Memory Management

Abstraction is our Business

- What I have
  - A single (or a finite number) of CPUs
  - Many programs I would like to run
- What I want: a Thread
  - Each program has full control of one or more CPUs

Abstraction is our Business

- What I have
  - A certain amount of physical memory
  - Multiple programs I would like to run
    - together, they may need more than the available physical memory
- What I want: an Address Space
  - Each program has as much memory as the machine's architecture will allow to name
  - All for itself

Address Space

- Set of all names used to identify and manipulate unique instances of a given resource
  - memory locations (determined by the size of the machine's word)
    - for 32-bit-register machine, the address space goes from 0x00000000 to 0xFFFFFFFF
  - phone numbers (XXX) (YYY-YYYY)
  - colors: R (8 bits) + G (8 bits) + B (8 bits)

Virtual Address Space: An Abstraction for Memory

- Virtual addresses start at 0
- Heap and stack can be placed far away from each other, so they can nicely grow
- Addresses are all contiguous
- Size is independent of physical memory on the machine

0 Program Code

Heap

free

63KB

Stack

64KB

Physical Address Space: How memory may actually look

- Processes loaded in memory at some memory location
  - virtual address 0 is not loaded at physical address 0
- Multiple processes may be loaded in memory at the same time, and yet…
- …physical memory may be too small to hold even a single virtual address space in its entirety
  - 64-bit registers, anyone?

0

OS

free

Process 2 code, data, etc

free

free

Process 1 code, data, etc

Process 3 code, data, etc

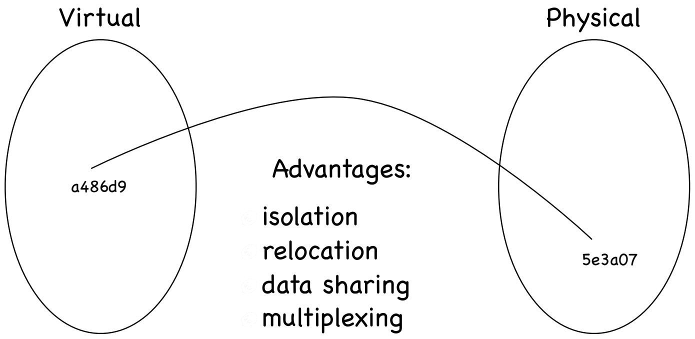
free

512K

II. Memory Isolation

Step 2: Address Translation

- Implement a function mapping into

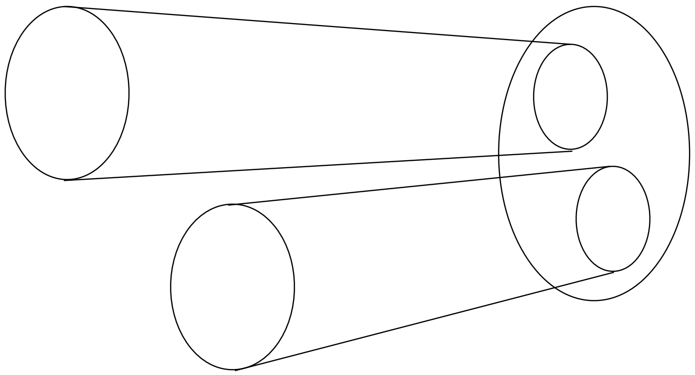


Advantages:

- isolation
- relocation
- data sharing
- multiplexing

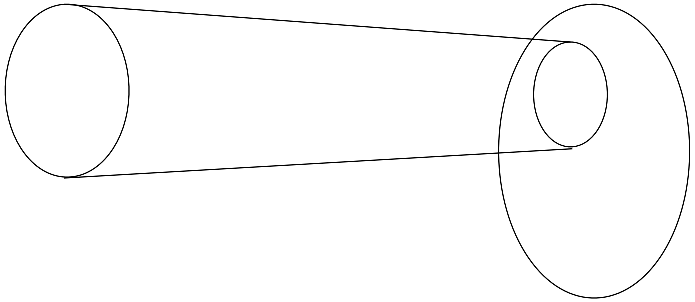
Isolation

- At all times, functions used by different processes map to disjoint ranges — aka “Stay in your room!”



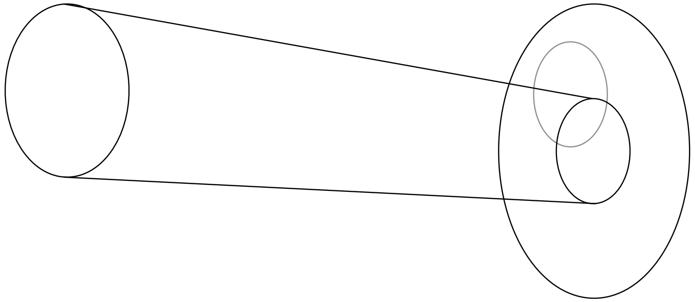
Relocation

- The range of the function used by a process can change over time



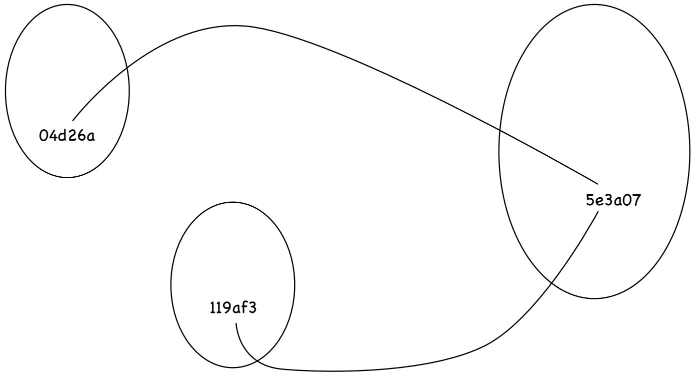
Relocation

- The range of the function used by a process can change over time — “Move to a new room!”



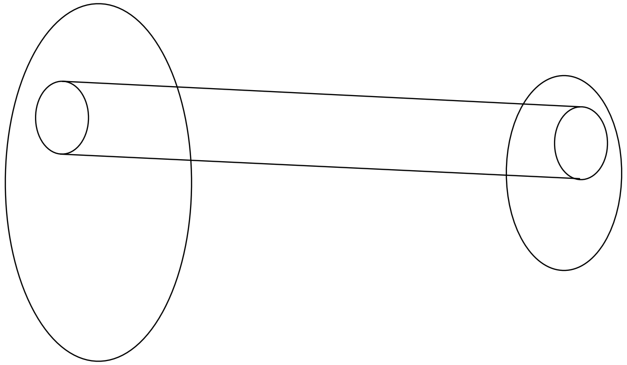
Data Sharing

- Map different virtual addresses of distinct processes to the same physical address — “Share the kitchen!”



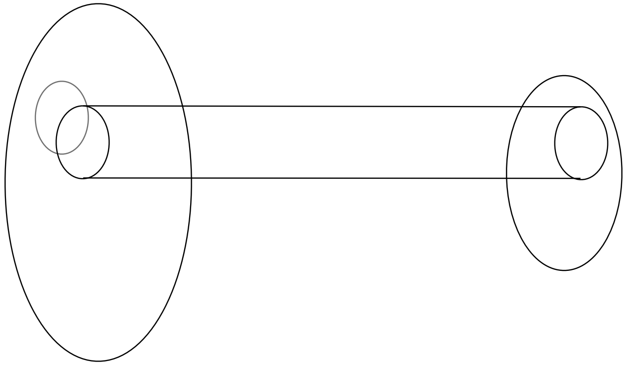
Multiplexing

- Create illusion of almost infinite memory by changing domain (set of virtual addresses) that maps to a given range of physical addresses — ever lived in a studio?



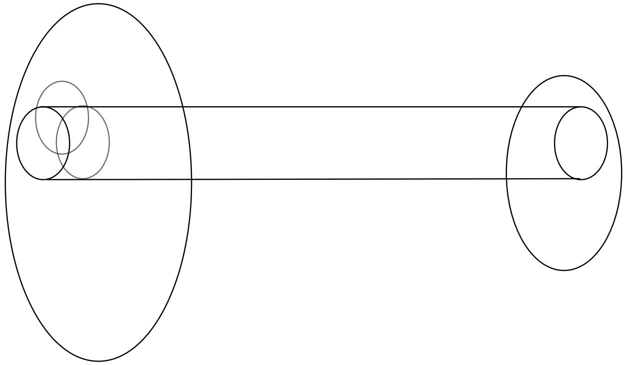
Multiplexing

- The domain (set of virtual addresses) that map to a given range of physical addresses can change over time



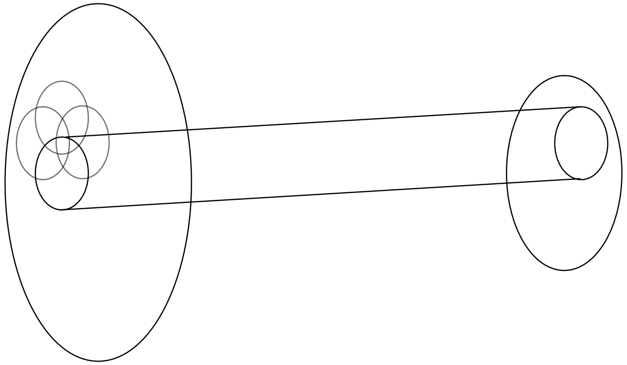
Multiplexing

- The domain (set of virtual addresses) that map to a given range of physical addresses can change over time



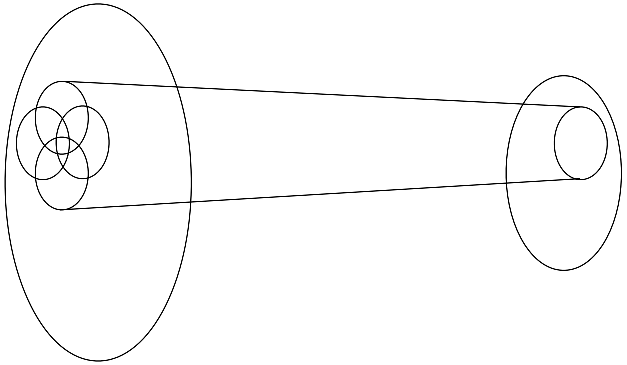
Multiplexing

- The domain (set of virtual addresses) that map to a given range of physical addresses can change over time



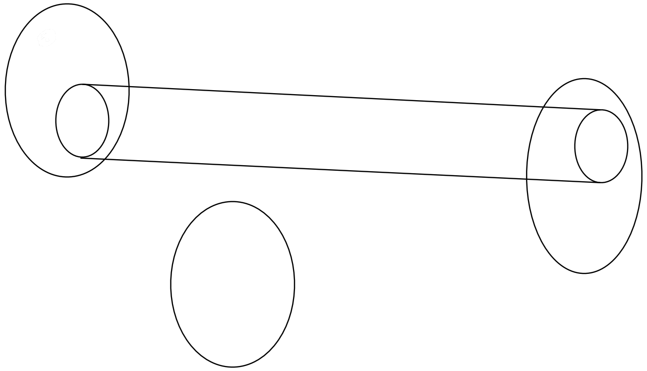
Multiplexing

- The domain (set of virtual addresses) that map to a given range of physical addresses can change over time



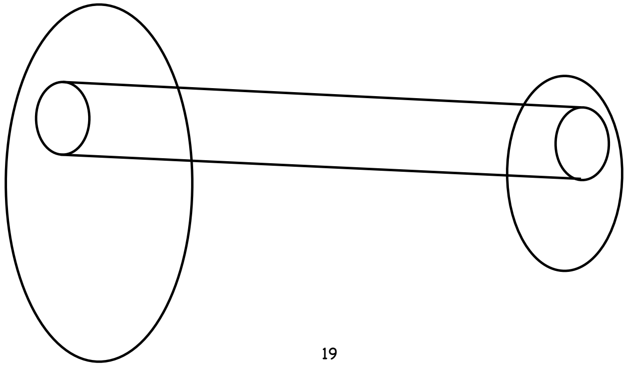
More Multiplexing

- At different times, different processes can map part of their virtual address space into the same physical memory — change tenants!



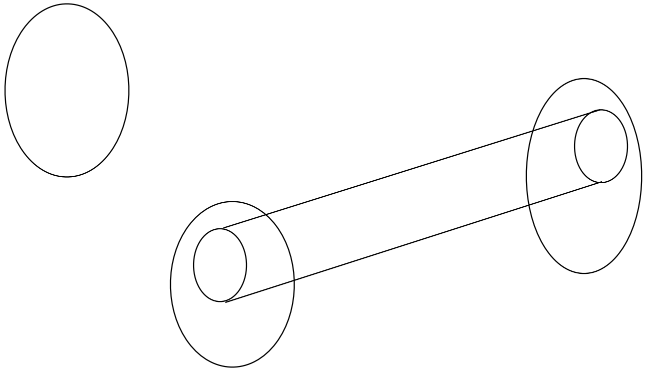
Contiguity

- Contiguous virtual addresses need not map to contiguous physical addresses



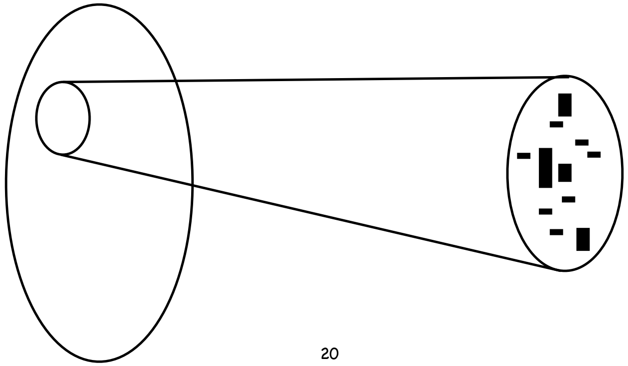
More Multiplexing

- At different times, different processes can map part of their virtual address space into the same physical memory — change tenants!



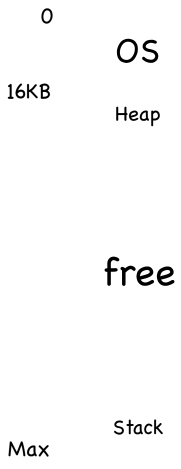
Contiguity

- Contiguous virtual addresses need not map to contiguous physical addresses



The Identity Mapping

- Map each virtual address onto the identical physical address
  - Virtual and physical address spaces have the same size
  - Run a single program at a time
    - OS can be a simple library
    - very early computers
- Friendly amendment: leave some of the physical address space for the OS
  - Use loader to relocate process
    - early PCs



More sophisticated address translation

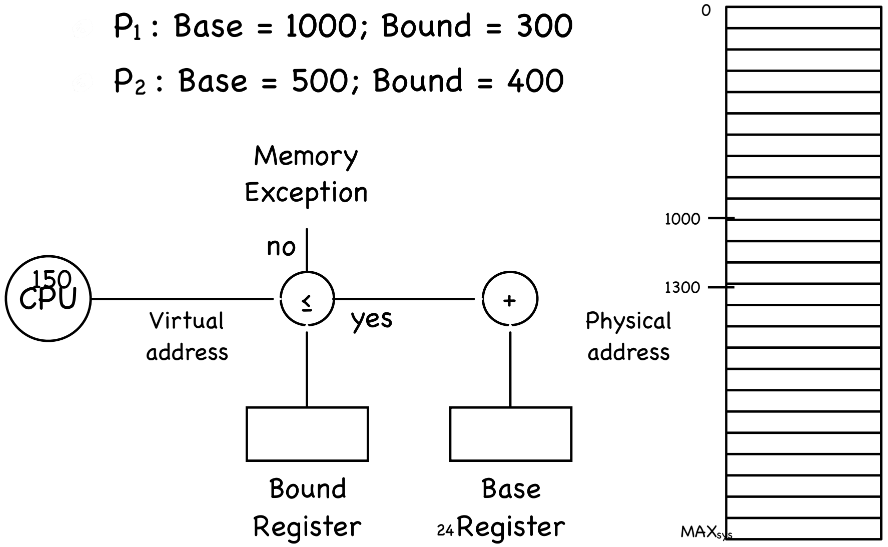
- How to perform the mapping efficiently?
  - So that it can be represented concisely?
  - So that it can be computed quickly?
  - So that it makes efficient use of the limited physical memory?
  - So that multiple processes coexist in physical memory while guaranteeing isolation?
  - So that it decouples the size of the virtual and physical addresses?
- Ask hardware for help!

Base & Bound

- Goal: allow multiple processes to coexist in memory while guaranteeing isolation
- Needed hardware
  - two registers: Base and Bound (a.k.a. Limit)
  - Stored in the PCB
- Mapping
  - pa = va + Base
    - as long as 0 ≤ va ≤ Bound
  - On context switch, change B&B (privileged instruction)

Base & Bound

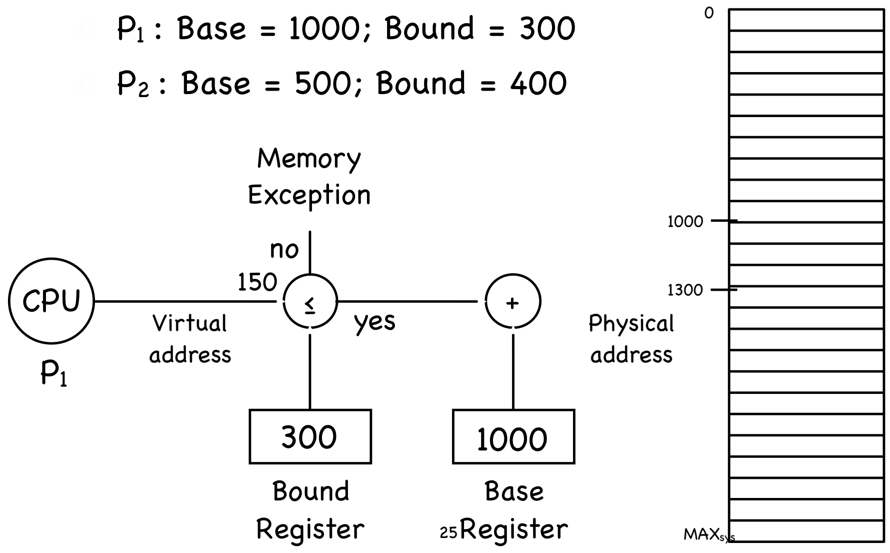
- P_1 : Base = 1000; Bound = 300
- P_2 : Base = 500; Bound = 400



Base & Bound

P_1 : Base = 1000; Bound = 300

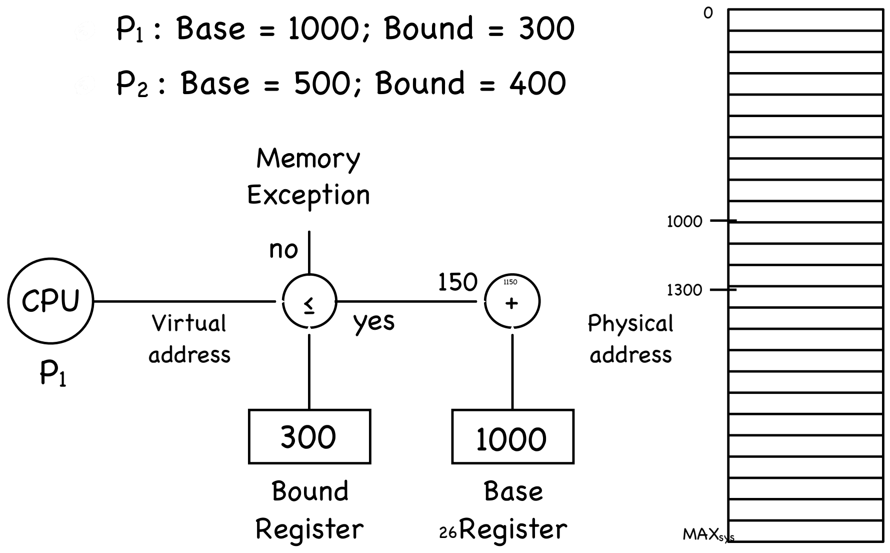
P_2 : Base = 500; Bound = 400



Base & Bound

P_1 : Base = 1000; Bound = 300

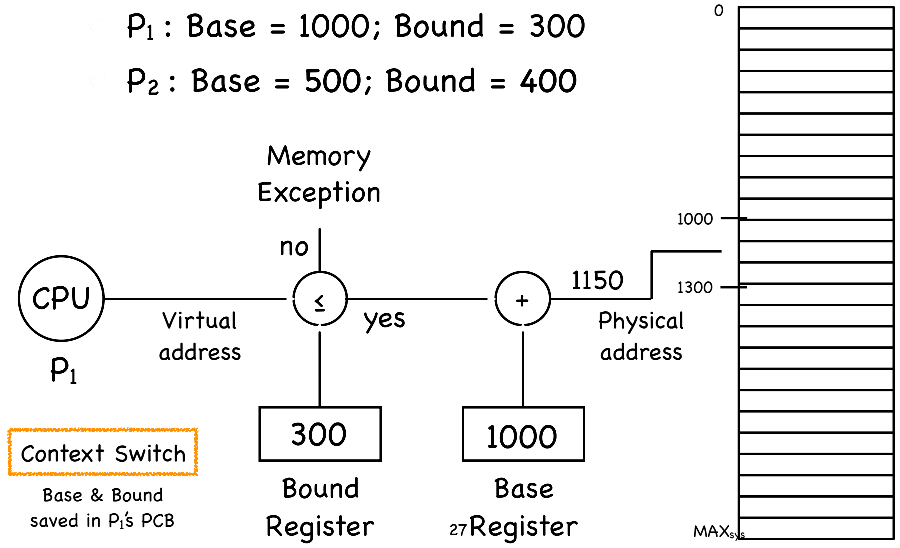
P_2 : Base = 500; Bound = 400



Base & Bound

P_1 : Base = 1000; Bound = 300

P_2 : Base = 500; Bound = 400



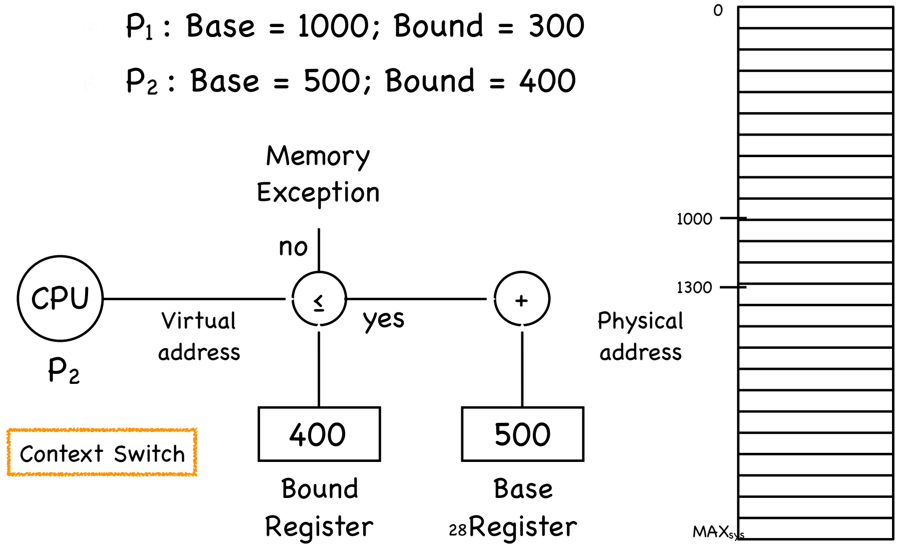
Context Switch

Base & Bound saved in P_1's PCB

Base & Bound

P_1 : Base = 1000; Bound = 300

P_2 : Base = 500; Bound = 400



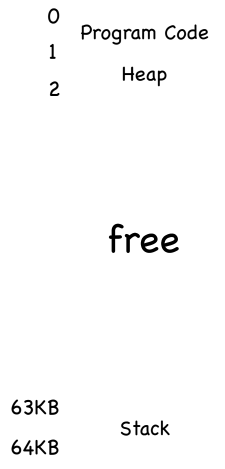
Context Switch

On Base & Bound

- Contiguous Allocation
  - contiguous virtual addresses are mapped to contiguous physical addresses
- But mapping entire address space to physical memory
  - is wasteful
    - lots of free space between heap and stack...
    - makes sharing hard
  - does not work if the address space is larger than physical memory
    - think 64-bit registers...

E Pluribus Unum

- Address spaces have structure!
- An address space comprises multiple segments
  - contiguous sets of virtual addresses, logically connected
    - heap, code, stack, (and also globals, libraries...)
  - each segment can be of a different size

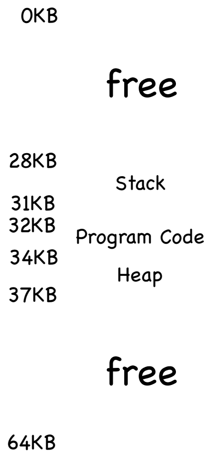


0
1
2
Program Code
Heap
free
63KB
64KB
Stack

Segmentation: Generalizing Base & Bound

- Base & Bound registers to each segment
  - each segment is independently mapped to a set of contiguous addresses in physical memory
    - no need to map unused virtual addresses

Segment	Base	Bound
Code	32K	2K
Heap	34K	3K
Stack	28K	3K



0KB
free
28KB
Stack
31KB
32KB
Program Code
34KB
Heap
37KB
free
64KB

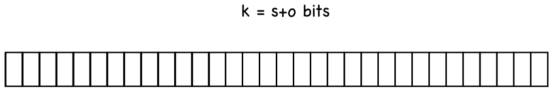
(not to scale)

Segmentation

- Goal: Supporting large address spaces (while allowing multiple processes to coexist in memory)
- Needed hardware
  - two registers (Base and Bound) per segment
    - Stored in the PCB
  - a segment table, stored in memory, at an address pointed to by a Segment Table Register (STBR)
    - STBR stored in the PCB

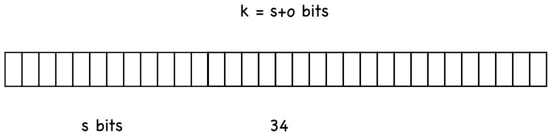
Segmentation: Mapping

- How do we map a virtual address to the appropriate segment?
  - Read VA as having two components
    - s most significant bits identify the segment
      - at most segments
    - o remaining bits identify offset within segment
      - each segment's size can be at most bytes



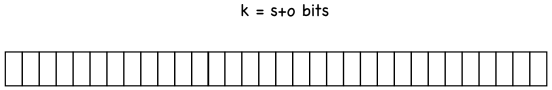
Segmentation: Mapping

- How do we map a virtual address to the appropriate segment?
  - Read VA as having two components
    - s most significant bits identify the segment
      - at most segments
    - o remaining bits identify offset within segment
      - each segment's size can be at most bytes



Segmentation: Mapping

- How do we map a virtual address to the appropriate segment?
  - Read VA as having two components
    - s most significant bits identify the segment
      - at most segments
    - o remaining bits identify offset within segment
      - each segment's size can be at most bytes



s bits

o bits

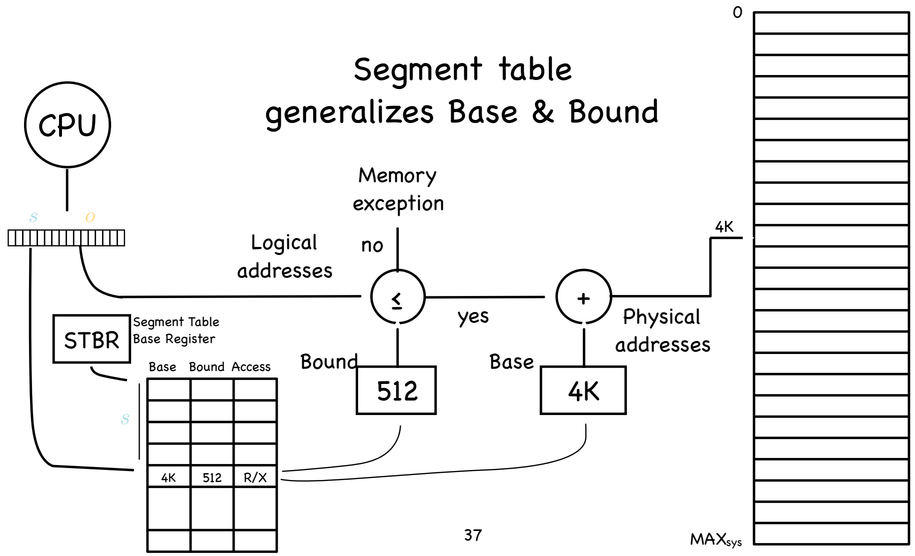
Segment Table

- Use s bits to index to the appropriate row of the segment table

	Base	Bound	Access
Code	32K	2K	Read/Execute
Heap	34K	3K	Read/Write
Stack	28K	3K	Read/Write

- Segments can be shared by different processes
  - use protection bits to determine if segment is shared Read only (maintaining isolation) or Read/Write
    - e.g., processes can share code segment while keeping data private

Implementing Segmentation



Segments and Dynamically Allocated Memory

- Memory on heap and stack dynamically allocated
  - memory reallocated to new process must be zeroed to avoid leaking info, but zeroing memory is expensive
- Zero-on-reference
  - Start with few KB
  - When program uses memory outside zero-ed area:
    - Segmentation fault into kernel, which
      - Allocates (and zeroes) some memory
      - Modifies segment table
      - Resumes process

Revisiting fork()

- Copying an entire address space can be costly…
  - especially if you proceed to obliterate it right away with exec()!

Revisiting fork(): Segments to the Rescue

- Instead of copying entire address space, copy just segment table (the VA->PA mapping)

	Base	Bound	Access
Code	32K	2K	RX
Heap	34K	3K	RW
Stack	28K	3K	RW

Parent

	Base	Bound	Access
Code	32K	2K	RX
Heap	34K	3K	RW
Stack	28K	3K	RW

Child

- but change all writeable segments to read only

Revisiting fork(): Segments to the Rescue

- Instead of copying entire address space, copy just segment table (the VA->PA mapping)

	Base	Bound	Access
Code	32K	2K	RX
Heap	34K	3K	R
Stack	28K	3K	R

Parent

	Base	Bound	Access
Code	32K	2K	RX
Heap	34K	3K	R
Stack	28K	3K	R

Child

- but change all writeable segments to read only
- Segments in VA spaces of parent and child point to same locations in physical memory

Copy on Write (COW)

- When trying to modify an address in a read-only segment:
  - exception!
    - exception handler copies just the affected segment, and changes both the old and new segment to writeable
- If exec() is immediately called, only stack segment is copied!