

Week 6 - Recap

Pamela Delgado

March 27, 2018

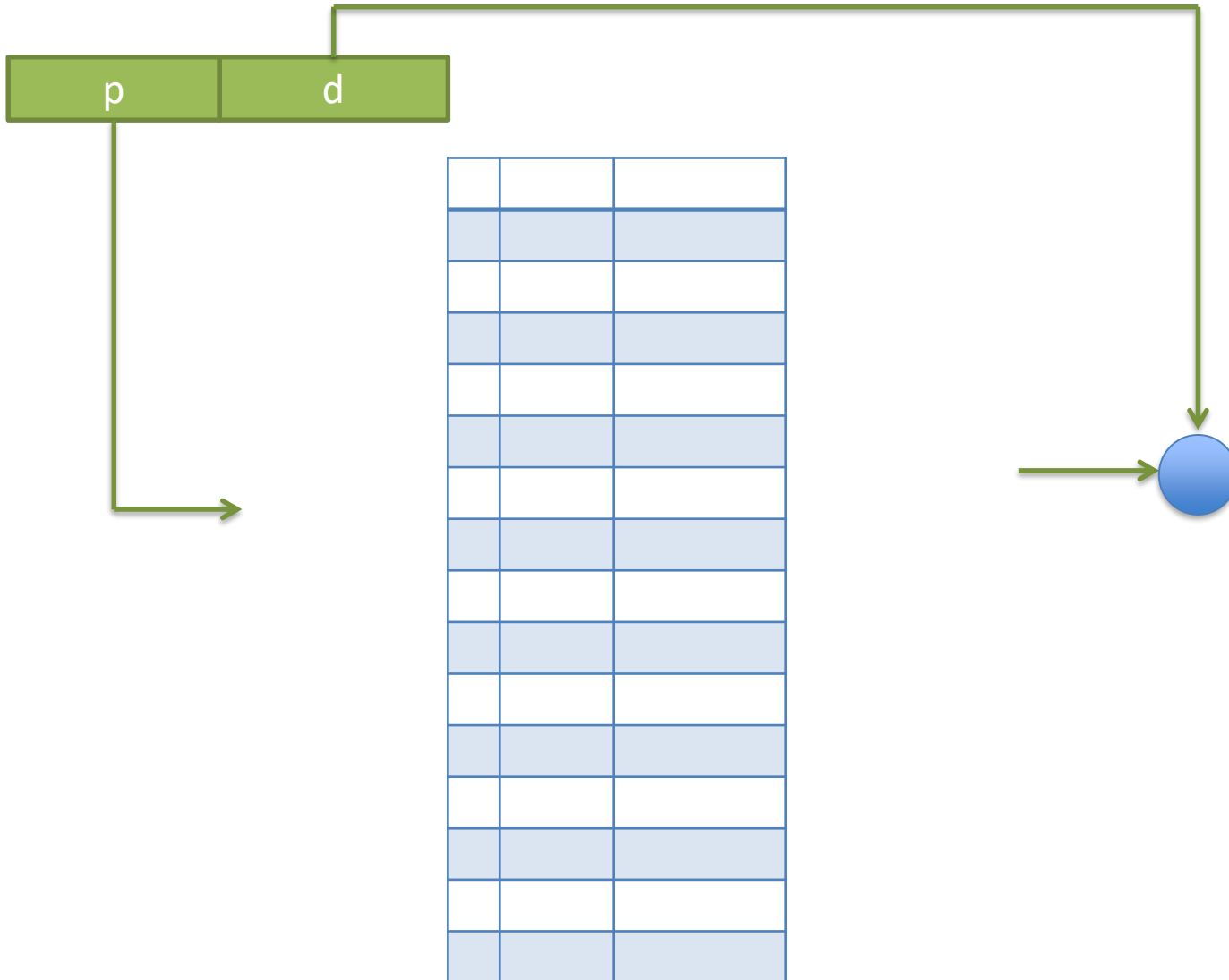
Large Sparse Address Spaces

- Hierarchical page tables
- Top-level page table for entire address space
- Lower-level page tables for subsets
- If subset not used, no need for page table
- Result: less space used for page tables

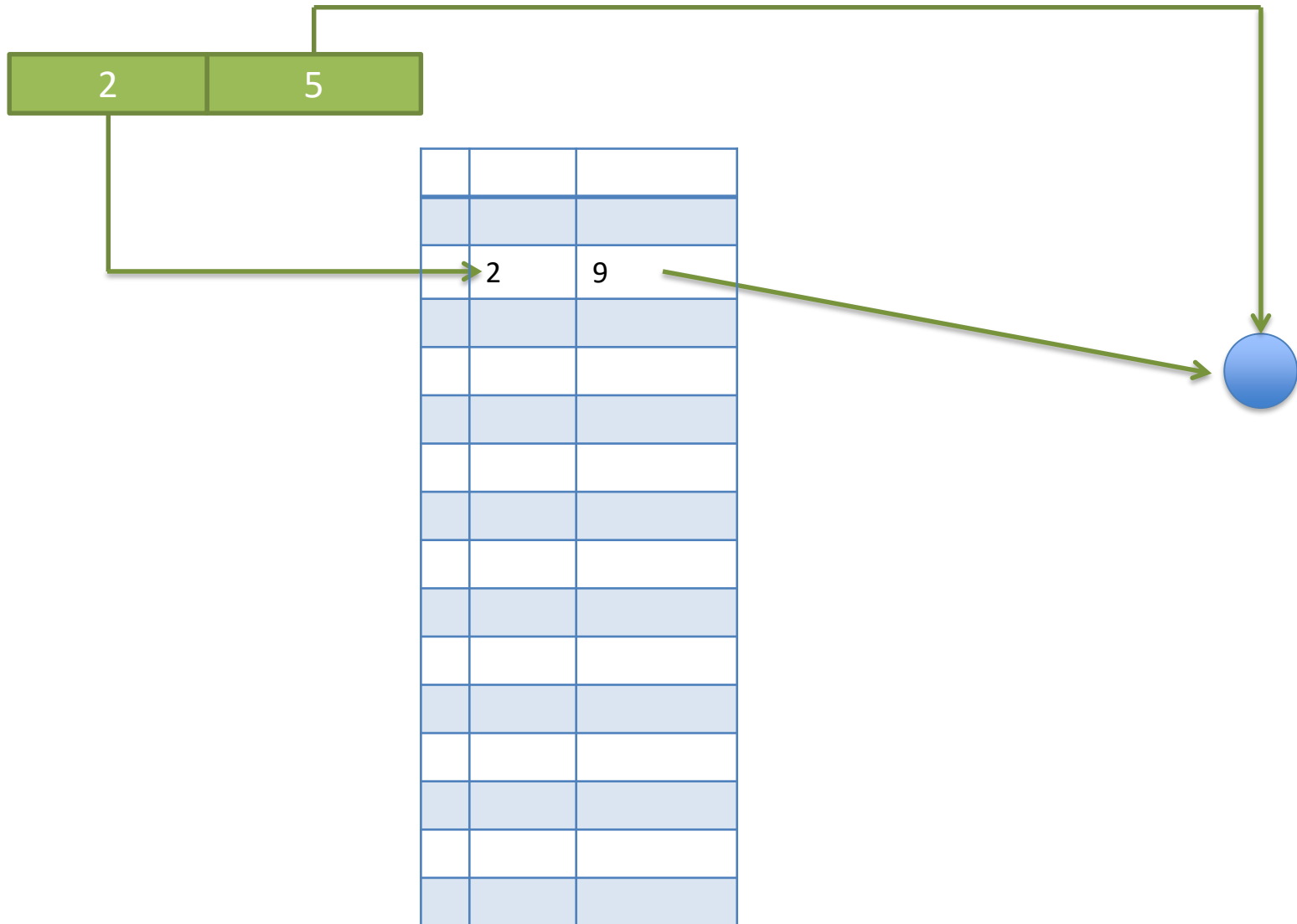
One-level Page Table

[illegible]

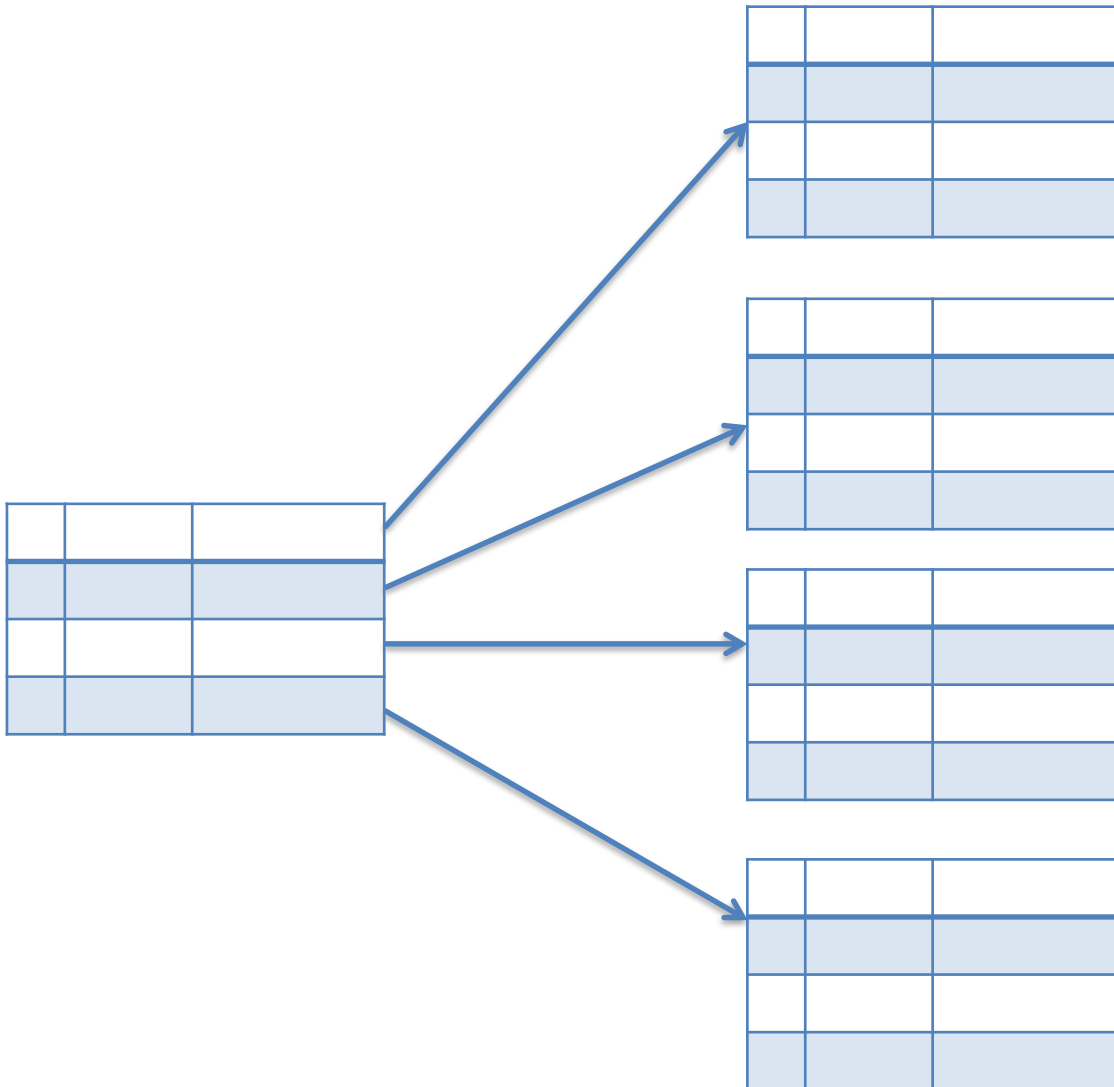
Address Translation



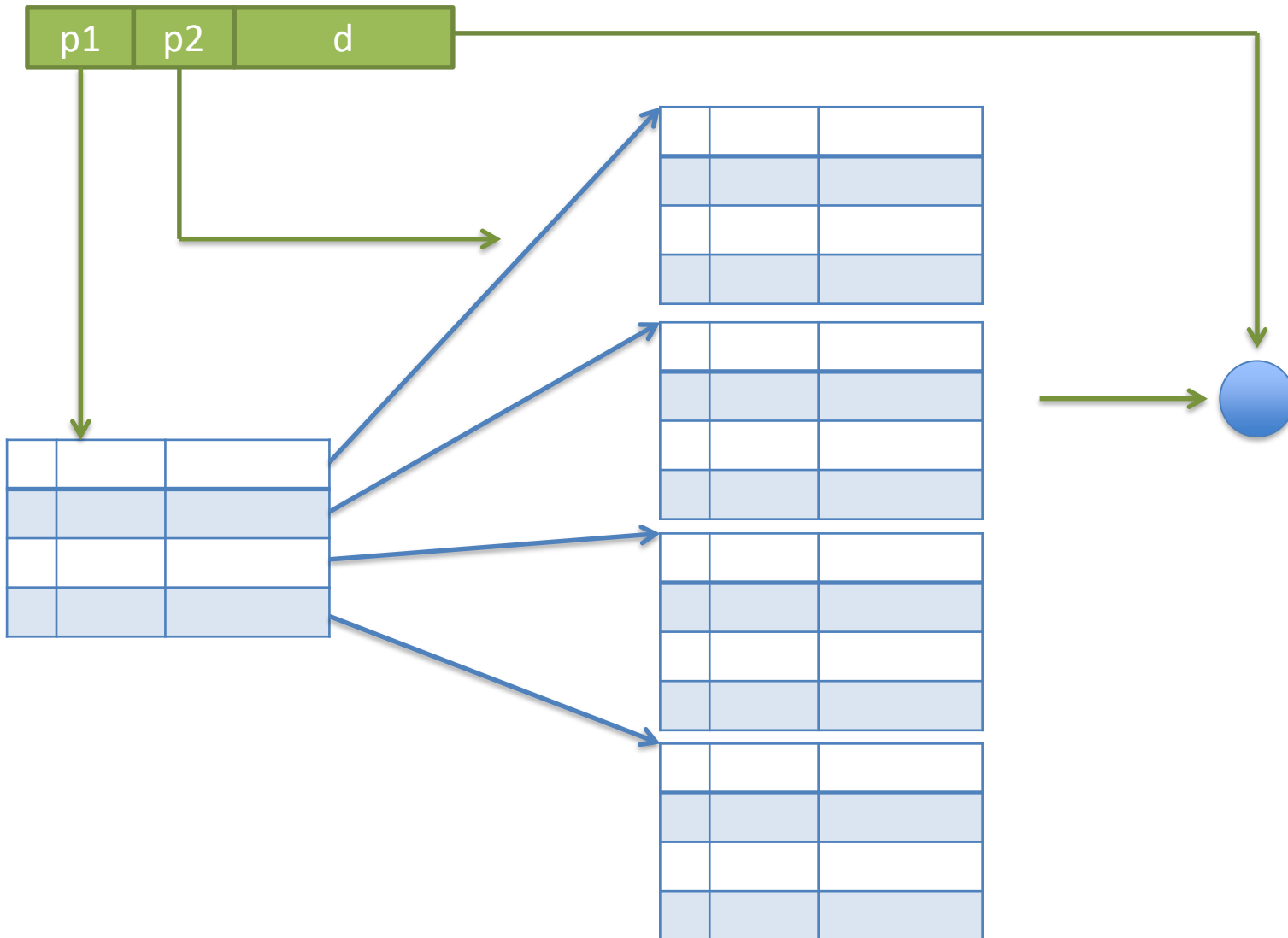
Address Translation Example



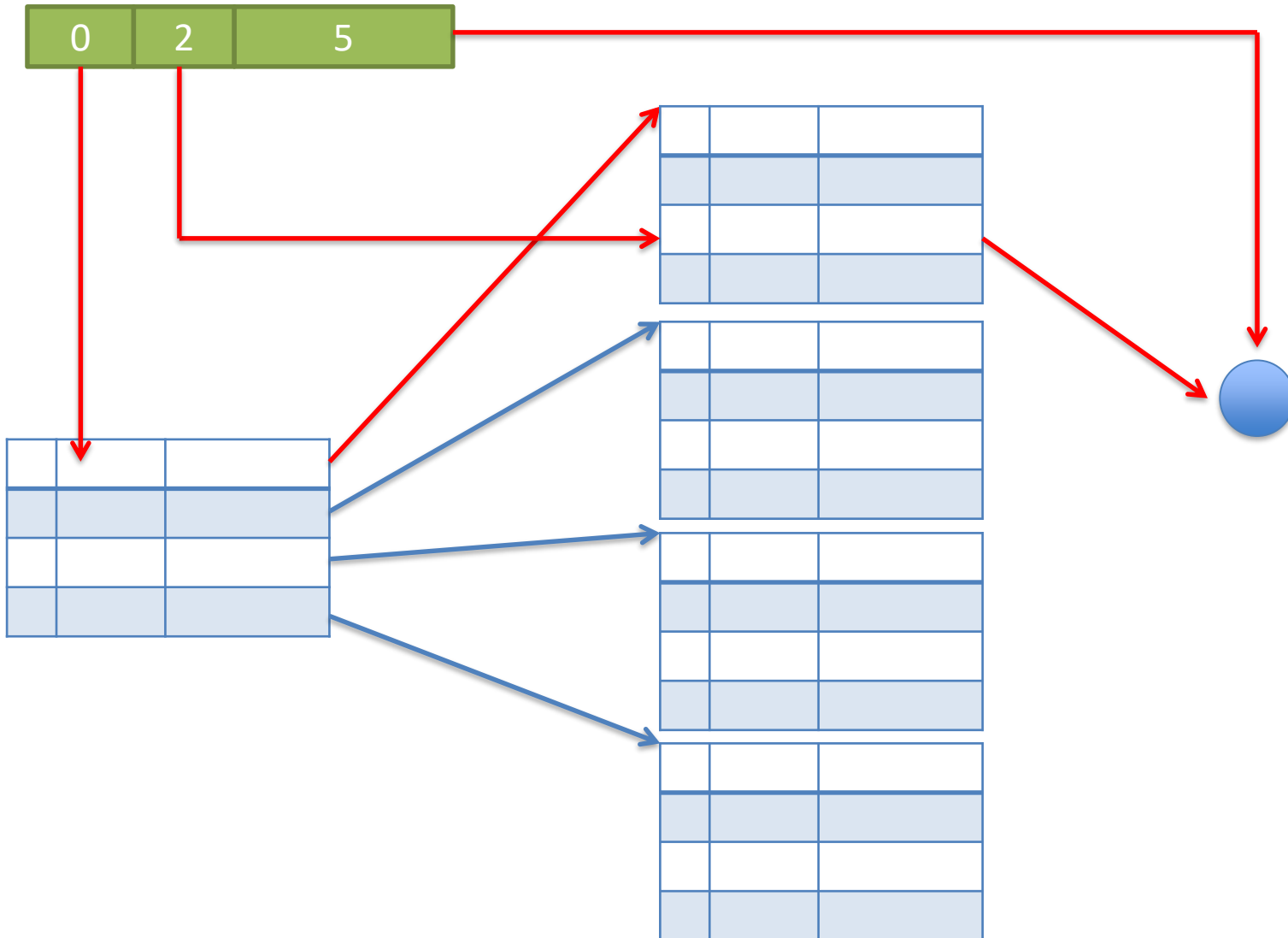
Two-level Page Table



Address translation



Address translation example



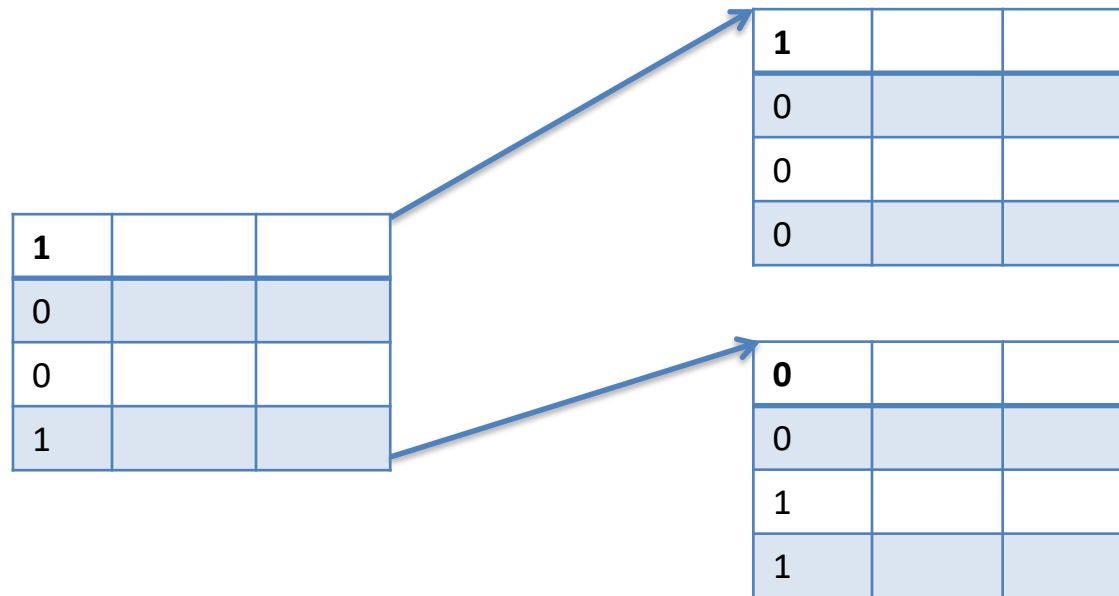
Sparse Address Space



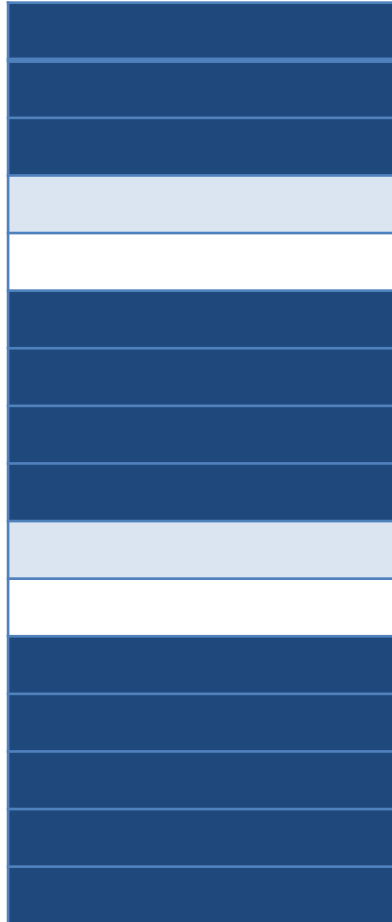
One-level Page Table for Sparse Address Space

1		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
1		
1		

Two-level Page Table for Sparse Address Space



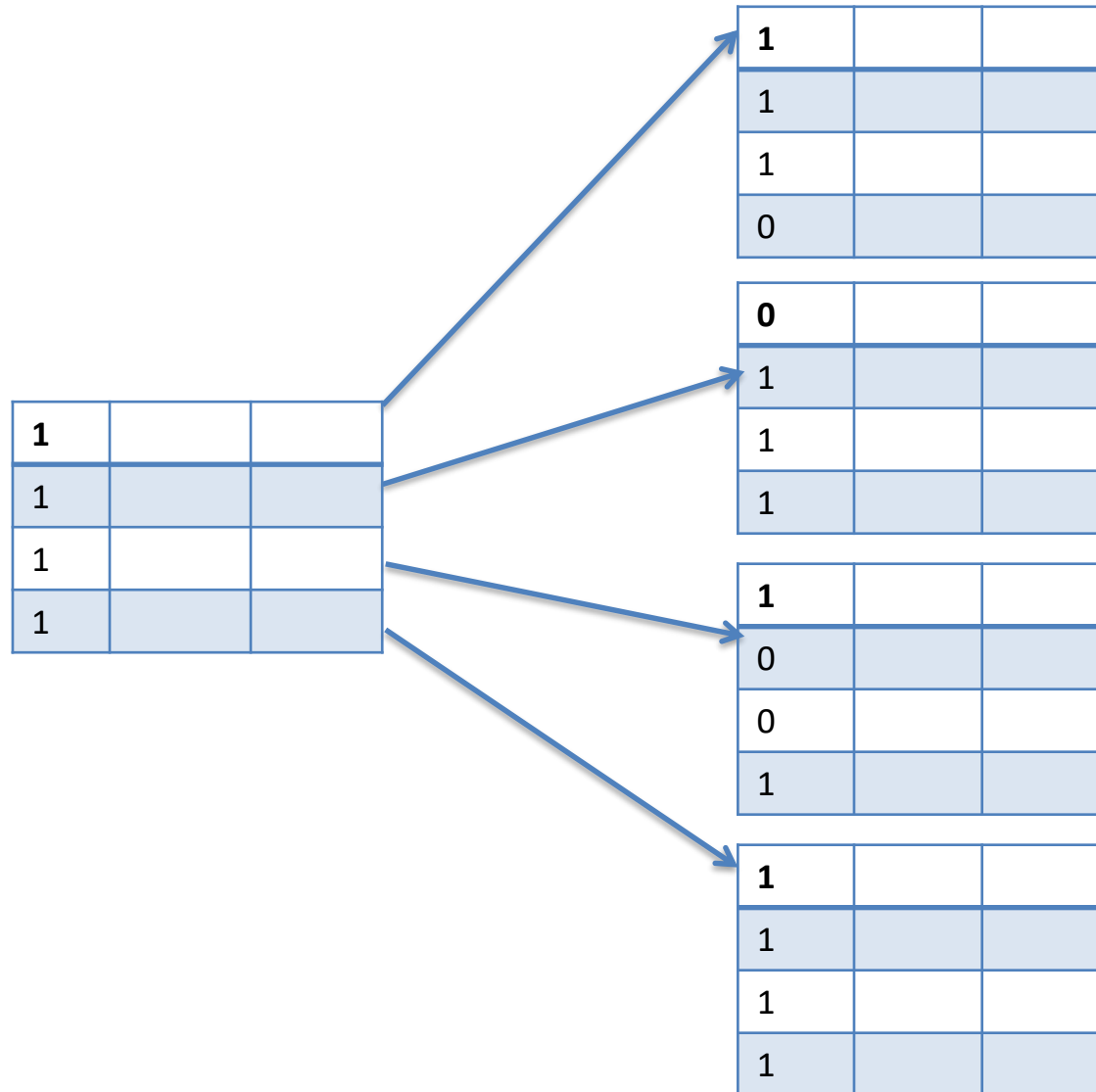
Dense Address Space



One-level Page Table for Dense Address Space

1		
1		
1		
0		
0		
1		
1		
1		
1		
0		
0		
1		
1		
1		
1		
1		

Two-level Page Table for Dense Address Space



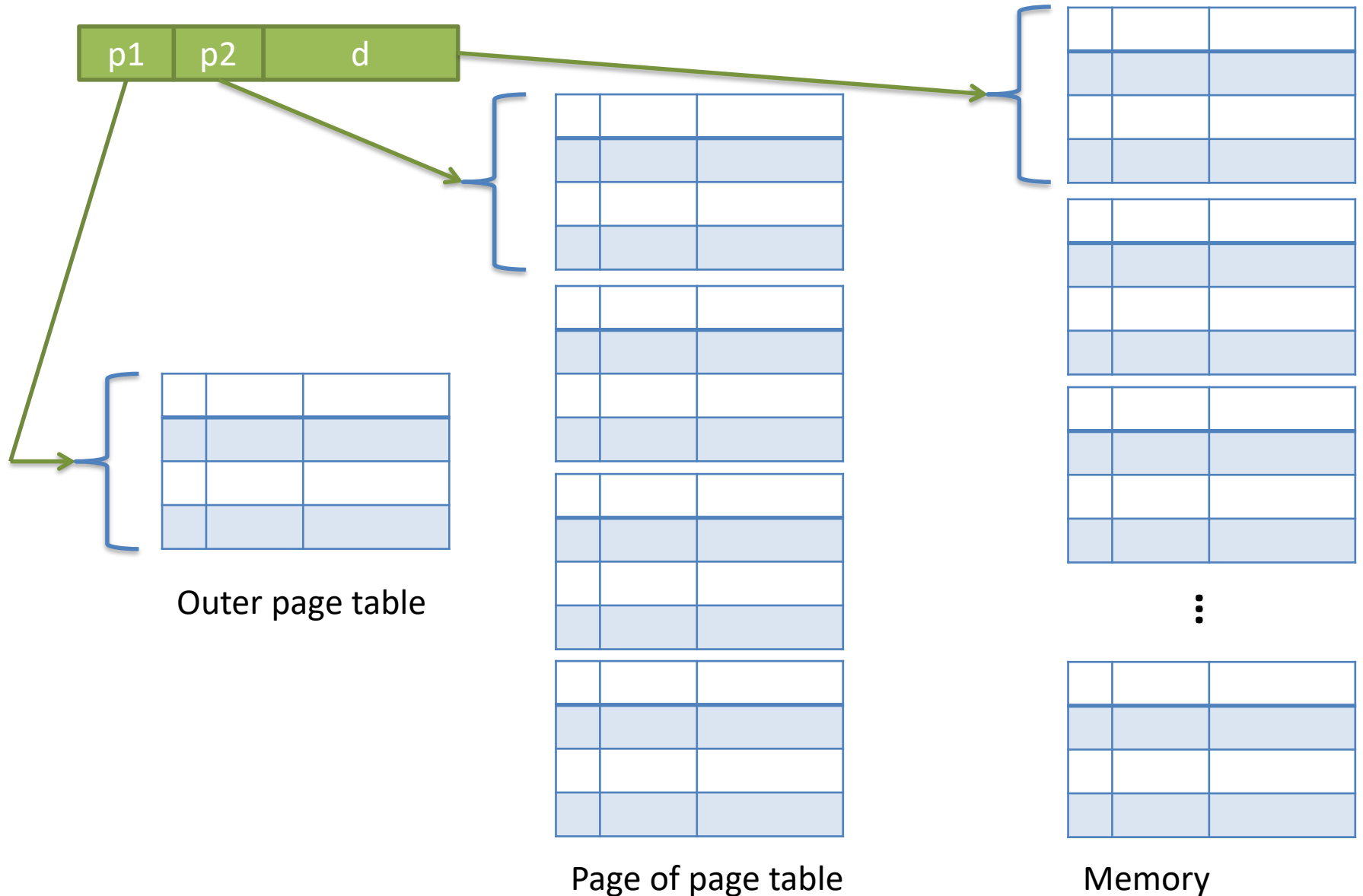
Multi-level Page Tables

- Reduce page table space for sparse address spaces
- Do not help for dense address spaces
- In fact, make it (slightly) worse
- In practice, most address spaces are *very* sparse

Question time!

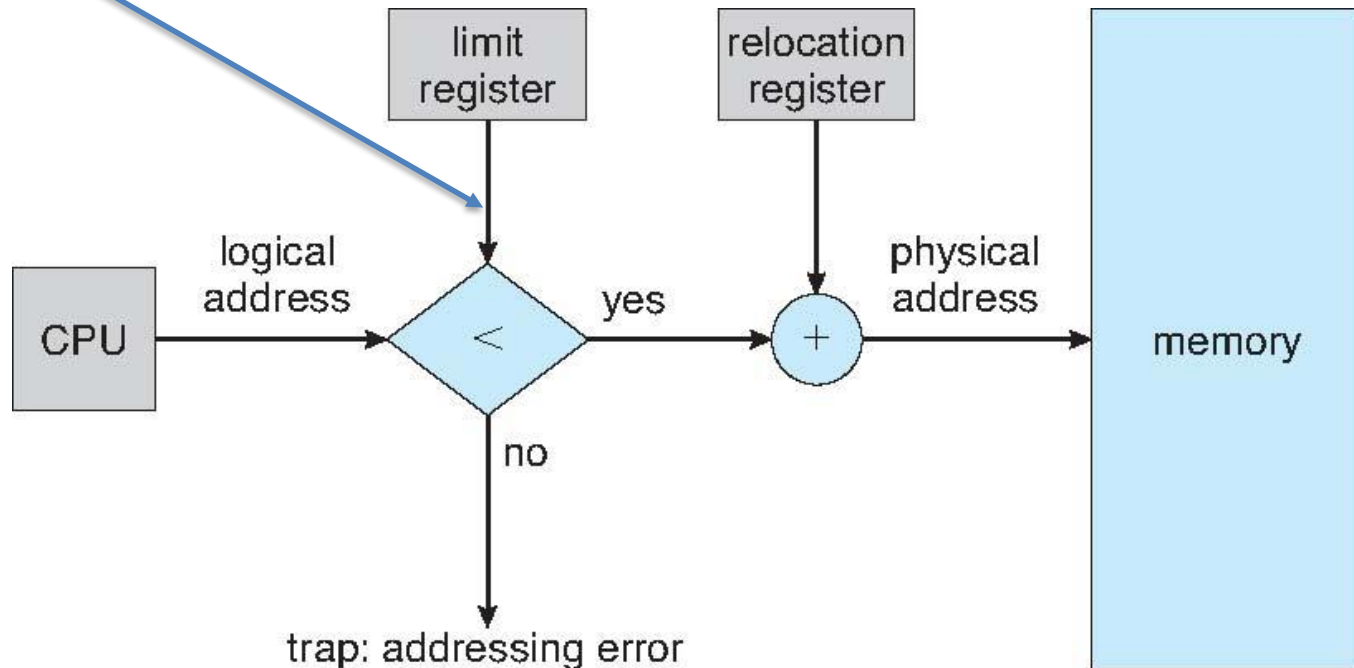


Q1. Number of bits for p2

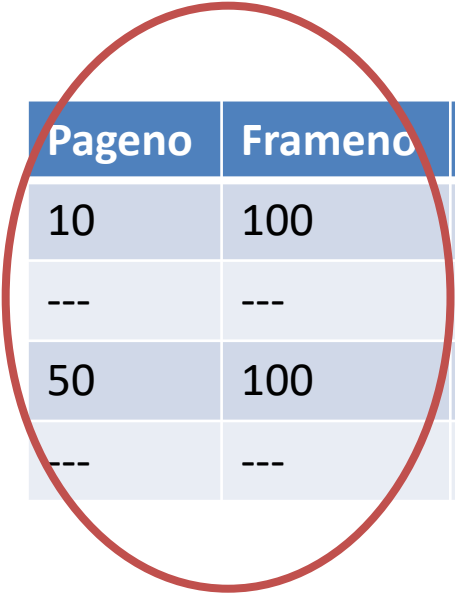


Q2. Base and bounds

Do we need relocation register input here?



Q3. TLB - can you have this situation?



Pageno	Frameno	valid	protection	ASID
10	100	1	rwX	1
---	---	0	---	---
50	100	1	rwX	2
---	---	0	---	---

Q4. Number of PTEs

- Virtual address – 32 bits
- Only low 20MB and upper 2MB valid
- Page size – 4KB or $d = 12$ bits, so $p = 20$ bits
- 1-level page table: $2^{20} = 1\text{M PTEs}$
- 2-level page table ($P1 = 8, P2 = 12$)
 - 2^8 for 1st level
 - $2 \times 2^{12} + 1 \times 2^{12}$ for 2nd level \longrightarrow Why?
 - $\sim 12\text{K PTEs}$

Week 7

Demand Paging

Pamela Delgado

March 27, 2018

based on:

- W. Zwaenepoel slides
- Arpaci-Dusseau book
- Silberschatz book

Key Concepts

- Demand paging (last week)
- Page fault handling (last week)
- Page replacement
- Frame allocation
- Optimizations

Demand Paging

- Part of the program is in memory
- (Typically) all of it is on disk

Remember

- CPU can only directly access memory
- CPU can only access data on disk through OS

Demand Paging

- What if program accesses part only on disk?

This is called a page fault

- Program is suspended
- OS runs, gets page from disk
- Program is restarted

This is called page fault handling

Issues

1. How to discover a page fault?
2. How to suspend process?
3. How to bring in a page from disk?
 3. a. How to find a free frame in memory?
4. How to restart process?

1. Discover Page Fault

- Use the valid bit in page table
- Without demand paging:
 - Valid bit = 0: page is invalid
 - Valid bit = 1: page is valid
- With demand paging
 - Valid bit = 0: page is invalid OR page is on disk
 - Valid bit = 1: page is valid AND page is in memory
- OS needs additional table: invalid / on-disk?

Demand Paging

0	A
1	B
2	C
3	D
4	E
5	F
6	G
7	H

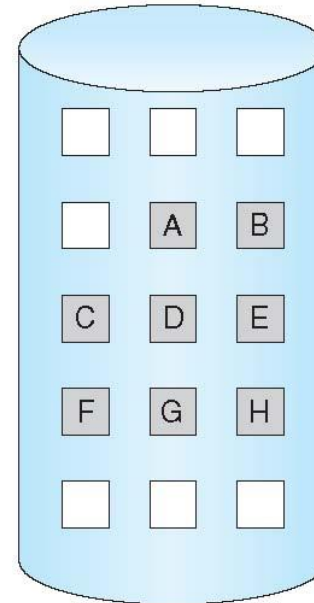
logical
memory

valid-invalid bit		
frame		
0	4	v
1		i
2	6	v
3		i
4		i
5	9	v
6		i
7		i

page table

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

physical memory



2. Suspending the Faulting Process

- Invalid bit access generates trap
- OS runs and suspends process
- As before, save process information in PCB

3. Getting the Page from Disk

- Assume there is at least one free frame
- Allocate a free frame to process
- Find page on disk
 - Note: need an extra table for that
- Get disk to transfer page from disk to frame

While the Disk is Busy

- Invoke scheduler to run another process
- When disk interrupt arrives
 - OS runs
 - Suspend running process
 - Get back to page fault handling

Completing Page Fault Handling

- `Pagetable[pageno].framenno = new framenno`
- `Pagetable[pageno].valid = 1`
- Set process state to ready
- Invoke scheduler

4. When Process Runs Again

- Restarts the previously faulting instruction
- Now finds
 - Valid bit to be set to 1
 - Page in corresponding frame in memory

3.a. If no Free Frame Available

- Pick a frame to be replaced
- Invalidate its page table entry (and TLB entry)
- You may have to write that frame to disk
- Page table entry has a modified bit
 - Set by hardware if page modified
 - If set, write out page to disk
 - If not, proceed with page fault handling

Page Replacement Policy

- How to pick with page/frame to replace?

Page Replacement Policies

- Random
- FIFO (First In, First Out)
- OPT
- LRU
- LRU approximations
- FIFO second-chance
- Clock



last week

The diagram consists of two vertical curly braces on the right side of the list. The top brace is grey and groups the first four items: Random, FIFO (First In, First Out), OPT, and LRU. The bottom brace is blue and groups the last three items: LRU approximations, FIFO second-chance, and Clock. The text 'last week' is positioned to the right of the top brace, and 'this week' is positioned to the right of the bottom brace.

this week

LRU: Least Recently Used recap

- Cannot look into the future
- But can try to predict future using past
- Replace least recently accessed page

LRU

reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

7	7	7	2		2		4	4	4	0			1		1		1	
	0	0	0		0		0	0	3	3			3		0		0	
		1	1		3		3	2	2	2			2		2		7	

page frames

9 page faults (not counting initial paging in)

LRU Implementation

- Can also not be implemented in general
- Need to timestamp every memory reference
- Too expensive
- But can be (well) approximated

Reference Bit

- Bit in page table
- Hardware sets bit when page is referenced

Simple LRU Approximation

- Replacement
 - Pick one of the pages with reference bit set
 - Reset all reference bits to zero

Better LRU Approximation

- Periodically
 - Read out and store all reference bits
 - Reset all reference bits to zero
- Keep all reference bits for some time
- The more bits kept, the better approximation
- Replacement:
 - Page with smallest value of reference bit history

LRU Approximation Operation

Current Ref	Ref at t-T	Ref at t-2T	Ref at t-3T	Ref at t-4T
0	0	0	1	0
0	1	0	0	0
1	1	1	1	0
1	1	1	1	1
0	1	1	0	0
1	0	0	0	0
0	0	1	1	1
0	0	0	0	1

LRU Approximation Operation

Current Ref	Ref at t-T	Ref at t-2T	Ref at t-3T	Ref at t-4T
0	0	0	1	0
0	1	0	0	0
1	1	1	1	0
1	1	1	1	1
0	1	1	0	0
1	0	0	0	0
0	0	1	1	1
0	0	0	0	1

FIFO Page Replacement

reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

7	7	7	2																
	0	0	0																
		1	1																

2	2	4	4	4	0														
3	3	3	2	2	2														
1	0	0	0	3	3														

0	0																		
1	1																		
3	2																		

7	7	7																	
1	0	0																	
2	2	1																	

page frames

12 page faults (not counting initial paging in)

FIFO – bad case example

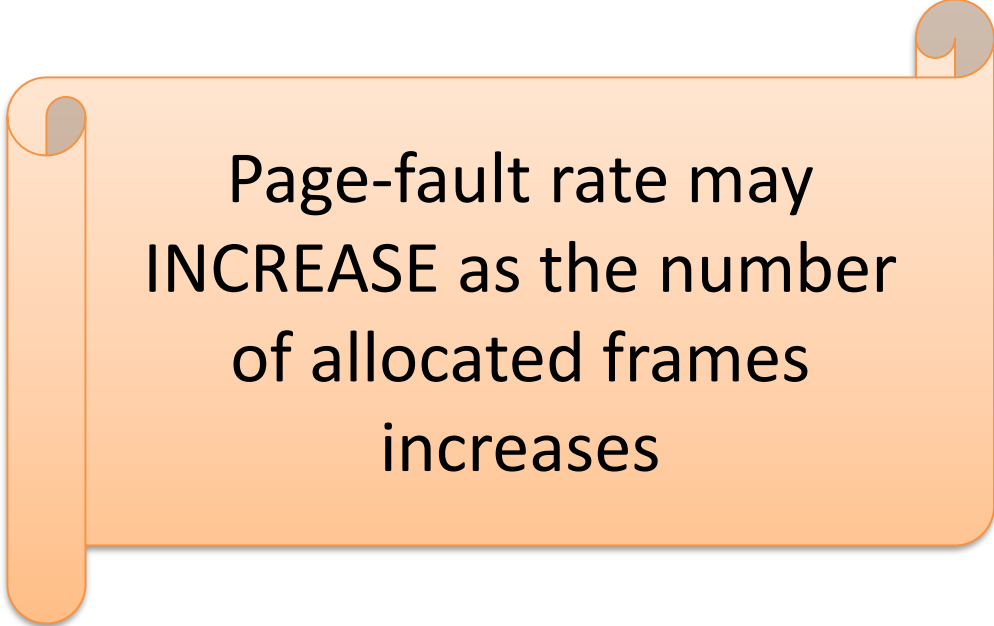
Reference string:

1 2 3 4 1 2 5 1 2 3 4 5

- Assume 3 frames
Q. Number of page faults?
- Now assume 4 frames
Q. Number of page faults?



Extra reading - Belady's anomaly

An orange scroll graphic with a light orange background and a darker orange border. The scroll is unrolled in the center, with the ends of the scroll visible on the left and right sides. The text is centered on the unrolled portion.

Page-fault rate may
INCREASE as the number
of allocated frames
increases

FIFO with Second Chance

- As FIFO, but each page gets a second chance
- Bring in page: put at tail of queue (as before)
- Replacement:
 - Look at head of queue
 - If reference bit is 0, replace
 - While reference bit is 1 /* give second chance */
 - Put at tail of queue
 - Set reference bit to 0
 - Look at head of queue

Why does it work? – Advantage 1

- Combination of
 - FIFO
 - LRU approximation with one reference bit
- In LRU approximation with one reference bit
 - Take any page with reference bit 0
- Here:
 - Take “oldest” page with reference bit 0

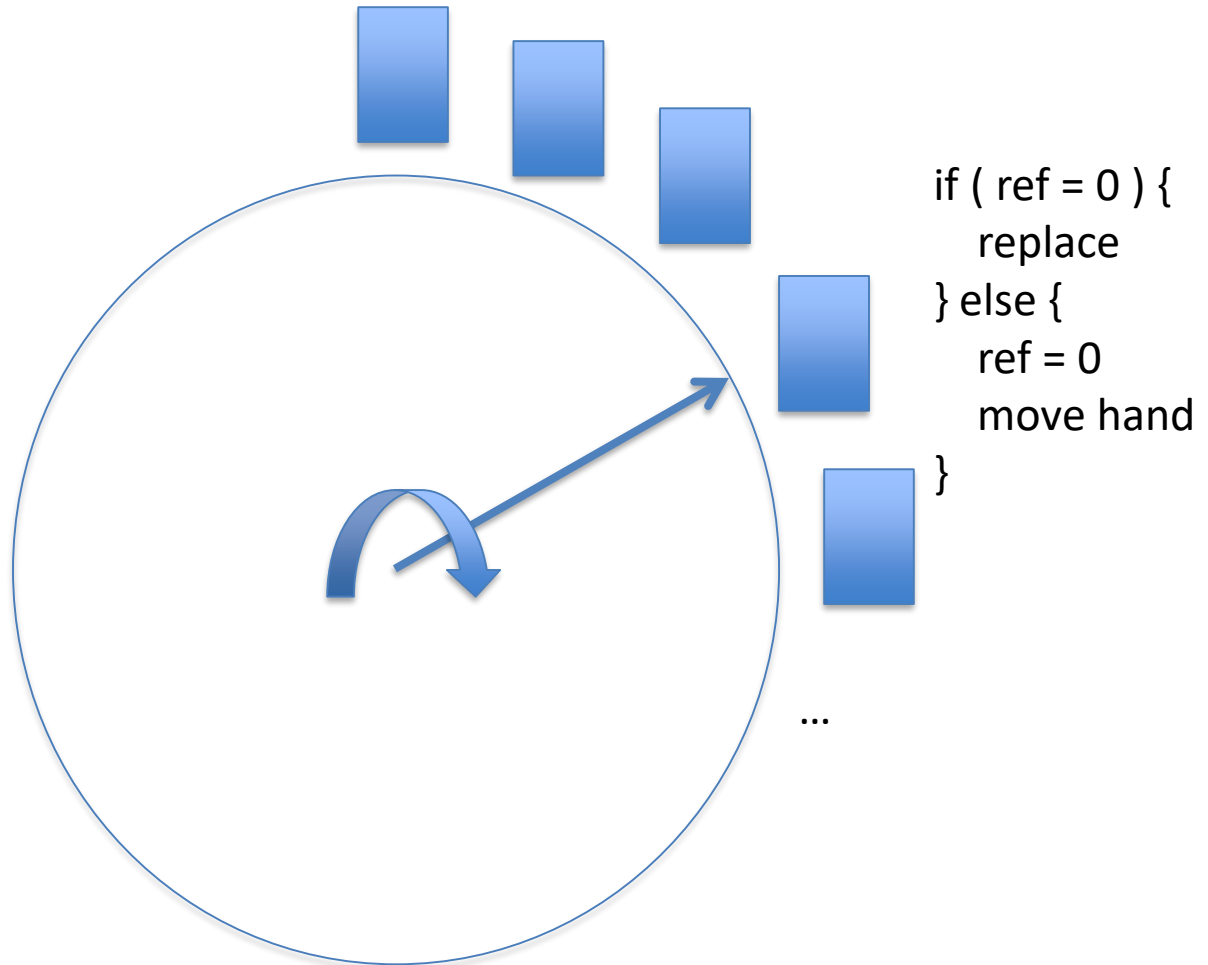
Why does it work? – Advantage 2

- Combination of
 - FIFO
 - LRU approximation with one reference bit
- In LRU approximation with one reference bit
 - Set all ref bits to 0
- Here:
 - Set ref bits of “old” pages to 0

Clock

- Imagine pages arranged around a clock
- Replacement:
 - Look at page where hand of clock is
 - If reference bit = 0, replace
 - If reference bit = 1
 - Set reference bit to 0
 - Move hand of clock to next page
 - Insert new page where old one was replaced

Clock



Clock vs FIFO + Second Chance

- The two are the same policy
 - The clock points points to the head of the queue
- Clock is more efficient
 - Doesn't need to move pages around in queue
 - Instead, clock hand moves
- Works well + efficient to implement
 - Variations used in many systems (e.g., Linux)

Page Replacement Policies

Policy	Implementation	Performance
Random	Easy	Poor
FIFO	Easy (queue)	Poor
OPT	Impossible	Optimal
LRU	Difficult (timestamp)	Good
LRU approximation	Moderate (ref bits)	Good (depending on approx.)
FIFO with second chance	Easy	Good
Clock	Easy and efficient	Good

Frame Allocation

- How many frames to give to each process?

Degree of Multiprocessing

- How many processes to keep in memory?
- Without demand paging:
 - All of process must be in memory
 - Severely limits degree of multiprocessing
- With demand paging:
 - Only part of process must be in memory
 - Can achieve high degree of multiprocessing

Close Link between Degree of Multiprocessing - Page Fault Rate

- Give each process frames for \sim all of its pages
 - Low degree of multiprocessing
 - Few page faults
 - Slow switching on i/o

Close Link between Degree of Multiprocessing - Page Fault Rate

- Give each process frames for \sim all of its pages
 - Low degree of multiprocessing
 - Few page faults
 - Slow switching on i/o
- Give each process 1 frame
 - High degree of multiprocessing
 - Many page faults (*thrashing*)
 - Quick switching on i/o



Close Link between Degree of Multiprocessing - Page Fault Rate

- Give each process frames for \sim all of its pages
 - Low degree of multiprocessing
 - Few page faults
 - Slow switching on i/o
- Give each process 1 frame
 - High degree of multiprocessing
 - Many page faults (*thrashing*)
 - Quick switching on i/o
- Where is the correct tradeoff?



Working Set of a Process

- Set of pages of process needed for execution
- Intuition:
 - Working set not in memory → many page faults
 - Working set in memory → no page faults
 - More than working set in memory → no gain

Tradeoff

Frame allocation	Page faults	Degree of Multiprocessing
All frames	None	Low
1 frame	Many	High
Working set	Few	Moderate

Why working set $<$ all pages?

- Principle of locality
- In given time interval:
 - Process only accesses part of its pages
 - Example: initialization, main, termination, error, ...

Frame Allocation Policy

- Give each process enough frames
 - To maintain its working set in memory
- If sum of all working sets $>$ memory
 - Swap out one or more processes
- If sum of all working sets $<$ memory
 - Swap in one or more processes

How to Predict Working Set?

- Working set for next 10'000 refs
= working set for last 10'000 refs
- Prediction is not perfect
 - Phase change (e.g., from initialization to main)
 - Will cause (temporary) high page fault rate
 - Will decrease during next phase



How to Measure Past Working Set?

- We do not really need working set
- We only need working set size

How to Measure Past Working Set?

- We do not really need working set
- We only need working set size
- Periodically (every 10'000 references)
 - Count reference bits set to 1
 - Set all references bits to 0
- Working set (for last 10'000 references)
 - = number of reference bits set to 1

Relationship between Page Replacement – Frame Allocation

- Frame allocation done periodically
- Page replacement done at page fault time

Global vs Local Replacement

- Local replacement: replace page of faulting process
- Global replacement: replace any page
- Done according to page replacement
- E.g., FIFO:
 - Local: replace oldest page of faulting process
 - Global: pick the oldest page overall

Tradeoff: Local - Global

- Assume using working set for frame allocation
- Local:
 - You cannot affect anyone else's working set
 - Hence, you cannot cause thrashing of others
 - But inflexible for yourself
 - If working set grows, cannot react
 - Hence, you can cause yourself to thrash

Tradeoff: Local - Global

- Assume using working set for frame allocation
- Global:
 - You can affect others' working set
 - Hence, you cause thrashing of others
 - But flexible for yourself
 - If working set grows, can react
 - Hence, you can avoid yourself to thrash

Tradeoff: Local vs Global

- Use frame allocation periodically
- Use local replacement in-between
- You may thrash for a short time
- But at next period get bigger allocation, so ok
- Cannot cause thrashing of others, so ok

Frame Allocation Driving Page Replacement

- Periodically run working set computation
- If allocation $>$ working set
- Immediately replace non-WS pages
- Or favor them during later replacement
- Algorithm: WSClock does this (not covered)

Some Optimizations

- Prepaging
- Cleaning
- Free frame pool
- Copy-on-write sharing

Prepaging

- So far: page in 1 page at time
- Prepaging: page in multiple pages at a time
- Usually, pages “surrounding” faulting page

Prepaging - Performance

- Relies on locality of virtual memory access
 - Nearby pages are often accessed soon after
- Avoids page faults, process switches, ..
- Can also get better disk performance (later)
- May bring in unnecessary pages

Cleaning

- So far: prefer to replace “clean” pages
- Cleaning: disk idle, write out “dirty” pages

Cleaning - Performance

- More “clean” pages at replacement time
 - Quicker replacement
- But page may be modified again
 - Useless disk traffic

Free Frame Pool

- So far: use all possible frames for pages
- Free pool: keep some frames unused

Free Frame Pool - Performance

- Page fault handling is quick
- Reduces effective main memory size

Copy-on-Write

- Clever trick for sharing pages between processes
 - That are initially the same
 - That are likely to be read-only
 - But that may (unlikely) be modified by a process

A Step Back: Read-Only Sharing

- Make page table entry point to same frame
- Set read-only bit so trap if process writes
- Trap treated as illegal memory access

Copy-on-Write Sharing

- Make page table entry point to same frame
- Set read-only bit so trap if process writes
- Fault not treated as illegal memory access
- Instead:
 - Create separate frame for faulting process
 - Insert (pageno, frameno) in page table
 - Set read-only bit to off (i.e., read-write)
 - Copy page into that frame
- Further accesses will not page fault

Works Well?

- If page is rarely written
- You save frames (as many as sharers – 1)

Works Poorly?

- Page is often written
- You take more page faults
- You don't gain in frame occupation

In Practice

- The conditions under which copy-on-write works well occur in practice
- All OS's provide it
- This is how Linux implements `fork()`

Summary

- Demand paging
- Page fault handling
- Page replacement
- Frame allocation
- Optimizations