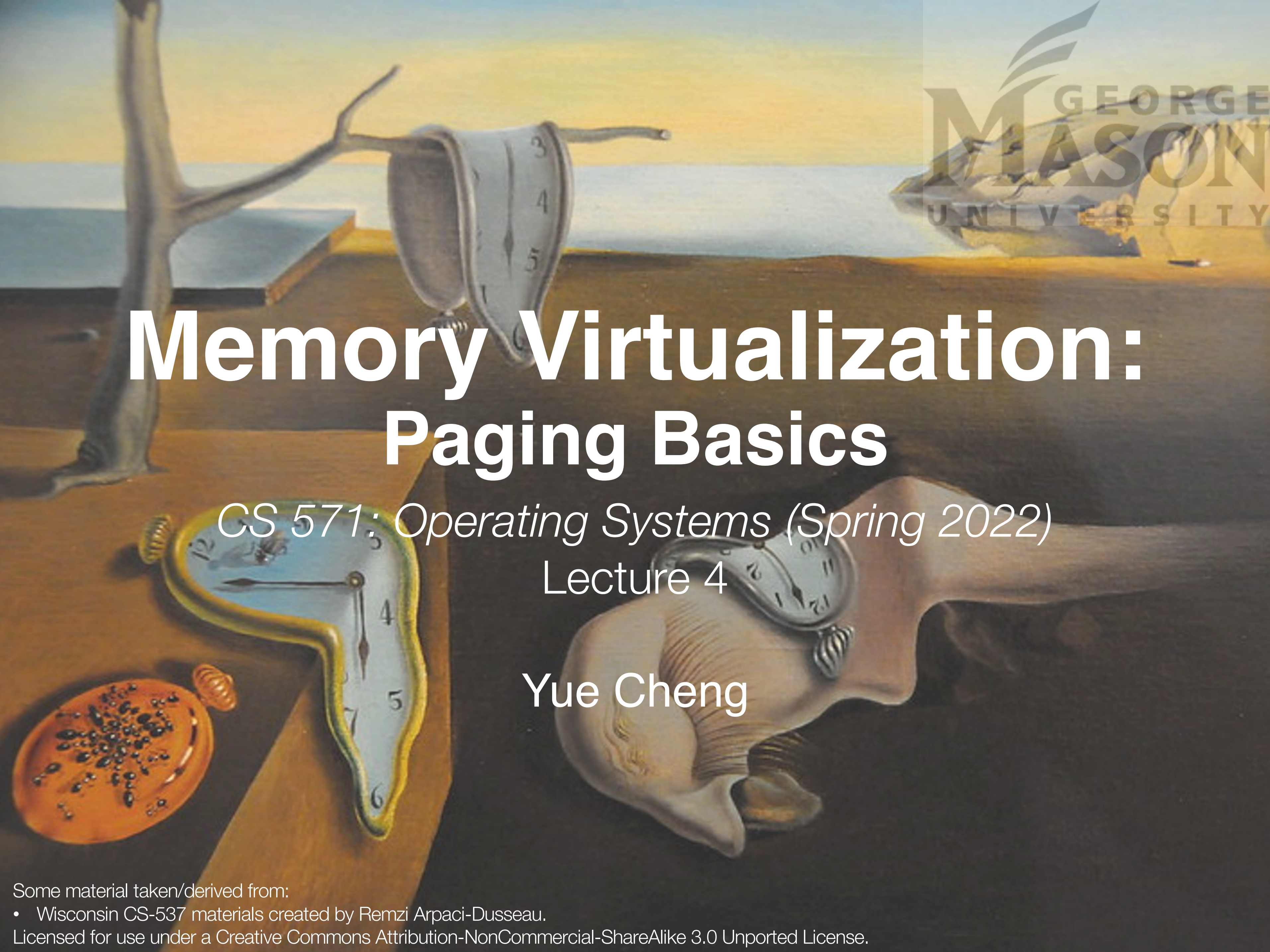


Memory Virtualization:
Paging Basics

CS 571: Operating Systems (Spring 2022)

Lecture 4

Yue Cheng

Some material taken/derived from:
- Wisconsin CS-537 materials created by Remzi Arpaci-Dusseau.

Licensed for use under a Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License.

Announcement

- Project proposal due by next Friday, 11:59pm, 02/25
    - https://tddg.github.io/cs571-spring22/project.html

- Read the ARC paper for Week 5's lecture

  ARC: A Self-Tuning, Low Overhead Replacement Cache

Today's outline

1. Address space and memory accesses

2. Segmentation and paging

3. Paging problems
    - PTs are too slow: Translation lookaside buffer (TLB)
    - PTs are too big: Small tables

All Memory Addresses You See are Virtual

Any address that a programmer can see is a virtual address

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]) {
    printf("location of code  : %p\n", (void *) main);
    printf("location of heap  : %p\n", (void *) malloc(1));
    int x = 3;
    printf("location of stack : %p\n", (void *) &x);
    return x;
}
```

Result:

```
location of code  : 0x1095afe50
location of heap  : 0x1096008c0
location of stack : 0x7fff691aea64
```

Virtual Memory Accesses

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    int x;
    x = x + 2;
}
```

```
_main:
0000000100000fa0 pushq %rbp
0000000100000fa1 movq  %rsp, %rbp
0000000100000fa4 xorl  %eax, %eax
0000000100000fa6 movl  %edi, -0x4(%rbp)
0000000100000fa9 movq  %rsi, -0x10(%rbp)
0000000100000fad movl  0x8(%rbp), %edi
0000000100000fb0 addl  $0x2, %edi
0000000100000fb3 movl  %edi, 0x8(%rbp)
0000000100000fb6 popq  %rbp
0000000100000fb7 retq
```

```
% objdump -S demo
```

Virtual Memory Accesses

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    int x;
    x = x + 2;
}
```

```
_main:
0000000100000fa0 pushq %rbp
0000000100000fa1 movq  %rsp, %rbp
0000000100000fa4 xorl  %eax, %eax
0000000100000fa6 movl  %edi, -0x4(%rbp)
0000000100000fa9 movq  %rsi, -0x10(%rbp)
0000000100000fad movl  0x8(%rbp), %edi
0000000100000fb0 addl  $0x2, %edi
0000000100000fb3 movl  %edi, 0x8(%rbp)
0000000100000fb6 popq  %rbp
0000000100000fb7 retq
```

```
% objdump -S demo
```

Virtual Memory Accesses

```
%rip = 0x100000fad
%rbp = 0x200
```

Memory accesses:

```
0x100000fad movl 0x8(%rbp), %edi
0x100000fb0 addl $0x2, %edi
0x100000fb3 movl %edi, 0x8(%rbp)
```

Virtual Memory Accesses

```
%rip = 0x100000fad
%rbp = 0x200
```

Memory accesses:

Fetch instr. at addr 0x100000fad

```
0x100000fad movl 0x8(%rbp), %edi
0x100000fb0 addl $0x2, %edi
0x100000fb3 movl %edi, 0x8(%rbp)
```

Virtual Memory Accesses

%rsp

```
%rip = 0x100000fad
%rbp = 0x200
```

offset.

```
0x100000fad movl 0x8(%rbp), %edi
0x100000fb0 addl $0x2, %edi
0x100000fb3 movl %edi, 0x8(%rbp)
```

Memory accesses:

Fetch instr. at addr 0x100000fad

Exec, load from addr 0x208

Virtual Memory Accesses

```
%rip = 0x100000fb0
%rbp = 0x200
```

Memory accesses:

Fetch instr. at addr 0x100000fad
Exec, load from addr 0x208

```
0x100000fad movl 0x8(%rbp), %edi
0x100000fb0 addl $0x2, %edi
0x100000fb3 movl %edi, 0x8(%rbp)
```

Virtual Memory Accesses

```
%rip = 0x100000fb0
%rbp = 0x200
```

Memory accesses:

Fetch instr. at addr 0x100000fad
Exec, load from addr 0x208

Fetch instr. at addr 0x100000fb0

```
0x100000fad movl 0x8(%rbp), %edi
0x100000fb0 addl $0x2, %edi
0x100000fb3 movl %edi, 0x8(%rbp)
```

Virtual Memory Accesses

```
%rip = 0x100000fb0
%rbp = 0x200
```

```
0x100000fad movl 0x8(%rbp), %edi
0x100000fb0 addl $0x2, %edi
0x100000fb3 movl %edi, 0x8(%rbp)
```

Memory accesses:

Fetch instr. at addr 0x100000fad
Exec, load from addr 0x208

Fetch instr. at addr 0x100000fb0
Exec, no load

Virtual Memory Accesses

```
%rip = 0x100000fb3
%rbp = 0x200
```

Memory accesses:

Fetch instr. at addr 0x100000fad
Exec, load from addr 0x208

Fetch instr. at addr 0x100000fb0
Exec, no load

```
0x100000fad movl 0x8(%rbp), %edi
0x100000fb0 addl $0x2, %edi
0x100000fb3 movl %edi, 0x8(%rbp)
```

Virtual Memory Accesses

```
%rip = 0x100000fb3
%rbp = 0x200
```

```
0x100000fad movl 0x8(%rbp), %edi
0x100000fb0 addl $0x2, %edi
0x100000fb3 movl %edi, 0x8(%rbp)
```

Memory accesses:

Fetch instr. at addr 0x100000fad
Exec, load from addr 0x208

Fetch instr. at addr 0x100000fb0
Exec, no load

Fetch instr. at addr 0x100000fb3

Virtual Memory Accesses

```
%rip = 0x100000fb3
%rbp = 0x200
```

```
0x100000fad movl 0x8(%rbp), %edi
0x100000fb0 addl $0x2, %edi
0x100000fb3 movl %edi, 0x8(%rbp)
```

Memory accesses:

Fetch instr. at addr 0x100000fad
Exec, load from addr 0x208

Fetch instr. at addr 0x100000fb0
Exec, no load

Fetch instr. at addr 0x100000fb3
Exec, store to addr 0x208

Virtual Memory Accesses

```
%rip = 0x100000fb3
%rbp = 0x200
```

```
0x100000fad movl 0x8(%rbp), %edi
0x100000fb0 addl $0x2, %edi
0x100000fb3 movl %edi, 0x8(%rbp)
```

Memory accesses:

Fetch instr. at addr 0x100000fad
Exec, load from addr 0x208

Fetch instr. at addr 0x100000fb0
Exec, no load

Fetch instr. at addr 0x100000fb3
Exec, store to addr 0x208

Q: How to relocate the memory access in a way that is transparent to the process?

Virtual Memory Accesses

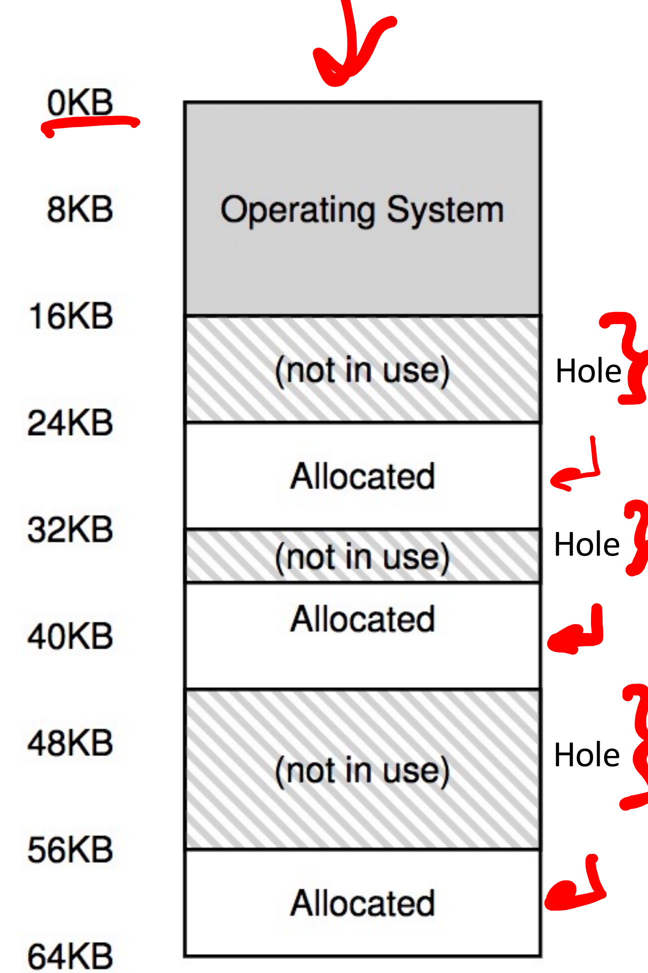
- Approaches:
  - Static Relocation
  - Dynamic Relocation
    - Base
    - Base-and-Bounds
  - Segmentation

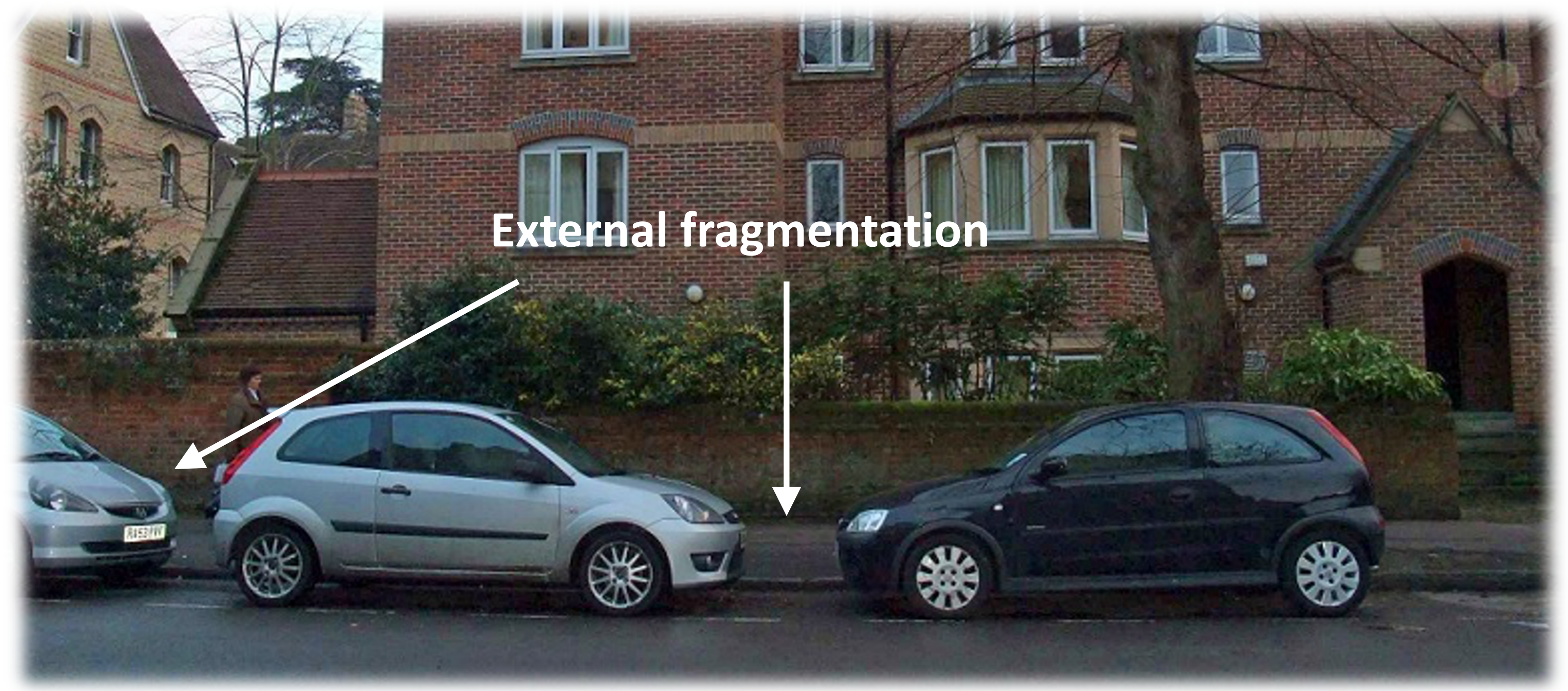
Virtual Memory Accesses

- Approaches:
  - Static Relocation: requires rewrite for the same code
  - Dynamic Relocation
    - Base: add a base to virtual address to get physical address
    - Base-and-Bounds: checks physical address is in range
  - Segmentation: many base+bounds pairs

Segmentation Issues: External Fragmentation

- As processes are loaded and removed from the main memory, the free memory is broken into small pieces
  - Hole: block of available memory; holes of various size are scattered throughout memory
- A new allocation request may have to be denied
  - When there is no contiguous free memory with requested size
  - The total free memory space may be much larger than the requested size!




External fragmentation

External fragmentation

Ideally, what we want…

Memory Compaction

- Reduce external fragmentation by `copy+compaction`
  - Shuffle memory contents to place all free memory together in one large block
  - Compaction is possible *only* if relocation is dynamic, and is done at execution time
  - Must be careful about pending I/O before initiating compaction
  - Problems
    - Too much perf overhead

Paging

Paging

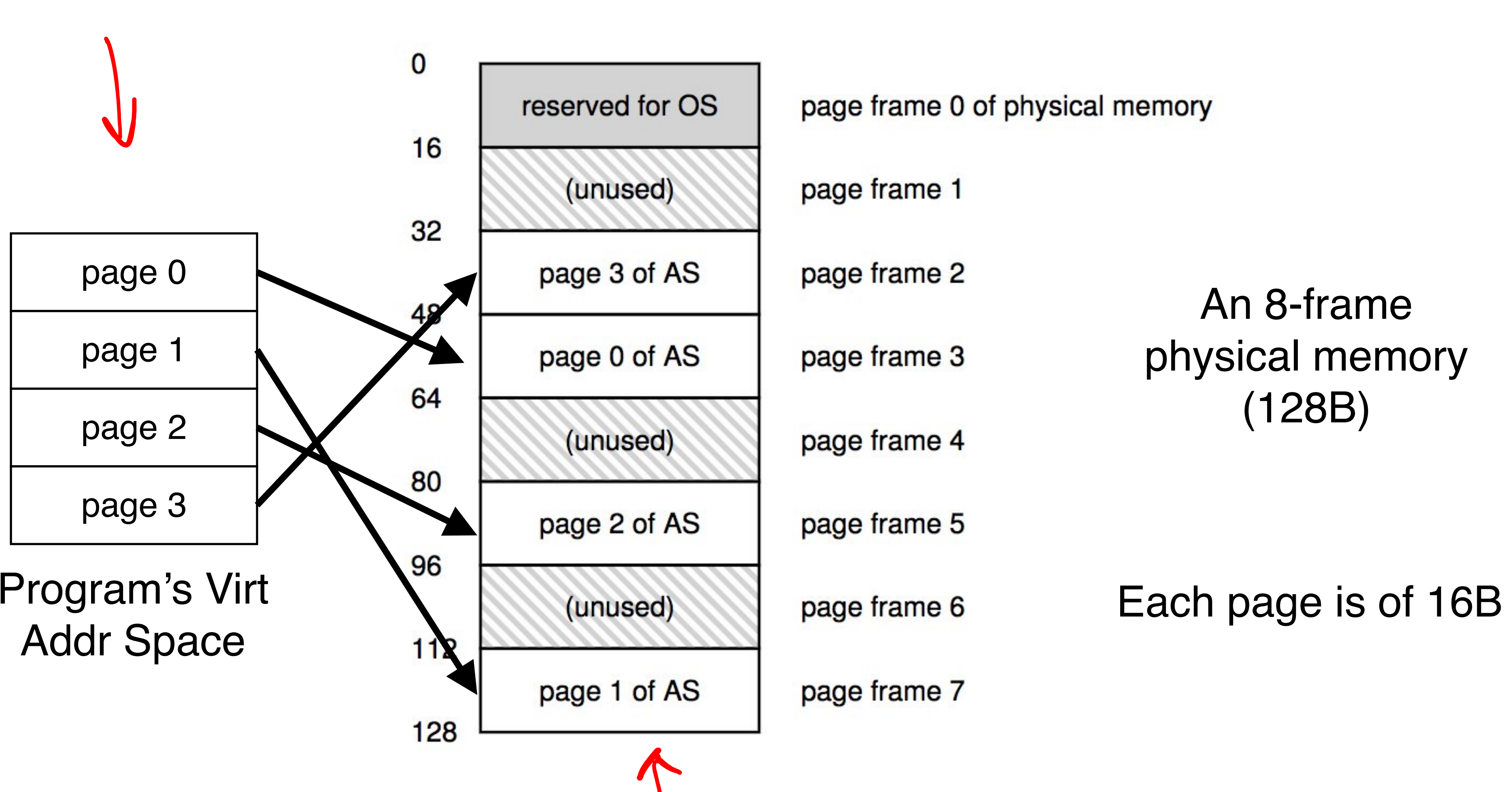
- Motivation: Segmentation is coarse-grained
  - Either waste space (external fragmentation) or
  - copy memory often (compaction)

- We need a finer-grained alternative!

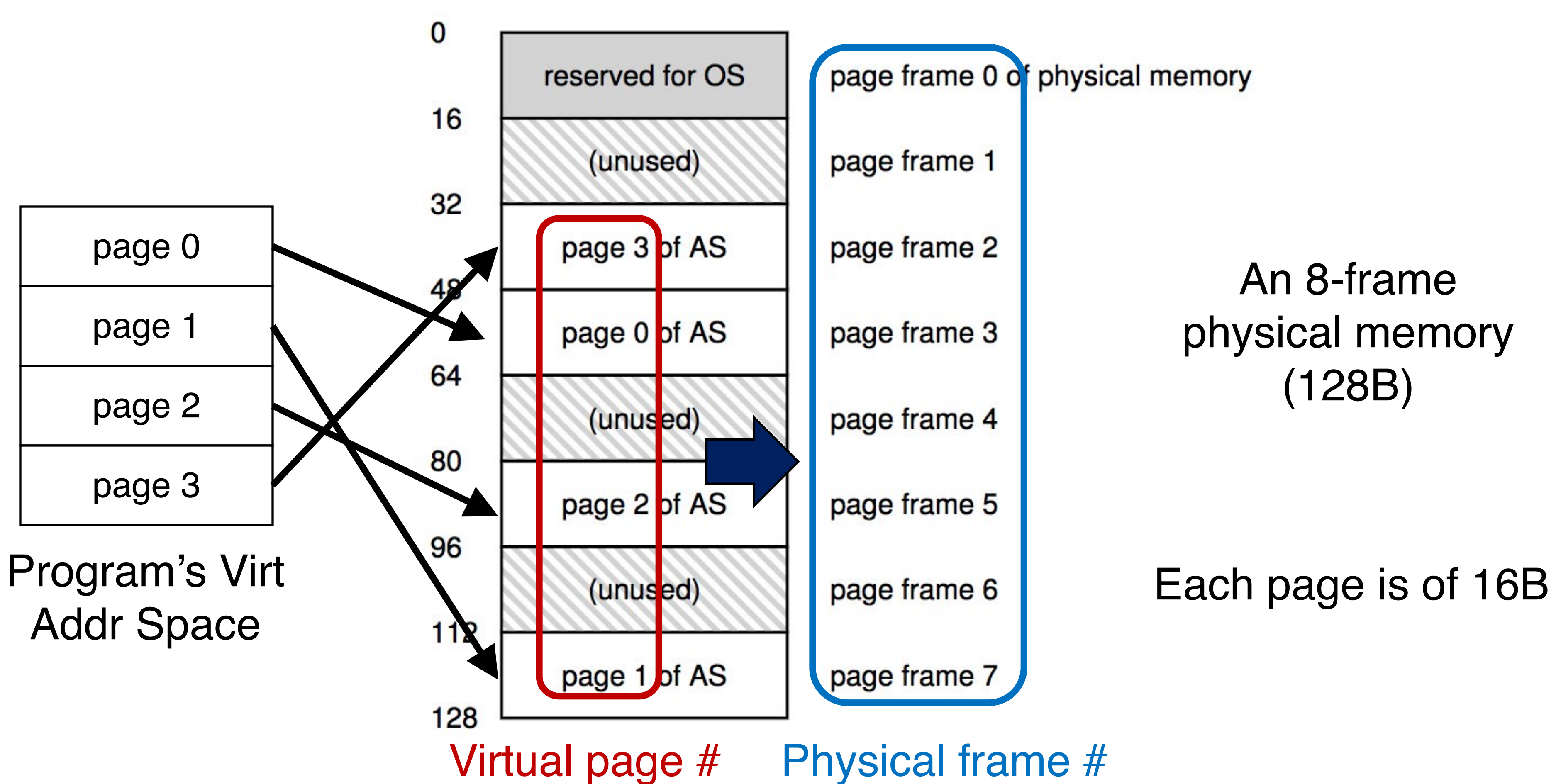
Paging Scheme

- A memory management scheme that allows the physical address space of a process to be non-contiguous
- Divide physical memory into fixed-sized blocks called **frames**
- Divide logical memory into blocks of same size called **pages**
- Flexible mapping: Any page can go to any free frame
- Scalability: To run a program of size *n* pages, need to find *n* free frames and load program
  - Grow memory segments wherever we please!

A Simple Example

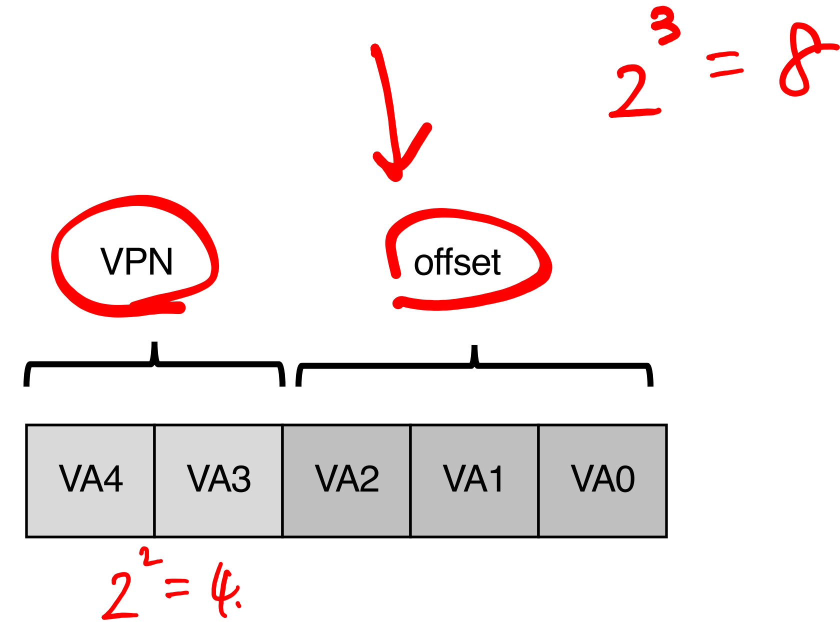


A Simple Example



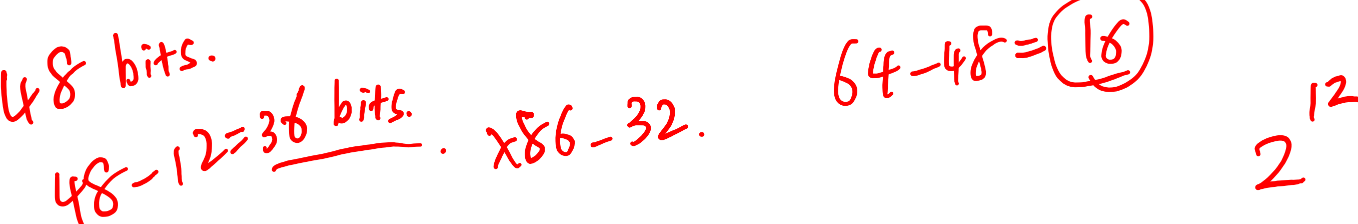
Addressing Basics

- For segmentation
  - High bits => segment #
  - Low bits => offset
- For paging
  - High bits => page #
  - Low bits => offset

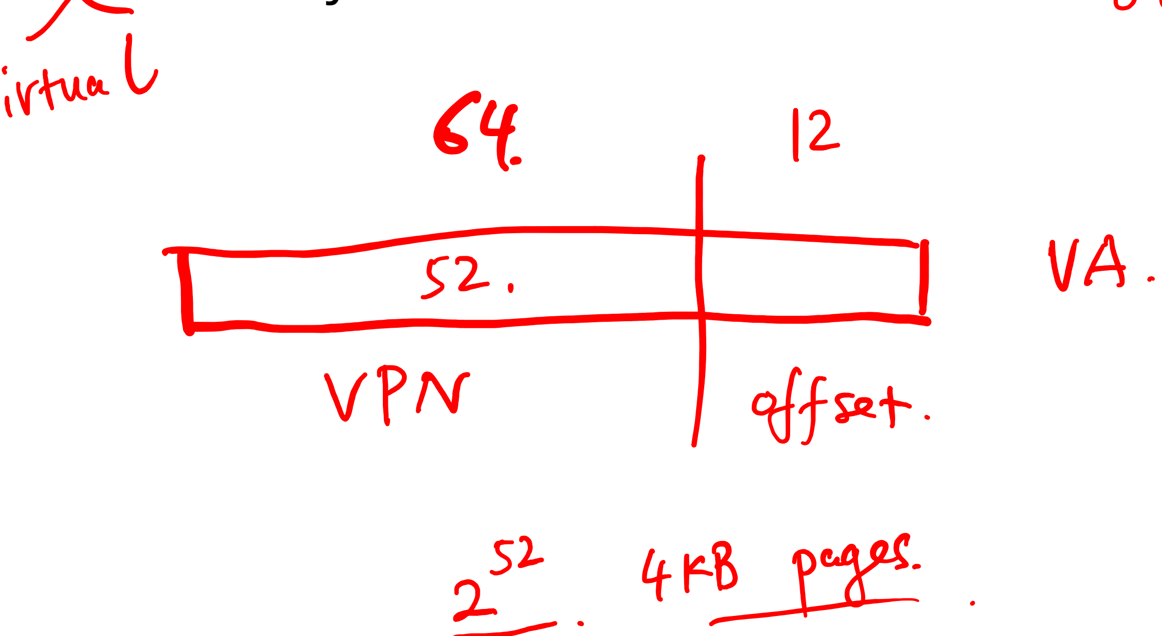


Q: How many offset bits do we need?

A: `log(page_size)`

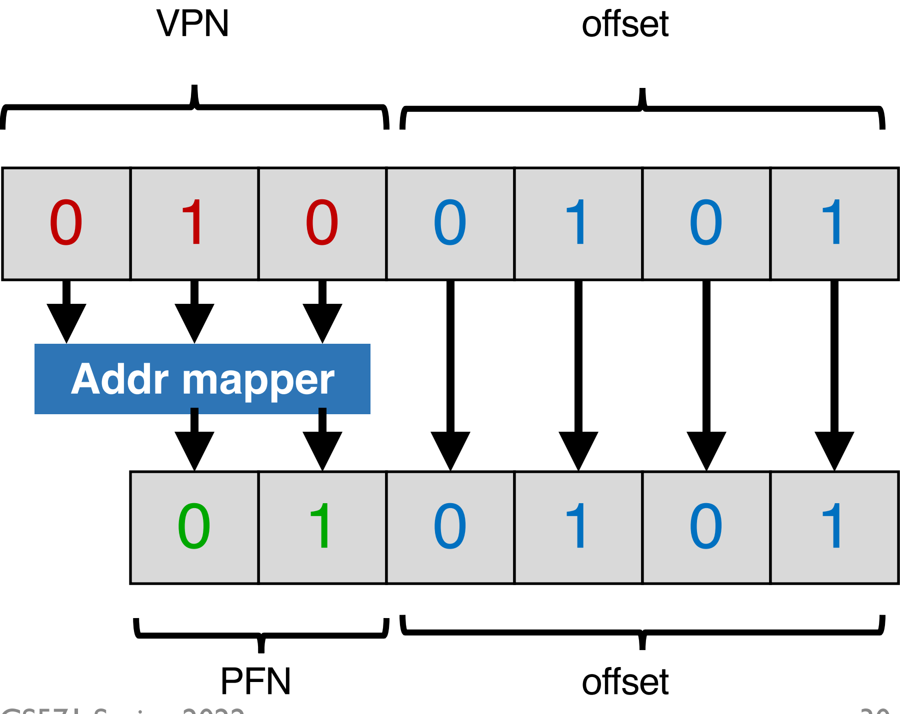


Question: An x86_64 Linux OS with 4KB page size. How many pages can we have assuming the maximum memory limit?



Virtual => Physical Addr Mapping

- We need a general mapping mechanism
- What data structure is good?
  - Big array



Virtual => Physical Addr Mapping

- We need a general mapping mechanism

- What data structure is good?
  - Big array
    (aka linear page table)

VPN offset

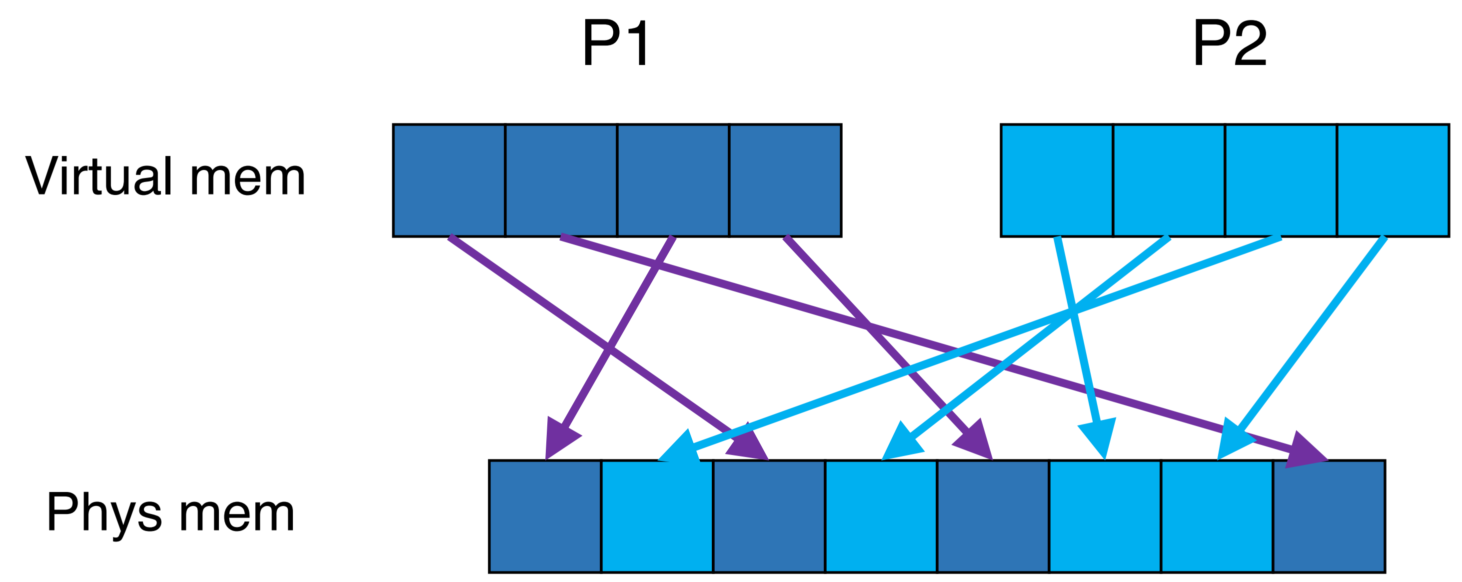
0 1 0 0 1 0 1

Addr mapper

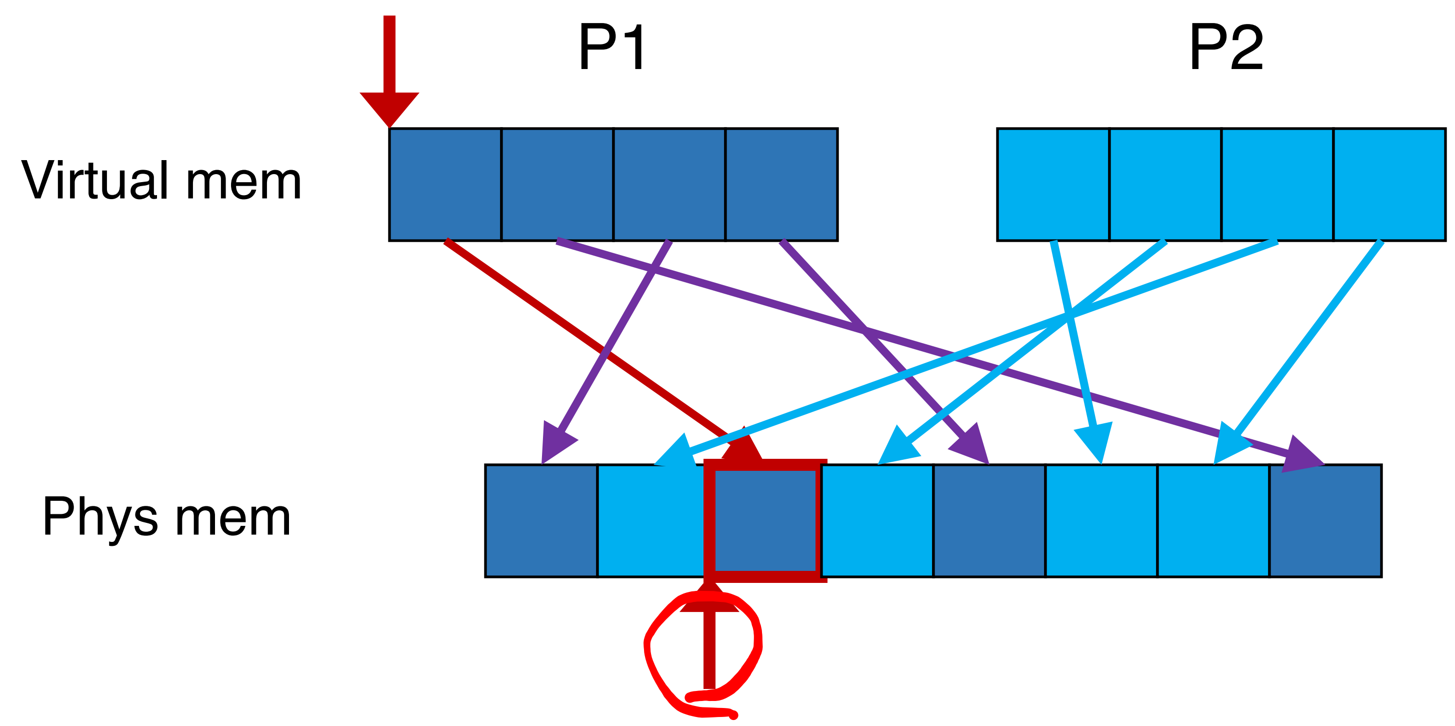
0 1 0 1 0 1

PFN offset

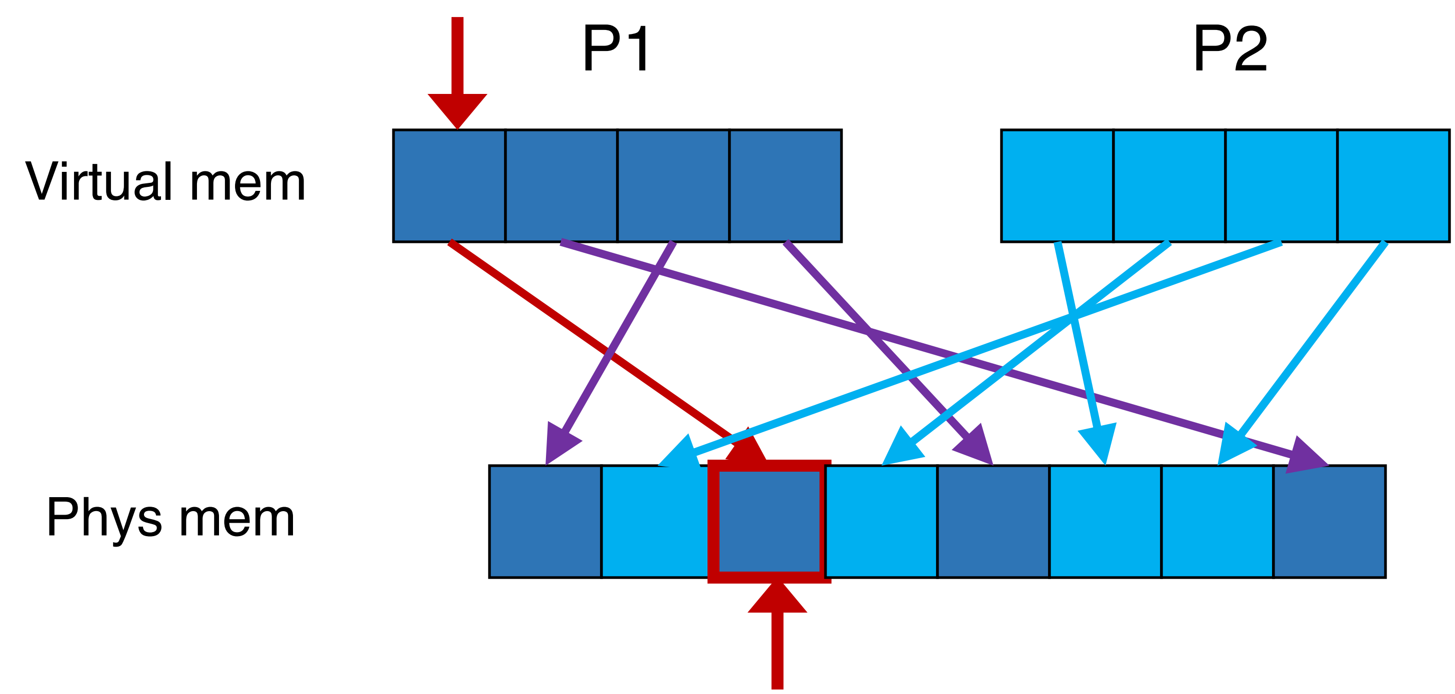
Mapping Example



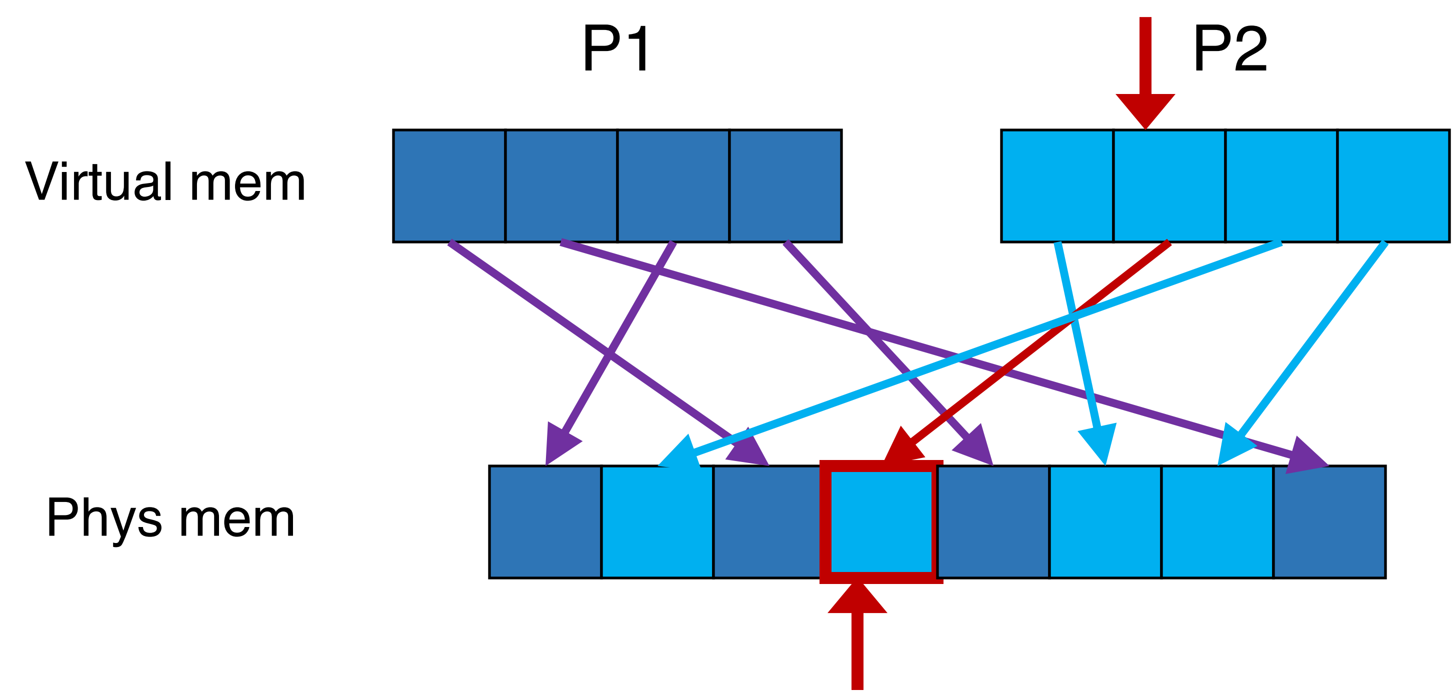
Mapping Example



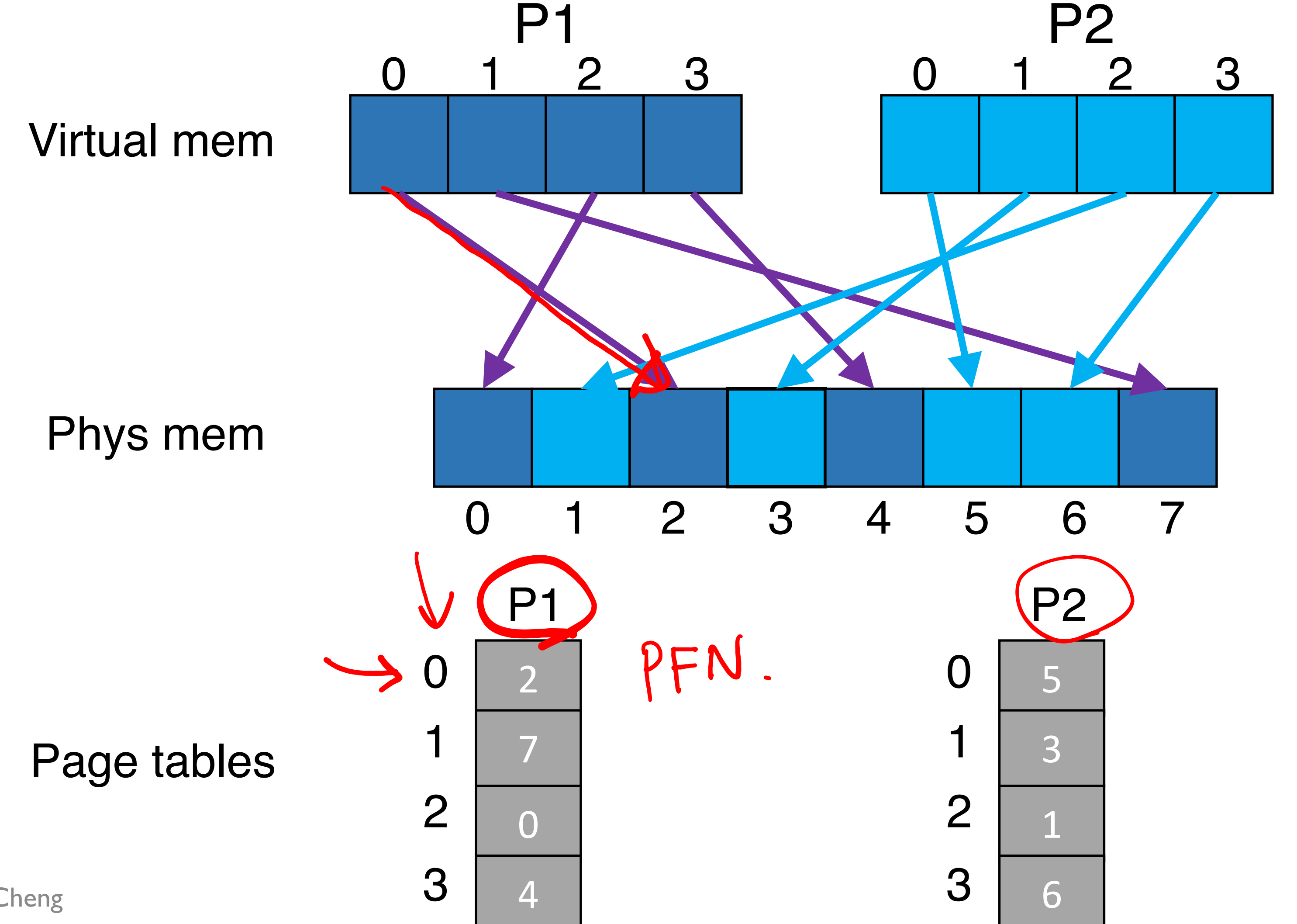
Mapping Example



Mapping Example



Mapping Example



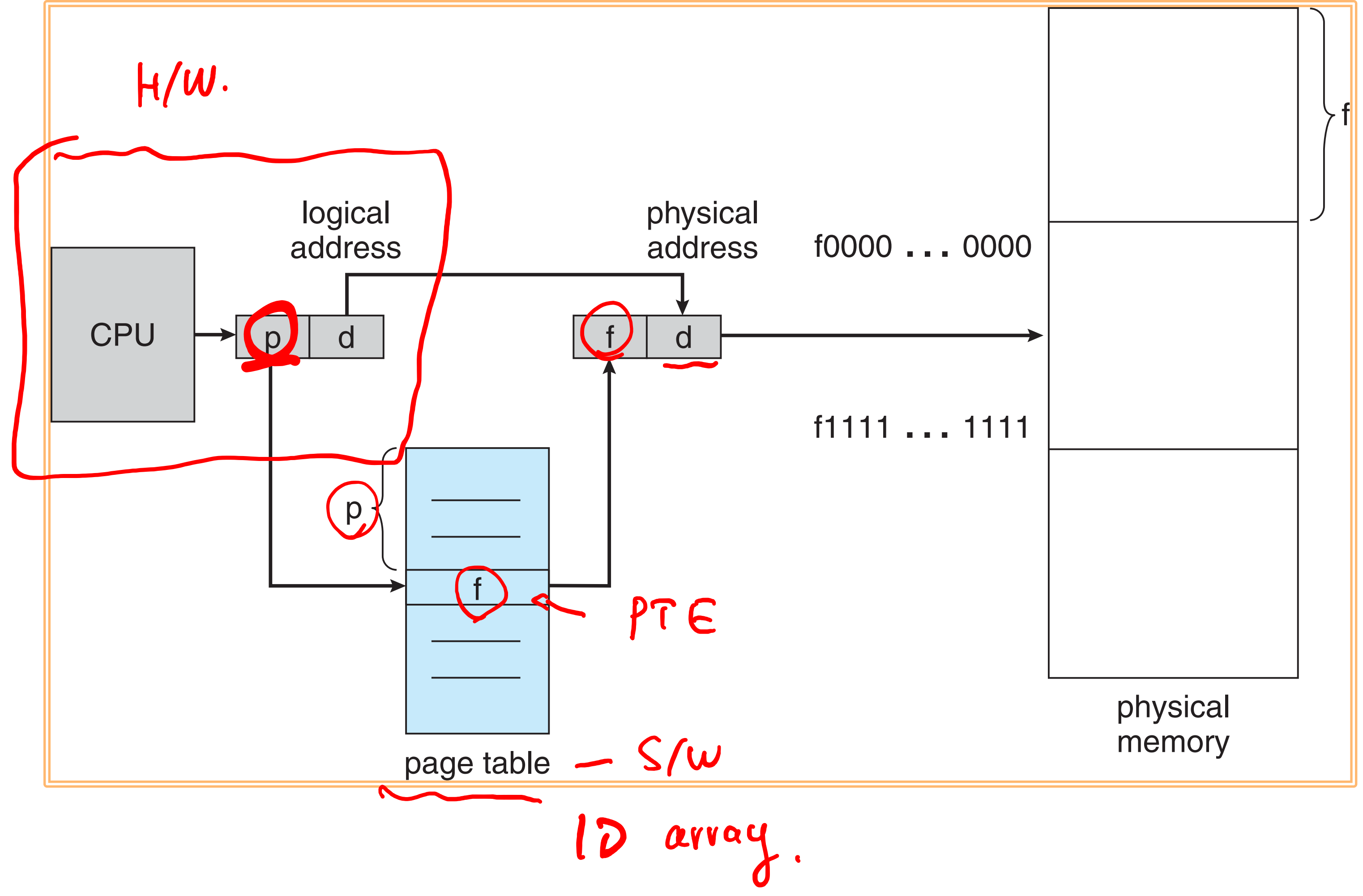
Page Table

- A per-process data structure used to keep track of virtual page to physical frame mapping
- Major role: store address translation

Address Translation Scheme

- Observe: The simple limit/relocation register pair mechanism is no longer sufficient
- m-bit virtual address generated by CPU is divided into:
  - Virtual Page number (p) – used as an index into a page table which contains base address of each page in physical memory
  - Page offset (d) – combined with base address to define the physical memory address that is sent to the memory unit

Address Translation Architecture



More on Page Table

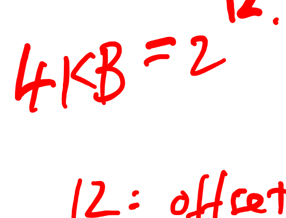
- The page table data structure is kept in main memory
- Each page table entry (PTE) holds

```
<physical translation + other info>
```

PFN.

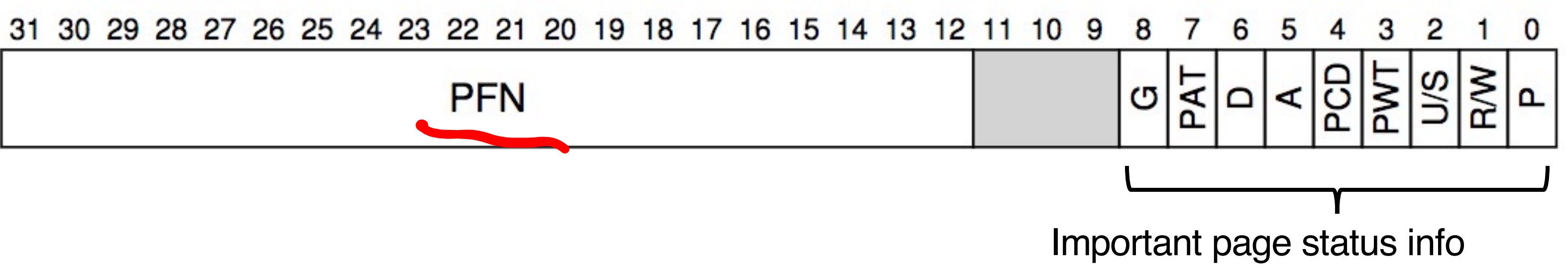
- Page-table base register (PTBR) points to the page table
  - E.g., CR3 on x86
- Page-table length register (PTLR), if it exists, indicates the size of the page table

Page Table Entry (PTE)



- The simplest form of a page table is a linear page table
    - Array data structure
    - OS indexes the array by virtual page number (VPN)
    - To find the desired physical frame number (PFN)

An 32-bit x86 page table entry (PTE)



Paging Problems

- Page tables are too slow

- Page tables are too big

Address Translation Steps

- Hardware (MMU): for each memory reference
  1. Extract VPN (virt page num) from VA (virt addr)
  2. Calculate addr of PTE (page table entry)
  3. Fetch PTE
  4. Extract PFN (phys page frame num)
  5. Build PA (phys addr)
  6. Fetch PA to register

Address Translation Steps

- Hardware (MMU): for each memory reference
  1. Extract VPN (virt page num) from VA (virt addr)
  2. Calculate addr of PTE (page table entry)
  3. Fetch PTE
  4. Extract PFN (phys page frame num)
  5. Build PA (phys addr)
  6. Fetch PA to register

- Q: Which steps are expensive??

Address Translation Steps

- Hardware (MMU): for each memory reference

cheap	1. Extract VPN (virt page num) from VA (virt addr)
cheap	2. Calculate addr of PTE (page table entry)
expensive	3. Fetch PTE
cheap	4. Extract PFN (phys page frame num)
cheap	5. Build PA (phys addr)
expensive	6. Fetch PA to register

- Q: Which steps are expensive??

Address Translation Steps

- Hardware (MMU): for each memory reference

cheap	1. Extract VPN (virt page num) from VA (virt addr)
cheap	2. Calculate addr of PTE (page table entry)
expensive	3. Fetch PTE
cheap	4. Extract PFN (phys page frame num)
cheap	5. Build PA (phys addr)
expensive	6. Fetch PA to register

- Q: Which expensive steps can we avoid??

Array Iterator

- A simple code snippet in array.c

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

- Compile it using gcc

```
prompt> gcc -o array array.c -Wall -O
prompt> ./array
```

- Dump the assembly code
  - `objdump` (Linux) or `otool` (Mac)

Trace the Memory Accesses

Virt

load 0x3000

load 0x3004

load 0x3008

load 0x300C

…

Trace the Memory Accesses

Virt	Phys
load 0x3000	load 0x100C
	load 0x7000
load 0x3004	load 0x100C
	load 0x7004
load 0x3008	load 0x100C
	load 0x7008
load 0x300C	load 0x100C
…	load 0x700C

Trace the Memory Accesses

Virt	Phys
load 0x3000	load 0x100C
	load 0x7000
load 0x3004	load 0x100C
	load 0x7004
load 0x3008	load 0x100C
	load 0x7008
load 0x300C	load 0x100C
…	load 0x700C

1st mem access: Fetch PTE

Trace the Memory Accesses

Virt	Phys
load 0x3000	load 0x100C
	load 0x7000
load 0x3004	load 0x100C
	load 0x7004
load 0x3008	load 0x100C
	load 0x7008
load 0x300C	load 0x100C
…	load 0x700C

Map VPN to PFN: 3 → 7

Trace the Memory Accesses

Virt	Phys
load 0x3000	load 0x100C
	load 0x7000
load 0x3004	load 0x100C
	load 0x7004
load 0x3008	load 0x100C
	load 0x7008
load 0x300C	load 0x100C
…	load 0x700C

2nd mem access: access **a[i]**

Trace the Memory Accesses

Virt	Phys
load 0x3000	load 0x100C
	load 0x7000
load 0x3004	load 0x100C
	load 0x7004
load 0x3008	load 0x100C
	load 0x7008
load 0x300C	load 0x100C
…	load 0x700C

Note: 1. Each virt mem access → **two** phys mem accesses
2. **Repeated** memory accesses!

Translation Lookaside Buffer (TLB)

Performance Problems of Paging

- A basic memory access protocol
  1. Fetch the translation from in-memory page table
  2. Explicit load/store access on a memory address

- In this scheme every data/instruction access requires two memory accesses
  - One for the page table
  - and one for the data/instruction

- Too much performance overhead!

Speeding up Translation

- The two memory access problem can be solved by the use of a special fast-lookup hardware cache called translation lookaside buffer (TLB)
- A TLB is part of the memory-management unit (MMU)
- A TLB is a **hardware** cache
- Algorithm sketch
  - For each virtual memory reference, hardware first checks the TLB to see if the desired translation is held therein

TLB Basic Algorithm

1. Extract VPN from VA

TLB Basic Algorithm

1. Extract VPN from VA
2. Check if TLB holds the translation

TLB Basic Algorithm

1. Extract VPN from VA
2. Check if TLB holds the translation
3. If it is a TLB hit – extract PFN from the TLB entry, concatenate it onto the offset to form the PA

TLB Basic Algorithm

1. Extract VPN from VA
2. Check if TLB holds the translation
3. If it is a TLB hit – extract PFN from the TLB entry, concatenate it onto the offset to form the PA

TLB Basic Algorithm

1. Extract VPN from VA
2. Check if TLB holds the translation
3. If it is a TLB hit – extract PFN from the TLB entry, concatenate it onto the offset to form the PA
4. If it is a TLB miss – access *page table* to get the translation, update the TLB entry with the translation

TLB Basic Algorithm

1. Extract VPN from VA
2. Check if TLB holds the translation
3. If it is a TLB hit – extract PFN from the TLB entry, concatenate it onto the offset to form the PA

4. If it is a TLB miss – access *page table* to get the translation, update the TLB entry with the translation

Array Iterator (w/ TLB)

```
int sum = 0;
for (i=0; i<1024; i++) {
    sum += a[i];
}
```

Trace the Memory Accesses (w/ TLB)

Virt

load 0x3000

load 0x3004

load 0x3008

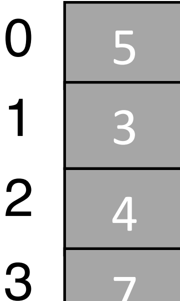
load 0x300C

…

Trace the Memory Accesses (w/ TLB)

4KB. → 12.

P1's page table

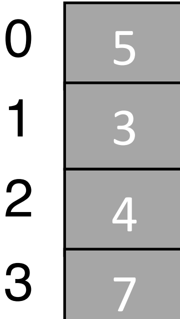


Virt	Phys
load 0x3000	
load 0x3004	
load 0x3008	
load 0x300C	
…	

CPU's TLB cache

Valid	Virt	Phys
0		
0		
0		
0		

Trace the Memory Accesses (w/ TLB)



Virt	Phys
load 0x3000	
load 0x3004	
load 0x3008	
load 0x300C	
…	

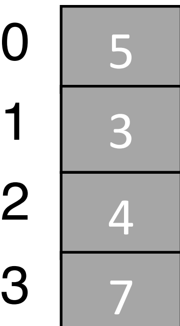
CPU's TLB cache

Miss

Valid	Virt	Phys
0		
0		
0		
0		

Trace the Memory Accesses (w/ TLB)

0
1
2
3



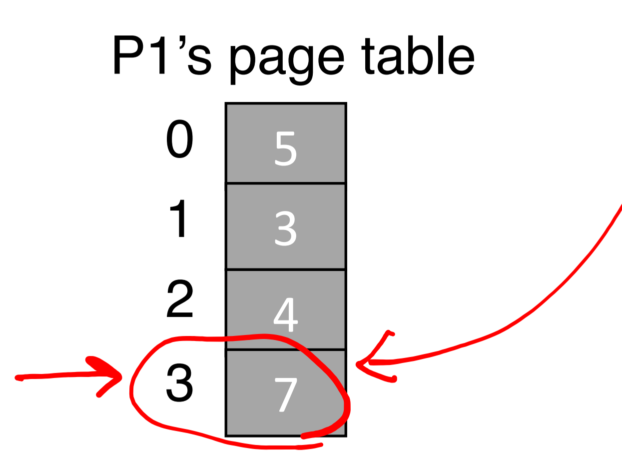
Virt	Phys
load 0x3000	load 0x100C
load 0x3004	
load 0x3008	
load 0x300C	
…	

CPU's TLB cache

Valid	Virt	Phys
0		
0		
0		
0		

Miss

Trace the Memory Accesses (w/ TLB)

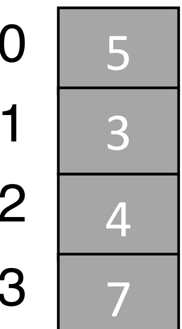


Virt	Phys
load 0x3000	load 0x100C
load 0x3004	
load 0x3008	
load 0x300C	
…	

CPU's TLB cache

Valid	Virt	Phys
1	3	7
0		
0		
0		

Trace the Memory Accesses (w/ TLB)



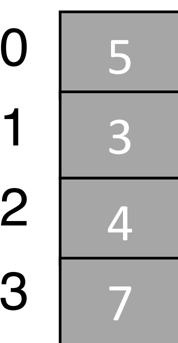
CPU's TLB cache

Valid	Virt	Phys
1	3	7
0		
0		
0		

Virt	Phys
load 0x3000	load 0x100C load 0x7000
load 0x3004	
load 0x3008	
load 0x300C	
…	

Trace the Memory Accesses (w/ TLB)

P1's page table

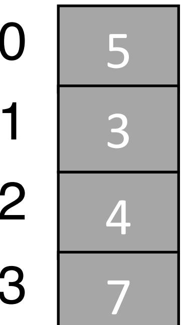


Virt	**Phys**
load 0x3000	load 0x100C
	load 0x7000
load 0x3004	
load 0x3008	
load 0x300C	
…	

CPU's TLB cache

Valid	Virt	Phys
1	3	7
0		
0		
0		

Trace the Memory Accesses (w/ TLB)



CPU's TLB cache

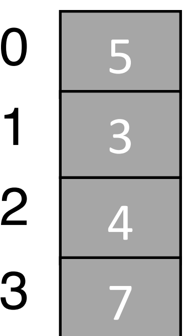
Valid	Virt	Phys
1	3	7
0		
0		
0		

Hit → (row 1)

Virt	Phys
load 0x3000	load 0x100C load 0x7000
load 0x3004	(TLB hit)
load 0x3008	
load 0x300C	
…	

Trace the Memory Accesses (w/ TLB)

P1's page table



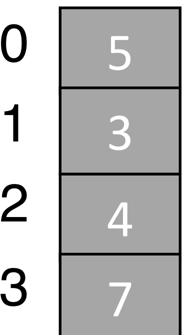
Virt	Phys
load 0x3000	load 0x100C
	load 0x7000
load 0x3004	(TLB hit)
	→ load 0x7004
load 0x3008	
load 0x300C	
…	

CPU's TLB cache

Valid	Virt	Phys
1	3	7
0		
0		
0		

Trace the Memory Accesses (w/ TLB)

P1's page table



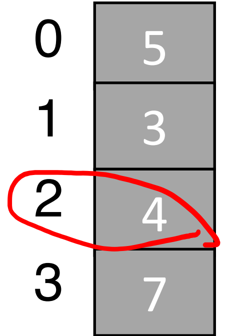
CPU's TLB cache

Valid	Virt	Phys
1	3	7
0		
0		
0		

Virt	Phys
load 0x3000	load 0x100C
	load 0x7000
load 0x3004	(TLB hit)
	load 0x7004
load 0x3008	(TLB hit)
	load 0x7008
load 0x300C	(TLB hit)
…	load 0x700C

Trace the Memory Accesses (w/ TLB)

P1's page table



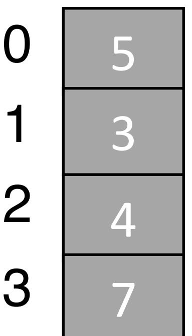
CPU's TLB cache

Valid	Virt	Phys
1	3	7
0		
0		
0		

Virt	Phys
load 0x3000	load 0x100C load 0x7000
load 0x3004	(TLB hit) load 0x7004
load 0x3008	(TLB hit) load 0x7008
load 0x300C	(TLB hit) load 0x700C
…	
load 0x2000	

Trace the Memory Accesses (w/ TLB)

P1's page table



Virt	Phys
load 0x3000	load 0x100C
	load 0x7000
load 0x3004	(TLB hit)
	load 0x7004
load 0x3008	(TLB hit)
	load 0x7008
load 0x300C	(TLB hit)
	load 0x700C
…	
load 0x2000	load 0x100F

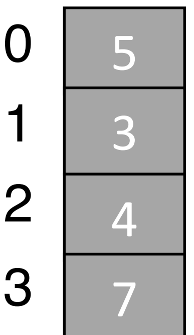
CPU's TLB cache

Valid	Virt	Phys
1	3	7
1	2	4
0		
0		

Miss

Trace the Memory Accesses (w/ TLB)

P1's page table

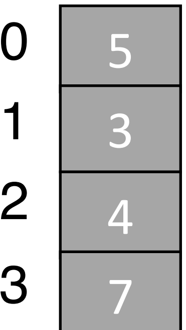


CPU's TLB cache

Valid	Virt	Phys
1	3	7
1	2	4
0		
0		

Virt	Phys
load 0x3000	load 0x100C
	load 0x7000
load 0x3004	(TLB hit)
	load 0x7004
load 0x3008	(TLB hit)
	load 0x7008
load 0x300C	(TLB hit)
	load 0x700C
…	load 0x100F
load 0x2000	load 0x4000

Trace the Memory Accesses (w/ TLB)

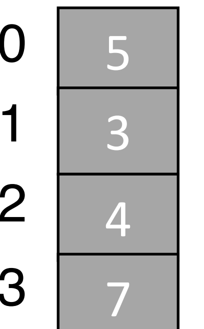


CPU's TLB cache

Valid	Virt	Phys
1	3	7
1	2	4
0		
0		

Virt	Phys
load 0x3000	load 0x100C
	load 0x7000
load 0x3004	(TLB hit)
	load 0x7004
load 0x3008	(TLB hit)
	load 0x7008
load 0x300C	(TLB hit)
…	load 0x700C
load 0x2000	load 0x100F
	load 0x4000
load 0x2004	

Trace the Memory Accesses (w/ TLB)



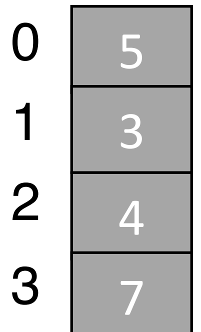
CPU's TLB cache

Valid	Virt	Phys
1	3	7
1	2	4
0		
0		

Hit →

Virt	Phys
load 0x3000	load 0x100C
	load 0x7000
load 0x3004	(TLB hit)
	load 0x7004
load 0x3008	(TLB hit)
	load 0x7008
load 0x300C	(TLB hit)
…	load 0x700C
load 0x2000	load 0x100F
	load 0x4000
load 0x2004	(TLB hit)

Trace the Memory Accesses (w/ TLB)



CPU's TLB cache

Valid	Virt	Phys
1	3	7
1	2	4
0		
0		

Virt	Phys
load 0x3000	load 0x100C load 0x7000
load 0x3004	(TLB hit) load 0x7004
load 0x3008	(TLB hit) load 0x7008
load 0x300C	(TLB hit) load 0x700C
…	
load 0x2000	load 0x100F load 0x4000
load 0x2004	(TLB hit) load 0x4004

How Many TLB Lookups

Assume 4KB pages

```
int sum = 0;
for (i=0; i<1024; i++) {
    sum += a[i];
}
```

How Many TLB Lookups

Assume 4KB pages

```
int sum = 0;
for (i=0; i<1024; i++) {
    sum += a[i];
}
```

Array `a[]` has 1024 items, each item is 4 bytes:

`Size(a) = 4096`

How Many TLB Lookups

Assume 4KB pages

```
int sum = 0;
for (i=0; i<1024; i++) {
    sum += a[i];
}
```

Array `a[]` has 1024 items, each item is 4 bytes:

`Size(a) = 4096`

Num of TLB miss: `4096/4096 = 1 or 2`

How Many TLB Lookups

Assume 4KB pages

```
int sum = 0;
for (i=0; i<1024; i++) {
    sum += a[i];
}
```

Array `a[]` has 1024 items, each item is 4 bytes:

`Size(a) = 4096`

Best case: Num of TLB miss: `4096/4096 = 1`

TLB miss rate: `1/1024 = 0.09%`

How Many TLB Lookups

Assume 4KB pages

```
int sum = 0;
for (i=0; i<1024; i++) {
    sum += a[i];
}
```

Array `a[]` has 1024 items, each item is 4 bytes:

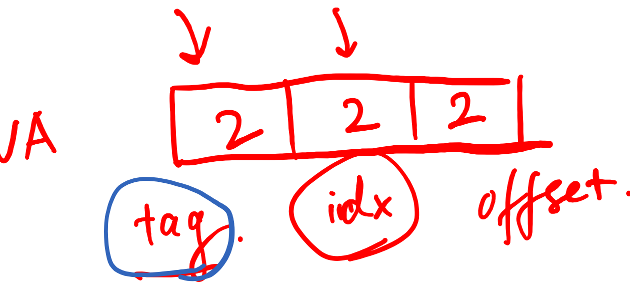
`Size(a) = 4096`

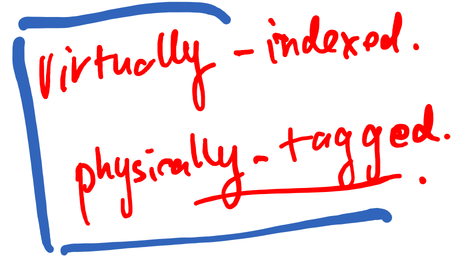
Best case: Num of TLB miss: `4096/4096 = 1`

TLB miss rate: `1/1024 = 0.09%`

TLB hit rate : `99.91%` (almost 100%)

TLB Content



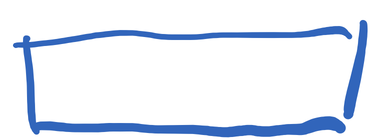


- Some entries are [wired down or reserved] for permanently valid translations

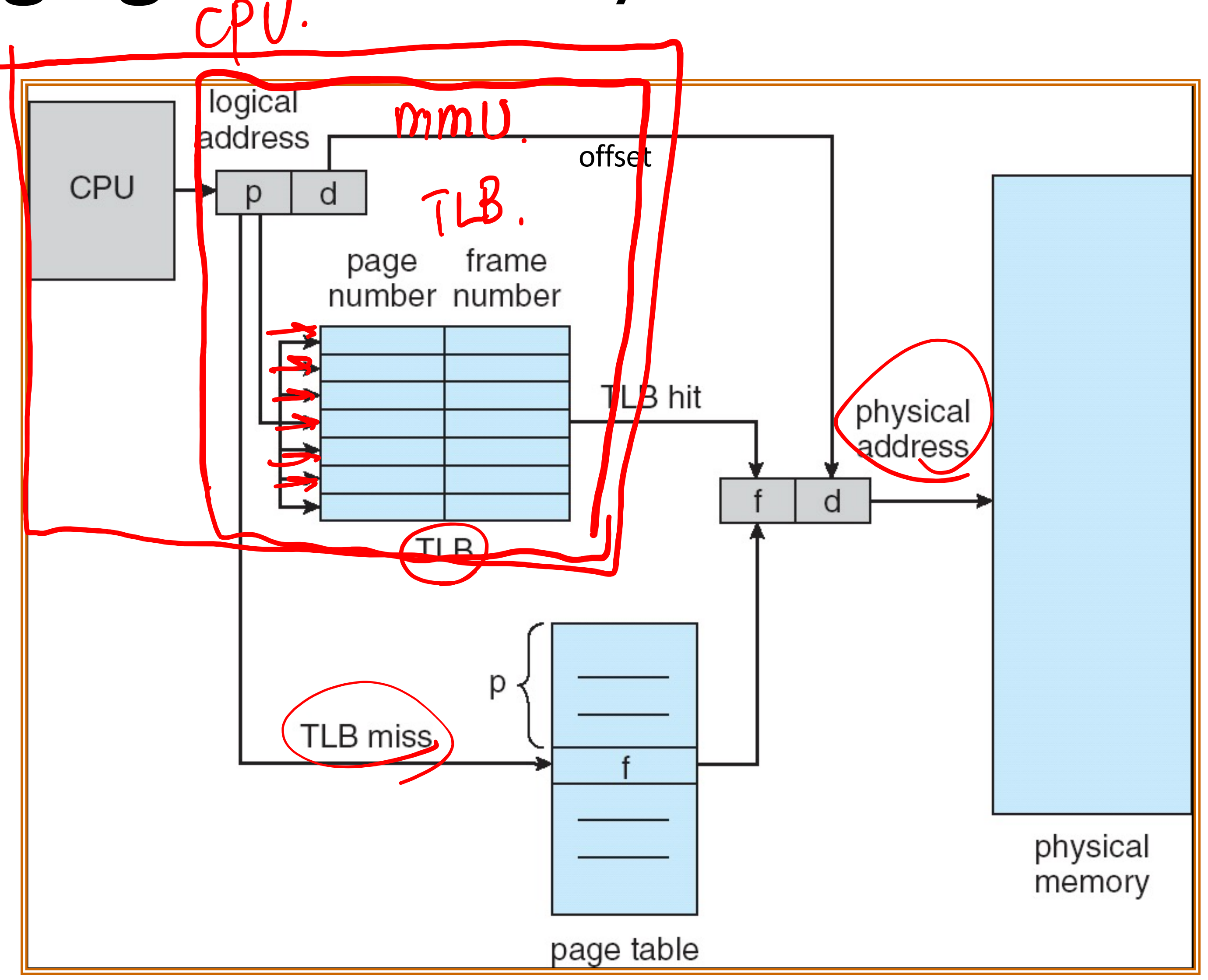
- TLB is a fully associative cache
  - Any given translation can be anywhere in the TLB
  - Hardware searches entire TLB in parallel to find a match

- A typical TLB entry

```
VPN | PFN | other bits
```



Paging Hardware w/ TLB



TLB Issue: Context Switch

- TLB contains translations only valid for the currently running process

- Switching from one process to another requires OS or hardware to do more work

One Example

- How does OS distinguish which entry is for which process?

	VPN	PFN	valid	prot
P1	10	100	1	rwx
	—	—	0	—
P2	10	170	1	rwx
	—	—	0	—

One Simple Solution: Flush

- OS flushes the whole TLB on context switch

- Flush operation sets all valid bit to 0

One Simple Solution: Flush

- OS flushes the whole TLB on context switch

- Flush operation sets all valid bit to 0

- Problem: the overhead is too high if OS switches processes too frequently

Optimization: ASID

- Some hardware systems provide an address space identifier (ASID) field in the TLB

- Think of ASID as a process identifier (PID)
  - An 8-bit field

VPN	PFN	valid	prot	ASID
10	100	1	rwx	1
—	—	0	—	—
10	170	1	rwx	2
—	—	0	—	—

Paging Problems

- Page tables are too slow (just covered)
  - *TLB* to the rescue!

- Page tables are too big

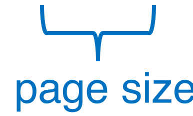
How Large are Page Tables?

- A linear page table array for 32-bit address space (2^{32} bytes) and 4KB page (2^{12} bytes)
  - How many pages: 2^{20} pages
  - How much memory: 4MB assuming each page-table entry is of 4 bytes
    - 2 ^ (32-log(4KB)) * 4 = 4MB

20 = 32 - 12

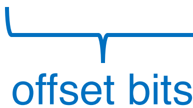
How Large are Page Tables?

- A linear page table array for 32-bit address space (2^{32} bytes) and 4KB page (2^{12} bytes)
  - How many pages: 2^{20} pages
  - How much memory: **4MB** assuming each page-table entry is of 4 bytes
    - 2 ^ (32-log(4KB)) * 4 = 4MB



How Large are Page Tables?

- A linear page table array for 32-bit address space (2^{32} bytes) and 4KB page (2^{12} bytes)
  - How many pages: 2^{20} pages
  - How much memory: 4MB assuming each page-table entry is of 4 bytes
    - 2 ^ (32-log(4KB)) * 4 = 4MB



How Large are Page Tables?

- A linear page table array for 32-bit address space (2^{32} bytes) and 4KB page (2^{12} bytes)
  - How many pages: 2^{20} pages
  - How much memory: 4MB assuming each page-table entry is of 4 bytes
    - 2 ^ (32-log(4KB)) * 4 = 4MB

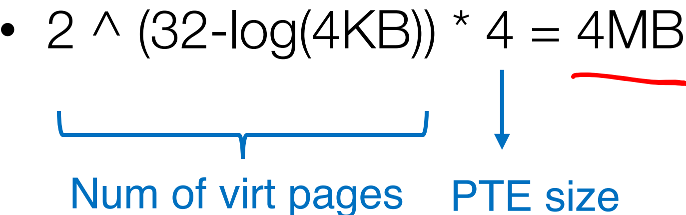
How Large are Page Tables?

- A linear page table array for 32-bit address space (2^{32} bytes) and 4KB page (2^{12} bytes)
  - How many pages: 2^{20} pages
  - How much memory: 4MB assuming each page-table entry is of 4 bytes
    - 2 ^ (32-log(4KB)) * 4 = 4MB

Num of virt pages

How Large are Page Tables?

- A linear page table array for 32-bit address space (2^{32} bytes) and 4KB page (2^{12} bytes)
  - How many pages: 2^{20} pages
  - How much memory: **4MB** assuming each page-table entry is of 4 bytes
    - 2 ^ (32-log(4KB)) * 4 = 4MB



Page Tables are Too Big

- A linear page table array for 32-bit address space (2^{32} bytes) and 4KB page (2^{12} bytes)
  - How many pages: 2^{20} pages
  - How much memory: 4MB assuming each page-table entry is of 4 bytes
    - 2 ^ (32-log(4KB)) * 4 = 4MB
- One page table for one process!
  - A system with 100 process: 400MB only for storing page tables in memory
- Solution??

Naïve Solution

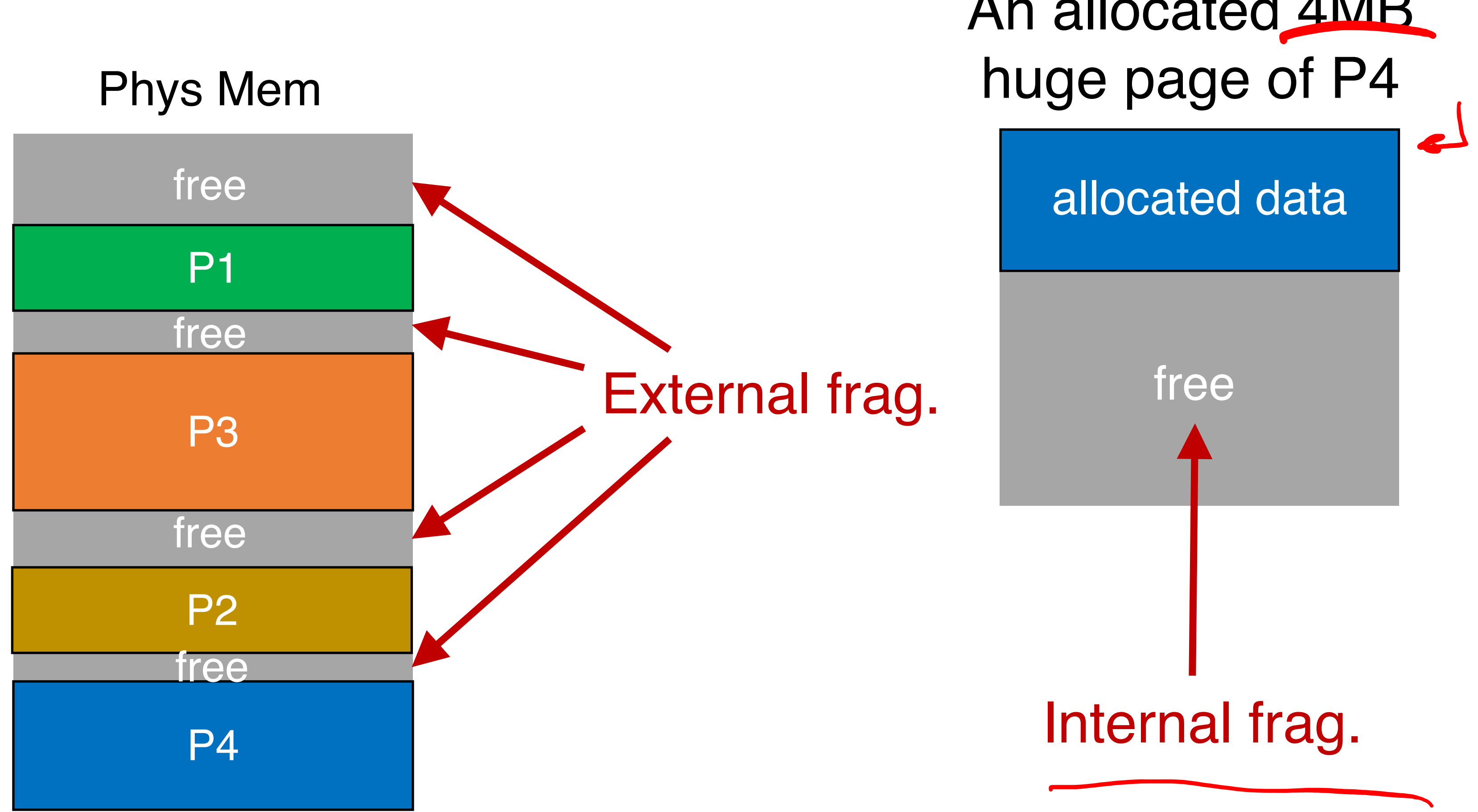
- Reduce the granularity
  - by increasing the page size

Naïve Solution

- Reduce the granularity
  - by increasing the page size

- Why are 4MB pages bad?
  - Internal fragmentation!
  - Well, it's not always bad…

Fragmentation



Assume each process consists of multiple 4MB pages

Approaches

- Approach 1: Linear Inverted Page Table
- Approach 2: Hashed Inverted Page Table
- Approach 3: Multi-level Page Table

Approaches

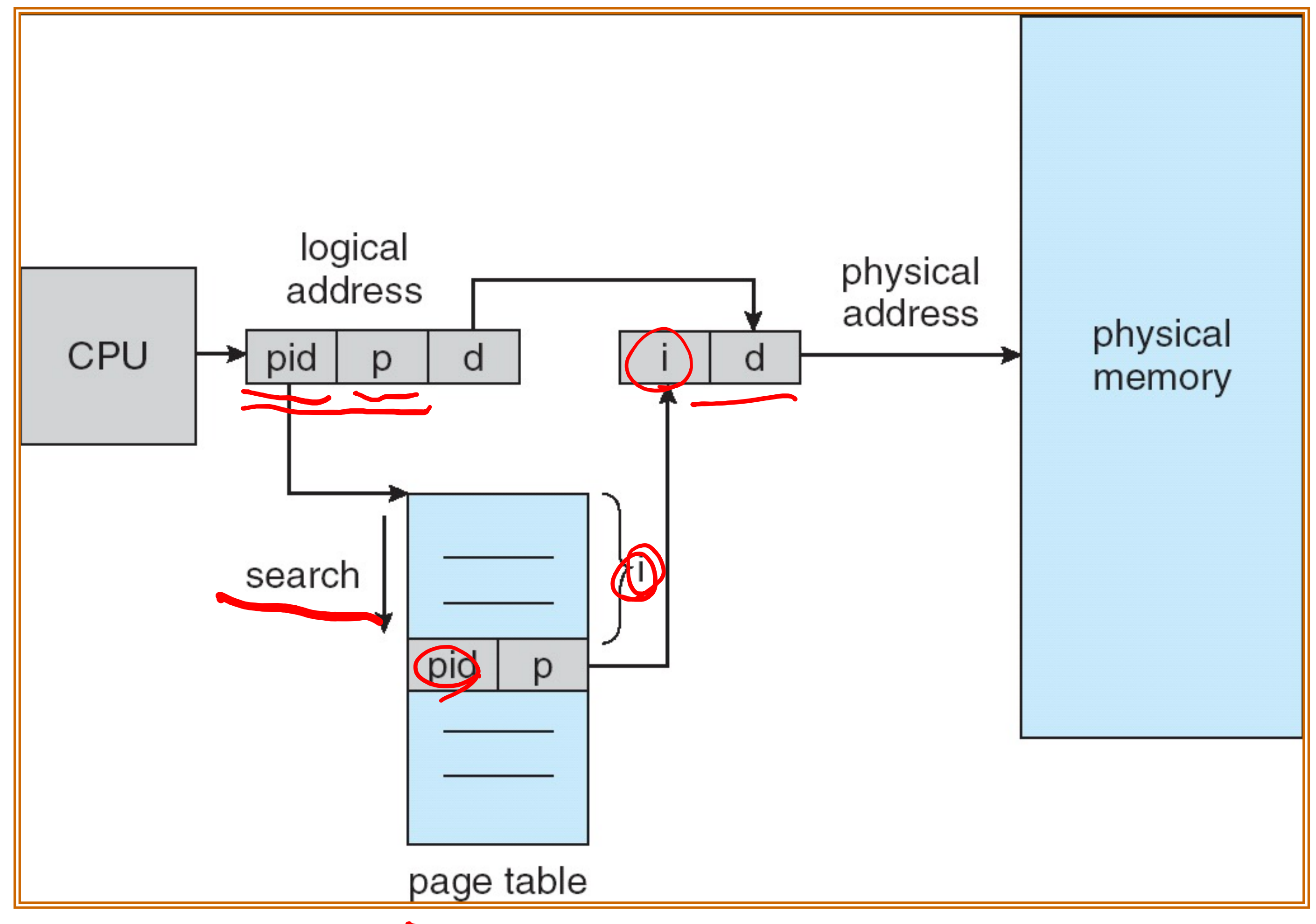
- Approach 1: Linear Inverted Page Table
- Approach 2: Hashed Inverted Page Table
- Approach 3: Multi-level Page Table

Linear Inverted Page Table

- Idea: Instead of keeping one page table per process, the system keeps a single page table that has an entry for each physical frame of the system

- Each entry tells which process owns the page, and VPN to PFN translation

Linear Inverted Page Table Example



Linear Inverted Page Table

- Idea: Instead of keeping one page table per process, the system keeps a single page table that has an entry for each physical frame of the system
- Each entry tells which process owns the page, and VPN to PFN translation
- Goal: use linear search to find the index `i`
  - The reason why it's "**inverted**"

- Pros: Extreme memory savings
- Cons: A linear search is expensive
  - **Solution??**

Approaches

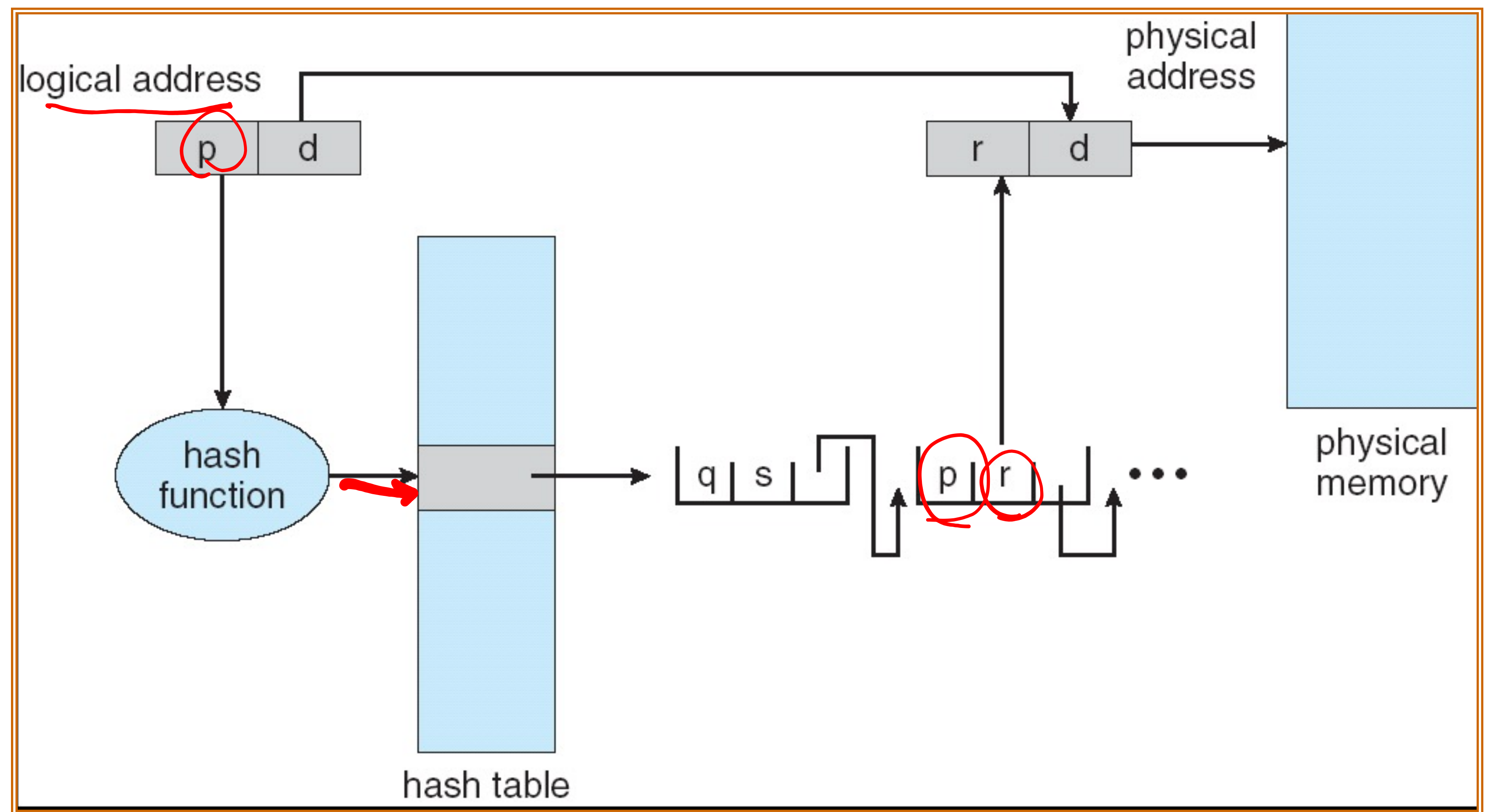
- Approach 1: Linear Inverted Page Table
- Approach 2: Hashed Inverted Page Table
- Approach 3: Multi-level Page Table

Hashed Inverted Page Table

- For large address spaces, a hashed page table can be used, with the hash value being the VPN

- Idea:
  - A PTE contains a chain of elements hashing to the same location (to handle collisions) within PT
  - Each element has three fields: (a) VPN, (b) PFN, (c) a pointer to the next element in the linked list
  - VPNs are compared in this chain searching for a match. If a match is found, the corresponding PFN is extracted

Hashed Inverted Page Table Example



Approaches

- Approach 1: Linear Inverted Page Table
- Approach 2: Hashed Inverted Page Table
- Approach 3: Multi-level Page Table

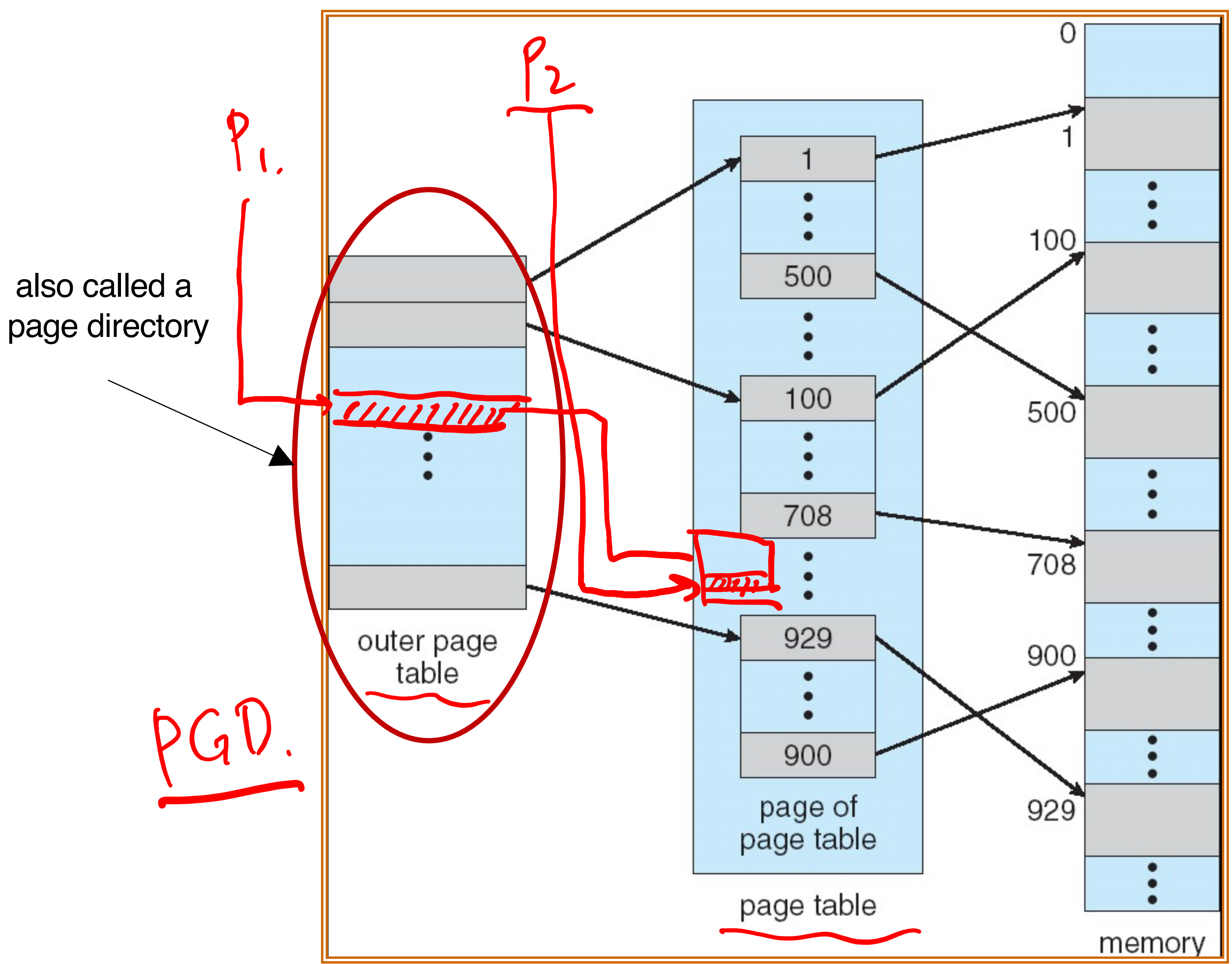
Multi-level Page Table: Radix PT

- Idea:
  - Break the page table into pages (the entire page table is paged!)
  - Only have pieces with >0 valid entries
    - Don't allocate the page of page table if the entire page of page-table entries is invalid

- Used by x86: also called radix page table

- A simple technique is a two-level page table

Two-level Page Table Example



Two-level Paging

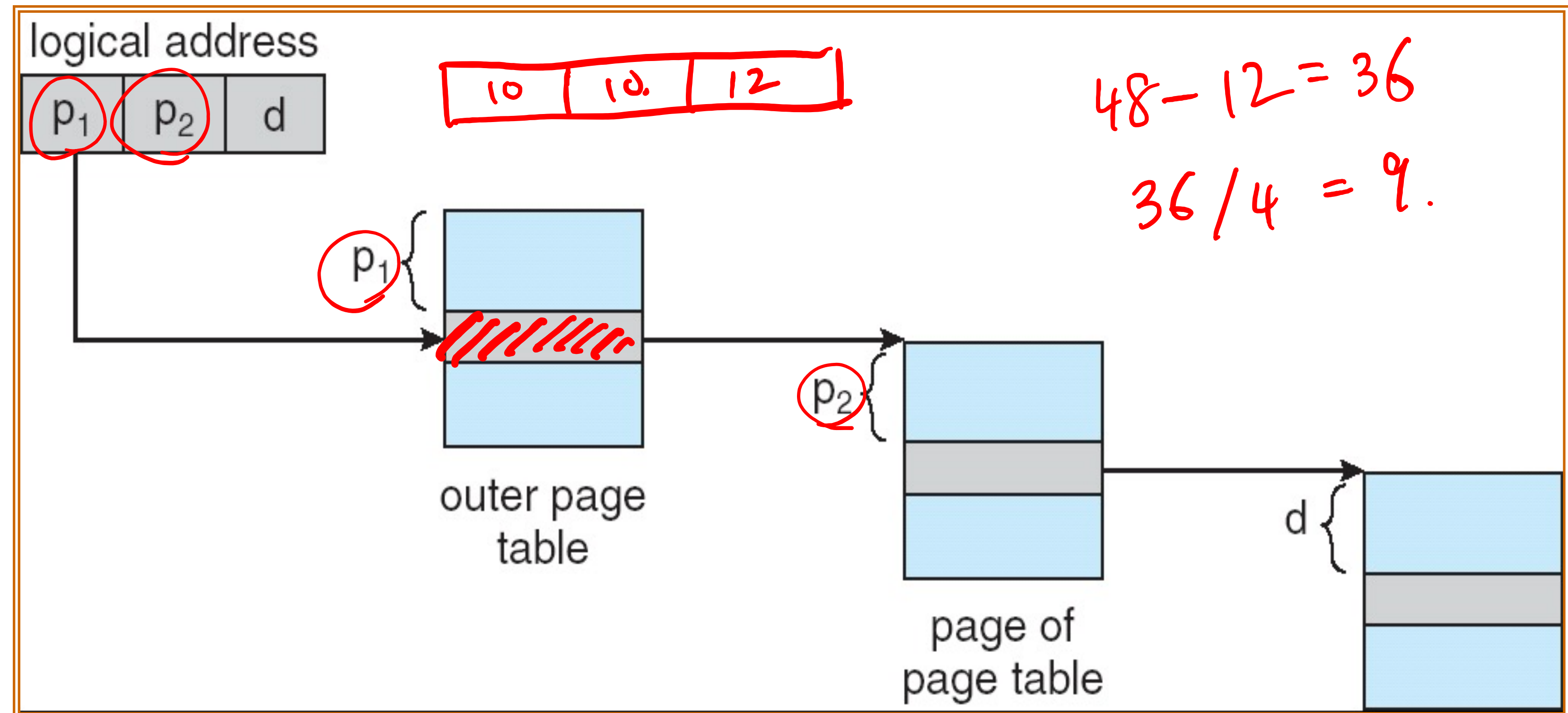
- A logical address (on 32-bit machine with 4KB page size) is divided into
  - a page number consisting of 20 bits
  - a page offset consisting of 12 bits
- A page table entry is 4 bytes
- Since the page table is paged, the page number is further divided into
  - p_1: a 10-bit page directory index
  - p_2: a 10-bit page table index
- The logical address is as follows:

page number		page offset
p_1	p_2	d
10	10	12

where p_1 is an index into the outer page table, and p_2 is the displacement within the page of the inner page table

Address Translation Scheme

- Address translation scheme for a two-level 32-bit paging architecture



> 2 Levels

- Problem: page directory may not fit in a page

- Solution:
  - Split page directories into pieces
  - Use another page dir to refer to the page dir pieces

> 2 Levels

- Problem: page directory may not fit in a page

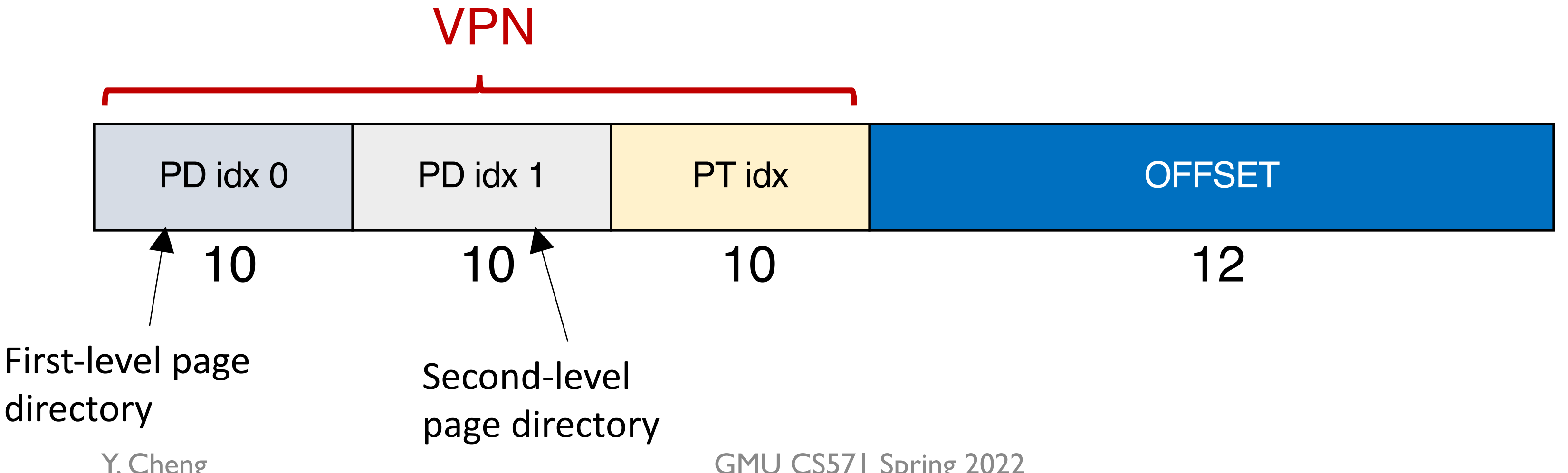
- Solution:
  - Split page directories into pieces
  - Use another page dir to refer to the page dir pieces

- Possible to extend to 3- or 4-level

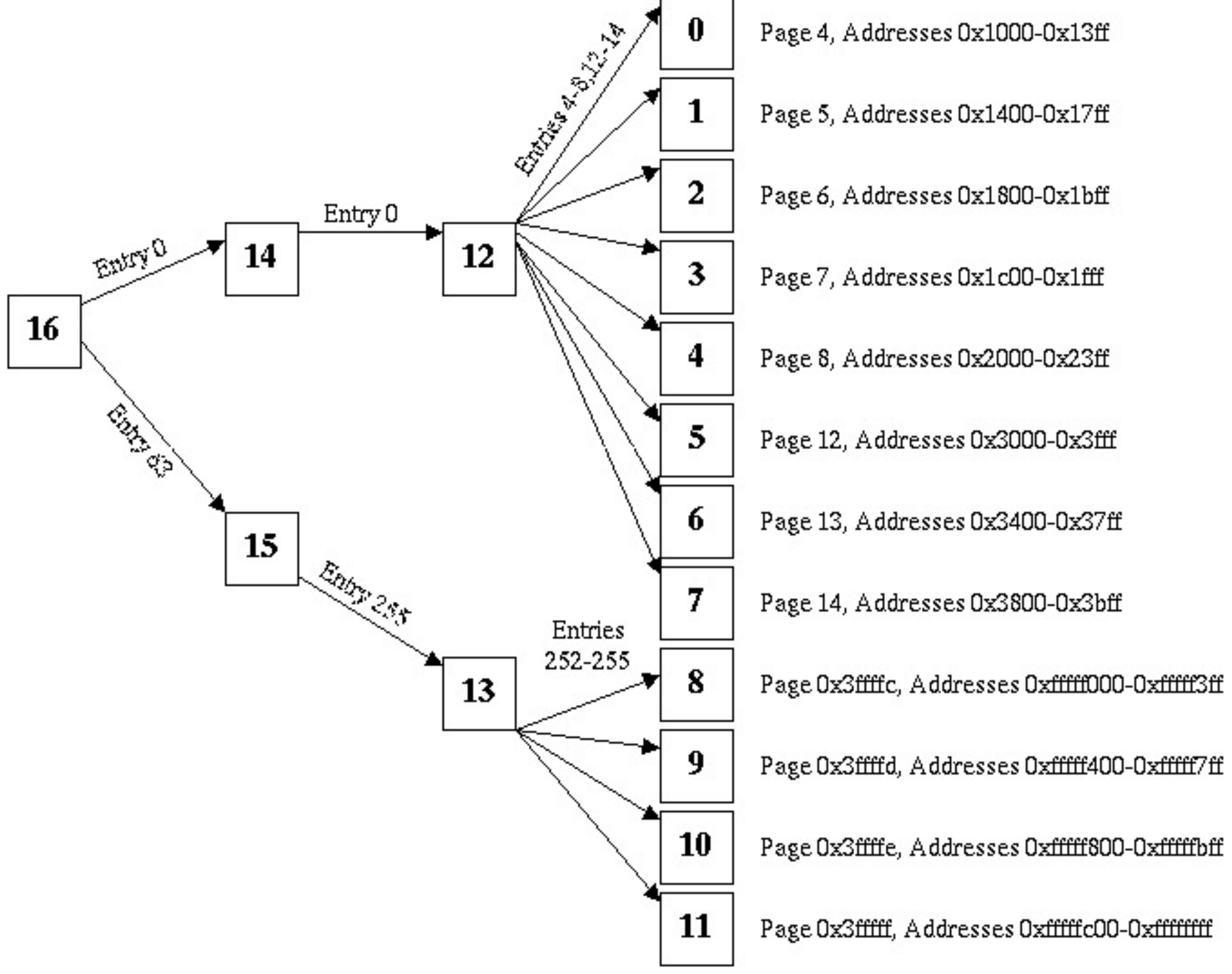
- E.g., 64-bit Ultra-SPARC would need 7 levels of paging

> 2 Levels

- Problem: page directory may not fit in a page
- Solution:
  - Split page directories into pieces
  - Use another page dir to refer to the page dir pieces



Multi-level Page Table Example



http://web.eecs.utk.edu/~mbeck/classes/cs560/560/oldtests/t2/2003/Answers.html