Paging

Operating System Design – MOSIG 1

Instructor: Arnaud Legrand
Class Assistants: Benjamin Negrevergne, Sascha Hunold

September 27, 2010

Flashback

Remember the last lecture

- **Virtual Memory is required to enforce:**
  - Protection/Isolation: a process should only mess with its own memory
  - Transparency: memory references and size need to be dynamically adjusted ; give each process its own adress space
  - Resource exhaustion management: handle (efficiently) situation where there is not enough memory to fit all process
- **The MMU is here to help us!**
  - Hardware support for adress translation
- **Segmentation is a first approach that suffers from a terrible drawback: Fragmentation**
- **Fragmentation is caused by**
  - size heterogeneity;
  - isolated deaths;
  - time-varying behavior;

Alternative approaches

- **Segmentation**
  - Part of each memory reference implicit in segment register
    segreg $\leftarrow \langle \mathrm{offset}, \mathrm{limit} \rangle$
  - By loading segment register code can be relocated
  - Can enforce protection by restricting segment register loads

Alternative approaches

- **Segmentation**
  - Part of each memory reference implicit in segment register segreg $\leftarrow \langle \text{offset}, \text{limit} \rangle$
  - By loading segment register code can be relocated
  - Can enforce protection by restricting segment register loads
- **Language-level protection (Java)**
  - Single address space for different modules
  - Language enforces isolation
  - Singularity OS does this [Hunt]
- **Software fault isolation**
  - Instrument compiler output
  - Checks before every store operation prevents modules from trashing each other
  - Google Native Client does this with only about 5% slowdown [Yee]

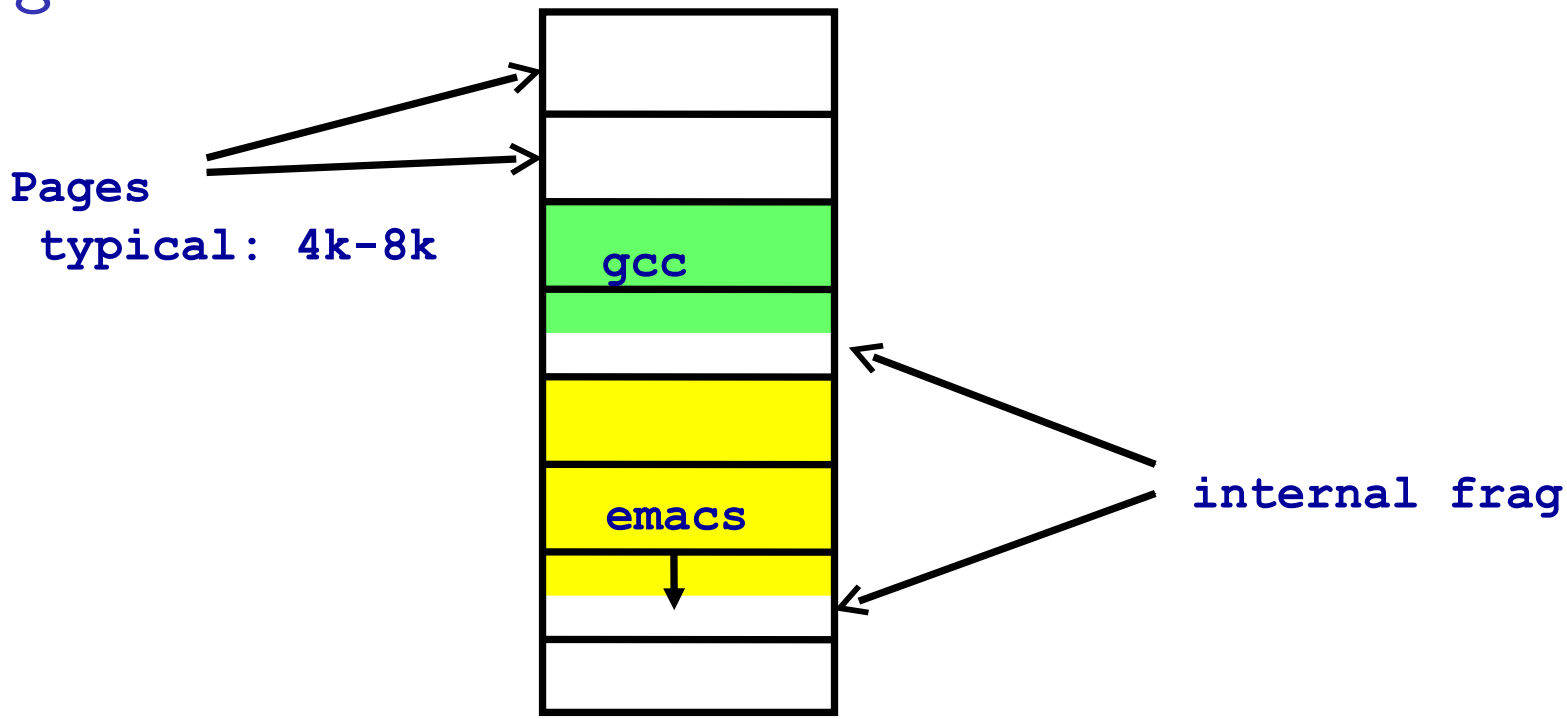
Outline

Outline

Paging

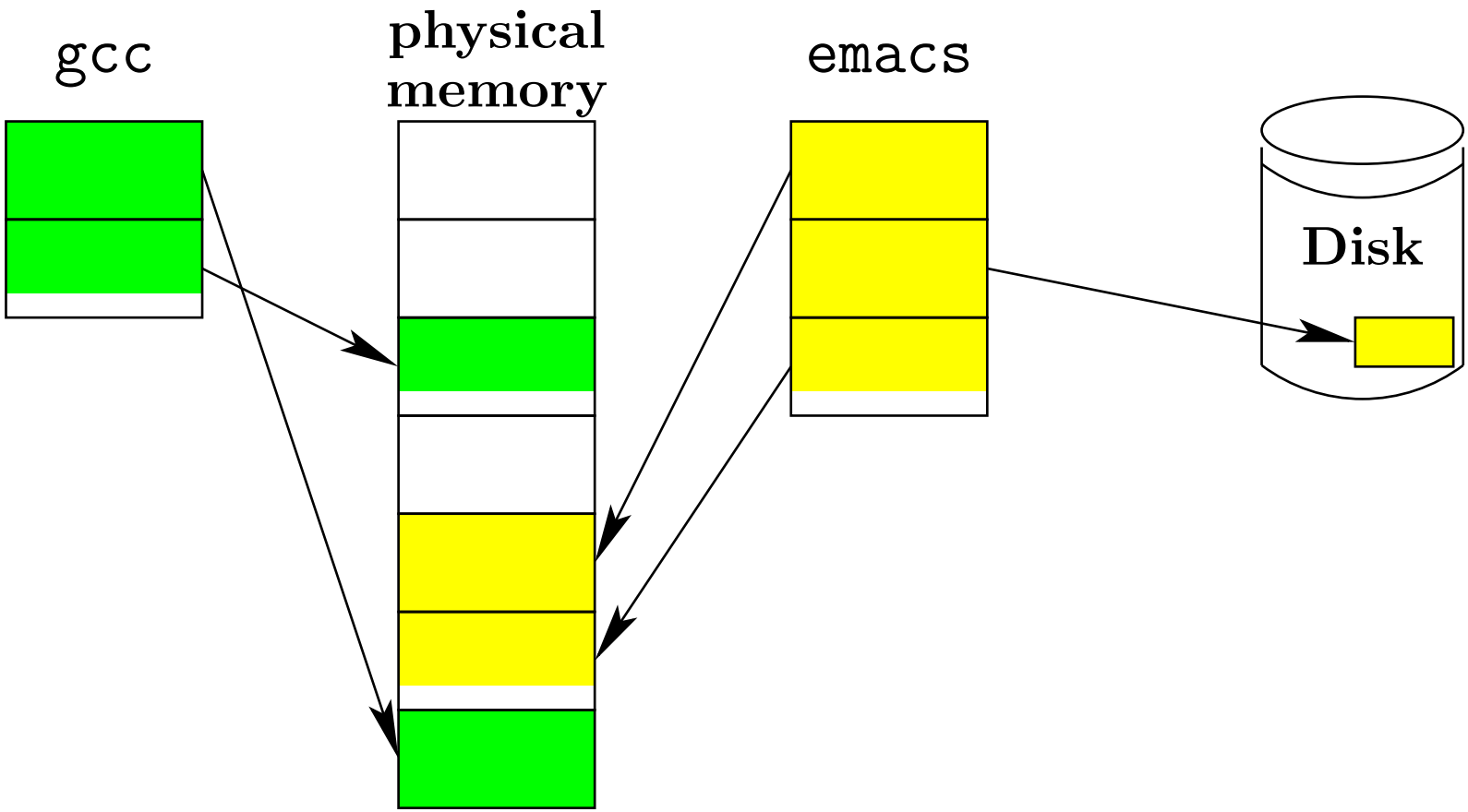
- **Divide memory up into small pages**
- **Map virtual pages to physical pages**
  - Each process has separate mapping
- **Allow OS to gain control on certain operations**
  - Read-only pages trap to OS on write
  - Invalid pages trap to OS on read or write
  - OS can change mapping and resume application
- **Other features sometimes found:**
  - Hardware can set “accessed” and “dirty” bits
  - Control page execute permission separately from read/write
  - Control caching of page

Paging trade-offs



- **Eliminates external fragmentation**
- **Simplifies allocation, free, and backing storage (swap)**
- **May leverage internal fragmentation**
- **Average internal fragmentation of .5 pages per "segment"**

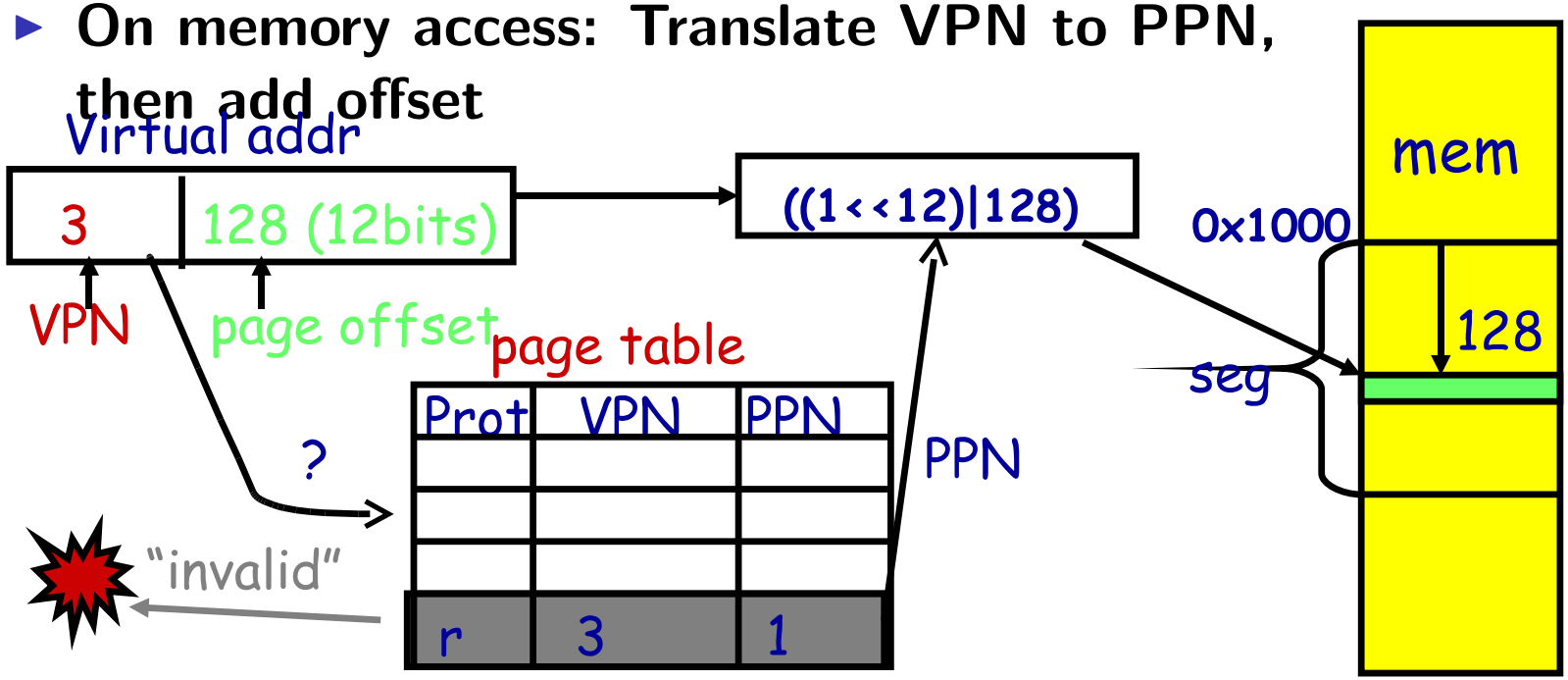
Simplified allocation



- **Allocate any physical page to any process**
- **Can store idle virtual pages on disk**

Paging data structures

- **Pages are fixed size, e.g., 4K**
  - Least significant 12 (log 4K) bits of address are *page offset*
  - Most significant bits are *page number*
- **Each process has a page table**
  - Maps *virtual page numbers* to *physical page numbers*
  - Also includes bits for protection, validity, etc.
- **On memory access: Translate VPN to PPN, then add offset**



Example: Paging on PDP-11

- **64K virtual memory, 8K pages**
  - Separate address space for instructions & data
  - I.e., can't read your own instructions with a load
- **Entire page table stored in registers**
  - 8 Instruction page translation registers
  - 8 Data page translations
- **Swap 16 machine registers on each context switch**

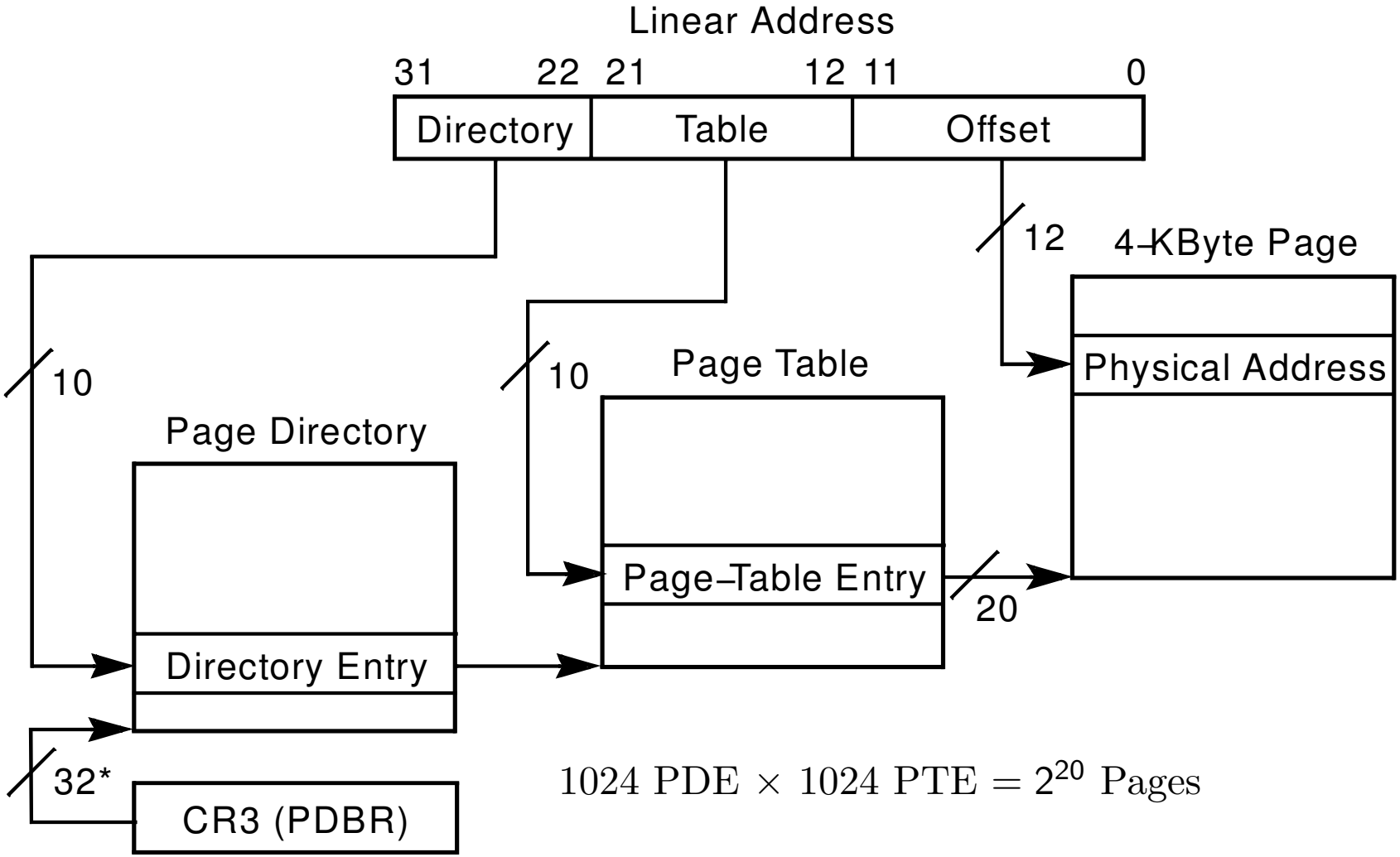
x86 Paging

- **Paging enabled by bits in a control register (`%cr0`)**
  - Only privileged OS code can manipulate control registers
- **Normally 4KB pages**
  - x86 use 32-bits words $\rightsquigarrow$ 4GB of adressable memory
  - offset=12bits /page index=20 bits $\rightsquigarrow$ flat page table = 1MB! ☹

x86 Paging

- **Paging enabled by bits in a control register (`%cr0`)**
  - Only privileged OS code can manipulate control registers
- **Normally 4KB pages**
  - x86 use 32-bits words $\rightsquigarrow$ 4GB of adressable memory
  - offset=12bits /page index=20 bits $\rightsquigarrow$ flat page table = 1MB! ☹
- `%cr3`: **points to 4KB page directory** (1 directory per process)
- **Page directory: 1024 PDEs (page directory entries)**
  - Each contains physical address of a page table
  - table index=10bits
- **Page table: 1024 PTEs (page table entries)**
  - Each contains physical address of virtual 4K page
  - Page table covers 4 MB of Virtual mem
  - page index=10bits
- **See old intel manual for simplest explanation**
  - Also volume 2 of AMD64 Architecture docs
  - Also volume 3A of latest Pentium Manual

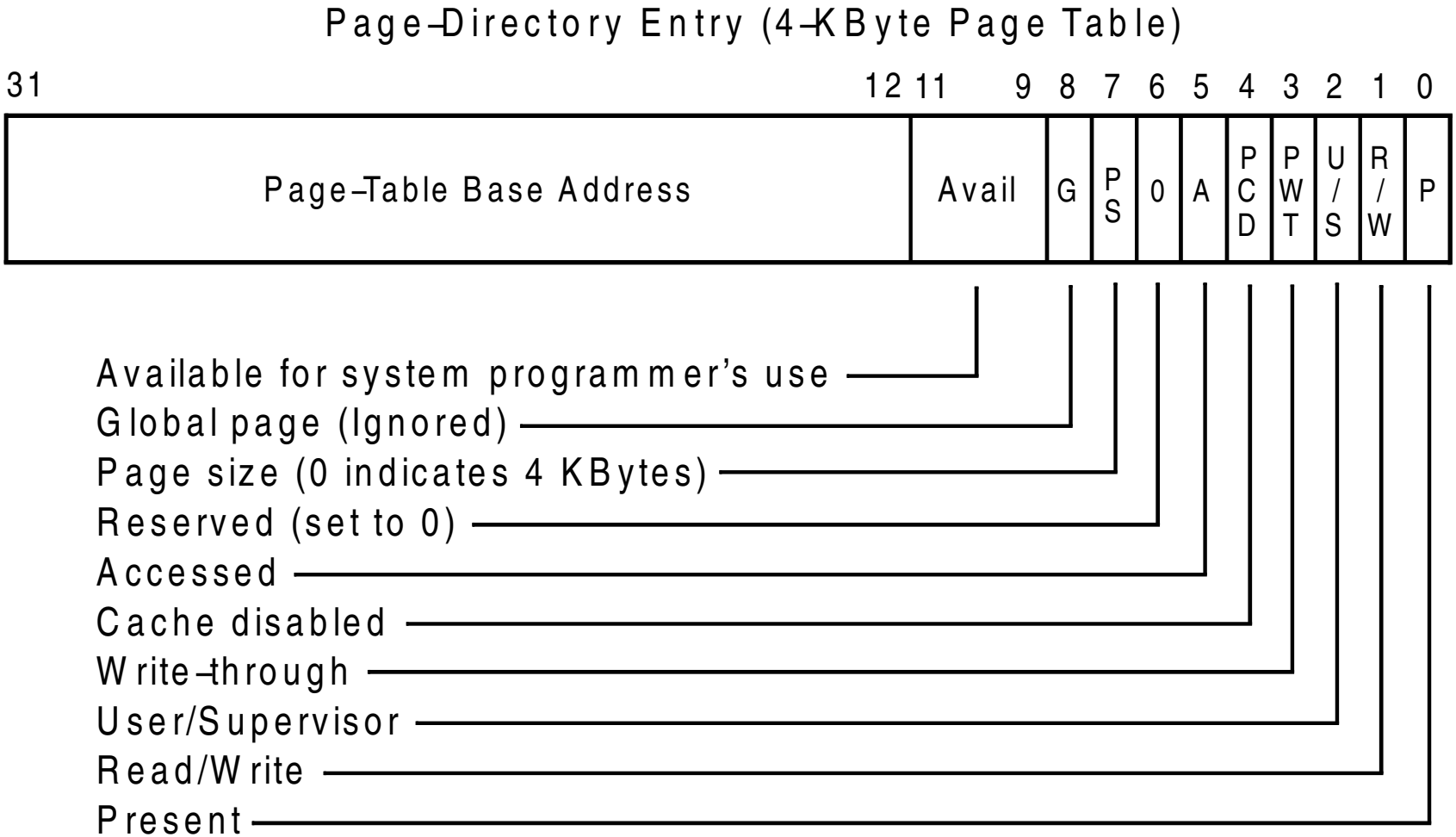
x86 page translation



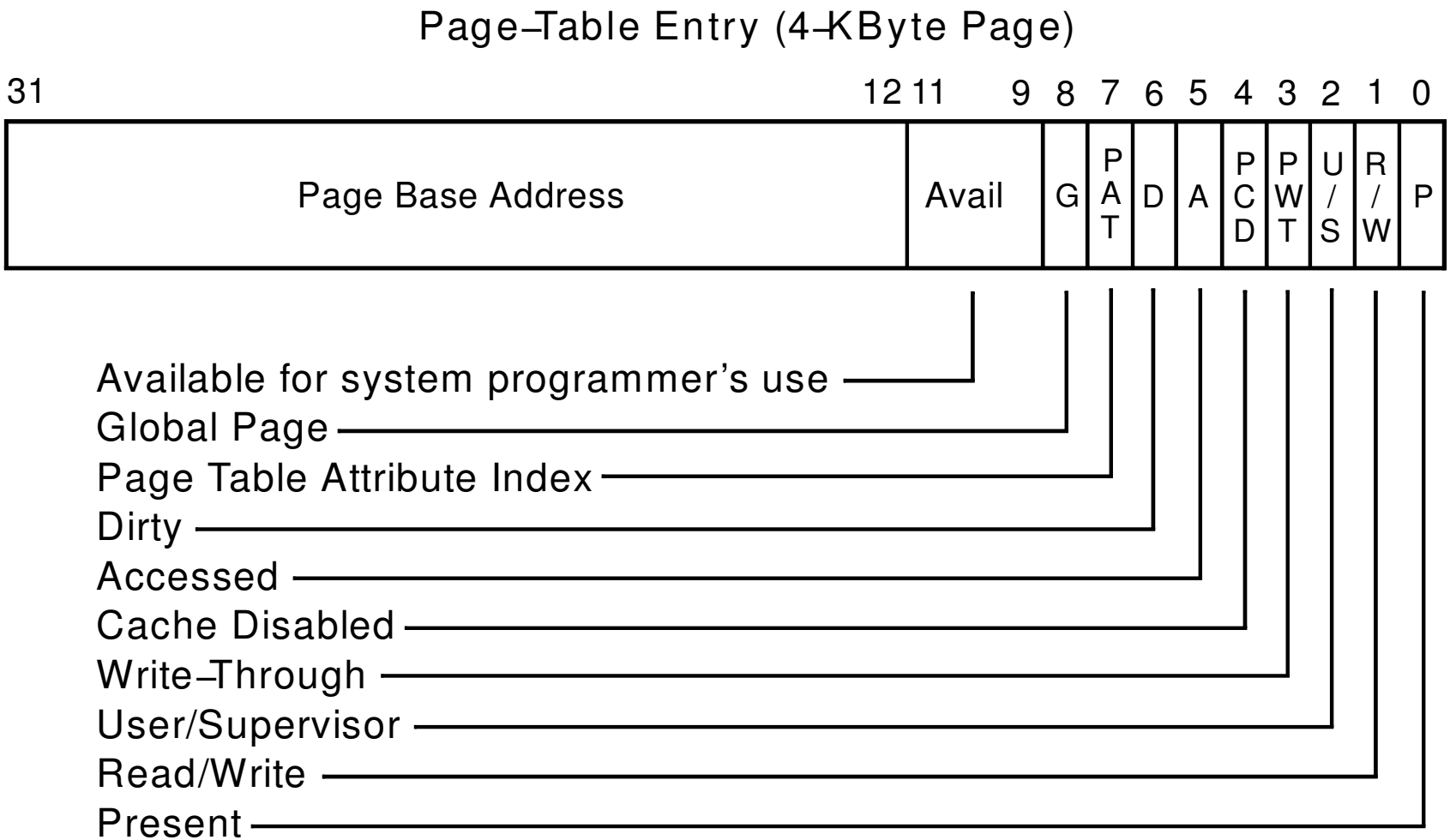
$1024 \text{ PDE} \times 1024 \text{ PTE} = 2^{20} \text{ Pages}$

*32 bits aligned onto a 4–KByte boundary

x86 page directory entry



x86 page table entry



x86 hardware segmentation

- **x86 architecture also supports segmentation**
  - Segment register base + pointer val = *linear address*
  - Page translation happens on linear addresses
- **Two levels of protection and translation check**
  - Segmentation model has four privilege levels (CPL 0–3)
  - Paging only two, so 0–2 = kernel, 3 = user
- **Why do you want both paging and segmentation?**

x86 hardware segmentation

- **x86 architecture also supports segmentation**
  - Segment register base + pointer val = *linear address*
  - Page translation happens on linear addresses
- **Two levels of protection and translation check**
  - Segmentation model has four privilege levels (CPL 0–3)
  - Paging only two, so 0–2 = kernel, 3 = user
- **Why do you want both paging and segmentation?**
- **Short answer: You don't – just adds overhead**
  - Most OSes use "flat mode" – set base = 0, bounds = 0xffffffff in all segment registers, then forget about it
  - x86-64 architecture removes most segmentation support
- **Long answer: Has some fringe/incidental uses**
  - VMware runs guest OS in CPL 1 to trap stack faults
  - OpenBSD used CS limit for W∧X when no PTE NX bit

64-bit address spaces

- **Recall x86-64 only has 48-bit virtual address space**
- **What if you want a 64-bit virtual address space?**
  - Straight hierarchical page tables not efficient
- **Solution 1: Guarded page tables [Liedtke]**
  - Omit intermediary tables with only one entry
  - Add predicate in high level tables, stating the only virtual address range mapped underneath + # bits to skip
- **Solution 2: Hashed page tables**
  - Store Virtual $\rightarrow$ Physical translations in hash table
  - Table size proportional to physical memory
  - Clustering makes this more efficient [Talluri]

Outline

The Memory Bottleneck

The memory is a very common bottleneck that programmers often don't think about

- When you look at code, you often pay more attention to computation
- `a[i] = b[j] + c[k]`
  - The access to the 3 arrays take more time than doing an addition
  - For the code above, the memory is the bottleneck for most machines!
- In the 70's, everything was balanced. The memory kept pace with the CPU (n cycles to execute an instruction, n cycles to bring in a word from memory)
- No longer true
  - CPUs have gotten 1,000x faster
  - Memory have gotten 10x faster and 1,000,000x larger
- Flops are free and bandwidth is expensive and processors are STARVED for data

Memory Latency and Bandwidth

- The performance of memory is typically defined by Latency and Bandwidth (or Rate)
- Latency: time to read one word from memory (measured in nanoseconds these days)
- Bandwidth: how many bytes can be read per seconds (measured in GB/sec)
- Note that you don't have bandwidth = 1 / latency!
- There is pipelining: Reading 2 words in sequence is much cheaper than twice the time reading one word only

Memory	Latency	Peak Bandwidth
DDR400 SDRAM	10 ns	6.4 GB/sec
DDR533 SDRAM	9.4 ns	8.5 GB/sec
DDR2-533 SDRAM	11.2 ns	8.5 GB/sec
DDR2-800 SDRAM	???	12.8 GB/sec
DDR2-667 SDRAM	???	10.6 GB/sec
DDR2-600 SDRAM	13.3 ns	9.6 GB/sec

Memory Bottleneck

Example and crude estimation

- Fragment of code: `a[i] = b[j] + c[k]`
  - Three memory references: 2 reads, 1 write
  - One addition: can be done in one cycle
- If the memory bandwidth is 12.8GB/sec, then the rate at which the processor can access integers (4 bytes) is: $12.8 \times 1024 \times 1024 \times 1024/4 = 3.4GHz$
- The above code needs to access 3 integers
- Therefore, the rate at which the code gets its data is $\simeq 1.1GHz$
- But the CPU could perform additions at $4GHz$!
- Therefore: The memory is the bottleneck
  - And we assumed memory worked at the peak!!!
  - We ignored other possible overheads on the bus
  - In practice the gap can be around a factor 15 or higher

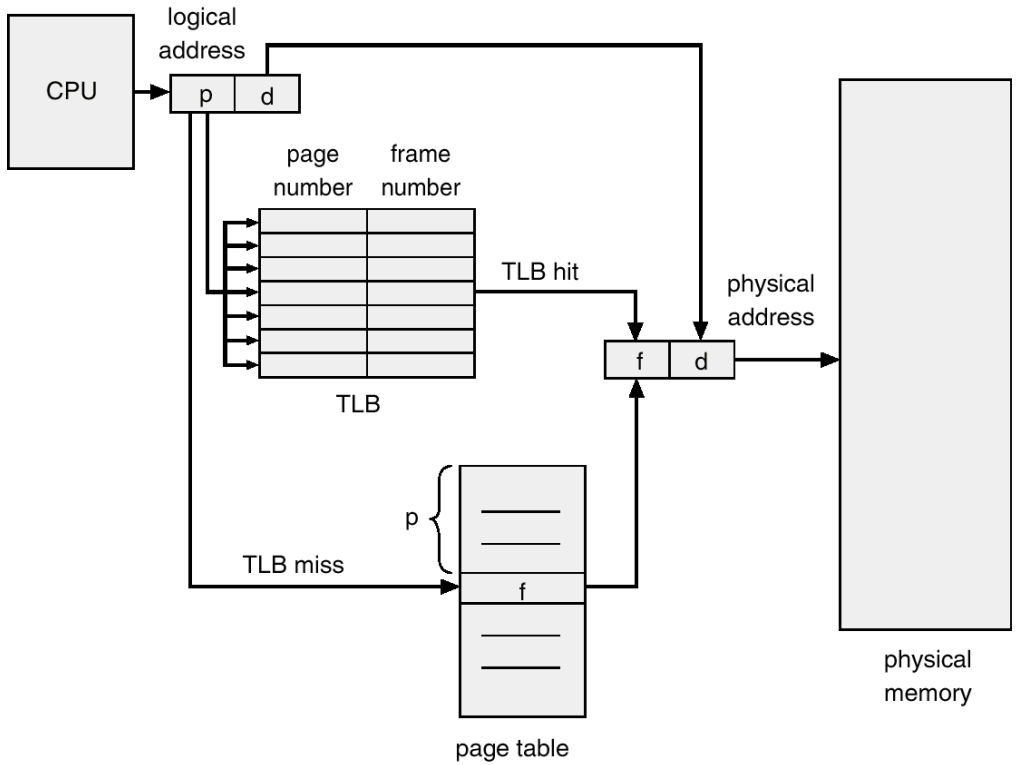
Dealing With Memory

- How have people been dealing with the memory bottleneck?
- Computers are built with a memory hierarchy
  - Registers, Multiple Levels of Cache, Main memory
  - Data is brought in in bulk (cache line) from a lower level (slow, cheap, big) to a higher level (fast, expensive, small)
  - When the cache is full, we need a policy to decide what should stay in the cache and what should be replaced
  - Hopefully brought in in a cache line will be (re)used soon
    - temporal locality
    - spatial locality
- Programs must be aware of the memory hierarchy (at least to some extent)
  - Makes life difficult when writing for performance
  - But is necessary on most systems

Making paging fast

- **x86 PTs require 3 memory reference per load/store**
  - Look up page table address in page directory
  - Look up PPN in page table
  - Actually access physical page corresponding to virtual address
- **For speed, CPU caches recently used translations**
  - Called a *translation lookaside buffer* or TLB
  - Typical: 64-2K entries, 4-way to fully associative, 95% hit rate
  - Each TLB entry maps a VPN $\rightarrow$ PPN + protection information
- **On each memory reference**
  - Check TLB, if entry present get physical address fast
  - If not, walk page tables, insert in TLB for next time
    (Must evict some entry. We'll discuss eviction soon.)

TLB Principle



- A TLB is a fast (small) associative memory which can perform a parallel search
- It acts as a cache for the paging table
- TLB management can either be done at hardware or software level

TLB: Effective Access Time

- The percentage that a particular page number is found in the TLB is called **hit ratio**
- **Typical TLB:**
  - Size: 8 - 4,096 entries
  - Hit time: 0.5 - 1 clock cycle
  - Miss penalty: 10 - 100 clock cycles
  - Miss rate: 0.01 - 1%

  If a TLB hit takes 1 clock cycle, a miss takes 30 clock cycles, and the miss rate is 1%, the effective memory cycle rate is an average of

$$1 \times 0.99 + (1 + 30) \times 0.01 = 1.021$$

  (1.021 clock cycles per memory access).
  A 10% miss rate would lead to 4 cycles...
- There may be **multiple TLBs** (e.g., a very small and fully associative one, then a larger and smaller TLB, and so on)

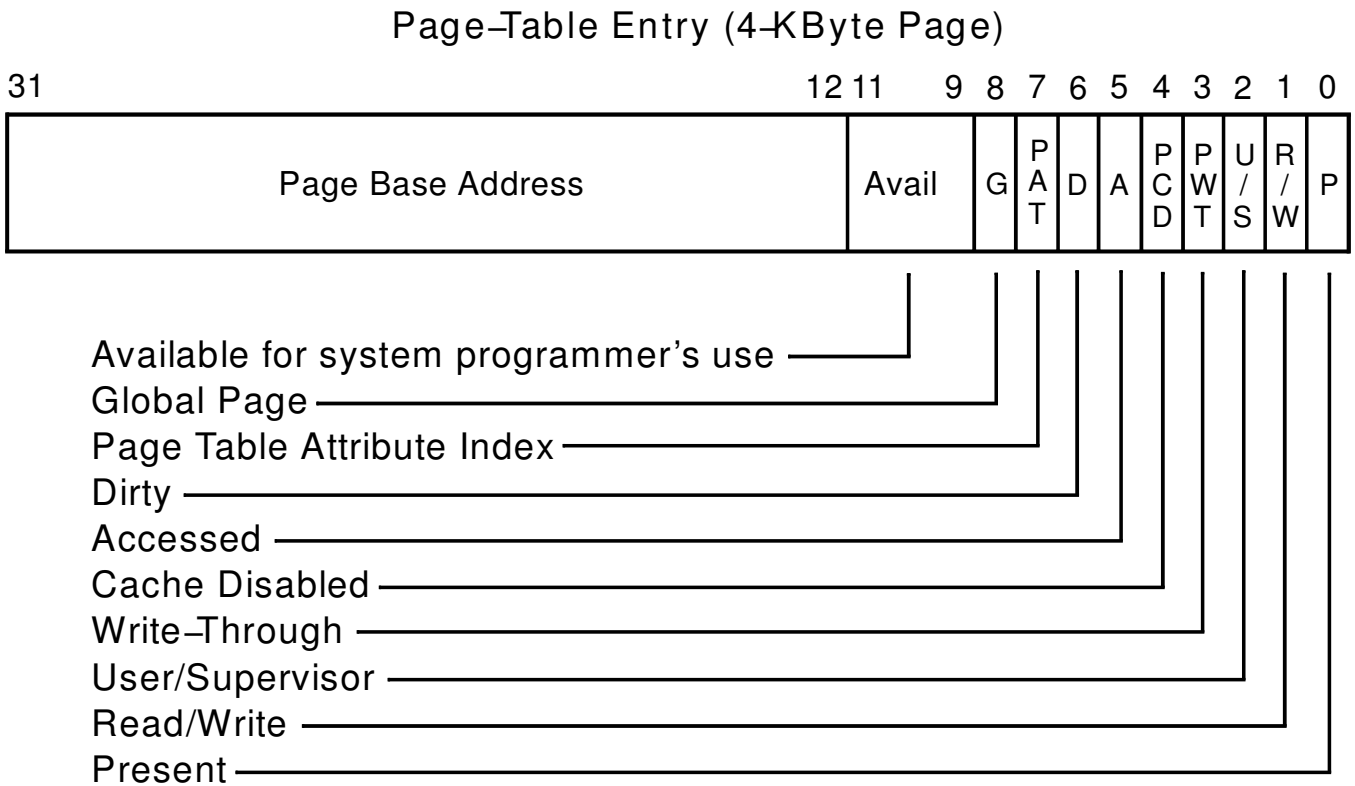
TLB details

- **TLB operates at CPU pipeline speed $\Longrightarrow$ small, fast**
- **Complication: what to do when switch address space?**
  - Flush TLB on context switch (e.g., x86 until recently)
  - Tag each entry with associated process's ID (e.g., MIPS): ASIDs
  - With the advent of virtualization for server consolidation, the x86 architecture has started introducing such mechanism
- **In general, OS must manually keep TLB valid**
- **E.g., x86 invlpg instruction**
  - Invalidates a page translation in TLB
  - Must execute after changing a possibly used page table entry
  - Otherwise, hardware will miss page table change
- **More Complex on a multiprocessor since every core has its own TLB. Maintaining consistency is non-trivial (TLB shootdown)**

Outline

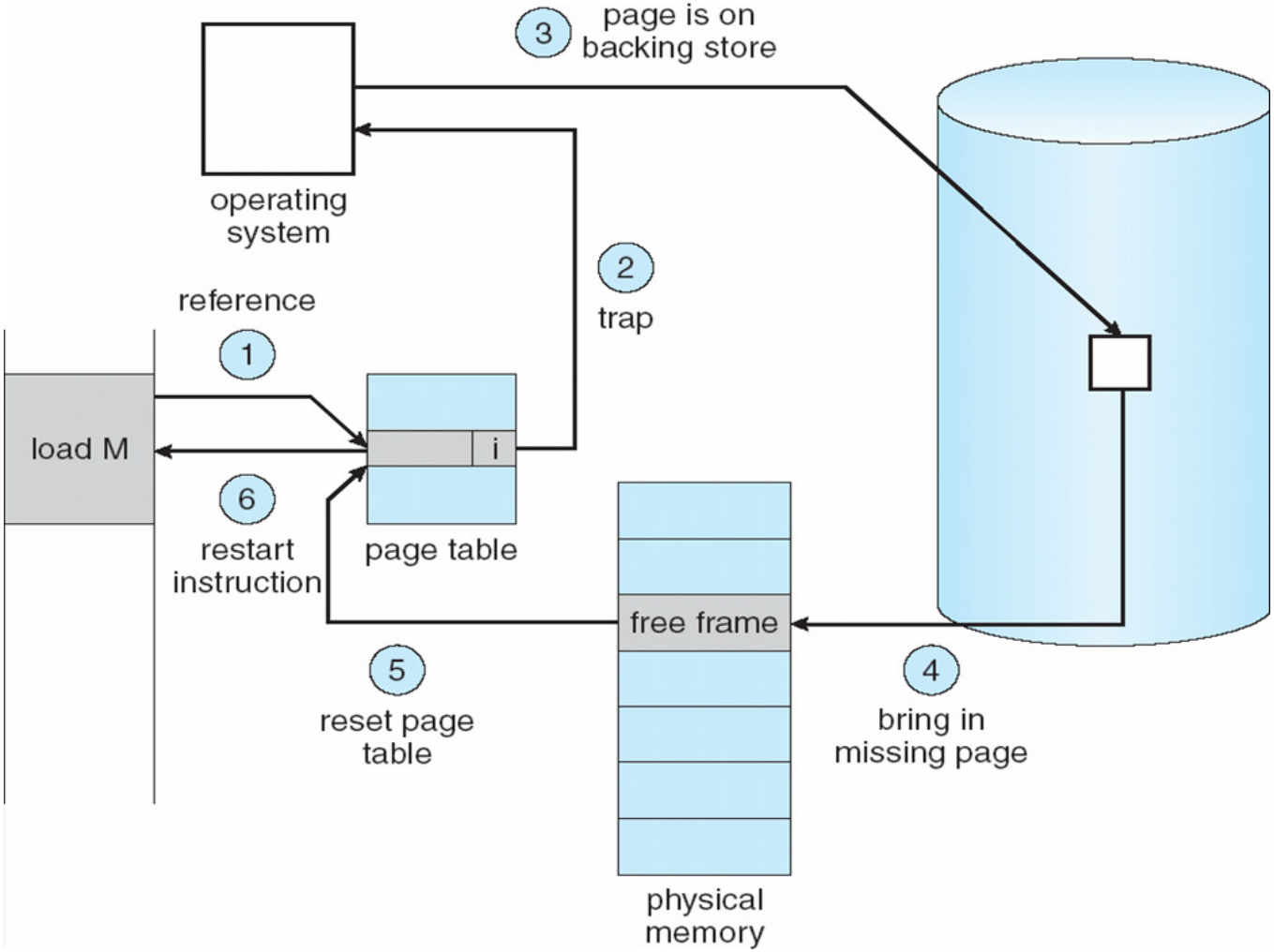
Paging

Use disk to simulate larger virtual than physical mem



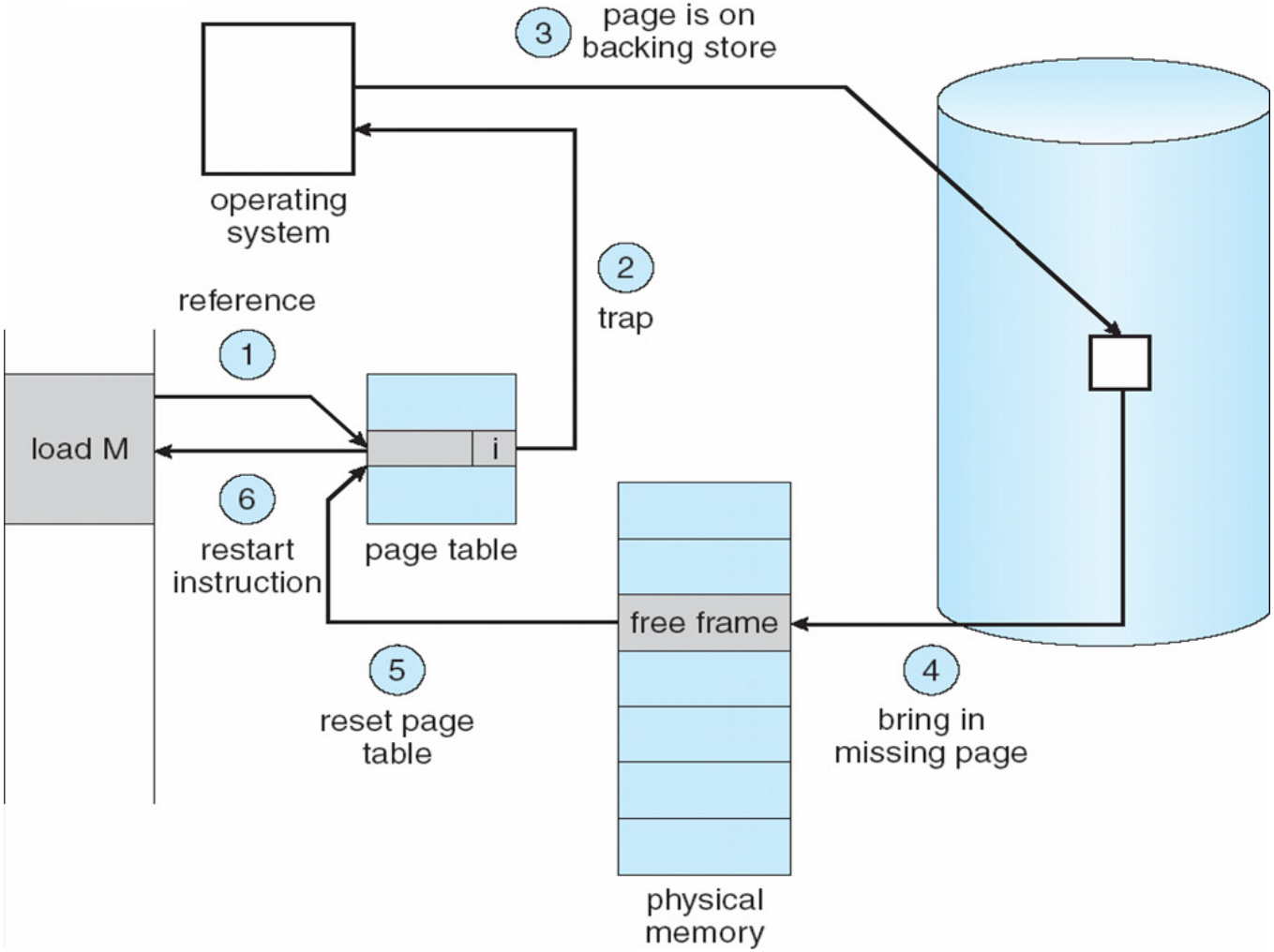
Paging

Use disk to simulate larger virtual than physical mem



Paging

Use disk to simulate larger virtual than physical mem



"The RAM acts like a cache for the disk".

Example: Paging to disk

- `gcc` **needs a new page of memory**
- **OS re-claims an idle page from** `emacs`
- **If page is clean (i.e., also stored on disk):**
  - E.g., page of text from emacs binary on disk
  - Can always re-read same page from binary
  - So okay to discard contents now & give page to `gcc`
- **If page is dirty (meaning memory is only copy)**
  - Must write page to disk first before giving to `gcc`
- **Either way:**
  - Mark page invalid in `emacs`
  - `emacs` will fault on next access to virtual page
  - On fault, OS reads page data back from disk into new page, maps new page into `emacs`, resumes executing

Performance

- **Page fault service time is depends on**
  - Servicing the page fault interrupt ($\approx$ 1-100 ns).
  - Reading the page ($\approx$ **8 ms.**)
    - HD average latency $\approx 3ms$
    - HD average seek $\approx 5ms$
    - HD transfer time $\approx .05ms$/page
  - Restarting the process ($\approx$ 1-100 ns)
- **Effective access time:**

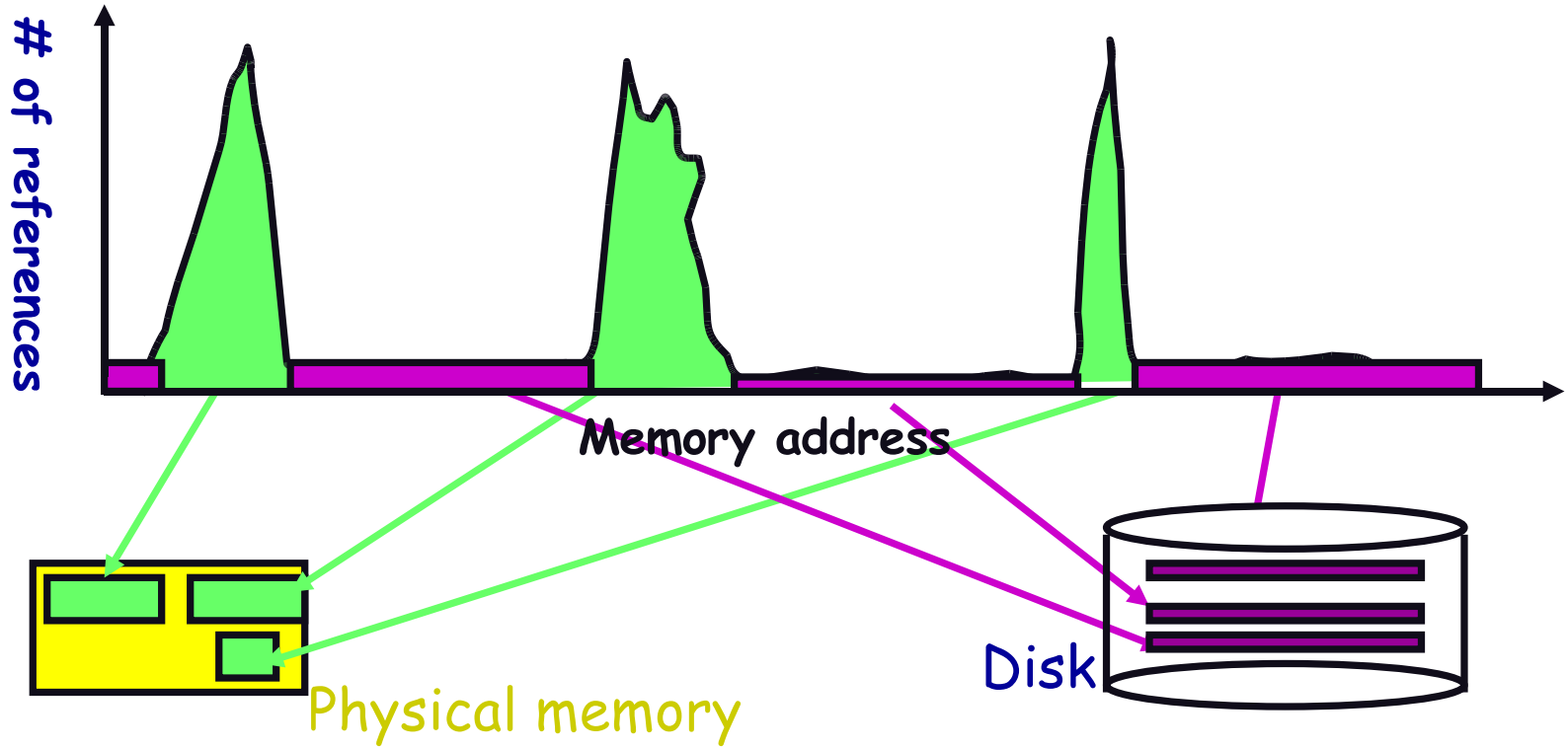
$$\begin{aligned}\textbf{Effective access time} &= (1-p) \times 200ns + p \times 8ms \\ &= (1-p) \times 200 + p \times 8,000,000ns \\ &= 200 + 7,999,800 \times p\end{aligned}$$

p = 1‰ $\Rightarrow$ EAT = 8200 (slowdown = 40)
A degradation smaller than 10% requires p $< 2.5\ 10^{-6}$!!!

Paging in day-to-day use

- **Demand paging**
- **Growing the stack**
- **BSS page allocation**
- **Shared text**
- **Shared libraries**
- **Shared memory**
- **Copy-on-write (`fork`, `mmap`, etc.)**
- **Bypass the File System (direct access to the H.D.)**

Working set model



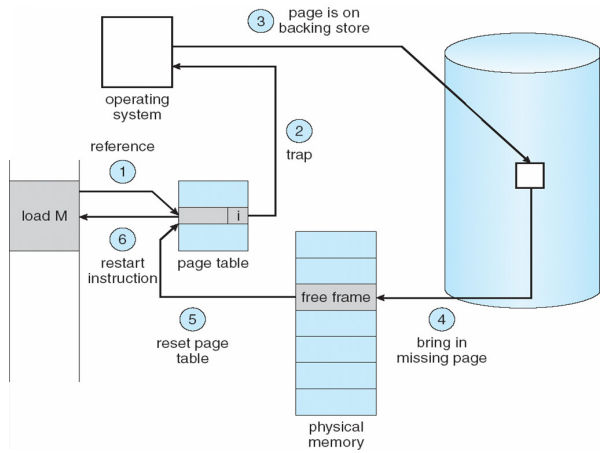
- **Disk much, much slower than memory**
  - Goal: Run at memory, not disk speeds
- **90/10 rule: 10% of memory gets 90% of memory refs**
  - So, keep that 10% in real memory, the other 90% on disk
  - How to pick which 10%?

Paging challenges

- **How to resume a process after a fault?**
  - Need to save state and resume
  - Process might have been in the middle of an instruction!
- **What to fetch?**
  - Just needed page or more?
- **What to eject?**
  - How to allocate physical pages amongst processes?
  - Which of a particular process's pages to keep in memory?

Re-starting instructions

- **Hardware provides kernel w. info about page fault**
  - Faulting virtual address (In `%cr2` reg on x86)
  - Address of instruction that caused fault
  - Was the access a read or write? Was it an instruction fetch?
    Was it caused by user access to kernel-only memory?



- **Hardware must allow resuming after a fault**
- **Idempotent instructions are easy**
  - E.g., simple load or store instruction can be restarted
  - Just re-execute any instruction that only accesses one address
- **Complex instructions must be re-started, too**
  - E.g., x86 move string instructions
  - Specify srd, dst, count in `%esi`, `%edi`, `%ecx` registers
  - On fault, registers adjusted to resume where move left off

What to fetch?

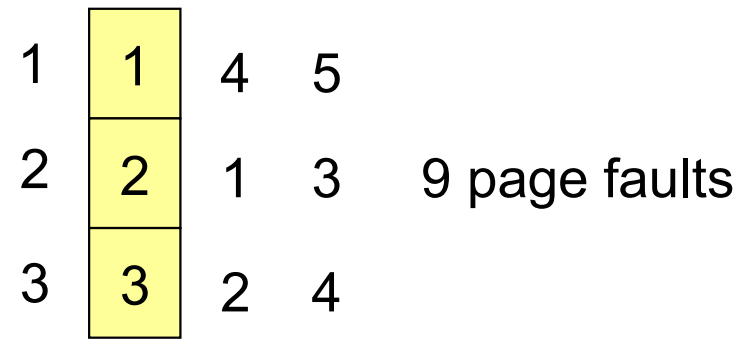
- **Bring in page that caused page fault**
- **Pre-fetch surrounding pages?**
  - Reading two disk blocks approximately as fast as reading one
  - As long as no track/head switch, seek time dominates
  - If application exhibits spacial locality, then big win to store and read multiple contiguous pages
- **Also pre-zero unused pages in idle loop**
  - Need 0-filled pages for stack, heap, BSS, anonymously mmapped memory
  - Zeroing them only on demand is slower
  - So many OSes zero freed pages while CPU is idle

Selecting physical pages

- **May need to eject some pages**
  - More on eviction policy in two slides
- **May also have a choice of physical pages**
- **Direct-mapped physical caches**
  - Virtual $\rightarrow$ Physical mapping can affect performance
  - Applications can conflict with each other or themselves
  - Scientific applications benefit if consecutive virtual pages to not conflict in the cache
  - Many other applications do better with random mapping

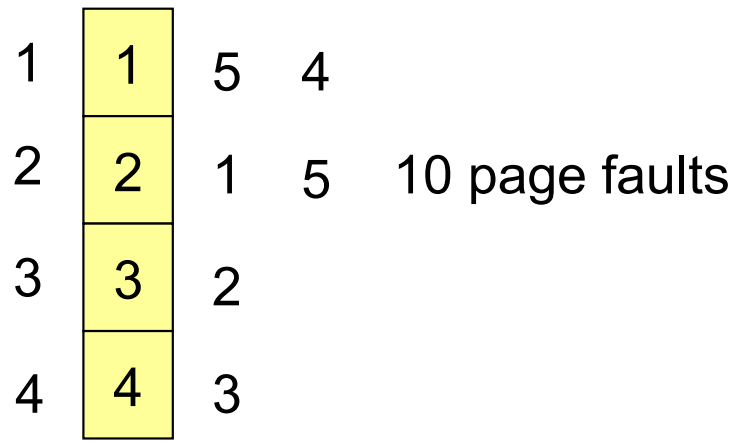
Straw man: FIFO eviction

- **Evict oldest fetched page in system**
- **Example—reference string 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5**
- 3 physical pages: 9 page faults

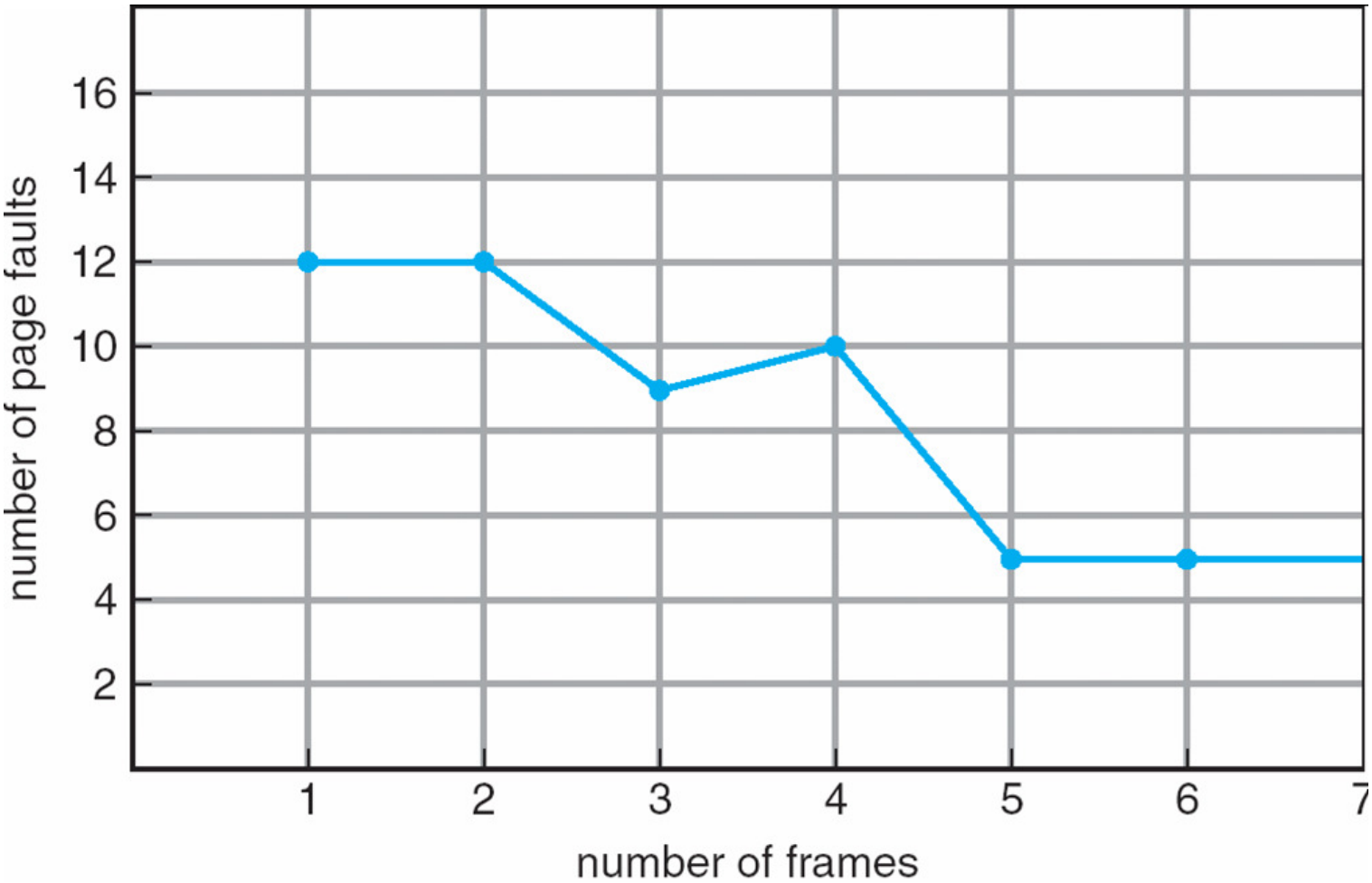


Straw man: FIFO eviction

- **Evict oldest fetched page in system**
- **Example—reference string 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5**
- 3 physical pages: 9 page faults
- 4 physical pages: 10 page faults



Belady's Anomaly



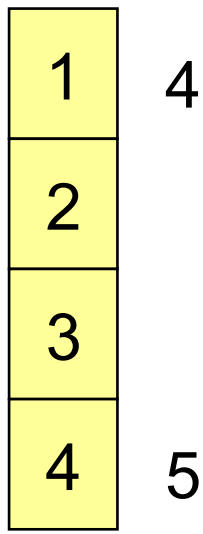
- **More phys. mem. doesn't always mean fewer faults**

Optimal page replacement

- **What is optimal (if you knew the future)?**

Optimal page replacement

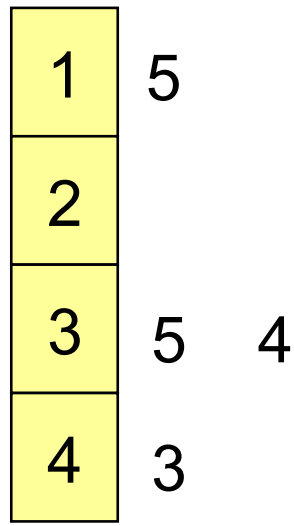
- **What is optimal (if you knew the future)?**
  - Replace page that will not be used for longest period of time
- **Example—reference string 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5**
- **With 4 physical pages:**



6 page faults

LRU page replacement

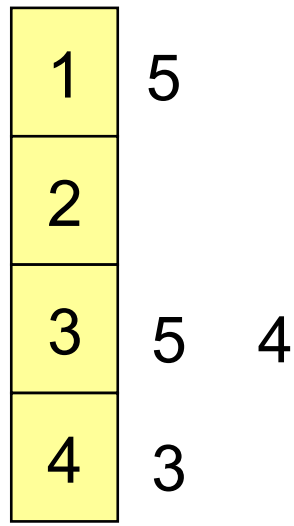
- **Approximate optimal with least recently used**
  - Because past often predicts the future
- **Example—reference string 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5**
- **With 4 physical pages: 8 page faults**



- **Problem 1: Can be pessimal – example?**

LRU page replacement

- **Approximate optimal with least recently used**
  - Because past often predicts the future
- **Example—reference string 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5**
- **With 4 physical pages: 8 page faults**



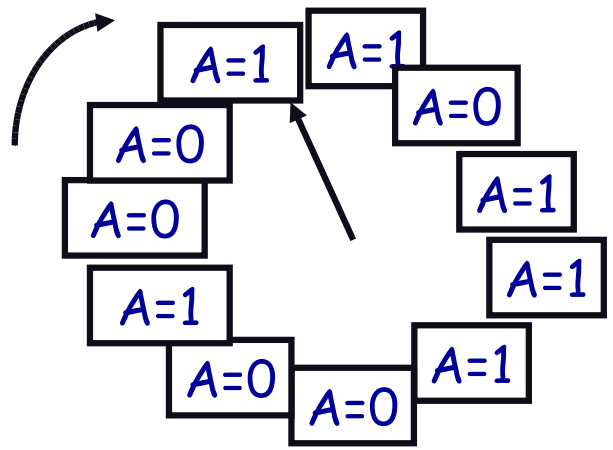
- **Problem 1: Can be pessimal – example?**
  - Looping over memory (then want MRU eviction)
- **Problem 2: How to implement?**

Straw man LRU implementations

- **Stamp PTEs with timer value**
  - E.g., CPU has cycle counter
  - Automatically writes value to PTE on each page access
  - Scan page table to find oldest counter value = LRU page
  - Problem: Would double memory traffic!
- **Keep doubly-linked list of pages**
  - On access remove page, place at tail of list
  - Problem: again, very expensive
- **What to do?**
  - Just approximate LRU, don't try to do it exactly

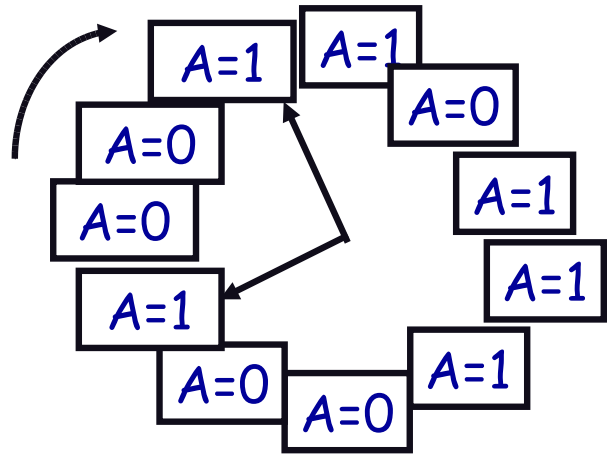
Clock algorithm

- **Use accessed bit supported by most hardware**
  - E.g., Pentium will write 1 to A bit in PTE on first access
  - Software managed TLBs like MIPS can do the same
- **Do FIFO but skip accessed pages**
- **Keep pages in circular FIFO list**
- **Scan:**
  - page's A bit = 1, set to 0 & skip
  - else if A == 0, evict
- **A.k.a. second-chance replacement**



Clock alg. (continued)

- **Large memory may be a problem**
  - Most pages reference in long interval
- **Add a second clock hand**
  - Two hands move in lockstep
  - Leading hand clears A bits
  - Trailing hand evicts pages with A=0

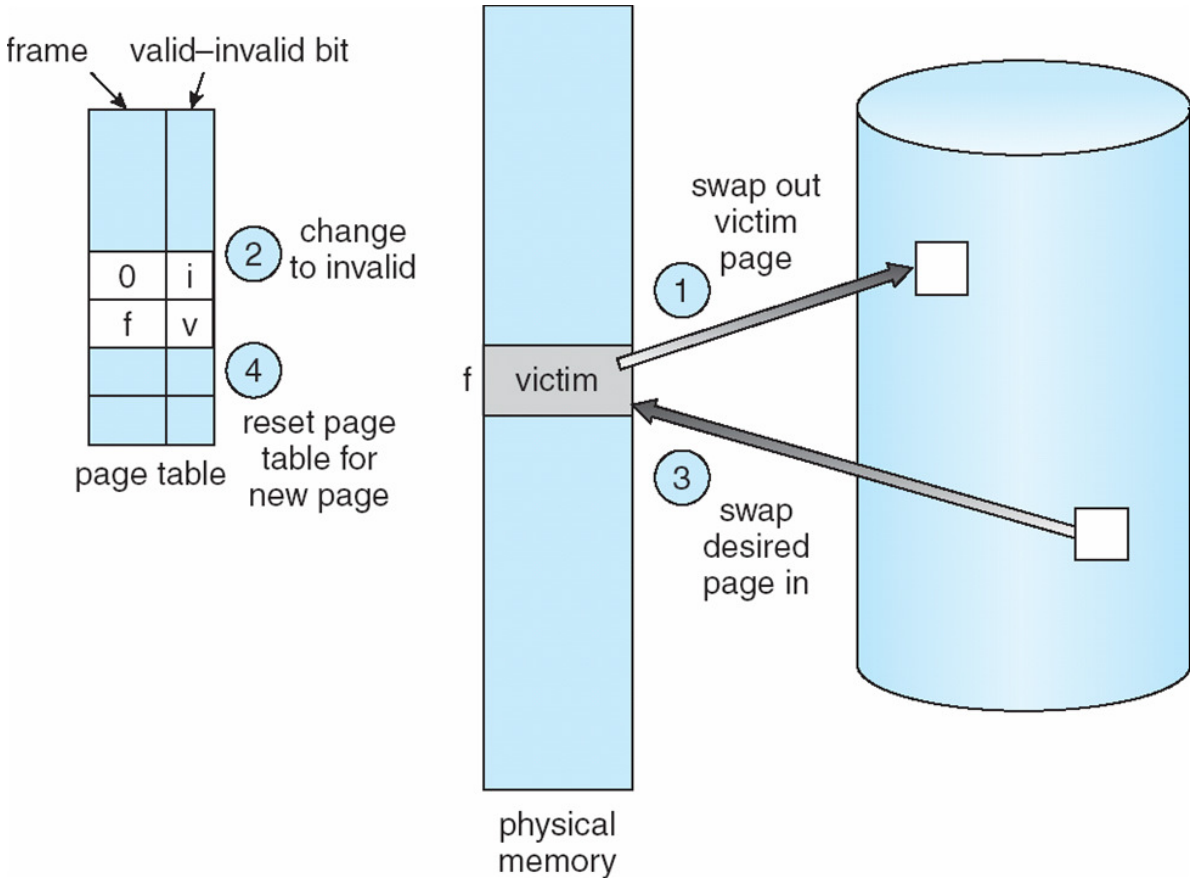


- **Can also take advantage of hardware Dirty bit**
  - Each page can be (Unaccessed, Clean), (Unaccessed, Dirty), (Accessed, Clean), or (Accessed, Dirty)
  - Consider clean pages for eviction before dirty
- **Or use n-bit accessed count instead just A bit**
  - On sweep: $count = (A \ll (n-1)) \mid (count \gg 1)$
  - Evict page with lowest *count*

Other replacement algorithms

- **Random eviction**
  - Dirt simple to implement
  - Not overly horrible (avoids Belady & pathological cases)
- **LFU (least frequently used) eviction**
  - instead of just A bit, count # times each page accessed
  - least frequently accessed must not be very useful (or maybe was just brought in and is about to be used)
  - decay usage counts over time (for pages that fall out of usage)
- **MFU (most frequently used) algorithm**
  - because page with the smallest count was probably just brought in and has yet to be used
- **Neither LFU nor MFU used very commonly**

Naïve paging



- **Naïve page replacement: 2 disk I/Os per page fault**

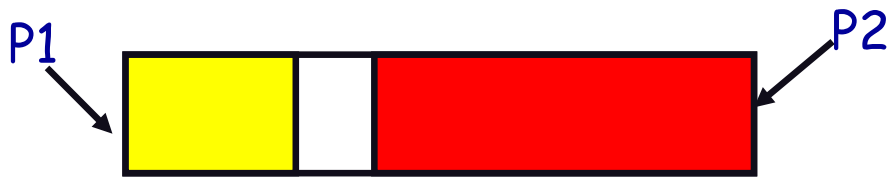
Page buffering

- **Idea: reduce # of I/Os on the critical path**
- **Keep pool of free page frames**
  - On fault, still select victim page to evict
  - But read fetched page into already free page
  - Can resume execution while writing out victim page
  - Then add victim page to free pool
- **Can also yank pages back from free pool**
  - Contains only clean pages, but may still have data
  - If page fault on page still in free pool, recycle

Outline

Page allocation

- **Allocation can be global or local**
- **Global allocation doesn't consider page ownership**
  - E.g., with LRU, evict least recently used page of any proc
  - Works well if P1 needs 20% of memory and P2 needs 70%:



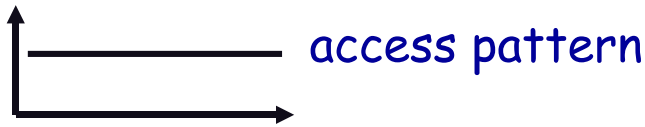
  - Doesn't protect you from memory pigs
    (imagine P2 keeps looping through array that is size of mem)
- **Local allocation isolates processes (or users)**
  - Separately determine how much mem each proc. should have
  - Then use LRU/clock/etc. to determine which pages to evict within each process

Thrashing

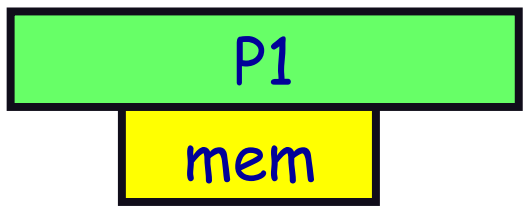
- **Thrashing: processes on system require more memory than it has**
  - Each time one page is brought in, another page, whose contents will soon be referenced, is thrown out
  - Processes will spend all of their time blocked, waiting for pages to be fetched from disk
  - I/O devs at 100% utilization but system not getting much useful work done
- **What we wanted: virtual memory the size of disk with access time the speed of physical memory**
- **What we have: memory with access time of disk**

Reasons for thrashing

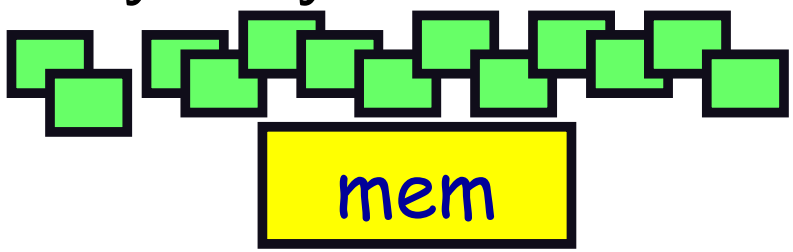
- **Process doesn't reuse memory, so caching doesn't work (past != future)**



- **Process does reuse memory, but it does not "fit"**

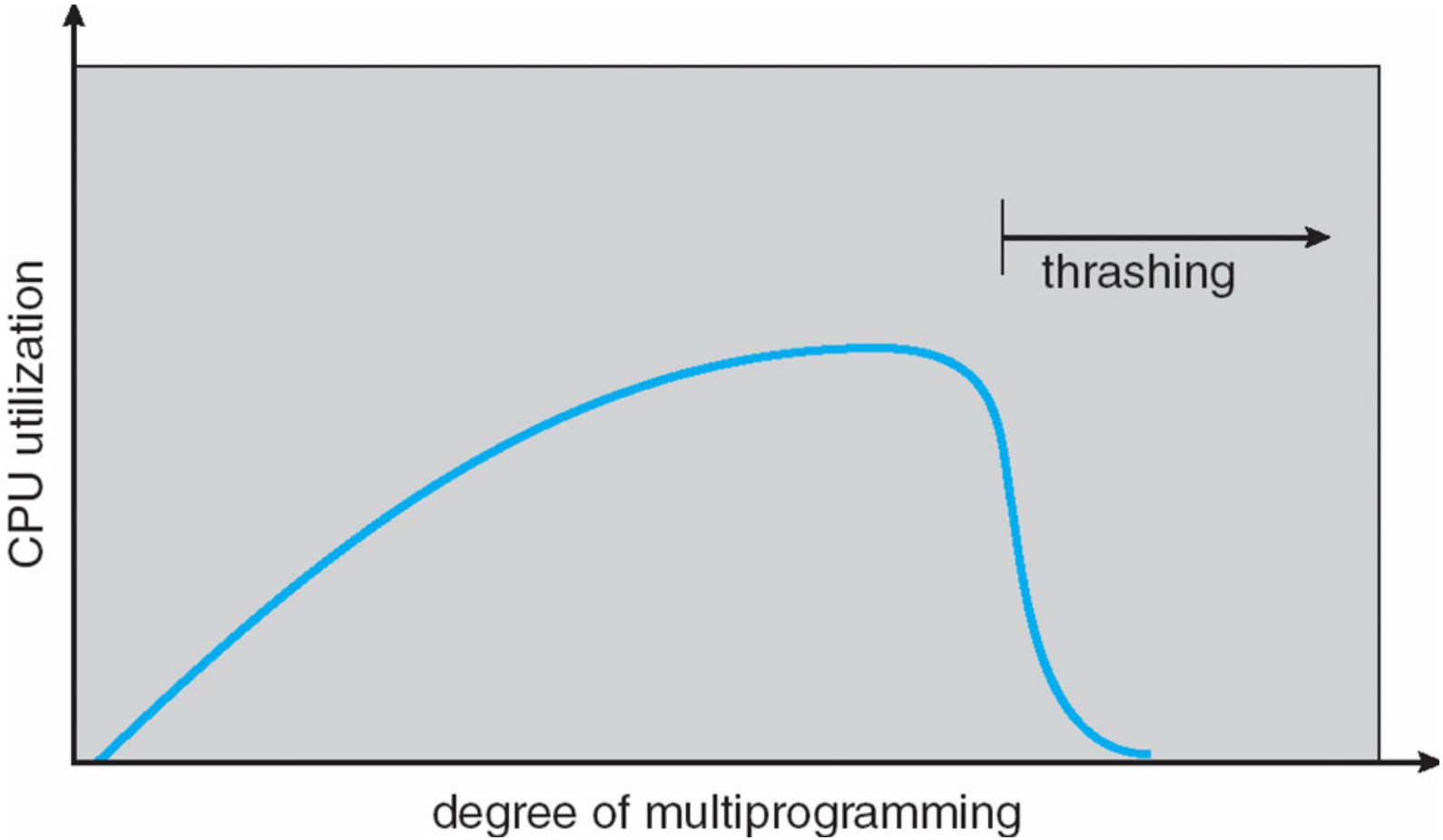


- **Individually, all processes fit and reuse memory, but too many for system**



  - At least this case is possible to address

Multiprogramming & Thrashing

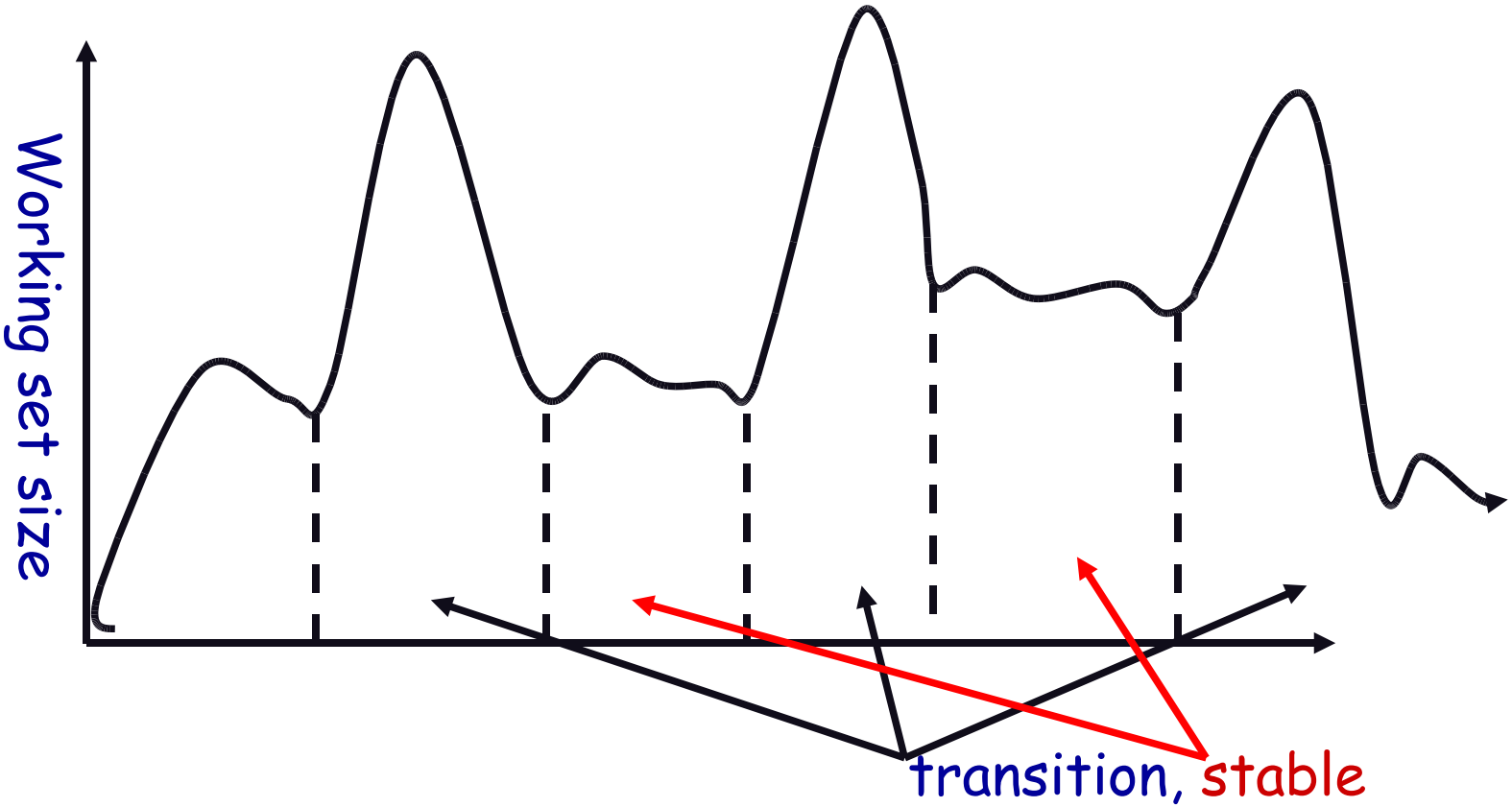


- **Need to shed load when thrashing**

Dealing with thrashing

- **Approach 1: working set**
  - Thrashing viewed from a caching perspective: given locality of reference, how big a cache does the process need?
  - Or: how much memory does process need in order to make reasonable progress (its working set)?
  - Only run processes whose memory requirements can be satisfied
- **Approach 2: page fault frequency**
  - Thrashing viewed as poor ratio of fetch to work
  - PFF = page faults / instructions executed
  - If PFF rises above threshold, process needs more memory
    not enough memory on the system? Swap out.
  - If PFF sinks below threshold, memory can be taken away

Working sets



- **Working set changes across phases**
  - Baloons during transition

Calculating the working set

- **Working set: all pages proc. will access in next T time**
  - Can't calculate without predicting future
- **Approximate by assuming past predicts future**
  - So working set $\approx$ pages accessed in last T time
- **Keep idle time for each page**
- **Periodically scan all resident pages in system**
  - A bit set? Clear it and clear the page's idle time
  - A bit clear? Add CPU consumed since last scan to idle time
  - Working set is pages with idle time $< T$

Two-level scheduler

- **Divide processes into active & inactive**
  - Active – means working set resident in memory
  - Inactive – working set intentionally not loaded
- **Balance set: union of all active working sets**
  - Must keep balance set smaller than physical memory
- **Use long-term scheduler**
  - Moves procs active $\rightarrow$ inactive until balance set small enough
  - Periodically allows inactive to become active
  - As working set changes, must update balance set
- **Complications**
  - How to chose idle time threshold T?
  - How to pick processes for active set
  - How to count shared memory (e.g., libc.so)

Some complications of paging

- **What happens to available memory?**
  - Some physical memory tied up by kernel VM structures
- **What happens to user/kernel crossings?**
  - More crossings into kernel
  - Pointers in syscall arguments must be checked
    (can't just kill proc. if page not present—might need to page in)
- **What happens to IPC?**
  - Must change hardware address space
  - Increases TLB misses
  - Context switch flushes TLB entirely on old x86 machines
    (But not on MIPS. . . Why?)

Outline

Recap

Paging nice features

- removes the fragmentation issue
- enables to offload the RAM (demand paging) and thus to fit more process in RAM
- enables to run process requiring more memory than available RAM

Replacement issues

- when the RAM is full, a page must be evicted, stored back on the disk and replaced in RAM by the requested one
- this content management is similar to the one in caches, TLB, ...
- Good policies build on locality, regularity of memory access.
- Workload and speed/size of the different components (TLB vs. cache L1 vs. cache L2 vs. RAM vs. disk, disk cache vs. cylinders, ...) call for different policies, data structures and tradeoffs.