CS 537
Lecture 8
Paging and Page Replacement

Michael Swift

Hardware and Kernel structures for paging

- Hardware:
  - Page table base register
  - TLB
- Software:
  - Page table
    - Virtual --> physical or virtual --> disk mapping
  - Page frame database
    - One entry per physical page
    - Information on page, owning process
  - Swap file

Page Frame Database

```
/*
 * Each physical page in the system has a struct page associated with
 * it to keep track of whatever it is we are using the page for at the
 * moment. Note that we have no way to track which tasks are using
 * a page.
 */
struct page {
  unsigned long flags;             // Atomic flags: locked,referenced,dirty,slab,disk
  atomic_t _count;                 // Usage count, see below. */
  atomic_t _mapcount;              // Count of ptes mapped in mms,
                                   // to show when page is mapped
                                   // & limit reverse map searches.
struct {
      unsigned long private;         // Used for managing pages used in file I/O
      struct address_space *mapping;  // Used for memory mapped files
   };
   pgoff_t index;                    // Our offset within mapping. */
   struct list_head lru;             // Lock on Pageout list, active_list
   void *virtual;                    // Kernel virtual address *
};
```


Shared memory

- Exploit level of indirection between VA and PA
    - regions of two separate processes' address spaces map to the same physical frames
      - read/write: access to share data
      - execute: shared libraries!
    - will have separate PTEs per process, so can give different processes different access privileges
    - must the shared region map to the same VA in each process?

Saving memory to disk

- When there is not enough memory for all our processes, the OS can copy data to disk and re-use the memory for something else
  - Copying a whole process is called "swapping"
  - Copying a single page is called "paging"
- Where does data go?
  - If it came from a file and was read only, it stays in the file
    - E.g. executable code
  - Unix: a swap partition
    - A region of the disk reserved for "backing store"
  - Windows: a swap file
    - A designated file in the regular file system
- When does data move?
  - Swapping: in advance of running a process
  - Paging: when a virtual page is accessed

Demand Paging

- We've hinted that pages can be moved between memory and disk
  - this process is called demand paging
  - OS uses main memory as a (page) cache of all of the data allocated by processes in the system
    - initially, pages are allocated from physical memory frames
    - when physical memory fills up, allocating a page in requires some other page to be evicted from its physical memory frame
  - evicted pages go to disk (only need to write if they are dirty)
    - to a swap file
    - movement of pages between memory / disk is done by the OS
    - is transparent to the application
      - except for performance…

Why does this work?

- Locality!
  - temporal locality
    - locations referenced recently tend to be referenced again soon
  - spatial locality
    - locations near recently references locations are likely to be referenced soon (think about why)
- Locality means paging can be infrequent
  - once you've paged something in, it will be used many times
  - on average, you use things that are paged in
  - but, this depends on many things:
    - degree of locality in application
    - page replacement policy and application reference pattern
    - amount of physical memory and application footprint

Why is this "demand" paging?

- Think about when a process first starts up:
  - it has a brand new page table, with all PTE valid bits 'false'
  - no pages are yet mapped to physical memory
  - when process starts executing:
    - instructions immediately fault on both code and data pages
    - faults stop when all necessary code/data pages are in memory
    - only the code/data that is needed (demanded!) by process needs to be loaded
    - what is needed changes over time, of course…

Page Faults

- What happens to a process that references a VA in a page that has been evicted?
  - when the page was evicted, the OS sets the PTE as invalid and stores (in PTE) the location of the page in the swap file
  - when a process accesses the page, the invalid PTE will cause an exception (page fault) to be thrown
  - the OS will run the page fault handler in response
    - handler uses invalid PTE to locate page in swap file
      - With multiple files, how do you know which?
    - handler reads page into a physical frame, updates PTE to point to it and to be valid
    - handler restarts the faulted process
- But: where does the page that's read in go?
  - have to evict something else (page replacement algorithm)
    - OS typically tries to keep a pool of free pages around so that allocations don't inevitably cause evictions

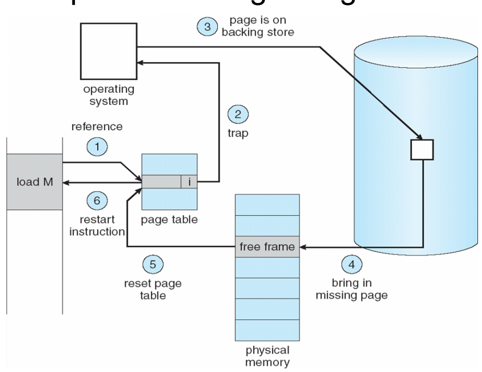
Page Fault

- If there is a reference to a page, first reference to that page will trap to operating system:

    page fault

1. Operating system looks at another table to decide:
   - Invalid reference ⇒ abort
   - Just not in memory
2. Get empty frame
3. Swap page into frame
4. Reset tables
5. Set validation bit = **v**
6. Restart the instruction that caused the page fault

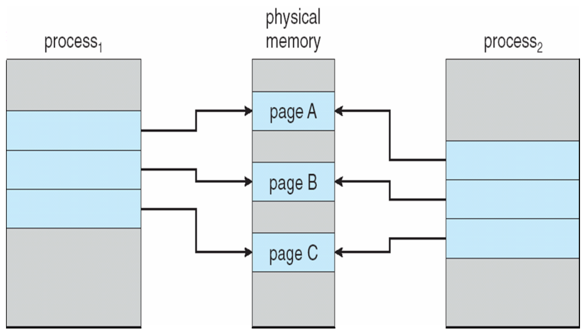
Steps in Handling a Page Fault



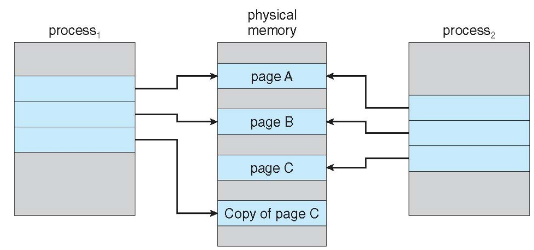
Copy-on-Write

- copy-on-write (COW), e.g. on fork( )
  - instead of copying all pages, created shared mappings of parent pages in child address space
    - make shared mappings read-only in child space
    - when child does a write, a protection fault occurs, OS takes over and can then copy the page and resume client
- Copy-on-Write (COW) allows both parent and child processes to initially *share* the same pages in memory
- If either process modifies a shared page, only then is the page copied
- COW allows more efficient process creation as only modified pages are copied
- Free pages are allocated from a **pool** of zeroed-out pages

Before Process 1 Modifies Page C



After Process 1 Modifies Page C



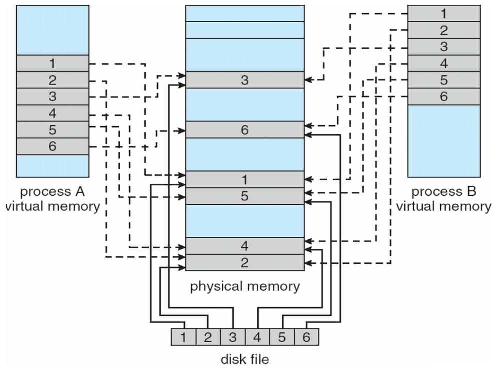
A cool trick

- Memory-mapped files
  - instead of using open, read, write, close
    - “map” a file into a region of the virtual address space
      - e.g., into region with base ‘X’
    - accessing virtual address ‘X+N’ refers to offset ‘N’ in file
    - initially, all pages in mapped region marked as invalid
  - OS reads a page from file whenever invalid page accessed
  - OS writes a page to file when evicted from physical memory
    - only necessary if page is dirty

Memory-Mapped Files

- Memory-mapped file I/O allows file I/O to be treated as routine memory access by **mapping** a disk block to a page in memory
- A file is initially read using demand paging. A page-sized portion of the file is read from the file system into a physical page. Subsequent reads/writes to/from the file are treated as ordinary memory accesses.
- Simplifies file access by treating file I/O through memory rather than **`read()`** **`write()`** system calls
- Also allows several processes to map the same file allowing the pages in memory to be shared

Memory Mapped Files



Evicting the best page

- The goal of the page replacement algorithm:
  - reduce fault rate by selecting best victim page to remove
  - the best page to evict is one that will never be touched again
    - as process will never again fault on it
  - "never" is a long time
    - Belady's proof: evicting the page that won't be used for the longest period of time minimizes page fault rate
- Rest of this lecture:
  - survey a bunch of replacement algorithms

#1: Belady's Algorithm

- Pick the page that won't be used for longest time in future
  - Provably optimal lowest fault rate (remember SJF?)
    - Why?
  - Problem: impossible to predict future
- Why is Belady's algorithm useful?
  - as a yardstick to compare other algorithms to optimal
    - if Belady's isn't much better than yours, yours is pretty good
- Is there a lower bound?
  - unfortunately, lower bound depends on workload
    - but, random replacement is pretty bad

#2: FIFO

- FIFO is obvious, and simple to implement
  - when you page in something, put in on tail of list
  - on eviction, throw away page on head of list
- Why might this be good?
  - maybe the one brought in longest ago is not being used
- Why might this be bad?
  - then again, maybe it is being used
  - have absolutely no information either way
- FIFO suffers from Belady's Anomaly
  - fault rate might increase when algorithm is given more physical memory
    - a very bad property

Example of Belady's Anomaly

3 pages

Page Requests	3	2	1	0	3	2	4	3	2	1	0	4
Newest Page	*3*	*2*	*1*	*0*	*3*	*2*	*4*	4	4	*1*	*0*	0
		3	2	1	0	3	2	2	2	4	1	1
Oldest Page			3	2	1	0	3	3	3	2	4	4

4 pages

Page Requests	3	2	1	0	3	2	4	3	2	1	0	4
Newest Page	*3*	*2*	*1*	*0*	0	0	*4*	*3*	*2*	*1*	*0*	*4*
		3	2	1	1	1	0	4	3	2	1	0
			3	2	2	2	1	0	4	3	2	1
Oldest Page				3	3	3	2	1	0	4	3	2

(red *italics* indicates page fault)

#3: Least Recently Used (LRU)

- LRU uses reference information to make a more informed replacement decision
  - idea: past experience gives us a guess of future behavior
  - on replacement, evict the page that hasn't been used for the longest amount of time
    - LRU looks at the past, Belady's wants to look at future
  - when does LRU do well?
    - when does it suck?
- Implementation
  - to be perfect, must grab a timestamp on every memory reference and put it in the PTE (way too $$)
  - so, we need an approximation…

Approximating LRU

- Many approximations, all use the PTE reference bit
  - keep a counter for each page
  - at some regular interval, for each page, do:
    - if ref bit = 0, increment the counter (hasn't been used)
    - if ref bit = 1, zero the counter (has been used)
    - regardless, zero ref bit
  - the counter will contain the # of intervals since the last reference to the page
    - page with largest counter is least recently used
- Some architectures don't have PTE reference bits
  - can simulate reference bit using the valid bit to induce faults
    - hack, hack, hack

#4: LRU Clock

- AKA Not Recently Used (NRU) or Second Chance
  - replace page that is "old enough"
- Arrange all physical page frames in a big circle (clock)
    - just a circular linked list
  - a "clock hand" is used to select a good LRU candidate
    - sweep through the pages in circular order like a clock
    - if ref bit is off, it hasn't been used recently, we have a victim
      - so, what is minimum "age" if ref bit is off?
    - if the ref bit is on, turn it off and go to next page
  - arm moves quickly when pages are needed
  - low overhead if have plenty of memory
- if memory is large, "accuracy" of information degrades
  - add more hands to fix
- SHOW EXAMPLE!

Another Problem: allocation of frames

- In a multiprogramming system, we need a way to allocate physical memory to competing processes
  - what if a victim page belongs to another process?
  - family of replacement algorithms that takes this into account
- Fixed space algorithms
  - each process is given a limit of pages it can use
  - when it reaches its limit, it replaces from its own pages
  - local replacement: some process may do well, others suffer
- Variable space algorithms
  - processes' set of pages grows and shrinks dynamically
  - global replacement: one process can ruin it for the rest
    - linux uses global replacement

#5: 2nd Chance FIFO

- LRU Clock is a **global** algorithm
  - It looks at all physical pages, from all processes
  - Every process gets its memory taken away gradually
- Local algorithms: run page replacement separately for each process
- 2nd Chance FIFO:
  - Maintain 2 FIFO queues per process
  - On first access, pages go at end of queue 1
  - When the drop off queue 1, page are invalidated and move to queue 2
  - When they drop off queue 2, they are replaced
  - If they are accessed in queue 2, they are put back on queue 1
- Comparison to LRU clock:
  - Per-process, not whole machine
  - No scanning
  - Replacement order is FIFO, not PFN
  - Used in Windows NT, VMS

Important concept: working set model

- A working set of a process is used to model the dynamic locality of its memory usage
  - i.e., working set = set of pages process currently "needs"
  - formally defined by Peter Denning in the 1960's
- Definition:
  - WS(t,w) = {pages P such that P was referenced in the time interval (t, t-w)}
    - t – time, w – working set window (measured in page refs)
    - a page is in the working set (WS) only if it was referenced in the last w references

#6: Working Set Size

- The working set size changes with program locality
  - during periods of poor locality, more pages are referenced
  - within that period of time, the working set size is larger
- Intuitively, working set must be in memory, otherwise you'll experience heavy faulting (thrashing)
  - when people ask "How much memory does Netscape need?", really they are asking "what is Netscape's average (or worst case) working set size?"
- Hypothetical algorithm:
  - associate parameter "w" with each process = # of unique pages referenced in the last "t" ms that it executed
  - only allow a process to start if it's "w", when added to all other processes, still fits in memory
    - use a local replacement algorithm within each process (e.g. clock, 2nd chance FIFO)

Thrashing

- What the OS does if page replacement algo's fail
  - happens if most of the time is spent by an OS paging data back and forth from disk
    - no time is spent doing useful work
    - the system is overcommitted
    - no idea which pages should be in memory to reduced faults
    - could be that there just isn't enough physical memory for all processes
  - solutions?
- Yields some insight into systems research[ers]
  - if system has too much memory
    - page replacement algorithm doesn't matter (overprovisioning)
  - if system has too little memory
    - page replacement algorithm doesn't matter (overcommitted)
  - problem is only interesting on the border between overprovisioned and overcommitted
    - many research papers live here, but not many real systems do…

Summary

- demand paging
  - start with no physical pages mapped, load them in on demand
- page replacement algorithms
  - #1: Belady's – optimal, but unrealizable
  - #2: Fifo – replace page loaded furthest in past
  - #3: LRU – replace page referenced furthest in past
    - approximate using PTE reference bit
  - #4: LRU Clock – replace page that is "old enough"
  - #5: 2nd Chance FIFO – replace local page that is "old enough"
  - #6: working set – keep set of pages in memory that induces the minimal fault rate
- local vs. global replacement
  - should processes be allowed to evict each other's pages?