EECS-343 Operating Systems
Lecture 7:
Swapping

Steve Tarzia

Spring 2019

Northwestern

Announcements

- Project 2 due Monday
  - It's much more difficult than Project 1!
- Midterm exam next Thursday, May 2nd

Last Lecture: VM & Paging optimizations

- **Latency cost**, because each memory access must be translated.
  - **Translation lookaside buffer (TLB)** caches recent virtual to physical page number translations.
  - Software-controlled paging removes page tables from the CPU spec and lets OS handle translations in software, in response to TLB miss exceptions.
- **Space cost**, due to storing a page table for each process.
  - Linear (one-level) page tables are large.
  - Smaller pages lead to less wasted space during allocation, but more space is consumed by page tables.
  - **Multi-level page tables** are the only way to truly conserve space.
  - Mixed-size pages reduce TLB misses.
- Copy-on-write fork, demand zeroing, lazy loading, and library sharing all reduce physical memory demands.

Motivation for swapping

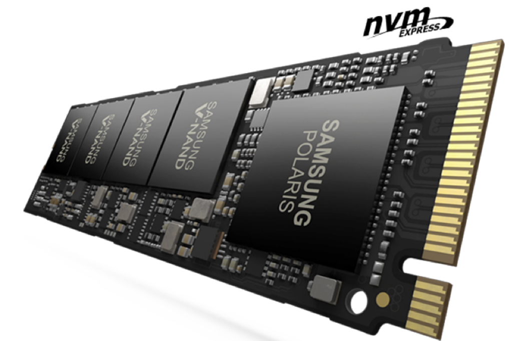
- Paging allows many processes to share the same physical memory
- Each process has a large virtual address space
    - Programs aren't really aware of how much RAM is in the system
    - This is very convenient to the programmer, but actually RAM space is limited!
- When physical RAM is fully consumed, OS must somehow react:
    - *Option 1:* Give up and kill the memory-hog process(es)
    - *Option 2:* Free some physical memory by temporarily moving pages to disk
        - Disk is large (~100x RAM size), but *very* slow.
        - Hopefully, some of the memory used by processes is infrequently used.

- **Swapping** is temporarily moving memory pages to disk

Computer storage hierarchy

Larger, but slower ↓

delay		*capacity*
0.3ns	CPU Registers	1 kB (kilobyte)
5ns	CPU Caches (L2)	16 MB
50ns	Random Access Memory (RAM)	16 GB
100µs	Flash Storage (SSD)	1 TB
5ms	Magnetic Disk	8 TB



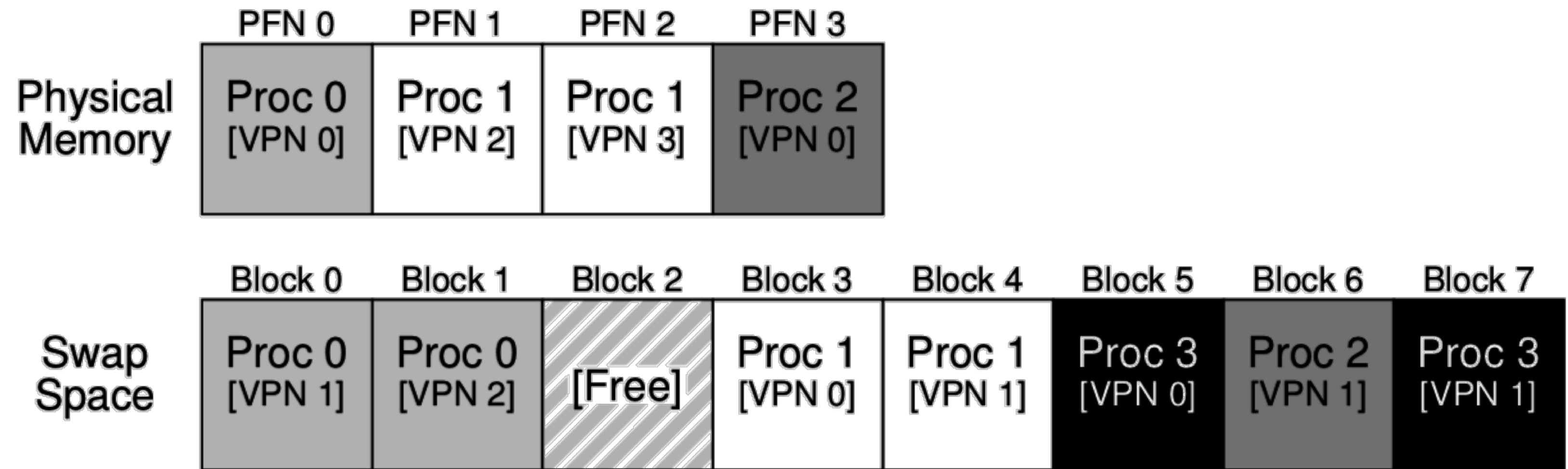
- Disk is about **100 times larger** than RAM, but has about **10,000 times higher latency** (delay) if magnetic, or **1000 times higher latency** if SSD.
- As always,
  - Goal is to work as much as possible in the top levels.
  - Large, rarely-needed data is stored at the bottom level

Swapped pages are stored on disk

- On Linux, swap space is a special (optional) disk partition
  - Allocated when you install the OS
  - This part of the disk is only used for swapping, never for regular files
  - Its formatting is optimized for storing page-sized chunks of data
- On Windows, swap space is in a hidden file called pagefile.sys
  - Allows swap space to grow and shrink on demand, sharing space with the main filesystem.
  - Compared to Linux's strategy it's more flexible, but slower.
- SSDs are better than magnetic disks for swap space, because of much lower latency (relatively little space is needed for swapping).

A simple swapping example

- Process's memory pages are distributed between RAM and disk:

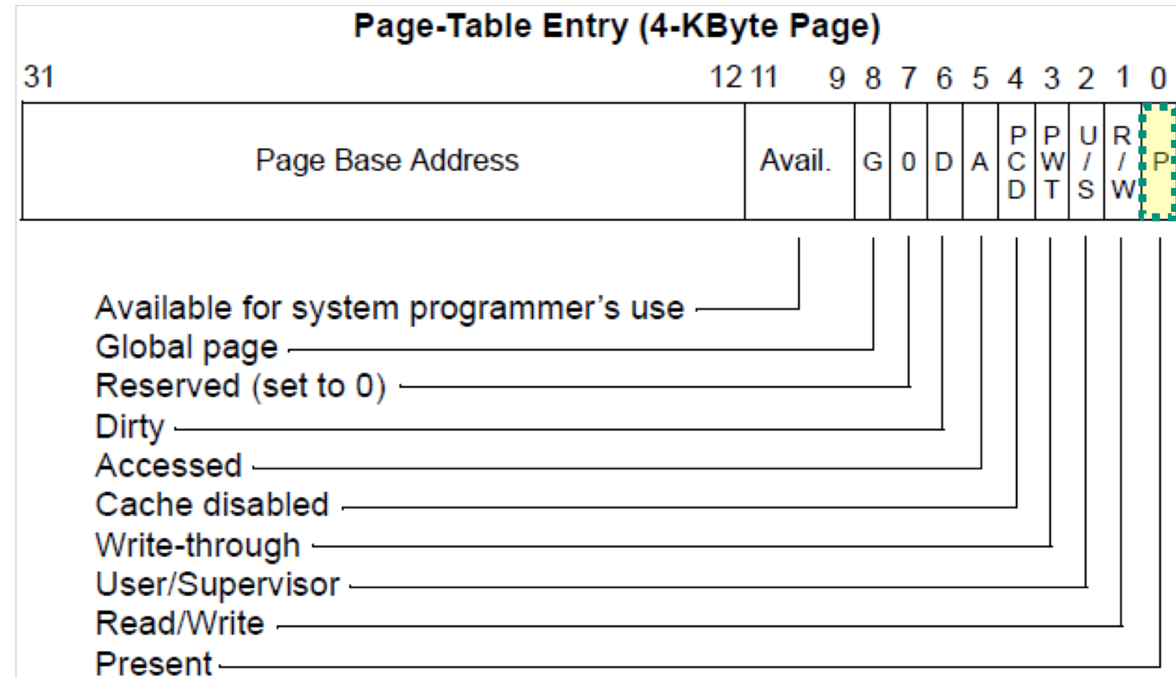


- Processes 0, 1, & 2 are *partially* in RAM
- Process 3 is entirely in Swap Space
- System's total memory is effectively tripled (12 pages instead of 4)
- ***But***, swapped memory cannot be used by the CPU. It's in *deep storage*.
  - A page fault exception and some kernel action will be needed to retrieve it.

Swapping Mechanics

To swap a page to disk:

- Set the PTE *present* bit to zero
- To distinguish from an invalid page, also store the *disk address* of the page in a kernel data structure.
  - We can actually store the disk address in the high bits of the PTE!
  - The top 31 bits can be used
- Copy the page to the disk location



Page faults occur when PTE is marked "not present"

If CPU sees a present bit set to zero when doing a virtual address translation, it:

- sets the `%CR2` register to the offending virtual address
- generates a *page fault* exception #14

OS handles the page fault interrupts in two possible ways:

- ***Option 1:*** if process made an invalid memory access
  - Read an invalid (or privileged) address or wrote a read-only page.
  - Kill the process and report "segmentation fault" to the user
- ***Option 2:*** if page was *swapped out* by the OS
  - Program didn't do anything wrong, it was just unlucky (or a memory hog)
  - Have to *swap in* the page and retry the last instruction

- How to distinguish these two cases? Interrupt handler checks the high bits of PTE:
  - OS can choose to store all zeros for invalid pages vs disk location for swapped pages.

Swapping **in**

- If swap space is being used, that means that RAM is in short supply
- Probably have to ***evict*** a page to make room for the demanded page.
- ***Page replacement policy*** determines which page to evict
  - Recall that disk read/write is very slow, so we should chose wisely.
  - Goal is to minimize future page faults by evicting an **unpopular** page
  - Evicted page can live happily on disk while it is unused

Magnetic disk demo

- Don't try this at home:
- https://www.youtube.com/watch?v=3owqvmMf6No

Throughput versus *Latency*

- There are two very different metrics for measuring I/O "speed"
  - ***Throughput*** is bytes per second
  - ***Latency*** is the delay before the action starts
- Latency is always higher for large, distant storage devices
  - Magnetic disk > SSD > RAM > L3 cache > L2 > L1 > Registers
  - At best, speed of light limits information transfer to 30cm/ns
  - Disk latency includes mechanical ***rotation*** and ***seek***
- However, throughput can be increased using parallelization
- A small I/O to disk is just as slow as a larger one
  - So, we try to batch I/O requests together.

A Postal analogy

- Letter represents as small disk I/O
- Package represents a big disk I/O
- A small letter and a big package both take the same time to reach Los Angeles.
  - ***Latency*** is identical
- However, the big package transfers more info at once.
- Package is a ***batched*** transfer
  - Package ***throughput*** is higher

When to swap?

- ***Demand swapping*** – swap pages in response to page faults
  - The simplest approach
  - For efficiency, the swap would happen ***asynchronously***:
    - Interrupt handler should be fast – just start the disk I/O and block the process to let another process run.
- ***Background swapping*** – swap pages preemptively
  - A swap ***daemon*** (background process) periodically runs (like the scheduler)
  - If # free physical frames < "low water mark," evict some pages until the quantity reaches the "high water mark"
    - The system will swap many pages at once to get from the *low* to *high* level.
    - We do this because writing a *large* block to disk is more efficient (batched transfer)

Background swapping

- Avoids long, unexpected delays in allocating memory
- Also reduces page fault latency because there is no need to evict a page prior to swapping in.
- Can be scheduled with low priority
- Takes advantage of idle time to prepare future work
- Linux swap daemon is a process called ***kswapd***

- Note: swapping is sometimes called "paging," so the background process can be called a "page daemon."

Page faults enable ***lazy allocation*** and ***lazy loading***

- In practice, paging is not just used to handle memory overflow
  - Paging provides an opportunity to be lazy about loading requested data
  - This is an important performance optimization, *reducing program start time*
- If a process uses *sbrk* or *mmap* to request a huge chunk of memory, maybe it will not use all that memory immediately (or ever!).
  - Programmers and compilers are sometimes ***greedy*** in their requests
  - We can *virtually* allocate memory, but mark most of the pages “not present”
  - Let the CPU raise an exception when the memory is really used
  - Then really allocate the demanded page
- Lazy loading also works for large code binaries
  - Delay loading a page of instructions until it’s needed

Speaking of laziness – break time!

Page Replacement Policies

- Same idea as the Translation Lookaside Buffer (TLB) in last lecture
  - Again memory access either ***hit*** or ***miss***. (Miss triggers a slow swap-in)
- When time comes to evict a page, we want to choose one that will not be used again soon, and we use recent history to predict the future
- Page faults are more costly than TLB misses, so we can afford to spend some time and energy implementing a sophisticated good policy.

- Definition: ***memory pressure*** is high demand for memory

Page replacement policy – problem definition

Given:

- A small number of page frames
- A larger number of virtual pages
- A sequence of memory accesses (a sequence of virtual page demands)

Choose:

- An mapping of VPs to PFs at each time step (page table config.)

Such that:

- No two VPs are mapped to the same PF at the same time
- Demanded virtual page at each time step is mapped to a physical page
- We ***minimize*** changes in the mapping over time. (Minimize swaps.)

Optimal page replacement policy

- Replace page that will be accessed ***furthest in the future***
- This makes sense because it will evict rarely-used pages
- However, it's not practical because we can't observe the future
  - But it's useful to construct simulations comparing real policies to the optimal policy – a *performance upper-bound*.
- Notice that even an optimal policy has misses due to ***cold-start*** and ***capacity***.

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		0
1	Miss		0, 1
2	Miss		0, 1, 2
0	Hit		0, 1, 2
1	Hit		0, 1, 2
3	Miss	2	0, 1, 3
0	Hit		0, 1, 3
3	Hit		0, 1, 3
1	Hit		0, 1, 3
2	Miss	3	0, 1, 2
1	Hit		0, 1, 2

Table 22.1: **Tracing the Optimal Policy**

Least Recently Used (LRU) approximates optimal

- Evict the LRU virtual page
- Assuming temporal and spatial locality, future memory accesses should be similar to past accesses.

Access	Hit/Miss?	Evict	Resulting Cache State	
0	Miss		LRU→	0
1	Miss		LRU→	0, 1
2	Miss		LRU→	0, 1, 2
0	Hit		LRU→	1, 2, 0
1	Hit		LRU→	2, 0, 1
3	Miss	2	LRU→	0, 1, 3
0	Hit		LRU→	1, 3, 0
3	Hit		LRU→	1, 0, 3
1	Hit		LRU→	0, 3, 1
2	Miss	0	LRU→	3, 1, 2
1	Hit		LRU→	3, 2, 1

Table 22.4: **Tracing the LRU Policy**

LRU suffers from "corner case" behaviors

- Certain access patterns can cause LRU to make the wrong prediction every time.
- Imagine a system with 3 page frames and the following access pattern:
  - Virtual pages: 1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4, …
  - In this case the LRU page is exactly the one we're going to use next!
  - Every access would be a miss.
  - The problem is that LRU's "remembers" only the past 3 accesses, but the repetition happens on a longer time scale (4).
- ***Random*** eviction policy avoids this "worst case" behavior
  - Simple to implement
  - But performs worse that LRU on many workloads

OS performance analysis

- As we have seen, no one OS policy works best for every program.
- When testing an OS (or any complex software system), must choose a sample ***workload***:
  - A workload (sometimes called a benchmark) is a repeatable execution scenario that is meant to mimic a variety of realistic use patterns.
  - Every program is written differently
  - Every user places different demands on those programs
- Any performance evaluation is just an *experiment* in conditions that should be designed to answer a particular question.

Some page replacement experiments (varying RAM size)

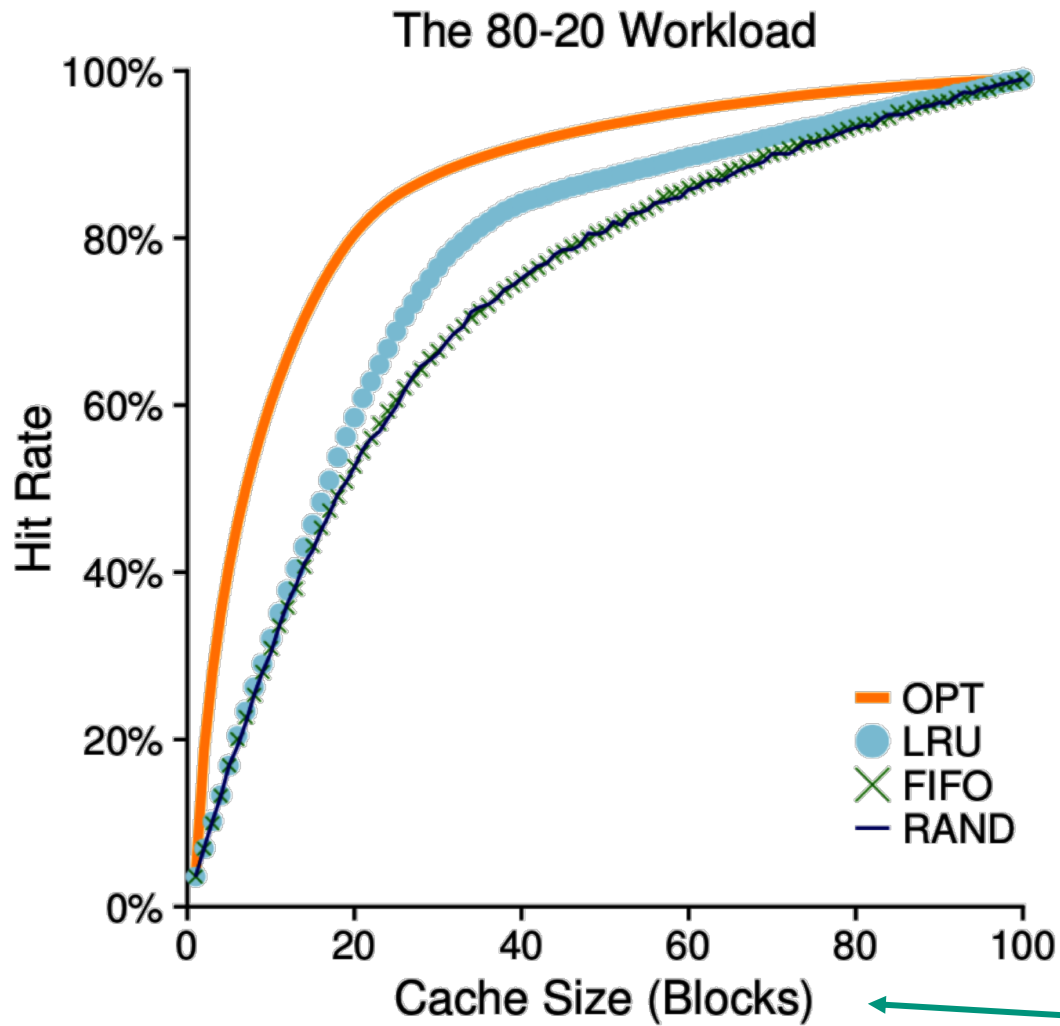
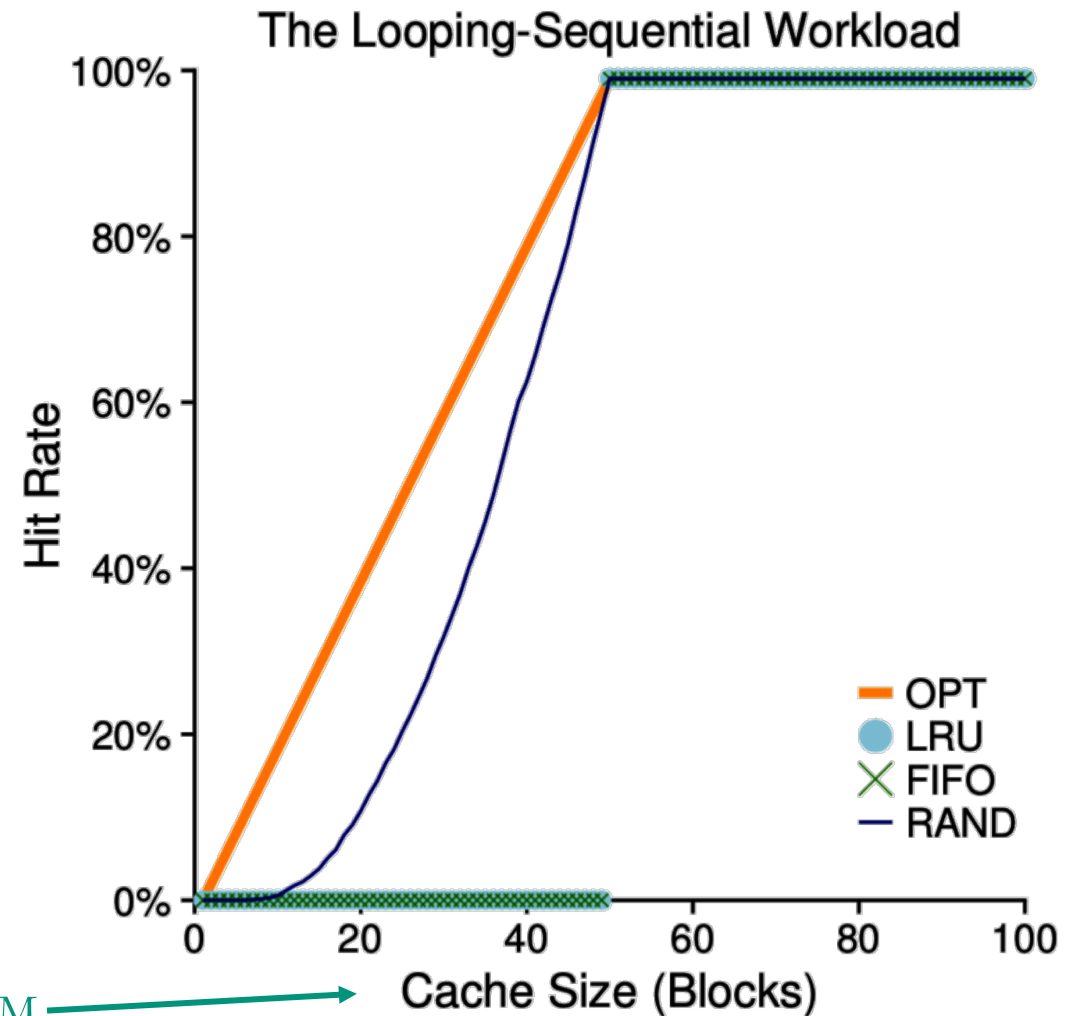


Figure 22.3: **The 80-20 Workload**

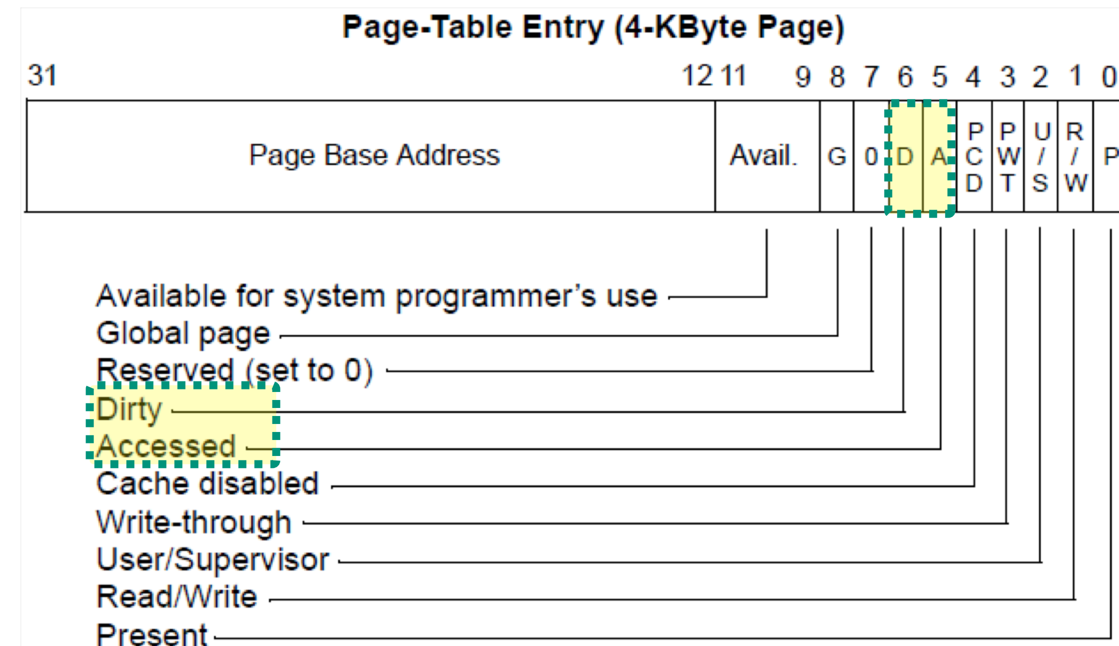


RAM

Figure 22.4: **The Looping Workload**

How to implement LRU

- So far, OS has no way of knowing which pages were used recently
- The CPU hardware provides some additional features to help the OS
  - x86 sets an ***accessed*** bit in PTE if a page is read/written
  - additionally sets ***dirty*** bit if page has been written
- ***Clock algorithm***:
  - OS evicts the first page it sees having accessed = 0.
  - If it sees accessed = 1, reset the bit and move on to next page.
  - Start looking where you left off last time, using a circular linked list of pages.



Keeping your hands clean

- Read-only pages are good candidates for eviction because we can simply throw out the page (without writing to disk).
  - This assumes that the OS is being clever about tracking the source of pages.
- For example:
  - Instructions from a code binary can be be re-read from the filesystem.
  - A data page can be duplicated in both swap space and in memory
    - If the in-memory copy was never written, it can be evicted without re-writing it to disk.
- ***Dirty bit*** allows OS to recognize data pages that have been changed and thus are inconsistent with the copy on disk.
- In other words, the eviction policy should not touch the dirty pages.

Thrashing

- Thrashing is a condition where swapping happens constantly
  - This is due to processes simply using way too much memory.
  - Swapping is very slow, so all programs are essentially frozen
- Linux `top` command lets you see the page fault count per process to detect thrashing.
- When thrashing, it's usually a good idea to kill some processes.
  - Better to do one thing well than to do many things very poorly

Approximating LRU for page replacement

- ***Clock algorithm*** is one option:
  - Basically, just scan through pages until we find one with *accessed bit* = 0.
  - Not truly LRU, but will find a page that was *not recently used*.
- What if our CPU architecture doesn't set an accessed bit (like VAX)?
  - Emulate an accessed bit using the present bit and page faults.
  - Set *present bit* = 0 but leave page in physical memory and leave address in PTE.
    - Set one of the OS-reserved bits in the PTE to recognize that the page is given a second chance.
  - If we see a page fault then the page was read, set *present* = 1 (no need for I/O)
- The above is called the ***Second chance*** algorithm:
  - Pretend page was evicted, but just test to see if it will be accessed again

Disk buffering and filesystem caching

- Swapping gives the illusion that RAM is as big as the disk.
- Similarly, ***filesystem caching*** gives the illusion that disk is as fast as RAM.
- Programs explicitly access three kinds of storage:
  (1) registers (2) memory (3) files
- File I/O can be a significant ***performance bottleneck:***
  - A bottleneck is the one slow component that limits performance
- **Filesystem caching:**
  - To improve performance, OS stores most recently used "pages" of disk in RAM (physical page frames).

`top` also reports filesystem cache size

```
top - 10:25:45 up 7 days, 48 min,  3 users,  load average: 0.04, 0.06, 0.09
Tasks: 650 total,   1 running, 649 sleeping,   0 stopped,   0 zombie
Cpu(s):  0.0%us,  0.0%sy,  0.0%ni, 99.9%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:  132144848k total, 129331984k used,  2812864k free, 37895660k buffers
Swap: 16383996k total,      436k used, 16383560k free, 45074412k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 9213 mysql     20   0 1263m 156m  14m S  0.0  0.1   3:57.24 mysqld
10001 root      20   0 5748m 219m  14m S  0.3  0.2  15:02.22 dsm_om_connsvcd
 9382 root      20   0  337m  18m  11m S  0.0  0.0   0:10.67 httpd
 8304 apache    20   0  352m  19m  10m S  0.0  0.0   0:00.29 httpd
 8302 apache    20   0  339m  14m 7144 S  0.0  0.0   0:00.16 httpd
 8298 apache    20   0  339m  14m 7140 S  0.0  0.0   0:00.12 httpd
 8299 apache    20   0  339m  14m 7136 S  0.0  0.0   0:00.17 httpd
 8303 apache    20   0  339m  14m 7136 S  0.0  0.0   0:00.17 httpd
 8300 apache    20   0  339m  14m 7120 S  0.0  0.0   0:00.13 httpd
 8301 apache    20   0  339m  14m 7120 S  0.0  0.0   0:00.16 httpd
 8305 apache    20   0  339m  14m 7112 S  0.0  0.0   0:00.13 httpd
 1386 apache    20   0  339m  14m 7096 S  0.0  0.0   0:00.06 httpd
 1387 apache    20   0  339m  14m 7084 S  0.0  0.0   0:00.07 httpd
 1122 spt175    20   0  251m  14m 6484 S  0.0  0.0   0:00.26 emacs
 2615 root      20   0 92996 6200 4816 S  0.0  0.0   0:00.93 NetworkManager
 9865 root      20   0 1043m  23m 4680 S  0.3  0.0   9:44.98 dsm_sa_datamgrd
 8737 postgres  20   0  219m 5380 4588 S  0.0  0.0   0:01.00 postmaster
 2786 haldaemo  20   0 45448 5528 4320 S  0.0  0.0   0:03.99 hald
 9956 root      20   0  491m 7268 3280 S  0.0  0.0   3:16.30 dsm_sa_snmpd
  990 root      20   0  103m 4188 3172 S  0.0  0.0   0:00.01 sshd
 1014 root      20   0  103m 4196 3172 S  0.0  0.0   0:00.02 sshd
19701 root      20   0  103m 4244 3172 S  0.0  0.0   0:00.01 sshd
```

- ***buffers*** and ***cached*** both represent file data that is being stored in memory for improved performance
  - There is a slight difference between the two, but it's not important.
- This machine has lots of RAM (128GB)
  - The majority of RAM is now being used to cache files (~83GB)

Unified Page Cache

- In modern OSes, the page replacement policy simultaneously considers placement of both VM pages and disk blocks in physical RAM.
  - May choose to evict either a VM page or a cached disk block.
  - Eventually, disk blocks are really written to the disk (flushed).
    - Certainly when the page is evicted, but may also be flushed earlier to make sure the file data is not lost if the machine crashes or loses power.
- A VM page is either:
  - ***file-backed*** if it can be reloaded from a file (code)
    - We can discard this page if evicting
  - ***anonymous*** if it is memory that was created by a process
    - Must save page to swap space if evicting

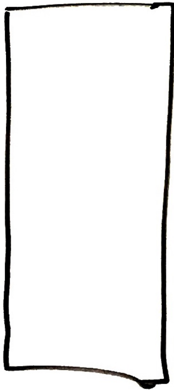
Programmer's view

CPU

Registers

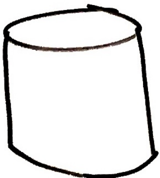
%eax
%ebx
...

Virt. Mem.



mov
push
pop

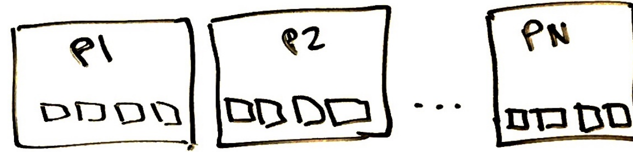
Disk



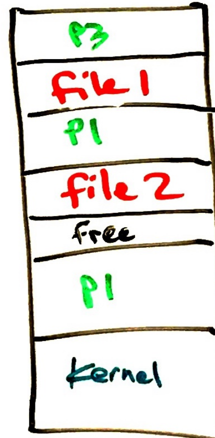
Syscalls:
- open
- read
- write

OS's view

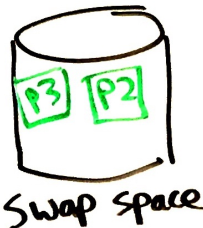
Per-process copies of Registers



Active VM & Disk pages in RAM



Overflow pages & files on disk



swap space

main filesystem

Reality ...

Hardware has further optimizations which the OS cannot see:

- CPU places most recently accessed RAM in L3, L2, and L1 caches.
- Disks have caches of ~128 Mb.
- Hybrid disks migrate data between flash and magnetic platters.

The benefits filesystem caching

Filesystem caching allows programmers to focus on functionality rather than performance.

- Write data to disk if it needs to be ***persistent.***
- Don't worry about disk speed, OS will somehow make it seem fast.
- Actually, accessing a file can be almost as fast as a register if the disk block is stored in an L1 cache.
  - Although there will be some overhead for the file read/write syscall

- A great example of why intelligent OSes are important!

Recap

- Disk is slow, but large, and can be used to store RAM's overflow
  - Disks have high ***throughput*** (transfer bitrate) but high ***latency*** (delay)
  - Magnetic disks have even higher latency than SSDs, due to moving parts.
- Paging and swapping work together, using the same CPU mechanisms
  - If a page is marked "not present" it may be either invalid or swapped to disk.
    - Or it might indicate *lazy allocation*, *lazy loading*, or *copy-on-write*, as we saw last time.
  - High bits of page table entry can store disk location of swapped page.
- ***Page replacement policy*** decides which page(s) to ***evict*** to free memory
  - Swapping can be done ***on demand*** or in the ***background***
  - Having some free physical frames will prevent delays for allocations.
  - ***Accessed bit*** and ***Dirty bit*** in PTEs inform the page replacement policy
- ***Thrashing*** is when swapping prevents the system from doing any work.
- ***Unified page cache*** handles both traditional paging and ***file caching***.
  - Makes filesystem access seem just as fast as memory access.