PRINCIPLES OF COMPUTER SYSTEMS DESIGN

CSE130

Winter 2020

Memory Management II - Allocation

Baskin Engineering UC SANTA CRUZ

Notices

- **Lab 3** due **Sunday March 1**
- **Assignment 3** due **Sunday February 16**
  - Equivalent to two questions in the final
- My office hours changed *today only*
  - 3pm to 4pm
- No class on Monday February 17
  - President's Day

Today's Lecture

- Brief Recap
- Memory Allocation
  - Partitioning Schemes
  - Fragmentation
  - Allocation Schemes
  - Paging
  - Paging on the Intel 80386
- Lab 3 - Secret Sauce 

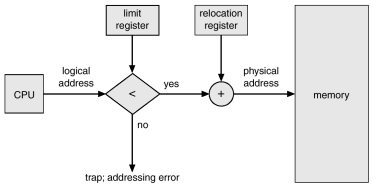
Operating Systems & Memory

- Computers consist of **CPUs**, **memory**, and **I/O devices**
- An Operating System's primary concerns are **equitable CPU scheduling**, and **safe and secure memory management**
- This series of lectures largely ignores the on-chip L1 & L2, and on-die L3 caches to concentrate on the off-chip, but on-board, Random Access Memory (RAM)
  - However, L cache misses still have to be taken into consideration in CPU scheduling algorithms, so don't forget about the L caches
- Memory Types:
  - Registers fastest tiny bytes
  - L Caches fast small KB to MB
  - **RAM** **slow** **large** **GB**
  - Hard Disk very slow massive TB

Memory Protection

The relocation-register scheme:

- **Relocation Register**
  - contains the smallest permitted ***physical*** address
- **Limit Register**
  - contains the maximum ***logical*** address

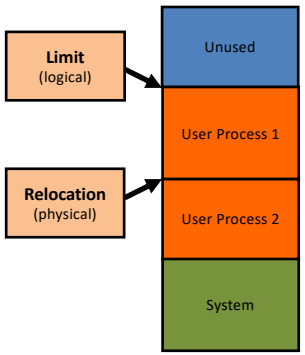


Ensures processes only accesses their own address space

Memory Partitioning

- Main memory must hold the code and data for both the OS and all user processes
- Main memory needs to be **divided into partitions**:
  - The resident operating system, usually held in low memory
  - User processes held in high memory
- Typically we want several user processes simultaneously in memory (multiprogramming / time-sharing)
- Need to partition memory to protect processes from each other
- Also must **allocate memory** to processes in a **safe** & **timely** manner

Memory Allocation

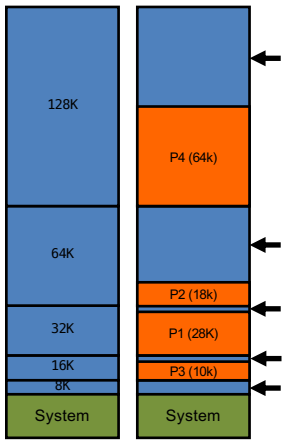


Questions:

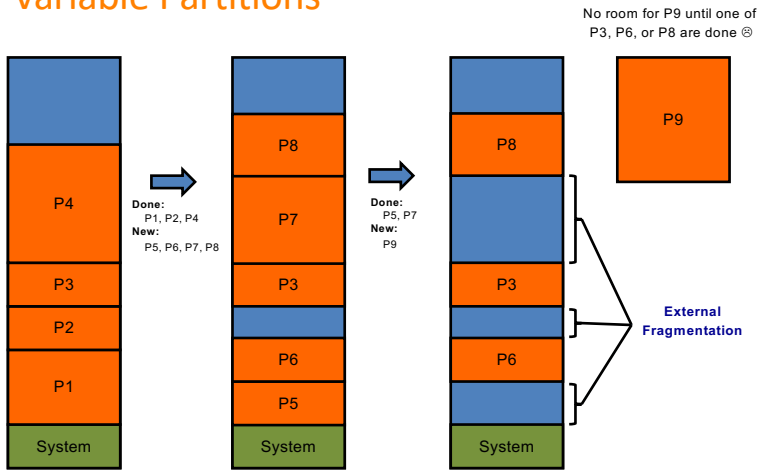
- When is memory allocated?
- How much memory?
- Which memory?
- Contiguous memory?
- Can process be swapped?

Fixed Partitions

- Traditional first, simple approach
- Memory statically partitioned
- Partitions of different, fixed sizes
- Processes allocated **smallest available partition large enough for their needs**
- Addresses bound at load time
- Fixes partition in which process runs
- **Problem?**
  - **Internal fragmentation** (wasted space)



Variable Partitions



Fragmentation

- **External Fragmentation**
  - Total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation**
  - Allocated memory may be slightly larger than requested memory
    - This size difference is memory internal to a partition, but not being used

Allocation Algorithms

How to satisfy a request from a list of free holes?

- **First-fit**
  - Allocate the first hole large enough
- **Best-fit**
  - Allocate the smallest hole large enough
    - Must search entire list, unless ordered by size
  - Produces the smallest leftover hole
- **Worst-fit**
  - Allocate the largest hole available
    - Must also search entire list
  - Produces the largest leftover hole

External Fragmentation

We can reduce external fragmentation via:

- **Compaction**
  - Shuffle memory contents around to place all free memory together in one large block
  - Compaction is possible only if relocation is dynamic, and is done at execution time
  - **Problem?**
    - All user processes have to wait whilst OS re-shuffles the free memory
- **OR…**
  - Map logical address space of a process to non-contiguous physical memory
  - Process is allocated physical memory from where ever it is available
  - This is known as **PAGING**

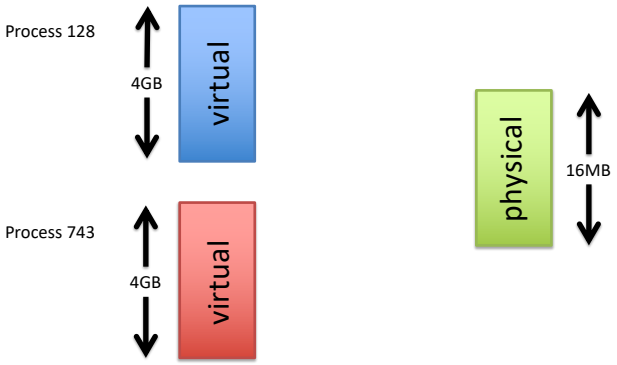
Paging: Definitions

- **Page** (verb) *"Divide a piece of software or data into sections, keeping the most frequently accessed in main memory and storing the rest in virtual memory"* – Google
- **Paging** (noun) *"The transfer of pages of data between the main memory of a computer and its auxiliary memory"* – Collins English Dictionary
- **Paging** (noun) *"A technique of storage management that transfers pages from secondary storage to main storage when they are required, and returns them to secondary storage when they are not"* – dictionary.com

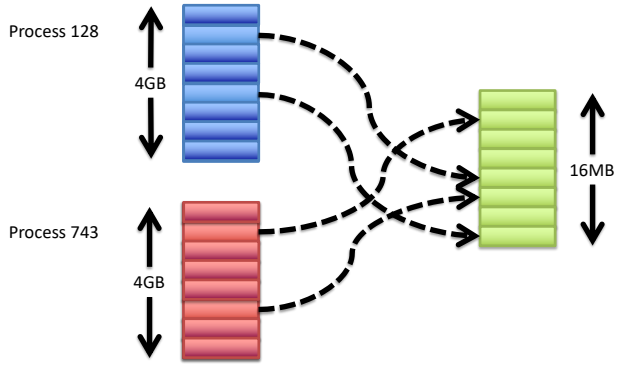
Paging for Memory Management

- **Paging** (the concept) is usually what people are talking about today when they say "**virtual memory**"
- Virtual memory requires hardware and software support
  - Hardware component is the **MMU** (memory management unit)
  - Operating System decides **how to translate virtual <=> physical addresses**

Paging Overview



Paging Overview



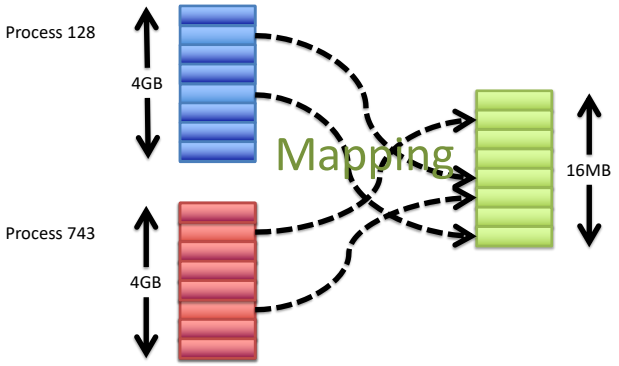
Paging Overview

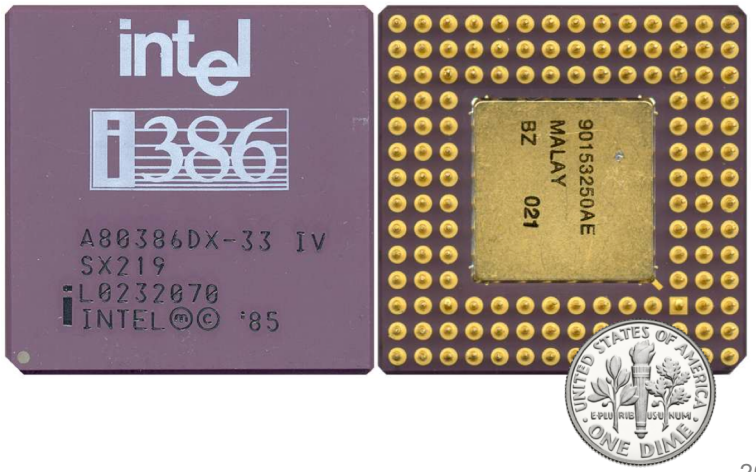
- Based on the notion of a virtual address space
  - **The *illusion* of a large, contiguous address space**
    - **Virtual address space >> Physical address space**
  - Each **process** gets its own virtual address space
    - Threads do not get their own
- Divides the address space into fixed size **pages**
  - **Virtual Page:** A "chunk" of the virtual address space
  - **Physical Page:** A "chunk" of the physical address space
    - Also known as a **frame**
  - **Size of virtual page == size of physical page**

Paging Overview

- We need to **MAP** virtual pages to physical pages
  - By itself, a virtual page is an illusion
    - Cannot actually store anything
    - Needs to be backed by a physical page
  - Before a virtual page can be accessed…
    - It must be paired with a physical page
      - i.e. it must be **mapped** to physical page
    - This mapping must be stored somewhere
  - On every access to the virtual page…
    - It's mapping is looked up
    - Access is then directed to the relevant physical page

Paging Overview

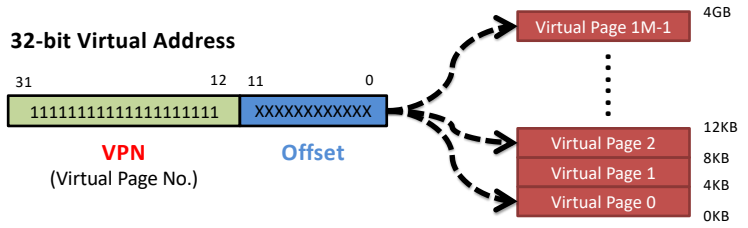




Paging in the Intel 80386

- Intel 80386 (Mid 1980's)
  - 32-bit processor
  - 4KB page size
- What is the **virtual address space**? (the addressable memory)
  - 2^{32} = 4GB
- How many **virtual pages** in the virtual address space?
  - 4GB/4KB = 1M
- What is the size of the **physical address space**?
  - It depends, but <= 4GB (i.e. no point in installing > 4GB memory as it can't be addressed)
- How many **physical pages** are there?
  - It depends, but <= 1M
- Even though physical address space may be less than 4GB, we'll assume physical addresses are 32bits

Intel 80386: Virtual Pages

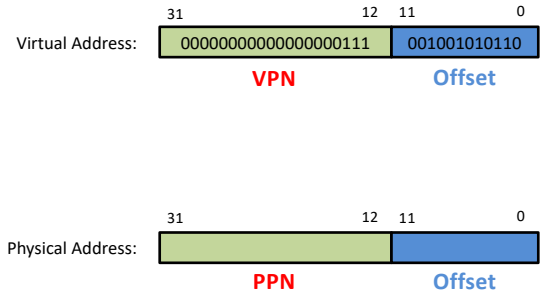


Highest 20 bits are the virtual page number
Lowest 12 bits are the offset within that page

See class video for animation

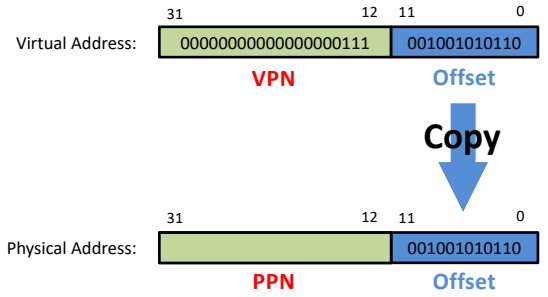
Intel 80386: Address Translation

- Assume Virtual Page 7 is mapped to Physical Page 34
- We want to access some offset in Virtual Page 7



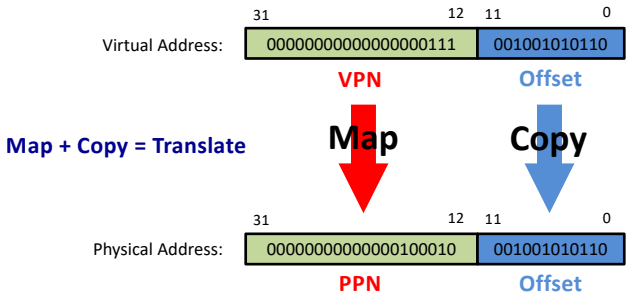
Intel 80386: Address Translation

- Assume Virtual Page 7 is mapped to Physical Page 34
- We want to access some offset in Virtual Page 7



Intel 80386: Address Translation

- Assume Virtual Page 7 is mapped to Physical Page 34
- We want to access some offset in Virtual Page 7



A Note on Notation

- We **MAP** pages
  - Specifically, virtual pages to physical pages
- We **TRANSLATE** addresses
  - Specifically, virtual addresses to physical addresses

Intel 80386: VPN->PPN

- **Problem**: Need to keep track of VPN -> PPN mappings
  VPN 65 -> PPN 981
  VPN 3161 -> PPN 1629
  VPN 9327 -> PPN 524

- **Solution**: **Page Table**
  - A "lookup table" for the mappings
  - Can be thought of as an array
  - Each array element is called a **page table entry** (PTE)

```
uint32_t PAGE_TABLE[1<<20];
PAGE_TABLE[65] = 981;
PAGE_TABLE[3161] = 1629;
PAGE_TABLE[9327] = 524;
```


Intel 80386: Page Tables

Two problems with page tables:

- **Page table is too large**
  - Page table has 1M entries
  - Each entry is 4B (20-bit PPN)
  - **Page table is 4MB** (ouch!)
    - Very expensive in the 1980s
    - Still problematic today on embedded devices

Intel 80386: Page Table Too Large

- Typically, the majority of PTEs are empty

```
PAGE_TABLE[0] = 141;
...
PAGE_TABLE[532] = 1190;

Unallocated (i.e. wasted) space!

PAGE_TABLE[1048402] = 845
...
PAGE_TABLE[1048575] = 742; // 1048575 ==(2^20)-1
```

- Unallocating every empty PTE is tedious
  - Instead, only unallocate long sections of empty PTEs

Intel 80386: Page Table Too Large

- How do we allow PTEs to become "unallocated"?
  - The page table must be restructured
- Before restructuring: **flat**

```
uint32_t PAGE_TABLE[1024*1024];
```

- After restructuring: **hierarchical**

```
uint32_t *PAGE_DIRECTORY[1024];
PAGE_DIRECTORY[0] = NULL; // 1024 PTEs unallocated
PAGE_DIRECTORY[1] = NULL; // 1024 PTEs unallocated
PAGE_DIRECTORY[2] = malloc(sizeof(uint32_t)*1024);
PAGE_DIRECTORY[2][0] = 753;
PAGE_DIRECTORY[2][1023] = 21;
PAGE_DIRECTORY[3] = NULL; // 1024 PTEs unallocated
PAGE_DIRECTORY[4] = NULL; // 1024 PTEs unallocated
```


Intel 80386: Page Tables

Two problems with page tables:

- Page table is too large
  - Page table has 1M entries
  - Each entry is 4B (20-bit PPN)
  - Page table is 4MB (ouch!)
    - Very expensive in the 80s
    - Still problematic today on embedded devices
- **Page table is stored in memory**
  - Before every memory access, we always have to fetch the PTE from comparatively slow RAM => big performance penalty

Intel 80386: Accelerating Translation

- **Problem:** Retrieving PTEs from memory is slow ☹
- **Solution:** "Cache" PTEs inside the processor ☺
  - **Translation Lookaside Buffer** (TLB)
    - "Lookaside Buffer" is an archaic term for a cache
  - Whenever a virtual address needs to be translated, the TLB is searched: "**hit**" vs. "**miss**" (we'll look at these next week)
  - 32-entry TLB on 80386
  - Each TLB entry has a **tag** and some **data**
    - Tag: 20-bit VPN + 4-bit metadata
    - Data: 20-bit PPN

Next Lecture

- Address Translation
- TLB Hits and Misses
- Demand Paging