - Applications have access to the latest code at run-time, e.g. most recent patched dlls,
 - Smaller size – stubs stay stubs unless activated
 - Can have only one copy of the code that is shared among all applications
 - We'll see later how code is shared between address spaces using page tables

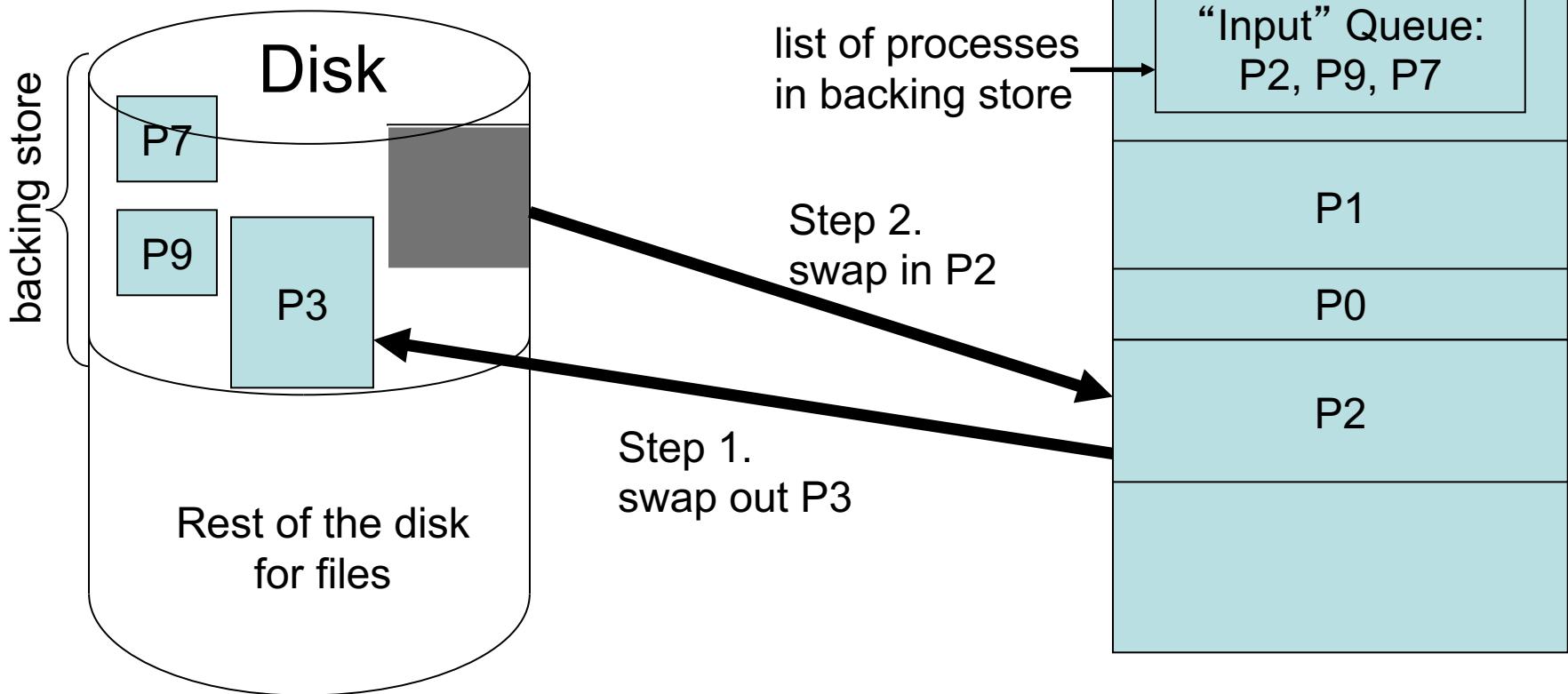




Swapping, Fragmentation, and Segmentation

Swapping

- When OS scheduler selects process P2, dispatcher checks if P2 is in memory.
- If not, there is not enough free memory, then swap out some process P_k , to make room



Swapping

- If run time binding is used, then a process can be easily swapped back into a different area of memory.
- If compile time or load time binding is used, then process swapping will become very complicated and slow - basically undesirable

Swapping Difficulties

- Context-switch time of swapping is very slow
 - Disks take on the order of 10s-100s of ms per access
 - When adding the size of the process to transfer, then transfer time can take seconds
 - Ideally hide this latency by having other processes to run while swap is taking place behind the scenes,
 - e.g. in RR, swap out the just-run process, and have enough processes in round robin to run before swap-in completes & newly swapped-in process is ready to run
 - can't always hide this latency if in-memory processes are blocked on I/O
 - avoids swapping unless the memory usage exceeds a threshold

Swapping Difficulties

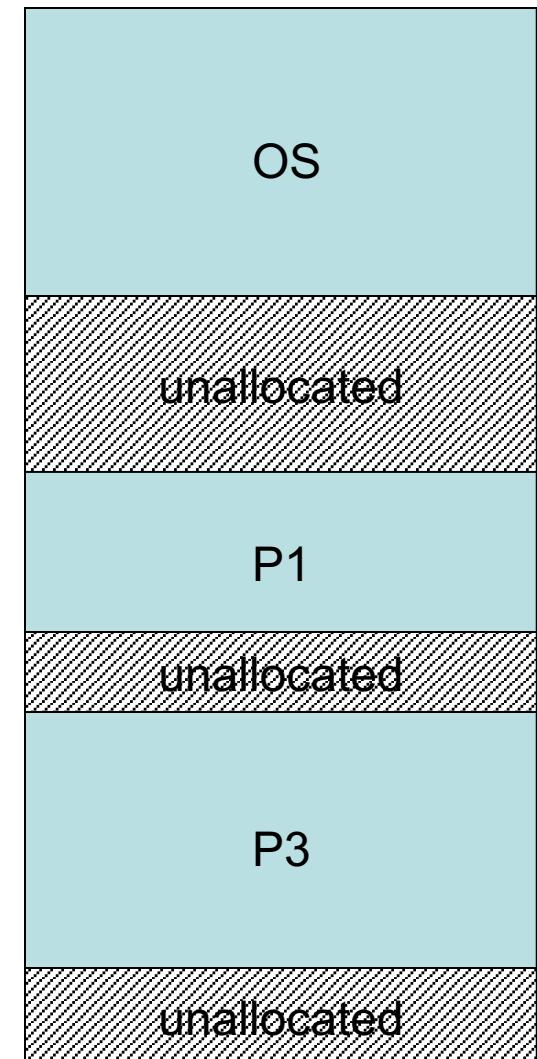
- swapping of processes that are blocked or waiting on I/O becomes complicated
 - one rule is to simply avoid swapping processes with pending I/O
- fragmentation of main memory becomes a big issue
 - can also get fragmentation of backing store disk
- Modern OS's swap portions of processes in conjunction with virtual memory and demand paging

Memory Allocation

as processes arrive, they're allocated a space in main memory

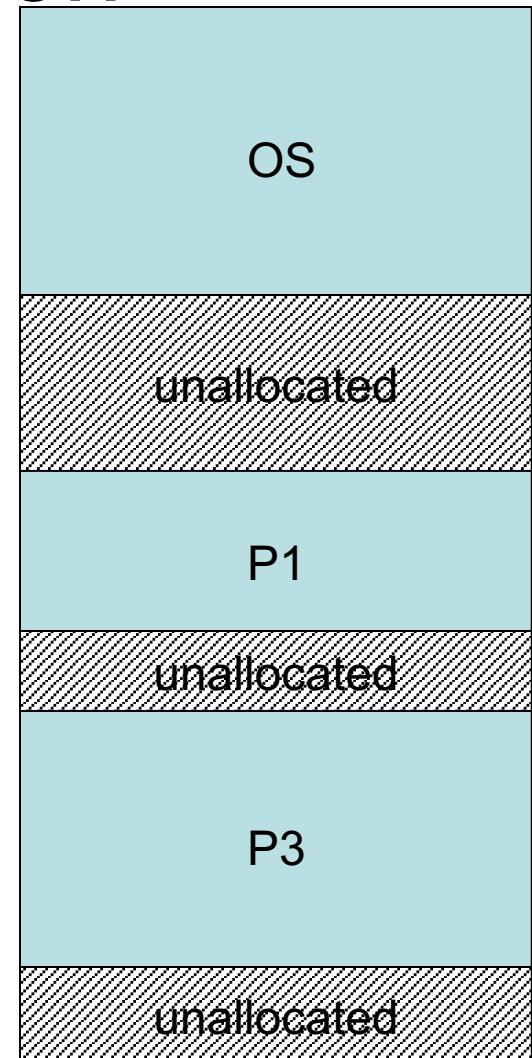
- **Allocation Strategies:**

- *best fit* - find the smallest chunk that is big enough
 - This results in more and more fragmentation
- *worst fit* - find the largest chunk that is big enough
 - this leaves the largest contiguous unallocated chunk for the next process
- *first fit* - find the 1st chunk that is big enough
 - This tends to fragment memory near the beginning of the list
- *next fit* – view fragments as forming a circular buffer
 - find the 1st chunk that is big enough after the most recently chosen fragment



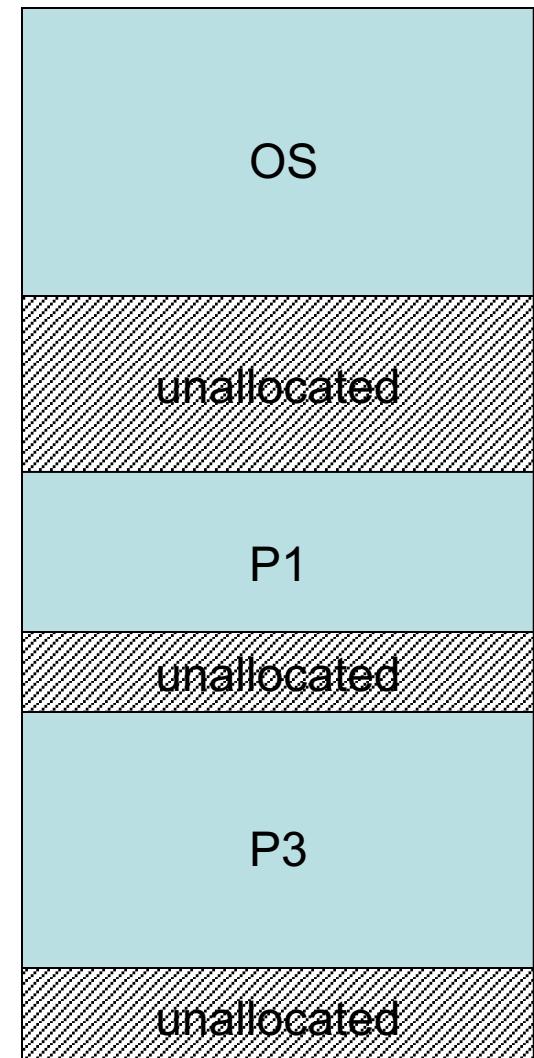
Memory Allocation

- over time, processes leave, and memory is deallocated
- results in *fragmentation* of main memory
 - There are many small chunks of non-contiguous unallocated memory between allocated processes in memory
- for the next process,
 - OS must find a large enough unallocated chunk in fragmented memory
 - May have enough memory, but not contiguous to allow a process to load

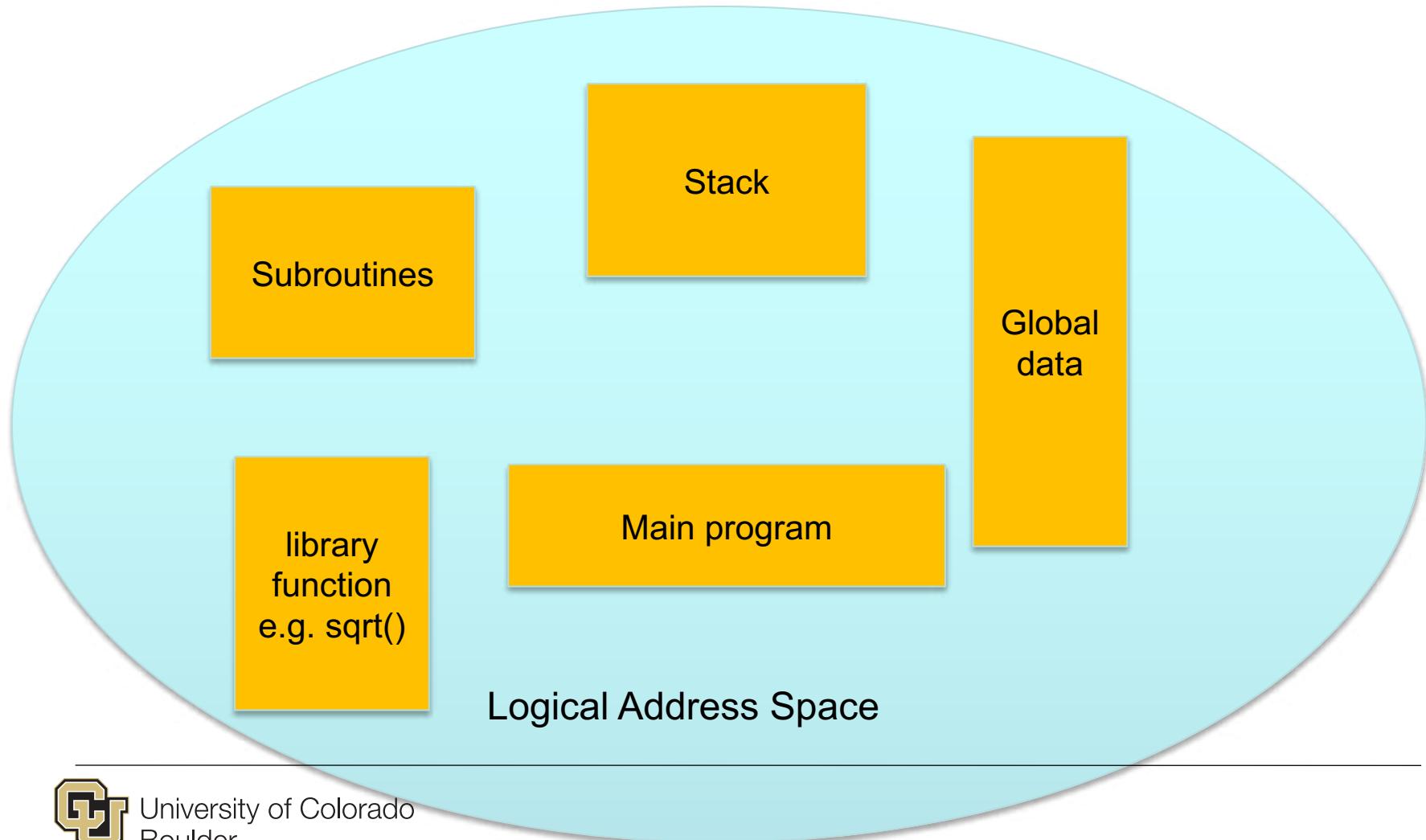


Fragmentation Problem

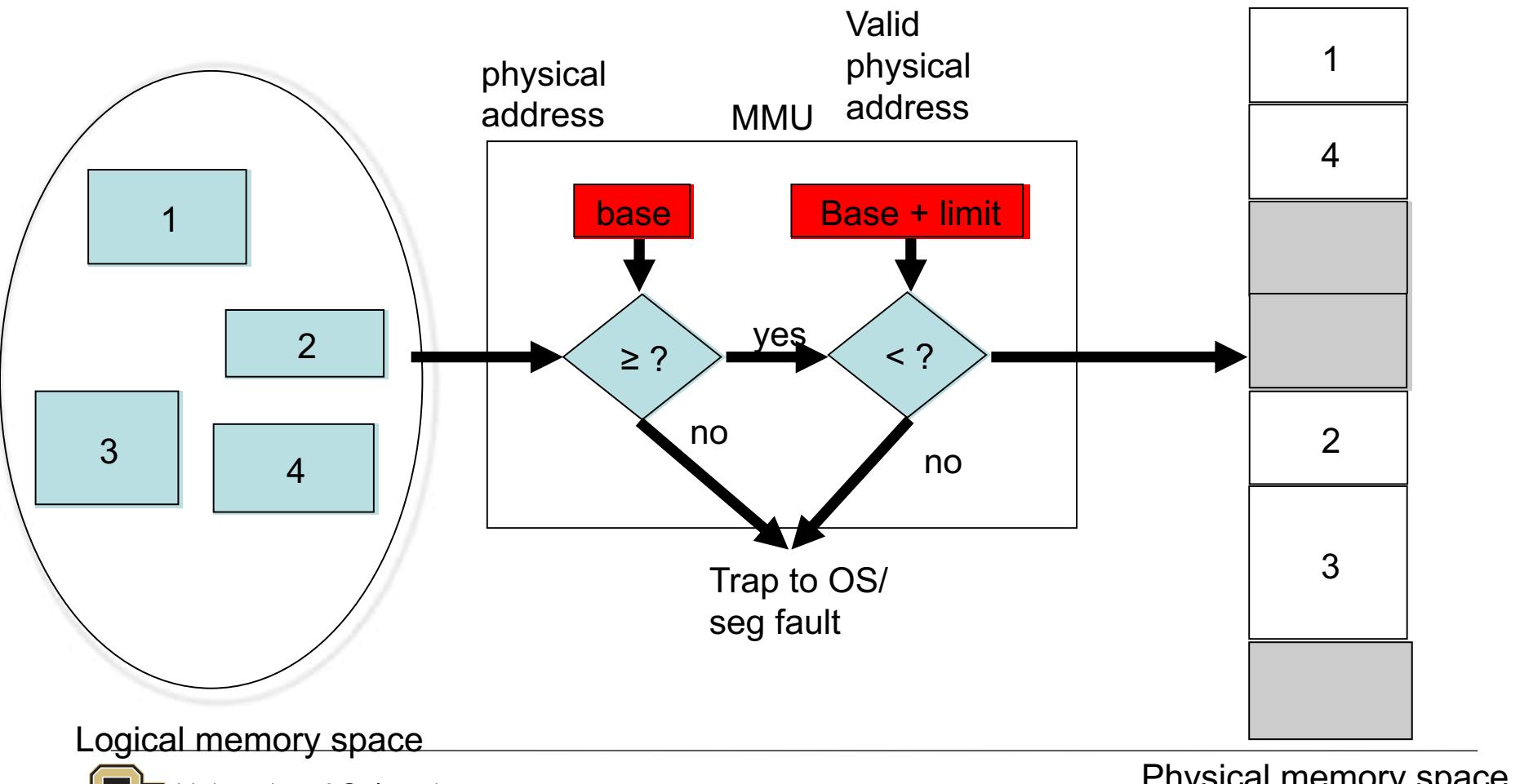
- This results in ***external fragmentation*** of main memory
 - There are many small chunks of non-contiguous unallocated memory between allocated processes in memory
- OS must find a large enough unallocated chunk in fragmented memory that a process will fit into
- De-fragmentation/Compaction
 - Algorithms for recombining contiguous segments is used, but memory still gets fragmented
 - Moving memory is expensive



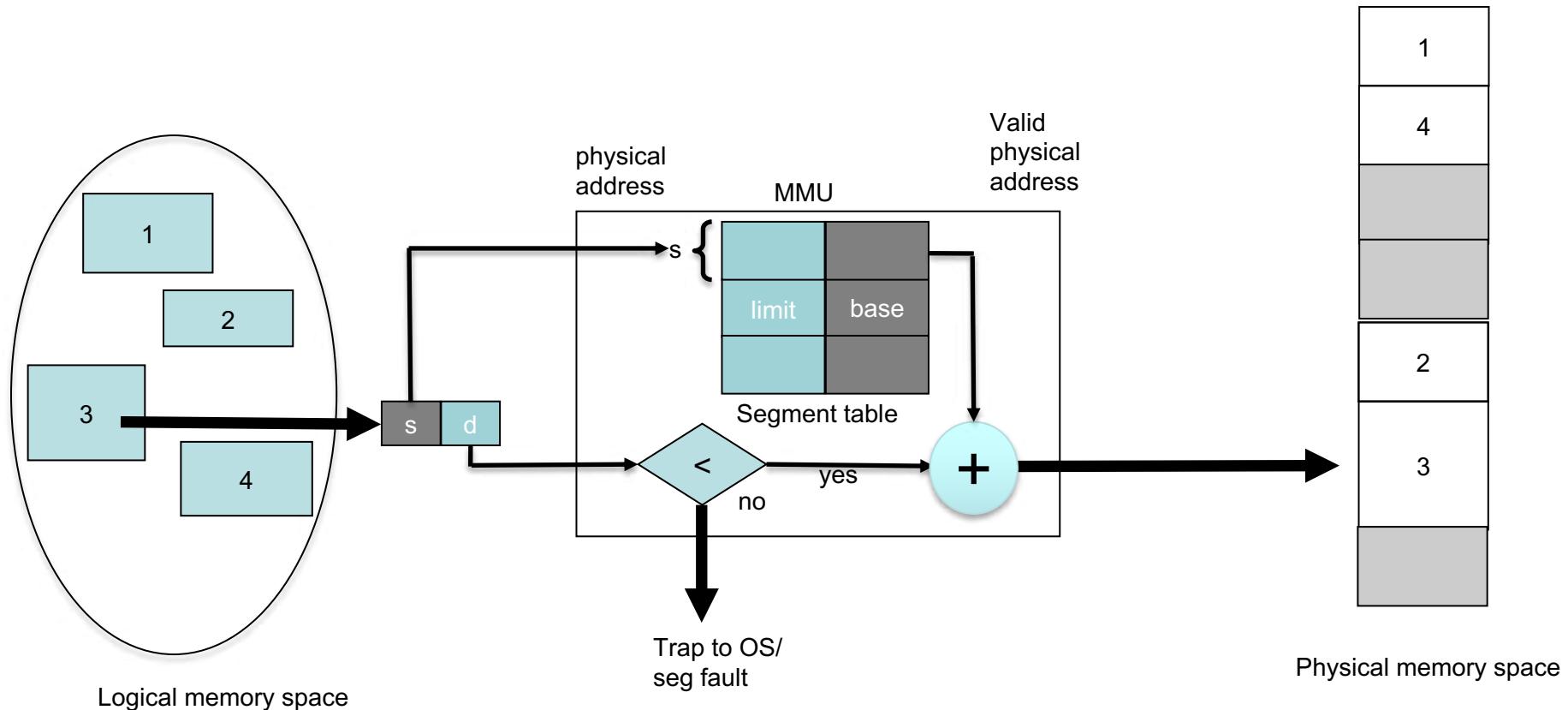
Instead of one big address space break it up into smaller segments



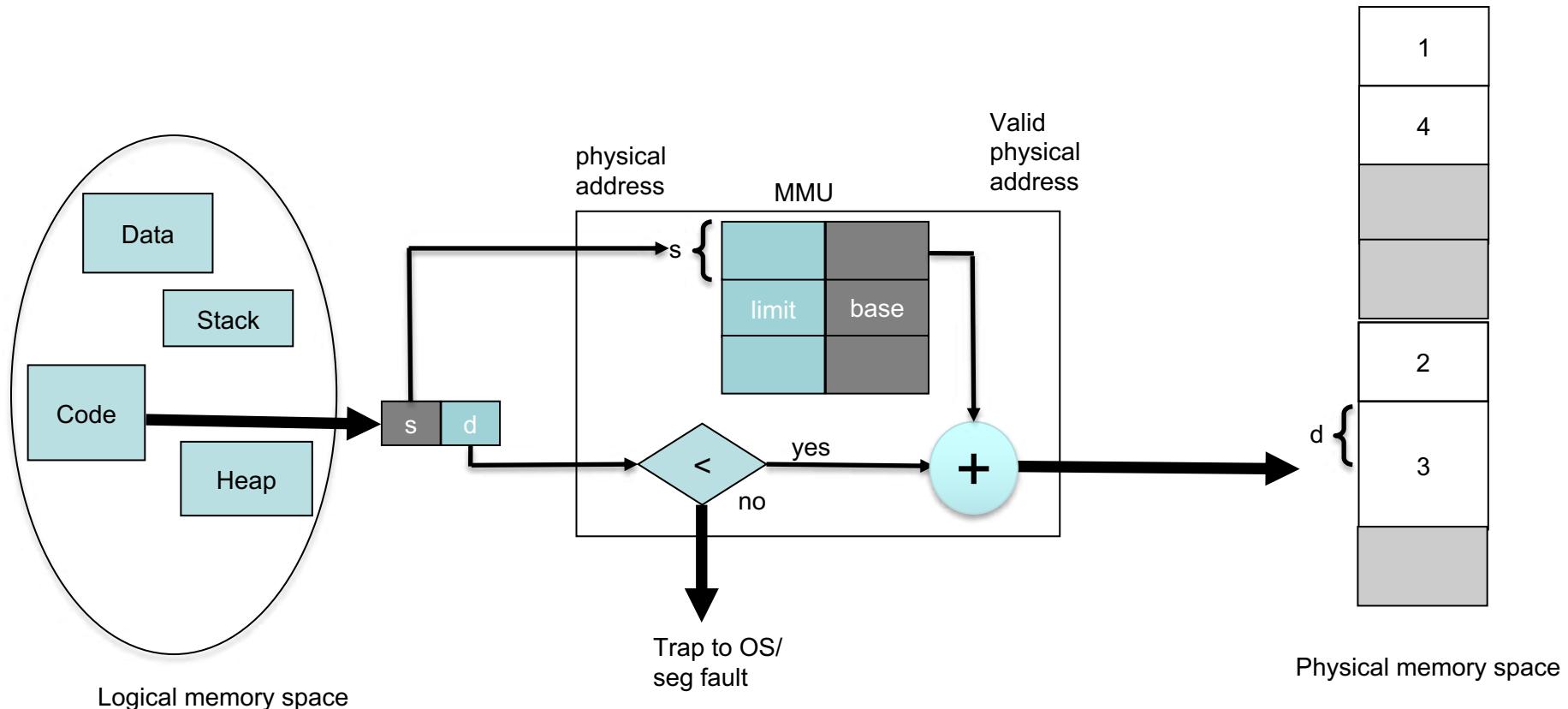
Segmentation



Segmentation of Memory



Segmentation of Memory

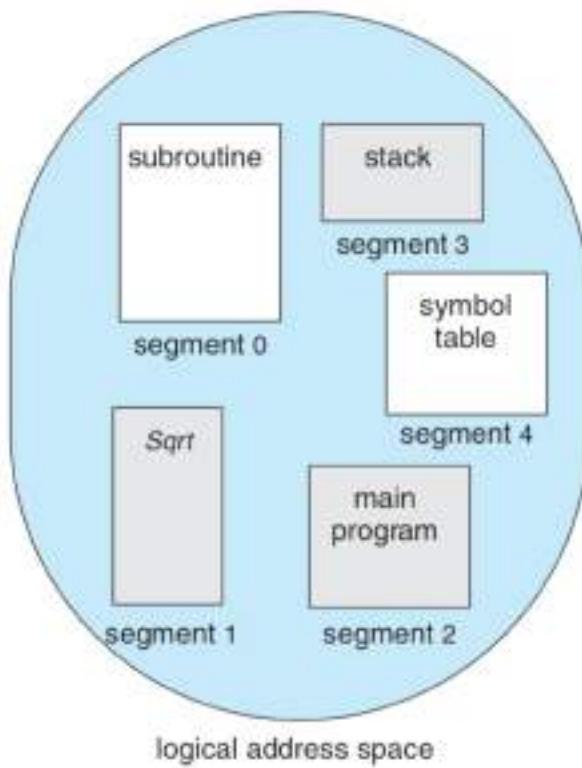


Segmentation allows physical address of a process to be non-contiguous



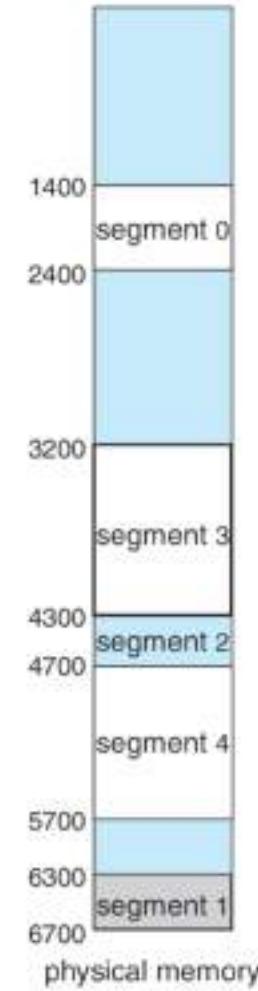
University of Colorado
Boulder

Example



	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

segment table



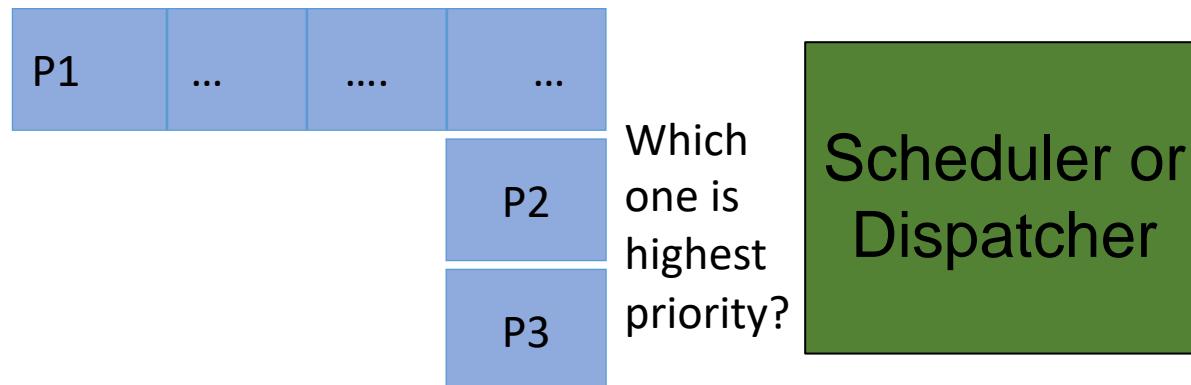


Lecture 15

More Scheduling Policies

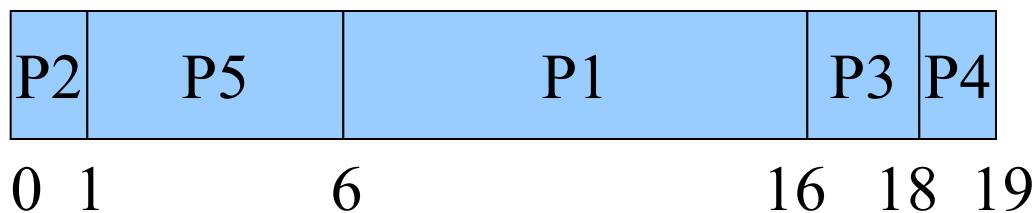
Priority-based Scheduling

- Assign each task a priority, and schedule higher priority tasks first, before lower priority tasks
- Any criteria can be used to decide on a priority
 - measurable characteristics of the task
 - external criteria based on the “importance” of the task
 - example: foreground processes may get high priority, while background processes get low priority



Priority-based Scheduling

Process	CPU Execution Time	Priority
P1	10	3
P2	1	1
P3	2	4
P4	1	5
P5	5	2



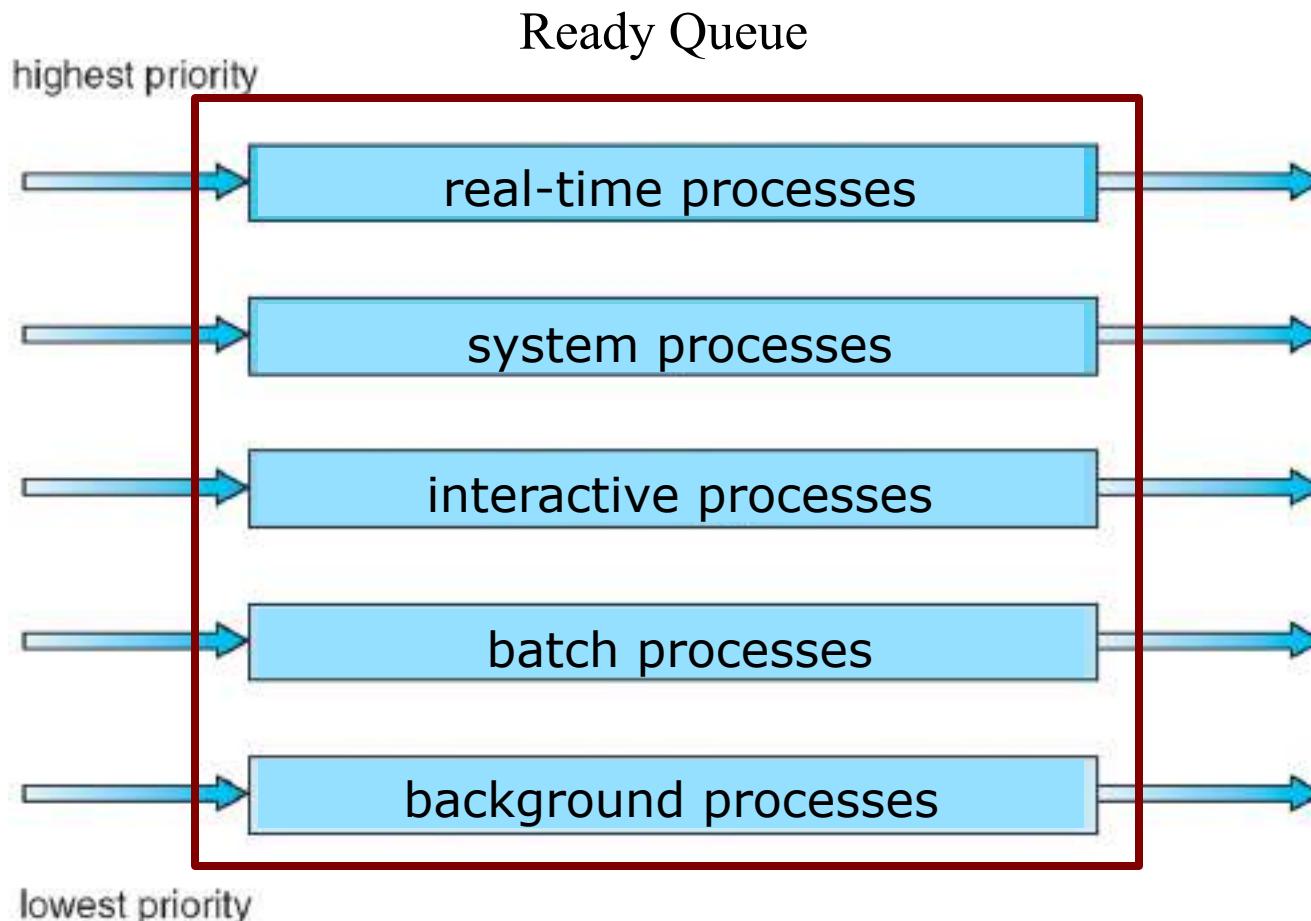
Priority-based Scheduling

- Can be preemptive:
 - A higher priority process arriving in the ready queue can preempt a lower priority running process
 - Switch can occur if the lower priority process:
 - Yields CPU with a system call
 - Is interrupted by a timer interrupt
 - Is interrupted by a hardware interrupt
 - Each of these cases gives control back to the OS, which can then schedule the higher priority process

Priority-based Scheduling

- **Multiple tasks with the same priority are scheduled according to some policy**
 - FCFS, round robin, etc.
- **Each priority level has a set of tasks, forming a *multi-level queue***
 - Each level's queue can have its own scheduling policy
- **We use priority-based scheduling and multi-level queue scheduling interchangeably**

Multilevel Queue Scheduling

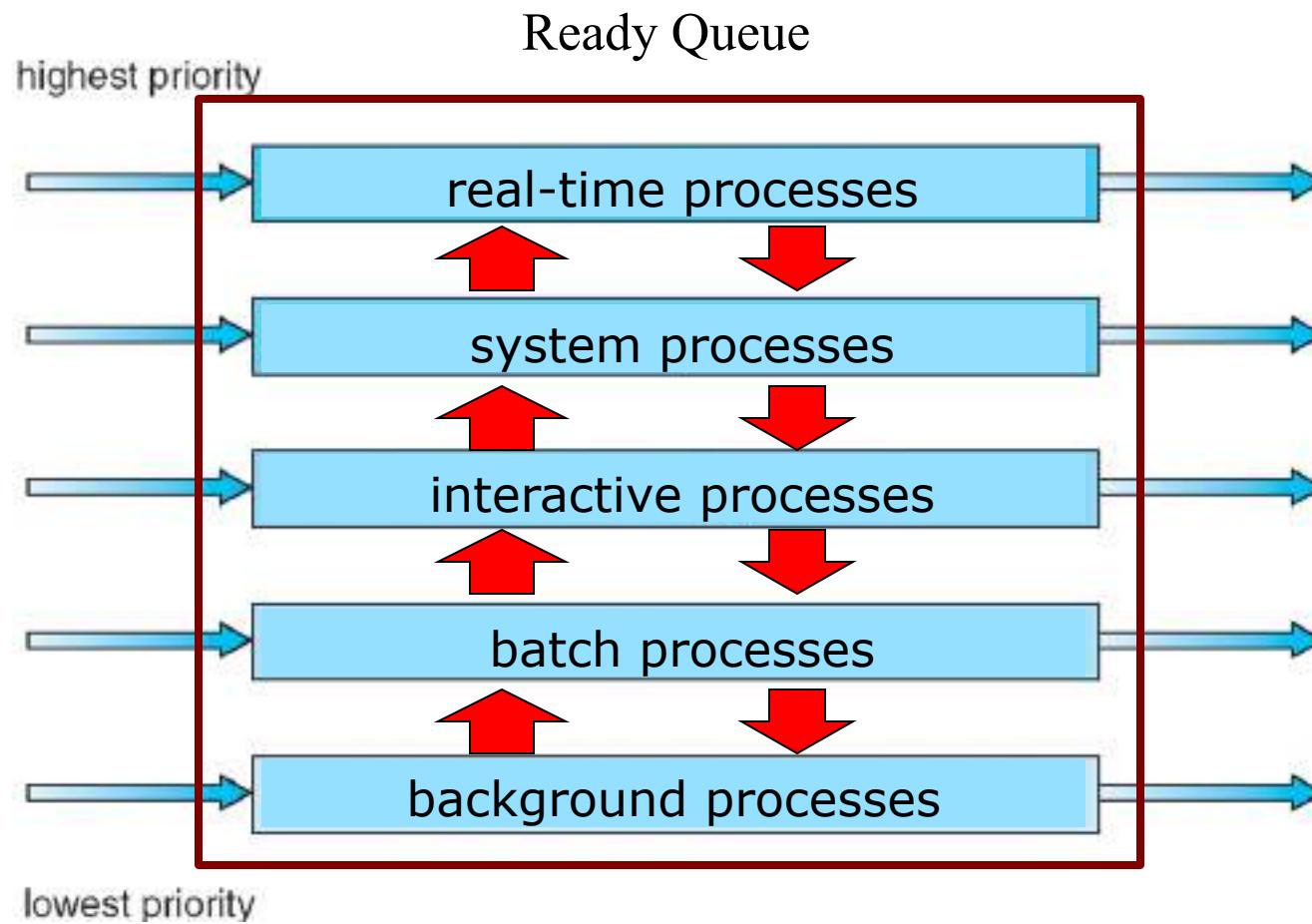


Priority-based Scheduling

- **Preemptive priorities can starve low priority processes**
 - A higher priority task always gets served ahead of a lower priority task, which never sees the CPU
- **Some starvation-free solutions:**
 - Assign each priority level a proportion of time, with higher proportions for higher priorities, and rotate among the levels
 - Similar to weighted round robin, except across levels
 - Create a *multi-level feedback queue* that allows a task to move up/down in priority
 - Avoids starvation of low priority tasks

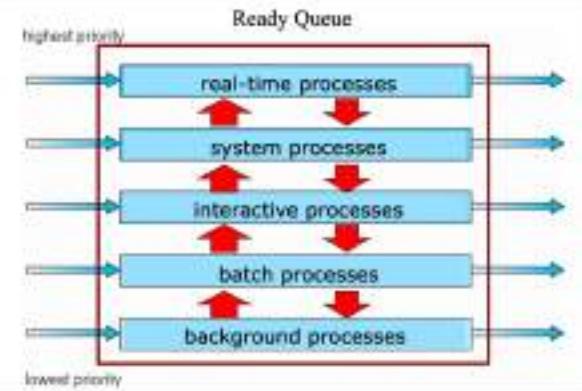


Multilevel Feedback Queue Scheduling



Multilevel Feedback Queue

- Multilevel-feedback-queue scheduler defined by the following parameters:
 - number of queues
 - scheduling algorithms for each queue
 - method used to determine when to upgrade a process
 - method used to determine when to demote a process
 - method used to determine which queue a process will enter when that process needs service



Multi-level Feedback Queues

- **Criteria for process movement among priority queues could depend upon age of a process:**
 - old processes move to higher priority queues, or conversely, high priority processes are eventually demoted
 - sample aging policy: if priorities range from 1-128, can decrease (increment) the priority by 1 every T seconds
 - eventually, the low priority process will get scheduled on the CPU

Multi-level Feedback Queues

- **Criteria for process movement among priority queues could depend upon behavior of a process:**
 - CPU-bound processes move down the hierarchy of queues, allowing interactive and I/O-bound processes to move up
 - give a time slice to each queue, with smaller time slices higher up
 - if a process doesn't finish by its time slice, it is moved down to the next/lower queue
 - over time, a process gravitates towards the time slice that typically describes its average local CPU burst



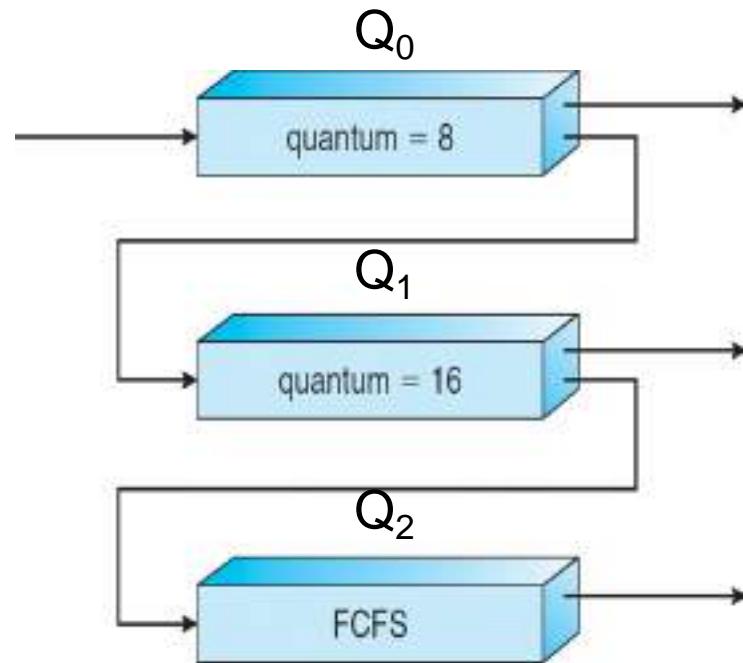
Example of Multilevel Feedback Queue

- **Three queues:**

- Q_0 – RR with time quantum 8 milliseconds
- Q_1 – RR time quantum 16 milliseconds
- Q_2 – FCFS

- **Scheduling**

- A new job enters queue Q_0 , job receives 8 ms
- If it does not finish in 8 ms, job is preempted and moved to Q_1
- At Q_1 job receives additional 16 ms.
- If it still does not complete, it is preempted and moved to Q_2
- Interactive processes are more likely to finish early, processing only a small amount of data
- Compute-bound processes will exhaust their time slice



Interactive processes will gravitate towards higher priority queues, while Compute bound will move to lower priority queues

Priority-based Scheduling

- In Unix/Linux, you can *nice* a process to set its priority, within limits
 - e.g. priorities can range from -20 to +20, with lower values giving higher priority, a process with ‘nice +15’ is “nicer” to other processes by incrementing its value (which lowers its priority)
 - E.g. if you want to run a compute-intensive process compute.exe with low priority, you might type at the command line “nice –n 19 compute.exe”
 - To lower the niceness, hence increase priority, you typically have to be root
 - Different schedulers will interpret/use the nice value in their own ways



Multi-level Feedback Queues

- In Windows XP and Linux, system & real-time tasks are grouped in a priority range higher than the priority range for non-real-time tasks
 - XP has 32 priorities
 - 1-15 are for normal processes, 16-31 are for real-time processes.
 - One queue for each priority.
 - XP scheduler traverses queues from high priority to low priority until it finds a process to run
 - Linux has
 - priorities 0-99 are for important/real-time processes
 - 100-139 are for 'nice' user processes.
 - Lower values mean higher priorities.
 - Also, longer time quanta for higher priority tasks
 - 200 ms for highest priority
 - Only 10 ms for lowest priority



Linux Priorities and Timeslice length

numeric priority	relative priority	time quantum
0	highest	200 ms
•		
•		
•		
99		
100		
•		
•		
•		
140	lowest	10 ms

Multi-level Feedback Queues

- **Most modern OSs use or have used multi-level feedback queues for priority-based preemptive scheduling**
 - e.g. Windows NT/XP, Mac OS X, FreeBSD/NetBSD and Linux pre-2.6
 - Linux 1.2 used a simple round robin scheduler
 - Linux 2.2 introduced scheduling classes (priorities) for real-time and non-real-time processes and SMP (symmetric multi-processing) support

More Linux Scheduler History

- **Linux 2.4 introduced an $O(N)$ scheduler – help interactive processes**
 - If an interactive process yields its time slice before it's done, then its "goodness" is rewarded with a higher priority next time it executes
 - Keep a list of goodness of all tasks.
 - But this was unordered. So had to search over entire list of N tasks to find the "best" next task to schedule – hence $O(N)$
 - doesn't scale well

More Linux Scheduler History

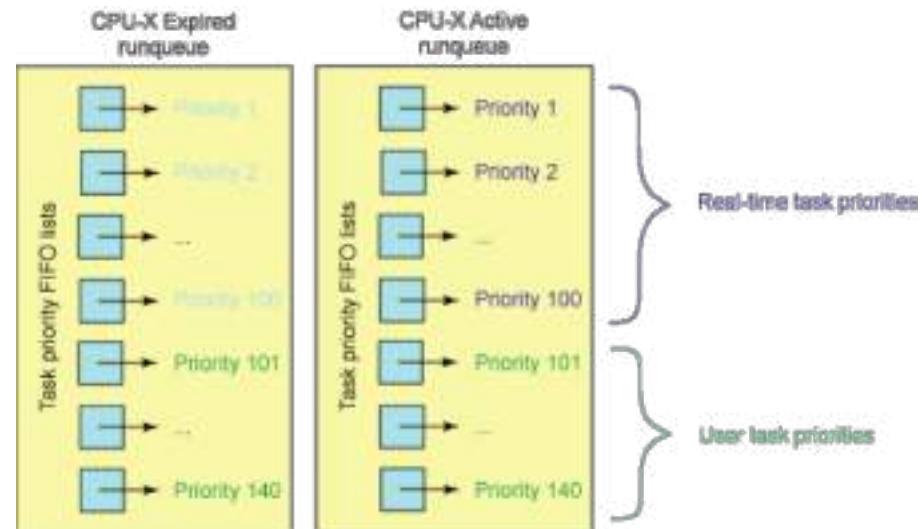
- **Linux 2.6-2.6.23 uses an O(1) scheduler**
 - Iterate over fixed # of 140 priorities to find the highest priority task
 - The amount of search time is bounded by the # priorities, not the # of tasks.
 - Hence O(1) is often called “constant time”
 - scales well because larger # tasks doesn’t affect time to find best next task to schedule



O(1) Scheduler in Linux

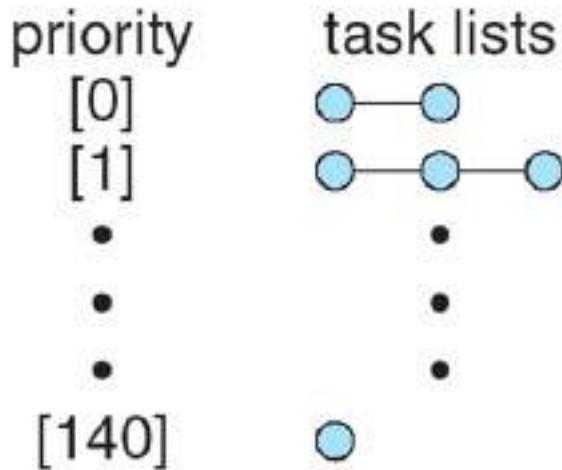
- **Linux maintains two queues:**
 - an active array or run queue and an expired array/queue, each indexed by 140 priorities
- **Active array contains all tasks with time remaining in their time slices, and expired array contains all expired tasks**

- Once a task has exhausted its time slice, it is moved to the expired queue

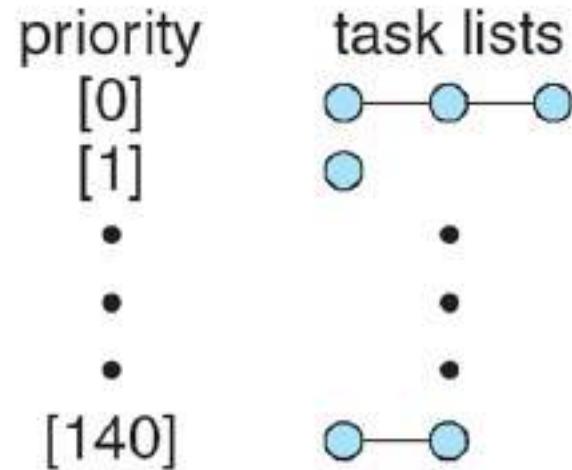


$O(1)$ Scheduler in Linux

**active
array**

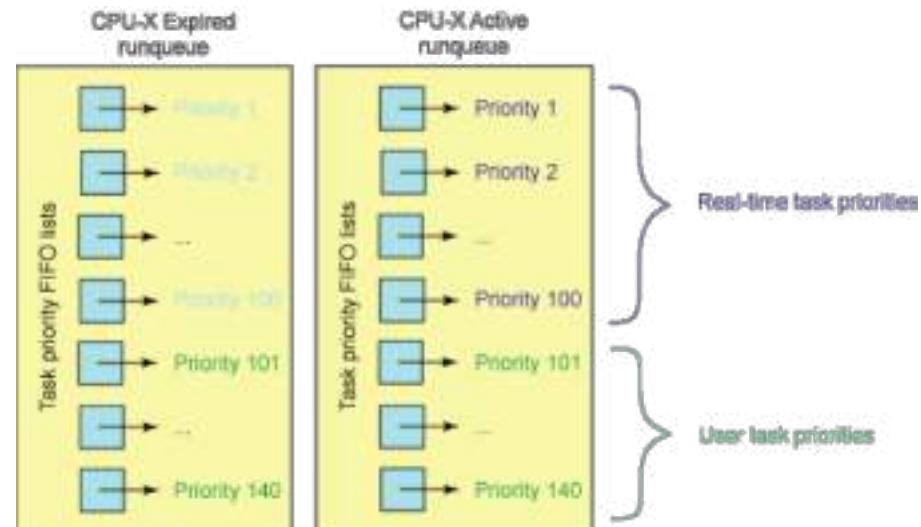


**expired
array**



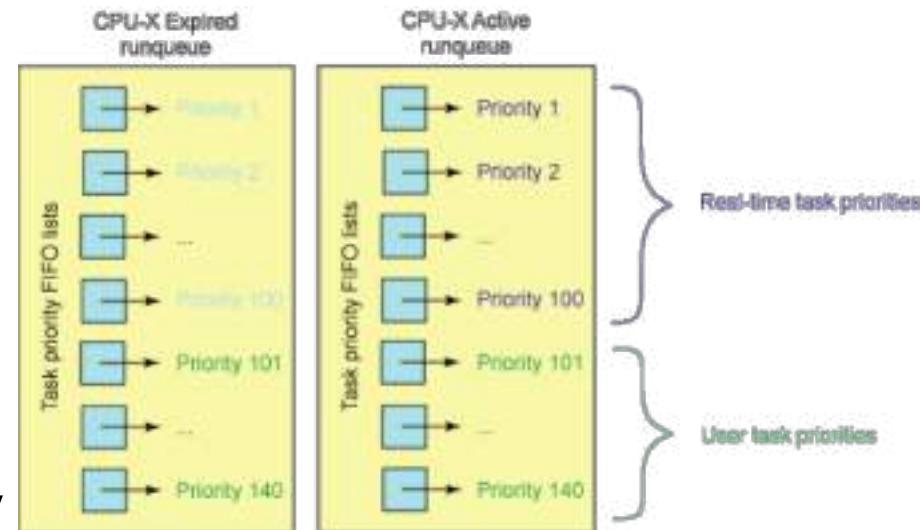
O(1) Scheduler in Linux

- An expired task is not eligible for execution again until all other tasks have exhausted their time slice
- Scheduler chooses task with highest priority from active array
 - Just search linearly through the active array from priority 1 until you find the first priority whose queue contains at least one unexpired task



O(1) Scheduler in Linux

- **# of steps to find the highest priority task is in the worst case 140**
 - This search is bounded and depends only on the # priorities, not # of tasks, unlike the O(N) scheduler
 - hence this is O(1) in complexity
- When all tasks have exhausted their time slices, the two priority arrays are exchanged
 - the expired array becomes the active array



O(1) Scheduler in Linux

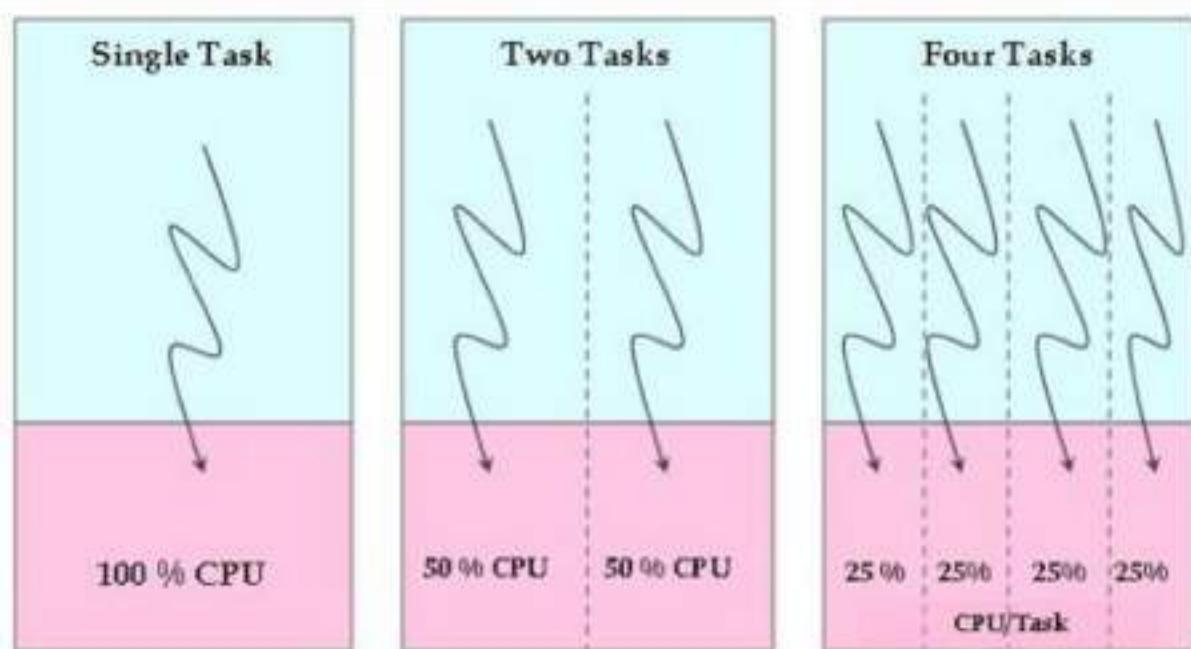
- When a task is moved from run to expired, Linux recalculates its priority according to a heuristic
 - New priority = nice value +/- f(interactivity)
 - $f()$ can change the priority by at most +/-5, and is closer to -5 if a task has been sleeping while waiting for I/O
 - interactive tasks tend to wait longer times for I/O, and thus their priority is boosted -5, and closer to +5 for compute-bound tasks
 - This dynamic reassignment of priorities affects only the lowest 40 priorities for non-RT/user tasks (corresponds to the nice range of +/- 20)
 - The heuristics became difficult to implement/maintain

Linux CFS

Completely Fair Scheduler (CFS) in Linux

- Linux 2.6.23+/3.* has a “completely fair” scheduler
- Based on concept of an “ideal” multitasking CPU

- If there are N tasks, an ideal CPU gives each task $1/N$ of CPU at every instant of time



CFS Intuition

- On an ideal CPU, N tasks would run truly in parallel, each getting $1/N$ of CPU and each executing at every instant of time
 - Example: for a 4 GHz processor, if there are 4 tasks, each gets a 1 GHz processor for each instant of time
 - Each such task makes progress at every instant of time
 - This is “fair” sharing of the CPU among each of the tasks

CFS Intuition

- In practice, we know a real (1-core) CPU cannot run N tasks truly in parallel
 - Only 1 task can run at a time
 - Time slice in/out the N tasks, so that in steady state each task gets $\sim 1/N$ of CPU
 - This gives the illusion of parallelism
 - Thus, what we have is concurrency, i.e. the N tasks run concurrently, but not truly in parallel



CFS Intuition

- Ingo Molnar (designer of CFS):
 - “CFS basically models an 'ideal, precise multitasking CPU' on real hardware.”
- So CFS’s goal is to approximate an ideally shared CPU
- Approach: when a task is given T seconds to execute, keep a running balance of the amount of time owed to other tasks as if they all ran on an ideal CPU



CFS Intuition



- Example:
 - Task T1 is given a T second time slice on the CPU
 - Suppose there are 3 other tasks T2, T3, and T4
 - On an ideal CPU, in any interval of time T, then T1, T2, T3 and T4 would each have had the equivalent of time $T/4$ on the CPU
 - Instead, on a real CPU
 - T1 is given T instead of $T/4$, so T1 has been overallocated $3T/4$
 - T2, T3 and T4 are owed time $T/4$ on the CPU, i.e. they have each been forced to *wait* the equivalent of $T/4$



CFS Intuition

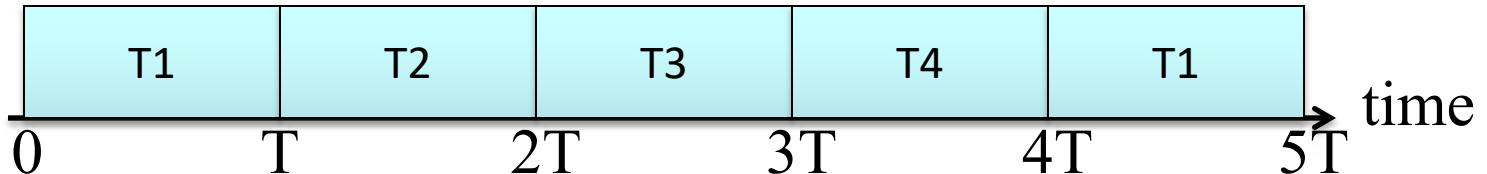


- Example:
 - The current accounting balance is summarized in the table below

	Time owed to task, i.e. wait time W_i :			
Giving time T to task:	$\underline{T_1}$	$\underline{T_2}$	$\underline{T_3}$	$\underline{T_4}$
T1	-3T/4	T/4	T/4	T/4

- In general, at any given time t in the system, each task T_i has an amount of time owed to it on an ideal CPU, i.e. the amount of time it was forced to wait, its *wait time* $W_i(t)$,

CFS Intuition



- Example: let's have round robin over 4 tasks

		Time owed to task, i.e. wait time W_i :			
Giving time T to task:		<u>T1</u>	<u>T2</u>	<u>T3</u>	<u>T4</u>
T1		-3T/4	T/4	T/4	T/4
then T2		-T/2	-T/2	T/2	T/2
then T3		-T/4	-T/4	-T/4	3T/4
then T4		0	0	0	0

- After 1 round robin, the balances owed all = 0, so every task receives its fair share of CPU over time 4T

CFS Intuition

- Suppose a 5th task T5 is added to the round robin
 - Now the amount owed/wait time is calculated as $T/5$ for each task not chosen for a time slice, and as $-4T/5$ for the chosen task for a time slice
 - In general, if there are N runnable tasks, then
 - $(N-1)T/N$ is subtracted from the balance owed/wait time of the chosen task
 - T/N is added to the balanced owed/wait time of all other ready-to-run tasks
 - T5 is initially owed no CPU time, so $W_5 = 0$
 - Example: If T5 had arrived just after T2's time slice, then T5's wait time =0 would place it above T1 and T2 but below T3 and T4 in terms of amount of time owed on the CPU



CFS Scheduler in Linux

- Goal of CFS Scheduler: *select the task with the longest wait time*
 - i.e. choose $\max_i W_i$
 - This is the task that is owed the most time on the CPU and so should be run next to achieve fairness most quickly

Wait Time Calculation

- Each scheduling decision at time k incurs a wait time $W_i(k)$, either positive or negative, to each task i
- Total accumulated wait time for each task i at time k is:

$$W_{\text{total},i}(k) = \sum_{j=1}^k W_i(j)$$

Wait Time Calculation

- Each wait time $W_i(k) =$
 - Either a penalty of T/N added to $W_{\text{total},i}$ if task i is not chosen to be scheduled, or
 - $(N-1)T/N$ is *subtracted* from the sum ($=T-T/N$) if task i is chosen to be scheduled
- So $W_i(k)$ = either T/N or $-T+N/T$
 - note how T/N is added regardless of the case!
- Hence $W_i(k) = T/N - \text{execution/run time given to task } i \text{ at time } k$, which may be zero
 - Define run time $R_i(k)$ as the execution/run time given to task i at time k , which may be zero
 - So $W_i(k) = T/N - R_i(k)$



Wait Time Calculation

- In general, each scheduling decision at time k may choose:
 - An arbitrary amount of time $T(k)$ to schedule the chosen task, i.e. it doesn't have to be a fixed time slot T
 - The number of runnable tasks $N(k)$ may change at each decision time k
- So $W_i(k) = T(k)/N(k) - R_i(k)$

Wait Time Calculation

- Total accumulated wait time for each task i at time k is:

$$\begin{aligned} W_{\text{total}_i}(k) &= \sum_{j=1}^k W_i(j) = \sum_{j=1}^k [T(j)/N(j) + R_i(j)] \\ &= \underbrace{\sum_{j=1}^k T(j)/N(j)}_{\text{Global fair clock measuring how system time advances in an ideal CPU with } N \text{ varying tasks, also called } rq->\text{fair_clock in CFS' 1st implementation}} + \underbrace{\sum_{j=1}^k R_i(j)}_{\text{Total run time given task } i.} \end{aligned}$$

Global fair clock measuring how system time advances in an ideal CPU with N varying tasks, also called `rq->fair_clock` in CFS' 1st implementation

Total run time given task i .
Let's define it as $R_{\text{total}_i}(k)$

CFS Scheduler in Linux

- Recall: CFS scheduler chooses task with max $W_{total,i}(k)$ at each scheduling decision k
- Maximizing $W_{total,i}(k)$ equivalent to minimizing the quantity [Global fair clock - $W_{total,i}(k)$]
- 1st CFS scheduler:
 - Had to track global fair clock and $W_{total,i}(k)$ for each task i
 - Then would compute the values [Global fair clock - $W_{total,i}(k)$]
 - Then ordered these values in a Red-Black tree
 - Then selected leftmost node in tree (has minimum value) and scheduled the task corresponding to this node



CFS Scheduler in Linux

- Revised CFS scheduler:
 - We note that [Global fair clock - $W_{total_i}(k)$] = run time $R_{total_i}(k)$!
 - Minimizing over the quantities [Global fair clock - $W_{total_i}(k)$] is equivalent to minimizing over the accumulated run times $R_{total_i}(k)$
 - 1st CFS scheduler had to track complex values like the global fair clock, and accumulated wait times
 - These both needed the # runnable tasks $N(k)$ at each scheduling decision time k , which keeps changing
 - New approach just sums run times given each task
 - this simple approach still achieves fairness according to our derivation

Virtual Run Time

- Revised CFS scheduler simply sums the run times given each task and chooses the one to schedule with the minimum sum
 - This is equivalent to choosing the task owed the most time on an ideal fair CPU according to our derivation, and thus achieves fairness
 - Caveat: when a new task is added to the run queue, it may have been blocked a long time, so its run time may be very low compared to other tasks in the run queue
 - Such a task would consume a long time before its accumulated run time rises to a level close to the other executing tasks' total run times, which would effectively block other tasks from running in a timely manner



Virtual Run Time

- Revised CFS scheduler accommodates new tasks as follows:
 - Define a virtual run time $vruntime$
 - As before, each normally running task i simply adds its given run times to its own accumulated sum $vruntime_i$,
 - When a new task is added to the run queue (or an existing task becomes unblocked from I/O), assign it a new virtual run time = minimum of current $vruntimes$ in the run queue
 - This quantity is defined as $\min_vruntime$
 - This approach re-normalizes the newly active task's run time to about the level of the virtual run times of the currently runnable tasks



Virtual Run Time

- Since each newly active task's is given a re-normalized run time, then the run time calculated is not the actual execution time given a task
 - Hence we need to define a new term $vruntime_i(k)$, rather than use the absolute accumulated run time $R_{total,i}(k)$
- *Intuitively, CFS choosing the task with the minimum virtual run time prioritizes the task that been given the least time on the CPU*
 - *This is the task that should get service first to ensure fairness*



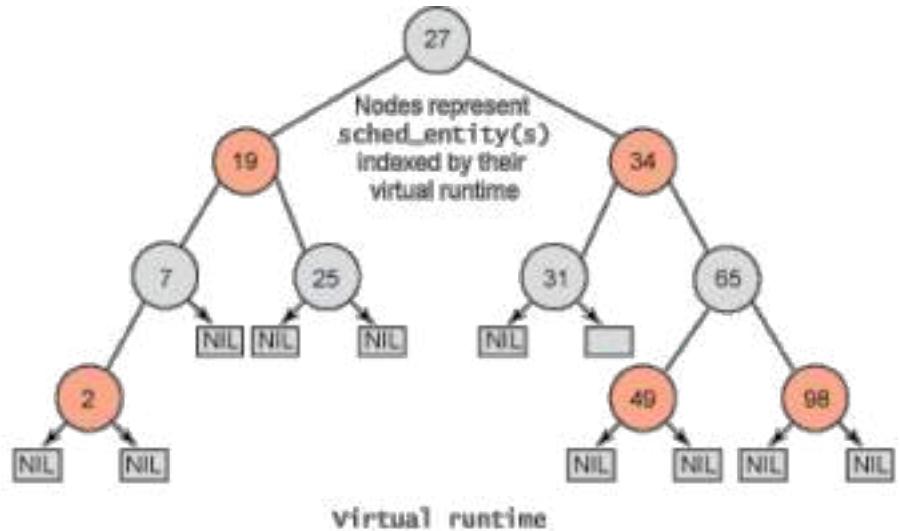
CFS Scheduler in Linux

- *So revised CFS scheduler chooses the task with the minimum $vruntime_i(k)$ at each scheduling decision time k*
- This approach is responsive to interactive tasks!
 - They get instant service after they unblock from their I/O
 - This is because they are given a re-normalized $vruntime_i(k) = \min_vruntimes$,
 - Since CFS chooses the next task to schedule as the one with the minimum $vruntime$, then the interactive task will be chosen first and get service immediately



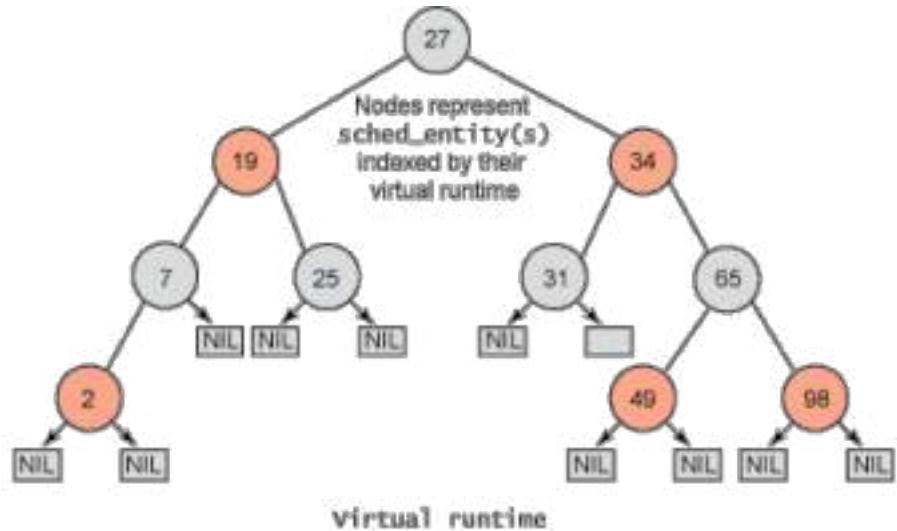
CFS' Red Black Tree

- To quickly find the task with the minimum vruntime, order the vruntimes in a Red-Black tree
 - This is a balanced tree, ordered from left (minimum vruntime) to right (maximum vruntime)
- Finding the minimum is fast, simple and constant time!
 - Choose leftmost task in tree with lowest virtual run time to schedule next



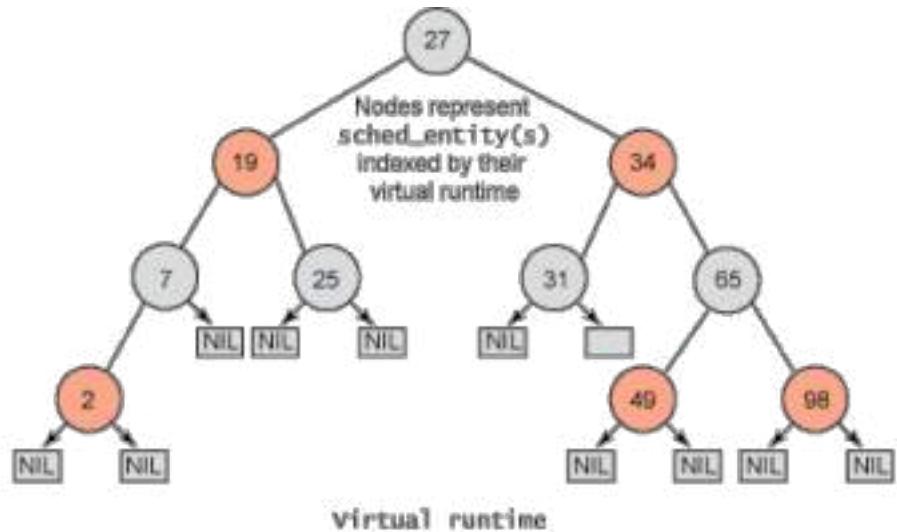
CFS' Red Black Tree

- As tasks run more, their virtual run time increases
 - so they migrate to other positions further to the right in the tree
 - Must re-insert nodes to tree, and rearrange tree, but the RB tree is self-balancing
- Inserting nodes is an $O(\log N)$ operation due to RB tree
 - This is viewed as acceptable overhead



CFS' Red Black Tree

- Tasks that haven't had CPU execution in a while will migrate left and eventually get service
 - Intuitively, this eventual migration leftwards makes CFS fair
- Newly active tasks, e.g. interactive ones, will be added to the left of the tree and get service quickly



CFS Scheduler and Priorities

- All non-Real Time tasks of differing priorities are combined into one RB tree
- don't need 40 separate run queues, one for each priority - elegant!
- Higher priority tasks get larger run time slices
- lower niceness => higher the priority => more run time is given on the CPU

CFS Scheduler and Priorities

- Higher priority tasks are scheduled more often
 - $\text{virtual runtime} += (\text{actual CPU runtime}) * \text{NICE}_0 / \text{task's weight}$
 - Higher priority
 - ⇒ higher weight
 - ⇒ less increment of $vruntime$
 - ⇒ task is further left on the RB tree and is scheduled sooner

CFS Scheduler and Priorities

- While CFS is fair to tasks, it is not necessarily fair to applications
 - Suppose application A1 has 100 threads T1-T100
 - Suppose application A2 is interactive and has one thread T101
 - CFS would give
 - A1 100/101 of CPU
 - A2 only 1/101 of the CPU
- Instead, Linux CFS supports fairness across groups:
 - A1 is in group 1 and A2 is in group 2
 - Groups 1 and 2 each get 50% of CPU – fair!
 - Within Group 1, 100 threads share 50% of CPU
- Multi-threaded apps don't overwhelm single thread apps



Realtime and Multi-core Scheduling

Real Time Scheduling in Linux

- Linux also includes three **real-time** scheduling classes:
 - Real time FIFO – soft real time (SCHED_FIFO)
 - Real time Round Robin – soft real time (SCHED_RR)
 - Real time Earliest Deadline First – hard real time as of Linux 3.14 (SCHED_DEADLINE)
- Only processes with the priorities 0-99 have access to these RT schedulers



Real Time Scheduling in Linux

- A real time FIFO task continues to run until it voluntarily yields the processor, blocks or is preempted by a higher-priority real-time task
 - no timeslices
 - all other tasks of lower priority will not be scheduled until it relinquishes the CPU
 - two equal-priority Real time FIFO tasks do not preempt each other

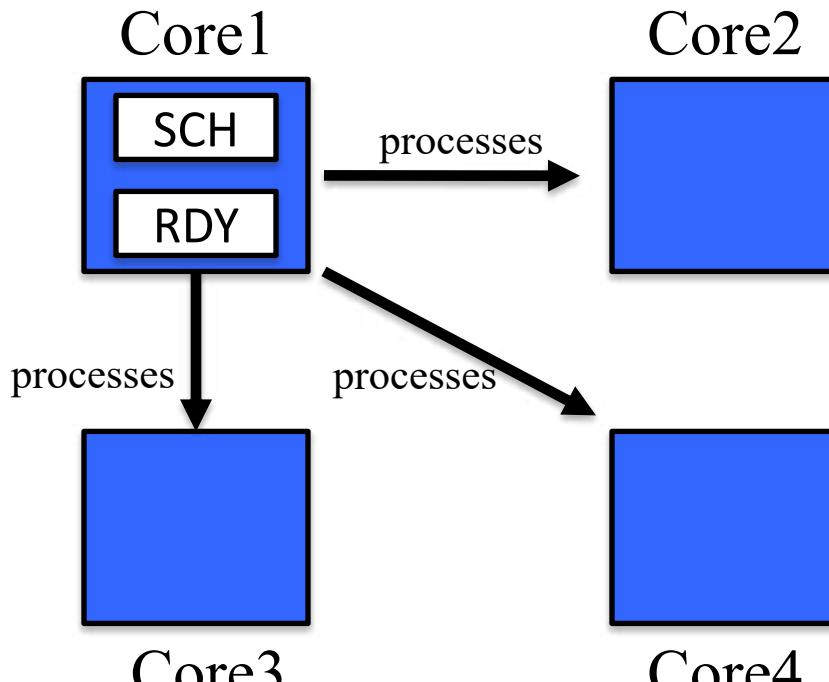


Real Time Scheduling in Linux

- SCHED_RR is similar to SCHED_FIFO, except that such tasks are allotted timeslices based on their priority and run until they exhaust their timeslice
- Non-real time tasks continue to use CFS algorithm
- SCHED_DEADLINE uses an Earliest Deadline First algorithm to schedule each task.

Multi-core Scheduling

- Scheduling over multiple processors or cores is a new challenge.
 - A single CPU/processor may support multiple cores



SCH = Scheduler, RDY = Ready Queue

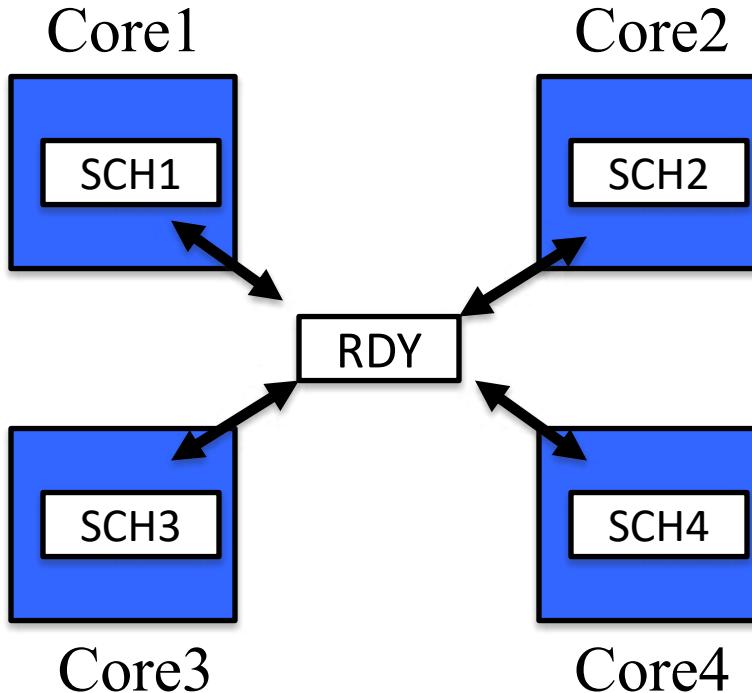
- Variety of multi-core schedulers being tried. We'll just mention some design themes.
- In *asymmetric multiprocessing* - one CPU handles all scheduling, decides which processes run on which cores

Multi-core Scheduling

- In symmetric multi-processing (SMP), each core is self-scheduling.
 - All modern OSs support some form of SMP

Two types:

1. All cores share a single global ready queue.

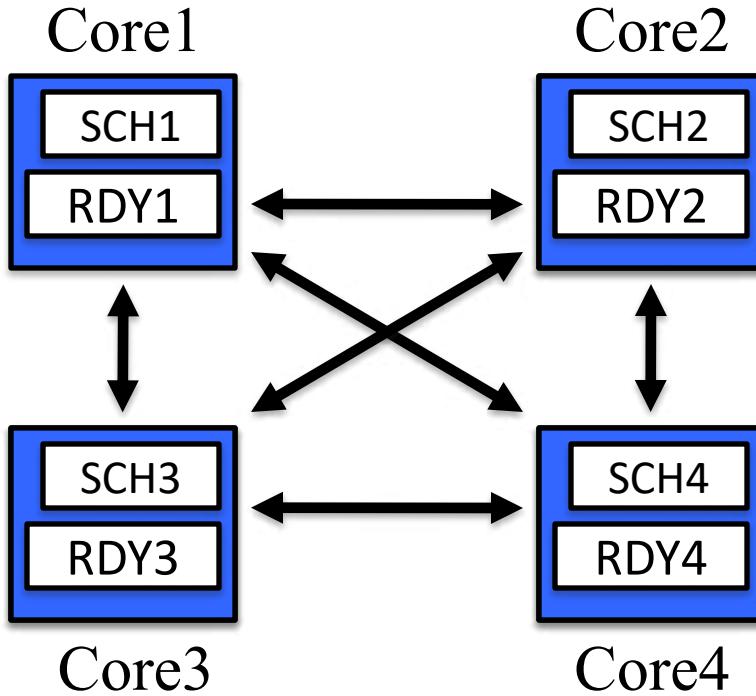


- Here, each core has its own scheduler. When idle, each scheduler requests another process from the shared ready queue. Puts it back when time slice done.
 - Synchronization needed to write/read shared ready queue



Multi-core Scheduling

2. Another self-scheduling SMP approach is when each core has its own ready queue
 - Most modern OSs support this paradigm



- a typical OS scheduler plus a ready queue designed for a single CPU can run on each core...
- Except that processes now can migrate to other cores/processors
 - There has to be some additional coordination of migration



Multi-core Scheduling - Cache

- Caching is important to consider
 - Each CPU has its own cache to improve performance
 - If a process migrates too much between CPUs
 - rebuild L1 and L2 caches each time a process starts on a new core/processor
 - L3 caches that span multiple cores can help alleviate this, but there is a performance hit, because L3 is slower than L1 and L2.
 - In any case, L1 and L2 caches still have to be rebuilt.



Multi-core Scheduling - Affinity

- To maximally exploit caching, processes tend to stick to a given core/processor = processor affinity
 - In hard affinity, a process specifies via a system call that it insists on staying on a given CPU core
 - In soft affinity, there is still a bias to stick to a CPU core, but processes can on occasion migrate.
 - Linux supports both

Multi-core Scheduling

Load balancing

- Goal: Keep workload evenly distributed across cores
- Otherwise, some cores will be under-utilized.
- When there is a single shared ready queue, there is automatic load balancing
- cores just pull in processes from the ready queue whenever they're idle.
- push migration – a dedicated task periodically checks the load on each core, and if imbalance, pushes processes from more-loaded to less-loaded cores



Multi-core Scheduling Load Balancing

- Load balancing can conflict with caching
 - Push/pull migration causes caches to be rebuilt
- Load balancing can conflict with power management
 - Mobile devices typically want to save power
 - One approach is to power down unused cores
 - Load balancing would keep as many cores active as possible, thereby consuming power
- In systems, often conflicting design goals

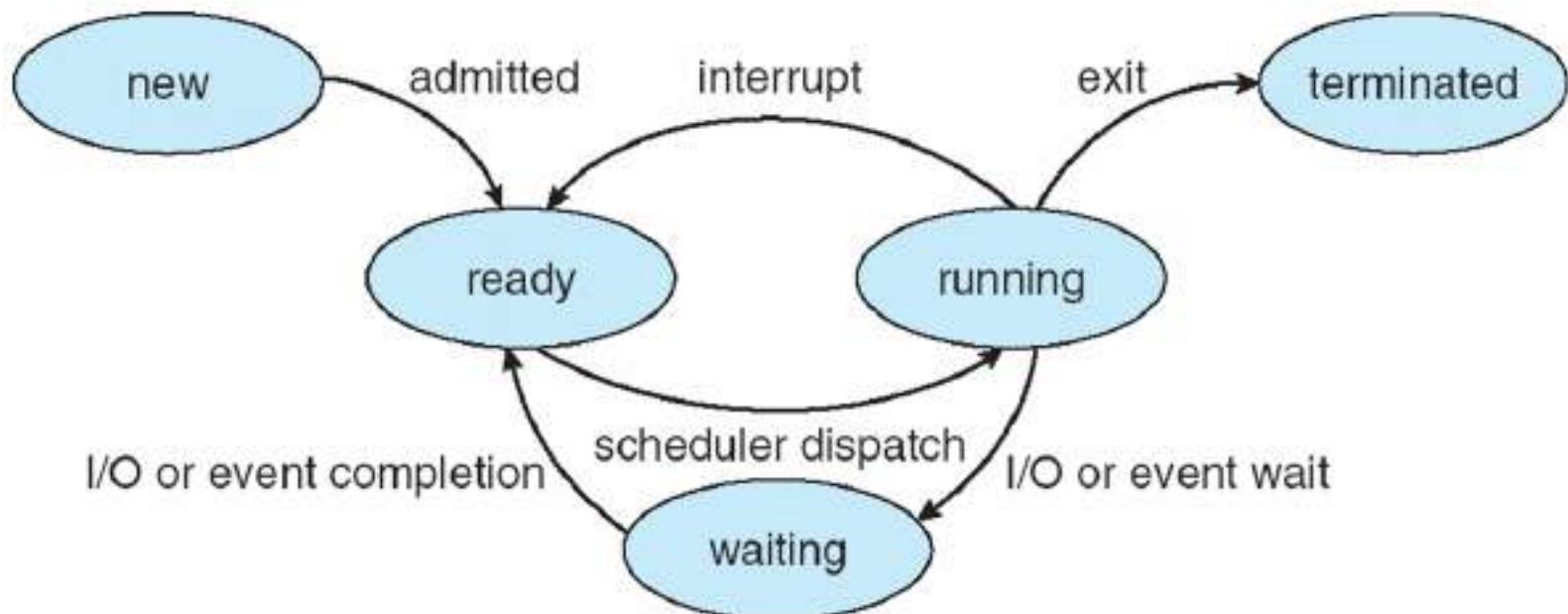




Lecture 14

Process Scheduling

Diagram of Process State



Also called “blocked” state



Switching Between Processes

- A process can be switched out due to:
 - blocking on I/O
 - voluntarily yielding the CPU, e.g. via other system calls
 - being preemptively time sliced, i.e interrupted
 - Termination



Switching Between Processes

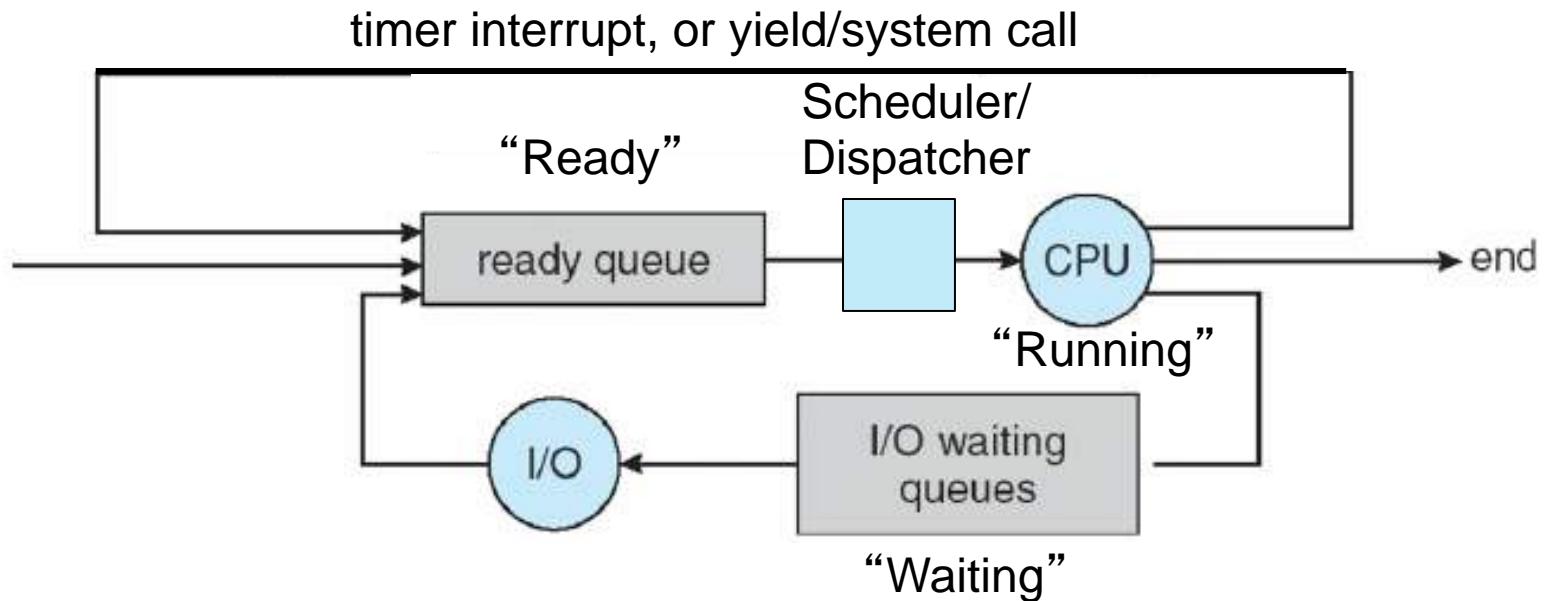
- The dispatcher gives control of CPU to the process selected by the scheduler, causing context switch:
 - save old state
 - select next process
 - load new state
 - switch to user mode, jumping to the proper location in the user process to restart that process
- Separate
 - *mechanism* of scheduling
 - from the *policy* of scheduling



Context Switch Overhead

- Typically take 10 microseconds to copy register state to/from memory
 - on a 1 GHz CPU, that's 10000 wasted cycles per context switch!
- if the time slice is on the order of a context switch, then CPU spends most of its time context switching
 - Typically choose time slice to be large enough so that only 10% of CPU time is spent context switching
 - Most modern systems choose time slices of 100 us

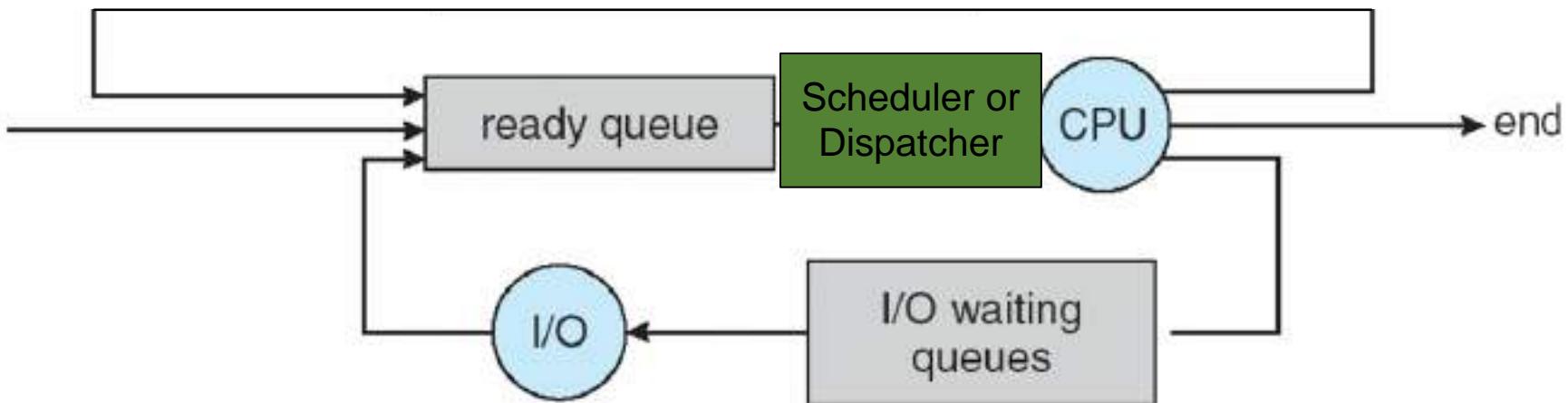
Process Scheduling



If threads are implemented as kernel threads, then OS can schedule threads as well as processes

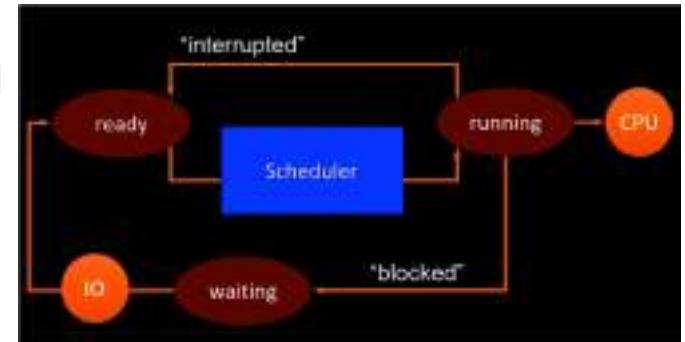
Scheduling Policy

- Scheduler's job is to decide the next process (or kernel thread) to run
 - From among the set of processes/kernel threads in the ready queue



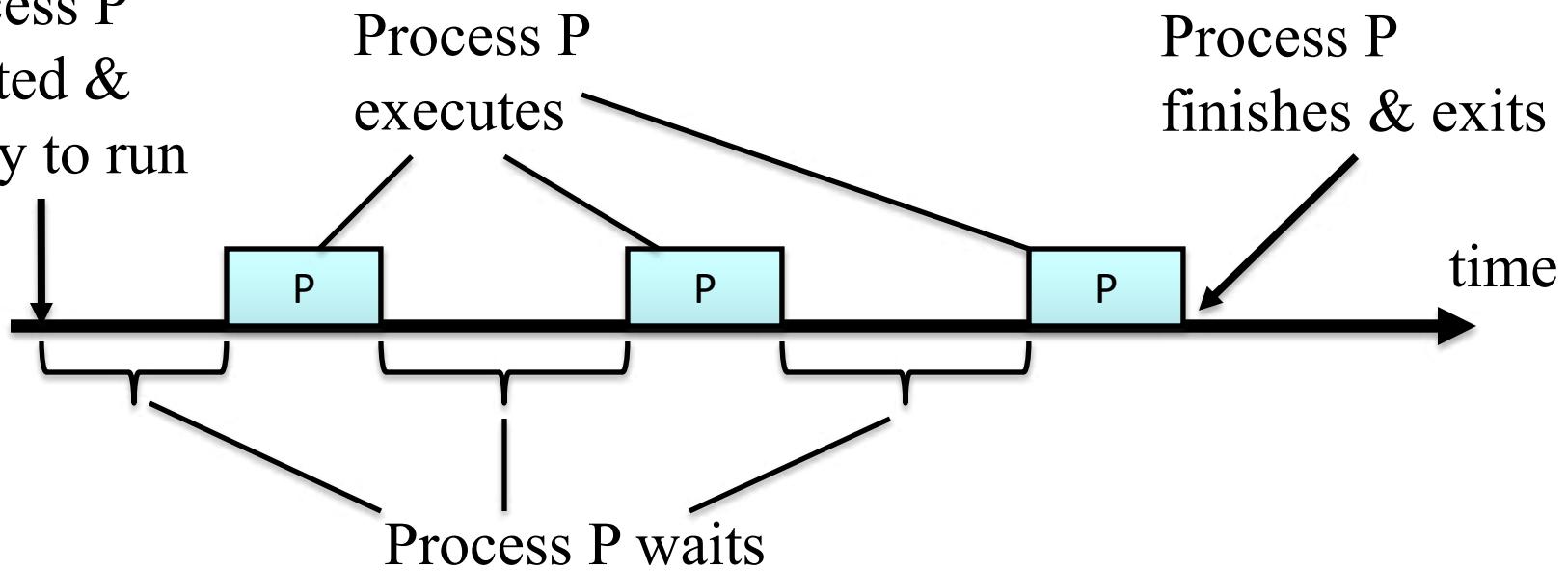
Scheduling Policy

- Scheduler implements a scheduling policy based on some of the following **goals**:
 - maximize CPU utilization: 40 % to 90 %
 - maximize throughput: # processes completed/second
 - maximize fairness
 - Meet deadlines or delay guarantees
 - Ensure adherence to priorities
 - Minimize average or peak turnaround time: from 1st entry to termination
 - Min avg or peak waiting time: sum of time in ready queue
 - Min avg or peak response time: time until first response.



Scheduling Definitions

Process P
created &
ready to run



- *execution time* $E(P_i)$ = the time on the CPU required to fully execute process i
 - Sum up the time slices given to process i
 - Also called the “burst time” by textbook

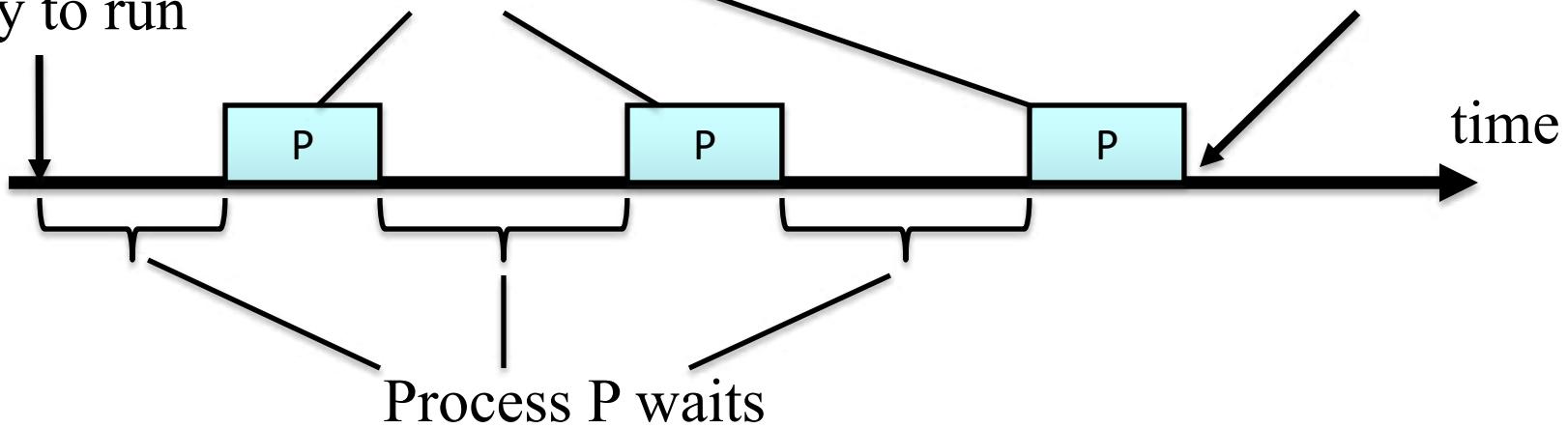


Scheduling Definitions

Process P
created &
ready to run

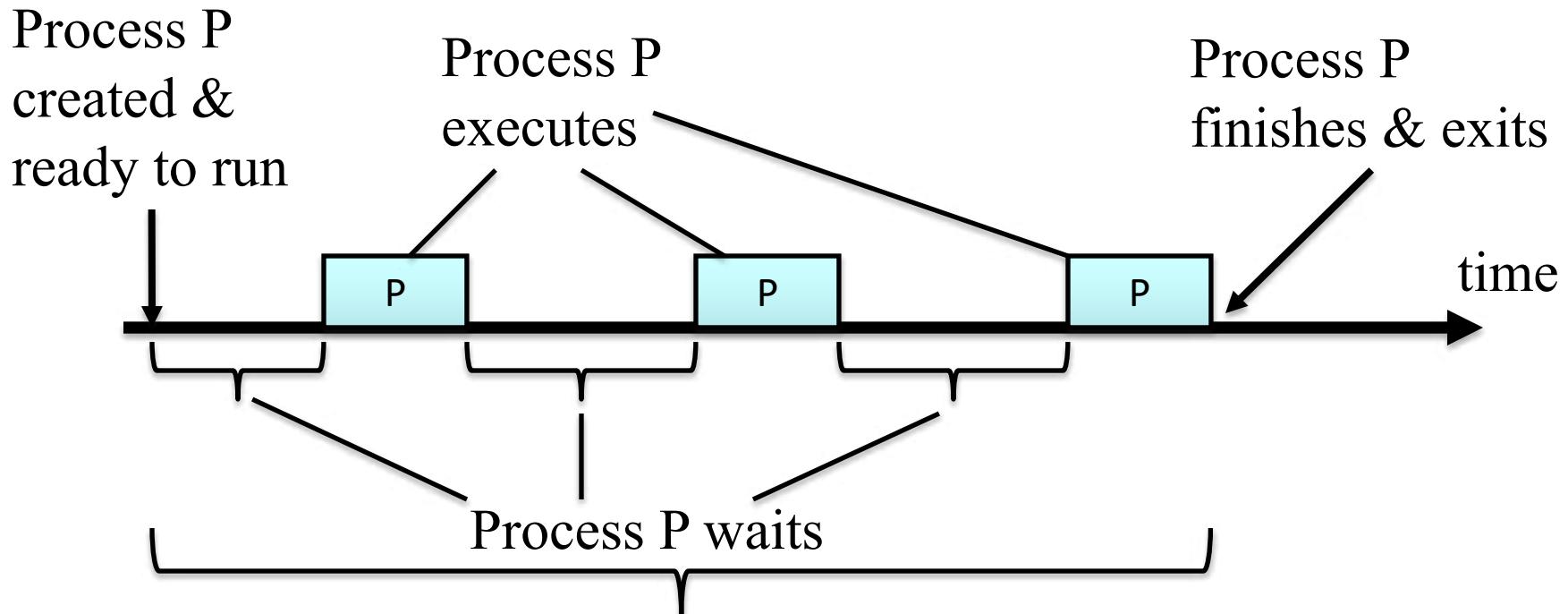
Process P
executes

Process P
finishes & exits



- ***wait time*** $W(P_i)$ = the time process i is in the ready state/queue waiting but not running
 - Sum up the gaps between time slices given to process i
 - But, does NOT include I/O waiting time

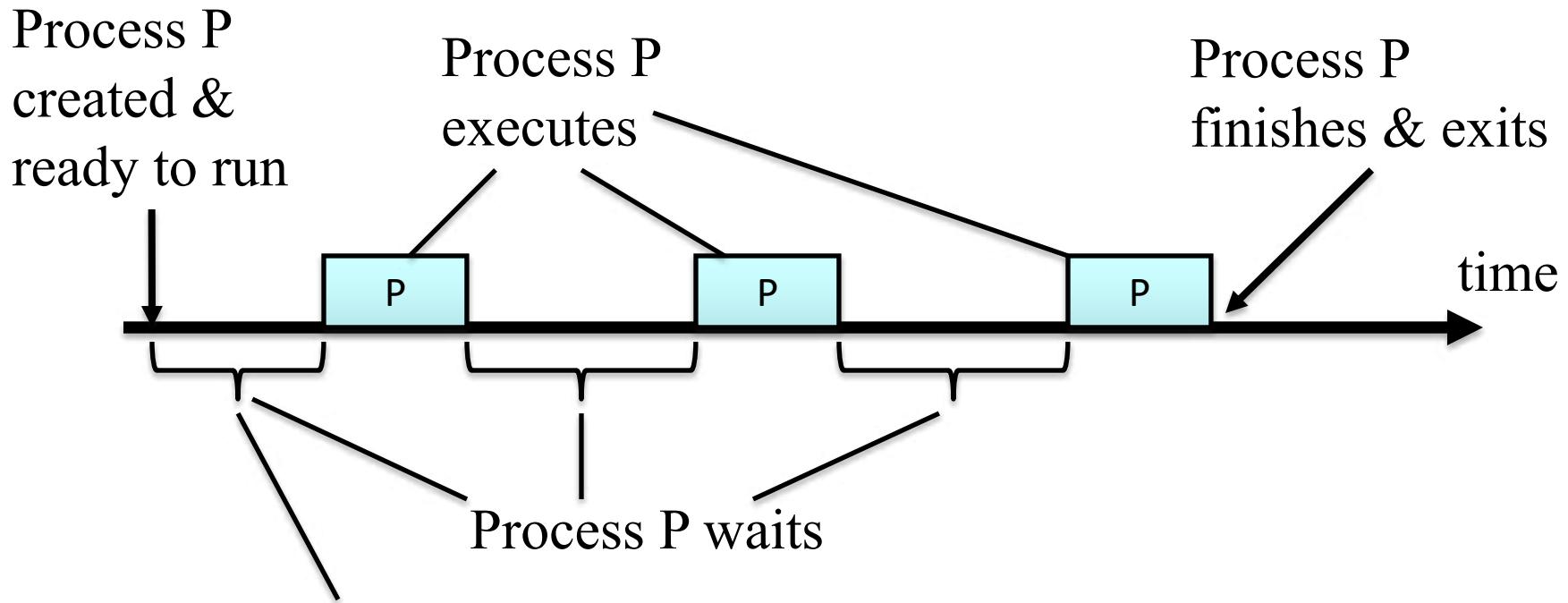
Scheduling Definitions



- ***turnaround time*** $T(P_i)$ the time from 1st entry of process i into the ready queue to its final exit from the system (exits last run state)
 - Does include time waiting and time for IO to complete



Scheduling Definitions



- **response time** $R(P_i)$ = the time from 1st entry of process i into the ready queue to its 1st scheduling on the CPU (1st occurrence in running state)
 - Useful for interactive tasks

Scheduling Analysis

- We analyze various scheduling policies to see how efficiently they perform with respect to metrics
 - wait time, turnaround time, response time, etc.
- Some algorithms will be optimal in certain metrics
- To simplify our analysis, assume:
 - No blocking I/O. Focus only on scheduling processes/tasks that have provided their execution times
 - Processes execute until completion, unless otherwise noted



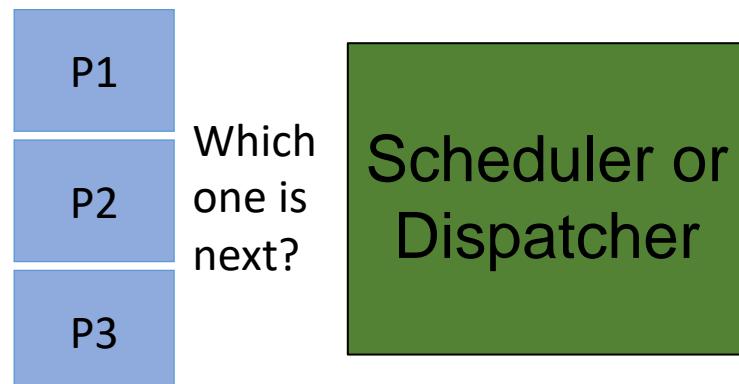


Scheduling Policies

Scheduling Policy

- How does Scheduler pick the next process to be run?
 - depends on which policy is implemented
- What is the simplest policy you can think of for picking from a group of processes?

- How about something complicated?



First Come First Serve (FCFS) Scheduling

- order of arrival dictates order of scheduling
 - Non-preemptive, processes execute until completion
- If processes arrived in order P1, P2, P3 before time 0, then CPU service time is:

Process	CPU Execution Time
P1	24
P2	3
P3	3



FCFS Scheduling

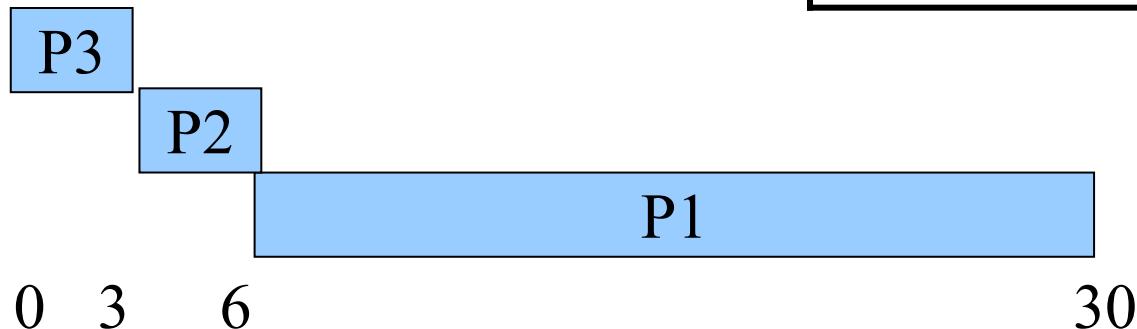
- If processes arrive in reverse order P3, P2, P1 before time 0, then CPU service time is:

Process	CPU Execution Time
P1	24
P2	3
P3	3



Gantt Chart

- We used a different format to describe the process times of multi-tasking and multi-programming
- These formats are equivalent



Process	CPU Execution Time
P1	24
P2	3
P3	3

FCFS Scheduling

Case I



Case II



Lets calculate the ***average wait time*** for each of the cases

All processes arrived just before time 0

P1 does not wait at all

P2 waits for P1 to complete before being scheduled

P3 waits until P2 has completed

$$(0 + 24 + 27)/3 = 17$$

FCFS Scheduling

Case I



Case II



- Case I: **average wait time** is $(0+24+27)/3 = 17$ seconds
- Case II: **average wait time** is $(0+3+6)/3 = 3$ seconds
- FCFS wait times are generally not minimal - vary a lot if order of arrival changed, which is especially true if the process service times vary a lot (are spread out)

FCFS Scheduling

Case I



Case II

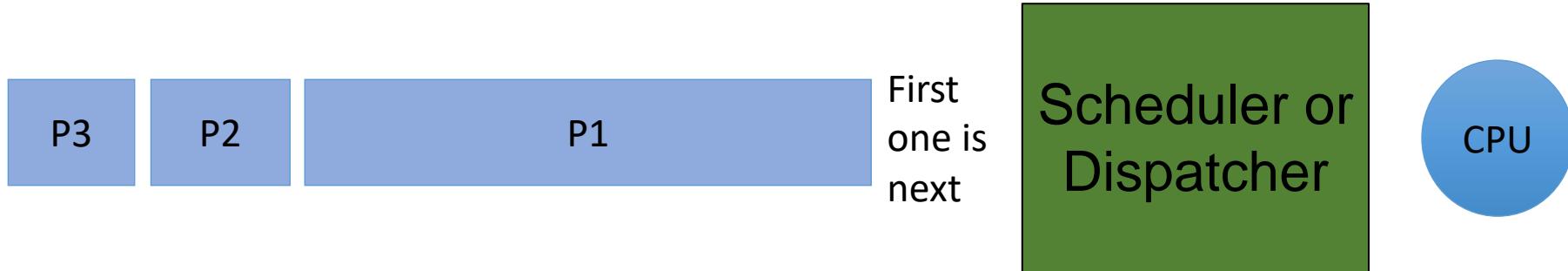


Lets calculate the ***average turnaround time*** for each of the cases

- Case I: average turnaround time is $(24+27+30)/3 = 27$ seconds
- Case II: average turnaround time is $(3+6+30)/3 = 13$ seconds
- A lot of variation in turnaround time too, depending on the task's arrival.

FCFS Scheduling

- Just pick the next process in the queue to be run?
 - No other information about the process is required
- Does it meet our goals?
 - maximize CPU utilization: 40% to 90%?
 - maximize throughput: # processes completed/second?
 - minimize average or peak wait, turnaround, or response time?
 - meet deadlines or delay guarantees?
 - maximize fairness?



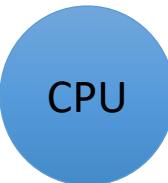
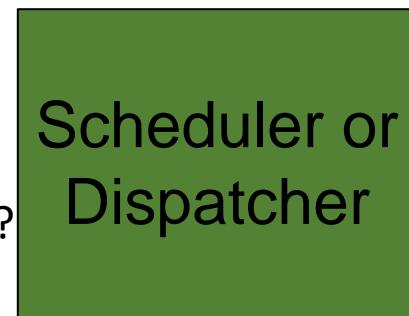
Shortest Job First (SJF) Scheduling

Choose the process/thread with
the lowest execution time

- gives priority to shortest or briefest processes
- minimizes the average wait time
 - intuition: one long process will increase wait time for all short processes that follow
- the impact of the wait time on other long processes moved towards the end is minimal



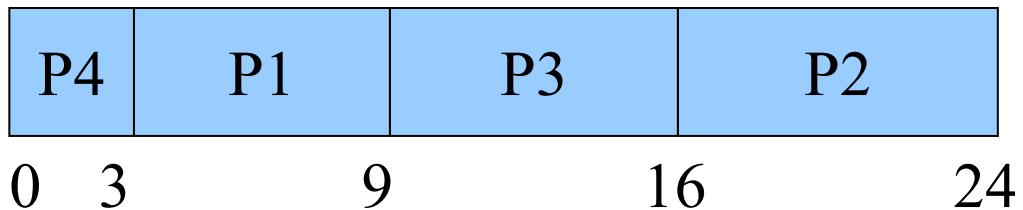
Which
one is
next???



Shortest Job First Scheduling

- In this example, P1 through P4 are in ready queue at time 0:
 - can prove SJF minimizes wait time*
 - out of 24 possibilities of ordering P1 through P4, the SJF ordering has the lowest average wait time

Process	CPU Execution Time
P1	6
P2	8
P3	7
P4	3



average wait time
 $= (0+3+9+16)/4$
 $= 7$ seconds



Shortest Job First (SJF) Scheduling

- *It has been proved that SJF minimizes the average wait time out of all possible scheduling policies.*
- *Sketch of proof:*
 - Given a set of processes $\{P_a, P_b, \dots, P_n\}$, suppose one chooses a process P from this set to schedule first
 - The wait times for all the remaining processes in $\{P_a, \dots, P_n\} - P$ will be increased by the run time of P
 - If P has the shortest run time (SJF), then the wait times will increase the least amount possible



Shortest Job First (SJF) Scheduling

- ***Sketch of proof (continued):***
 - Apply this reasoning iteratively to each remaining subset of processes.
 - At each step, the wait time of the remaining processes is increased least by scheduling the process with the smallest run time.
 - The average wait time is minimized by minimizing each process' wait time,
 - Each process' wait time is the sum of all earlier run times, which is minimal if the shortest job is chosen at each step above.

Shortest Job First Scheduling

- **Problem?**
 - must know run times $E(p_i)$ in advance unlike FCFS
- **Solution: estimate CPU demand in the next time interval from the process/thread's CPU usage in prior time intervals**
 - Divide time into monitoring intervals, and in each interval n , measure the CPU time each process P_i takes as $CPU(n,i)$.
 - For each process P_i , estimate the amount of CPU time $EstCPU(n,i)$ for the next interval as the average of the current measurement and the previous estimate



Shortest Job First Scheduling

- **Solution (continued):**

$$\text{EstCPU}(n+1,i) = \alpha * \text{CPU}(n,i) + (1-\alpha) * \text{EstCPU}(n,i)$$

where $0 < \alpha < 1$

- If $\alpha > 1/2$, then estimate is influenced more by recent history.
- If $\alpha < 1/2$, then bias the estimate more towards older history
- This kind of average is called an **exponentially weighted average**
- See textbook for more



Shortest Job First Scheduling

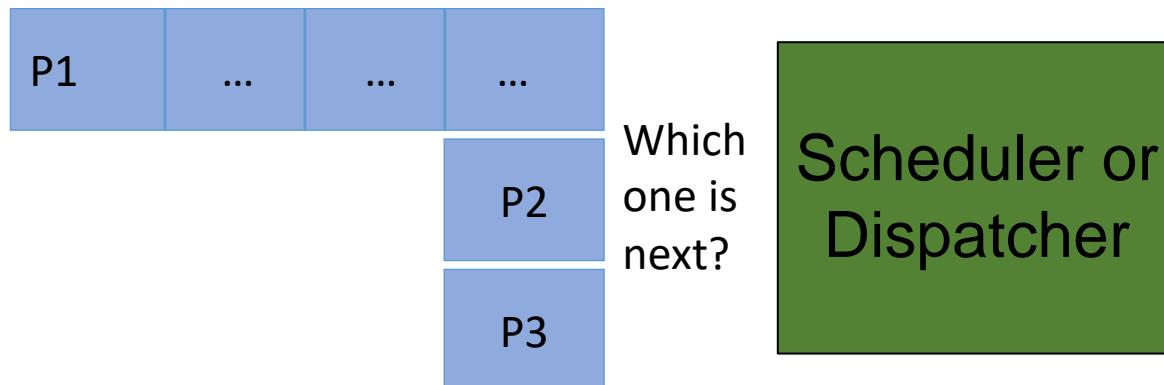
- **SJF can be preemptive:**
 - i.e. when a new job arrives in the ready queue, if its execution time is less than the currently executing job's remaining execution time, then it can preempt the current job
 - Compare to FCFS: a new process can't preempt earlier processes, because its order is later than the earlier processes
 - For simplicity, we assumed in the preceding analysis that jobs ran to completion and no new jobs arrived until the current set had finished

Scheduling Criteria

Is it FAIR that long processes use the CPU as much as they need?

Is it FAIR to make long processes wait for all shorter processes?

How can we be more fair with the CPU resource?



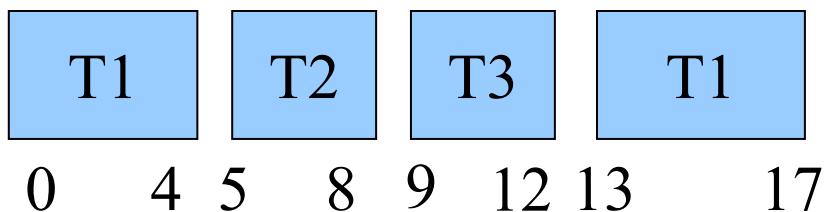
Round Robin Scheduling

- **Use preemptive time slicing**
 - a task is forced to relinquish the CPU before it's necessarily done
- **Rotate among the tasks in the ready queue.**
 - periodic timer interrupt transfers control to the CPU scheduler, which *rotates* among the processes in the ready queue, giving each a time slice
 - e.g. if there are 3 tasks T1, T2, & T3, then the scheduler will keep rotating among the three: T1, T2, T3, T1, T2, T3, T1, ...
 - treats the ready queue as a circular queue (or removes the scheduled item from the front and places it at the back)

Round Robin Scheduling

- Example: let time slice = 4 ms
- Now T1 is time sliced out, and T2 and T3 are allowed to run sooner than FCFS
- ***average response time?***
 - assuming a 1ms switching time

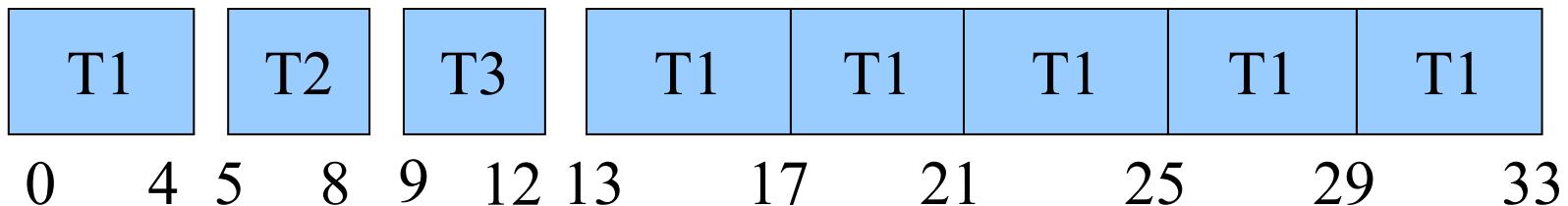
Task	CPU Execution Time (ms)
T1	24
T2	3
T3	3



Round Robin Scheduling

- Example: let time slice = 4 ms
- Now T1 is time sliced out, and T2 and T3 are allowed to run sooner than FCFS
- ***average response time is fast at 4.66ms***
 - assuming a 1ms switching time
 - Compare to FCFS w/ long 1st task

Task	CPU Execution Time (ms)
T1	24
T2	3
T3	3



Round Robin Scheduling

- Useful to support interactive applications in multitasking systems
 - hence is a popular scheduling algorithm
- Properties:
 - Simple to implement: just rotate, and don't need to know execution times a priori
 - Fair: If there are n tasks, each task gets $1/n$ of CPU
- A task can finish before its time slice is up
 - Scheduler just selects the next task in the queue



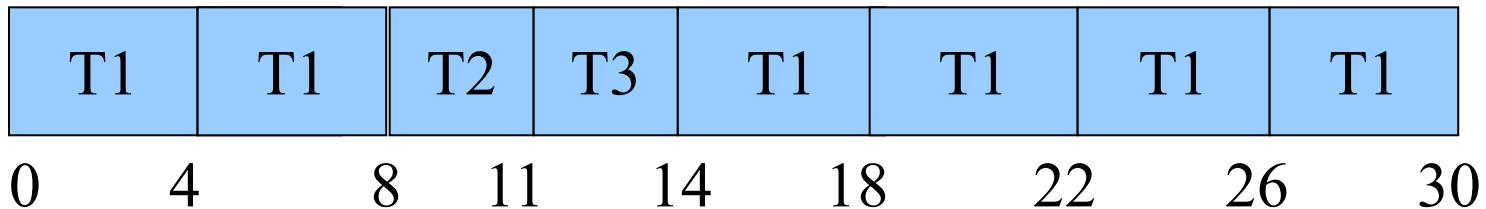
Weighted Round Robin

- **Give some tasks more time slices than others**
 - This is a way of implementing priorities – higher priority tasks get more time slices per round
 - If task T_i gets N_i slots per round, then the fraction α_i of the CPU bandwidth that task i gets is:

$$\alpha_i = \frac{N_i}{\sum_i N_i}$$

Weighted Round Robin

- In previous example – assuming switching time = 0ms:
 - could give T1 two time slices
 - T2 and T3 only 1 each round



Deadline Scheduling

- Hard real time systems require that certain tasks *must* finish executing by a certain time, or the system fails

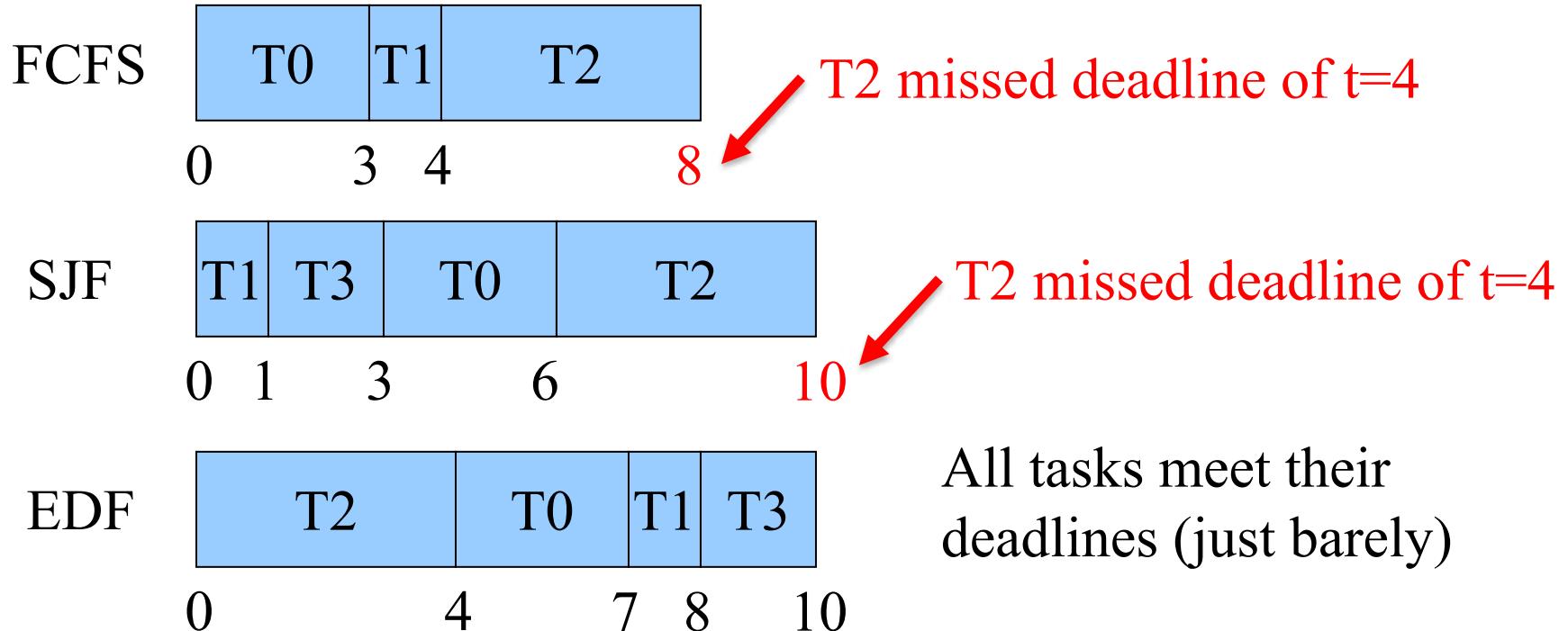
– e.g. robots and self-driving cars need a real time OS (RTOS) whose tasks (actuating an arm/leg or steering wheel) must be scheduled by a certain deadline

Task	CPU Execution Time	Deadline from now
T0	3	7
T1	1	9
T2	4	4
T3	2	10

Earliest Deadline First (EDF) Scheduling

- Choose the task with the earliest deadline
 - Pick task that most urgently needs to be completed

Task	CPU Time	Deadline from now
T0	3	7
T1	1	9
T2	4	4
T3	2	10



Deadline Scheduling

- Even EDF may not be able to meet all deadlines:

- In previous example, if T3's deadline was t=9, then EDF cannot meet T3's deadline

Task	CPU Time	Deadline from now
T0	3	7
T1	1	9
T2	4	4
T3	2	10

- When EDF fails, the results of further failures, i.e. missed deadlines, are unpredictable

- Which tasks miss their deadlines depends on when the failure occurred and the system state at that time
 - Could be a cascade of failures
- This is one disadvantage of EDF

Deadline Scheduling

- Admission control policy
 - Check on entry to system whether a task's deadline can be met,
 - Examine the current set of tasks already in the ready queue and their deadlines
 - If all deadlines can be met with the new task, then admit it.
 - The *schedulability* of the set of real-time tasks has been verified
 - Else when deadlines cannot be met with new task,
deny admission to this task if its deadline can't be met
 - Note FCFS, SJF and priority/weighted RRobin had no notion of refusing admission



EDF and Preemption

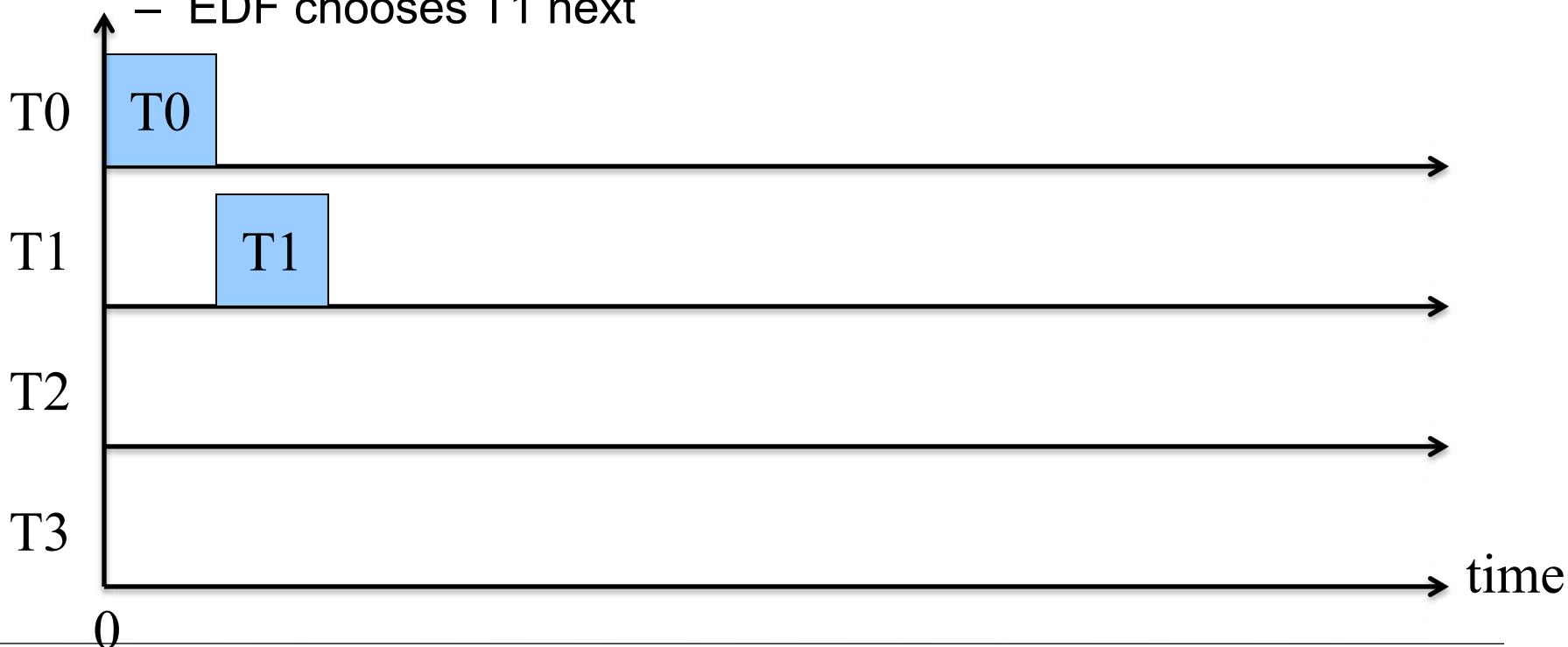
- Assume a preemptively time sliced system
 - A task arriving with an earlier deadline can preempt one currently executing with a later deadline.

Task	CPU Execution Time	Absolute Deadline	Arrival time
T0	1	2	0
T1	2	5	0
T2	2	4	2
T3	2	10	3

Assume in this example time slice = 1, i.e. the executing task is interrupted every second and a new scheduling decision is made

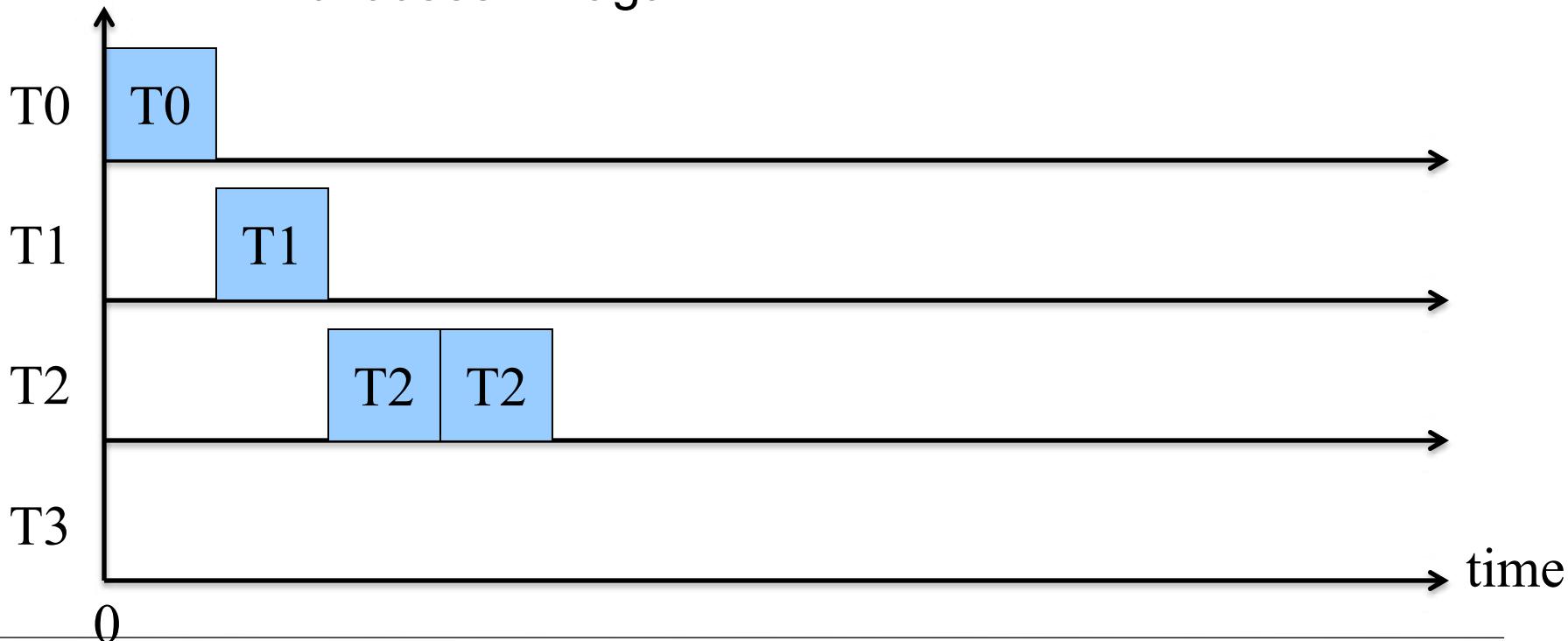
EDF and Preemption

- At time 0, tasks T0 and T1 have arrived
 - EDF chooses T0
- At time 1, T0 finishes, makes deadline
 - EDF chooses T1 next



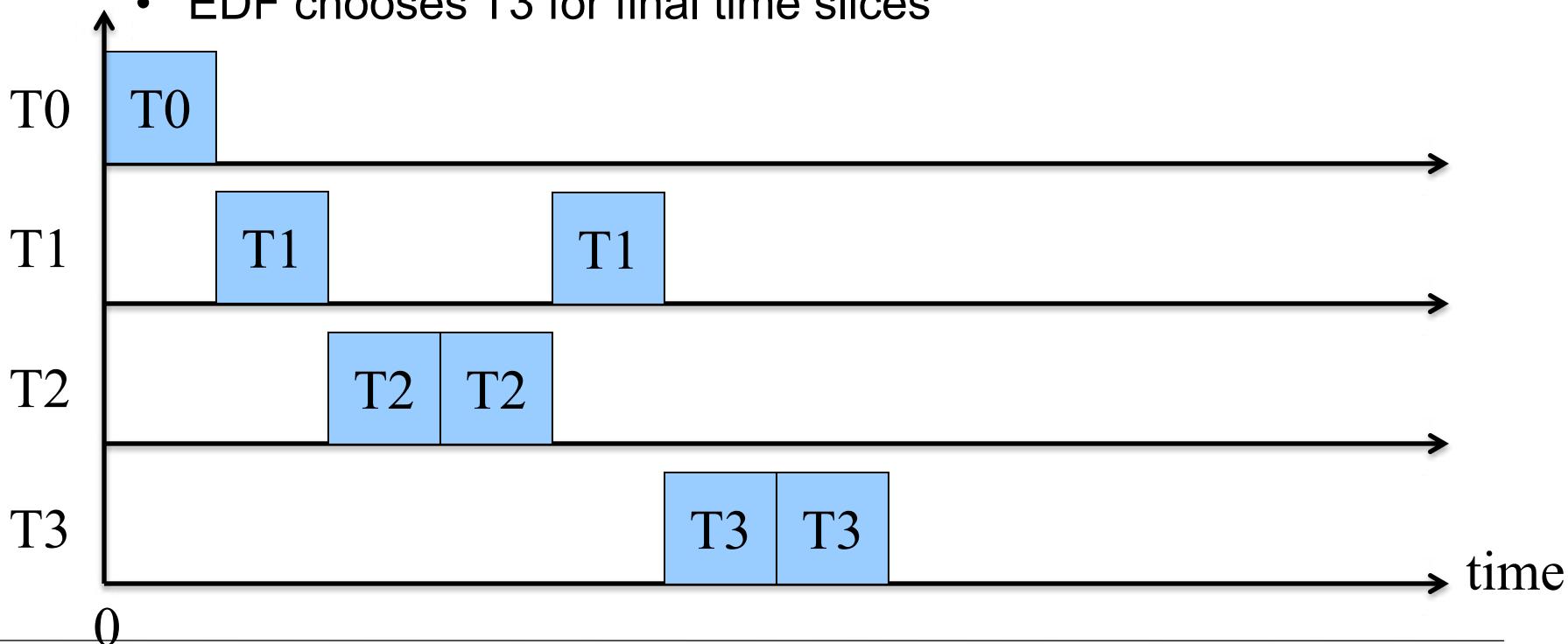
EDF and Preemption

- At time 2, preempt T1
 - EDF chooses newly arrived T2 with earlier deadline
- At time 3, preempt T2
 - EDF chooses T2 again



EDF and Preemption

- At time 4, T2 finishes and makes deadline
 - EDF chooses T1
- At time 5, T1 finishes and makes deadline
 - EDF chooses T3 for final time slices



Deadline Scheduling

- **There are other types of deadline schedulers**
 - Example: a Least Slack algorithm chooses the task with the smallest slack time = time until deadline – remaining execution time
 - i.e. slack is the maximum amount of time that a task can be delayed without missing its deadline
 - Tasks with the least slack are those that have the least flexibility to be delayed given the amount of remaining computation needed before their deadline expires
- **Both EDF and Least Slack are optimal according to different criteria**

Soft Real Time Systems

- ***Soft real time systems seek to meet most deadlines, but allow some to be missed***
 - Unlike hard real time systems, where every deadline must be met or else the system fails
 - Soft real time scheduler may seek to provide probabilistic guarantees
 - e.g. if 60% of deadlines are met, that may be sufficient for some systems
 - Linux supports a soft real-time scheduler based on priorities – we'll see this next

