

Outline for Today's Lecture

- Administrative:
 - Exam next Tuesday
 - Review in Discussion Sessions this week – come armed with questions.
- Objective for today:
 - Replacement policies

Policies for Paged Virtual Memory

The OS tries to minimize page fault costs incurred by all processes, balancing fairness, system throughput, etc.

- ✓ (1) **fetch policy**: When are pages brought into memory?
 - prepaging: reduce page faults by bring pages in before needed
 - on demand: in direct response to a page fault.
- (2) **replacement policy**: How and when does the system select victim pages to be evicted/discarded from memory?
- ✓ (3) **placement policy**: Where are incoming pages placed? Which frame?
- ✓ (4) **backing storage policy**:
 - Where does the system store evicted pages?
 - When is the backing storage allocated?
 - When does the system write modified pages to backing store?
 - Clustering: reduce seeks on backing storage

Paged Virtual Memory

- General notion of a cache:
 - copies of data temporarily moved into storage of faster, higher cost, lower capacity technology
 - Achieving a high hit ratio gives performance equivalent to having most of memory built using this \$\$ technology
- The paging system manages physical memory as a **cache** over a larger virtual address space.
 - Data “lives” on disk, and a *copy* is in physical memory only while in active use.
 - Hardware and OS software cooperate to maintain the illusion that the machine’s memory “looks like” the virtual memory.
 - The OS controls data placement/movement and establishes the set of translations in effect at any time.
 - The VM abstraction is (mostly) transparent to user code.

Demand Paging

- Missing pages are loaded from disk into memory at *time of reference (on demand)*.
The alternative would be to prefetch into memory in anticipation of future accesses (need good predictions).
- Page fault occurs because *valid bit* in page table entry (PTE) is *off*. The OS:
 - allocates an empty frame*
 - reads the page from disk
 - updates the PTE when I/O is complete
 - restarts faulting process

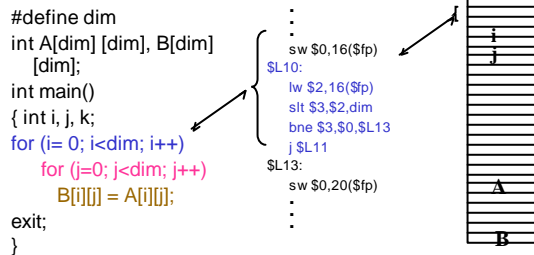
*Page Replacement

- 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.

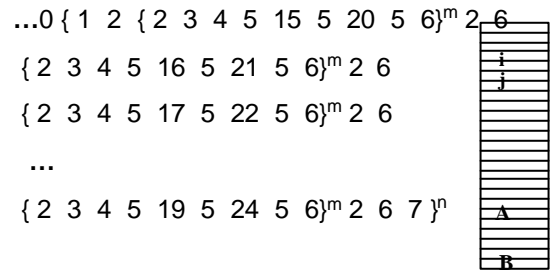
Terminology

- 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..

Example (Artificially Small Pagesize)



Example Reference String



Assessing Replacement Algs

- ✓ Model program execution as a reference string
- ✓ Metric of algorithm performance is **fault rate**
- Comparison to base line of **Optimal Algorithm**.
- For a specific algorithm: What is the information needed? How is that information gathered? When is it acted upon?
 - At each memory reference
 - At fault time
 - At periodic intervals

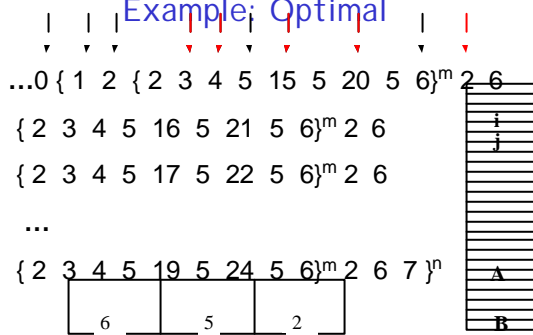
"What did he know and when did he know it?"
 (history lesson - 1972-74)

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

Example: Optimal



VAS

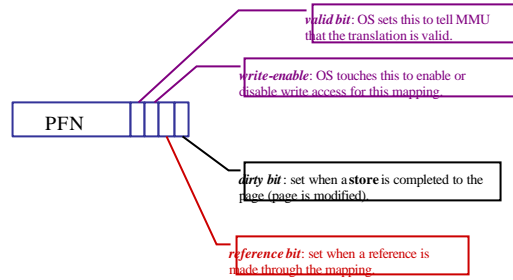
FIFO

- No extra hardware assistance needed, No per-reference overhead (we have no information about actual access pattern)
- At fault time: maintain a first-in first-out queue of pages resident in physical memory
- Replace oldest resident page
- Why it might make sense - straight-line code, sequential scans of data
- Belady's anomaly - fault rate can increase with more memory

Approximations to LRU

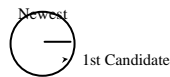
- HW support: usage/reference bit in every PTE - set on each reference.
 - Set in TLB entry
 - Written back to PTE on TLB replacement
- We now know whether or not a page has been used *since the last time the bits were cleared*.
 - Algorithms differ on when & how that's done.
 - Unless it's on every reference, there will be "ties"

A Page Table Entry (PTE)



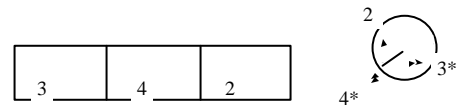
Clock Algorithm

- 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*



Example Reference String

$\downarrow \downarrow \downarrow \downarrow \downarrow \downarrow$
 $\dots 0 \{ 1 \ 2 \ \{ 2 \ 3 \ 4 \ 5 \ 15 \ 5 \ 20 \ 5 \ 6 \}^m \ 2 \ 6$
 $\{ 2 \ 3 \ 4 \ 5 \ 16 \ 5 \ 21 \ 5 \ 6 \}^m \ 2 \ 6$
 $\{ 2 \ 3 \ 4 \ 5 \ 17 \ 5 \ 22 \ 5 \ 6 \}^m \ 2 \ 6 \dots$



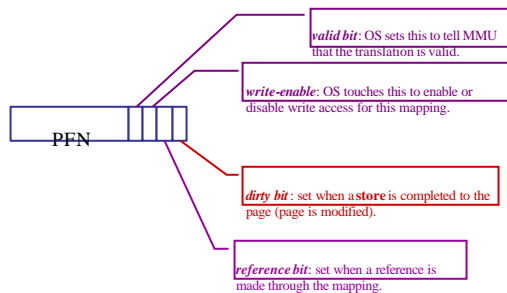
Approximating a Timestamp

- Maintain a supplemental data structure (a counter) for each page
- Periodically (on a regular timer interrupt) gather info from the usage bits and zero them.
for each page i {if (used _{i}) counter _{i} = 0; else counter _{i} ++; used _{i} = 0;}
- At fault time, replace page with largest counter value (time intervals since last use)

Practical Considerations

- Dirty bit - modified pages require a writeback to secondary storage before frame is free to use again.
- Variation on Clock 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.

A Page Table Entry (PTE)



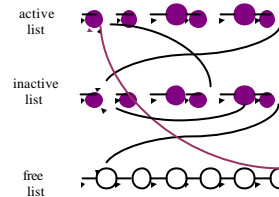
The Paging Daemon

- 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 (Mach)

- **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? *page_protect* and place on tail of free list.
3. Dirty page on inactive list? Push to disk and return to inactive list tail.

Consume frames from the head of the free list.

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

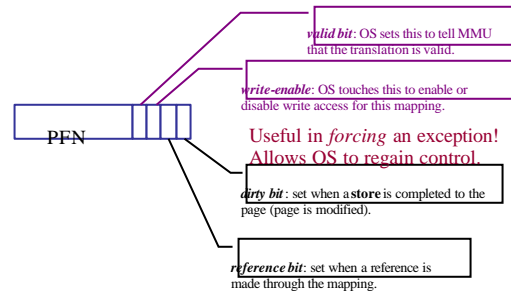
MMU Games

Vanilla Demand Paging

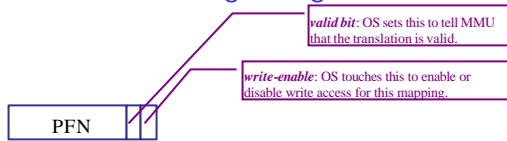
- Valid bit in PTE means non-resident page. Resulting page fault causes OS to initiate page transfer from disk.
- Protection bits in PTE means page should not be accessed in that mode (usually means non-writable)

What *else* can you do with them?

A Page Table Entry (PTE)



Simulating Usage Bits



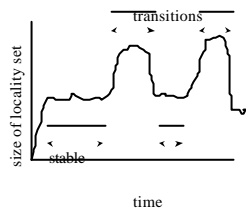
- Turn off both valid bit and write-protect bit
- On first reference - fault allows recording the reference bit information by OS in an auxiliary data structure. Set it valid for subsequent accesses to go through HW.
- On first write attempt - protection fault allows recording the dirty bit information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

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.



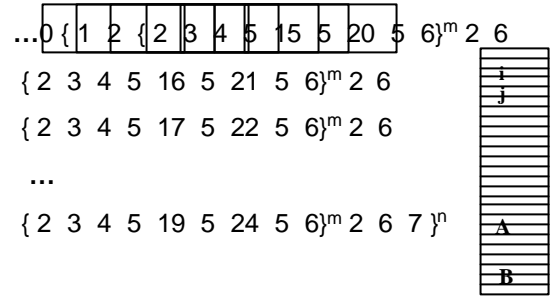
Variable Space Algorithms

- **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.

Working Set Model

- Working set at time t is the set of pages referenced in the interval of time $(t-w, t)$ where w is the working set **window**.
 - Implies per-reference information captured.
 - How to choose w ?
- Identifies the “active” pages. Any page that is resident in memory but not in any process's working set is a candidate for replacement.
- Size of the working set can vary with locality changes

Example Reference String

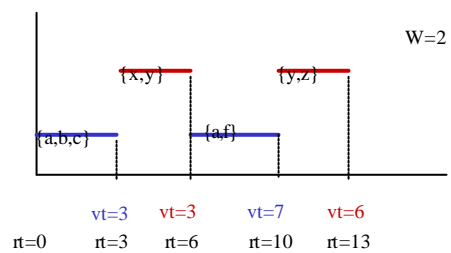


WSClock

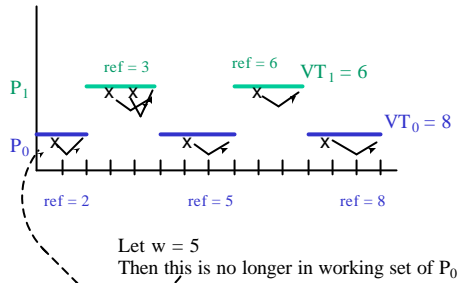
- The implementable approximation
- At fault time: scan usage bits of resident pages. For each page i

```
{if (usedi) {time_of_refi = vtowner[i] /*virtual time of
owning process*/; usedi = 0;}
else if ( |vtowner[i] - time_of_refi| >= w )
replaceable; //else still in "working set"}
```

WSClock

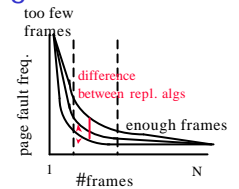


WSClock (the picture)



Thrashing

- Page faulting is dominating performance
- Causes:
 - Memory is overcommitted - not enough to hold locality sets of all processes at this level of multiprogramming
 - Lousy locality of their programs
 - Positive feedback loops reacting to paging I/O rates



- Load control is important (how many slices in frame pie?)
 - LT/RT strategy (limit # processes in load phase)

Pros and Cons of VM

- Demand paging gives the OS flexibility to manage memory...
 - programs may run with pages missing
 - unused or "cold" pages do not consume real memory
 - improves degree of multiprogramming
 - program size is not limited by physical memory
 - program size may grow (e.g., stack and heap)
- ...but VM takes control away from the application.
 - With traditional interfaces, the application cannot tell how much memory it has or how much a given reference costs.
 - Fetching pages on demand may force the application to incur I/O stalls for many of its references.