8.18 [M] Give a critique of the assumption made in Example 8.1, in Section 8.7.1, that the miss penalty is the same for both read and write accesses. Consider both the write-through and write-back cases, as described in Section 8.6, in formulating your answer.

8.7.1 : Hit Rate and Miss Penalty

write-through:

- both the cache & main memory location updated

\+ simple

\- unnecessary writes

write-back:

- update only the cache location, marking the block containing it as dirty (i.e modified).
- main memory updated only once the dirty block is evicted from the cache

\+ faster

8.1 : access :

cache : τ

main : 10τ

miss : block of B transferred from main memory to cache : 10τ for first, then remaining 7 in τ with initial access delay τ

$\therefore$ miss penalty, $M = \tau + 10\tau + 7\tau + \tau = 19\tau$

Assumptions:

1. 30% of the instructions are R/W
2. hit rates cache : 0.95, data : 0.9
3. same miss penalty for both R/W

read

$$\frac{\text{Time without cache}}{\text{Time with cache}} = \frac{130 \times 10\tau}{100(0.95\tau + 0.05 \times 19\tau) + 30(0.9\tau + 0.1 \times 19\tau)} = 4.7$$

ideal

$$\frac{\text{Time for real cache}}{\text{Time for ideal cache}} = \frac{100(0.95\tau + 0.05 \times 19\tau) + 30(0.9\tau + 0.1 \times 19\tau)}{130\tau} = 2.1$$

Critique:

the assumptions are an oversimplification for easier calculations. It fails due to the fact that cpu has SEPARATE R/W paths, thus the "3." assumption is not true.

The R/W penalty depends heavily on the cache to main writing policy.

a write miss doesn't require memory to be fetched, thus this is immediately smaller than a read miss. additionally a write hit is slower than a read hit as the path to main memory is further than cache. In terms of write back & write through, write through miss vs read miss is lower than the write back miss vs read miss penalty.

Because of write back & write through behavior stated above, there are some critical flaws in the assumption

Matt Krueger

8.19 [M] Consider a computer system in which the available pages in the physical memory are divided among several application programs. The operating system monitors the page transfer activity and dynamically adjusts the number of pages allocated to various programs. Suggest a suitable strategy that the operating system can use to minimize the overall rate of page transfers.

Page transfers i.e. page faults are costly to the computer system as memory from secondary memory such as a disk or SSD must be accessed. This is costly as it must pause the running process, locate the missing page on disk (slow memory access time) to load it into ram using an algorithm such as LRU to evict a page.

After doing some research, I found two potential strategies to minimize the overall rate of page transfers:

1. Working set model: exploits locality by using a working set window, 'Δ' This works by tracking the set of pages actively used in Δ. Note that Δ is a short time window. There must be enough memory to hold the working set to avoid faults. If the total demand is greater than the available memory then we reduce multiprogramming.

2. Page fault frequency: monitors rate of faults
This works by monitoring the frequency of page faults for each process currently being run. If this is low, then we reclaim the pages and distribute them to other processes. If frequency is high, we give it more pages. Our goal is to minimize the number of pages per processes, keeping processes just below critical fault rate. This makes sense as resources for processes differs. A clear example of this would be a large, volatile program versus a smaller, less volatile program. Obviously, these shouldn't be allocated the same resources by the computer system.

Learning comes from:

YT - Academia - #24, #28

8.20 [M] In a computer with a virtual-memory system, the execution of an instruction may be interrupted by a page fault. What state information has to be saved so that this instruction can be resumed later? Note that bringing a new page into the main memory involves a DMA transfer, which requires execution of other instructions. Is it simpler to abandon the interrupted instruction and completely re-execute it later? Can this be done?

Yes, we can save state of the current instructions being processed by the cpu in the 5-stage pipeline. Interruptions from page faults should prompt the computer to save:

1. program counter → to resume execution
2. Register contents → save state
3. CC's → if instructions in interrupt affect cc's.
4. pipeline state → save state
5. Page table & TLB state → save state

Yes, it is simpler if we can just restart the instruction - IF the instruction CAN be restarted. An example of an exception to restarting is the Auto increment/decrement operations.

Yes, this can be done, but with caution.

8.21 [E] When a program generates a reference to a page that does not reside in the physical main memory, execution of the program is suspended until the requested page is loaded into the main memory from a disk. What difficulties might arise when an instruction in one page has an operand in a different page? What capabilities must the processor have to handle this situation?

Having the operands split between physical and virtual memory is problematic if the memory from virtual space hasn't been loaded into main yet. Obviously this isn't always avoidable. Two possible difficulties are:

1. mid instruction page fault. (after fetch of inst.). Fault!
2. Auto increment/decrement may reach across into memory that's currently loaded. Fault!

To handle these. The processor must be built to handle restartable instructions, (or fully execute or completely rollback current instruction... similar to say RDBs).
Additionally, processor should maintain robust page tables