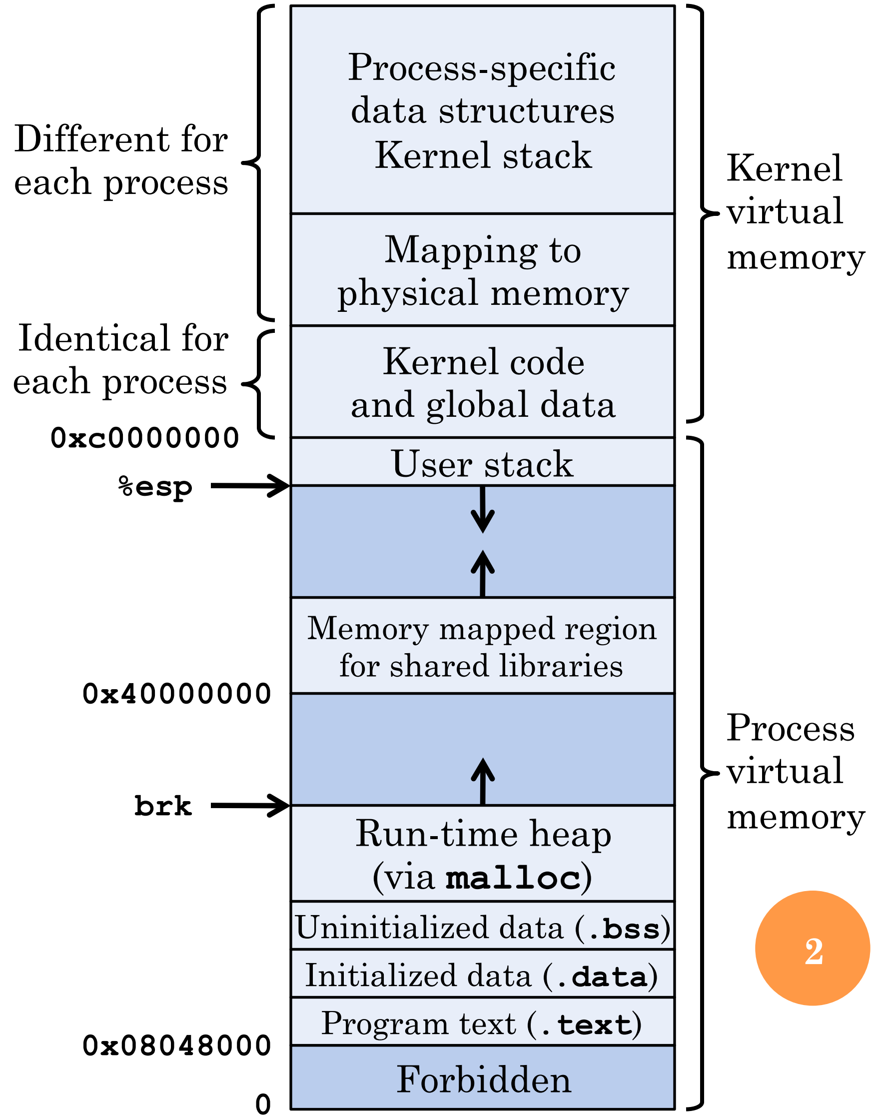
CS24: Introduction to Computing Systems

Spring 2015

Lecture 25

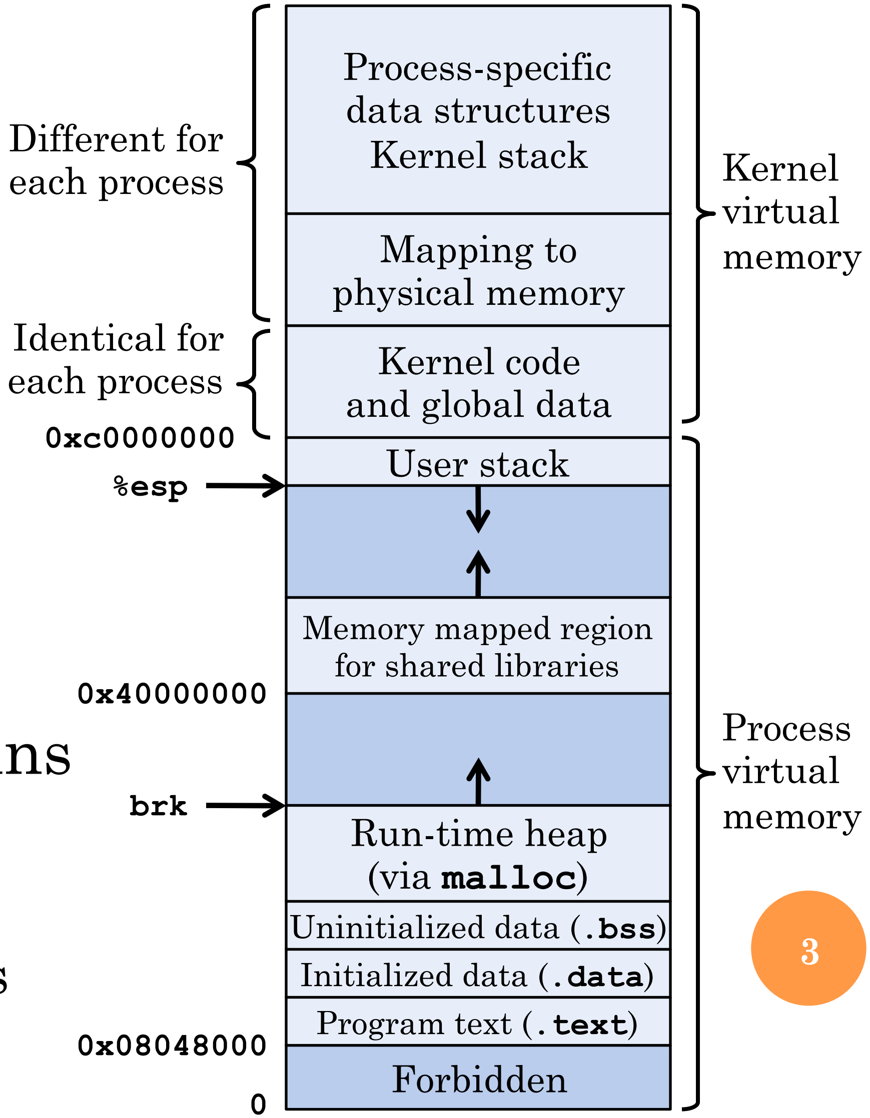
Last Time: Process Memory Layout

- Explored how Linux uses IA32 virtual memory
- All processes have a similar memory layout
  - Each process has its own page table structure
  - Processes have isolated address spaces
  - Program entry-point is at **`0x08048000`**
  - Program's stack grows down from **`0xc0000000`**



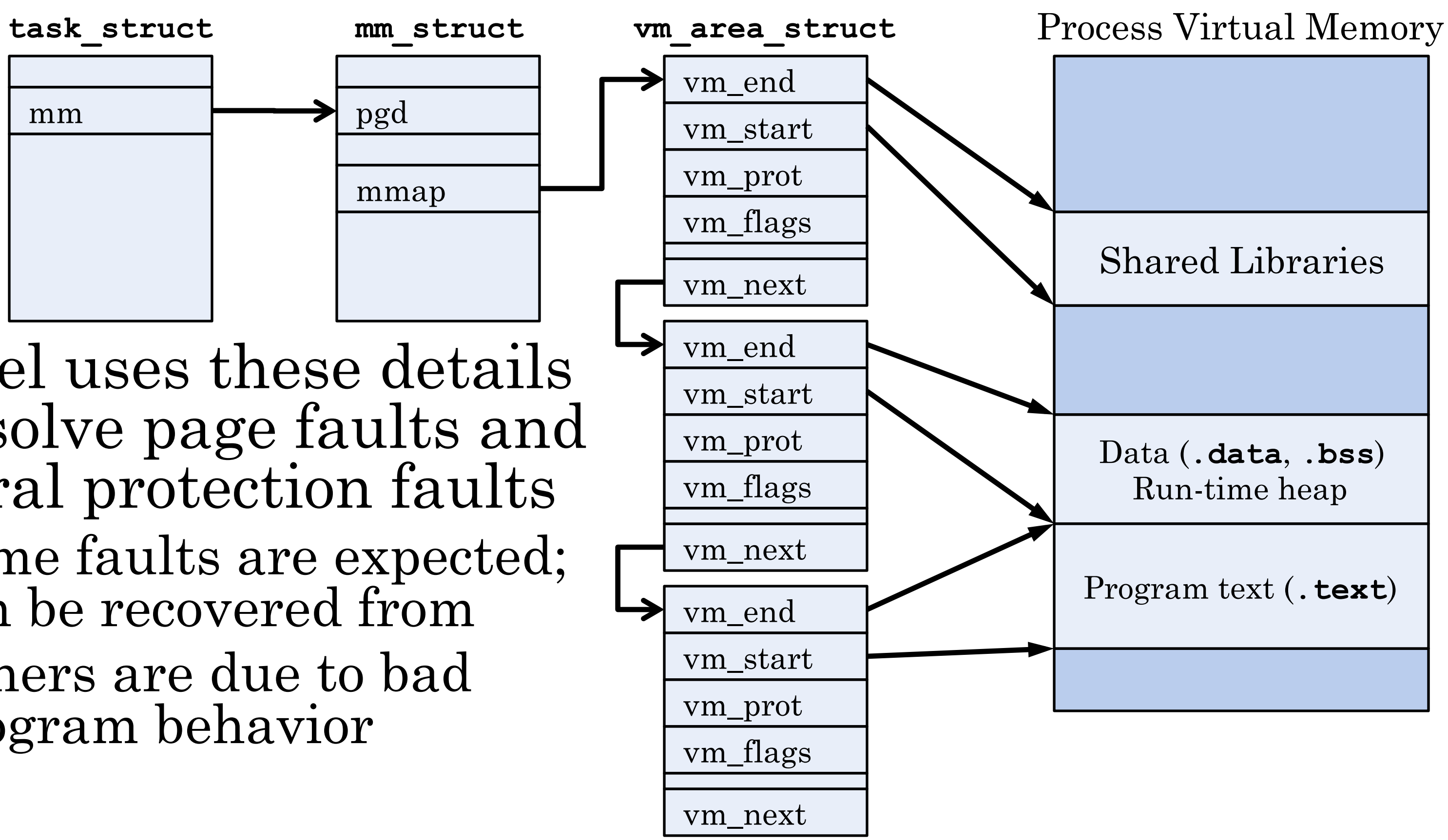
Last Time: Process Memory Layout (2)

- Kernel maps some of its own code and data into each process' memory
  - Data structures that all processes need
  - Support for system calls
  - Processes cannot access this memory directly!
    - Must use **`int 0x80`** trap, or **`sysenter/syscall`**
- Kernel memory also contains data specific to the process
  - Page table for the process
  - Kernel-stack for the process



Process Virtual Memory Areas

- Kernel maintains details about virtual memory regions in each process
  - Supplemental details beyond what is recorded in the IA32 page-directory structure



- Kernel uses these details to resolve page faults and general protection faults
  - Some faults are expected; can be recovered from
  - Others are due to bad program behavior

Memory Mapping Objects

- In Linux, each memory area is associated with an *object* on disk:
  - A *regular file* in the UNIX filesystem, such as a program binary or a shared library
  - An *anonymous file*, presented by the kernel, containing only zero values
    - *(Not an actual file, but presented via the file abstraction)*
- The anonymous file is used when a process allocates new virtual pages
  - (e.g. when the heap or stack is expanded)
  - A victim page is evicted, and then the page's contents are overwritten with zero values
  - Once a new page is allocated, it is saved to disk in a special swap area

Memory Mapping Objects (2)

- The anonymous file is used when the kernel allocates new virtual pages to a process
- Reason:
  - A process should never be able to see data from another process, unless it is explicitly shared
- Example: a process that prompts the user for a password
  - Another process allocates a virtual page, which maps to the same physical page in DRAM…
  - …but the password data wasn't overwritten before the second process gets the page!
- The kernel must ensure that such situations cannot happen

Shared and Private Objects

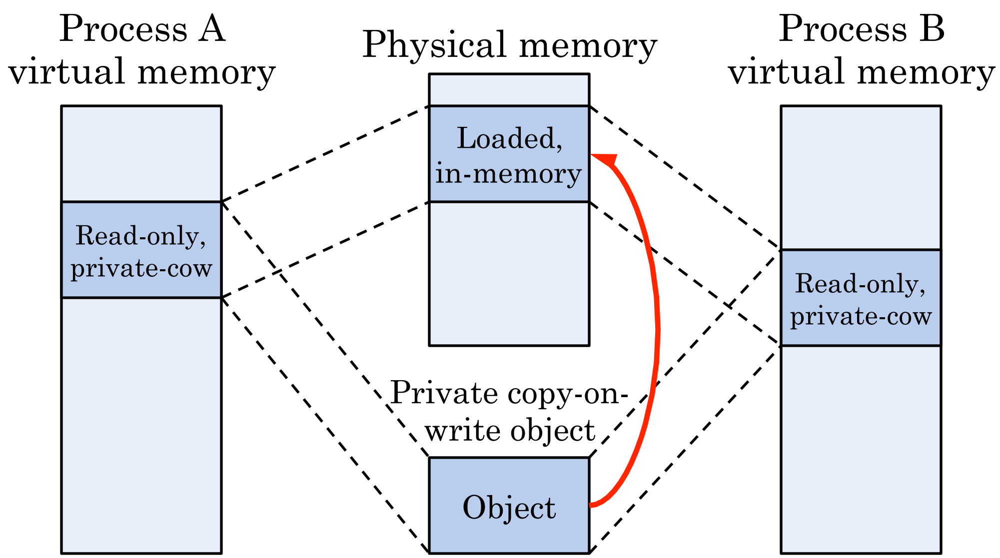
- Objects can be mapped into a virtual memory area as either a *shared object* or a *private object*
  - Memory area that the shared object is mapped into is called a *shared area*
  - Similarly, memory area that a private object is mapped into is called a *private area*
- Multiple processes may be running same code (e.g. **`/bin/bash`**), or using same shared library (**`libc.so`**)
  - Load the shared object into physical memory <u>once</u>, and then map it into the address space of multiple processes
- Writes to a shared object are visible to <u>all</u> processes accessing the object!
  - Writes will also modify the shared object stored on disk!
- With shared objects, *very* important to enable writes only when appropriate!

Shared and Private Objects (2)

- When a process writes to a private object, only that process should see the modification
  - Additionally, the change should not modify the private object on disk
- A simple technique:
  - Each time a specific private object is mapped into a process, load another copy into physical memory
- This approach can become *very* expensive
  - Particularly in situations where a specific private object is loaded into multiple processes, but none of the processes are making changes to the object!
- A much better technique: *copy-on-write*

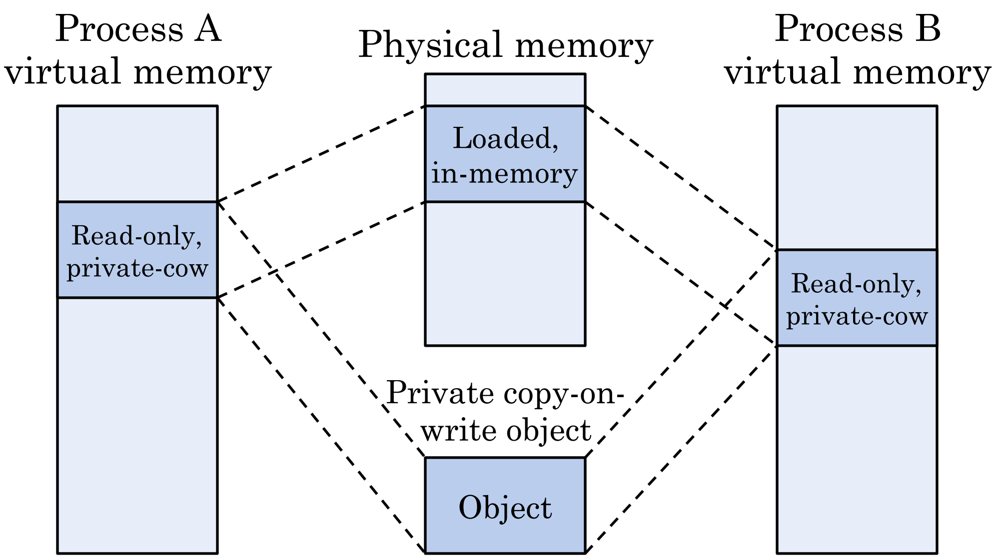
Private Objects and Copy-on-Write

- Like shared objects, a private object is initially loaded into physical memory only once
  - Kernel sets the object's pages to be read-only, and flags the memory area as *private copy-on-write*
- Example:
  - Two processes using a private copy-on-write object
  - Initially, object is loaded into physical memory only once
- Both processes have virtual memory area marked as read-only, private copy-on-write



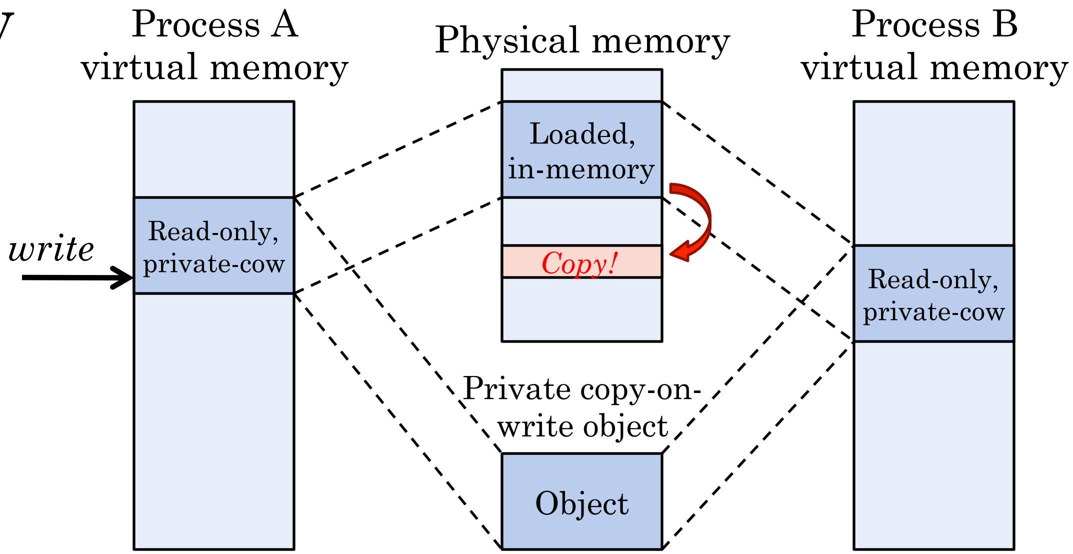
Private Objects, Copy-on-Write (2)

- When a process writes to a private copy-on-write page, this generates a general protection fault
  - Process tried to write to a read-only memory page!
- Kernel handles these faults in a special way:
  - If write was to a read-only area that is also copy-on-write, make a copy of the page that was accessed
  - Create a new page
  - Copy data from old page into new page
  - In the process' page table, replace the old page with new page
    - Now change is local to the writing process



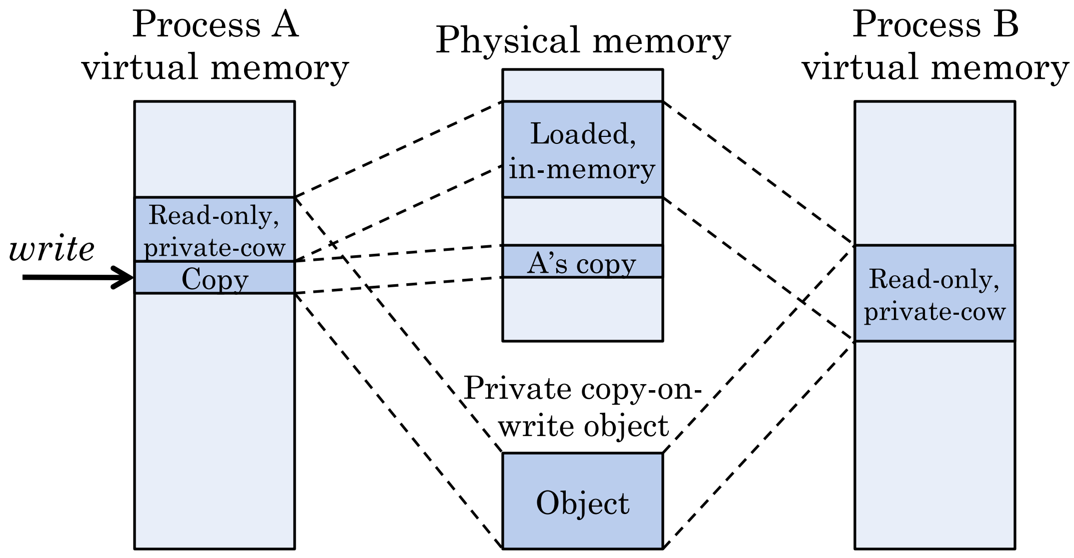
Private Objects, Copy-on-Write (3)

- Example: Process A writes to the private object
- Kernel receives a general protection fault
  - Process tried to write to a read-only page…
  - …but, the page is flagged as "private copy-on-write," so the kernel performs copy-on-write steps
- Step 1: Allocate a new page, and copy the old data into it



Private Objects, Copy-on-Write (3)

- Step 2: Update Process A's virtual memory space to reference the *copied* page, not the original
  - Copied page is also marked read-write, not read-only
- Step 3: Return from the protection-fault handler
  - CPU retries the instruction that caused the fault, and this time it succeeds without any problems

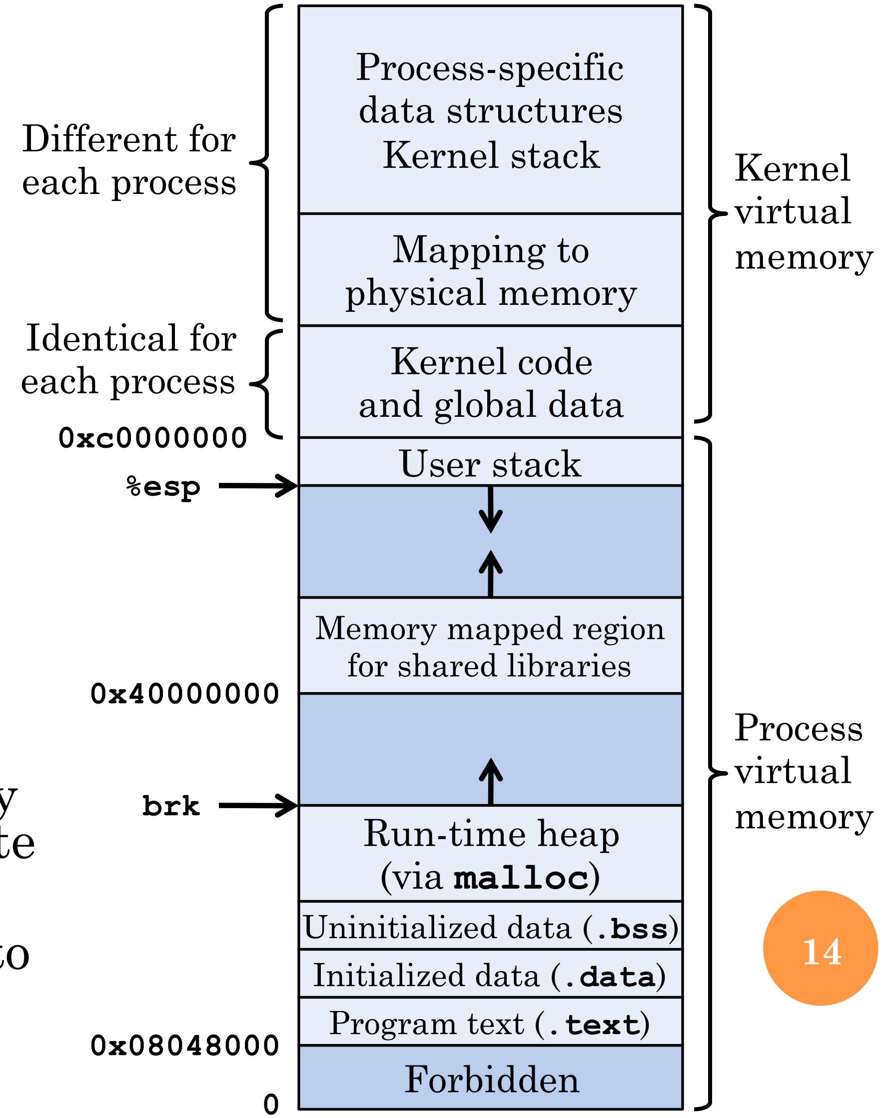


Copy-on-Write and `fork()`

- When a process calls **`fork()`**, it spawns an identical child-process
  - Most significant differences are that the process ID and parent-process ID are different
  - All code, data, and I/O state of parent process is exactly replicated in the child process
- Creating an actual copy of the parent process would be inefficient and slow
  - Parent and child process share the same program text and shared libraries, and these are read-only…
  - Plus, parent and child processes might not actually change all of the data they now share
- Instead, kernel can use copy-on-write technique to create the child process

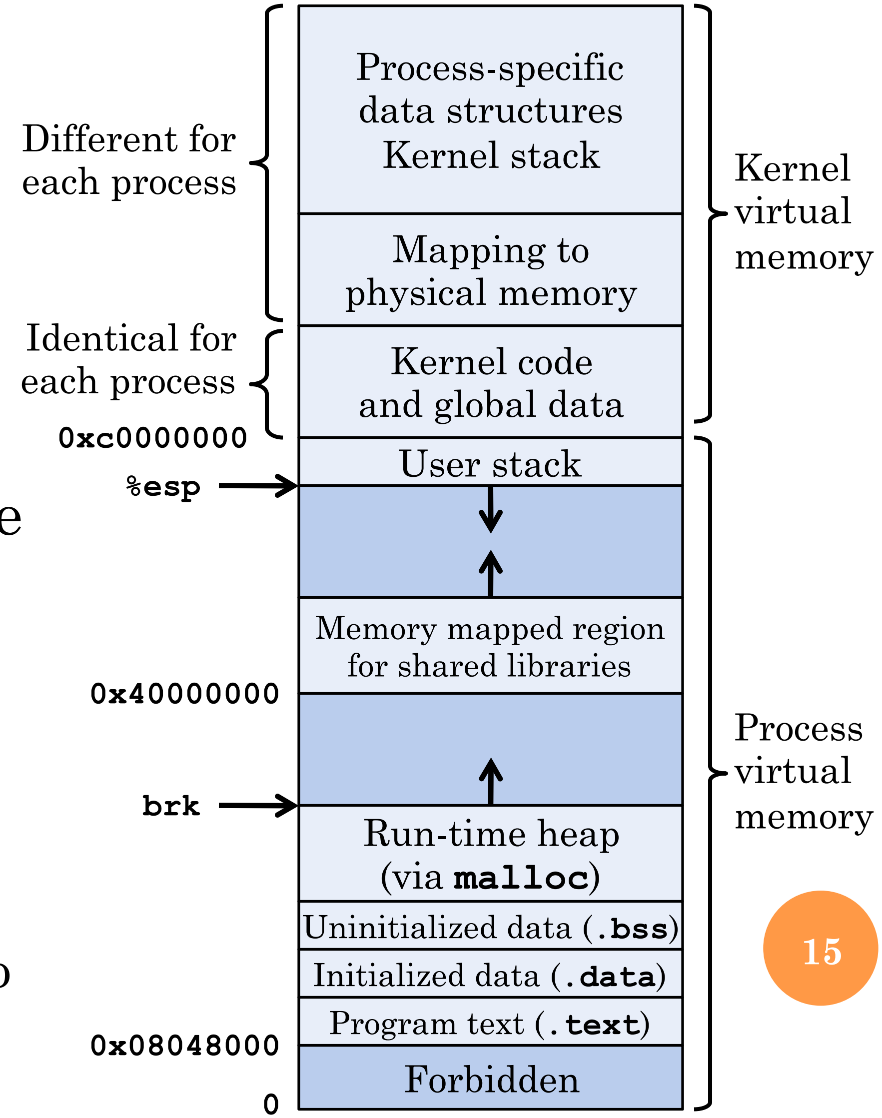
Copy-on-Write and `fork()` (2)

- When process calls **`fork()`**, the kernel can use the copy-on-write technique:
  - Duplicate process-specific data structures, including page tables and **`mm_struct`** virtual memory details
  - Flag all pages in both processes as read-only
  - Mark all memory areas as private copy-on-write
- When either parent or child process writes to memory:
  - Modified page is automatically duplicated by the copy-on-write mechanism
  - Parent and child still appear to have isolated address spaces



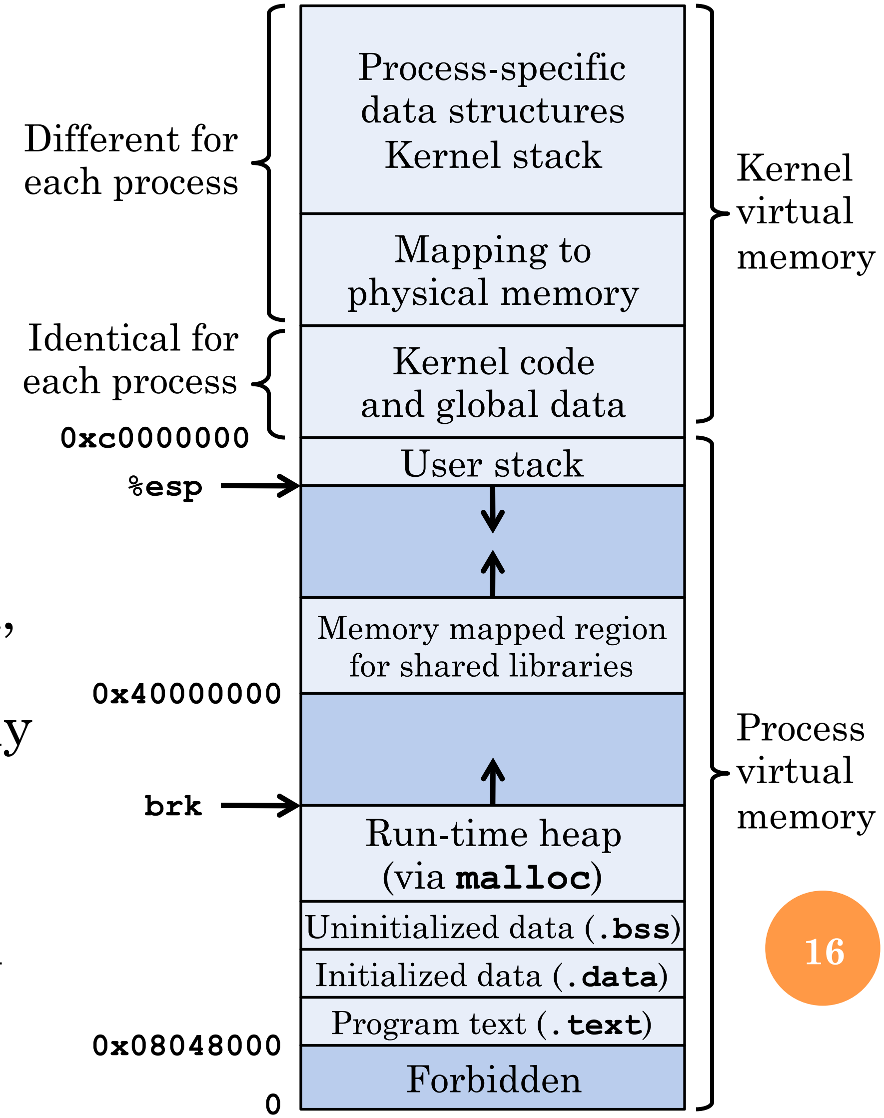
VIRTUAL MEMORY AND **execve()**

- UNIX **execve()** function also relies heavily on the virtual memory system
  - Loads and runs a new program in the current process context
- Step 1: clean up the current process' virtual memory state
  - Reset the process' virtual address space in preparation for the new program
- Iterate through the **vm_area_struct** list:
  - Unmap virtual memory areas
  - Delete **vm_area_struct**s, too



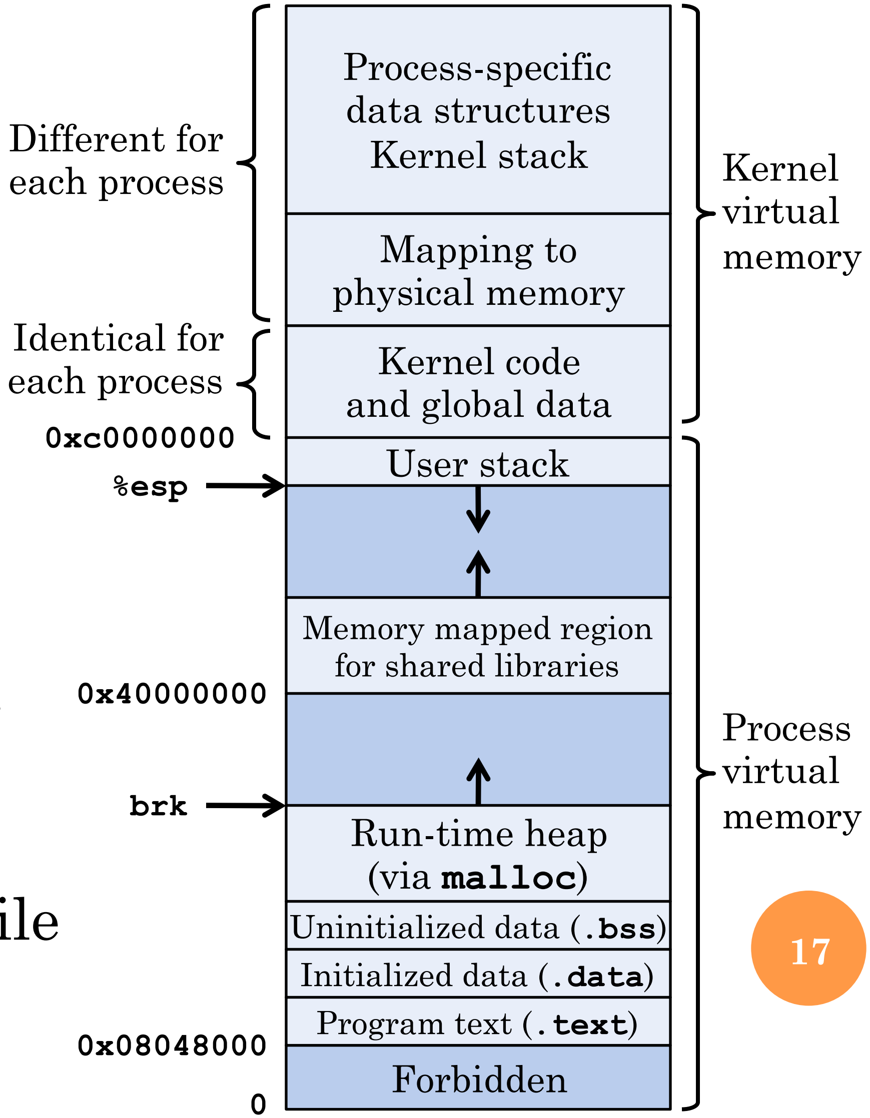
VIRTUAL MEMORY AND `execve()` (2)

- <u>Step 2</u>: map private memory areas
- Program text (`.text`) and initialized data (`.data`) are specified in binary file
  - Map these virtual pages directly to the appropriate areas of the binary file
  - When pages are referenced, kernel will swap them into main memory automatically
- Both areas are marked private copy-on-write
  - (Or, `.text` may be marked read-only instead…)



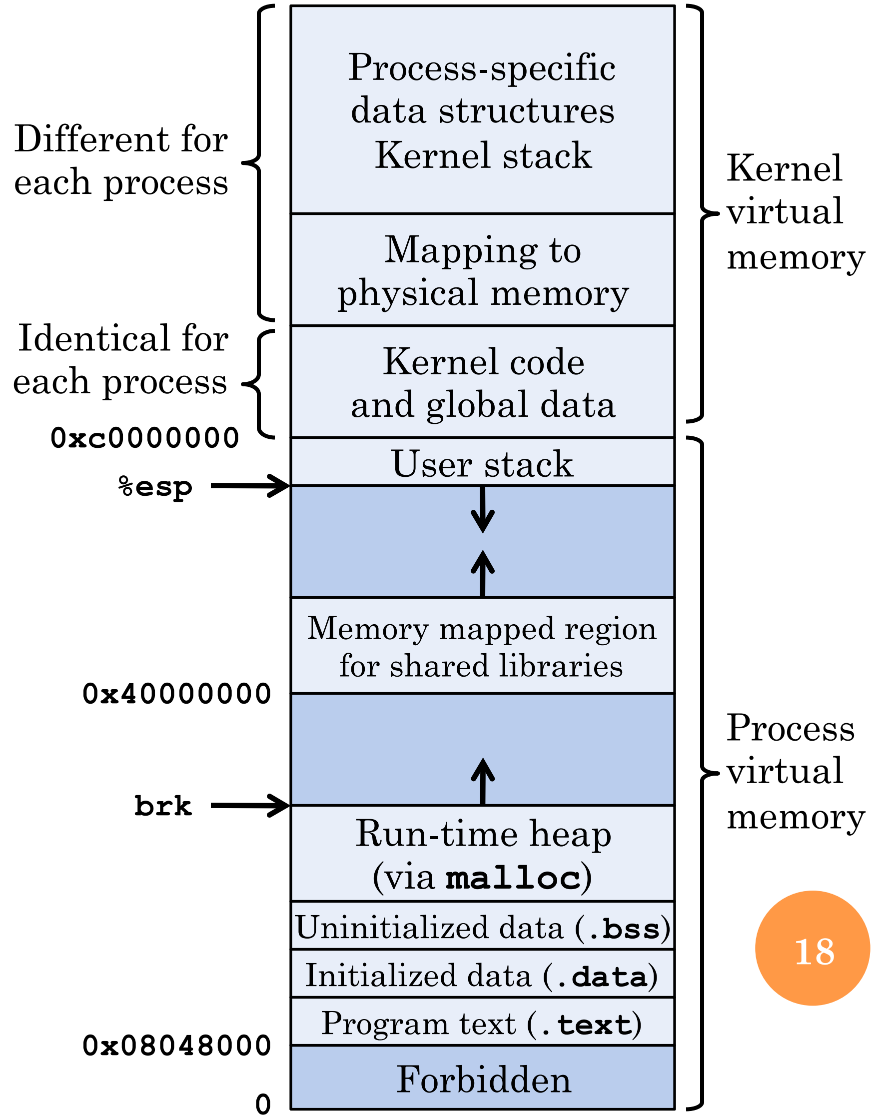
VIRTUAL MEMORY AND `execve()` (2)

- <u>Step 2</u>: map private memory areas, *cont.*
- Uninitialized data (`.bss`) isn't contained in the binary file
  - Binary simply specifies the <u>size</u> of this region
  - Kernel maps the `.bss` pages to the *anonymous file*, which contains all zero values…
- Uninitialized data is initially set to all zero values
- Kernel also maps user stack and heap to the anonymous file
  - Sizes are increased as needed



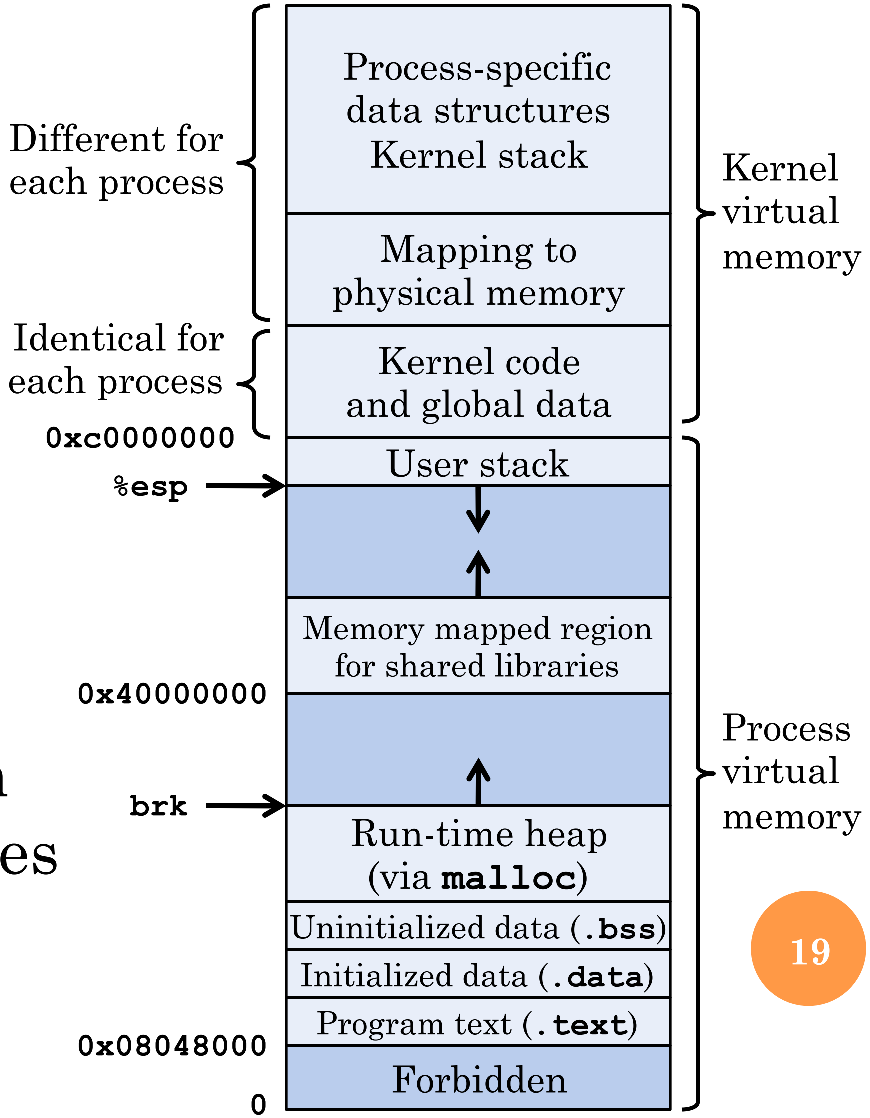
Virtual Memory and `execve()` (3)

- Step 3: map shared memory areas
- If the program is linked with any shared objects:
  - e.g. **`libc.so`**, the C standard library
- Libraries are dynamically linked into the program
- Then, the shared objects are mapped into the process' virtual address space
  - e.g. shared read-only
  - *(otherwise, changes are visible to all other processes, and also modify the original file!)*



Virtual Memory and `execve()` (4)

- Step 4: set the process' Program Counter to the program's entry-point
  - Next time the process is scheduled for execution, it will start running from the entry-point
- As new program executes, the virtual memory system will swap in necessary pages
  - e.g. program text, data, shared library code, etc.



Summary: Virtual Memory

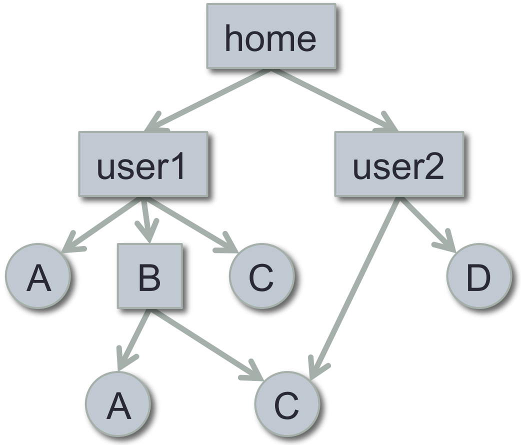
- Virtual memory is an essential component of modern operating systems
- Used extensively to implement many features
  - Process memory and isolation of address-spaces
  - Fast context-switches and process-forking
  - Simplifies loading of programs and libraries into main memory
  - Facilitates efficient memory use by sharing common data across many processes
    - Data is only duplicated when strictly necessary

SOLID STATE DRIVES AND TRIM

Supplemental Material

File Systems and File Deletion

- File systems generally separate directory information and file contents
  - One or more directories can contain a link to a given file (e.g. hard links, symlinks)



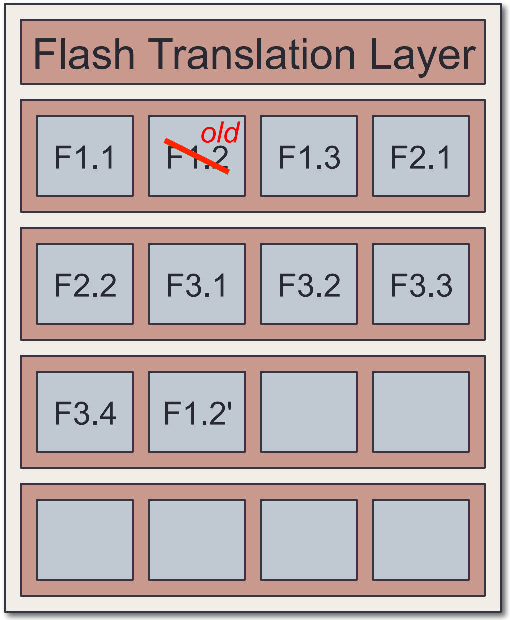
- When a file is deleted, two operations:
  - Remove directory entry to file
  - If no more directory entries reference the file, mark the file's space as available
- Normally, the file's actual data isn't modified by deletion
  - (Can securely delete file data by overwriting it one or more times)
- The operating system simply stops using the disk sectors where the file previously resided…
  - …at least until the area is used by a new file

Free Space and SSDs

- Solid State Drives (SSDs) and other flash-based devices often complicate management of free space
- SSDs are block devices; reads and writes are a fixed size
- Problem: can only write to a block that is currently empty
- Blocks can only be erased in groups, not individually!
  - An **erase block** is a group of blocks that are erased together
- Erase blocks are <u>much</u> larger than read/write blocks
  - A read/write block might be 4KiB or 8KiB…
  - Erase blocks are often 128 or 256 of these blocks (e.g. 2MiB)!
- As long as some blocks on the SSD are empty, writes can be performed immediately
- If the SSD has no more empty blocks, a group of blocks must be erased to provide more empty blocks

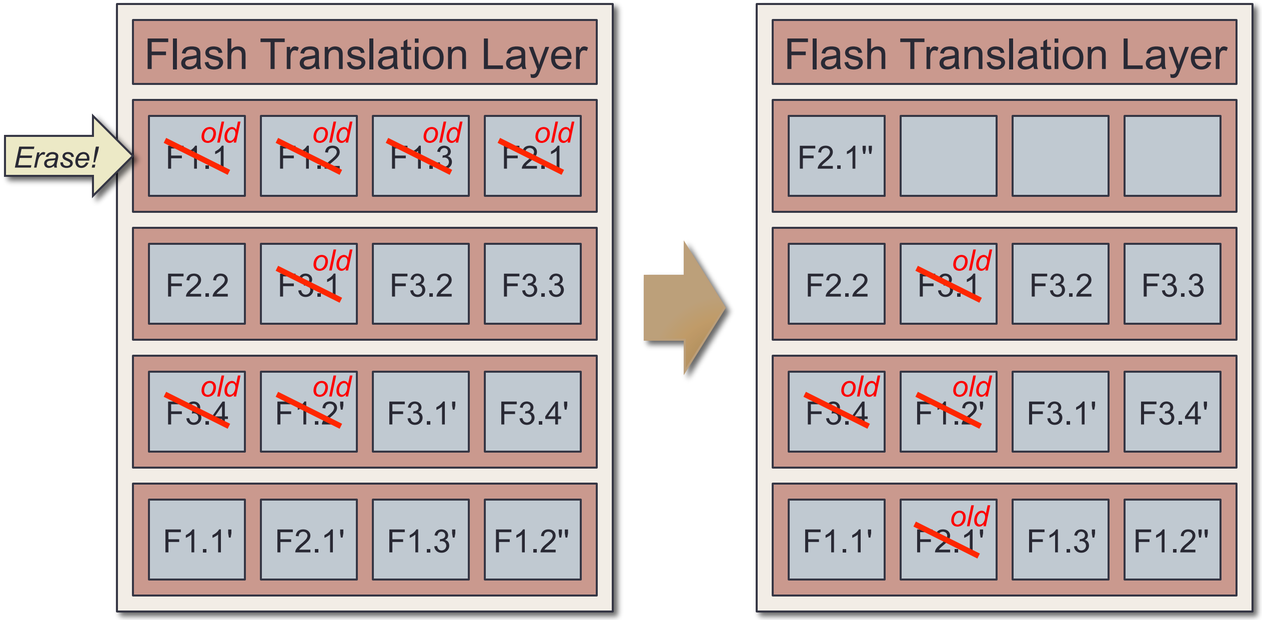
Solid State Drives

- Solid State Drives include a **flash translation layer** that maps logical block addresses to physical memory cells
  - Recall: system uses Logical Block Addressing to access disks
- When files are written to the SSD, data must be stored in empty cells (i.e. old contents can't simply be overwritten)
- If a file is edited, the SSD sees a write issued against the same logical block
  - e.g. block 2 in file F1 is written
- SSD can't just replace block's contents…
- SSD marks the cell as "old," then stores the new block data in another cell, and updates the mapping in the FTL



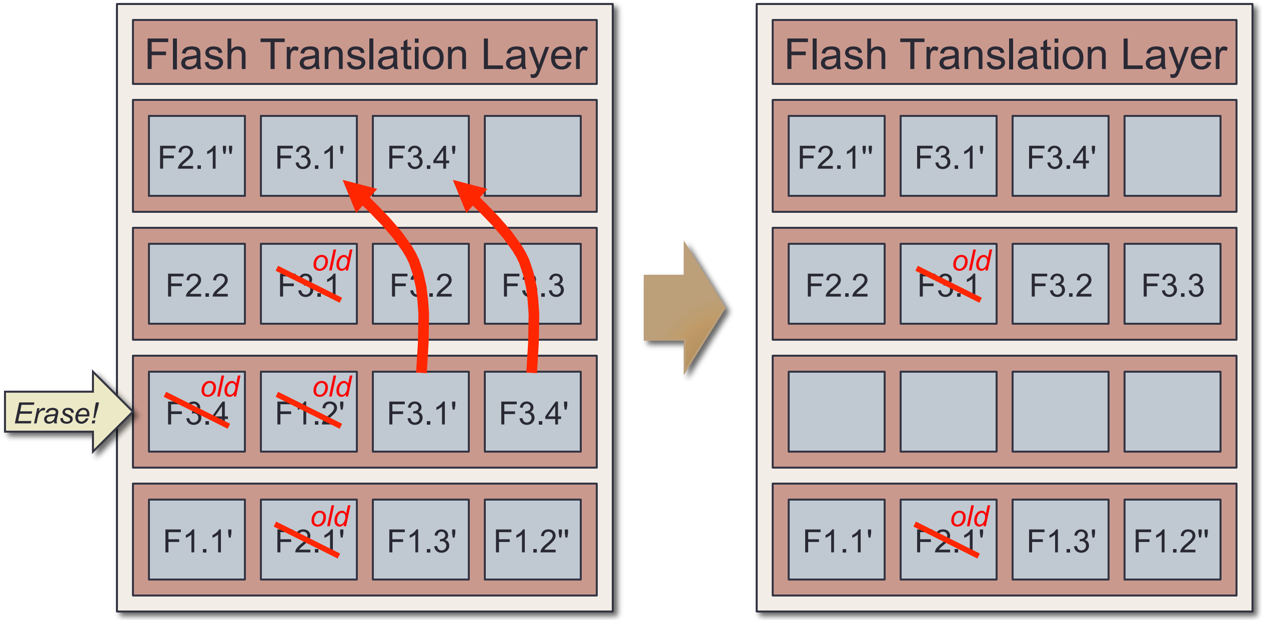
Solid State Drives (2)

- Over time, SSD ends up with few or no available cells
  - e.g. a series of writes to our SSD that results in all cells being used
- SSD must erase at least one block of cells to be reused
- Best case is when an entire erase-block can be reclaimed
  - SSD erases the entire block, and then carries on as before



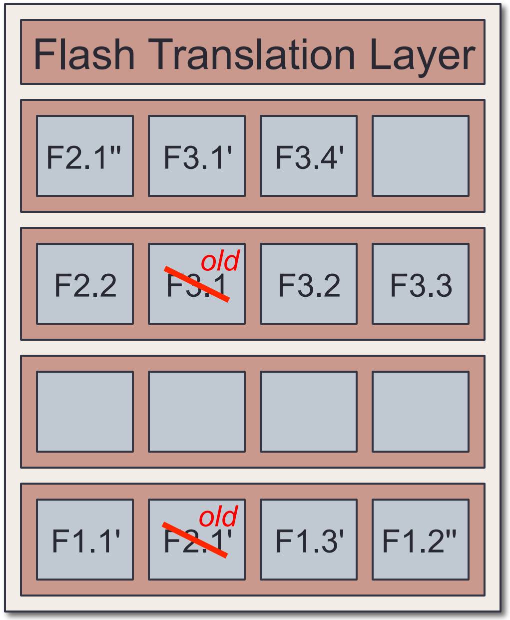
Solid State Drives (3)

- More complicated when an erase block still holds data
  - e.g. SSD decides it must reclaim the third erase-block
- SSD must relocate the current contents before erasing
- Result: sometimes a write *to* the SSD incurs additional writes *within* the SSD
  - Phenomenon is called **write amplification**



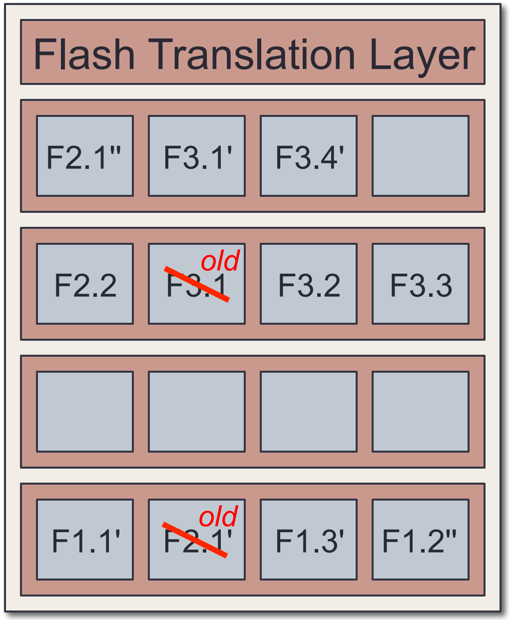
Solid State Drives (4)

- SSDs must carefully manage this process to avoid uneven wear of its memory cells
  - Cells can only survive so many erase cycles, then become useless
- How does the SSD know when a cell's contents are no longer needed? (i.e. when to mark the cell "old")
- The SSD only knows because it sees multiple writes to the same logical block
  - The new version replaces the old version
  - The SSD knows that the old cell is no longer used for storage



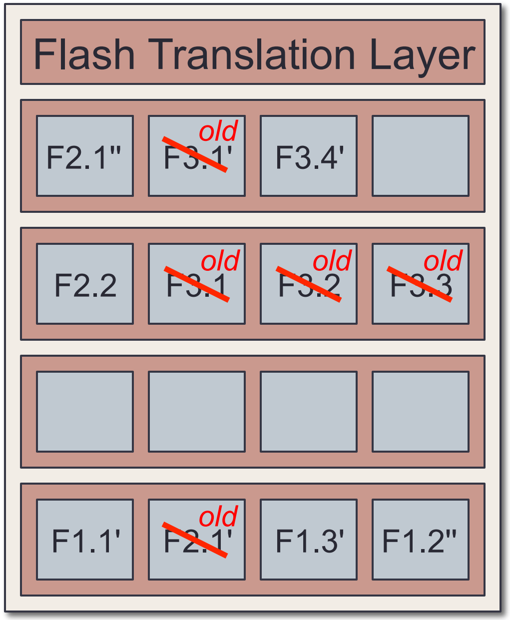
SSDs and File Deletion

- Problem: for most file system formats, file deletion doesn't actually touch the blocks in the file themselves!
  - File systems try to avoid this anyway, because storage I/O is slow!
  - Want to only update directory entry and other bookkeeping data, and we want this to be as efficient as possible
- Example: File F3 is deleted from the SSD
  - SSD will only see the block with the directory entry change, and maybe a few other blocks
- The SSD has no idea that file F3's data no longer needs to be preserved
  - e.g. if the SSD decides to erase bank 2, it will still move F3.2 and F3.3 to other cells, even though the OS and the users don't care!



SSDs, File Deletion and TRIM

- To deal with this, SSDs introduced the TRIM command
  - (TRIM is not an acronym)
- When the filesystem is finished with certain logical blocks, it can issue a TRIM command to inform the SSD that the data in those blocks can be discarded
- Previous example: file F3 is deleted
  - The OS can issue a TRIM command to inform SSD that all associated blocks are now unused
- TRIM allows the SSD to manage its cells much more efficiently
  - Greatly reduces write magnification issues
  - Helps reduce wear on SSD memory cells



SSDs, File Deletion and TRIM (2)

- Still a few issues to resolve with TRIM at this point
- Biggest one is TRIM wasn't initially a queued command
  - Couldn't include TRIM commands in a mix of other read/write commands being sent to the device
  - TRIM must be performed separately, in isolation of other operations
- TRIM must be issued in a batch-mode way, when it won't interrupt other work
  - e.g. can't issue TRIM commands immediately after each delete operation
- This was fixed in SATA 3.1 specification
  - A queued version of TRIM was introduced
- Another issue: not all OSes/filesystems support TRIM (or not enabled by default)