Memory Management and Virtual Memory - 2

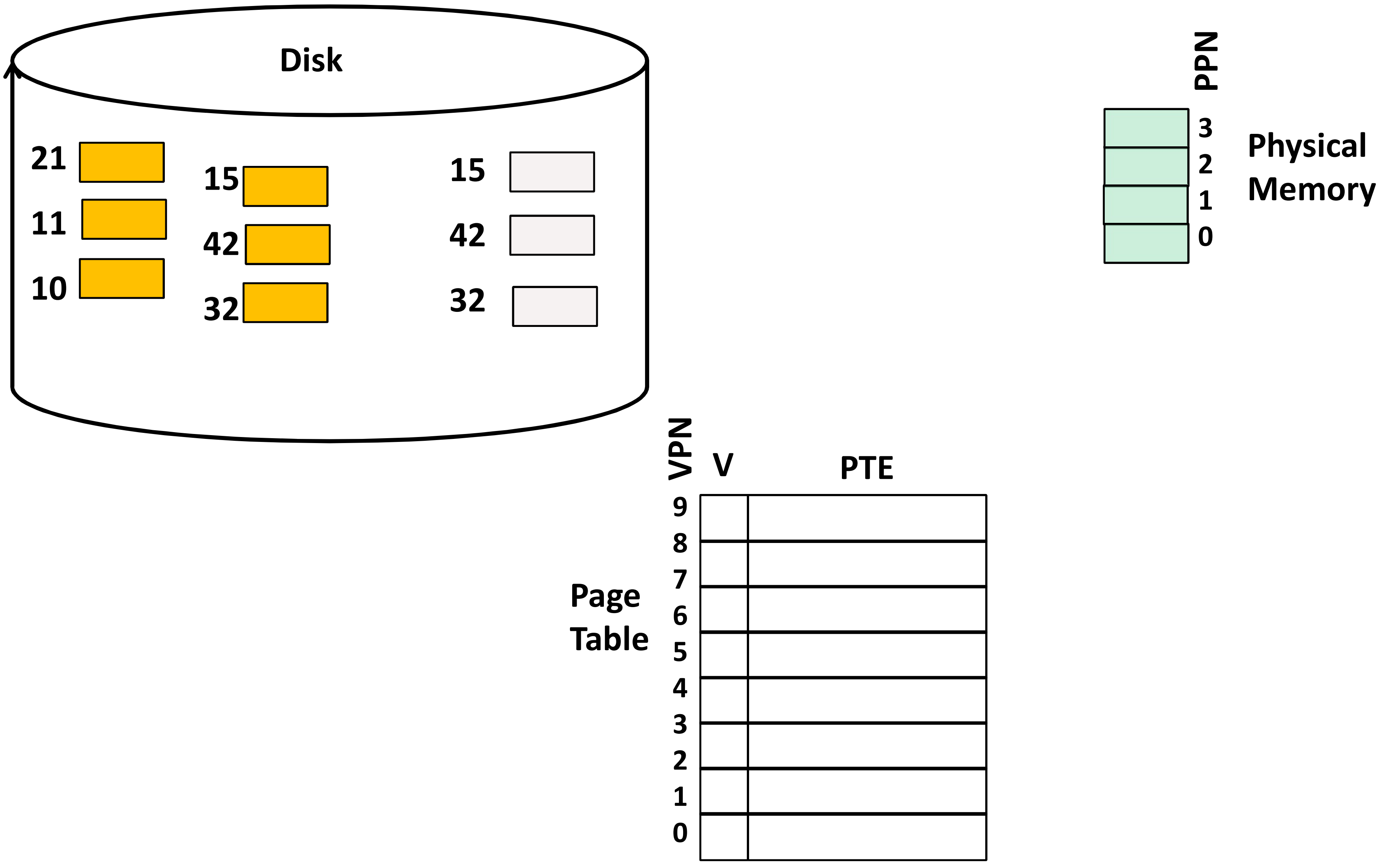
Some of the slides are adapted from Matt Welsh's.

Some slides are from Tanenbaum, Modern Operating Systems 3 e, (c) 2008 Prentice-Hall, Inc. All rights reserved. 0-13-6006639

Some slides are from Silberschatz, and Gagne.

Page Replacement

- How do we decide which pages to page-out (a.k.a kick out) of physical memory when memory is tight?
- How do we decide how much memory to allocate to a process?



Disk
21
11
10
15
42
32
15
42
32
PPN
3
2
1
0
Physical Memory
VPN
V
PTE
9
8
7
6
5
4
3
2
1
0
Page Table

Basic Page Replacement

- **How do we replace pages?**
  - 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.
  - Restart the process

Evicting the best page

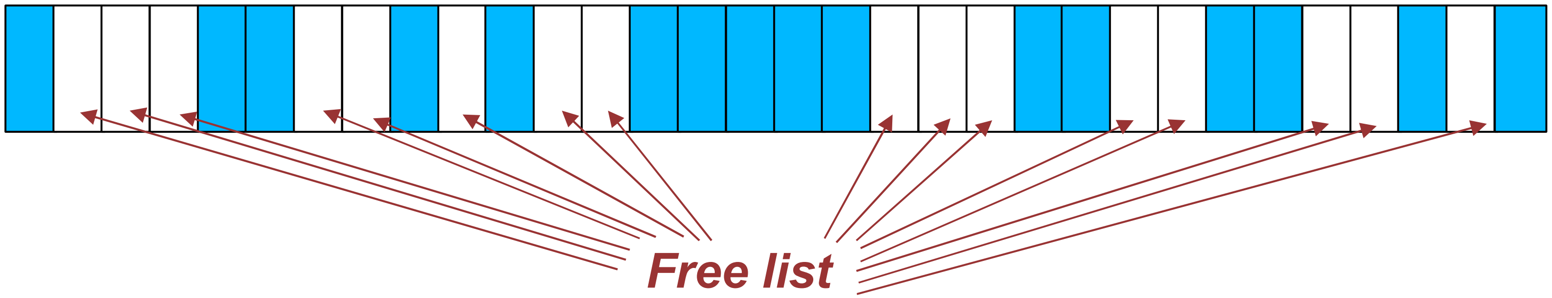
- **Goal of the page replacement algorithm:**
  - Reduce **page fault rate** by selecting the best page to evict
- **The "best" pages are those that will never be used again**
  - However, it's impossible to know in general whether a page will be touched
  - If you have information on future access patterns, 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)

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 policy*
  - The amount of physical memory and the *application footprint*

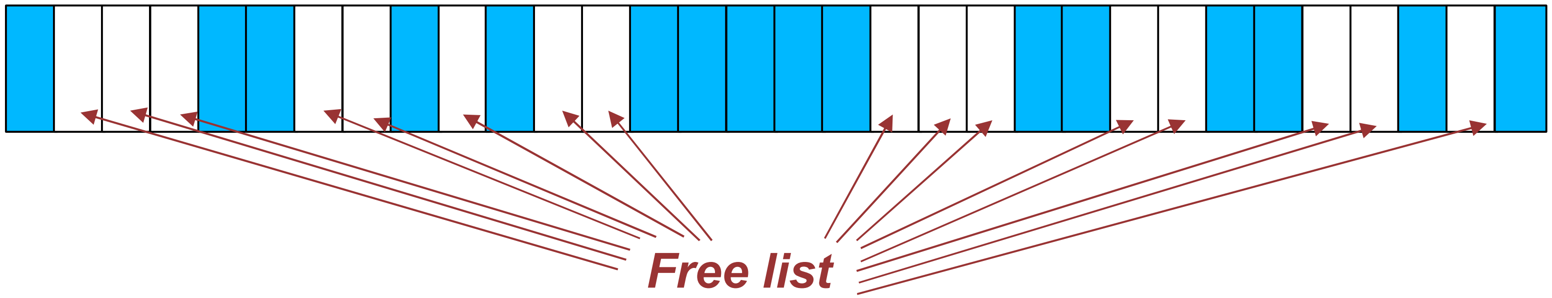
Page Replacement Basics

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



Page Replacement Basics

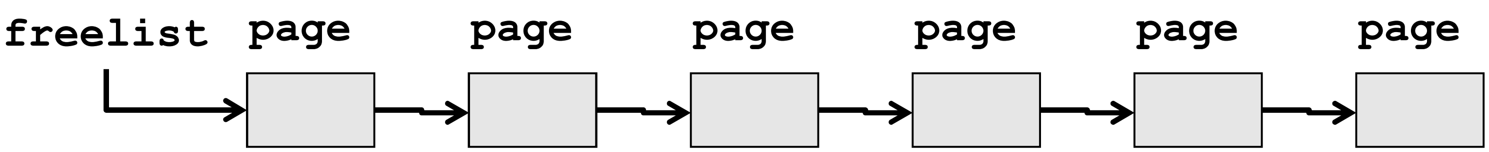
- **Most page replacement algorithms operate on some data structure that represents physical memory:**



  - Might consist of a bitmap, one bit per physical page
  - Might be more involved, e.g., a reference count for each page (remember Shared memory/CoW?)
  - Free list consists of pages that are unallocated
- **Several ways of implementing this data structure**
  - Scan all process PTEs that correspond to mapped pages (valid bit == 1)
  - Keep separate linked list of physical pages
  - ***Inverted page table***: One entry per physical page, each entry points to PTE

Free List

- **Bitmap representation: n/8 bytes.**
  - i.e. 4GB = 4M pages requires 512KB
  - More information per frame required if page is not free. i.e. invalidate PTE's of address translation tables referring an evicted frame.
- **Linked list of `page` structures:**



  - Allocating a free page and inserting an evicted page is fast. Insert/remove from the head
  - Non-free page structures keep reference count, reference to task memory maps, file block info if loaded from a file, state and protection.

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: 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 being used anymore
- **Why might FIFO not be so good?**
  - Doesn't consider spatial locality!
  - 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: 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”
  - Also called “MIN” (for “minimal”)
- **OPT is useful as a “yardstick” to compare the performance of other (implementable) algorithms against**

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 MIN 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?**

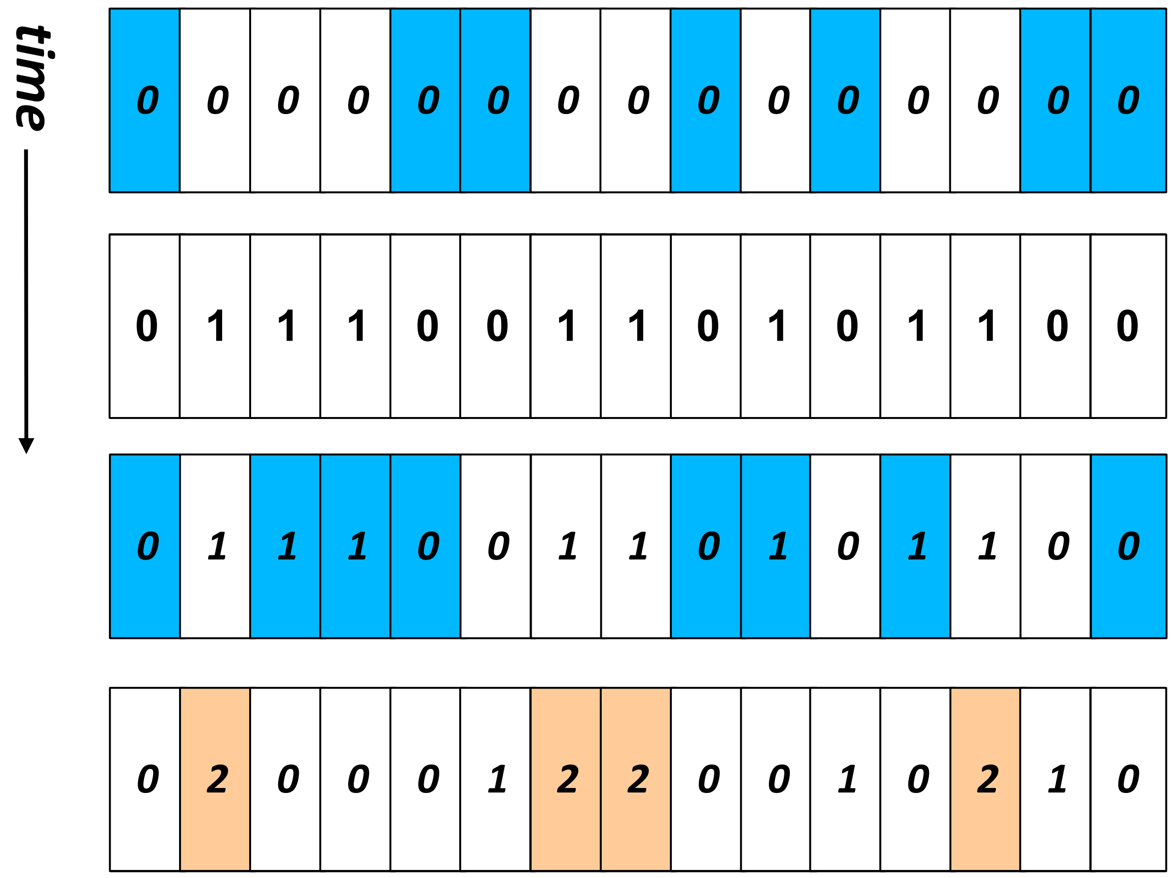
Algorithm: Least Recently Used (LRU)

- **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 MIN 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 for each page would double the size of every PTE
  - Scanning all of the PTEs for the lowest timestamp would be slow

Approximating LRU: Additional-Reference-Bits

- **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**
  - If the page has not been accessed (PTE reference bit == 0), **increment (or shift right)** the counter
  - If the page has been accessed (reference bit == 1), **set counter to zero (or shift right)**
  - **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**

LRU example



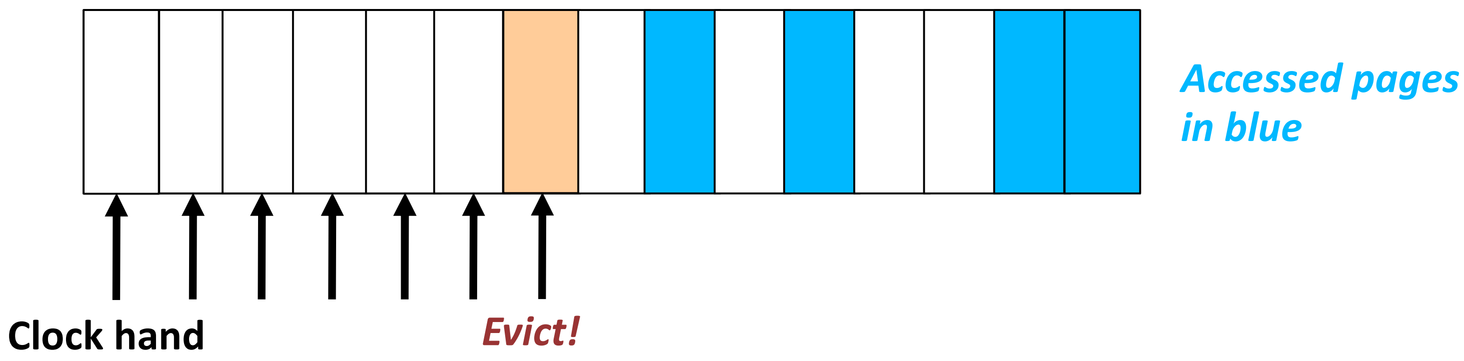
Accessed pages in blue

Increment counter for untouched pages

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

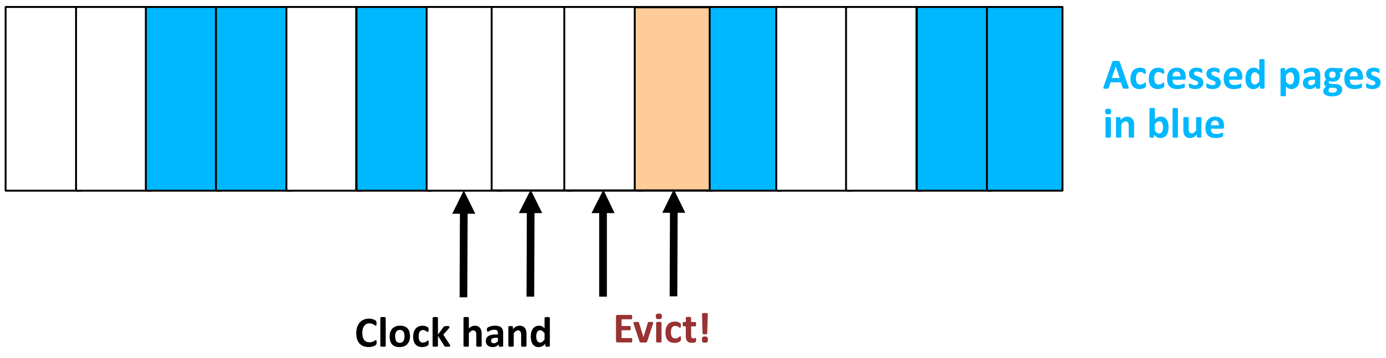
Algorithm: LRU Second-chance (Clock)

- **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)

- **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

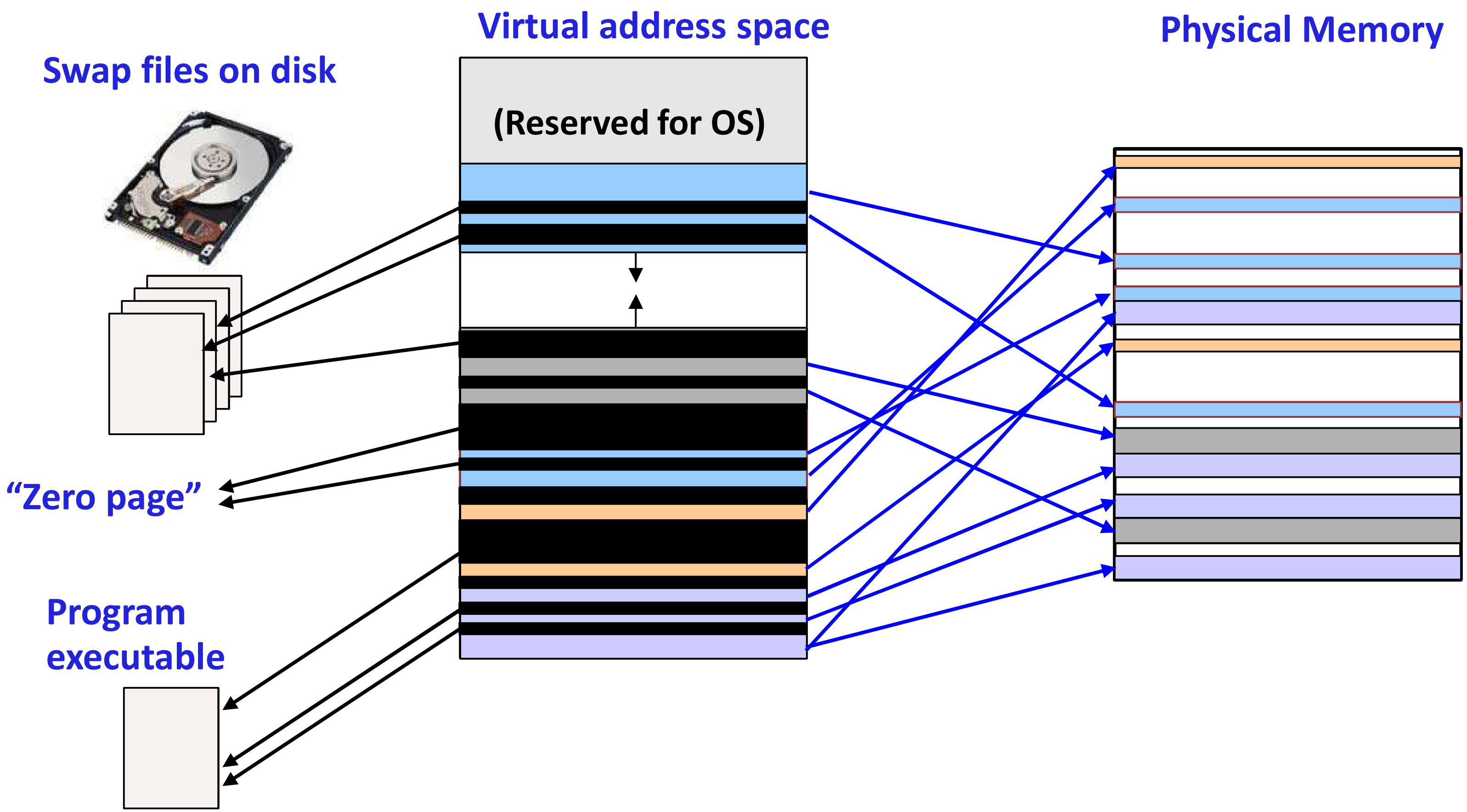
Algorithm: LRU Enhanced Second-chance (Clock)

- **Be even smarter: Consider the R(eference) bit and the M(odified) bit as an ordered pair to classify pages into four classes**
  - **(0,0)** : Neither recently used not modified – best page to replace
  - **(0,1)** : Not recently used but modified – not quite as good, since the page has to be written out before replacement
  - **(1,0)** : recently used but clean – probably will be used again
  - **(1,1)** :recently used and modified – probably will be used again and the page will be need to be written out before it can be replaced

- **We may need to scan the circular queue several times.**
- **The number of required I/O's reduced.**

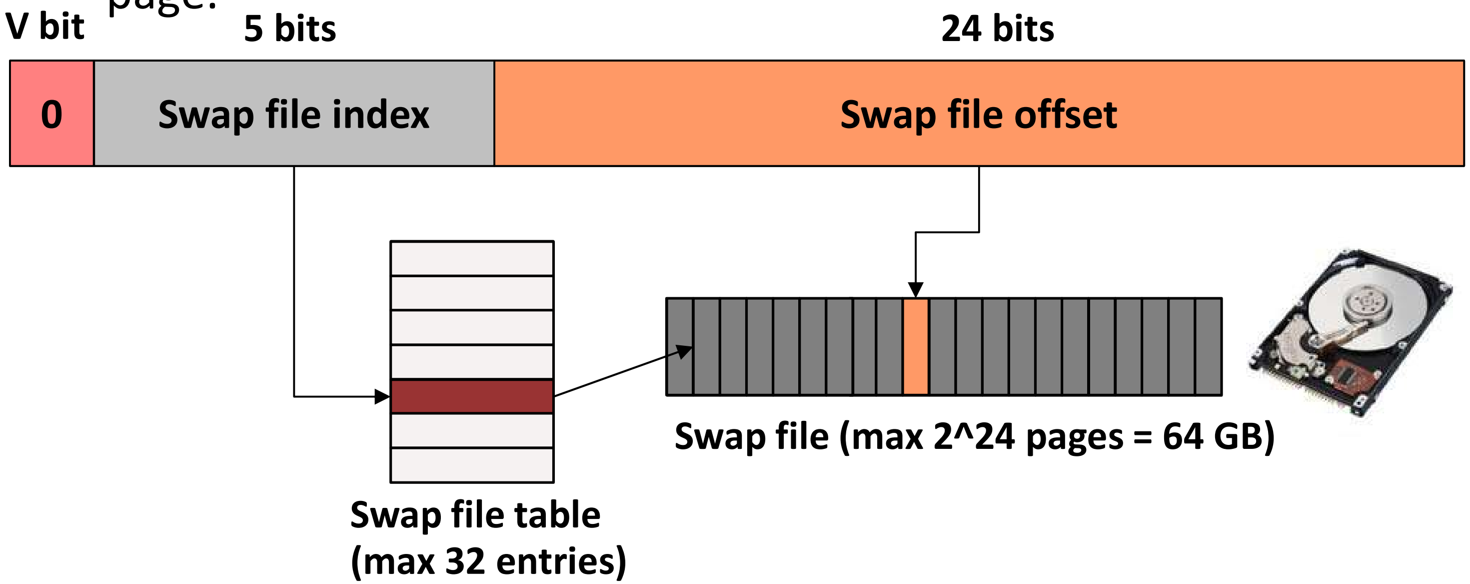
Swap Files

- **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

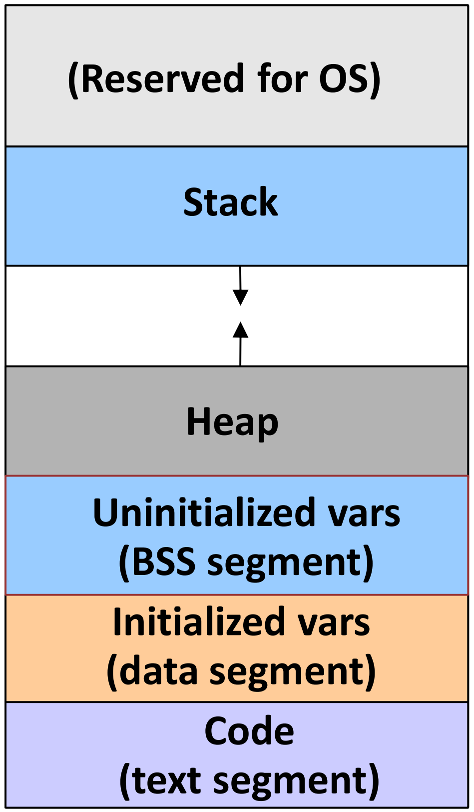
- **How do we keep track of where things are on disk?**
  - Recall PTE format
  - When V bit is 0, can recycle the PFN field to remember something about the page.



- **But … not all pages are swapped in from swap files!**
  - What about executables?
  - Or “zero pages”?
  - How do we deal with these file types?

VM map structure

- **OS keeps a "map" of the layout of the process address space.**
  - This is separate from the page tables.
  - In fact, the VM map is used by the OS to lay out the page tables.
- **This map can indicate where to find pages that are not in memory**
  - e.g., the disk file ID and the offset into the file.



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**

When to Evict/Page-Out Pages

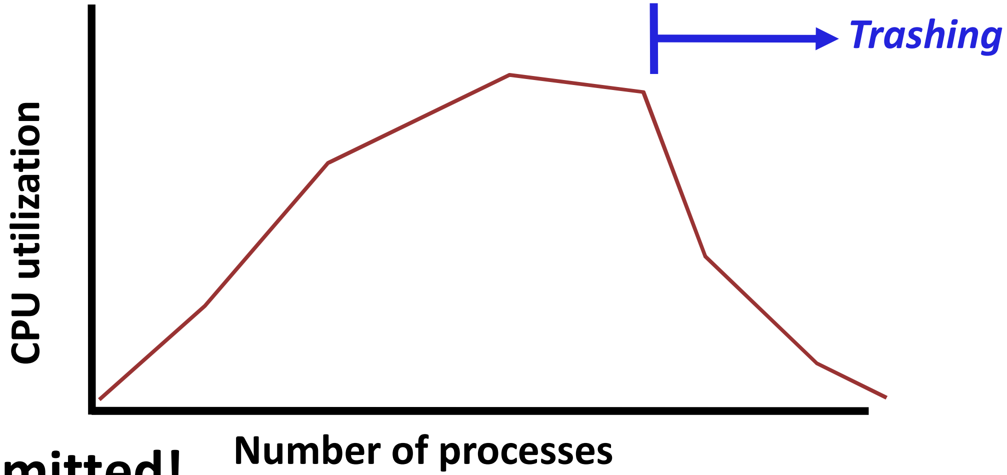
- **On page fault, when a free page is required**
  - In a loaded system most requests need replacement algorithm to work.
  - When replacement requires I/O, task needs to sleep.
  - Performance of tasks reduces, replacement time is added.
- **Solution: Page Daemon (or swap daemon)**
  - Watches system free memory. Start replacing pages as free memory drops below a threshold.
  - Maintains a pool of free memory all the time so tasks requiring a new page can find a new page instantly.
  - It sleeps when there is plenty of memory. Adaptively wake ups more often and replaces more pages as system is low on memory.
  - In extreme cases, it starts replacing whole memory of tasks (trashing)

Paging and swapping

- **However, on heavily-loaded systems, memory can fill up**
- **To achieve good system performance, must move "inactive" pages out to disk**
    - If we didn't do this, what options would the system have if memory is full???
  - 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 a page back into memory?
- **Swapping**
  - Usually refers to moving the memory for an entire process out to disk
  - This effectively puts the process to sleep until OS decides to swap it back in
- **Paging out/in**
  - Refers to moving individual pages out to disk (and back)
  - We often use the terms "paging out" and "swapping" interchangeably

Trashing

- **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?

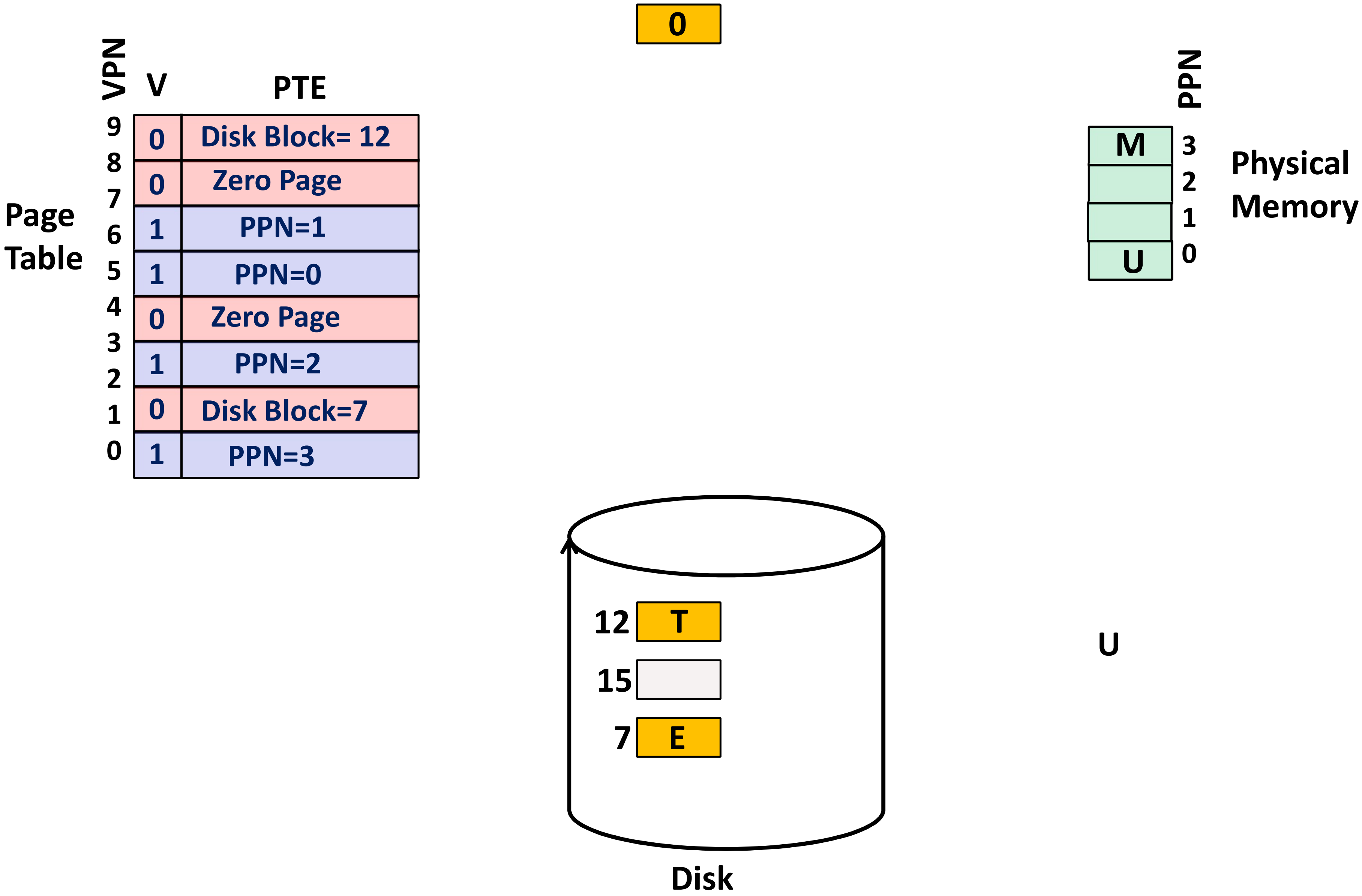
Benefits of sharing pages

- **How much memory savings do we get from sharing pages across identical processes?**
  - A lot! Use the “top” command...

```
Terminal — top — 88x26
Processes:  68 total, 2 running, 1 stuck, 65 sleeping... 246 threads    13:17:30
Load Avg:  0.75, 0.58, 0.52      CPU usage:  7.7% user, 17.9% sys, 74.4% idle
SharedLibs: num =  223, resident = 33.3M code, 4.61M data, 4.80M LinkEdit
MemRegions: num = 17413, resident =  208M + 11.0M private,  546M shared
PhysMem:   618M wired,  261M active,  130M inactive, 1010M used, 13.9M free
VM: 9.79G +  150M   635052(61) pageins, 455424(0) pageouts

  PID COMMAND      %CPU   TIME    #TH #PRTS #MREGS RPRVT  RSHRD  RSIZE  VSIZE
 3784 Grab         5.0%  0:00.51   3   126   159  2.23M+ 7.25M+ 16.8M+  216M+
 3781 less         0.0%  0:00.02   1    13    17   148K   304K   484K  26.7M
 3778 sh           0.0%  0:00.00   1     8    16  88.0K   608K   364K  27.1M
 3777 sh           0.0%  0:00.00   1    13    16  68.0K   608K   544K  27.1M
 3776 man          0.0%  0:00.01   1    13    16   184K   264K   460K  26.7M
 3752 bash         0.0%  0:00.01   1    14    16   228K   696K   816K  27.1M
 3751 login        0.0%  0:00.01   1    16    40   172K   380K   636K  26.9M
 3748 top         12.8%  0:23.16   1    25    20   704K   300K  1.14M  27.0M
 3725 bash         0.0%  0:00.02   1    14    16   228K   696K   812K  27.1M
 3724 login        0.0%  0:00.01   1    16    40   172K   380K   636K  26.9M
 3722 Terminal     0.2%  0:02.31   6    92   140  2.25M  11.1M  10.3M   218M
 3719 WinAppHelp   0.0%  0:00.05   1    57    95   716K  4.10M  3.00M   198M
 3713 mdimport     0.0%  0:00.90   4    68   119  1.59M  3.16M  4.64M  57.8M
 3675 iTunes       3.5%  6:51.76   9   193   370  7.12M  12.1M+ 10.2M   263M
 3670 Address Bo   0.0%  0:02.58   1    92   179  2.21M  5.56M  15.2M   216M
 3659 Mail         0.0%  0:59.65   8   172   415  25.3M  10.9M+ 27.2M   258M
 3084 Skype        0.7% 17:20.32  18   240   452  23.9M  8.65M+ 20.0M   304M
  655 vfstool      0.0%  0:00.07   2    14    29   120K   308K   256K  32.1M
```

Page Replacement



VM map structure

- **OS keeps a "map" of the layout of the process address space.**
  - This is separate from the page tables.
  - In fact, the VM map is used by the OS to lay out the page tables.
- **This map can indicate where to find pages that are not in memory**
  - e.g., the disk file ID and the offset into the file.