EECS 482
Introduction to Operating Systems

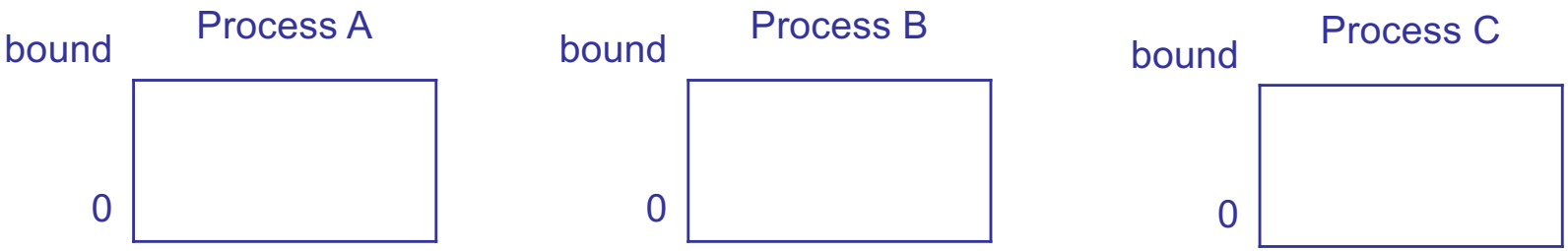
Winter 2018

Harsha V. Madhyastha

Address Spaces

- Hardware interface:
  - All processes share physical memory

- OS abstraction:



Recap of Address Spaces

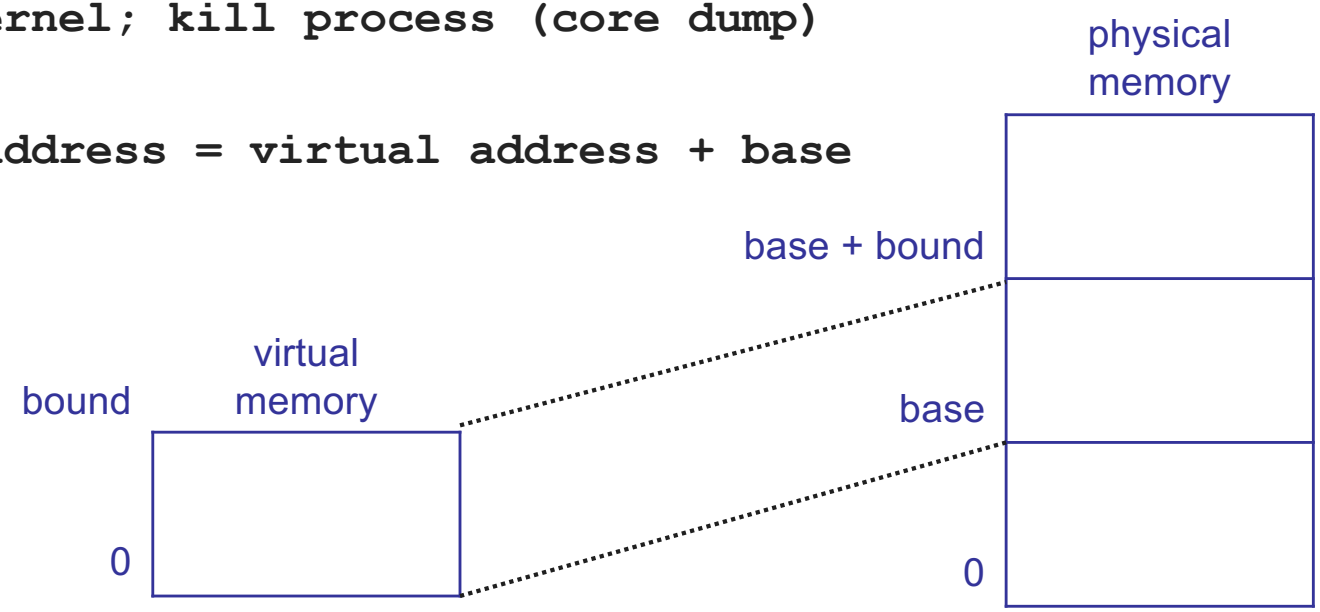
- Must offer three properties:
  - Address independence
  - Protection
  - Virtual memory

- Need dynamic instead of static translation

Base and bounds

- Load each process into contiguous region of physical memory
  - Prevent process from accessing data outside its region

```
if (virtual address > bound) {
        trap to kernel; kill process (core dump)
} else {
        physical address = virtual address + base
}
```



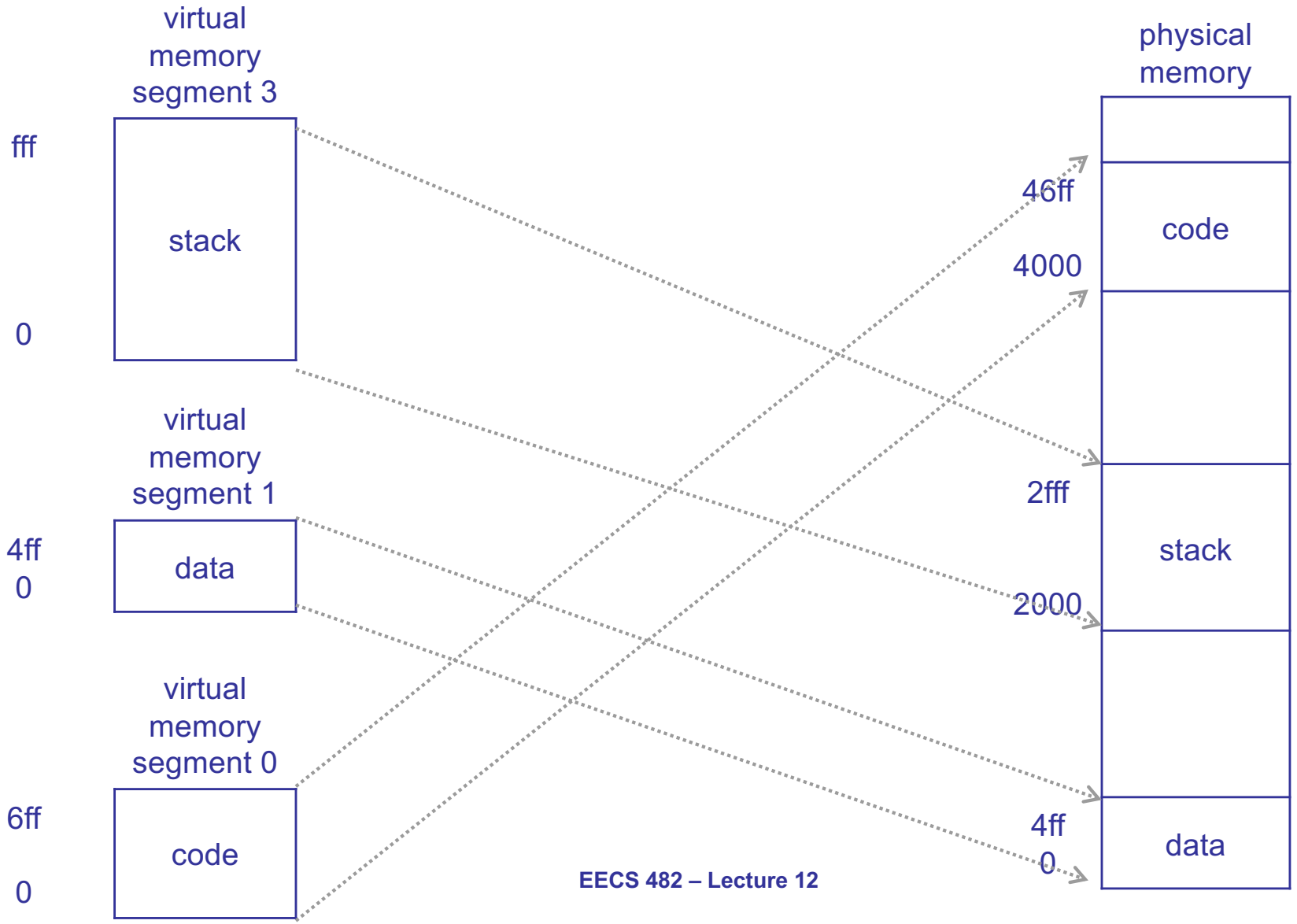
Base and bounds

- Pros:
  - Fast
  - Simple hardware support
- Cons:
  - No virtual memory
  - External fragmentation
  - Cannot grow parts of address space independently
  - No controlled sharing
- Cause: Address space is indivisible unit

Segmentation

- Divide address space into segments
- Segment: region of memory contiguous in both physical memory and in virtual address space

Segmentation



Segmentation

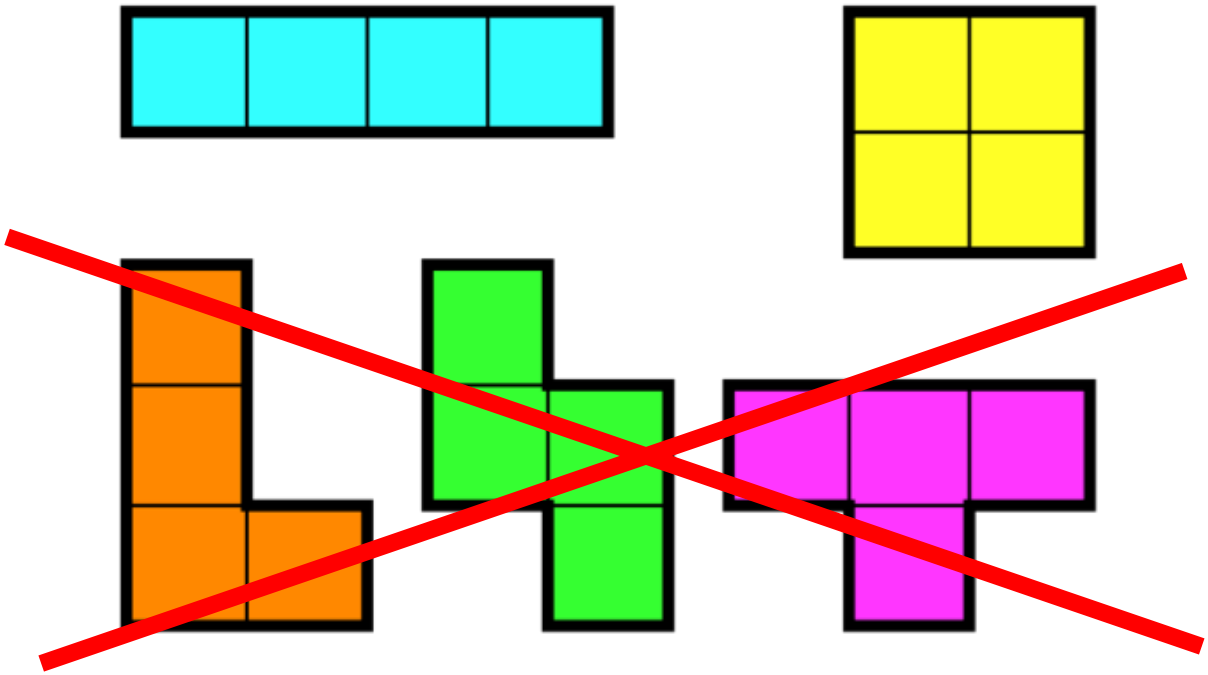
Segment #	Base	Bounds	Description
0	4000	700	code segment
1	0	500	data segment
2	n/a	n/a	unused
3	2000	1000	stack segment

- Virtual address is of the form: (segment #, offset)
  - Physical address = base for segment + offset

Segmentation

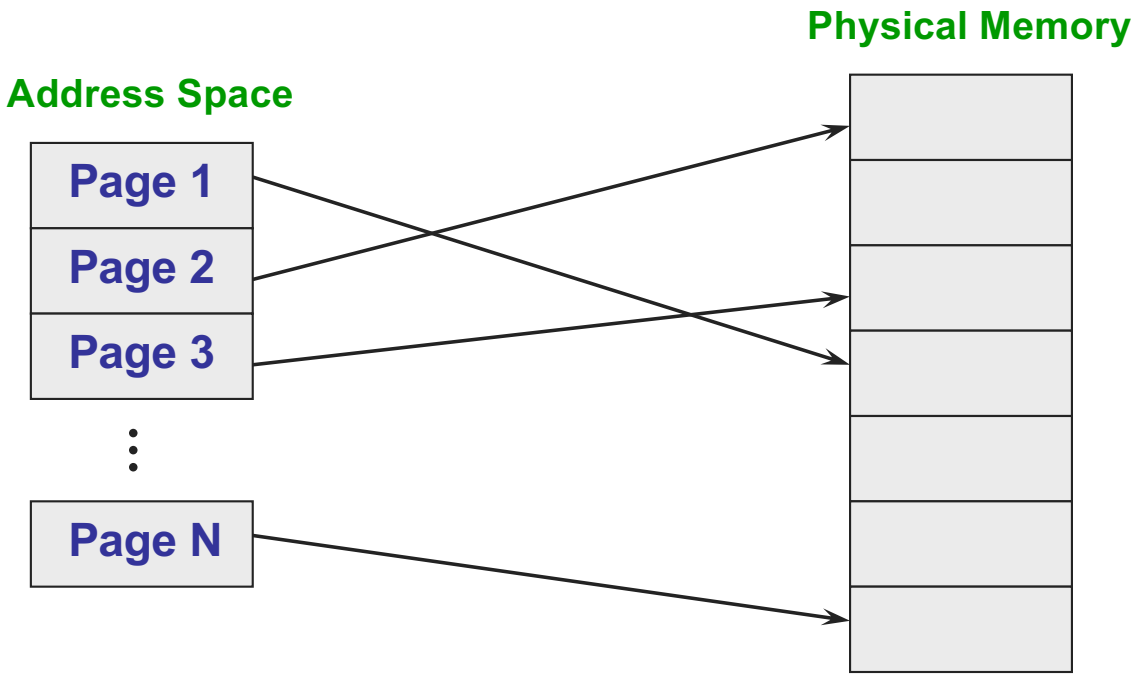
- Pros:
  - Can grow each segment independently
  - Can share segments across address spaces
- Cons:
  - Every segment must be smaller than phys. memory
  - Segment allocation is hard
  - External fragmentation
- Cause: Variable amount of contiguous memory


GAME
OVER



Paging

- Allocate phys. memory in fixed-size units (pages)
  - Any free physical page can store any virtual page



Paging

- Translation data is the page table

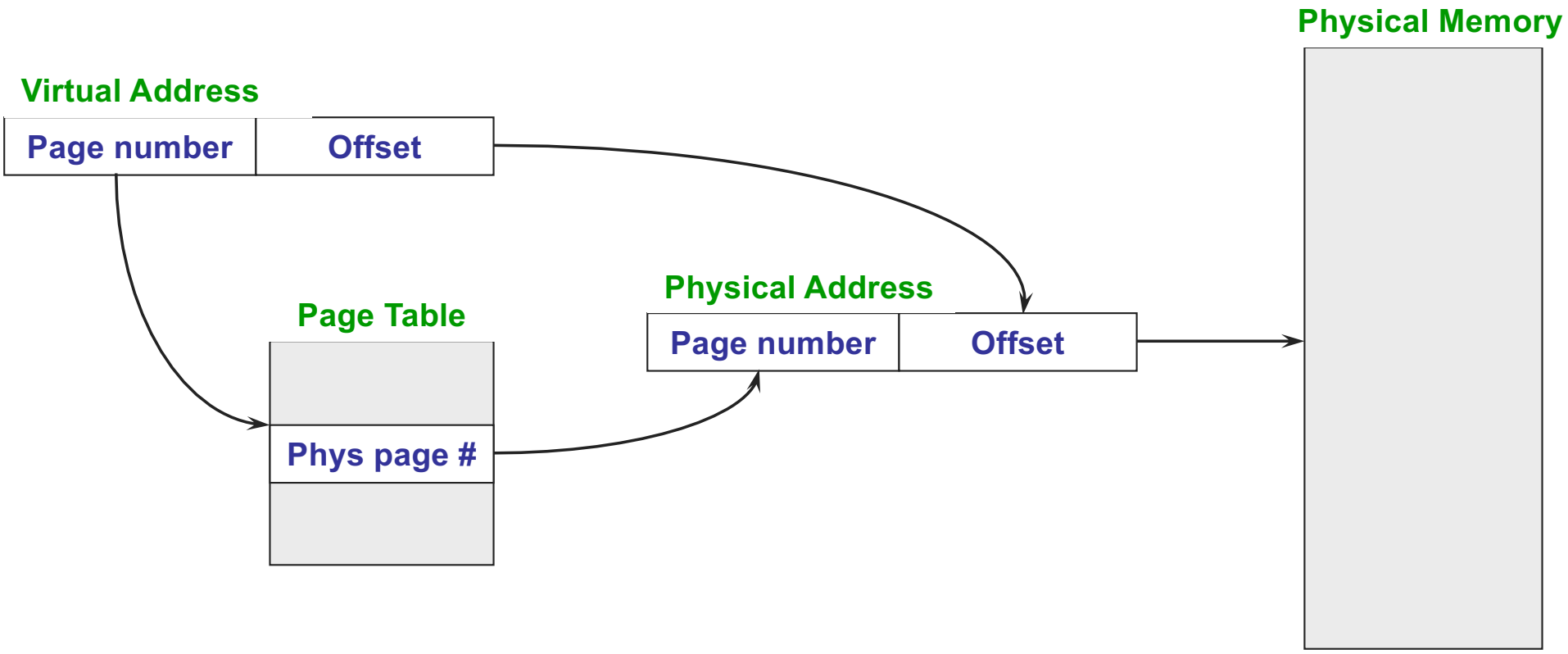
Why no column for bound?

Virtual page #	Physical page #
0	105
1	15
2	283
3	invalid
...	invalid
1048575	invalid

Function to translate VA to PA?

- Virtual address is split into
  - Virtual page # (high bits of address, e.g., bits 31-12)
  - Offset (low bits of address, e.g., bits 11-0, for 4 KB page size)

Page Lookups



Paging

- Translating virtual address to physical address

```
if (virtual page is invalid) {
        trap to OS fault handler
} else {
        physical page # = pageTable[virtual page #].physPageNum
        physical address = {Physical page #}{offset}
}
```

- What must be changed on a context switch?
  - Indirection via Page Table Base Register

Paging

- Each virtual page can be in physical memory or “paged out” to disk
- How does processor know that a virtual page is not in physical memory?
- Like segments, pages can have different protections (e.g., read, write, execute)

```
if (virtual page is invalid or non-resident or protected) {
        trap to OS fault handler
} else {
        physical page # = pageTable[virtual page #].physPageNum
        physical address = {Physical page #}{offset}
}
```

Paging

- Revised page table:

Virtual page #	Physical page #	Resident	Protection
0	105	0	RX
1	15	1	R
2	283	1	RW
3	invalid		
...	invalid		
1048575	invalid		

Valid versus Resident

- Valid → virtual page is legal for process to access
- Resident → virtual page is in physical memory
- Error to access invalid page, but not to access non-resident page

- Who makes a virtual page resident/non-resident?
- Who makes a virtual page valid/invalid?
- Why would a process want one of its virtual pages to be invalid?

Picking Page Size

- What happens if page size is really small?
- What happens if page size is really big?

- Typically a compromise, e.g., 4 KB or 8 KB
  - Some architectures support multiple page sizes

Growing Address Space

Virtual page #	Physical page #
0	105
1	15
2	282
...	invalid
1048572	invalid
1048573	1078
1048574	48136
1048575	60

Why less space wastage than base and bounds?

Paging

- Pros
  - Simple memory allocation
  - Flexible sharing
  - Easy to grow address space
- Cons
  - 32-bit virtual address, 4 KB pages, 4 byte PTEs
  - Page table size?

Page table size

- 32-bit address → 2^32 unique addresses
- 4 KB page → (2^32)/4 KB = 2^20 virtual pages
- 4 bytes per PTE → 4 MB page table
- 25 processes → 100 MB for page tables!

- How to reduce page table overhead?

Project 2

- Due on Saturday!
  - Check calendar on web page for extra office hours

- For every thread, think about “where is the current context?”
- Think about memory leaks and how to test if they exist
- Think about why your thread library may cause a program to run for longer than correct library

Mid-Term

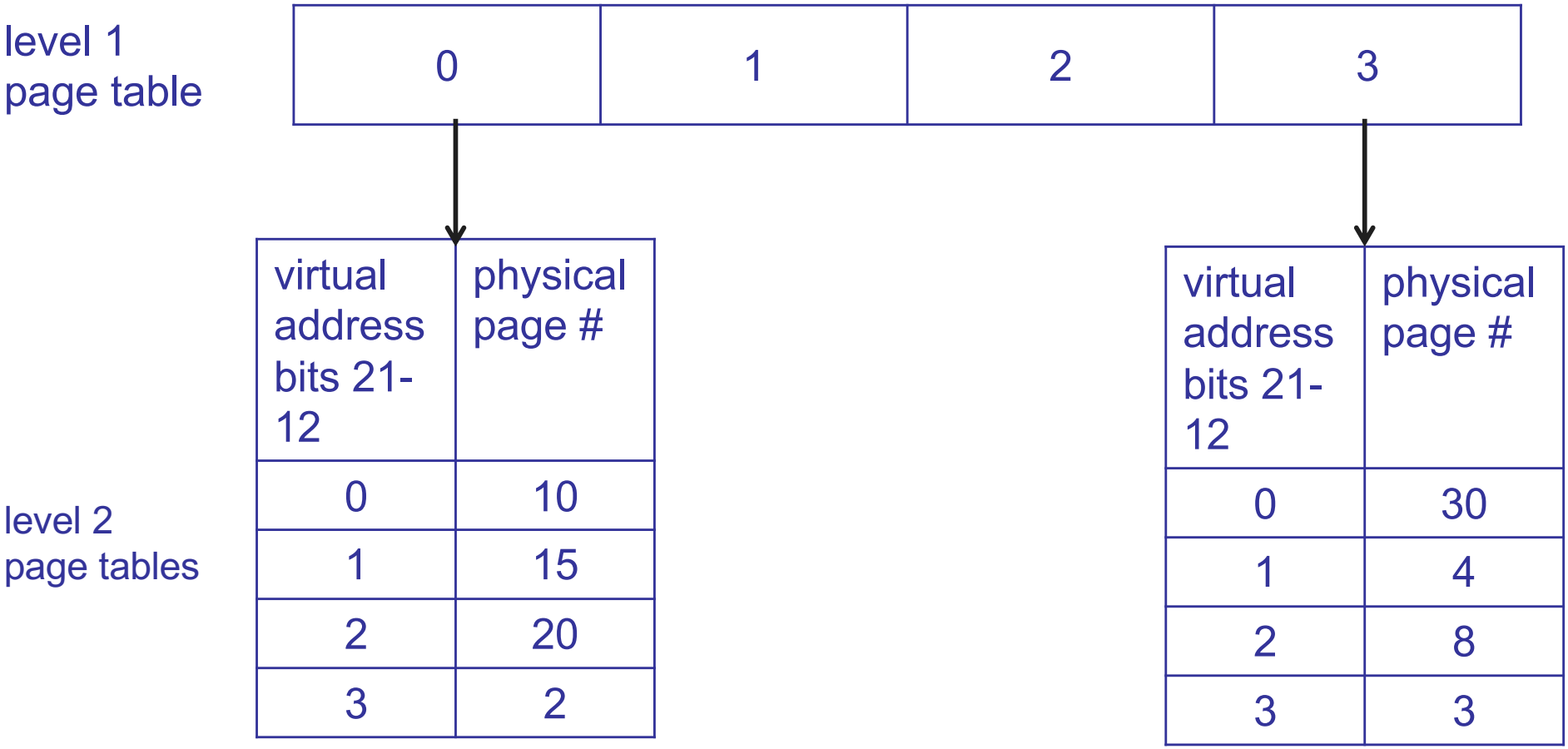
- Next Tuesday!
  - Covers all material up to and including deadlock
- Email me if you would like me to review any material on Wednesday
- Review of sample exams on Sunday (Feb 18th)
  - Important to try questions yourself first!

Multi-level Paging

- Standard page table is a simple array
- Multi-level paging generalizes this into a tree

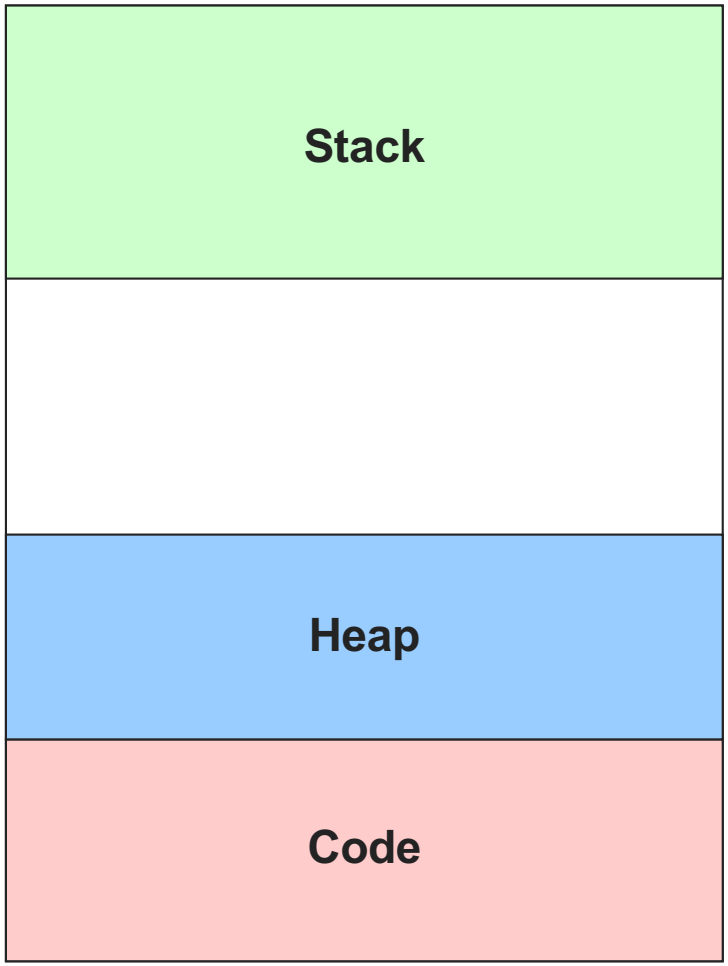
- Example: Two-level page table with 4KB pages
  - Index into level 1 page table: virtual address bits 31-22
  - Index into level 2 page table: virtual address bits 21-12
  - Page offset: bits 11-0

Multi-level Paging



How does this let translation data take less space?

Sparse Address Space



Virtual page #	Physical page #
0	105
1	15
2	283
3	invalid
...	invalid
1048572	invalid
1048573	1078
1048574	48136
1048575	60

Sparse Address Space

Bits 31-22	Physical address
0	0xfffff389
1	Invalid
2	Invalid
…	Invalid
1021	Invalid
1022	Invalid
1023	0xffff7046

Bits 21-12	Physical page #
0	105
1	15
2	283
3	invalid
...	invalid

Bits 21-12	Physical page #
...	invalid
1020	invalid
1021	1078
1022	48136
1023	60

Multi-level paging

- How to share memory between address spaces?
- What must be changed on a context switch?
- Pros
  - Easy memory allocation
  - Flexible sharing
  - Space efficient for sparse address spaces
- Cons
  - Two or more extra lookups per memory reference

Translation lookaside buffer

- TLB caches virtual page # to PTE mapping
  - Cache hit → Skip all the translation steps
  - Cache miss → Get PTE, store in TLB, restart instruction
- Does TLB change what happens on a context switch?

End-to-end look at paging

- New process → allocate new L1 page table
  - All entries in L1 page table invalid
- As process makes virtual pages valid, allocate new L2 page tables and add entries
- To serve load/store on a virtual page:
  - CPU looks up TLB to find PTE for virtual page #
  - If absent, lookup PTE in memory and load TLB
- When process ends, deallocate L1 and L2 page tables

Page replacement

- Not at all valid pages can be in phys memory
- How to handle loads/stores on non-resident pages?