Memory Management - Demand Paging and Multi-level Page Tables

CS 416: Operating Systems Design, Spring 2011

Department of Computer Science
Rutgers University

Rutgers Sakai: 01:198:416 Sp11
(https://sakai.rutgers.edu)

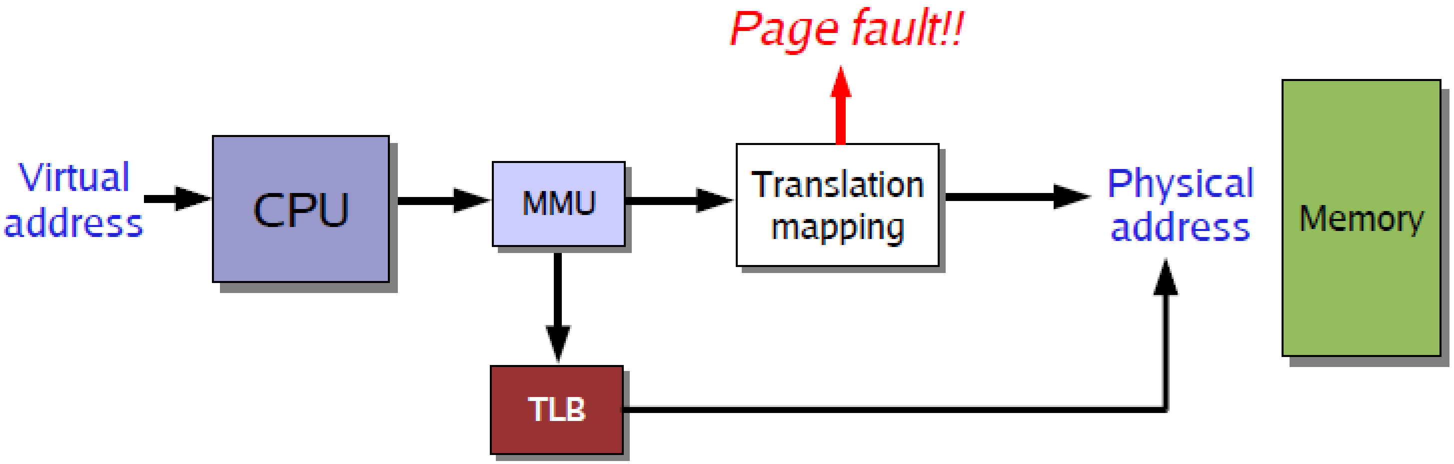
Topics for the day

- What happens when a page is not in memory ?
- How do we prevent having page tables take up a huge amount of physical memory themselves ?

Questions from last week that needs clarification

- Question 1: With 20 bits allocated to the number of pages, the number of page table entries could be : 4bytes * 2^{20}pages = 4MB
  - Is this entire 4MB space allocated contiguously in Physical Memory ?
  - (My Answer was Yes, But the answer is NO. We will see how it is held today)
- Are all processes having 4MB of Page Tables ?
  - They "could" have, but in practice, they are allocated at the time of process creation to be "size of code" + fixed size partitions. The number of page tables entries would be the "Maximum" number of pages allocated to this process.
- When there is a page fault, can a process kick out frames belonging to other process ?
  - Yes, the OS handles Page replacement. Therefore, it can access the process's page tables and mark the entry corresponding to the frame as "invalid".

Page Faults

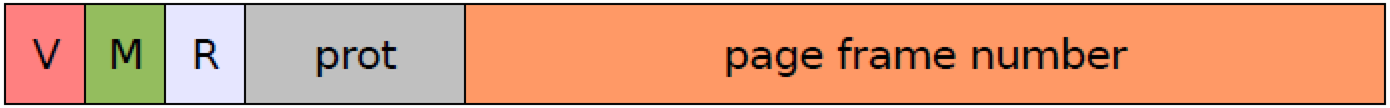


- When a virtual address translation cannot be performed, it's called a page fault

Handling Page Fault

- Trap to the OS
- Save user Register and Process State
- Check whether the page reference was legal and determine the location of the page on memory
- Issue a read from disk to a free frame
- Block for the disk operation to be complete
- On receiving “Interrupt” for disk transfer completion, save other process state
- Serve the interrupt from the disk and Fix the page table entry
- Wait for the CPU to be allocated to this process again
- Restore state and continue execution

Page Faults

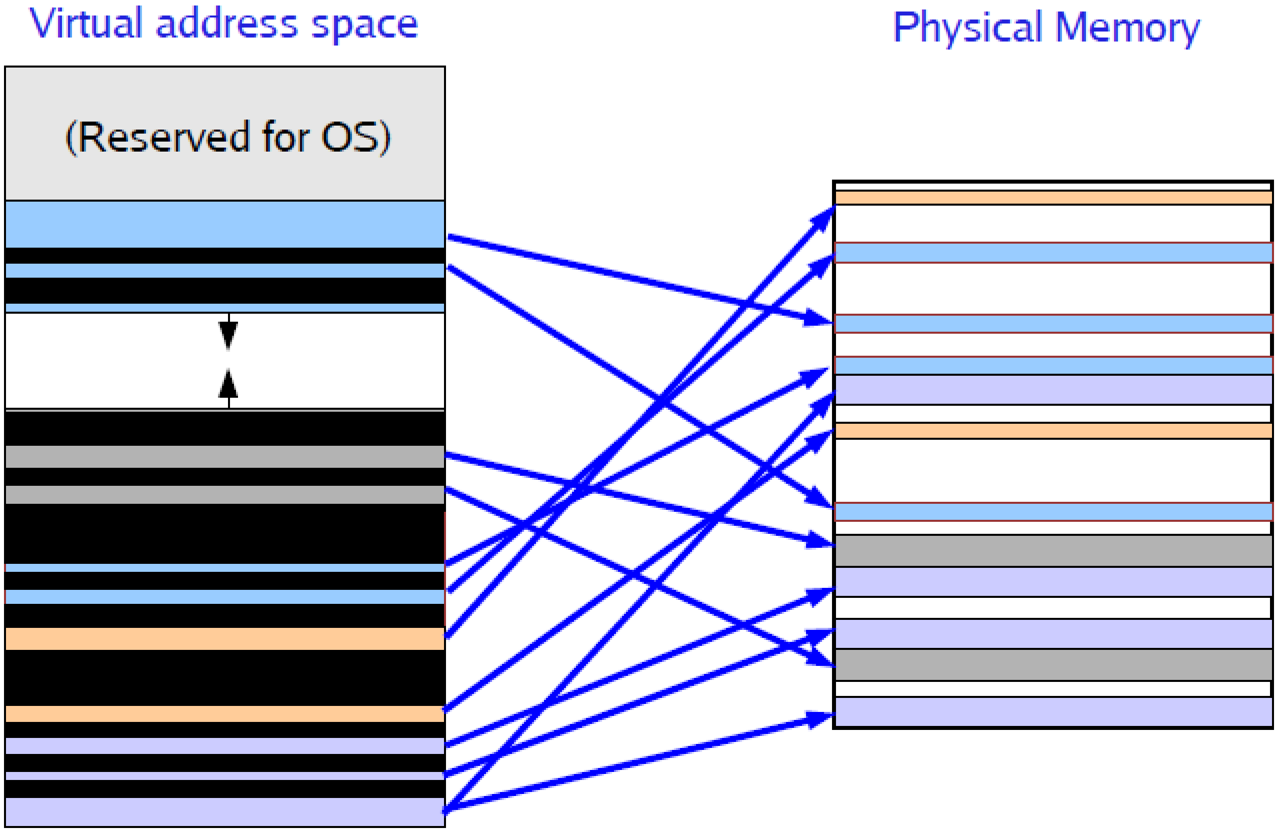


- Valid Bit indicates whether a page translation is valid
  - If Valid bit is set to 0, then a page fault will occur
- Protection Bits tells whether a page is readable, writeable, executable
  - Page fault occurs when we attempt to write a read-only page
  - This is sometimes called “Protection Fault”

Demand Paging

- Does it make sense to read an entire program into memory at once
  - No! Remember we talked about an example where some code never executes
  - For example, if you never use the “Save as PDF” function in office

What are these holes in the virtual address space mean ?



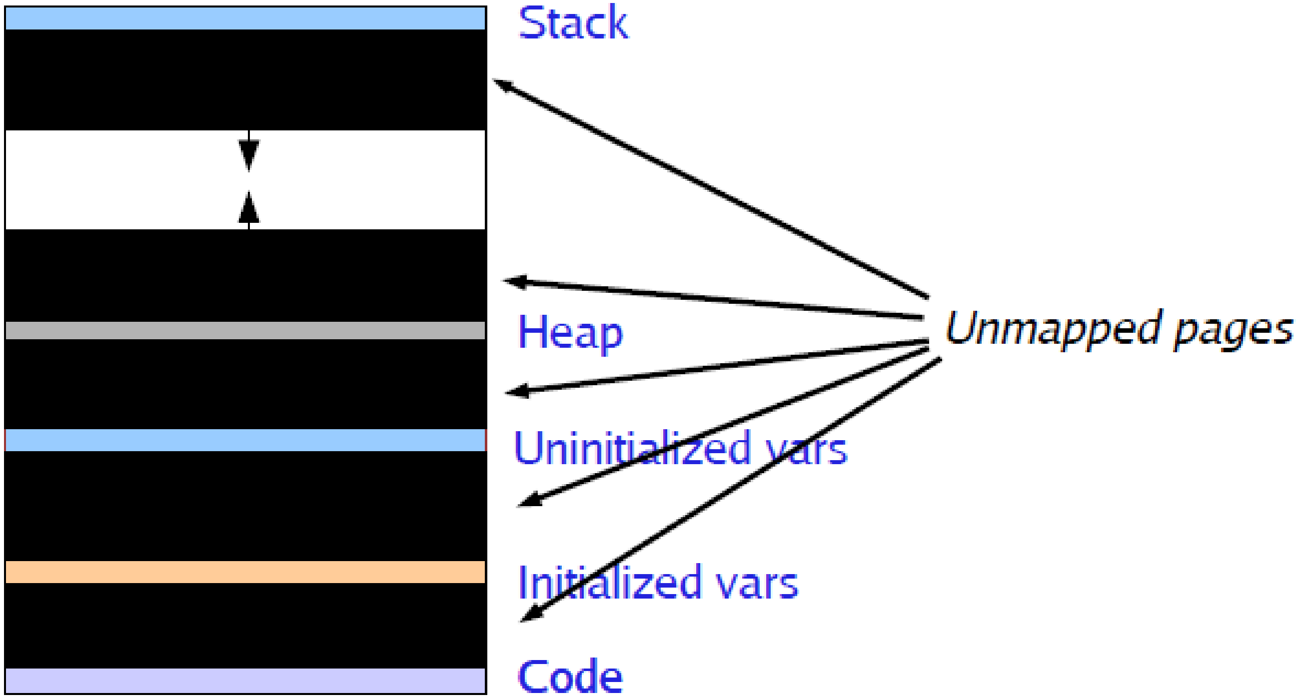
What are these holes ?

Three kinds of holes in a process's page tables:

- Pages that are on disk
  - Swapped out to disk due to lack of space in Physical Memory
    - When a page fault occurs, load the corresponding page from the disk
- Pages that have not been accessed yet
  - For Example, newly allocated memory
    - When a page fault occurs, allocate a new physical page
- Pages that are invalid
  - For example, the NULL POINTER always points to page at address 0x0
    - When a NULL address is accessed, we get segmentation fault !
    - Trying to access 0x0 creates page fault, and the OS kills the offending process

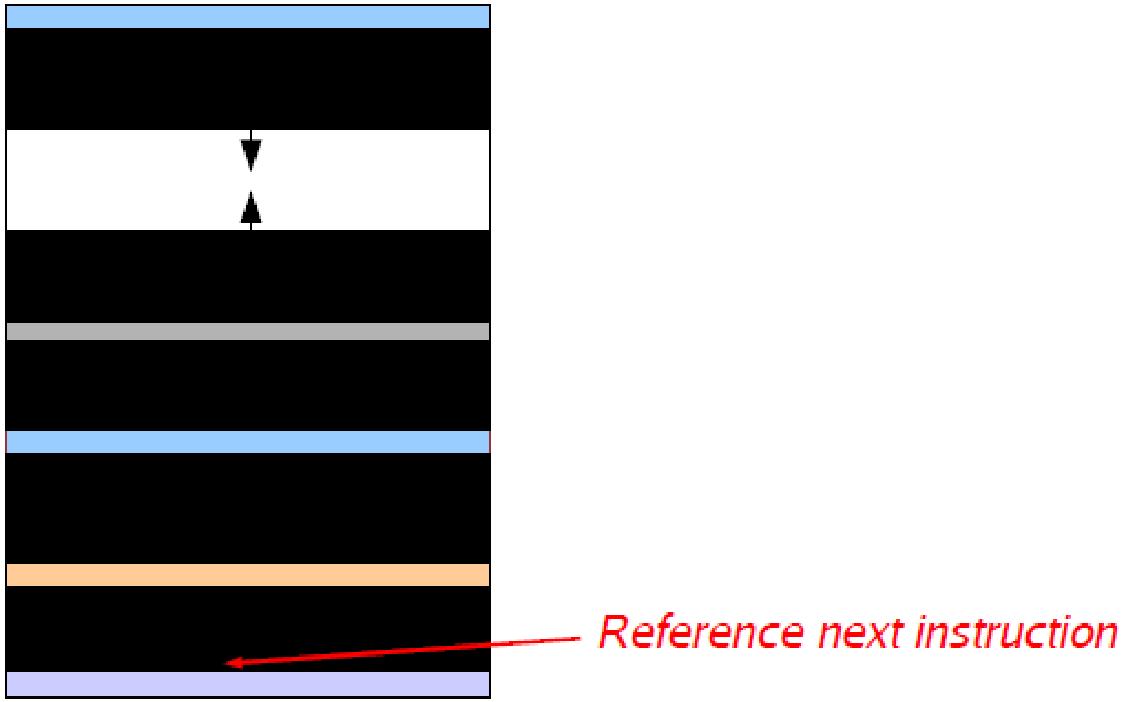
Starting up a process

- What does a process address space looks like when it starts ?



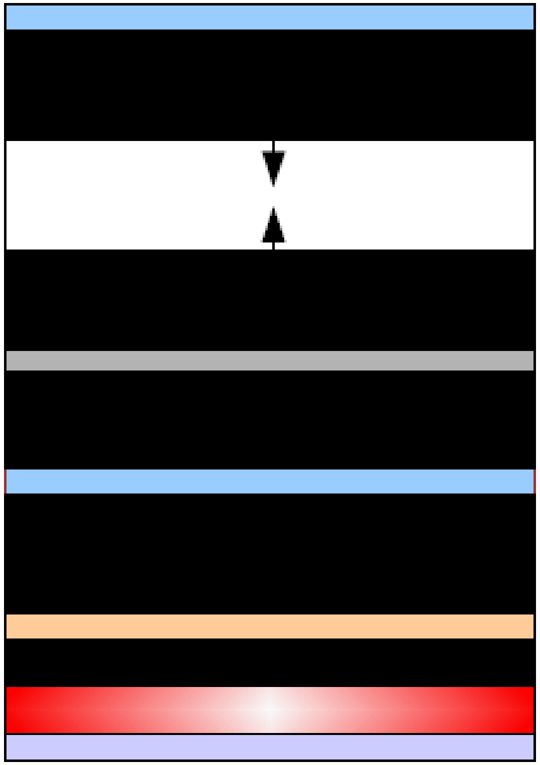
Starting up a process

- What does the process's address space look like when it first starts up



Starting up a process

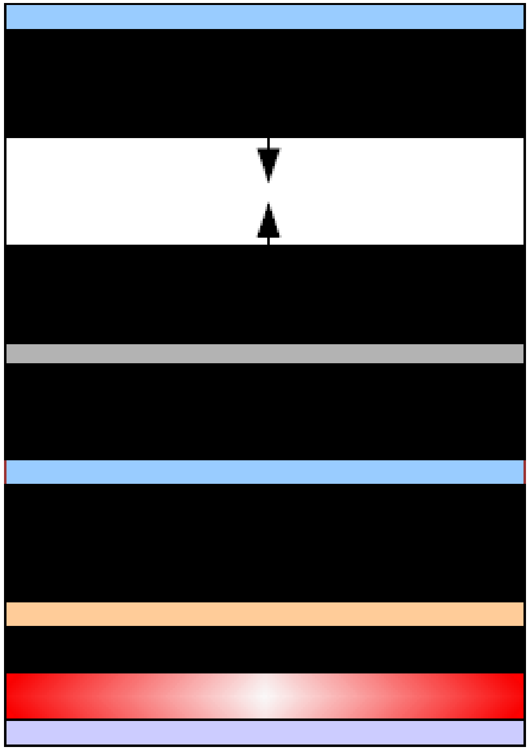
- What does the process's address space look like when it first starts up

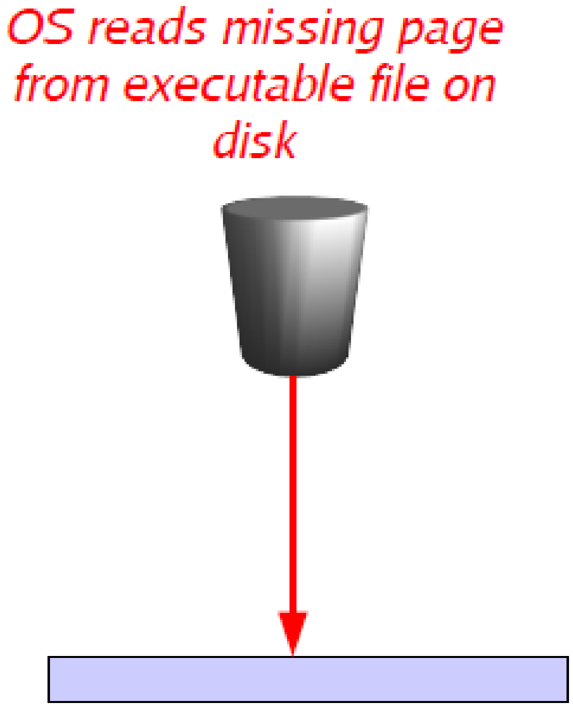


Page fault!!!

Starting up a process

- What does the process's address space look like when it first starts up





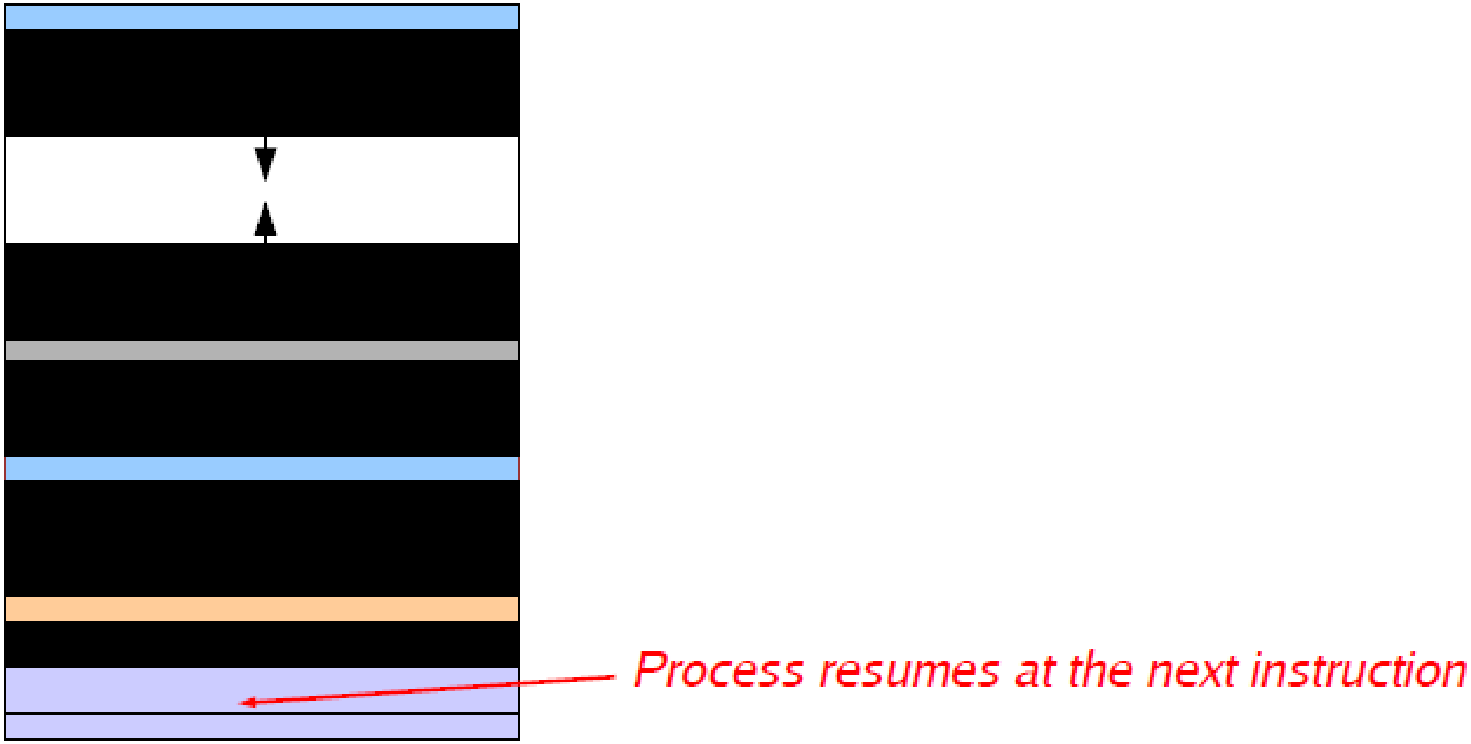
Starting up a process

- What does the process’s address space look like when it first starts up

OS adds page to process's page table

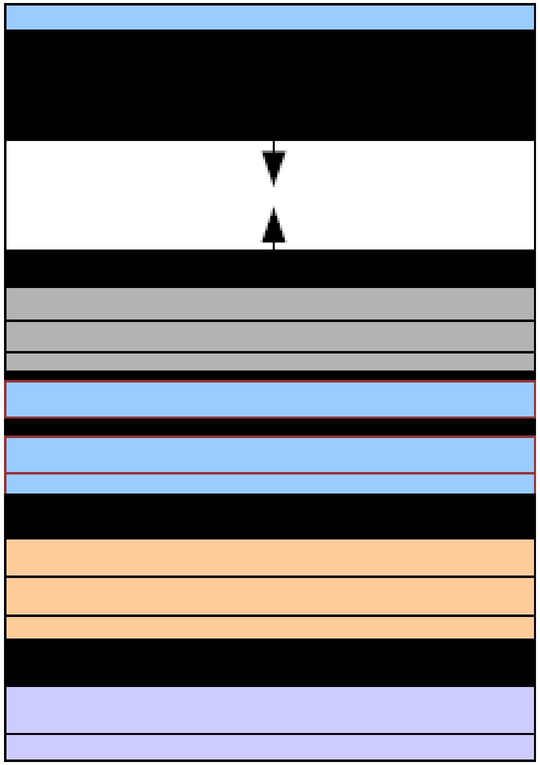
Starting up a process

- What does the process’s address space look like when it first starts up



Starting up a process

- What does the process’s address space look like when it first starts up



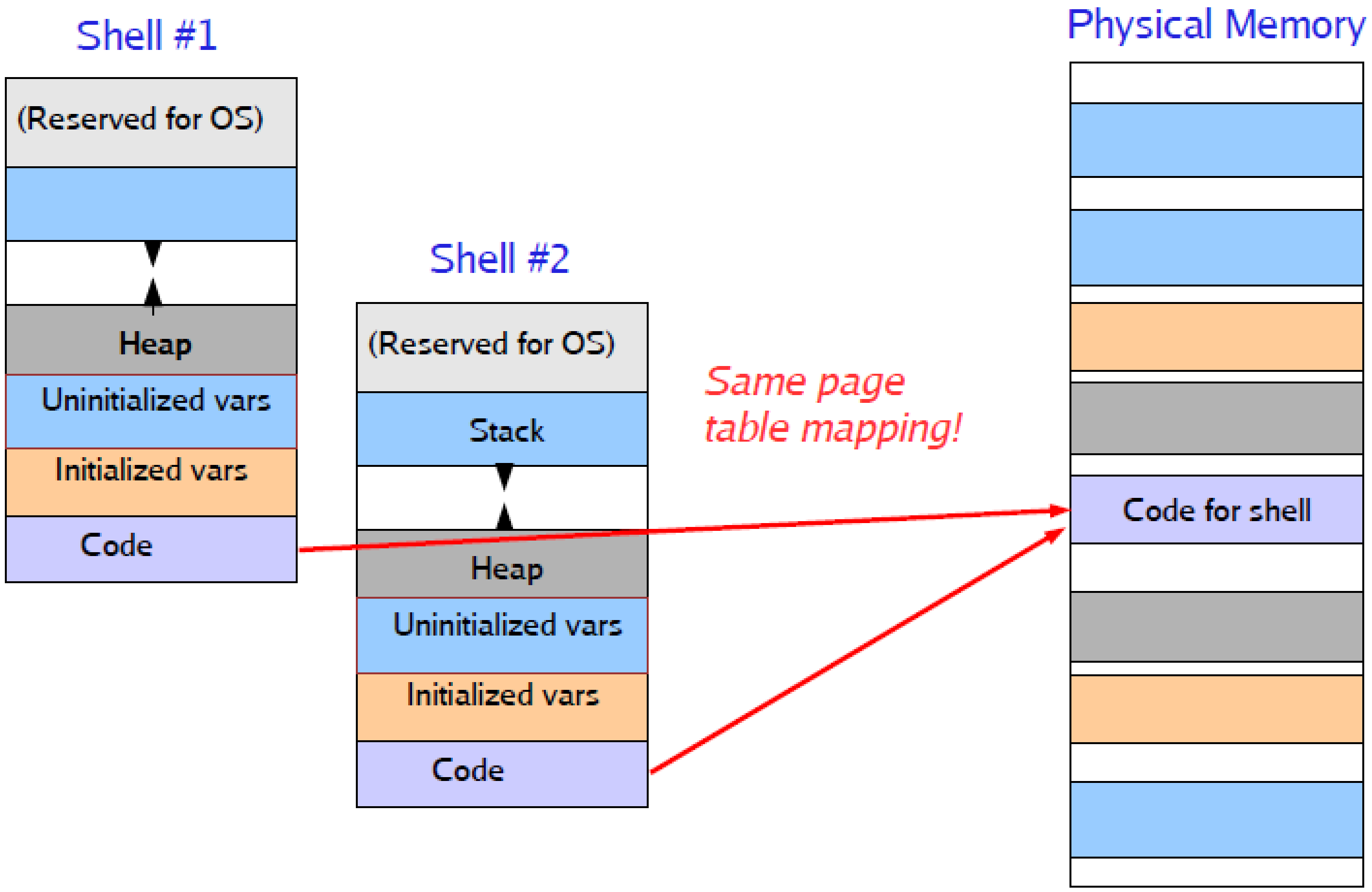
Over time, more pages are brought in from the executable as needed

Uninitialized Variables and the heap

- Page faults bring in pages from the executable file for:
  - Code (text segment) pages, Initialized variables
- What about Un-initialized variables and the heap ?
- Say I have a global variable “int c” in the program ..What happens when the process first accesses it ?
  - Page fault occurs
  - OS looks at the page and realizes that it corresponds to a Zero Page
  - Allocates a new frame in the main memory and **sets all bytes to ZERO**
  - Maps the frame into the address space
- What about the heap ?
  - malloc() **just maps** new zero pages into the address space
  - Brings in new empty pages into the frame only when page fault occurs

More Demand Paging Tricks

- Paging can be used by processes to share memory
  - A significant portion of many process's address space is identical



More Demand Paging Tricks

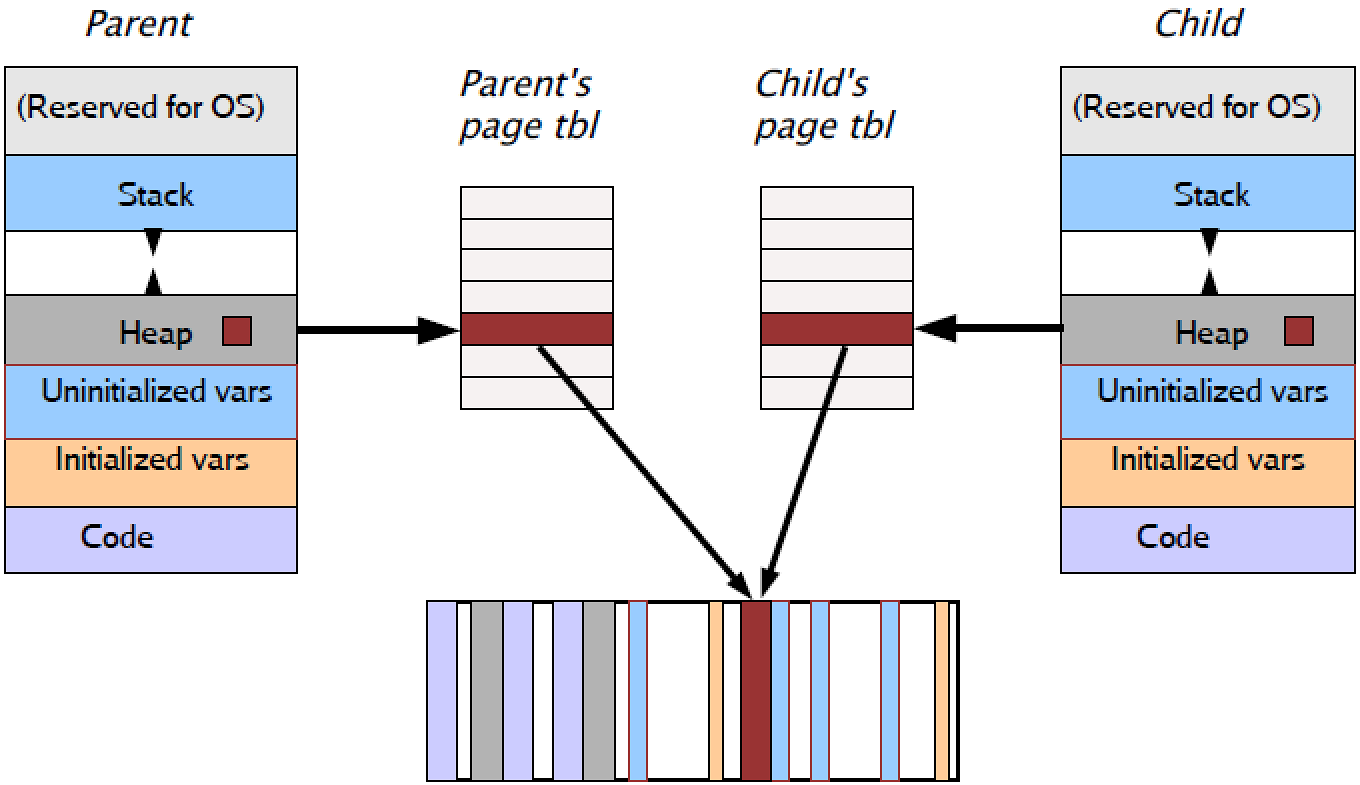
- This can be used to let different processes share memory
  - UNIX supports shared memory through the shmget/shmat/shmdt system calls
  - Allocates a region of memory that is shared across multiple processes
  - Some of the benefits of multiple threads per process, but the rest of the processes address space is protected
    - *Why not just use multiple processes with shared memory regions?*
- Memory-mapped files
  - Idea: Make a file on disk look like a block of memory
  - Works just like faulting in pages from executable files
  - *In fact, many OS's use the same code for both*
  - One wrinkle: Writes to the memory region must be reflected in the file
  - **How does this work?**
    - **When writing to the page, mark the “modified” bit in the PTE**
    - **When page is removed from memory, write back to original file**

Remember fork()

- fork() creates an exact copy of a process
  - What does this imply about page tables?
- When we fork a new process, does it make sense to make a copy of all of its memory?
  - Why or why not?
- What if the child process doesn't end up touching most of the memory the parent was using?
  - What happens if a process does an exec() immediately after fork()?

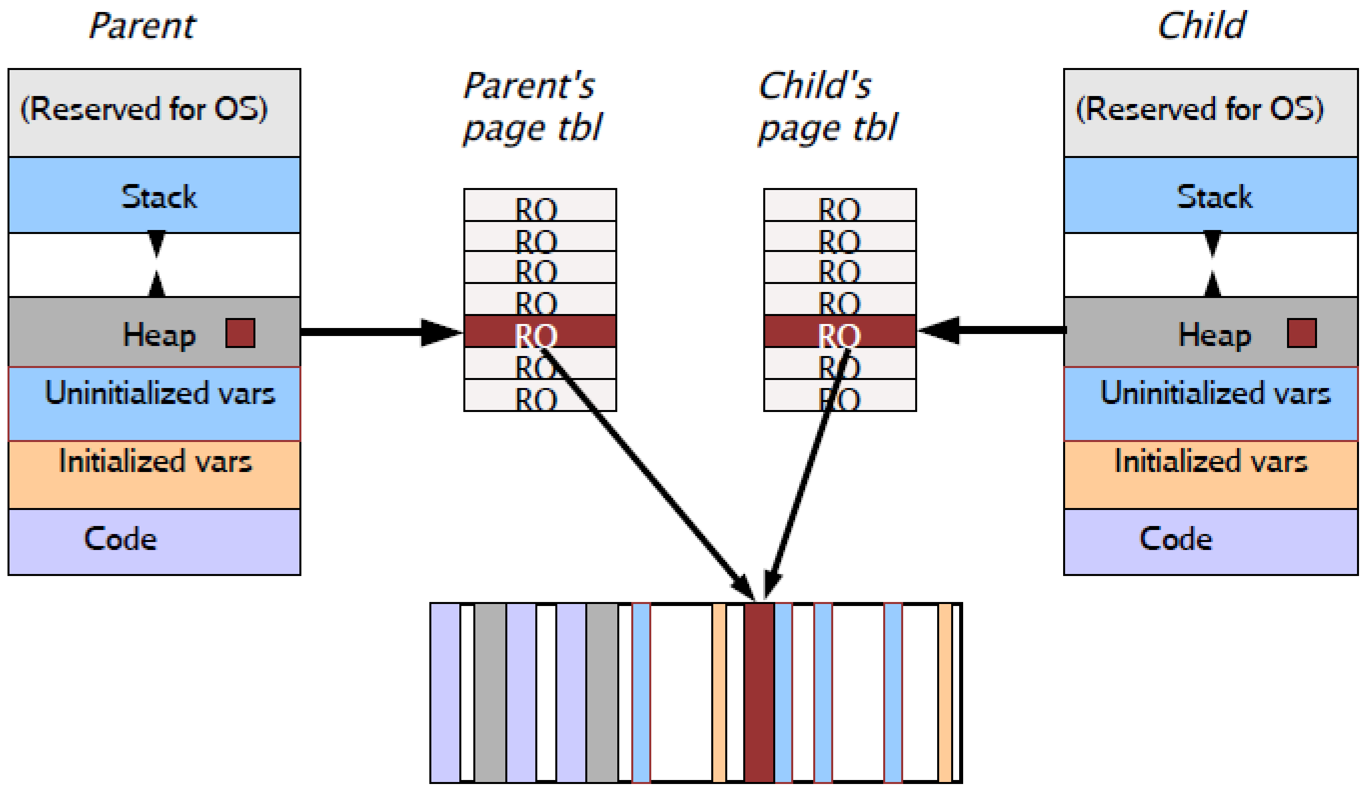
Copy on Write

- Share the pages among parent and child, but don't let the child write to any pages directly
  - Parents forks a child, Child gets a copy of the parent's page tables.



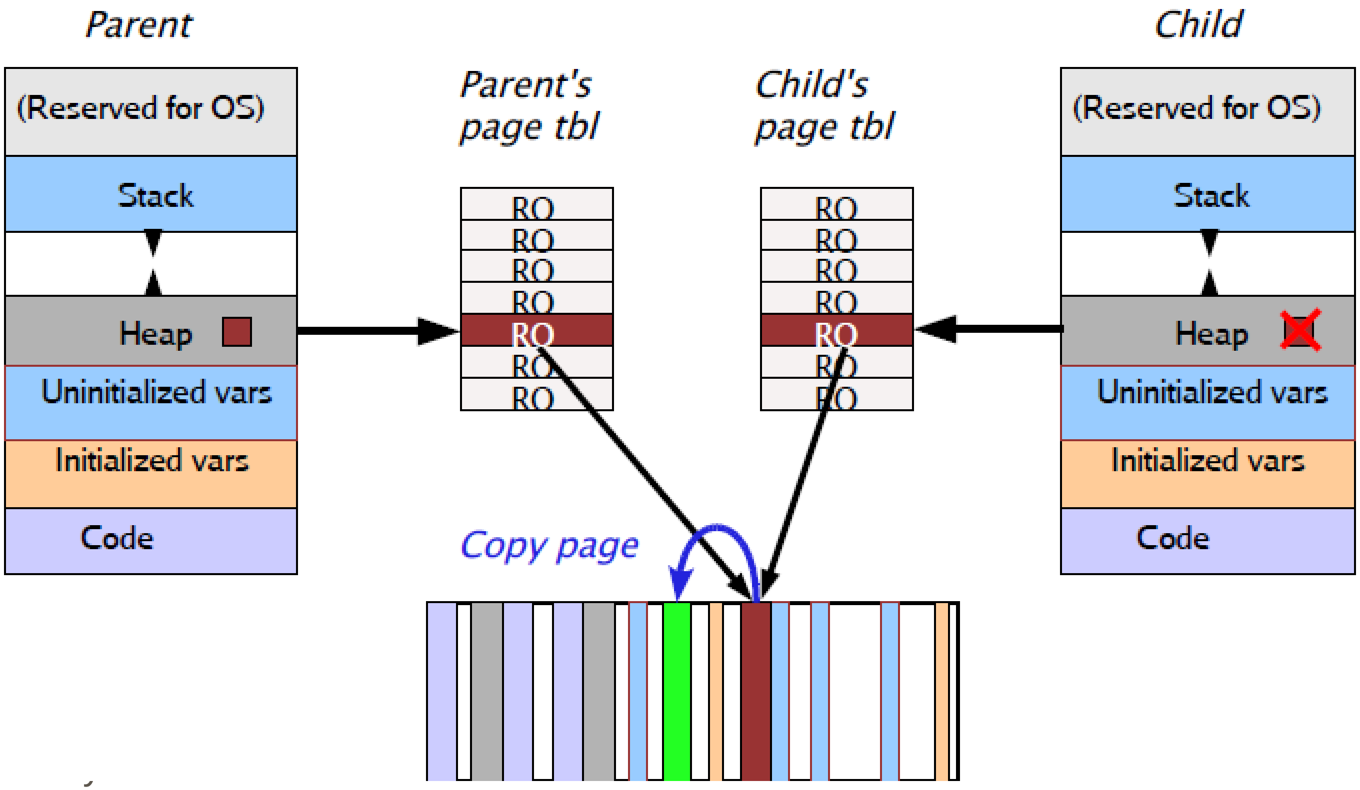
Copy on Write

- All Pages (both parent and child) marked read-only
  - Why ?

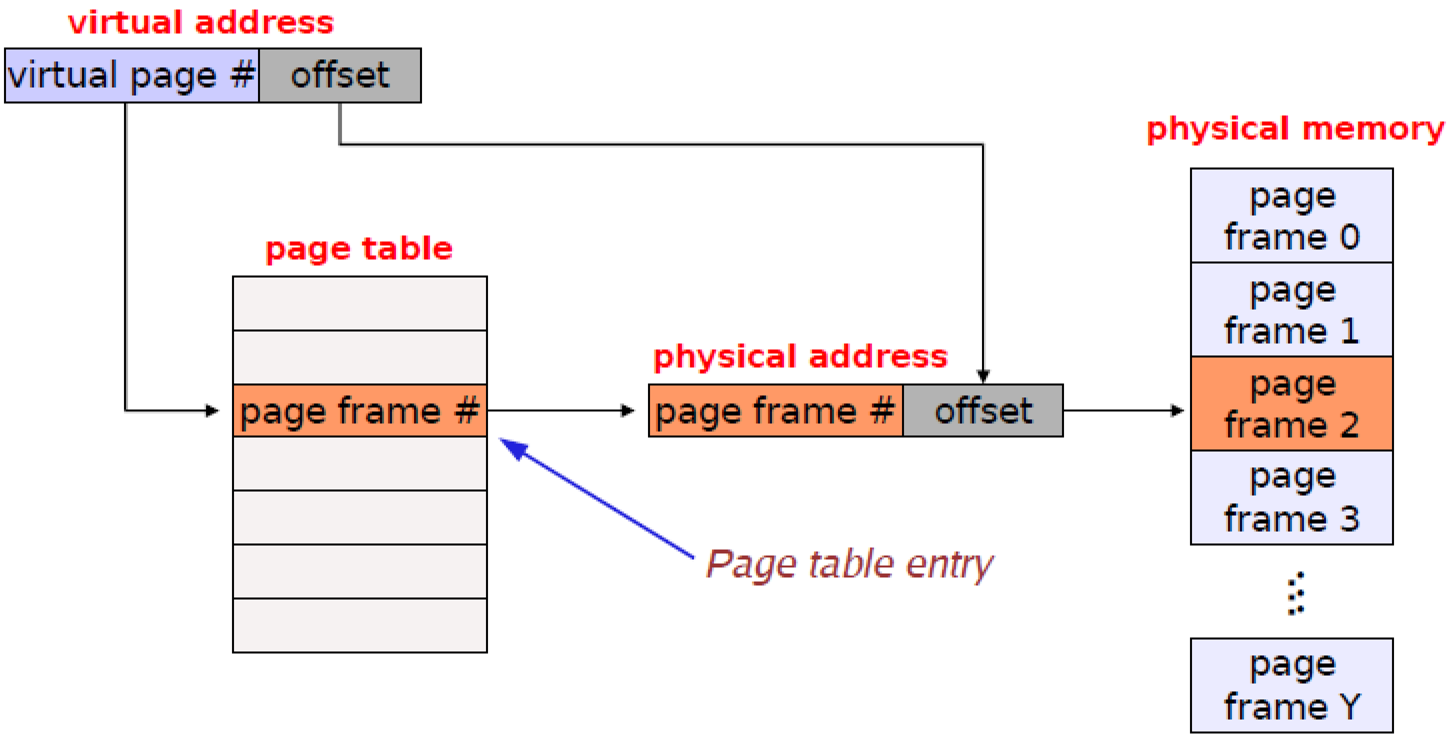


Copy on Write

- What happens when the child “writes” the pages
  - Protection fault occurs
  - OS copies the page and maps it R/W into child’s address space



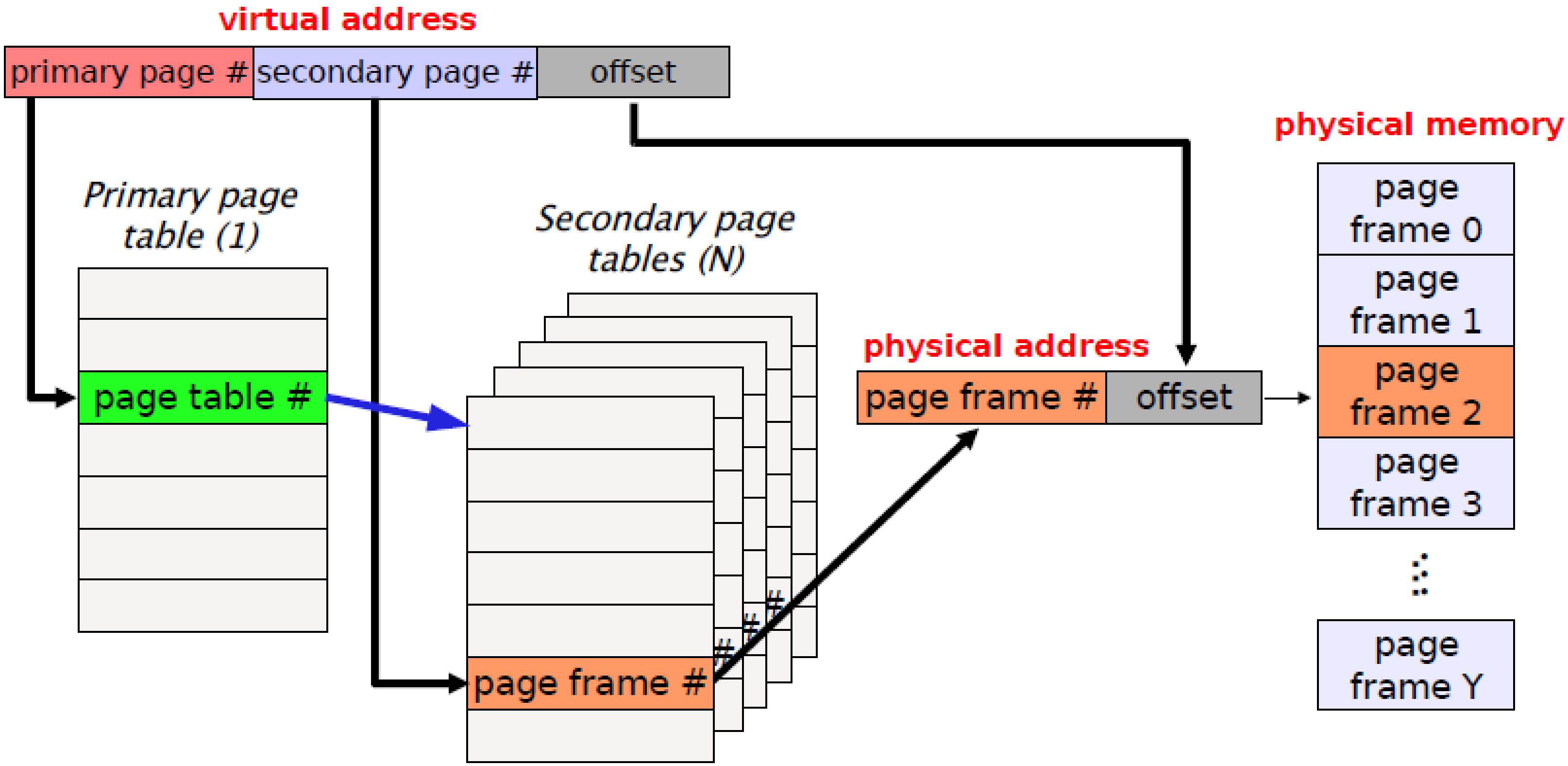
Page Tables



- Recall that page tables for every process could be as large as 4MB

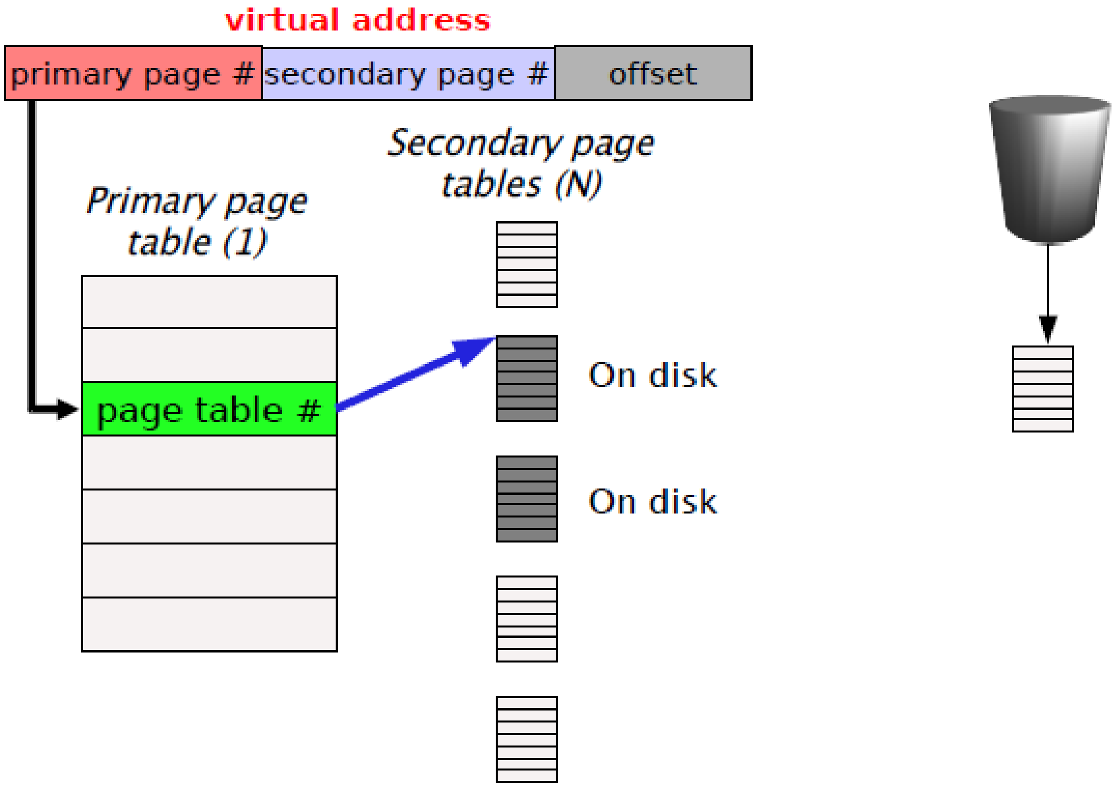
Multi-Level Page Tables

- Can't hold all of the page tables in memory
- Solution: Page the page tables
  - Allow portions of the page tables to be kept in memory at one time



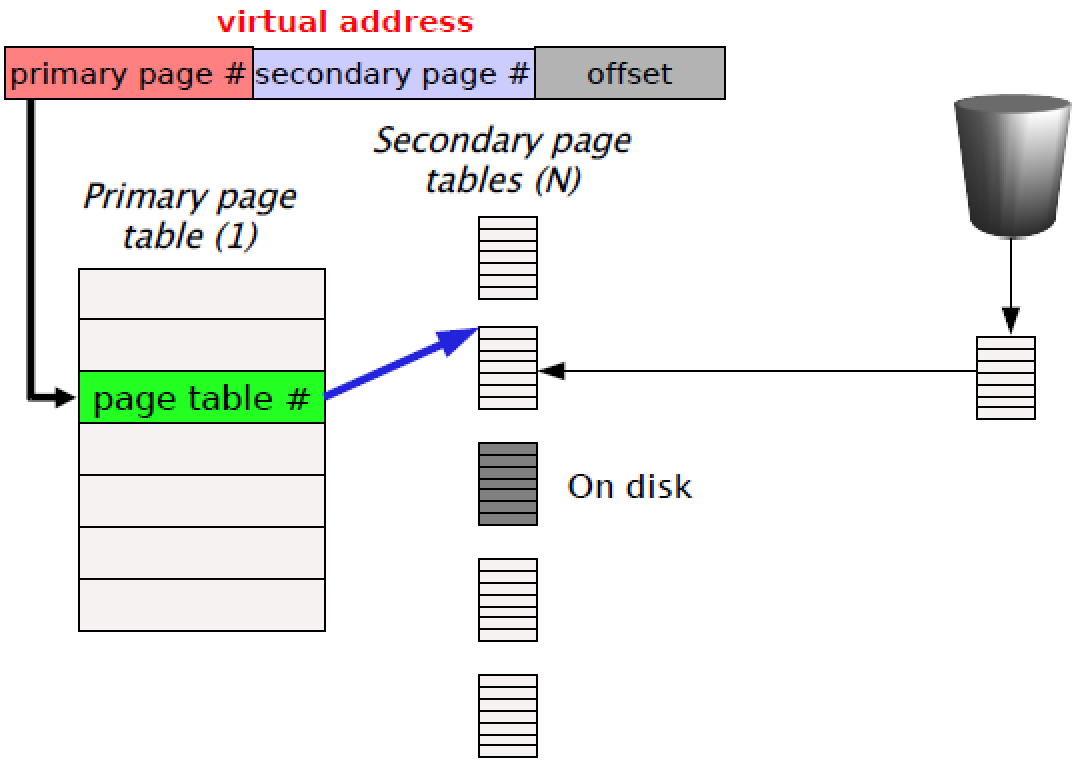
Multi-Level Page Tables

- Can’t hold all of the page tables in memory
- Solution: Page the page tables
  - Allow portions of the page tables to be kept in memory at one time



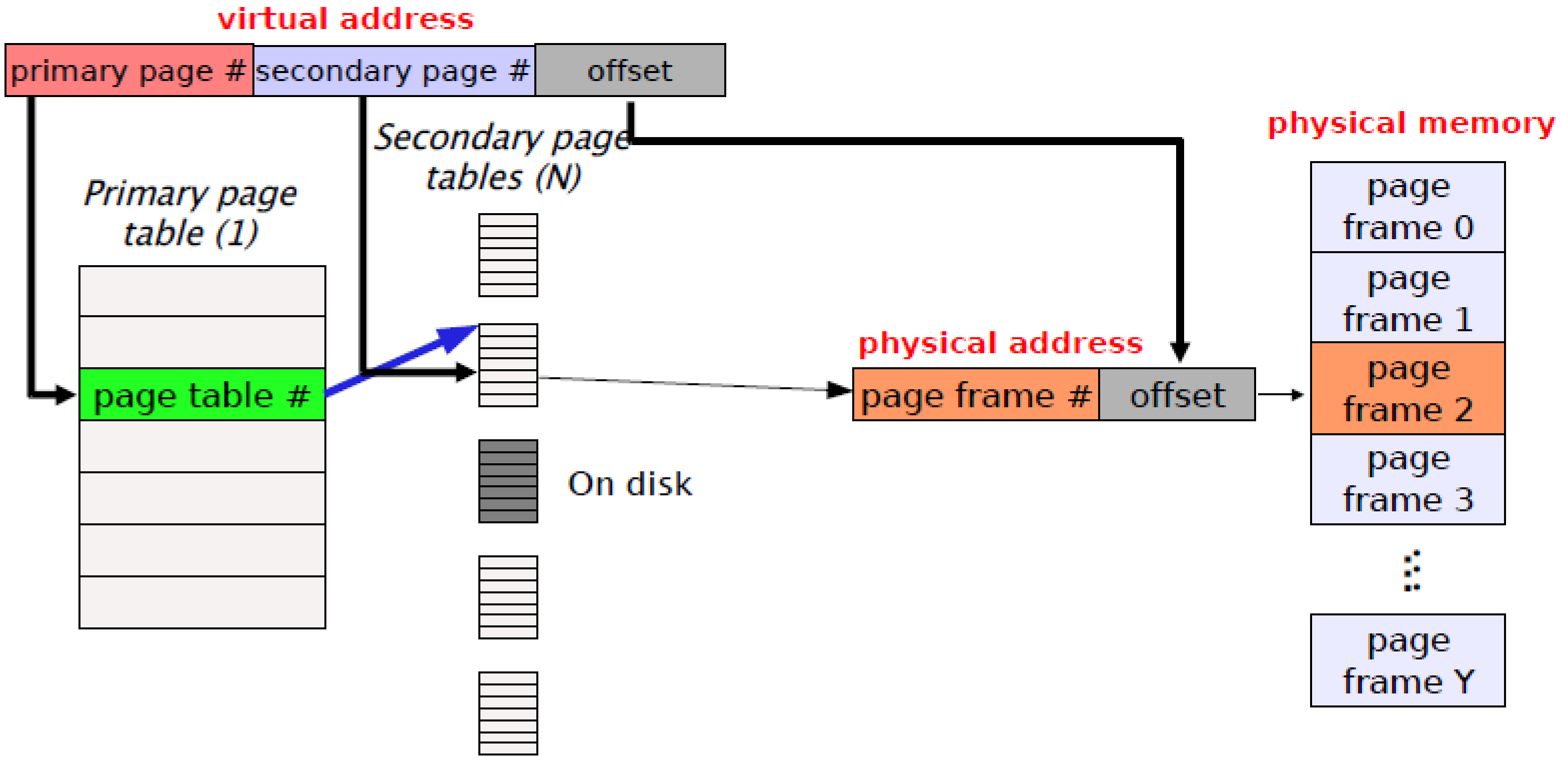
Multi-Level Page Tables

- Can’t hold all of the page tables in memory
- Solution: Page the page tables
  - Allow portions of the page tables to be kept in memory at one time



Multi-Level Page Tables

- Can’t hold all of the page tables in memory
- Solution: Page the page tables
  - Allow portions of the page tables to be kept in memory at one time



Multilevel page tables

- With two levels of page tables, how big is each table?
  - Say we allocate 10 bits to the primary page, 10 bits to the secondary page, 12 bits to the page offset
  - Primary page table is then 2^10 * 4 bytes per PTE == 4 KB
  - Secondary page table is also 4 KB
    - *Hey ... that's exactly the size of a page on most systems ... cool*
- What happens on a page fault?
  - MMU looks up index in primary page table to get secondary page table
    - *Assume this is "wired" to physical memory*
  - MMU tries to access secondary page table
    - *May result in another page fault to load the secondary table!*
  - MMU looks up index in secondary page table to get PFN
  - CPU can then access physical memory address
- Issues
  - Page translation has very high overhead
    - *Up to three memory accesses plus two disk I/Os!!*
  - TLB usage is clearly very important.

Multilevel Paging and Performance

- Since each level is stored as a separate table in memory, covering a logical address to a physical one may take **three** memory accesses.
  - CPU Generates an address
  - Use the first 10 bits to read a memory location (outer page table) – first access
  - Use the first page table to locate the frame for second page table – Second access
  - Get the (frame number + offset) and read the actual memory location. – Third access

Average page fault service time = 8ms

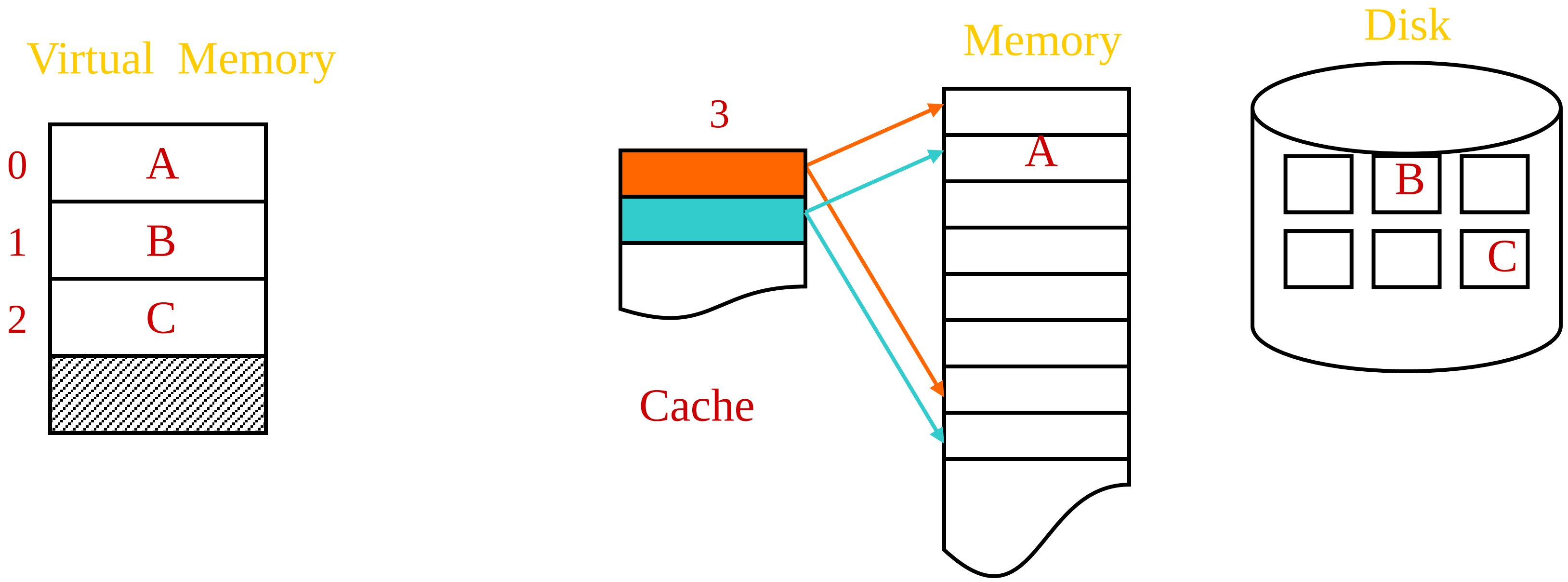
Average memory access time = 200ns

Let the probability of page fault be ‘p’

Effective Access time per memory access : (1-p)*200ns + p*8ms

Where does caching fit here ?

- After the Physical Address is Identified, the CPU first check the Cache to see if the entire block is already in cache !
- If yes, accesses are much faster
- If not, the block is transferred to the cache.



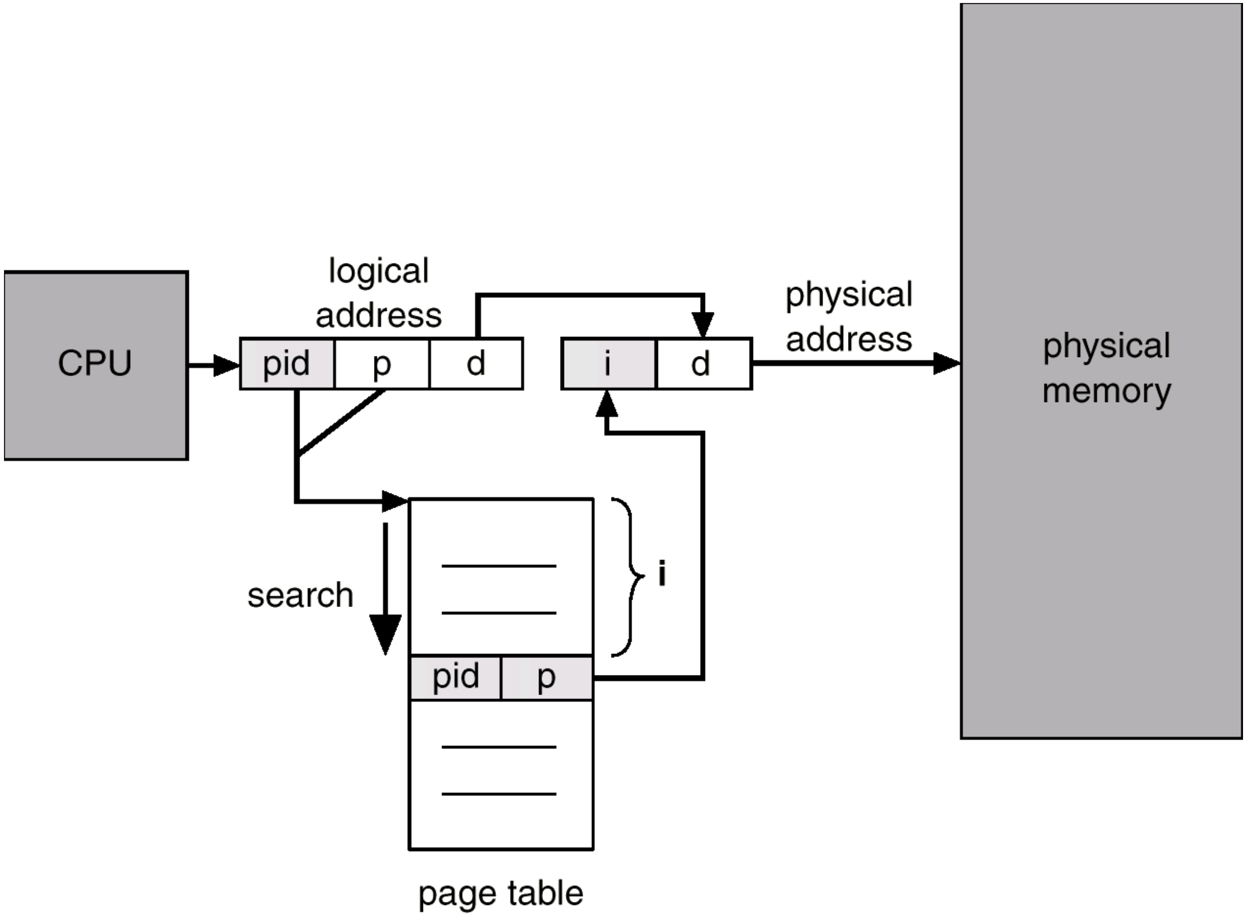
Inverted Page Table

- One entry for each real frame of memory.

- Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page.

- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs.

Inverted Page Table Architecture



Next Lecture

- Page Replacement Algorithms !