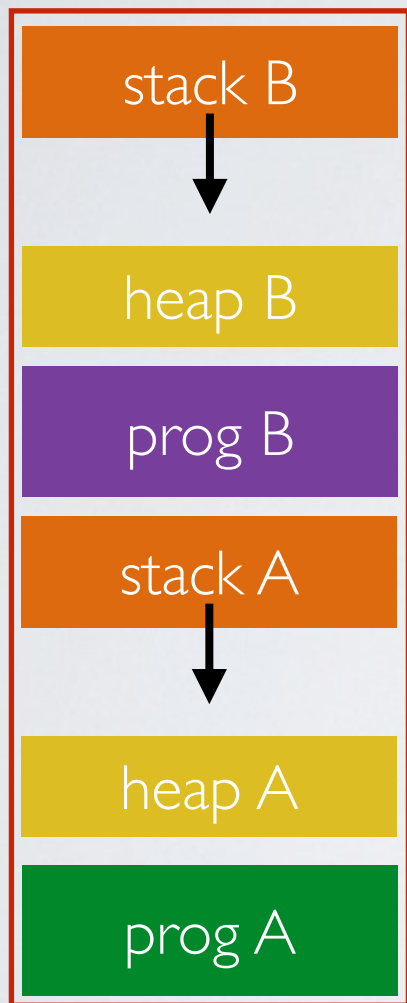


Virtual Memory

Thierry Sans

The problem of managing the memory



How to make programs and execution contexts co-exists in memory?

- ✓ Placing multiple execution contexts (stack and heap) at random locations in memory is not a problem ...
... well, as long as you have enough memory
- ⦿ However having programs placed at random locations is problematic

(recap) Compiling and linking

- **Compiler** takes source code files and translates (binds) symbolic addresses to logical, relocatable addresses within compilation unit (object file)
- **Linker** takes collection of object files and translates addresses to logical, absolute addresses within executable (resolves references to symbols defined in other files/modules)

Let's look at some C code and its binary

```
#include <stdio.h>

int foo(){
    printf("hello world!");
}

int main(int argc, char **argv){
    foo();
}
```

Since function addresses and others are hard-encoded in the binary, the program cannot be placed at random locations in memory

```
0804840b <foo>:
804840b: 55                push    ebp
804840c: 89 e5            mov     ebp,esp
804840e: 83 ec 08        sub     esp,0x8
8048411: 83 ec 0c        sub     esp,0xc
8048414: 68 d0 84 04 08   push    0x80484d0
8048419: e8 c2 fe ff ff   call    80482e0 <printf@plt>
804841e: 83 c4 10        add     esp,0x10
8048421: 90              nop
8048422: c9              leave
8048423: c3              ret

08048424 <main>:
8048424: 8d 4c 24 04      lea     ecx,[esp+0x4]
8048428: 83 e4 f0        and     esp,0xffffffff0
804842b: ff 71 fc        push    DWORD PTR [ecx-0x4]
804842e: 55              push    ebp
804842f: 89 e5            mov     ebp,esp
8048431: 51              push    ecx
8048432: 83 ec 04        sub     esp,0x4
8048435: e8 d1 ff ff ff   call    804840b <foo>
804843a: b8 00 00 00 00   mov     eax,0x0
804843f: 83 c4 04        add     esp,0x4
8048442: 59              pop     ecx
8048443: 5d              pop     ebp
8048444: 8d 61 fc        lea     esp,[ecx-0x4]
8048447: c3              ret
8048448: 66 90           xchg    ax,ax
804844a: 66 90           xchg    ax,ax
804844c: 66 90           xchg    ax,ax
804844e: 66 90           xchg    ax,ax
```


Naive Idea : load time linking

How about doing the linking when process executed, not at compile time

- ➡ Determine where process will reside in memory and adjust all references within program
- How to relocate the program in memory during execution? (consider functions but also data pointers now)
- What if no contiguous free region fits program?
- How to avoid programs interfering with each others?

Issues in sharing physical memory

Transparency

- A process shouldn't require particular physical memory bits
- A process often require large amounts of contiguous memory (for stack, large data structures, etc.)

Resource exhaustion

- Programmers typically assume machine has “enough” memory
- Sum of sizes of all processes often greater than physical memory

Protection

- How to prevent A from even observing B's memory
- How to prevent process A from corrupting B's memory (whether it is intentional or not)

Virtual Memory Goals

- Provide a convenient abstraction for programming by giving each program its own virtual address space
- Allow programs to see more memory than exists
- Allocate scarce memory resources among competing processes to maximize performance with minimal overhead
- Enforce protection by preventing one process from messing with another's memory

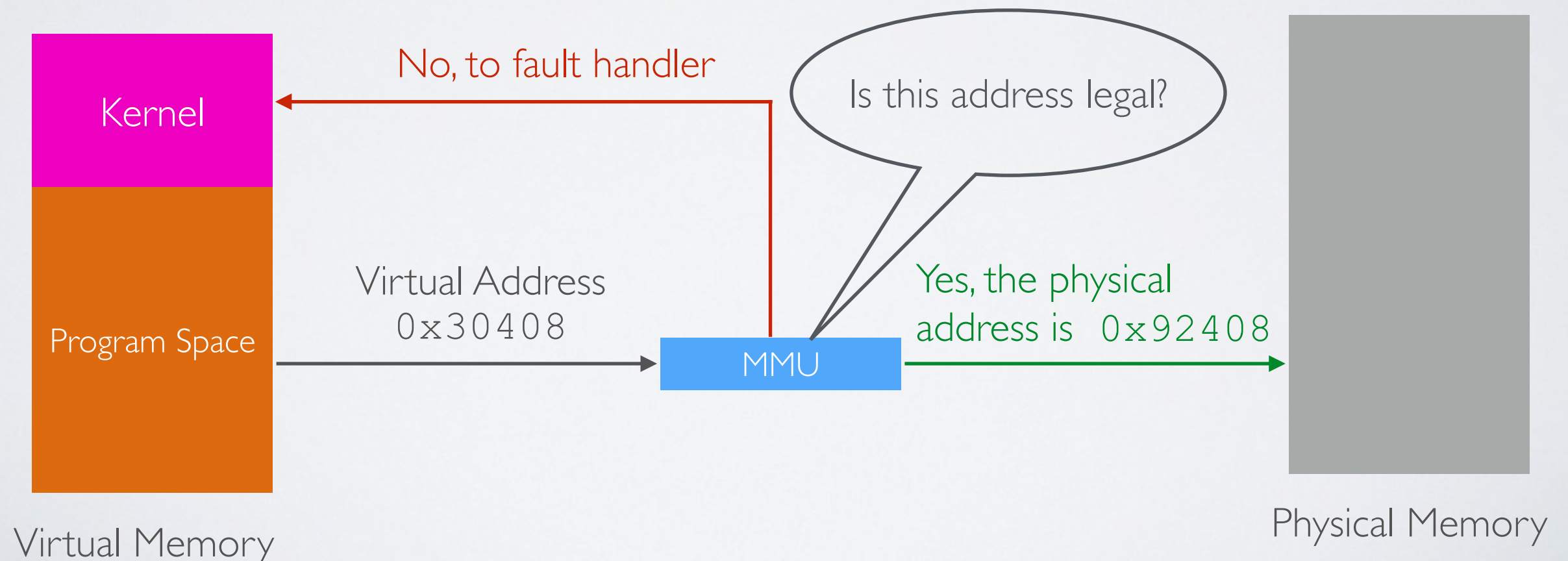
Definitions

- Programs load/store to **virtual addresses**
- Actual memory uses **physical addresses**
- Virtual memory hardware is the **MMU (Memory Management Unit)**
 - Usually part of CPU and configured through privileged instructions (e.g., load bound reg)
 - Translates from virtual to physical addresses
 - Gives per-process view of memory called **address space**

Virtual Memory in a nutshell

The application does not see physical memory addresses

➔ **Memory-Management Unit** (MMU) relocates each load/store at runtime



Virtual Memory Advantages

- ✓ Can re-locate process while running either in memory or to disk (a.k.a swap)

Techniques for implementing virtual memory

- Basic address translation
- Segmentation (the old way)
- **Paging (the new way)**

Basic Address Translation

Base & Bound registers

Two special privileged registers : base and bound
On each load/store/jump

- Physical address = virtual address + base
 - Check $0 \leq \text{virtual address} < \text{bound}$, else trap to kernel
- ✓ OS can change these registers to move the process in memory
- ✓ OS must re-load base these register on context switch

Base + Bound Trade-offs

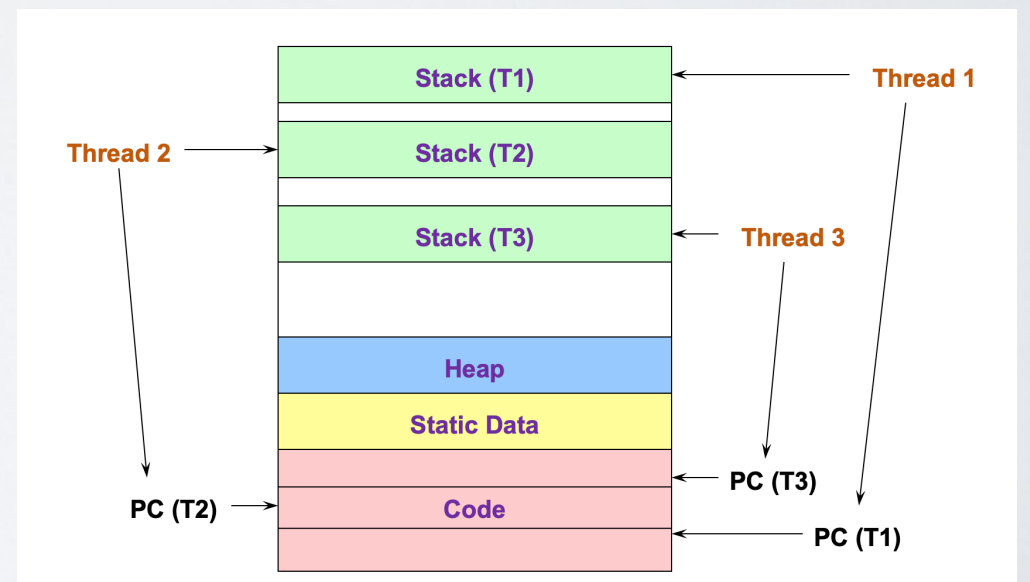
Advantages

- ✓ Cheap in terms of hardware : only two registers
- ✓ Cheap in terms of cycles : do add and compare in parallel

Disadvantages

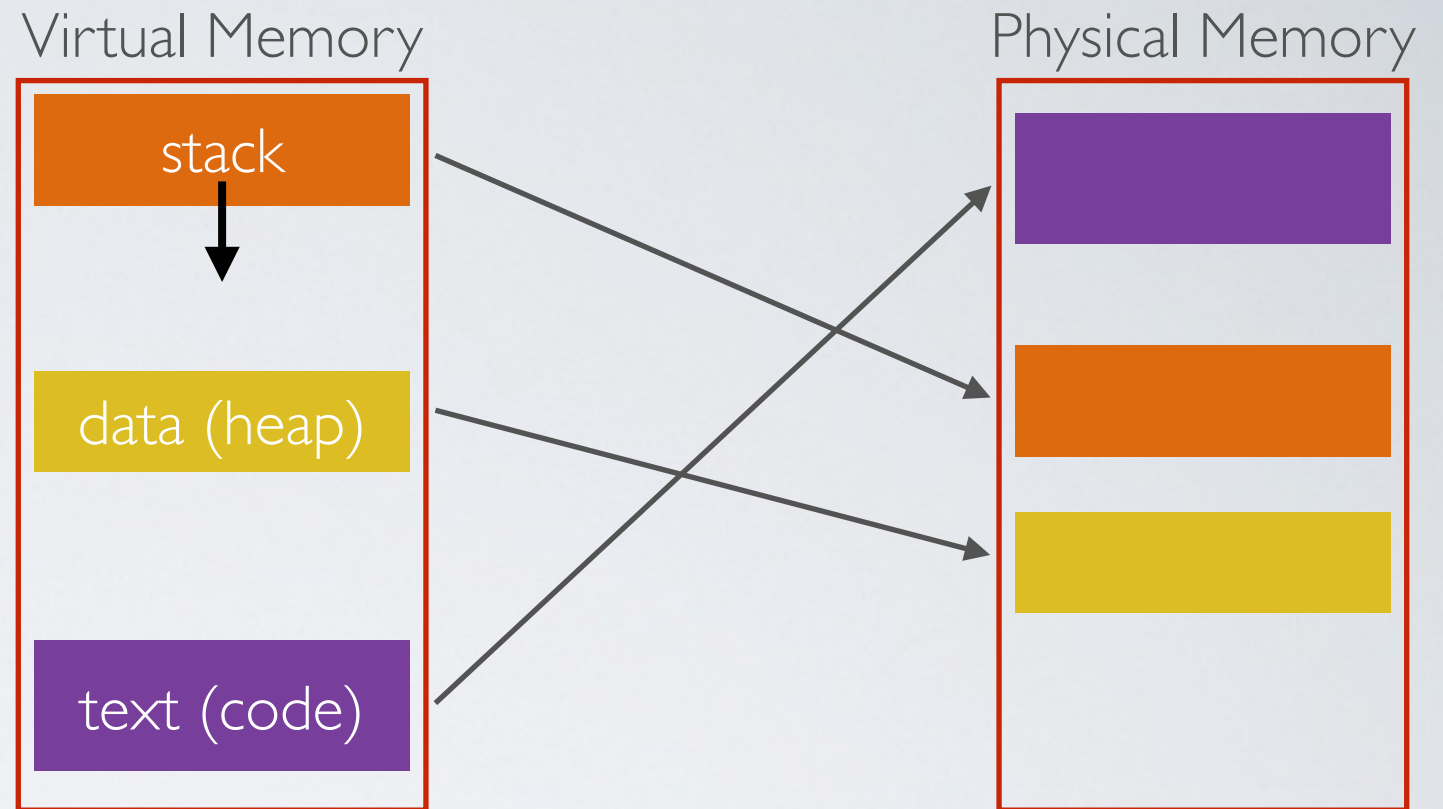
- Growing a process is expensive
- No way to share code or data

➔ **Solution :** segmentation i.e separate code, stack and data segments



Segmentation

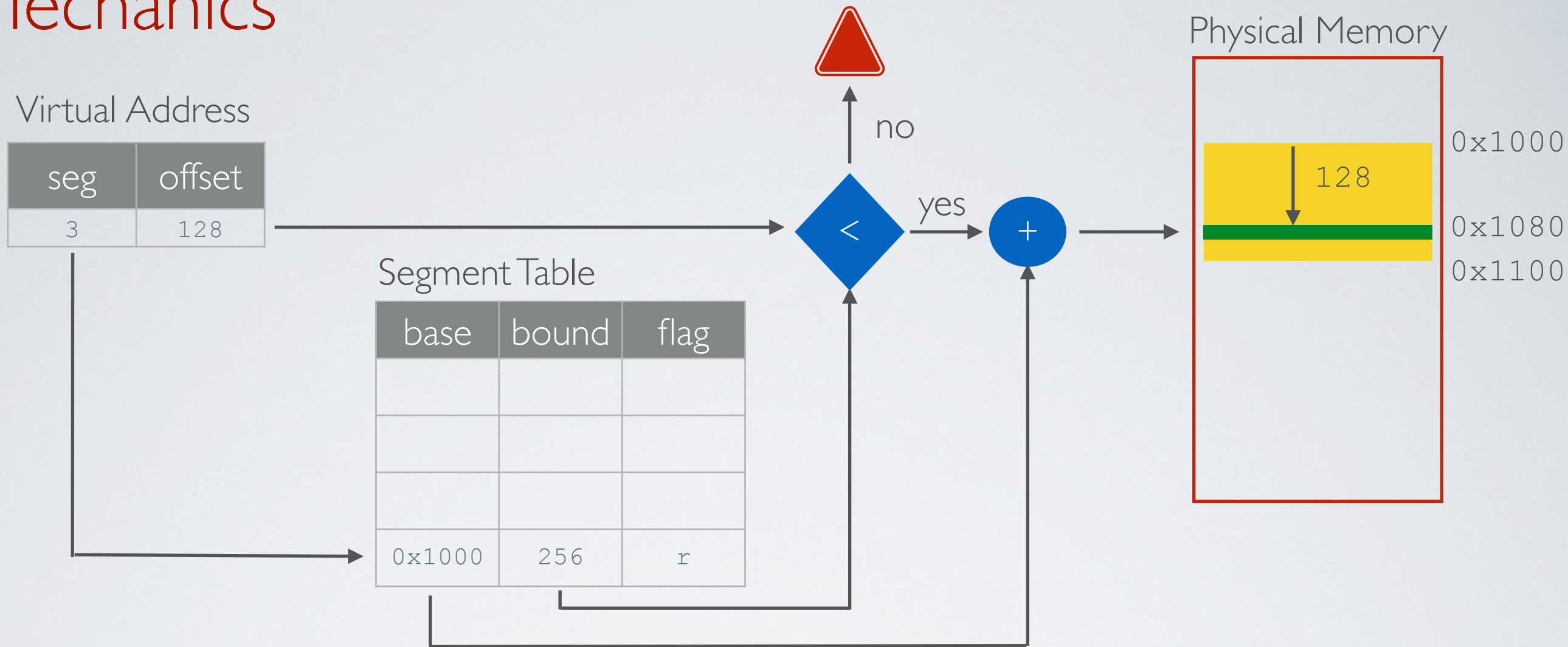
Idea



Each process has a collection of multiple base/bound registers

- ➔ Address space is built from many segments (a.k.a segmentation table)
- ✓ Can share/protect memory at segment granularity

Mechanics

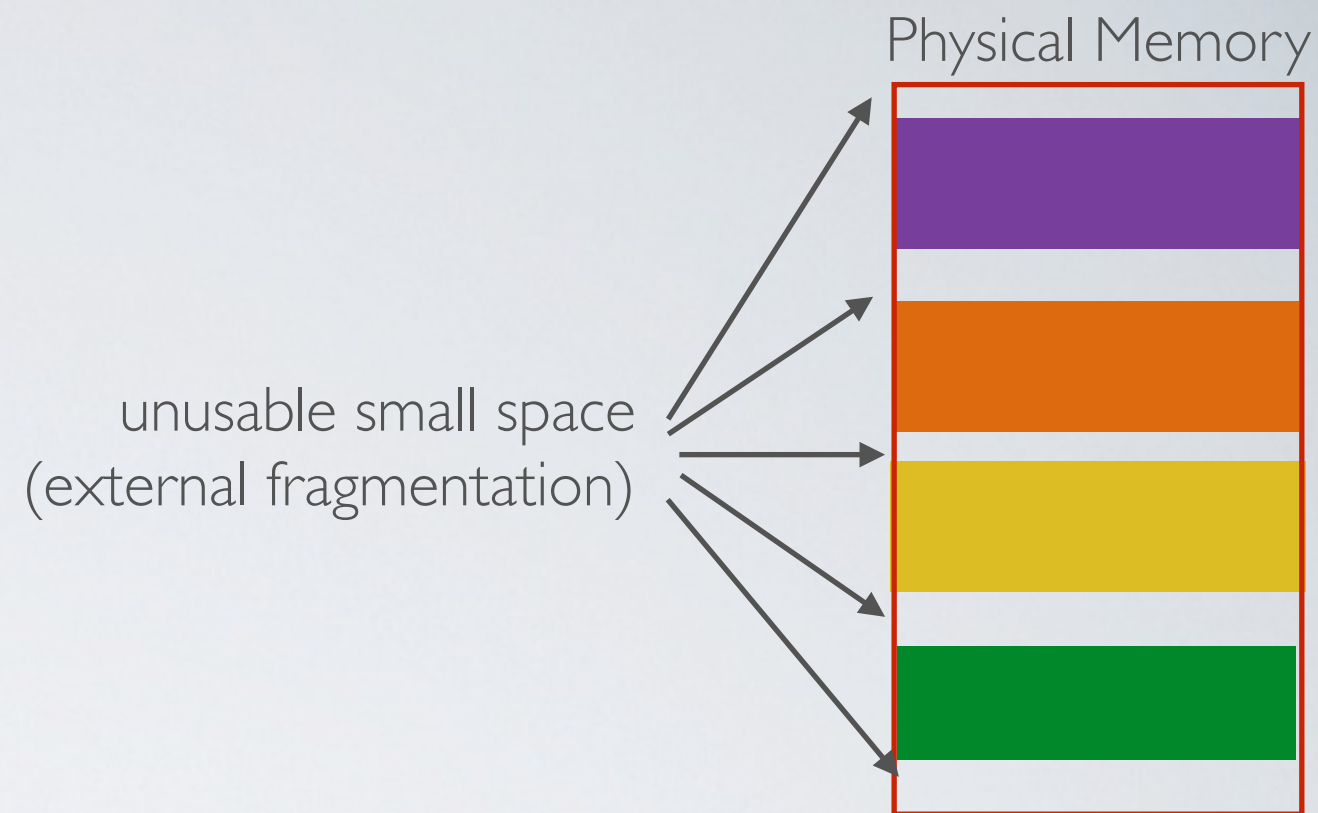


Each virtual address indicates

- a segment index in the table (top bits)
- and an offset (low bits)

➔ x86 stores segment #s in registers (CS, DS, SS, ES, FS, GS)

Segmentation Trade-offs



Advantages

- ✓ Multiple segments per process (sparse memory)
- ✓ Can easily share memory
- ✓ Do not need entire process in memory (swap)

Disadvantages

- ⦿ Requires translation, which could limit performance
- ⦿ Makes external **fragmentation** a real problem

Fragmentation

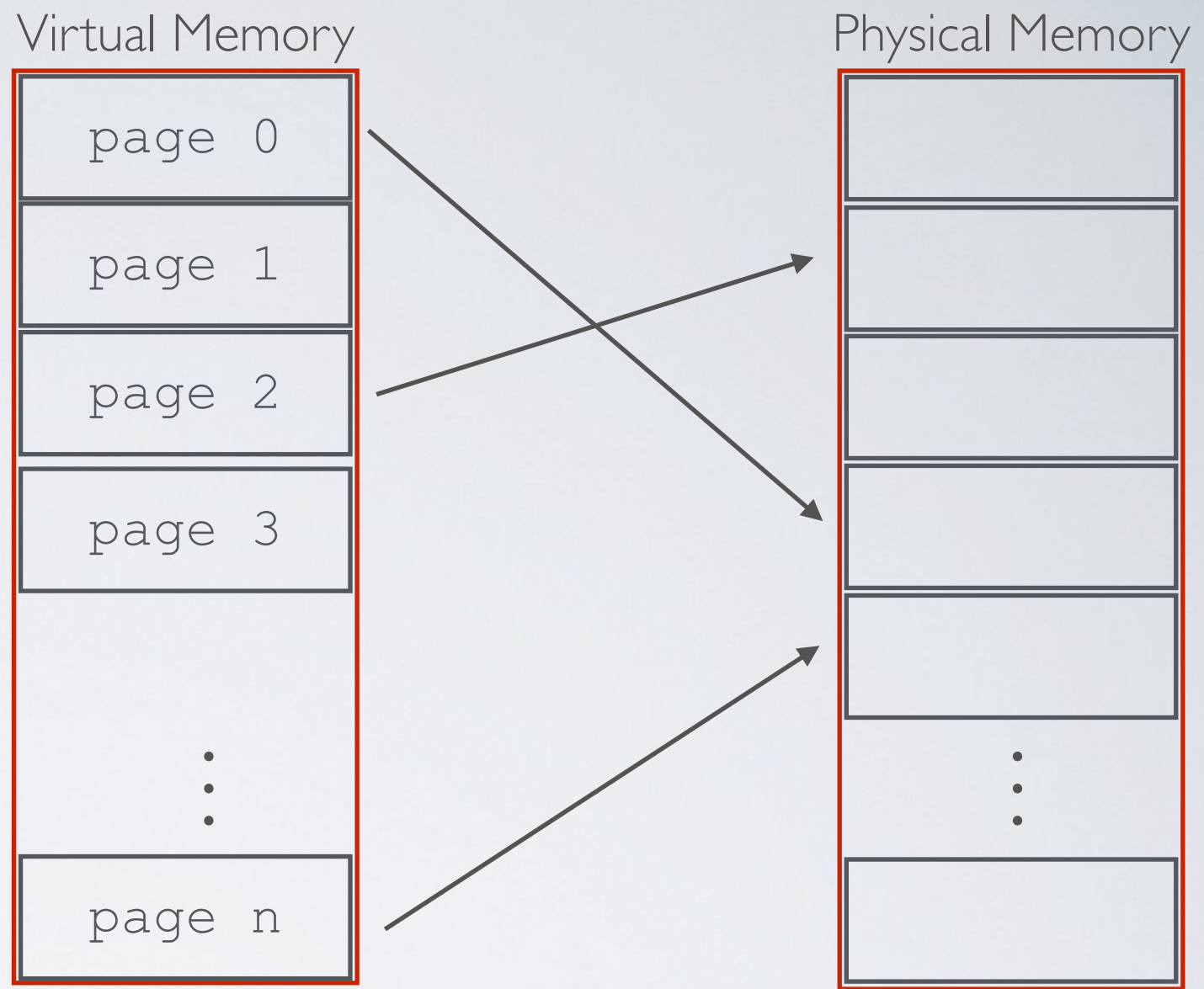
Fragmentation is the inability to use free memory

➡ Over time

- **External fragmentation**
because of variables sized pieces (i.e many small holes)
- **Internal fragmentation**
because of fixed size pieces (i.e no external hole but internal waste of space)

Paging (Introduction)

Idea

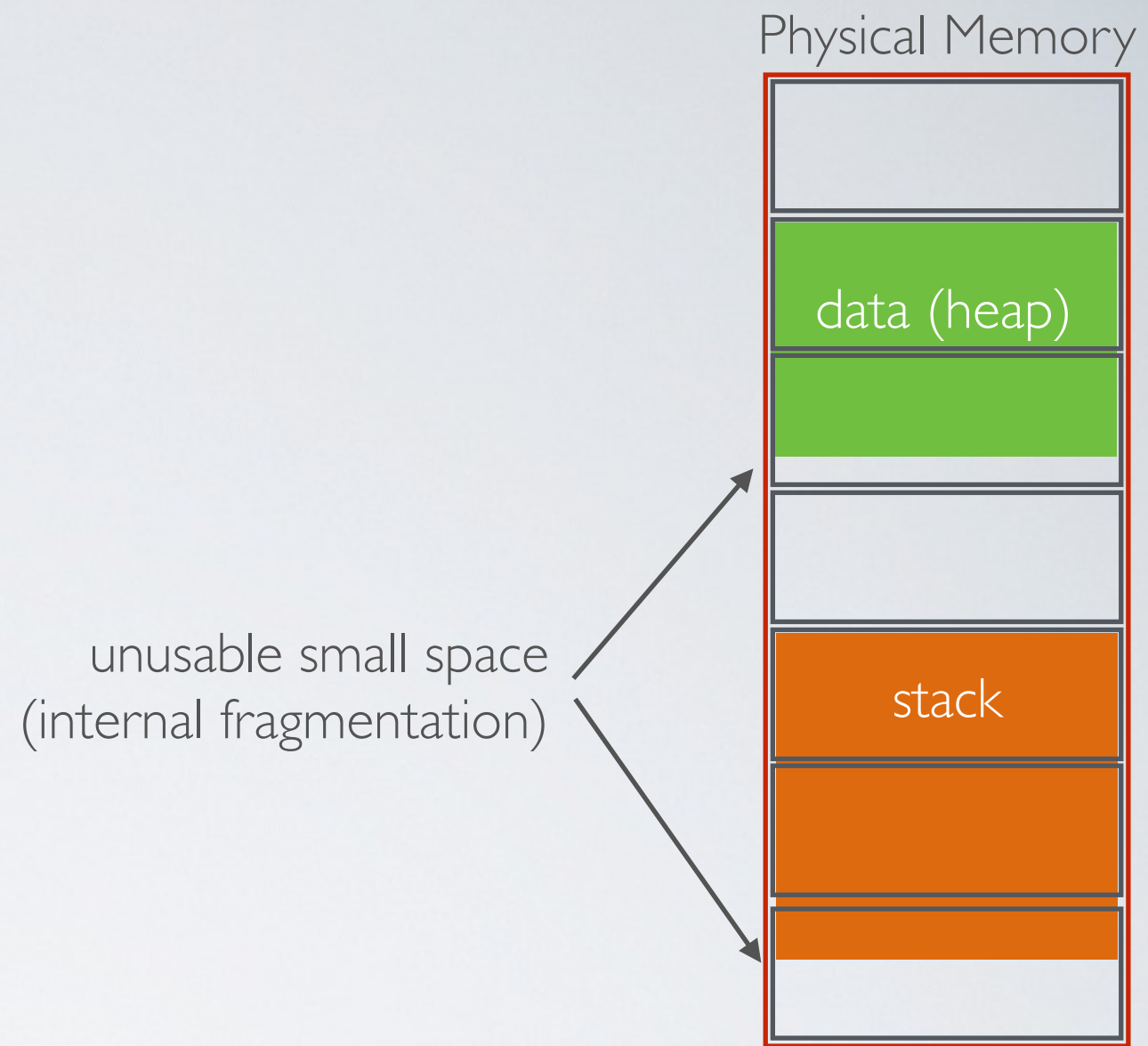


- ➡ Divide memory up into fixed-size pages to eliminate external fragmentation

Each process has a collection of maps from virtual pages to physical pages

- ✓ Can share/protect memory at page granularity

Paging Trade-offs



- ✓ Eliminates external fragmentation
- ✓ Simplifies allocation, free, and backing storage (swap)
- Average internal fragmentation of .5 pages per "segment"

Paging Data Structures

Pages are fixed size (e.g. 4K) so a virtual address has two parts:

- **virtual page number** : most significant bits
- and the **page offset** : least significant 12 bits ($\log_2 4k$)

The page table is a collection of **page table entry (PTE)** that maps

- a **virtual page number (VPN)**
i.e the index in the page table
- to **physical page numbers (PPN)** a.k.a frame number
- and includes bits for protection, validity, etc ...

Page Table Entries (PTEs)

- The **Modify bit** says whether or not the page has been written (set when the write to a page occurs)
- The **Reference bit** says whether the page has been accessed (set when a read or write to a page occurs)
- The **Valid bit** says whether or not the PTE can be used (checked each time the virtual address is used)
- The **Protection bits** say what operations (read, write, execute) are allowed on page
- The **Physical page number (PPN)** determines the physical page

Page Lookup

Virtual Address

page	offset
3	128

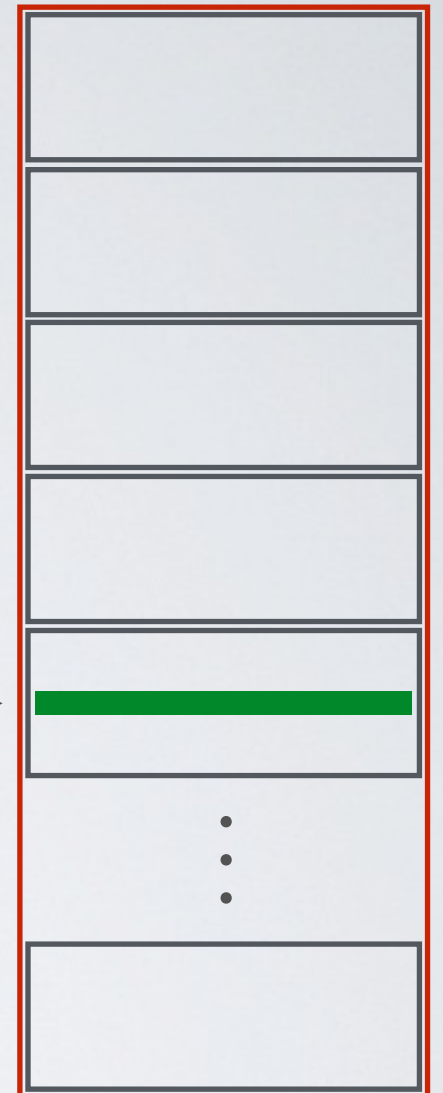
Page Table

m	r	v	p	ppe
1	1	1	rw	5

Physical Address

page	offset
5	128

Physical Memory



Paging Advantages

- ✓ Easy to allocate memory
 - Memory comes from a free list of fixed size chunks
 - Allocating a page is just removing it from the list
 - External fragmentation not a problem
- ✓ Easy to swap out chunks of a program
 - All chunks are the same size
 - Use valid bit to detect references to swapped pages
 - Pages are a convenient multiple of the disk block size

Paging Limitations

- Can still have internal fragmentation
 - Requires 2 or more references, which could limit performance
- ➔ **Solution:** use a hardware cache of lookups (coming next)
- The amount of memory to store the page table is significant
 - Need one PTE per page, with 32 bit address space w/ 4KB pages = 2^{20} PTEs
 - 4 bytes/PTE = 4MB/page table
 - 25 processes = 100MB just for page tables!
- ➔ **Solution :** page the page tables (coming next)

x86 Paging and Segmentation

x86 architecture supports both paging and segmentation

- Segment register base + pointer val = linear address
- Page translation happens on linear addresses
- Two levels of protection and translation check
 - Segmentation model has four privilege levels (CPL 0–3)
 - Paging only two, so 0–2 = kernel, 3 = user

Acknowledgments

Some of the course materials and projects are from

- Ryan Huang - teaching CS 318 at *John Hopkins University*
- David Mazière - teaching CS 140 at *Stanford*