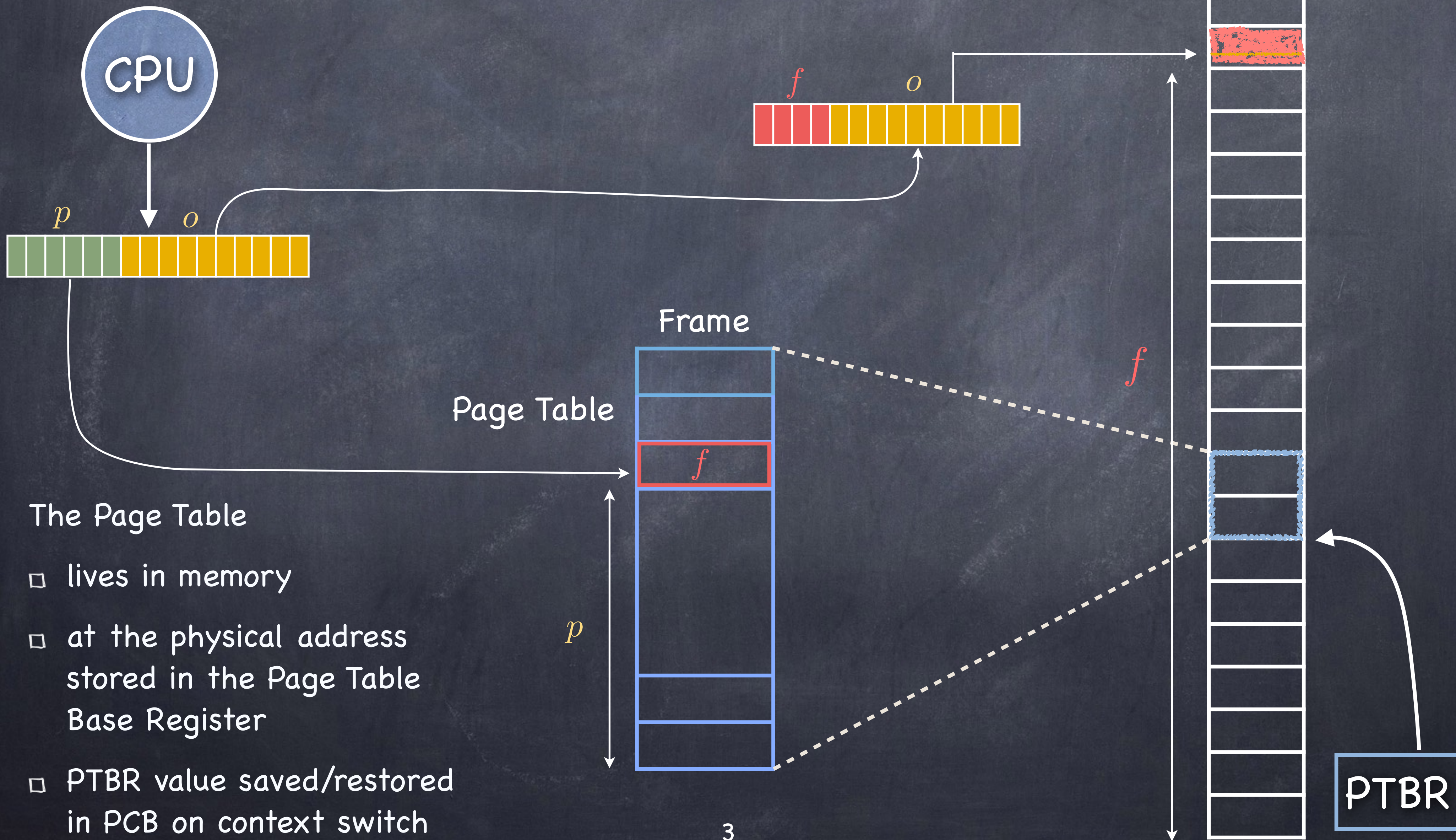
Lecture 15:
Memory Management

Page Tables, and page replacement algorithms

Recall: Paging

- Allocate VA & PA memory in chunks of the same, fixed size (pages and frames, respectively)
- Adjacent pages in VA need not map to contiguous frames in PA!
  - free frames can be tracked using a simple bitmap
    - 0011111001111011110000 one bit/frame
  - no more external fragmentation!
  - possible internal fragmentation
    - when memory needs are not a multiple of a page
    - typical size of page/frame: 4KB to 16KB

Recall: Basic Paging



The Page Table

- lives in memory
- at the physical address stored in the Page Table Base Register
- PTBR value saved/restored in PCB on context switch

Recall: Basic goals in paging

- **Minimize Storage overhead**
  - data structure overhead (the Page Table itself)
  - fragmentation
    - How large should a page be?
- **Fast Address translation**
  - We need "fast" lookups on page table
- **Efficient sharing of physical memory**
  - By multiple processes

Paging—first attempt

- Divide virtual address space into fixed-sized pages (e.g., 4KB)
- Linear array: one entry for each page, maps it to a frame
- Storage overheads:
  - Number of entries * size of entry
  - Number of entries = number of pages = (VAS size / page size)
  - Size of entry ~ $\log_2$ (PAS Size / frame size) + control bits
  - 32-bit virtual address space, 4GB physical memory, 4KB pages = 4MB
  - 64-bit virtual address space, 4GB physical memory, 4KB pages = 16 PB

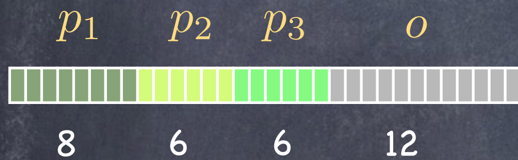
Paging—second attempt

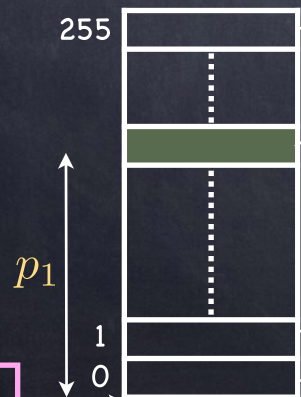
- Divide virtual address space into fixed-sized pages (e.g., 4KB)
- Multi-level page tables: store a tree
  - But only those nodes/edges that are required to map pages to frames

Multi-level Paging

Structure virtual address space as a tree

Virtual address





PTBR

63

p_2

0

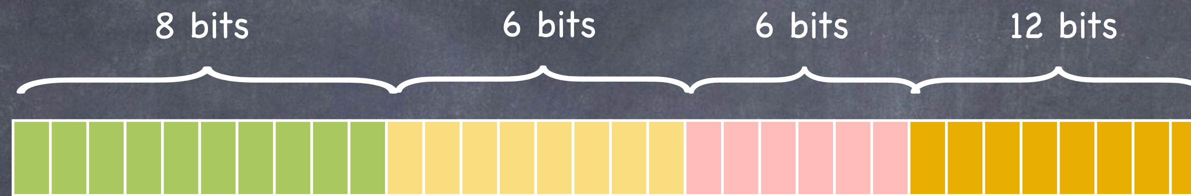
63

v

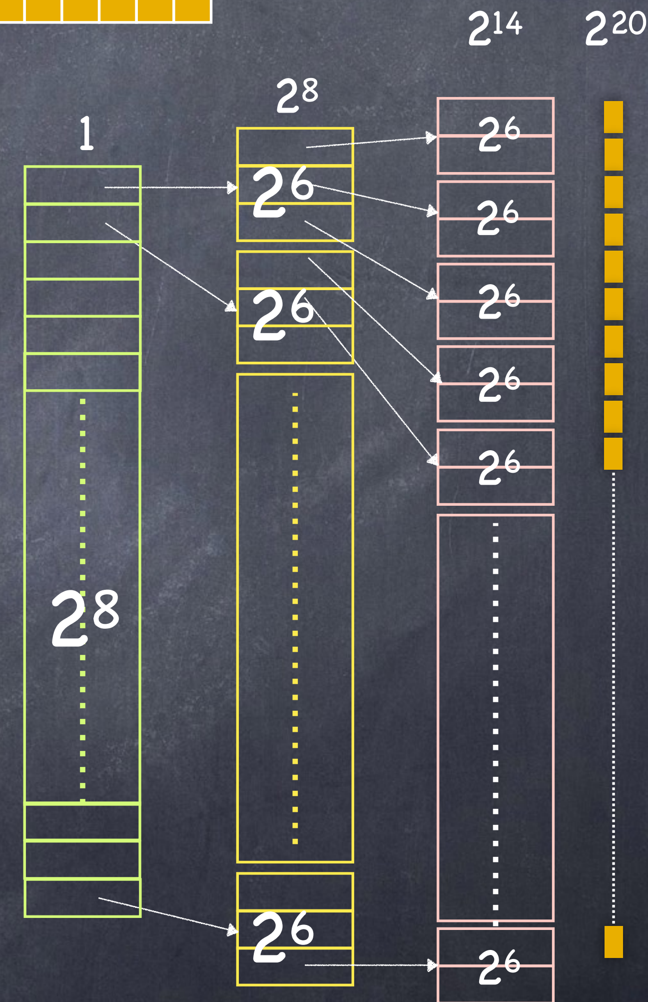
p_3

0

32-bit, 4GB, 4KB pages Example



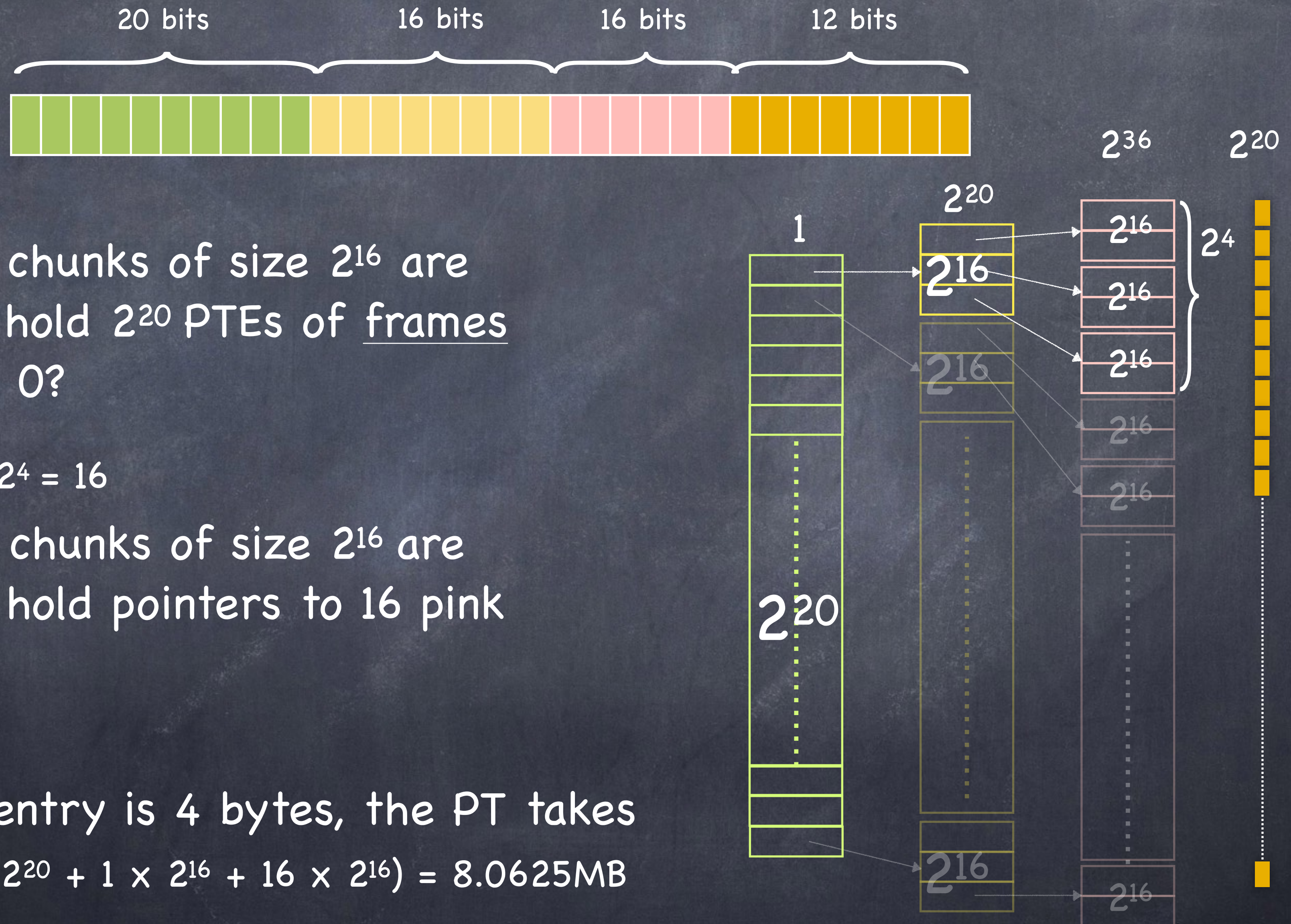
- If we use a tree…
  - Last 12 bits index into the page
    - page size = 2^{12}
    - #pages = 2^{20} (since total memory = 4GB)
  - Next 6 bits index into last-level of the tree
    - #entries in each chunk = 2^6
    - #chunks = 2^{14} (to account for 2^{20} pages)
  - Next 6 bits index into second-last-level of the tree
    - #entries in each chunk = 2^6
    - #chunks = 2^8 (to account for 2^{14} last-level chunks)
  - Next 8 bits index into first-level of the tree
    - #entries in each chunk = 2^8
    - #chunks = 1 (to account for 2^8 second-last-level chunks)



64-bit, 4GB, 4KB pages Example

- If we use a tree...
  - Last 12 bits index into the page
    - page size = 2^{12}
    - #pages = 2^{20} (since total memory = 4GB)
  - Next 16 bits index into last-level of the tree
    - #entries in each chunk = 2^{16}
    - #chunks needed = 2^4 (to account for 2^{20} pages)
  - Next 16 bits index into second-last-level of the tree
    - #entries in each chunk = 2^{16}
    - #chunks needed = 1 (to account for 2^4 last-level chunk)
  - Next 20 bits index into first-level of the tree
    - #entries in each chunk = 2^{20}
    - #chunks = 1 (to account for 1 second-last-level chunk)

64-bit, 4GB, 4KB pages Example



- How many chunks of size 2^{16} are needed to hold 2^{20} PTEs of frames starting at 0?
  - $2^{20}/2^{16} = 2^4 = 16$
- How many chunks of size 2^{16} are needed to hold pointers to 16 pink chunks?
  - 1
- So, if each entry is 4 bytes, the PT takes
  - $4 * (1 \times 2^{20} + 1 \times 2^{16} + 16 \times 2^{16}) = 8.0625\text{MB}$
  - Can be further reduced a bit

Questions?

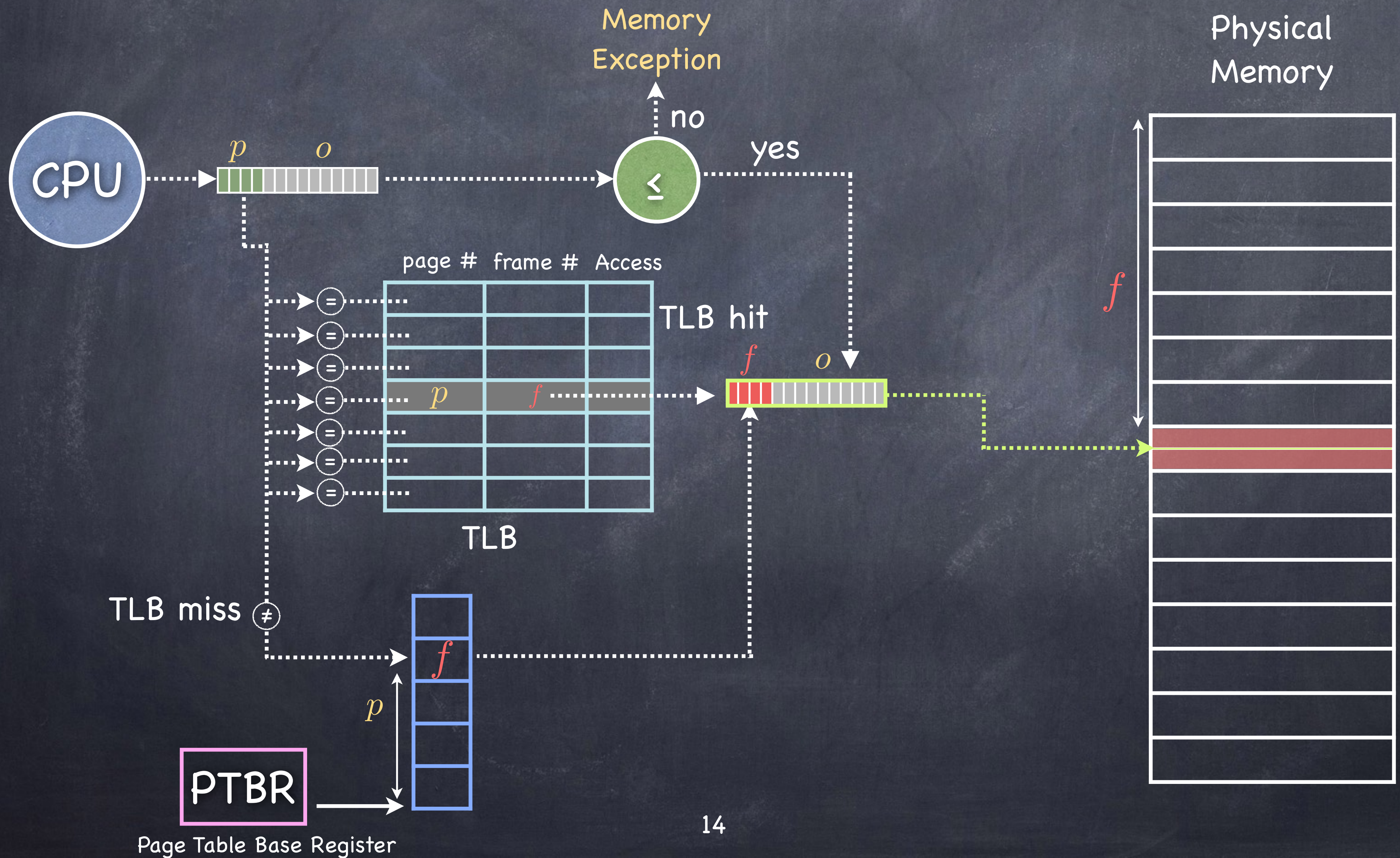
Where are we?

- Storage overheads
  - Minimized! Using multi-level page tables.
- How about address translation time?
  - Every new level of paging
    - reduces the memory overhead for computing the mapping function…
    - … but increases the time necessary to perform the mapping function

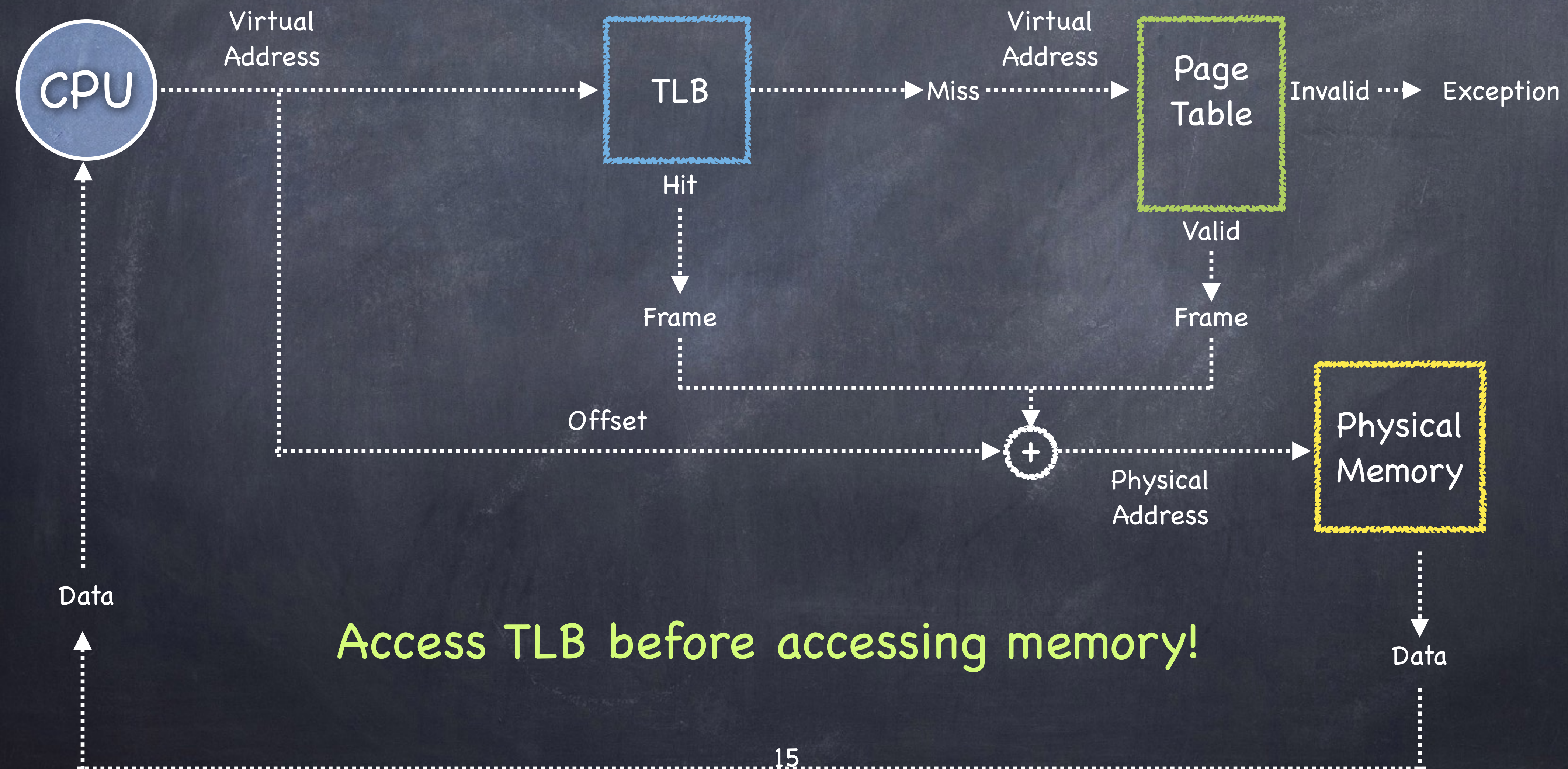
Caching!

- Keep the results of recent virtual address to physical address translations in a structure called Translation Lookaside Buffer (TLB)

Speeding things up: The TLB



Address Translation with TLB



Access TLB before accessing memory!

TLB Hit and Miss

- The TLB is small; it cannot hold all PTEs
  - it can be fast only if it is small!
- Some translations will inevitably miss the TLB
- Must access memory to find the appropriate PTE
  - called walking the page table
  - incurs large performance penalty

Handling TLB Misses

- Hardware-managed (e.g., x86)
  - The hardware does the page walk
  - Hardware fetches PTE and inserts it in TLB
    - If TLB is full, must replace another TLB entry
  - Done transparently to system software
- Software-managed (e.g., MIPS)
  - Hardware raises an exception
  - OS does the page walk, fetches PTE, and inserts evicts entries in TLB

Tradeoffs, Tradeoffs...

- Hardware-managed TLB
  - \+ No exception on TLB miss. Instruction just stalls
  - \+ No extra instruction/data brought into the cache
  - \- OS has no flexibility in deciding Page Table organization
  - \- OS has no flexibility in TLB entry replacement policy
- Software-managed TLB
  - \+ OS can define Page Table organization
  - \+ More flexible TLB entry replacement policies
  - \- Slower: exception causes to flush pipeline; execute handler; pollute cache

TLB Consistency - I

- On context switch
  - VAs of old process should no longer be valid
  - Change PTBR — but what about the TLB?

TLB Consistency - I

- On context switch
  - VAs of old process should no longer be valid
  - Change PTBR — but what about the TLB?
    - Option 1: Flush the TLB
    - Option 2: Add pid tag to each TLB entry

	PID	VirtualPage	PageFrame	Access
TLB Entry	1	0x0053	0x0012	R/W

Ignore entries with wrong PIDs

TLB Consistency - II

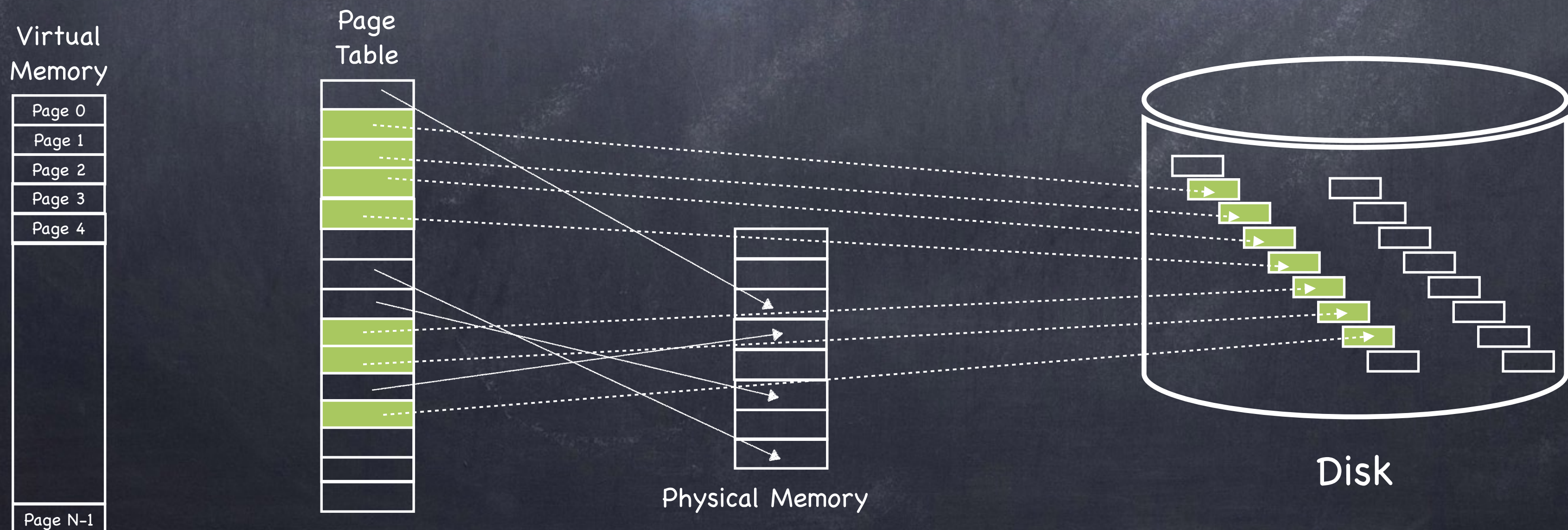
- What if OS changes permissions on page?
  - If permissions are reduced, OS must ensure affected TLB entries are purged
  - If permissions are expanded, no problem
    - new permissions will cause an exception and OS will restore consistency

Virtual memory

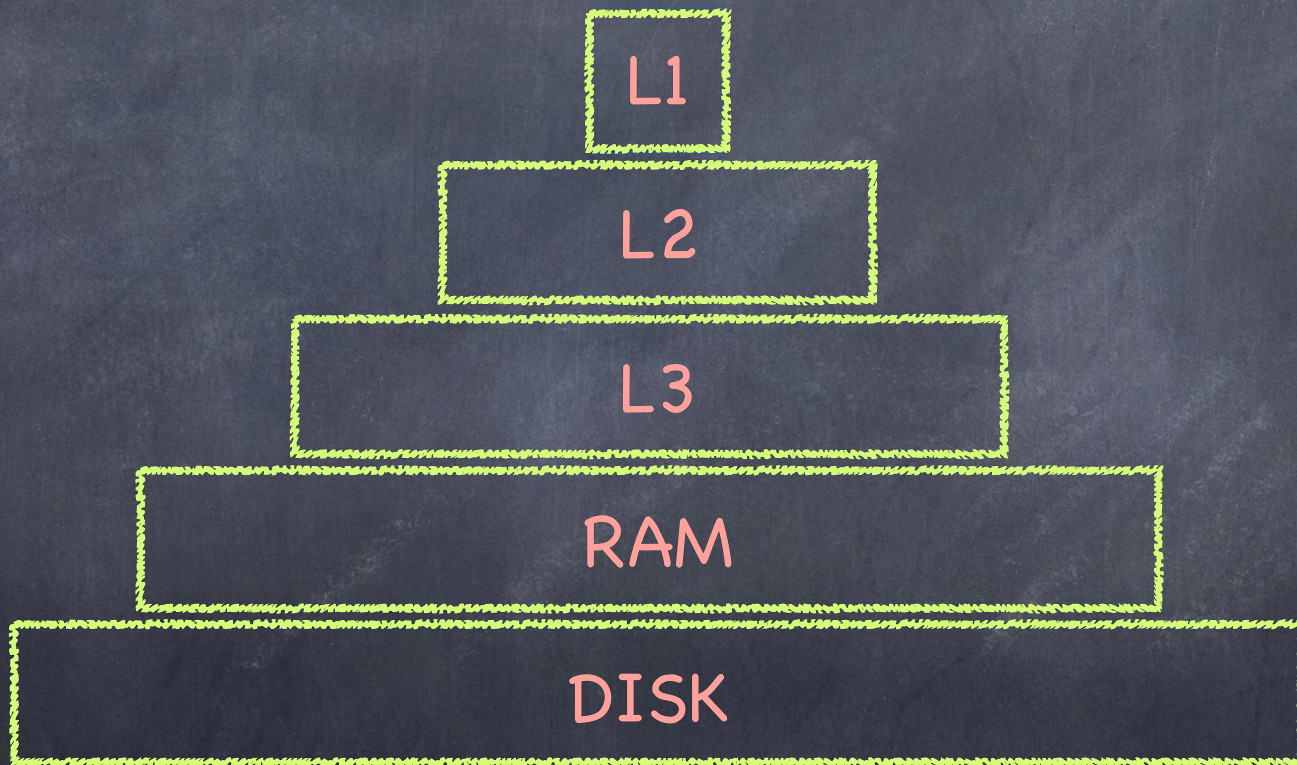
- Consider a server with 4GB memory.
- What if a process has 16GB requirement?
- What if we have two concurrently running processes
    - each having 4GB requirements?

Virtual Memory

- Each process has the illusion of a large address space
  - 2^x bytes for x-bit addressing
- However, physical memory is usually much smaller
  - and we want to run multiple processes concurrently
- How do we give this illusion to multiple processes?
  - Virtual Memory: back every memory address with a file on disk



Processes execute from disk!



RAM is just another layer of cache!

A Virtual Page can be...

- Mapped (present bit set in PTE)
  - to a physical frame, with certain r/w/x permissions
    - (may trigger Page Fault)
- Not mapped (present bit not set in PTE)
  - in some physical frame, but not currently mapped
  - or still in the original program file
  - or needing to be zero-filled (heap, BSS, stack)
  - or on backing store (paged or swapped out)
    - (Page Fault)
  - or not part of one of the processes' segment
    - Segmentation Fault!

Handling a Page Fault

- Identify page and reason
  - access inconsistent with segment access rights
    - terminate process
  - access a page currently on disk
    - does frame with the code/data already exist?
      - if not, allocate a frame and load page in
  - access of zero-initialized data (BSS) or stack
    - allocate a frame, initialize all bytes to zero

When a page must be brought in...

- Find a free frame
  - evict a page if there are no free frames
- Issue disk request to fetch data for page
- Move "current process" to disk queue
- Context switch to new process
- Update PTE when disk completes
  - frame number, present bit, RWX bits, etc.
- Move "current process" to ready queue

When a page must be swapped out…

- Find all page table entries that refer to old page
  - Frame might be shared
- Set each page table entry to not present (invalid)
- Remove any TLB entries
  - “TLB Shootdown”: in multiprocessors, TLB entry must be eliminated from the TLB of all processors
- Write page back to disk, if needed
  - Dirty bit in PTE indicates need

Demand Paging

MIPS Style

1. TLB Miss
2. Exception to kernel
3. Page Table walk
4. Page fault (present bit not set in Page Table)
5. Convert VA to file offset
6. Allocate page frame (evict page if needed)
7. Initiate disk block read into page frame
8. Disk interrupt when DMA completes
9. Mark page as present
10. Update TLB
11. Resume process at faulting instruction
12. TLB hit
13. Execute instruction

Software handling page fault

Demand Paging:

x86 Style

1. TLB Miss
2. Page Table walk
3. Page fault (page not present in Page Table)
4. Exception to kernel
5. Convert VA to file offset
6. Allocate page frame (evict page if needed)
7. Initiate disk block read into page frame
8. Disk interrupt when DMA completes
9. Mark page as present
10. Resume process at faulting instruction
11. TLB miss
12. Page Table walk - success!
13. TLB updated
14. Execute instruction

Software handling page fault

Page Replacement

- When physical memory is full, we need to choose a "victim" to evict
- Local vs Global replacement
  - Local: victim chosen from frames of process experiencing page fault
    - fixed allocation of frames per process
  - Global: victim chosen from frames allocated to any process
    - variable allocation of frames per process
- Goal: minimizing number of page faults

Page Replacement Algorithms

- Random: Pick any page to eject at random
  - Used mainly for comparison
- FIFO: The page brought in earliest is evicted
  - Ignores usage
- LRU: Evict page not been used the longest
  - Assumes past is good predictor of the future
- MRU: Evict most recently used page
  - Good for data accessed only once, e.g., a movie
- LFU: Evict least frequently used page
- OPT: Belady's algorithm

How do we pick a victim?

- We want:
  - low page fault-rate
  - page faults as inexpensive as possible
- We need:
  - a way to compare the relative performance of different page replacement algorithms
  - some absolute notion of what a "good" page replacement algorithm should accomplish

Comparing Page Replacement Algorithms

- Record a trace of the pages accessed by a process
  - E.g. 3,1,4,2,5,2,1,2,3,4 (or c,a,d,b,e,b,a,b,c,b)
- Simulate behavior of page replacement algorithm on trace
- Record number of page faults generated

Optimal Page Replacement

- Replace page needed furthest in future

Time		0	1	2	3	4	5	6	7	8	9	10	11	12
Trace		a												
Page Frames	0													
	1													
	2													
Faults		X												

Process can use 3 frames
(3 pages in memory)

Optimal Page Replacement

- Replace page needed furthest in future

Time		0	1	2	3	4	5	6	7	8	9	10	11	12
Trace		a	b											
Page Frames	0		a											
	1													
	2													
Faults		X	X											

Process can use 3 frames
(3 pages in memory)

Optimal Page Replacement

- Replace page needed furthest in future

Time		0	1	2	3	4	5	6	7	8	9	10	11	12
Trace		a	b	c										
Page Frames	0		a	a										
	1			b										
	2													
Faults		X	X	X										

Process can use 3 frames
(3 pages in memory)

Optimal Page Replacement

- Replace page needed furthest in future

Time		0	1	2	3	4	5	6	7	8	9	10	11	12
Trace		a	b	c	d	a	b	e	a	b	c	d	e	
Page Frames	0		a	a	a									
	1			b	b									
	2				c									
Faults		X	X	X	X									

Process can use 3 frames
(3 pages in memory)

Optimal Page Replacement

- Replace page needed furthest in future

Time		0	1	2	3	4	5	6	7	8	9	10	11	12
Trace		a	b	c	d	a								
Page Frames	0		a	a	a	a								
	1			b	b	b								
	2				c	d								
Faults		✗	✗	✗	✗	✓								

Process can use 3 frames (3 pages in memory)

Optimal Page Replacement

- Replace page needed furthest in future

Time		0	1	2	3	4	5	6	7	8	9	10	11	12
Trace		a	b	c	d	a	b							
Page Frames	0		a	a	a	a	a							
	1			b	b	b	b							
	2				c	d	d							
Faults		✗	✗	✗	✗	✓	✓							

Process can use 3 frames
(3 pages in memory)

Optimal Page Replacement

- Replace page needed furthest in future

Time		0	1	2	3	4	5	6	7	8	9	10	11	12
Trace		a	b	c	d	a	b	e	a	b	c	d	e	
Page Frames	0		a	a	a	a	a	a						
	1			b	b	b	b	b						
	2				c	d	d	d						
Faults		✗	✗	✗	✗	✓	✓	✗						

Process can use 3 frames
(3 pages in memory)

Optimal Page Replacement

- Replace page needed furthest in future

Time		0	1	2	3	4	5	6	7	8	9	10	11	12
Trace		a	b	c	d	a	b	e	a					
Page Frames	0		a	a	a	a	a	a	a					
	1			b	b	b	b	b	b					
	2				c	d	d	d	e					
Faults		✗	✗	✗	✗	✓	✓	✗	✓					

Process can use 3 frames (3 pages in memory)

Optimal Page Replacement

- Replace page needed furthest in future

		0	1	2	3	4	5	6	7	8	9	10	11	12
Time		0	1	2	3	4	5	6	7	8	9	10	11	12
Trace		a	b	c	d	a	b	e	a	b				
Page Frames	0		a	a	a	a	a	a	a	a				
	1			b	b	b	b	b	b	b				
	2				c	d	d	d	e	e				
Faults		✗	✗	✗	✗	✓	✓	✗	✓	✓				

Process can use 3 frames
(3 pages in memory)

Optimal Page Replacement

- Replace page needed furthest in future

Time		0	1	2	3	4	5	6	7	8	9	10	11	12
Trace		a	b	c	d	a	b	e	a	b	c	d	e	
Page Frames	0		a	a	a	a	a	a	a	a	a			
	1			b	b	b	b	b	b	b	b			
	2				c	d	d	d	e	e	e			
Faults		✗	✗	✗	✗	✓	✓	✗	✓	✓	✗			

Process can use 3 frames
(3 pages in memory)

Optimal Page Replacement

- Replace page needed furthest in future

Time		0	1	2	3	4	5	6	7	8	9	10	11	12
Trace		a	b	c	d	a	b	e	a	b	c	d	e	
Page Frames	0		a	a	a	a	a	a	a	a	a	c		
	1			b	b	b	b	b	b	b	b	b		
	2				c	d	d	d	e	e	e	e		
Faults		✗	✗	✗	✗	✓	✓	✗	✓	✓	✗	✗		

Process can use 3 frames
(3 pages in memory)

Optimal Page Replacement

- Replace page needed furthest in future

Time		0	1	2	3	4	5	6	7	8	9	10	11	12
Trace		a	b	c	d	a	b	e	a	b	c	d	e	
Page Frames	0		a	a	a	a	a	a	a	a	a	c	c	
	1			b	b	b	b	b	b	b	b	b	d	
	2				c	d	d	d	e	e	e	e	e	
Faults		✗	✗	✗	✗	✓	✓	✗	✓	✓	✗	✗	✓	

Process can use 3 frames
(3 pages in memory)

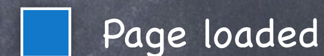
Optimal Page Replacement

- Replace page needed furthest in future

Time		0	1	2	3	4	5	6	7	8	9	10	11	12
Trace		a	b	c	d	a	b	e	a	b	c	d	e	
Page Frames	0		a	a	a	a	a	a	a	a	a	c	c	c
	1			b	b	b	b	b	b	b	b	b	d	d
	2				c	d	d	d	e	e	e	e	e	e
Faults		✗	✗	✗	✗	✓	✓	✗	✓	✓	✗	✗	✓	

7 page faults

Process can use 3 frames (3 pages in memory)


Page loaded

Time		0	1	2	3	4	5	6	7	8	9	10	11	12
Trace		a	b	c	d	a	b	e	a	b	c	d	e	
Page Frames	0		a	a	a	a	a	a	a	a	a	a	d	d
	1			b	b	b	b	b	b	b	b	b	a	e
	2				c	c	c	c	c	c	c	c	b	b
	3					d	d	d	e	e	e	e	e	c
Faults		✗	✗	✗	✗	✓	✓	✗	✓	✓	✓	✗	✓	

6 page faults

Process can use 4 frames (4 pages in memory)

FIFO Replacement

- Replace pages in the order they come into memory

Time		0	1	2	3	4	5	6	7	8	9	10	11	12
Trace		a	b	c	d	a	b	e	a	b	c	d	e	
Page Frames	0		a	a	a	d	d	d	e	e	e	e	e	e
	1			b	b	b	a	a	a	a	a	c	c	c
	2				c	c	c	b	b	b	b	b	d	d
Faults		✗	✗	✗	✗	✗	✗	✗	✓	✓	✗	✗	✓	

Process can use 3 frames
(3 pages in memory)

9 page faults

FIFO Replacement

- Replace pages in the order they come into memory

Time		0	1	2	3	4	5	6	7	8	9	10	11	12
Trace		a	b	c	d	a	b	e	a	b	c	d	e	
Page Frames	0		a	a	a	a	a	a	e	e	e	e	d	d
	1			b	b	b	b	b	b	a	a	a	a	e
	2				c	c	c	c	c	c	b	b	b	b
	3					d	d	d	d	d	d	c	c	c
Faults		X	X	X	X	✓	✓	X	X	X	X	X	X	

Process can use 4 frames
(4 pages in memory)

10 page faults

More frames —> more page faults?

Belady's Anomaly

Locality of Reference

- If a process access a memory location, then it is likely that
  - the same memory location is going to be accessed again in the near future (temporal locality)
  - nearby memory locations are going to be accessed in the future (spatial locality)
- 90% of the execution of a program is sequential
- Most iterative constructs consist of a relatively small number of instructions

LRU: Least Recently Used

- Replace page not referenced for the longest time

Time		0	1	2	3	4	5	6	7	8	9	10	11	12
Trace		a	b	c	d	a	b							
Page Frames	0		a	a	a	a	a	a						
	1			b	b	b	b	b						
	2				c	c	c	c						
	3					d	d	d						
Faults		✗	✗	✗	✗	✓	✓							

Process can use 4 frames
(4 pages in memory)

LRU: Least Recently Used

- Replace page not referenced for the longest time

Time		0	1	2	3	4	5	6	7	8	9	10	11	12
Trace		a	b	c	d	a	b	e						
Page Frames	0		a	a	a	a	a	a						
	1			b	b	b	b	b						
	2				c	c	c	c						
	3					d	d	d						
Faults		✗	✗	✗	✗	✓	✓	✗						

Process can use 4 frames
(4 pages in memory)

LRU: Least Recently Used

- Replace page not referenced for the longest time

Time		0	1	2	3	4	5	6	7	8	9	10	11	12
Trace		a	b	c	d	a	b	e						
Page Frames	0		a	a	a	a	a	a	a					
	1			b	b	b	b	b	b					
	2				c	c	c	c	e					
	3					d	d	d	d					
Faults		✗	✗	✗	✗	✓	✓	✗						

Process can use 4 frames
(4 pages in memory)

LRU: Least Recently Used

- Replace page not referenced for the longest time

Time		0	1	2	3	4	5	6	7	8	9	10	11	12
Trace		a	b	c	d	a	b	e	a	b	c	d	e	
Page Frames	0		a	a	a	a	a	a	a	a	a	a	a	e
	1			b	b	b	b	b	b	b	b	b	b	b
	2				c	c	c	c	e	e	e	e	d	d
	3					d	d	d	d	d	d	c	c	c
Faults		✗	✗	✗	✗	✓	✓	✗	✓	✓	✗	✗	✗	

8 page faults

Process can use 4 frames
(4 pages in memory)

Implementing LRU

- On reference: timestamp each page
- On eviction: scan for oldest page
- Problems:
  - Large page lists
  - Timestamps are costly
- Solution: approximate LRU
  - after all, LRU is already an approximation! (of OPT)
  - Next lecture