Lecture 12: Virtual Memory Optimizations

CS343 – Operating Systems

Branden Ghena – Spring 2022

Some slides borrowed from:
Stephen Tarzia (Northwestern), Shivaram Venkataraman (Wisconsin), and UC Berkeley CS162

Today's Goals

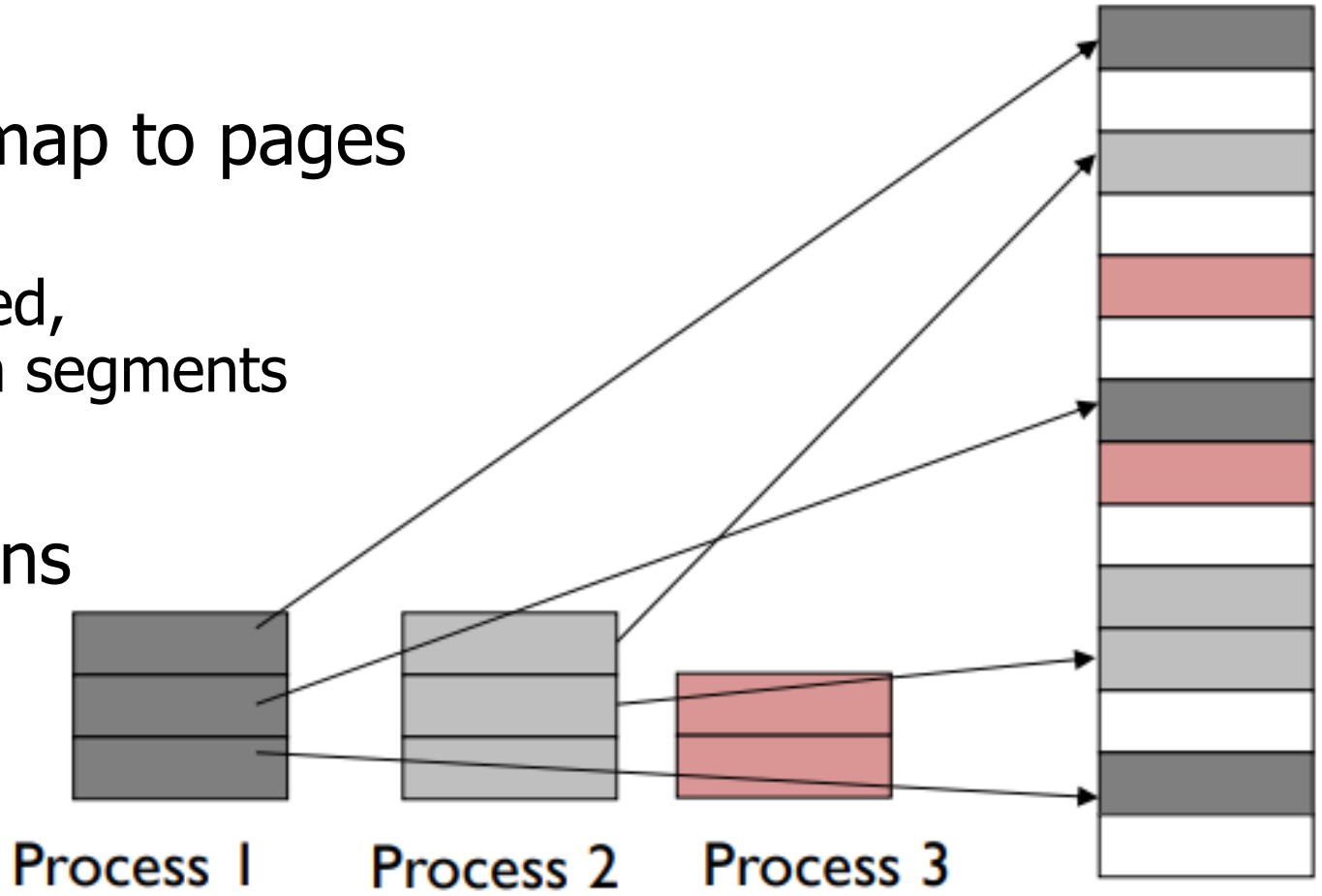
- Explore optimizations to memory paging.
- Insight into how virtual memory is used and what it looks like in today's systems.
- Review of the memory hierarchy and how the OS interacts with each level.
- Introduce swapping as a mechanism for enabling more virtual memory than physical memory.
- Explore several page replacement policies that control swapping.

Memory paging

- Divide memory into small, **fixed-sized** pages

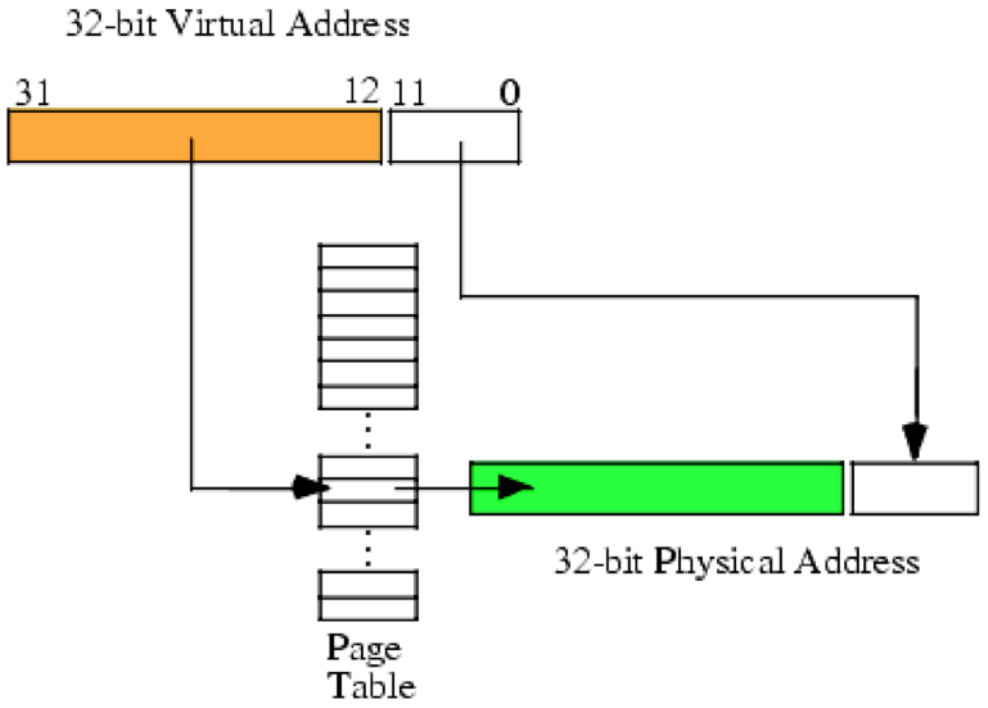
- Pages of virtual memory map to pages of physical memory
  - Like segments were mapped, but *many* more pages than segments

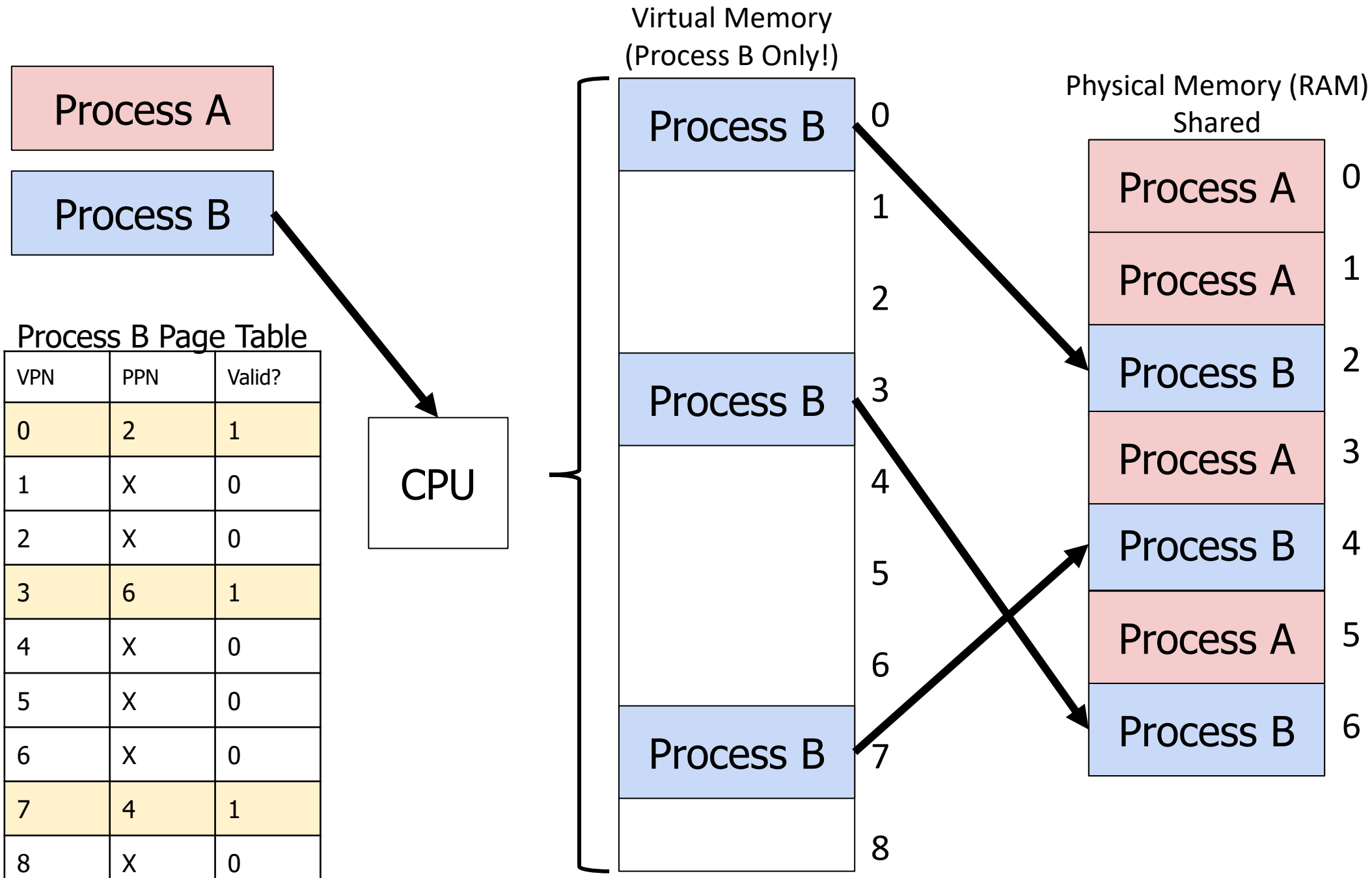
- Processes and their sections can be mapped to any place in memory



Page table translates virtual addresses to physical addresses

- Use topmost bits of virtual address to select page table entry
  - One page table entry per each virtual page
- Add address at page table entry to bottommost bits
  - Actually just concatenate the two
- Just like segment tables, there will be a different page table for each process




Process A
Process B
Process B Page Table
VPN PPN Valid?
0 2 1
1 X 0
2 X 0
3 6 1
4 X 0
5 X 0
6 X 0
7 4 1
8 X 0
CPU
Virtual Memory
(Process B Only!)
Process B
Process B
Process B
0
1
2
3
4
5
6
7
8
Physical Memory (RAM)
Shared
Process A
Process A
Process B
Process A
Process B
Process A
Process B
0
1
2
3
4
5
6

Paging challenges

- Page tables are slow to access
  - Page tables need to be stored in memory due to size
  - MMU only holds the base address of the page table and reads from it
  - Two memory loads per load!!!
  - Going to have to fix this…

- Page tables require a lot of storage space
  - Mapping must exist for each virtual page, even if unused
  - Becomes a serious issue on 64-bit systems

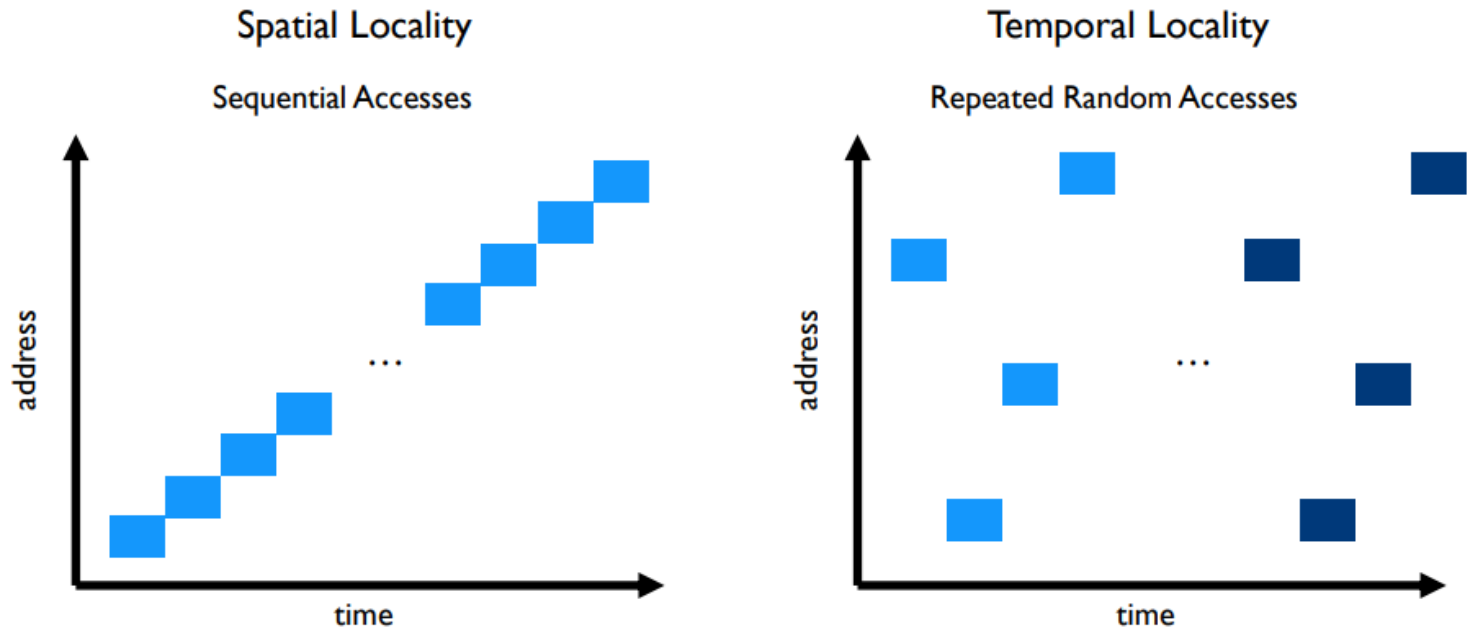
Outline

…continued from last lecture

- **Paging improvements**
  - **Improving translation speed**
  - Improving table storage size

Caching can speed up page table access

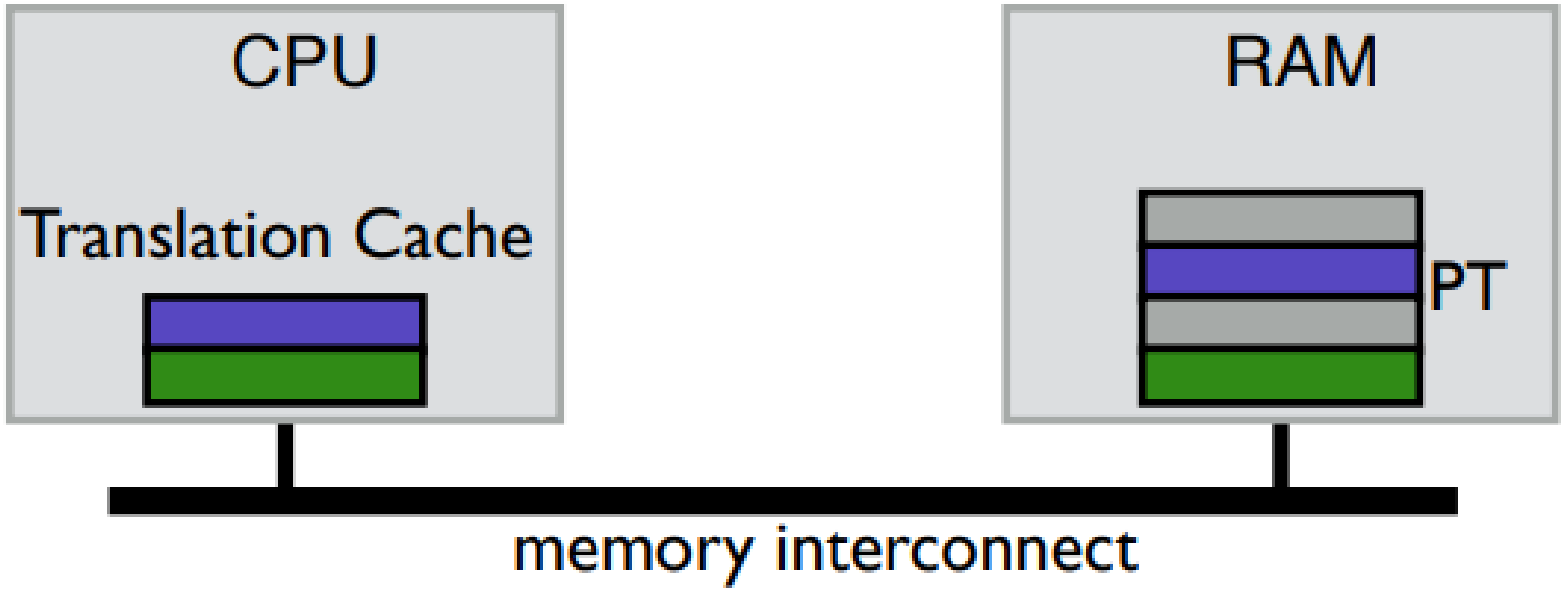
- How do we make page table access faster?
  - How do we make memory access faster?
  - Cache it!
- Code and Stack have very high spatial locality



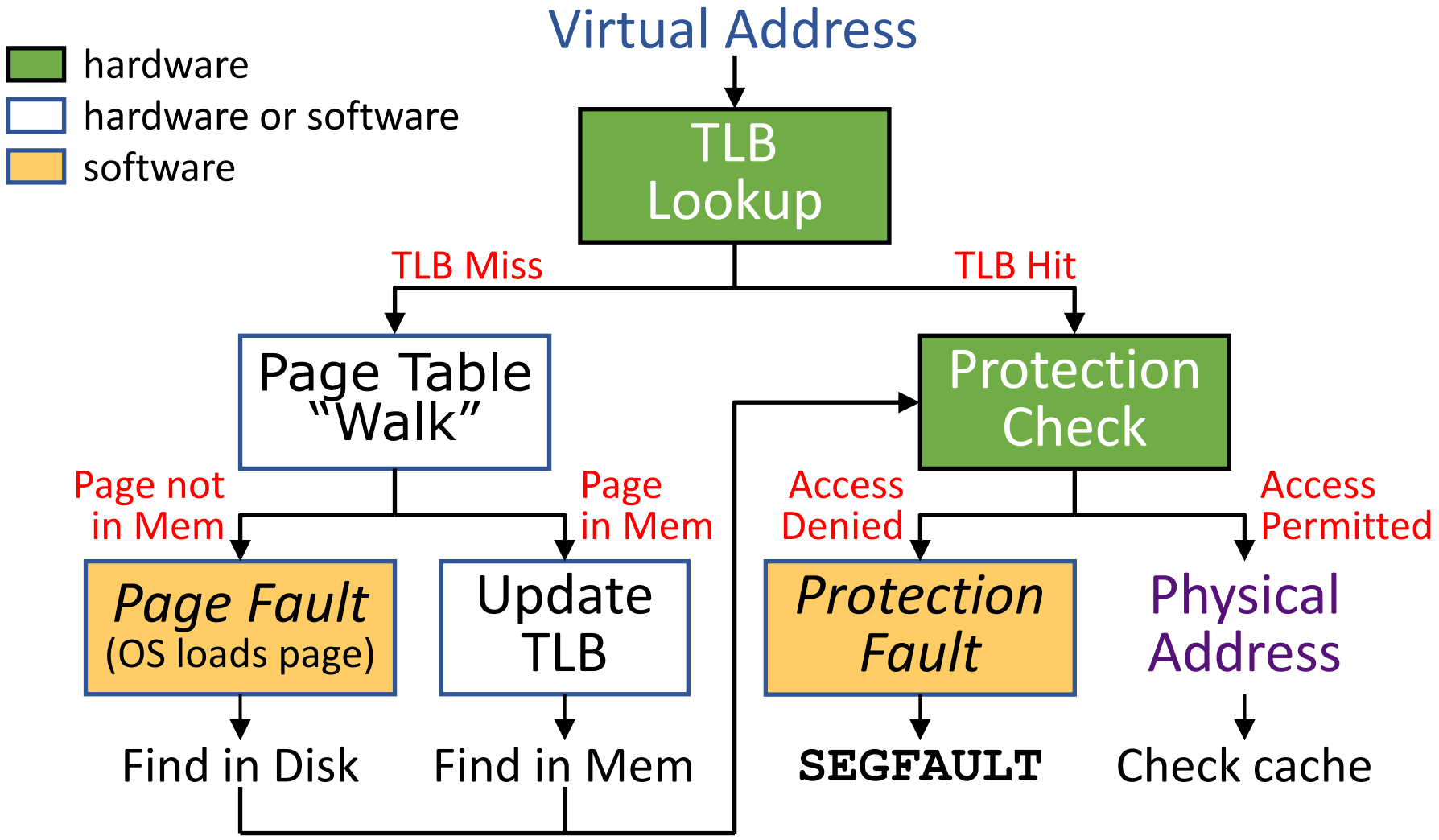
TLB caches page table entries

- Translation Lookaside Buffer
  - Fully-associative cache (only compulsory misses)
  - Holds a subset of the page table (VPN->PPN mapping and permissions)

- On a TLB miss, go check the real page table (done in hardware)



Address translation with TLB



Context switches with a TLB

- A process must only access its own page table entries in the TLB!
  - Otherwise, the mapping is wrong, and it accesses another process…
  - OS needs to manage the TLB

- Option 1: Flush TLB on each context switch
  - Costly to lose recently cached translations

- Option 2: Track with process each entry corresponds to
  - x86-64 Process Context Identifiers (12-bit -> 4096 different processes)
    - Extra state for the OS to manage if it has more processes than that

Software controlled TLBs

- Some RISC CPUs have a software-managed TLB
  - TLB still used for translation, but a miss causes a fault for OS to handle
    - OS looks in page table for proper entry
    - OS evicts an existing entry from TLB
    - OS inserts correct entry into TLB
  - Special instruction allows OS to write to TLB
  - Hardware is simpler and OS has control over the TLB functionality
    - Can prefetch page table entries it thinks might be important
    - Can flush entries relevant to other processes
  - TLB misses take longer to complete, however

Outline

…continued from last lecture

- **Paging improvements**
  - Improving translation speed
  - **Improving table storage size**

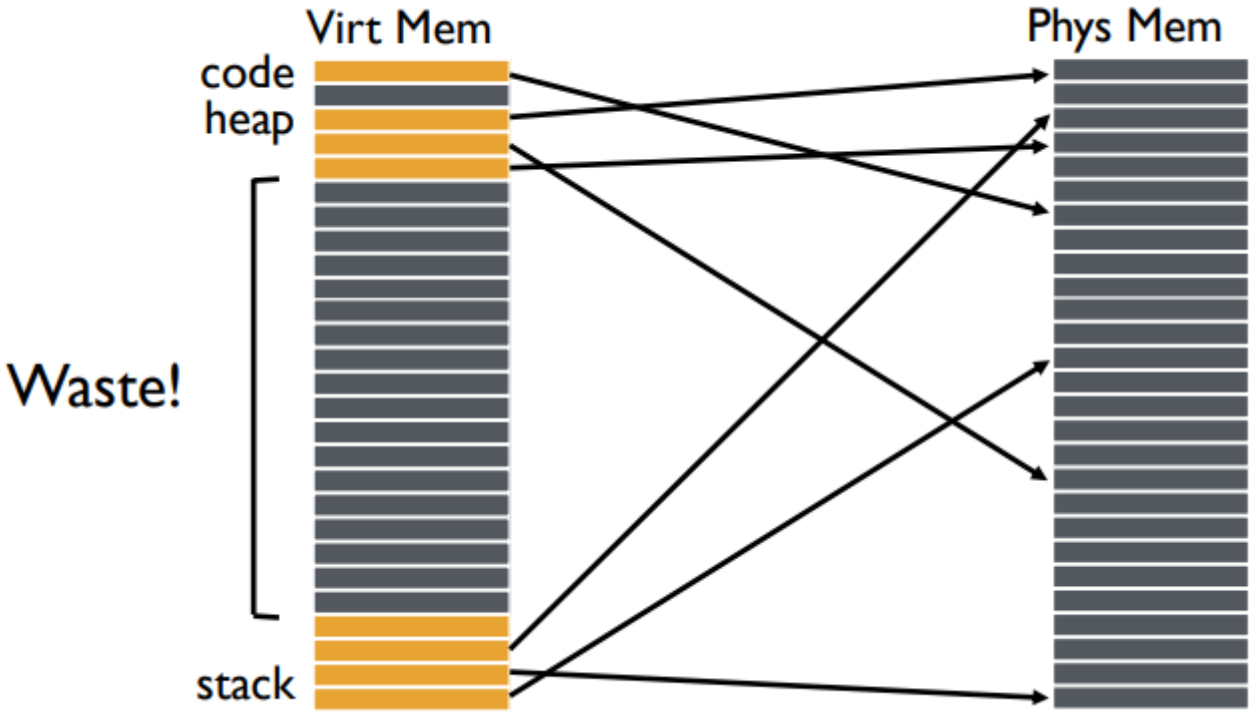
Paging disadvantages

1. Page tables are slow to access
   - Memory access for page table before any other memory access
   - TLB can speed this up considerably for common execution

2. Page tables require a lot of storage space
   - Mapping must exist for each virtual page, even if unused
   - Becomes a serious issue on 64-bit systems

Why do page tables take so much storage space?

- For every virtual page, there must exist an entry in the page table
  - Even though most virtual addresses aren't used!



- 32-bit address space with 4 kB pages -> 1 million entries
  - At least 8 MB of storage
  - 64-bit address space would require 36 exabytes of page table storage…

Create multiple page tables, each with useful mappings only

- How do we eliminate extraneous entries from the page tables?

Virtual Page Number	Valid?	Physical Page Number
0	1	2
1	1	3
2	0	
3	0	
4	0	
5	1	7
6	0	
7	0	

Create multiple page tables, each with useful mappings only

- Collect groups of page table entries
  (call them "page table entry pages"?)

Virtual Page Number	Valid?	Physical Page Number
0	1	2
1	1	3
2	0	
3	0	
4	0	
5	1	7
6	0	
7	0	

Create multiple page tables, each with useful mappings only

- Collect groups of page table entries
- Only keep groups that have valid mappings in them

Virtual Page Number	Valid?	Physical Page Number
0	1	2
1	1	3
4	0	
5	1	7

Create multiple page tables, each with useful mappings only

- Collect groups of page table entries
- Only keep groups that have valid mappings in them
- Remaining groups are now separate tables

Virtual Page Number	Valid?	Physical Page Number
0	1	2
1	1	3

Virtual Page Number	Valid?	Physical Page Number
4	0	
5	1	7

Create multiple page tables, each with useful mappings only

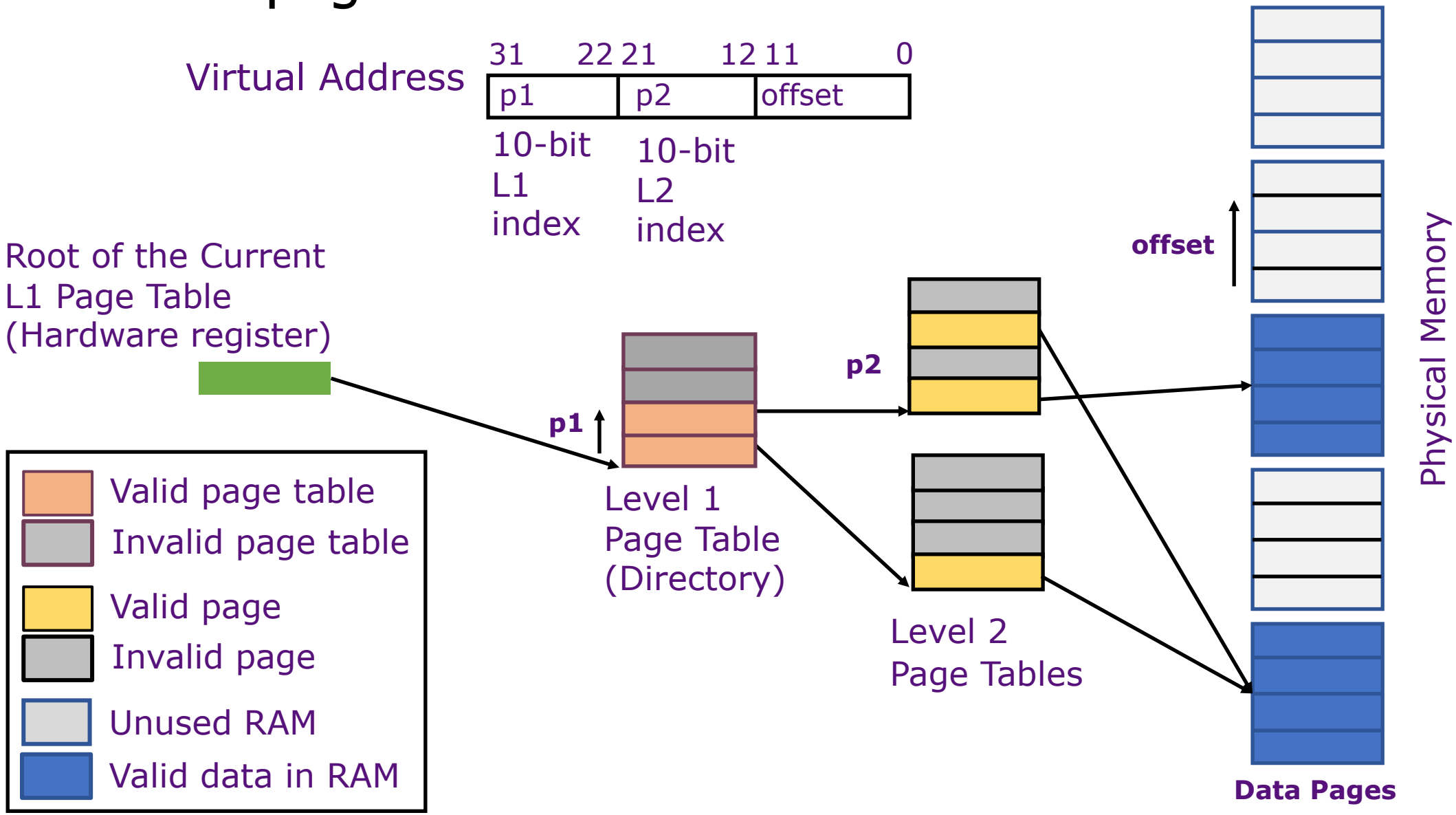
- Collect groups of page table entries
- Only keep groups that have valid mappings in them
- Remaining groups are now separate tables
- Create a directory of page tables to collect existing page tables

Virtual Page Number Range	Valid?	Page Table Address
0-1	1	
2-3	0	
4-5	1	
6-7	0	

Virtual Page Number	Valid?	Physical Page Number
0	1	2
1	1	3

Virtual Page Number	Valid?	Physical Page Number
4	0	
5	1	7

Multilevel page tables



Multilevel page table logistics

- Virtual address is broken down into three or more parts
  - Highest bits index into highest-level page table
- A missing entry at any level triggers a page fault

- Size of tables in memory proportional to number of pages of virtual memory used
  - Small processes can have proportionally small page tables

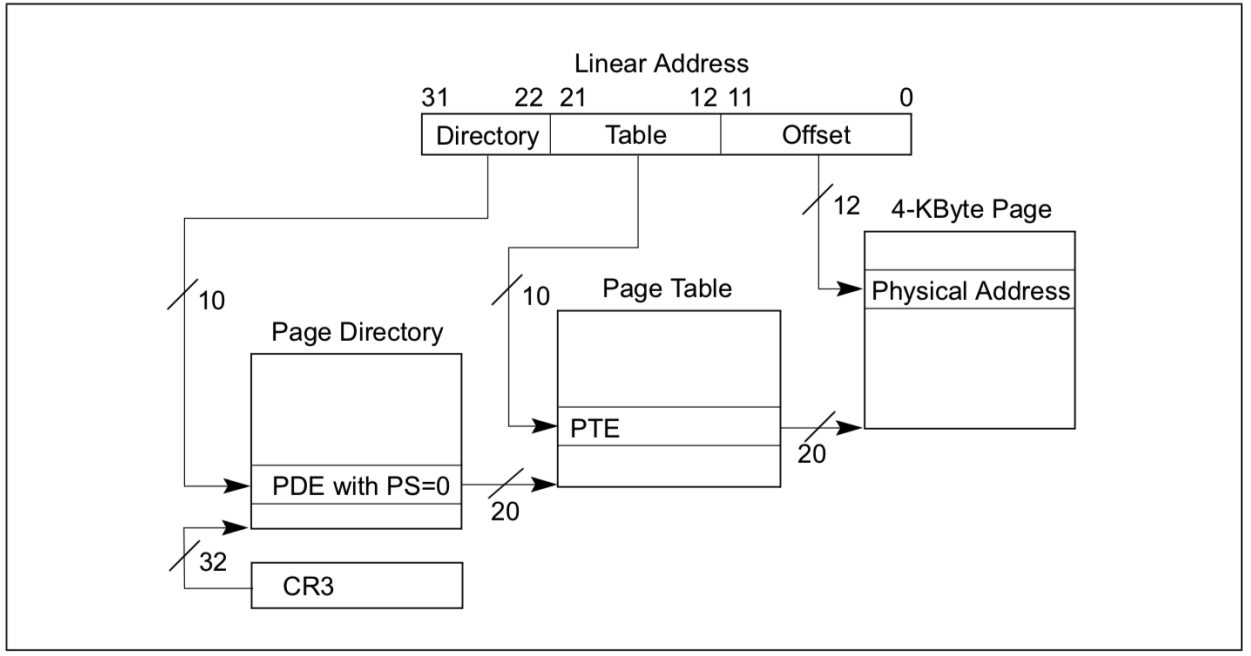
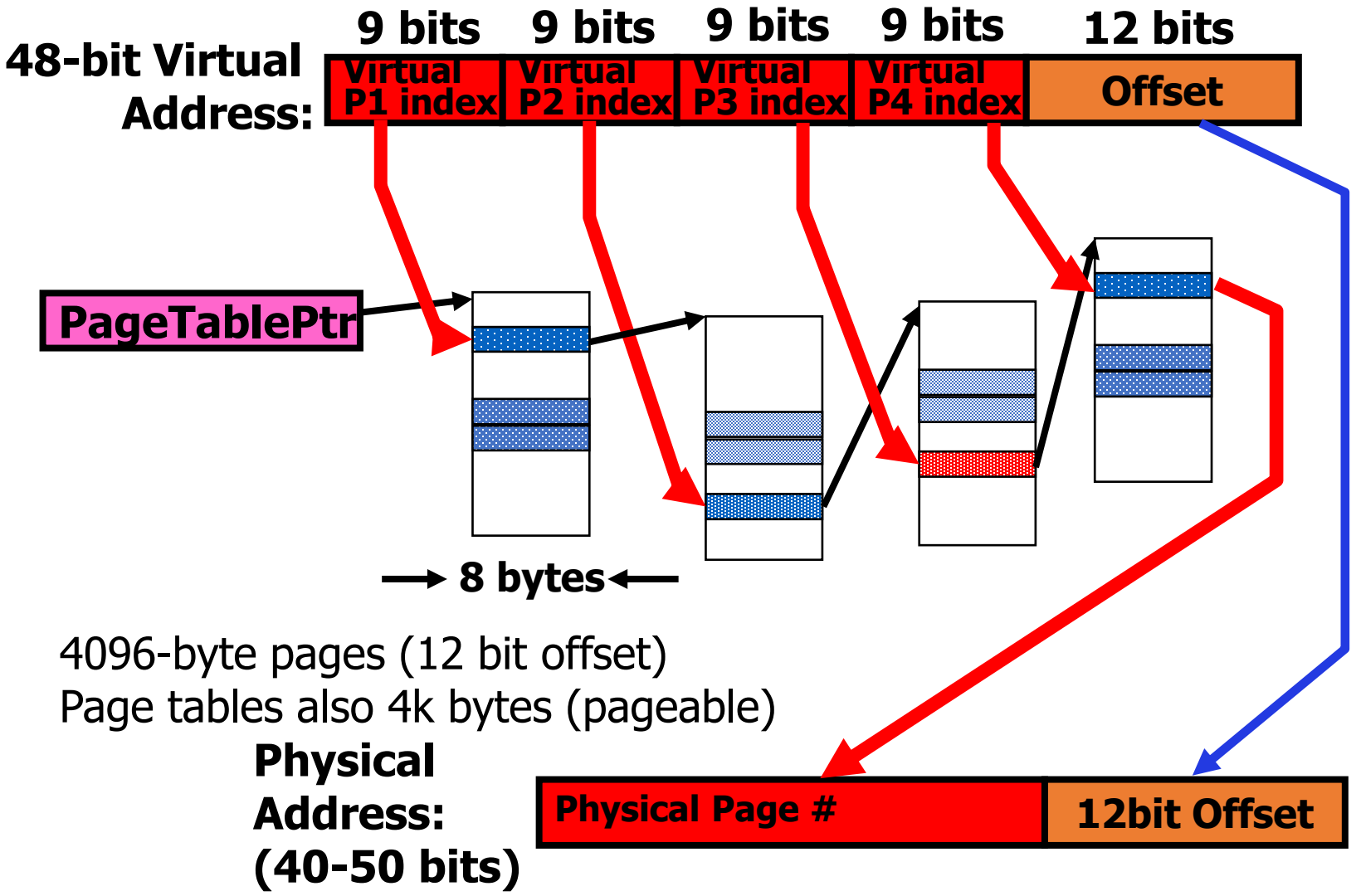


Figure 4-2. Linear-Address Translation to a 4-KByte Page using 32-Bit Paging

Multilevel page tables can keep nesting

- Even page table directory is often sparse, so break it up too
- x86-64
  - Four levels of page table
  - 48-bit addresses (256 TB RAM ought to be enough for everyone right?)



Intel Ice Lake (2019): 5 layers!!

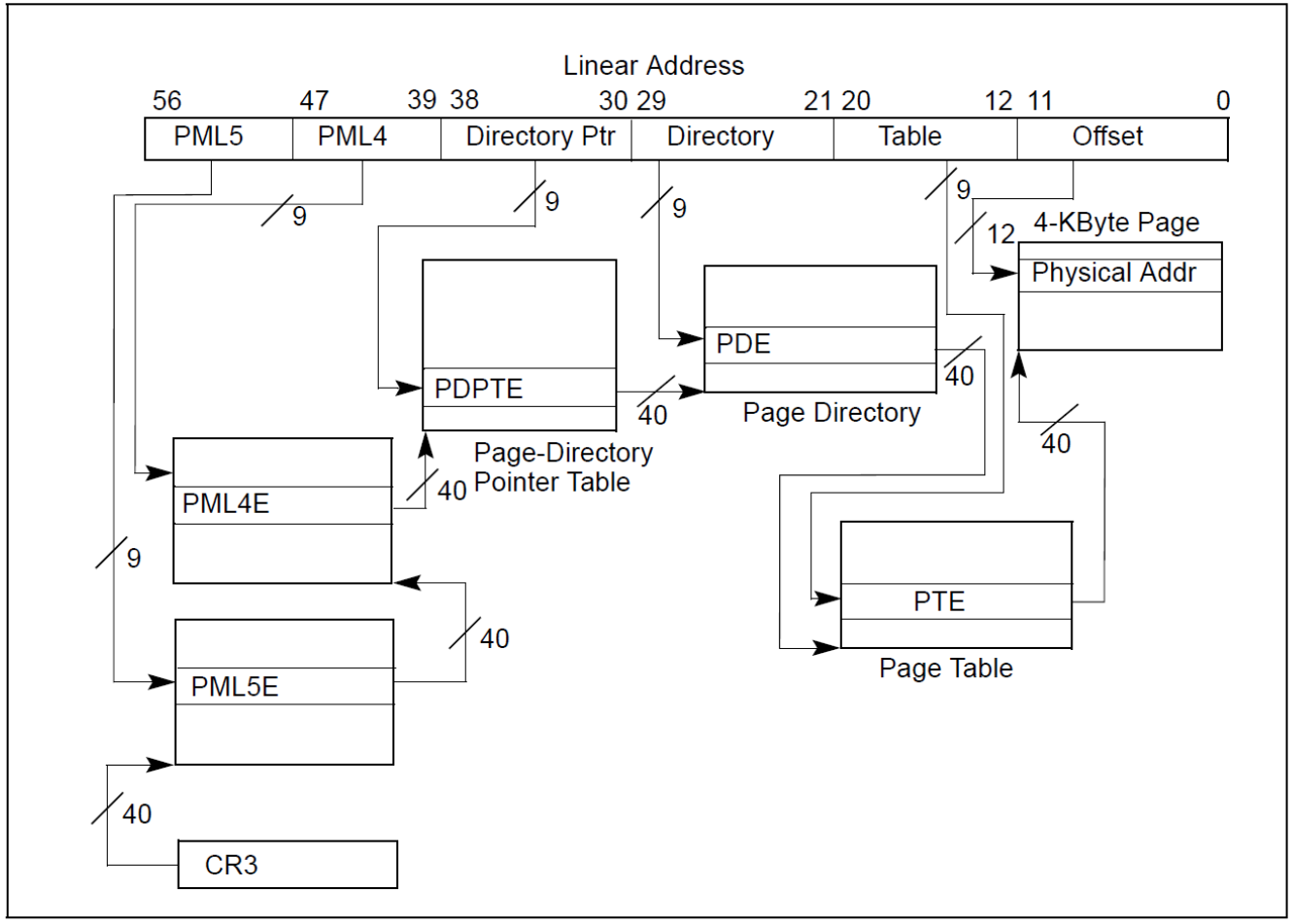


Figure 2-1. Linear-Address Translation Using 5-Level Paging

Check your understanding – multilevel page table

- How many memory loads per read are there now?

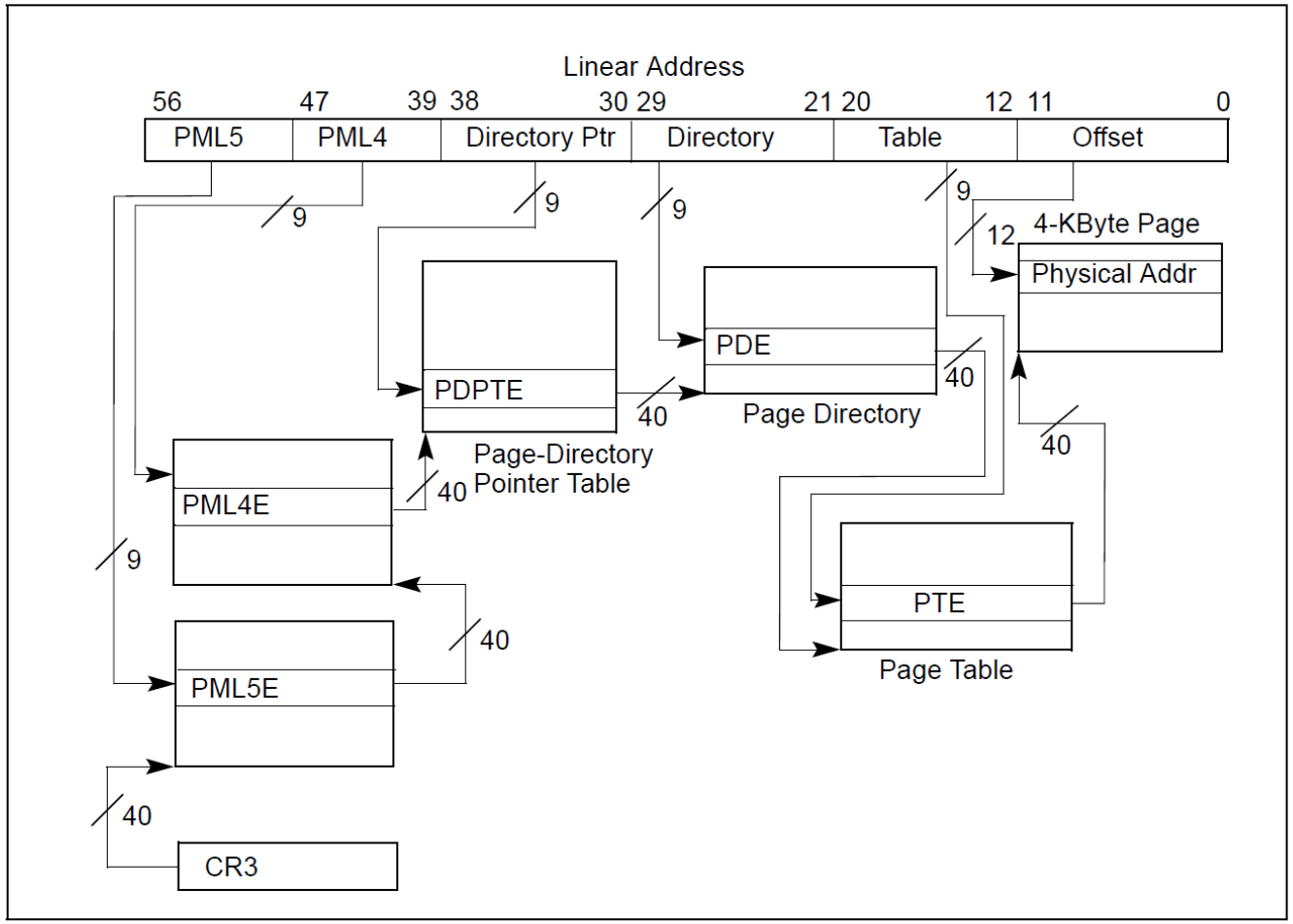


Figure 2-1. Linear-Address Translation Using 5-Level Paging

Check your understanding – multilevel page table

- How many memory loads per read are there now?
  - 6
  - As in each memory access takes six times as long
- TLB is ***extremely*** important

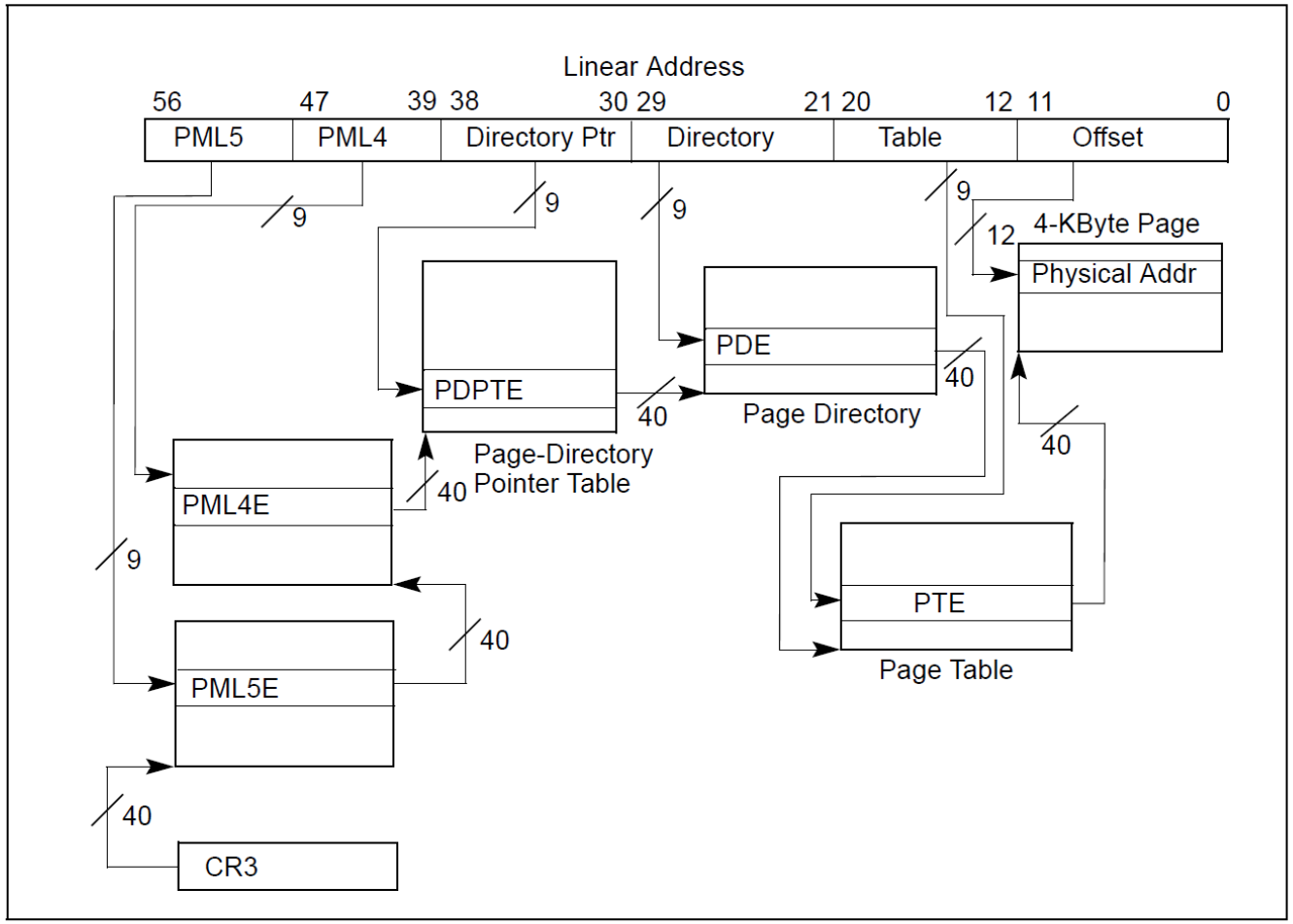


Figure 2-1. Linear-Address Translation Using 5-Level Paging

Additional optimization: large pages

- Always using large pages results in wasted memory
  - Example: 1 MB page where only 1 KB is used
- Always using small pages results in unnecessary page table entries
  - Example: 250 entries in a row to represent 1 MB of memory
- Can we mix in larger pages opportunistically?
  - Small pages normally
  - Large pages occasionally
  - Huge pages rarely

x86-64 allows multiple-sized pages: 4 KB

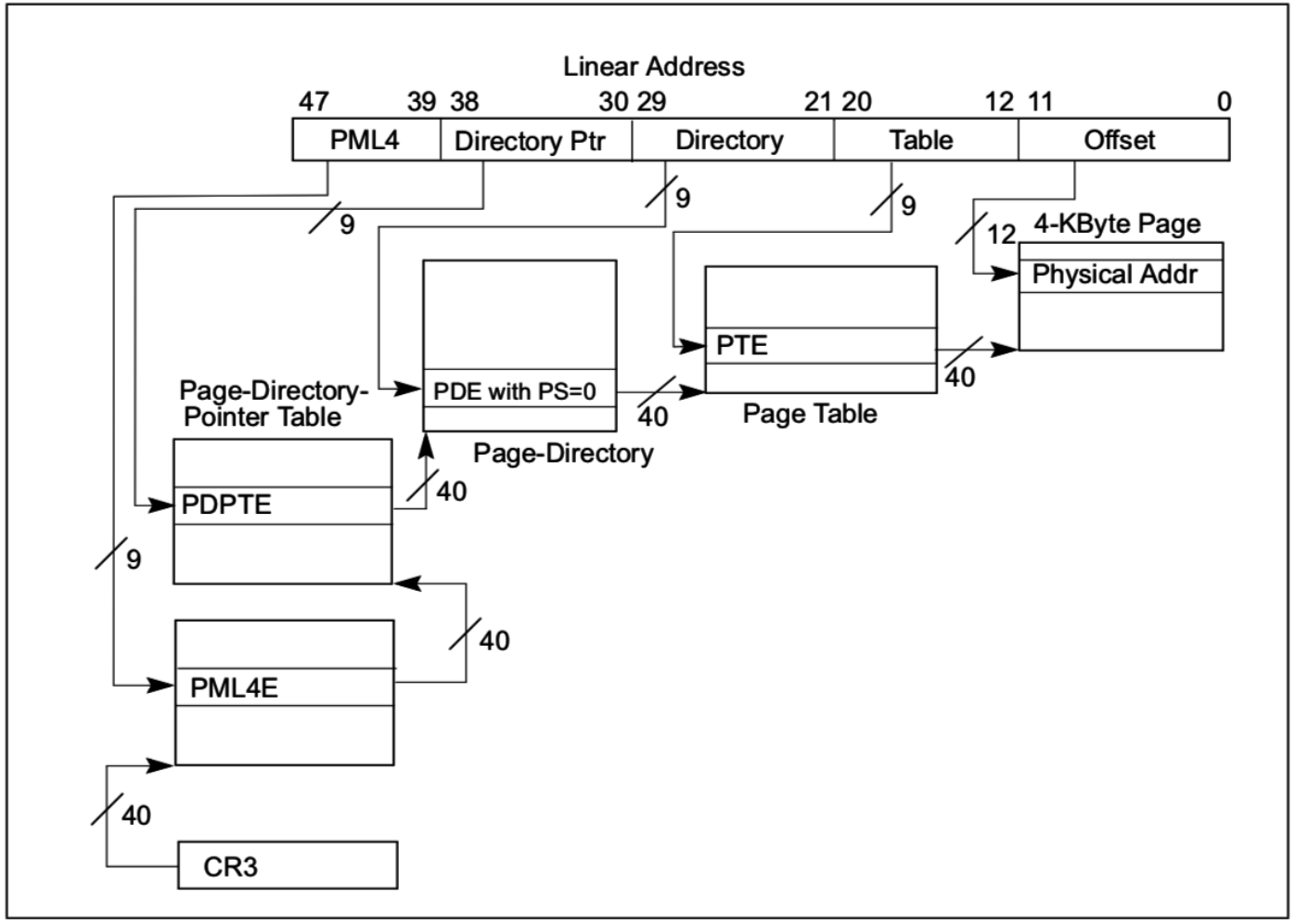


Figure 4-8. Linear-Address Translation to a 4-KByte Page using 4-Level Paging

- Normal x86-64 paging

x86-64 allows multiple-sized pages: 2 MB

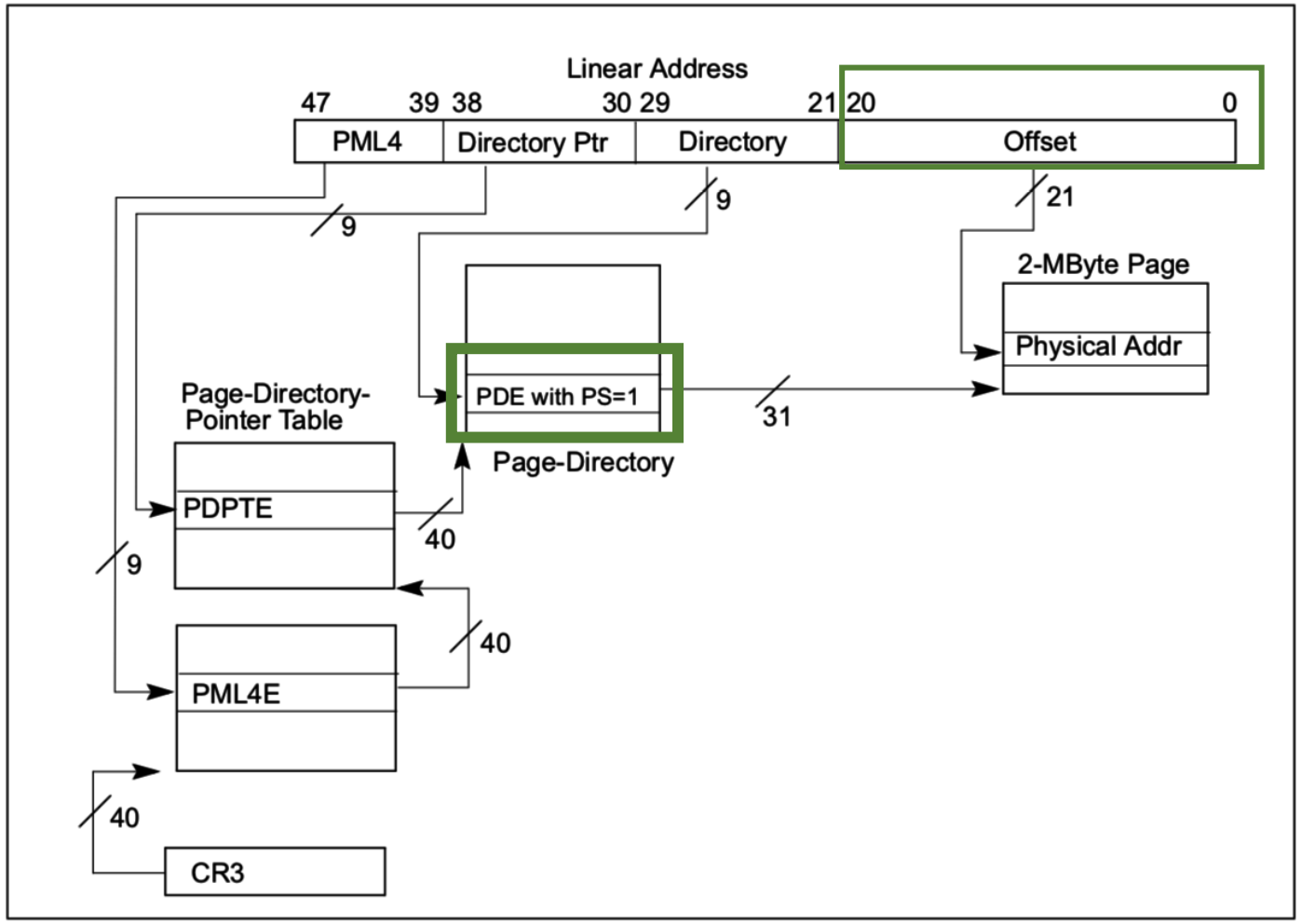


Figure 4-9. Linear-Address Translation to a 2-MByte Page using 4-Level Paging

- Page Size bit triggers walk to skip next table and go straight to 2 MB page in memory
- Remaining address bits are used as offset into larger page

x86-64 allows multiple-sized pages: 1 GB

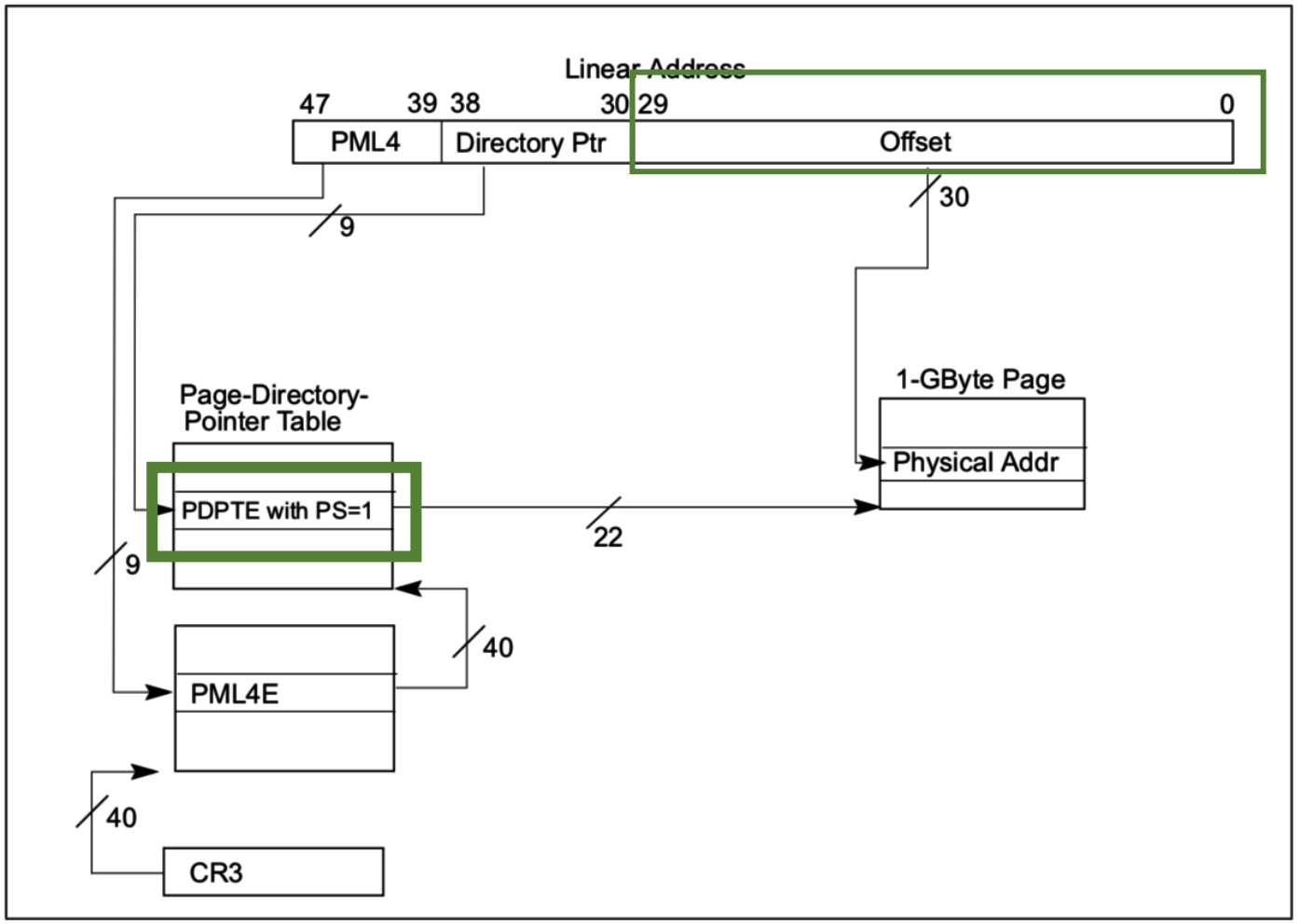


Figure 4-10. Linear-Address Translation to a 1-GByte Page using 4-Level Paging

- Can also skip straight to 1 GB pages
- With a bit of extra hardware, TLB can hold large page entries
  - Occupies a single TLB entry for 1 GB of data (250000 normal entries)

Other data structures for paging

- If hardware handles TLB misses
  - Need a regular structure it can "walk" to find page table entry
  - x86-64 needs to use multilevel page tables

- If software handles TLB misses
  - OS can use whatever data structure it pleases
  - Example: inverted page tables
    - Only store entries for virtual pages with valid physical mappings
    - Use hash of VPN+PCID to find the entry you need

Break + Question

- If every page of virtual memory was used, would a multi-level page table take more or less space than a “flat” page table?

- How often is every page of virtual memory used?

Break + Question

- If every page of virtual memory was used, would a multi-level page table take more or less space than a “flat” page table?
  - More! Still need an entry for every “used” page
  - Now would have to add tree structure as well

- How often is every page of virtual memory used?
  - Never! That would be 18 exabytes of storage in one process

  - For refence: ~44000 exabytes is all of human digital storage (2022)

Outline

- **Paging in modern OS**
- Memory Hierarchy
- Swapping
  - Page Replacement Policies

OS management of processes with paging

- When loading a process
  - OS places actual memory into physical pages in RAM
  - OS creates page table for the process
    - OS decides access permissions to different pages
    - OS connects to shared libraries already in RAM

- When a context switch occurs
  - OS changes which page table is in use (%CR3 register in x86)

- When a fault occurs
  - OS decides how to handle it. (Invalid access or missing page?)

Page faults enable lazy allocation and lazy loading

- Paging is not just translation and overflow
  - Paging provides an opportunity to be lazy about loading requested data

- Trick: don’t load data upfront, do it later when it’s first needed!
  - This is an important performance optimization, *reducing program start time*

Lazy loading in practice

- If a process requests a huge chunk of memory, maybe it will not use all that memory immediately (or ever!).
  - Programmers and compilers are sometimes ***greedy*** in their requests
  - We can *virtually* allocate memory, but mark most of the pages "not present"
  - Let the CPU raise an exception when the memory is really used
  - Then really allocate the demanded page

- Lazy allocation minimizes latency of fulfilling the request and it prevents OS from allocating memory that will not be used.

Extra features of lazy loading

- Lazy loading also works for large code binaries
  - Delay loading a page of instructions until it's needed

- OS must also write zeros to newly assigned physical frames
  - Program does not necessarily expect the new memory to contain zeros,
  - But we clear the memory for security, so that other process' data is not leaked.
  - OS can keep one read-only physical page filled with zeros and just give a reference to this at first.
    - After the first page fault (due to writing a read-only page), allocate a real page.

Lazy allocation via copy-on-write with Fork

- Recall that ***fork + exec*** is the only way to create a child process in unix

- Fork clones the entire process, including all virtual memory
  - This can be very slow and inefficient, especially if the memory will just be overwritten by a call to **exec**.

Lazy allocation via copy-on-write with Fork

- ***Copy on write*** is a performance optimization:
  - Don't copy the parent's pages, ***share*** them
    - Make the child process' page table point to the parent's physical pages
    - Mark all the pages as "read only" in the PTEs (temporarily)
  - If parent or child writes to a shared page, a page fault exception will occur
  - OS handles the page fault by:
    - Copying parent's page to the child & marking both copies as writeable
    - When the faulting process is resumed, it retries the memory write.

Virtual memory in practice

- On Linux, the `pmap` command shows a process' VM mapping.
- We see:
  - OS tracks which file code is loaded from, so it can be lazily loaded
  - The main process binary and libraries are ***lazy loaded***, not fully in memory
  - Libraries have read-only sections that can be shared with other processes
- `cat /proc/<pid>/smaps` shows even more detail

References:

- https://unix.stackexchange.com/a/116332
- https://www.akkadia.org/drepper/dsohowto.pdf

pmap on emacs

```
[spt175@murphy ~]$ pmap -x 1122
1122:   emacs kernel/proc.c
Address           Kbytes     RSS   Dirty Mode   Mapping
0000000000400000    2032    1344       0 r-x--  emacs-23.1
00000000007fb000    8856    8192    6140 rw---  emacs-23.1
0000000001dd5000    1204    1204    1204 rw---    [ anon ]
00000035cc600000      16      12       0 r-x--  libuuid.so.1.3.0
00000035cc604000    2044       0       0 -----  libuuid.so.1.3.0
00000035cc803000       4       4       4 rw---  libuuid.so.1.3.0
00000035cca00000      28      12       0 r-x--  libSM.so.6.0.1
00000035cca07000    2048       0       0 -----  libSM.so.6.0.1
00000035ccc07000       4       4       4 rw---  libSM.so.6.0.1
00000035d0e00000      32      12       0 r-x--  libgif.so.4.1.6
00000035d0e08000    2048       0       0 -----  libgif.so.4.1.6
00000035d1008000       4       4       4 rw---  libgif.so.4.1.6
0000003f65a00000     128     116       0 r-x--  ld-2.12.so
0000003f65c20000       4       4       4 r----  ld-2.12.so
0000003f65c21000       4       4       4 rw---  ld-2.12.so
0000003f65c22000       4       4       4 rw---    [ anon ]
0000003f65e00000    1576     536       0 r-x--  libc-2.12.so
0000003f65f8a000    2048       0       0 -----  libc-2.12.so
0000003f6618a000      16      16       8 r----  libc-2.12.so
0000003f6618e000       8       8       8 rw---  libc-2.12.so
```

- "Mapping" shows source of the section, more code can be loaded from here later.
  - "**anon**" are regular program data, requested by *sbrk* or *mmap*. (In other words, heap data.)
- Each library has several sections:
  - "r-x--" for code
  - "r----" for constants

  } *can be shared*
  - "rw---" for global data
  - "-----" for guard pages:

    (not mapped to anything, just reserved to generate page faults)
- **RSS** means resident in physical mem.
- **Dirty** pages have been written and therefore cannot be shared with others

top has a column showing shared memory

```
top - 10:25:45 up 7 days, 48 min,  3 users,  load average: 0.04, 0.06, 0.09
Tasks: 650 total,   1 running, 649 sleeping,   0 stopped,   0 zombie
Cpu(s):  0.0%us,  0.0%sy,  0.0%ni, 99.9%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:  132144848k total, 129331984k used,  2812864k free, 37895660k buffers
Swap: 16383996k total,      436k used, 16383560k free, 45074412k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 9213 mysql     20   0 1263m 156m  14m S  0.0  0.1   3:57.24 mysqld
10001 root      20   0 5748m 219m  14m S  0.3  0.2  15:02.22 dsm_om_connsvcd
 9382 root      20   0  337m  18m  11m S  0.0  0.0   0:10.67 httpd
 8304 apache    20   0  352m  19m  10m S  0.0  0.0   0:00.29 httpd
 8302 apache    20   0  339m  14m 7144 S  0.0  0.0   0:00.16 httpd
 8298 apache    20   0  339m  14m 7140 S  0.0  0.0   0:00.12 httpd
 8299 apache    20   0  339m  14m 7136 S  0.0  0.0   0:00.17 httpd
 8303 apache    20   0  339m  14m 7136 S  0.0  0.0   0:00.17 httpd
 8300 apache    20   0  339m  14m 7120 S  0.0  0.0   0:00.13 httpd
 8301 apache    20   0  339m  14m 7120 S  0.0  0.0   0:00.16 httpd
 8305 apache    20   0  339m  14m 7112 S  0.0  0.0   0:00.13 httpd
 1386 apache    20   0  339m  14m 7096 S  0.0  0.0   0:00.06 httpd
 1387 apache    20   0  339m  14m 7084 S  0.0  0.0   0:00.07 httpd
 1122 spt175    20   0  251m  14m 6484 S  0.0  0.0   0:00.26 emacs
 2615 root      20   0 92996 6200 4816 S  0.0  0.0   0:00.93 NetworkManager
 9865 root      20   0 1043m  23m 4680 S  0.3  0.0   9:44.98 dsm_sa_datamgrd
 8737 postgres  20   0  219m 5380 4588 S  0.0  0.0   0:01.00 postmaster
 2786 haldaemo  20   0 45448 5528 4320 S  0.0  0.0   0:03.99 hald
 9956 root      20   0  491m 7268 3280 S  0.0  0.0   3:16.30 dsm_sa_snmpd
  990 root      20   0  103m 4188 3172 S  0.0  0.0   0:00.01 sshd
 1014 root      20   0  103m 4196 3172 S  0.0  0.0   0:00.02 sshd
19701 root      20   0  103m 4244 3172 S  0.0  0.0   0:00.01 sshd
```

- The duplicate processes are using a lot of shared memory:
  - ~50% of resident memory for `httpd` is shared
    ~75% of resident memory for `sshd` is shared
- Even if there is just one instance of `emacs` running, it may share many libraries with other running programs.
- Total virtual memory is ~10x larger than resident memory
  - Processes only use a small fraction of their VM!
  - Due to sharing and lazy loading.

To see virtual memory info on Linux

- cat /proc/meminfo
- vmstat
- top
  - (resident)

```
[spt175@murphy ~]$ cat /proc/meminfo
MemTotal:       132144848 kB
MemFree:        130263996 kB
Buffers:           63880 kB
Cached:           539824 kB
SwapCached:            0 kB
Active:           665300 kB
Inactive:         323932 kB
Active(anon):     385768 kB
Inactive(anon):     2460 kB
Active(file):     279532 kB
Inactive(file):   321472 kB
Unevictable:           0 kB
Mlocked:               0 kB
SwapTotal:      16383996 kB
SwapFree:       16383996 kB
Dirty:                96 kB
Writeback:             0 kB
AnonPages:        387972 kB
Mapped:            61012 kB
Shmem:              2688 kB
Slab:              88844 kB
SReclaimable:      28140 kB
SUnreclaim:        60704 kB
KernelStack:       12672 kB
PageTables:        15000 kB
NFS_Unstable:          0 kB
Bounce:                0 kB
WritebackTmp:          0 kB
CommitLimit:    82456420 kB
Committed_AS:    1659096 kB
VmallocTotal:   34359738367 kB
VmallocUsed:       486616 kB
VmallocChunk:   34291646280 kB
HardwareCorrupted:     0 kB
AnonHugePages:    276480 kB
HugePages_Total:       0
HugePages_Free:        0
HugePages_Rsvd:        0
HugePages_Surp:        0
Hugepagesize:       2048 kB
DirectMap4k:        5604 kB
DirectMap2M:     2078720 kB
DirectMap1G:    132120576 kB
```

Requesting memory from the OS – brk()

- System call to change data segment size (the program "break")
  - Either set a new virtual address pointer for top of data segment
  - Or increment the size of the data segment by N bytes

- These are the old system calls to dynamically change program memory
  - How malloc creates space

- "sbrk() and brk() are considered legacy even by 1997 standards"
  - Removed from POSIX in 2001
  - Still exists in some form in lots of OSes (including Nautilus)

Modern requesting memory from the OS – mmap()

- Map (or unmap) files or devices into memory

- Given a file, places the file in the process's virtual address space
  - Process can request an address to place it at, which OS *might* follow

- Given flag MAP_ANONYMOUS, creates empty memory
  - Initialized to zero and accessible from process
  - Malloc implementation uses this

- Many other options
  - Create huge page, create memory for a stack, shared memory

Break + Consideration

- Why use mmap() to put a file in your address space, when you could just read()/write() it instead?

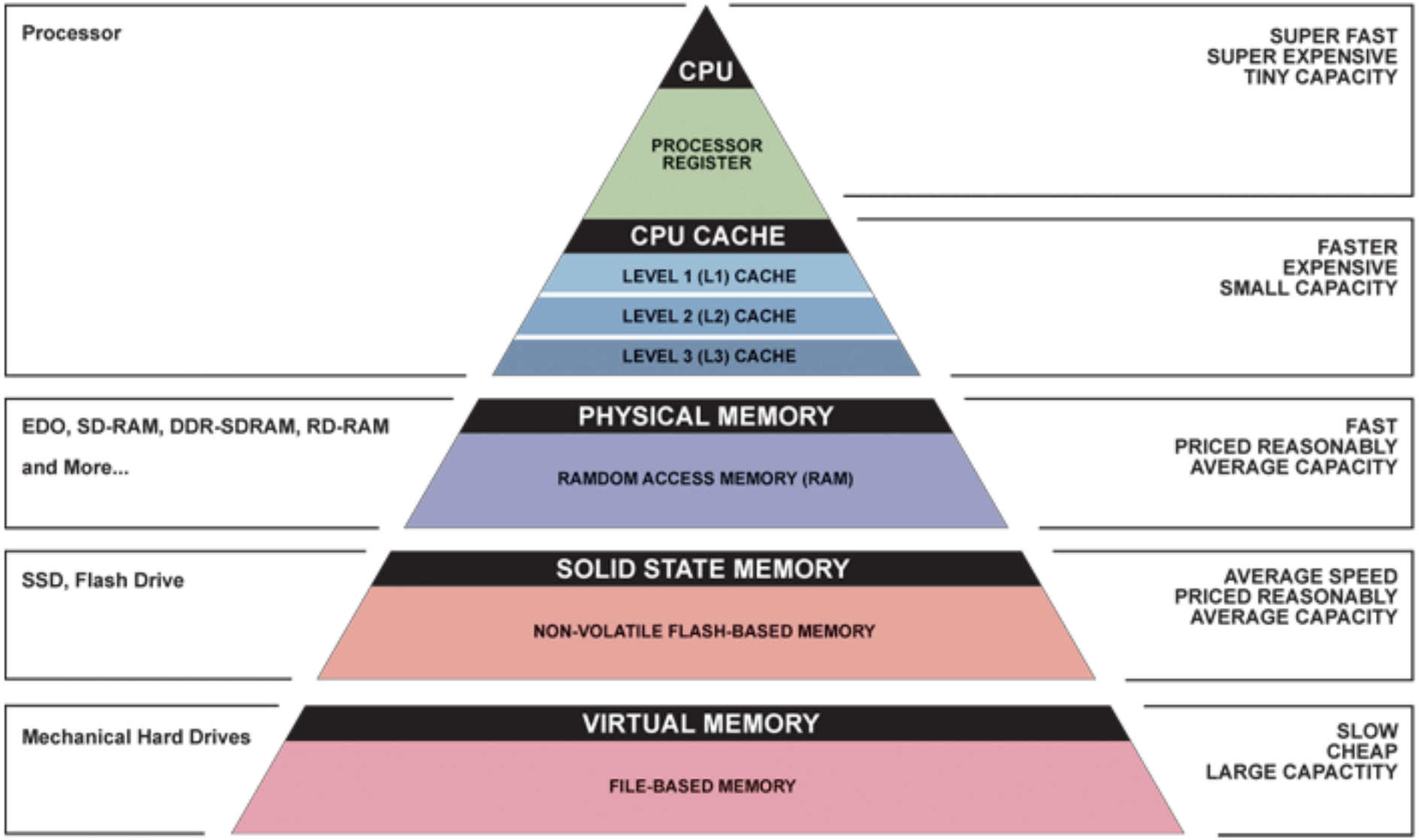
Break + Consideration

- Why use mmap() to put a file in your address space, when you could just read()/write() it instead?
  - Speed! No longer need to make system calls for each file access
  - A downside: now you need to handle file interactions yourself
    - Track offset for reading and writing
    - Make sure you don’t go past the end of the file

Outline

- Paging in modern OS
- **Memory Hierarchy**
- Swapping
  - Page Replacement Policies

Memory Hierarchy



The OS view on registers

- Illusion: separate set for each process
- Reality: separate set for each core (or each thread in a core)
- OS needs to save and update registers whenever the currently running process changes
- Process and hardware handle moving memory into registers

The OS view on caches

- Mostly ignore them, handled by the hardware automatically
  - Occasionally might need to clear them for security purposes

- Addresses in the caches are either entirely physical addresses
- Or are virtually indexed, physically tagged
  - Cache lookup and TLB lookup happen in parallel
  - TLB result is used as Tag for cache to determine if there was a hit

The OS view on memory

- Managed through virtual memory translation
  - Paging (or Segmentation) that we talked about last time

- OS chooses which portions of processes go in RAM
  - Other portions of memory get “swapped” to disk
  - Writeable memory regions (stack, heap, global data) must be preserved
  - Read-only memory regions (code) can be reloaded from original location

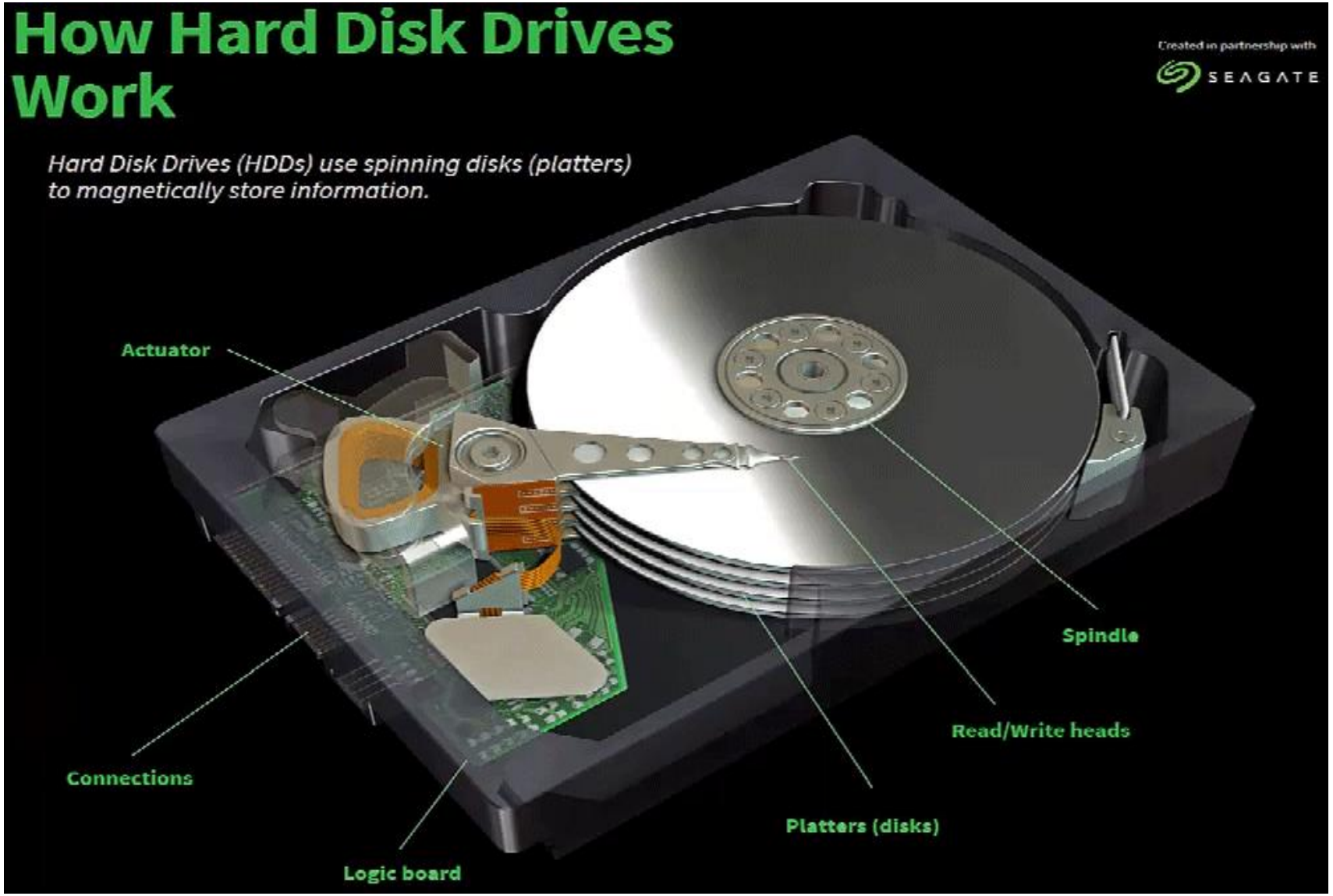
The OS view on disk

- Non-volatile memory store
  - Everything else on the system disappears when power is removed (and cannot be trusted across reboots)

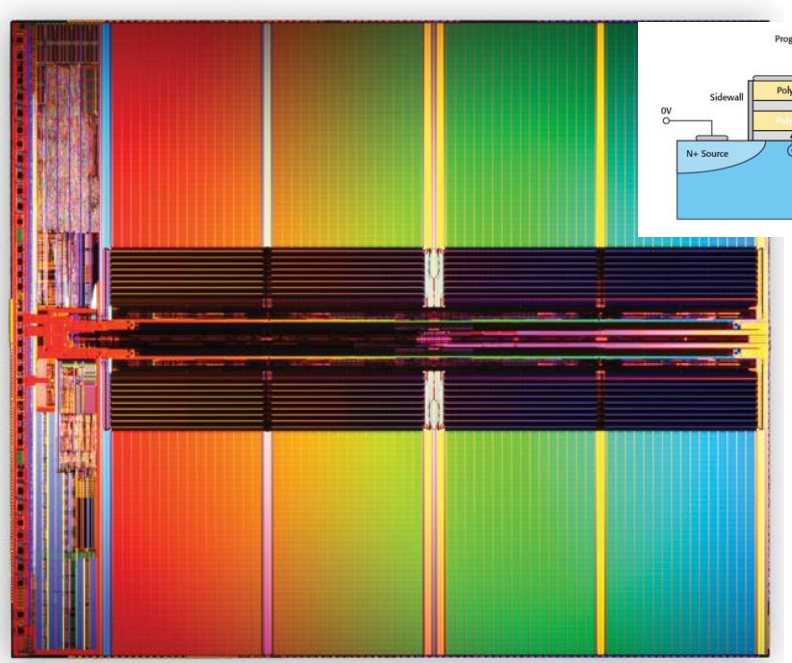
- Backing store for lots of information
  - Boot information: via "Master Boot Record" on disk
  - Filesystem, which the OS manages access to through system calls
  - Swap space, which the OS moves extra pages in and out of
    - Disk is significantly bigger than RAM, so this will work

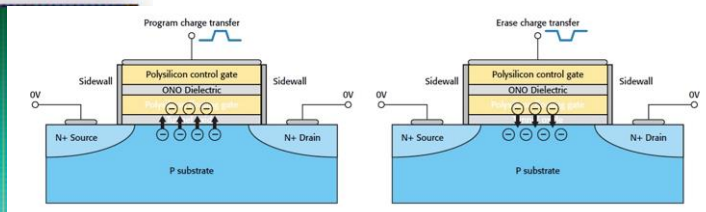
- Disk is a device that the OS manages and reads in "blocks"
  - Compare to memory, which is directly addressed by processes

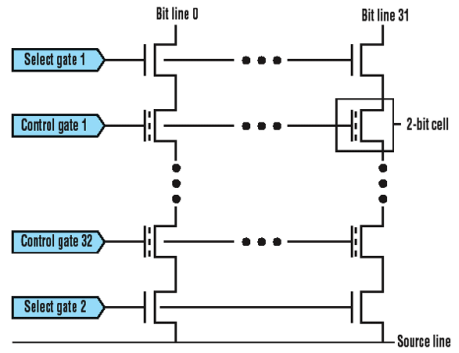
Traditional hard disk drives (HDDs) use magnetic regions



Solid state drives (SSDs) use flash memory







In the basic functional block used in multilevel NAND flash memories, 32 rows of bit lines and 32 control-gate lines form a building block that's repeated many times to form the memory array. The select gate lines are used with the control gate lines to control access to the array.

2. Micron's triple-level cell (TLC) flash memory stores 3 bits of data in each transistor.

NMOS transistor with an additional conductor between gate and source/drain which "traps" electrons. The presence/absence is a 1 or 0

- Still non-volatile
- Significantly faster
  - 0.1 ms to access (10 ms for disk)
- More limited lifetime than disk
  - Limited writes

Outline

- Paging in modern OS
- Memory Hierarchy
- **Swapping**
  - Page Replacement Policies

Motivation for swapping

- Processes should be independent of the amount of physical memory
  - Should be correct, even if not performant

- OS goal: support processes when not enough physical memory
  - Multiple processes combining to more than physical memory
  - Single process with very large address space
    - Video games: Red Dead Redemption 2 – 150 GB
    - Large-scale data processing: Compiling Android – 16 GB

- OS provides illusion of more physical memory by using disk

Locality of reference

- If disk is involved with memory, won't this be ridiculously slow?

- Leverage *locality of reference* within process
  - **Spatial**: memory addresses near referenced address likely to be next
  - **Temporal**: referenced addresses likely to be referenced again
  - Processes spend majority of time in a small portion of code
    - Estimate: 90% of time spent in 10% of code (loops)

- Implication
  - Process only uses small amount of address space at any moment
  - Only small amount of address space needs to be in physical memory
  - RAM acts as a sort of cache for program memory

How swapping works

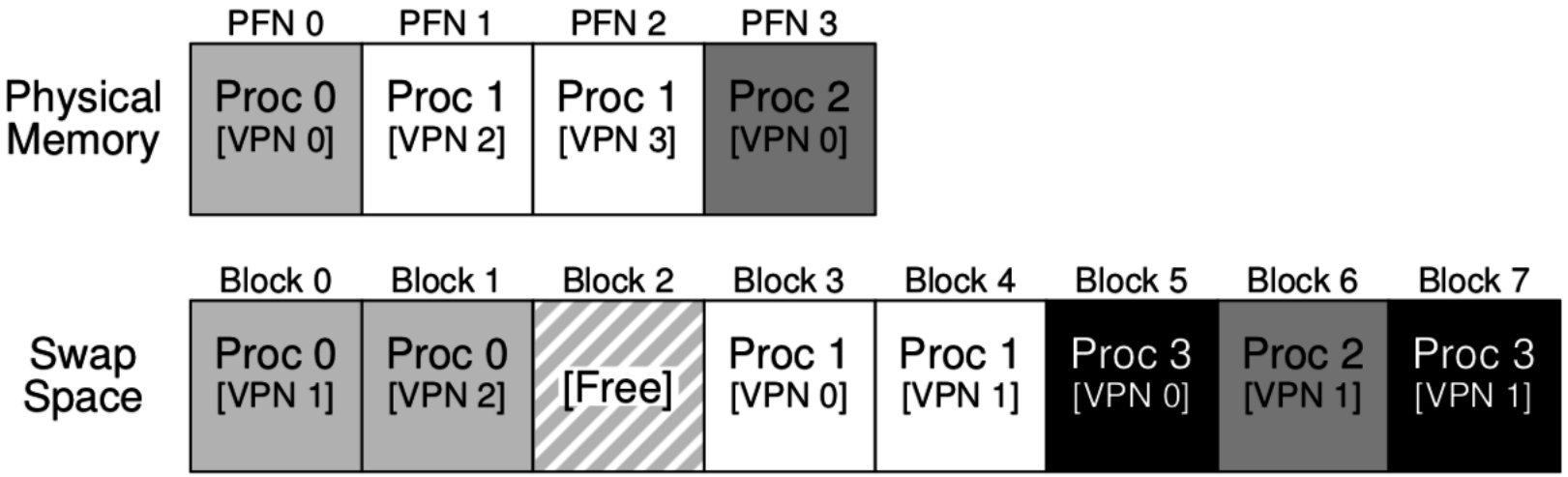
- OS moves unreferenced pages to disk

- Processes can still run when not all pages are in physical memory
- OS and hardware cooperate to make memory available when needed
  - Same behavior as if all of address space always was in memory
  - Except in terms of time, but processes don’t know about time…

- Requirements
  - OS needs **mechanism** to identify location of address space pages on disk and move them into RAM when necessary
  - OS needs **policy** to determine which pages go in RAM or disk

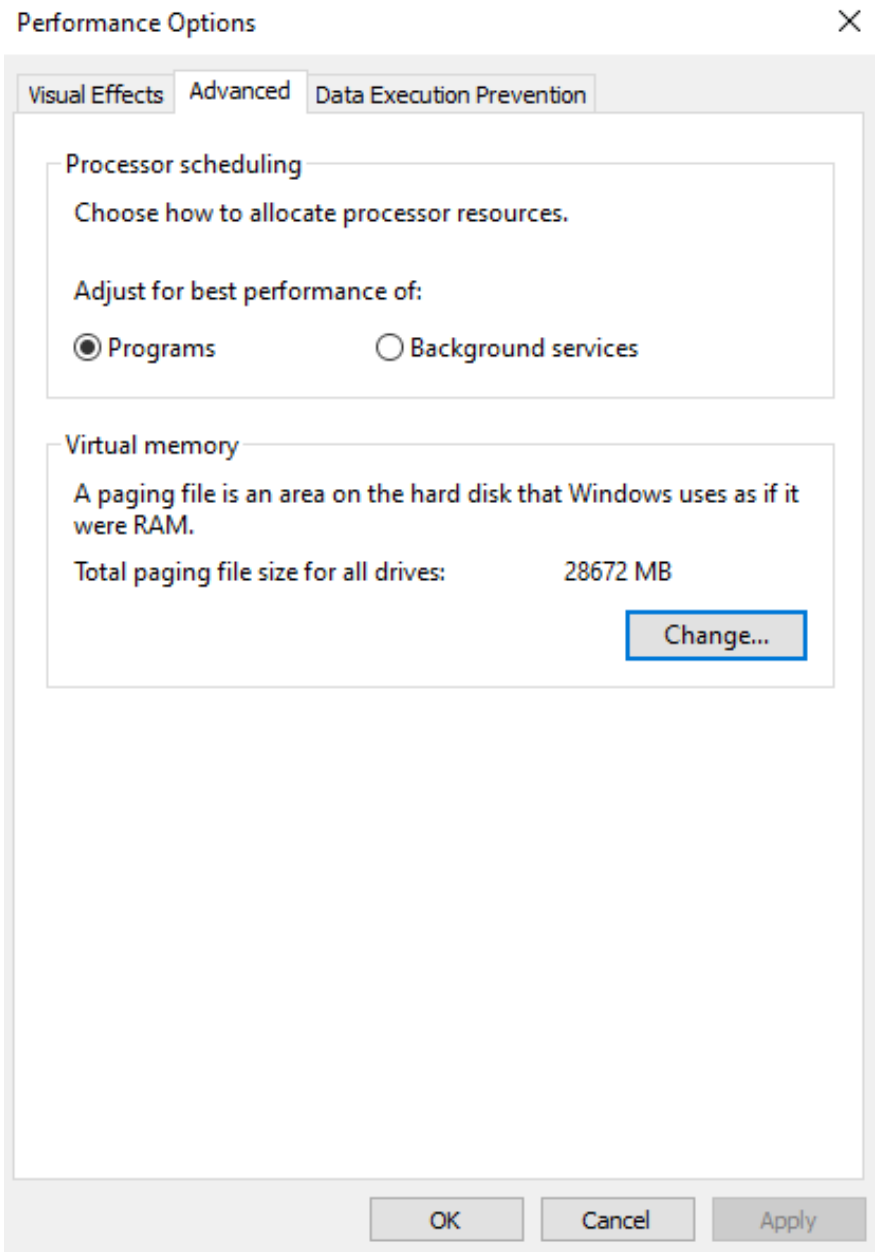
Combination of swapping and paging

- Processes have memory pages, which are distributed among RAM and Disk
- Example:
  - Processes 0, 1, and 2 are partially in RAM
  - Process 3 is entirely in "swap space" on disk



Paging on Windows

- Windows lets you see and even set the size of swap space on disk



Mechanisms to support swapping

- Each page in virtual address space lives in a location
  1. Physical memory
  2. Disk
  3. Nowhere

- Extend page tables with an extra bit – present
  - Physical Page Number, Permissions, Valid, Present
    - Page in memory, valid and present

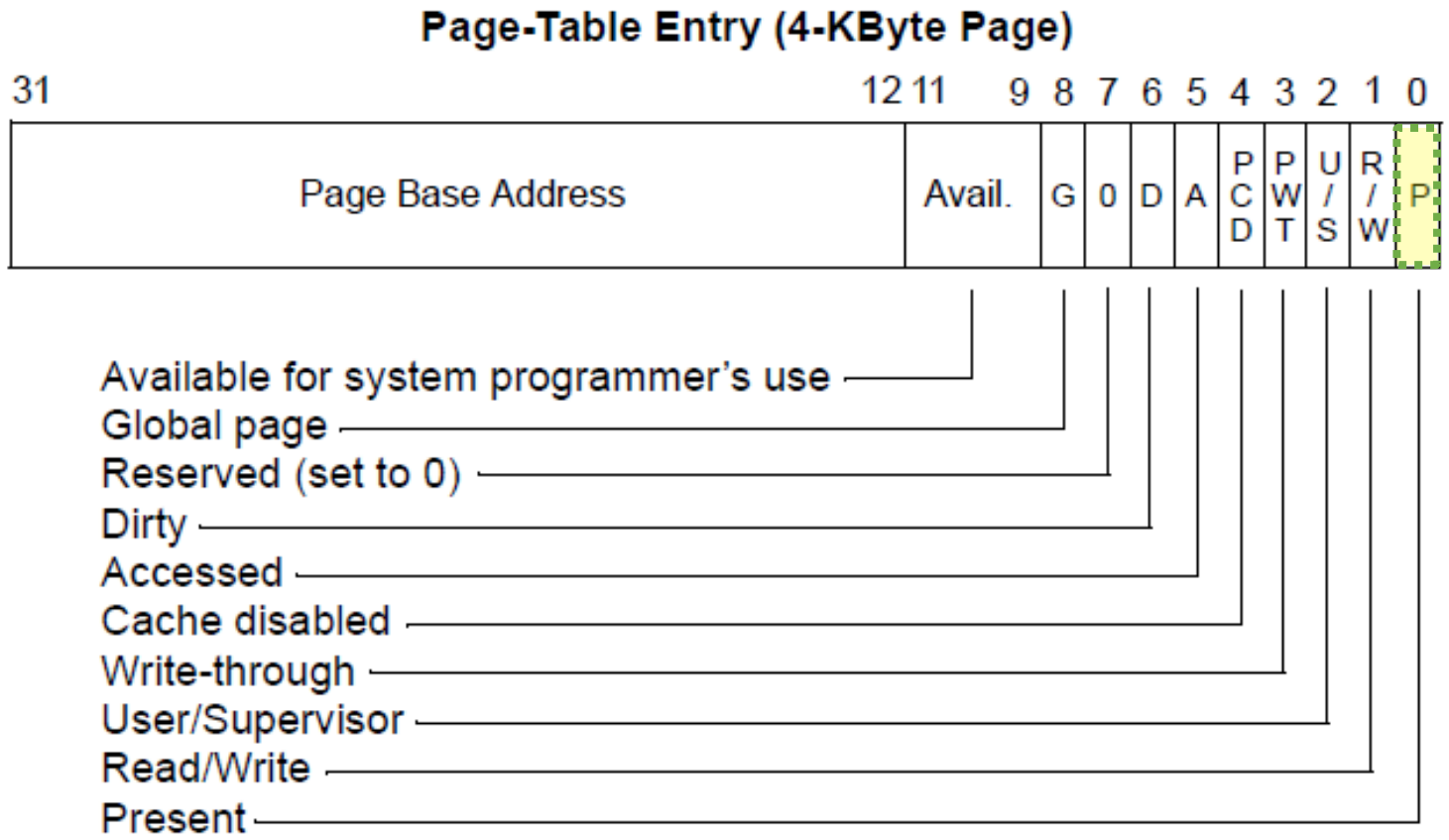
    - Page on disk, valid but not present
      - Page Table Entry points to block on disk instead!
      - Trap to OS on reference

    - Invalid page, not valid and not present OR bad permissions
      - Trap to OS on reference

Other bits in a page table entry

- Page Base Address can be reused to hold disk block

- Dirty bit
  - Whether page has been modified
  - If page needs to be swapped out, only preserve if modified



Steps to a memory access with swapping

1. Hardware checks TLB for virtual address
   - If Hit, address translation complete AND page in physical memory
2. Hardware (or OS) walks page tables
   - If valid and present, then page in physical memory
3. Trap into OS
   - If invalid or bad permissions, fault process (segmentation fault)
   - If valid but not present
     - If memory is full, select a victim page in memory to replace
       - If modified (dirty), write to disk
       - Invalidate TLB entry for that page
     - OS reads referenced page from disk into memory
     - Page table is updated, present bit is set
     - Resume process execution (could be really complicated on CISC machines)

Types of page faults

- **Minor/soft**: Page is loaded in memory, but PTE is not configured:
  - Memory could be a shared library already in memory from another process.
  - OS could be tracking accesses to this page. (hardware without a dirty bit)

  Response: update the PTE.

- **Major/hard**: A disk access will be needed:
  - Anonymous page (process data) may have been swapped out.
  - Lazy-loading program executable.

  Response: load the page from disk

- **Invalid**: User program misbehaved:
  - Dereference null or invalid pointer.
  - Write to page that is read-only.
  - Execute code on a page that is not executable (for security).

  Response: terminate the process.

Policies to determine swapping evictions

- Goal: minimize the number of page faults
  - Page faults need to read/write from disk and are very slow
  - So the OS can take plenty of time to make a *good* decision

- OS has two decisions
  1. Page Selection
     - When should a page be brought into memory?

  2. Page Replacement
     - When should a page be swapped into disk?
     - Which page should be swapped out of physical memory?

When do we load in pages? (page selection)

- Demand paging: Load page only when page fault occurs
  - Intuition: Wait until page must absolutely be in memory
  - When process starts: No pages are loaded in memory
  - Problems: Pay cost of page fault for every newly accessed page

- Pre-paging (prefetching): Load page before referenced
  - OS predicts future accesses and brings pages into memory early
  - Works well for some access patterns (e.g., sequential)

- Hints: Combine above with user-supplied hints about page references
  - User specifies: may need page in future, don't need this page anymore, or sequential access pattern, ...
  - Example: `madvise`() in POSIX – "give advice about use of memory"

When do we swap out pages? (page replacement)

- Demand swapping: whenever the page fault actually occurs
  - Simplest method
  - Swap actually occurs asynchronously
    - Start the disk I/O and block the process that faulted

- Background swapping: preemptively when RAM is getting full
  - Background service in kernel periodically runs (kswapd)
  - If number of free physical pages < “low water mark”, evict a bunch
    - Writing many pages to disk in one operation is way more efficient

Thrashing

- Thrashing: when swapping happens frequently
  - Policy could be bad (working set keeps getting swapped to disk)
  - More likely RAM is too small

- Frequent swapping slows down the whole computer to a crawl
  - Constantly waiting on disk I/O

- Solution for thrashing
  - Kill processes until it stops (relieves memory pressure)
  - Install more RAM in the computer

Outline

- Paging in modern OS
- Memory Hierarchy
- Swapping
  - **Page Replacement Policies**

Which page should be evicted?

- **Page replacement policy** determines page to evict

- Very similar process as cache eviction or TLB eviction
  - Misses are expensive, so make sure you evict the right page
  - Page faults are extremely long, so a sophisticated policy is possible

Optimal page replacement policy

- Optimal page replacement
  - Evict page that will be accessed furthest in the future
- Advantages
  - Guaranteed to minimize the number of page faults
- Disadvantages
  - Requires the OS to predict the future and therefore cannot exist
- Performance upper bound
  - This is the best anything can do, so it is useful to compare against
  - Still has misses due to cold-start and capacity

First-In-First-Out replacement policy

- FIFO replacement
  - Evict page that has been in memory the longest

- Advantages
    - Fair as all pages have equal residency
    - Easy to implement

- Disadvantages
    - Some pages of memory are always needed (stack)
      - Memory doesn’t really need “fairness” like processes did

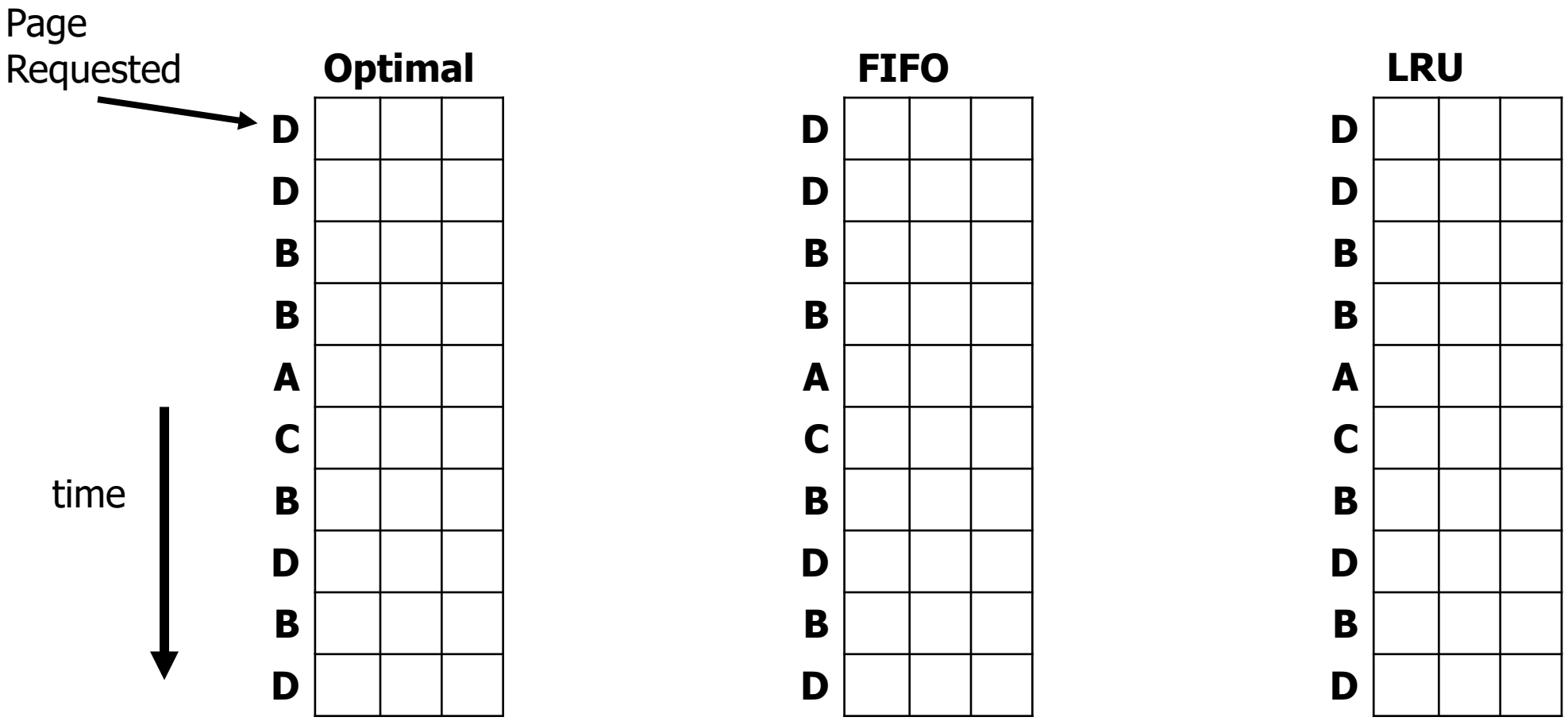
Least Recently Used replacement policy

- LRU replacement
  - Replace page not accessed for longest time
  - Using the past to predict the future (temporal locality)

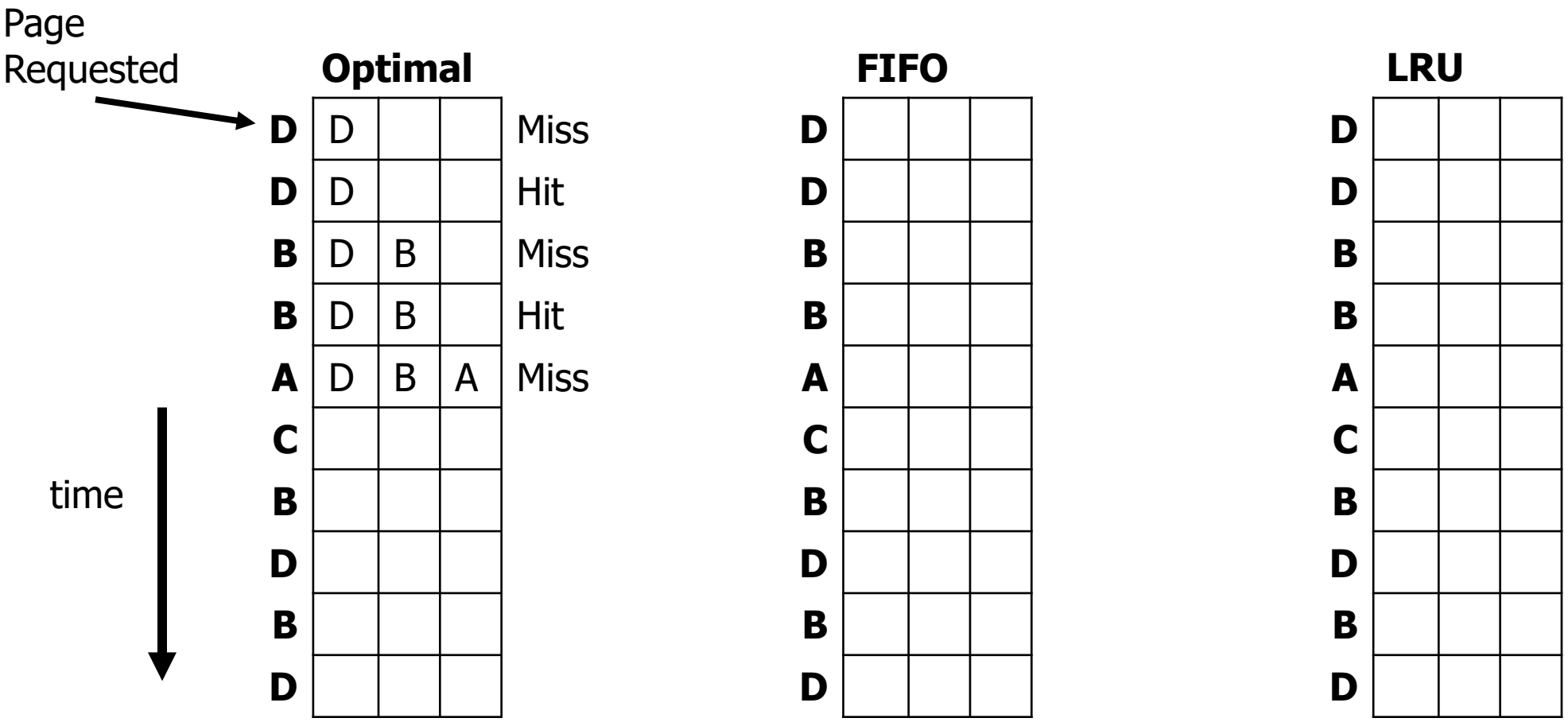
- Advantages
  - With locality, LRU approximates Optimal

- Disadvantages
  - Harder to implement as we need to track when pages are accessed
  - Cyclical patterns can make LRU fail (bigger concern for cache than RAM)

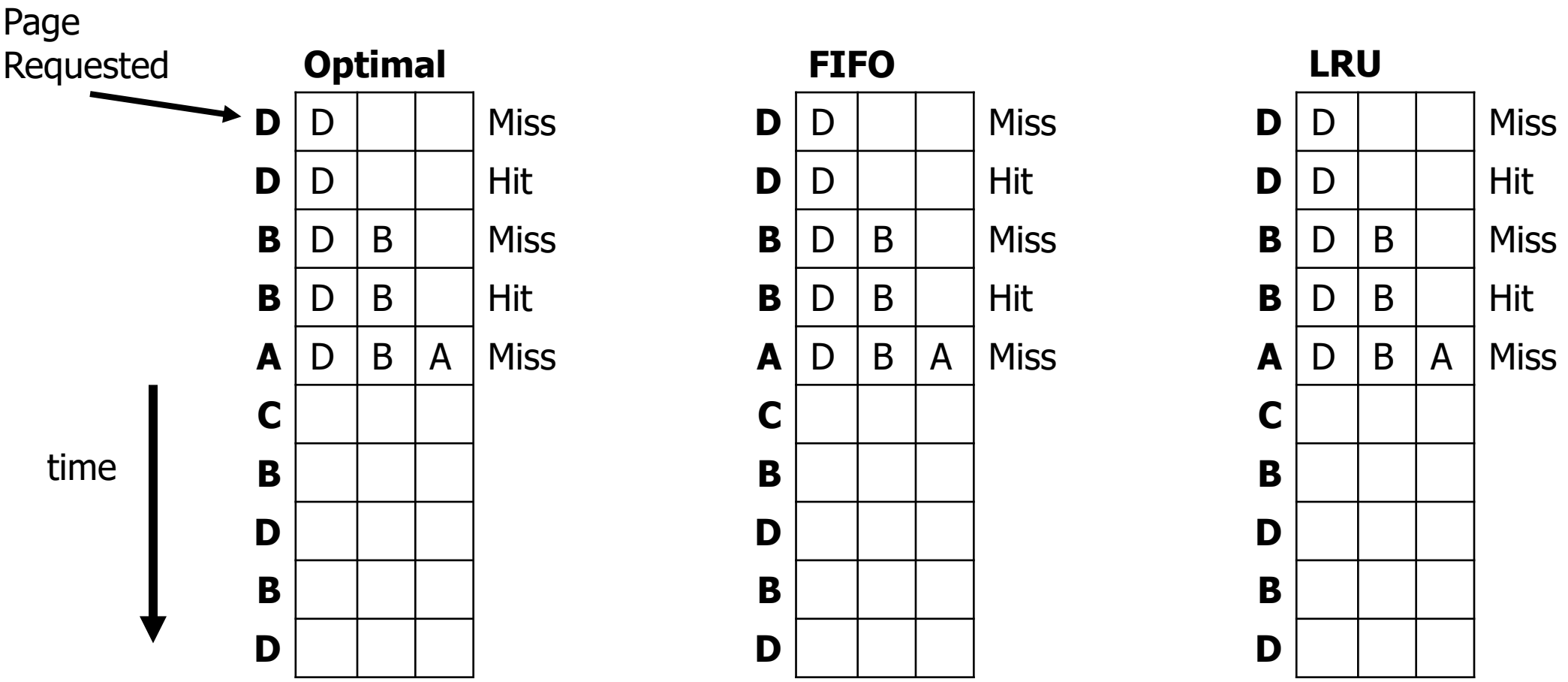
Check your understanding – simple replacement policies



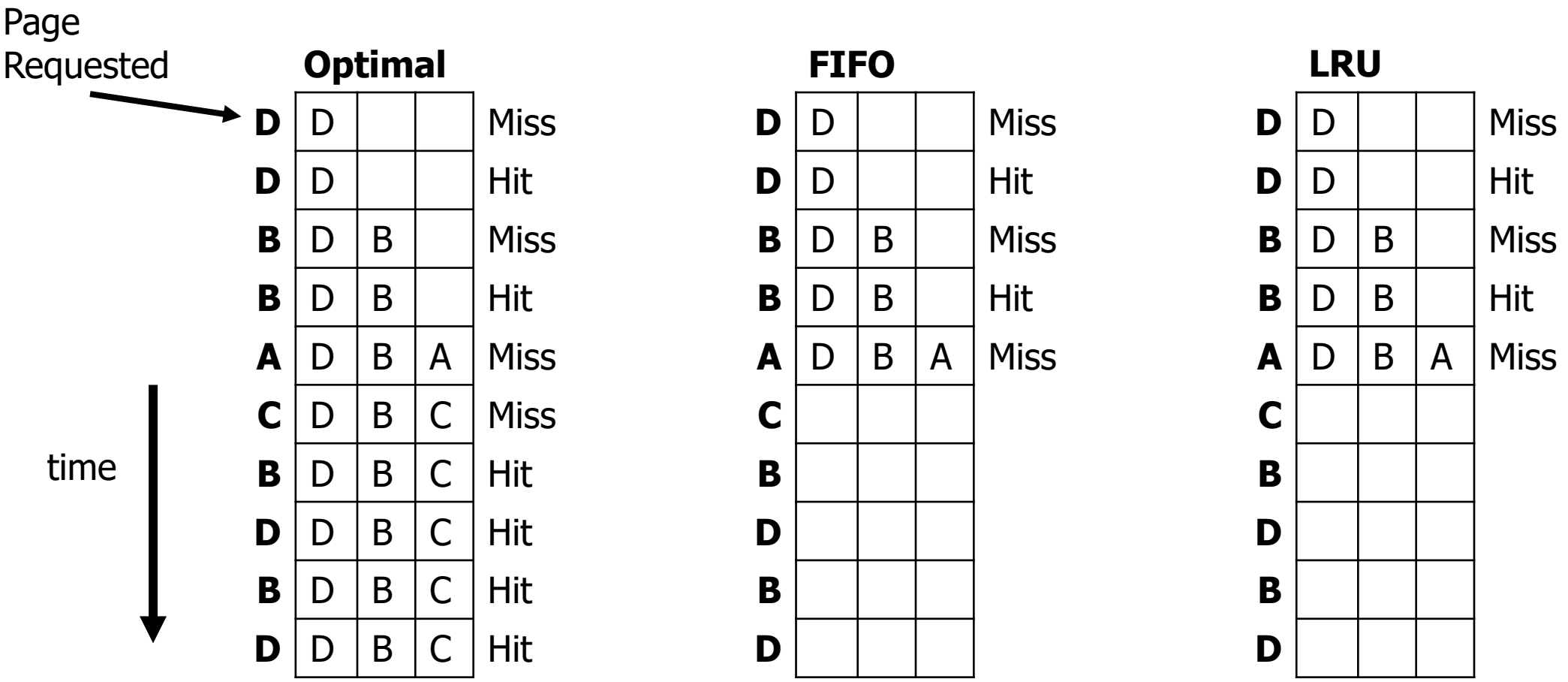
Check your understanding – simple replacement policies



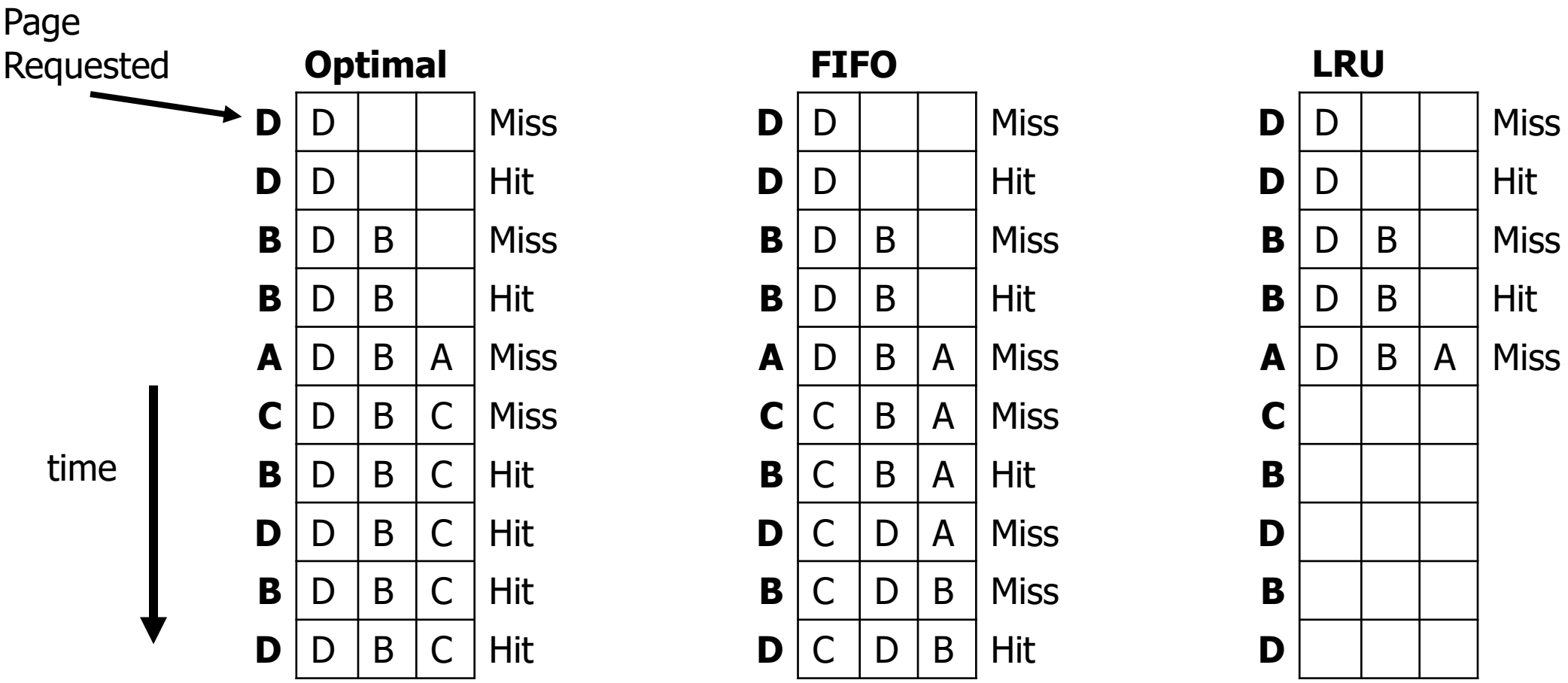
Check your understanding – simple replacement policies



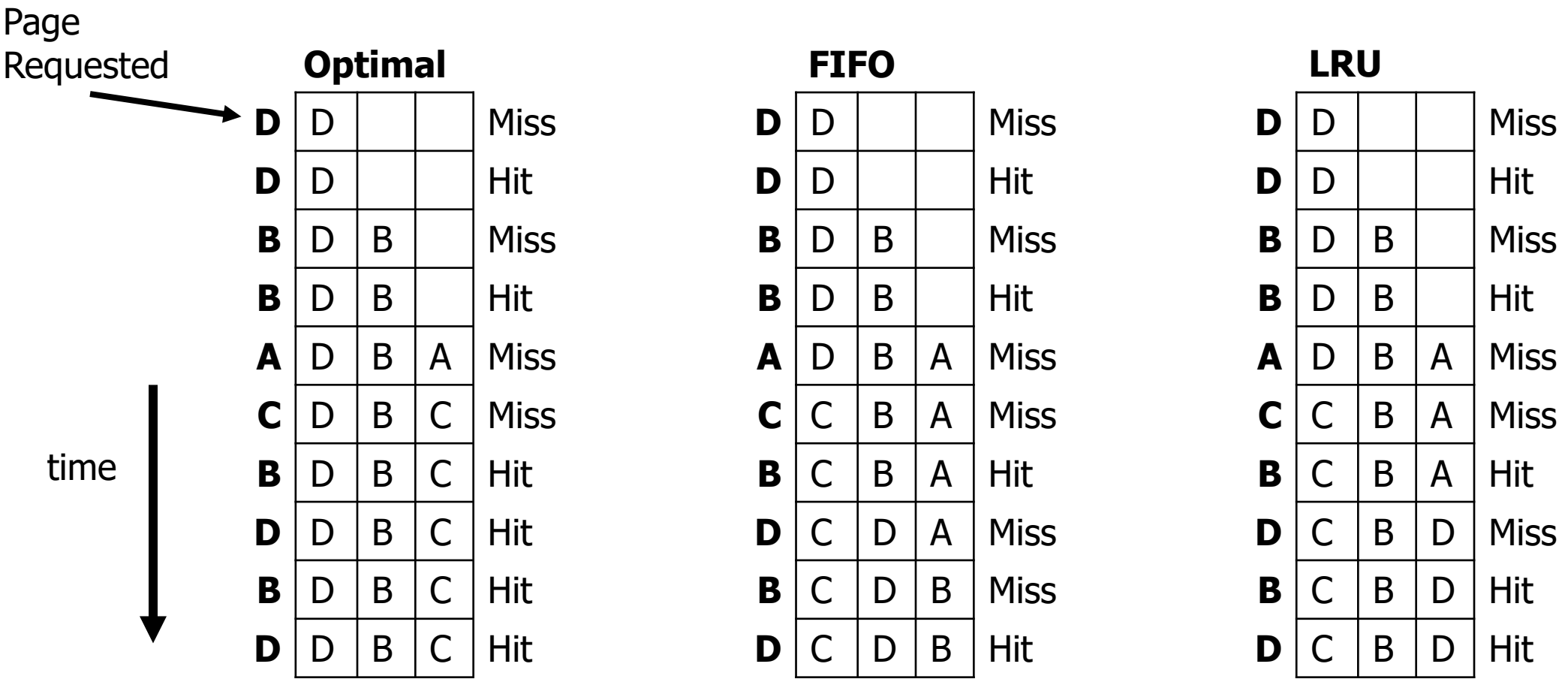
Check your understanding – simple replacement policies



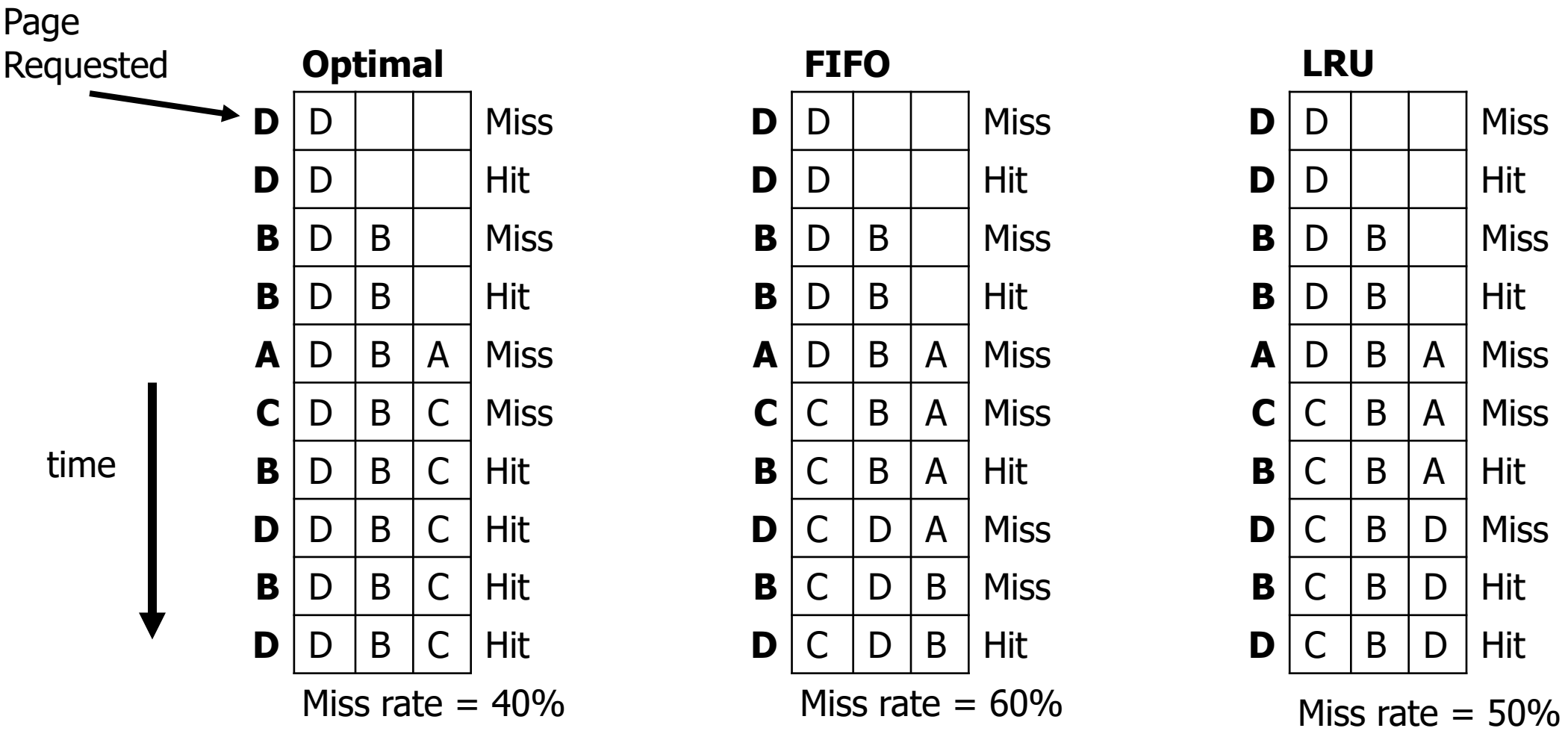
Check your understanding – simple replacement policies



Check your understanding – simple replacement policies



Check your understanding – simple replacement policies



Implementing LRU

- Implementing *perfect* LRU is difficult in practice

- Software perfect LRU
  - OS maintains an ordered list of physical pages by reference time
    - When page is referenced: move to end of list
    - When swap is needed: evict front of list
  - Tradeoff: slow on memory reference, fast on replacement (unacceptable)

- Hardware perfect LRU
  - Associate a timestamp with each physical page
    - When page is referenced: hardware updates timestamp for page
    - When swap is needed: OS searches through all pages for oldest
  - Tradeoff: fast on memory reference, extremely slow on replacement and needs special hardware

Clock algorithm

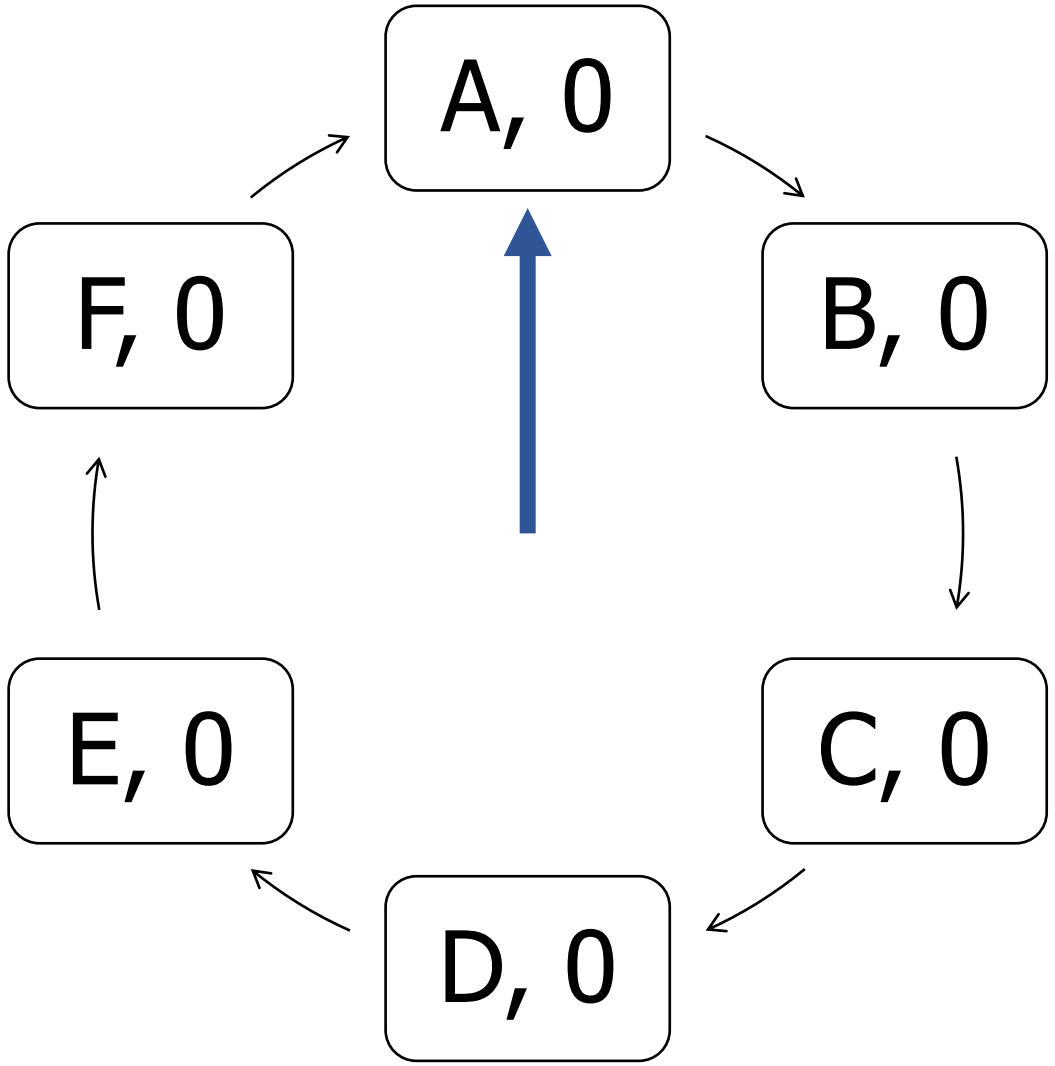
- LRU approximates Optimal anyways, so approximate a little more
  - Goal: find *an* old page, not necessarily the *oldest* page

- Clock algorithm
  - One "accessed" bit added to each page
    - When page is referenced: accessed bit is set to one (hardware)
    - When swap is needed:
      - Cycle through pages looking for one with accessed bit zero
      - Update accessed bit to zero after checking a page
      - Continue from where you left off when next swap is needed
  - Essentially looks for page that hasn't been referenced this "cycle"

Clock algorithm example

(Page Name, Accessed Bit)

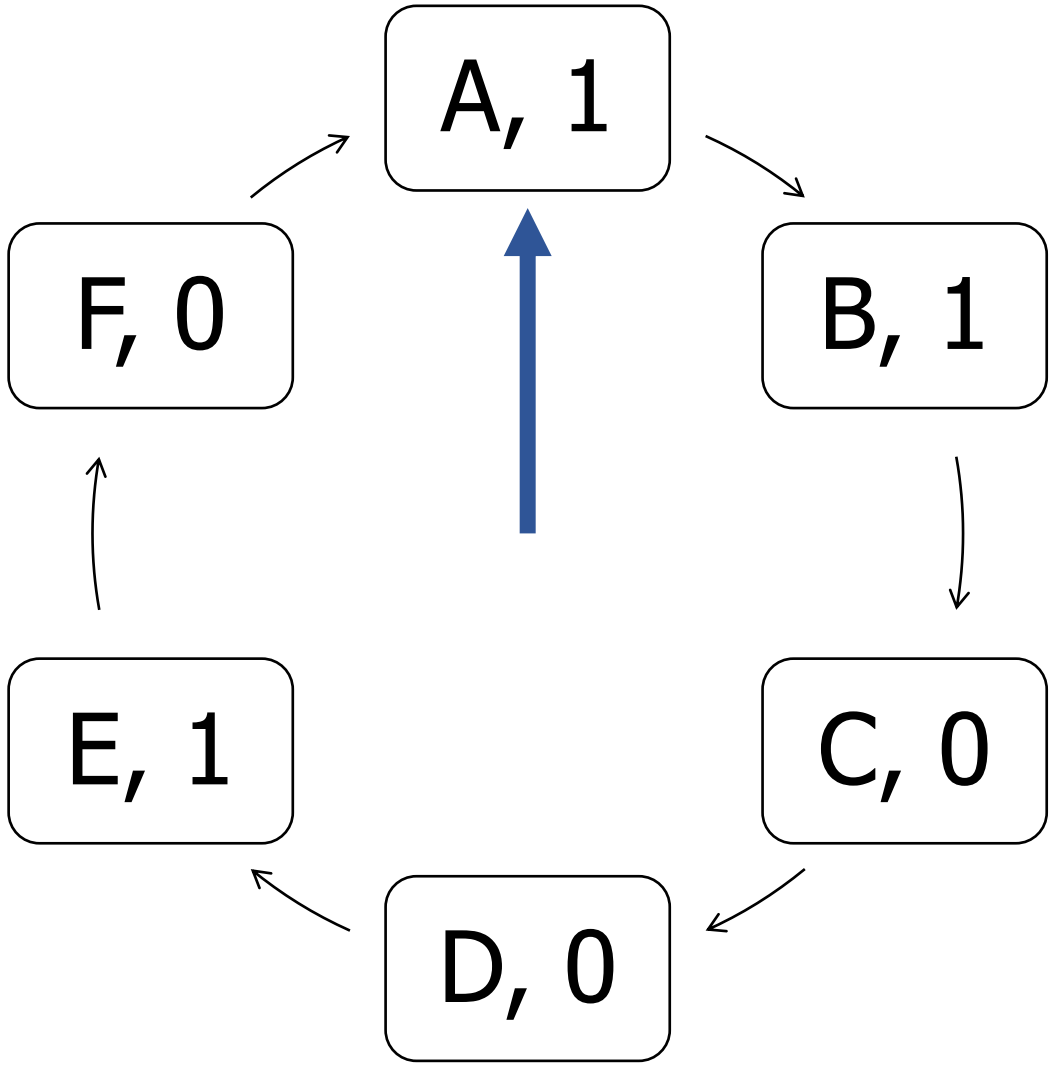
- Initial setup
  - 6 pages total fit in memory
  - Accessed starts as zero
  - “clock hand” points at first page



Clock algorithm example

- After running a little while
  - Pages A, B, E are accessed

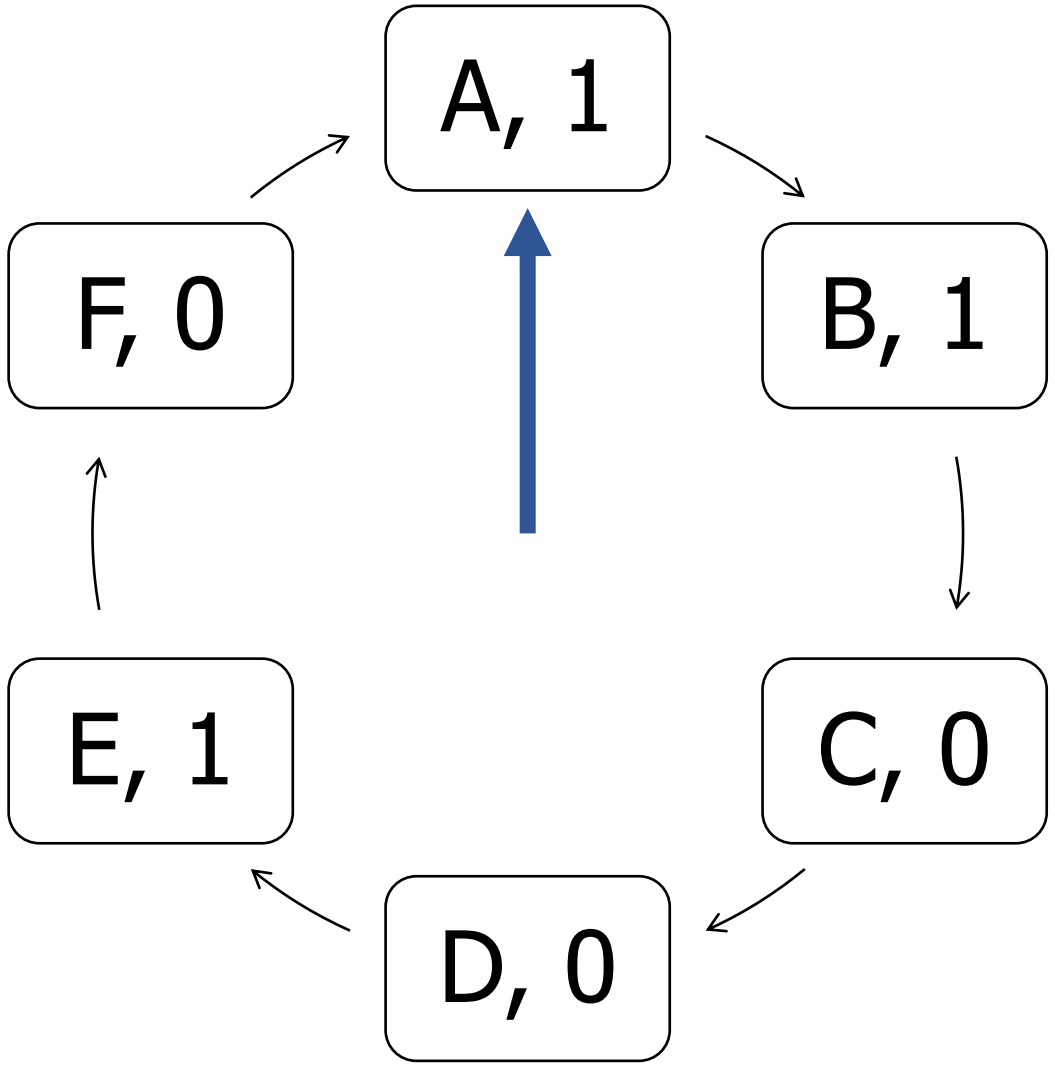
(Page Name, Accessed Bit)



Clock algorithm example

- OS needs to swap pages
  - Algorithm starts
- A is recently accessed

(Page Name, Accessed Bit)

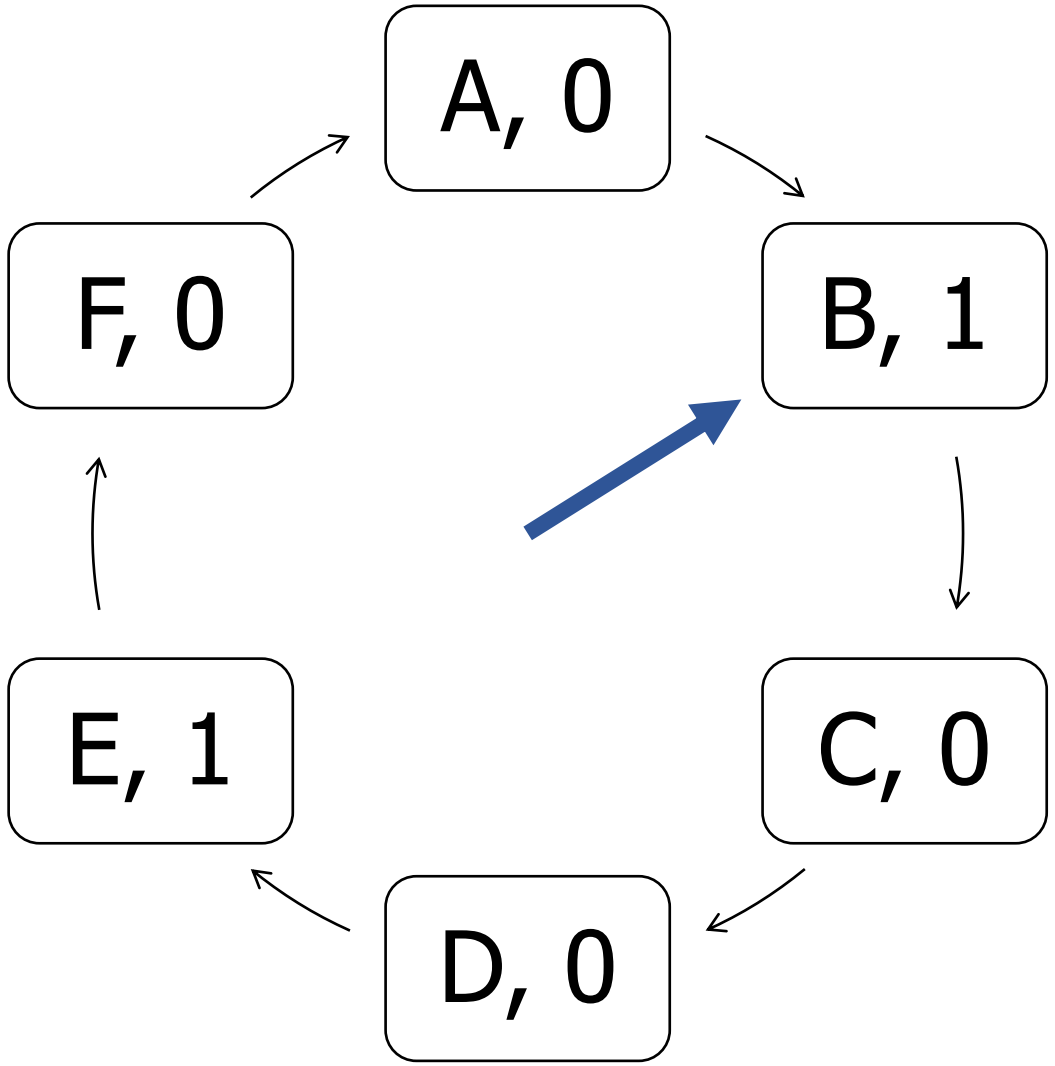


Clock algorithm example

- OS needs to swap pages
  - Algorithm starts

- A is recently accessed
- B is recently accessed

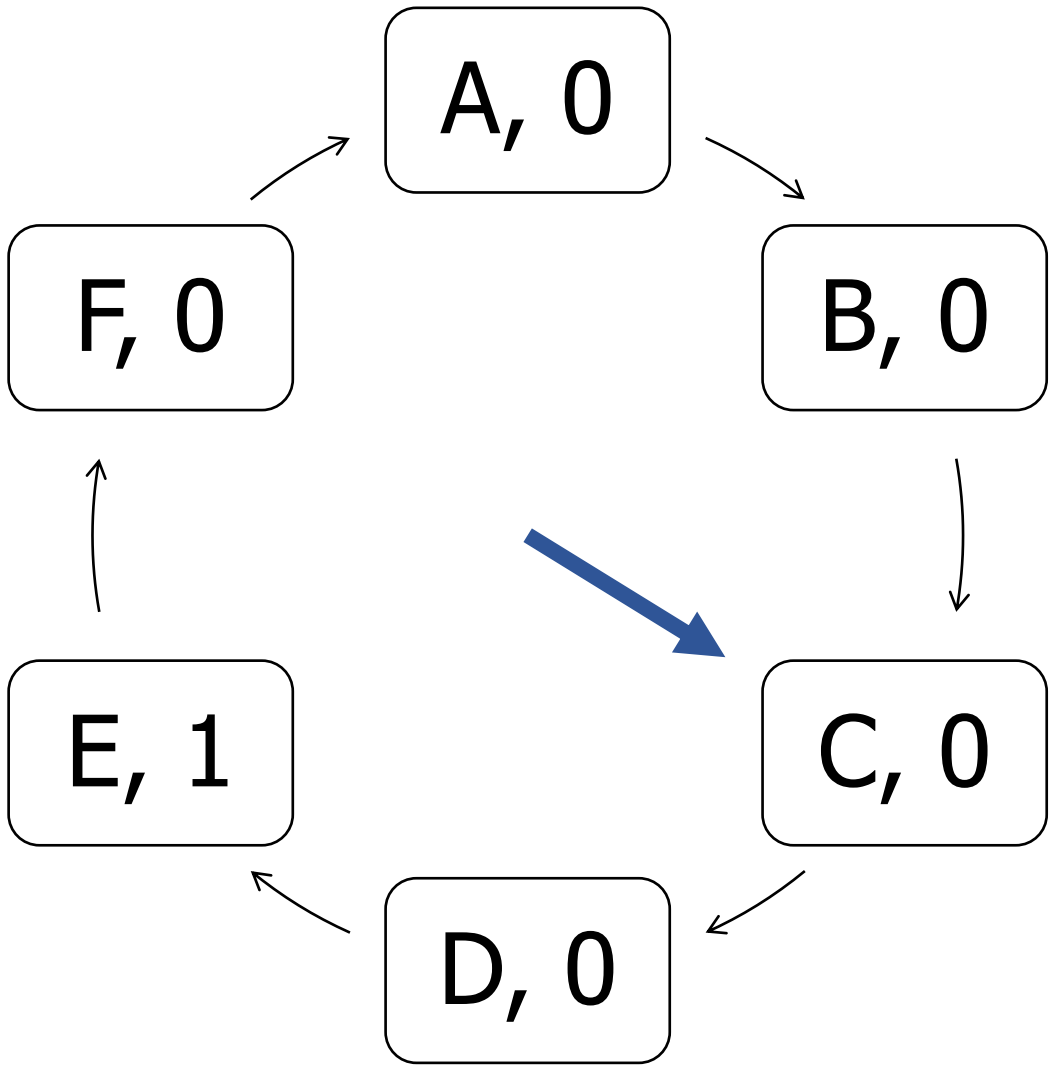
(Page Name, Accessed Bit)



Clock algorithm example

- OS needs to swap pages
  - Algorithm starts

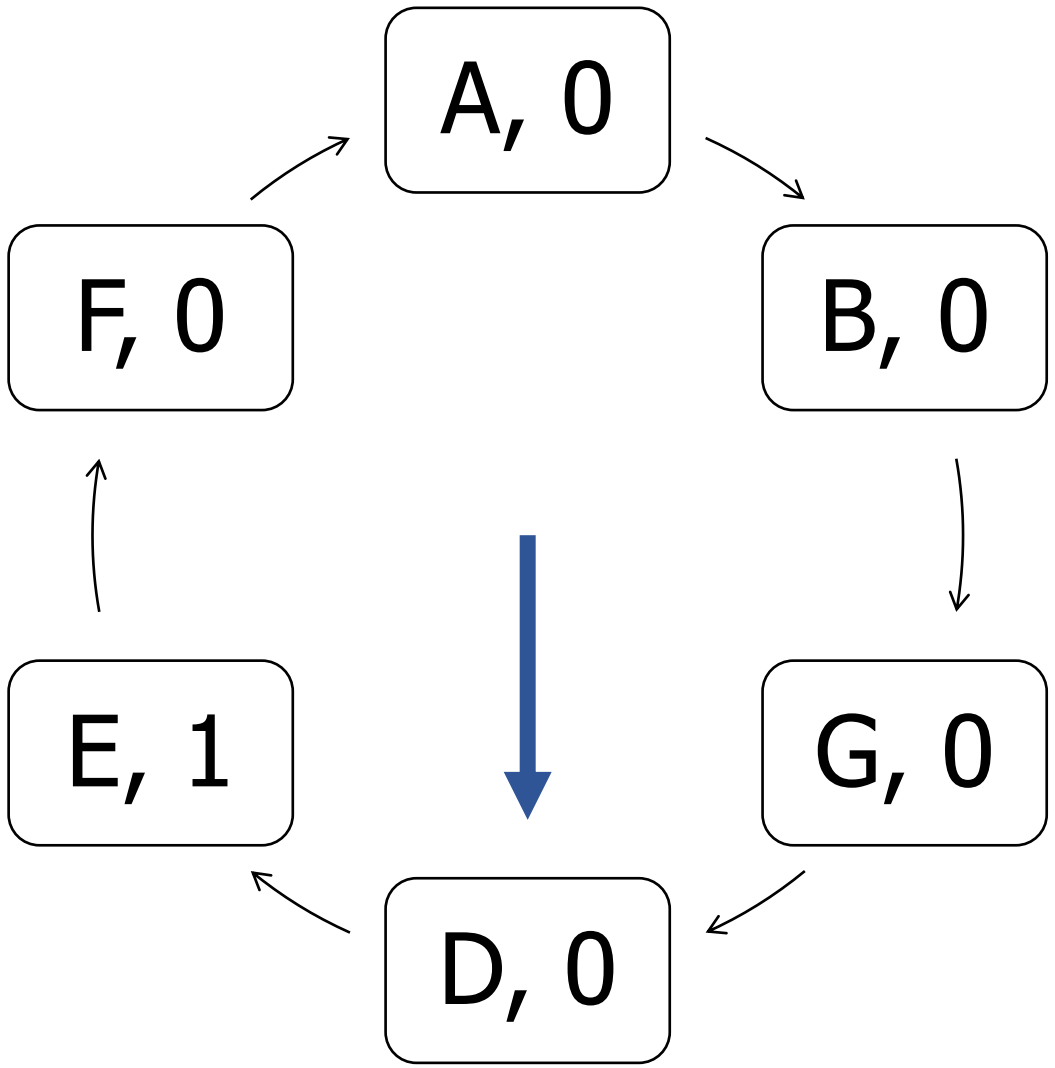
- A is recently accessed
- B is recently accessed
- C has not been recently accessed



Clock algorithm example

- OS needs to swap pages
  - Algorithm starts

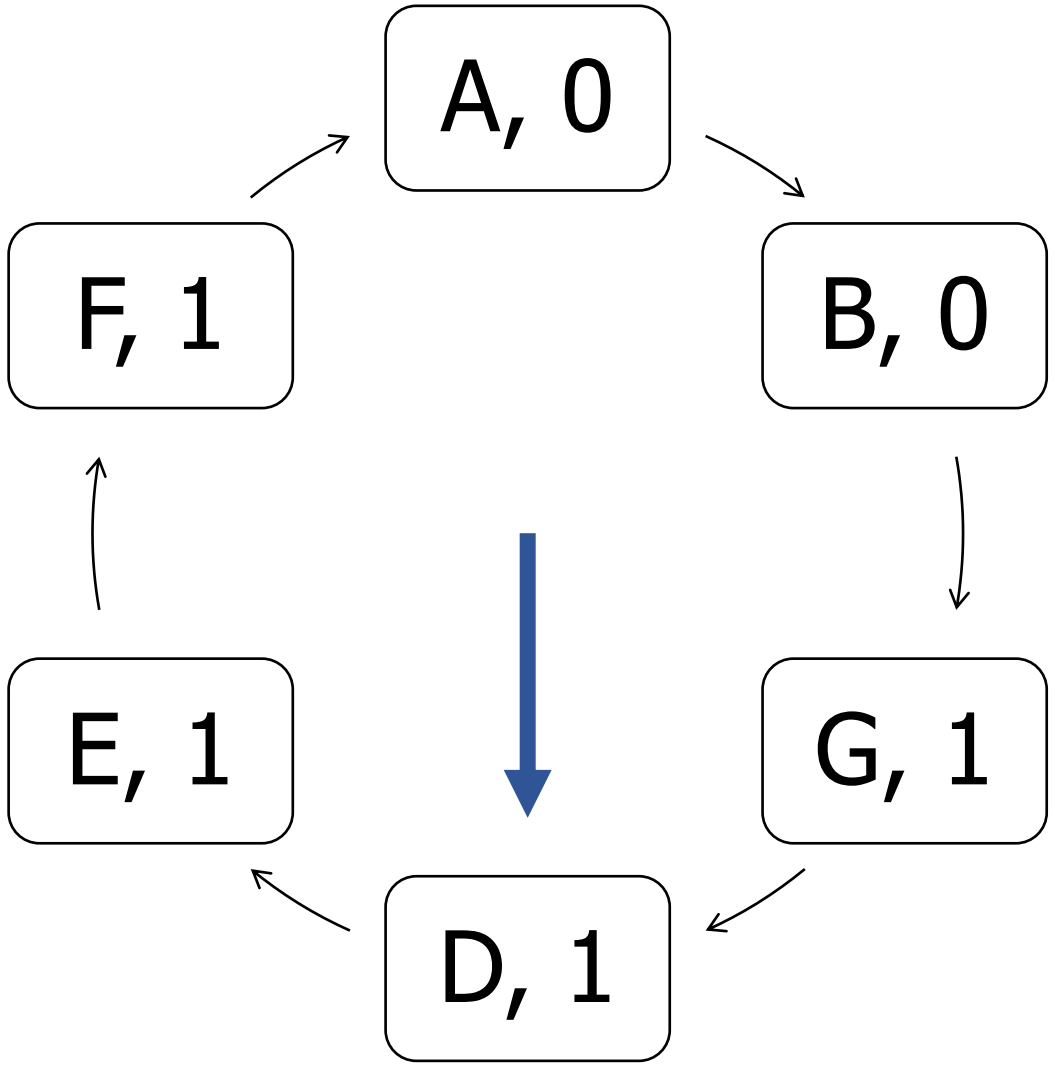
- A is recently accessed
- B is recently accessed
- C has not been recently accessed
  - So swap it
  - And advance hand once more



Clock algorithm example

- Programs continue running for a while
  - Pages G, D, F are accessed

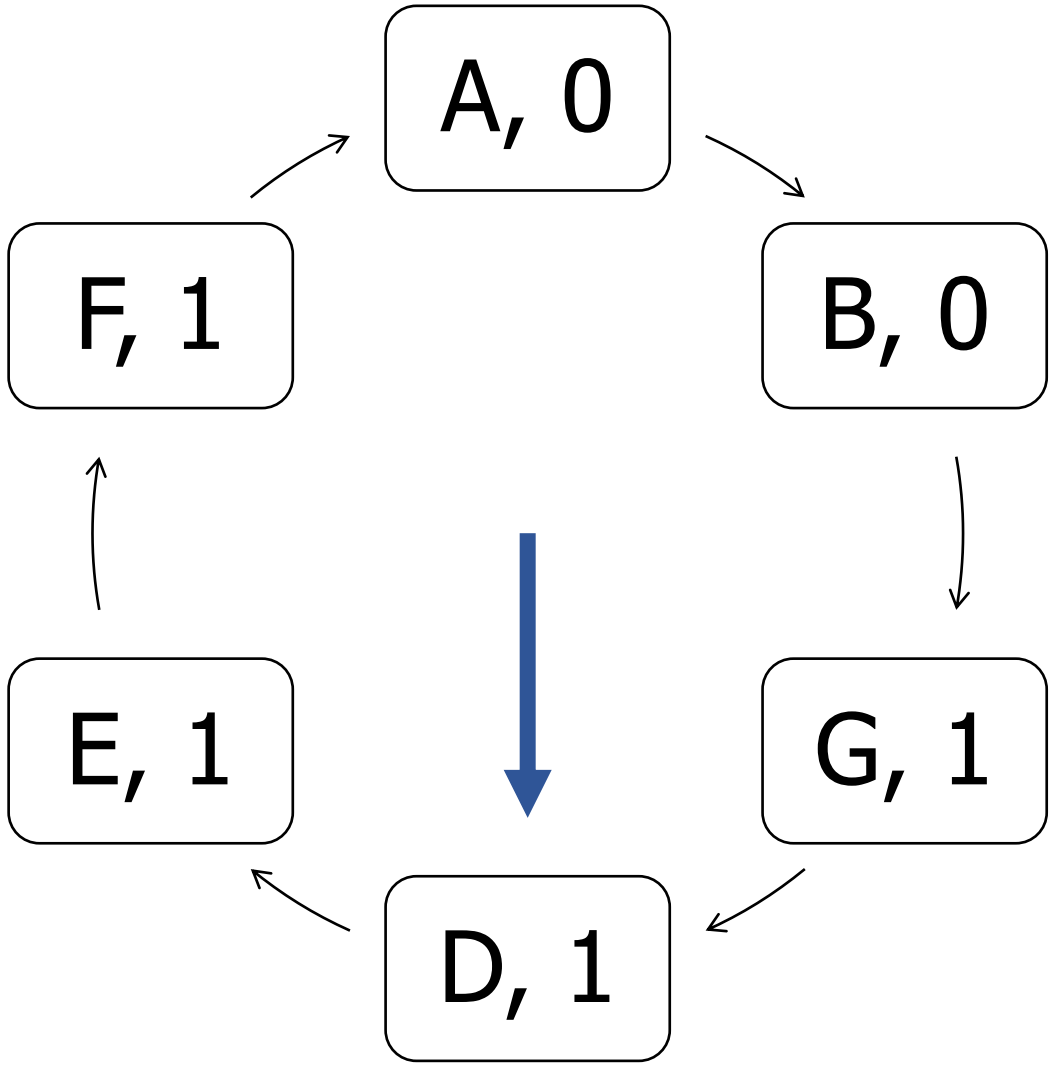
(Page Name, Accessed Bit)



Clock algorithm example

- OS needs to swap again
  - Algorithm begins again
  - But with hand starting somewhere new

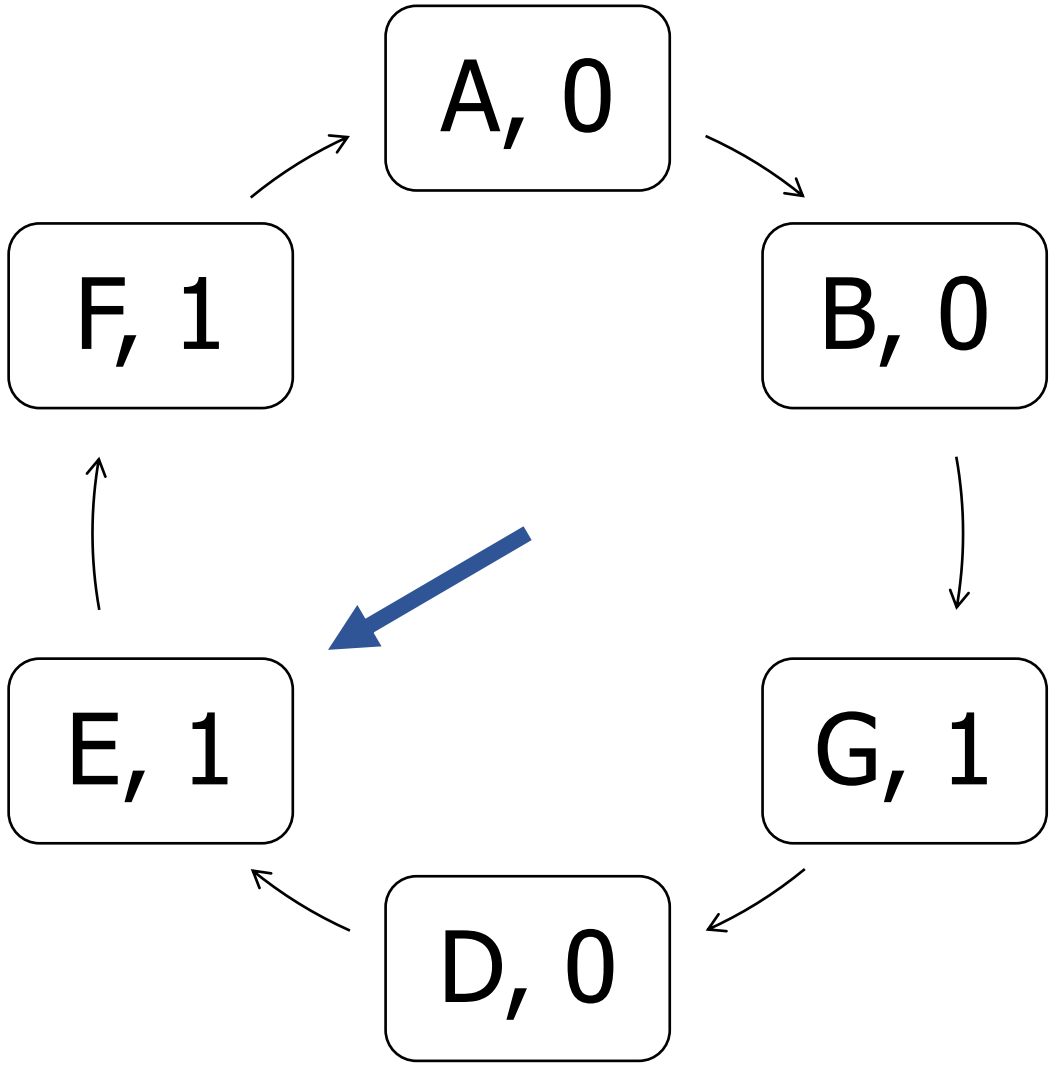
(Page Name, Accessed Bit)



Clock algorithm example

- OS needs to swap again
  - Algorithm begins again
  - But with hand starting somewhere new
- D recently accessed

(Page Name, Accessed Bit)

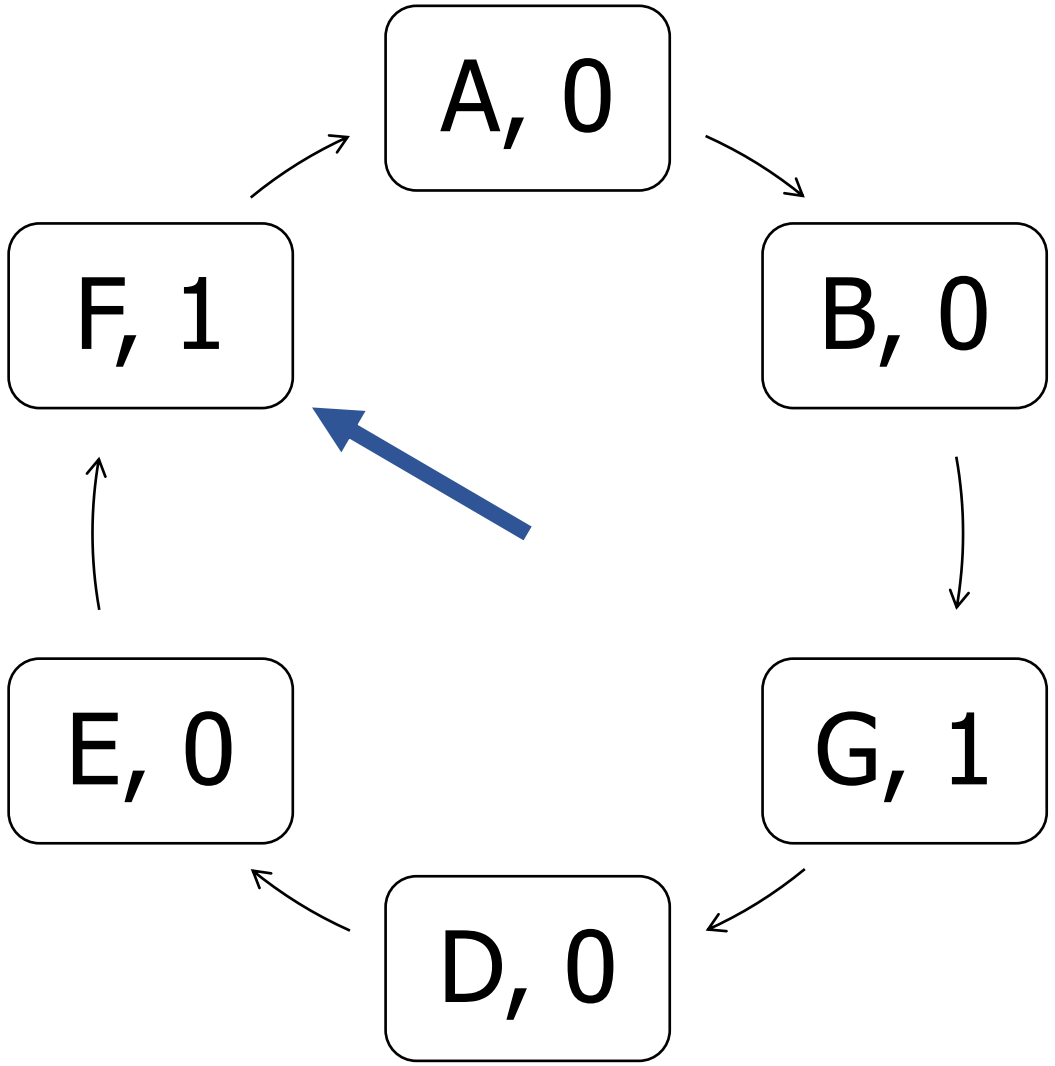


Clock algorithm example

- OS needs to swap again
  - Algorithm begins again
  - But with hand starting somewhere new

- D recently accessed
- E recently accessed

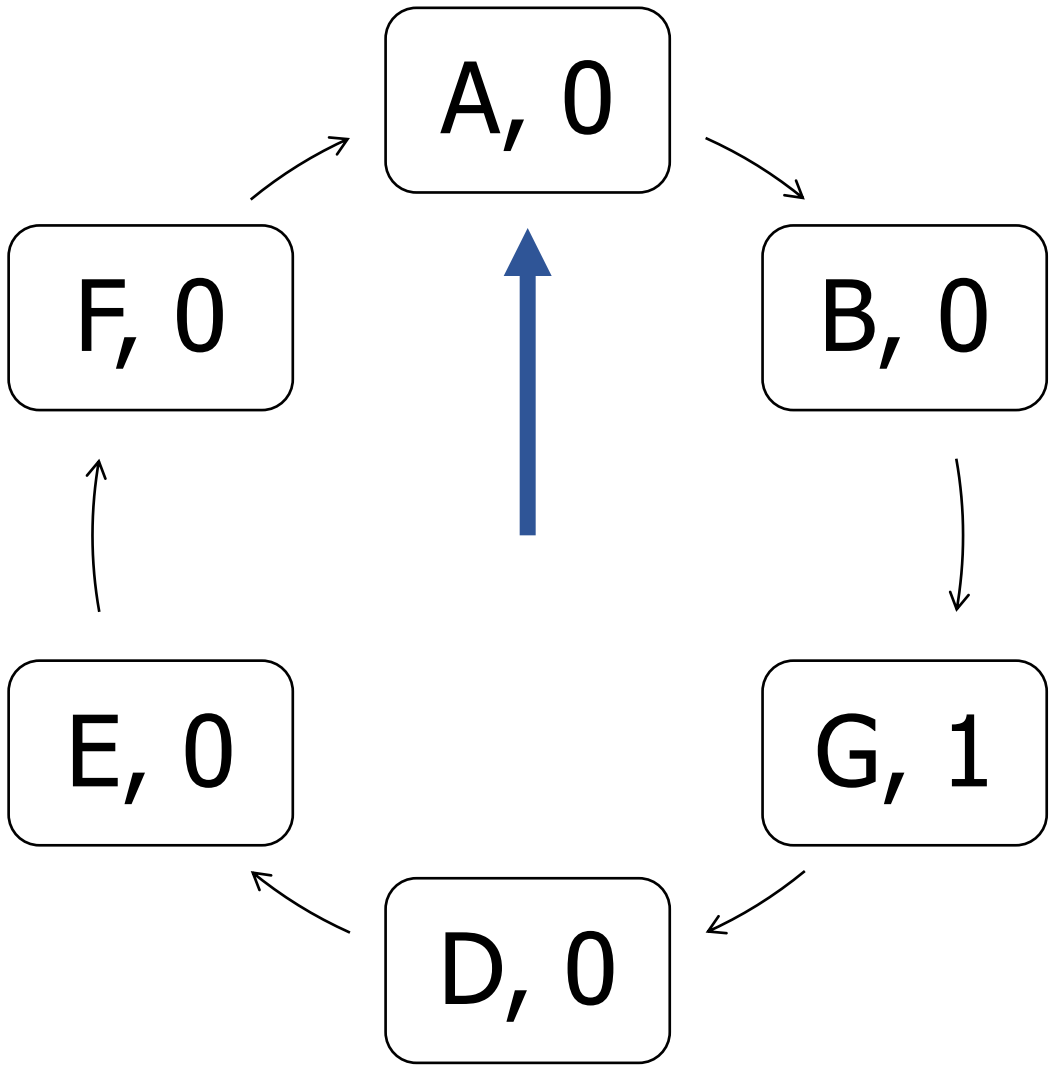
(Page Name, Accessed Bit)



Clock algorithm example

- OS needs to swap again
  - Algorithm begins again
  - But with hand starting somewhere new

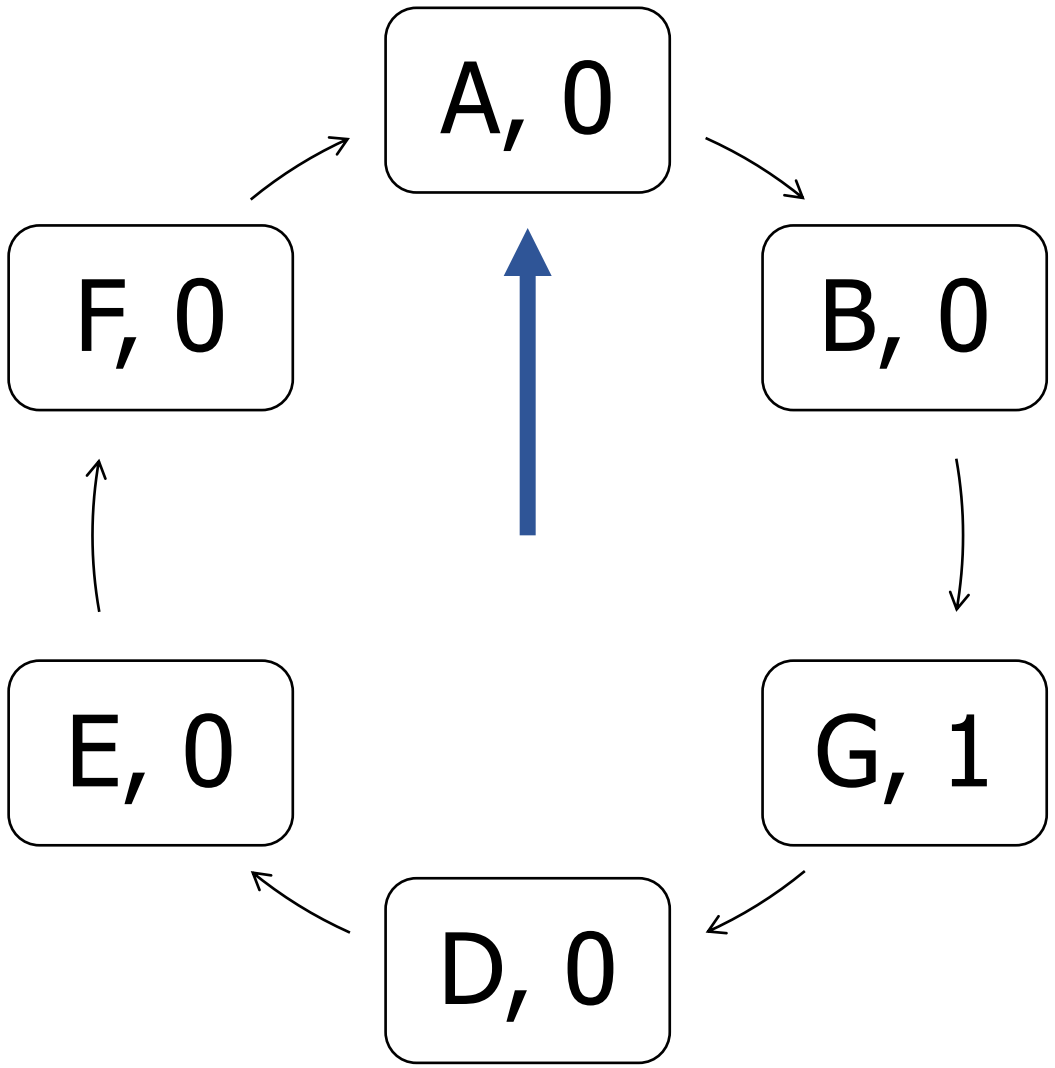
- D recently accessed
- E recently accessed
- F recently accessed



Clock algorithm example

- OS needs to swap again
  - Algorithm begins again
  - But with hand starting somewhere new

- D recently accessed
- E recently accessed
- F recently accessed
- A gets swapped!

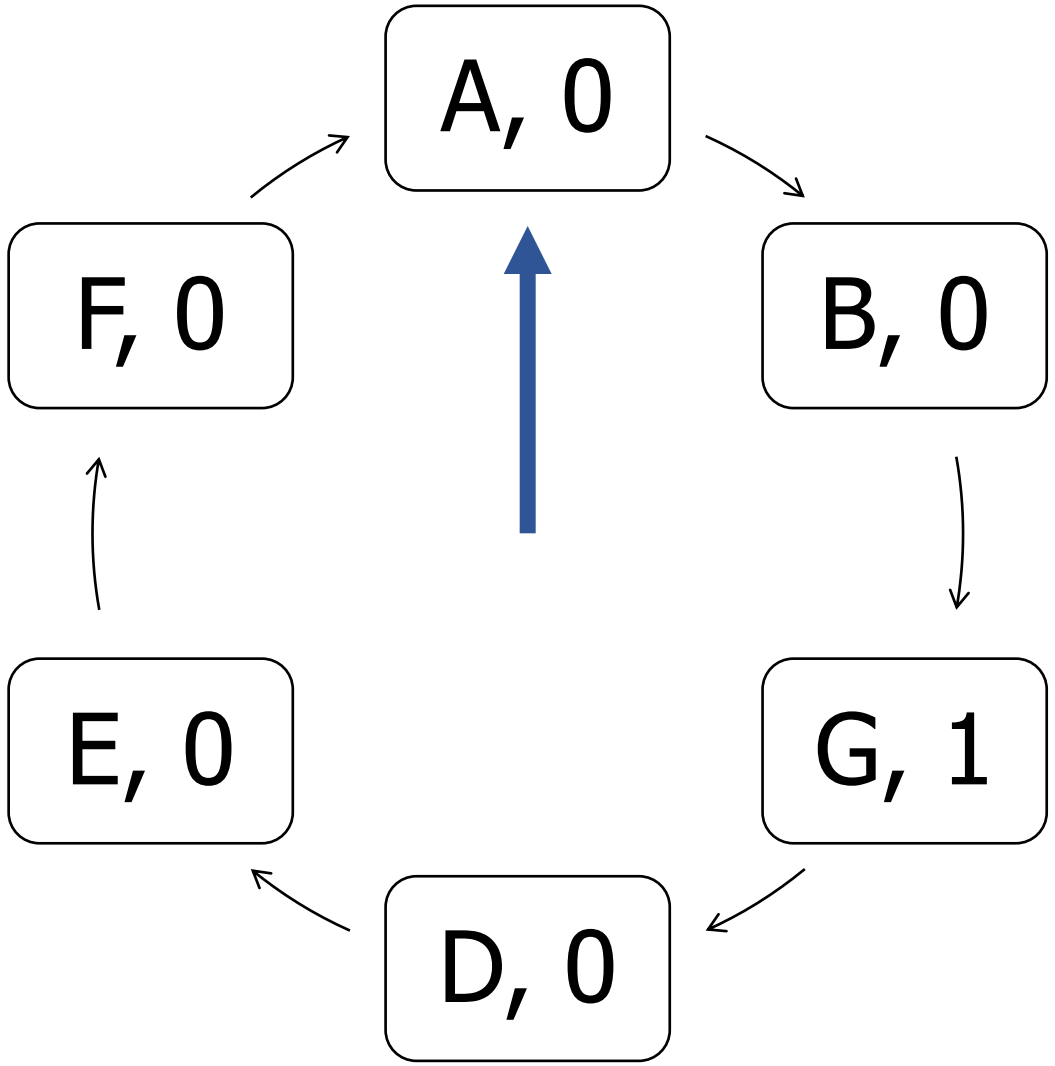


Clock algorithm example

- OS needs to swap again
  - Algorithm begins again
  - But with hand starting somewhere new

- D recently accessed
- E recently accessed
- F recently accessed
- A gets swapped
  - Was A or B the actual LRU?
  - Probably doesn't matter

(Page Name, Accessed Bit)



Clock algorithm is actually used in real computers

- Modern OSes use some variation on Clock Algorithm

- x86 hardware supports an accessed bit in page table entries

- Clock algorithm can be built without hardware support
  - Mark all pages as valid but not present initially (soft page fault)
  - On OS fault, update accessed bit for page, mark as present
    - Only fault on *first* access per clock-hand-cycle
  - Reset page to not present whenever accessed is reset to zero

Improving clock algorithm access notion

- Add multiple "accessed" bits to create accessed counter
  - Increment or decrement bits on use or clock-hand-pass respectively
  - Only remove pages with 0 accessed (or less than some minimum)

- Combine with timestamp notion to ensure page is "old" (WSClock)
  - Keep a timestamp in addition to accessed bit
  - Only remove pages with 0 accessed and older than some amount
    - Still not necessarily oldest, but definitely old

Improving clock algorithm evictions

- Keep track of number of times a page re-enters memory (Clock-PRO)
  - Give eviction preference to pages that haven't been brought back a bunch
  - Bringing it back implies it was important, even if it was old

- Keep track of which pages are dirty
  - Give eviction preference to clean pages (also to read-only pages)
  - Means no write to disk is necessary!

- Evict several pages at once each time it is required
  - Find first N with accessed bit of zero
  - Takes advantage of disk I/O properties

Outline

- Paging in modern OS
- Memory Hierarchy
- Swapping
  - Page Replacement Policies