Replacement policies for multiprogramming

Hardware support for page replacement algorithms

- Enhanced Second Chance
- Second Chance
- LRU approximations

Today

and improve performance

A good page replacement algorithm can reduce the number of page faults

- Switch occurs.
- Processes can share memory more efficiently, reducing the costs when a context switch occurs.
- Processes start faster because they only need to load a few pages (for code and data).
- Processes can run without being fully loaded into memory.
- Virtual address space can be larger than physical address space.

Benefits of demand paging:

Last Class: Demand Paged Virtual Memory
Why

- With LRU, increasing the number of frames always decreases the number of page faults.

<table>
<thead>
<tr>
<th>Frame 1</th>
<th>Frame 2</th>
<th>Frame 3</th>
<th>Frame 4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
</tr>
</tbody>
</table>

LRU:

Adding Memory with LRU

With FIFO, the contents of memory can be completely different with a different number of page frames.

<table>
<thead>
<tr>
<th>Frame 1</th>
<th>Frame 2</th>
<th>Frame 3</th>
<th>Frame 4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
</tr>
</tbody>
</table>

FIFO:

Does adding memory always reduce the number of page faults?

Adding Memory
Page fault still requires a search through all the pages.

- Faster, since setting a single bit on each memory access.
- Approximate, since it does not guarantee a total order on the pages.
- On a page fault, the lowest numbered page is kicked out:
  - High order bit:
  - All regular entries on each memory access, shift the byte right, placing a 0 in the
  - Additional Reference-Bits: Maintain more than 1 bit, say 8 bits.

Hardware Requirements: Maintain reference bits with each page.

Implementations of LRU

- Problems: still too expensive, since the OS must modify 6 pointers on each
  memory access (in the worst case)
  - The LRU page is the least recently used.
  - On a page access, move the page to the front of the list. Doubly link the list.
  - 2. Keep a list of pages, where the front of the list is the most recently used page, and
  - LRU page.
  - Problems: OS must record time stamp for each memory access, and to throw out
  - a page that the OS may not look at all pages. Exponential overflow.
  - I. Keep a time stamp for each page with the time of the last access. Throw out the

Perf ect LRU:

- All implementations and approximations of LRU require hardware

Implementations of LRU:
What if all bits are 1?

Will it always find a page?

Simple hardware requirements.

- a 0, reference bit.
- Page fault if faster, since we only search the pages until we find one with

shift. Fast, since setting a single bit on each memory access, and no need for a

the algorithm.

indicates if the page was used at all since the last time it was checked by

Less accurate than additional-reference-bits, since the reference bit only

Second Chance Algorithm

Second Chance Algorithm (a.k.a. Clock)

1. OS keeps frames in a circular list.

2. On a page fault, the OS

(a) Checks the reference bit of the next frame.
(b) If the reference bit is 0, replace the page, and set its bit to 1.
(c) If the reference bit is 1, set bit to 0, and advance the pointer to the next frame.
0: page is the same as the copy on disk
1: page is modified (different from the copy on disk)

• Hardware keeps a modify bit (in addition to the reference bit)

OS can give preference to paging out un-modified pages
OS need not write the page back to disk

It is cheaper to replace a page that has not been written

Enhanced Second Chance

Why not partition pages into more than two categories?

You are and old pages

One way to view the clock algorithm is as a crude partitioning into two categories:

Clock Example
By the third pass, all the pages will be at (0, 0).

If the page is being written out, wait for the I/O to complete and then remove the
changed to (0, 0) page.

On the second pass, a page that was originally (0, 0) or (1, 0) might have been
for pages with the reference bit set, the reference bit is cleared
until the I/O completes. Clear the modified bit and continue the search
in memory.

I. Page with (0, 0) ⇐ replace the page.
2. Page with (0, 1) ⇐ iterate an I/O to write out the page, locks the page in memory
3. Page with (1, 1) ⇐ iterate an I/O to write out the page, waits for the I/O to complete, and then remove the
changed to (0, 0) page.

The OS goes around at most three times searching for the (0, 0) class.

Page Replacement in Enhanced Second Chance

Page Replacement

On a page fault, the OS searches for the first page in the lowest
nonempty class.

Write it, or else replace it

write out the page, but it might not be needed anymore.
write out the page, but it might not be needed anymore.
write out this page, but it might not be needed anymore.
write out this page, but it might not be needed anymore.
write out the page, but it might not be needed anymore.
write out the page, but it might not be needed anymore.
write out the page, but it might not be needed anymore.
write out the page, but it might not be needed anymore.
write out the page, but it might not be needed anymore.
write out the page, but it might not be needed anymore.
write out the page, but it might not be needed anymore.
write out the page, but it might not be needed anymore.
write out the page, but it might not be needed anymore.
write out the page, but it might not be needed anymore.
write out the page, but it might not be needed anymore.
write out the page, but it might not be needed anymore.
write out the page, but it might not be needed anymore.
write out the page, but it might not be needed anymore.
write out the page, but it might not be needed anymore.
write out the page, but it might not be needed anymore.
write out the page, but it might not be needed anymore.
write out the page, but it might not be needed anymore.
write out the page, but it might not be needed anymore.
write out the page, but it might not be needed anymore.
write out the page, but it might not be needed anymore.
write out the page, but it might not be needed anymore.
write out the page, but it might not be needed anymore.
write out the page, but it might not be needed anymore.
write out the page, but it might not be needed anymore.
write out the page, but it might not be needed anymore.
write out the page, but it might not be needed anymore.
write out the page, but it might not be needed anymore.
write out the page, but it might not be needed anymore.
write out the page, but it might not be needed anymore.
write out the page, but it might not be needed anymore.
write out the page, but it might not be needed anymore.
write out the page, but it might not be needed anymore.
write out the page, but it might not be needed anymore.
Disadvantages: Throttling might become even more likely (why?)

Advantages: Flexible, adjusts to diverse process needs

What can we do in a multiprogrammed environment to limit thrashing?

- Results in a serious and very noticeable loss of performance
- Causes page faults
- Memory access times approach disk access times since many memory references tossed out while they are still in use
- Throttling: the memory is over-committed and pages are continuously

Replacement Policies for Multiprogramming

Clock Example
Advantages: A threshold is less likely to process only completed with itself.

- May need to suspend a process until overall memory demands decrease.
- The time it takes to handle a page fault.

Goal: The system-wide mean time between page faults should equal to each process instead.

Working sets are expensive to compute = track page fault frequency of

Per-Process Replacement

Replacement Policies For Multiprogramming

- What happens if \( T \) is too small? too big?
  \( T \) needs to be a whole lot bigger than 2 million instructions.
  \( T \) is 2 million instructions.
  \( T \) page fault = 10 msec

How does the OS pick T?

- More formally, it is the set of all pages that a process referenced in the past 1 seconds.
  Informally, the working set is the set of pages the process is using right now.
  "Working" set size?

How do we figure out how many pages a process needs, i.e., "its working"

- Run only a group of processes that fit in memory, and kick out the rest.

Per-Process Replacement: Each process has its own pool of pages.
A critical issue the OS must decide is how many processes and the frames can have.

The more processes running concurrently, the less physical memory each process approaches the virtual memory size.

All algorithms approach optimal as the physical memory allocated to a insufficient physical memory (less than half of their virtual address space).

Experiments show that all algorithms do poorly if processes have Unix and Linux use variants of Clock, Windows NT uses FIFO.

Summary of Page Replacement Algorithms

- Performance
  - Larger slow down than they used to. Reducing the number of page faults is critical to CPU speed is increasing faster than disk speed. As a result, page faults result in a
  - Also, internal fragmentation is less of a concern with abundant memory.
  - Physical memory is cheap. As a result, page tables could get huge with small pages.

Page sizes are growing because:

- Fewer page faults (few processes that exhibit locality of reference)
- Amortizes disk overheads over a larger page
- Simpler page tables

Reasons for large pages:

- Higher degree of multiprogramming possible.
- More effective memory use.

Reasons for small pages:

Page Sizes