

Lecture 8

Demand Paging

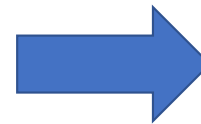
Prof. Yinqian Zhang

Spring 2023

Demand Paging Mechanisms

How To Go Beyond Physical Memory?

- How to support large address space?
 - 64-bit machine supports up to 4EB address space
 - Applications may use more space than available in physical memory
- Solution: stash away portions of address spaces that aren't currently in use
 - in the next-level of storage (e.g., hard disk drive)
 - slower but much larger

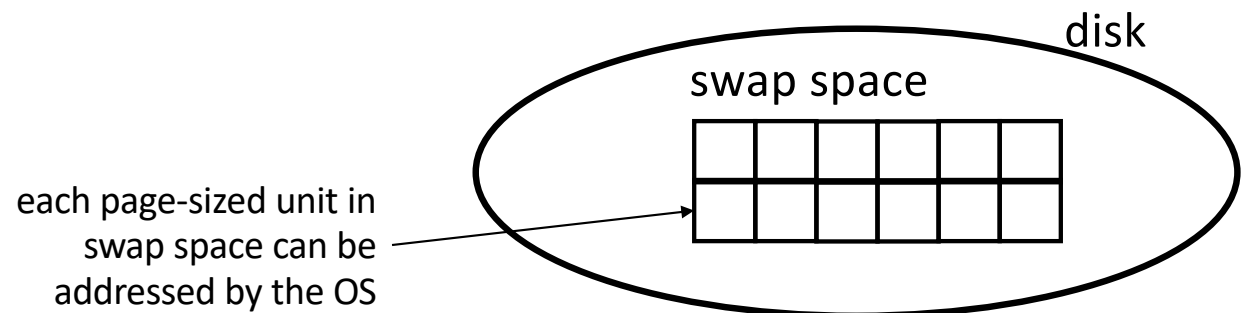


An Abstraction of Address Space

- Who is responsible for moving data?
- Application: **memory overlays**
 - Application in charge of moving data between memory and disk
 - e.g., calling a function needs to make sure the code is in memory!
- OS: **demand paging**
 - OS configures page table entries
 - Virtual page maps to physical memory or files in disk
 - Process sees an abstraction of address space
 - OS determines where the data is stored

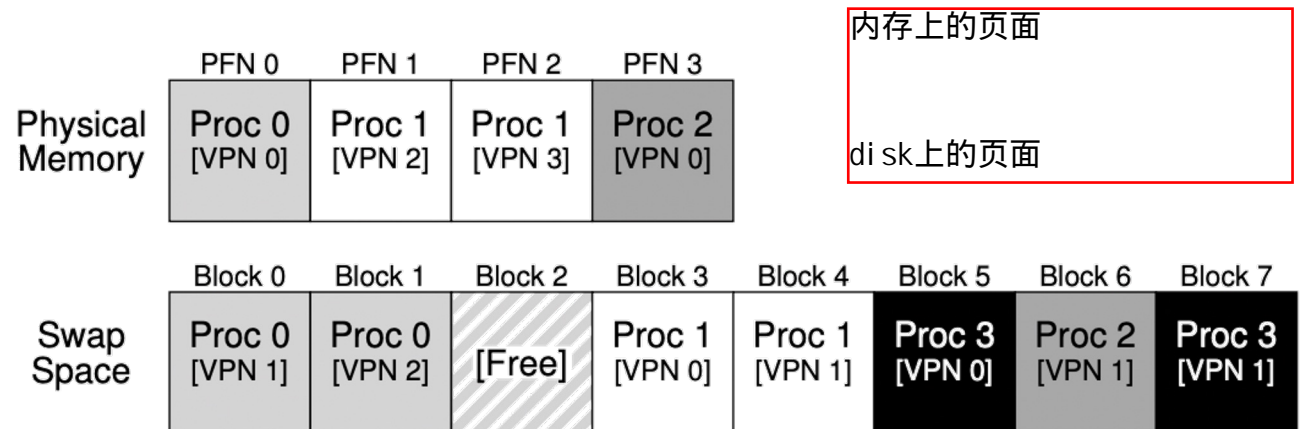
Swap Space

- **Swap space** is a partition or a file stored on the disk
 - OS swaps pages out of memory to it
 - OS swaps pages from it into memory
- Swap space conceptually divided into page-sized units
 - OS maintains a disk address of each page-sized unit



Swap Space Example

- 4-page physical memory and an 8-page swap space
 - Proc 0 has three virtual pages
 - Proc 1 has four virtual pages
 - Proc 2 and Proc 3 each has two virtual pages



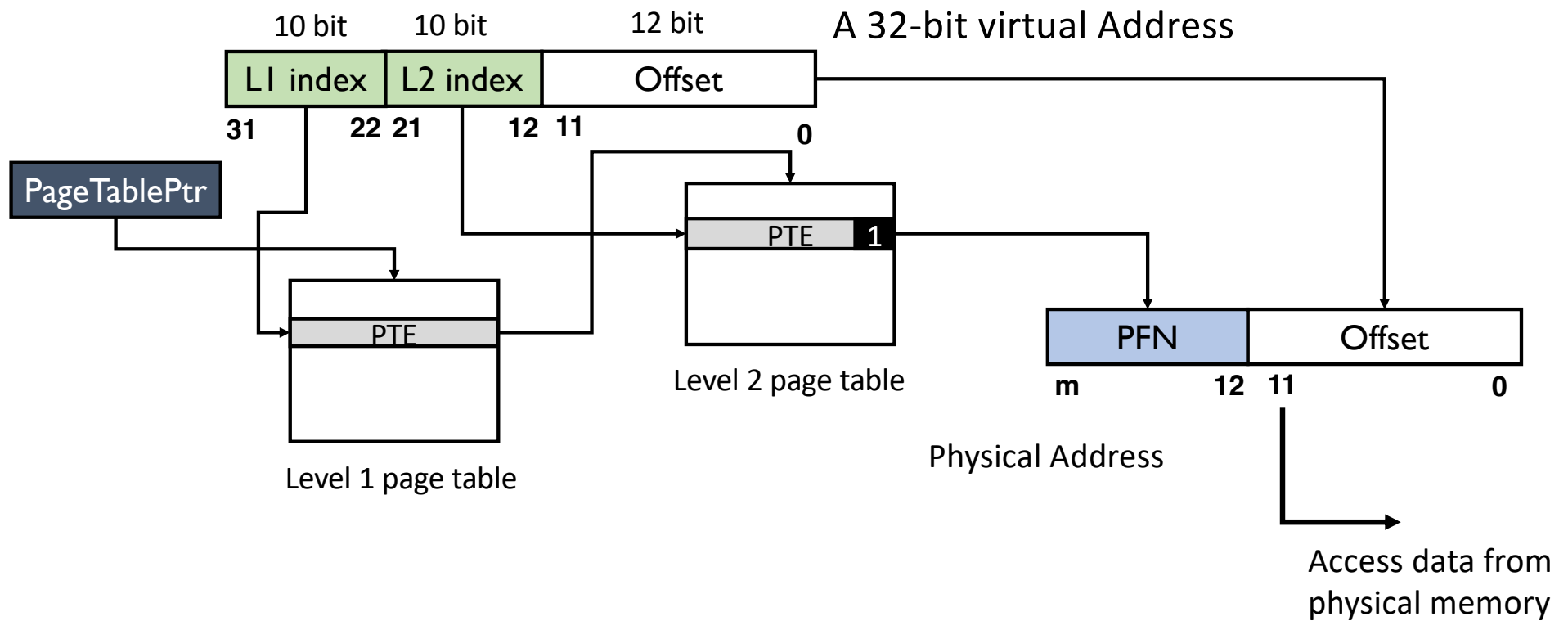
Demand Paging

- Load pages from disk to memory only as they are needed
 - Pages are loaded “on demand”
- Data transferred in the unit of pages
- Two possible on-disk locations
 - Swap space:
 - created by OS for temporary storage of pages on disk
 - e.g., pages for stack and heap
 - Program binary files:
 - The code pages from this binary are only loaded into memory when they are executed
 - Read-only, thus never write back

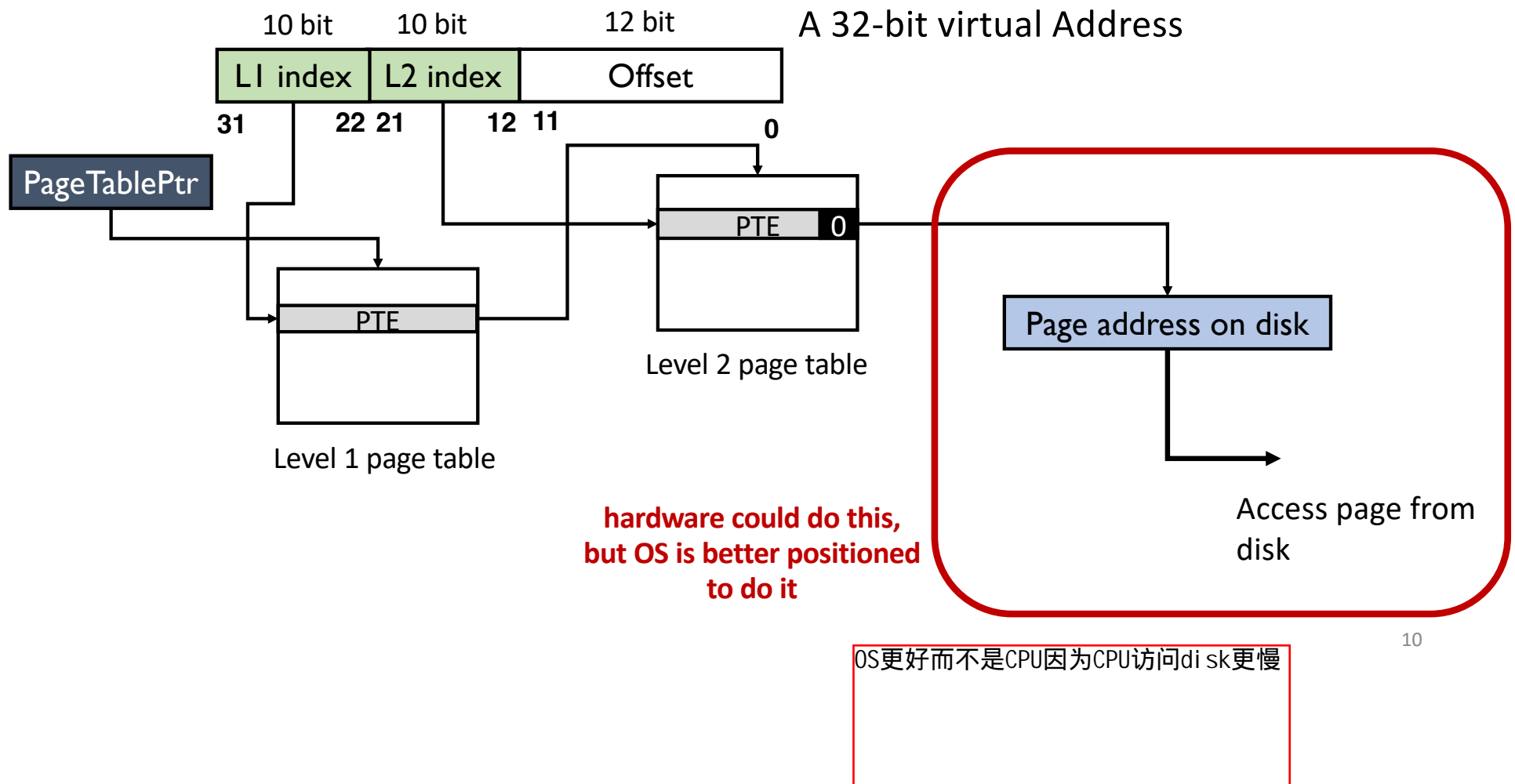
Physical Memory as a Cache

- Physical memory can be regarded as a cache of on-disk swap space
- Block size of the cache?
 - 1 page (4KB)
- Cache organization (direct-mapped, set-associative, fully-associative)?
 - Fully associative: any disk page maps to any page frame
- What is page replacement policy?
 - LRU, Random, FIFO
- What happens on a miss?
 - Go to lower level to fill page (i.e. disk)
- What happens on a write, write-through or write back?
 - write-back: changes are written back to disk when page is evicted

Present Bit



Present Bit



Page Faults

- Present bit = 0 raises a page fault exception
 - OS gets involved in address translation
- Page fault handler
 - (1) Find free page frame in physical memory
 - (2) Fetch page from disk and store it in physical memory
- After page fault
 - Return from page fault exception
 - CPU re-execute the instruction that accesses the virtual memory
 - No more page fault since present bit is set this time
 - TLB entry loaded from PTE

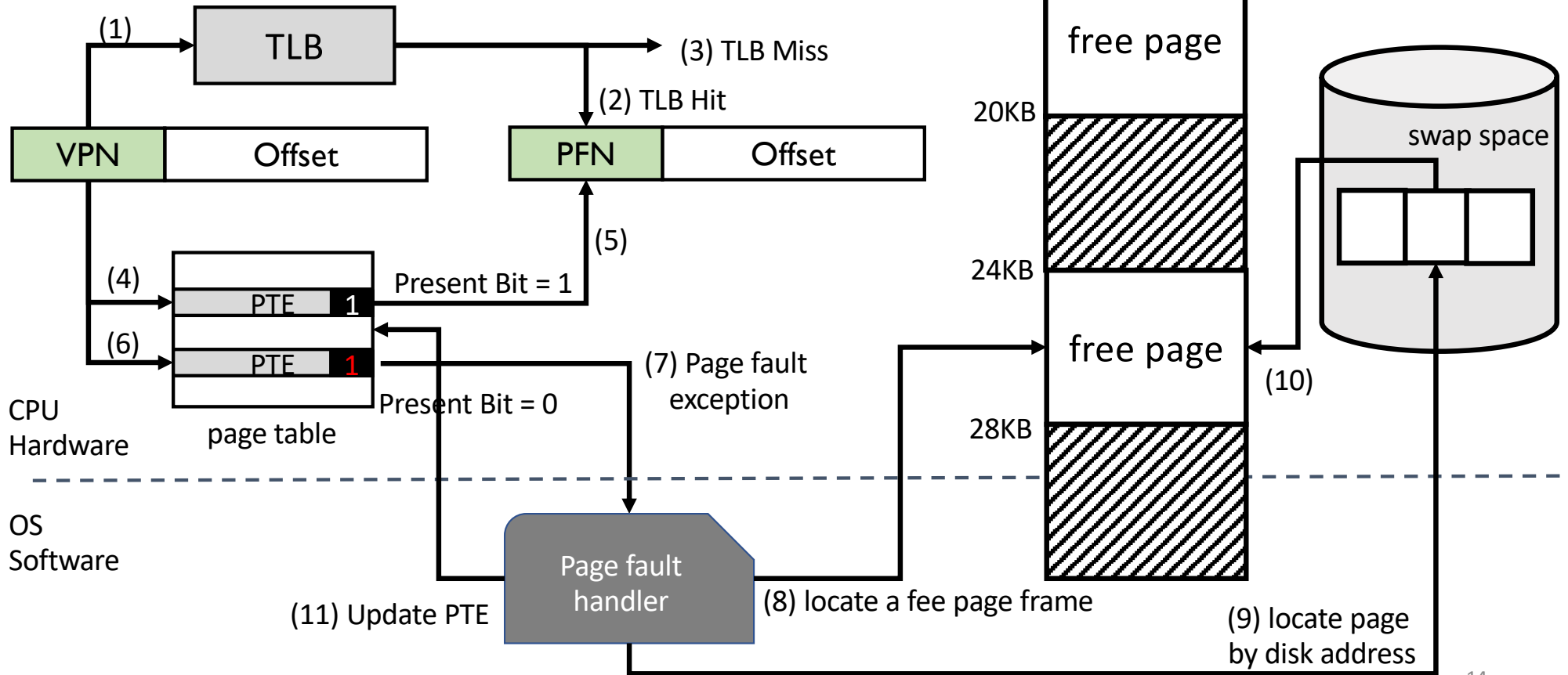
Page Faults (Cont'd)

- (1) Find free page frame in physical memory
 - Find one free page frame from a free-page list
 - If no free page, trigger **page replacement**
- **Page Replacement**
 - find a page frame to be replaced
 - Page replacement policy decides which one to replace
 - If page frame to be replaced is dirty, write it back to disk
 - Update all PTEs pointing to the page frame
 - Invalidate all TLB entries for these PTEs

Page Faults (Cont'd)

- (2) Fetch page from disk
 - Determine the faulting virtual address from register
 - Locate the disk address of the page in PTE (where PFN should be stored)
 - It is a very natural choice to make use of the space in PTE
 - Issues a request to disk to fetch the page into memory
 - Wait (could be a very long time, context switch!)
 - When I/O completes, update page table entry: PFN, present bit

Put It All Together



When to Trigger Page Replacement

- Proactive page replacement usually leads to better performance
 - Page replacement even though no one needs free page frames (yet)
 - Always reserve some free page frames in the system
- Swap daemon
 - background process for reclaiming page frames
 - Low watermark: a threshold to trigger swap daemon
 - High watermark: a threshold to stop reclaiming page frames

Page Replacement Policy

Effective Access Time

Hit Rate: 访问数据时，数据在内存里的概率

Miss Rate: 访问数据时，数据在disk上的概率

- $EAT = \text{Hit Rate} \times \text{Hit Time} + \text{Miss Rate} \times \text{Miss Penalty}$
- Example:
 - Memory access time = 200 nanoseconds
 - Average page-fault service time = 8 milliseconds
 - Suppose p = Probability of miss, $1-p$ = Probability of hit
 - Then, we can compute EAT as follows:

$$\begin{aligned} EAT &= (1-p) \times 200\text{ns} + p \times 8\text{ ms} \\ &= (1-p) \times 200\text{ns} + p \times 8,000,000\text{ns} \end{aligned}$$

Effective Access Time (Cont'd)

$$\begin{aligned} \text{EAT} &= (1-p) \times 200\text{ns} + p \times 8 \text{ ms} \\ &= (1-p) \times 200\text{ns} + p \times 8,000,000\text{ns} \end{aligned}$$

- If one access out of 1,000 causes a page fault, then EAT is about 8.2 μs :
 - This is a slowdown by a factor of 40!
- What if we want slowdown by less than 10%?
 - $200\text{ns} \times 1.1 < \text{EAT} \Rightarrow p < 2.5 \times 10^{-6}$
 - This is about 1 page fault in 400,000!

Types of Cache Misses: Three Cs

- Compulsory Misses:
 - Cold-start miss: pages that have never been fetched into memory before
 - Prefetching: loading them into memory before needed
- Capacity Misses:
 - Not enough memory: must somehow increase available memory size
 - One option: Increase amount of DRAM (not quick fix!)
 - Another option: If multiple processes in memory: adjust percentage of memory allocated to each one!
- Conflict Misses:
 - fully-associative cache (OS page cache) does not have conflict misses

Page Replacement Policies

- **Optimal** (also called MIN):
 - Replace page that will not be used for the longest time
 - Lead to minimum page faults in theory
- **FIFO** (First In, First Out)
 - Throw out oldest page first
 - May throw out heavily used pages instead of infrequently used
- **RANDOM**:
 - Pick random page for every replacement
 - Pretty unpredictable – makes it hard to make real-time guarantees

Replacement Policies (Con't)

- **Least Recently Used (LRU):**
 - Replace page that has not been used for the longest time
 - **Temporal locality of program**
 - If a page has not been used for a while, it is unlikely to be used in the near future
- **Least Frequently Used (LFU)**
 - Replace page that has not been accessed many times
 - **Spatial locality of program**
 - if a page has been accessed many times, perhaps it should not be replaced as it clearly has some value.

Example: Optimal (MIN)

前三个一定会page fault因为A、B、C，一开始对应的page都还没从disk加载到内存里

- Suppose we have 3 page frames, 4 virtual pages, and following reference string:
 - A B C A B D A D B C B

Ref:	A	B	C	A	B	D	A	D	B	C	B
Page:											
1	A									C	
2		B									
3			C			D					

- MIN: 5 faults
 - Where will D be brought in? Look for page not referenced farthest in future

Example: FIFO

- Suppose we have 3 page frames, 4 virtual pages, and following reference string:
 - A B C A B D A D B C B

Ref:	A	B	C	A	B	D	A	D	B	C	B
Page:											
1	A					D				C	
2		B					A				
3			C						B		

- FIFO: 7 faults
- When referencing D, replacing A is bad choice, since need A again right away

Example: LRU

- Suppose we have 3 page frames, 4 virtual pages, and following reference string:
 - A B C A B D A D B C B

Ref:	A	B	C	A	B	D	A	D	B	C	B
Page:											
1	A									C	
2		B									
3			C			D					

- LRU performs the same as Optimal

Is LRU Always Close to Optimal?

- Consider the following reference string: A B C D A B C D A B C D
- LRU performs as follows (the same as FIFO):

Ref:	A	B	C	D	A	B	C	D	A	B	C	D
Page:												
1	A			D			C			B		
2		B			A			D			C	
3			C			B			A			D

Is LRU Always Close to Optimal? (Cont'd)

- Consider the following: A B C D A B C D A B C D
- MIN performs better:

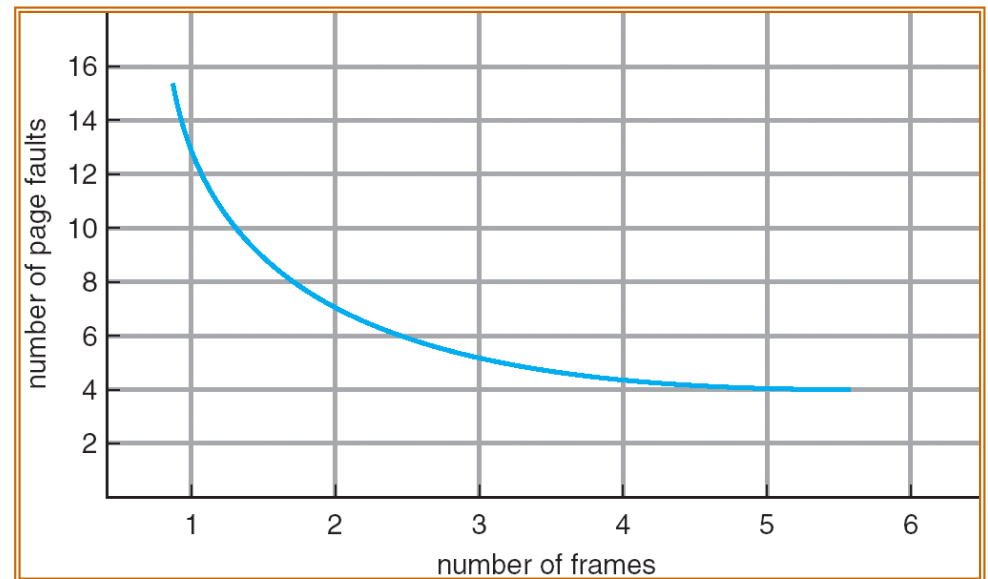
Ref:	A	B	C	D	A	B	C	D	A	B	C	D
Page:												
1	A									B		
2		B					C					
3			C	D								

Quiz

- Consider the following reference string with three page frames:
 - 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1
- What are the number of page faults with the following policy:
 - Optimal (MIN)
 - LRU
 - FIFO

Bélády's Anomaly

- One desirable property:
When you add memory the miss rate drops
 - Yes for LRU and MIN
 - Not necessarily for FIFO!
- Bélády's anomaly
 - For FIFO, more page frames may lead to more page faults!



Bélády's Anomaly Example

- Page replacement with 3 page frames

Ref: Page:	A	B	C	D	A	B	E	A	B	C	D	E
1	A			D			E					
2		B			A					C		
3			C			B					D	

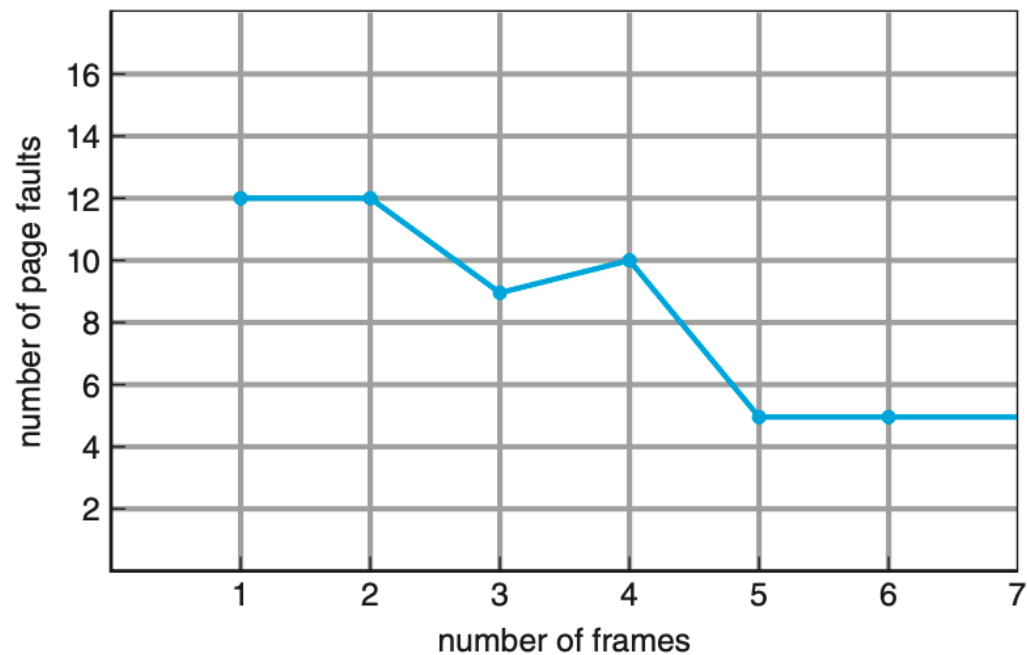
- Page replacement with 4 page frames

Ref: Page:	A	B	C	D	A	B	E	A	B	C	D	E
1	A						E				D	
2		B						A				E
3			C						B			
4				D						C		

Page Fault Curve

- Page fault curve for FIFO on reference string
 - 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

How do you plot
a chart like this?



LRU Implementation

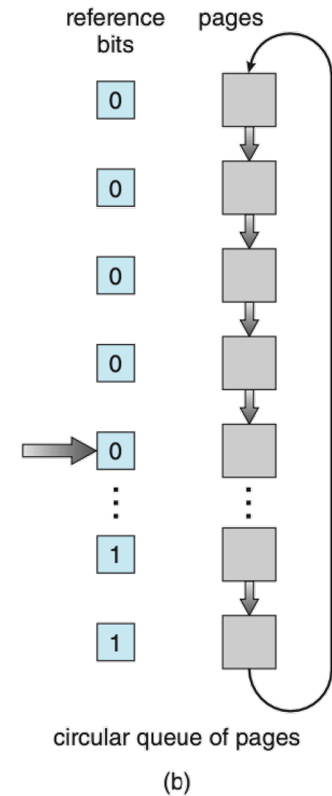
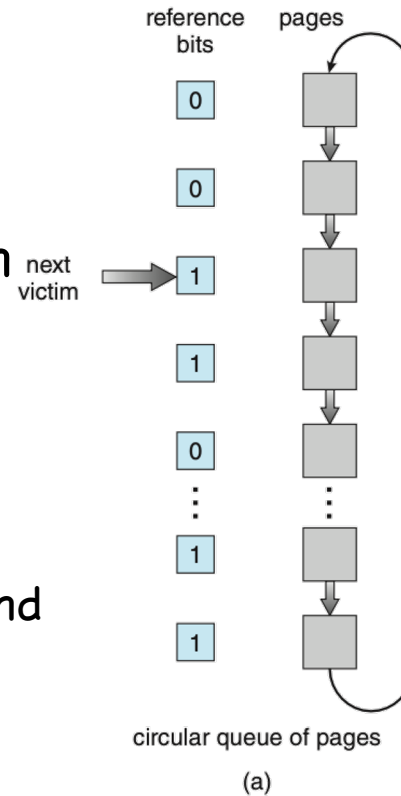
- Hardware support is necessary
 - Update a data structure in OS upon every memory access
 - E.g., a timestamp counter for each page frame
- Overhead
 - One additional memory write for each memory access
 - TLB hit does not save the extra memory access
 - Scan the entire memory to find the LRU one
 - 4GB physical memory has 1 million page frames
 - sorting is time consuming

LRU Approximation with Reference Bit

- Reference bit
 - One reference bit per page frame
 - All bits are cleared to 0 initially
 - The first time a page is referenced, the reference bit is set **by CPU**
 - Can be integrate with page table walk
 - The order of page accesses approximated by two clusters: **used** and **unused** pages
- Examples:
 - Clock algorithm (also called second-chance algorithm)
 - Enhanced clock algorithm with dirty bits

Clock Algorithm

- Arrange physical pages in a circular list
- CPU sets reference bit to 1 upon first access
- OS maintains a pointer
 - When a replacement occur, check reference bit of the current page
 - If 1: the page has been accessed recently, clear the bit (set to 0) and move to the next page
 - If 0: the page has not been accessed recently, good candidate for replacement, stop



Clock Algorithm with Dirty Bit

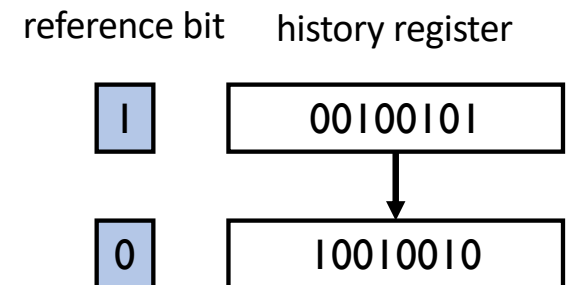
- Enhance clock algorithm with a dirty bit
 - dirty bit = 1: the page has recently been modified
- CPU sets dirty bit to 1 upon **write** access
- When a replacement occurs, OS checks (ref bit, dirty bit), and selects a candidate page in decreasing order
 - (0, 0) neither recently used nor modified — best page to replace
 - (0, 1) not recently used but modified — not quite as good, because the page will need to be written out before replacement
 - (1, 0) recently used but clean — probably will be used again soon
 - (1, 1) recently used and modified — probably will be used again soon, and the page will need to be written out to secondary storage before it can be replaced

LRU Approximation with Reference Bit and Counter

- Each physical page frame is associated with one reference bit and a counter
 - Reference bit indicate recent access
 - set by CPU hardware, cleared by OS
 - Counter records history of accesses
 - Maintained by OS
- Examples
 - Additional-reference-bits algorithm
 - N^{th} -chance clock algorithm

Additional-reference-bits Algorithm

- 8-bit history register associated with each page frame
- Timer interrupt every 100ms
 - reference bit shifts to highest bit in the history register
 - other bits shift right and discard the lowest bit
 - 00000000 unused page in 800ms
- Compare history register as unsigned integer
 - Larger value more recently used
 - 11000100 > 01110111
- Approximate LRU with more bits and more frequent interrupts



Nth-chance Clock Algorithm

- All page frames arranged in a circular list and each page frame is associated with a reference bit and a counter
- CPU hardware sets reference bit upon memory accesses
- OS checks the reference bit of the page pointed to by the clock hand
 - 1 → clear reference bit and the counter
 - 0 → increment counter; if count=N, replace page
- How do we pick N?
 - Large N? Better approximation to LRU
 - If $N \sim 1K$, really good approximation
 - Small N? More efficient
 - Otherwise might have to look a long way to find free page

Page Frame Allocation

Allocation of Page Frames

- How do we allocate memory among different processes?
 - Does every process get the same fraction of memory? Different fractions?
 - Should we completely swap some processes out of memory?
- Minimum number of pages per process
 - Depends on the computer architecture
 - How many pages would one instruction use at one time
 - x86 only allows data movement between memory and register and no indirect reference
 - needs at least one instruction page, one data page, and some page table pages
- Maximum number of pages per process
 - Depends on available physical memory

Global versus Local Allocation

- Global replacement
 - Process selects replacement frame from all page frames
 - One process can take a frame from another process
- Local replacement
 - Each process selects from only its own set of allocated frames

Allocation Algorithms

- Equal allocation:
 - Every process gets same amount of memory
 - Example: 100 frames, 5 processes → process gets 20 frames
- Proportional allocation
 - Number of page frames proportional to the size of process
 s_i = size of process p_i and m = total number of frame
 a_i = allocation for $p_i = m \times \frac{s_i}{\sum s_j}$
- Priority Allocation:
 - Number of page frames proportional to the priority of process
 - Possible behavior: If process p_i generates a page fault, select for replacement a frame from a process with lower priority number

Thrashing

- The memory demands of the set of running processes simply exceeds the available physical memory
- Early OS
 - Working set: the pages used actively of a process
 - Reduce the # of process so their working set fit into memory
- Modern OS
 - Out-of-memory killer when memory is oversubscribed
 - May need a reboot

Thank you!

