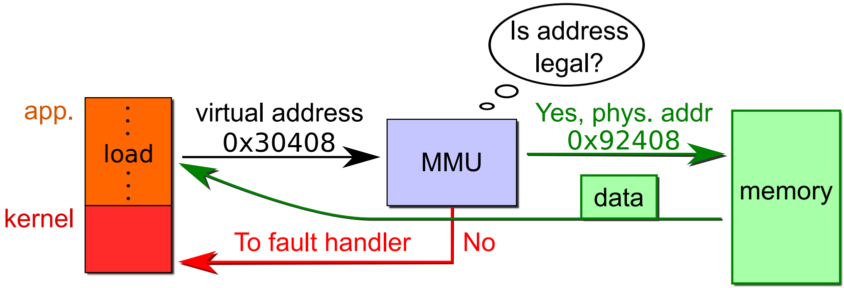
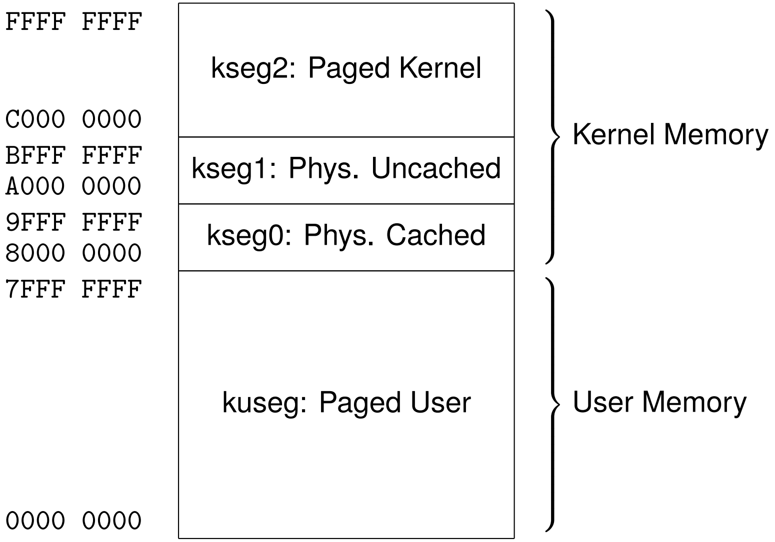
Virtual memory goals



- Give each program its own “virtual” address space
  - At run time, Memory-Management Unit relocates each load, store to actual memory… App doesn’t see physical memory
- Enforces protection
- Allows programs to see more memory than exists

MIPS Memory Layout



Heap allocators

- Simplify memory management for application developers
- Applications use `malloc()` and `free()` to manage memory
  - Backs `new`/`delete` in C++ or similar mechanisms
- Uses system calls `brk()`, `sbrk()`, or `mmap()` to ask the kernel to map pages into the application address space

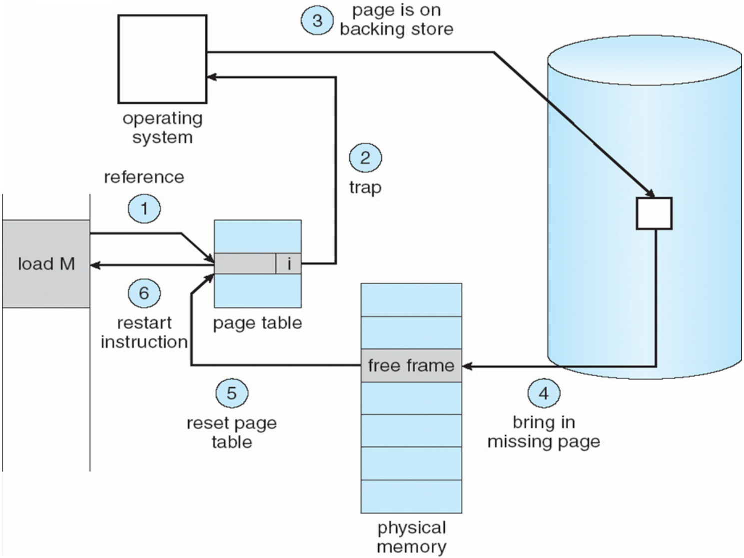
Virtual memory everywhere

- Used throughout applications and the operating system
- Virtual memory system calls
  - Map and unmap memory in userspace
  - Mapping files as memory
  - Control page permissions and caching behavior
- Larger applications than physical memory
  - Large databases or accessing huge files as memory
  - Operating system places infrequently used pages on-disk
- Security
  - Applications use virtual memory to protect themselves from attack
  - E.g. making Write/eXecute pages exclusive

Outline

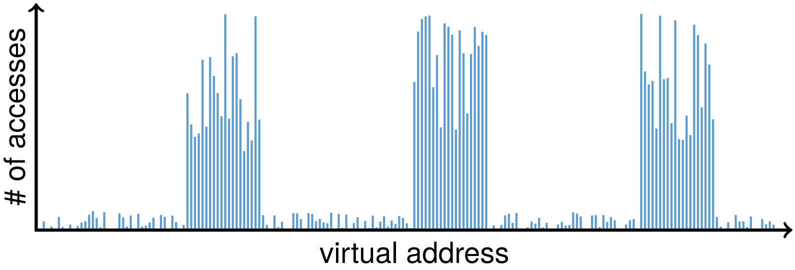
1. Paging
2. Eviction policies
3. Thrashing

Paging



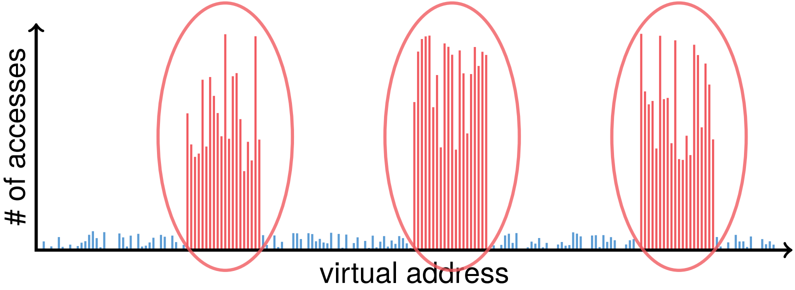
- Use disk to simulate larger virtual than physical mem

Working set model



- Disk much, much slower than memory
  - Goal: run at memory speed, not disk speed
- 80/20 rule: 20% of memory gets 80% of memory accesses
  - Keep the hot 20% in memory
  - Keep the cold 80% on disk

Working set model



- Disk much, much slower than memory
  - Goal: run at memory speed, not disk speed
- 80/20 rule: 20% of memory gets 80% of memory accesses
  - → Keep the hot 20% in memory
  - Keep the cold 80% on disk

Working set model

- Disk much, much slower than memory
  - Goal: run at memory speed, not disk speed
- 80/20 rule: 20% of memory gets 80% of memory accesses
  - Keep the hot 20% in memory

⟶ Keep the cold 80% on disk

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 from disk?
  - 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 with information about page fault
  - Faulting virtual address (In `%c0_vaddr` reg on MIPS)
  - Address of instruction that caused fault (`%c0_epc` reg)
  - 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

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, anonymously mmapped memory
  - Zeroing them only on demand is slower
  - Hence, 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
  - In old days: Physical address A conflicts with $kC + A$ (where k is any integer, C is cache size)
  - Applications can conflict with each other or themselves
  - Scientific applications benefit if consecutive virtual pages do not conflict in the cache
  - Many other applications do better with random mapping
  - These days: CPUs more sophisticated than $kC + A$

Superpages

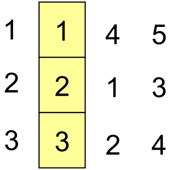
- How should OS make use of "large" mappings
  - x86 has 2/4MB pages that might be useful
  - Alpha has even more choices: 8KB, 64KB, 512KB, 4MB
- Sometimes more pages in L2 cache than TLB entries
  - Don't want costly TLB misses going to main memory
- Or have two-level TLBs
  - Want to maximize hit rate in faster L1 TLB
- OS can transparently support superpages [Navarro]
  - "Reserve" appropriate physical pages if possible
  - Promote contiguous pages to superpages
  - Does complicate evicting (esp. dirty pages) – demote

Outline

1. Paging
2. Eviction policies
3. Thrashing

Straw man: FIFO eviction

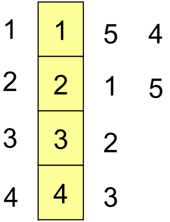
- Evict oldest fetched page in system
- Example—reference string 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 3 physical pages: 9 page faults



9 page faults

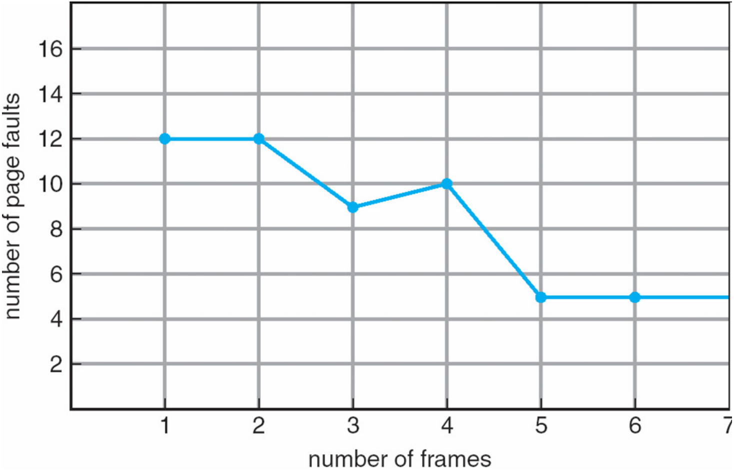
Straw man: FIFO eviction

- 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**



10 page faults

Belady's Anomaly



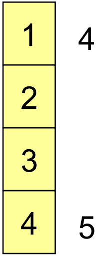
- More physical memory doesn't always mean fewer faults

Optimal page replacement

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

Optimal page replacement

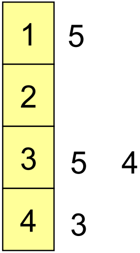
- What is optimal (if you knew the future)?
  - Replace page that will not be used for longest period of time
- Example—reference string 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- With 4 physical pages:



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

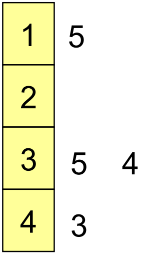


- Problem 1: Can be pessimal – example?

- Problem 2: How to implement?

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?

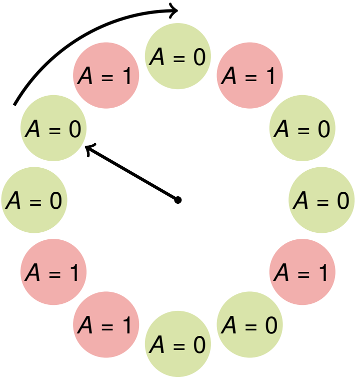
Straw man LRU implementations

- 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

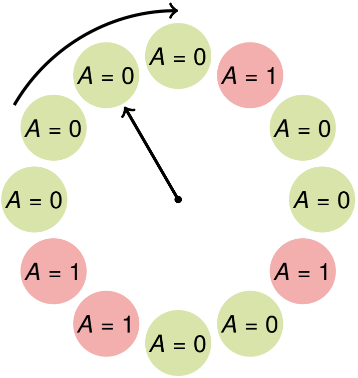
- Use accessed bit supported by most hardware
  - E.g., Pentium will write 1 to A bit in PTE on first access
  - Software managed TLBs like MIPS can do the same

- 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



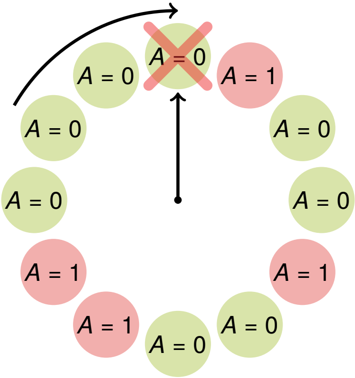
Clock algorithm

- 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

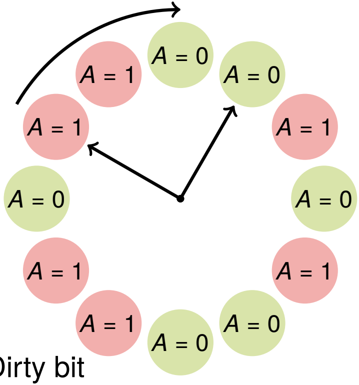


Clock algorithm

- 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

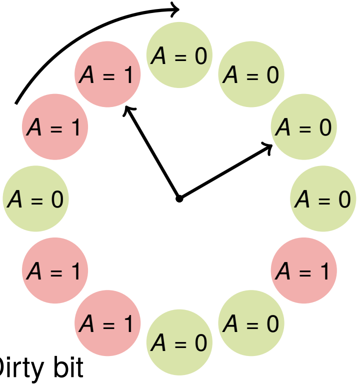


Clock algorithm (continued)



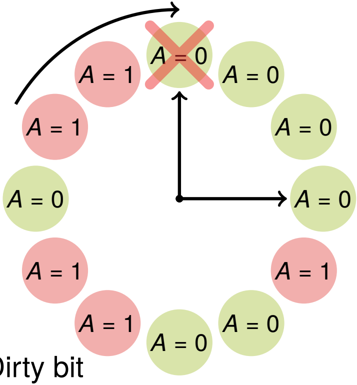
- Large memory may be a problem
  - Most pages referenced in long interval
- Add a second clock hand
  - Two hands move in lockstep
  - Leading hand clears A bits
  - Trailing hand evicts pages with A=0
- Can also take advantage of hardware Dirty bit
  - Each page can be (Unaccessed, Clean), (Unaccessed, 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*

Clock algorithm (continued)



- Large memory may be a problem
  - Most pages referenced 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*

Clock algorithm (continued)

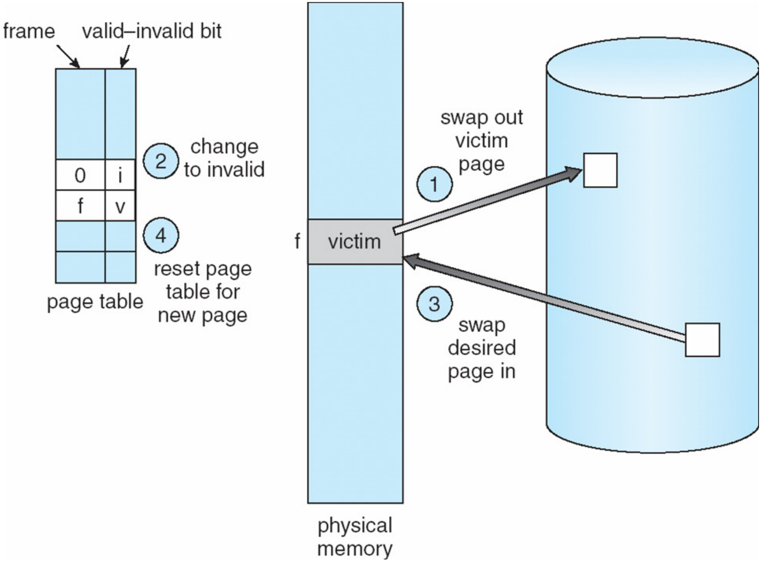


- Large memory may be a problem
  - Most pages referenced in long interval
- Add a second clock hand
  - Two hands move in lockstep
  - Leading hand clears A bits
  - Trailing hand evicts pages with A=0
- Can also take advantage of hardware Dirty bit
  - Each page can be (Unaccessed, Clean), (Unaccessed, 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

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 P_1 needs 20% of memory and P_2 needs 70%:

  - Doesn't protect you from memory pigs
    (imagine P_2 keeps looping through array that is size of mem)
- Local allocation isolates processes (or users)
  - Separately determine how much memory each process should have
  - Then use LRU/clock/etc. to determine which pages to evict within each process

Outline

1. Paging
2. Eviction policies
3. Thrashing

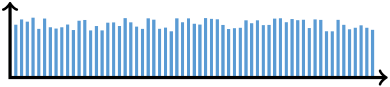
Thrashing

Thrashing is when an application is in a constantly swapping pages in and out preventing the application from making forward progress at any reasonable rate.

- Processes require more memory than system 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 got: memory with access time of disk

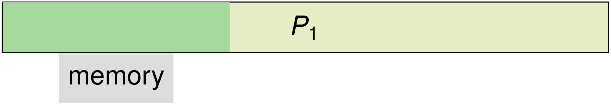
Reasons for thrashing

- Access pattern has no temporal locality (past ≠ future)

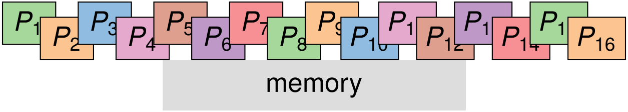


(80/20 rule has broken down)

- Hot memory does not fit in physical memory

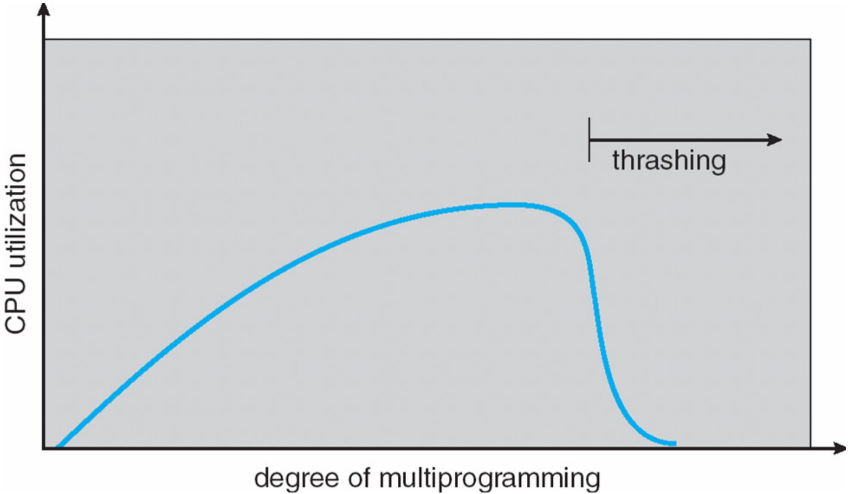


- Each process fits individually, but too many for system



  - At least this case is possible to address

Multiprogramming & Thrashing

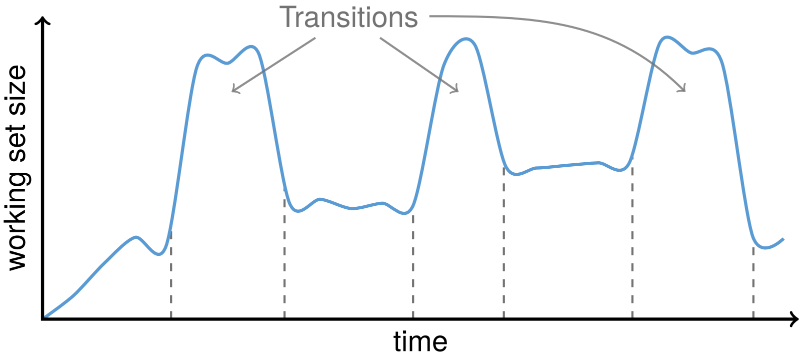


- Must shed load when thrashing

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 the 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 phase transitions

Calculating the working set

- Working set: all pages process 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 [recall from lecture 4]
  - 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)