Memory Management – Page Replacement

CS 416: Operating Systems Design, Spring 2011

Department of Computer Science
Rutgers University

Rutgers Sakai: 01:198:416 Sp11
(https://sakai.rutgers.edu)

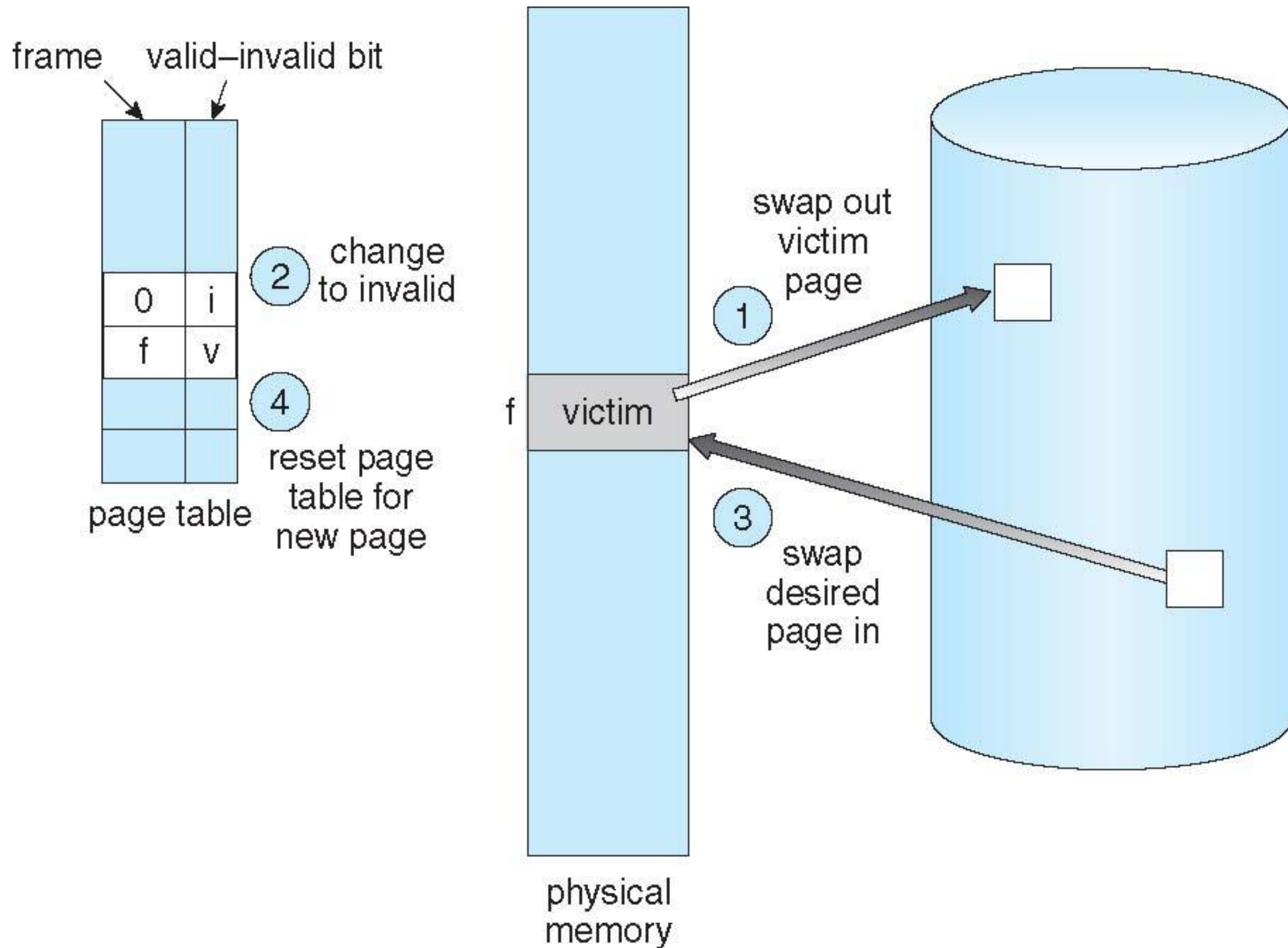
Page Replacement

- How do we decide which frames to kick out if the memory is tight?
- How do we decide how much of physical memory should be allocated to each process ?

Paging and Swapping

- To achieve good performance, the OS must kick “inactive” frames out of main memory into the disk
  - What constitutes an “inactive” page?
  - How do we choose the right set of pages to copy out to disk ?
  - How do we decide when to move back a page into memory?
- Swapping
  - Usually refers to moving the memory for an entire process onto the disk
  - This effectively puts the process to sleep until the OS decided to swap it in
- Paging
  - Refers to moving individual pages out to disk

Page Replacement



Page eviction and locality

- When do we decide to evict a page from memory?
  - Usually at the time when we are trying to allocate a frame for currently executing process
  - However, the OS keeps a pool of "free pages" around, even when memory is tight, so that allocating a new page can be done quickly.
  - Therefore, OS does this periodically ***in the background***.
- Exploiting locality: Locality helps reduce frequency of paging
  - **Temporal Locality:** Memory accessed recently tends to be accessed again
  - **Spatial Locality:** Memory locations near recently-accessed memory is likely to be referenced soon
- Frequency of paging depends on
  - The amount of locality and reference patterns in a program
  - The **page replacement** policy
  - The amount of physical memory and the application footprint

Locality Example

- Program structure

Array A[1024, 1024] of integer

Each row is stored in one page

One frame

Program 1

```
for j := 1 to 1024 do
for i := 1 to 1024 do
     A[i,j] := 0;
```

1024 x 1024 page faults ! – Poor Locality

Program 2

```
for i := 1 to 1024 do
for j := 1 to 1024 do
     A[i,j] := 0;
```

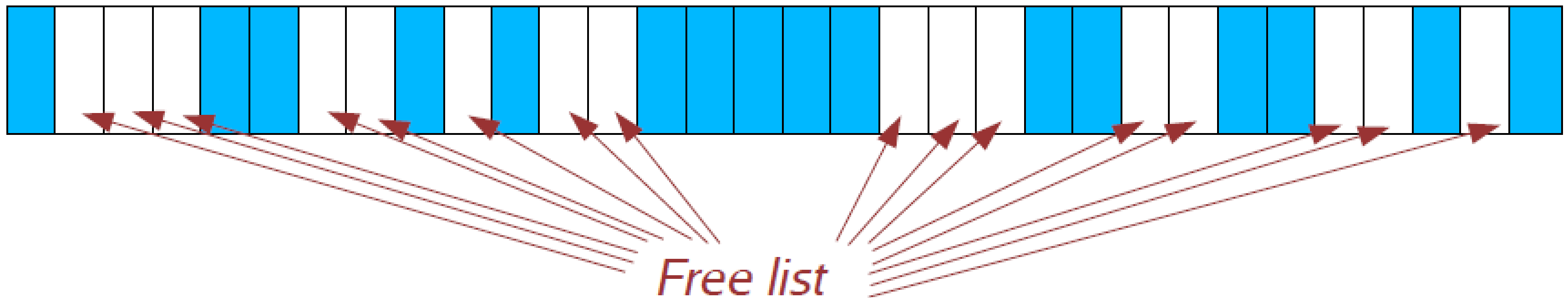
1024 page faults !

Evicting the best pages

- 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 happened to have information on future access patterns, you can 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?

Page Replacement Basics

- Most page replacement algorithms operate on some data structure that represents physical memory:



  - Might consist of a bitmap, one bit per physical page
  - Might be more involved, e.g., a reference count for each page
  - Free list consists of pages that are unallocated

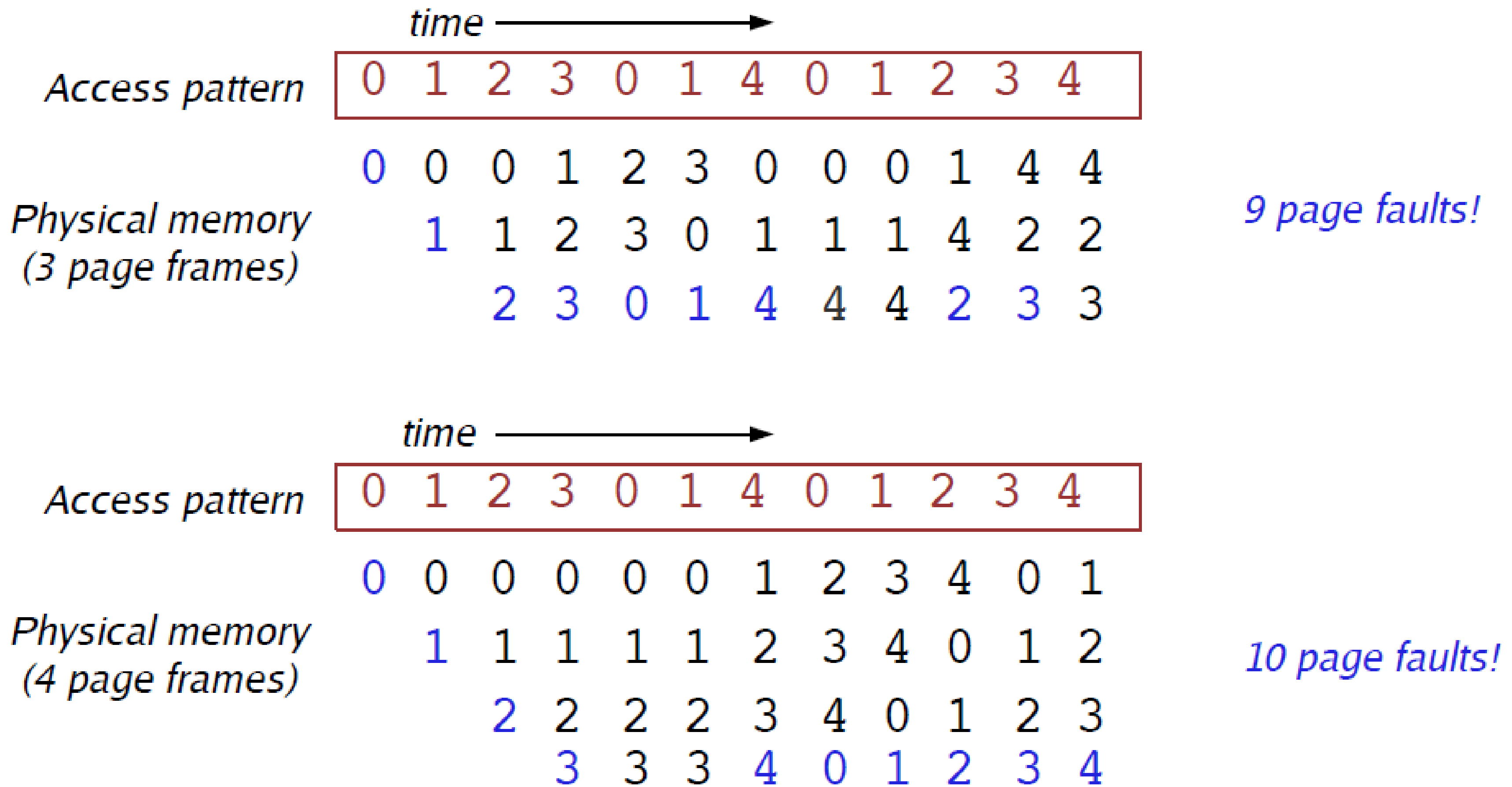
Algorithm #1: OPT (a.k.a MIN)

- Evict page that won't be used for the longest time in the future
  - Of course, this requires that we can see into the future...
  - So OPT cannot be implemented!
- This algorithm has the provably optimal performance
  - Hence the name "OPT"
  - Also called "MIN" (for "minimal")
- OPT is useful as a "yardstick" to compare the performance of other (implementable) algorithms against

Algorithms #2 and 3: 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?
  - Maybe the page allocated very long ago isn't being used anymore
- Why might FIFO not be so good?
  - For Example, a variable initialized early on in the code gets referenced later.
  - Suffers from ***Belady's Anomaly****:* Performance of an application might get ***worse*** as the size of physical memory ***increases***!!!

Belady's Anamoly



In a system with smaller memory, whatever that was recently accessed kept coming in at the end of the queue since it had higher chances of being chucked out due to small size.

Algorithm #4: Least Recently Used (LRU)

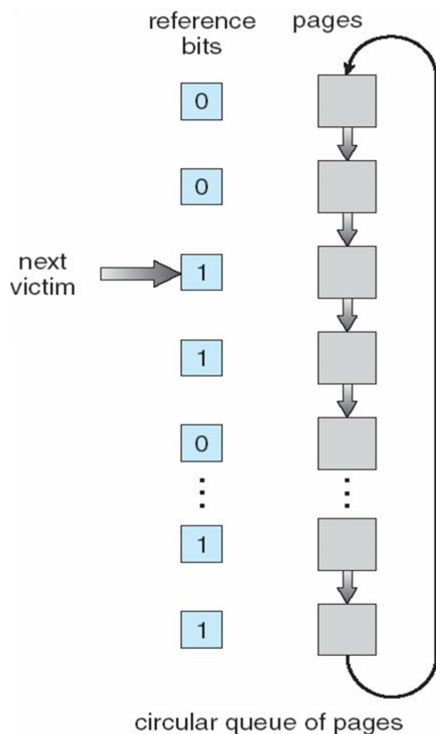
- Evict the page that was used the **longest time ago**
  - Keep track of when pages are referenced to make a better decision
  - Use past behavior to predict future behavior
    - *LRU uses past information, while MIN uses future information*
- Implementation
  - Every time a page is accessed, record 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 for each page would double the size of every PTE
  - Scanning all of the PTEs for the lowest timestamp would be slow
  - So, we need an approximation!
- **Why doesn't LRU suffer from Belady's Anamoly ?**

Approximating LRU : Counter

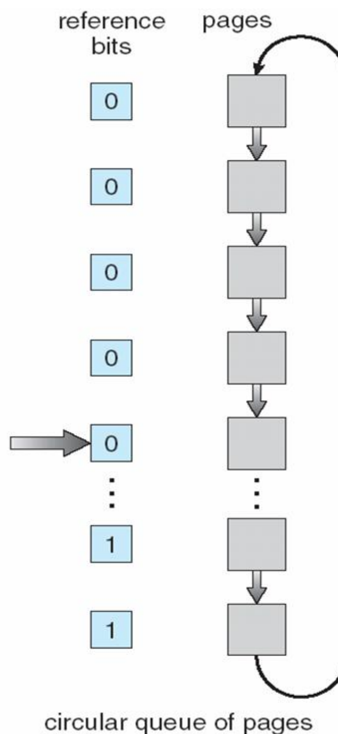
- Have a reference bit and software counter for each page frame
- At each clock interrupt, the OS adds the reference bit of each frame to its counter and then clears the reference bit
- When need to evict a page, choose frame with lowest counter
- What's the problem?
  - Doesn't forget anything, no sense of time – hard to evict a page that was reference a lot sometime in the past but is no longer relevant to the computation
  - Updating counters is expensive, especially since memory is getting rather large these days
- Can be improved with an ***aging scheme***: counters are shifted right before adding the reference bit and the reference bit is added to the leftmost bit (rather than to the rightmost one)

Approximating LRU : Using Clock (Second Chance)

- "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**
- If PTE reference bit == 0, **evict this page**
  - No need for a counter in the PTE!



(a)



(b)

What is the problem with this scheme ?

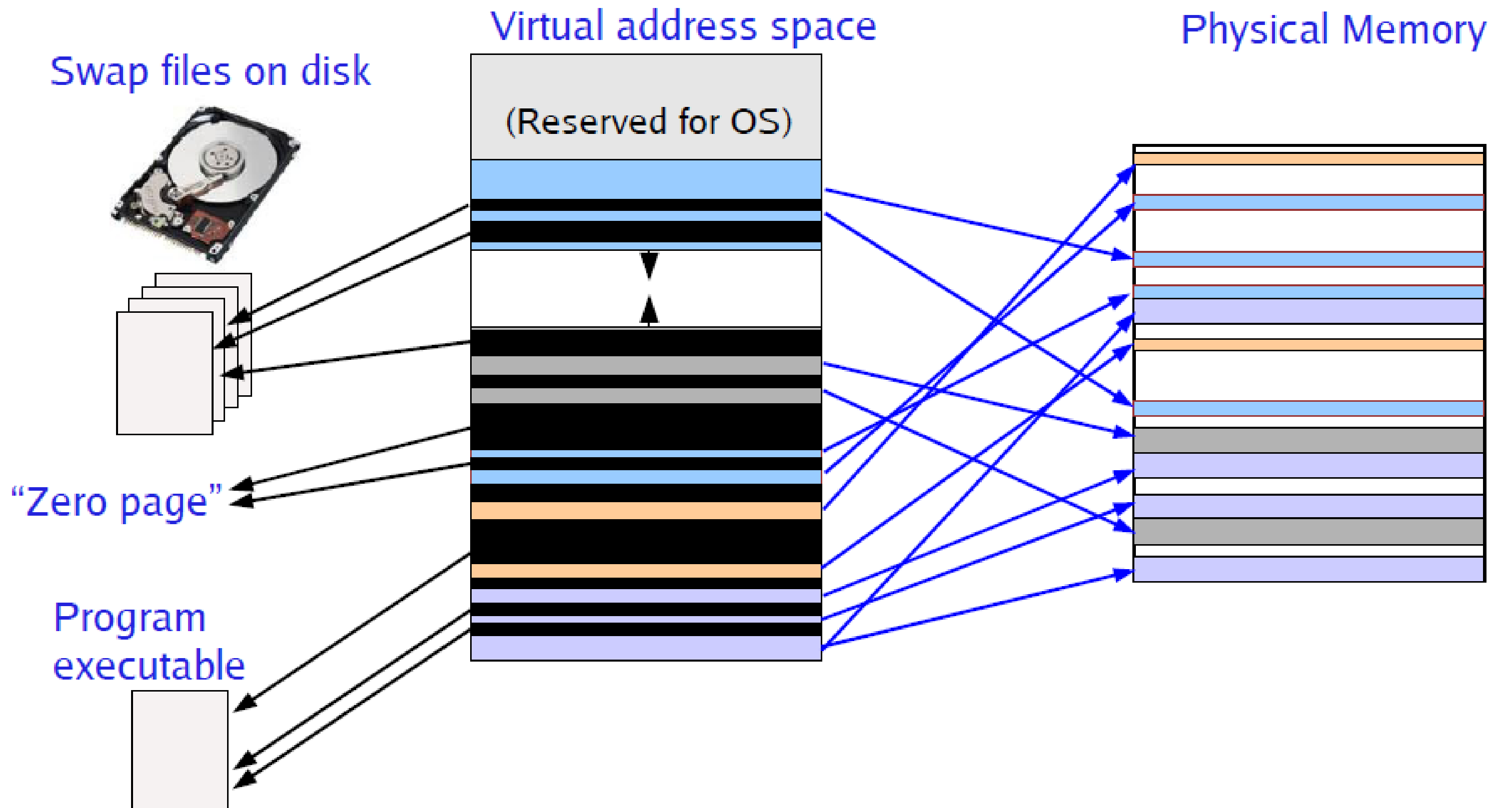
- Don't have a count of number of times any page was accessed recently.

Approximating LRU : Nth Chance (Counter + Clock)

- 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)
- On Page fault, Advance clock hand.
  - If the page has not been accessed (PTE reference bit == 0), **increment the counter**
  - If the page has been accessed (reference bit == 1), **set counter to zero**
    - **(WHY? Is this necessary ?)**
  - **Clear** the PTE reference bit in either case!
- Counter will contain the number of scans since the last reference to this page.
  - If counter < N, go on. Otherwise this is our Victim
- **What is the problem if N is too large ?**

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



Page Eviction

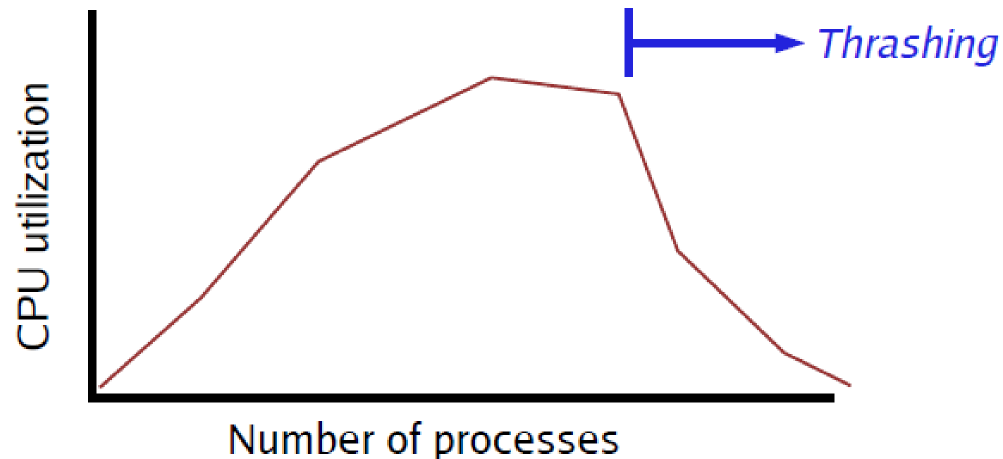
- How we evict a page depends on its type.
- Code page:
  - Just chuck it from memory – can recover it from the executable file on disk!
- Unmodified (*clean*) data page:
  - If the page has previously been swapped to disk, just chuck it from memory
    - *Assuming that page's backing store on disk has not been overwritten*
  - If the page has never been swapped to disk, allocate new swap space and write the page to it (This is just an optimization since swapping the page in is faster from swap space)
  - 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?
  - When we evict a page, which process should we evict it from?
  - How do we ensure fairness?
  - How do we avoid one process hogging the entire memory of the system?
- Fixed-space algorithms
  - Per-process limit on the physical memory usage of each process
  - When a process reaches its limit, it evicts pages *from itself*
- Variable-space algorithms
  - Physical size of processes can grow and shrink over time
  - Allow processes to evict pages from other processes
- One process paging can impact performance of entire system!
  - One process that does a lot of paging will induce more disk I/O

Thrashing

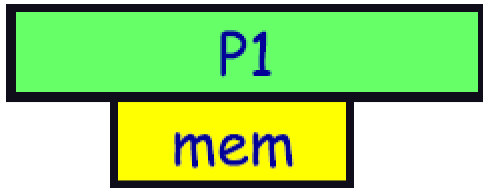
- As system becomes more loaded, spends more of its time paging
  - Eventually, no useful work gets done!
- System is overcommitted!
  - If the system has too little memory, the page replacement algorithm doesn't matter



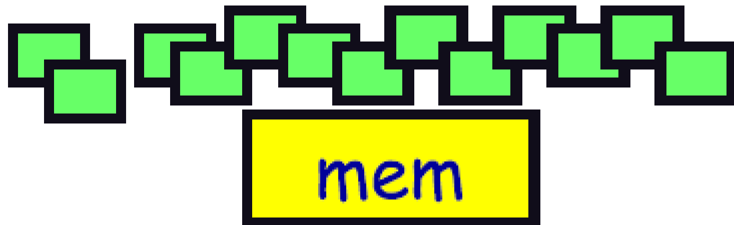
- Solutions?
  - Change scheduling priorities to "slow down" processes that are thrashing
  - Identify process that are hogging the system and kill them?

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



  - This could be solved !

Dealing with Thrashing

Approach 1: Working set

- 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

- 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 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 of a process?
- Simple approach
  - Approximate with interval timer + a reference bit
- Example: t = 10,000
  - Timer interrupts after every 5000 time units.
  - Keep in memory 2 bits for each page.
  - Whenever a timer interrupts, shift the bits to right and copy the reference bit value onto the high order bit and sets the values of all reference bits to 0.
  - If one of the bits in memory = 1 ⇒ page in working set.
- Why is this not completely accurate?
  - Not sure when exactly in the last 5000 time units was this page accessed
- Improvement = 10 bits and interrupt every 1000 time units.

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.
  - 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

Page-Fault Frequency Scheme

- Page Fault Rate = (#Page Faults)/No of Executed Instructions
- Establish "acceptable" page-fault rate
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame (or is swapped out)