Paging algorithms

CS 241

February 10, 2012

Copyright ©: University of Illinois CS 241 Staff

Announcements

- MP2 due Tuesday
- Fabulous Prizes Wednesday!

Paging

- On heavily-loaded systems, memory can fill up
- Need to make room for newly-accessed pages
  - Heuristic: try to move “inactive” pages out to disk
    - What constitutes an “inactive” page?
- Paging
  - Refers to moving individual pages out to disk, and back
  - We often use the terms “paging” and “swapping” interchangeably
  - Different from context switching
    - Background processes often have their pages remain resident in memory
    - Paging could occur even with only one process running

Basic Page Replacement

- Find the location of the desired page on disk
- Find a free frame
  - If there is a free frame, use it
  - If there is no free frame, use a page replacement algorithm to select a *victim* frame
- Read the desired page into the (newly) free frame. Update the page and frame tables.

- Note: can also evict in advance
  - OS keeps pool of “free pages” around, even when memory is tight
  - Makes allocating a new page fast
  - The process of evicting pages to disk is then performed in the background

Exploiting Locality

- Exploiting locality
  - Temporal locality: Memory accessed recently tends to be accessed again soon
  - Spatial locality: Memory locations near recently-accessed memory is likely to be referenced soon
- Locality helps to reduce the frequency of paging
  - Once something is in memory, it should be used many times
- This depends on many things:
  - The amount of locality and reference patterns in a program
  - The *page replacement algorithm*
  - The amount of physical memory and the *application footprint*

Fundamental technique: caching

- A **cache** keeps a subset of a data set in a more accessible but space-limited location

- Caches are **everywhere** in systems
  - Such as…?

Fundamental technique: caching

- A **cache** keeps a subset of a data set in a more accessible but space-limited location

- Caches are **everywhere** in systems
  - **Registers** are a cache for **L1 cache** which is a cache for **L2 cache** which is a cache for **memory** which is a cache for **disk** which might be a cache for a **remote file server**
  - Web proxy servers make downloads faster & cheaper
  - Web browser stores downloaded files
  - Local DNS servers remember recently-resolved DNS names
  - Google servers remember your searches

- Key goal: **minimize cache miss rate**
  - = minimize page fault rate (in context of paging)
  - Requires a good algorithm

Evicting the Best Page

- Goal of the page replacement algorithm:
  - Reduce **page fault rate** by selecting the best page to evict
- The "best" pages are those that will never be used again
  - However, it's impossible to know in general whether a page will be touched
  - If you have information on future access patterns, it is possible to *prove* that evicting those pages that will be used the *furthest in the future* will *minimize* the page fault rate
- What is the best algorithm for deciding the order to evict pages?
  - Much attention has been paid to this problem.
  - Used to be a very hot research topic.
  - These days, widely considered solved (at least, solved well enough)

Algorithm: OPT (a.k.a. MIN)

- Evict page that won't be used for the longest time in the future
  - Of course, this requires that we can foresee the future...
  - So OPT cannot be implemented!

- This algorithm has the provably optimal performance
  - Hence the name “OPT”

- OPT is useful as a “yardstick” to compare the performance of other (implementable) algorithms against

The Optimal Page Replacement Algorithm

- Idea:
  - Select the page that will not be needed for the longest time in the future

Time		0	1	2	3	4	5	6	7	8	9	10
Requests			c	a	d	b	e	b	a	b	c	d
Page Frames	0	a	a	a	a	a						
	1	b	b	b	b	b						
	2	c	c	c	c	c						
	3	d	d	d	d	d						
Page faults							x					

Copyright ©: University of Illinois CS 241 Staff

The Optimal Page Replacement Algorithm

- Idea:
  - Select the page that will not be needed for the longest time in the future

Time		0	1	2	3	4	5	6	7	8	9	10
Requests			c	a	d	b	e	b	a	b	c	d
Page Frames	0	a	a	a	a	a	a	a	a	a	a	
	1	b	b	b	b	b	b	b	b	b	b	
	2	c	c	c	c	c	c	c	c	c	c	
	3	d	d	d	d	d	e	e	e	e	e	
Page faults							x					x

Copyright ©: University of Illinois CS 241 Staff

The Optimal Page Replacement Algorithm

- Idea:
  - Select the page that will not be needed for the longest time in the future

Time		0	1	2	3	4	5	6	7	8	9	10
Requests			c	a	d	b	e	b	a	b	c	d
Page Frames	0	a	a	a	a	a	a	a	a	a	a	a
	1	b	b	b	b	b	b	b	b	b	b	b
	2	c	c	c	c	c	c	c	c	c	c	c
	3	d	d	d	d	d	e	e	e	e	e	d
Page faults							x					x

Copyright ©: University of Illinois CS 241 Staff

Algorithms: Random and FIFO

- Random: Throw out a random page
  - Obviously not the best scheme
  - Although very easy to implement!
- FIFO: Throw out pages in the order that they were allocated
  - Maintain a list of allocated pages
  - When the length of the list grows to cover all of physical memory, pop first page off list and allocate it
- Why might FIFO be good?
- Why might FIFO not be so good?

Algorithms: Random and FIFO

- FIFO: Throw out pages in the order that they were allocated
  - Maintain a list of allocated pages
  - When the length of the list grows to cover all of physical memory, pop first page off list and allocate it
- Why might FIFO be good?
  - Maybe the page allocated very long ago isn't used anymore
- Why might FIFO not be so good?
  - Doesn't consider locality of reference!
  - Suffers from Belady's anomaly: Performance of an application might get *worse* as the size of physical memory *increases!!!*

Belady's Anomaly

time →

Access pattern	0	1	2	3	0	1	4	0	1	2	3	4
Physical memory (3 page frames)	**0**	0	0	1	2	3	0	0	0	1	4	4
		1	1	2	3	0	1	1	1	4	2	2
			2	**3**	**0**	**1**	**4**	4	4	**2**	**3**	3

9 page faults!

time →

Access pattern	0	1	2	3	0	1	4	0	1	2	3	4
Physical memory (4 page frames)	**0**	0	0	0	0	0	1	2	3	4	0	1
		1	1	1	1	1	2	3	4	0	1	2
			2	2	2	2	3	4	0	1	2	3
				3	3	3	**4**	**0**	**1**	**2**	**3**	**4**

10 page faults!

Algorithm: Least Recently Used (LRU)

- Evict the page that was used the longest time ago
  - Keep track of when pages are referenced to make a better decision
  - Use past behavior to predict future behavior
    - LRU uses past information, while OPT uses future information
  - When does LRU work well, and when does it not?
- Implementation
  - Every time a page is accessed, record a timestamp of the access time
  - When choosing a page to evict, scan over all pages and throw out page with oldest timestamp
- Problems with this implementation?

Algorithm: Least Recently Used (LRU)

- Evict the page that was used the longest time ago
  - Keep track of when pages are referenced to make a better decision
  - Use past behavior to predict future behavior
    - LRU uses past information, while OPT uses future information
  - When does LRU work well, and when does it not?
- Implementation
  - Every time a page is accessed, record a timestamp of the access time
  - When choosing a page to evict, scan over all pages and throw out page with oldest timestamp
- Problems with this implementation?
  - 32-bit timestamp would double size of PTE
  - Scanning all of the PTEs for lowest timestamp: slow

Least Recently Used (LRU)

- Keep track of when a page is used
- Replace the page that has been used least recently

Time		**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Requests			**c**	**a**	**d**	**b**	**e**	**b**	**a**	**b**	**c**	**d**
Page Frames	**0**	**a**										
	1	**b**										
	2	**c**										
	3	**d**										
Page faults												

Copyright ©: University of Illinois CS 241 Staff

Least Recently Used (LRU)

- Keep track of when a page is used
- Replace the page that has been used least recently (farthest in the past)

Time		0	1	2	3	4	5	6	7	8	9	10
Requests			c	a	d	b	e	b	a	b	c	d
Page Frames	0	a	a	a	a	a						
	1	b	b	b	b	b						
	2	c	c	c	c	c						
	3	d	d	d	d	d						
Page faults							x					

Copyright ©: University of Illinois CS 241 Staff

Least Recently Used (LRU)

- Keep track of when a page is used
- Replace the page that has been used least recently (farthest in the past)

Time		0	1	2	3	4	5	6	7	8	9	10
Requests			c	a	d	b	e	b	a	b	c	d
Page Frames	0	a	a	a	a	a	a	a	a	a		
	1	b	b	b	b	b	b	b	b	b		
	2	c	c	c	c	c	e	e	e	e		
	3	d	d	d	d	d	d	d	d	d		
Page faults							x				x	

Copyright ©: University of Illinois CS 241 Staff

Least Recently Used (LRU)

- Keep track of when a page is used
- Replace the page that has been used least recently (<u>farthest in the past</u>)

Time		0	1	2	3	4	5	6	7	8	9	10
Requests			c	a	d	b	e	b	a	b	c	d
Page Frames	0	a	a	a	a	a	a	a	a	a	a	
	1	b	b	b	b	b	b	b	b	b	b	
	2	c	c	c	c	c	e	e	e	e	e	
	3	d	d	d	d	d	d	d	d	d	c	
Page faults							x				x	x

Copyright ©: University of Illinois CS 241 Staff

Least Recently Used (LRU)

- Keep track of when a page is used
- Replace the page that has been used least recently (<u>farthest in the past</u>)

Time		0	1	2	3	4	5	6	7	8	9	10
Requests			c	a	d	b	e	b	a	b	c	d
Page Frames	0	a	a	a	a	a	a	a	a	a	a	a
	1	b	b	b	b	b	b	b	b	b	b	b
	2	c	c	c	c	c	e	e	e	e	e	d
	3	d	d	d	d	d	d	d	d	d	c	c
Page faults							x				x	x

Copyright ©: University of Illinois CS 241 Staff

Least Recently Used

- 3 frames of physical memory
- Run this for a long time with LRU page replacement:

```
while true
    for (i = 0; i < 4; i++)
        read from page i
```

- Q1: What fraction of page accesses are faults?
  - ○ None or almost none
  - ○ About 1 in 4
  - ○ About 2 in 4
  - ○ About 3 in 4
  - ○ All or almost all
- Q2: How well does OPT do?

Least Recently Used

- 3 frames of physical memory
- Run this for a long time with LRU page replacement:

```
while true
    for (i = 0; i < 4; i++)
        read from page i
```

- Q1: What fraction of page accesses are faults?
  - ○ None or almost none
  - ○ About 1 in 4
  - ○ About 2 in 4
  - ○ About 3 in 4
  - ○ **All or almost all –** least recently used is always next to be used!
- Q2: How well does OPT do?

Least Recently Used Issues

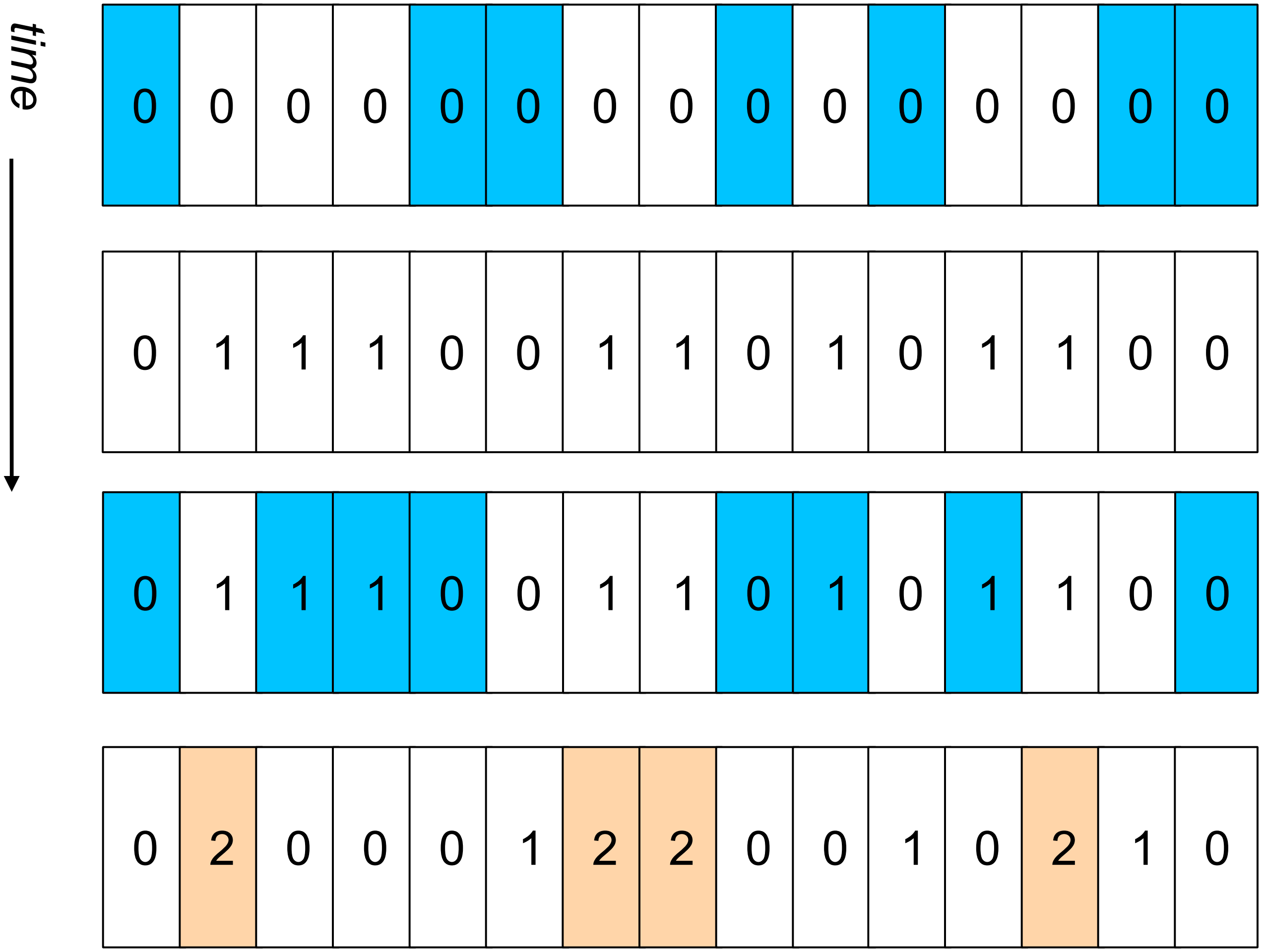
- Not optimal
- Does not suffer from Belady's anomaly
- Implementation
  - Use time of last reference
    - Update every time page accessed (use system clock)
    - Page replacement - search for smallest time
  - Use a stack
    - On page access : remove from stack, push on top
    - Victim selection: select page at bottom of stack
- Both approaches require large processing overhead, more space, and hardware support.

Copyright ©: University of Illinois CS 241 Staff

Approximating LRU

- Use the PTE reference bit and a small `counter` per page
  - (Use a counter of, say, 2 or 3 bits in size, and store it in the PTE)
- Periodically (say every 100 msec), scan all physical pages in the system. For each page:
  - If not accessed recently, (PTE reference bit == 0), `counter++`
  - If accessed recently (PTE reference bit == 1), `counter = 0`
  - Clear the PTE reference bit in either case!
- Counter will contain the number of scans since the last reference to this page.
  - PTE that contains the highest counter value is the least recently used
  - So, evict the page with the highest counter

Approximate LRU Example



0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Accessed pages in blue

0	1	1	1	0	0	1	1	0	1	0	1	1	0	0

Increment counter for untouched pages

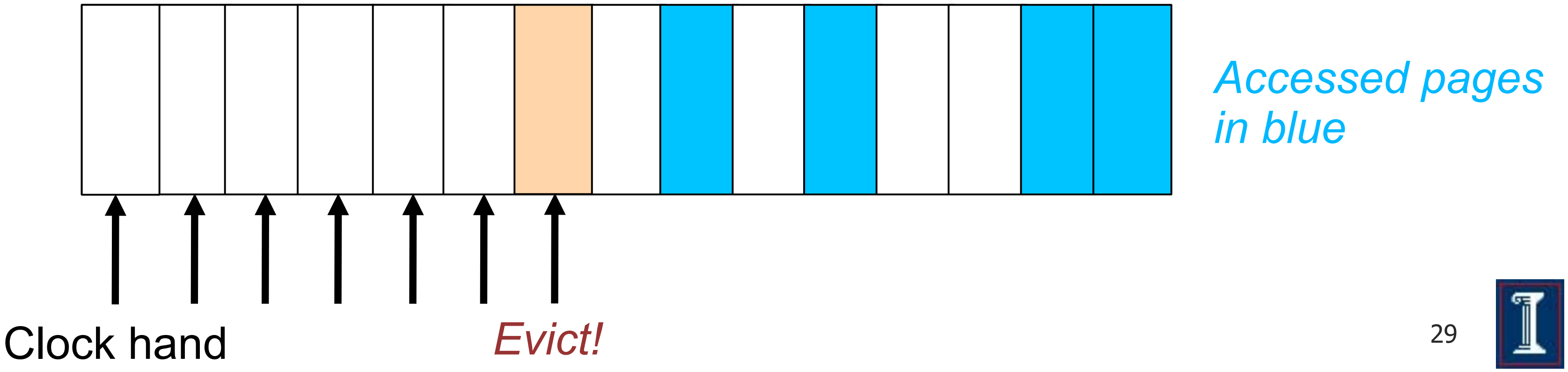
0	1	1	1	0	0	1	1	0	1	0	1	1	0	0

0	2	0	0	0	1	2	2	0	0	1	0	2	1	0

These pages have the highest counter value and can be evicted.

Algorithm: LRU Second-Chance (Clock)

- LRU requires searching for the page with the highest last-ref count
  - Can do this with a sorted list or a second pass to look for the highest value
- Simpler technique: Second-chance algorithm
  - "Clock hand" scans over all physical pages in the system
    - Clock hand loops around to beginning of memory when it gets to end
  - If PTE reference bit == 1, clear bit and advance hand to give it a second-chance
  - If PTE reference bit == 0, evict this page
    - No need for a counter in the PTE!



Algorithm: LRU Second-Chance (Clock)

- This is a lot like LRU, but operates in an iterative fashion
  - To find a page to evict, just start scanning from current clock hand position
  - What happens if all pages have ref bits set to 1?
  - What is the minimum "age" of a page that has the ref bit set to 0?
- Slight variant -- "nth chance clock"
  - Only evict page if hand has swept by N times
  - Increment per-page counter each time hand passes and ref bit is 0
  - Evict a page if counter ≥ N
  - Counter cleared to 0 each time page is used

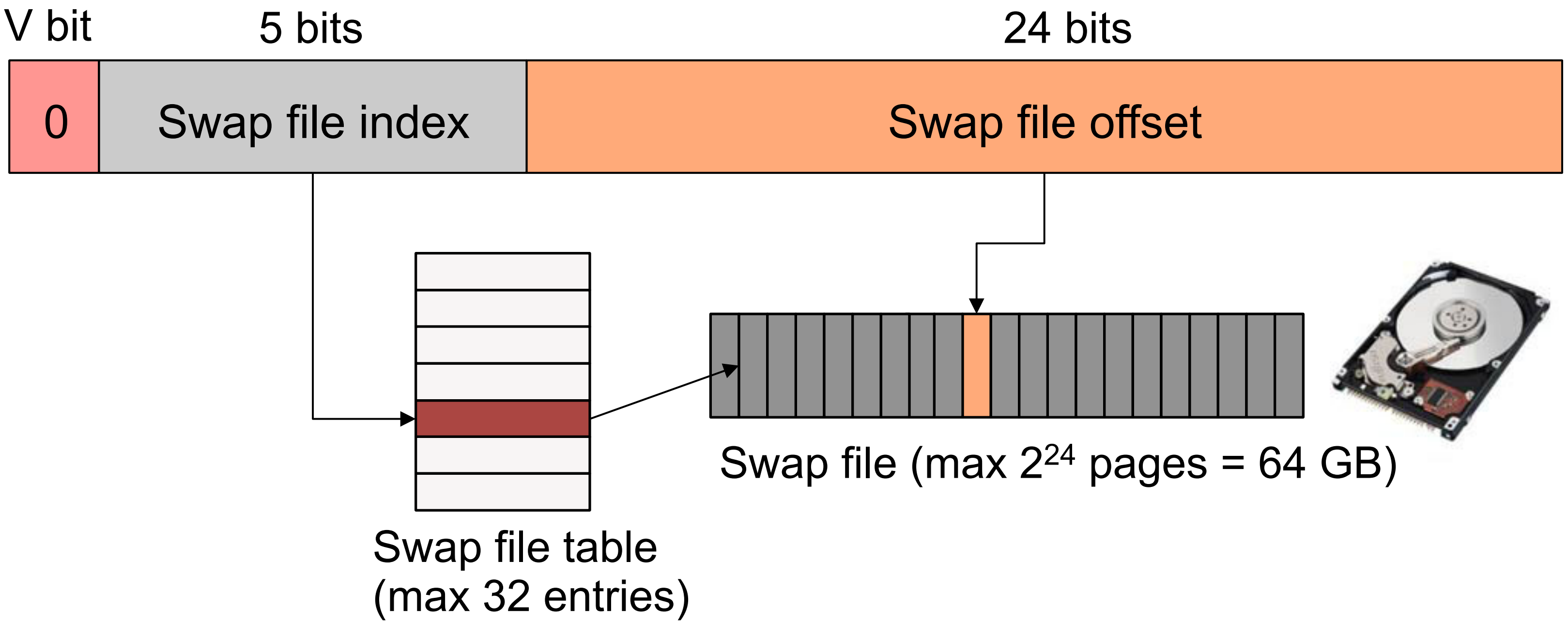
Swap Files

- 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.



- 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!
- Unmodified (*clean*) data page:
  - If the page has previously been swapped to disk, just remove it from memory
    - Assuming that page's backing store on disk has not been overwritten
  - If the page has 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!
- Modified (*dirty*) data page:
  - If the page has previously been swapped to disk, write page out to the swap space
  - If the page has never been swapped to disk, allocate new swap space and write the page to it

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?
- 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 entire system!
  - One process that does a lot of paging will induce more disk I/O

Working Set

- A process's *working set* is the set of pages that it currently "needs"
- Definition:
  - WS(P, t, w) = the set of pages that process P accessed in the time interval [t-w, t]
  - "w" is usually counted in terms of number of page references
    - A page is in WS if it was referenced in the last w page references
- 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
- Basic idea: Give process enough memory for its working set
  - If WS is larger than physical memory allocated to process, it will tend to swap
  - If WS is smaller than memory allocated to process, it's wasteful
  - This amount of memory grows and shrinks over time

Estimating the Working Set

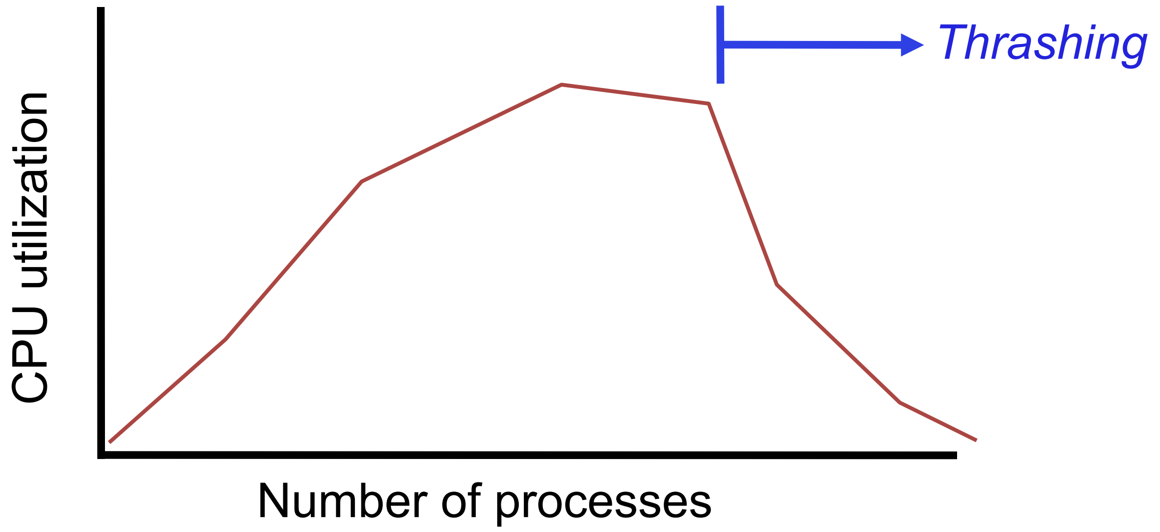
- How do we determine the working set?
- Simple approach: modified clock algorithm
  - Sweep the clock hand at fixed time intervals
  - Record how many seconds since last page reference
  - All pages referenced in last T seconds are in the working set
- Now that we know the working set, how do we allocate memory?
  - If working sets for all processes fit in physical memory, done!
  - Otherwise, reduce memory allocation of larger processes
    - Idea: Big processes will swap anyway, so let the small jobs run unencumbered
  - Very similar to shortest-job-first scheduling: give smaller processes better chance of fitting in memory
- How do we decide the working set time limit T?
  - If T is too large, very few processes will fit in memory
  - If T is too small, system will spend more time swapping
    - Which is better?

Page Fault Frequency

- Dynamically tune memory size of process based on # page faults
- Monitor page fault rate for each process (faults per sec)
- If page fault rate above threshold, give process more memory
  - Should cause process to fault less
  - Doesn't always work!
    - *Recall Belady's Anomaly*
- If page fault rate below threshold, reduce memory allocation
- What happens when **everyone’s** page fault rate is high?

Thrashing

- As system becomes more loaded, spends more of its time paging
  - Eventually, no useful work gets done!



- System is overcommitted!
  - If the system has too little memory, the page replacement algorithm doesn't matter
- Solutions?
  - Change scheduling priorities to “slow down” processes that are thrashing
  - Identify process that are hogging the system and kill them?
    - Is thrashing a problem on systems with only one user?