CSE 120
Principles of Operating Systems

Fall 2024

Lecture 10: Paging

Geoffrey M. Voelker

Lecture Overview

We'll cover more 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

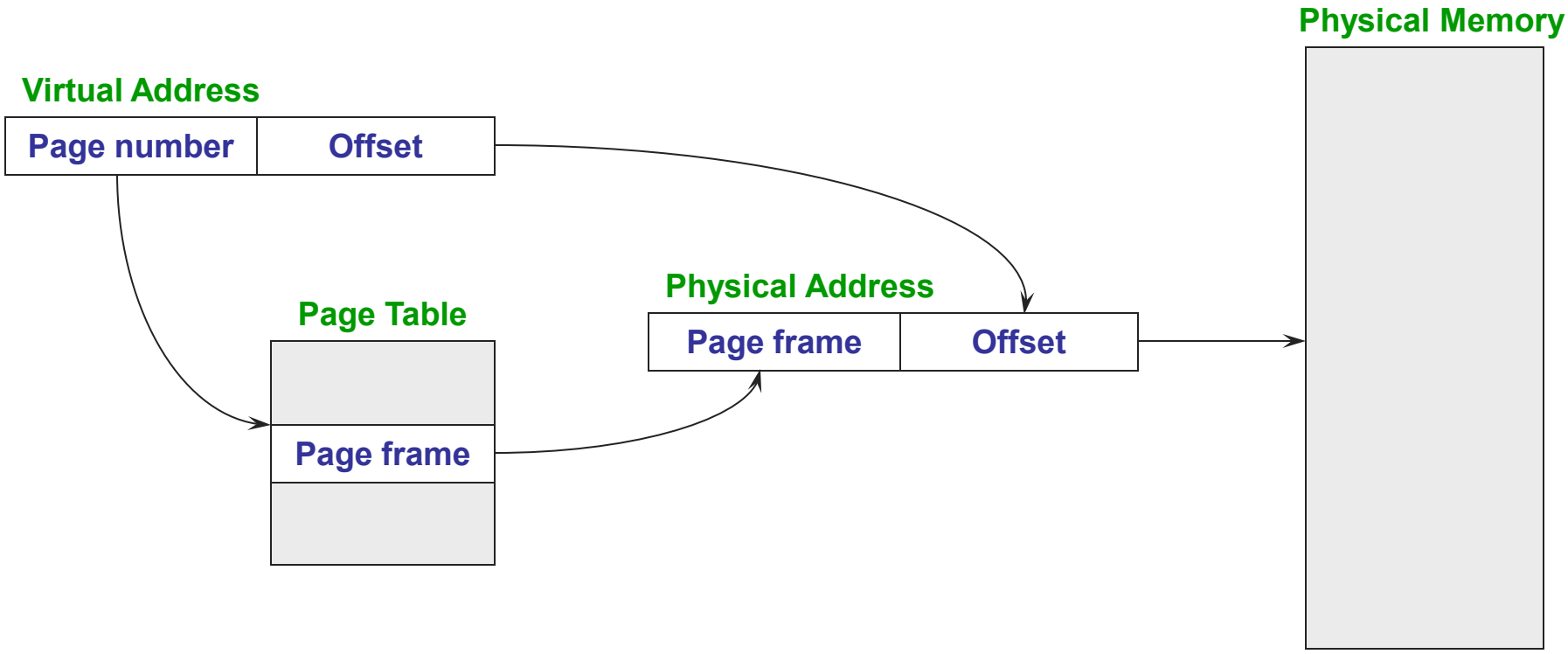
Managing Page Tables

- Last lecture we computed the size of the page table for a 32-bit address space w/ 4K pages to be 4MB
  - This is too much overhead for each process
- How can we reduce this overhead?
  - Observation: Only need to map the portion of the address space actually being used (a fraction of entire addr space)
- How do we only map what is being used?
  - Can dynamically extend page table…
  - Does not work if addr space is sparce (internal fragmentation)
- Use another level of indirection: two-level page tables

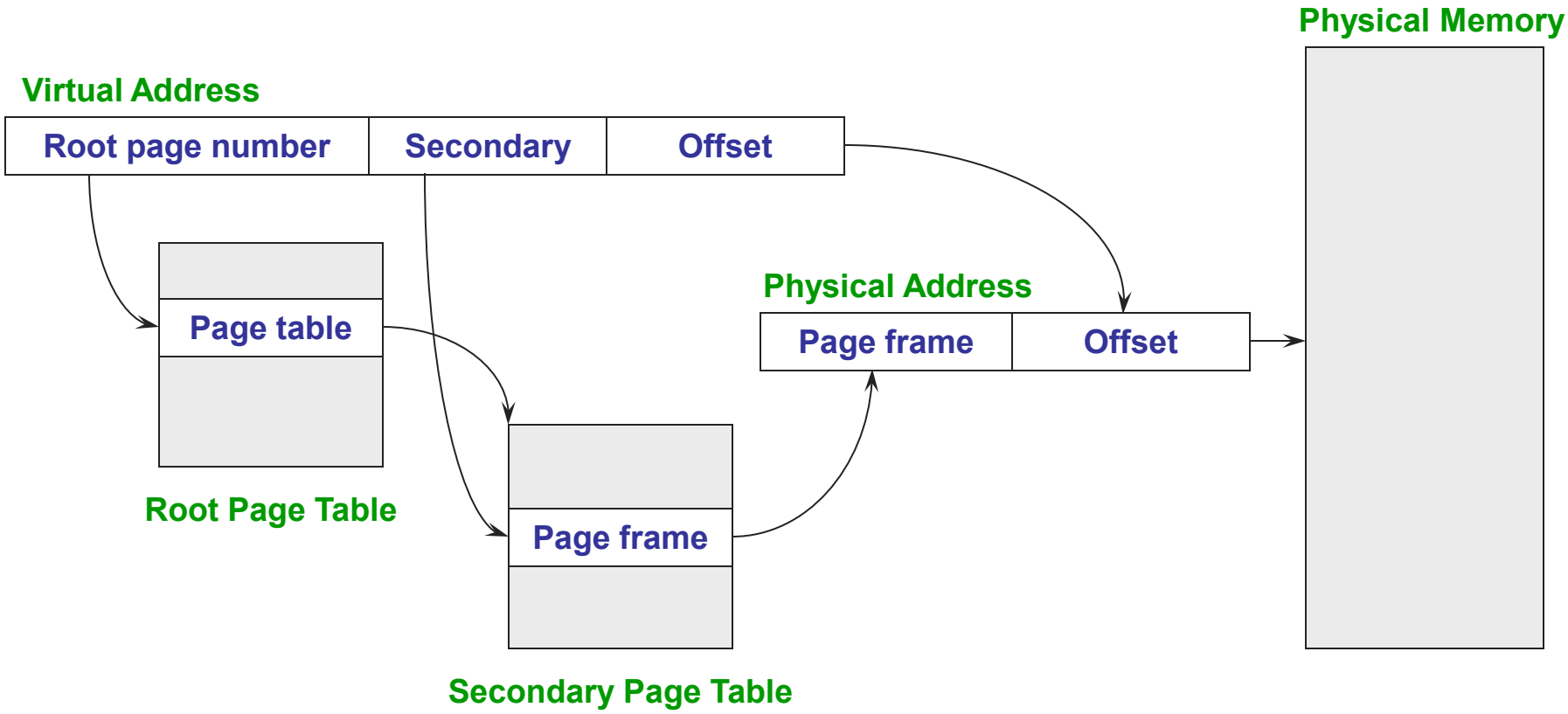
Two-Level Page Tables

- Two-level page tables for 32-bit address spaces
  - Virtual addresses (VAs) have three parts:
    - Root page number, secondary page number, and offset
  - Root page table maps VAs to secondary page table
  - Secondary page table maps page number to physical page
  - Offset indicates where in physical page address is located

Single-Level Page Tables



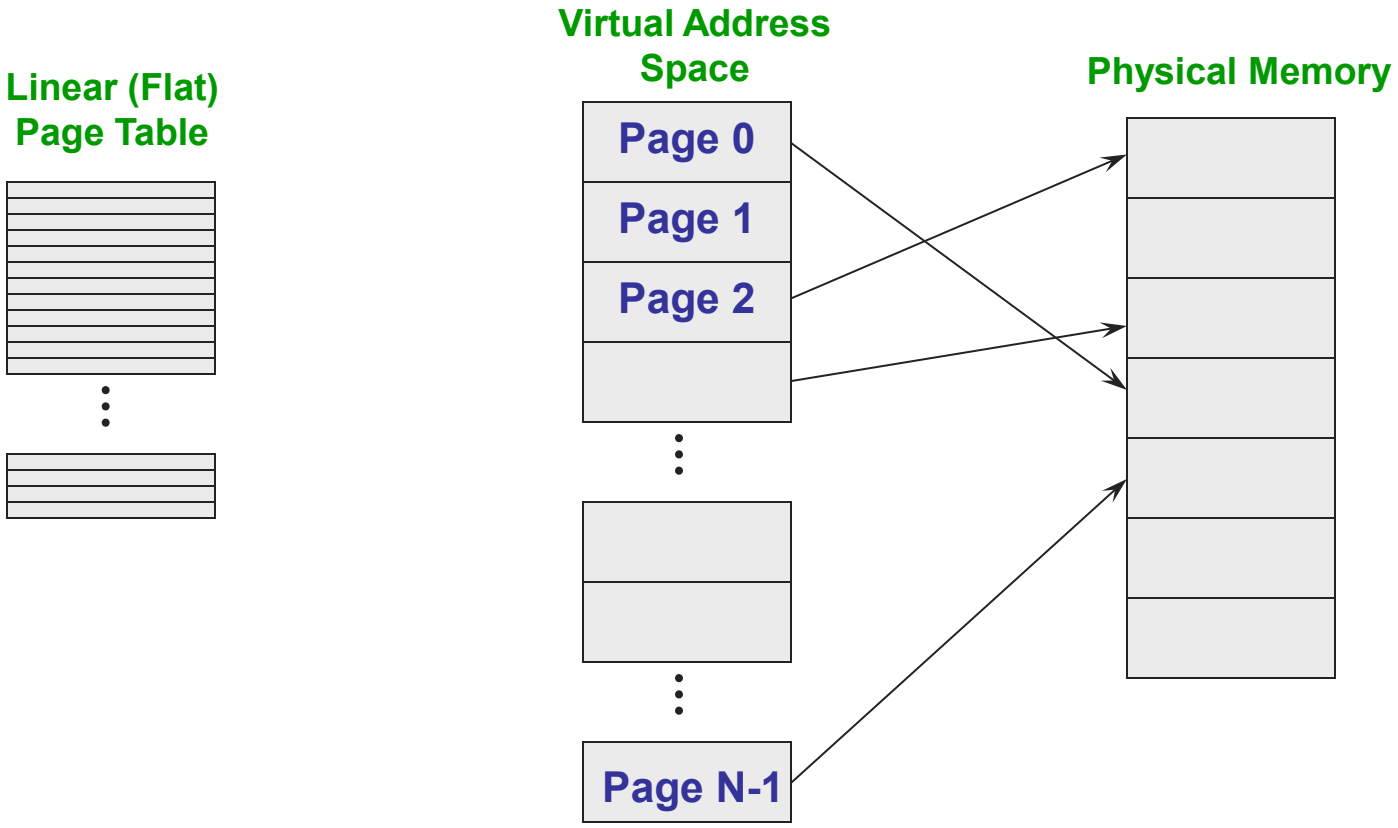
Two-Level Page Tables



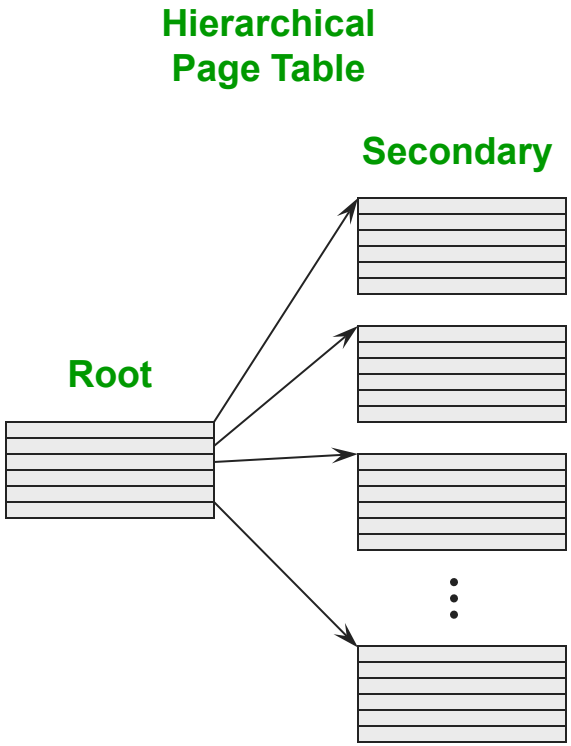
Two-Level Page Tables

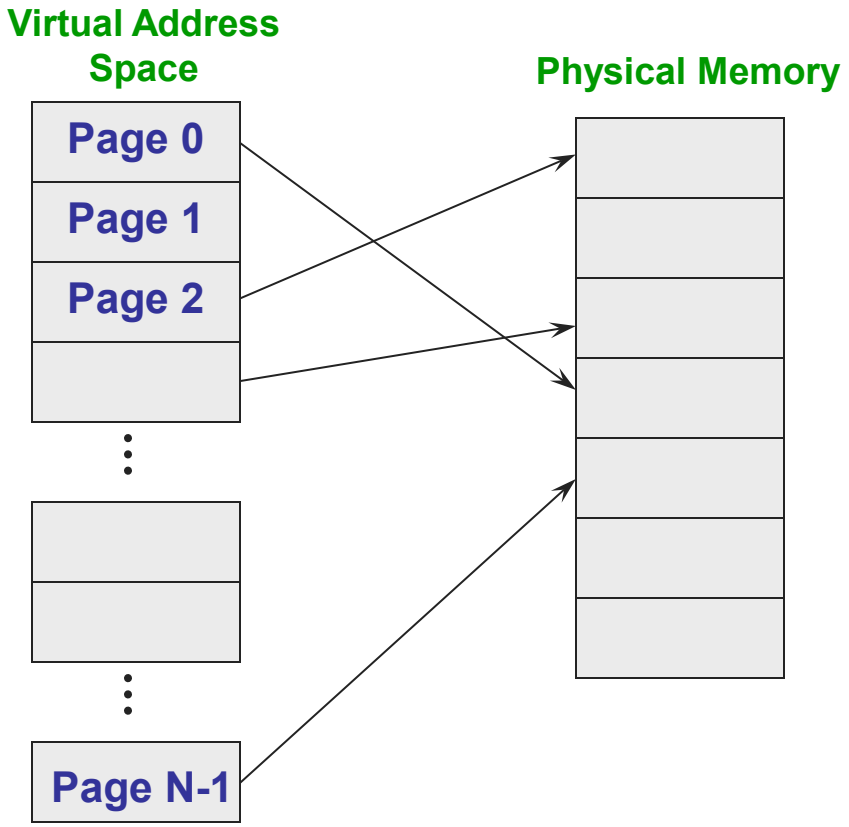
- Two-level page tables for 32-bit address spaces
  - Virtual addresses (VAs) have three parts:
    - » Root page number, secondary page number, and offset
  - Root page table maps VAs to secondary page table
  - Secondary page table maps page number to physical page
  - Offset indicates where in physical page address is located
- Example
  - 4K pages, 4 bytes/PTE
  - How many bits in offset? 4K = 12 bits
  - Want root page table in one page: 4K/4 bytes = 1K entries
  - Hence, 1K secondary page tables. How many bits?
  - Root (1K) = 10, offset = 12, inner = 32 – 10 – 12 = 10 bits

Page Table Evolution

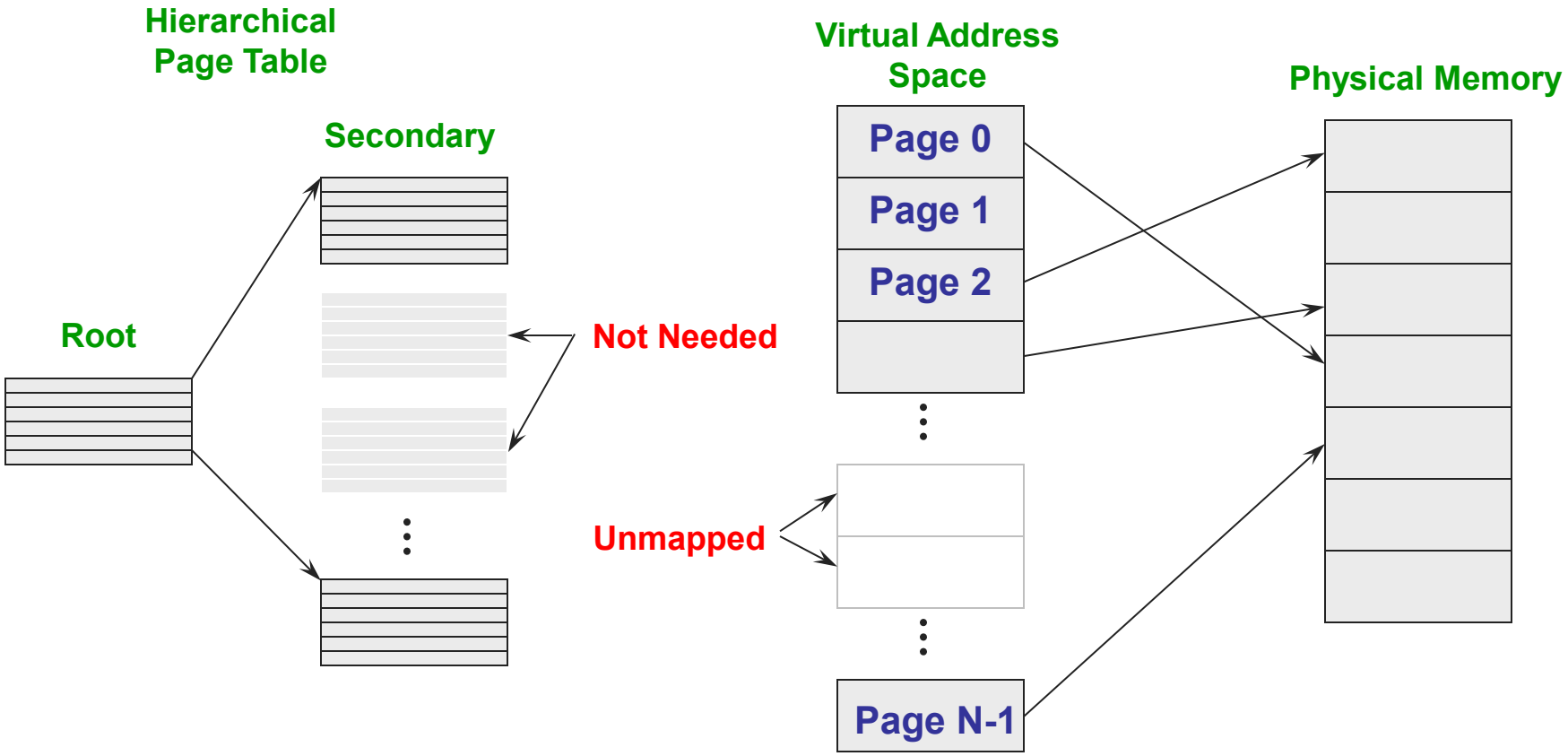


Page Table Evolution

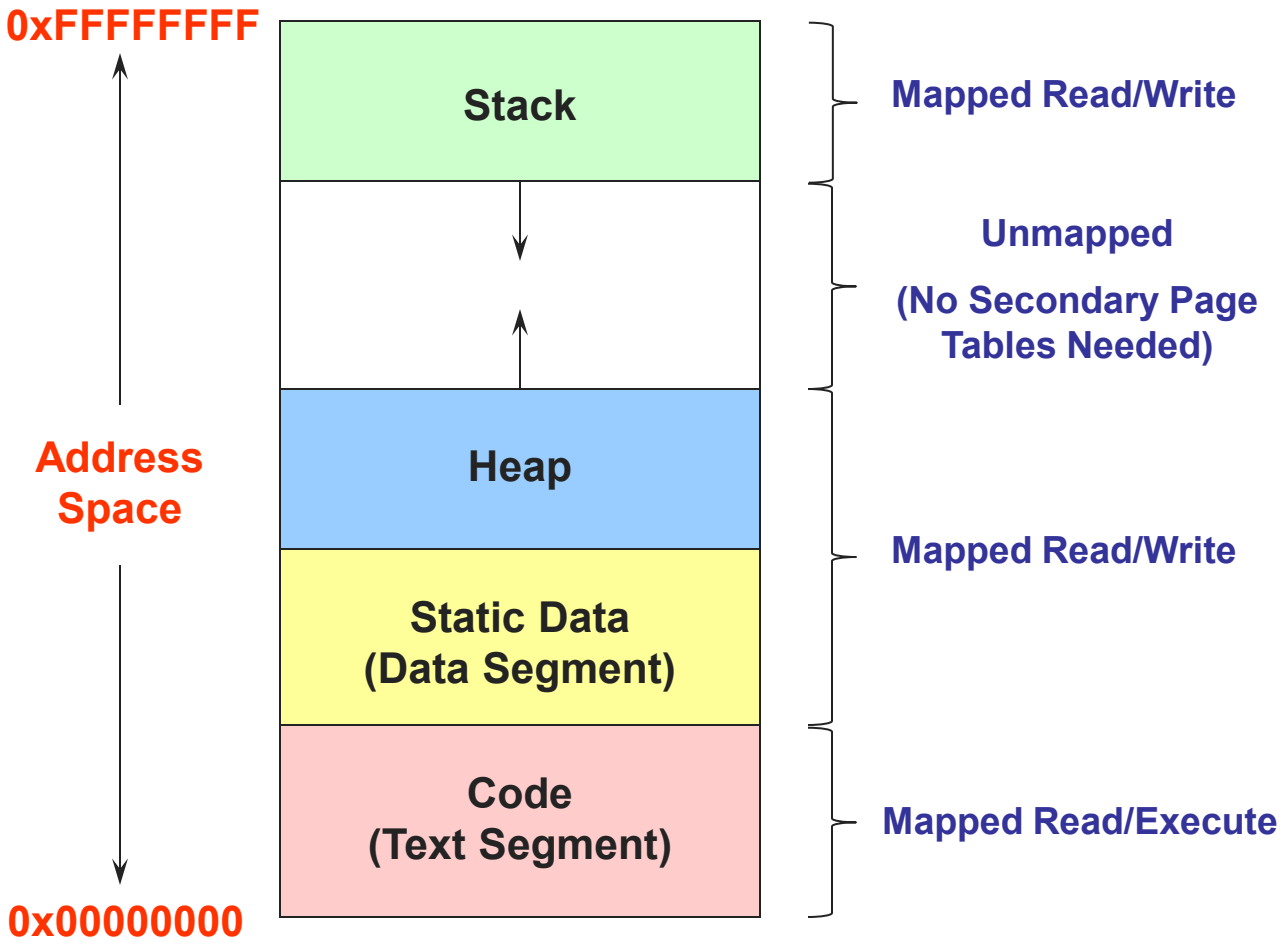




Page Table Evolution



Simple Address Space



Heap

- The heap is the region of address space for dynamic memory allocation (malloc, new)
  - "Raw" address space region managed by malloc library
  - End of the heap is known as the "break" (Unix)
- Processes can ask the OS to grow their heap
  - brk/sbrk: system call to change the heap boundary
  - malloc library calls brk/sbrk when it runs out of space
  - brk allocates more PTEs in the page table
  - Physical pages allocated on demand like other VM regions

Addressing Page Tables

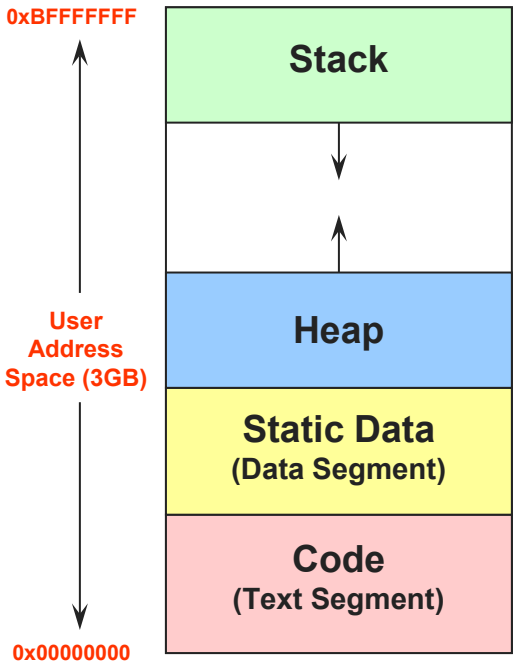
Where do we store page tables (which address space)?

- Physical memory
  - Easy to address, no translation required
  - But, allocated page tables consume memory for lifetime of VAS
- Virtual memory (OS virtual address space)
  - Cold (unused) page table pages can be paged out to disk
  - But addressing page tables requires translation
  - How do we stop recursion?
  - Do not page the outer page table (called wiring)
- If we're going to page the page tables, might as well page the entire OS address space, too
  - Need to wire special code and data (fault, interrupt handlers)

Kernel Address Space

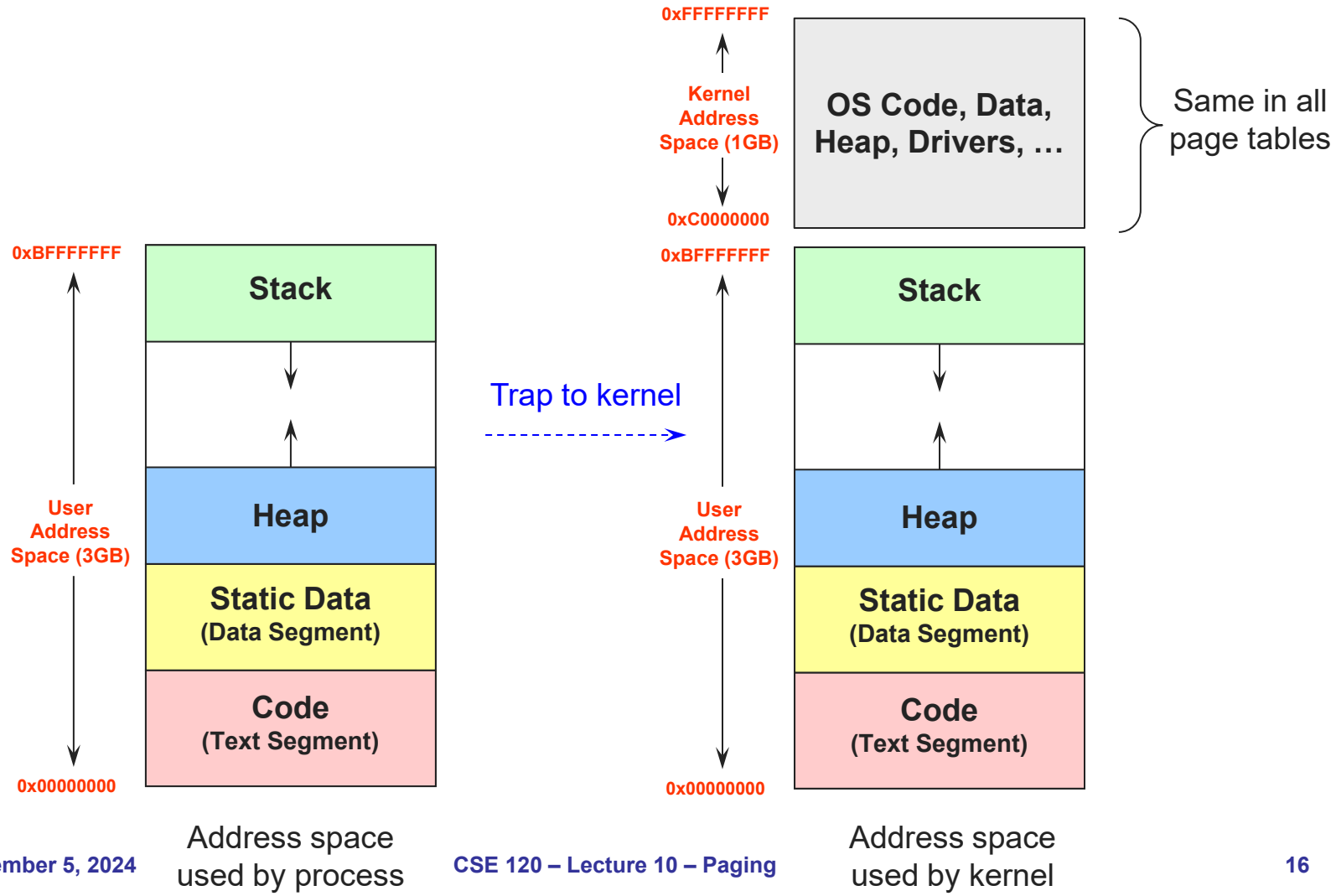
- Wait…how does the OS virtual address space work?
- We have talked about it as a separate address space
- But it is typically implemented as an extension of the user-level process address space
  - The bottom portion is for the user-level process
  - The top portion is for the operating system/kernel
  - VMS, early Unix: user 2GB, kernel 2GB (32-bit)
  - Linux, Windows: user 3GB, kernel 1GB (32-bit)

Process Address Space



Address space used by process

Kernel Address Space



Kernel Address Space

- When CPU is in user mode, a process can only access the user-level portion
- When CPU is in kernel mode, the OS can access the entire region
- This arrangement is very convenient for the OS
  - The OS can access any memory in the user-level portion of the current process (e.g., copying system call arguments)
  - But the OS region is protected from the process
- As a result, the OS is mapped into every process
  - The upper portion of every process address space is the OS
  - Context switching effectively just switches the bottom portion
- Can use same page table, or an extended copy (KPTI)

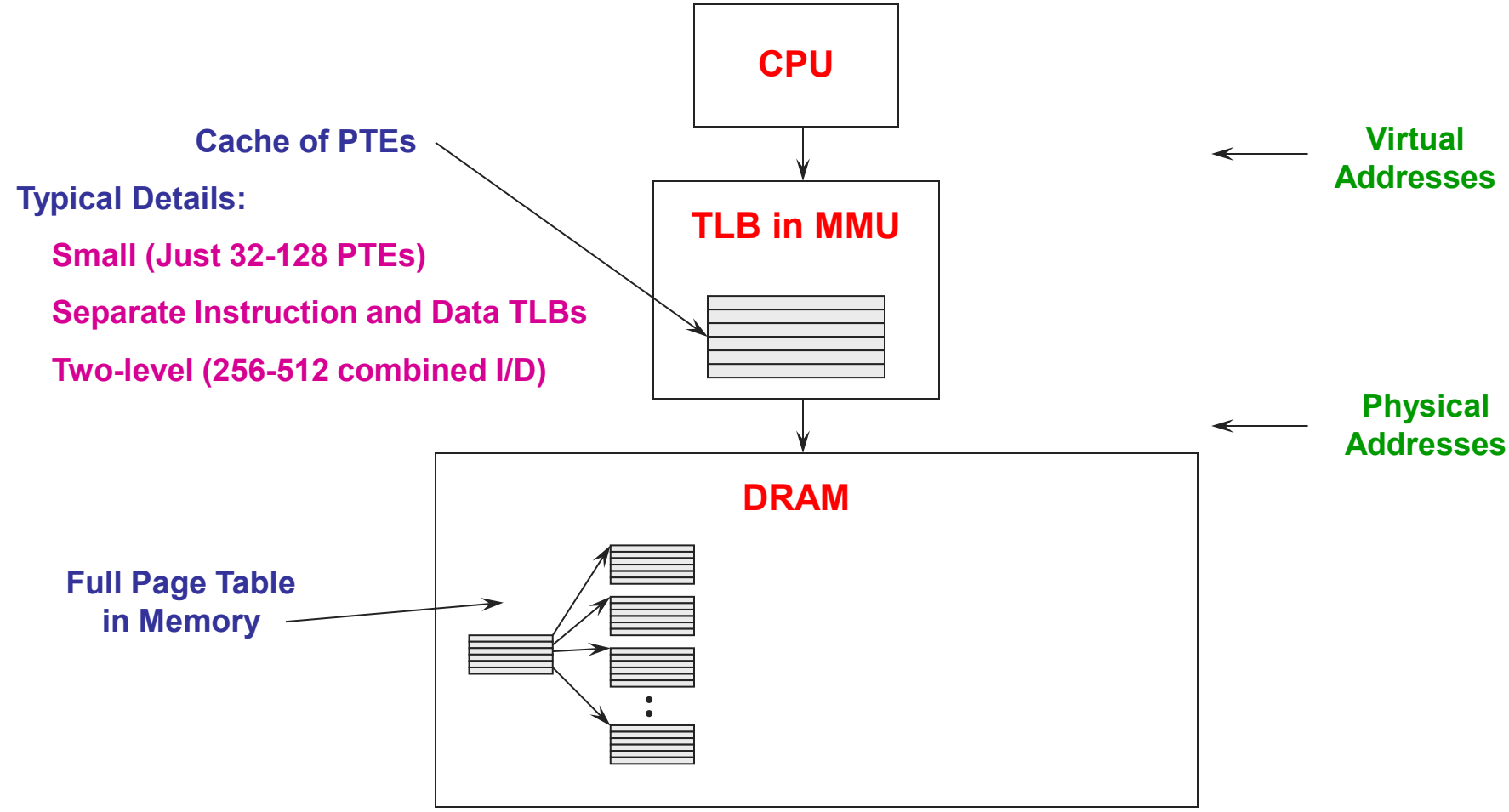
Efficient Translations

- Our original page table scheme already doubled the cost of doing memory lookups
  - One lookup into the page table, another to fetch the data
- Now two-level page tables triple the cost!
  - Two lookups into the page tables, a third to fetch the data
  - Worse, 64-bit architectures support 4+-level page tables
  - And this assumes the page table is in memory
- How can we use paging but avoid the time overhead of virtual address translation?
  - Cache translations in hardware
  - Translation Lookaside Buffer (TLB)
  - TLB managed by Memory Management Unit (MMU)

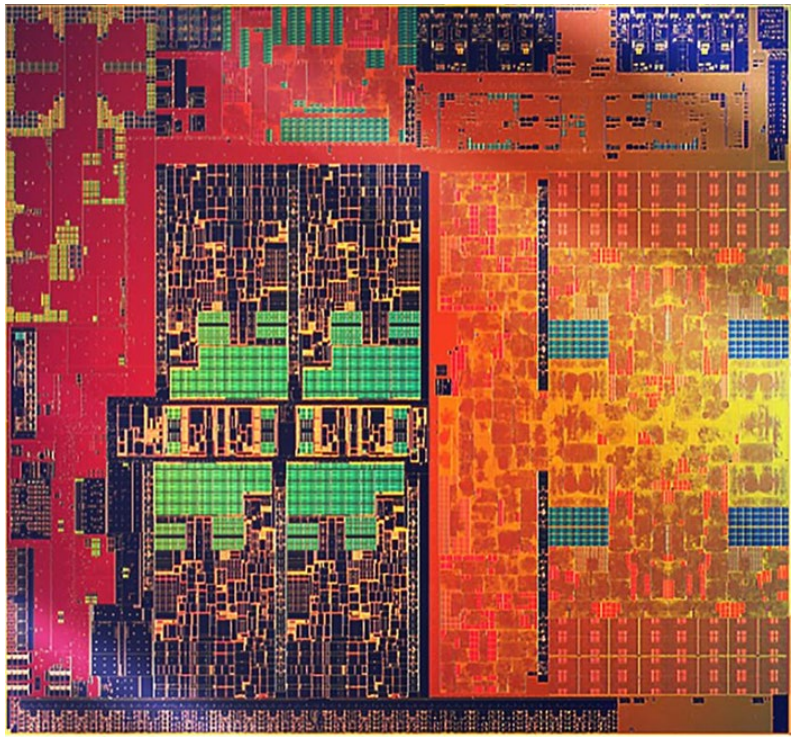
TLBs

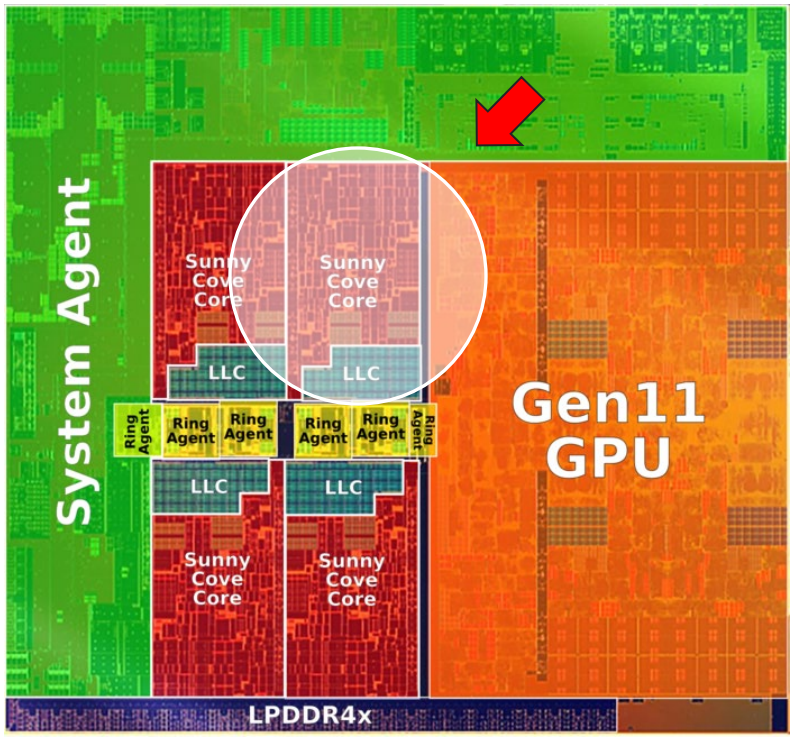
- Translation Lookaside Buffers
  - Translate virtual page #s into PTEs (not physical addrs)
  - Can be done in a single machine cycle
- TLBs implemented in hardware
  - Fully associative cache (all entries looked up in parallel)
  - Cache tags are virtual page numbers
  - Cache values are PTEs (entries from page tables)
  - With PTE + offset, can directly calculate physical address
- TLBs exploit locality
  - Processes only use a handful of pages at a time
    - 32-128 entries/pages (128-512K)
    - Only need those pages to be “mapped” by TLB
  - Hit rates are therefore very important

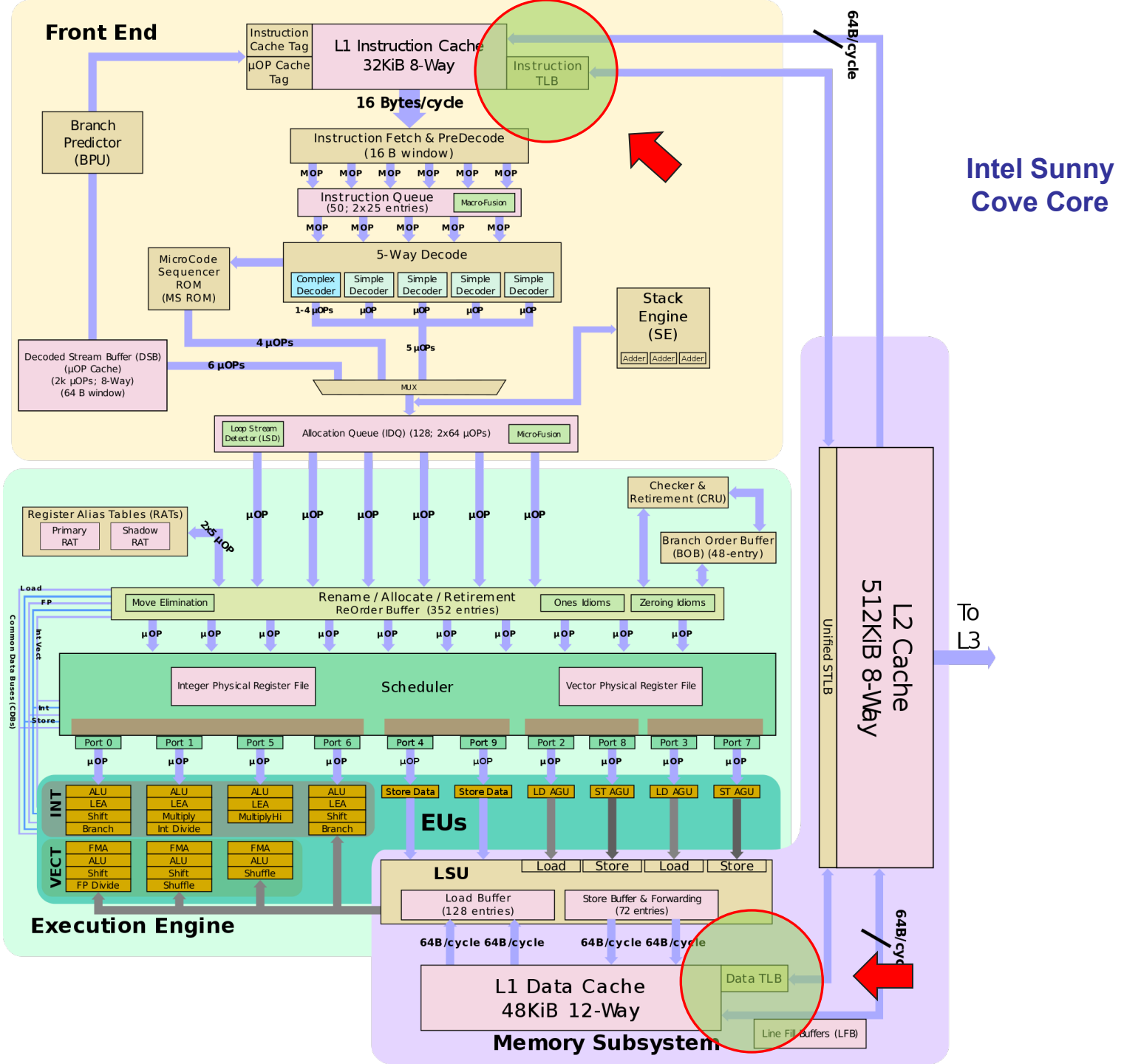
TLBs


CPU
Cache of PTEs
Typical Details:
Small (Just 32-128 PTEs)
Separate Instruction and Data TLBs
Two-level (256-512 combined I/D)
TLB in MMU
Virtual Addresses
Physical Addresses
DRAM
Full Page Table in Memory

Intel Ice Lake






Front End
Instruction Cache Tag
µOP Cache Tag
L1 Instruction Cache 32KiB 8-Way
Instruction TLB
64B/cycle
16 Bytes/cycle
Branch Predictor (BPU)
Instruction Fetch & PreDecode (16 B window)
MOP
Instruction Queue (50; 2x25 entries)
Macro-Fusion
MicroCode Sequencer ROM (MS ROM)
5-Way Decode
Complex Decoder
Simple Decoder
1-4 µOPs
µOP
Stack Engine (SE)
Adder
Decoded Stream Buffer (DSB) (µOP Cache) (2k µOPs; 8-Way) (64 B window)
4 µOPs
6 µOPs
5 µOPs
MUX
Loop Stream Detector (LSD)
Allocation Queue (IDQ) (128; 2x64 µOPs)
MicroFusion
Intel Sunny Cove Core
Register Alias Tables (RATs)
Primary RAT
Shadow RAT
2-5 µOP
Checker & Retirement (CRU)
Branch Order Buffer (BOB) (48-entry)
Move Elimination
Rename / Allocate / Retirement
ReOrder Buffer (352 entries)
Ones Idioms
Zeroing Idioms
Load
FP
Int Vect
Int
Store
Common Data Buses (CDBs)
Integer Physical Register File
Scheduler
Vector Physical Register File
Port 0
Port 1
Port 5
Port 6
Port 4
Port 9
Port 2
Port 8
Port 3
Port 7
INT
ALU
LEA
Shift
Branch
Multiply
Int Divide
MultiplyHi
VECT
FMA
Shuffle
FP Divide
Store Data
LD AGU
ST AGU
EUs
LSU
Load
Store
Load Buffer (128 entries)
Store Buffer & Forwarding (72 entries)
64B/cycle 64B/cycle
Execution Engine
L1 Data Cache 48KiB 12-Way
Data TLB
Line Fill Buffers (LFB)
Memory Subsystem
Unified STLB
L2 Cache 512KiB 8-Way
To L3

Managing TLBs

- Address translations for most instructions are handled using the TLB
  - >99% of translations, but there are misses (TLB miss)…
- Who places translations into the TLB (loads the TLB)?
  - Hardware (Memory Management Unit) [x86, ARM, RISC-V]
    - Knows where page tables are in main memory
    - OS maintains tables, HW accesses them directly
    - Tables have to be in HW-defined format (inflexible)
  - Software loaded TLB (OS) [MIPS, Alpha, Sparc, PowerPC]
    - TLB faults to the OS, OS finds appropriate PTE, loads it in TLB
    - Must be fast (but still 20-200 cycles)
    - CPU ISA has instructions for manipulating TLB
    - Tables can be in any format convenient for OS (flexible)

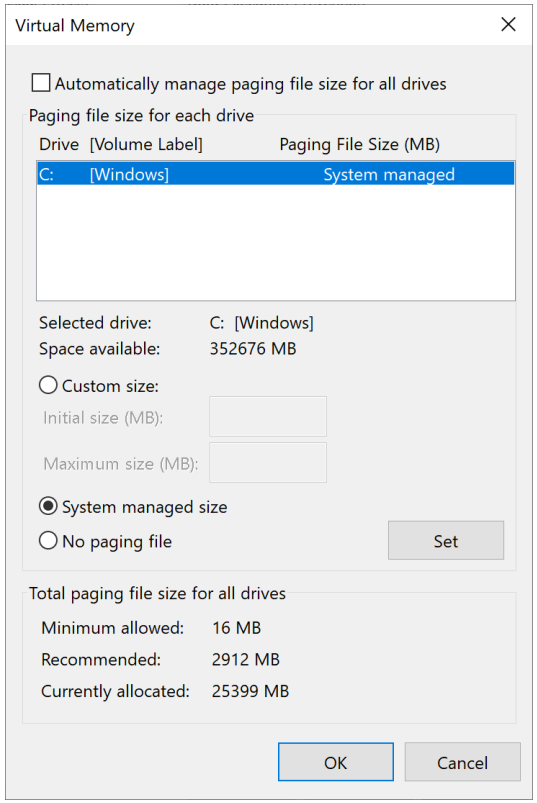
Managing TLBs (2)

- OS ensures that TLB and page tables are consistent
  - When it changes the protection bits of a PTE, it needs to invalidate the PTE if it is in the TLB
- Reload TLB on a process context switch
  - Invalidate all entries (causes overhead to fill it again)
  - Why? What is one way to fix it?
- When the TLB misses and a new PTE has to be loaded, a cached PTE must be evicted
  - Choosing PTE to evict is called the TLB replacement policy
  - Implemented in hardware, often simple (e.g., Last-Not-Used)

Paged Virtual Memory

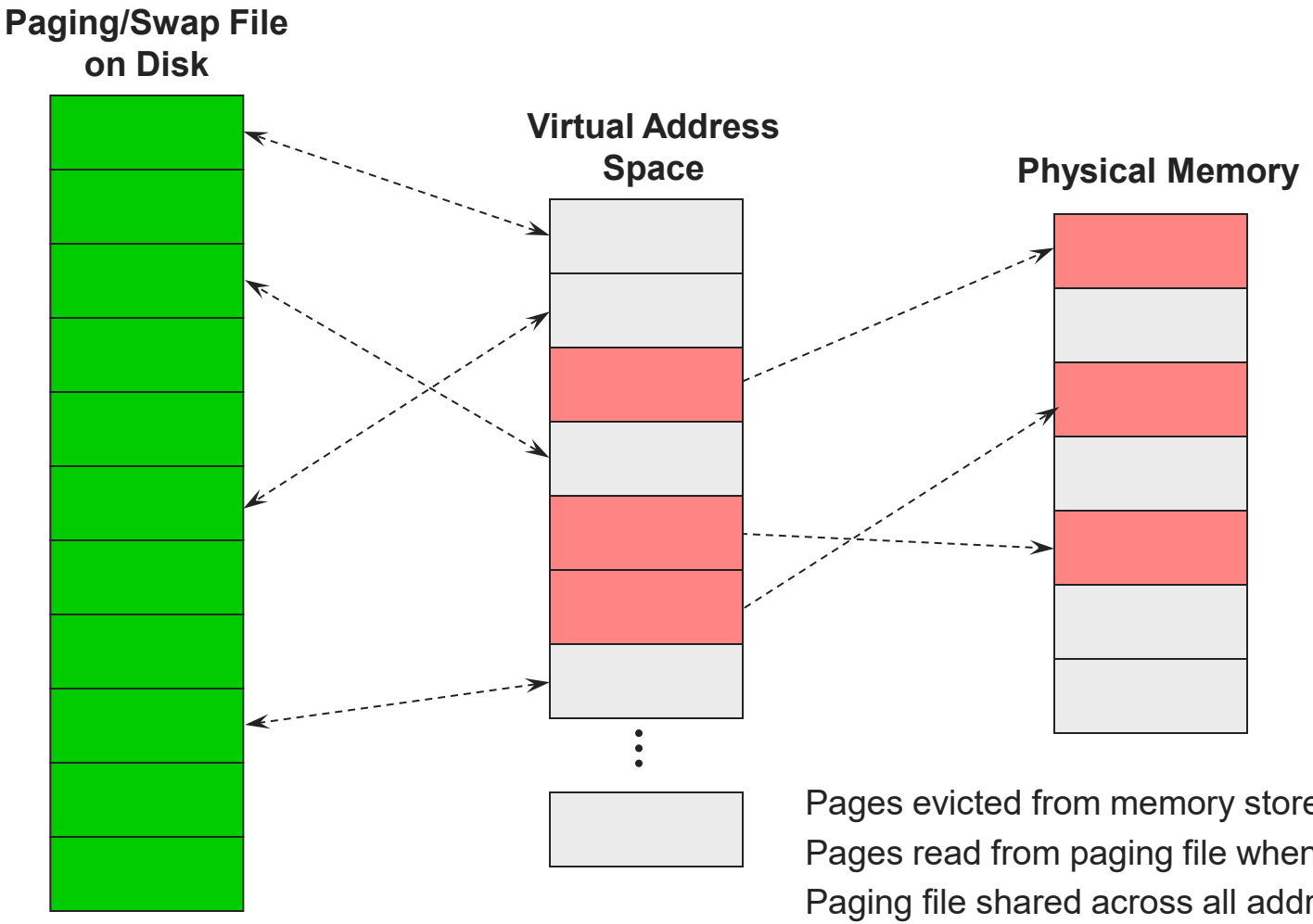
- We've mentioned before that pages can be moved between memory and disk
  - This feature is called demand paging
- OS uses main memory as a page cache of all the data allocated by processes in the system
  - Initially, pages are allocated from memory
  - When memory fills up, allocating a page in memory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

Windows




```
C:\>ls -l pagefile.sys
-rw-r----- 1 Unknown+User Unknown+Group 27173613568 Nov 11 13:58 pagefile.sys
```

Paged Virtual Memory



Pages evicted from memory stored in paging file

Pages read from paging file when accessed again

Paging file shared across all address spaces

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 exception)
  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 instruction that caused the page fault
- 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

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 in TLB, 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 CPU 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
  - Assuming page tables are in OS virtual address space
  - Not a problem if tables are in physical memory
  - Solve by wiring page tables for OS address space
- 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 fault
  - Read/write/execute – operation not permitted on page
  - Invalid – virtual page not mapped, 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 use for other purposes (e.g., copy on write, mapped files)
  - Invalid
    - » Virtual page not mapped in address space
      - OS sends fault to process (e.g., segmentation fault)
    - » Page not in physical memory
      - Used to implement demand paging
      - 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
  - How are we going to coordinate access to shared data?

NAME top

objects

SYNOPSIS top

```
#include <sys/mman.h>
#include <sys/stat.h>        /* For mode constants */
#include <fcntl.h>           /* For O_* constants */

int shm_open(const char *name, int oflag, mode_t mode);

int shm_unlink(const char *name);
```

Link with *-lrt*.

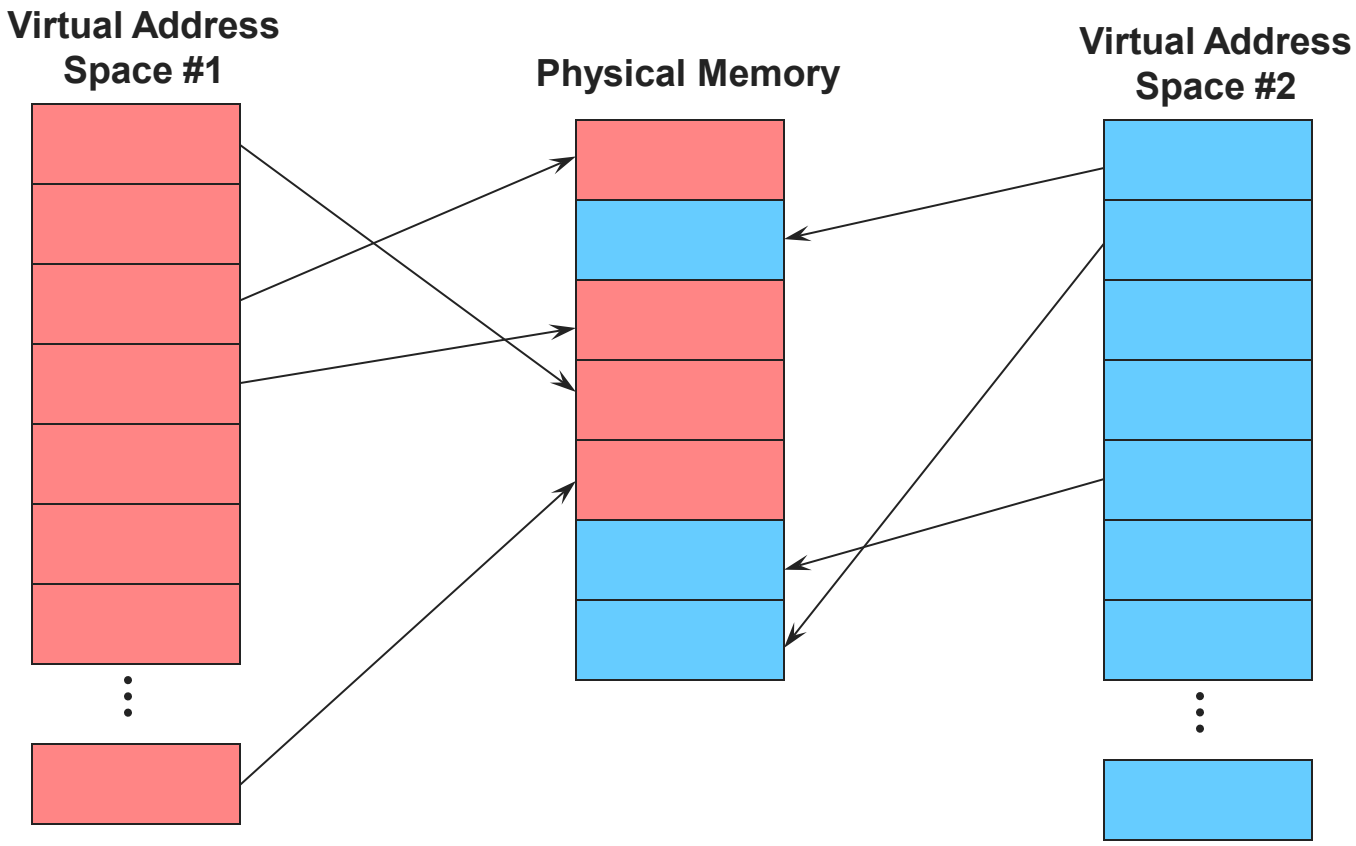
DESCRIPTION top

shm_open() creates and opens a new, or opens an existing, POSIX shared memory object. A POSIX shared memory object is in effect a handle which can be used by unrelated processes to mmap(2) the same region of shared memory. The **shm_unlink**() function performs the converse operation, removing an object previously created by **shm_open**().

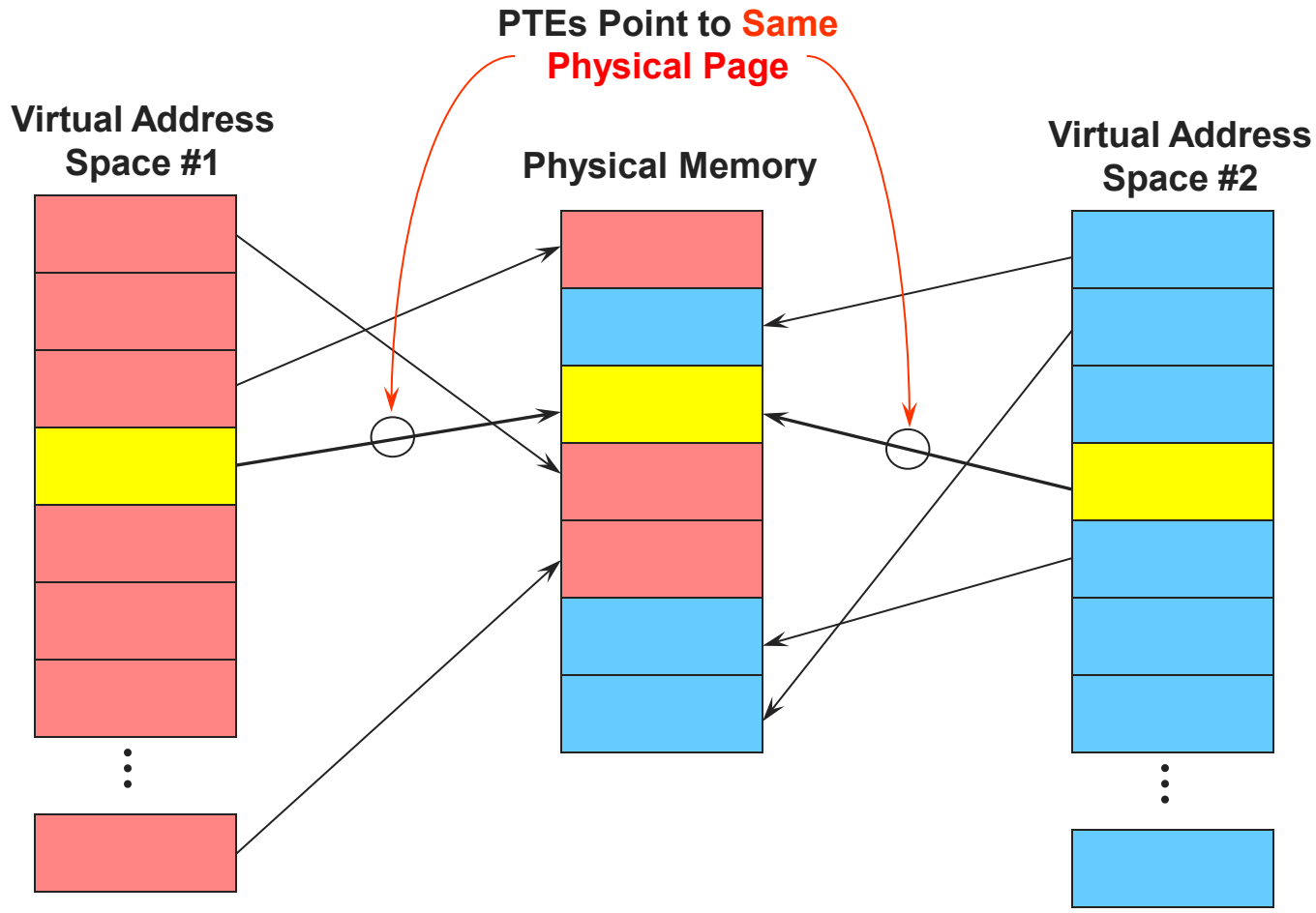
Sharing (2)

- 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
- Can map shared memory at same or different virtual addresses in each process address space
  - Different: Flexible (no address space conflicts), but pointers inside the shared memory segment are invalid
  - Same: Less flexible, but shared pointers are valid

Isolation: No Sharing



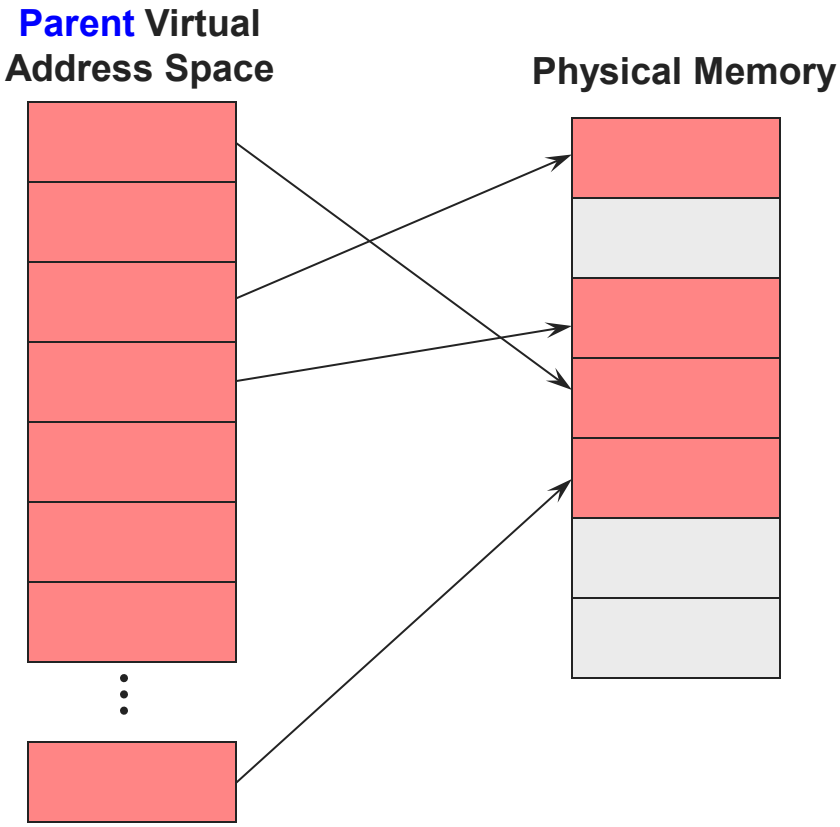
Sharing Pages



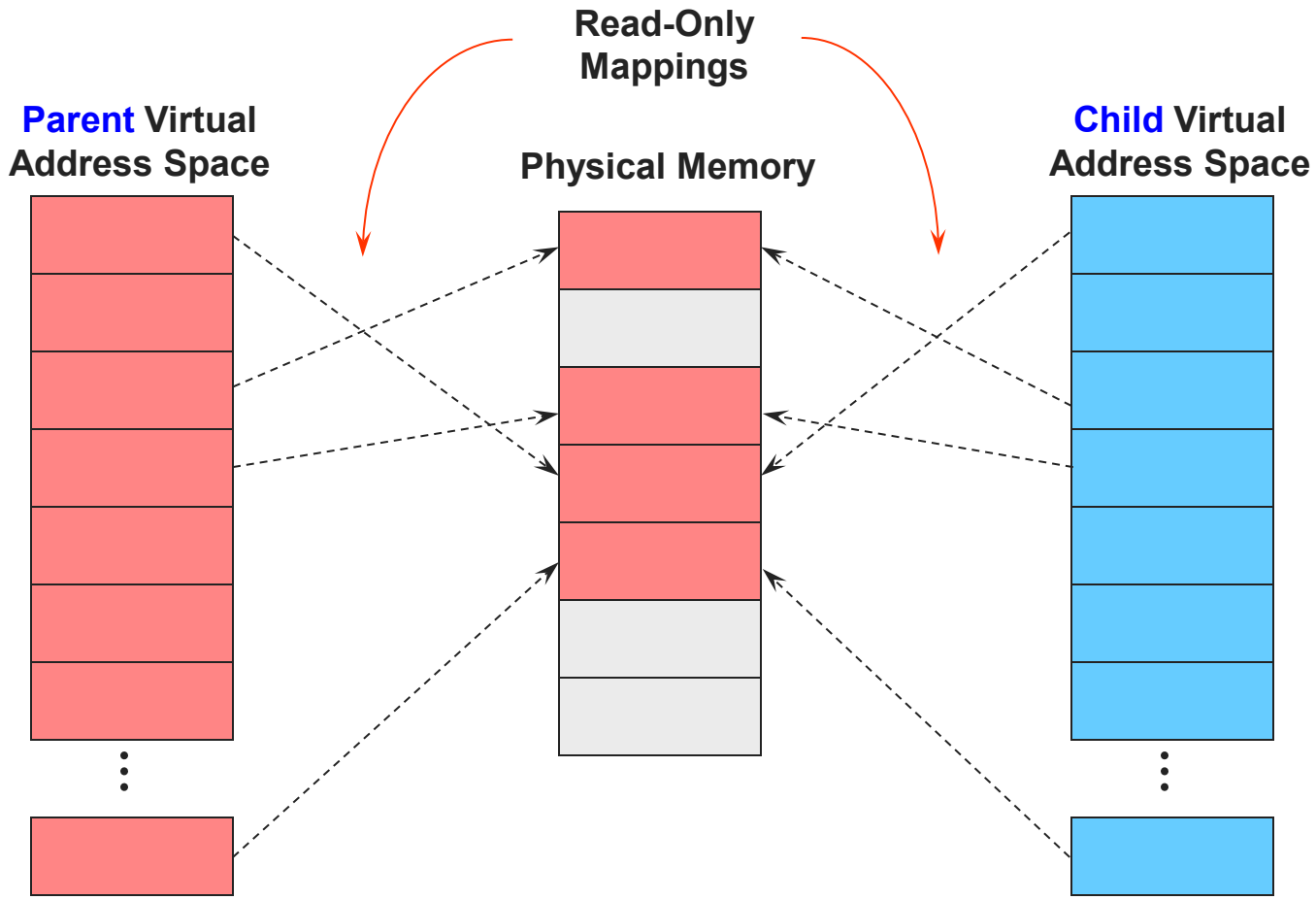
Copy on Write

- OSes spend a lot of time copying data
  - Entire address spaces to implement process 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()?

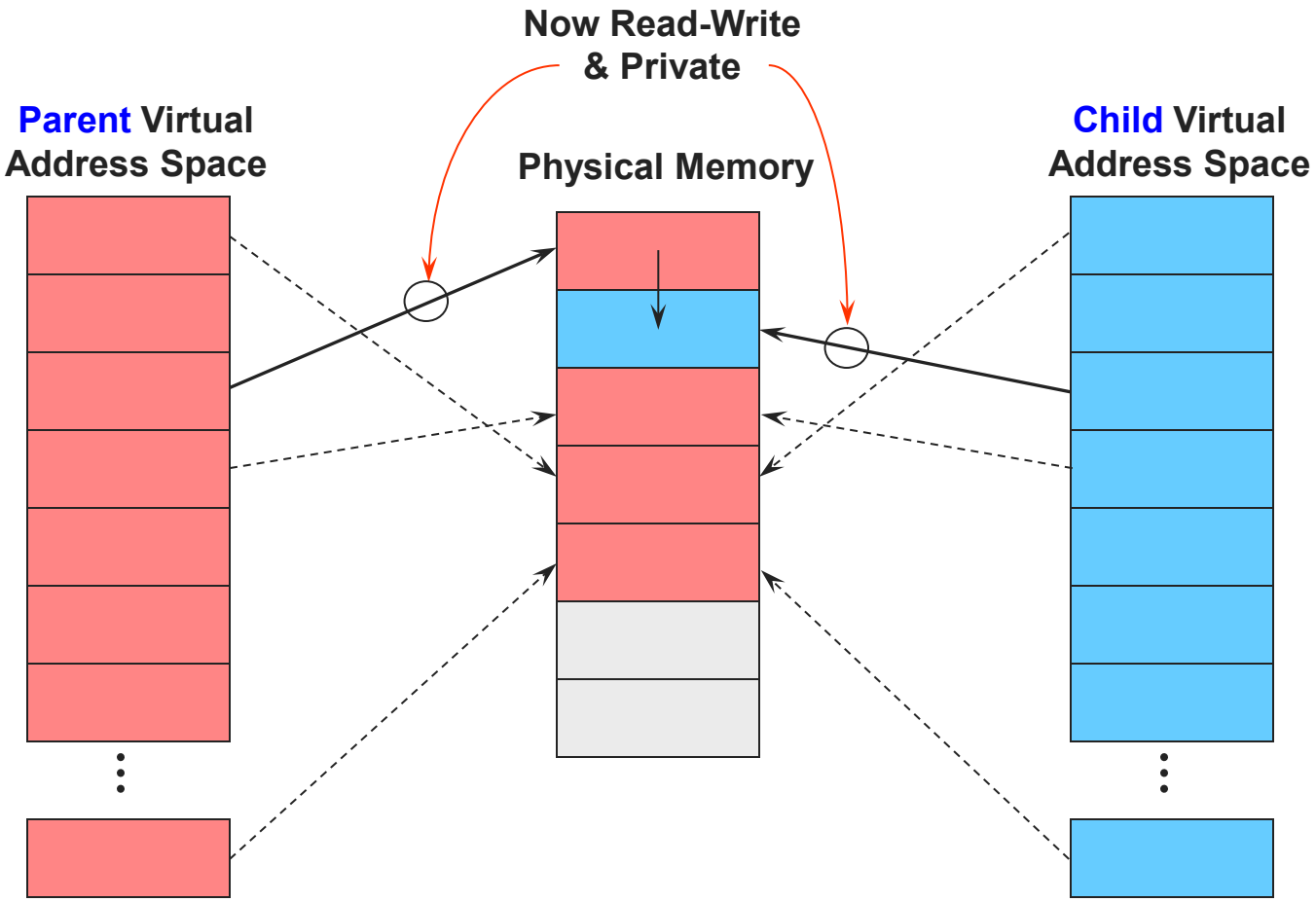
Copy on Write: Before Fork



Copy on Write: Fork



Copy on Write: On A Write



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
  - 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

NAME top

mmap, munmap - map or unmap files or devices into memory

SYNOPSIS top

```
#include <sys/mman.h>

void *mmap(void *addr, size_t length, int prot, int flags,
           int fd, off_t offset);
int munmap(void *addr, size_t length);
```

See NOTES for information on feature test macro requirements.

DESCRIPTION top

mmap() creates a new mapping in the virtual address space of the calling process. The starting address for the new mapping is specified in *addr*. The *length* argument specifies the length of the mapping (which must be greater than 0).

If *addr* is NULL, then the kernel chooses the (page-aligned) address at which to create the mapping; this is the most portable method of creating a new mapping. If *addr* is not NULL, then the kernel takes it as a hint about where to place the mapping; on Linux, the mapping will be created at a nearby page boundary. The address of the new mapping is returned as the result of the call.

The contents of a file mapping (as opposed to an anonymous mapping; see **MAP_ANONYMOUS** below), are initialized using *length* bytes starting at offset *offset* in the file (or other object) referred to by the

MapViewOfFile function

Maps a view of a file mapping into the address space of a calling process.

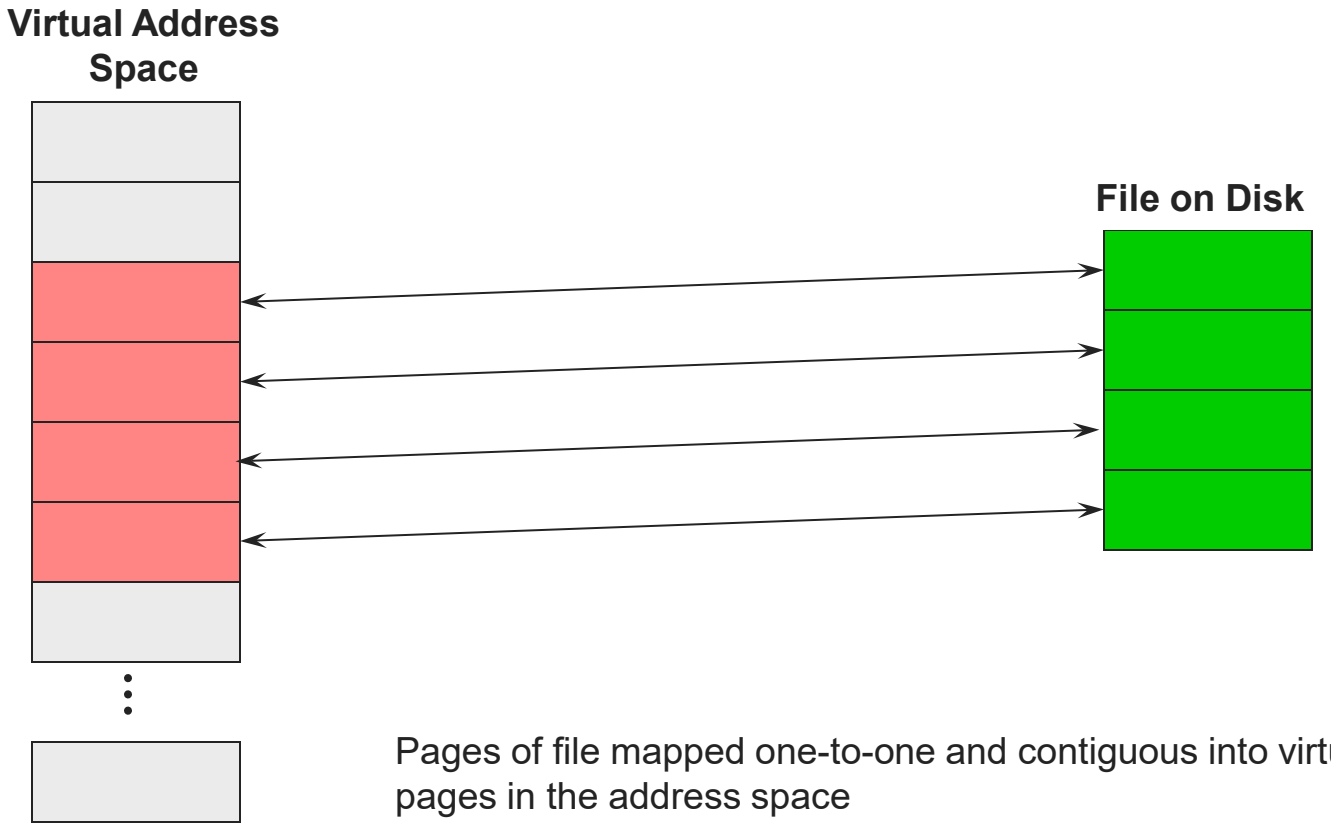
To specify a suggested base address for the view, use the MapViewOfFileEx function. However, this practice is not recommended.

Syntax

C++

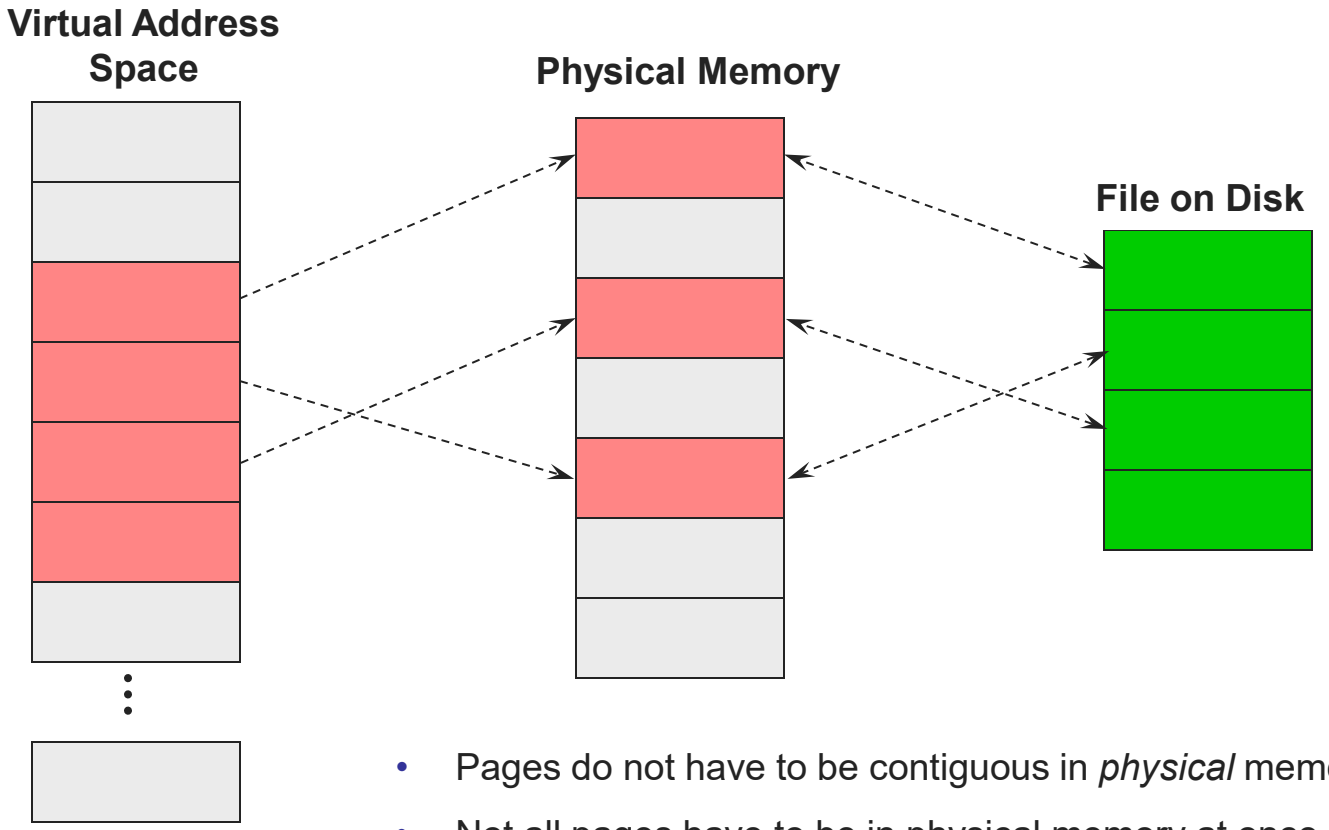
```
LPVOID WINAPI MapViewOfFile(
  _In_ HANDLE hFileMappingObject,
  _In_ DWORD  dwDesiredAccess,
  _In_ DWORD  dwFileOffsetHigh,
  _In_ DWORD  dwFileOffsetLow,
  _In_ SIZE_T dwNumberOfBytesToMap
);
```

Mapped Files



Pages of file mapped one-to-one and contiguous into virtual pages in the address space

Mapped Files



- Pages do not have to be contiguous in *physical* memory
- Not all pages have to be in physical memory at once

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.)

Program Loading

- Loading programs into a process uses memory mapping and copy-on-write
- Memory mapped files
  - Executable file is mapped into the address space
  - Faults will read from executable file
- Copy-on-write
  - Code regions mapped read-only (never need to copy)
  - Data regions from exe are mapped read/write
    - Only need to make copies if a process modifies a page
- Also applies to shared libraries
  - "Shared" → library mapped into multiple processes

Address Space Demo

$ cat /proc/self/maps

Executable File Mapped Copy-on-Write

```
9:19 (12) ~> cat /proc/self/maps
55d63f849000-55d63f851000 r-xp 00000000 08:02 12058626                   /bin/cat
55d63fa50000-55d63fa51000 r--p 00007000 08:02 12058626                   /bin/cat
55d63fa51000-55d63fa52000 rw-p 00008000 08:02 12058626                   /bin/cat
55d6406df000-55d640700000 rw-p 00000000 00:00 0                          [heap]
7f833a41b000-7f833a6f9000 r--p 00000000 08:02 8403808                    /usr/lib/locale/locale-archive
7f833a6f9000-7f833a8e0000 r-xp 00000000 08:02 10357928                   /lib/x86_64-linux-gnu/libc-2.27.so
7f833a8e0000-7f833aae0000 ---p 001e7000 08:02 10357928                   /lib/x86_64-linux-gnu/libc-2.27.so
7f833aae0000-7f833aae4000 r--p 001e7000 08:02 10357928                   /lib/x86_64-linux-gnu/libc-2.27.so
7f833aae4000-7f833aae6000 rw-p 001eb000 08:02 10357928                   /lib/x86_64-linux-gnu/libc-2.27.so
7f833aae6000-7f833aaea000 rw-p 00000000 00:00 0
7f833aaea000-7f833ab13000 r-xp 00000000 08:02 10355697                   /lib/x86_64-linux-gnu/ld-2.27.so
7f833ace0000-7f833ad04000 rw-p 00000000 00:00 0
7f833ad13000-7f833ad14000 r--p 00029000 08:02 10355697                   /lib/x86_64-linux-gnu/ld-2.27.so
7f833ad14000-7f833ad15000 rw-p 0002a000 08:02 10355697                   /lib/x86_64-linux-gnu/ld-2.27.so
7f833ad15000-7f833ad16000 rw-p 00000000 00:00 0
7fffd5064000-7fffd5085000 rw-p 00000000 00:00 0                          [stack]
7fffd5112000-7fffd5115000 r--p 00000000 00:00 0                          [vvar]
7fffd5115000-7fffd5117000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
9:19 (13) ~>
```

Address Range

Protection

File Backing Store

Shared Libs Mapped Copy-on-Write

Summary

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 time: Paging policies

Next time…

- Chapters 21-23