

Memory - Paging

[Physical Frame Allocation]

- How do we allocate physical memory across multiple processes?
 - What if Process A needs to evict a page from Process B?
 - How do we ensure fairness?
 - How do we avoid having one process hogging the entire memory of the system?



Physical Frame Allocation

- Local replacement algorithms
 - Per-process limit on the physical memory usage of each process
 - When a process reaches its limit, it evicts pages from itself
- Global-replacement algorithms
 - Physical size of processes can grow and shrink over time
 - Allow processes to evict pages from other processes
- Note that one process' paging can impact performance of the entire system!
 - One process that does a lot of paging will induce more disk I/O



Allocation of Page Frames

- Scenario
 - Several physical pages allocated to processes A, B, and C. Process B page faults.
 - Which page should be replaced?
- Allocating memory across processes?
 - Does every process get the same fraction of memory?
 - Different fractions?
 - Should we completely swap some processes out of memory?



Allocation of Page Frames

- Each process needs minimum number of pages
 - Want to make sure that all processes that are loaded into memory can make forward progress
 - Example: IBM 370 – 6 pages to handle SS MOVE instruction:
 - Instruction is 6 bytes, might span 2 pages
 - 2 pages to handle from
 - 2 pages to handle to



[Fixed Allocation]

- Allocate a minimum number of frames per process
- Consider minimum requirements
 - One page from the current executed instruction
 - Most instructions require two operands
 - Include an extra page for paging out and one for paging in



[Equal Allocation]

- Allocate an equal number of frames per job
 - Example
 - 100 frames
 - 5 processes
 - Each process gets 20 frames
- Issues
 - But jobs use memory unequally
 - High priority jobs have same number of page frames and low priority jobs
 - Degree of multiprogramming might vary



[Proportional Allocation]

- Allocate a number of frames per job proportional to job size
 - How do you determine job size
 - Run command parameters ?
 - Dynamically?
- Priority Allocation
 - May want to give high priority process more memory than low priority process
 - Use a proportional allocation scheme using priorities instead of size



Local Replacement: Per Process

- Each process has separate pool of pages
 - Fixed number of pages (e.g., Digital VMS)
 - Fixed fraction of physical memory (1/P)
 - Proportional to size of allocated address space
- Page fault in one process only replaces pages of that process
 - Perform replacement (e.g., LRU) over only those pages
- Advantage
 - No interference across processes
- Disadvantage
 - Potentially inefficient allocation of memory
 - How to handle sharing of pages?



[Local Replacement: Per User]

- Each user has separate pool of pages
- Advantage
 - Fair across different users
- Disadvantage
 - Inefficient allocation



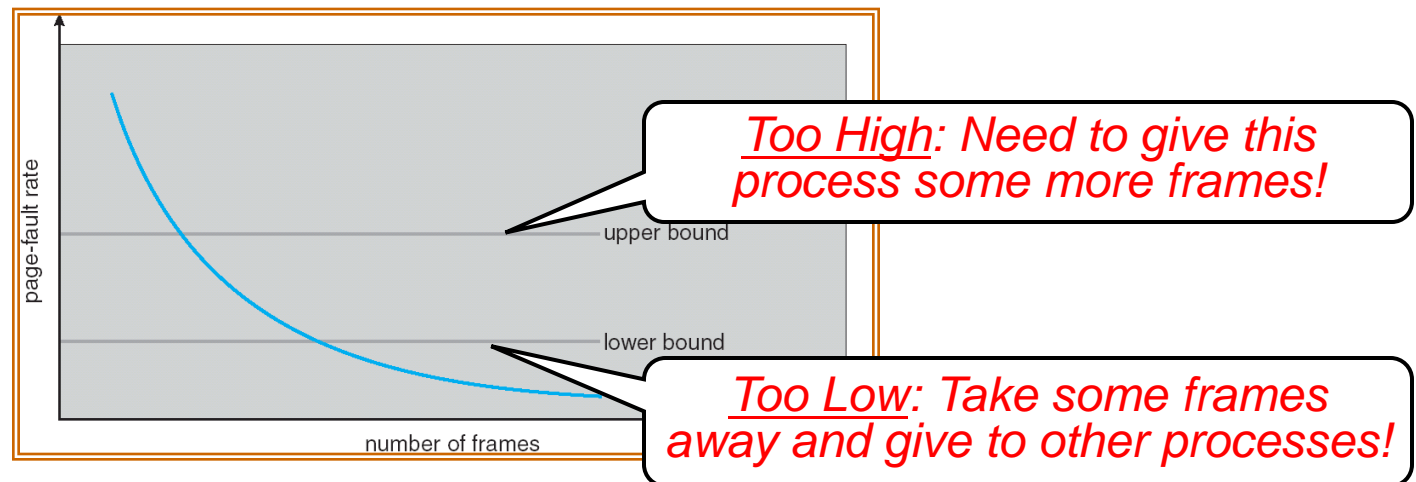
[Global Replacement]

- Pages from all processes lumped into single replacement pool
 - Example: Run clock over all page frames
- Each process competes with other processes for frames
- Advantages
 - Flexibility of allocation
 - Minimize total number of page faults
- Disadvantages
 - One memory-intensive process can hog memory, hurt all processes



Page Fault Frequency Allocation

- Can we reduce Capacity misses by dynamically changing the number of pages/application?



- Establish “acceptable” page-fault rate
 - If actual rate too low, process loses frame
 - If actual rate too high, process gains frame
- Question: What if we just don’t have enough memory



[Overcommitting Memory]

- When does the Virtual Memory illusion break?
- Example
 - Set of processes frequently references 33 important pages
 - Physical memory can fit 32 pages
- What happens?
 - Process A references page not in physical memory
 - OS runs another process B
 - OS replaces some page in memory with page for A
 - How long can B run before it page faults?
 - Cycle continues...



[Overcommitting Memory]

- If a process does not have enough pages, the page-fault rate is very high
 - Low CPU utilization
 - OS thinks that it needs to increase the degree of multiprogramming
 - Another process is added to the system.
 - System throughput plunges...
- System is reading and writing pages instead of executing useful instructions
 - Average memory access time = disk access time
 - Memory appears as slow as disk, instead of disk appearing as fast as memory



[Thrashing]

- If a process does not have enough frames, the page-fault rate is very high
 - Low CPU utilization
- Thrashing
 - A process is busy swapping pages in and out
 - In other words, a process is spending more time paging than executing



[Thrashing]

- Example
 - Process has 3 frames allocated to it (use LRU)
 - Reference string is 123412341234... - 4th access onwards all cause page faults
- Cannot be fixed with better replacement policies
 - Do not indicate that a page must be kept in memory
 - Only show which pages are better than others to replace



[Thrashing]

- Student's analogy to thrashing: Too many courses
 - Drop a course
- OS solution: Admission control
 - Determine how much memory each process needs
 - Long-term scheduling policy
 - Run only those processes whose memory requirements can be satisfied
 - What if memory needs of one process are too large????



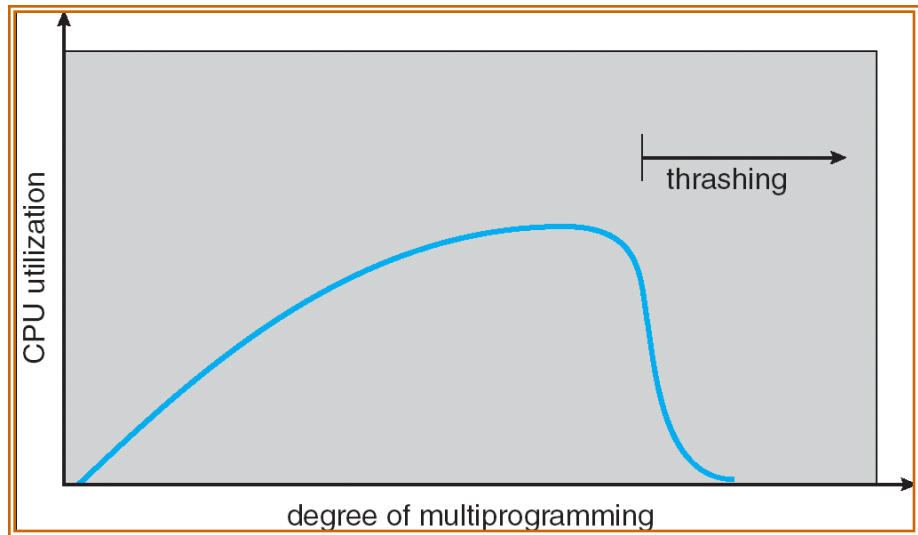
[Why Thrashing?]

- Computations have locality
 - Set of pages that are actively used together
- As the number of page frames decreases
 - There are not enough available page frames to contain the locality of the process
- Processes start faulting heavily
 - Pages that are read in, are used and immediately paged out



[Thrashing]

- Page fault rate goes up
 - Processes get suspended for page out to disk
- The system may start new jobs
 - Reduces number of available page frames
 - Increases page faults
- System throughput plunges!



[Working Set]

- Question
 - How much memory does a process need to keep the most recent computation in memory with very few page faults?
- How can we determine this?
 - Determine the working set of a process
 - The principle of locality
 - A program clusters its access to data and text temporally
 - A recently accessed page is more likely to be accessed again



[Working Set (1968, Denning)]

- Need
 - Set of pages process needs to avoid thrashing
 - Requires knowing the future
- Working set
 - Pages referenced by process in last T seconds of execution
 - Approximates locality
- Working set changes over the lifetime of the process
 - Periods of high locality exhibit smaller working set
 - Periods of low locality exhibit larger working set

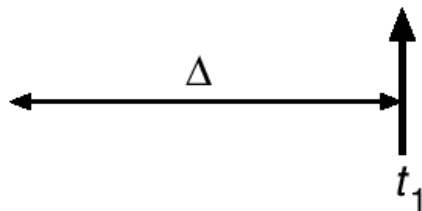


[Working Set (1968, Denning)]

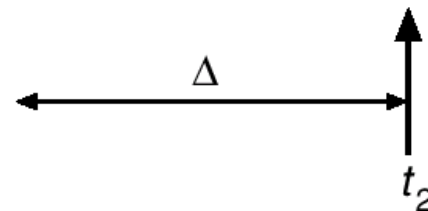
- $\Delta \equiv$ working-set window
 - Fixed number of page references
 - Example: 10,000 instruction
- Working set of P_i
 - Pages referenced in most recent Δ

page reference table

... 2 6 1 5 7 7 7 7 5 1 6 2 3 4 1 2 3 4 4 4 3 4 3 4 4 4 1 3 2 3 4 4 4 3 4 4 4 ...



$$WS(t_1) = \{1, 2, 5, 6, 7\}$$



$$WS(t_2) = \{3, 4\}$$

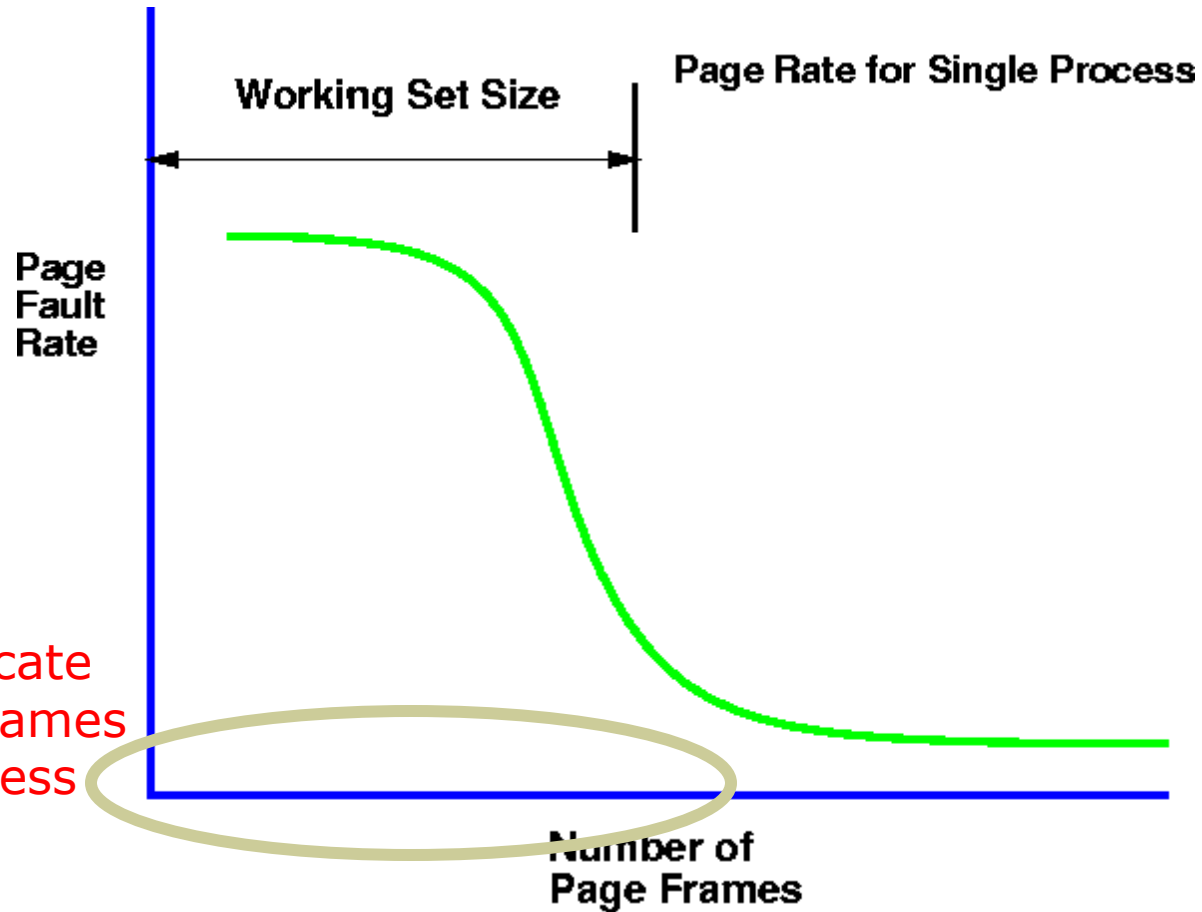


[Working Set (1968, Denning)]

- Using working set sizes
 - Cache partitioning
 - Give each app enough space for WS
 - Page replacement
 - Preferentially discard non-WS pages
 - Scheduling
 - Process not executed unless WS in memory



Working Set Size



At least allocate
this many frames
for this process

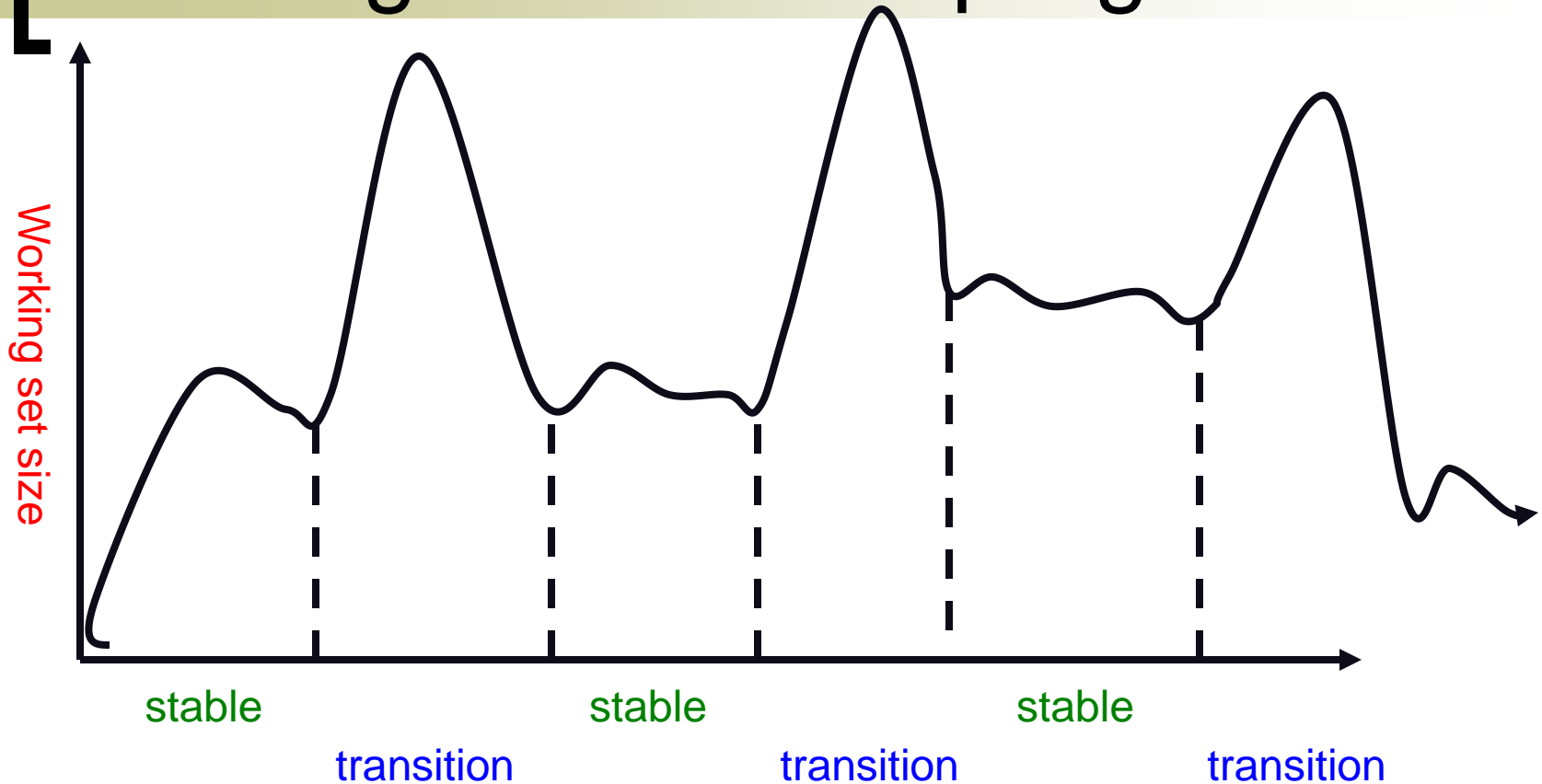


[Working Set Size]

- Choosing T
 - T too small
 - Will not encompass entire locality
 - T too large
 - Will encompass several localities
 - $T = \infty$
 - Will encompass entire program



Working sets of real programs



- Typical programs have phases



Working Set in Action to Prevent Thrashing

■ Algorithm

- If number free page frames $>$ working set of some suspended process_{*i*}
 - Activate process_{*i*} and map in its working set
- If working set size of some process_{*k*} increases and no page frame is free
 - Suspend process_{*k*} and release all its pages

■ Tracking the working set

- Moving window over reference string
- Approximate with Interval timer + a reference bit



Working Set Implementation

- Example: $T = 10,000$
 - Timer interrupts after every 5000 time units
 - Copy and set the values of all reference bits to 0
 - Keep in memory 2 bits for each page
 - Indicates if page was used within last 10,000 to 15,000 references
 - If one of the bits in memory = 1
 - Page in working set
- Not completely accurate - cannot tell where reference occurred.
 - Improvement - 10 bits and interrupt every 1000 time units



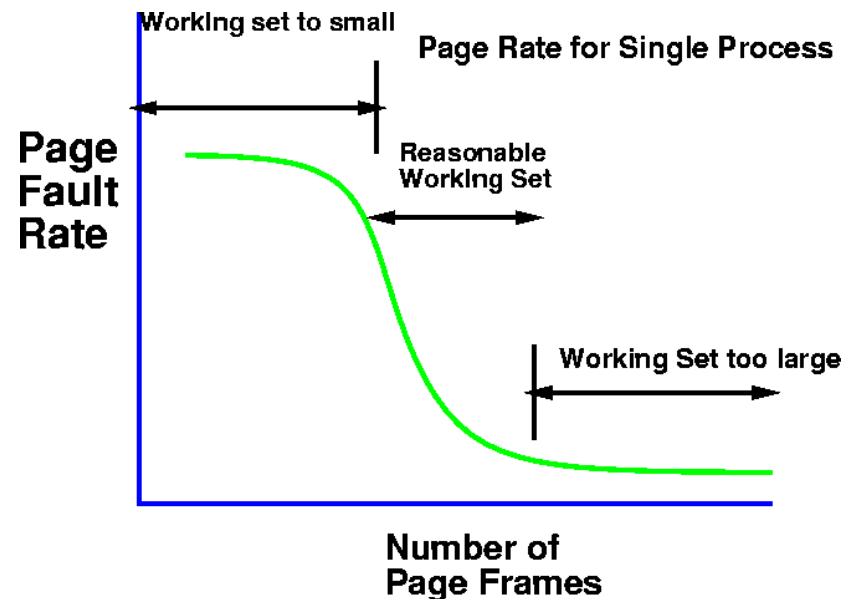
Page Fault Frequency Working Set

- Approximation of pure working set
- Assumption
 - If the working set is correct
 - Not many page faults
- Approach
 - Control thrashing by establishing acceptable page-fault rate



Page Fault Frequency Working Set

- Algorithm
 - If page fault rate increases beyond assumed knee of curve
 - Increase number of page frames available to process
 - If page fault rate decreases below foot of knee of curve
 - Decrease number of page frames available to process



Page Size Considerations

■ Small pages

- Large page tables
- Minimizes internal fragmentation
- Good for locality of reference (~256)
- Page tables are larger
- Disk-seek time dominates transfer time (It takes the same time to read a large page as a small page)

■ Large pages

- Significant amounts of a page may not be referenced
- Enables more data per seek

■ Real systems (can be reconfigured)

- Windows: default 8KB
- Linux: default 4 KB



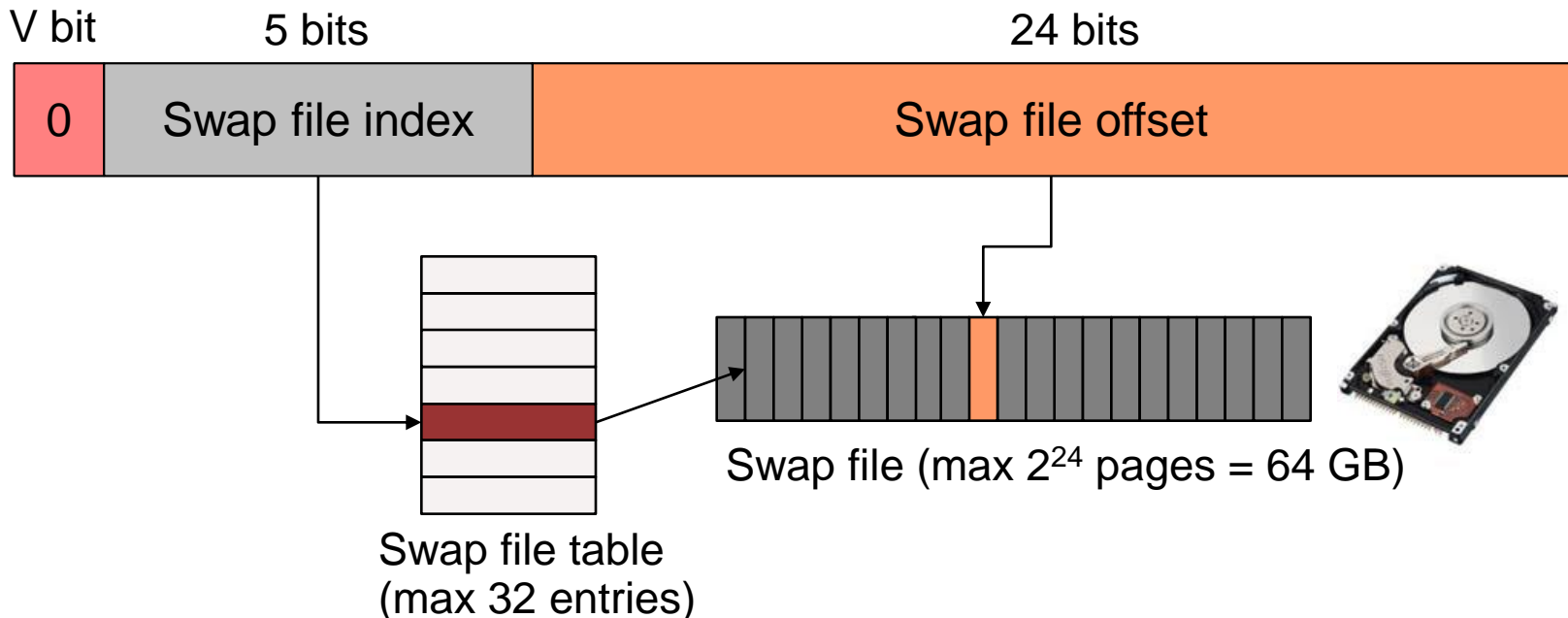
Page Replacement – Swap Files

- What happens to the page that we choose to evict?
 - Depends on what kind of page it is and what state it's in!
- OS maintains one or more swap files or partitions on disk
 - Special data format for storing pages that have been swapped out



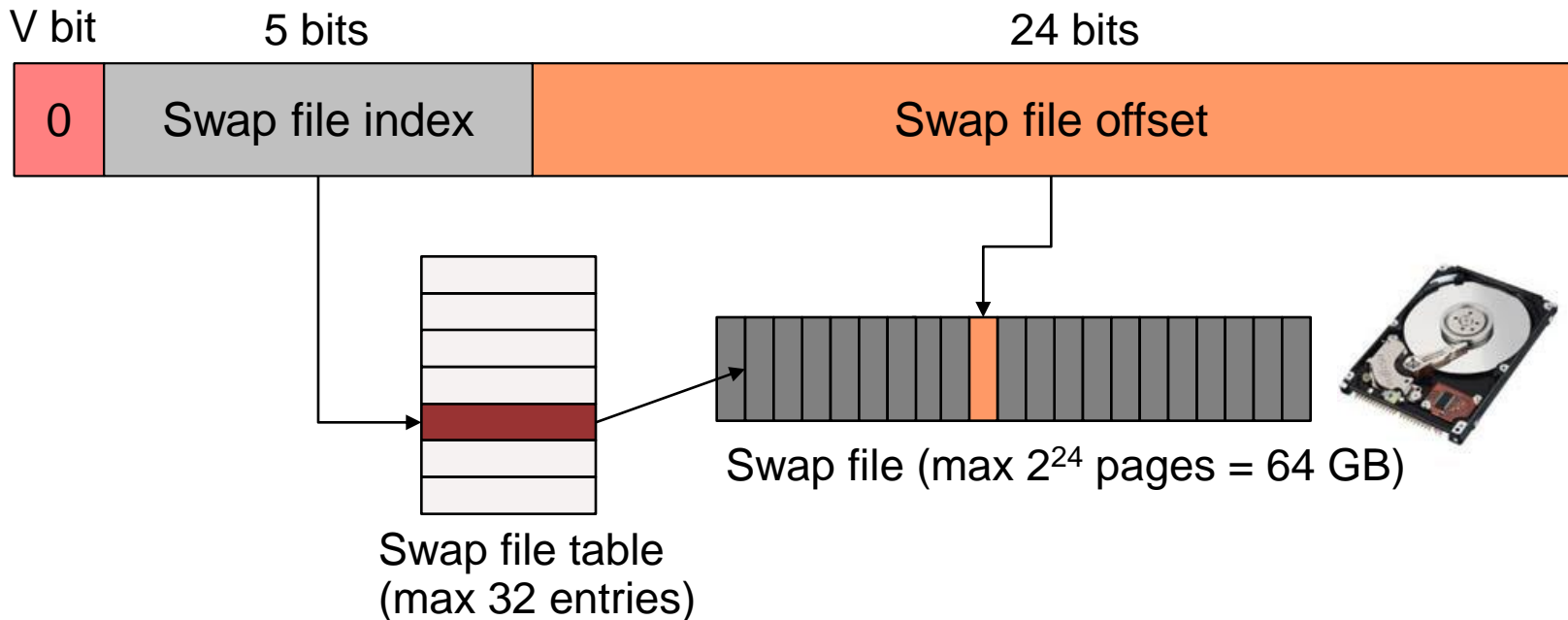
[Swap Files]

- How do we keep track of where things are on disk?
 - Recall PTE format
 - When V bit is 0, can recycle the PFN field to remember something about the page.



[Swap Files]

- But ... not all pages are swapped in from swap files!
 - e.g., what about executables?



[Page Eviction]

- How we evict a page depends on its type?
- Code page
 - Just remove it from memory – can recover it from the executable file on disk!



[Page Eviction]

- How we evict a page depends on its type?
- Unmodified (clean) data page
 - Previously swapped to disk
 - Just remove it from memory
 - Assuming that page's backing store on disk has not been overwritten
 - Never been swapped to disk
 - Allocate new swap space and write the page to it
 - Exception: unmodified zero page
 - No need to write out to swap at all!



[Page Eviction]

- How we evict a page depends on its type?
- Modified (dirty) data page:
 - Previously swapped to disk
 - Write page out to the swap space
 - Never been swapped to disk
 - Allocate new swap space and write the page to it

