Last Class: Demand Paged Virtual Memory

Benefits of demand paging:

- Virtual address space can be larger than physical address space.
- Processes can run without being fully loaded into memory.
  - Processes start faster because they only need to load a few pages (for code and data) to start running.
  - Processes can share memory more effectively, reducing the costs when a context switch occurs.
- A good page replacement algorithm can reduce the number of page faults and improve performance

Today

- Page Replacement
- LRU behavior
- LRU approximations:
  - Second Chance
  - Enhanced Second Chance
- Hardware support for page replacement algorithms
- Replacement policies for multiprogramming

Page Replacement Algorithms

On a page fault, we need to choose a page to evict

Random: amazingly, 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 [Belady'66]). 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.

Example: FIFO

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?

Example: MIN

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?

Example: LRU

- **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?

Example: LRU

- When will LRU perform badly?

	A	B	C	A	B	D	A	D	B	C	B
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 or on each memory access, 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

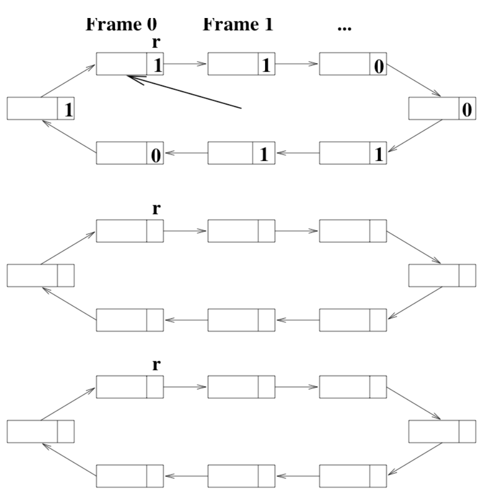
Second Chance Algorithm

- 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



=> 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?

Enhanced Second Chance

- It is cheaper to replace a page that has not been written
  - OS need not be write the page back to disk

  => OS can give preference to paging out un-modified pages

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

  '1': page is modified (different from the copy on disk)

  '0': page is the same as the copy on disk

Enhanced Second Chance

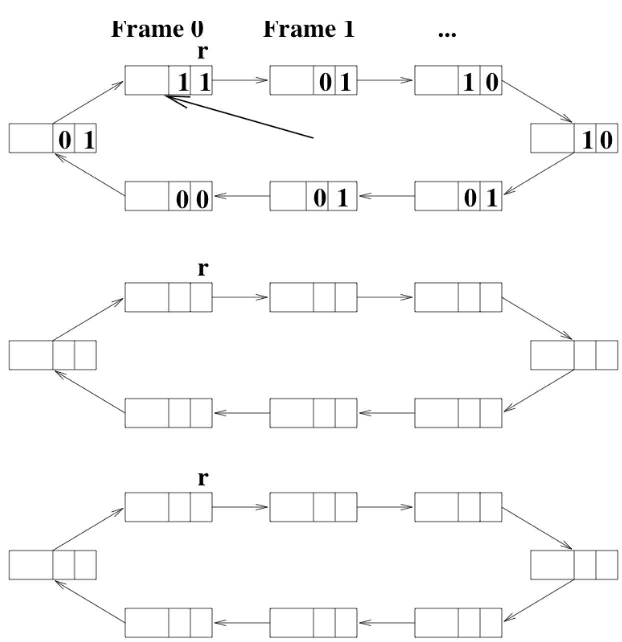
- The reference bit and modify bit form a pair (r,m) where
  1. (0,0) neither recently used nor modified - replace this page!
  2. (0,1) not recently used but modified - not as good to replace, since the OS must write out this page, but it might not be needed anymore.
  3. (1,0) recently used and unmodified - probably will be used again soon, but OS need not write it out before replacing it
  4. (1,1) recently used and modified - probably will be used again soon and the OS must write it out before replacing it

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

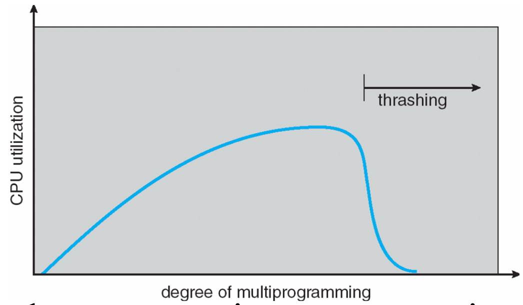
Page Replacement in Enhanced Second Chance

- The OS goes around at most three times searching for the (0,0) class.
  1. Page with (0,0) => replace the page.
  2. Page with (0,1) => initiate an I/O to write out the page, locks the page in memory until the I/O completes, clears the modified bit, and continue the search
  3. For pages with the reference bit set, the reference bit is cleared.
  4. If the hand goes completely around once, there was no (0,0) page.
     - On the second pass, a page that was originally (0,1) or (1,0) might have been changed to (0,0) => replace this page
     - If the page is being written out, waits for the I/O to complete and then remove the page.
     - A (0,1) page is treated as on the first pass.
     - By the third pass, all the pages will be at (0,0).

Clock Example



Multiprogramming and Thrashing



- **Thrashing:** the memory is over-committed and pages are continuously tossed out while they are still in use
  - memory access times approach disk access times since many memory references cause page faults
  - Results in a serious and very noticeable loss of performance.
- What can we do in a multiprogrammed environment to limit thrashing?

Replacement Policies for Multiprogramming

- **Proportional allocation:** allocate more page frames to large processes.
  - alloc = s/S * m
- **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 might become even more likely (Why?)

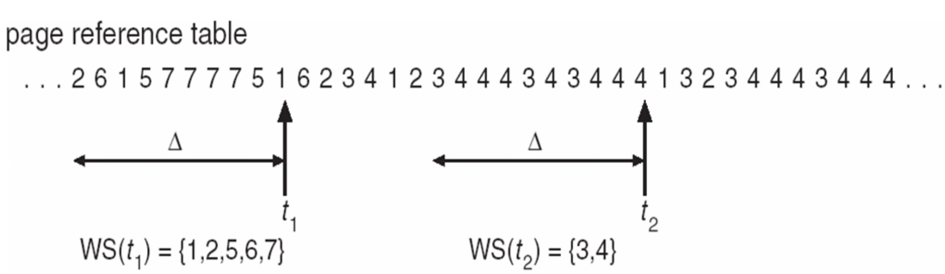
Replacement Policies for Multiprogramming

- **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?
  - Informally, the working set is the set of 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

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

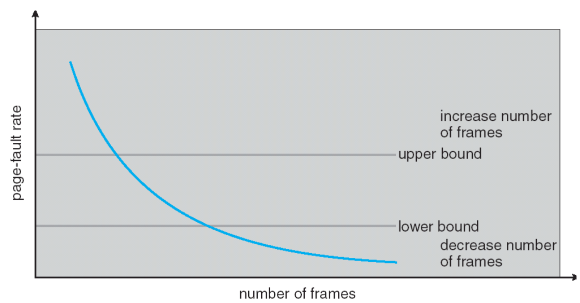
  - What happens if T is too small? too big?

Working Set Determination



Per-process Replacement

- Working sets are expensive to compute => track page fault frequency of each process instead
  - If the page fault frequency > some threshold, give it more page frames.
  - If the page fault frequency < a second threshold, take away some page frames
- **Goal:** the system-wide mean time between page faults should be equal to the time it takes to handle a page fault.
  - May need to suspend a process until overall memory demands decrease.



Page-fault Frequency Scheme

- **Advantages:** Thrashing is 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.

Kernel Memory Allocators

- Buddy allocator
  - Allocate memory in size of 2^n
  - Can lead to internal fragmentation
- Slab allocator
  - Group objects of same size in a "slab"
  - Object cache points to one or more slabs
  - Separate cache for each kernel data structure (e.g., PCB)
  - Used in solaris, linux

Page Sizes

- Reasons for small pages:
  - More effective memory use.
  - Higher degree of multiprogramming possible.
- Reasons for large pages:
  - Smaller page tables
  - Amortizes disk overheads over a larger page
  - Fewer page faults (for processes that exhibit locality of references)
- Page sizes are growing because:
  - Physical memory is cheap. As a result, page tables could get huge with small pages. Also, internal fragmentation is less of a concern with abundant memory.
  - CPU speed is increasing faster than disk speed. As a result, page faults result in a larger slow down than they used to. Reducing the number of page faults is critical to performance.

Summary of Page Replacement Algorithms

- Unix and Linux use variants of Clock, Windows NT uses FIFO.
- Experiments show that all algorithms do poorly 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 the frames per process that may share memory simultaneously.