CS161: Operating Systems

Matt Welsh
mdw@eecs.harvard.edu

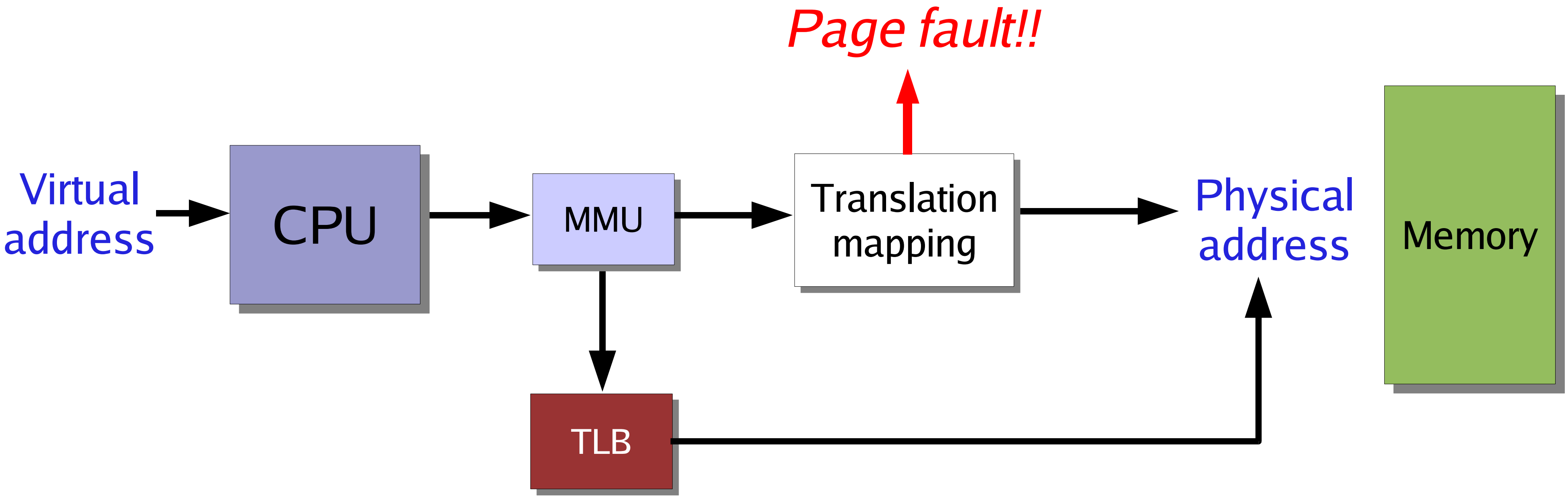
Lecture 10: Demand Paging and Multi-level Page Tables
March 8, 2007

Topics for today

What happens when a page is not in memory?

How do we prevent having page tables take up a huge amount of memory themselves?

Page Faults



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

Page Faults

Recall the PTE format:

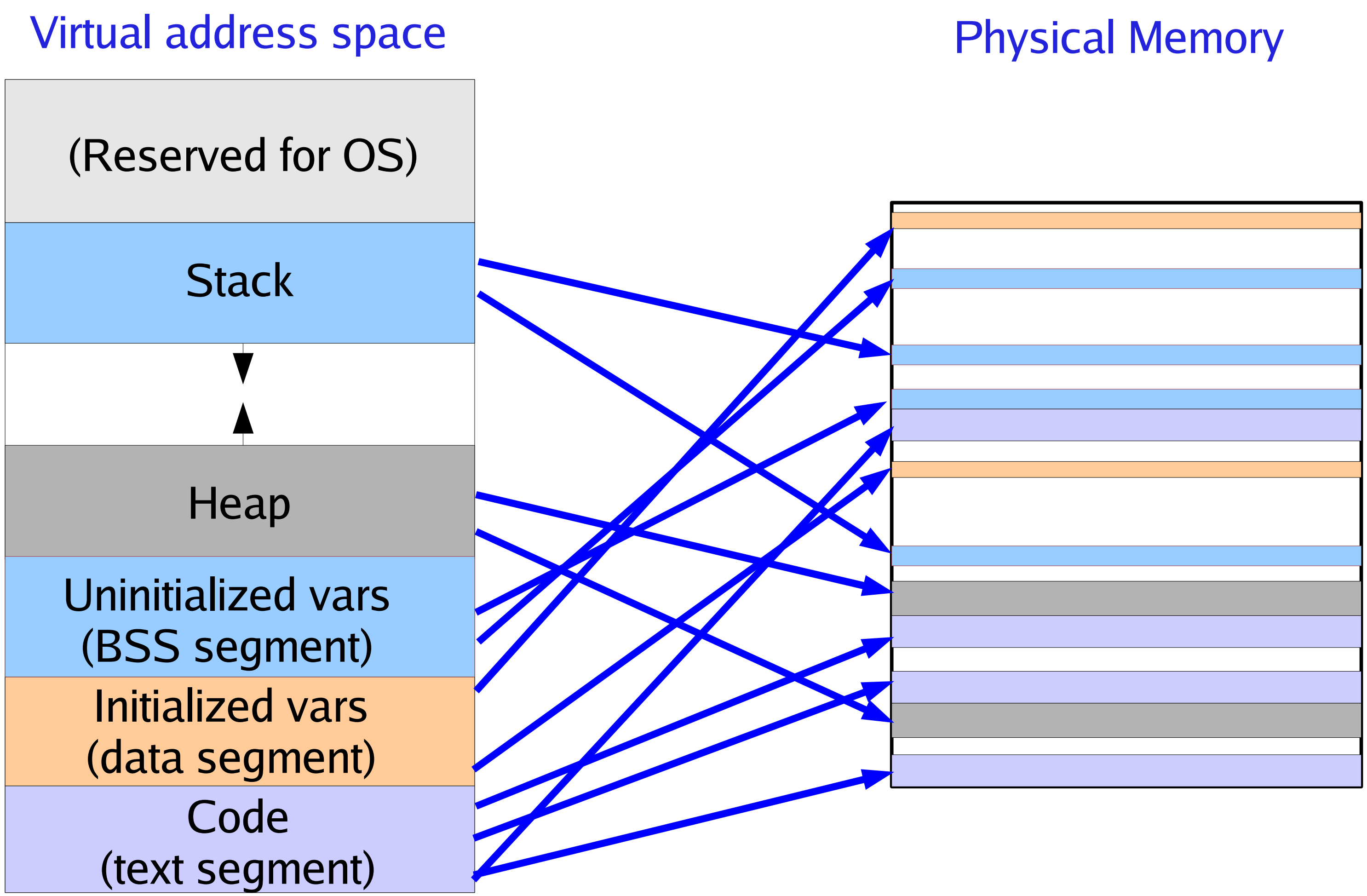
V	M	R	prot	page frame number

- Valid bit indicates whether a page translation is valid
- If Valid bit is 0, then a page fault will occur
- Page fault will also occur if attempt to write a read-only page (based on the Protection bits, not the valid bit)
    - *This is sometimes called a protection fault*

Demand Paging

Does it make sense to read an entire program into memory at once?

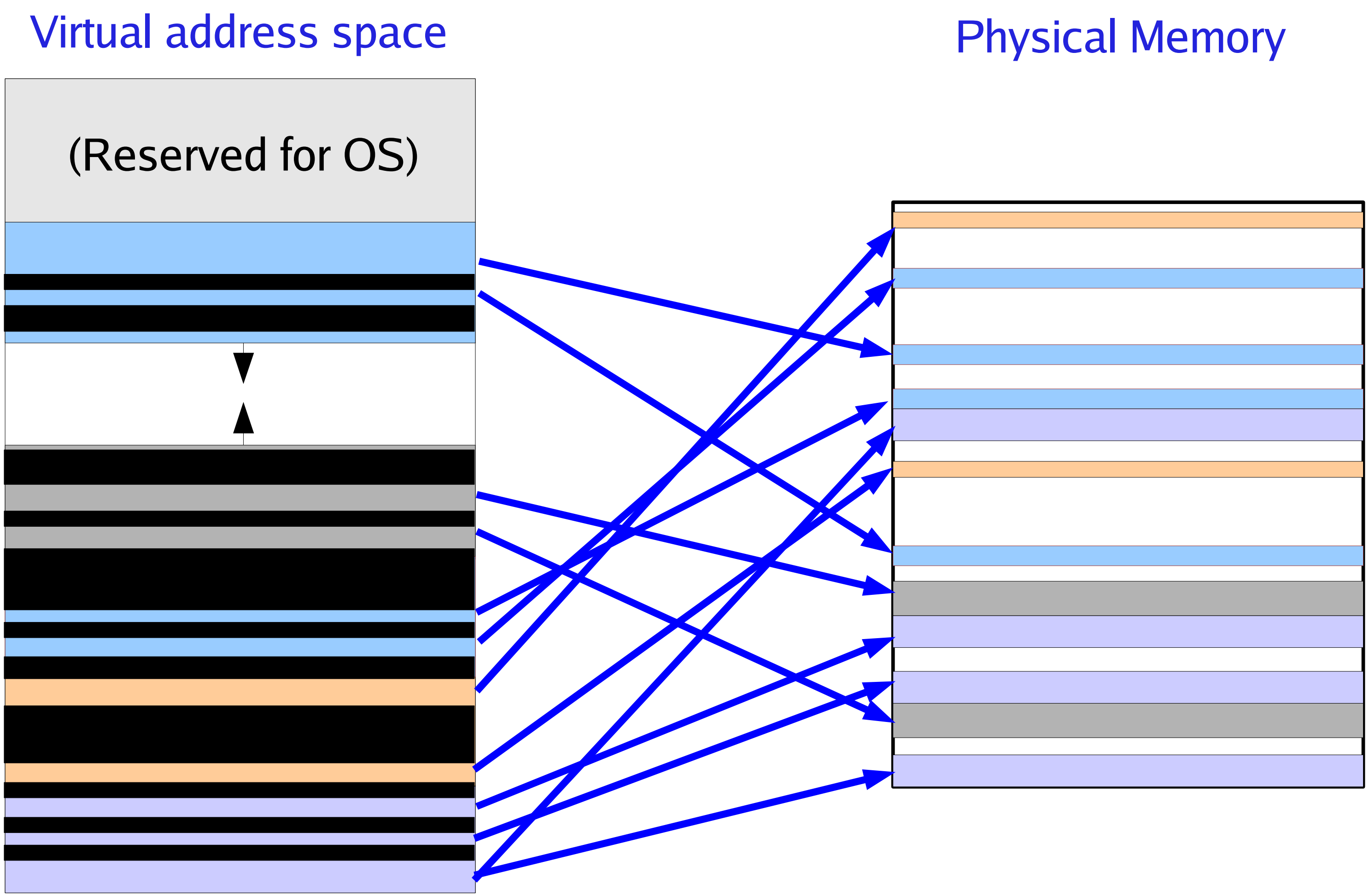
- No! Remember that only a small portion of a program's code may be used!
- For example, if you never use the "save as PDF" feature in OpenOffice...



Demand Paging

Does it make sense to read an entire program into memory at once?

- No! Remember that only a small portion of a program's code may be used!
- For example, if you never use the “save as PDF” feature in OpenOffice...



What are these “holes” ??

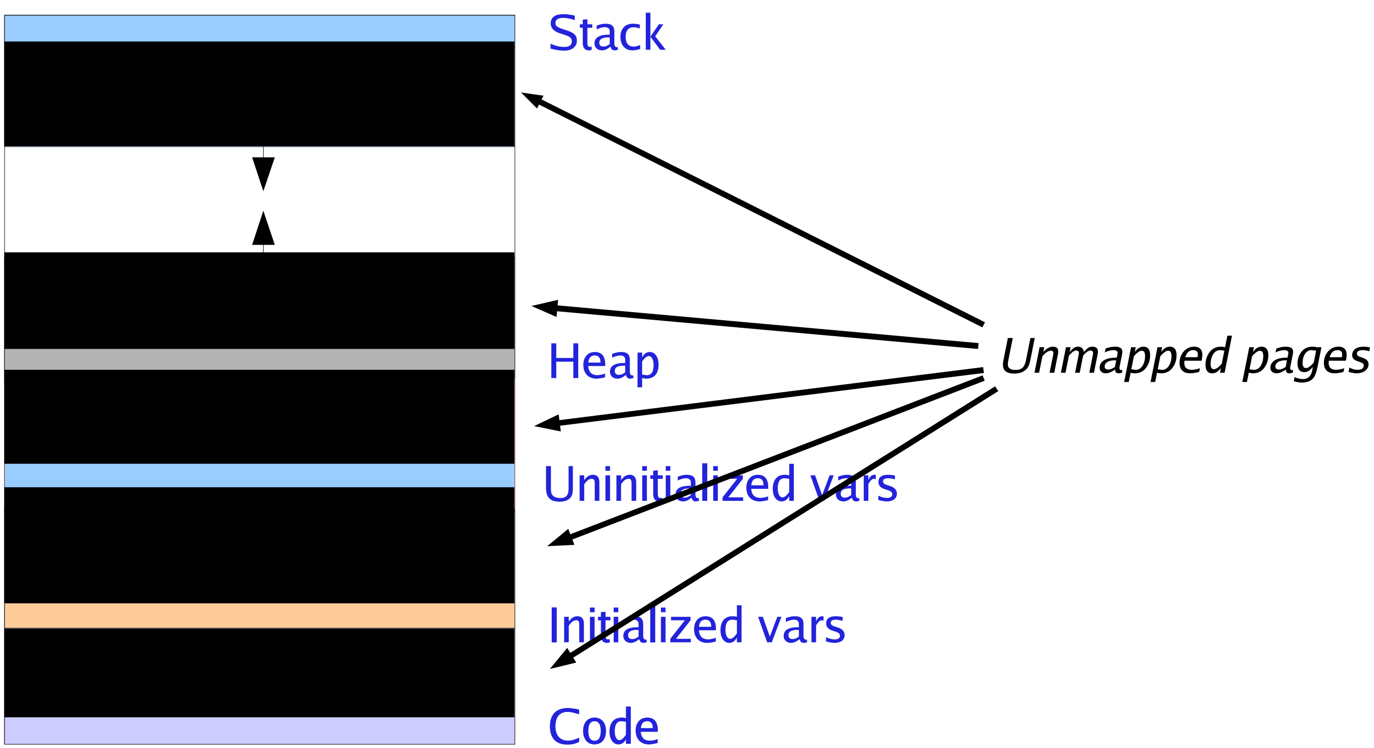
What are these “holes”?

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

1. Pages that are on disk
   - Pages that were swapped out to disk to save memory
   - Also includes code pages in an executable file
     - *When a page fault occurs, load the corresponding page from disk*
2. Pages that have not been accessed yet
   - For example, newly-allocated memory
     - *When a page fault occurs, allocate a new physical page*
   - What are the contents of the newly-allocated page???
3. Pages that are invalid
   - For example, the “null page” at address 0x0
     - *When a page fault occurs, kill the offending process*

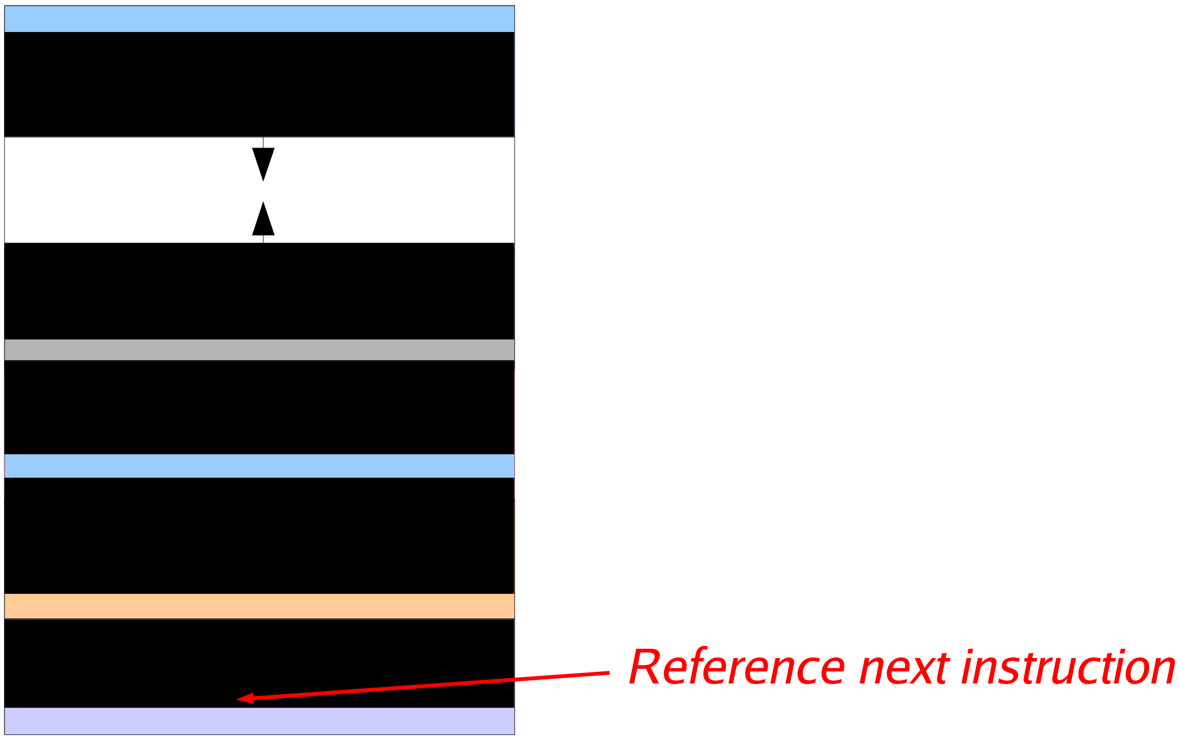
Starting up a process

What does a process's address space look like when it first starts up?



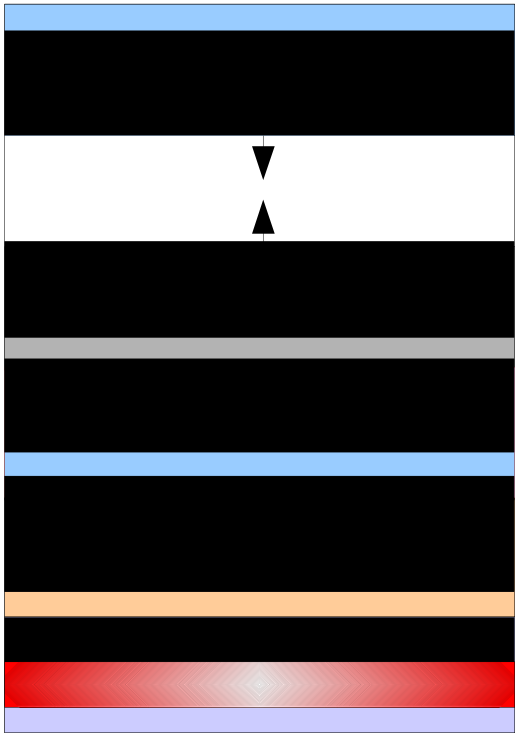
Starting up a process

What does a process's address space look like when it first starts up?



Starting up a process

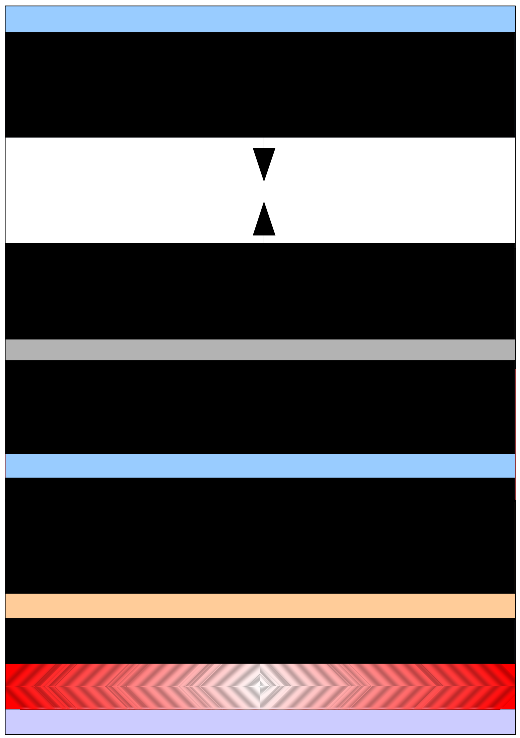
What does a process's address space look like when it first starts up?



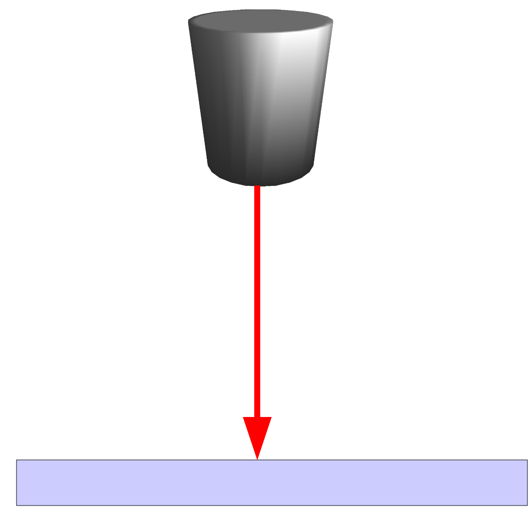
Page fault!!!

Starting up a process

What does a process's address space look like when it first starts up?



OS reads missing page from executable file on disk

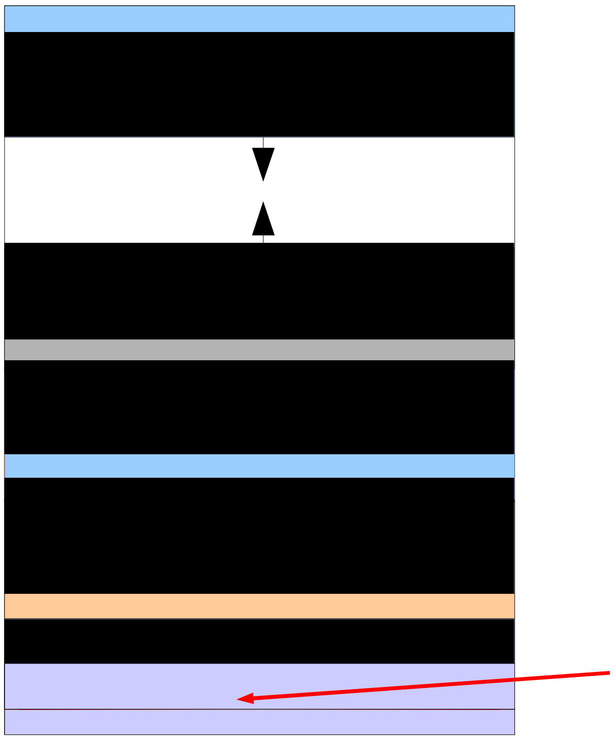


Starting up a process

What does a process's address space look like when it first starts up?

Starting up a process

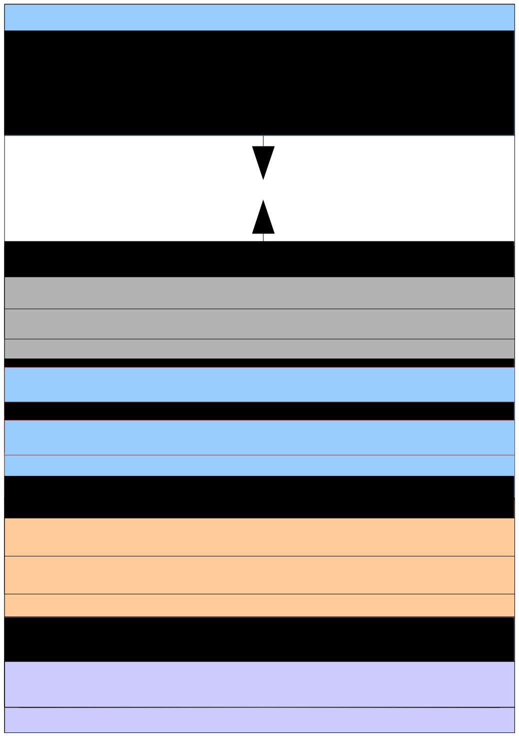
What does a process's address space look like when it first starts up?



Process resumes at the next instruction

Starting up a process

What does a 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 uninitialized variables and the heap?

Say I have a global variable "`int c`" in the program ... what happens when the process first accesses it?

Uninitialized variables and the heap

Page faults bring in pages from the executable file for:

- Code (text segment) pages
- Initialized variables

What about uninitialized 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 it corresponds to a *zero page*
- Allocates a new physical frame in memory **and sets all bytes to zero**
  - *Why???*
- Maps the frame into the address space
  - *What do I mean by this?*

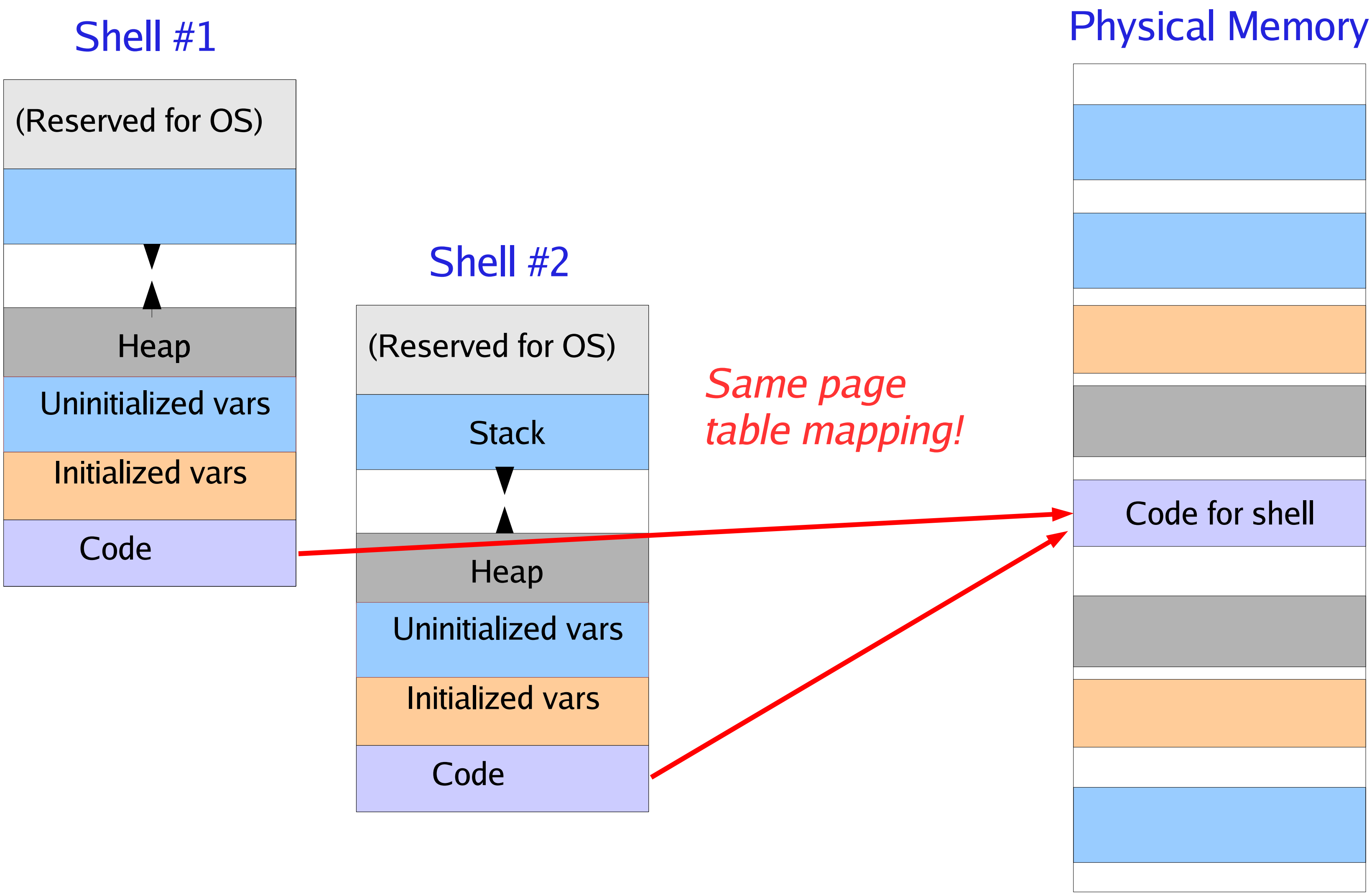
What about the heap?

- `malloc()` just asks the OS to map new zero pages into the address space
- Page faults allocate new empty pages as above

More Demand Paging Tricks

Paging can be used to allow processes to share memory

- A significant portion of many process's address space is identical
- For example, multiple copies of your shell all have the same exact code!



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?**

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?

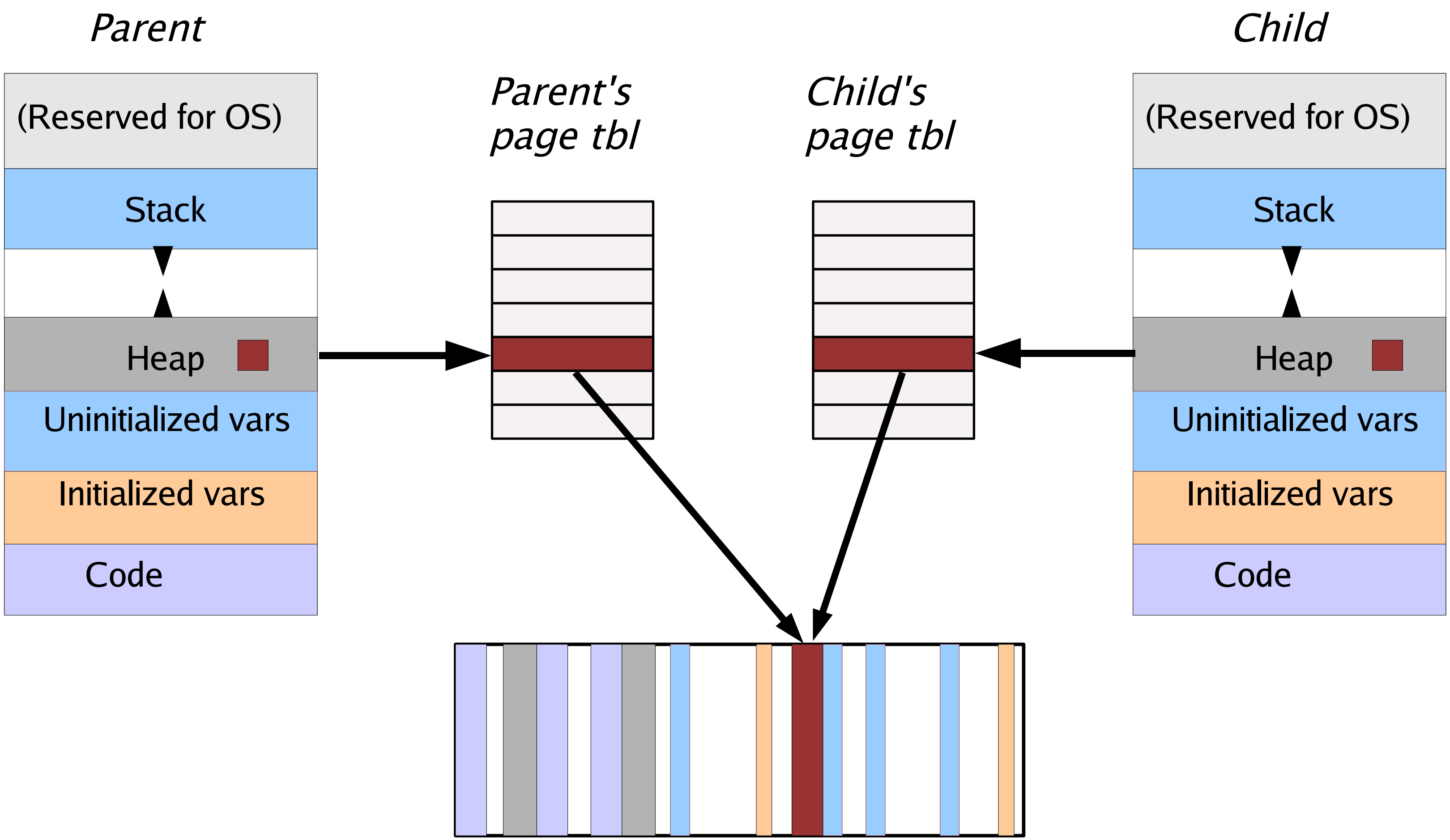
What if the child process doesn't end up touching most of the memory the parent was using?

- Extreme example: What happens if a process does an exec() immediately after fork()?

Copy-on-write

Idea: Give the child process access to the same memory, but don't let it write to any of the pages directly!

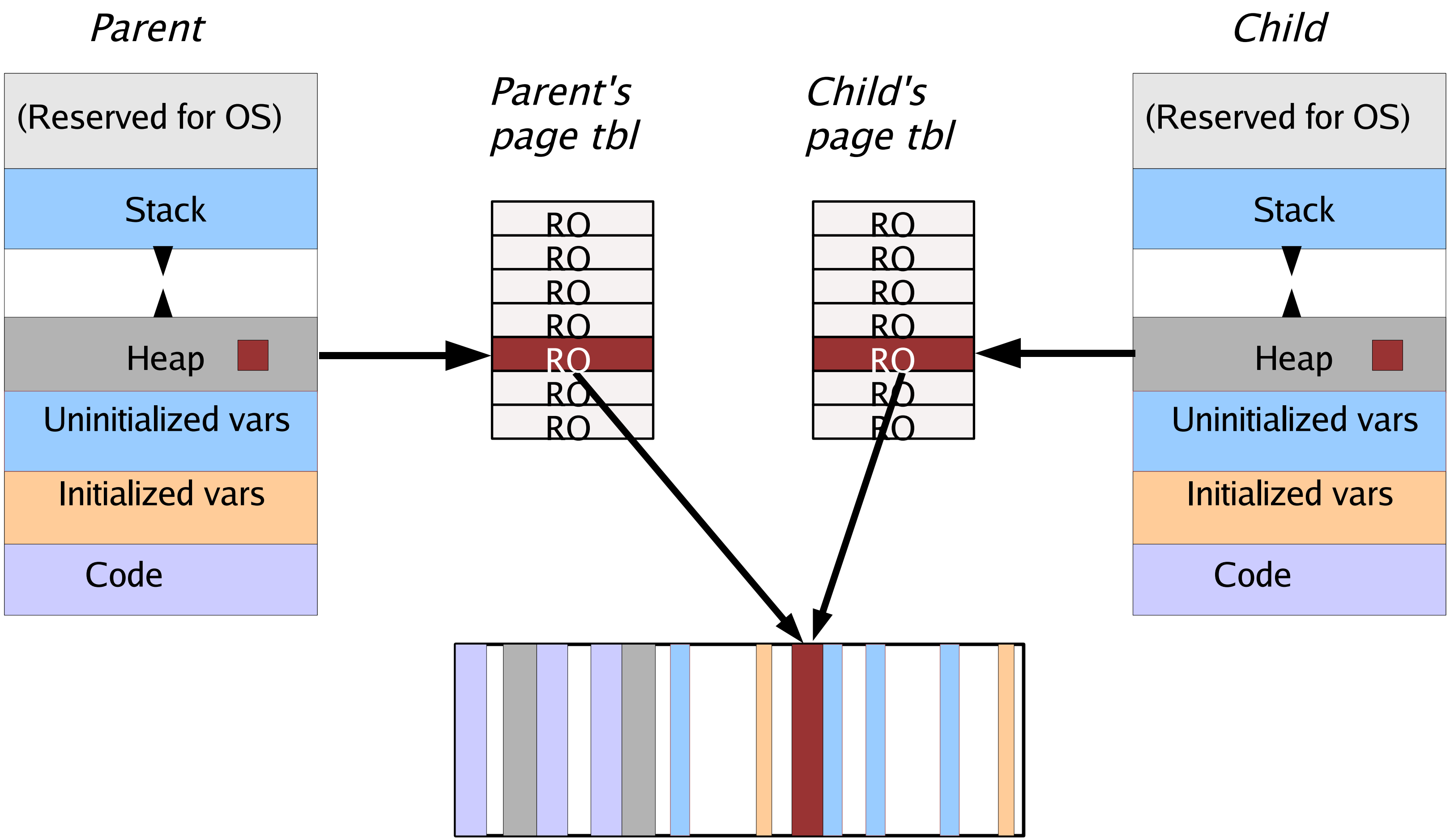
- 1) Parent forks a child process
- 2) Child gets a copy of the parent's page tables
  - *They point to the same physical frames!!!*



Copy-on-write

All pages (both parent and child) marked read-only

- Why???



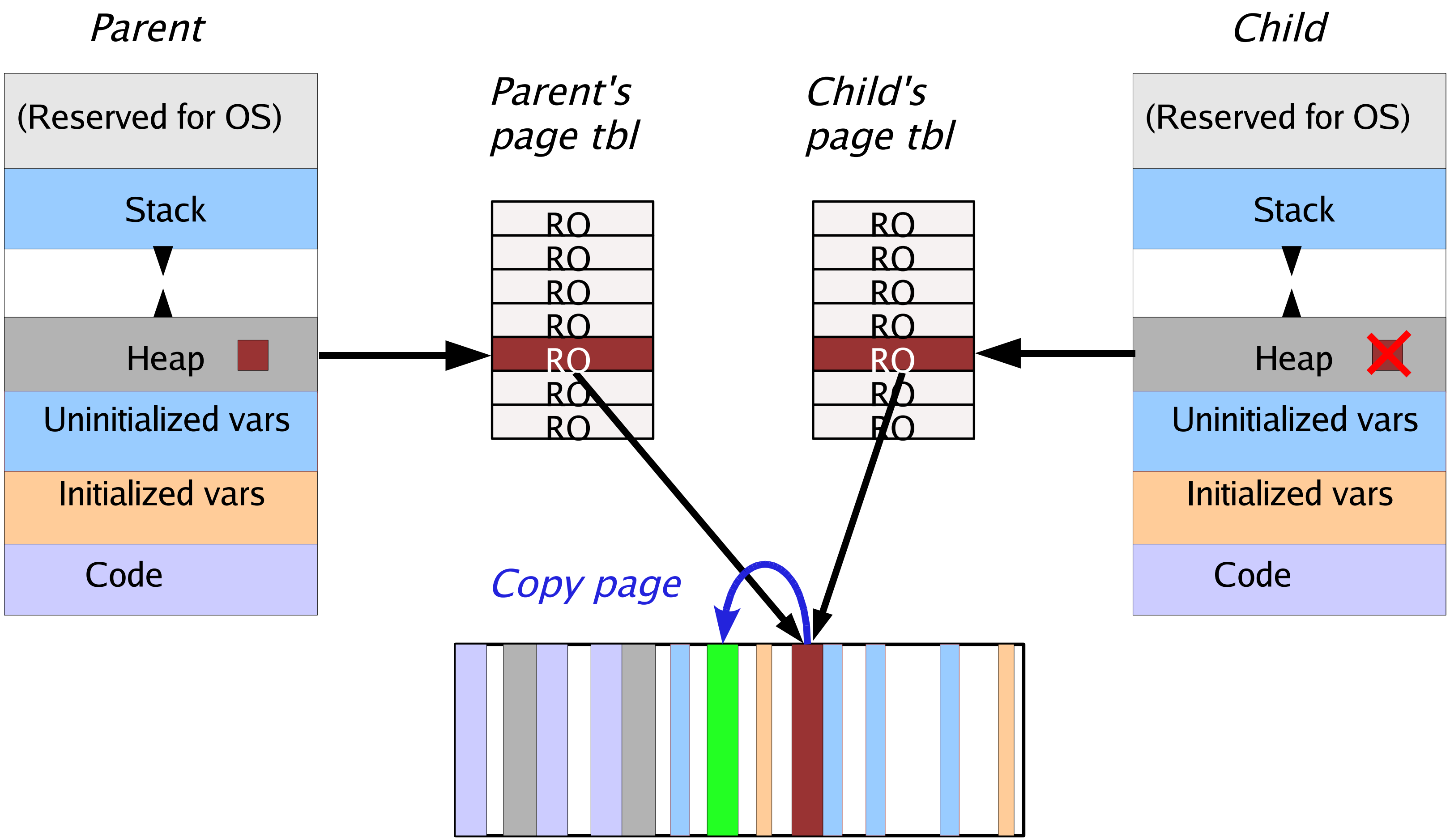
Copy-on-write

What happens when the child *reads* the page?

- Just accesses same memory as parent .... niiiiiice

What happens when the child *writes* the page?

- Protection fault occurs (page is read-only!)
- OS copies the page and maps it R/W into the child's addr space



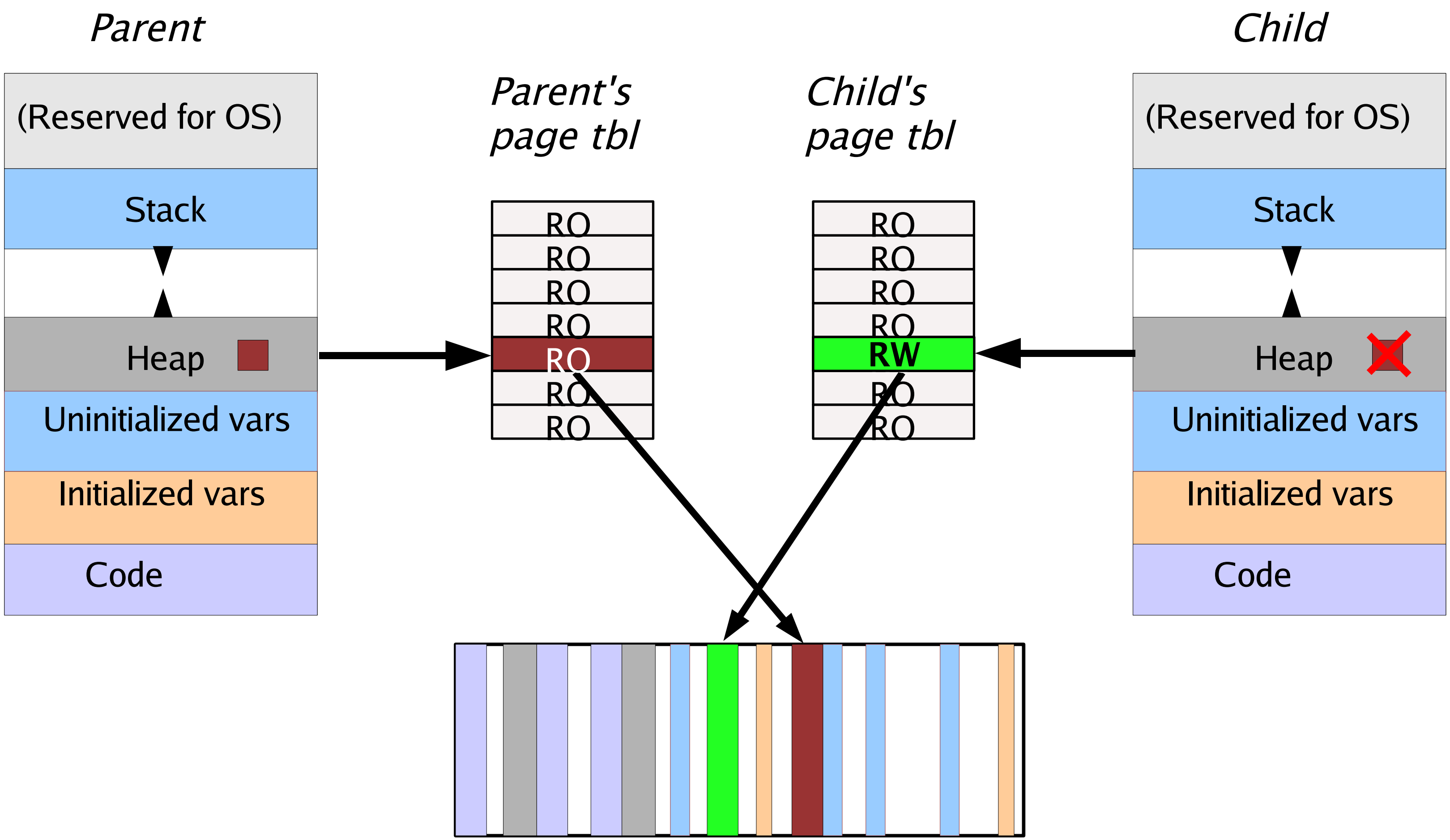
Copy-on-write

What happens when the child *reads* the page?

- Just accesses same memory as parent .... niiiiiice

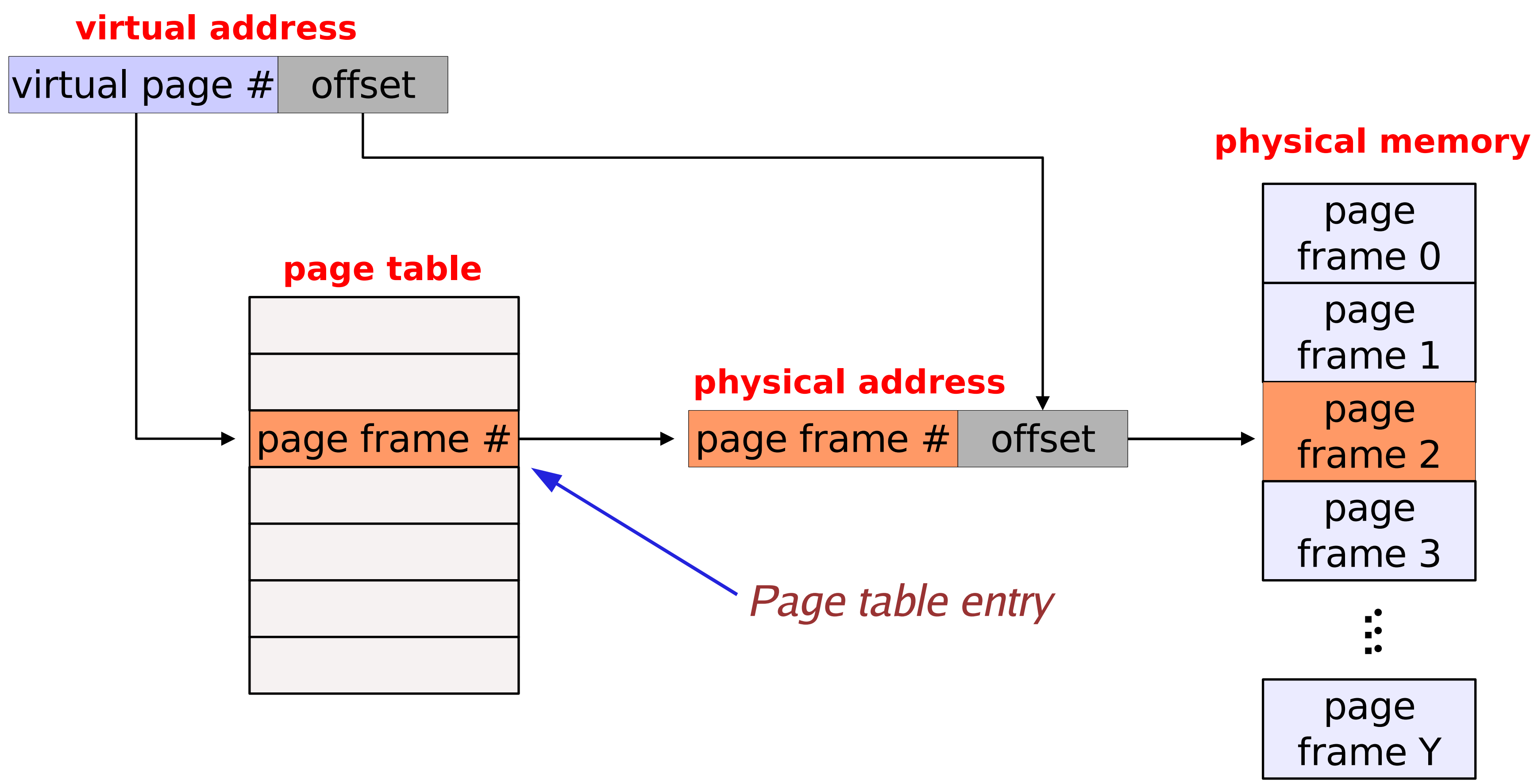
What happens when the child *writes* the page?

- Protection fault occurs (page is read-only!)
- OS copies the page and maps it R/W into the child's addr space



Page Tables

Remember how paging works:



Recall that page tables for one process can be very large!

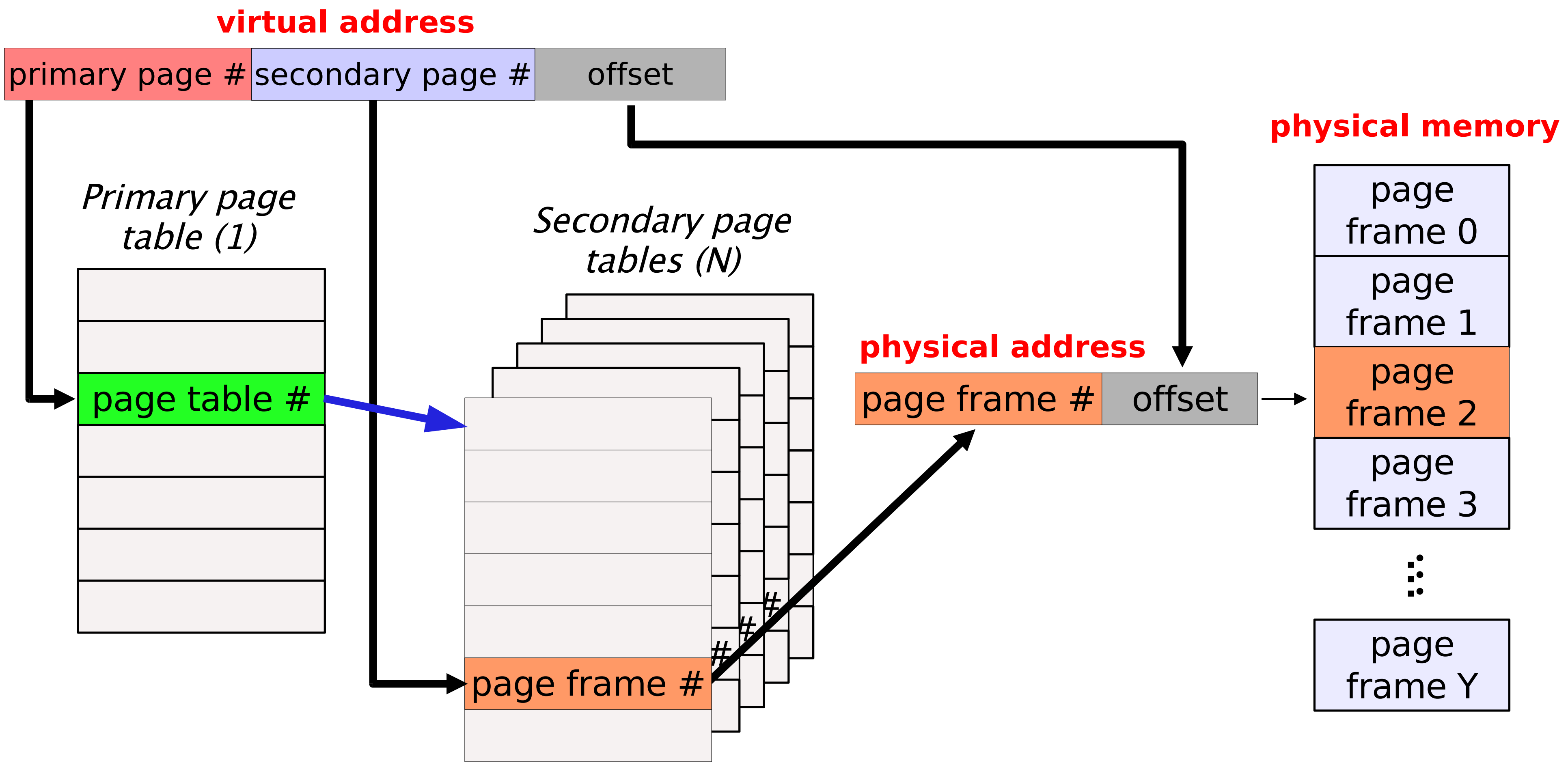
- 2^20 PTEs * 4 bytes per PTE = 4 Mbytes per process

Multilevel Page Tables

Problem: 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

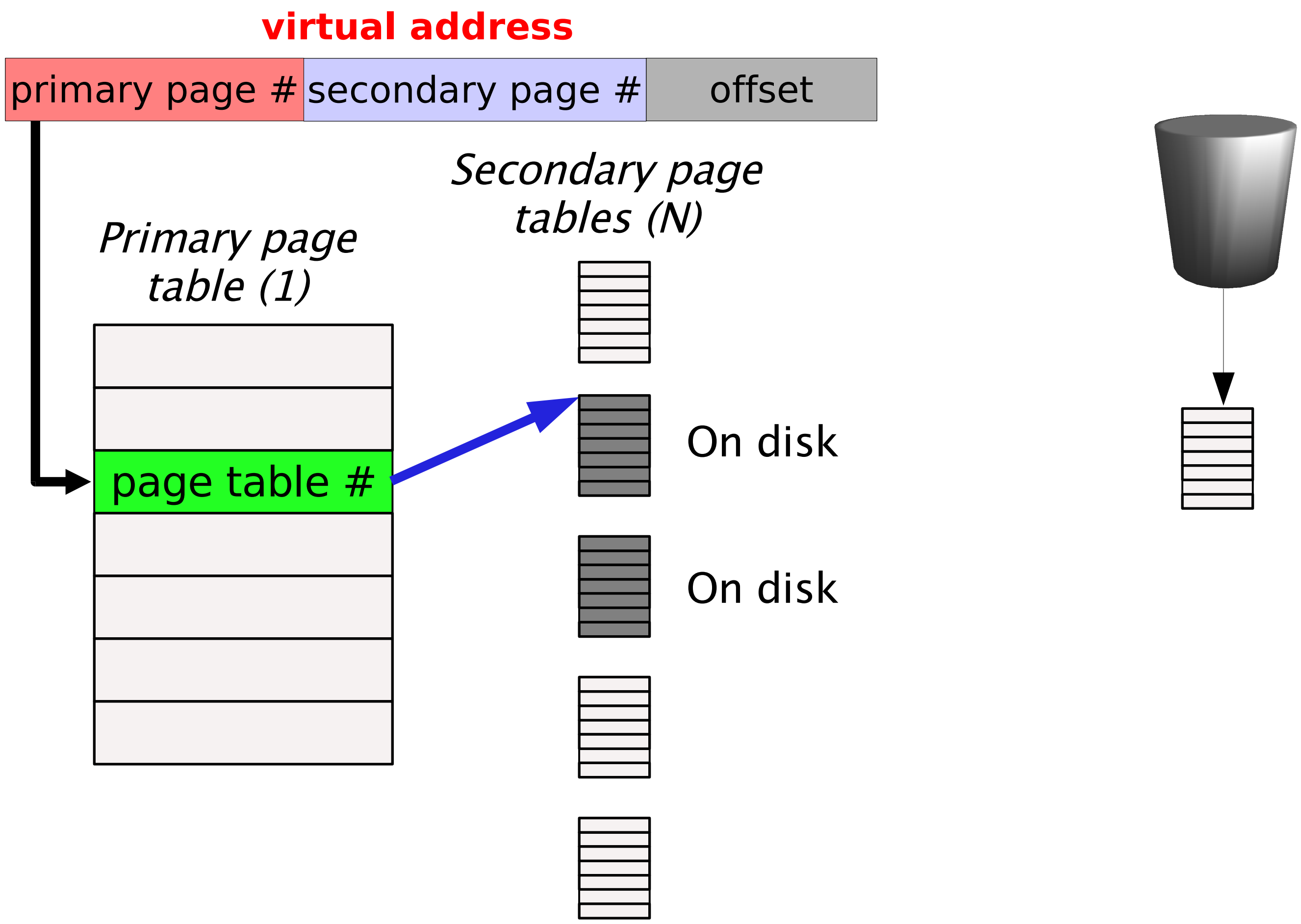


Multilevel Page Tables

Problem: 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

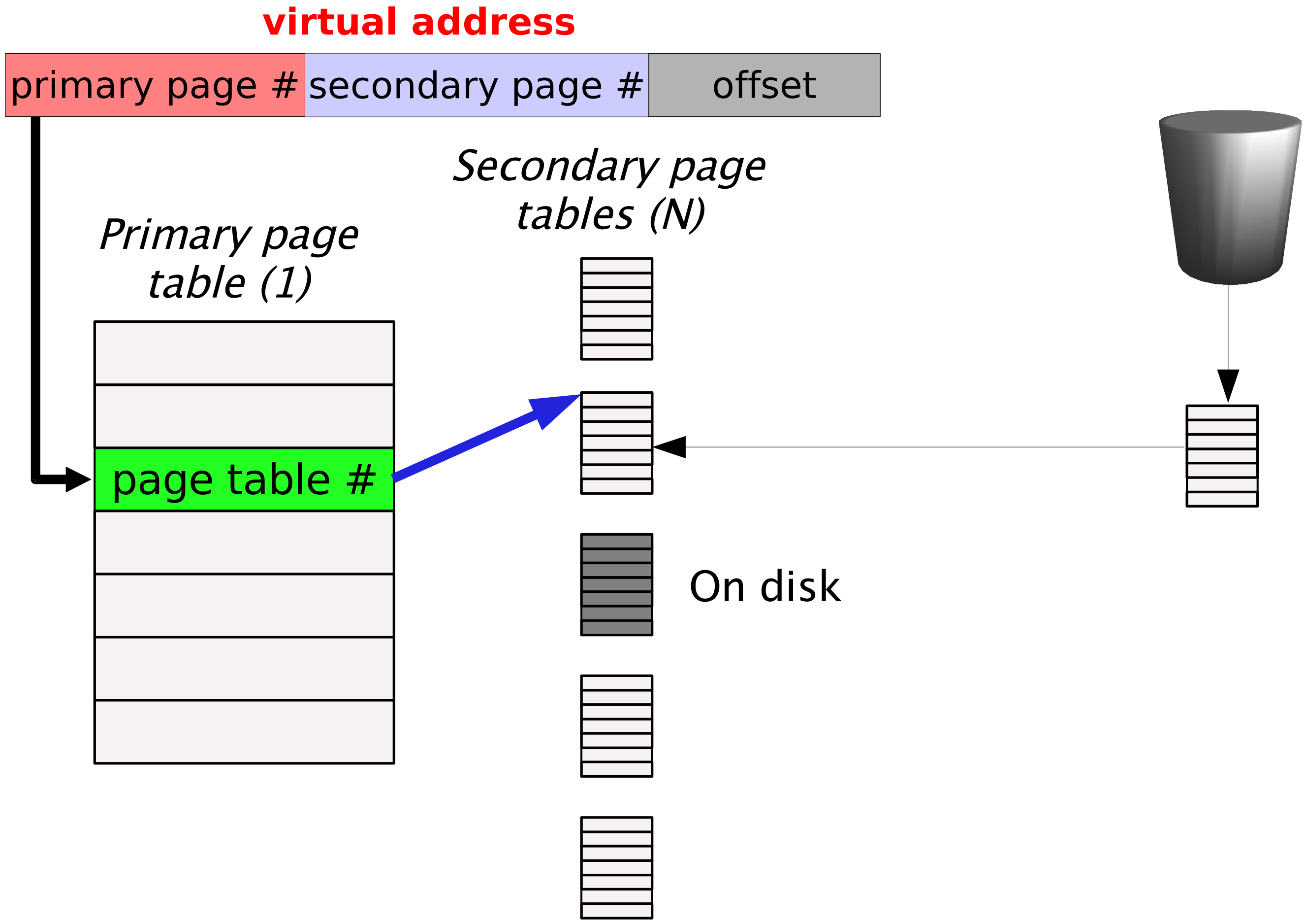


Multilevel Page Tables

Problem: 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

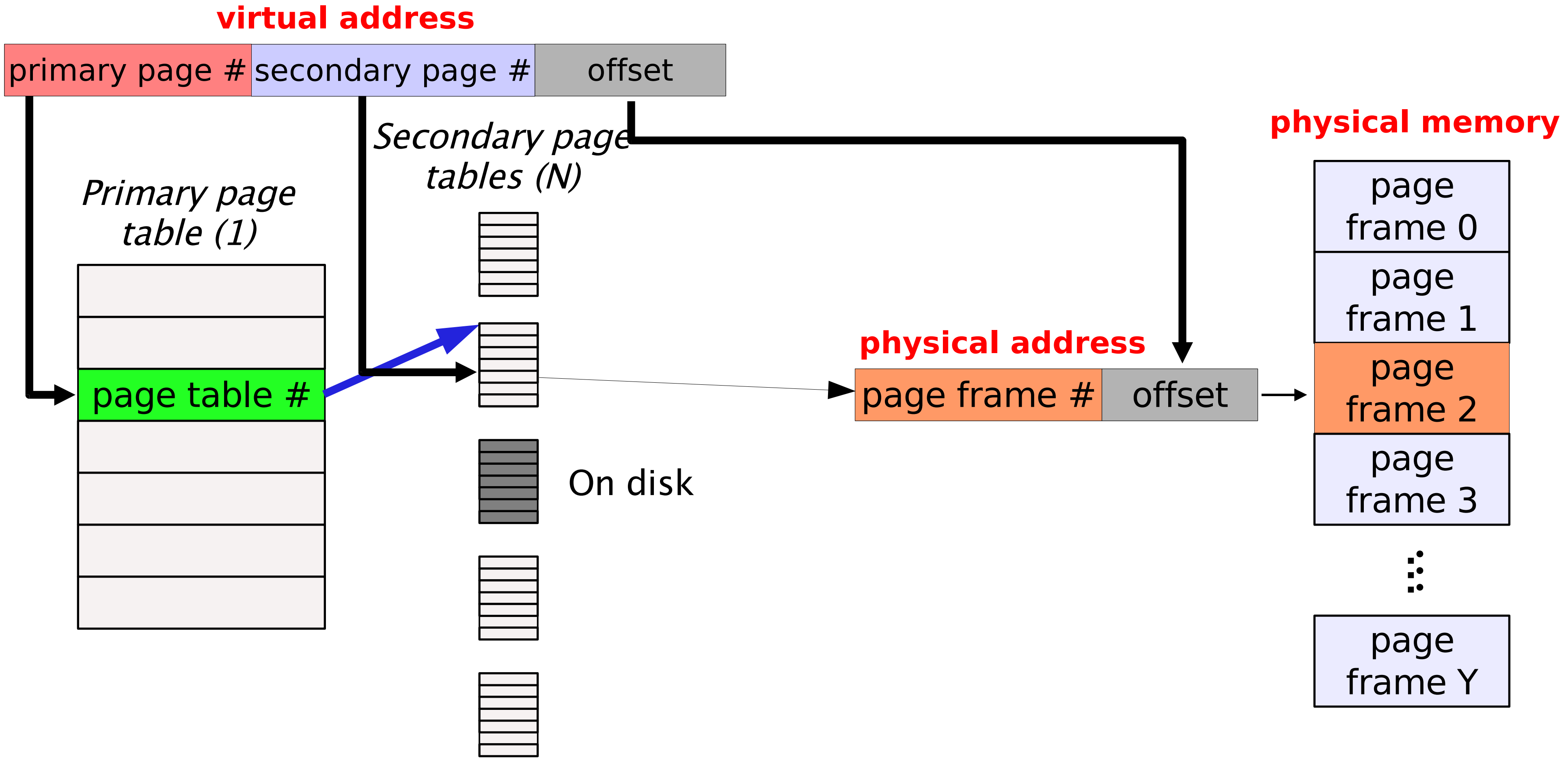


Multilevel Page Tables

Problem: 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.

Next Lecture

Page Replacement Policies

- How do we decide which pages to kick out to disk?
- How do we bring kicked out pages back into memory?