CS 390
Chapter 9 Homework Solutions

9.2 `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 will 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 a pure LRU page replacement algorithm.

9.5 `Discuss the hardware support required ...`

At a minimum, demand paging requires hardware to support the logical to physical address translation. Required hardware includes a translation look-aside buffer to hold page table entries. In addition, each page must have an associated valid bit (to determine if the page is in memory).

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. In addition, if the architecture supports DMA I/O, a lock bit is required.

Finally, the machine must implement restartable instructions, which will complicate the CPU design.

9.15 `A simplified view of thread ...`

a. Yes, the thread will change to the blocked state. The thread will remain blocked until the faulted-for page is in memory, when it will become ready.

b. No, the thread state will not change. A TLB miss is one of the few places in the kernel where an interrupt does not need to send a process either to the ready queue or the wait queue. When the TLB miss occurs, we trap to the kernel. The kernel simply fetches the page table entry from memory, loads it into the TLB, and restarts the instruction. If the kernel is carefully designed, it may not even have to perform a full-fledged context switch to service a TLB miss.

   The x86 architecture contains a special privileged instruction that loads a page table entry into the TLB without using any general purpose registers of the CPU. This allows a TLB miss to be handled without a context switch.

c. No, thread state will not change. In a system using paging, every process address reference (which is a logical address after all) must be resolved through the page table, and this resolution is done in hardware. (See part .)

9.16 `Consider a system that uses ...`

a. If the system is using pure demand paging, the page fault rate will be very high because none of the pages that the process needs will be in memory. If the system is using pre-paging, and the pre-paging algorithm guesses correctly, the page fault rate will be low.

b. If the working set is in memory, the page fault rate will be 0%.

c. The process needs more memory than is currently available. The system designer could:

   1. Swap the entire process out until enough memory is available.
   2. Allow the process to steal pages from other processes. However, since no free memory is available, this could mean that the total current memory demand is greater than the amount of physical memory. If so, stealing pages will lead to thrashing and wreck the performance of every process in the system.
   3. Allow the process to continue to execute, but not allocate any more frames to it. The process will have a very high

page fault rate and will appear to execute very slowly, but the other processes on the system will continue to execute normally (except when they fault).

9.17 `What is the copy-on-write ...`

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.

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.

Since COW is implemented using demand paging, paging hardware is required. In addition, the system must be able to intercept writes to a COW page (usually through a trap) before the write hits memory. Thus, we need some way of indicating that a page is a COW page. This can be implemented using an additional bit in the page table entry, or by marking the page as unwritable, if the page table has a 'W' bit.

9.18 `A certain computer provides its users ...`

When the process generates the logical address, the memory management hardware calculates the page number and offset from the address. Since the page size is a power of two, this can be done by dividing the address into two parts: the least-significant 12 bits ($4096 = 2^{12}$) give the offset, and the most significant 20 bits ($20 = 32 - 12$) give the page number.

The page number is sent to the paging hardware which looks it up in the TLB. If the page is not in the TLB, a request for the correct page table entry is sent to memory.

Once the page-table entry is found and loaded into the TLB, the valid bit is checked. If the page is valid, the frame number is combined

with the offset and the physical address is sent to memory. These steps occur in hardware.

If the page number is not valid, a trap occurs. The page fault service routine locates a free frame (possibly running the page replacement algorithm and writing the victim to disk), and then loads the desired page into memory from the backing store. The valid bit of the process is then updated, and the entry for the page is added to the TLB. These steps are accomplished in software.

9.19 `Assume that we have a demand-paged ...`

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 $p \leq 6.098 \cdot 10^{-6}$.

9.21 `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%.

9.23 `Assume that you are monitoring the rate ...`

a. `pointer is moving fast`

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.

b. `pointer is moving slow`

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.

9.28 `Consider a demand-page system ...`

a. `Install a faster CPU.`

Unlikely to improve CPU utilization. The current CPU is underutilized. A faster one won't help.

b. `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.

c. `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.

d. `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.

e. `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.

f. `Install a faster hard disk or multiple controllers with multiple hard disks.`

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

g. `Add prepaging to the page-fetch algorithms.`

We did not discuss "prepaging" in class. In a system with prepaging, a page fault will cause the faulted-for page to be paged in as usual. In addition (under the assumption that memory is being accessed sequentially) the kernel will also page-in several of the pages that come immediately after the faulted-for page.

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

h. `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 from root to leaf. Parent and child nodes of this tree are unlikely to reside on the same page. T the kernel, the process will appear to be accessing memory in a non-linear fashion.

9.32 `What is the cause of thrashing? ...`

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 processes until the remaining processes' memory needs can be met.

9.a Suppose that a process generates the following string of page references: `0,4,5,0,6,7,1,4,5,1,2,3,6,7`.

Calculate the working sets generated by this reference string from time t_0 to t_{13} for a window size of

1. 3
2. 5

t	$\Delta = 3$	$\Delta = 5$
t_0	$\{0\}$	$\{0\}$
t_1	$\{0, 4\}$	$\{0, 4\}$
t_2	$\{0, 4, 5\}$	$\{0.4, 5\}$
t_3	$\{0, 4, 5\}$	$\{0, 4, 5\}$
t_4	$\{0, 5, 6\}$	$\{0, 4, 5, 6\}$
t_5	$\{0, 6, 7\}$	$\{0, 4, 5, 6, 7\}$
t_6	$\{1, 6, 7\}$	$\{0, 1, 5, 6, 7\}$
t_7	$\{1, 4, 7\}$	$\{0, 1, 4, 6, 7\}$
t_8	$\{1, 4, 5\}$	$\{1, 4, 6, 7\}$
t_9	$\{1, 4, 5\}$	$\{1, 4, 5, 7\}$
t_{10}	$\{1, 2, 5\}$	$\{1, 2, 4, 5\}$
t_{11}	$\{1, 2, 3\}$	$\{1, 2, 3, 4, 5\}$
t_{12}	$\{2, 3, 6\}$	$\{1, 2, 3, 5, 6\}$
t_{13}	$\{3, 6, 7\}$	$\{1, 2, 3, 6, 7\}$