TCSS 422: OPERATING SYSTEMS

Three Easy Pieces:
Beyond Physical Memory,
I/O Devices

Wes J. Lloyd
Institute of Technology
University of Washington - Tacoma

OBJECTIVES

- Assignment 3 – Page Table Walker
- <u>**Memory Virtualization**</u>
- Beyond Physical Memory – Ch. 21/22
- I/O Devices – Ch. 36
- Final Exam – June 4th

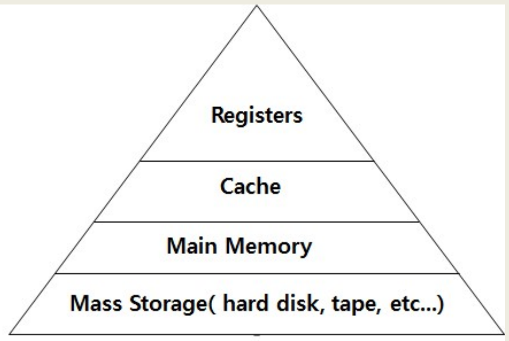
FEEDBACK – 5/23

- Questions on assignment #3…

CHAPTER 21/22: BEYOND PHYSICAL MEMORY

MEMORY HIERARCHY

- Disks (HDD, SSD) provide another level of storage in the memory hierarchy



Memory Hierarchy in modern system

MOTIVATION FOR EXPANDING THE ADDRESS SPACE

- Can provide illusion of an address space larger than physical RAM

- For a single process
  - Convenience
  - Ease of use

- For multiple processes
  - Large virtual memory space for many concurrent processes

LATENCY TIMES

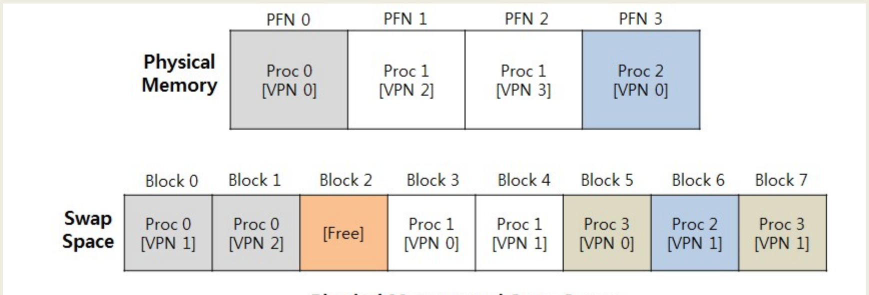
- Design considerations
  - SSDs 4x the time of DRAM
  - HDDs 80x the time of DRAM

Action	Latency (ns)	(μs)	
L1 cache reference	0.5ns		
L2 cache reference	7 ns		14x L1 cache
Mutex lock/unlock	25 ns		
Main memory reference	100 ns		20x L2 cache, 200x L1
Read 4K randomly from SSD*	150,000 ns	150 μs	~1GB/sec SSD
Read 1 MB sequentially from memory	250,000 ns	250 μs	
Read 1 MB sequentially from SSD*	1,000,000 ns	1,000 μs	1 ms ~1GB/sec SSD, 4X memory
Read 1 MB sequentially from disk	20,000,000 ns	20,000 μs	20 ms 80x memory, 20X SSD

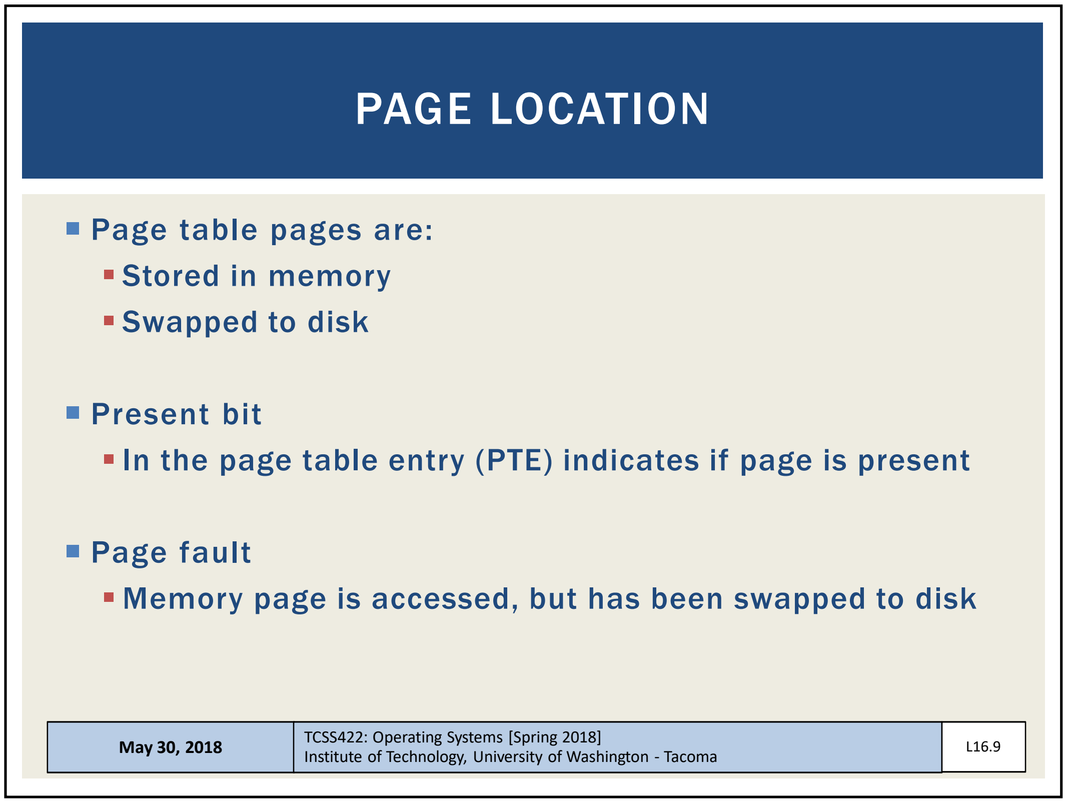
- Latency numbers every programmer should know
- From: https://gist.github.com/jboner/2841832#file-latency-txt

SWAP SPACE

- Disk space for storing memory pages
- "Swap" them in and out of memory to disk as needed

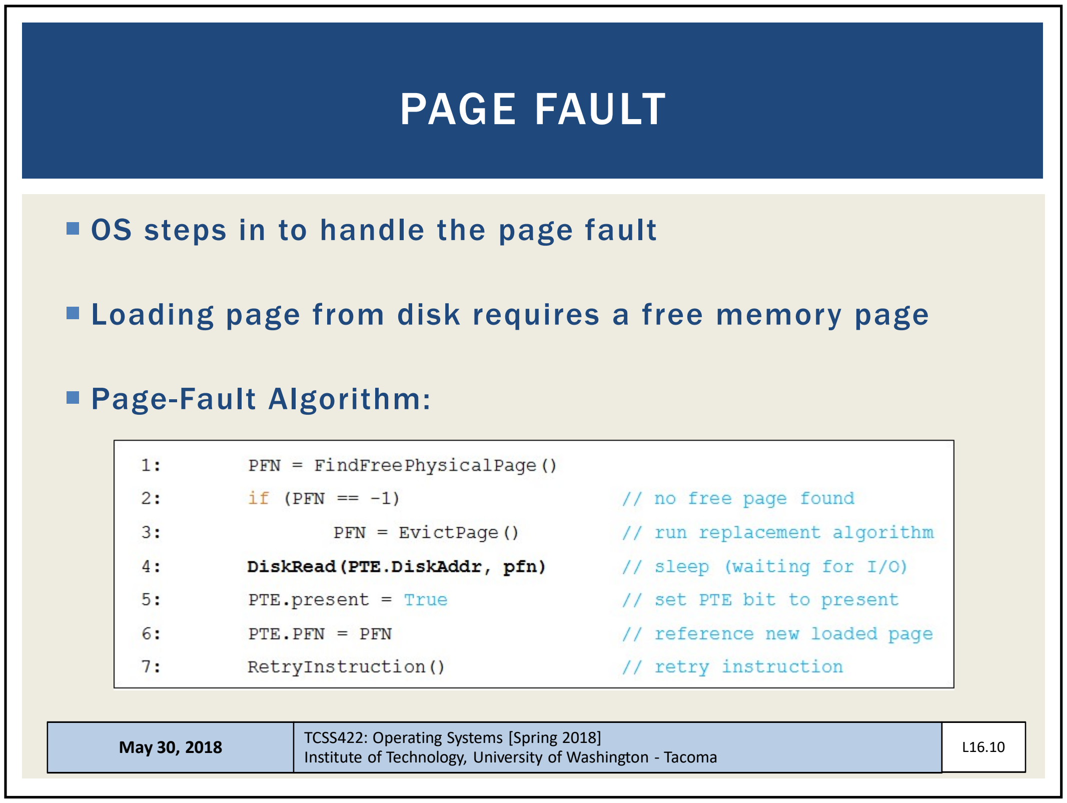


Physical Memory and Swap Space



PAGE LOCATION

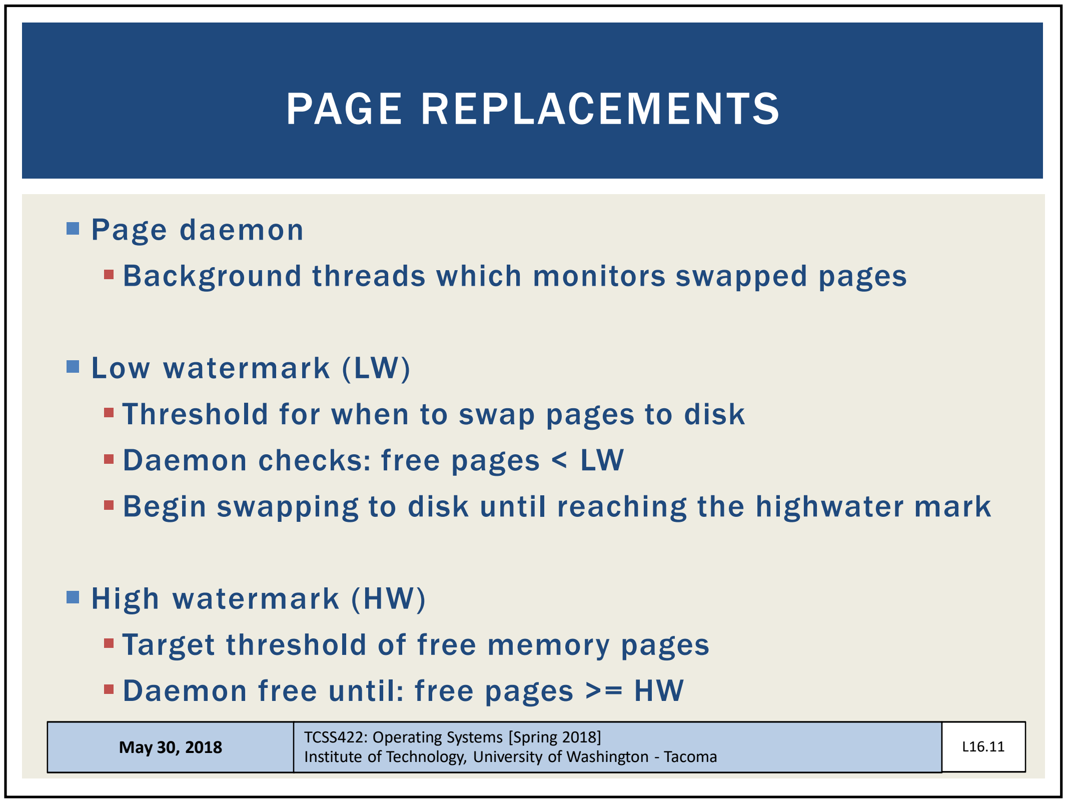
- Page table pages are:
  - Stored in memory
  - Swapped to disk
- Present bit
  - In the page table entry (PTE) indicates if page is present
- Page fault
  - Memory page is accessed, but has been swapped to disk



PAGE FAULT

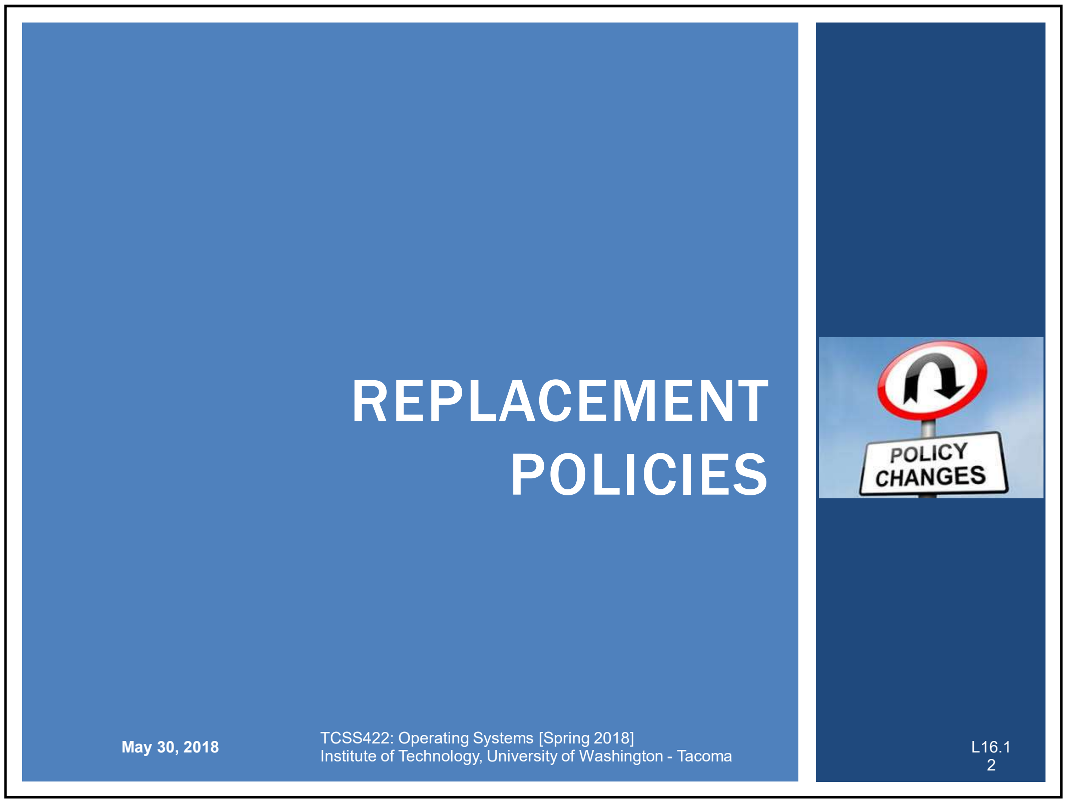
- OS steps in to handle the page fault
- Loading page from disk requires a free memory page
- Page-Fault Algorithm:

```
PFN = FindFreePhysicalPage()
if (PFN == -1)                    // no free page found
        PFN = EvictPage()         // run replacement algorithm
DiskRead(PTE.DiskAddr, pfn)       // sleep (waiting for I/O)
PTE.present = True                // set PTE bit to present
PTE.PFN = PFN                     // reference new loaded page
RetryInstruction()                // retry instruction
```

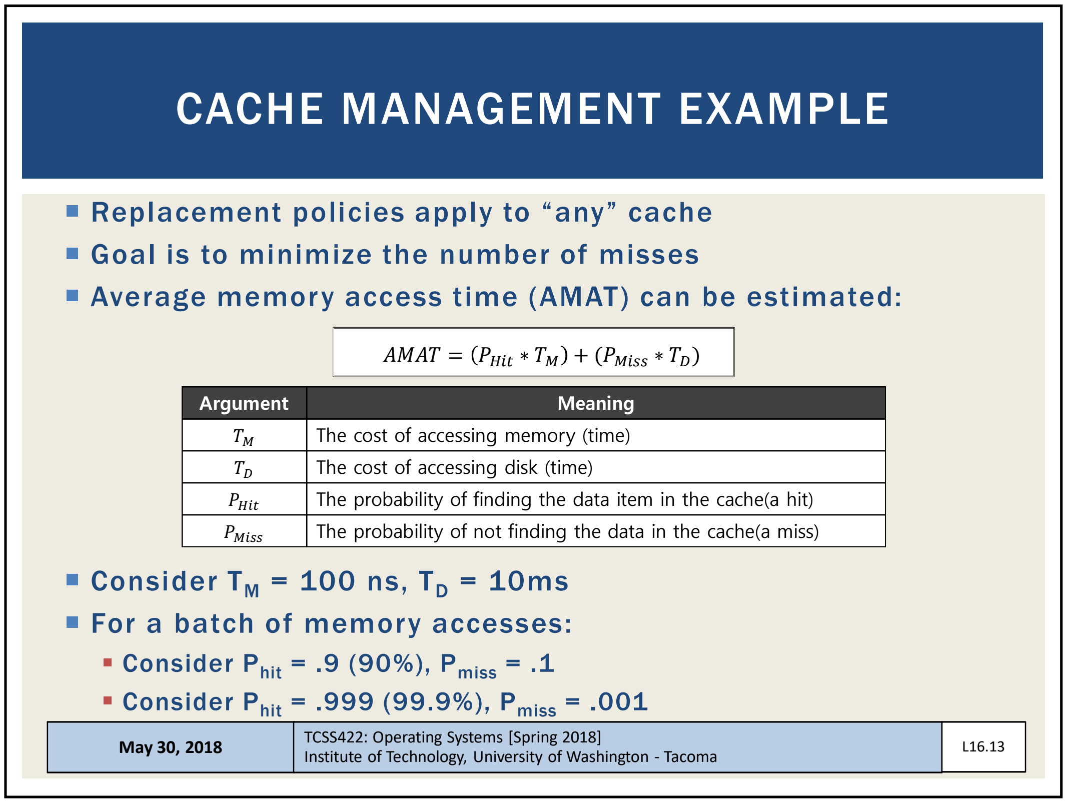


PAGE REPLACEMENTS

- Page daemon
  - Background threads which monitors swapped pages
- Low watermark (LW)
  - Threshold for when to swap pages to disk
  - Daemon checks: free pages < LW
  - Begin swapping to disk until reaching the highwater mark
- High watermark (HW)
  - Target threshold of free memory pages
  - Daemon free until: free pages >= HW



REPLACEMENT POLICIES



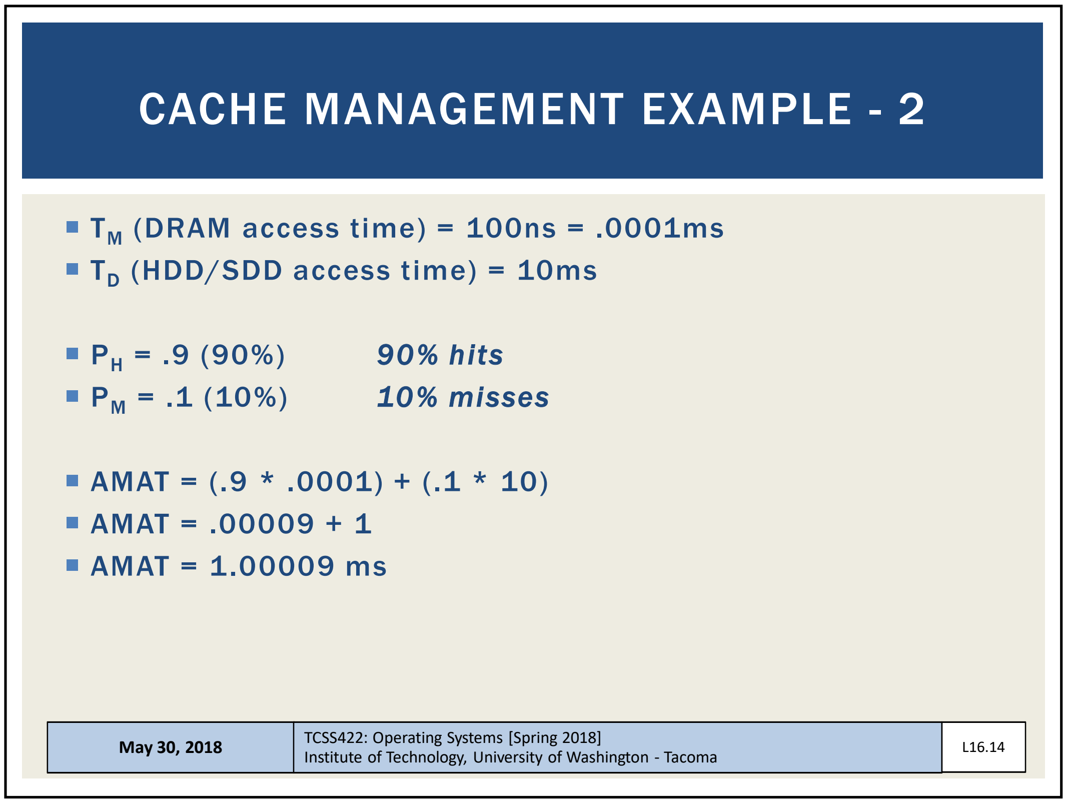
CACHE MANAGEMENT EXAMPLE

- Replacement policies apply to “any” cache
- Goal is to minimize the number of misses
- Average memory access time (AMAT) can be estimated:

$$AMAT = (P_{Hit} * T_M) + (P_{Miss} * T_D)$$

Argument	Meaning
T_M	The cost of accessing memory (time)
T_D	The cost of accessing disk (time)
P_{Hit}	The probability of finding the data item in the cache(a hit)
P_{Miss}	The probability of not finding the data in the cache(a miss)

- Consider T_M = 100 ns, T_D = 10ms
- For a batch of memory accesses:
  - Consider P_{hit} = .9 (90%), P_{miss} = .1
  - Consider P_{hit} = .999 (99.9%), P_{miss} = .001

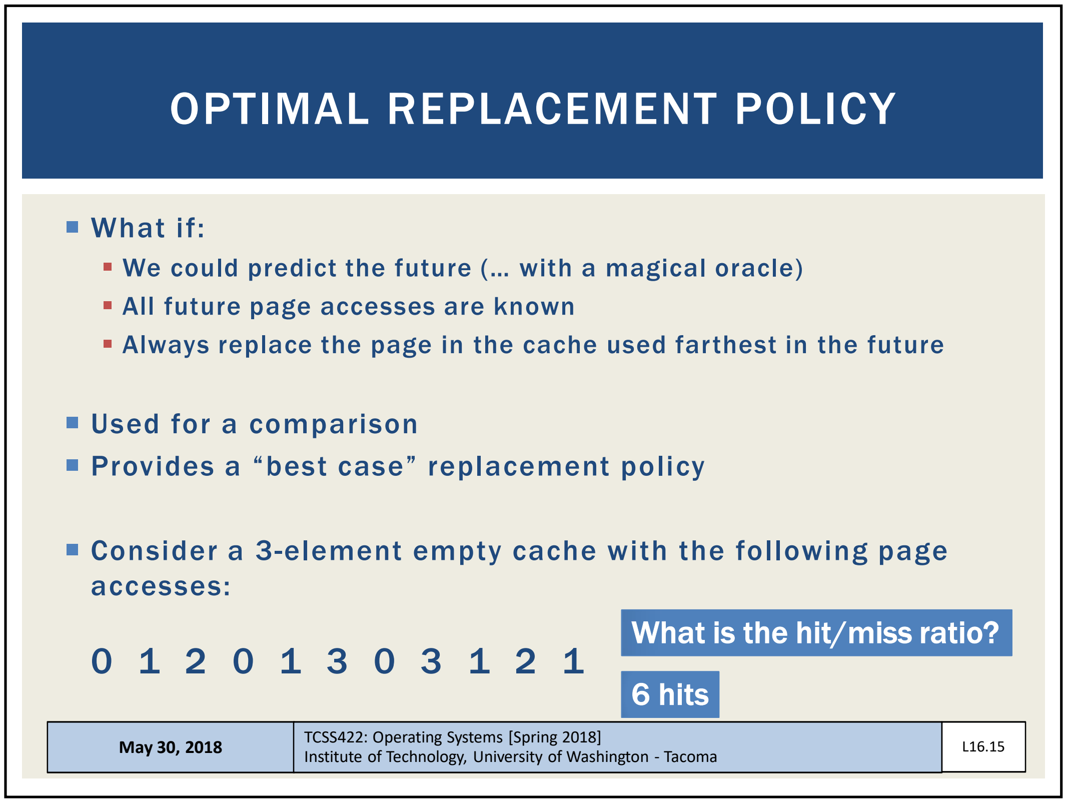


CACHE MANAGEMENT EXAMPLE - 2

- T_M (DRAM access time) = 100ns = .0001ms
- T_D (HDD/SDD access time) = 10ms

- P_H = .9 (90%)   ***90% hits***
- P_M = .1 (10%)   ***10% misses***

- AMAT = (.9 * .0001) + (.1 * 10)
- AMAT = .00009 + 1
- AMAT = 1.00009 ms



OPTIMAL REPLACEMENT POLICY

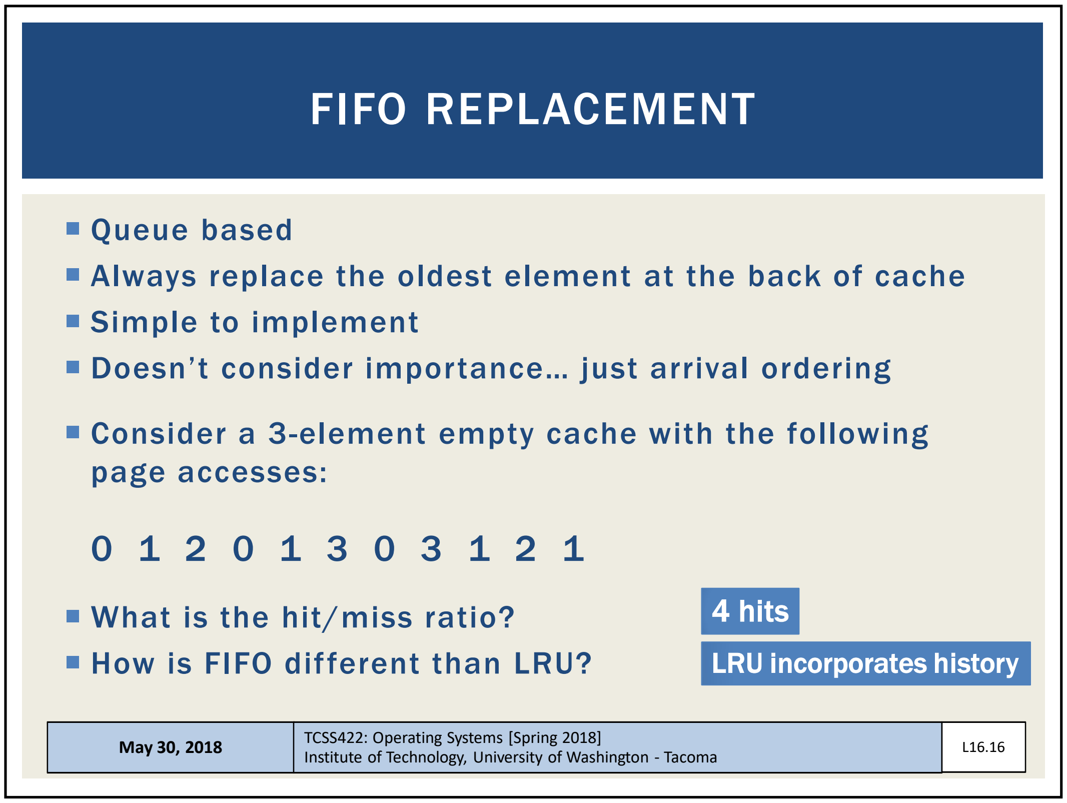
- What if:
  - We could predict the future (… with a magical oracle)
  - All future page accesses are known
  - Always replace the page in the cache used farthest in the future
- Used for a comparison
- Provides a "best case" replacement policy
- Consider a 3-element empty cache with the following page accesses:

0 1 2 0 1 3 0 3 1 2 1

What is the hit/miss ratio?

6 hits

May 30, 2018 | TCSS422: Operating Systems [Spring 2018] Institute of Technology, University of Washington - Tacoma | L16.15



FIFO REPLACEMENT

- Queue based
- Always replace the oldest element at the back of cache
- Simple to implement
- Doesn't consider importance… just arrival ordering
- Consider a 3-element empty cache with the following page accesses:

0 1 2 0 1 3 0 3 1 2 1

- What is the hit/miss ratio?
- How is FIFO different than LRU?

4 hits

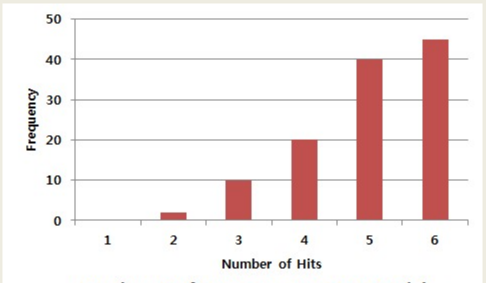
LRU incorporates history

May 30, 2018 | TCSS422: Operating Systems [Spring 2018] Institute of Technology, University of Washington - Tacoma | L16.16

RANDOM REPLACEMENT

- Pick a page at random to replace
- Simple and fast implementation
- Performance depends on luck of random choices

0 1 2 0 1 3 0 3 1 2 1



Random Performance over 10,000 Trials

HISTORY-BASED POLICIES

- LRU: Least recently used <u>*(adds "a memory" to the cache)*</u>
- Always replace page with oldest access time (front)
- Always move end of cache when element is read again
- Considers temporal locality (*when pg was last accessed*)

<u>3-element cache:</u>

0 1 2 0 1 3 0 3 1 2 1

What is the hit/miss ratio?

6 hits

- LFU: Least frequently used
- Always replace page with fewest accesses (front)
- Consider frequency of page accesses

<u>3-element cache:</u>

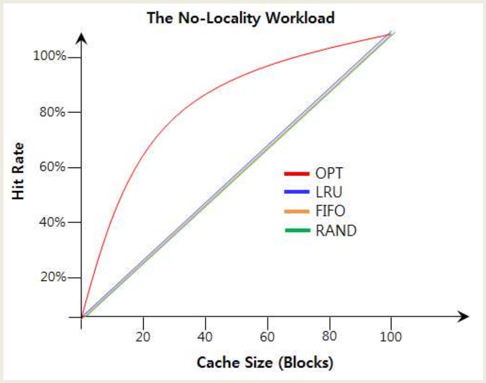
0 1 2 0 1 3 0 3 1 2 1

Hit/miss ratio is=

6 hits

WORKLOAD EXAMPLES: NO-LOCALITY

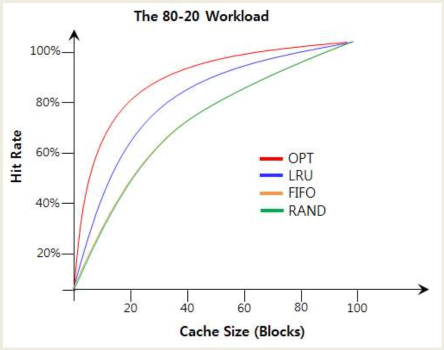
- No-Locality (Random Access) Workload
  - Perform 10,000 random page accesses
  - Across set of 100 memory pages



When the cache is large enough to fit the entire workload, it doesn't matter which policy you use.

WORKLOAD EXAMPLES: 80/20

- 80/20 Workload
  - Perform 10,000 page accesses, against set of 100 pages
  - 80% of accesses are to 20% of pages (hot pages)
  - 20% of accesses are to 80% of pages (cold pages)

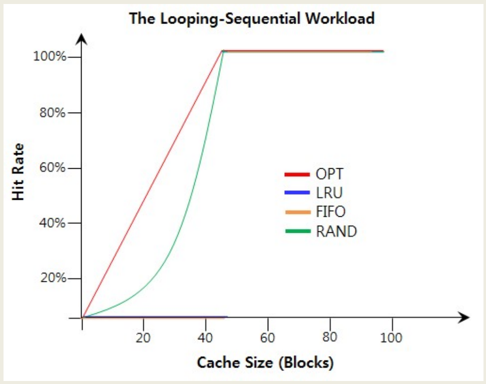


LRU is more likely to hold onto hot pages

(recalls history)

WORKLOAD EXAMPLES: SEQUENTIAL

- Looping sequential workload
  - Refer to 50 pages in sequence: 0, 1, …, 49
  - Repeat loop



Random performs better than FIFO and LRU for cache sizes < 50

Algorithms should provide "scan resistance"

IMPLEMENTING LRU

- Implementing last recently used (LRU) requires tracking access time for all system memory pages
- Times can be tracked with a list
- For cache eviction, we must scan an entire list
- Consider: 4GB memory system (2^{32}),
  with 4KB pages (2^{12})
- This requires 2^{20} comparisons !!!
- Simplification is needed
  - Consider how to approximate the oldest page access

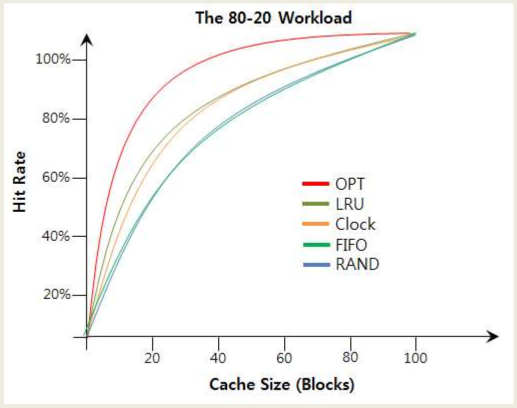
IMPLEMENTING LRU - 2

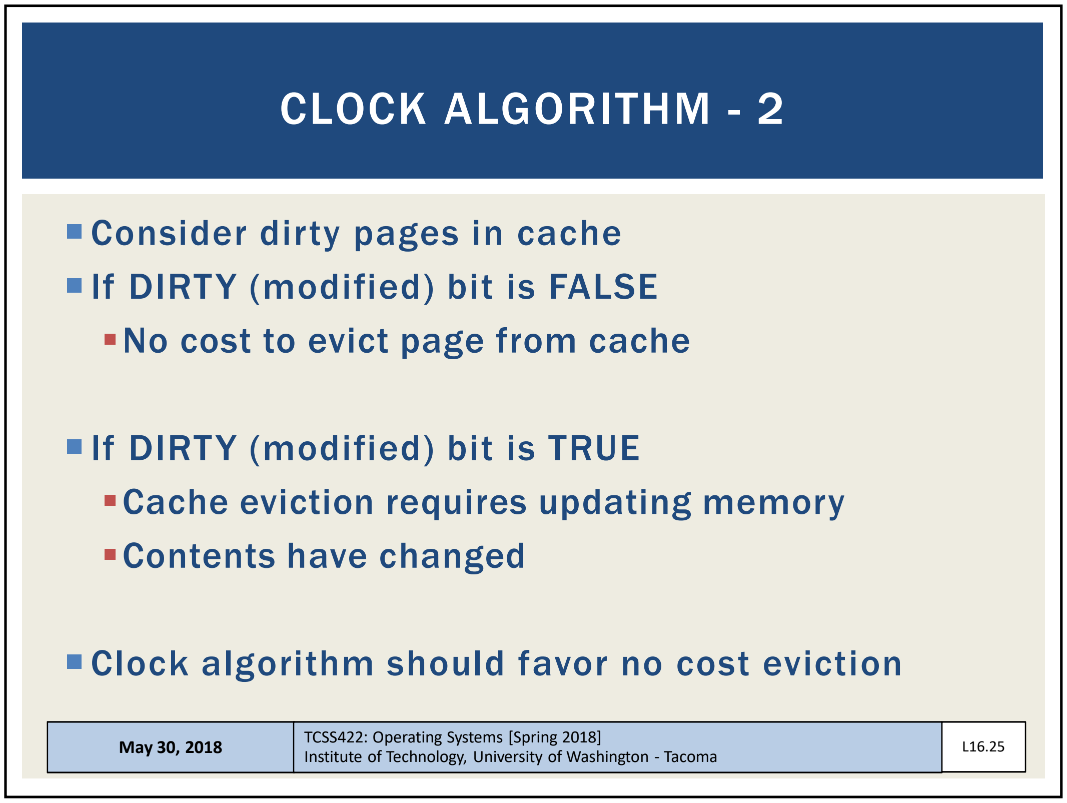
- Harness the Page Table Entry (PTE) Use Bit
- HW sets to 1 when page is used
- OS sets to 0

- Clock algorithm (*approximate LRU*)
  - Refer to pages in a circular list
  - Clock hand points to current page
  - Loops around
    - IF USE_BIT=1 set to USE_BIT = 0
    - IF USE_BIT=0 replace page

CLOCK ALGORITHM

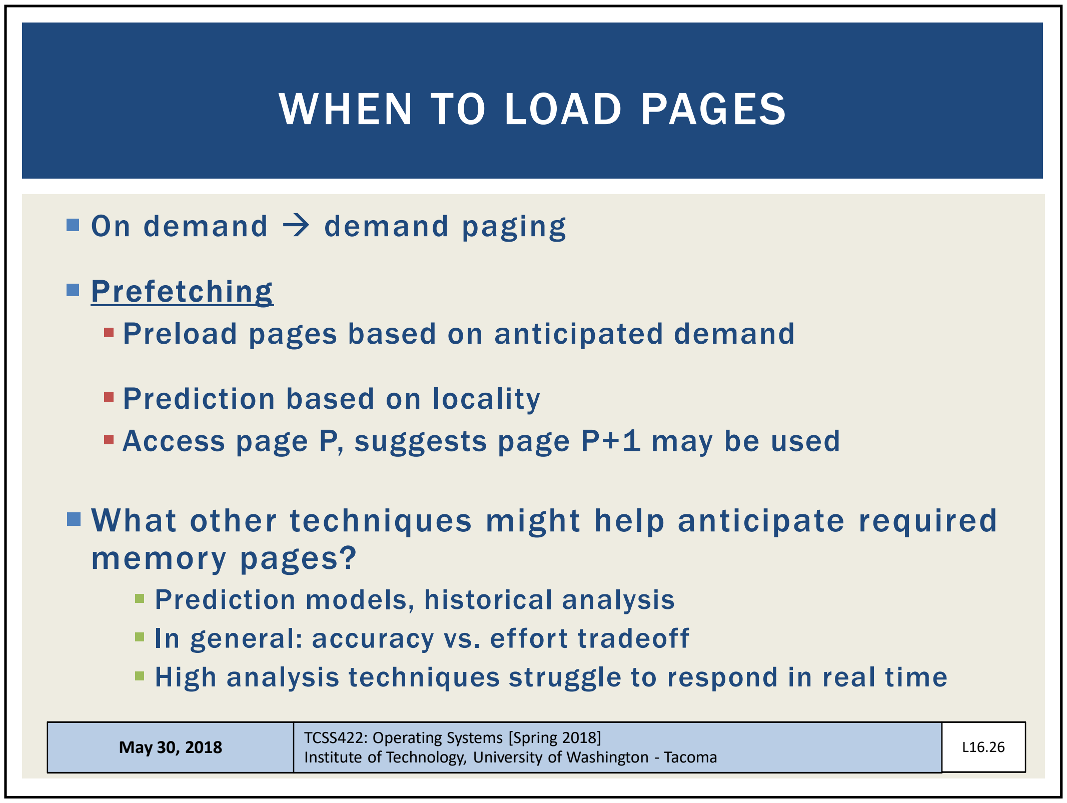
- Not as efficient as LRU, but better than other replacement algorithms that do not consider history





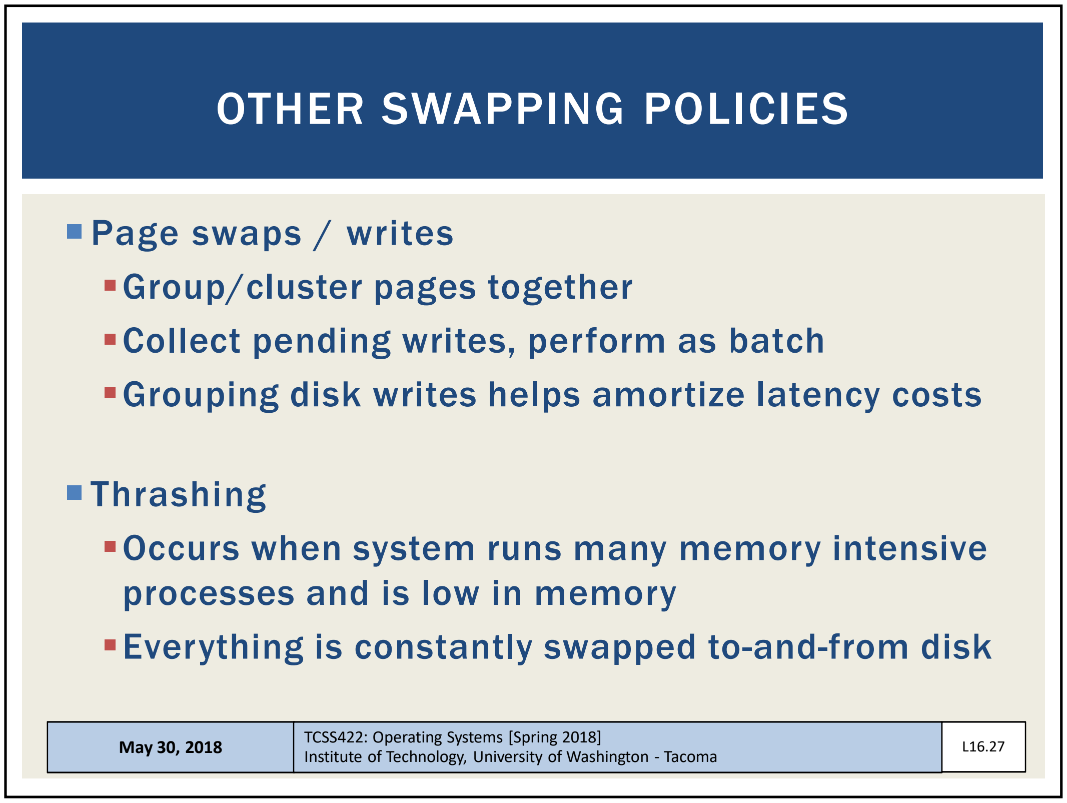
CLOCK ALGORITHM - 2

- Consider dirty pages in cache
- If DIRTY (modified) bit is FALSE
  - No cost to evict page from cache
- If DIRTY (modified) bit is TRUE
  - Cache eviction requires updating memory
  - Contents have changed
- Clock algorithm should favor no cost eviction



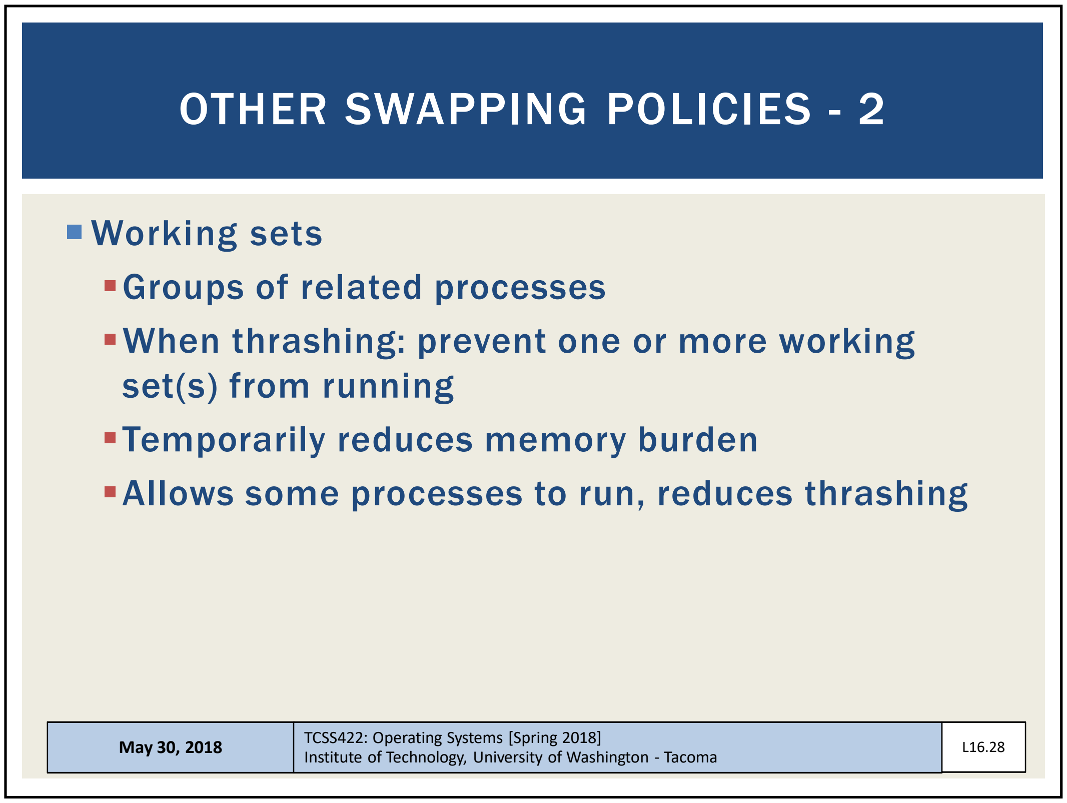
WHEN TO LOAD PAGES

- On demand → demand paging
- Prefetching
  - Preload pages based on anticipated demand
  - Prediction based on locality
  - Access page P, suggests page P+1 may be used
- What other techniques might help anticipate required memory pages?
  - Prediction models, historical analysis
  - In general: accuracy vs. effort tradeoff
  - High analysis techniques struggle to respond in real time



OTHER SWAPPING POLICIES

- Page swaps / writes
  - Group/cluster pages together
  - Collect pending writes, perform as batch
  - Grouping disk writes helps amortize latency costs
- Thrashing
  - Occurs when system runs many memory intensive processes and is low in memory
  - Everything is constantly swapped to-and-from disk



OTHER SWAPPING POLICIES - 2

- Working sets
  - Groups of related processes
  - When thrashing: prevent one or more working set(s) from running
  - Temporarily reduces memory burden
  - Allows some processes to run, reduces thrashing

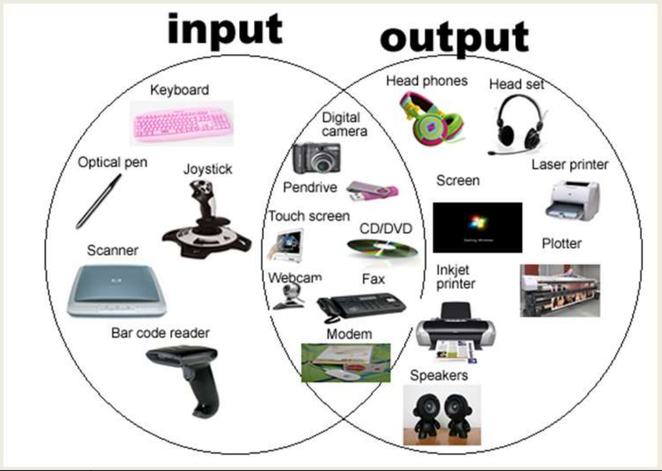
CHAPTER 36: I/O DEVICES

OBJECTIVES

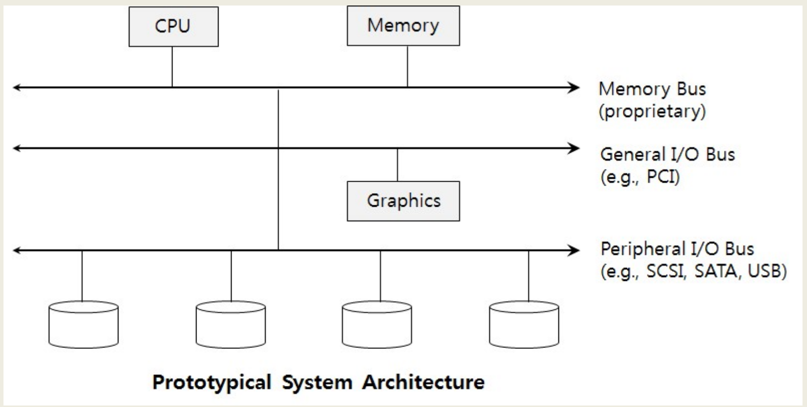
- Chapter 36
  - Polling vs Interrupts
  - Programmed I/O (PIO)
    - Port-mapped I/O (PMIO)
    - Memory-mapped I/O (MMIO)
  - Direct memory Access (DMA)

I/O DEVICES

- Modern computer systems interact with a variety of devices



COMPUTER SYSTEM ARCHITECTURE


CPU
Memory
Memory Bus (proprietary)
General I/O Bus (e.g., PCI)
Graphics
Peripheral I/O Bus (e.g., SCSI, SATA, USB)

Prototypical System Architecture

VERY FAST: CPU is attached to main memory via a Memory bus.

FAST: High speed devices (e.g. video) are connected via a General I/O bus.

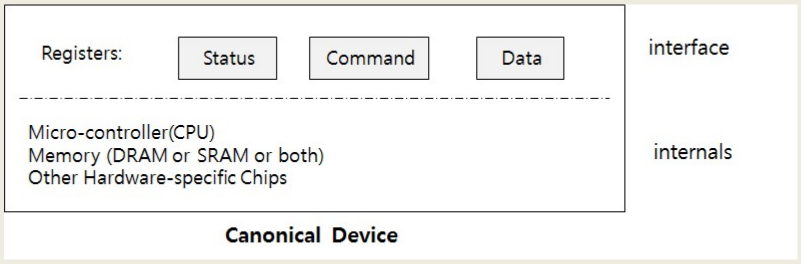
SLOWER: Disks are connected via a Peripheral I/O bus.

I/O BUSES

- Buses
  - Buses closer to the CPU are faster
  - Can support fewer devices
  - Further buses are slower, but support more devices
- Physics and costs dictate "levels"
  - Memory bus
  - General I/O bus
  - Peripheral I/O bus
- Tradeoff space: speed vs. locality

CANONICAL DEVICE

- Consider an arbitrary canonical *"standard/generic"* device



- Two primary components
  - Interface (registers for communication)
  - Internals: Local CPU, memory, specific chips, firmware (embedded software)

CANONICAL DEVICE: HARDWARE INTERFACE

- Status register
  - Maintains current device status
- Command register
  - Where commands for interaction are sent
- Data register
  - Used to send and receive data to the device

General concept:
The OS interacts and controls device behavior by reading and writing the device registers.

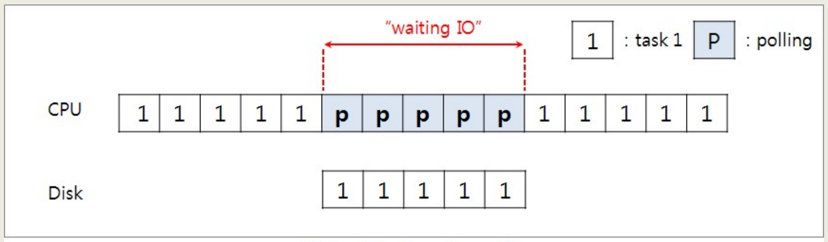
OS DEVICE INTERACTION

- Common example of device interaction

```
while ( STATUS == BUSY)                  ⇦ Poll- Is device available?
    ; //wait until device is not busy
write data to data register              ⇦ Command parameterization
write command to command register        ⇦ Send command
    Doing so starts the device and executes the command
while ( STATUS == BUSY)                  ⇦ Poll – Is device done?
    ; //wait until device is done with your request
```

POLLING

- OS checks if device is *READY* by repeatedly checking the STATUS register
  - Simple approach
  - CPU cycles are wasted without doing meaningful work
  - Ok if only a few cycles, for rapid devices that are often READY
  - BUT polling, as with “spin locks” we understand is inefficient



CPU utilization by polling

INTERRUPTS VS POLLING

- For longer waits, put process waiting on I/O to sleep
- Context switch (C/S) to another process
- When I/O completes, fire an interrupt to initiate C/S back
  - Advantage: better multi-tasking and CPU utilization
  - Avoids: unproductive CPU cycles (polling)

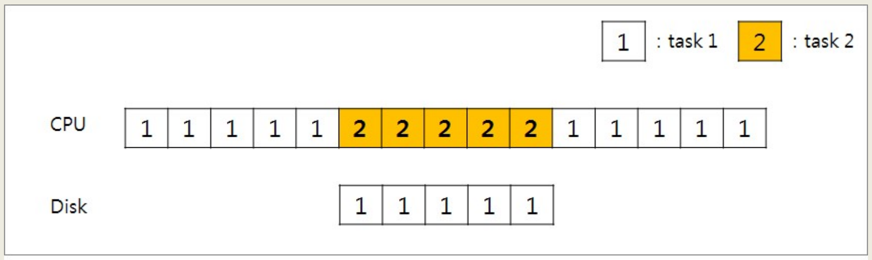
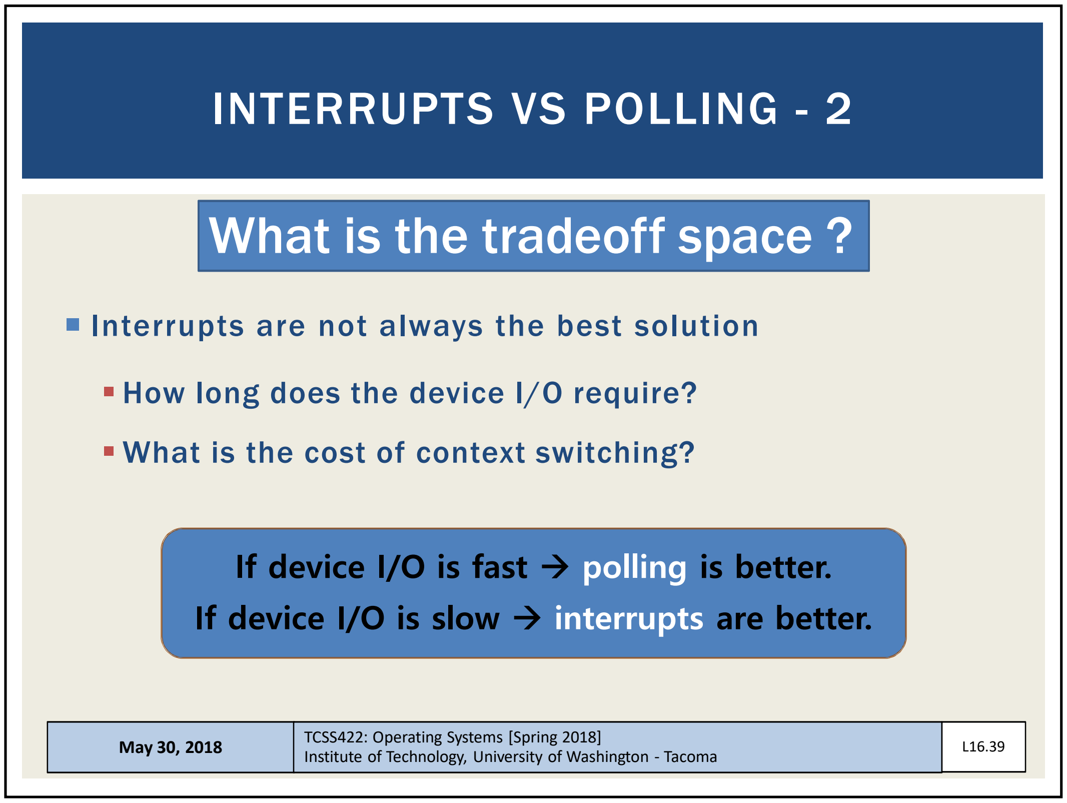


Diagram of CPU utilization by interrupt



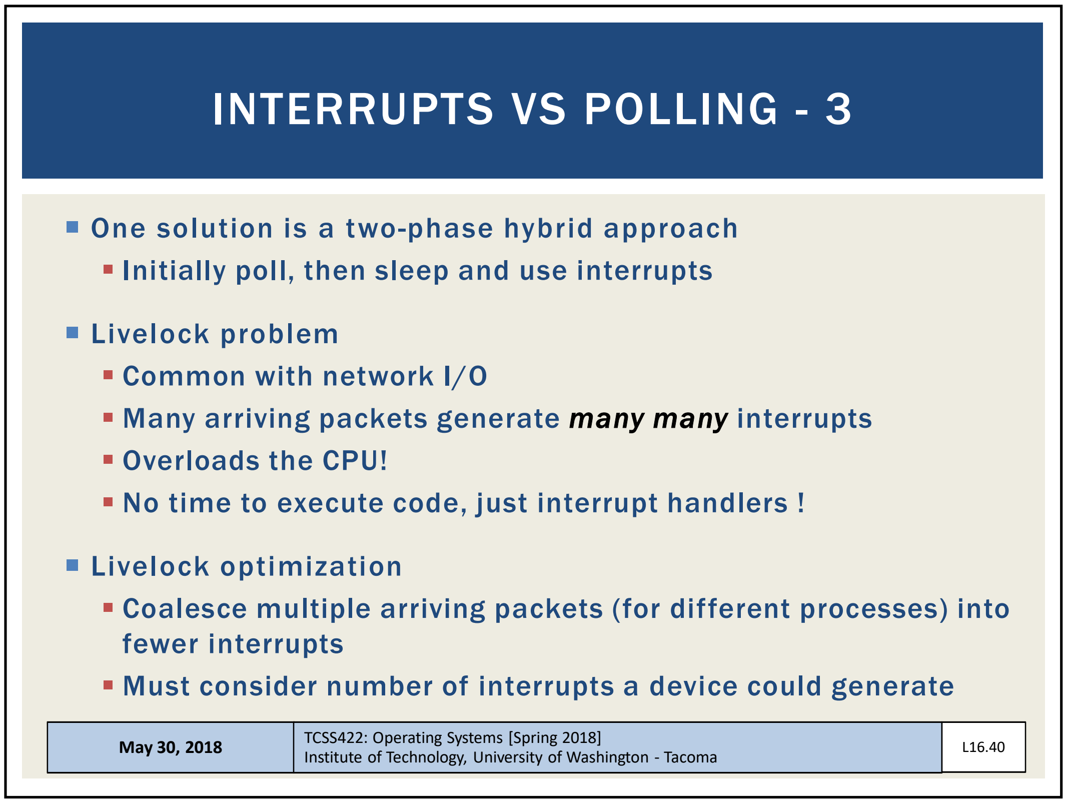
INTERRUPTS VS POLLING - 2

What is the tradeoff space ?

- Interrupts are not always the best solution
  - How long does the device I/O require?
  - What is the cost of context switching?

If device I/O is fast → polling is better.
If device I/O is slow → interrupts are better.

May 30, 2018 | TCSS422: Operating Systems [Spring 2018] Institute of Technology, University of Washington - Tacoma | L16.39



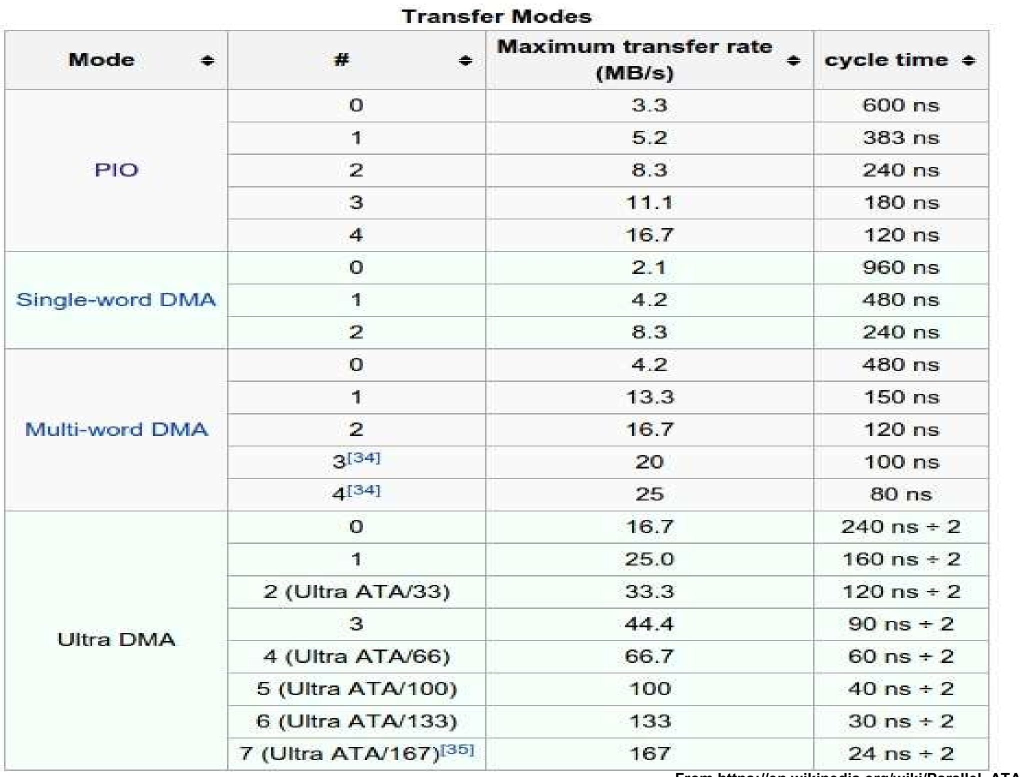
INTERRUPTS VS POLLING - 3

- One solution is a two-phase hybrid approach
  - Initially poll, then sleep and use interrupts
- Livelock problem
  - Common with network I/O
  - Many arriving packets generate ***many many*** interrupts
  - Overloads the CPU!
  - No time to execute code, just interrupt handlers !
- Livelock optimization
  - Coalesce multiple arriving packets (for different processes) into fewer interrupts
  - Must consider number of interrupts a device could generate

May 30, 2018 | TCSS422: Operating Systems [Spring 2018] Institute of Technology, University of Washington - Tacoma | L16.40

DEVICE I/O

- To interact with a device we must send/receive DATA
- There are two general approaches:
  - Programmed I/O (PIO)
  - Direct memory access (DMA)



Transfer Modes

Mode	#	Maximum transfer rate (MB/s)	cycle time
PIO	0	3.3	600 ns
	1	5.2	383 ns
	2	8.3	240 ns
	3	11.1	180 ns
	4	16.7	120 ns
Single-word DMA	0	2.1	960 ns
	1	4.2	480 ns
	2	8.3	240 ns
Multi-word DMA	0	4.2	480 ns
	1	13.3	150 ns
	2	16.7	120 ns
	3[34]	20	100 ns
	4[34]	25	80 ns
Ultra DMA	0	16.7	240 ns ÷ 2
	1	25.0	160 ns ÷ 2
	2 (Ultra ATA/33)	33.3	120 ns ÷ 2
	3	44.4	90 ns ÷ 2
	4 (Ultra ATA/66)	66.7	60 ns ÷ 2
	5 (Ultra ATA/100)	100	40 ns ÷ 2
	6 (Ultra ATA/133)	133	30 ns ÷ 2
	7 (Ultra ATA/167)[35]	167	24 ns ÷ 2

From https://en.wikipedia.org/wiki/Parallel_ATA

PROGRAMMED I/O (PIO)

- Spend CPU time to perform I/O
- CPU is involved with the data movement (input/output)
- PIO is slow –CPU is occupied with meaningless work

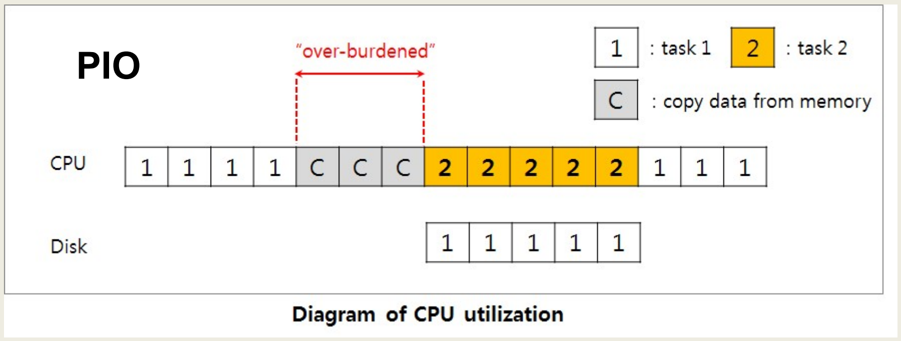
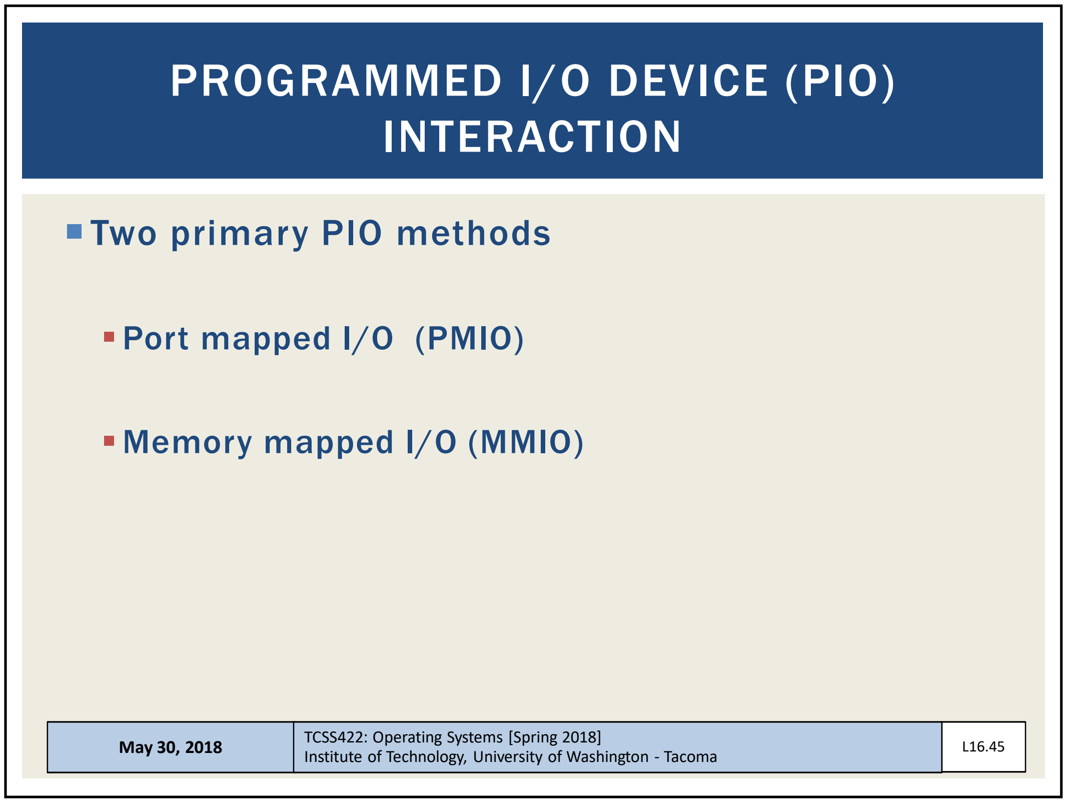


Diagram of CPU utilization

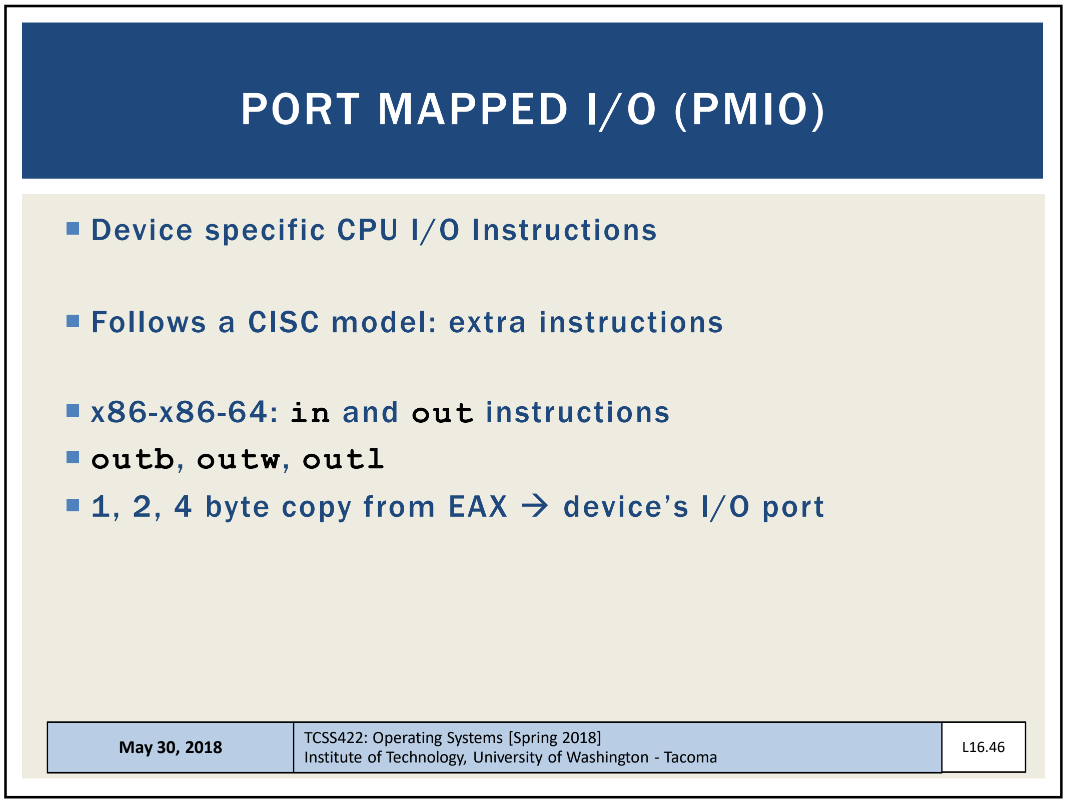
PIO DEVICES

- Legacy serial ports
- Legacy parallel ports
- PS/2 keyboard and mouse
- Legacy MIDI, joysticks
- Old network interfaces



PROGRAMMED I/O DEVICE (PIO) INTERACTION

- Two primary PIO methods
  - Port mapped I/O (PMIO)
  - Memory mapped I/O (MMIO)



PORT MAPPED I/O (PMIO)

- Device specific CPU I/O Instructions
- Follows a CISC model: extra instructions
- x86-x86-64: `in` and `out` instructions
- `outb`, `outw`, `outl`
- 1, 2, 4 byte copy from EAX → device's I/O port

MEMORY MAPPED I/O (MMIO)

- Device's memory is mapped to CPU memory
- Tenet of RISC CPUs: instructions are eliminated, CPU is simpler
- Old days: 16-bit CPUs didn't have a lot of spare memory space
- Today's CPUs: 32-bit (4GB addr space) & 64-bit (128 TB addr space)
- Regular CPU instructions used to access device: mapped to memory
- Devices monitor CPU address bus and respond to their addresses
- I/O device address areas of memory are **reserved** for I/O
  - Must not be available for normal memory operations.

DIRECT MEMORY ACCESS (DMA)

- Copy data in memory by ***offloading*** to "DMA controller"
- Many devices (including CPUs) integrate DMA controllers
- CPU gives DMA: memory address, size, and copy instruction
- DMA performs I/O independent of the CPU
- DMA controller generates CPU interrupt when I/O completes

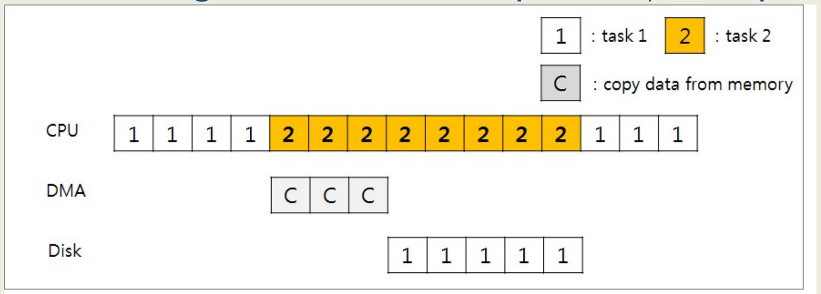
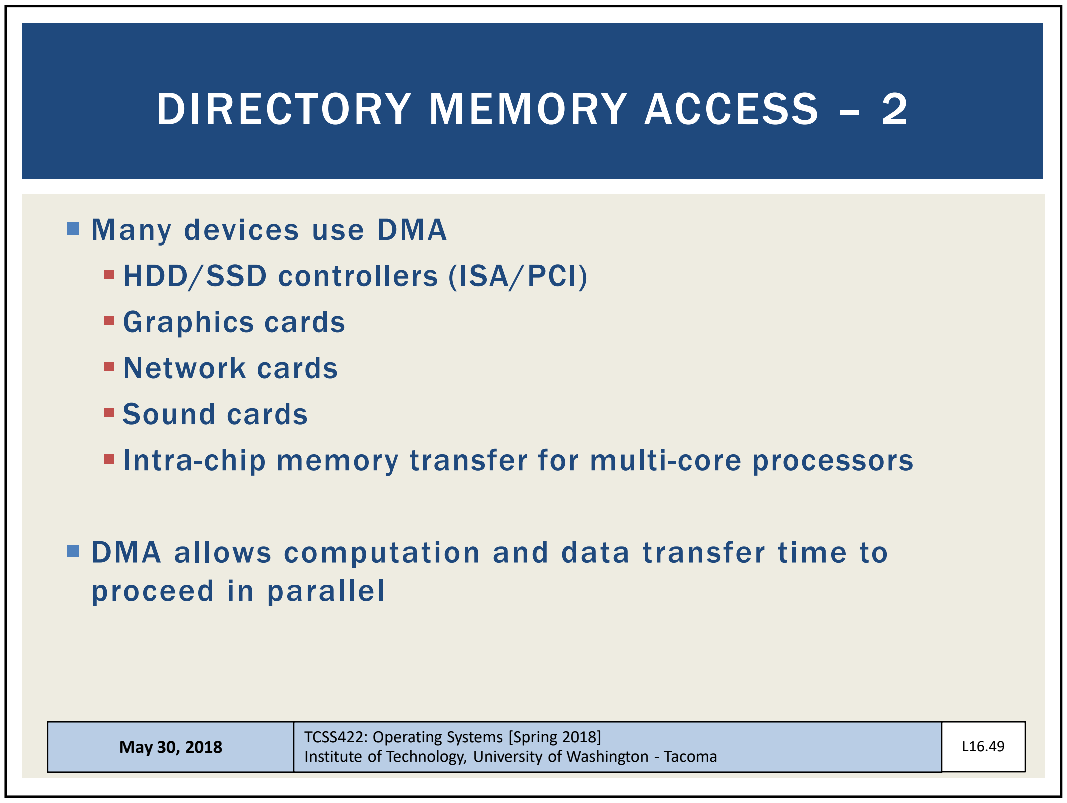


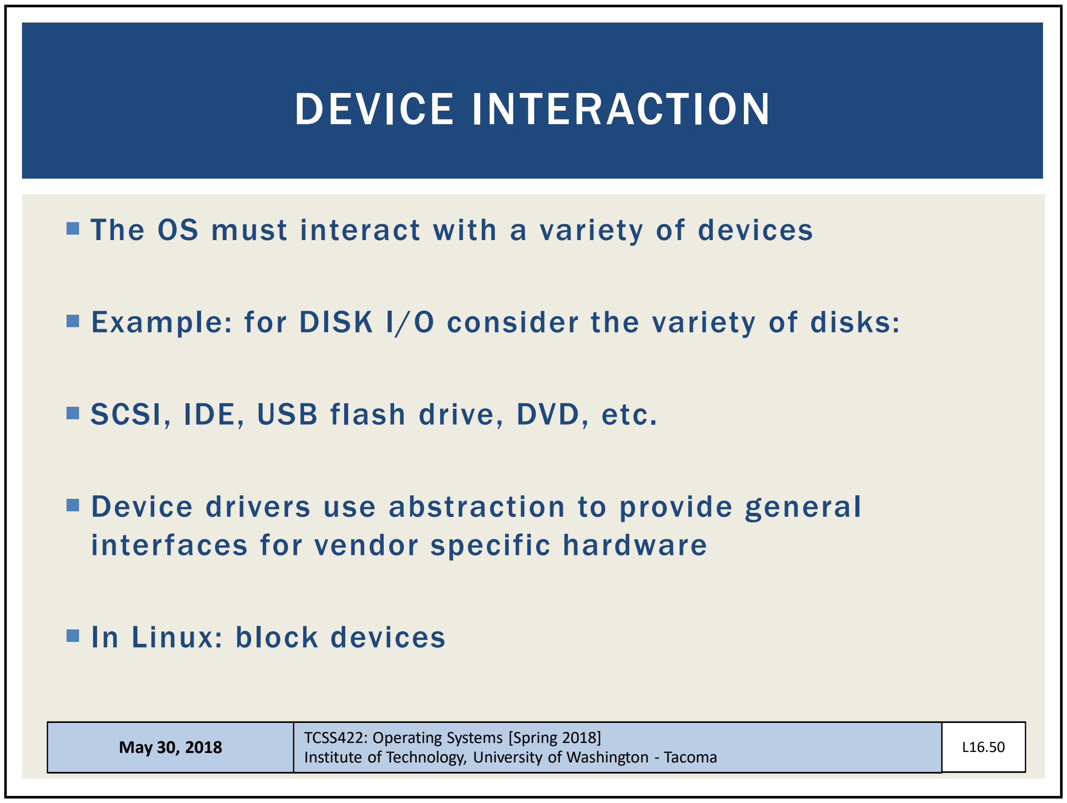
Diagram of CPU utilization by DMA



DIRECTORY MEMORY ACCESS – 2

- Many devices use DMA
  - HDD/SSD controllers (ISA/PCI)
  - Graphics cards
  - Network cards
  - Sound cards
  - Intra-chip memory transfer for multi-core processors
- DMA allows computation and data transfer time to proceed in parallel

May 30, 2018 | TCSS422: Operating Systems [Spring 2018] Institute of Technology, University of Washington - Tacoma | L16.49



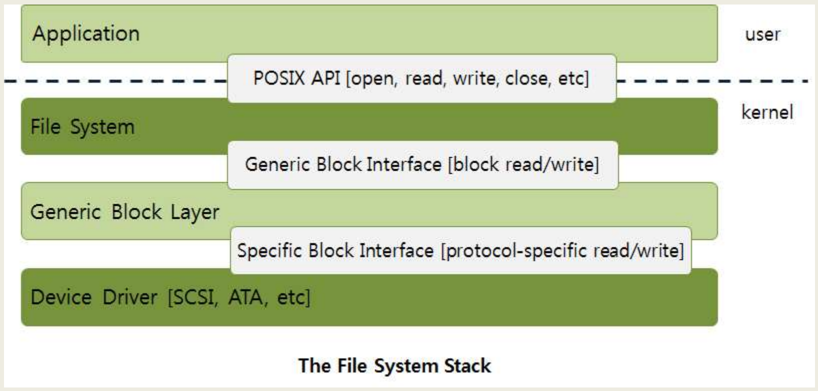
DEVICE INTERACTION

- The OS must interact with a variety of devices
- Example: for DISK I/O consider the variety of disks:
- SCSI, IDE, USB flash drive, DVD, etc.
- Device drivers use abstraction to provide general interfaces for vendor specific hardware
- In Linux: block devices

May 30, 2018 | TCSS422: Operating Systems [Spring 2018] Institute of Technology, University of Washington - Tacoma | L16.50

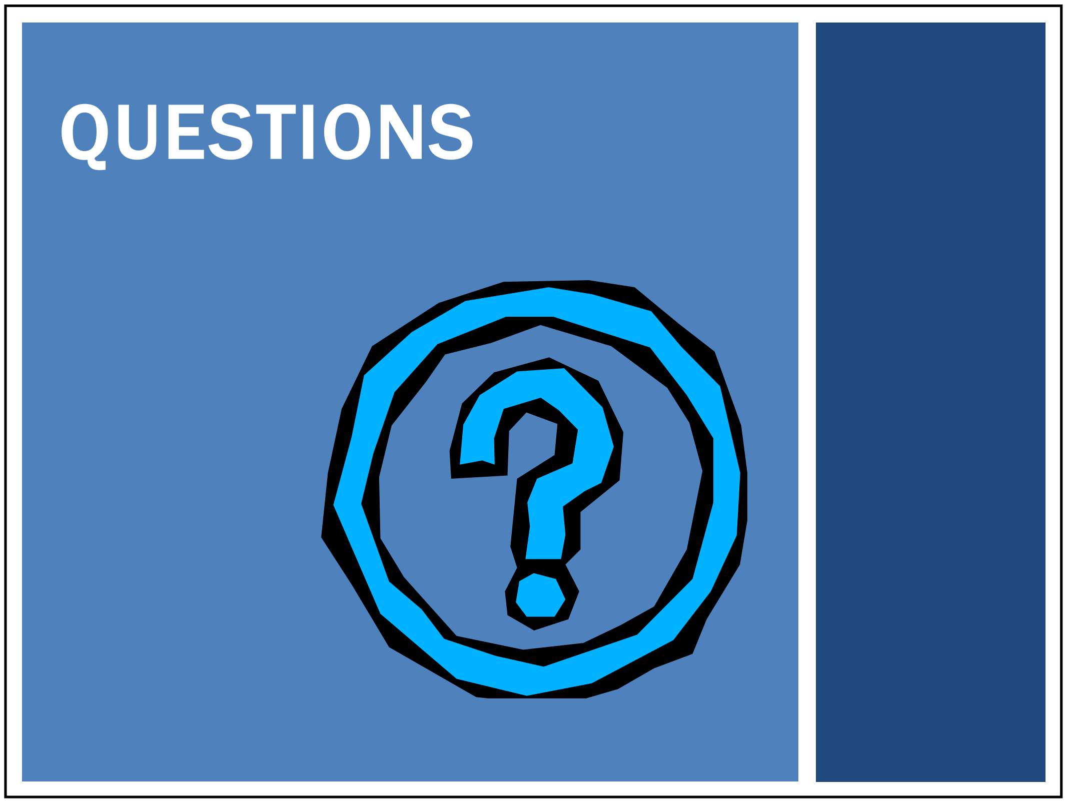
FILE SYSTEM ABSTRACTION

- Layers of I/O abstraction in Linux
- C functions (open, read, write) issue **block read** and **write** requests to the generic block layer


Application
user
POSIX API [open, read, write, close, etc]
kernel
File System
Generic Block Interface [block read/write]
Generic Block Layer
Specific Block Interface [protocol-specific read/write]
Device Driver [SCSI, ATA, etc]
The File System Stack

FILE SYSTEM ABSTRACTION ISSUES

- **Too much abstraction**
- Many devices provide special capabilities
- Example: SCSI Error handling
- SCSI devices provide extra detail which are lost to the OS
- **Buggy device drivers**
- 70% of OS code is in device drivers
- Device drivers are required for every device plugged in
- Drivers are often 3rd party, which is not quality controlled at the same level as the OS (Linux, Windows, MacOS, etc.)



QUESTIONS