Review of Memory Management

- The traditional memory hierarchy, the virtual memory abstraction.
- Hardware and software mechanisms to support the abstraction.
- Management policies.
- Where are the opportunities for current research? The underlying assumptions that are changing.

Page Replacement Policy

When there are no free frames available, the OS must replace a page (*victim*), removing it from memory to reside only on disk (*backing store*), writing the contents back if they have been modified since fetched (*dirty*).

Replacement algorithm - goal to choose the best victim, with the metric for "best" (usually) being to reduce the fault rate.

– FIFO, LRU, Clock, Working Set… (defer to later)

The Page Caching Problem (aka Replacement Policy)

- Each thread/process/job utters a stream of page references.
  - Model execution as a *page reference string:* e.g., "abcabcdabce.."
- The OS tries to minimize the number of faults incurred.
  - The set of pages (the *working set*) actively used by each job changes relatively slowly.
  - Try to arrange for the *resident set* of pages for each active job to closely approximate its working set.
- Replacement policy is the key.
  - Determines the resident subset of pages..

Replacement Algorithms

Assume fixed number of frames in memory assigned to this process:

- Optimal - baseline for comparison - future references known in advance - replace page used furthest in future.
- FIFO
- Least Recently Used (LRU)

  stack algorithm - don't do worse with more memory.
- LRU approximations for implementation

  Clock, Aging register

LRU

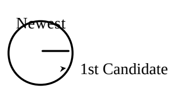
- At fault time: replace the resident page that was last used the longest time ago
- Idea is to track the program's temporal locality
- To implement exactly: we need to order the pages by time of most recent reference
  (per-reference information needed –> HW, too $$)
  - timestamp pages at each ref, stack operations at each ref
- Stack algorithm - doesn't suffer from Belady's anomaly -- if i > j then set of pages with j frames is a subset of set of pages with i frames.

LRU Approximations for Paging

- Pure LRU and LFU are prohibitively expensive to implement.
  - most references are hidden by the TLB
  - OS typically sees less than 10% of all references
  - can't tweak your ordered page list on every reference
- Most systems rely on an *approximation* to LRU for paging.
  - periodically *sample* the reference bit on each page
    - visit page and set reference bit to zero
    - run the process for a while (the *reference window*)
    - come back and check the bit again
  - reorder the list of eviction candidates based on sampling

Clock Algorithm

- Maintain a circular queue with a pointer to the next candidate (clock hand).
- At fault time: scan around the clock, looking for page with usage bit of zero (that's your victim), clearing usage bits as they are passed.
- We now know whether or not a page has been used *since the last time the bits were cleared*



Practical Considerations

- Dirty bit - modified pages require a writeback to secondary storage before frame is free to use again.
- Variation tries to maintain a healthy pool of clean, free frames
  - on timer interrupt, scan for unused pages, move to free pool, initiate writeback on dirty pages
  - at fault time, if page is still in frame in pool, reclaim; else take free, clean frame.

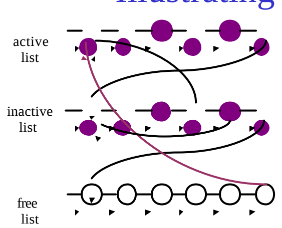
The Paging Daemon

- Most OS have one or more system processes responsible for implementing the VM page cache replacement policy.
  - A *daemon* is an autonomous system process that periodically performs some housekeeping task.
- The *paging daemon* prepares for page eviction before the need arises.
  - Wake up when free memory becomes low.
  - Clean dirty pages by pushing to backing store.
    - *prewrite* or *pageout*
  - Maintain ordered lists of eviction candidates.
  - Decide how much memory to allocate to UBC, VM, etc.

FIFO with Second Chance

- ***Idea***: do simple FIFO replacement, but give pages a "second chance" to prove their value before they are replaced.
  - Every frame is on one of three FIFO lists:
    - *active*, *inactive* and *free*
  - Page fault handler installs new pages on tail of active list.
  - "Old" pages are moved to the tail of the inactive list.
    - Paging daemon moves pages from head of active list to tail of inactive list when demands for free frames is high.
    - Clear the refbit and protect the inactive page to "monitor" it.
  - Pages on the inactive list get a "second chance".
    - If referenced while inactive, *reactivate* to the tail of active list.

Illustrating FIFO-2C



Restock inactive list by pulling pages from the head of the active list: knock off the reference bit and inactivate.

Inactive list scan:
1. Page on inactive list has been referenced? Return to tail of active list (*reactivation*).

2. Page at head of inactive list has not been referenced? *pmap_page_protect* and place on tail of free list.

3. Dirty page on inactive list? Push and return to inactive list tail.

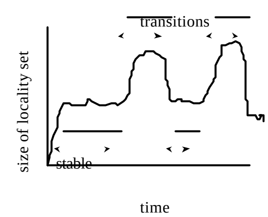
Paging daemon scans a few times per second, even if not needed to restock free list.

Consume frames from the head of the free list.

If free shrinks below threshhold, kick the paging daemon to start a scan.

Variable / Global Algorithms

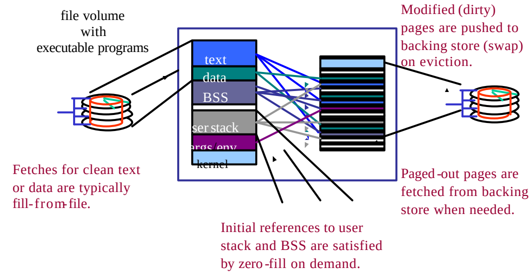
- Not requiring each process to live within a fixed number of frames, replacing only its own pages.
- Can apply previously mentioned algorithms globally to victimize any process's pages
- Algorithms that make number of frames explicit.



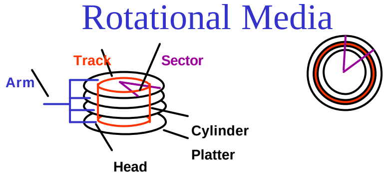
Variable Space Algorithms

- Working Set
  Tries to capture what the set of active pages currently is. The whole working set should be resident in memory for the process to bother running. WS is set of pages referenced during window of time (now-t, now).
  – Working Set Clock - a hybrid approximation
- Page Fault Frequency
  Monitor fault rate, if pff > high threshold, grow # frames allocated to this process, if pff < low threshold, reduce # frames. Idea is to determine the right amount of memory to allocate.

Backing Store = Disk



Rotational Media



Access time = seek time + rotational delay + transfer time

seek time = 5-15 milliseconds to move the disk arm and settle on a cylinder
rotational delay = 8 milliseconds for full rotation at 7200 RPM: average delay = 4 ms
transfer time = 1 millisecond for an 8KB block at 8 MB/s

Bandwidth utilization is less than 50% for any noncontiguous access at a block grain.

Layout issues: clustering

A Case for Large Pages

- Page table size is inversely proportional to the page size
  – memory saved
- Transferring larger pages to or from secondary storage (possibly over a network) is more efficient
- Number of TLB entries are restricted by clock cycle time,
  – larger page size maps more memory
  – reduces TLB misses

A Case for Small Pages

- Fragmentation
  – not *that* much spatial locality
  – large pages can waste storage
  – data must be contiguous within page

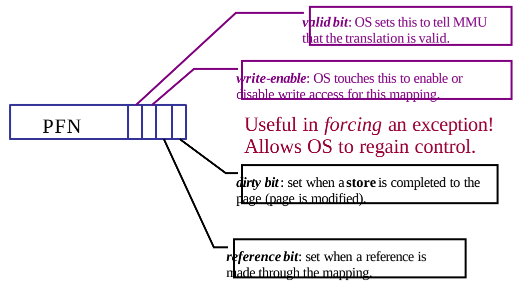
MMU Games

Vanilla Demand Paging

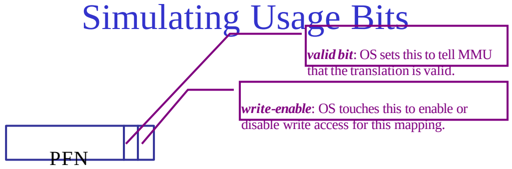
- Valid bit in PTE means non-resident page. Resulting page fault causes OS to initiate page transfer from disk.
- Protection bits in PTE means page should not be accessed in that mode (usually means non-writable)

What *else* can you do with them?

A Page Table Entry (PTE)



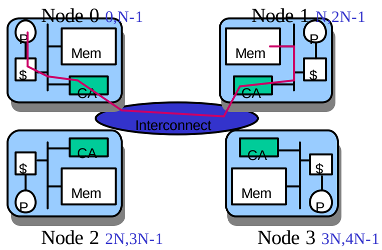
Simulating Usage Bits



- Turn off both valid bit and write-protect bit
- On first reference - fault allows recording the reference bit information by OS in an auxillary data structure. Set it valid for subsequent accesses to go through HW.
- On first write attempt - protection fault allows recording the dirty bit information by OS in aux. data structure.

Copy-on-Write

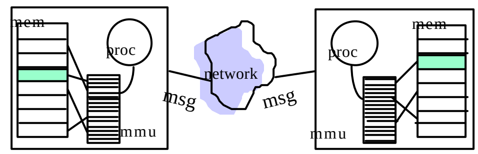
- Operating systems spend a lot of their time copying data.
  - particularly Unix operating systems, e.g., *fork()*
  - cross-address space copies are common and expensive
- *Idea*: defer big copy operations as long as possible, and hope they can be avoided completed.
  - create a new *shadow* object backed by an existing object
  - shared pages are mapped readonly in participating spaces
    - read faults are satisfied from the original object (typically)
    - write faults trap to the kernel
      - make a (real) copy of the faulted page
      - install it in the shadow object with writes enabled



- Each node owns some of physical memory
- OS can allocate physical memory anywhere in system
- May or may not be hardware support for shared memory across nodes (does the MMU recognize remote addresses?)
  - NUMA (non-uniform memory access) vs. distributed memory architectures.

Distributed Shared Memory (DSM)

Allows use of a shared memory programming model (shared address space) in a distributed system (processors with only local memory)



DSM Issues

- Can use the local memory management hardware to generate fault when desired page is not locally present or when write attempted on read-only copy.
- Locate the page remotely - current "owner" of page (last writer) or "home" for page.
- Page sent in message to requesting node (read access makes copy; write migrates)
- Consistency protocol - invalidations or broadcast of changes (update)
  - directory kept of caches holding copies

DSM States

Forced faults are key to consistency operations

- Invalid local mapping, attempted read access - data flushed from most recent writer, set write-protect bit for all copies.
- Invalid local mapping, attempted write access - migrate data, invalidate all other copies.
- Local read-only copy, write-fault - invalidate all other copies

Consistency Models

- Sequential consistency
  - All memory operations *appear* to execute one at a time. A write is considered *done* only when invalidations or updates have propagated to all copies.
- Weaker forms of consistency
  - Guarantees associated with synchronization primitives; at other times, it doesn't matter
  - For example:
    acquire lock - make sure others' writes are done
    release lock - make sure all my writes are seen by others

70