Jin-Soo Kim
(*jinsoo.kim@snu.ac.kr*)

Systems Software &
Architecture Lab.

Seoul National University

Fall 2020

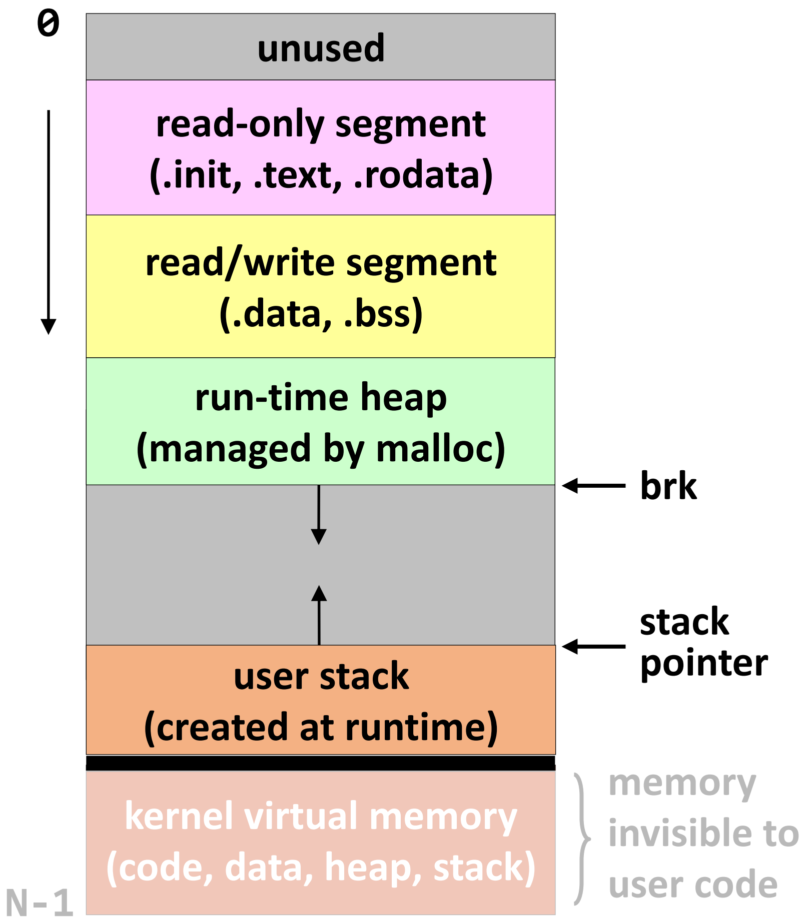
Virtual Memory

Virtual Memory: Goals

- Transparency
  - Processes should not be aware that memory is shared
  - Provides a convenient abstraction for programming (a large, contiguous space)
- ______________
  - Minimizes fragmentation due to variable-sized requests (space)
  - Gets some hardware support (time)
- Protection
  - Protect processes and the OS from another process
  - Isolation: a process can fail without affecting other processes
  - Cooperating processes can share portions of memory

(Virtual) Address Space

- Process' abstract view of memory
  - OS provides illusion of private address space to each process
  - Contains all of the memory state of the process
  - Static area
    - Allocated on `exec()`
    - Code & Data
  - Dynamic area
    - Allocated at runtime
    - Can grow or shrink
    - Heap & Stack

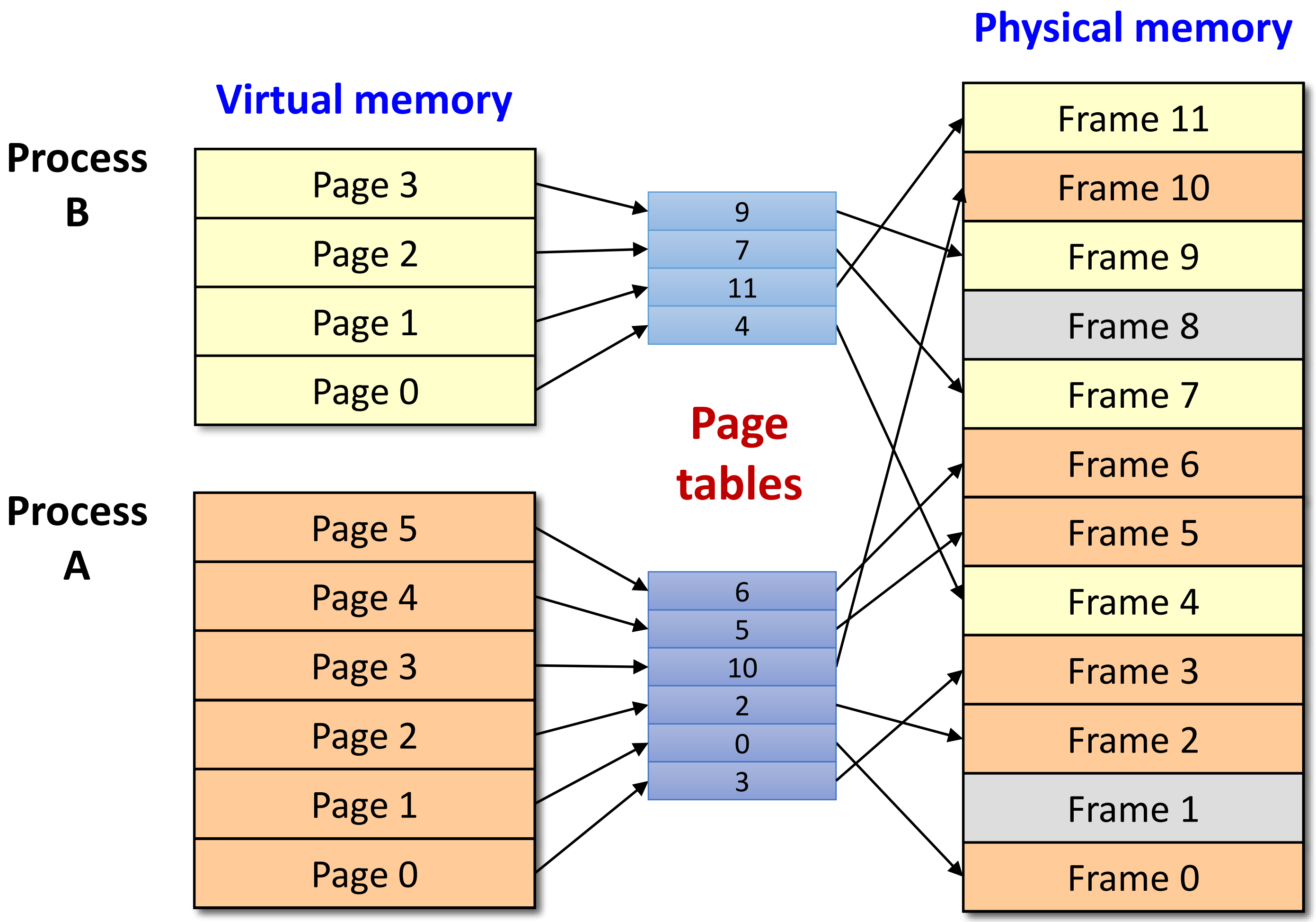


Paging

- Allows the physical address space of a process to be noncontiguous
  - Divide virtual memory into blocks of same size (pages)
  - Divide physical memory into fixed-size blocks (frames)
  - Page (or frame) size is power of 2 (typically 512B – 8KB)

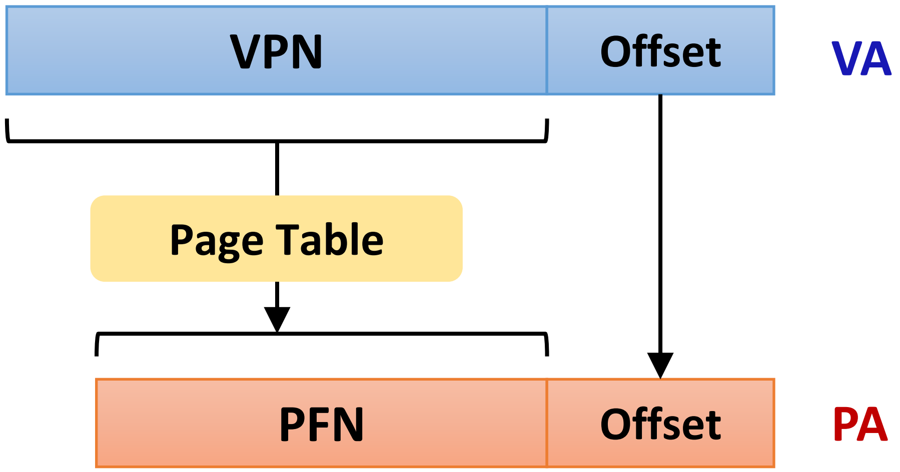
- Eases memory management
  - OS keeps track of all free frames
  - To run a program of size *n* pages, need to find *n* free frames and load the program
  - Set up a page table to translate virtual to physical addresses
  - No __________ fragmentation

Paging Overview



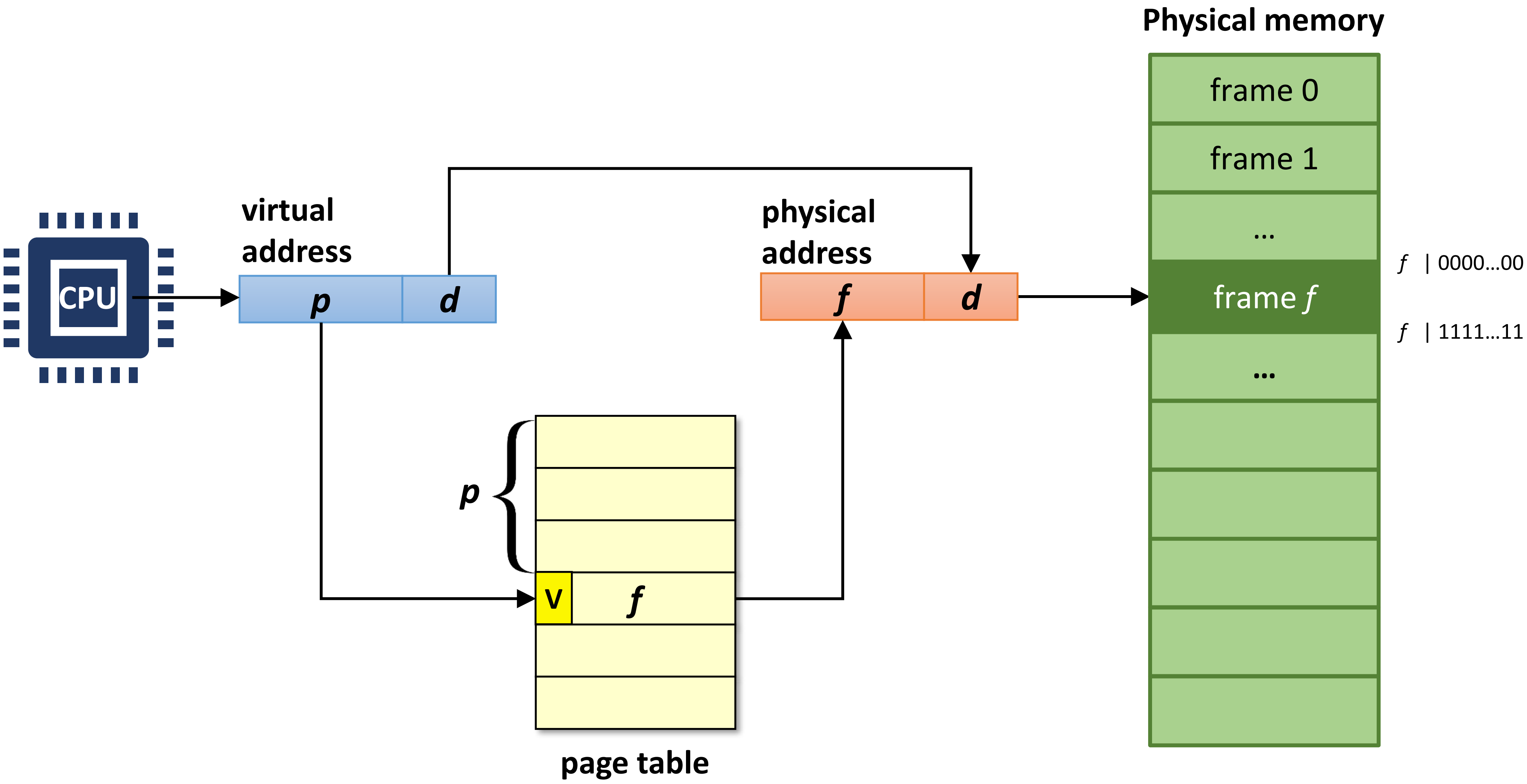
Address Translation (1)

- Translating virtual addresses
  - A virtual address has two parts:
    <Virtual Page Number (VPN), Offset>
  - VPN is an index into the page table
  - Page table determines Page Frame Number (PFN)
  - Physical address is <PFN, Offset>
  - Usually, |VPN| >= |PFN|



- Page tables
  - Managed by ________
  - Map VPN to PFN
  - One Page Table Entry (PTE) per page in virtual address space

Address Translation (2)



Protection

- Separate page table for each process
  - No way to access the physical memory of other processes
  - On context switch, an MMU register is set to point to the base address of the current page table (e.g., CR3 in x86, `satp` in RISC-V)

- Page-level protection
  - Memory protection is implemented by associating protection bits with each PTE
  - Valid / invalid bit
    - "Valid": the page is in the process' address space and in use
    - "Invalid": the page is not allocated
  - Finer level of protection is possible for valid pages
    - Read-only, Read-write, or execute-only protections

PTE

- Page Table Entry

1	1	1	2	20
V	**R**	**M**	**Prot**	**Page Frame Number (PFN)**

- V (Valid) bit says whether or not the PTE can be used
  - It is checked each time a virtual address is used
- R (Reference) bit says whether the page has been accessed
  - It is set when a read or write to the page occurs
- M (Modify) bit says whether the page is dirty
  - It is set when a write to the page occurs
- Prot (Protection) bits control which operations are allowed
  - Read, Write, Execute, User/Kernel, etc.
- PFN (Page Frame Number) determines the physical frame

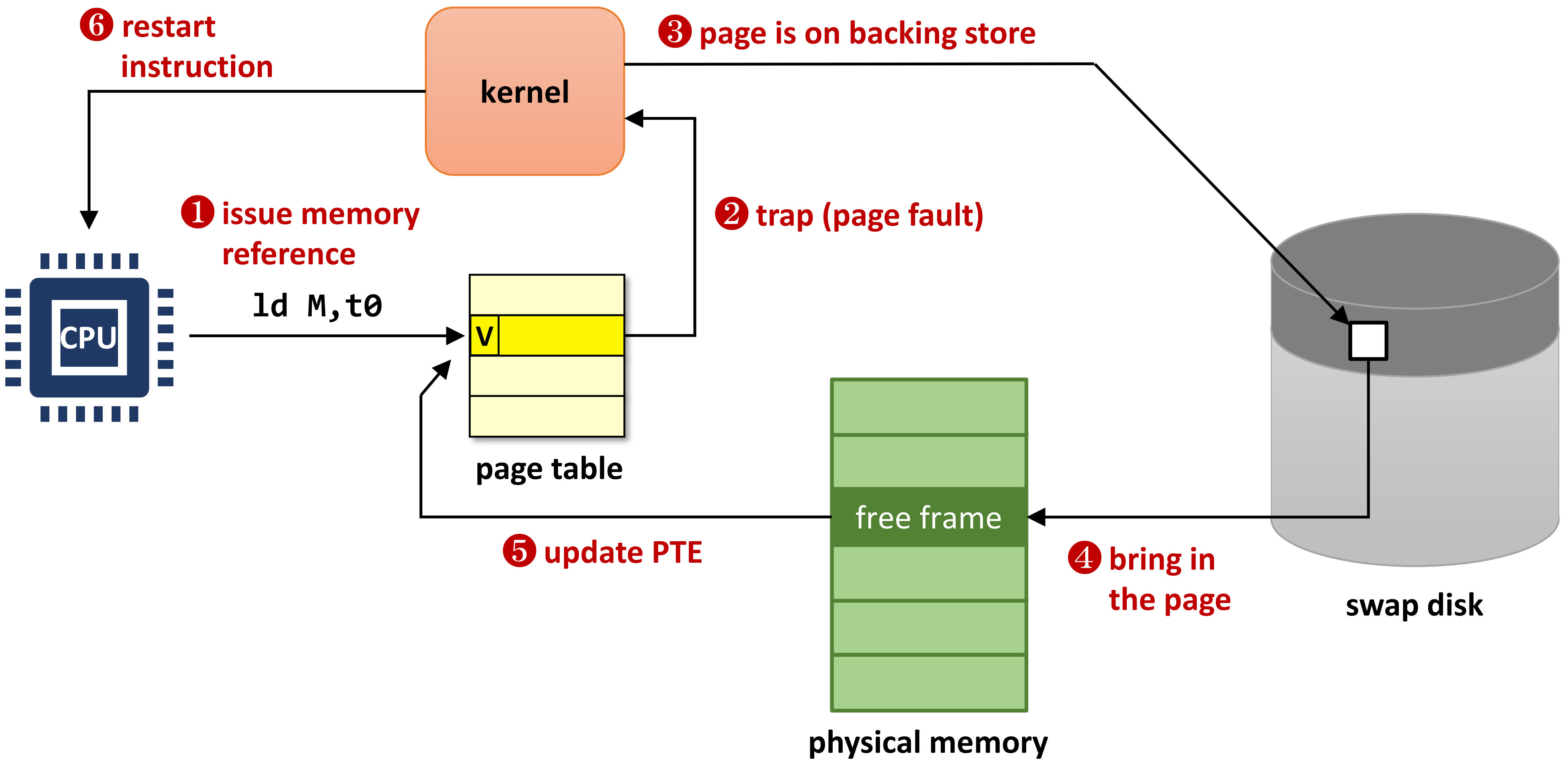
Demand Paging

- OS uses main memory as a (page) cache of all the data allocated by processes in the system
  - Bring a page into memory only when it is needed
  - Pages can be evicted from their physical memory frames
  - Evicted pages go to disk (only dirty pages are written)
  - Movement of pages is transparent to processes
- Benefits
  - ____________________
  - Less memory needed
  - Faster response
  - More processes

Page Fault

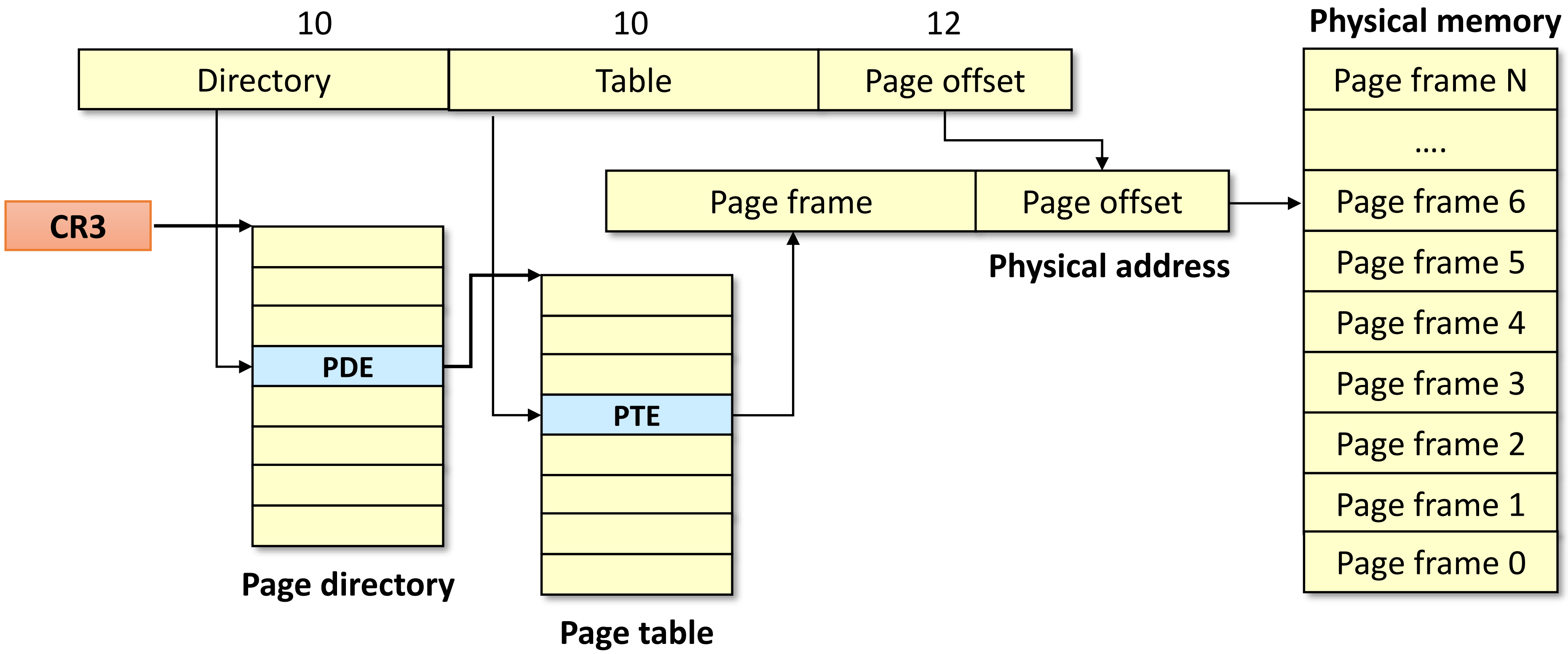
- An exception raised by CPU when accessing invalid PTE
- _______ page faults
  - The page is valid but not loaded into memory
  - OS maintains information on where to find the contents
  - Require disk I/Os
- _______ page faults
  - Page faults can be resolved without disk I/O
  - Used for lazy allocation (e.g., accesses to stack & heap pages)
  - Accesses to prefetched pages, etc.
- Invalid page faults
  - Segmentation violation: the page is not in use

Handling Page Faults



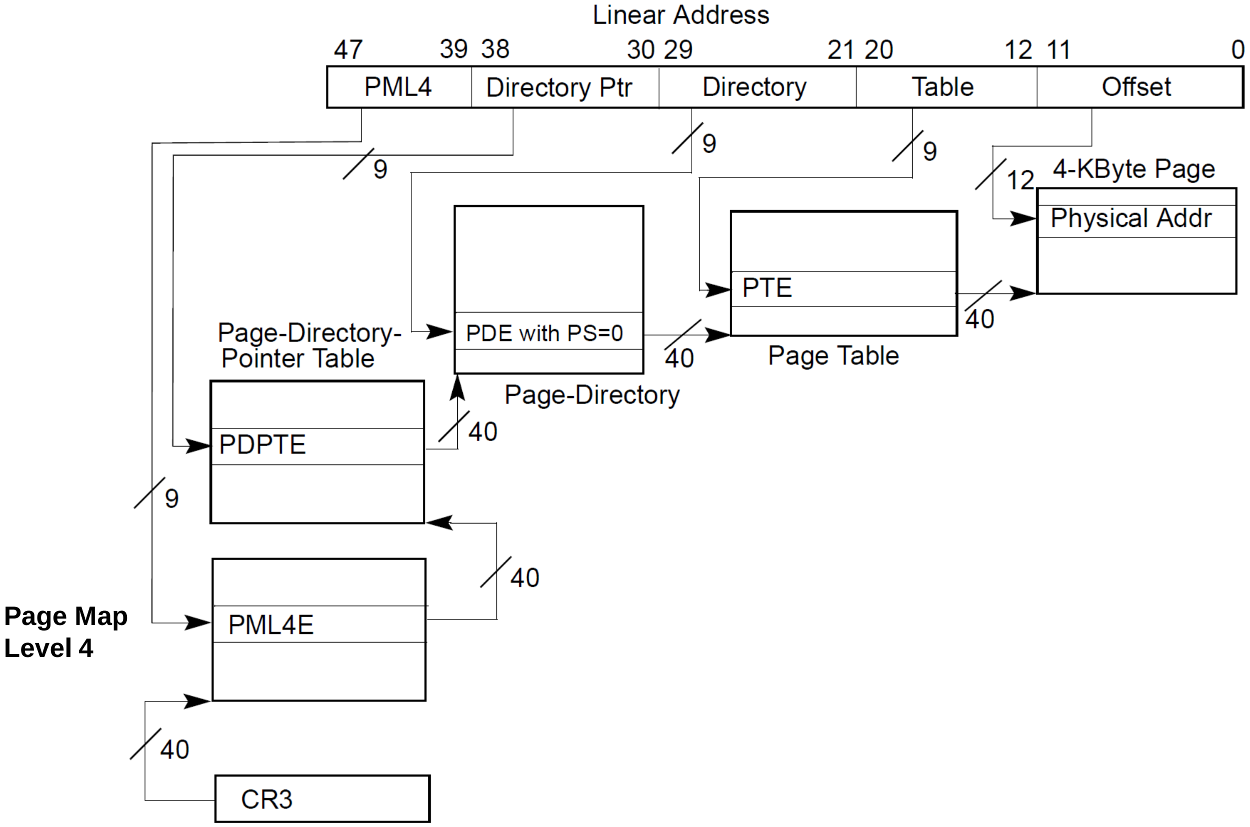
Multi-level Page Table: IA-32

- 32-bit paging
  - 32-bit address space, 4KB pages, 4 bytes/PTE
  - Want every page table fit into a page



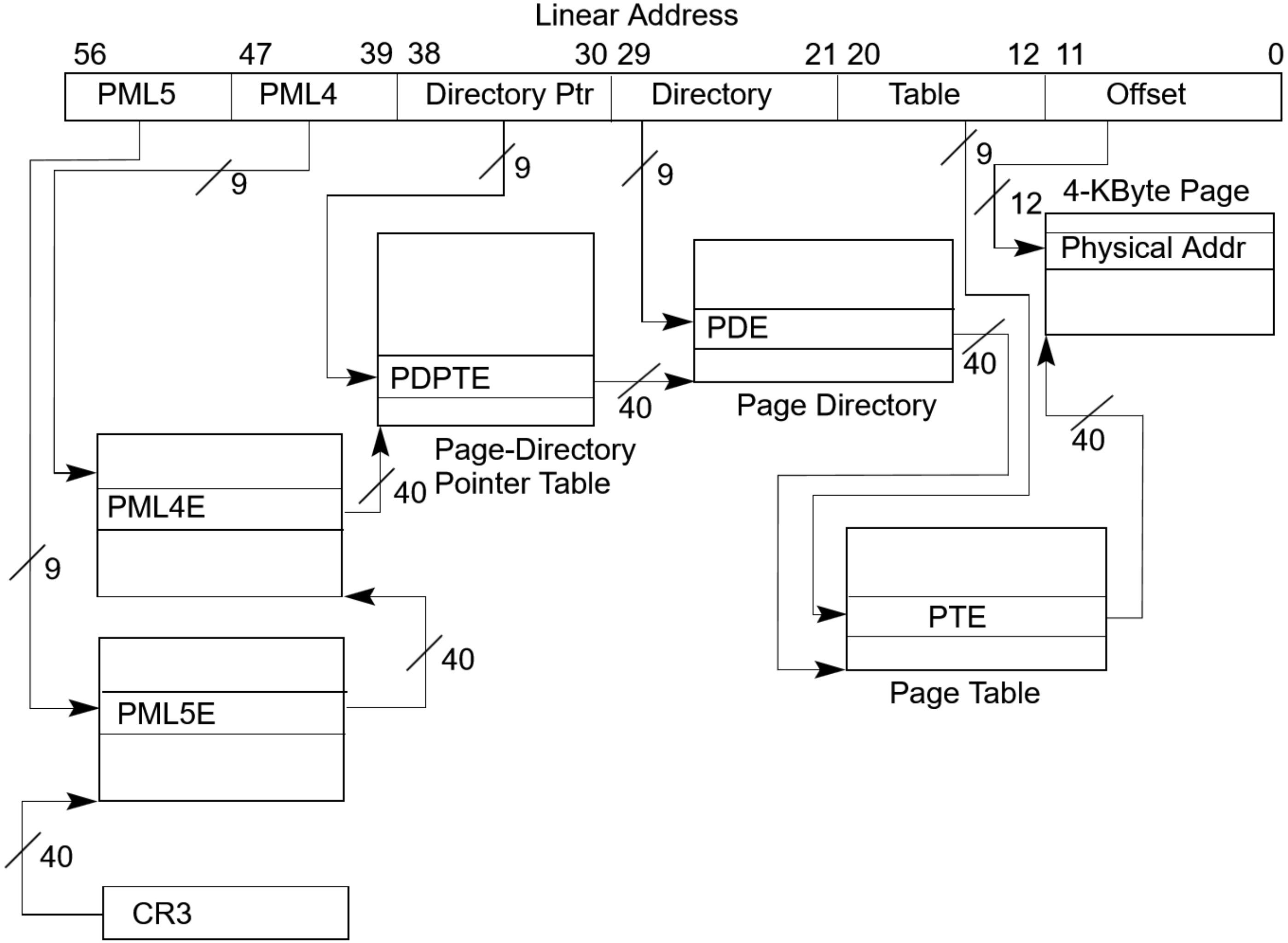
Four-level Page Table

- IA-32e paging mode in Intel 64
  - 48-bit "*linear*" address → ________ physical address (4KB page)



Five-level Page Table

- 57-bit virtual address space
- For Intel Xeon Scalable "Ice Lake" server processors and beyond
- Supported by Linux since 4.14
- Enabled by default since 5.5



TLB

- Translation ___________ Buffer
  - A hardware cache of popular virtual-to-physical address translations
  - Essential component which makes virtual memory possible

- TLB exploits locality
  - Temporal locality: an instruction or data item that has been recently accessed will likely be re-accessed soon
    - Instructions and data accesses in loops, …
  - _________ locality: if a program accesses memory at address *x*, it will likely soon access memory near *x*
    - Code execution, array traversal, stack accesses, …

TLB Organization

- TLB is implemented in hardware
  - Processes only use a handful of pages at a time
    - 16~256 entries in TLB is typical
  - Usually fully associative
    - All entries looked up in parallel
    - But may be set associative to reduce latency
  - Replacement policy: LRU (Least Recently Used)
  - TLB actually caches the whole PTEs, not just PFNs

Valid	Tag (VPN)	Value (PTE)				
1	0x1000	V	R	M	Prot	PFN 0x1234
1	0x2400	V	R	M	Prot	PFN 0x8800
0	-	-				

Handling TLB Misses

- Software-managed TLB

?

- Hardware-managed TLB
  - CPU knows where page tables are in memory
    - e.g., CR3 (or PDBR) register in IA-32 / Intel 64, `satp` in RISC-V
  - ________ maintains page tables
  - CPU "walks" the page table and fills TLB
  - Page tables have to be in hardware-defined format

TLB on Context Switch

- Flush TLB on each context switch
  - TLB is flushed automatically when PTBR is changed in a hardware-managed TLB
  - Some architectures support the pinning of pages into TLB
    - For pages that are globally-shared among processes (e.g., kernel pages)
    - MIPS, Intel, etc.

- Track which entries are for which process
  - Tag each TLB entry with an ASID (Address Space ID)
  - A privileged register holds the ASID of the current process
  - MIPS / ARMv7-A support 8-bit ASID
  - ARMv8-A supports 8-bit/16-bit ASID
  - Intel 64 supports 12-bit PCID (Process Context ID) – Since Westmere (2010)

TLB on Multi-core

- TLB coherence
  - Page-table changes may leave stale entries in the TLBs
  - Flushing the local TLB is not enough
  - Unlike memory caches, TLBs of different cores are not maintained coherent by hardware
  - TLB coherence should be restored by the OS

- TLB ______________
  - The initiating core sends an IPI (Inter-Processor Interrupt) to the remote cores
  - The remote cores invalidate their TLBs (may need to flush the entire TLB)
  - The IPI may take several hundreds of cycles

TLB Performance

- TLB is the source of many performance problems
  - Performance metric: hit rate, lookup latency, …
- Increase TLB ________ (= # TLB entries * Page size)
  - Use **superpages**: e.g., 2MB, 1GB page support in x86_64
  - Increase the TLB size
- Use multi-level TLBs
  - e.g., Intel Haswell (4KB pages): L1 ITLB 128 entries (4-way), L1 DTLB 64-entries (4-way), L2 STLB 1024 entries (8-way)
- Change your algorithms and data structures to be TLB-friendly

Paging: Pros

- No external fragmentation
- Fast to allocate and free
  - A list or bitmap for free page frames
  - Allocation: no need to find contiguous free space
  - Free: no need to coalesce with adjacent free space
- Easy to "page out" portions of memory to disk
  - Page size is chosen to be a multiple of disk block sizes
  - Use valid bit to detect reference to "paged-out" pages
  - Can run process when some pages are on disk
- Easy to protect and share pages

Paging: Cons

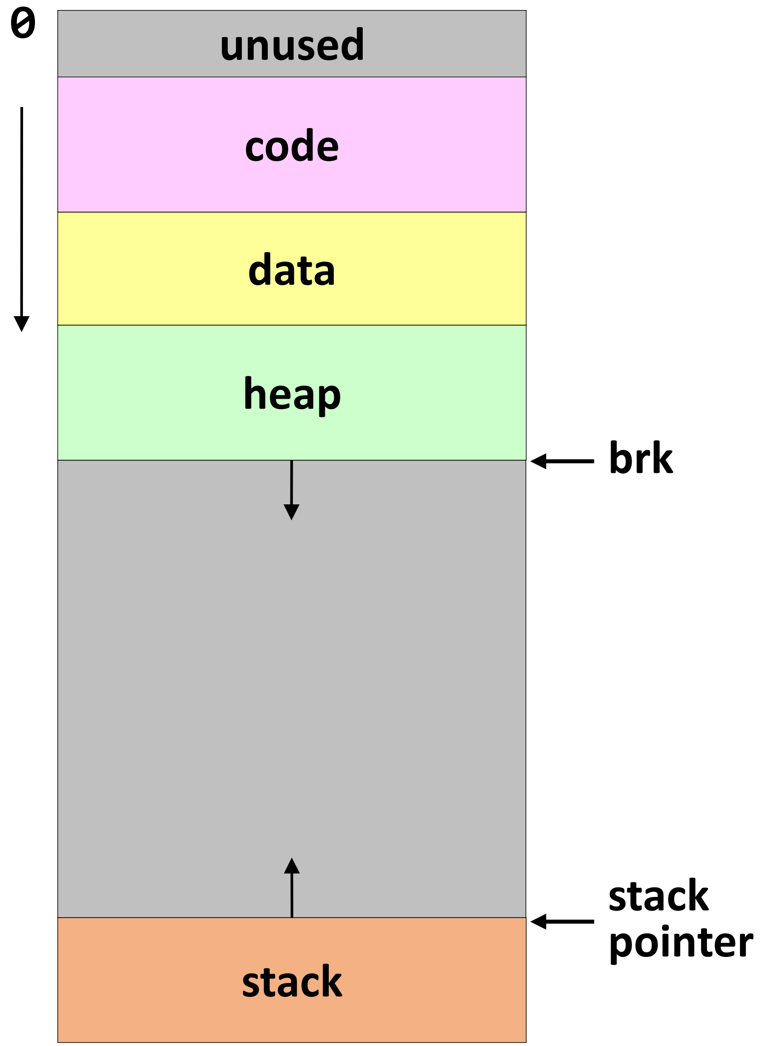
- Internal fragmentation
  - Wasted memory grows with larger pages
- Memory reference overhead
  - Page table stored in memory
  - Address translation increases latency
  - Solution: get hardware support (TLBs)
- Storage needed for page tables
  - Needs one PTE for each page in virtual address space
  - 32-bit virtual address space with 4KB pages: 4MB per page table
  - Page table for each process
  - Solution: use multi-level page table

Memory Mapping

Virtual Memory Area

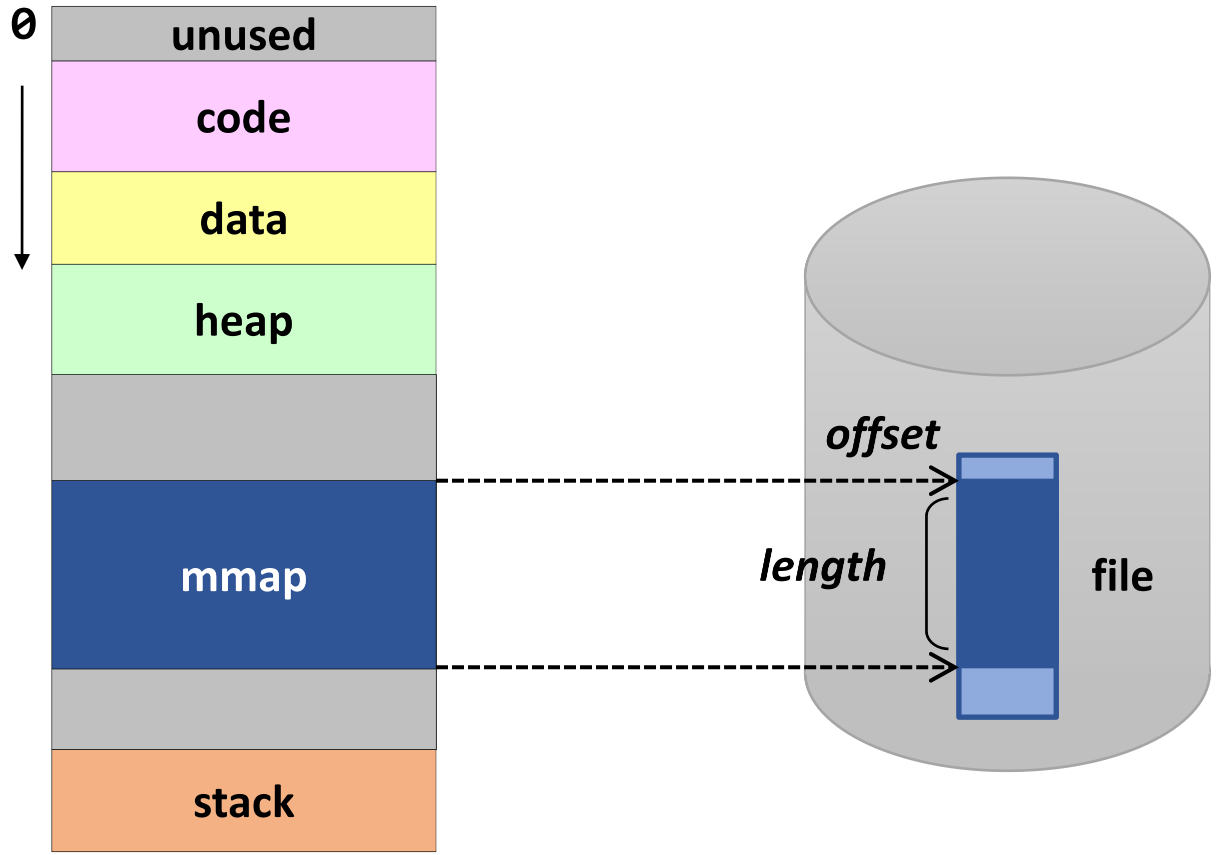
- Virtual address space is a resource
  - Every memory area should be allocated in the virtual address space
  - If you run out of the virtual address space, you can not access any more memory (even if you have space in the physical memory)

- Some of memory areas are backed by files and some aren't



Memory Mapping

- A dynamically allocated virtual memory area that has a backing store
  - File
  - Shared memory
  - ________________
  - None
    (Anonymous mapping)



File vs. Anonymous Mapping

- File mapping (memory-mapped file)
  - Backing store: regular file
  - Maps a memory region to a file region
  - The content of the file can be read from or written to using load/store instructions

- Anonymous mapping
  - Virtual address space not backed by a file
  - Maps a memory region to a memory area filled with 0
  - Zero-page mapping

Shared vs. Private Mapping

- Several processes can map the same backing store in their own virtual address space

- Shared mapping
  - Modifications to shared pages are visible to all involved processes

- Private mapping
  - Modifications are not visible to other processes
  - Copy-on-write

	File mapping	Anonymous mapping
Private	Private file mapping	Private anonymous mapping
Shared	Shared file mapping	Shared anonymous mapping

mmap()

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

- Creates a new mapping in the virtual address space of the calling process
  - `addr`: the starting address for the new mapping (should be aligned to page boundary)
    - If NULL, the kernel chooses the address
    - Otherwise, the kernel takes it as a hint about where to place the mapping
  - `length`: the length of the mapping
  - `prot`: protection info. (`PROT_EXEC`, `PROT_READ`, `PROT_WRITE`, `PROT_NONE`)
  - `flags`: mapping flags (`MAP_PRIVATE`, `MAP_SHARED`, `MAP_ANONYMOUS`, ...)
  - `fd`, `offset`: file descriptor & file offset (used for file mapping)

Memory-Mapped File: Example

- Allows processes to perform file I/O using memory references
  - Instead of open(), read(), write(), close(), etc.
  - Map a file to a virtual memory region

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

int main(int argc, char *argv[]) {
    int fd = open("/bin/ls", O_RDONLY);
    char *p = (char *) mmap(0, 4096, PROT_READ, MAP_SHARED, fd, 0);
    printf("0x%02x 0x%02x 0x%02x 0x%02x\n", *p, *(p+1), *(p+2), *(p+3));
    close(fd);
}
```

Memory-Mapped File

- Implementation
  - Initially, all pages in mapped region are marked as invalid
  - OS reads a page from file whenever invalid page is accessed
  - PTEs map virtual addresses to page frames holding file data
  - <Virtual address base + *n*> refers to `offset` + *n* in file

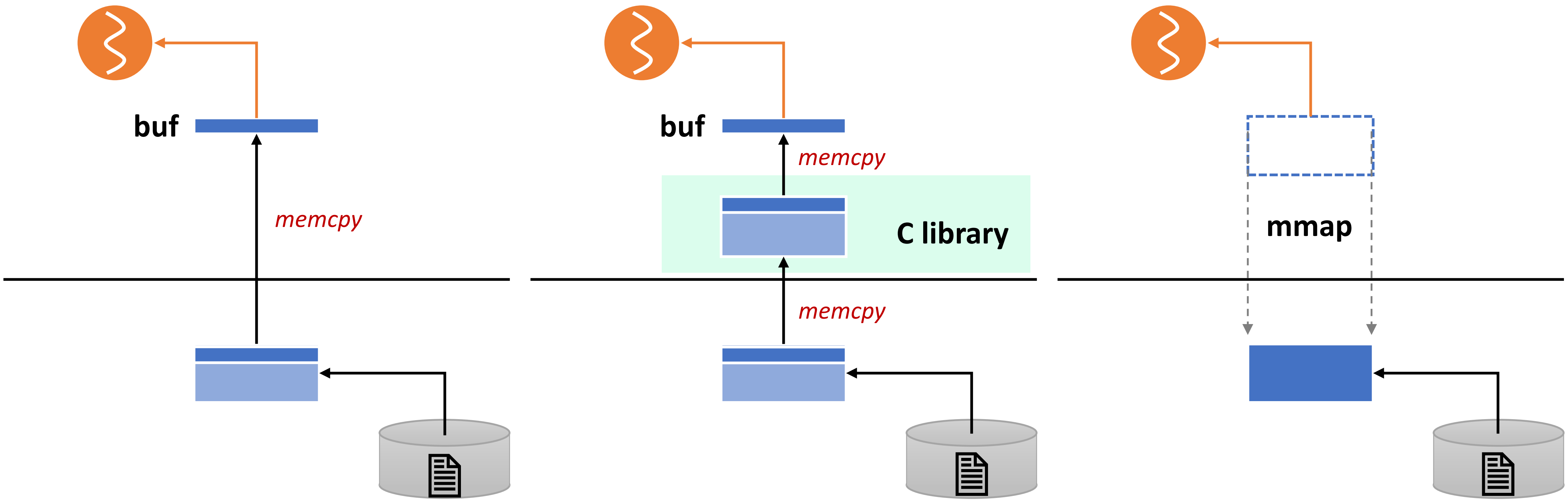
- Writes to the memory-mapped area
  - If MAP_SHARED,
    OS writes to a page and it is written to the file when evicted from physical memory
  - If MAP_PRIVATE,
    OS creates a private copy and then write data to the page (a.k.a. Copy-On-Write).
    File is not modified.

File I/O Comparisons

```
char buf[1024];
int fd = open(“a.txt”, ...);
read(fd, buf, 1024);
```

```
char buf[1024];
FILE *fp = fopen(“a.txt”,“r”);
fgets(buf, 1024, fp);
```

```
int fd = open(“a.txt”, ...);
char *p = mmap(0, .., fd, 0);
```



Summary: Memory-Mapped File

- Pros
  - Uniform access for files and memory (just use pointers)
  - ________________________
  - Several processes can map the same file allowing the pages in memory to be shared

- Cons
  - Process has less control over data movement
  - Does not generalize to streamed I/O (pipes, sockets, etc.)

Shared Memory: Example

- Allows (unrelated) processes to share data using direct memory reference

```
#include <sys/mman.h>
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
    int fd = shm_open("/shm1", O_CREAT | O_EXCL | O_RDWR, 0600);
    ftruncate(fd, 4096);      // set shmem size
    int *p = (int *) mmap(0, 4096, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    for (int i = 0; i < 1024; i++)  p[i] = i;
    close(fd);
}
```

Shared Memory

- Implementation
  - 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

- Mapping shared memory in the virtual address space
  - At the different address: flexible (no address space conflicts), but pointers inside the shared memory are invalid
  - At the same address: less flexible, but shared pointers are valid

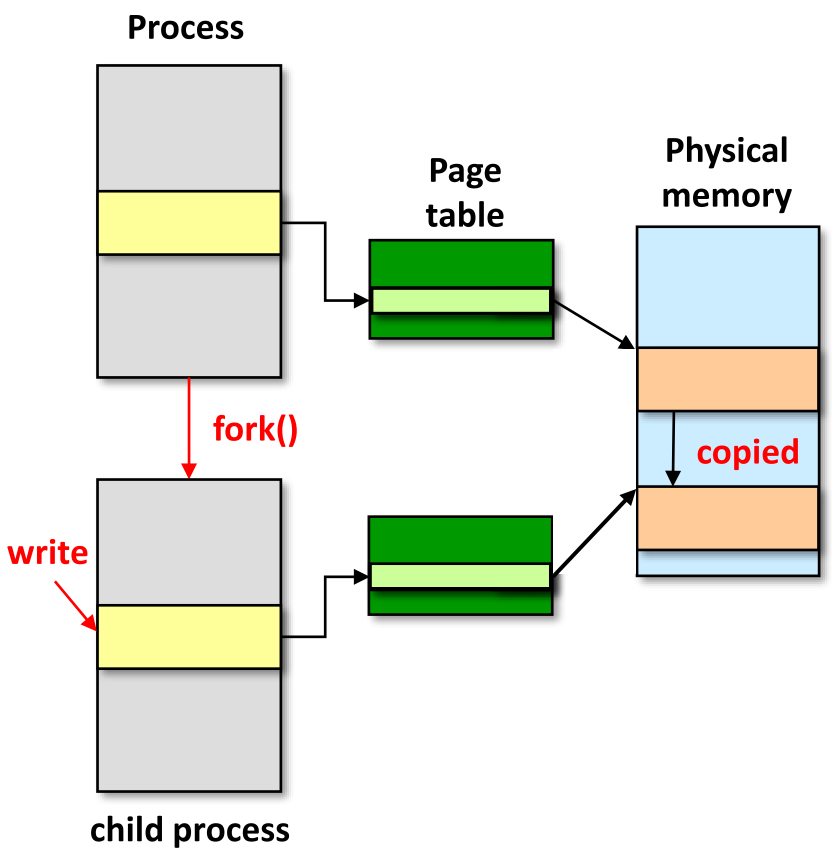
Copy-on-Write

- Defers memory copies as long as possible, hoping to avoid them altogether
- Implementation
  - Instead of copying pages, create shared mappings to the same page frames in physical memory
  - Shared pages are protected as read-only
  - When data is written to these pages, OS allocates new space in physical memory and directs the write to it
- Usage
  - `fork()`
  - Allocating data and heap pages, etc.

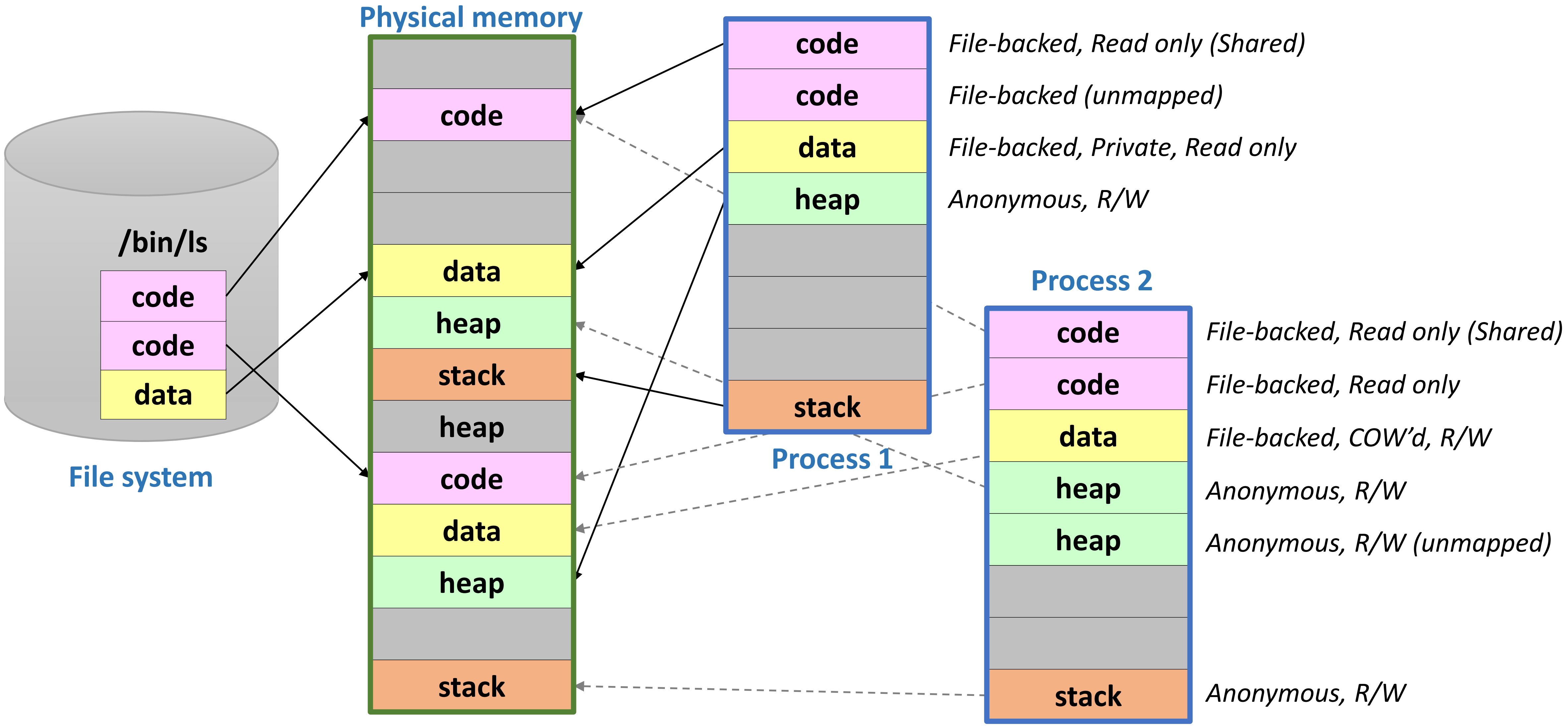
Copy-on-Write during fork()

- COW ensures that both processes do not see each other's changes
  - Instead of copying all pages, create shared mappings of parent pages in the child address space
  - Shared pages are protected as read-only
  - Reads happen as usual
  - Writes generate a protection fault and OS copies the page, changes page mapping, and restarts write instruction

- Efficient when the child process calls `exec()` immediately after `fork()`



Summary


Physical memory
code
data
heap
stack
heap
code
data
heap
stack
/bin/ls
code
code
data
File system
code
code
data
heap
stack
Process 1
File-backed, Read only (Shared)
File-backed (unmapped)
File-backed, Private, Read only
Anonymous, R/W
Process 2
code
code
data
heap
heap
stack
File-backed, Read only (Shared)
File-backed, Read only
File-backed, COW'd, R/W
Anonymous, R/W
Anonymous, R/W (unmapped)
Anonymous, R/W

Swapping

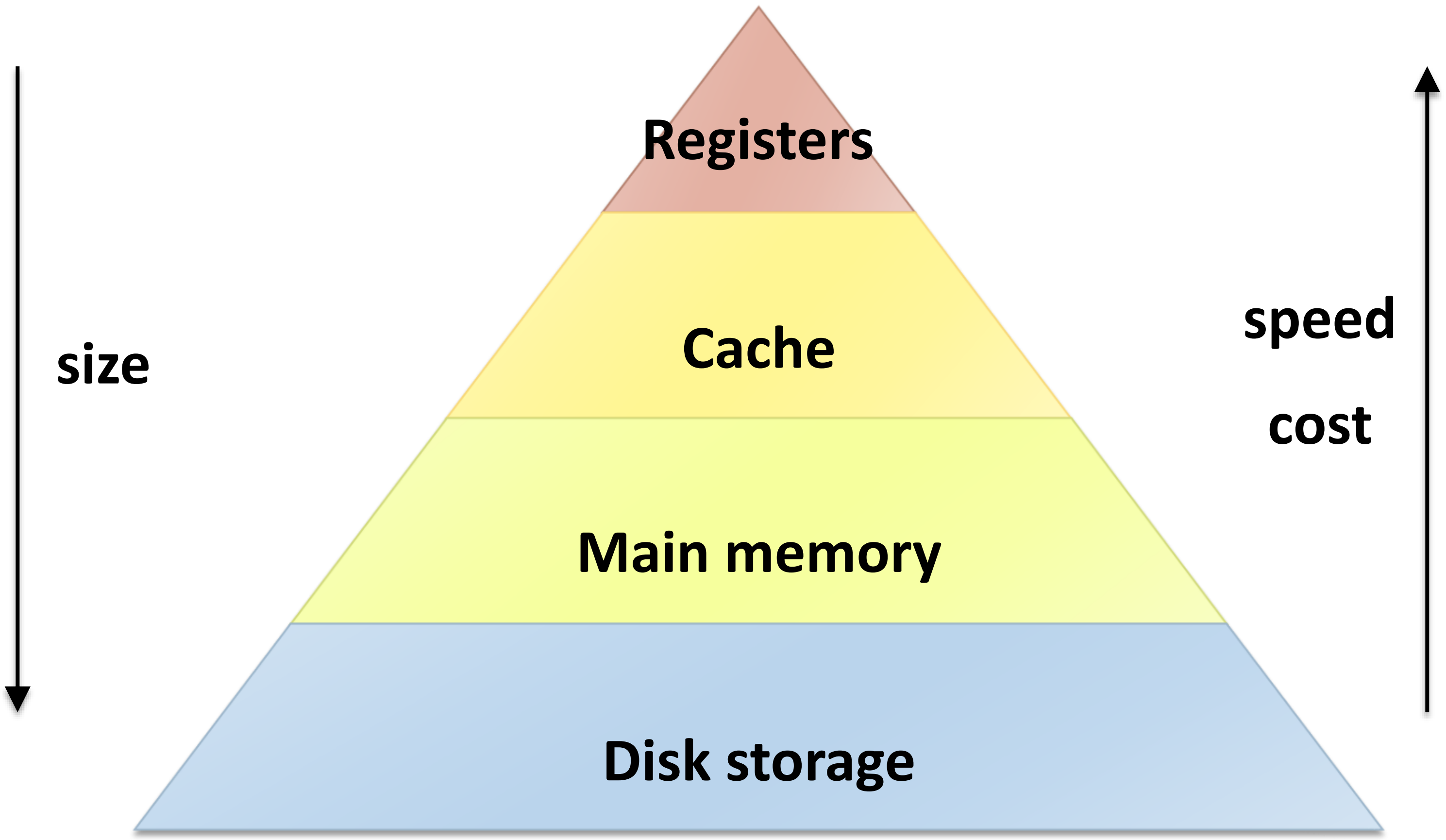
Swapping

- Support processes when not enough physical memory
  - User program should be independent of the amount of physical memory
  - Single process with very large address space
  - Multiple processes with combined address spaces

- Consider physical memory as a ________ for disks
  - Leverage locality of reference within processes
  - Process only uses small amount of address space at a moment
  - Only small amount of address space must be resident in physical memory
  - Store the rest of them to disk

Memory Hierarchy

- Each layer acts as “backing store” for layer above



Numbers Everyone Show Know

L1 cache reference	0.5 ns
Branch mispredict	5 ns
L2 cache reference	7 ns
Mutex lock/unlock	100 ns
Main memory reference	100 ns
Compress 1K bytes with Zippy	10,000 ns
Send 2K bytes over 1 Gbps network	20,000 ns
Read 1 MB sequentially from memory	250, 000 ns
Round trip within same datacenter	500,000 ns
Disk seek	10,000,000 ns
Read 1 MB sequentially from network	10,000,000 ns
Read 1 MB sequentially from disk	30,000,000 ns
Send packet CA → Netherlands → CA	150,000,000 ns

Source: Jeff Dean, "Designs, Lessons and Advice from Building Large Distributed Systems", LADIS Keynote, 2009

How to Swap

- ____________
  - Programmers manually move pieces of code or data in and out of memory as they were needed
  - No special support needed from OS
- Process-level swapping
  - A process is swapped temporarily out of memory to a backing store
  - It's brought back into memory later for continued execution
- Page-level swapping
  - Swap pages out of memory to a backing store (*swap-out* or *page-out*)
  - Swap pages into memory from the backing store (*swap-in* or *page-in*)

Where to Swap

- Swap space
  - Disk space reserved for moving pages back and forth
  - The size of the swap space determines the maximum number of memory pages that can be in use
  - Block size is same as the page size
  - Can be a dedicated partition or a file in the file system

	PFN 0	PFN 1	PFN 2	PFN 3
Physical Memory	PID 0 (VPN 0)	PID 1 (VPN 1)	PID 1 (VPN 2)	PID 2 (VPN 0)

	Blk 0	Blk 1	Blk 2	Blk 3	Blk 4	Blk 5	Blk 6	Blk 7
Swap Space	PID 0 (VPN 1)	PID 0 (VPN 2)	Free	PID 1 (VPN 0)	PID 1 (VPN 1)	PID 3 (VPN 0)	PID 2 (VPN 1)	PID 3 (VPN 1)

When to Swap

- Proactively based on thresholds
  - OS wants to keep a small portion of memory free
  - Two threshold values: *HW* (high watermark) and *LW* (low watermark)
  - A background thread called swap daemon (or page daemon) is responsible for freeing memory (e.g., `kswapd` in Linux)

  - If (# free pages < *LW*), the swap daemon starts to evict pages from physical memory
  - If (# free pages > *HW*), the swap daemon goes to sleep

  - What if the allocation speed is faster than reclamation speed?

What to Swap

- What happens to each type of page frame on low mem?
  - Kernel code → **Not swapped**
  - Kernel data → **??**
  - Page tables for user processes → **Not swapped**
  - Kernel stack for user processes → **??**
  - User code pages → **Dropped**
  - User data pages → **??**
  - User heap/stack pages → **Swapped**
  - Files mmap'ed to user processes → **??**
  - Page cache pages → **Dropped or go to file system**
- Page replacement policy chooses the pages to evict

Page Replacement

- Which page in physical memory should be selected as a victim?
  - Write out the victim page to disk if modified (dirty bit set)
  - If the victim page is clean, just discard
    - The original version is either in the file system or in the swap space
  - Why not use direct-mapped or set-associative design similar to CPU caches?

- Goal: minimize the page fault rate (miss rate)
  - The miss penalty (cost of disk access) is so high (> x100,000)
  - A tiny miss rate quickly dominates the overall AMAT (Average Memory Access Time)

OPT (or MIN)

- Belady's optimal replacement policy (1966)
  - Replace the page that will not be used for the longest time in the future
  - Shows the lowest fault rate for any page reference stream
  - Problem: have to predict the future
  - Not practical, but good for comparison

Reference:	**1**	**2**	**3**	**4**	**1**	**2**	**5**	**1**	**2**	**3**	**4**	**5**
	1	1	1	1	**1**	1	1	**1**	1	**3**	3	3
PF rate = 7 / 12		**2**	2	2	2	**2**	2	2	**2**	2	**4**	4
			3	**4**	4	4	**5**	5	5	5	5	**5**
	Miss	**Miss**	**Miss**	**Miss**	**Hit**	**Hit**	**Miss**	**Hit**	**Hit**	**Miss**	**Miss**	**Hit**

FIFO

- First-In First-Out
  - Replace the page that has been in memory the longest
  - Why might this be good?
    - Maybe, the one brought in the longest ago is not being used
  - Why might this be bad?
    - Maybe, it's not the case
    - Some pages may always be needed
  - Obvious and simple to implement
  - Fair: all pages receive equal residency
  - FIFO suffers from "Belady's anomaly"
    - The fault rate might increase when the algorithm is given more memory

FIFO: Belady's Anomaly

PF rate = 9 / 12

Reference:	1	2	3	4	1	2	5	1	2	3	4	5
	1	1	1	4	4	4	5	5	5	5	5	5
		2	2	2	1	1	1	1	1	3	3	3
			3	3	3	2	2	2	2	2	4	4
	Miss	Miss	Miss	Miss	Miss	Miss	Miss	Hit	Hit	Miss	Miss	Hit

PF rate = 10 / 12

Reference:	1	2	3	4	1	2	5	1	2	3	4	5
	1	1	1	1	1	1	5	5	5	5	4	4
		2	2	2	2	2	2	1	1	1	1	5
			3	3	3	3	3	3	2	2	2	2
				4	4	4	4	4	4	3	3	3
	Miss	Miss	Miss	Miss	Hit	Hit	Miss	Miss	Miss	Miss	Miss	Miss

LRU

- Least Recently Used
  - Replace the page that has not been used for the longest time in the past
  - Use past to predict the future
    - cf. OPT wants to look at the future

  - With locality, LRU approximates OPT
  - "Stack" algorithm: does not suffer from Belady's anomaly
  - Harder to implement: must track which pages have been accessed
  - Does not consider the frequency of page accesses
  - Does not handle all workloads well

Stack Property

- Stack algorithms
  - Policies that guarantee increasing memory size does not increase the number of page faults (e.g., OPT, LRU, etc.)
  - Any page in memory with *m* frames is also in memory with *m+1* frames

PF rate = 10 / 12

Reference:	**1**	**2**	**3**	**4**	**1**	**2**	**5**	**1**	**2**	**3**	**4**	**5**
Stack distance:	*∞*	*∞*	*∞*	*∞*	*4*	*4*	*∞*	*3*	*3*	*5*	*5*	*5*
	1	2	3	4	1	2	5	1	2	3	4	5
		1	2	3	4	1	2	5	1	2	3	4
			1	2	3	4	1	2	5	1	2	3
				1	2	3	4	4	4	5	1	2
							3	3	3	4	5	1
	Miss	Miss	Miss	Miss	Miss	Miss	Miss	Hit	Hit	Miss	Miss	Miss

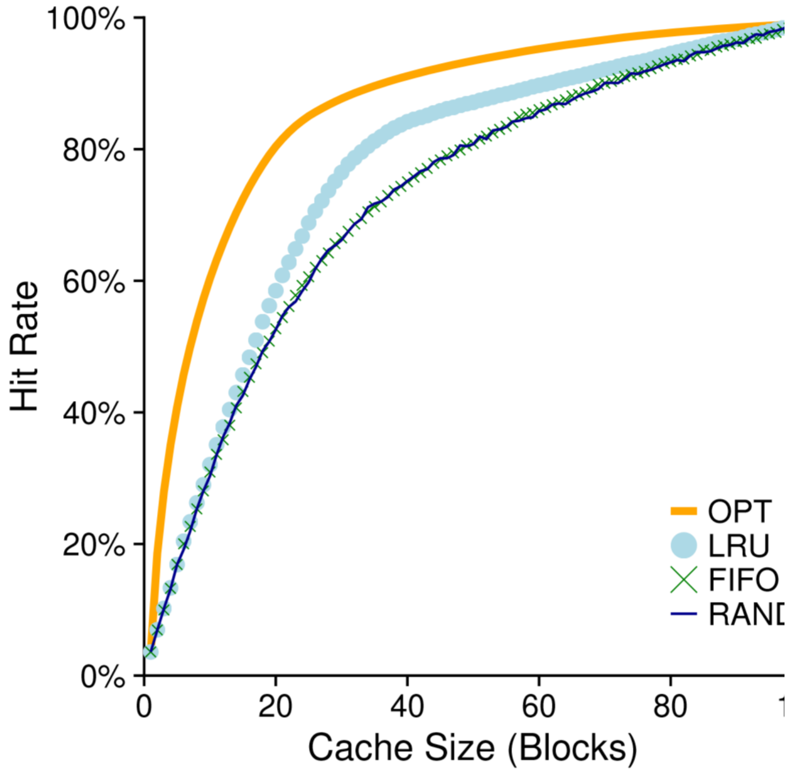
RANDOM

- Another simple policy
  - Simply picks a random page to replace under memory pressure
  - Simple to implement: no bookkeeping needed
  - Performance depends on the luck of the draw
  - Outperforms FIFO and LRU for certain workloads

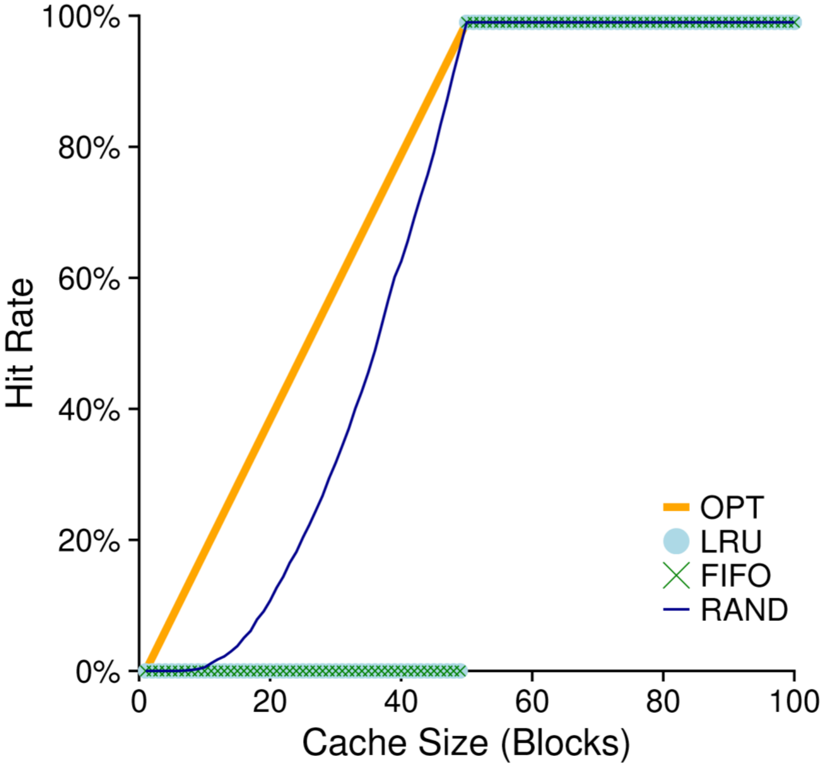
Reference:	**1**	**2**	**3**	**4**	**1**	**2**	**5**	**1**	**2**	**3**	**4**	**5**
	1	1	1	1	**1**	1	1	**1**	1	1	1	**5**
PF rate = 8 / 12		**2**	2	2	2	**2**	2	2	**2**	**3**	3	3
			3	**4**	4	4	**5**	5	5	5	**4**	4
	Miss	Miss	Miss	Miss	Hit	Hit	Miss	Hit	Hit	Miss	Miss	Miss

Comparisons

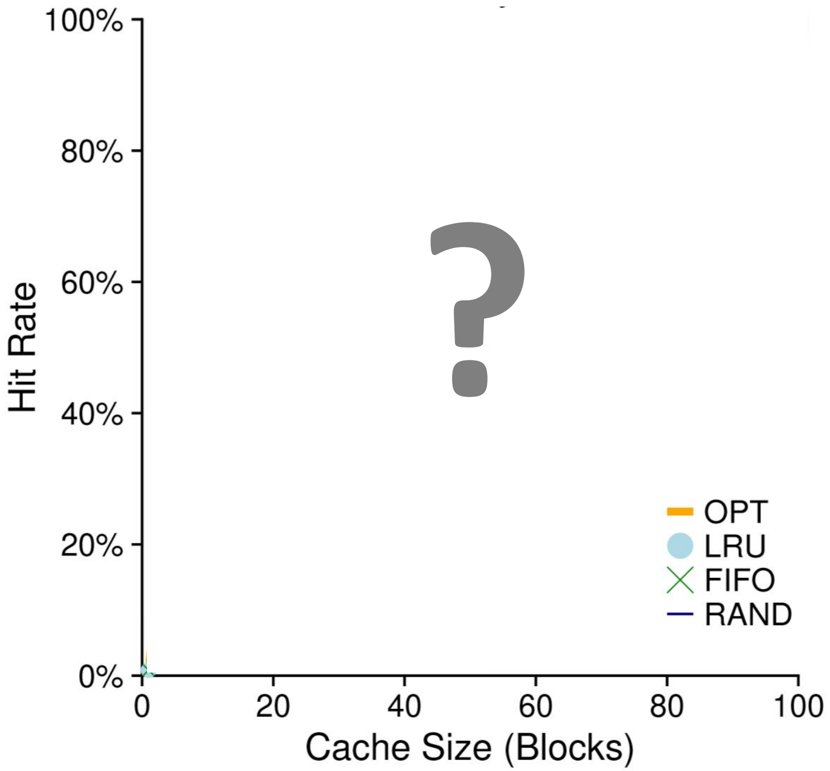
The 80-20 Workload



The Looping Workload (50 blocks)



The Random Workload



Implementing LRU

- Software approach
  - OS maintains ordered list of page frames by reference time
  - When page is referenced: move page to the front of the list
  - When need victim: pick the page in the back of the list
  - Slow on memory reference, fast on replacement
- Hardware approach

?

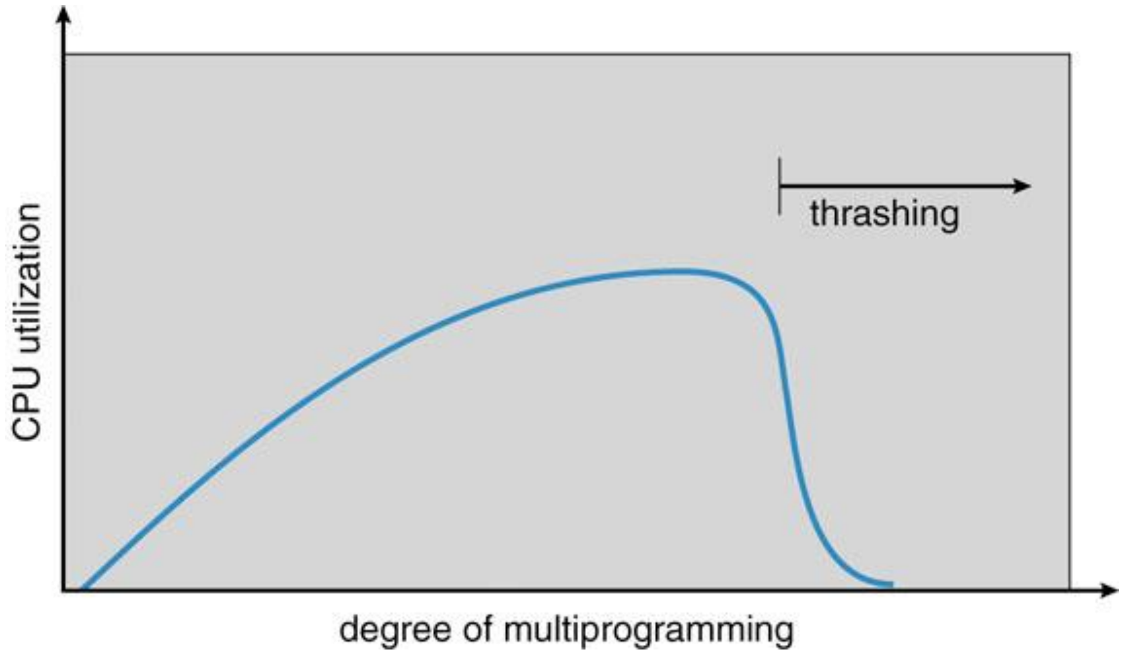
Replacement Algorithms

- I/O buffer cache replacement
  - "Page hit" is known to OS
  - Uses block I/O traces
  - LRU, LRU-2, 2Q, SEQ, LRFU, EELRU, MQ, LIRS, ARC, …

- VM page replacement
  - "Page hit" is only known to hardware, not to OS
  - Hardware sets the Reference / Dirty bits in the PTE
  - LRU approximation
  - Uses memory reference traces
  - CLOCK, WSClock, GCLOCK, CAR, CLOCK-Pro, …

Thrashing

- What happens when physical memory is not enough to hold all the "working sets" of processes
  - Working set: a set of pages that a process is using actively
  - Most of the time is spent by an OS paging data back and forth from disk
  - Possible solutions:
    - Kill processes
    - Buy more memory



- Android's LMK
  (Low Memory Killer)

Summary

- VM mechanisms
  - Physical and virtual addressing
  - Partitioning, segmentation, paging
  - Page table management, TLBs, etc.
- VM policies
  - Page replacement policy, page allocation policy
- VM optimizations
  - Demand paging, copy-on-write (space)
  - Multi-level page tables (space)
  - Efficient translation using TLBs (time)
  - Page replacement policy (time)