CS 390
Chapter 10 Homework Solutions

10.1 `Under what circumstances do page ...`

A page fault occurs when a process generates a logical address, and that address is on a page that is not resident in physical memory.

When a page fault occurs, the memory management hardware generates a trap. The kernel inspects the logical address and determines which page is not in memory. It then finds a free frame, either by consulting the free-frame list, or evicting a resident page, if no frames are free. It then moves the process to the wait queue, and schedules an IO operation to copy the page from the disk or backing store (which are usually the same thing) into the free frame. When the page becomes resident in memory, the kernel updates the process's page table, abd the process is added to the ready queue. When the process is dispatched, the faulting instruction is restarted.

10.2 `Assume that you have a ...`

a. The lower bound is n, the number of distinct page numbers. Each page will always be faulted for at least once, when it is first brought into memory.

b. If $m \geq n$ (the number of frames is greater than number of distinct pages) then the upper bound on the number of page faults is n. We fault once for each page.

If $m < n$ (that is, if the number of frames is less than the number of distinct pages) then the upper bound is p. A really bad page replacement algorithm might fault for every page. For example, consider what would happen if a program was to linearly access a very large array over and over again and the kernel was using an LRU page replacement algorithm.

10.3 `Consider the following page-replacement ...`

From best to worst:

c. Optimal replacement 5. Does not suffer from Belady's.

a. LRU replacement 4. Does not suffer from Belady's.

d. Second-chance replacement 3. Suffers from Belady's.

b. FIFO replacement 2. Suffers from Belady's.

You will have to trust my assertions about Belady's anomaly.

Bonus question: Can you think of a page-replacement algorithm that would generate the **worst** possible page-fault rate for any reference string?

10.5 `Consider the page table for ...`

- `0EF`
- `211`
- `D00` (page fault, page 7 placed in frame D)
- `EFF` (page fault, page 0 placed in frame E)

10.6 `Discuss the hardware functions required ...`

At a minimum, demand paging needs hardware to support the logical to physical address translation. Required hardware includes a translation look-aside buffer to hold part of the page table. In addition, each page table entry must have a valid bit.

While it is not required, additional bits such as a dirty bit and a reference bit can simplify the implementation of the page replacement algorithm. If the architecture supports DMA I/O, a lock bit is required to make sure that a page being used for I/O is not selected as a victim by the page replacement algorithm.

Finally, the machine must implement restartable instructions.

10.9 `Consider the following page reference ...`

Using LRU page replacement, the page fault rate is 90%. Using FIFO page replacement, the page fault rate is 85%. Using OPT page replacement, the page fault rate is 65%.

Additional Exercises

A Suppose that a system uses pure demand paging. How would you characterize the page fault rate when the process first begins to execute? How would you characterize the page fault rate when the working set of the process has been captured in memory?

B What is the copy-on-write feature, and under what circumstances is its use beneficial?

C Assume that we have a demand-paged memory. The page table is held in a TLB. It takes 8 ms to service a page fault if an empty frame is available or if the victim page is not modified, and 20 ms if the victim page is modified and needs to be saved. Memory-access time is 100 ns.

Assume that the page to be replaced is modified 70% of the time. What is the maximum acceptable page-fault rate for an effective access time of no more than 200 ns?

D Assume that you are monitoring the rate at which the pointer in the second-chance victim selection algorithm moves. (The pointer indicates the candidate page that the algorithm is considering for replacement.) What can you say about the system if you notice that the pointer is moving fast? If the pointer is moving slowly?

E Consider a demand-paging system with the following time-measured utilizations:

- CPU utilization: 20%
- Paging disk: 97.7%
- Other I/O devices: 5%

For each of the following, indicate whether it will or is likely to improve CPU utilization. Explain your answers.

- Upgrade to a faster CPU.
- Install a larger paging disk.
- Increase the degree of multi-programming.
- Decrease the degree of multi-programming.
- Install more main memory.
- Install a faster paging disk.
- Replace the paging algorithm with one that pre-pages more of the process.
- Increase the page size.

F What is the cause of thrashing? How does the system detect thrashing? What can the kernel do to eliminate thrashing, once it has been detected?

G Calculate the working sets generated by this reference string: 1, 2, 3, 4, 5, 1, 6, 7, 2, 6, 7, 1, 5, 1, 2 from time t_0 to t_{14} for a window size of 3. Repeat with a window size of 5.

Answers to additional exercises

A If the system is using pure demand paging, the page fault rate will be very high when the process first starts, because none of the pages that the process needs will be in memory.

If the paging algorithm is successful in capturing the working set of the process in memory, the page fault rate will be 0%.

B Copy-on-write is a kernel feature that takes advantage of a system's paging hardware to avoid copying the contents of a frame when a page needs to be duplicated. When a page copy is requested, the OS creates a new page table entry for the copied page.

However, this entry doesn't point to a new frame. Instead, it points to the frame that holds the original page. Thus, two page table entries point to the same frame. These page table entries can either be in the same process or different processes. The process or processes access the "copied" page just like normal memory, and the kernel translates the different logical addresses into the same physical address.

This scheme breaks down when a process attempts to write to the memory. At this point, the kernel must physically copy the page into a new frame and update the corresponding page table. Whether a page needs to be copied when it is written to is indicated by the COW bit in the page table entry.

C If p is the page fault rate and ma is the memory access time, then the effective access time is defined as

$$\text{EAT} = (1-p) \cdot \text{ma} + p \cdot \text{page fault service time}$$

In this example, the page fault service time is

$$\text{PFST} = 8 \text{ msecs} \cdot 0.3 + 20 \text{ msecs} \cdot 0.7$$

Converting to nanoseconds gives

$$\text{EAT} = (1-p) \cdot 100\text{nsecs} + p \cdot (8000000 \text{ nsecs} \cdot 0.3 + 20000000 \text{ nsecs} \cdot 0.7)$$

Set EAT to 200 nsecs and solve for p, giving $p \leq 6.098 \cdot 10^{-6}$.

D If the pointer is moving fast, then the page replacement algorithm must be running frequently. The page replacement algorithm only runs when there are no free frames. Thus, the system must be running under memory over-commit, and page faults must be happening frequently.

If the pointer is moving slow, then the page replacement algorithm must be running infrequently. Again, the system must be running under memory over-commit, since the clock hand would not be moving at all if there was no contention for memory. However, in this case, page faults are infrequent, which indicates that the system has successfully captured in memory most of the pages that processes are referencing.

E

- Install a faster CPU. Unlikely to improve CPU utilization. The current CPU is underutilized. A faster one won't help.
- Install a bigger paging disk.

  Unlikely to improve CPU utilization. The size of the backing store can effect the maximum degree of multiprogramming that the system can support, since a bigger backing store can hold more swapped-out pages. However, that does not seem to be the problem here. The backing store is very busy, and a bigger disk won't help that.
- Increase the degree of multiprogramming.

  Very unlikely to improve CPU utilization. It appears this system is thrashing. Increasing the degree of multiprogramming is the exact wrong thing to do.
- Decrease the degree of multiprogramming.

  Likely to improve CPU utilization. It appears that jobs are queued up, waiting on the backing store to serve their page faults. Swapping out entire jobs would allow the remaining processes to be allocated enough frames to reduce their page fault rates so they can actually spend time executing on the

CPU, and thus finish. Once those processes finish, memory will be freed up, and the kernel can begin gradually swapping processes back in.

- Install more main memory.

  Likely to improve CPU utilization. More memory would allow us to allocate more frames to each process, hopefully capturing more of each process's locality in memory, thus reducing its page-fault rate.

- Install a faster paging disk.

  May slightly improve CPU utilization. A faster disk may reduce page-fault service time, leading to a decrease in effective access time and higher CPU throughput.

- Replace the paging algorithm with one that pre-pages more of the process.

  Prepaging is unlikely to increase CPU utilization. The problem with this system is that too many pages are fighting over too few frames. Bringing more pages into memory won't solve the problem.

- Increase the page size.

  If data and code are accessed sequentially, a large page size might capture more of a process's locality in memory, reducing the page fault rate and thus increasing the CPU utilization.

  However, if the process does not access memory sequentially, a larger page size is unlikely to improve CPU utilization. A larger page size would tend to capture more code and data than is contained in the process's current locality, meaning code and data which are not being used are stored in memory, wasting memory and increasing the page fault rate. For example, imagine a process that is traversing a massive tree in a breadth-first manner from root to leaves. Parent and child nodes of this tree are unlikely to reside on the same page. To the kernel, the process will appear to be accessing memory in a non-linear fashion.

F The memory management algorithms of the kernel try to allocate enough frames to each process so that they can execute without

frequent page faults. Thrashing occurs when the sum of all the memory needs of all the processes on the ready queue is greater than the amount of physical memory; processes can only execute a few instructions before they fault for a page.

Since there are no unused frames, a frame must be stolen from another process, causing that process to fault. Another page is stolen, causing another process to fault, and so on, in a domino effect. Since no process has the correct page set in memory to execute for more than a few instructions, the system is spending all its time servicing page faults, and little real work is being accomplished.

Thrashing can be detected by monitoring CPU utilization and the system page-fault rate. A high page-fault rate coupled with a low CPU utilization indicates thrashing.

Thrashing can be eliminated by swapping out entire processes until the page-fault rate returns to normal.

G

t	$\Delta = 3$	$\Delta = 5$
t_0	$\{1\}$	$\{1\}$
t_1	$\{1, 2\}$	$\{1.2\}$
t_2	$\{1, 2, 3\}$	$\{1, 2, 3\}$
t_3	$\{2, 3, 4\}$	$\{1, 2, 3, 4\}$
t_4	$\{3, 4, 5\}$	$\{1, 2, 3, 4, 5\}$
t_5	$\{1, 4, 5\}$	$\{1, 2, 3, 4, 5\}$
t_6	$\{1, 5, 6\}$	$\{1, 3, 4, 5, 6\}$
t_7	$\{1, 6, 7\}$	$\{1, 4, 5, 6, 7\}$
t_8	$\{2, 6, 7\}$	$\{1, 2, 5, 6, 7\}$
t_9	$\{2, 6, 7\}$	$\{1, 2, 6, 7\}$
t_{10}	$\{2, 6, 7\}$	$\{2, 6, 7\}$
t_{11}	$\{1, 6, 7\}$	$\{1, 2, 6, 7\}$
t_{12}	$\{1, 5, 7\}$	$\{1, 2, 5, 6, 7\}$
t_{13}	$\{1, 5\}$	$\{1, 5, 6, 7\}$
t_{14}	$\{1, 2, 5\}$	$\{1, 2, 5, 7\}$