PRINCIPLES OF COMPUTER SYSTEMS DESIGN

CSE130

Winter 2020

Memory Management IV - Demand Paging

Baskin Engineering UC SANTA CRUZ

Notices

- **Lab 3** due **Sunday March 1**
- **Assignment 4** due **Monday February 24**
  - 24 hour delay due to Presidents' Day
  - Make sure you have install from Wednesday evening
    - `manpage` test was passing even when it wasn't ☺
    - `make submit` created incorrectly named archive ☹
    - `cartman -r` was sending all carts to the same line ☹

Today's Lecture

- Demand Paging
- Swapping
- Kernel Memory Allocation
- Assignment 4 & Lab 3 Secret Sauce 

Paging vs. Segmentation

Linux	Paging (with Copy-on-Write)
BSD 4.4	Paging (Least Recently Used replacement)
FreeBSD	Paging (Least Actively Used replacement)
Unix SVR4	Paging
SCO Unix	Paging
macOS	Paging
iOS	Paging
IBM AIX	Paging
Android	Paging
Windows	Paging
OS400	Segmentation









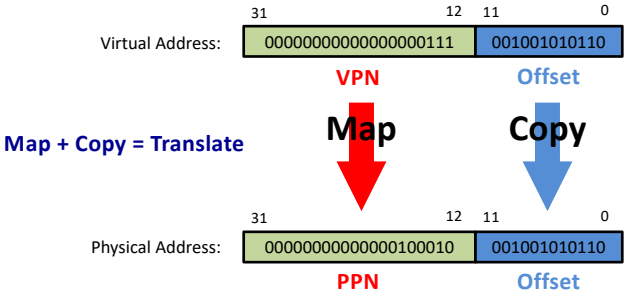




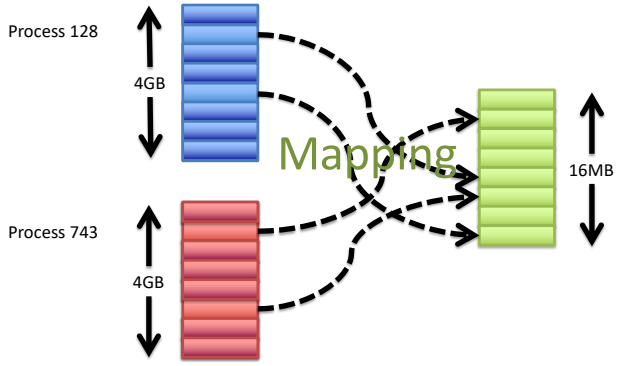


Intel 80386: Address Translation

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



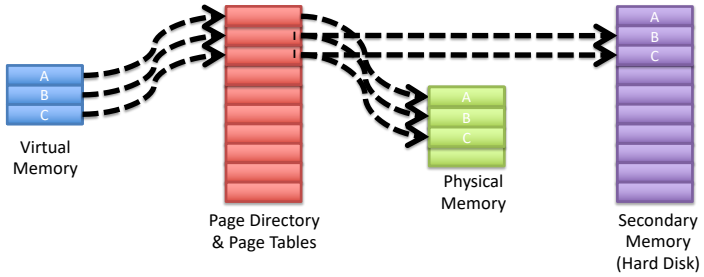
Paging Overview



Memory Pages on Disk

- Why would a virtual page be mapped to disk?
- **Demand Paging**
  - When a large executable file (a program) or a large data file is read from disk by a running process, not all of it is loaded into memory at once
  - Page size chunks are loaded **on-demand**
  - If most of the file is never read
    - **Saves time** - disks are slow
    - **Saves space** - memory is limited
  - **Problems?**

Demand Paging



PAGE FAULT!

See class video for animation

Demand Paging Algorithm

1. Interrupt detected to the operating system
2. Save the user registers and currently running process state
3. Determine that the interrupt was a page fault
4. Check that the page reference was legal
5. Determine the location of the page on the disk
6. Request a read from the disk to a free page
7. Wait in a queue for the disk until the read request is serviced
8. Wait for the device seek and/or latency time
9. Begin the transfer of the page to a free page
10. While waiting, allocate the CPU to some other process
11. Receive an interrupt from the disk subsystem (I/O completed)
12. Save the registers and process state for the other process
13. Determine that the interrupt was from the disk
14. Adjust the page table and other tables to show page is now in memory
15. Wait for the CPU to be allocated to the original process again
16. Restore the user registers, process state, and new page table, and then resume the interrupted instruction

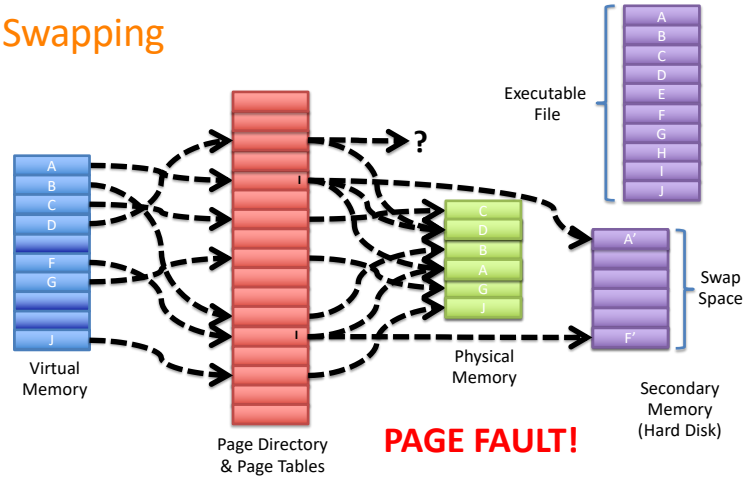
Demand Paging Problems

- A page fault may occur at any address (i.e. at any memory location)
- Consider the pseudo machine instruction `ADD C,A,B`
  - Adds the integer at address A to the integer at address B, saves the result at address C
  - **What's the worst case memory access scenario?**

Memory Pages on Disk

- Why else would a virtual page be mapped to disk?
- **Swapping**
  - Physical memory is exhausted
    - Many memory hungry processes are running
  - Or when a process requires "simultaneous" access to more virtual pages than there are physical pages available
    - "**Least used**" physical page is copied to disk for later re-instatement
  - When you try to start another process…
    - Some physical pages have to be "**swapped out**"
    - Data in these physical pages is written to disk
    - The physical pages are now tagged as being free
    - Their Virtual Memory PTEs are now invalid
      - (yet another) page fault occurs ☹

Swapping

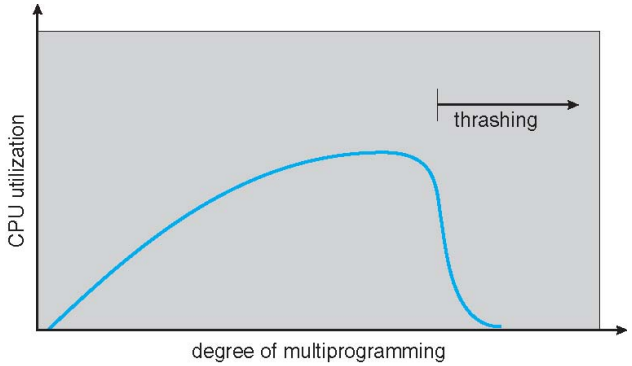


See class video for animation

Thrashing

- When a process later accesses a physical page that was swapped out, the page is swapped back in to physical memory
  - May cause another physical page to be swapped out
  - If this happens too rapidly, it's known as **THRASHING**
  - **Extreme performance degradation** ☹
- If a process does not have "enough" pages, the page-fault rate can be very high indeed
  - Page fault raised to get desired page
  - Replaces existing page
  - But quickly need replaced page back again
  - Leads to **Low CPU utilization**…
    - Operating system thinks that it needs to increase the degree of multiprogramming
    - Another process gets added to the system with negative results ☹

Thrashing



Page Fault Problems

- Three major components of page-fault service time (Intel 80386) :
  - Service the page-fault interrupt (≈ 1-100 microseconds)
  - Read in the page (≈ 24 milliseconds)
  - Restart the process (≈ 1-100 microseconds)
- Fault-access time ≈ 25 milliseconds
- Memory-access time ≈ 100 nanoseconds
- Let **p** denote the probability of a page fault
- Effective access time

  = (1-**p**) x memory access time + **p** x page fault time

  = (1-**p**) x 100 ns + **p** x 25 ms
- For performance degradation to be < 10% … **p** < $4x10^{-7}$

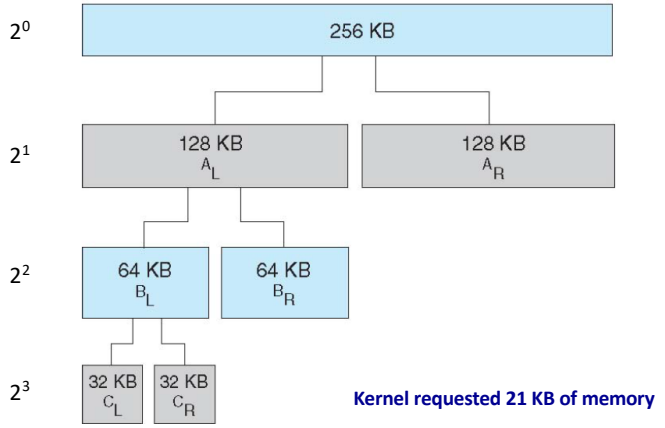
Page Fault Optimisation

- Processes can be loaded by page faulting
- Consider `fork()`
  - In many cases, child process may not make any changes to parent's data
- **Copy-on-Write** (COW) allows both parent and child processes to initially share the same pages in memory
  - If either process modifies a shared page, only then is the page copied
  - COW allows efficient process creation as only modified pages are copied
- In general, free pages are allocated from a pool of **zero-fill-on-demand** pages
  - **Why zero-out a page before allocating it?**
- `vfork()` variation of `fork()` system call has parent suspend and child uses the address space of parent

Allocating Kernel Memory

- Treated differently from user memory
- Often allocated from a free-memory pool
- Kernel requests memory for structures of varying sizes
- Some kernel memory needs to be contiguous
- **The Buddy System**
  - Allocates kernel memory from fixed-size segment consisting of physically contiguous pages
  - Memory allocated using a **power-of-2 allocator**
    - Satisfies request in units sized in proportion to a power of 2
    - Request rounded up to next highest power of 2
    - When smaller allocation needed than is available, current chunk split into two buddies of next-lower power of 2

Buddy System Allocator

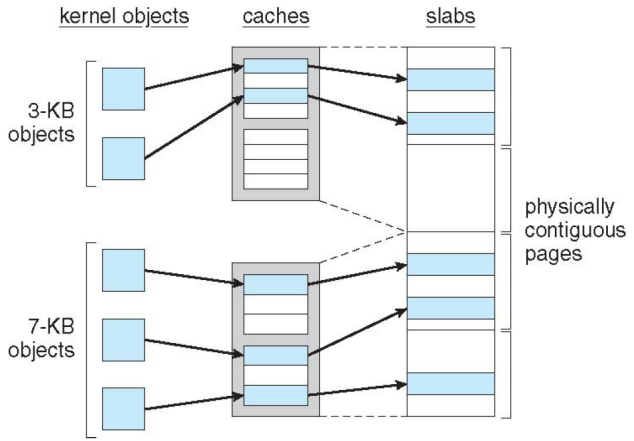


Kernel requested 21 KB of memory

Slab Allocator

- Alternate strategy to the Buddy System
- **Slab** is one or more physically contiguous pages
- **Cache** consists of one or more slabs
  - Single cache for each unique kernel data structure
    - Processes, Threads, File Handles, Locks, Semaphores, etc. etc.
  - When cache created, filled with objects marked as free
  - When structures stored, objects marked as used
- **Benefits**:
  - No fragmentation
  - Fast servicing of memory requests

Slab Allocation



Secret Sauce

Assignment 4 - Basic / Manpage

- Busy Wait & "who's turn is it" global variable
  - Works, but inefficient ☹
- Single Condition Variable & "who's turn is it" global variable
  - Condition is "it's somebody's turn"
  - Thread that just ran updates "who's turn is it" and broadcasts
  - All other threads check if "who's turn is it" is them
- Multiple Condition Variables
  - Condition is "it's my turn"
  - Thread that just ran signals next thread's condition variable
- Multiple Locks
  - Thread that just ran releases next thread's lock
- Multiple Binary Semaphores
  - Thread that just ran puts next thread's semaphore up

Assignment 4 - Advanced / Cartman

`cartman` initialise anything you need before your system runs

`arrive` a CART has just arrived at a junction
- Create a thread for the CART
- `reserve` the junctions you need for exclusive access to the critical section
- But, you must make sure you don't call `reserve` when the junctions you need are already reserved by another CART / thread
- Call `cross()` when you have the two junctions

`depart` a CART has reached the other side of a critical section
- `release` the two junctions it reserved

- Remember:
  - You must deal with concurrency
  - You must have multiple CARTs in motion at the same time to avoid timeout

Next Lecture

- Page Replacement
- Frame Allocation