MEMORY MANAGEMENT: PAGING

Kai Mast

CS 537

Fall 2022

ANNOUNCEMENTS

- Project 2a is out now
  - Due on 10/10
  - Discussion tomorrow will cover string handling and other shell-related things

- Grades for 1a and 1b should be out soon…

RECAP: VIRTUAL MEMORY

An abstraction that enables multiple processes to use the physical memory at the same time

Goals

- Transparency: Process is unaware of resources being shared
- Protection: Cannot corrupt or read data of OS or another process
- Efficiency: Do not waste resources or slow down processes

Applies to other abstractions for multiprocessing (e.g., file systems) as well!

SEGMENTATION

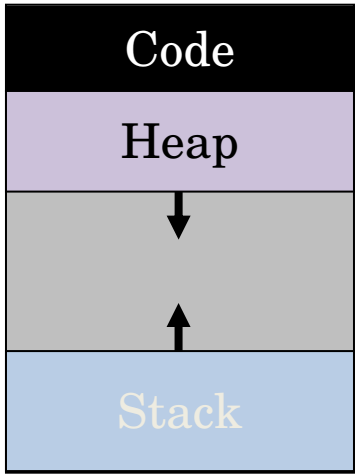
Divide logical address space into logical segments – visible to the OS

- Each segment corresponds to logical entity in address space (code, stack, heap)

Each segment has separate base + bounds register in the MMU

Advantages: Each segment can independently

- be placed separately in physical memory
- grow and shrink
- be protected (separate read/write/execute bits)



(Book Chapter 16)

SEGMENTED ADDRESSING

For virtual address, process specifies 1) segment and
2) offset within segment

How does process designate a particular segment?

- Use part of logical address
  - Top bits of logical address select segment
  - Low bits of logical address select offset within segment

SEGMENTED ADDRESSING

Example: 14 bit logical address, 4 segments

- We need two bits to indicate the segment

Consider Address 0x1108

Hex	**1**				**1**				**0**				**8**			
Bits	-	-	0	1	0	0	0	1	0	0	0	0	1	0	0	0

Unused

Top Bits (Segment)

Lower Bits (Offset)

SEGMENTED ADDRESSING (CONT.)

Consider Address 0x1108
14 bit logical address, top 2 bits for the segment number

```
// If the MMU was implemented in software
// (which it usually is not)
void* translate(void *ptr) {
    int addr = (int)ptr; // 0x1108
    int segment_no = (addr & (0xF000)) >> 12; // 0x1
    int offset = addr & 0x0FFF; // 0x108

    return (void*)segments[segment_no].base + offset;
}
```

ADDRESS TRANSLATIONS WITH SEGMENTATION

Segment Table in the MMU
14 bit logical address, 4 segments;

Segment	Base	Bounds	R W
0	0x2000	0x6ff	1 0
1	0x0000	0x4ff	1 1
2	0x3000	0xfff	1 1
3	0x0000	0x000	0 0

Remember:
1 hex digit 4 bits

Translate logical (in hex) to physical

0x0240:
Segment: 0
0x2000 + 0x240 = 0x2240

0x1108:
Segment: 1
0x0000 + 0x108 = 0x0x108

0x265c:
Segment: 2
0x3000 + 0x65c = 0x365c

0x3002:
Segment: 3
Offset > Bounds (and no R/W)

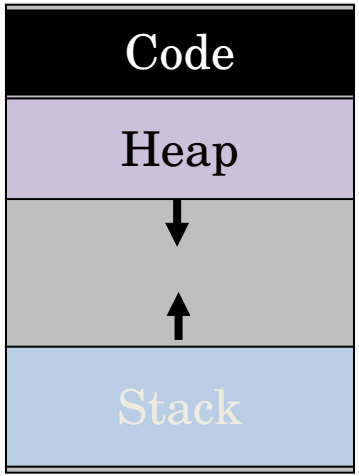
ADVANTAGES OF SEGMENTATION

Enables sparse allocation of address space

Stack and heap can grow independently

- Heap: If no data on free list, dynamic memory allocator requests more from OS

  (e.g., UNIX: malloc library makes sbrk() system call)
- Stack: OS recognizes reference near outside legal segment, extends stack implicitly



Different protection for different segments

- Enables sharing of selected segments
- Read-only status for code

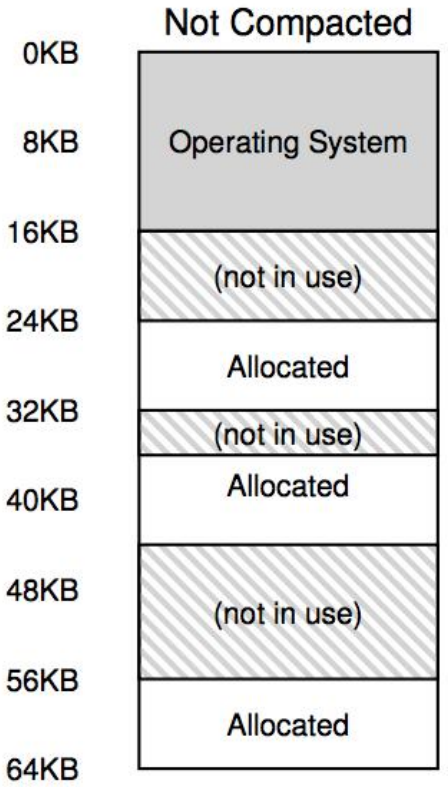
Supports dynamic relocation of each segment

DISADVANTAGES OF SEGMENTATION

Each segment must be allocated contiguously

May not have sufficient physical memory for large segments?

External Fragmentation
(memory fragmentation caused by the OS)



MEMORY MANAGEMENT WITH PAGING

(Book Chapter 18)

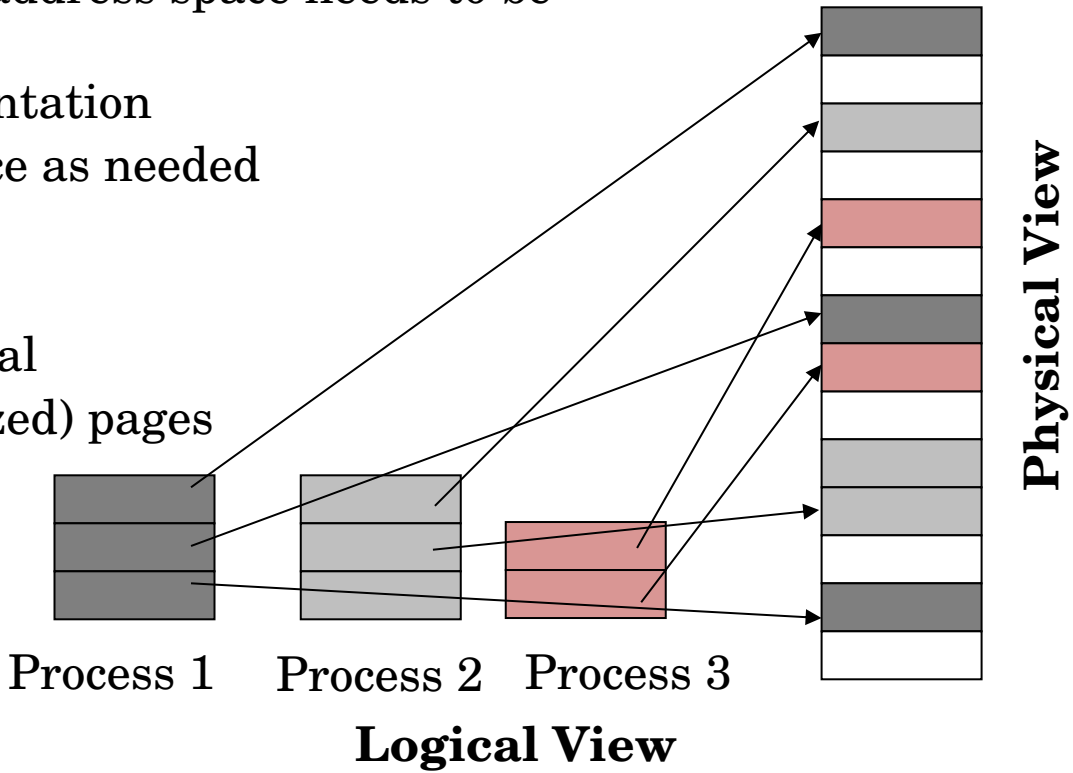
PAGING

Goal: Remove requirement that address space needs to be contiguous

- Eliminates external fragmentation
- Ability to grow address space as needed

Idea:

Divide address spaces and physical memory into fixed-sized (same sized) pages



PAGE TABLES

Maintain a mapping from virtual pages to physical frames.

How many pages tables are needed?*

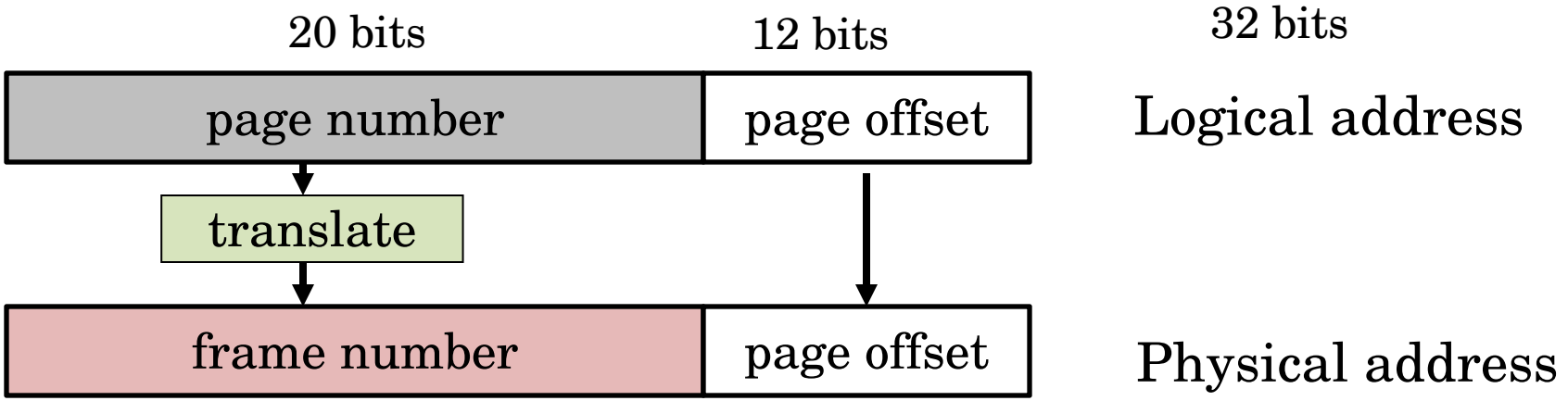
- One per process
  - Each process has its own virtual memory!
- One for the kernel's virtual memory (if the kernel uses VM)
- Additionally, we need a free list
  - Keeps track of which frames are free or currently mapped to a process

*Assume we do not use a reverse page table to do address translation. We will talk about that another lecture.

TRANSLATION OF PAGE ADDRESSES

How to translate logical address to physical address?

- High-order bits of address designate page number
- Low-order bits of address designate offset within page



No addition needed (unlike segmentation); just append bits correctly…

ADDRESS FORMAT

Given known page size, how many bits are needed in address to specify offset in page?

Given number of bits in virtual address and bits for offset, how many bits for virtual page number?

Given number of bits for VPN, how many virtual pages can there be in an address space?

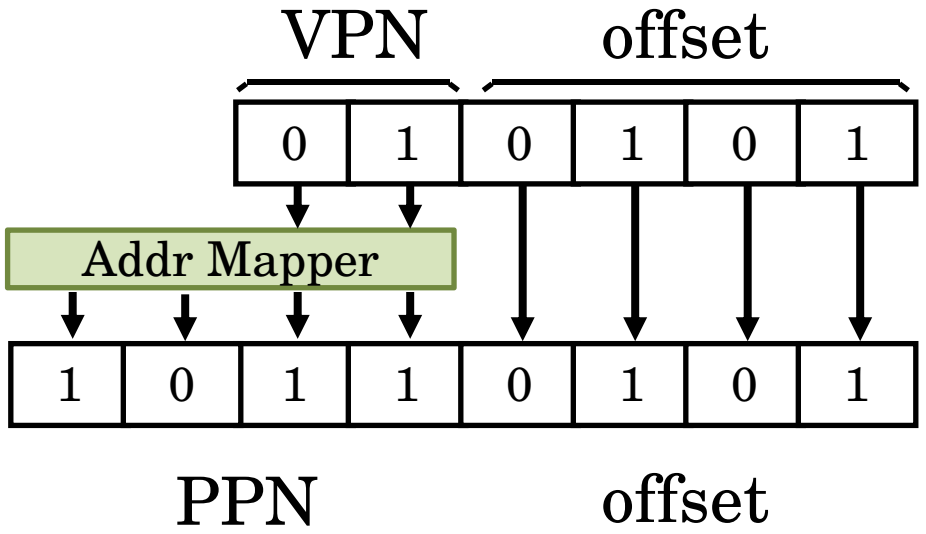
Page Size	Low Bits (Offset)	Virtual Address Length (Bits)	High Bits (VPN)	# of Virtual Pages
16 bytes	4	10	6	64
1 KB	10	20	10	1K
4 KB	12	32	20	1M
512 bytes	9	16	7	128
4 KB	12	16	4	16

VIRTUAL TO PHYSICAL PAGE MAPPING

Number of bits in virtual address

need not be equal to

number of bits in physical address

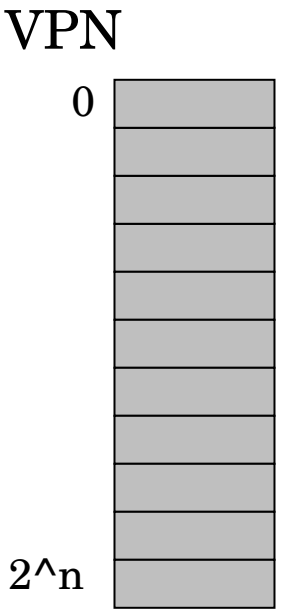


- How should OS translate VPN to PPN?
- For paging, OS needs a general mapping mechanism
- What data structure is good?

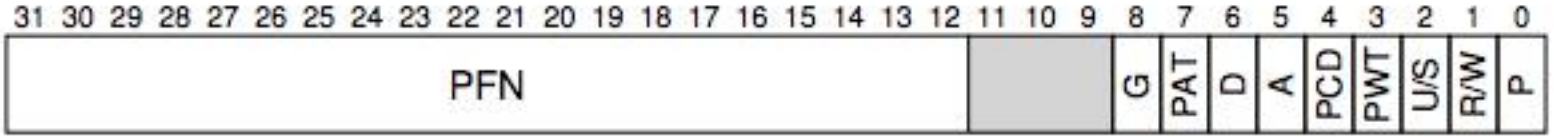
LINEAR PAGETABLES

What is a good data structure ?

Simple solution: Linear page table
a.k.a. *array*



Page Table Entry



Physical Frame Number

Flags

HOW BIG IS A PAGETABLE?

How big is a typical page table?

- assume **32-bit** address space
- assume 4 KB pages
- assume 4 byte page table entries (PTEs)

Final answer: 2 ^ (32 - log(4KB)) * 4 = **4 MB**

- Page table size = num entries * size of each entry
- Num entries = num virtual pages = 2^(bits for vpn)
- Bits for vpn = 32– number of bits for page offset
  = 32 – lg(4KB) = 32 – 12 = 20
- Num entries = 2^20 = 1M
- Page table size = Num entries * 4 bytes = 4 MB

WHERE ARE PAGETABLES STORED?

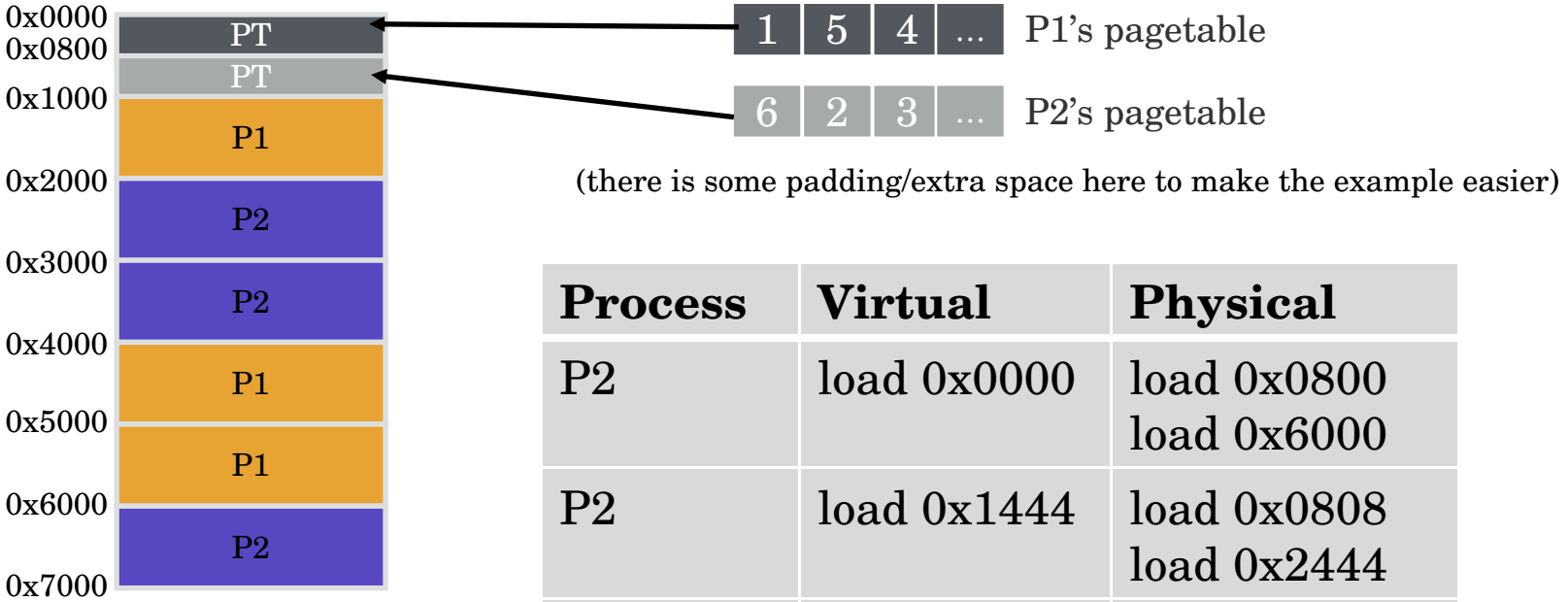
Implication: Store each page table in memory

- Hardware finds page table base with register (e.g., CR3 on x86)

What happens on a context-switch?

- Change contents of **page table base register** to newly scheduled process
- Save old **page table base register** in PCB of descheduled process

EXAMPLE: MEMORY ACCESSES WITH PAGING



(there is some padding/extra space here to make the example easier)

Process	Virtual	Physical
P2	load 0x0000	load 0x0800 load 0x6000
P2	load 0x1444	load 0x0808 load 0x2444
P1	load 0x0444	load 0x0000 load 0x1444

Assume

- 4KB pages, 16 total pages
- 4 bit VPN, 12 bit offset
- 8 bytes per page table entry

TIME FOR TRANSLATION STEPS WITH PAGING

For each memory reference:

(fast) 1. extract **VPN** (virt page num) from **VA** (virt addr)

(fast) 2. calculate addr of **PTE** (page table entry)

(slow) 3. read **PTE** from memory

(fast) 4. extract **PFN** (page frame num)

(fast) 5. build **PA** (phys addr)

(slow) 6. read contents of **PA** from memory into register

Which steps are fast and which are slow?

ADVANTAGES OF PAGING

No external fragmentation

- Any page can be placed in any frame in physical memory

Fast to allocate (sbrk) and free

- alloc: No searching for suitable free space
  - Just pick first free page from free list
  - Can be implemented as bitmap (“in use” or “free”)
- free: Just re-add to free list

Simple to swap-out portions of memory to disk (later lecture)

- Page size matches disk block size
- Can run process when some pages are on disk

PROBLEMS WITH PAGING (SO FAR)

Page table itself is stored in memory
- Will not fit in MMU or CPU registers
- Need to touch main memory twice for every memory access

Solution: Cache most recently/commonly used translations

Page tables are too big
- Array of size N, where N=(MemorySize)/(PageSize)
- Need one page table per process

Solution: Find a more efficient data structure

Will be covered in the next two lectures!