11. Page Faults & Advanced Features

CS 4352 Operating Systems

Adopted from slides of Drs. Silberschatz, Huang, & Bryant

Review

- Virtual-to-physical address translation via page table
- Multi-level page table
  - Why?
- TLB
  - Why?
- Demand paging

Faults Caused by PTE

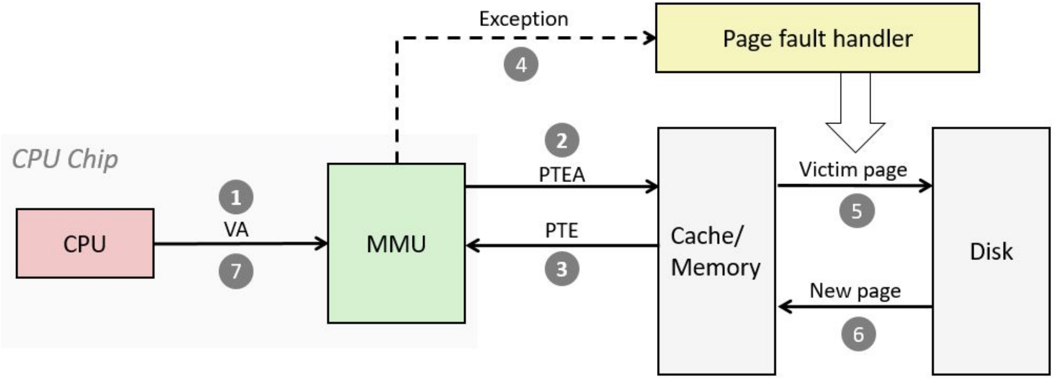
- What happens when a process accesses a page that is not in memory?
  - First of all, its PTE has the v bit set as invalid
  - When translation happens, a fault will be caused
    - Since PTE says its mapping is invalid
- Then what will happen?
  - A fault is generated and OS will get in to solve this problem
- Faults caused by memory accesses in paging system are called page faults
  - When a page fault occurs, OS **page fault handler** deals with it!
    - Remember we have learnt interrupts, traps, faults, and aborts?

Page Fault Types

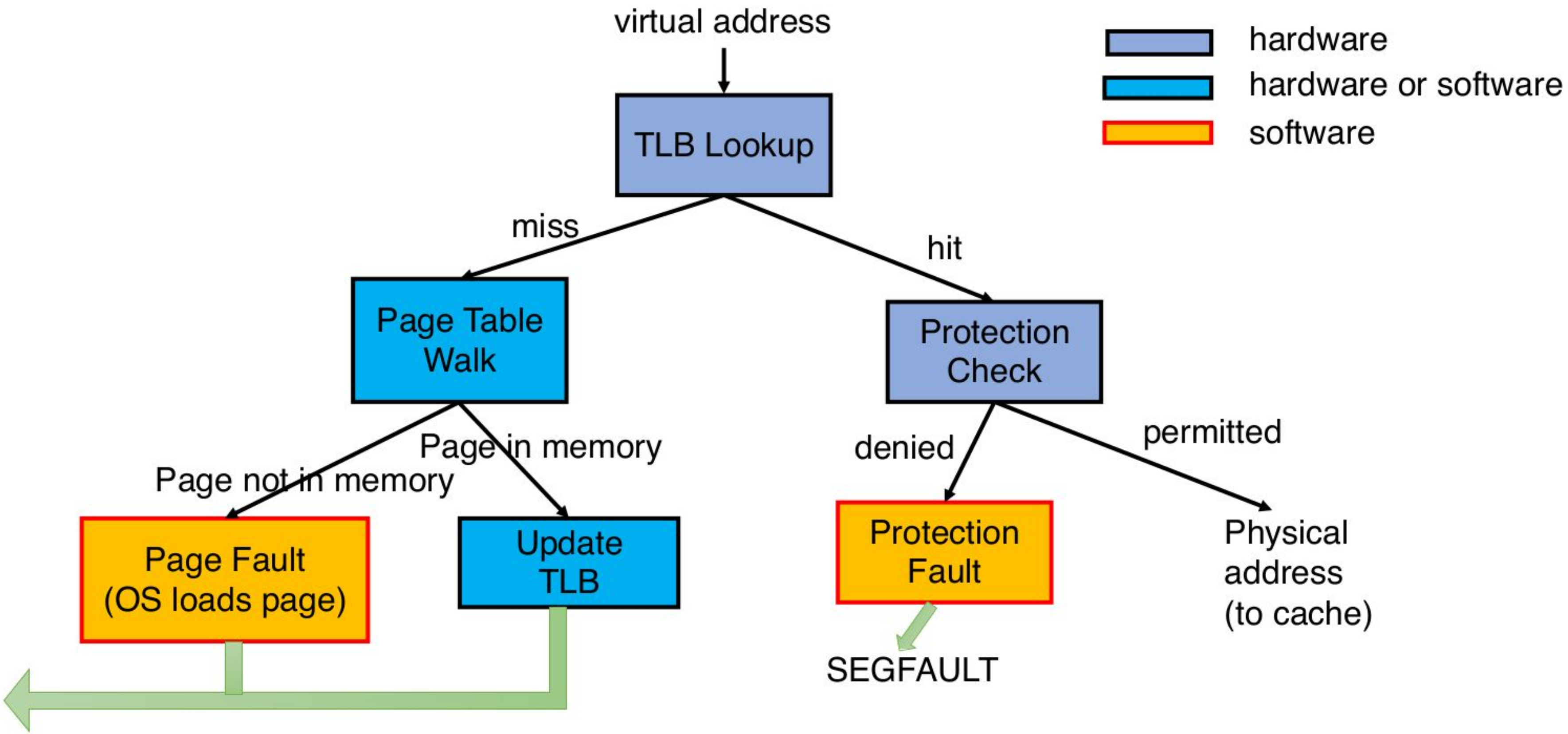
- Soft page fault: the page is in memory but not marked as in memory
  - OS page fault handler just needs to change the PTE to have the page in memory mapped
  - E.g., shared library
- Hard page fault: the page is not in memory
  - OS page fault handler brings the page into memory and changes the PTE
  - You often hear someone says **page fault** → they are referring to this hard page fault
- Invalid page fault: an address that is not part of the current virtual address space is accessed
  - OS page fault handler sends a SIGSEGV signal to the process
  - If no handlers installed, segmentation fault is what you often see…

Address Translation Example

1. Processor sends virtual address to MMU
2. TLB miss
3. PTE is fetched from page table
4. Valid bit is zero, so MMU triggers page fault exception
5. Handler identifies victim (and, if dirty, pages it out to disk)
6. Handler pages in new page and updates PTE in memory
7. Handler returns to original process, restarting faulting instruction



Whole Picture



Advanced Functionality

- OS can provide applications with some advanced functionality using virtual memory tricks
  - Shared memory
  - Copy on Write (CoW)
  - Mapped files

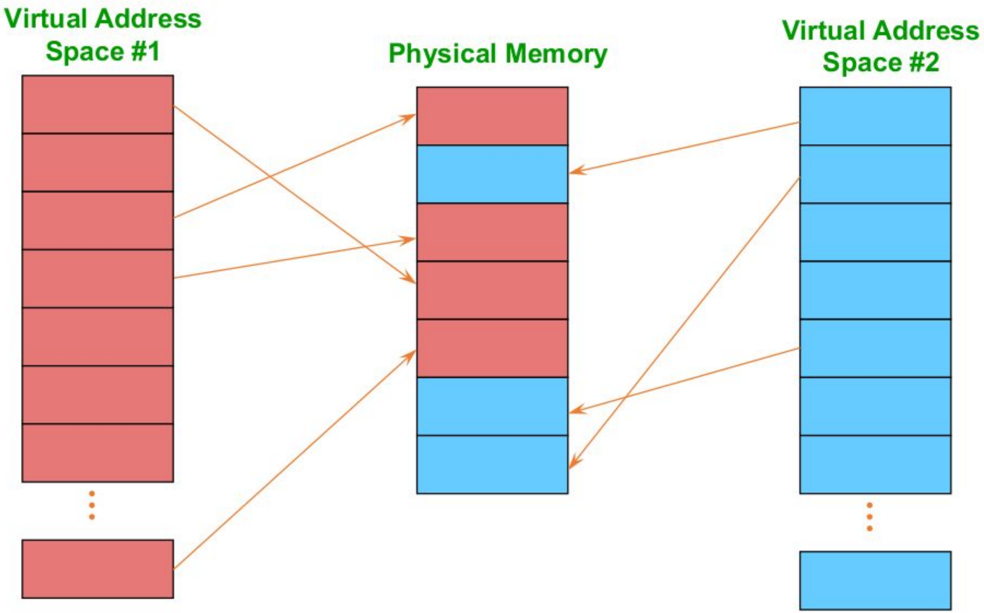
Sharing

- 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)
  - E.g., Parent and child processes in a forking Web server will want to share an in-memory data without copying
- We can use shared memory to allow processes to share data using direct memory references
  - Both processes can update the shared memory and see each other’s updates
    - What old problems we have learnt about this?

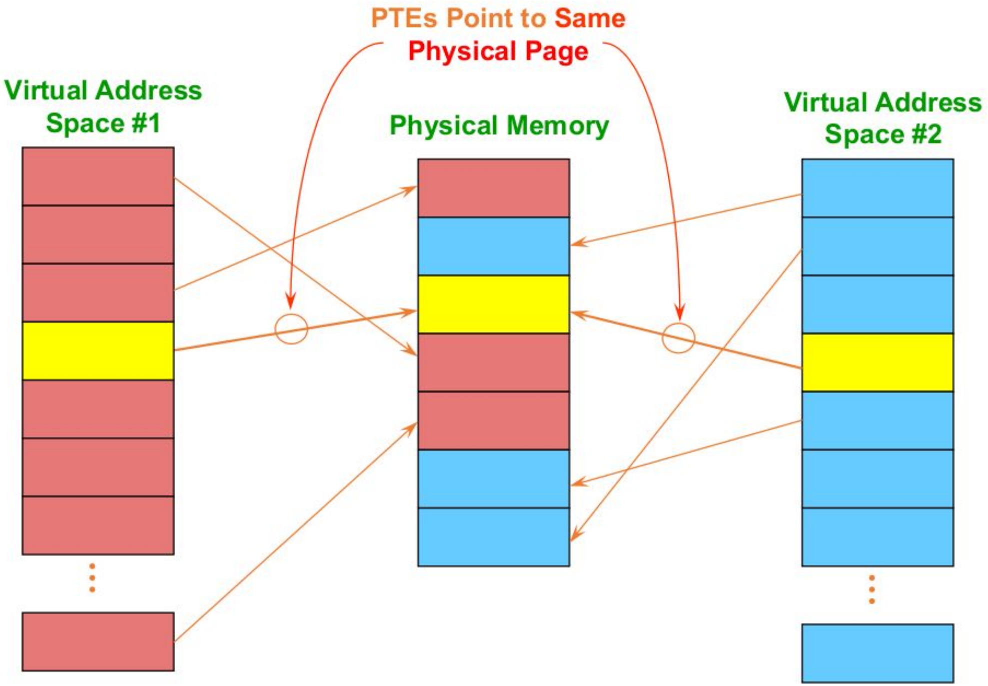
Implementation

- We can implement sharing using page tables
  - Have PTEs in both tables map to the same page frame
  - Each PTE can have different protection values
    - E.g., one can write and the other can only read
  - Must update both PTEs when page becomes invalid
  - Can map shared memory at same or different virtual addresses in each process’ address space

Isolation: No Sharing



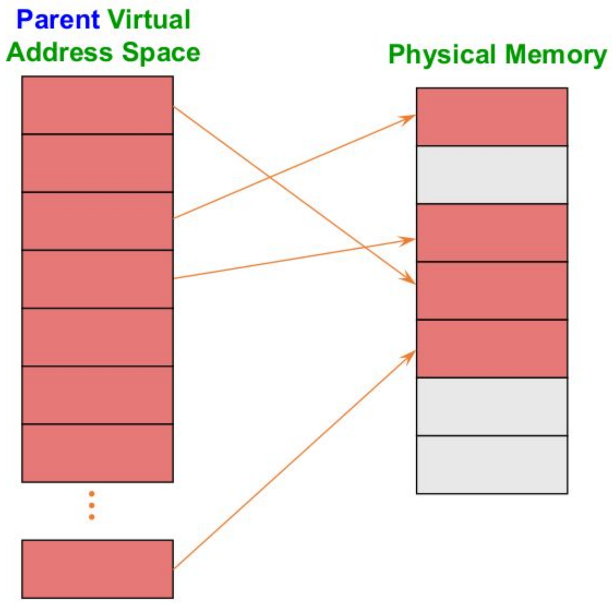
Sharing Pages



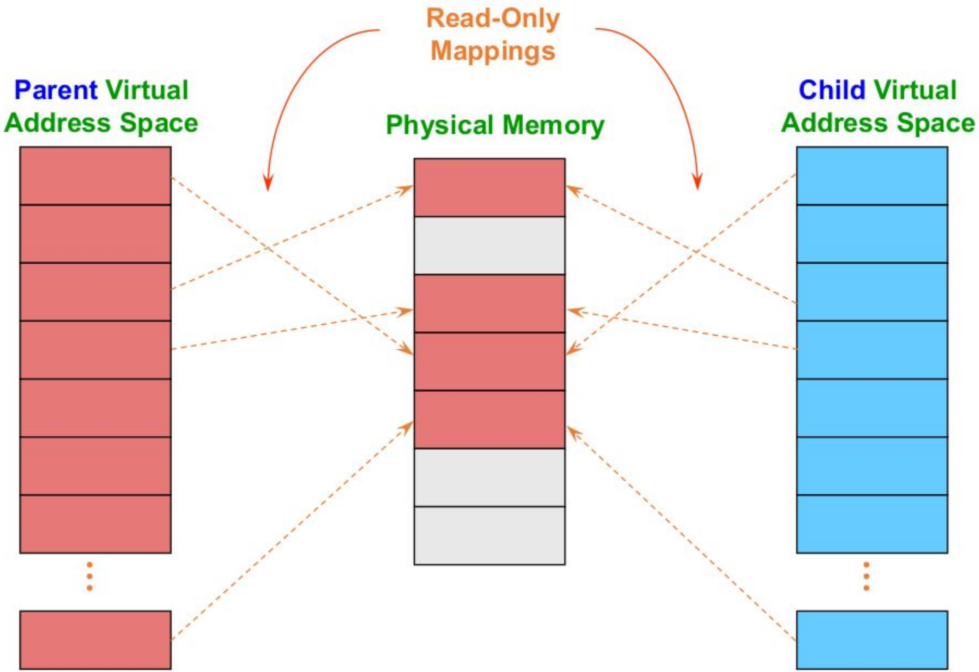
Copy on Write

- OS spends a lot of time copying data
  - E.g., the entire address spaces when calling fork()
- We can use Copy on Write (CoW) to defer 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 → OS takes over, copies page, changes page mapping in the page table, restart write instruction
- Why this is helpful in the case of fork()?

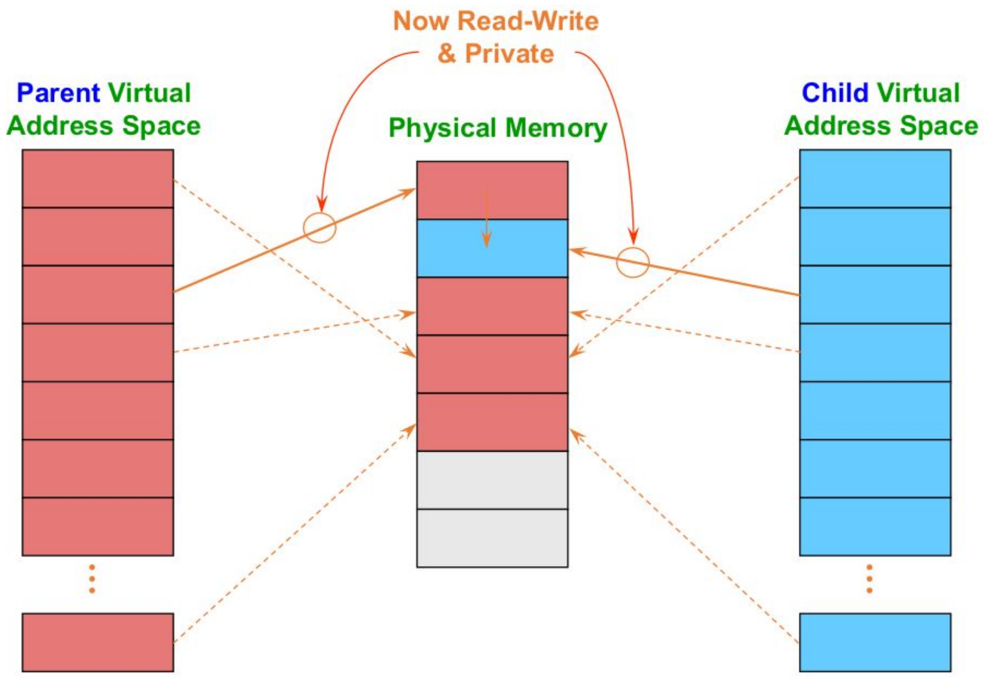
Copy on Write: Before Fork



Copy on Write: Fork



Copy on Write: On A Write



Memory Mapping

- VM areas initialized by associating them with disk objects
  - Memory mapping
- Area can be backed by (i.e., get its initial values from):
  - **Regular file** on disk (e.g., an executable object file)
    - Memory mapped files
    - Initial page bytes come from a section of a file
  - **Anonymous file** (e.g., nothing)
    - First fault will allocate a page frame full of 0's (demand-zero page)
    - Once the page is written to (dirtied), it is like any other page
- If not backed by a regular file, dirty pages are copied back and forth between memory and a special swap file

Memory Mapped Files

- Memory mapped files enable processes to do file I/O using loads and stores
  - Instead of read() and write() system calls
- Bind a file to a virtual memory region
  - Using mmap() system call in Linux
  - PTEs map virtual addresses to physical frames holding file data
  - Virtual address base + N refers to offset N in file
- Initially, all pages to which a file is mapped are invalid
  - OS reads a page from file when invalid page is accessed
  - OS writes a page to file when evicted, or region unmapped
  - If page is not dirty (has not been written to), no write needed
    - Another use of the dirty bit in PTE

mmap() System Call

- void * mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset)
    - The starting address for the new mapping is specified in addr
        - If addr is NULL, then the kernel chooses the (page-aligned) address at which to create the mapping
    - The length argument specifies the length of the mapping (which must be greater than 0)
    - The prot argument describes the desired memory protection of the mapping
    - The flags argument determines many things, e.g.,
        - MAP_SHARED: Share this mapping
        - MAP_PRIVATE: Create a private copy-on-write mapping
        - MAP_ANONYMOUS: The mapping is not backed by any file
    - The fd argument is the file descriptor
    - It starts at offset in the file

Example

```
#include <fcntl.h>
#include <stdio.h>
#include <sys/mman.h>
#include <unistd.h>

#define SIZE 10

int main()
{
  int i;
  int fd;
  char *buf;

  fd = open("test.txt",O_RDONLY);
  buf = mmap(0, SIZE, PROT_READ, MAP_PRIVATE, fd, 0);
  for (i = 0; i < SIZE; ++i)
    printf("%c", buf[i]);
  return 0;
}
```

Next Lecture

Replacement policies