EECS 482
Introduction to Operating Systems

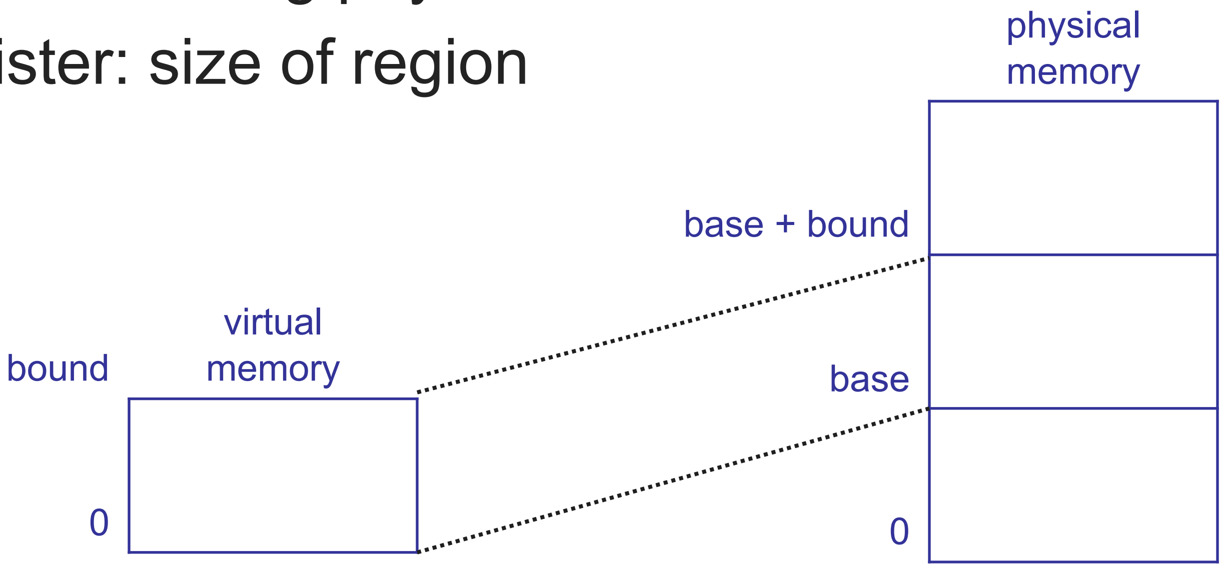
Fall 2018

Baris Kasikci

Slides by: Harsha V. Madhyastha

Base and bounds

- Load each process into contiguous region of physical memory
  - Prevent process from accessing data outside its region
  - Base register: starting physical address
  - Bound register: size of region



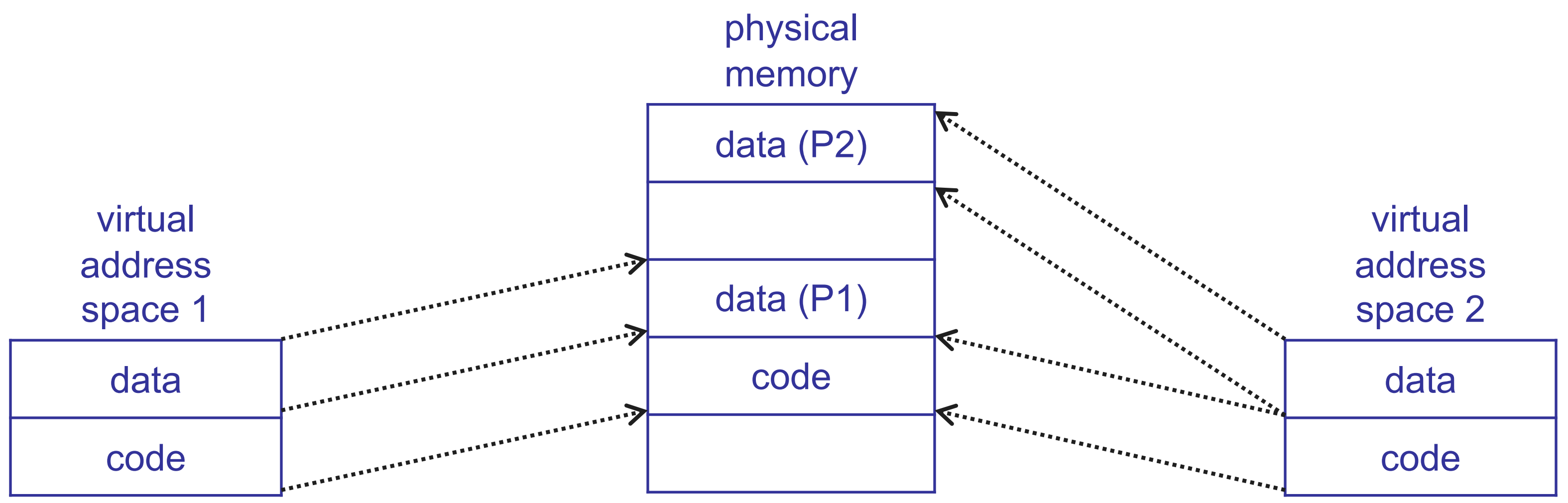
Base and bounds

- Pros?
  - Fast
  - Simple hardware support

- Cons?
  - Virtual address space limited by physical memory
  - No controlled sharing
  - External fragmentation

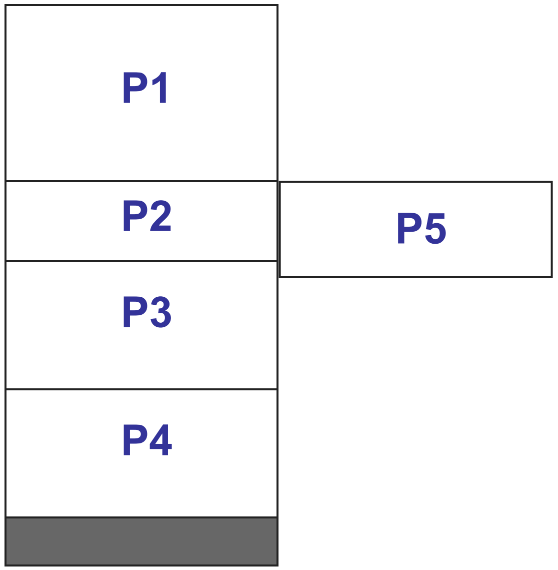
Base and bounds

- Can’t share part of an address space between processes

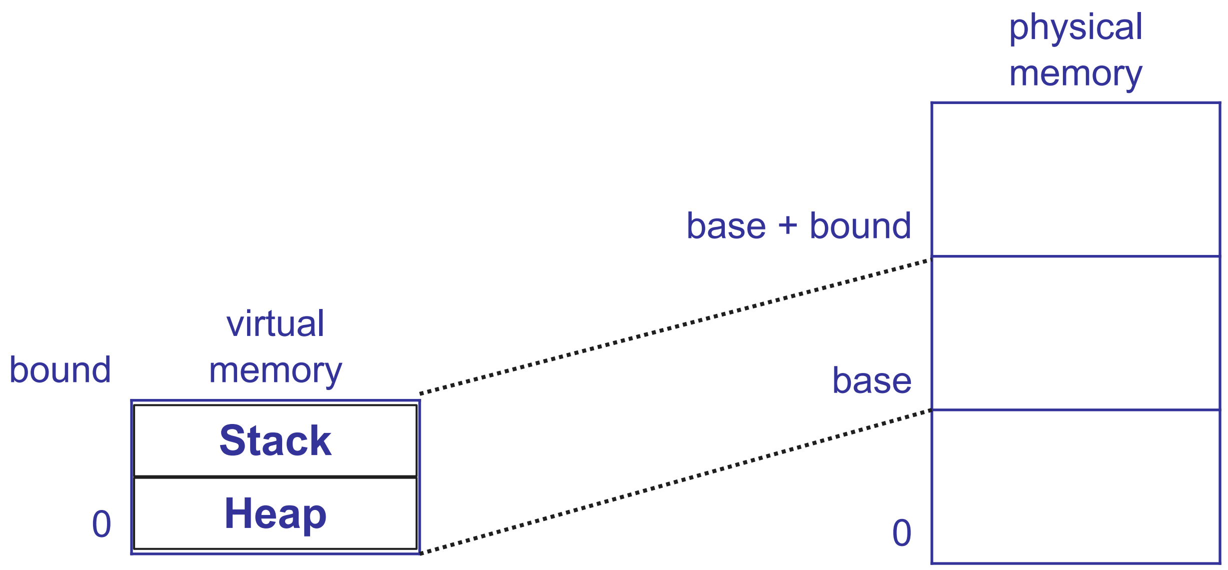


External fragmentation

- Processes come and go, leaving a mishmash of available memory regions
  - Wasted memory between allocated regions



Growing address space

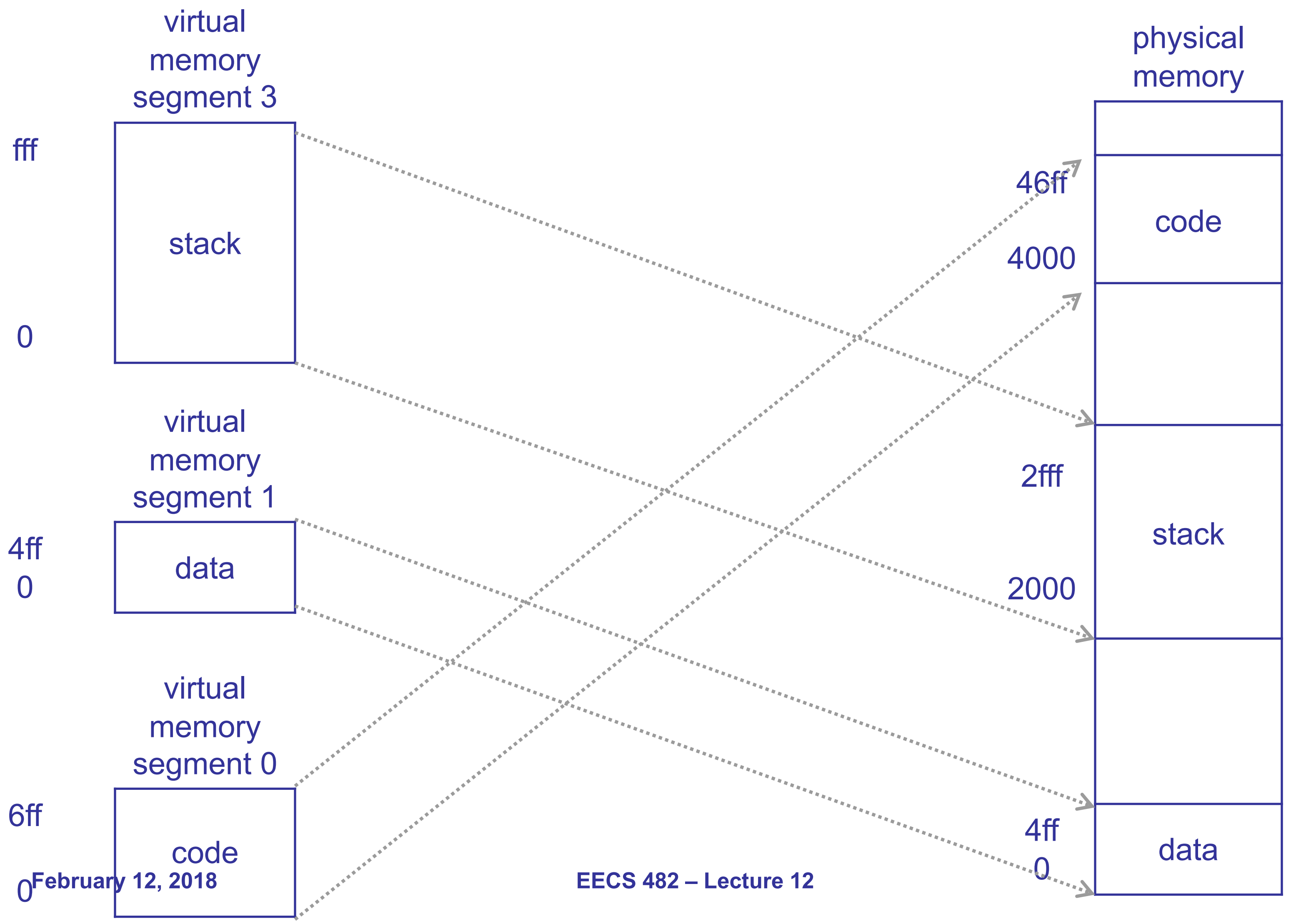


How can stack and heap grow independently?

Segmentation

- Divide address space into segments
- Segment: region of memory contiguous in both physical memory and in virtual address space
  - Multiple segments of memory with different base and bounds.

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
- Many ways to specify the segment #
  - High bits of address
  - Special register
  - Implicit to instruction opcode

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

- Physical address for virtual address (3, 100)?
  - 2100
- Physical address for virtual address (0, ff)?
  - 40ff
- Physical address for virtual address (2, ff)?
- Physical address for virtual address (1, 2000)?

Segmentation

- Not all virtual addresses are valid
  - Valid → region is part of process's address space
  - Invalid → virtual address is illegal to access
    - » Accessing an invalid virtual address causes a trap to OS (usually resulting in core dump)

- Reasons for virtual address being invalid?
  - Invalid segment number
  - Offset within valid segment beyond bound

Segmentation

- How to grow a segment?
- Different segments can have different protection
  - E.g., code is usually read only (allows fetch, load)
  - E.g., data is usually read/write (allows fetch, load, store)
  - Fine-grained protection in base and bounds?
- What must be changed on a context switch?

Benefits of Segmentation

- Multiple areas of address space can grow separately
- Easy to share part of address space

Process 1

Segment #	Base	Bounds	Description
0	4000	700	code segment ←
1	0	500	data segment
3	2000	1000	stack segment

Process 2

Segment #	Base	Bounds	Description
0	4000	700	code segment ←
1	1000	300	data segment
3	500	1000	stack segment

Drawbacks of Segmentation

- Have we eliminated external fragmentation?
- Can an address space be larger than physical memory?
- How can we:
  - Make memory allocation easy
  - Not have to worry about external fragmentation
  - Allow address space size to be > physical memory

Project 2

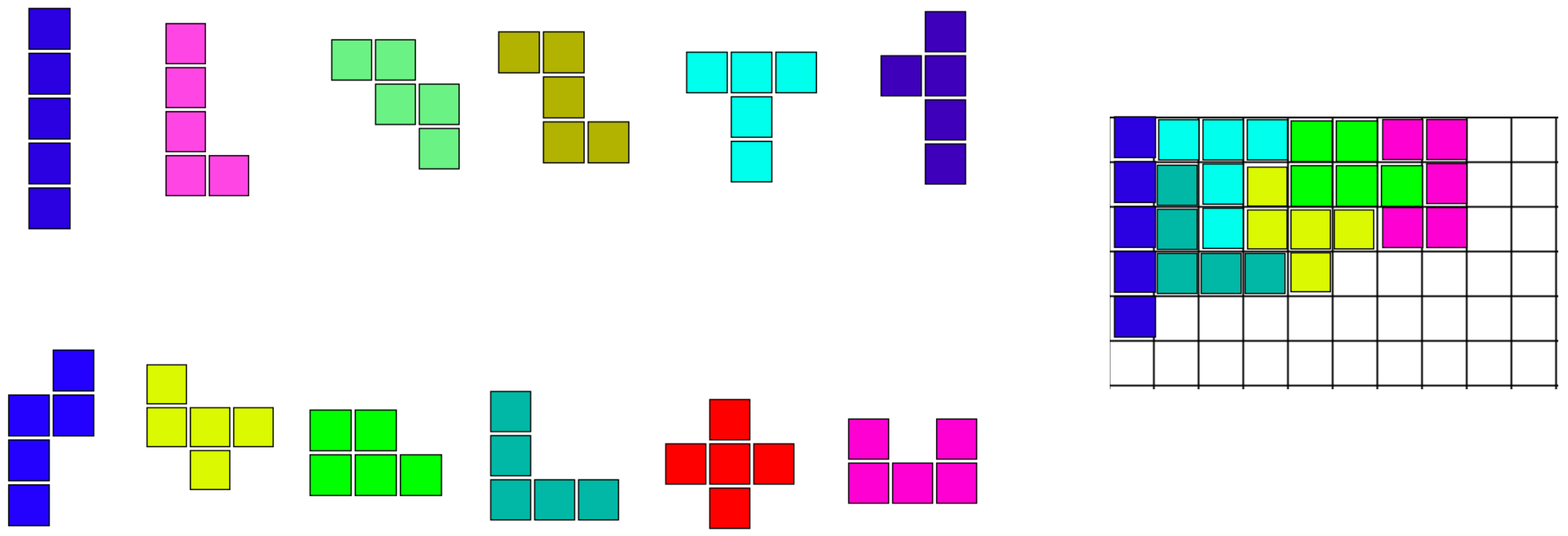
- Due in 6 days
  - Check calendar on web page for extra office hours

- 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

Drawbacks of Segmentation

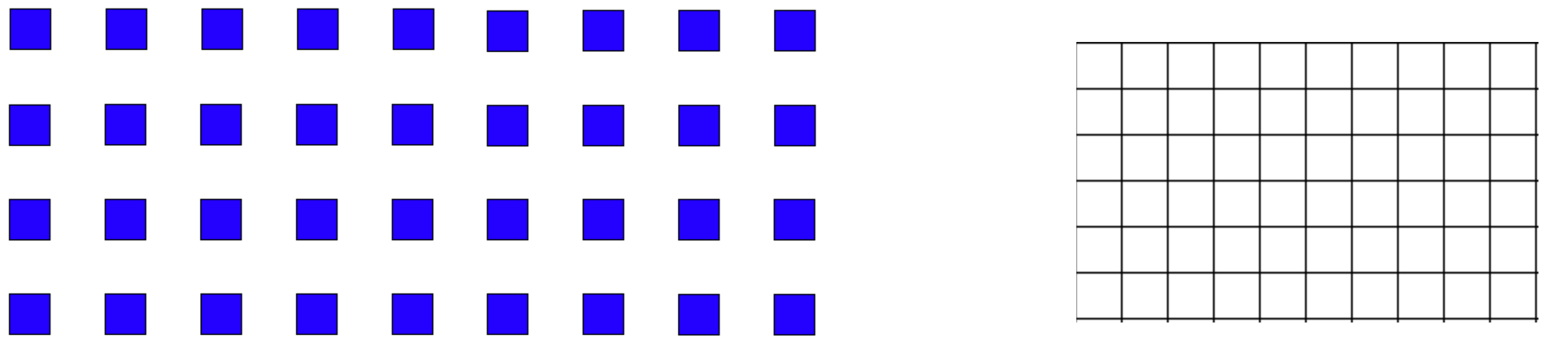
- Have we eliminated external fragmentation?
- Can an address space be larger than physical memory?
- How can we:
  - Make memory allocation easy
  - Not have to worry about external fragmentation
  - Allow address space size to be > physical memory

Drawbacks of Segmentation



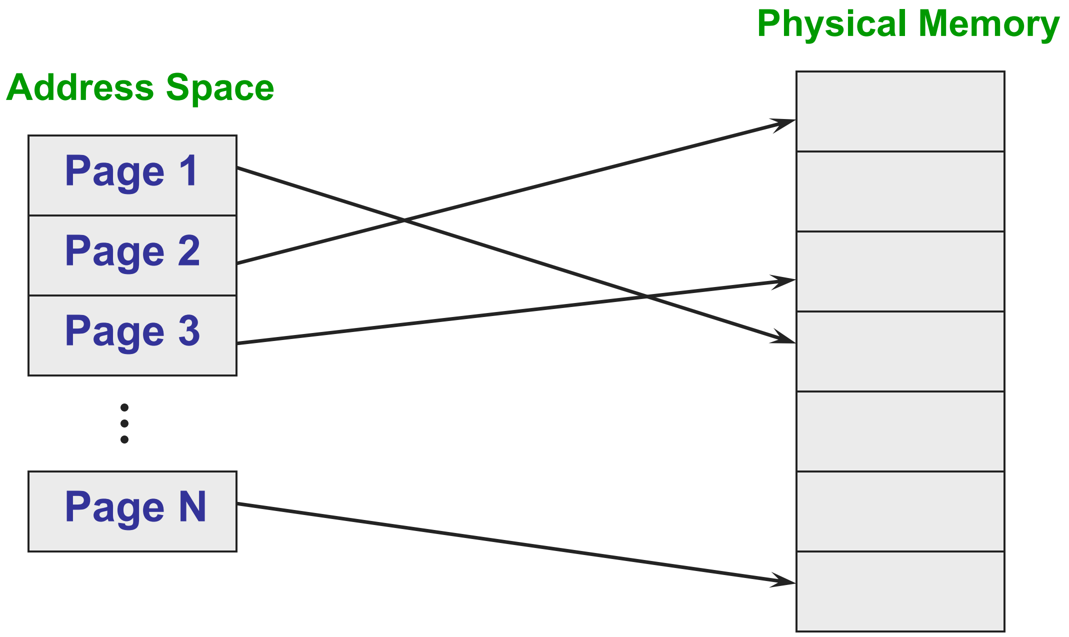
Paging

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



Paging

- 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)



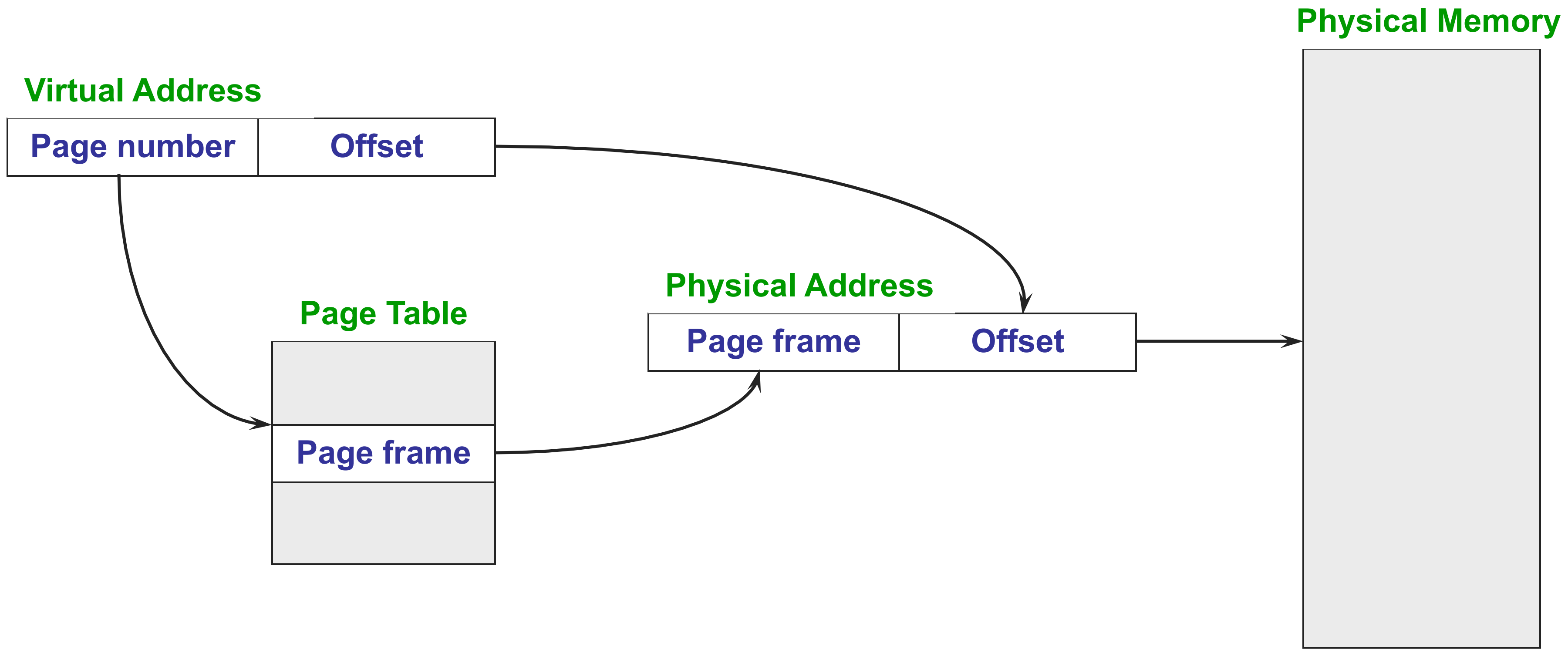
Paging

- Translation data is the page table

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

- Why no column for bound?
- How do I check whether offset is within bound?
- One entry per virtual page → Lots of entries!

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
}
```

- What must be changed on a context switch?
  - Page table (but can be large)
  - Use **indirection**: Page Table Base Register
    - » Points to physmem address of page table

Paging out

- 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?

Virt. page #	Phys. page #	resident
0	105	1
1	15	0
2	283	1
3	invalid	
…	invalid	

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

Paging

- Like segments, pages can have different protections (e.g., read, write, execute)

Virt. page #	Phys. page #	resident	protected
0	105	1	0
1	15	0	0
2	283	1	1
3	invalid		
…	invalid		

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

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?

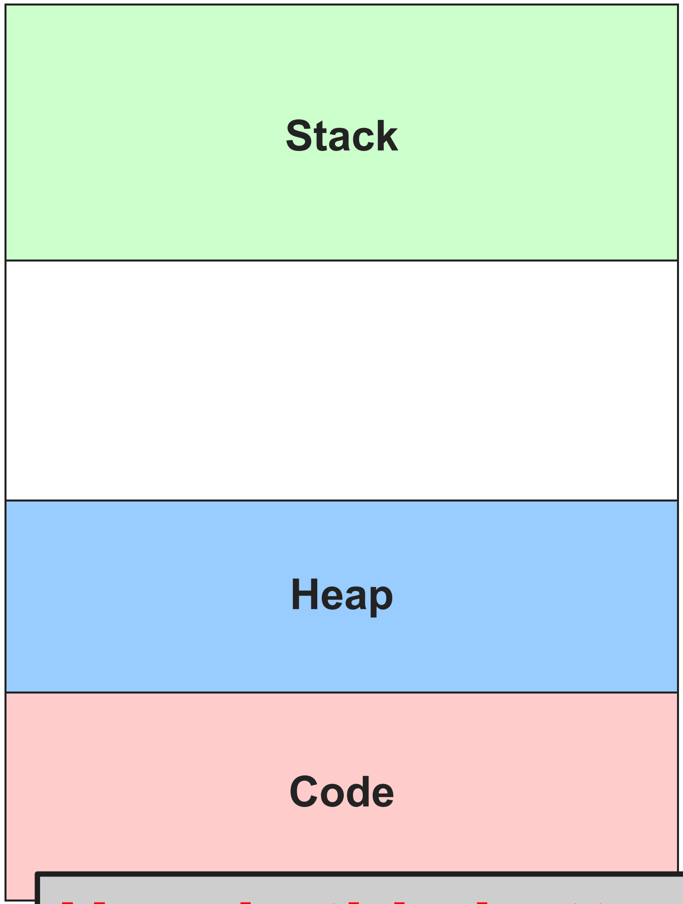
Picking Page Size

- 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

What must be changed on a context switch?

Growing Address Space



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

How is this better than base and bounds?

Paging

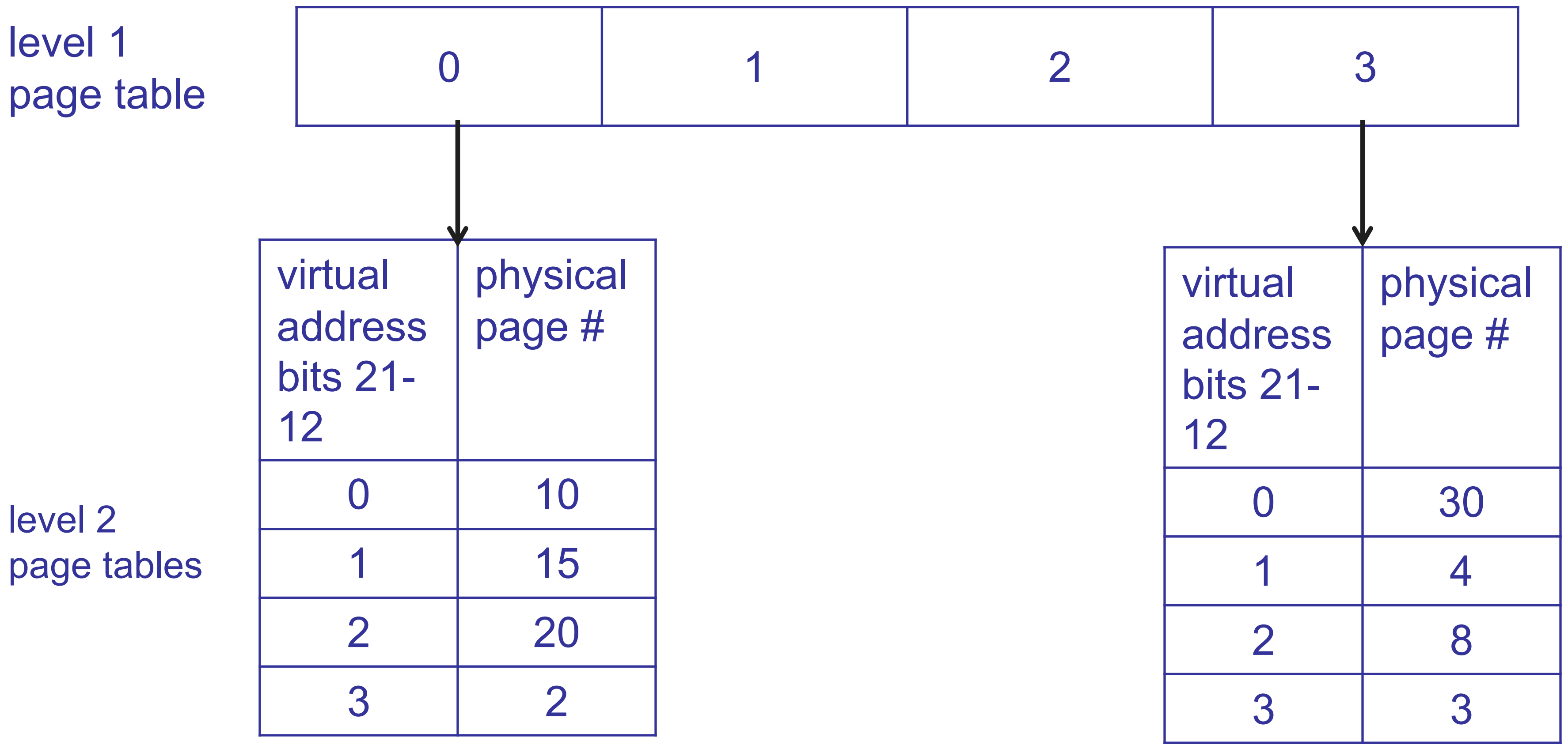
- Pros?
  - Flexible memory allocation/growing address space
  - Virtual memory
  - No external fragmentation
  - Flexible sharing
- Cons?
  - Large page tables
- How to modify paging to reduce space needed for translation data?

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



level 1
page table

0	1	2	3

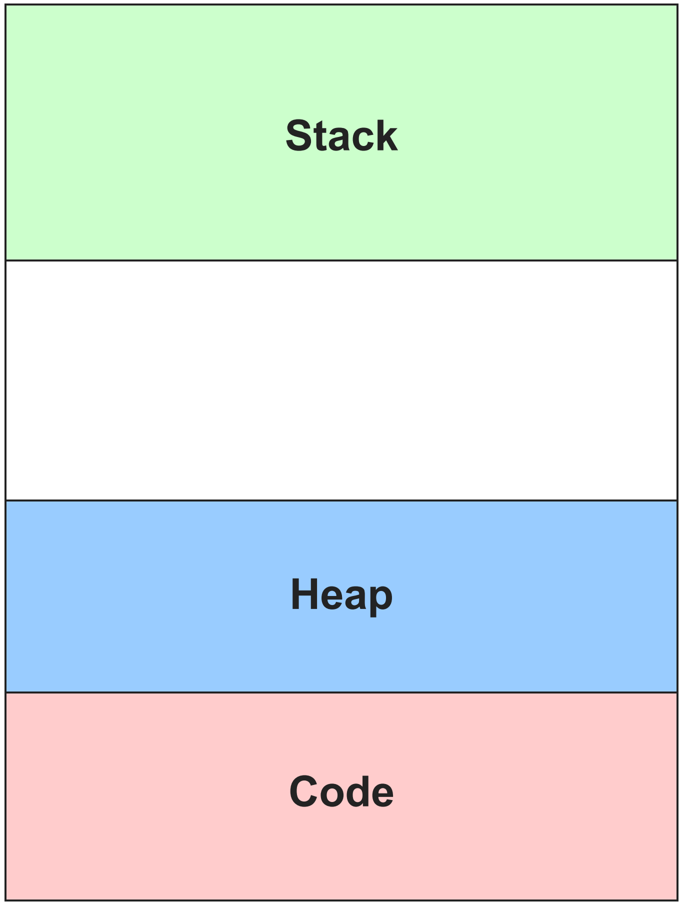
level 2
page tables

virtual address bits 21-12	physical page #
0	10
1	15
2	20
3	2

virtual address bits 21-12	physical page #
0	30
1	4
2	8
3	3

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 page #
0	389
1	invalid
2	invalid
…	invalid
1021	invalid
1022	invalid
1023	7046

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 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 this change what happens on a context switch?