

Recap - Week 5

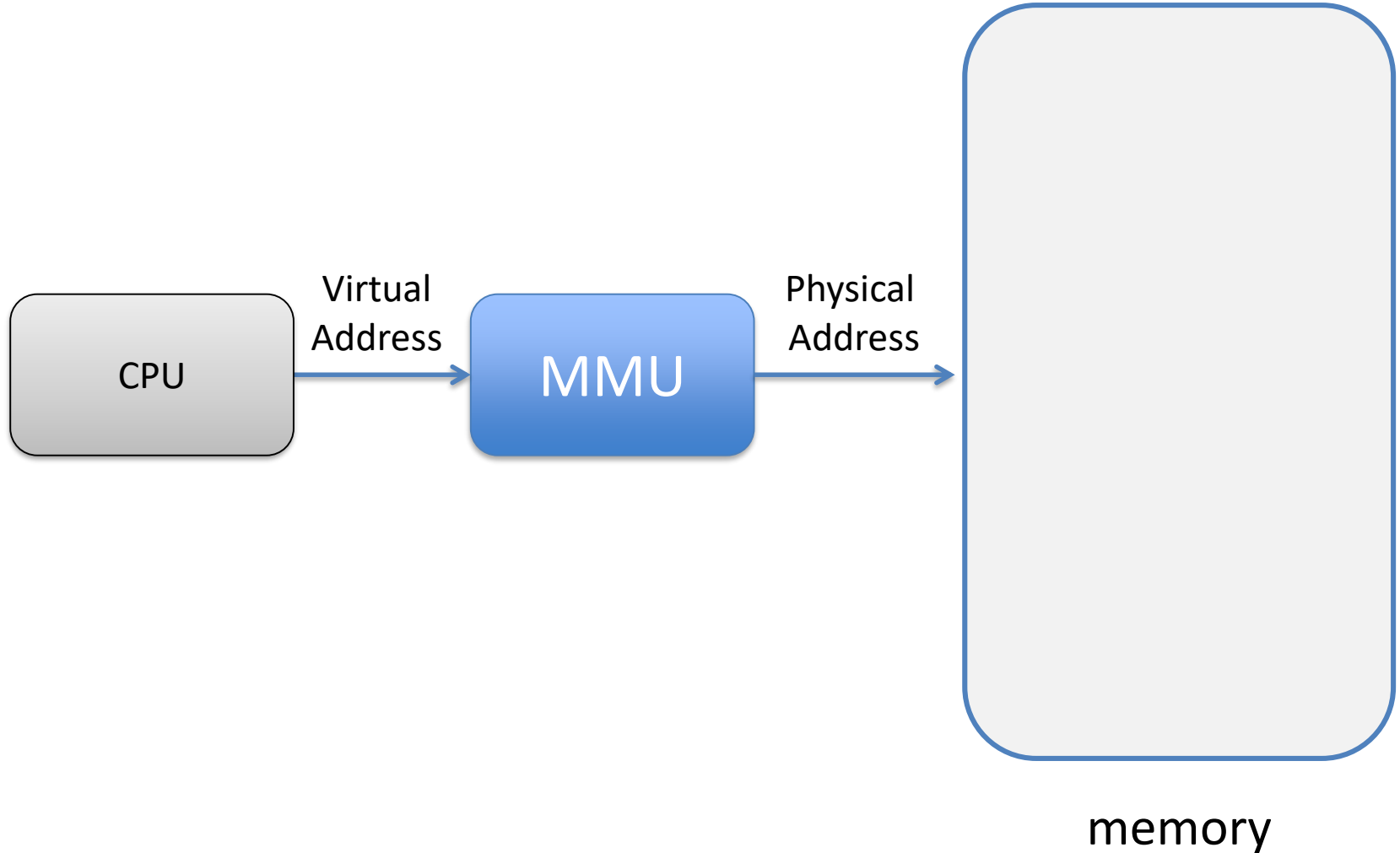
Pamela Delgado

March 20, 2019

Virtual vs Physical Address Space

- Virtual/logical address space =
 - What the program(mer) thinks is its memory
- Physical address space =
 - Where the program is in physical memory

MMU: Mapping Virtual to Physical



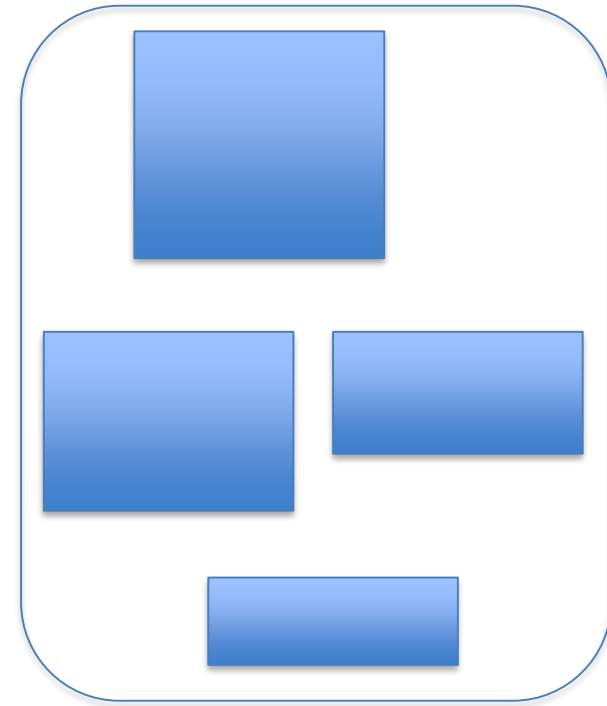
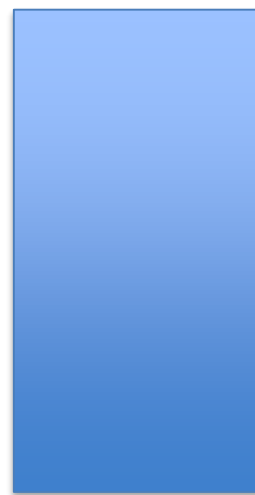
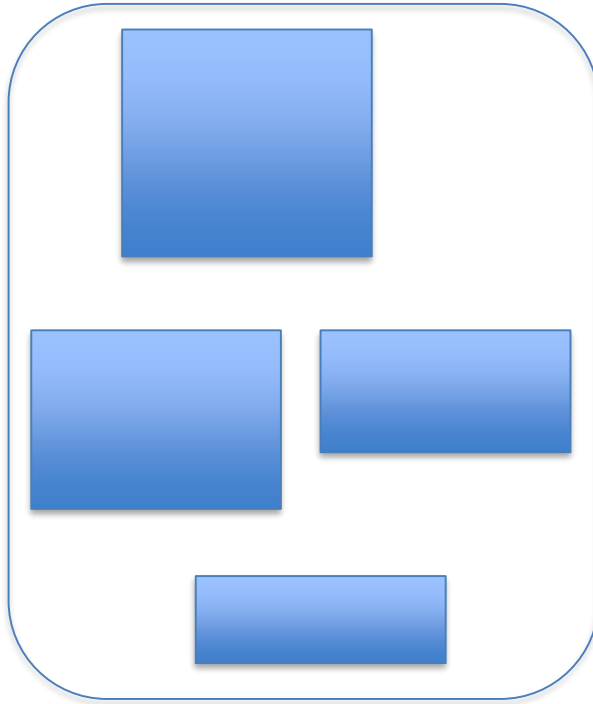
Virtual Address Space

Base and
Bounds

Segmentation

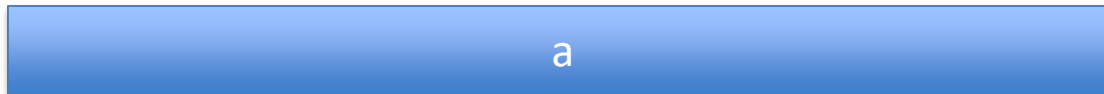
Paging

Segmentation with Paging



Virtual Address Format

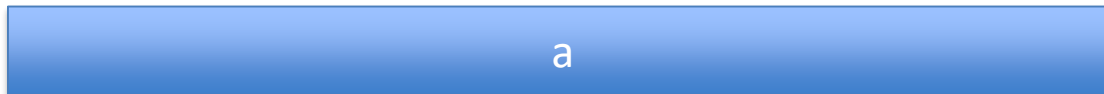
- Base and bounds



- Segmentation



- Paging

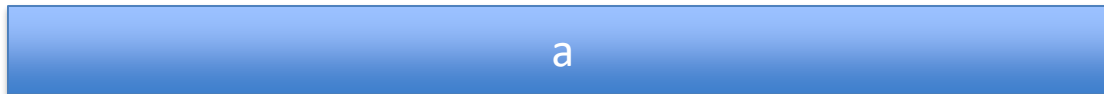


- Segmentation with Paging



Breakdown of Virtual Address for Mapping

- Base and bounds



- Segmentation



- Paging

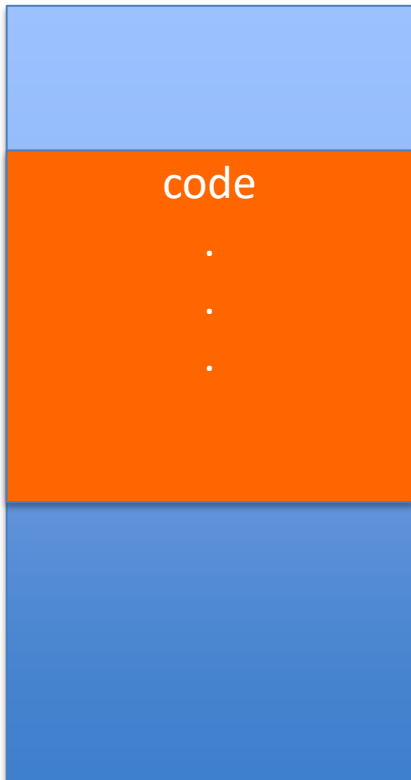


- Segmentation with Paging

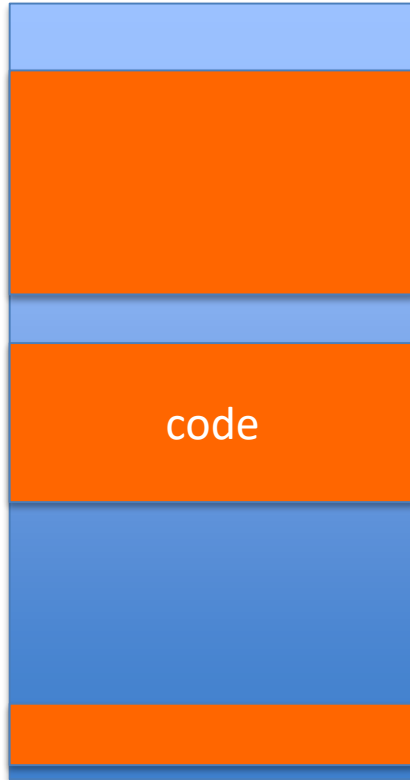


Physical Address Space

Base and
Bounds



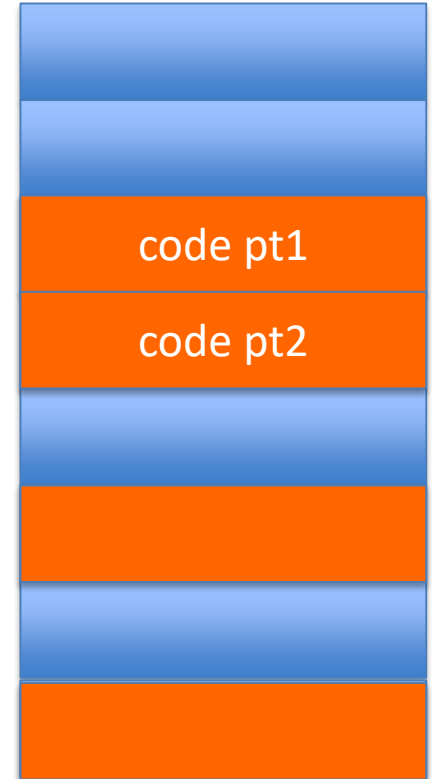
Segmentation



Paging



Segmentation with Paging



Main Memory Allocation

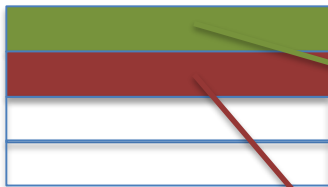
	Number	Size	Fragmentation
Base-and-bounds	1	Variable	Lots (external)
Segmentation	Many	Variable	Some (external)
Paging	Many	Fixed	Little (internal)
Segmentation with Paging	Many	Fixed	Little (internal)

How to Share Memory?

- With base and bounds, not possible
- With segmentation
 - Create segment for shared data
 - Entry in segment table of both processes
 - Points to shared segment in memory
- With paging
 - Need to share pages
 - Entries in page table of both processes
 - Point to shared pages

P1 and P2 Share One Segment

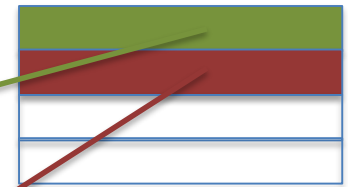
SegmentTable P1



Memory



SegmentTable P2

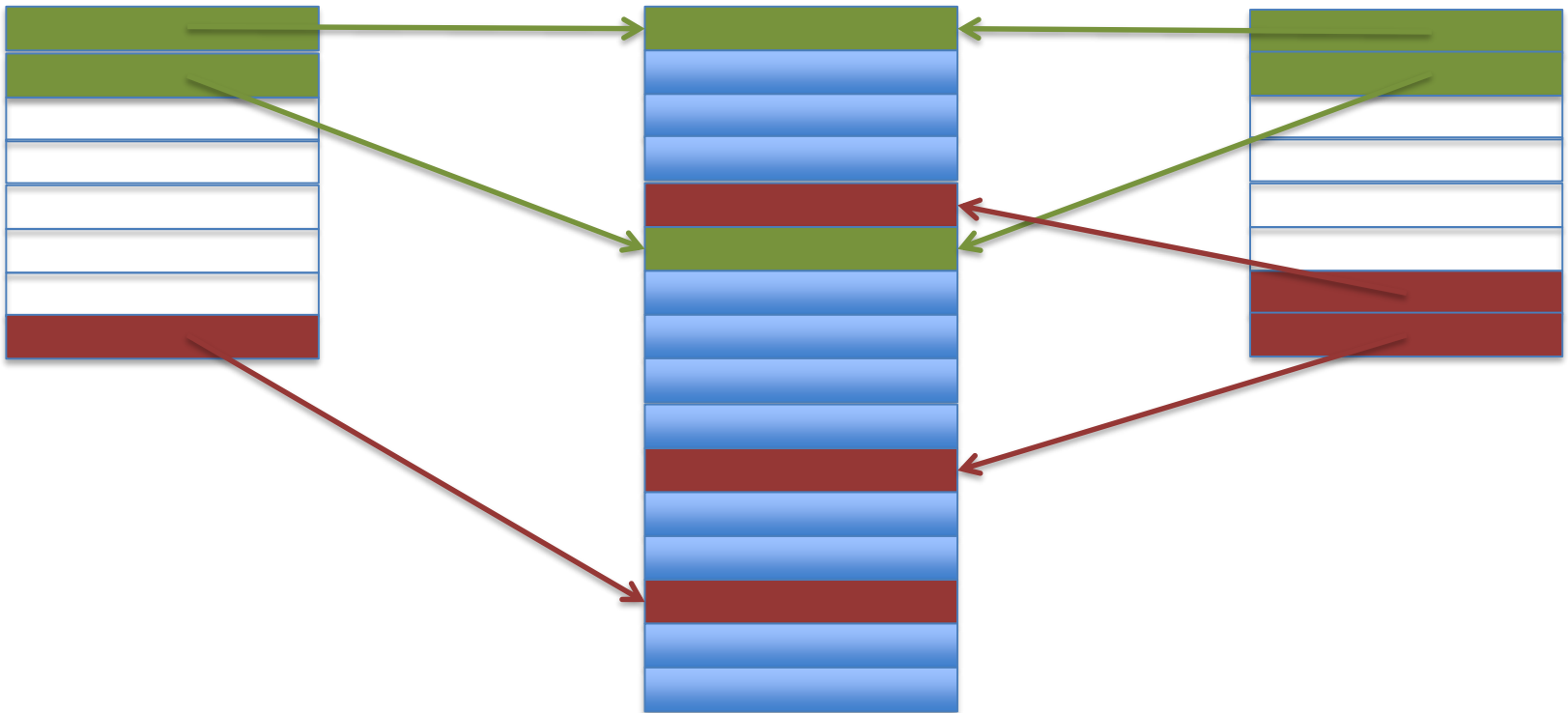


P1 and P2 Share Two Pages

PageTable P1

Memory

PageTable P2



Advantages / Disadvantages

	Segmentation	Paging	Segmentation with Paging
Sharing	<input checked="" type="checkbox"/>		<input checked="" type="checkbox"/>
Fine-grain protection	<input checked="" type="checkbox"/>		<input checked="" type="checkbox"/>
Memory allocation		<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>

In Reality

- Base-and-bounds only for niche
- Segmentation abandoned
 - Complexity for little gain
 - Effect approximated with paging with valid bits
- Paging is now universal

Address Translation Performance Issue

- Page table is in memory
- 1 virtual address → 2 physical memory accesses
- *Would reduce performance by factor of 2*
- Solution: Translation lookaside buffer (TLB)

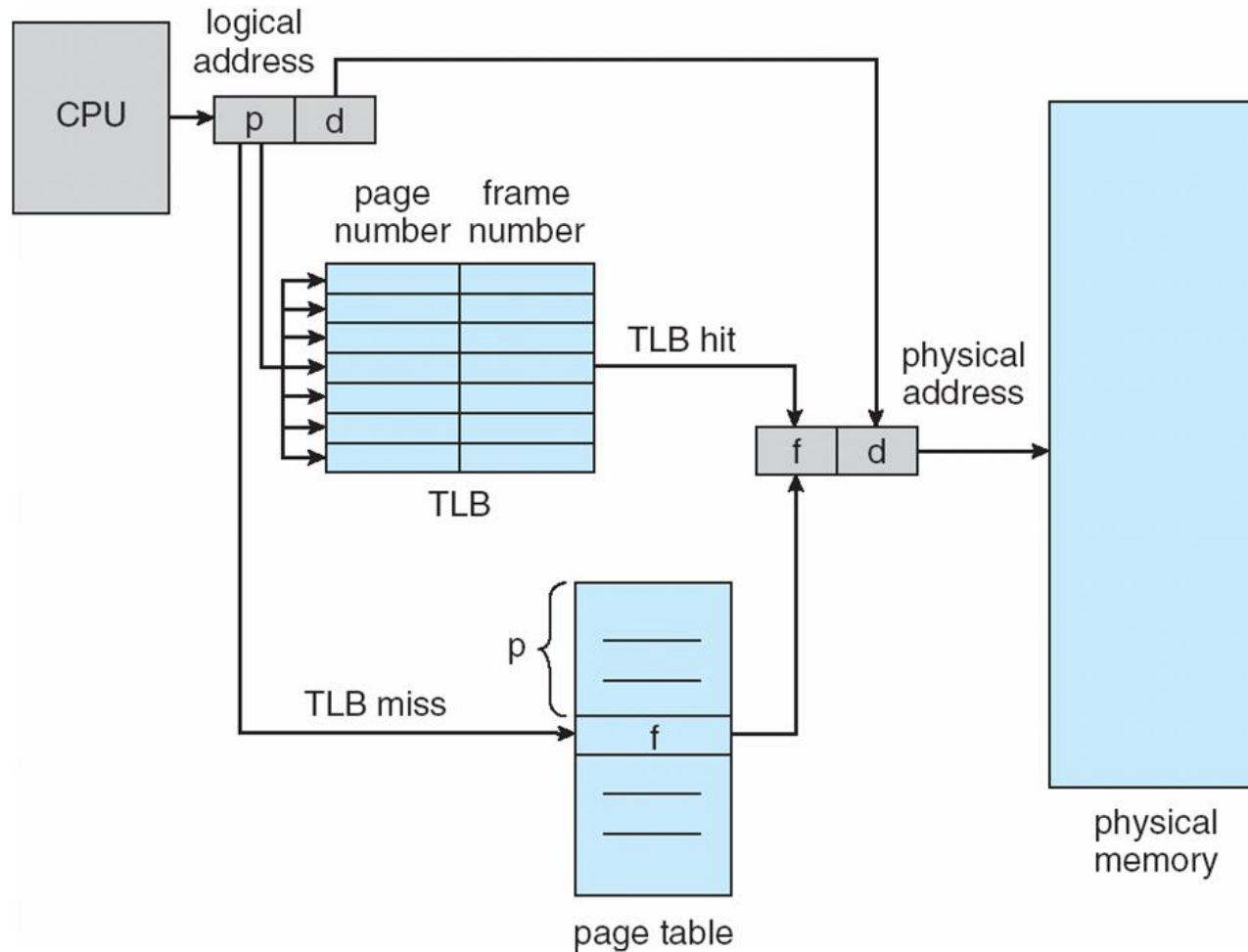


Time problem

TLB

- Small fast cache of (pageno, frameno) maps
- If mapping for pageno found in TLB
 - Use frameno from TLB
 - Abort mapping using page table
- If not
 - Perform mapping using page table
 - Insert (pageno, frameno) in TLB

Paging Mapping With TLB



TLB hit/miss example

```
int sum =0;
for (i=0; i<10; i++){
    sum += a[i];
}
```

- Assume we need only accesses to a
- Assume TLB is empty at the beginning
 - 3 TLB miss
 - 7 TLB hit



Offset	00	04	08	12	16	V. Pageno
						00
						01
						02
			a[0]	a[1]		03
	a[2]	a[3]	a[4]	a[5]		04
	a[6]	a[7]	a[8]	a[9]		05
						06
						07

Virtual address space

Paging bits examples

- Valid bit → used space by the program
- Protection bit → read/write/executed
- Present bit → memory or disk
- Dirty bit → modified content
- Reference bit → page has been accessed
- Few bits to determine HW caching

Week 6

Virtual Memory: OS Implications and Demand Paging

Pamela Delgado

March 27, 2019

based on:

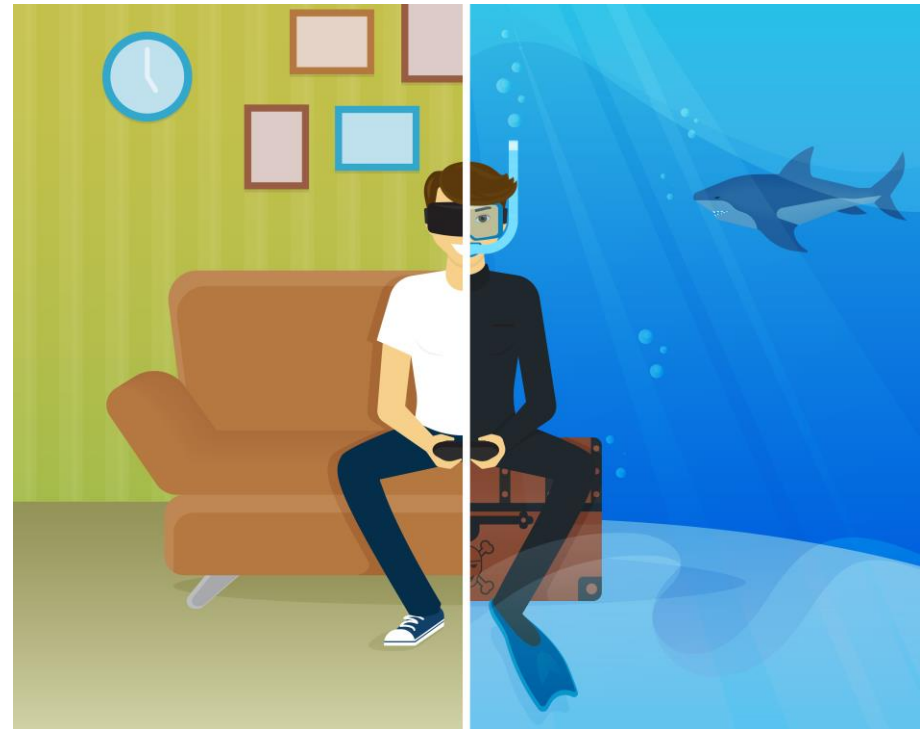
- W. Zwaenepoel slides
- Arpaci-Dusseau book
- Silberschatz book

Key Concepts

- Very large address spaces
- Process switching and memory management
- Demand paging

Virtual memory

- Address space ilusion!
- Appear to have much more memory than reality
- Address space
 - 32bit or 64bit
 - for each process!



virtual reality

Dealing with Large Virtual Address Spaces

- 64-bit virtual address space
- 4KB pages (12-bit page offset)

Dealing with Large Virtual Address Spaces

- 64-bit virtual address space
- 4KB pages (12-bit page offset)
- Leaves 52 bits for pageno
- Would require 2^{52} page table entries
- Let's say every page table entry 4 bytes
- Page table size = 2^{54} bytes
- More than main memory!

Large page table, mostly unused

Page table

Valid bit	...	Page frame no
1		
1		
0		
1		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
0		
1		

Not used

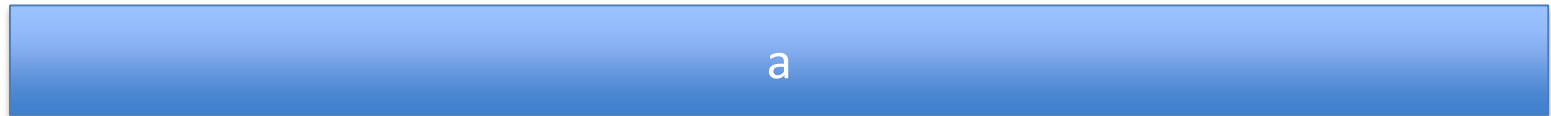
Space
problem

Solutions

- Hierarchical page tables
 - Example: two-level page tables
- Hashed page tables
- Inverted page tables

Single-level Page Tables

- Virtual address:

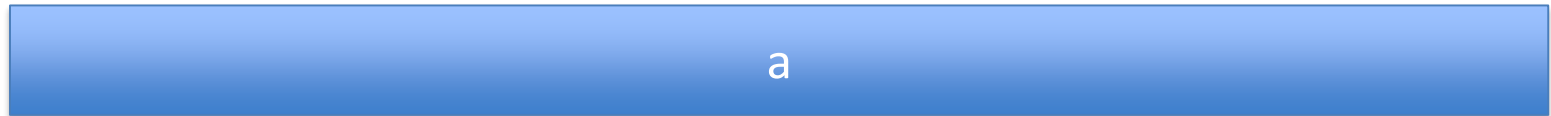


- Breakdown of virtual address for mapping:

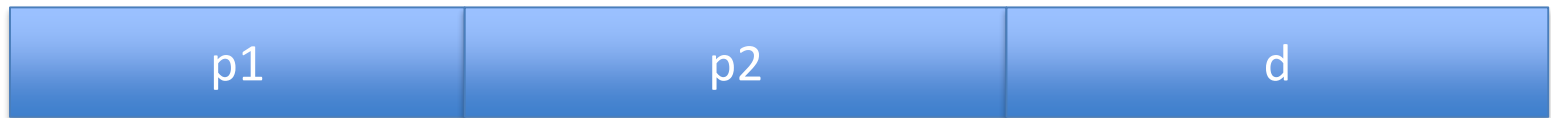


Two-level Page Tables

- Virtual address:



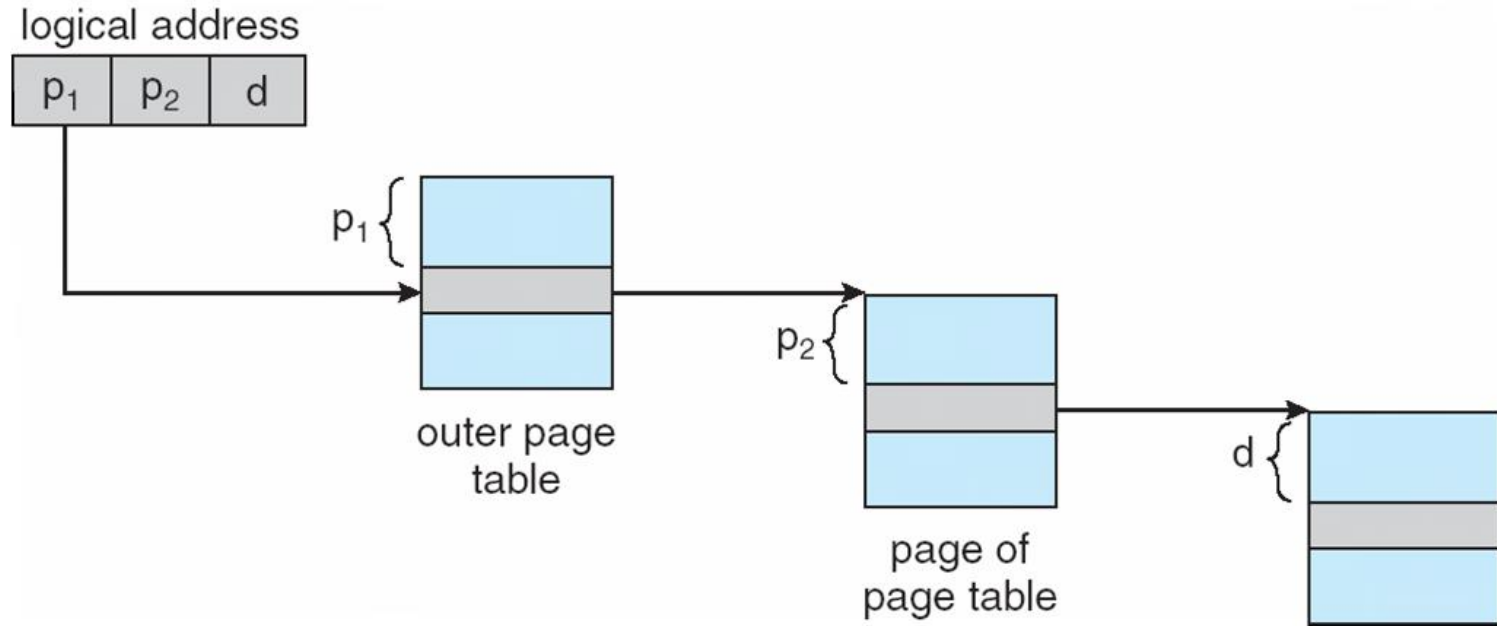
- Breakdown of virtual address for mapping:



MMU for Two-level Page Tables

- PTBR: points to top-level page table
- Top-level page table entry:
 - Indexed by p1
 - Pointer to second-level page table
 - Valid bit
- Second-level page table entry:
 - Indexed by p2
 - Frame containing page (p1,p2)
 - Valid bit

Address-Translation Scheme



Two-Level Page Table Memory Use

- For sparse address spaces
- One-level page table:
 - Need page table for entire address space
- Two-level page table:
 - Need top-level page table for entire address space
 - Need only second-level page tables for populated parts of the address space

(Small-Size) Example

- Total address length – 8 bits
- P – 4 bits (16 pages)
- D – 4 bits (16-byte pages)
- Let's say only page 0, 14 and 15 are used

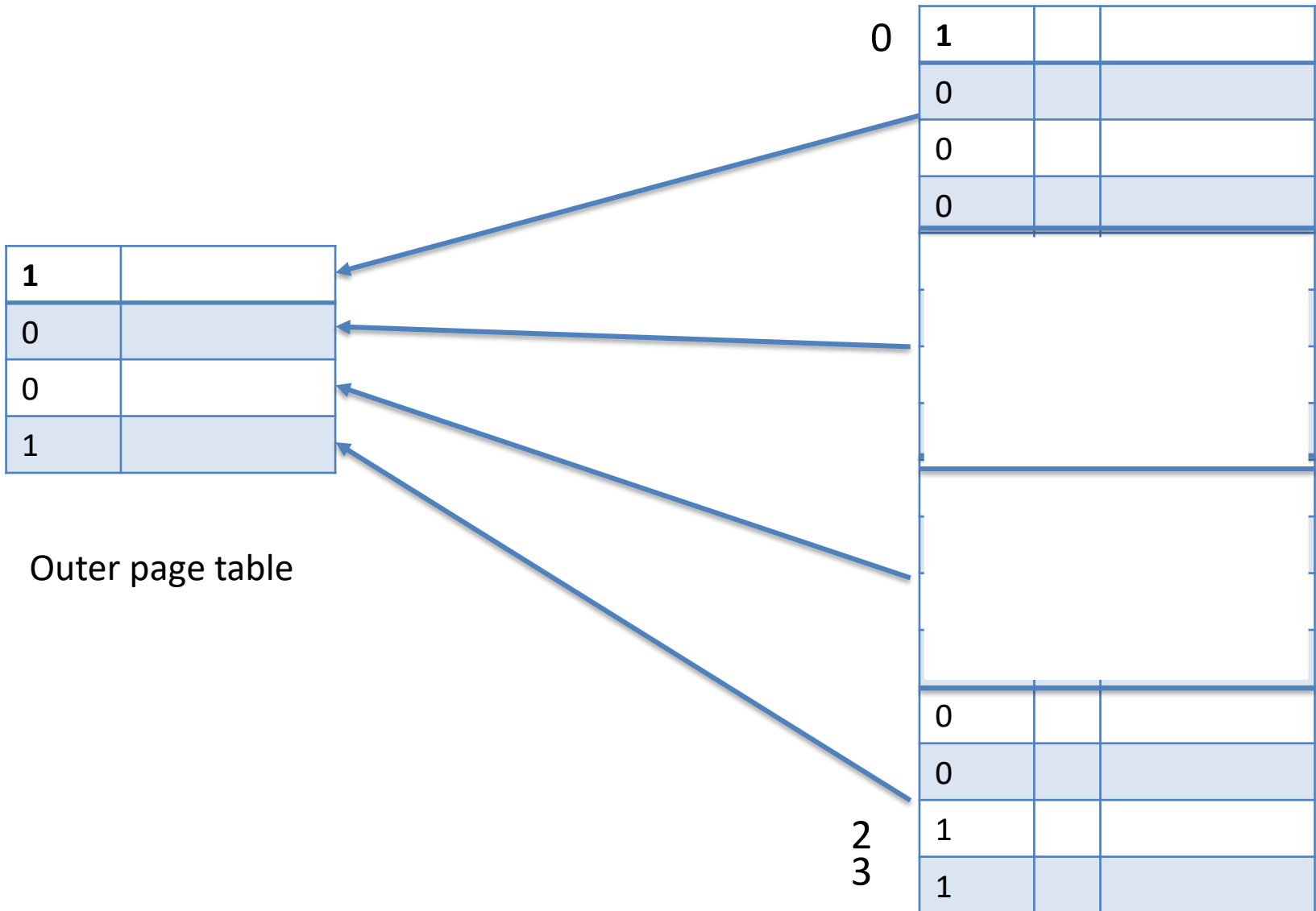
Flat (1-Level) Page Table

$2^4 = 16$ entries

0	1		
	0		
	0		
	0		
	0		
	0		
	0		
	0		
	0		
	0		
	0		
	0		
	0		
	0		
	0		
14	1		
15	1		

Two-level Page Table

$2^4 + 2 \times 2^4 = 12$ entries



Realistic Example

- Virtual address – 32 bits
- Only low 20MB and upper 2MB valid
- Page size – 4KB or $d = 12$ bits, so $p = 20$ bits

Realistic Example

- 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: ?? PTEs

Realistic Example

- 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: $2^{20} = 1\text{M PTEs}$

Realistic Example

- 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 ($P1 = 8, P2 = 12$)

Realistic Example

- 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

Realistic Example

- 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?
homework
 - $\sim 12\text{K PTEs}$

In General: Two-Level Page Tables

- For sparse address spaces
- One-level page table:
 - Need page table for entire address space
- Two-level page table:
 - Need top-level page table for entire address space
 - Need only second-level page tables for populated parts of the address space

Two-level Page Tables for Dense Address Spaces

- Not useful
- In fact, counter-productive
- But most address spaces are sparse

Are Two Levels Enough?

- Size second-level page table == size of page
- Why? Easy to allocate
- Let's assume
 - 4KB pages
 - 4 bytes per PTE
- It follows
 - $p_2 = ??$

Are Two Levels Enough?

- Size second-level page table == size of page
- Why? Easy to allocate
- Let's assume
 - 4KB pages
 - 4 bytes per PTE
- It follows
 - $p_2 = 10$

Are Two Levels Enough?

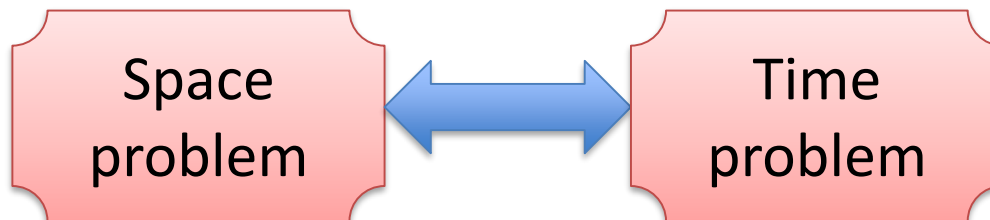
- 64-bit address space
- 4KB pages
- $d = 12$, $p_2 = 10$
- Thus, $p_1 = 42$
- Top level page table: 2^{42} entries

More Levels

- 3-level page table:
 - $d = 12$, $p_3 = 10$, $p_2 = 10$, $p_1 = 32$
- 4-level page table:
 - $d = 12$, $p_4 = 10$, $p_3 = 10$, $p_2 = 10$, $p_1 = 22$

The price to be paid

- Each level adds another memory access
- N-level page table
 - 1 memory access \rightarrow $n+1$ memory accesses
- But, TLB still works
 - If hit, 1 memory access \rightarrow 1 memory accesses
 - If miss, 1 memory access \rightarrow $n+1$ memory accesses
- TLB hit rate must be very high (99+ %)



Process Switching and Memory Management

Revisiting Process Switching

- What does kernel need to do on switch?
- Before we said:
 - Save and restore PC and registers
- Now we need to add:
 - Save and restore memory mapping information
- Additional fields in process control block (PCB)

Process Switch: Memory Mapping Info

- Base and bounds: base and limit register
- Segmentation: STBR and STLR
- Paging: PTBR and PTLR

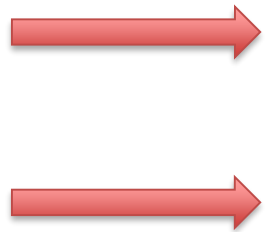
- Note: need *not* save and restore segment and page table (they are in memory)

Process Switch: TLB?

Process Switch: TLB Issue

- Suppose
 - Process P1 is running
 - Entry (pageno, frameno) in TLB
 - Switch from P1 to P2
 - P2 issues virtual address in page pageno
- P2 accesses P1's memory!

TLB and process switch



Pageno	Frameno	valid	protection
10	100	1	rwX
---	---	0	---
10	170	1	rwX
---	---	0	---

Process Switch: TLB Issue – Solution 1

- On process switch, invalidate all TLB entries
 - Simply requires invalid bit in each TLB entry
 - Makes process switch expensive
 - New process initially incurs 100% TLB misses

Process Switch: TLB Issue: Solution 2

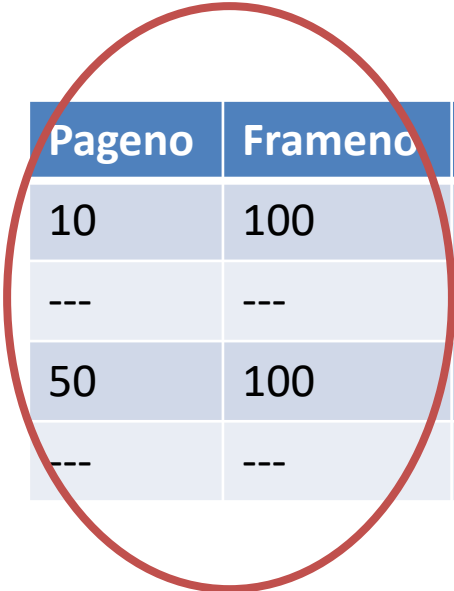
- Have process identifier in TLB entries
 - Match = match on pid and match on pageno
 - Makes TLB more complicated and expensive
- Process switch
 - Nothing to do
 - Cheaper
- All modern machines have this feature

TLB and process switch

Pageno	Frameno	valid	protection	ASID
10	100	1	rwX	1
---	---	0	---	---
10	170	1	rwX	2
---	---	0	---	---

Address space
identifier

Can you have this situation?



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

Demand Paging

Remember from Last Week

Simplifying Assumption

- For this week's lecture only
- All of a program must be in memory
- Will revisit assumption next week
- *We are now going to drop this assumption*

Main Reason

- Virtual address spaces > physical address space
- Implicitly we looked at this already
 - 64-bit virtual address space
 - No machine has 2^{64} bytes (16 exabytes) of memory
- Why such large virtual address space?
 - Don't have to worry about running out

Does it make sense?

- Code needs to be in memory to execute, but entire program rarely used
 - Error code, unusual routines, large data structures
- Entire program code not needed at same time

Partially-loaded program

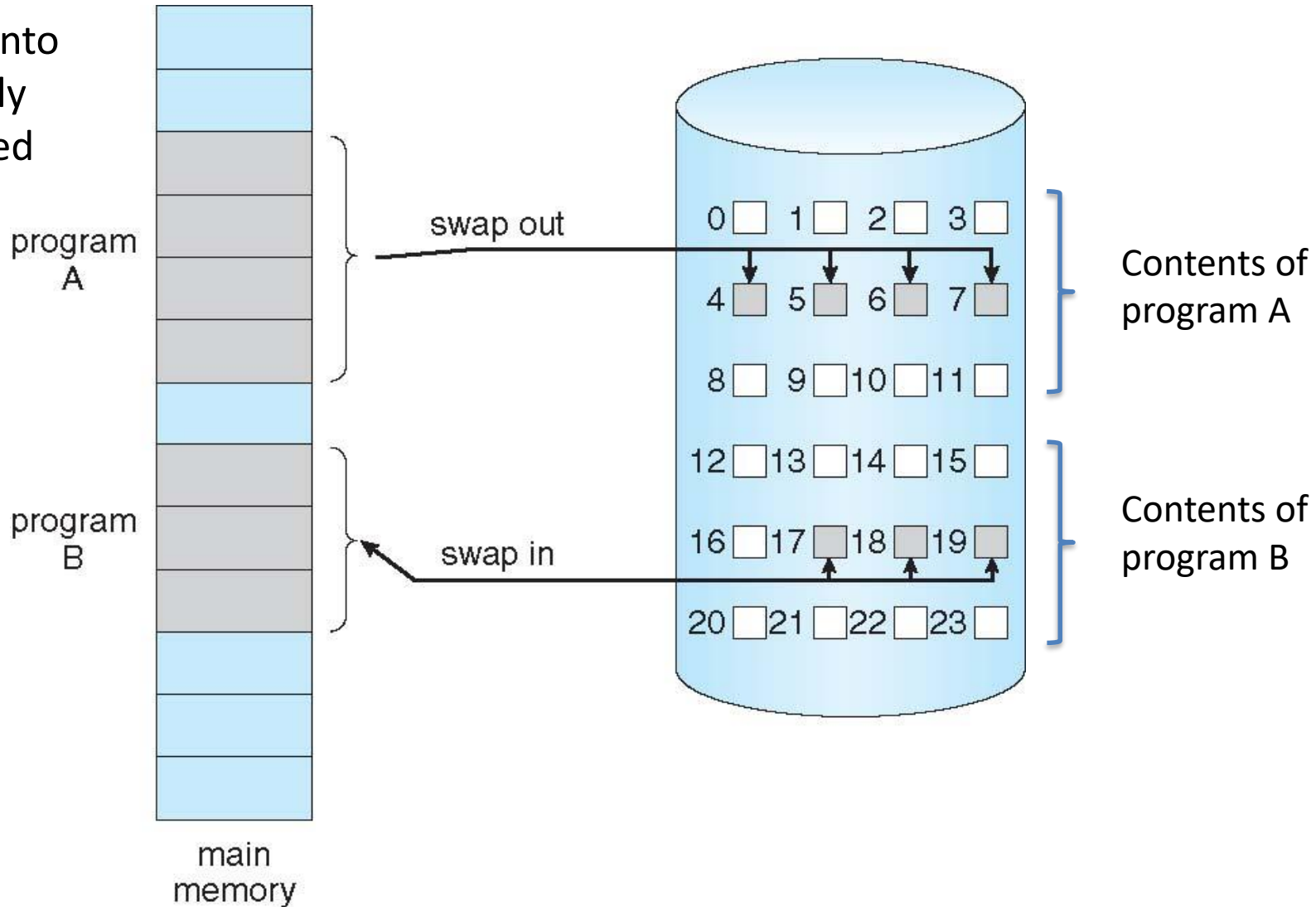
- Shorter process startup latency
 - Can start process without all of it in memory
 - Even 1 page suffices
- No longer constrained by limits of physical memory
- Takes less memory while running -> more programs run at the same time
 - Increased CPU utilization and throughput with no increase in response time
- Less I/O needed to load or swap programs into memory -> each user program runs faster

If the program is not in memory, then where is it?

- Part of it is in memory
- (Typically) all of it is on disk
- Note: difference with swapping
 - Swapping = all of program is in memory or all of program is on disk
 - Demand paging: part of program is in memory

Demand Paging

Bring page into memory only when needed



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?

Demand Paging

- What if program accesses part only on disk?
- Program is suspended
- OS runs, gets page from disk
- Program is restarted

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

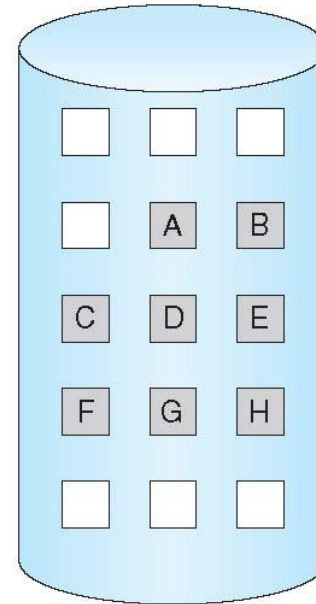
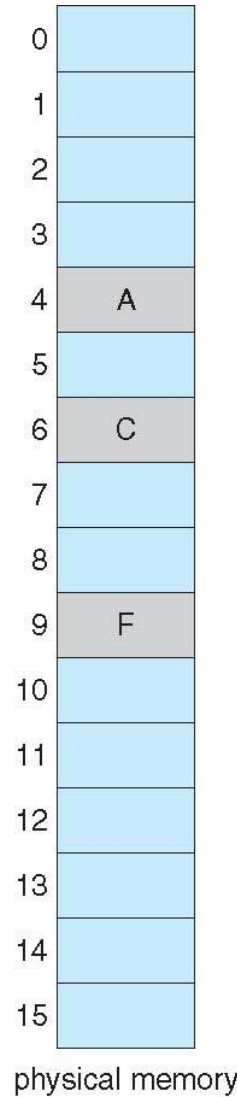
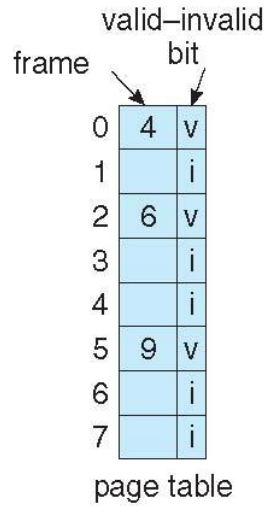
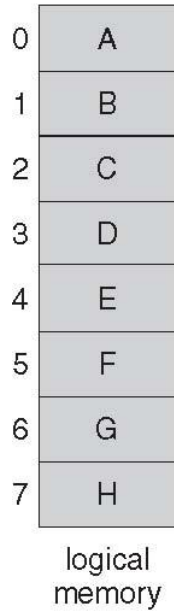
Issues

- How to discover a page fault?
- How to suspend process?
- How to bring in a page from disk?
 - How to find a free frame in memory?
- How to restart process?

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



Suspending the Faulting Process

- Invalid bit access generates trap
- As before, save process information in PCB

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
 - Suspend running process
 - Get back to page fault handling

Completing Page Fault Handling

- `Pagetable[pageno].frameno = new frameno`
- `Pagetable[pageno].valid = 1`

- Set process state to ready
- Invoke scheduler

When Process Runs Again

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

Remember this Assumption

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

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 has a modified bit
 - 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 Faults and Performance

- Normal memory access
 - ~ nanoseconds
- Faulting memory access
 - Disk I/O ~ 10 milliseconds
- Too many page faults -> program very slow
- Hence, importance of good page replacement

Page Replacement Policy

- In general, prefer replacing clean over dirty
- 1 disk I/O instead of 2

Page Replacement Policies

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

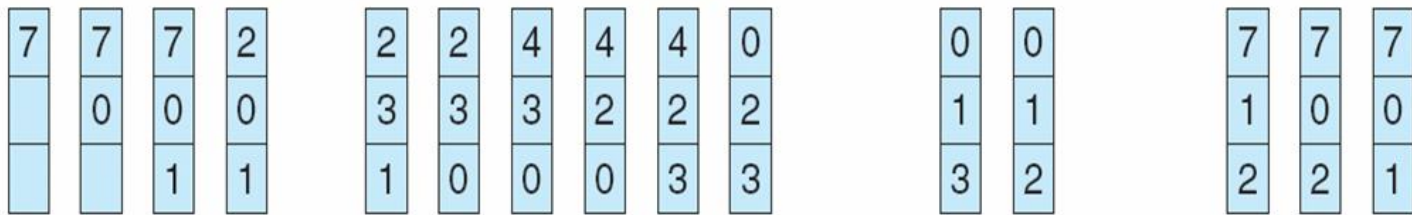
FIFO

- Oldest page is replaced
 - Age = Time since brought into memory
- Easy to implement
- Keep a queue of pages
- Bring in a page: stick at the end of the queue
- Need replacement: pick head of queue

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



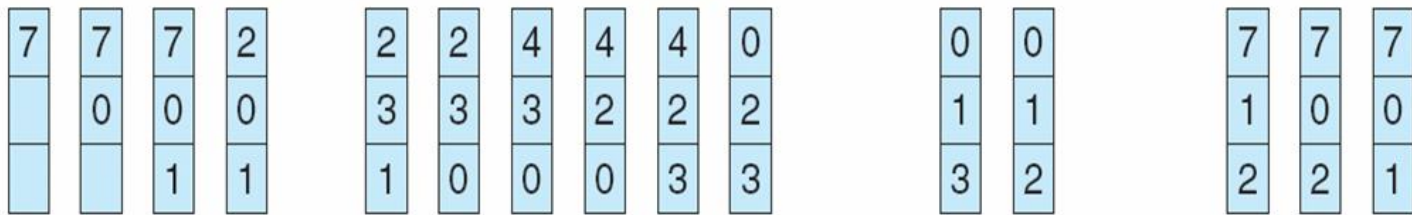
page frames

?? page faults (not counting initial paging in)

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



page frames

12 page faults (not counting initial paging in)

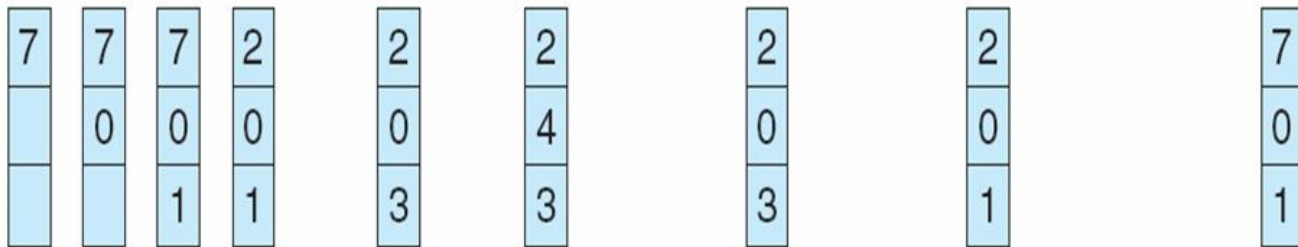
OPT: An Optimal Algorithm

- Replace the page that will be referenced the furthest in the future

Optimal Page Replacement

reference string

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



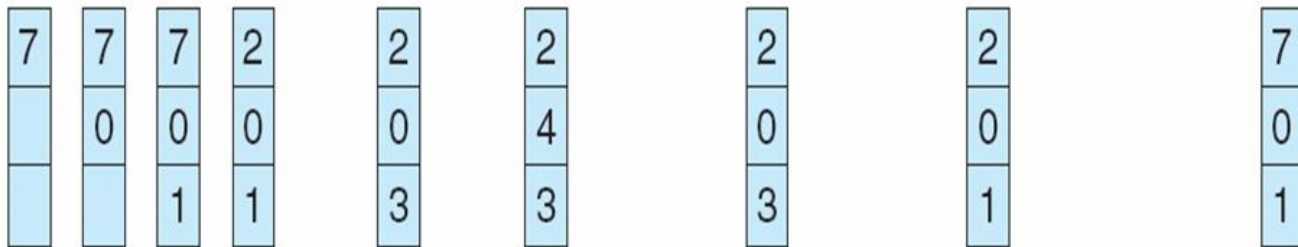
page frames

?? page faults (not counting initial paging in)

Optimal Page Replacement

reference string

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



page frames

6 page faults (not counting initial paging in)

OPT Implementation

- Cannot be implemented in general
 - Cannot predict future
- A basis of comparison for other algorithms

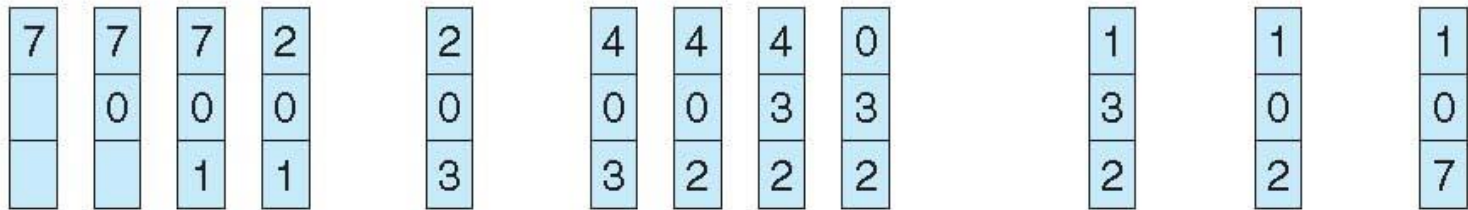
LRU: Least Recently Used

- 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



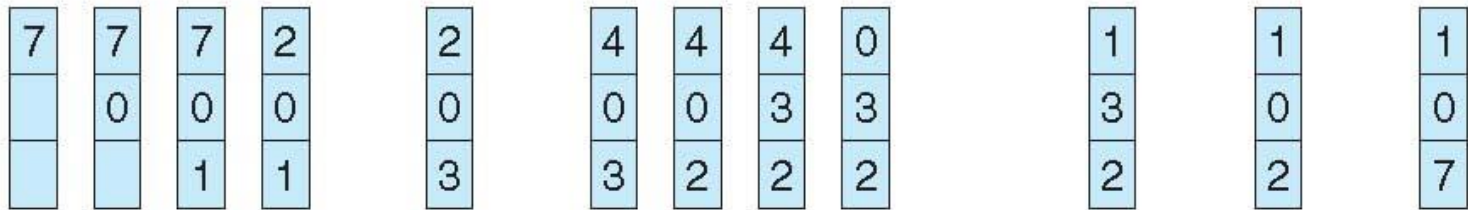
page frames

?? page faults (not counting initial paging in)

LRU

reference string

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



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

Some Optimizations

- Prepaging
- Cleaning
- Free frame pool

Prepaging

- So far: page in 1 page at time
- Prepaging: page in multiple pages at a time
- Usually, pages “surrounding” faulting page
- Relies on locality of virtual memory access
 - Nearby pages are often accessed soon after
- Avoids multiple page faults, process switches, ..
- Can also get better disk performance (later)

Cleaning

- So far: prefer to replace “clean” pages
- Cleaning: when disk is not busy, start writing out “dirty” pages
- More “clean” pages at replacement time

Free Frame Pool

- So far: use all possible frames for pages
- Free pool: keep some frames unused
- Page replacement when disk idle to keep free pool
- Advantage:
 - Page fault handling is quick (no disk I/O)
 - Disk I/O in background

Summary

- Demand paging
- Page fault handling
- Page replacement algorithms
- Optimizations