Recap

Physical memory: the reality

- Limited size
- Fragmented

Address space: the virtual memory view to process/programmer

- Layout (code/statical data, heap, stack)
- Contiguous
- Large (e.g., 0-2^{wordsize}-1)
- Every address in range is available

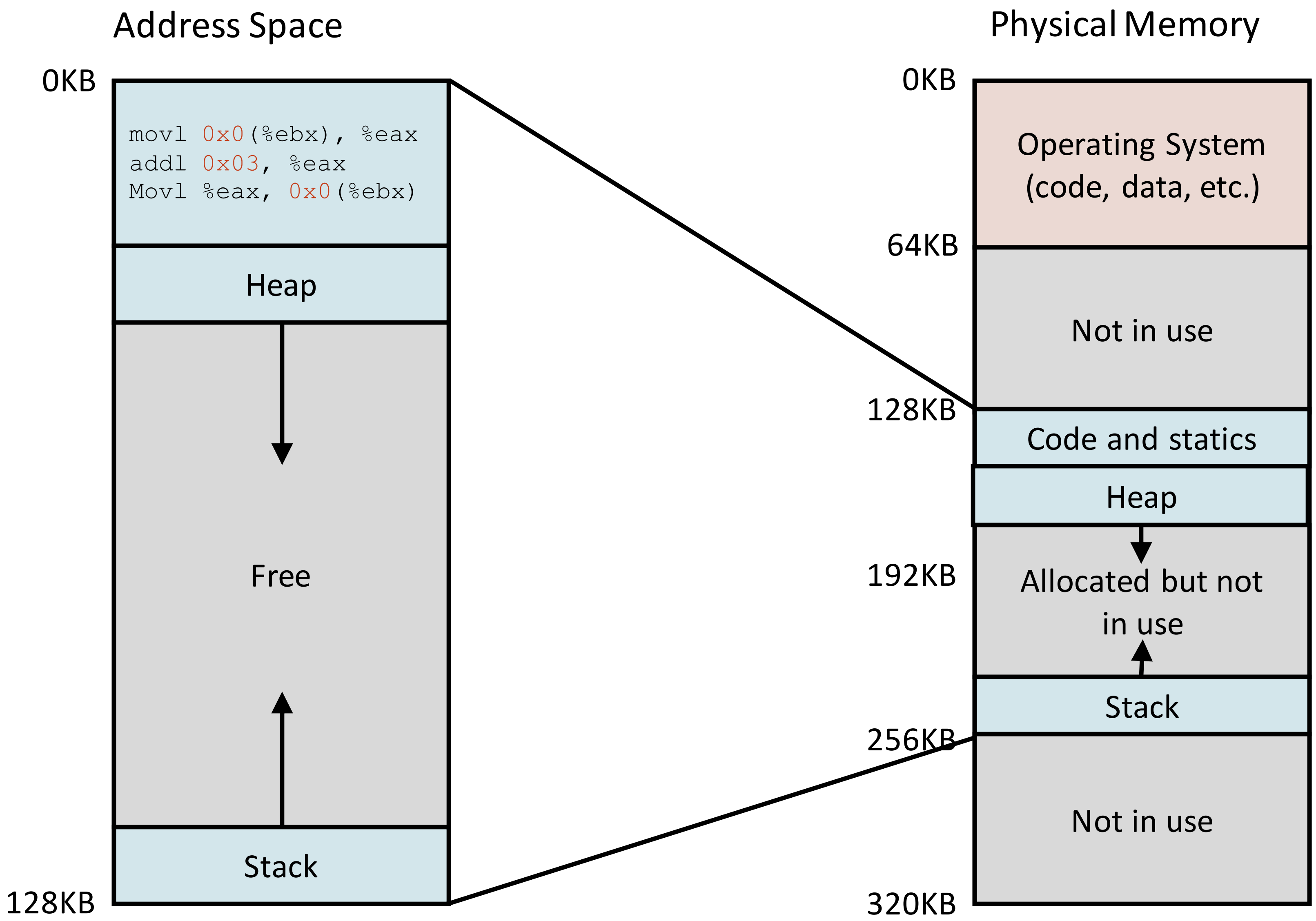
Memory virtualization: delivering address spaces to processes based on messy physical memory

Assumptions (only for now)

- A process' address space must be placed contiguously in physical memory.
- The size of the address space is not too big; it is less than the size of physical memory.
- Each address space is exactly the same size.

(We will remove these assumptions in later lectures)

Example: Memory Relocation (i.e., VM->PM Mapping)



Software-Based Translation Method

Loader: the purpose is to load the binary program on disk into the process memory

Some early loaders also had the job of translating all addresses found in instructions from virtual to physical locations

Translation is performed once (**statically**) before the process begins execution

`movl(1000, %eax)` → `movl(4000, %eax)`

Disadvantages:

- the loader needs to be trusted code (or no memory protection)
- relocation after the process starts is costly

Hardware-based Translation (Dynamic Relocation) Method: Base and Bounds

The **base and bounds** method requires two CPU registers

- **base** – points to start of process in physical memory
- **bounds** – points to maximum legal address for process

When instruction is executed, all addresses translated by hardware

`physical address = virtual address + base`

Bounds used in one of two ways:

- Bounds specify the virtual bounds: If virtual address > bounds, access is illegal and so trap to kernel
- Bounds specify the physical bounds: If physical address > bounds, access is illegal and so trap to kernel

Allows **dynamic relocation** of process memory

MMU

The hardware responsible is the **Memory Management Unit (MMU)**

It is typically part of the CPU but sits between the core and the address buss

Translates all addresses between CPU and main memory

Example

Instruction in code:

```
128: mov 1000, %eax
```

0. Assume base: 32,768 and bounds: 128K
1. Program Counter (PC) is incremented to 128
2. CPU begins fetching instruction by reading from address 128
3. MMU checks 128 < bounds (valid virtual address); translates 128 to 32,896 = 32768+128 and memory is read
4. CPU decodes instruction and requests a read from address 1000
5. MMU checks 1000<bounds (valid virtual address); translates 1000 to 33,768 = 32768 + 1000 and memory is read
6. CPU finishes execution of instruction

Exception Handling

Memory access out of bounds results in a trap

OS typically terminates process

Hardware Requirements

Hardware Requirement	Common Implementation
Privilege mode	Kernel mode
Base and bounds registers	
Translate virtual address	**MMU** intercepts all addresses between CPU and bus
Privileged instructions to update base and bounds	Write to registers (kernel mode)
Privileged instructions to register exception handlers	Write to interrupt vector table (kernel mode)
Ability to raise exceptions	**Trap** when read out of bounds

Put together: Limited Direct Execution (Dynamic Relocation)

OS @ boot (kernel mode)	**Hardware**	**(No Program Yet)**
initialize trap table		
	remember addresses of...	
	system call handler	
	timer handler	
	illegal mem-access handler	
	illegal instruction handler	
start interrupt timer		
	start timer; interrupt after X ms	
initialize process table		
initialize free list		

Limited Direct Execution (Dynamic Relocation)

OS @ run (kernel mode)	**Hardware**	**Program (user mode)**
To start process A:		
allocate entry in process table		
alloc memory for process		
set base/bound registers		
return-from-trap (into A)		
	restore registers of A	
	move to **user mode**	
	jump to A's (initial) PC	
		Process A runs
		Fetch instruction
	translate virtual address	
	perform fetch	
		Execute instruction
	if explicit load/store:	
	ensure address is legal	
	translate virtual address	
	perform load/store	
		(A runs...)
	Timer interrupt	
	move to **kernel mode**	
	jump to handler	

Limited Direct Execution (Dynamic Relocation)

Handle timer
decide: stop A, run B
call `switch()` routine
  save regs(A)
    to proc-struct(A)
  (including base/bounds)
  restore regs(B)
    from proc-struct(B)
  (including base/bounds)
return-from-trap (into B)

restore registers of B
move to **user mode**
jump to B's PC

Process B runs
  Execute bad load

Load is out-of-bounds;
move to **kernel mode**
jump to trap handler

Handle the trap
  decide to kill process B
  deallocate B's memory
  free B's entry
    in process table

L8: Segmentation

(based on Ch 16)

Slides revised from
Matthew Tancreti
Iowa State University

Limitations of Base-and-bounds

Base and bounds requires direct translation from a whole address space to physical memory

Assumes program knows in advance how much dynamic memory will be required

Growing process's memory dynamically is very difficult

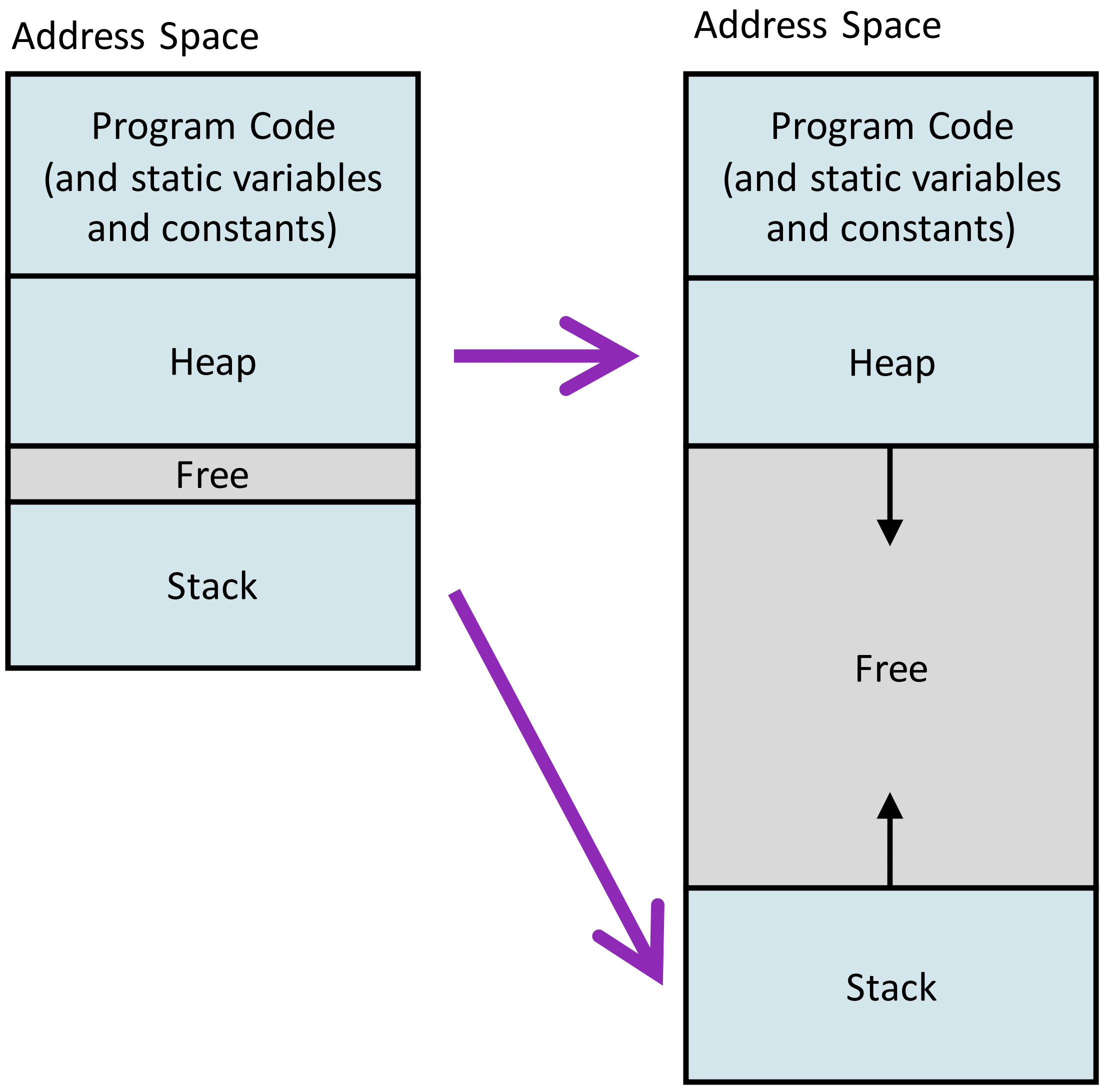
How to remove limitations of physical memory from address space?

Problem with Base-and-Bounds

What happens when address space is full?

Using base and bounds, process needs to be copied to larger region in memory

All pointers to stack need to be updated



Example Address Space in Linux

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

int main(int argc, char *argv[]) {
    printf("code  %p\n", main);
    printf("heap  %p\n", malloc(1));
    printf("stack %p\n", &argc);
    return 0;
}
```

```
code  0x401136       =           4,198,710
heap  0x1d096b0      =          30,447,280
stack 0x7ffc6a46bf4c = 140,722,091,507,532    140TB address space size!
```

The allowed address space can be very large;
the size of the actually used part can change very dynamically!

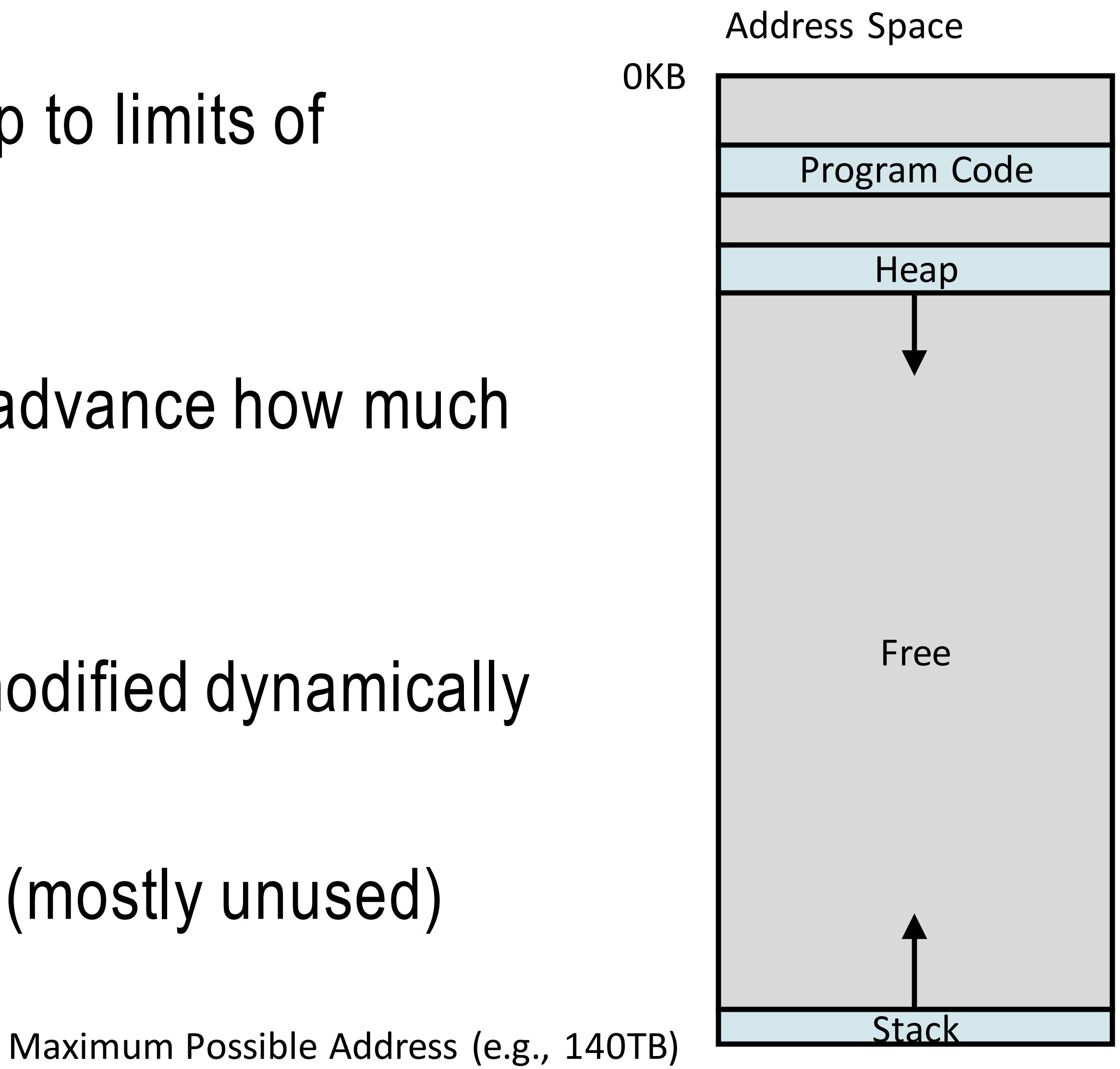
Goal: Remove Physical Limitations from Address Space

Make the address space massive (up to limits of addressable size)

Program doesn’t need to declare in advance how much memory it will need

Address space doesn’t need to be modified dynamically

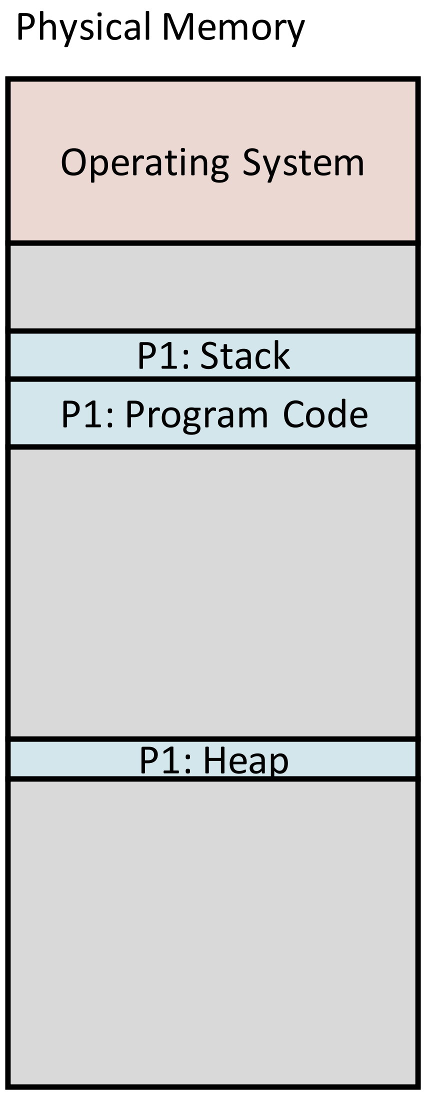
Known as a **sparse address space** (mostly unused)



Segmentation

How to allow for sparce address space in physical memory?

Segmentation means we can locate parts of the address space independently in physical memory

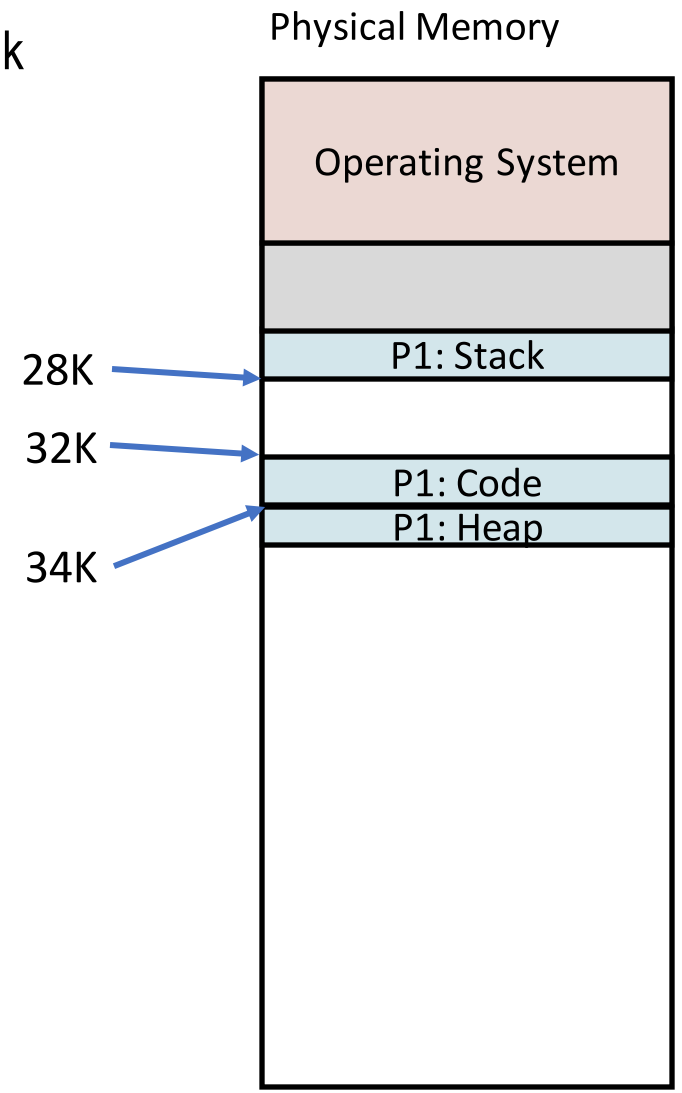


Hardware Requirements for Segmentation

Address space divided to 3 segments: Code (+ static data); Heap; Stack

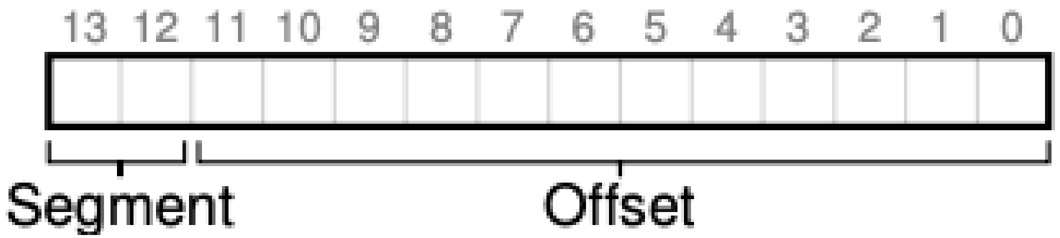
Registers for the start and size of each segment

Segment	Base register	Size register
Code	32K	2K
Heap	34K	2K
Stack	28K	2K



How to Translate Addresses?

Virtual Address:



12

0x3000

```
// get top 2 bits of 14-bit VA
Segment = (VirtualAddress & SEG_MASK) >> SEG_SHIFT
// now get offset
Offset = VirtualAddress & OFFSET_MASK
if (Offset >= Bounds[Segment])
    RaiseException(PROTECTION_FAULT)
else
    PhysAddr = Base[Segment] + Offset
    Register = AccessMemory(PhysAddr)
```

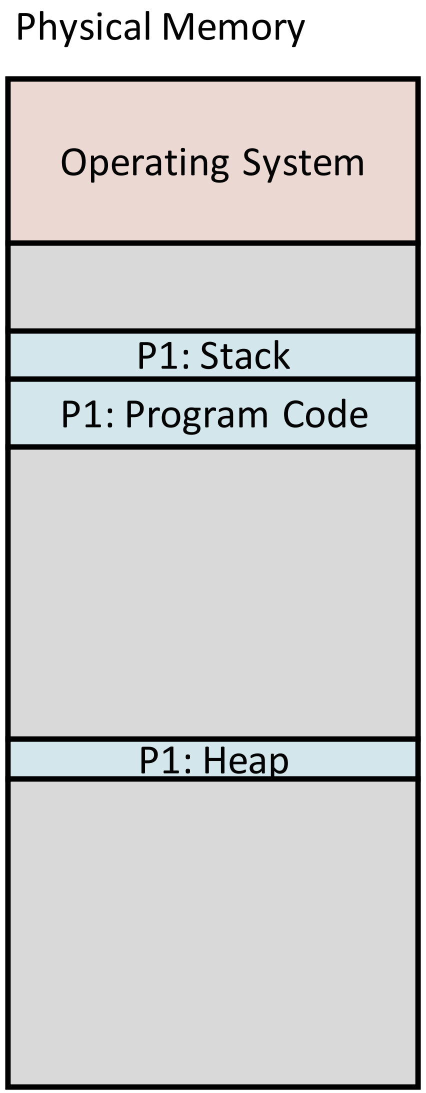
0x0fff

Question

What if physical memory runs out of space for a segment and needs to relocate it? Will pointers in the program need to be updated?

No, address space does not depend on where segments are located in physical memory.

Only base and bounds registers change.

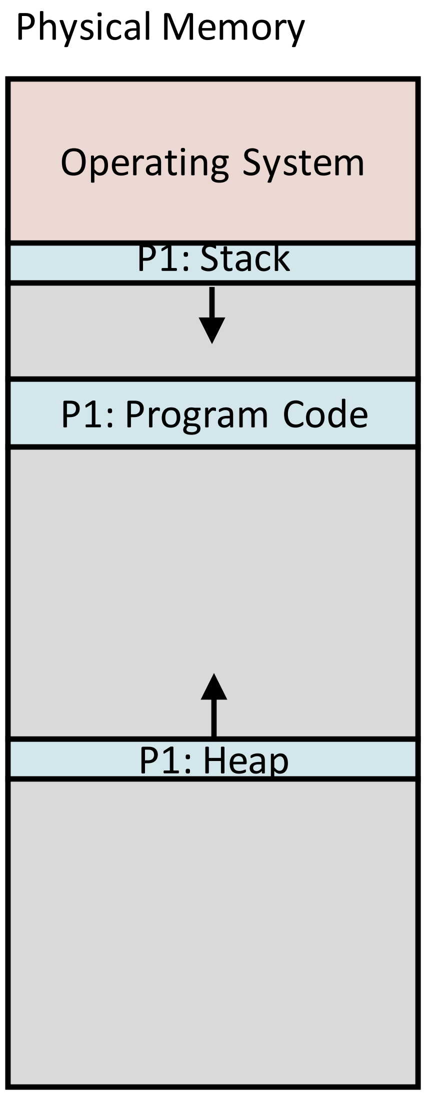


Independent Direction of Segment Growth

We can even allow segments to grow in different directions

A set of registers can indicate if a segment grows up or down

Segment	Base register	Size register	Grows Positive?
Code	32K	2K	1
Heap	34K	2K	0
Stack	28K	2K	1

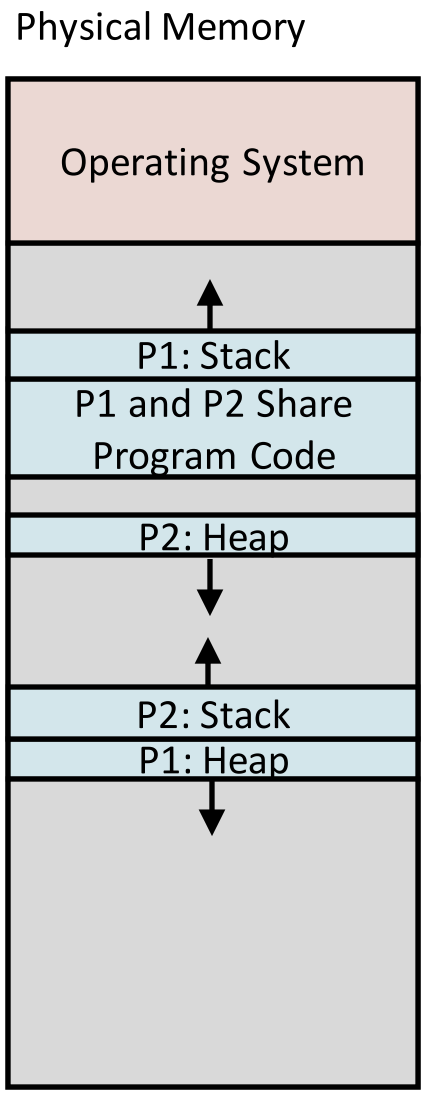


Sharing

Protection registers can enable **sharing**

Example: two processes are executing the same code. If code segment is read-only no danger of processes corrupting each other

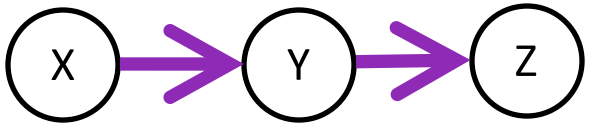
Segment	Base register	Size register	Grows Positive?	Protection
Code	32K	2K	1	Read-execute
Heap	34K	2K	1	Read-Write
Stack	28K	2K	0	Read-Write



Free Memory

Segments are in contiguous regions of physical memory

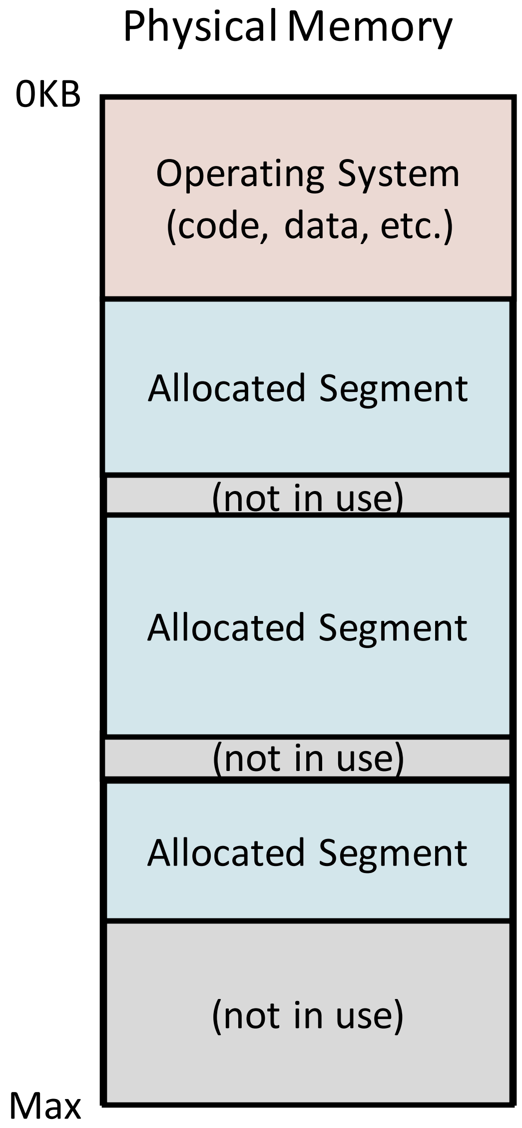
To allocate a new segment, OS must keep a list of free memory



Simple solution is a linked list of free regions of memory

On new allocation search for first open spot that has sufficient memory (**first fit** strategy)

Best fit strategy searches for smallest region of free memory that will fit the segment



Fragmentation

Segments are in contiguous regions of physical memory

Gaps result in **external fragmentation** (wasted physical memory)

Not big enough to fit a full segment, so can't be used

Compaction used to reclaim the fragments