Memory Management Basics

Don Porter

Portions courtesy Emmett Witchel and Kevin Jeffay

Background Detour: Splitting Numbers

- In elementary school, one typically learns about the “ones”, “tens”, “hundreds”, etc.
  - E.g., 13 + 24, can be modeled as: (10 * (1 + 2)) + (3 + 4)
- One can apply the same reasoning to space:
  - Room numbers in SN/FB: hundreds digit indicates the floor, remaining digits indicate position within floor
  - Street address: lower 2 digits indicate house number, upper digits indicate block
- Or an array of 10-byte sub-arrays:

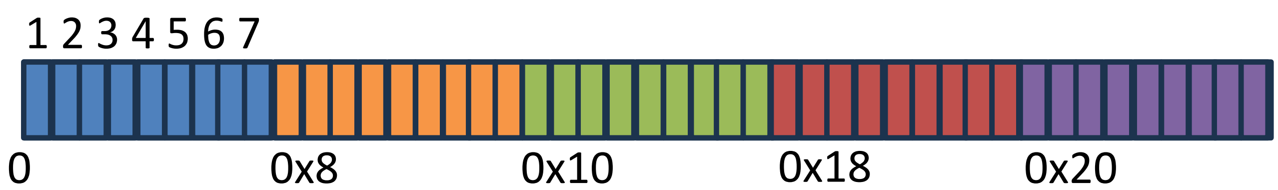
Background Detour: Splitting Numbers (2)

- One could rename "tens" to "index"
  - E.g., byte 34 is in sub-array index #3
- One could rename "ones" to "offset"
  - E.g., byte 34 is offset 4 in sub-array #3
- In this example, address "34" becomes a tuple (3,4)

- In base 10, this is an intuitive concept
- We will use this in base 2

Splitting Numbers in Base 2

- Same idea applies, just need to split on powers of two instead of ten
  - Say we go to sub-arrays of size 8:



Why do we care?

- We will see lots of variations on using modular arithmetic to calculate an index and offset in the next few lectures

- And why base 2?
  - How data is carried on wires in chip
  - Easier to implement modular arithmetic in base 2
  - Use cheap logical operators instead of expensive division
    - When dividing, if n is a power of two:

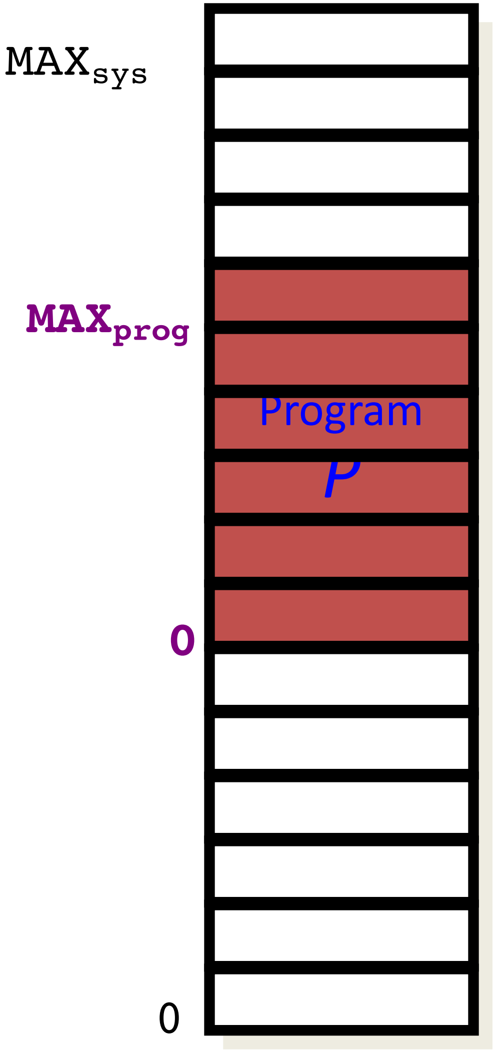
      $x / n == x >> \log_2 (n)$

      $x \% n == x \& (n-1)$

Review: Address Spaces

- *Physical address space* — The address space supported by the hardware
  - Starting at address 0, going to address $\mathbf{MAX_{sys}}$

- *Virtual address space* — A process's view of its own memory
  - Starting at address 0, going to address $\mathbf{MAX_{prog}}$



But where do addresses come from?

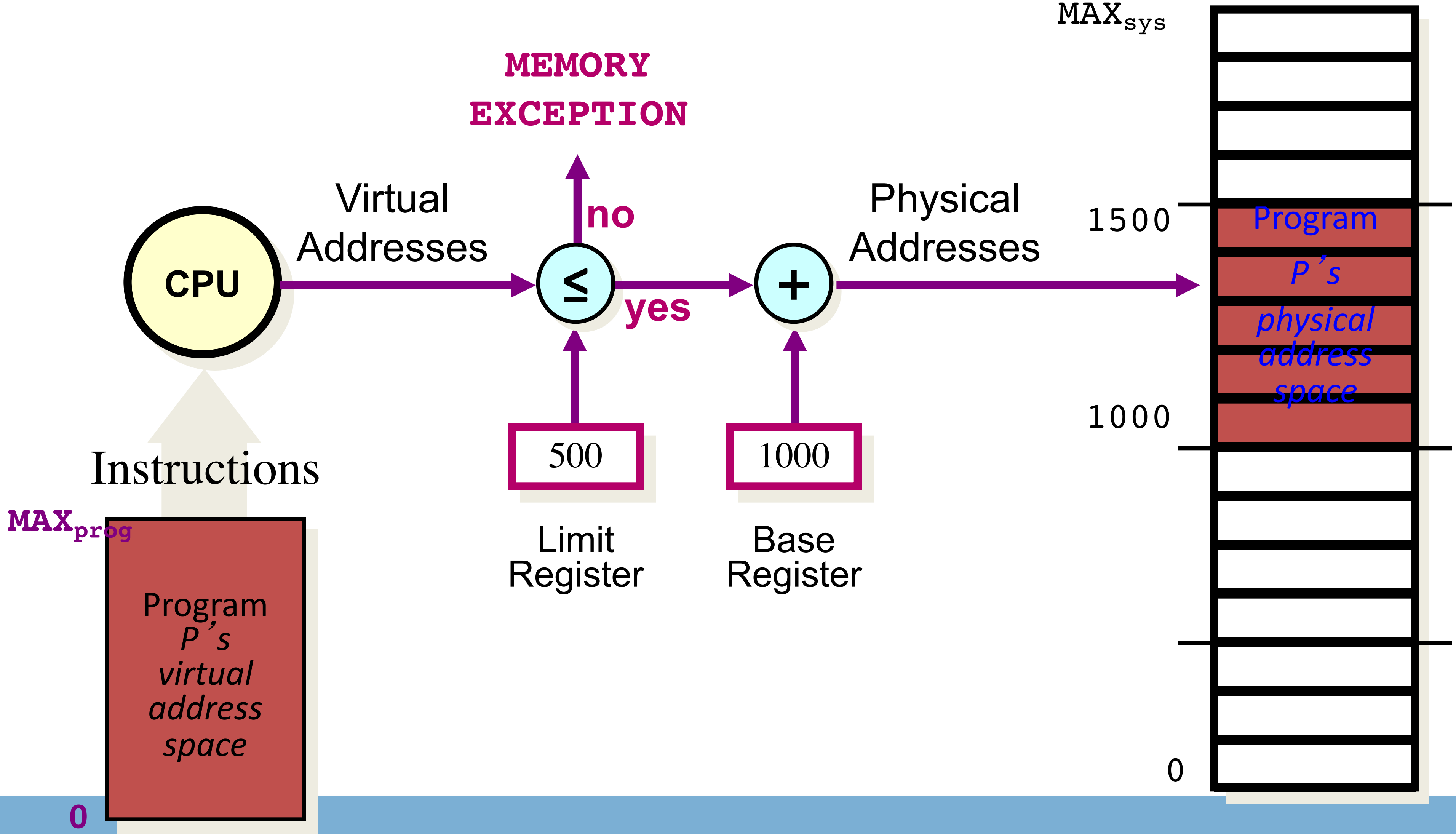
```
MOV r0, @0xfffa620e
```

- Which is bigger, physical or virtual address space?
  - A. Physical address space
  - B. Virtual address space
  - C. It depends on the system.

Program Relocation

- Program issues virtual addresses
- Machine has physical addresses.
- If virtual == physical, then how can we have multiple programs resident concurrently?
- Instead, relocate virtual addresses to physical at run time.
  - While we are relocating, also bounds check addresses for safety.
- I can relocate that program (safely) in two registers…

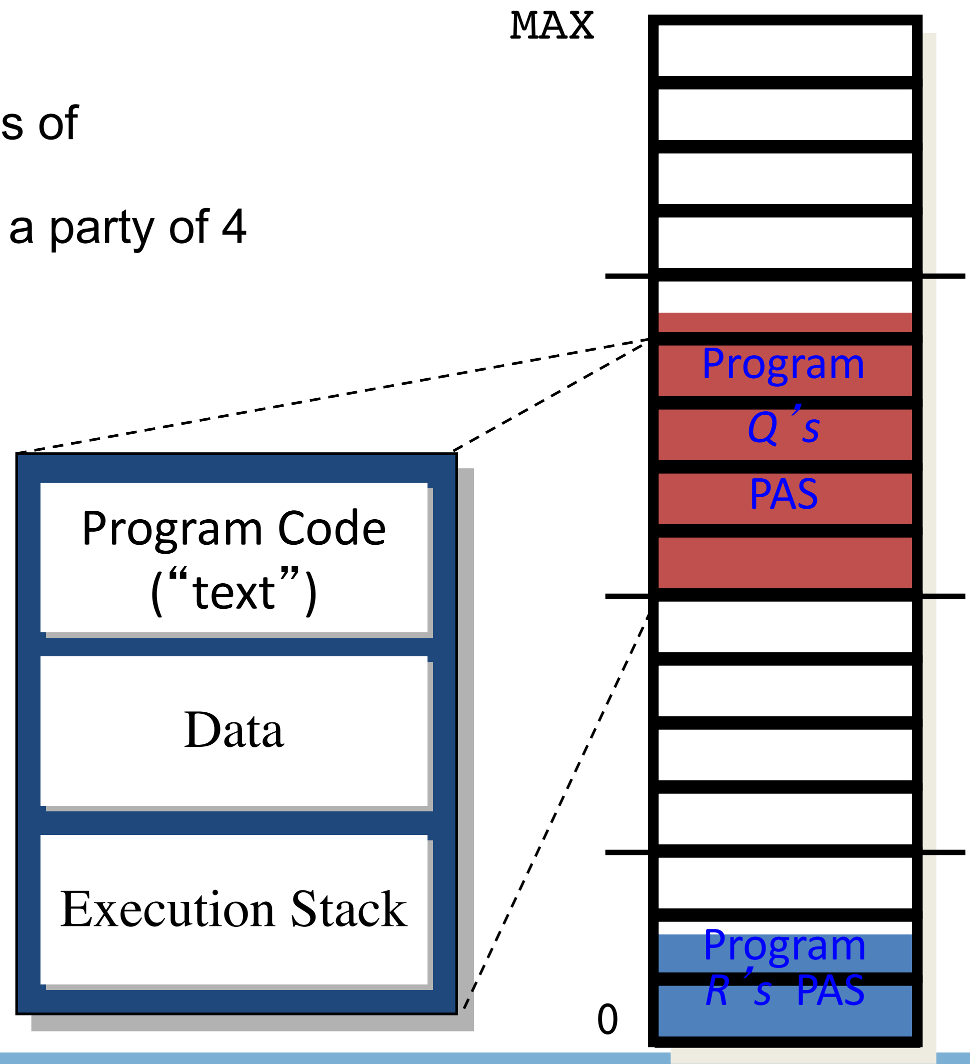
2 register translation



- With base and bounds registers, the OS needs a hole in physical memory at least as big as the process.
  - A. True
  - B. False

The Fragmentation Problem

- External fragmentation
  - Unused memory between units of allocation
  - E.g, two fixed tables for 2, but a party of 4
- Internal fragmentation
  - Unused memory within a unit of allocation
  - E.g., a party of 3 at a table for 4



Dynamic Allocation of Partitions

- Simple approach:
  - Allocate a partition when a process is admitted into the system
  - Allocate a contiguous memory partition to the process

OS keeps track of…

Full-blocks

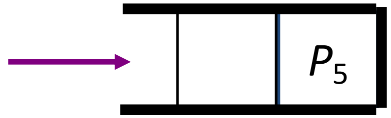
Empty-blocks (“holes”)

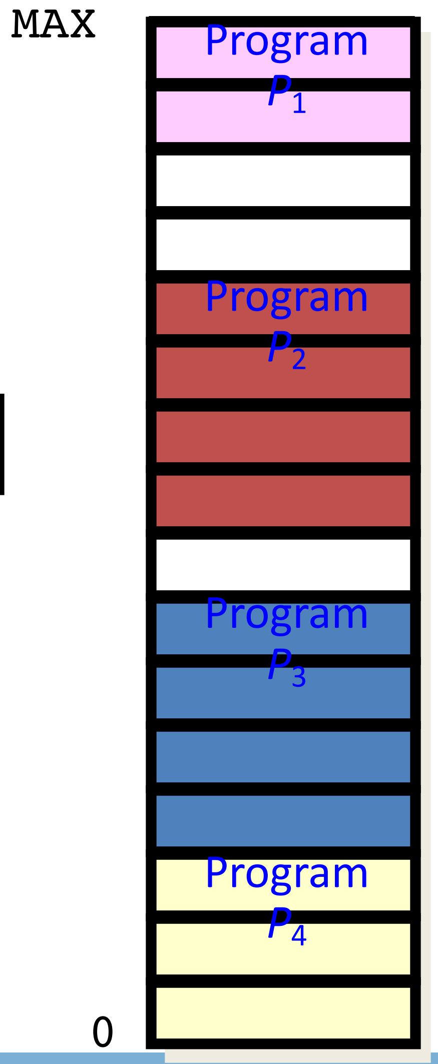
Allocation strategies

First-fit

Best-fit

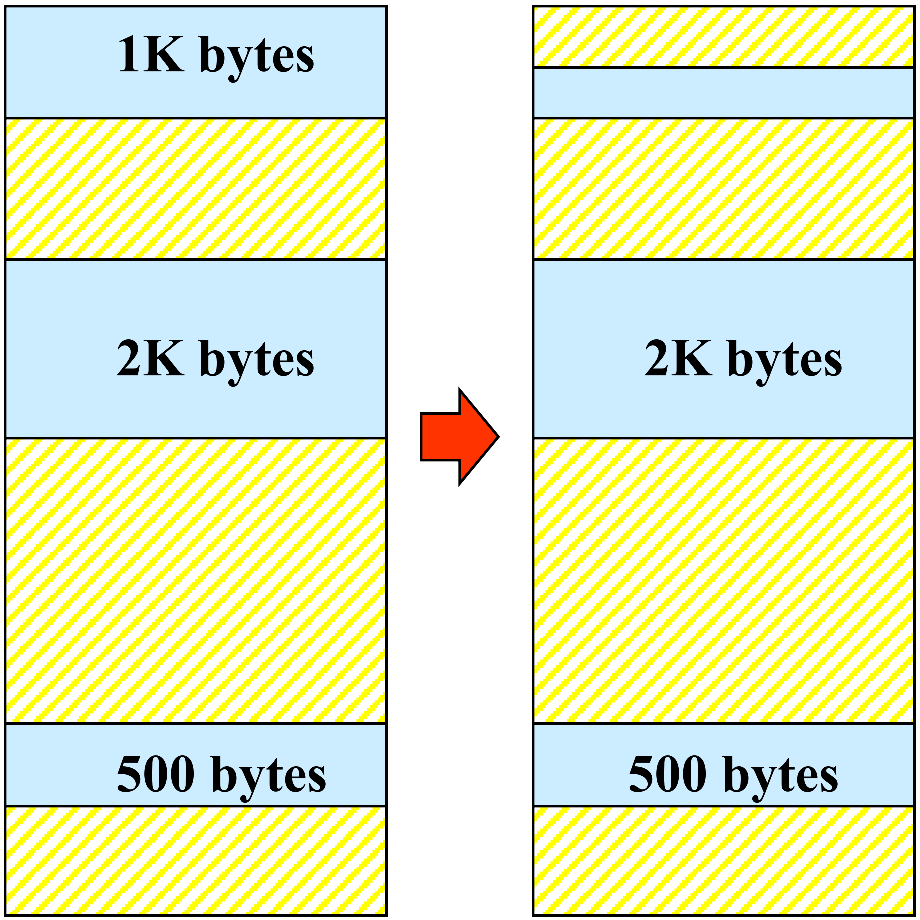
Worst-fit





First Fit Allocation

To allocate *n* bytes, use the *first* available free block such that the block size is larger than *n.*



To allocate 400 bytes, we use the 1st free block available

First Fit: Rationale and Implementation

- Simplicity!

- Requires:
  - Free block list sorted by address
  - Allocation requires a search for a suitable partition
  - De-allocation requires a check to see if the freed partition could be merged with adjacent free partitions (if any)

Advantages

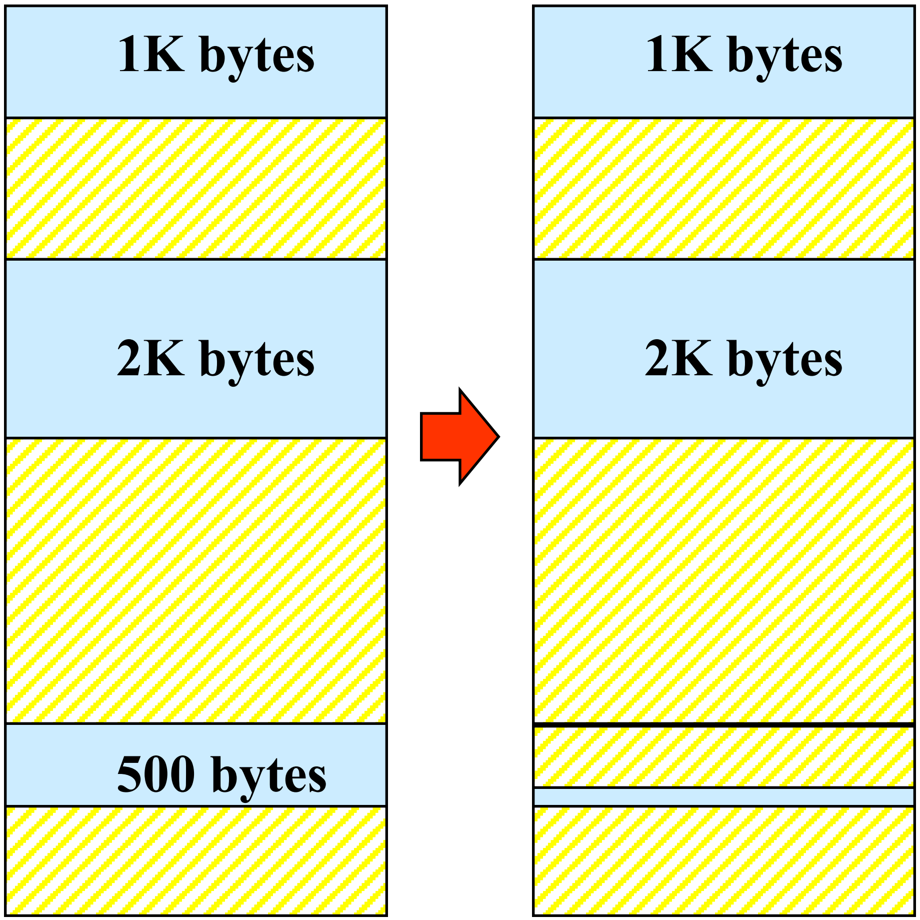
- Simple
- Tends to produce larger free blocks toward the end of the address space

Disadvantages

- Slow allocation
- External fragmentation

Best Fit Allocation

To allocate *n* bytes, use the *smallest* available free block such that the block size is larger than *(or equal to) n.*



To allocate 400 bytes, we use the 3rd free block available (smallest)

Best Fit: Rationale and Implementation

- Avoid fragmenting big free blocks
- To minimize the size of external fragments produced
- Requires:
  - Free block list sorted by size
  - Allocation requires search for a suitable partition
  - De-allocation requires search + merge with adjacent free partitions, if any

Advantages

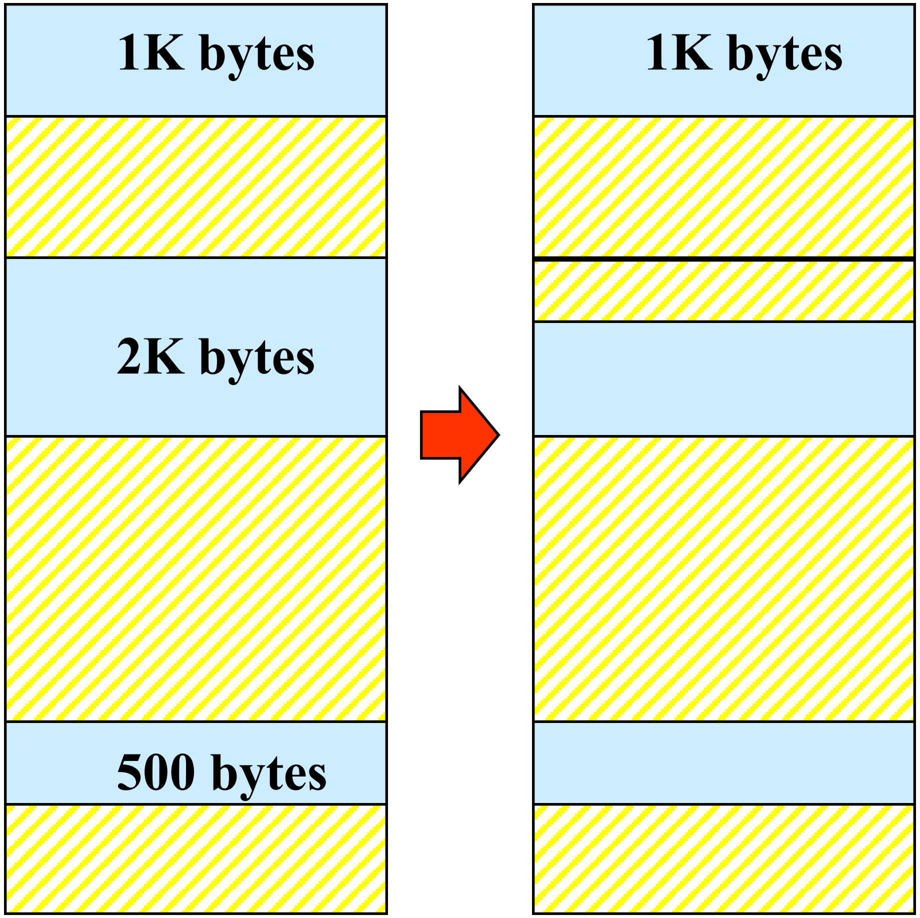
- Works well when most allocations are of small size
- Relatively simple

Disadvantages

- External fragmentation
- Slow de-allocation
- Tends to produce many useless tiny fragments (not really great)

Worst Fit Allocation

To allocate *n* bytes, use the *largest* available free block such that the block size is larger than *n.*



To allocate 400 bytes, we use the 2nd free block available (largest)

Worst Fit: Rationale and Implementation

- Avoid having too many tiny fragments

- Requires:
  - Free block list sorted by size
  - Allocation is fast (get the largest partition)
  - De-allocation requires merge with adjacent free partitions, if any, and then adjusting the free block list

Advantages

- Works best if allocations are of medium sizes

Disadvantages

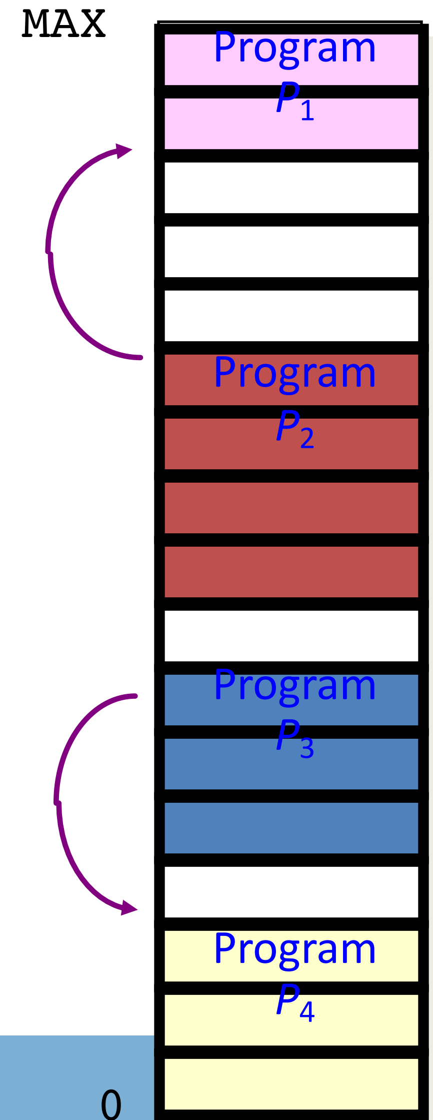
- Slow de-allocation
- External fragmentation
- Tends to break large free blocks such that large partitions cannot be allocated

Allocation strategies

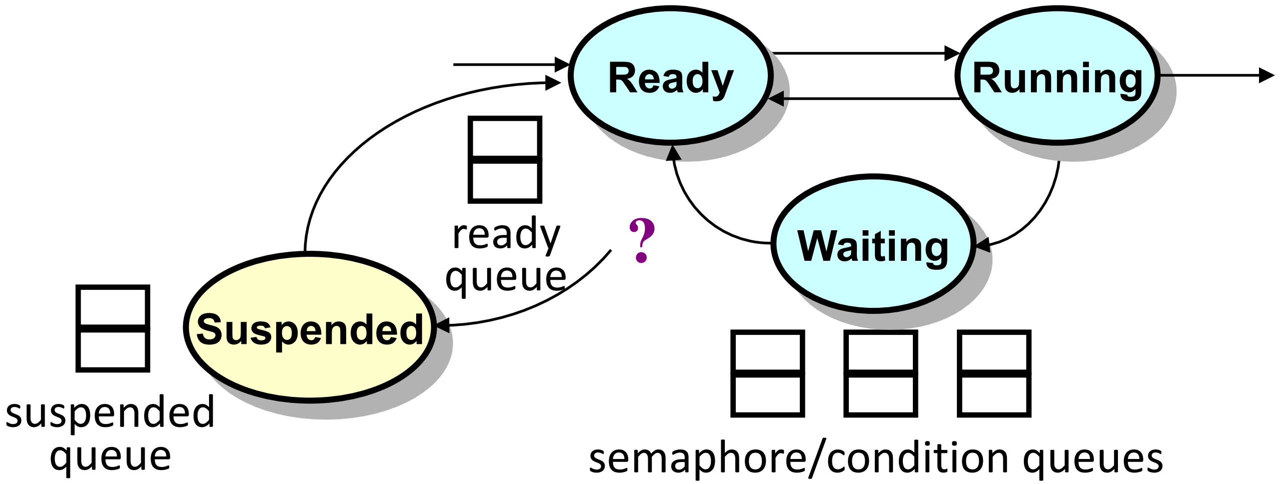
- First fit, best fit and worst fit all suffer from external fragmentation.
  - A. True
  - B. False

Eliminating Fragmentation

- Compaction
  - Relocate programs to coalesce holes



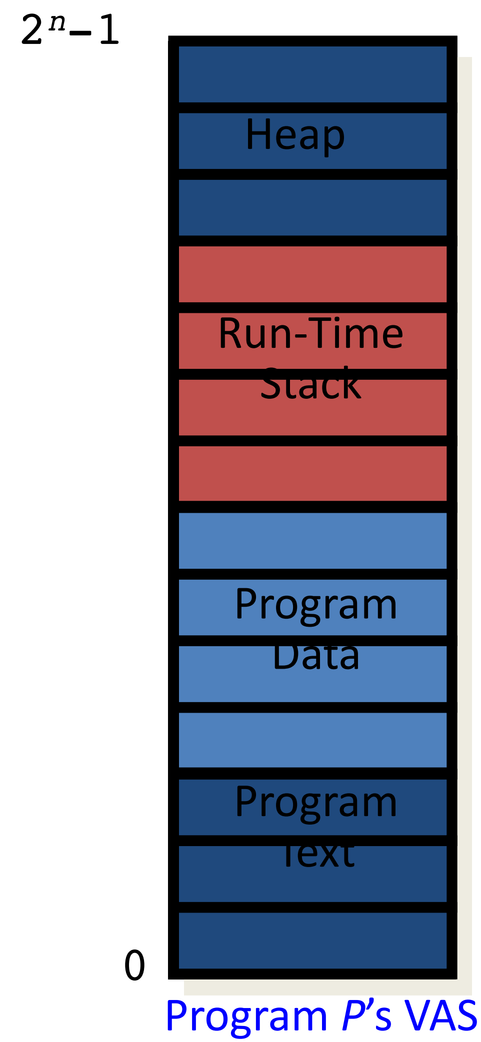
- Swapping
  - Preempt processes & reclaim their memory



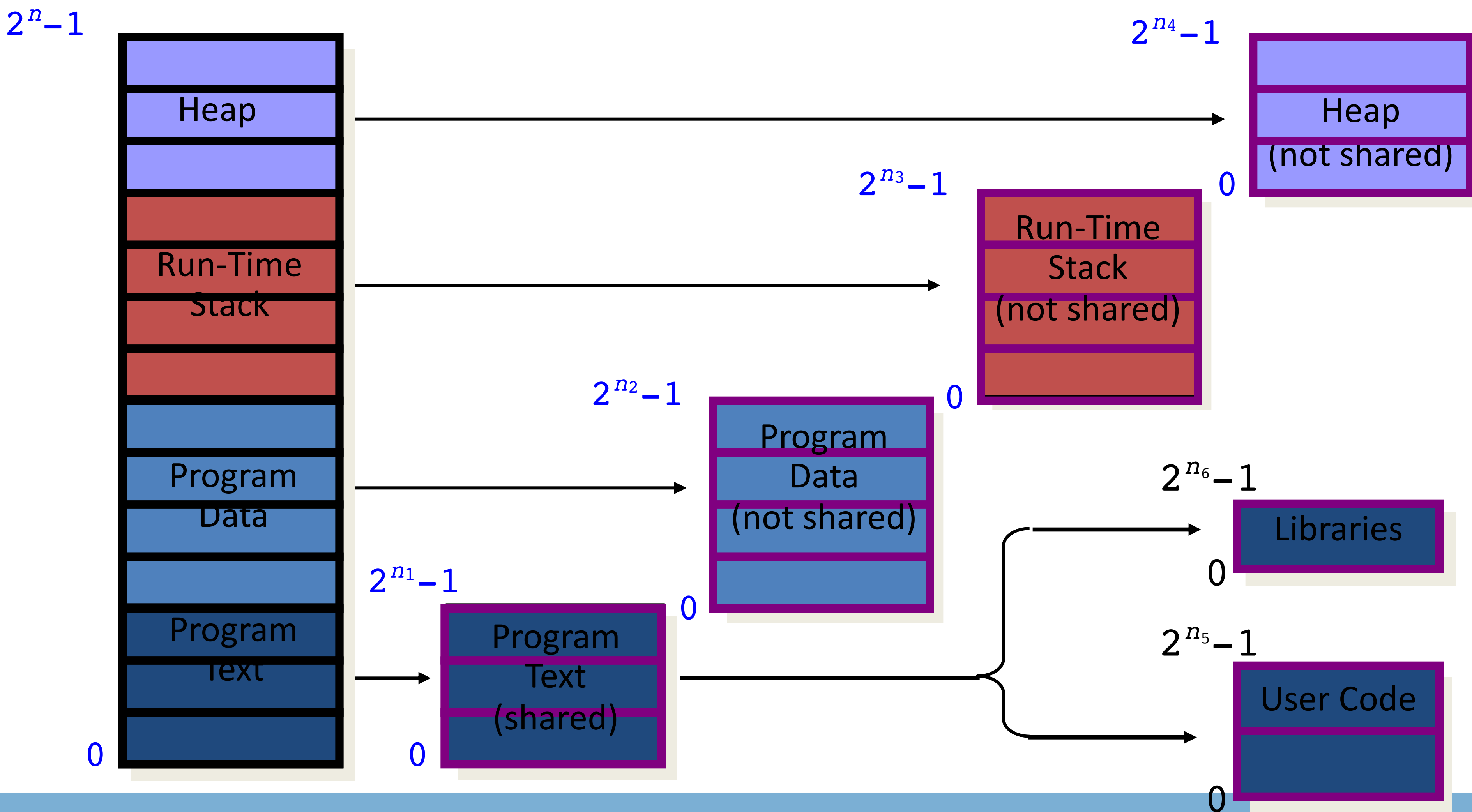
Sharing Between Processes

- Schemes so far have considered only a single address space per process
  - A single *name space* per process
  - No sharing

How can one share code and data between programs without paging?

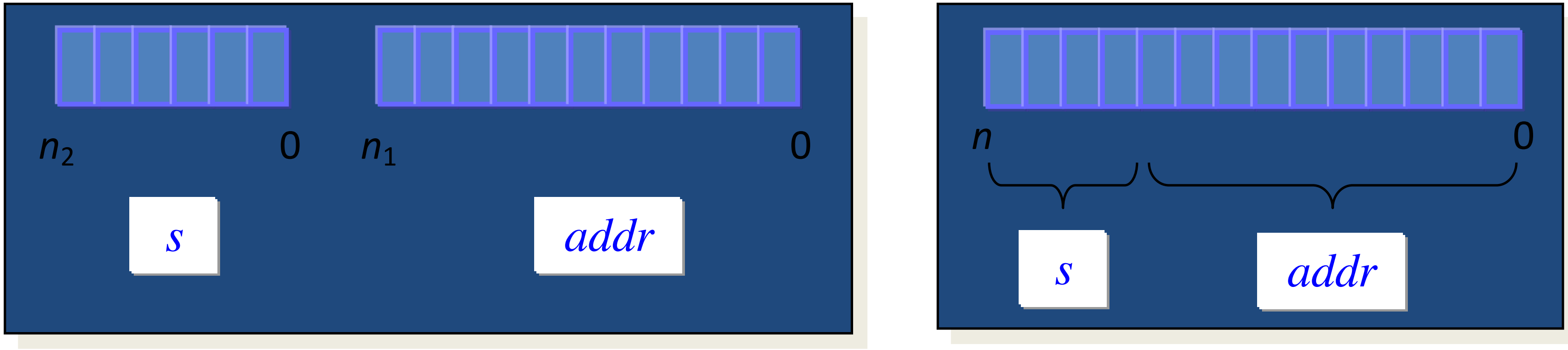


Multiple (sub) Name Spaces



Segmentation

- New concept: A *segment* — a memory "object"
  - A virtual address space
- A process now addresses objects —a pair (*s*, *addr*)
  - *s* — segment number
  - *addr* — an offset within an object
    - Don't know size of object, so 32 bits for offset?



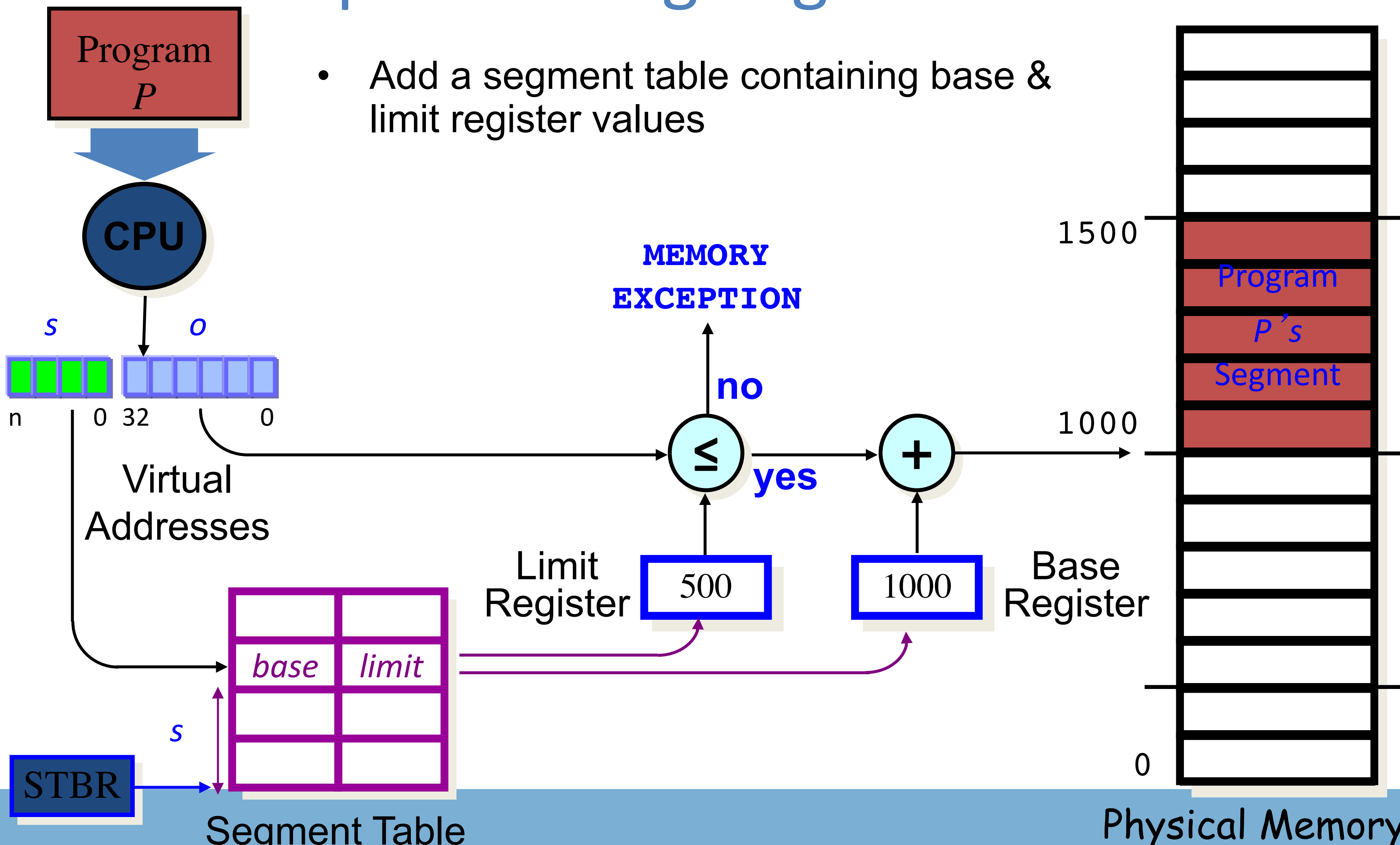
Segment + Address register scheme

Single address scheme

Two ways to encode a virtual address

Implementing Segmentation

- Add a segment table containing base & limit register values



Are we done?

- Segmentation allows sharing
  - And dead simple hardware
    - Can easily cache all translation metadata on-chip
  - Low latency to translate virtual addresses to physical addresses
    - Two arithmetic operations (add and limit check)

- … but leads to poor memory utilization
  - We might not use much of a large segment, but we must keep the whole thing in memory (bad memory utilization).
  - Suffers from external fragmentation
  - Allocation/deallocation of arbitrary size segments is complex

- How can we improve memory management?
  - stay tuned…

Trivia: Revisiting fork()

- I promised to explain the historical reason for fork()
- On the original machine Unix was designed for, there was only segmented memory protection, and very, very little DRAM.
- Easiest way to create a new process was to:
  - Write the relevant segments of the parent process to disk
    - Effectively, making a copy of the process memory on disk
  - Reload copied segments into memory to run child
- So they made a software abstraction that matched efficient use of early 1970s virtual memory hardware
  - And we still (inefficiently) emulate it on modern hardware