Physical Memory Management

Most Basic

Only 1 process, with only 1 kernel thread in memory at a time.

No memory sharing between user programs ⇒ no security

Next Simplest Way - Fixed Partitions

Memory is divided into a set of partitions of various sizes.

Load a program into a partition
- no sharing of partitions

\+ Multiprogramming is possible
\+ The partitions are the security

- Can only run a program up to the largest partition

- internal memory fragmentation
  - unused memory allocated to a process

Method #3 - Dynamic Partitions (Base & Bounds)

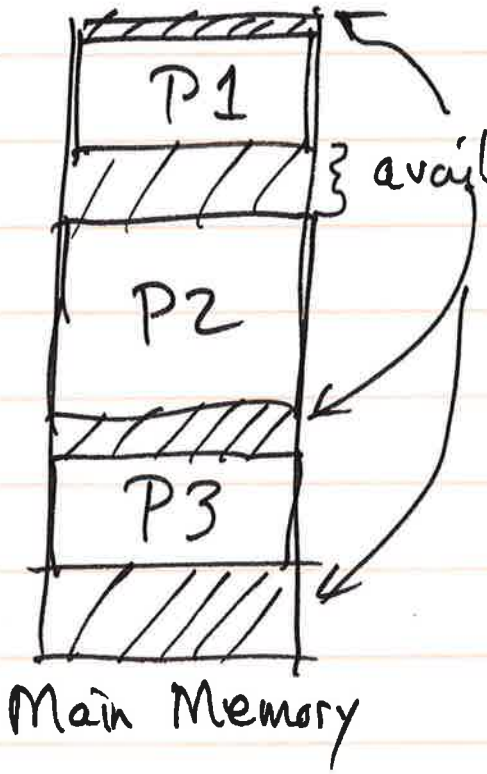
A partition is created at process initialization

+ A process can be loaded anywhere in memory
+ Can run one job almost the size of physical memory

- Need security between user programs
  - Need 2 new registers
    - Base: starting memory address
    - Bounds: partition size

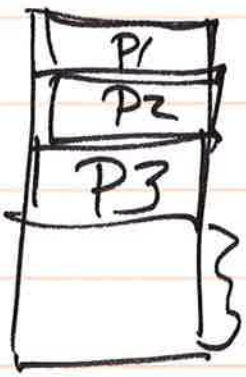
* One more disadvantage that all 3 methods, up to now: memory allocated to a process has been contiguous

Here's the problem w/ Dynamic Partitions



Problem: I need more memory for the new process than any single piece of available memory.

- I can wait until later
- Defragment memory



PURE OVERHEAD

Validating memory references & converting virtual addresses to real (physical) addresses is called Address Translation

⇓ done in hardware ⇓ mmu

For Base & Bounds.

① Validation

Compare needed virtual address to the Bounds reg.

V.A >= 0 && VA < Bounds Reg

② Getting Physical Address

P.A. = Base Reg + V.A.

Method #4 - Segmentation

Divide the address space into segments - these are smaller than whole address space

Each segment is contiguous in memory

Any segment can be loaded into an unused block of memory that is large enough

One set of Base & Bounds registers is not enough

- Too expensive to have lots of pairs of registers

We will store these B&B values in a table - Segment Tables

New + Memory sharing can now occur between processes
- entire segment

New + Virtual memory is now possible

→ allow user program to ignore physical memory limitations

(*) – Memory allocation is still a problem

* – External fragmentation exists

How to do Address Translation?

mmu Process

0, Receive V.A.

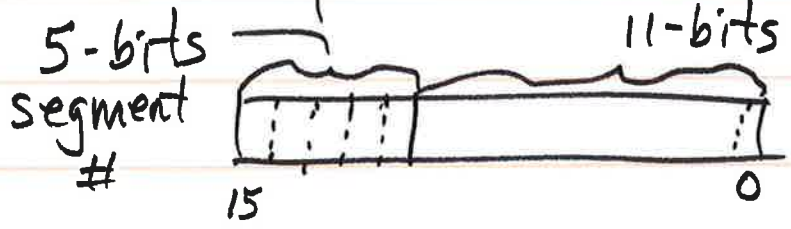
1. Validate V.A.
   - Split the V.A. into 2 parts
     - Segment number
     - Address offset in this segment

index position in Segment Table

an array

2. If valid, perform the translation

Example: 16-bit V.A.



VA: 10001 10000000111

17 1035

segment # V.A. offset in segment 17

Segment Table

	Valid	Base	Bounds
17	True	10,000	2000
20			

last segment

Validation Process

1. Does segment # exist in segment table – Yes – 21 entries
2. Is this segment already in memory? Yes – valid bit is true
3. Is address offset contained with segment 17?

offset : 1035 ⇒ Yes
size : 2000

4. Compute physical address

P.A = V.A. + Segment Start

= 1035 + 10,000

= 11,035

Method #5 - Paging

Allocate memory in fixed size blocks - all the same size ⇒ Pages

"Logically" divide the address space into Pages - the same size as physical memory pages
(virtual)

+ Any virtual page can be
− loaded into any available

Page of physical memory

O.S. tracks all virtual page locations, for each address space, in physical memory
- use a Page Table for this

Address Translation for Paging

- Similar like Segmentation
  - Split the V.A. into 2 parts
    - virtual page #
    - address offset within a page

Paging has a new problem:

- Page Table is an array
  - indexed by V.P. #
  - managed by kernel
  - no address translation

It must be physically contiguous in memory

Page tables can be very large & mostly empty

Ex: 32-bit OS

⇓

2^{32} bytes available to a user program

⇓

~4 billion bytes

⇓

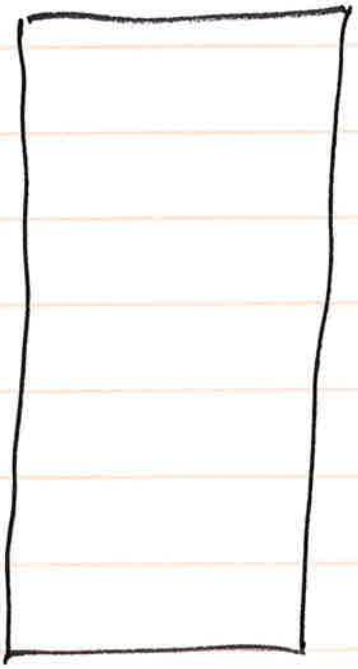
~1 billion addresses

Let's say a page is 10^4 addresses

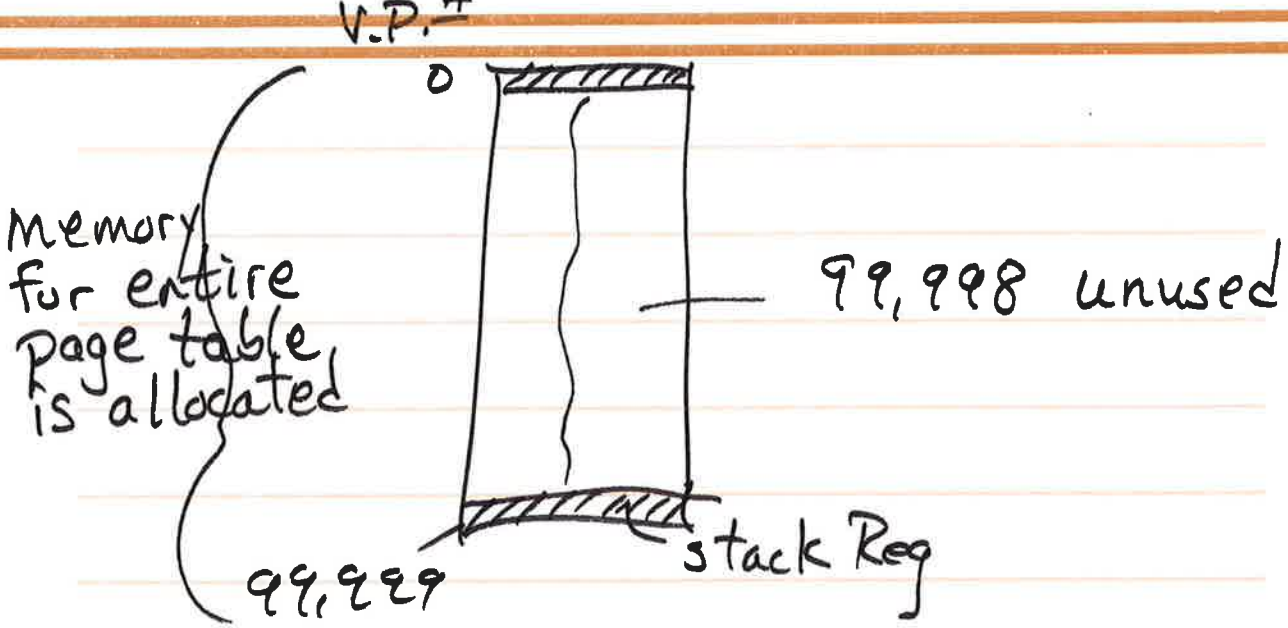
A page table could be 100,000 entries

User programs can specify how much heap & stack space they want.

```
while(true);
```



Page Table w/ 100,000 entries



Worst Case Scenario

In general, A.S.