Advanced Functionality

- Now we’re going to look at some advanced functionality that the OS can provide applications using virtual memory tricks
  - Shared memory
  - Copy on Write
  - Mapped files
Sharing

- Private virtual address spaces protect applications from each other
  - Usually exactly what we want

- But this makes it difficult to share data (have to copy)
  - Parents and children in a forking Web server or proxy will want to share an in-memory cache without copying

- We can use **shared memory** to allow 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
How can we implement sharing using page tables?
- Have PTEs in both tables map to the same physical frame
- Each PTE can have different protection values
- Must update both PTEs when page becomes invalid

How are we going to coordinate access to shared data?
Sharing (3)

- Can map shared memory at same or different virtual addresses in each process’ address space
  - Different:
    - 10th virtual page in P1 and 7th virtual page in P2 correspond to the 2nd physical page
    - Flexible (no address space conflicts), but pointers inside the shared memory segment are invalid
      - What happens if it points to data inside/outside the segment?
  - Same:
    - 2nd physical page corresponds to the 10th virtual page in both P1 and P2
    - Less flexible, but shared pointers are valid
OSes spend a lot of time copying data
- System call arguments between user/kernel space
- Entire address spaces to implement fork()

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 page, change page mapping in client page table, restart write instruction
- How does this help fork()?
Execution of fork()
fork() with Copy on Write

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

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

Under what circumstances such copies can be deferred forever?
Mapped Files

- 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

Pages are all invalid initially

A read occurs
Page 1

A read occurs
Page 2

Physical Memory
File Content 1
File Content 2

What happens if we unmap the memory?
How do we know whether we need to write changes back to file?
Writing Back to File

- OS writes a page to file when evicted, or region unmapped
- If page is not dirty (has not been written to), no write needed
  - Dirty bit trick (not protection bits)
Mapped Files (2)

- File is essentially backing store for that region of the virtual address space (instead of using the swap file)
  - Virtual address space not backed by “real” files also called Anonymous VM

- Advantages
  - Uniform access for files and memory (just use pointers)
  - Less copying

- Drawbacks
  - Process has less control over data movement
    - OS handles faults transparently
  - Does not generalize to streamed I/O (pipes, sockets, etc.)