CSE120
Principles of Operating Systems

Prof Yuanyuan (YY) Zhou

Advanced Memory Management

Review

- Page
  - How is different from variable size partition?
  - How is different from segmentation?
  - How is virtual address to physical address translated?
  - Why is page size always a power of 2?
  - For a 32 bit address, if the page size is 4KB, how many bits are used as the virtual page number? How many virtual pages can a process have?
  - Does an instruction use virtual address or physical address?

Review

- Page table
  - What is a page table?
  - Does each process have a page table?
  - What is stored in each entry of a page table?
- Two level page table
  - What is a two level page table?
  - Why is it introduced? For what purpose?
  - Why can it serve this purpose?
  - For a system with 4KB as the page size, the master page table fits into one page, each pointer & mapping is 4Bytes, how many entries for each secondary page table?

Review

- TLB
  - What is TLB?
  - What is it for?
  - Is it implemented in hardware or it is an OS data structure?
  - Why does it work?
  - Usually how big is it?
  - What is a TLB hit rate? Why is it important?
  - What is a TLB miss? How is it handled? What are the trade-offs between different approaches?

Agenda for this lecture

- Context switch
- Page Faults
- Put things together
  - TLB miss, page fault, etc
- Some optimizations leveraging VM
- Replacement algorithms

What happen at a context switch?

- Each process has its own virtual address from 0 to 0xFFFFFFFF (if 32 bits)
- Mapping is for EACH process
- So what happen at a context switch (from process 1 to process 2)?
  - Before the switch, TLB holds the mapping for process 1? Will Process 2 be able to use these mapping?
  - If not, what we should do?
    - TLB flushing: invalidate all TLB entries!
    - Next few memory accesses by process 2 will have to suffer from TLB misses----expensive!

Paged Virtual Memory

- We've mentioned before that pages can be moved between memory and disk
  - This process is called demand paging
- OS uses main memory as a page cache of all the data allocated by processes in the system
  - Initially, pages are allocated from memory
  - When memory fills up, allocating a page in memory requires some other page to be evicted from memory
    - Why physical memory pages are called "frames"
  - Evicted pages go to disk (where? the swap file/backing store)
  - The movement of pages between memory and disk is done by the OS, and is transparent to the application

Page Faults

- What happens when a process accesses a page that has been evicted?
  1. When it evicts a page, the OS sets the PTE as invalid and stores the location of the page in the swap file in the PTE
  2. When a process accesses the page, the invalid PTE will cause a trap (page fault)
  3. The trap will run the OS page fault handler
  4. Handler uses the invalid PTE to locate page in swap file
  5. Reads page into a physical frame, updates PTE to point to it
  6. Restarts process
- But where does it put it? Have to evict something else
  - OS usually keeps a pool of free pages around so that allocations do not always cause evictions

Address Translation Redux

- We started this topic with the high-level problem of translating virtual addresses into physical addresses
- We've covered all of the pieces
  - Virtual and physical addresses
  - Virtual pages and physical page frames
  - Page tables and page table entries (PTEs), protection
  - TLBs
  - Demand paging
- **Now let's put it together, bottom to top**

The Common Case

- Situation: Process is executing on the CPU, and it issues a read to an address
  - What kind of address is it? Virtual or physical?
- The read goes to the TLB in the MMU
  1. TLB does a lookup using the page number of the address
  2. Common case is that the page number matches, returning a page table entry (PTE) for the mapping for this address
  3. TLB validates that the PTE protection allows reads (in this example)
  4. PTE specifies which physical frame holds the page
  5. MMU combines the physical frame and offset into a physical address
  6. MMU then reads from that physical address, returns value to CPU
- Note: This is all done by the hardware

TLB Misses

- At this point, two other things can happen
  1. TLB does not have a PTE mapping this virtual address
  2. PTE exists, but memory access violates PTE protection bits
- We'll consider each in turn

Reloading the TLB

- If the TLB does not have mapping, two possibilities:
  1. MMU loads PTE from page table in memory
     - Hardware managed TLB, OS not involved in this step
     - OS has already set up the page tables so that the hardware can access it directly
  2. Trap to the OS
     - Software managed TLB, OS intervenes at this point
     - OS does lookup in page table, loads PTE into TLB
     - OS returns from exception, TLB continues
- A machine will only support one method or the other
- At this point, there is a PTE for the address in the TLB

TLB Misses (2)

Note that:

- Page table lookup (by HW or OS) can cause a recursive fault if page table is paged out

- When TLB has PTE, it restarts translation
  - Common case is that the PTE refers to a valid page in memory
    - These faults are handled quickly, just read PTE from the page table in memory and load into TLB
  - Uncommon case is that TLB faults again on PTE because of PTE protection bits (e.g., page is invalid)
    - Becomes a page fault…

Page Faults

- PTE can indicate a protection/page fault
  - Read/write/execute – operation not permitted on page
  - Invalid – virtual page not allocated, or page not in physical memory
- TLB traps to the OS (software takes over)
  - R/W/E – OS usually will send fault back up to process, or might be playing games (e.g., copy on write, mapped files)
  - Invalid
    - Virtual page not allocated in address space
      - OS sends fault to process (e.g., segmentation fault)
    - Page not in physical memory
      - OS allocates frame, reads from disk, maps PTE to physical frame

Advanced Functionality

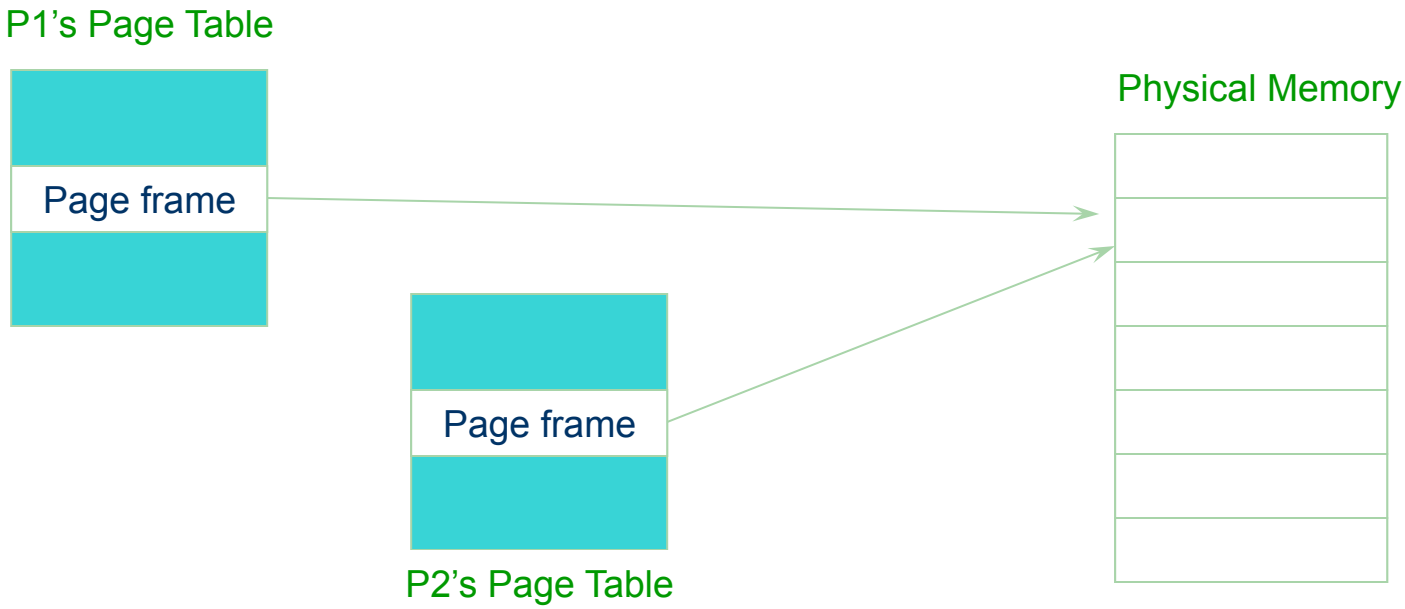
- Now we're going to look at some advanced functionality that the OS can provide applications using virtual memory tricks
  - Shared memory
  - Copy on Write
  - Mapped files

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

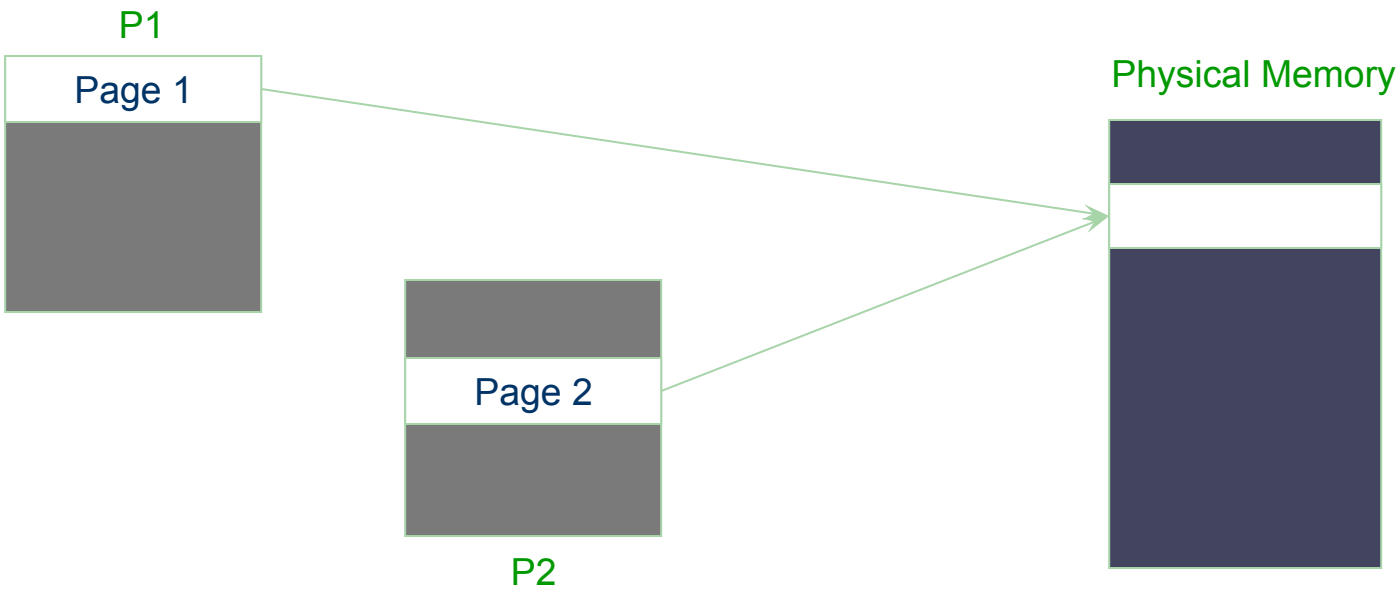
Sharing (2)

- 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?

Process perspective



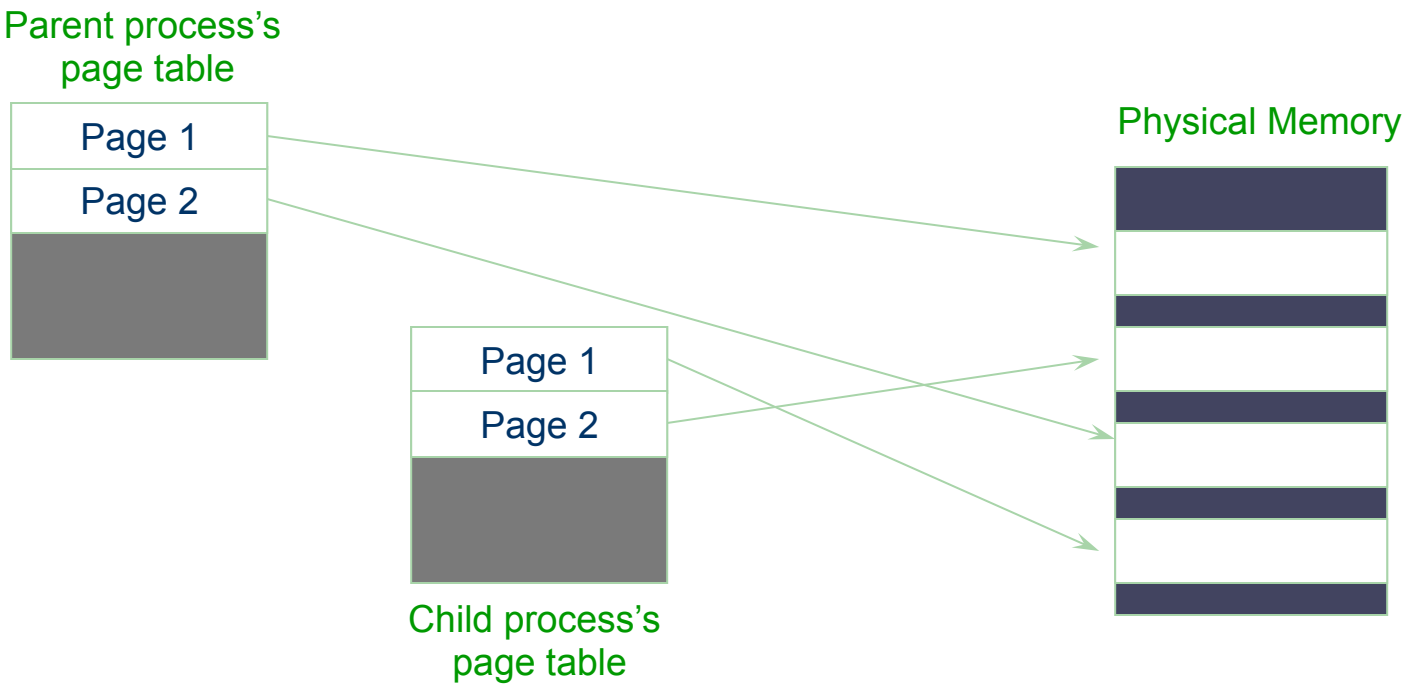
Sharing (3)

- 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

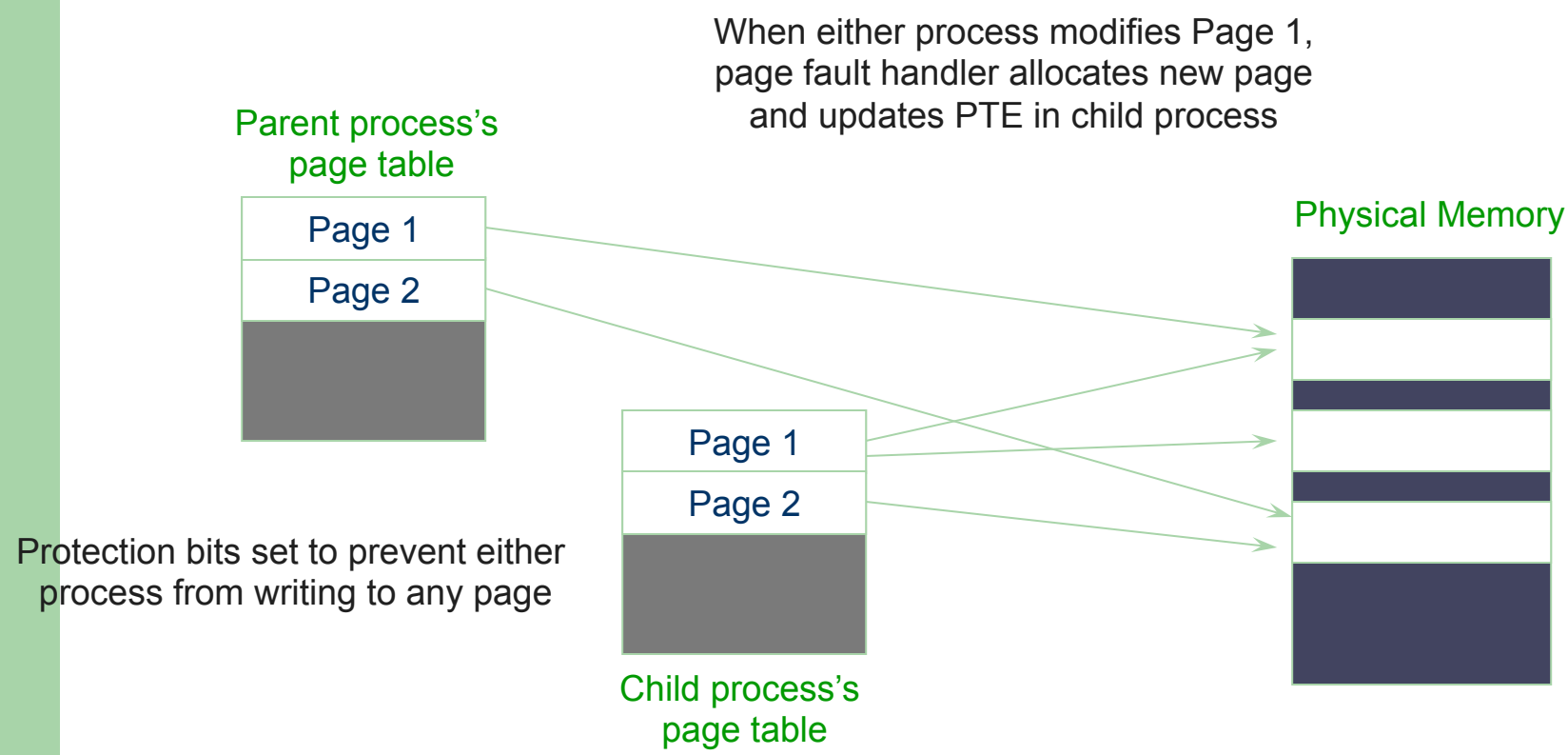
Copy on Write

- 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

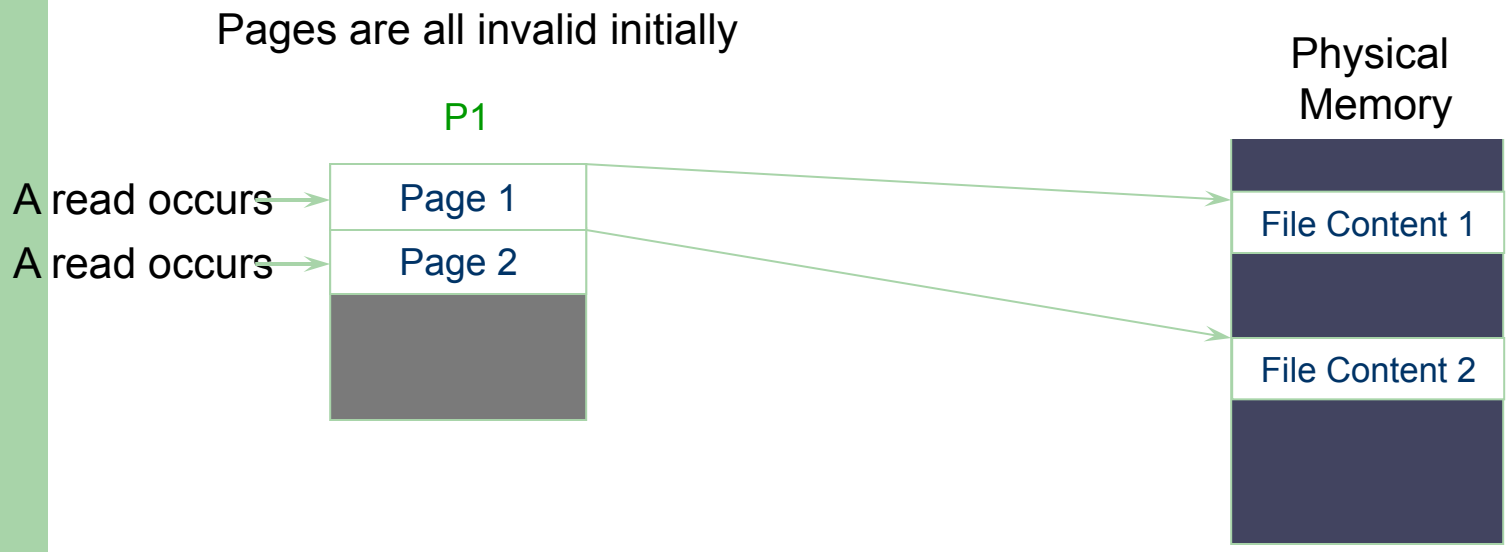


Under what circumstances such copies can be deferred forever?

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



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.)

Let's summarize before moving on

Paging mechanisms:

- Optimizations
  - Managing page tables (space)
  - Efficient translations (TLBs) (time)
  - Demand paged virtual memory (space)
- Recap address translation
- Advanced Functionality
  - Sharing memory
  - Copy on Write
  - Mapped files

Next : Paging policies