Page Replacement Algorithms

- What happens if a process needs a new page, but all of the frames in physical memory are in use?
- Reducing the frequency of page faults is critical to OS performance. What page replacement algorithms are best at reducing the number of page faults?
- A slow page replacement algorithm will also affect OS performance adversely. What page replacement algorithms are most efficient to implement?
- As usual, high speed requires hardware support. What hardware support is required to support the page replacement algorithms?

Page Replacement Algorithms

On a page fault, we need to choose a page to kick out.

Random: this algorithm works pretty well.

FIFO: First-In, First-Out. Throw out the oldest page.
Simple to implement, but the OS can easily throw out a page that is being accessed frequently.

MIN: (a.k.a. OPT) Look into the future and throw out the page that will be accessed farthest in the future (provably optimal [Belady66]). Problem?

LRU: Least Recently Used. Approximation of MIN that works well if the recent past is a good predictor of the future. Throw out the page that has not been used in the longest time.

These replacement algorithms are used in caches and TLBs as well. Many TLBs actually use Random.

Examples

3 page Frames
4 virtual Pages: A B C D
Reference stream: A B C A B D A D B C B

FIFO: First-In-First-Out

	A	B	C	A	B	D	A	D	B	C	B
frame 1											
frame 2											
frame 3											

Number of page faults?

MIN: Look into the future and throw out the page that will be accessed farthest in the future.

	A	B	C	A	B	D	A	D	B	C	B
frame 1											
frame 2											
frame 3											

Number of page faults?

Examples

LRU: Least Recently Used. Throw out the page that has not been used in the longest time.

	A	B	C	A	B	D	A	D	B	C	B
frame 1											
frame 2											
frame 3											

Number of page faults?

When will LRU perform badly?

	A	B	C	D	A	B	C	D	A	B	C	D
frame 1												
frame 2												
frame 3												

Adding Memory

Does adding memory always reduce the number of page faults?

FIFO:

	A	B	C	D	A	B	E	A	B	C	D	E
frame 1												
frame 2												
frame 3												
frame 1												
frame 2												
frame 3												
frame 4												

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

Adding Memory with LRU

LRU:

	A	B	C	D	A	B	E	A	B	C	D	E
frame 1												
frame 2												
frame 3												
frame 1												
frame 2												
frame 3												
frame 4												

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

Implementing LRU:

All implementations and approximations of LRU require hardware assistance.

Perfect LRU:

1. Keep a time stamp for each page with the time of the last access. Throw out the LRU page.

   Problems: OS must record time stamp for each memory access, and to throw out a page the OS has to look at all pages. Expensive!

2. Keep a list of pages, where the front of the list is the most recently used page, and the end is the least recently used. On a page access, move the page to the front of the list. Doubly link the list.

   Problems: still too expensive, since the OS must modify 6 pointers on each memory access (in the worst case).

Approximations of LRU

Hardware Requirements: Maintain reference bits with each page. On each access to the page, the hardware sets the reference bit to '1'. Set to 0 at varying times depending on the page replacement algorithm.

Additional-Reference-Bits: Maintain more than 1 bit, say 8 bits. At regular intervals (every 100 ms, for example) shift the byte right, placing a 0 in the high order bit. On a page fault, the lowest numbered page is kicked out.

- Approximate, since it does not guarantee a total order on the pages.
- Faster, since setting a single bit on each memory access.
- Page fault still requires a search through all the pages.

Second chance algorithm: (a.k.a. Clock)

Use a single reference bit per page.

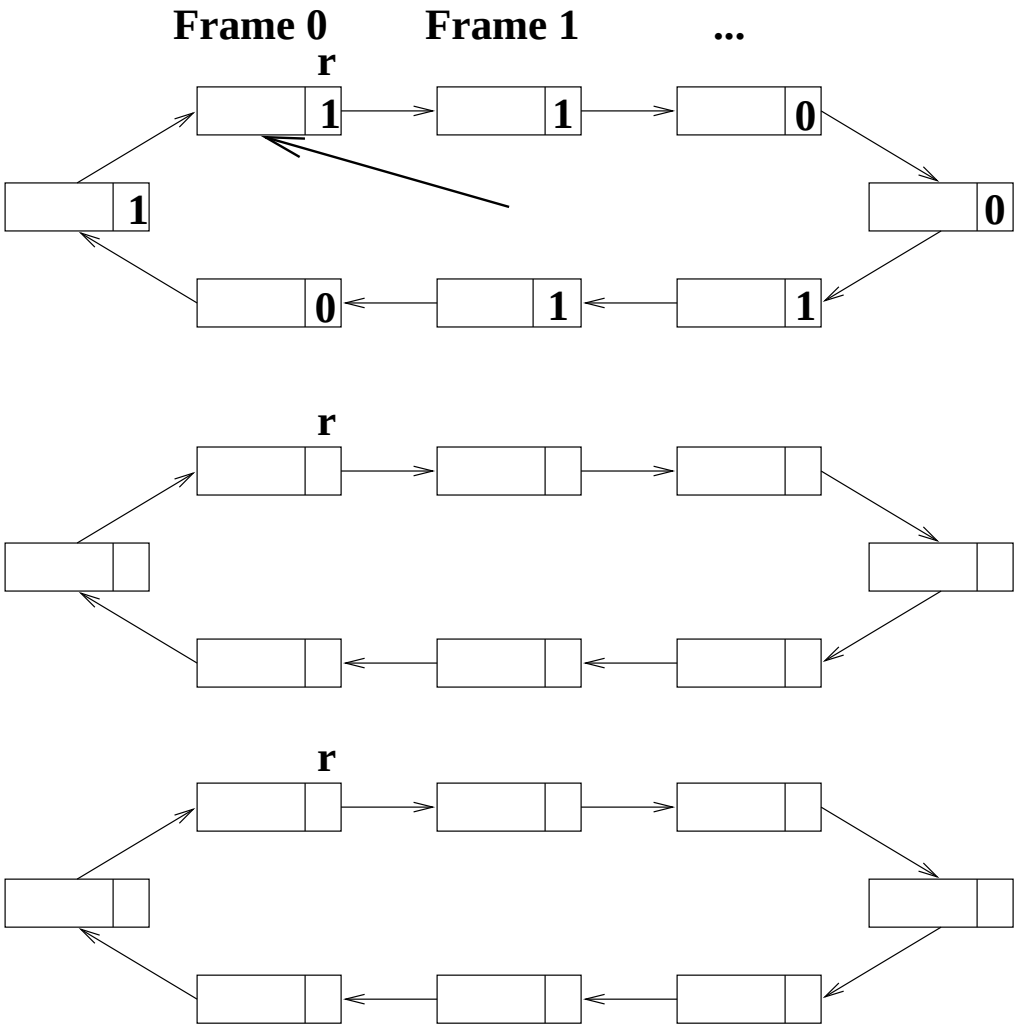
1. OS keeps frames in a circular list.
2. On a page fault, the OS
   (a) Checks the reference bit of the next frame.
   (b) If the reference bit is '0', replace the page, and set its bit to '1.'
   (c) If the reference bit is '1', set bit to '0', and advance the pointer to the next frame.

- Less accurate than additional-reference-bits, since the reference bit only indicates if the page was used at all since the last time it was checked by the algorithm.
- Fast, since setting a single bit on each memory access, and no need for a shift.
- Page fault is faster, since we only search the pages until we find one with a '0' reference bit.
- Simple hardware requirements.

Will it always find a page?

What if all bits are '1'?

Clock Example



$\Longrightarrow$ One way to view the clock algorithm is as a crude partitioning into two categories: young and old pages.

Why not partition pages into more than two categories?

In practice...

- Unix and Linux use variants of the Clock algorithm.
- Windows NT uses FIFO replacement.
- Experimentation shows that all algorithms do poorly on typical processes if processes have insufficient physical memory (less than half of their virtual address space).
- All algorithms approach optimal as the physical memory allocated to a process approaches the virtual memory size.
- The more processes running concurrently, the less physical memory each process can have.
- A critical issue the OS must decide is how many processes and may share memory simultaneously (how many frames per process).

Replacement Policies for Multiprogramming

Thrashing: the memory is over-committed and pages are tossed out while they are still in use. Instead of the illusion of RAM speed, memory references take disk time since many memory references require a page to be faulted in. The result is a serious and very noticeable loss of performance. This problem was severe in early systems, e.g., the VAX.

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

Proportional allocation: allocate more page frames to large processes.

Global replacement: put all pages from all processes in one pool so that the physical memory associated with a process can grow

Advantages: Flexible, adjusts to divergent process needs

Disadvantages: Thrashing can become even more likely. Why?

Replacement Policies for Multiprogramming

Per-process replacement: Each process has its own pool of pages.

- Run only groups of processes that fit in memory, and kick out the rest.
- How do we figure out how many pages a process needs, i.e., its working set size?
- The working set is informally the pages the process is using right now. More formally, it is the set of all pages that a process referenced in the past T seconds.
- How does the OS pick T?

  1 page fault = 10msec

  10msec = 2 million instructions

  So T needs to be a whole lot bigger than 2 million instructions.

  What happens if T is too small? too big?

Replacement Policies for Multiprogramming

Per-process replacement:

- Working sets are expensive to compute.
- Can track page fault frequency of each process.
- If the page fault frequency is above some threshold, give it more page frames.
- If the page fault frequency is below a second threshold, take away some page frame.
- Goal: the system-wide mean time between page faults should be equal to the time it takes to handle a page fault.
- Might be necessary to suspend a process until overall memory demands decrease.

Per-process replacement:

Advantages: Thrashing less likely as process only competes with itself. More consistent performance independent of system load.

Disadvantages: The OS has to figure out how many pages to give each process and if the working set size grows dynamically adjust its allocation.