Administrivia

• Homework
  • HW2 is due Friday May 19
Recap

- Virtual address space
  - Virtual-to-physical translation mechanisms
- Page translation
  - Memory consumption and translation efficiency
- Memory Hierarchy
  - Virtual memory and cache replacement policies
Advanced paging

- Shared memory
- Copy-on-Write (COW)
- Memory mapped files
- Segmentation + paging
- Page frame management
Shared memory (1)

- Virtual address spaces isolate process from each other
  - Protection against buggy or malicious processes
- But this makes it difficult to share data
  - Has to copy twice with the help of the kernel
    - Process1 -> kernel -> process2
  - Why kernel can access to both address space?
Shared memory (2)

- Shared memory allows processes to share data using direct memory references
  - Both processes see updates to the shared memory segment
    - Process B can immediately read an update by process A
- OS kernel space is also a large chunk shared memory
  - All processes share the same piece of kernel memory
  - This is reason why kernel can copy data between processes
Shared memory (3)

- How can we implement sharing using page tables?
  - Have PTEs in both tables map to the same physical frame
Shared memory (4)

- Process perspective
Shared memory (5)

- How is kernel space shared?
  - Share upper level page tables instead of individual page frames
  - Mapped as global (G bit in PTE)
    - Will not be evicted from TLB during address space switch
  - Shared memory can also happen within the same virtual address space
Copy-on-Write (1)

• Problem: OS spends a lot of time copying data during `fork()`

• Observations

  1. Memory can be safely shared if no one write to it
  2. After `fork()`, if `exec()` is not invoked, then many memory can be shared (e.g., code)
  3. After `fork()`, if `exec()` is invoked, then the entire virtual address space is freed

• Copy would be a total waste of time
Copy-on-Write (2)

• Solution: use Copy-on-Write (CoW) to defer large copies as long as possible, hoping to avoid them altogether
  • Instead of copying pages, create shared mappings of parent pages in child virtual address space
  • Shared pages are protected as read-only in parent and child
    • Reads happen as usual
    • Writes generate a protection fault, trap to OS, copy content, change page mapping in client page table, restart writes
fork() without CoW

Parent process’s page table

Page 1
Page 2

Physical Memory

Child process’s page table

Page 1
Page 2
fork() with CoW

- Parent process's page table:
  - Page 1
  - Page 2

- Child process's page table:
  - Page 1
  - Page 2

- Physical Memory

Protection bits set to prevent either process from writing to any page.

When either process modifies Page 1, page fault handler allocates new page and updates PTE in child process.

Under what circumstances such copies can be deferred forever?
Memory mapped files (1)

- Mapped files enable processes to do file I/O using loads and stores
  - Instead of `open`, `read` into buffer, operate on buffer, ...
- Bind a file to a virtual memory region (`mmap()` in Unix)
  - PTEs map virtual addresses to physical frames holding file data
  - Virtual address `base + N` refers to offset `N` in file
- Initially, all pages mapped to file are invalid
  - OS reads a page from file when invalid page is accessed
Memory mapped files (2)

Pages are all invalid initially

What happens if we unmap the memory?
How do we know whether we need to write changes back to file?
Memory mapped files (3)

- Writing back to file
  - OS writes a page to file when the page evicted, or unmapped
  - If page is not dirty (has not been written to), no write needed
    - Dirty bit trick (not protection bits)
Memory mapped files (4)

- Program loading
  - Memory map executables to the memory
Segmentation + paging

- x86 architecture allows both segmentation and paging
  - Segmentation: translates logical address (segment#, offset) to virtual address
  - Paging: translates virtual address to physical address
- Why this may help?
  - Intra virtual address space isolation
  - Faster memory region reference
    - Thread local storage (TLS)
Page frame management (1)

- VM as a tool for memory management
- Key idea: each process has its own virtual address space
  - It can view memory as a simple linear array
  - Mapping function scatters addresses through physical memory
    - Most memory allocations can be served by any physical page
    - Exceptions: large pages, memory accessed by hardware, etc.
Page frame management (2)

• Physical page frames can be easily managed as an array of reference counter
  • How many virtual pages are mapped to this physical page frame
    • 0 == available
  • An freelist to speed up searching for available page frames
Page frame management (3)

• On demand paging
  • **Problem**: process may allocate lots of memory but never use it
  • **Solution**: only creates a valid mapping when the allocated memory is truly accessed
    • Requires another data structure to manage mapped/available virtual addresses
Page frame management (4)

- Page allocation vs. heap
  - `malloc()`/`free()` allocates/frees memory from heap
  - Heap is managed by a user level library (standard C library)
  - Heap library allocates memory from the kernel in the granularity of pages, e.g., `brk()` and `mmap()`
  - Heap library then divides the memory allocated from kernel into smaller chunks and serve the requests
    - Goal: performance, minimize internal fragmentation
Summary

• Shared memory
• Copy-on-Write (COW)
• Memory mapped files
• Segmentation + paging
• Page frame management
Next class ...

- File systems
- Textbook
  - Module 36, 37, 39