Page Replacement Policies

General caching problem: which entry should be removed from the cache to make room for another?

When should page replacement policy be invoked:

- on a page fault
- independent of page faults (a paging daemon)

Min Policy (Belady's algorithm), Optimal Policy

Discard the page whose next access is farthest in the future.

- ideal policy: best possible choice
- cannot be implemented
- yardstick for comparing other policies

Least Recently Used

Use (recent) past as a predictor of the future:

- discard page that has not been accessed in longest time
- tough to implement exactly:
  - list of pages must be kept in sorted list by reference time, or
  - page must be stamped with time of reference or some type of increasing counter.

However, good approximations are possible.

Not Recently Used (NRU)

Approximate time stamp with 1-bit clock:

- if bit is true, page was referenced "recently"
- otherwise, it was referenced "significantly earlier"

One implementation:

- hardware supports a *reference* bit (similar to modified bit, but set on every page reference)
- Periodically (either on page faults or by time or on clock interrupt), cycle through the pages, clearing the reference bit.
- when the page fault handler needs a frame, it cycles through the page table looking for pages with a false reference bit

  Four classes (pick a page from the lowest numbered class). Approach shown in Tanenbaum.

  1. R=0, M=0
  2. R=0, M=1
  3. R=1, M=0
  4. R=1, M=1

Clock Algorithm

To increase fairness (and improve performance) start search for candidate page where previous search ended—Clock Algorithm. This algorithm does not divide pages into classes nor does it use the modify bit in making decisions (although a such an algorithm could be constructed).

Like advancing a clock hand through the pages.

```
if (R == 0)
    replace
else
    reset R
advance hand
```

FIFO Page Replacement

In the first in first out (FIFO) policy, the memory manager swaps out the oldest page, regardless of how recently it was accessed.

Evaluation

Inclusion (Stack) Property

Page-reference strings can be used to evaluate page replacement policies. Usually one counts the number of page faults for a given reference string, page replacement policy, and number of frames.

A *fault-rate graph* plots the number of page faults vs. the number of page frames.

Cold-start behavior refers to starting with an empty cache (e.g., pure demand paging).

Warm-start behavior refers to that part of the fault-rate graph after the cache has been loaded. Fill up initial frames.

The *characteristic number* is a single number that summarizes the page fault frequency; it is given by the area under the curve of the graph. Other analysis is given in Tanenbaum.

Does an increase in the number of pages in memory decrease the number of page faults a process generates?

Consider the reference string (sequence of pages a process references: 0 1 2 3 0 1 4 0 1 2 3 4

If physical store holds 3 pages, 9 faults occur (3 for cold start faults, 6 warm faults). If we increase the physical size to 4 pages, 10 faults occur (4 cold, 6 warm).

Anomalies can occur only in policies that lack the *stack property* or *inclusion property*. The inclusion property requires that:

- for any page reference string, the set of pages that would be in an n-frame store is a subset of those that would be in an (n+1)-frame store

FIFO does not satisfy the inclusion property, while LRU (among others) does.

Locality of Reference

Another way of viewing the problem. Rather than LRU and its approximations, try to take into account locality of reference.

Experiments show that some pages are accessed more frequently than others. In particular, programs exhibit a *locality of reference*:

- on stack machines, programs access pages near the top of the stack more frequently than those at the bottom
- programs executing loops continually fetch the same instructions
- sequential access of elements of an array (bring up an example of a 1024x1024 array—access each row or each column). Consider:

```
{
    int a[1024][1024];
    int i, j;

    for (i = 0; i < 1024; i++)
        for (j = 0; j < 1024; j++)
            a[i][j] = 6;    // arbitrary value
}
```

Locality of reference exploits observation that at any time a few pages are in *active* use, while others are unreferenced. A *phase change* occurs when the set of pages in active use changes.

Working Set

The *working set* policy (an ideal):

- estimates the number of pages a process needs to proceed at a reasonable rate.
- limits the number of processes on the ready list so that physical memory can accommodate all working sets
- other "ready" processes are kept in a "memory-wait" state
- when a process is swapped back into memory, the pages in its working set may be brought in (eliminates cold start). This is called *prepaging*, and is the opposite of demand paging

The working set is defined as follows:

- maintain the reference string over *virtual time*, the time a process is actually executing.
- a processes working set is the set of pages referenced in the previous w units of virtual time
- w is tunable *working size* parameter of the policy

Simulations show that the working set policy works well. In practice, it is difficult to implement exactly.

Page-Fault Frequency

The *page-fault frequency* policy approximates the working set policy.

Decisions made only when a page fault occurs

```
Check when the last fault occurred;
If (last fault occured more than {\em p\/} units ago) {
    // phase shift assumed
    Discard all pages not referenced since last page fault;
}
else {
    // building up
    Add page to the working set.
}
Reset all referenced bits.
```

A discarded page has its present bit set to false, but is not swapped out. It is added to the pool of good candidates to swap out.

If a fault occurs, the page may still be in the pool.

Global vs. Local Policies

So far, we have considered single process systems where there is only one page table. Multiple processes can be accommodated by giving each process a quota of frames and applying a policy individually to each process.

Local policies resolve competition for frames local to a process.

Global policies have processes competing against each other for frames.

Global Policies

Why global policies?

- one process may need more memory than another
- guessing the right per-process allocation difficult (a process' working set)
- blocked processes hold memory usable by another process

Problems with global policies:

- process that has been away from the ready list for a while experiences many page faults (cold start problem)
- number of processes on ready list may be so large that no process has enough memory
- processes that have been executing for a while keep referencing their pages, while other processes (perhaps blocked waiting for a page) lose even more of theirs

For global policies, scan frame table (rather than page table) for candidate pages to swap. Usually, the frame table is not directly supported by the hardware, so the OS maintains one in software.

Sharing Pages

Can think of allowing sharing of pages between processes. Useful, but causes problems.

- Who owns a frame?
- How to gain access to information for a shared frame?

How to approach? A central mapping table to use for keeping track of shared information?

Locking Pages in Memory

Some pages cannot safely be swapped out:

- frames being accessed by devices performing DMA. The OS must *tie down* such frames in memory
- parts of the operating system (e.g., page fault handler!)

Backing Store

Have an explicit swap area on disk. Sometimes the executable itself is used, which can result in the message "text file busy" when recompiling.

Memory Mapped Files

Memory mapped files cause file contents to be mapped into memory and all accesses to file and are done as memory operations, which may cause page faults. Files is used as backing store.

Merges caching done by file I/O with that done for virtual memory.