CS 326: Operating Systems

Paging

Lecture 12

Today's Schedule

- Introduction to Paging
- Addressing Pages

Today's Schedule

- **Introduction to Paging**
- Addressing Pages

Memory Fragmentation

- Thinking back to segmentation, one big issue is **memory fragmentation**
- We could move segments around… (say, on a fixed interval like every 10 seconds)
  - Garbage collector style
  - …but this is very inefficient
- Wouldn't it be great if we could assign parts of physical memory to processes bit by bit instead?

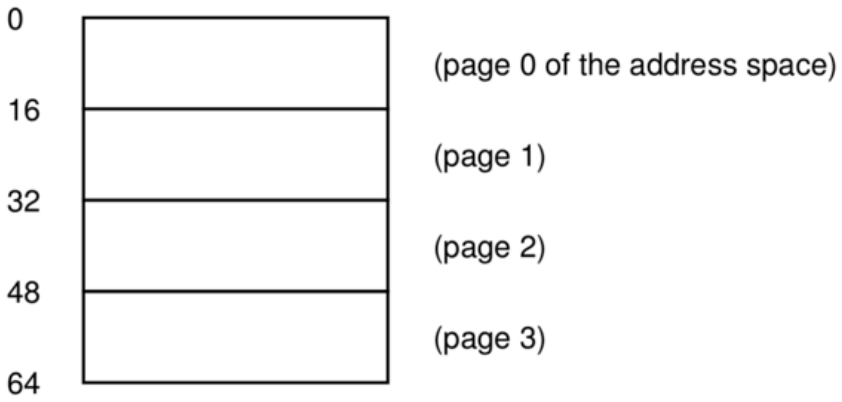
Paging

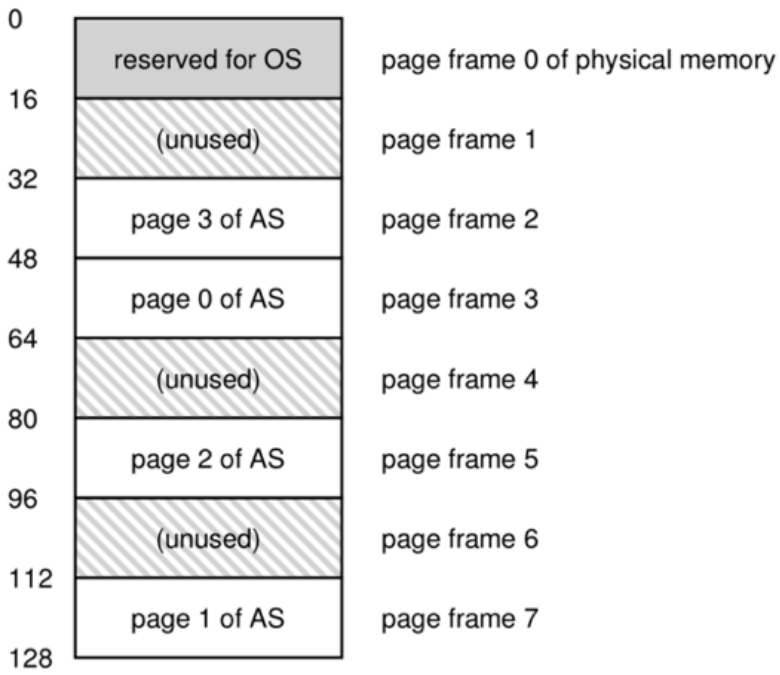
- We can use **paging** to allocate non-contiguous regions of memory
- With paging, each **page** of memory can be mapped to a process
  - Example: you get page 1, 3, 45, 682
    - Need more memory? Sure, here's page 981
- This approach avoids fragmentation and reduces waste
  - Most memory we can waste: page_sz - 1

Page Frames

- Physical memory is broken up into pieces called **page frames**
  - Each page frame contains a single **page**
  - Pages are virtual, page frames are physical
- Page frame sizes are OS and hardware specific
  - 4KB, 2MB, 1GB on Intel x86_64
  - 4KB is extremely common
  - Generally a power of 2

Address Space Example





Determining the Page Size

- Drumroll please!!

```
[magical-unicorn:~]$ getconf PAGE_SIZE
4096
```

Tracking Memory

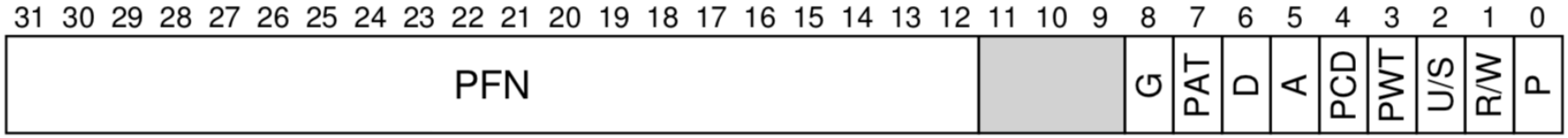
- Awesome! We've (mostly) eliminated fragmentation!
- Now comes the hard part: how do we figure out which pages are allocated to our processes?
- We achieve this with a **page table**

Page Table

- The page table is a structure maintained by the OS for each process
- When a process is swapped in or out, the page table is as well
- Page tables consist of **page table entries** (PTEs)

A Page Table Entry

- PFN: “Physical frame number”
- There are several other fields…



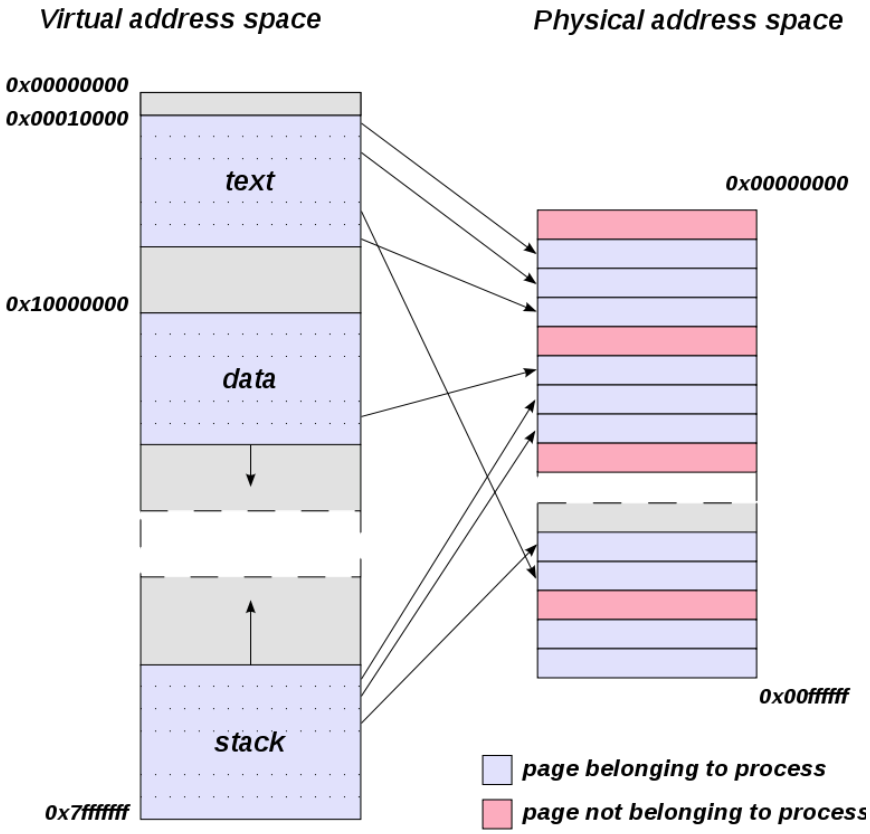
Page Table Entry Elements

- Valid/Invalid bit – whether the memory is allocated
  - Used to provide our illusion of a single, giant address space without having to actually allocate pages
- Protection Bits – similar to the NX bit, segment protection bits, etc.
- Present bit – whether the page is actually in memory or not
  - Why do we need this?
- Dirty bit – whether the page has been changed since it was read into memory
- And many more, depending on OS/hardware/etc.

Implementing the Page Table

- The page table's main job is map virtual addresses to physical addresses
- There are plenty of options for doing this…
- At a basic level, we can imagine using a simple array to keep track of the locations
  - Virtual Page 0 ➔ Physical Page Frame 3
  - Virtual Page 4 ➔ Physical Page Frame 1
  - That's really all paging is… Keeping track of what virtual pages map to particular physical frames

Page Table Translation



Address Translation

- To facilitate this process, we use **address translation** to convert virtual page ➔ physical page
- Split the memory address into two pieces: the **virtual page number** and the **offset**
- The virtual page number determines **where** in physical memory the data is located
  - VPN ➔ Page Table ➔ Physical Page Frame
  - If we're using an array as our PT , these are the array indices
- The **offset** indicates where the data is stored within the physical page

Paging Issues

- The main issue with paging is that page tables can become quite large
  - 4KB page size, 32-bit address space = 1m pages
- It also takes time to do all this translating…
- We can't just rely on a hardware component to save us here: it would require too much of its own, fast memory!
  - We have a few options, including caching portions of the address translations

Caching Translations

- Recently-used translations are maintained by the MMU in the **translation lookaside buffer** (TLB)
- Using the TLB will be faster than the page table, but we can only store a limited number of entries
  - If the TLB doesn't have the VPN we're looking up, it's called a TLB miss

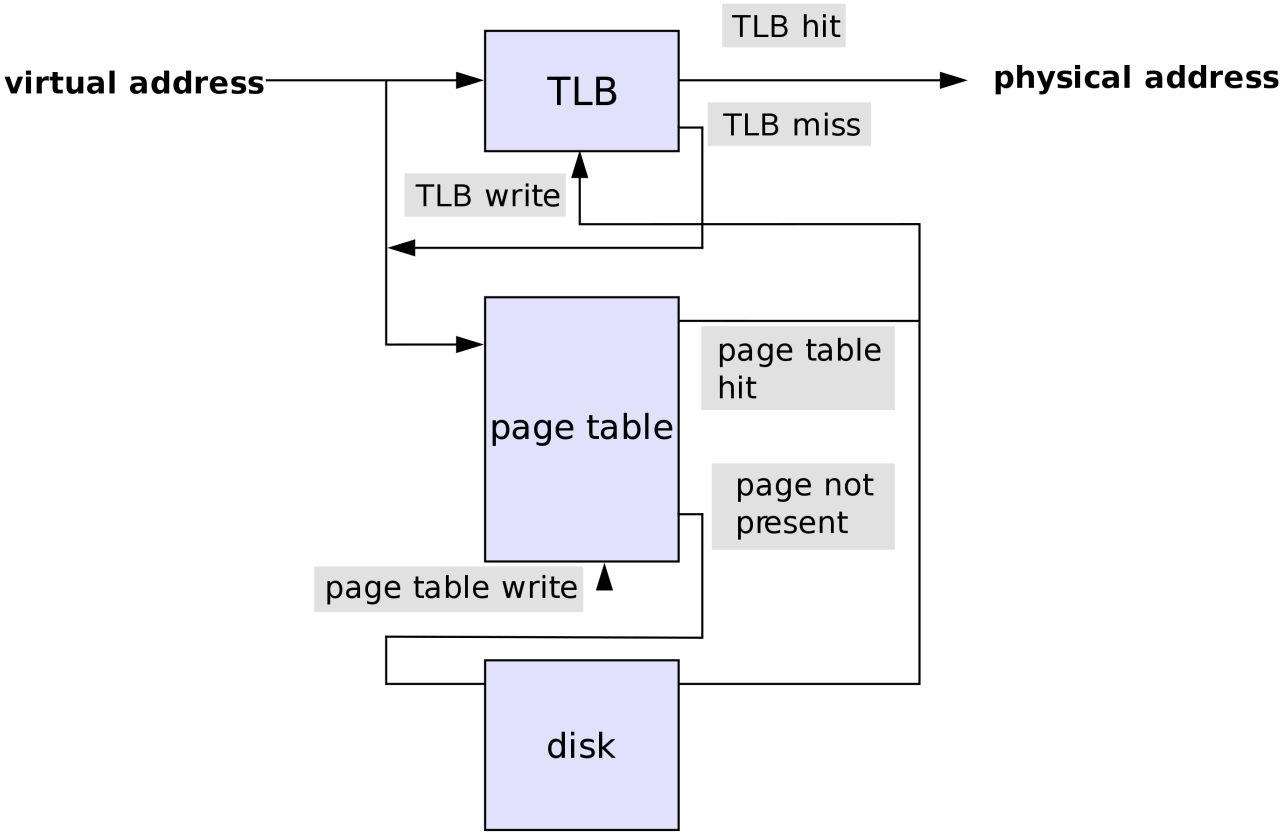
Backing Store

- In addition to fragmentation-free allocation, we can use paging with a **backing store**
- Commonly a disk, pages that are not used frequently can be **swapped out** to disk
  - Frees up memory for other processes to use
  - When swapped memory is referenced by a program, a **page fault** occurs
    - Must be reloaded back into main memory
- In modern times, SSDs make this very fast

Translation Process

1. Check the TLB
   - Mapping found? We're done!
   - Otherwise, we have a TLB miss…
2. Consult the page table
   - Page table hit? Add the entry to the TLB, and then we're done!
   - Otherwise, we have a **page fault**
3. Consult the backing store

Translation Process



Today's Schedule

- Introduction to Paging
- **Addressing Pages**

Virtual Addresses

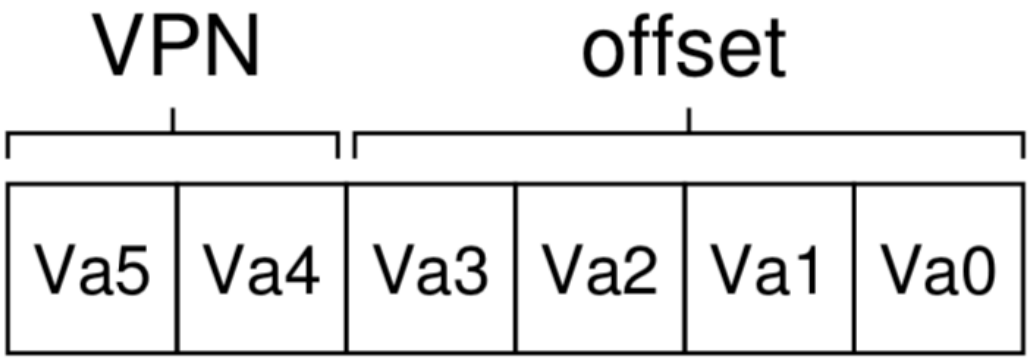
- With Paging, we can now piece together large, non-contiguous portions of physical memory
- These combine to give the illusion of a nice, big, de-fragmented address space
- As usual, processes do not need to know that they are running under this **virtualized** memory scheme

Address Translation

- To help the OS keep track of where pages are located, memory addresses are **split up**
- In the most basic case, the addresses are split into two pieces:
  - The virtual page number
  - The offset

Address Translation

- Here's a single address split into VPN/offset:



VPN

- The VPN (virtual page number) tells the OS which page the memory is located in via the page table
- The offset specifies what **part** of the page to move to
- We have a level of indirection here: determine the page number, then figure out what byte to start at
  - Note: the offset portion of the address is the same for both virtual and physical memory… why?

Virtual vs. Physical

- With Paging, the **physical** and **virtual** addresses can be different sizes
  - We don't necessarily have a 1:1 mapping between **pages** and **frames**
  - e.g., we have a 32-bit system (2^{32} = 4 GB address space), but only 1 GB of RAM installed
- If the virtual address space is larger than the physical, those extra pages will be swapped to disk

Inverted Page Table

- With an **inverted** page table, the page table actually does map directly to physical addresses
- Instead of a page table per process, there is a single page table for the system
- Address translation requires searching the entire page table
  - Often implemented as a hash table with PIDs stored as part of the memory address for access protection

Page Table Entries

- If we have a 32-bit machine with a page size of 4 KB, how many page table entries do we have?
- 4 KB = 4096 bytes, or 2^{12} ($log2(4096) = 12$)
  - So that means we need 12 bits for the offset
- 32 bits – 12 offset bits = 20 bits for the VPN
- Entries (one for each page) = 2^{20} (~1m)

Page Table Size

- To calculate our page table size:
  - Entries * sizeof(PTE)
  - (straightforward)
- So if each PTE takes up 4 bytes then we have:
  - 1 MB * 4 = 4 MB (on our example 32 bit machine)
- 4 MB is not too bad… except we need a page table for **every** process!

Example Address Structure: xv6

```
A virtual address 'la' has a three-part structure as follows:

+--------10------+-------10-------+---------12----------+
| Page Directory |   Page Table    | Offset within Page  |
|      Index     |      Index      |                     |
+----------------+-----------------+---------------------+
 \--- PDX(va) --/ \--- PTX(va) --/
```

Page Directory?

- Here, page directories point to page tables, which then point to physical addresses
- This allows the OS to omit page tables when large ranges of virtual addresses are unused
- It takes a **lot** of memory just to store the page table
  - Generally, portions of the page table itself can be stored in virtual memory and could even be swapped to the disk if needed

Mapped Memory

- We spent some time discussing how mmap lets us share memory across processes
- How can we implement this?
- …
- At a basic level: both processes' page tables have valid mappings for the shared memory region

Wrapping Up

- Paging gives us much more flexibility when managing memory compared to a simple base + bound or segmentation
- Most modern OS implement some form of paging
- The main thing to remember: those memory addresses you see are fake
  - They're getting translated to actual physical locations in RAM