EECS 482
Introduction to Operating Systems

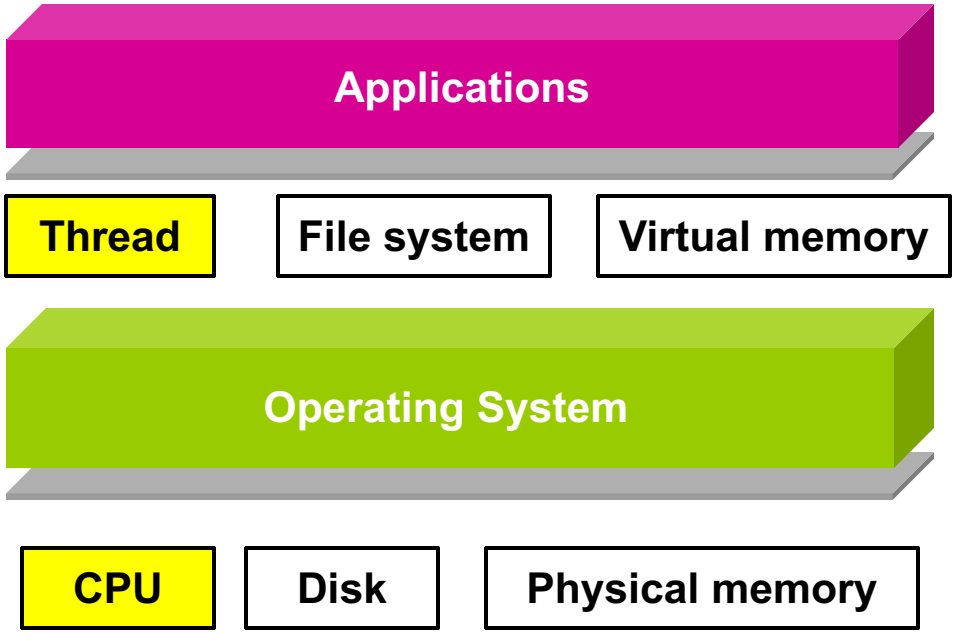
Winter 2018

Harsha V. Madhyastha

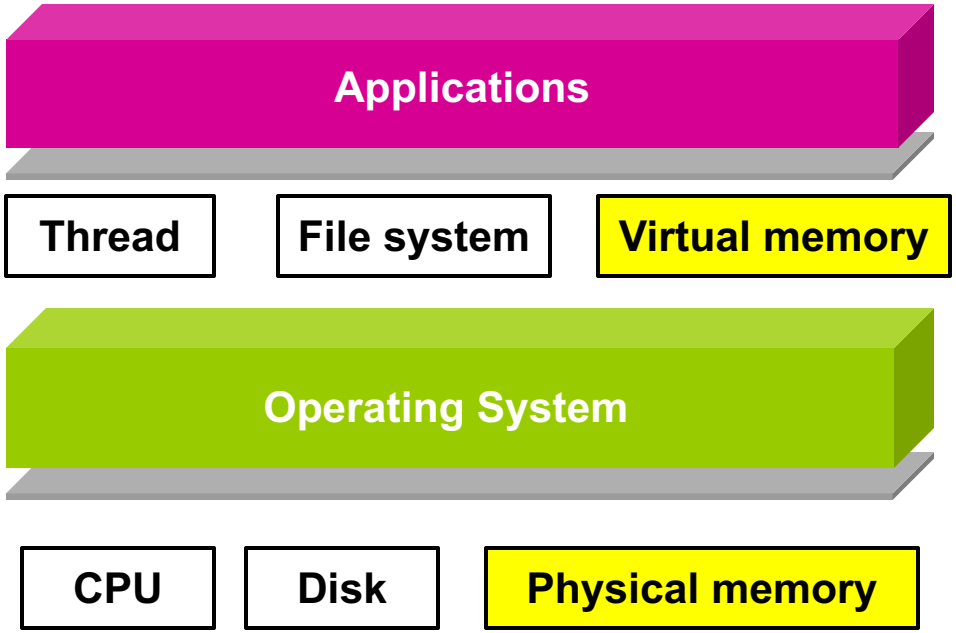
Threads and Concurrency

- Physical reality:
  - Limited # of CPUs operating independently
  - Low-level H/W support: interrupts and test_and_set
- Abstraction:
  - Threads can assume infinite CPUs
  - Locks for mutual exclusion, condition variables for ordering constraints, and semaphores for both
- Over-constrained synchronization → deadlock

OS abstractions



OS Abstractions



Memory management

- Recall: Process = Set of threads + address space
- Address space
  - All the memory space the process can use as it runs

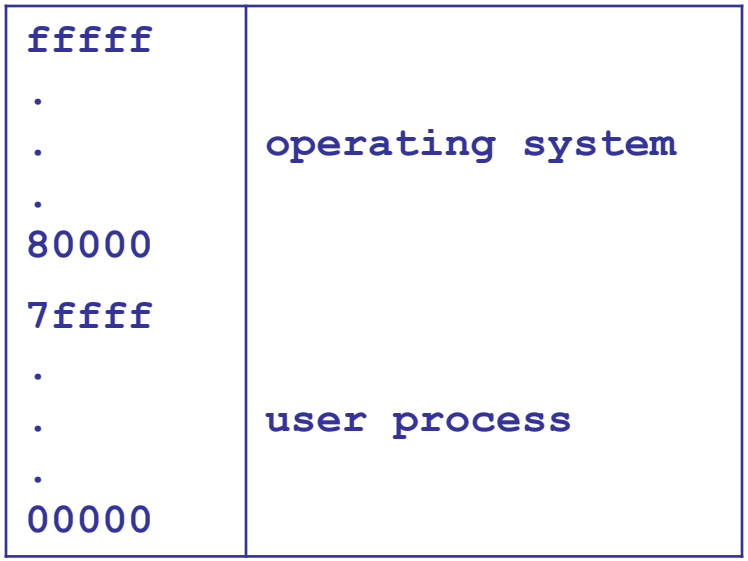
- Hardware interface:
  - Physical memory shared between processes
- Potential abstractions:
  - Allow direct access to addresses in physical memory?
  - Partition physical memory across processes?

Abstraction provided by OS

- Virtual memory: address space can be larger than physical memory
- Address independence: same numeric address can be used in different address spaces, yet remain logically distinct
- Protection: one process can't access data in another process's address space (actually controlled sharing)

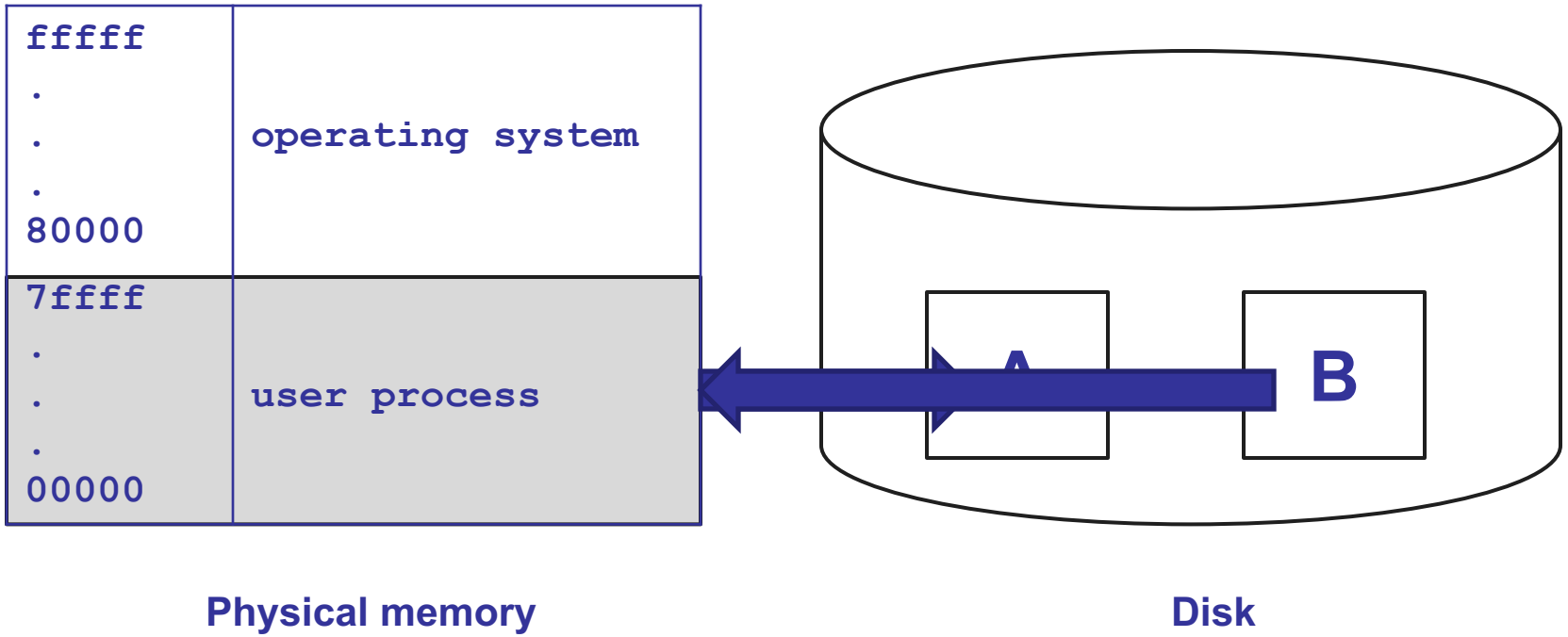
Uni-programming

- 1 process runs at a time
- Always load process into same spot in memory
- Reserve space for OS
- Virtual address = physical address



Problems?

Context switch



Uni-programming summary

- Address space abstractions:
  - Virtual memory: No
  - Address Independence: Yes
  - Protection: Yes

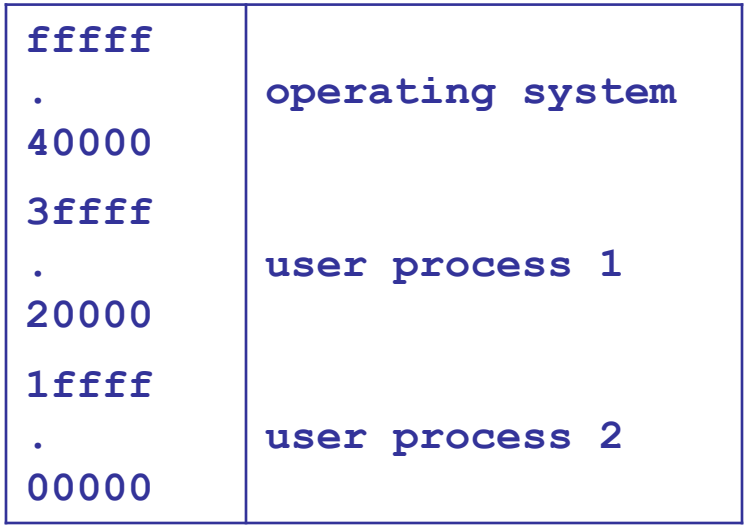
- Pro: Simple (early OSes for PCs used this)
- Con: Expensive context switch

Multi-programming

- Allow >1 address space in physical memory
- Programs written assuming addr range starts at 0

- Only 1 process can start at physical address 0
- Implies address translation necessary for address independence

Static address translation



- Compiler generates addresses starting at 0
- Linker-loader adds offset to instructions
  - E.g., MOV 0x10, %eax becomes MOV 0x20010, %eax

Register offset addressing

- Arrays, structs, pointers use indexing
  - E.g., MOV %ebx(0x10), %eax
  - (Add 0x10 to %ebx and load value at that address)
  - Any issues with this?
- %ebx could be negative or > partition size!

- Could add dynamic checks before loads/stores
  - Very expensive for general-purpose language (C)

Dynamic address translation

- Problem is application gets “last move”
  - Compiler generates machine code (app)
  - Linker-loader translates addresses (OS)
  - Register values used to calculate addresses (app)

- Dynamic translation: system has the last move
  - Hardware (MMU) translates all memory references
  - Virtual address: address used by the process
  - Physical address: address in physical memory

Dynamic address translation

- Address independence
  - Virtual addresses are scoped to 1 process
- Protection
  - One process can't refer to another's address space
- Virtual memory
  - VA only needs to be in phys. mem. when accessed
  - Allows changing translations on the fly

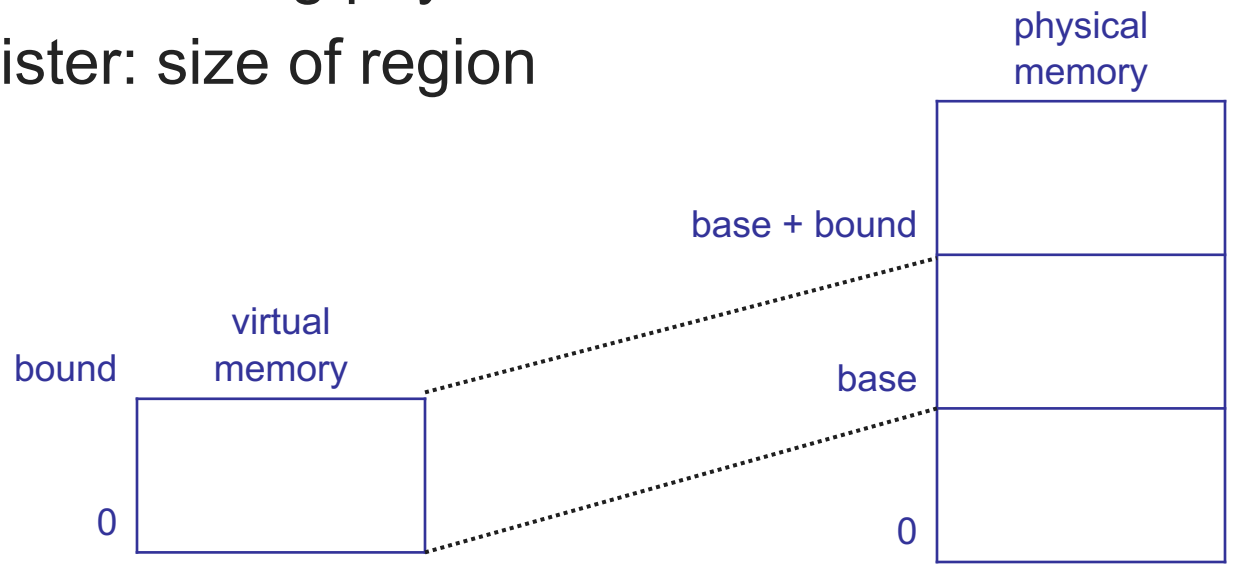
Address translation

- Many ways to implement translator

- Tradeoffs
  - Flexibility (sharing, growth, virtual memory)
  - Size of data needed to support translation
  - Speed of translation

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



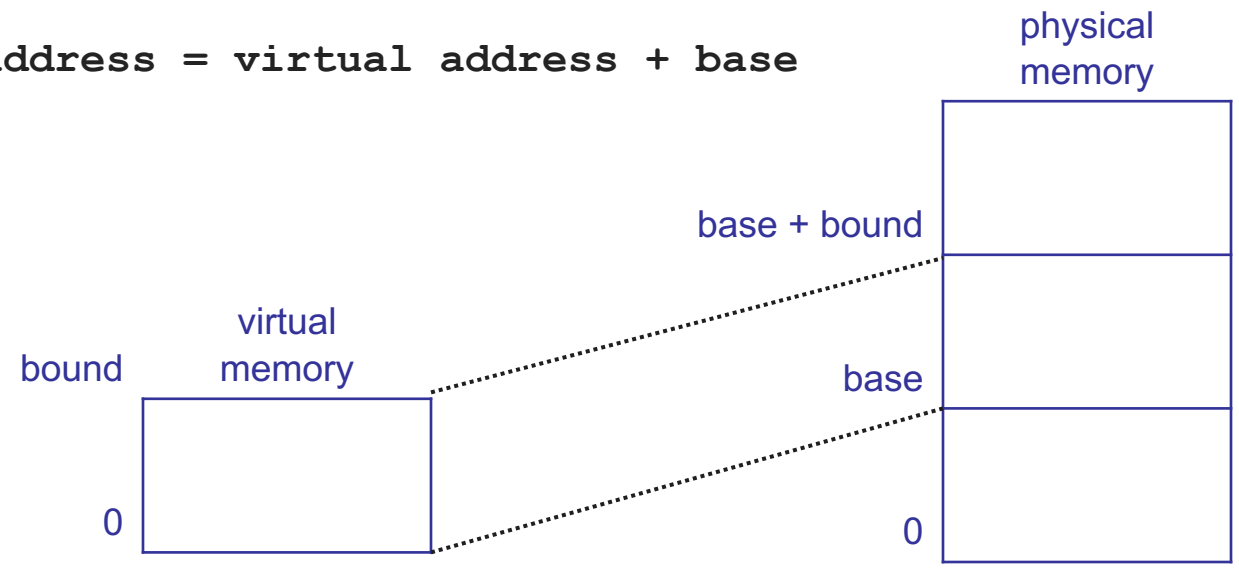
Base and bounds

- MMU Translation:

What to change on context switch?

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

How to grow address space?



Base and bounds

- Pros:
  - Fast
  - Simple hardware support
- Cons:
  - No virtual memory
  - External fragmentation
  - No controlled sharing

Writing good test cases

- Need targeted test cases and stress tests
- Targeted test case:
  - Tests 1 or a few specific things
  - Failure tells you exactly what's wrong
  - Autograder gives you results for your tests (!)
- Stress tests:
  - Lots of concurrency, activities
  - Tests for rare, non-deterministic interleavings
  - Failure doesn't say why – run under debugger?

Targeted test case

- lock() blocks when mutex held, finishes when lock released()

Thread A:

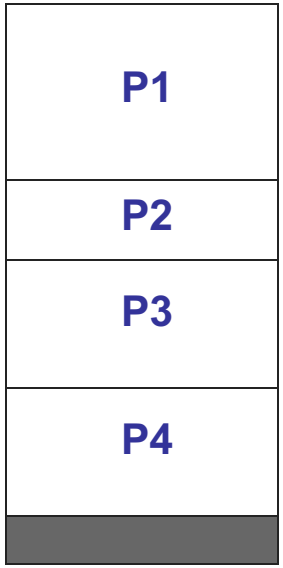
```
create thread B
m.lock()
yield()


m.unlock()
cout << "unlocked\n"
```

Thread B:

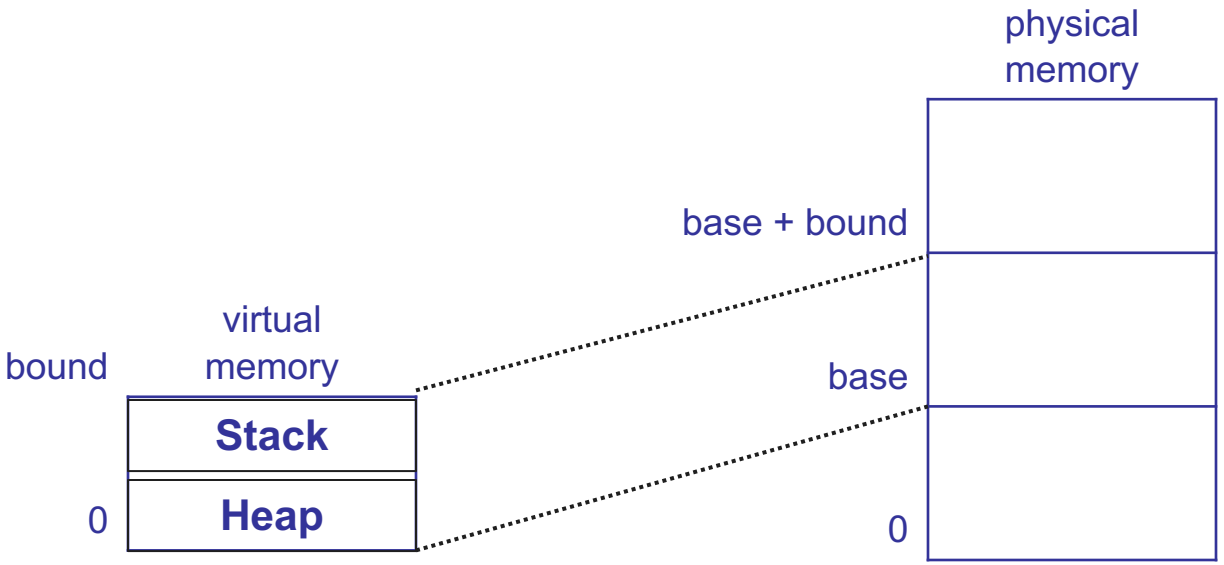
```


cout << "About to lock\n"
m.lock()


cout << "Lock released\n"
```

External fragmentation

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



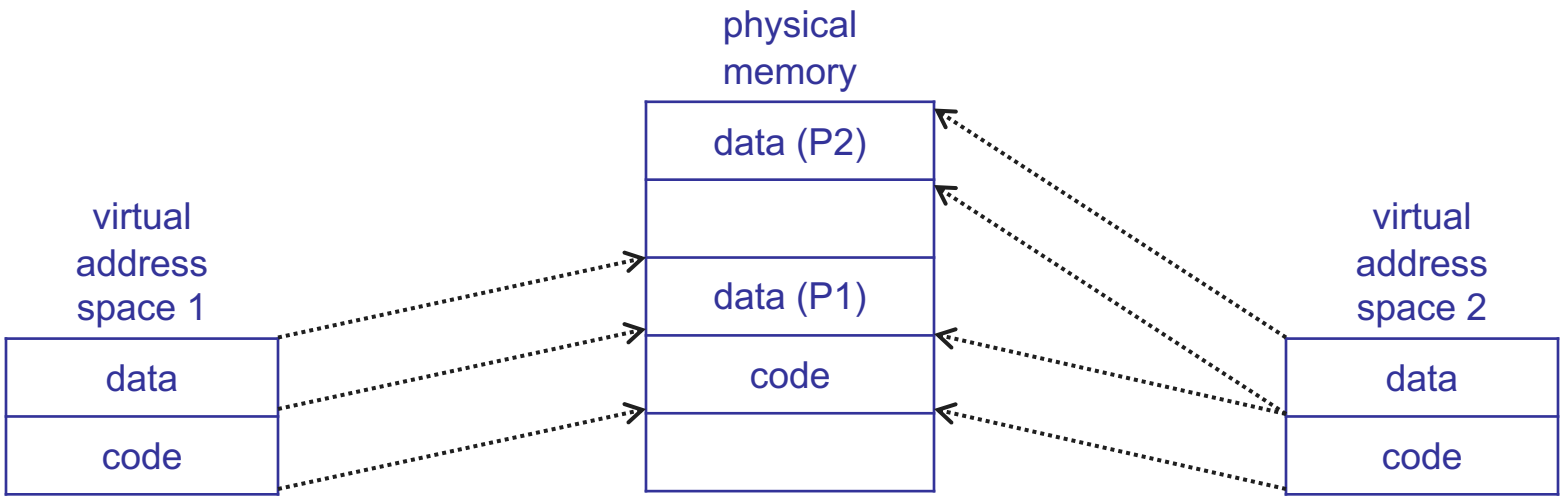
Growing address space



How can stack and heap grow independently?

Base and bounds: Sharing

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

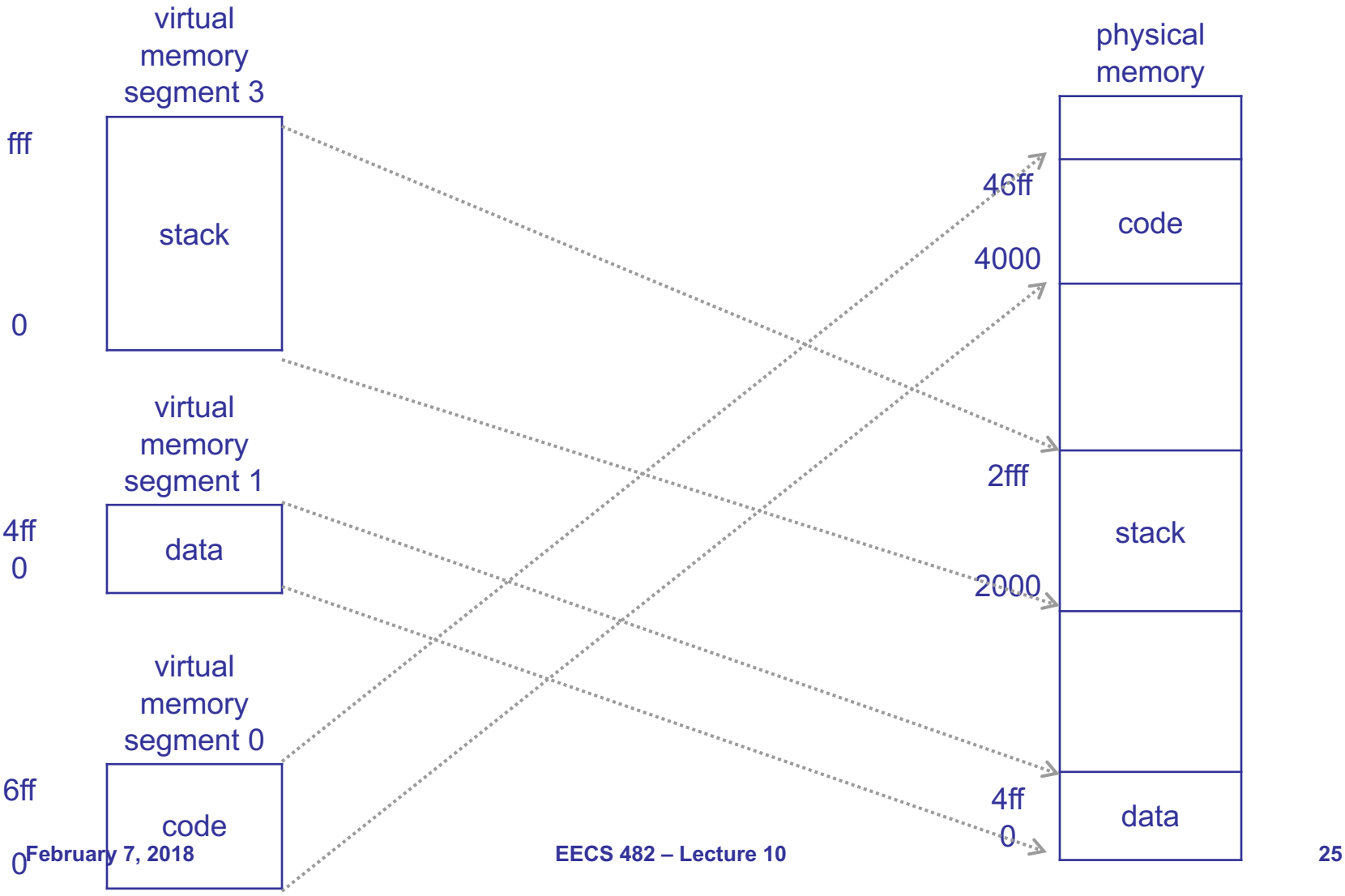


Segmentation

- Divide address space into segments

- Segment: region of memory that is:
  - Contiguous in physical memory
  - Contiguous in virtual address space
  - Variable size

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
- Ways to specify the segment number:
  - High bits of address
  - Special register
  - Implicit to instruction opcode

Segmentation: Translation

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)?

Valid vs. invalid addresses

- Not all virtual addresses are valid
  - Valid → address is part of virtual address space
  - Invalid → virtual address is illegal to access
    - » Accessing invalid address causes trap to OS

- 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 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

- External fragmentation
- Can address space be larger than physmem?

- How can we:
  - Make memory allocation easy
  - Not have to worry about external fragmentation
  - Allow address space size to be > physical memory

Next time ...

- Paging