SOLUTION NOTES

Operating Systems 2002 Paper 1 Question 11 (GMB)

This is also Paper 10 Question 7 ("Operating System Foundations").

(*a*) **Paging** [Slides 81–89] Paging allows a process to reside in non-contiguous memory. We divide physical memory into small blocks of fixed size called frames. Logical memory is divided into blocks of the same size called pages. Each address generated by the CPU is composed of a page number p and an offset o. The memory management unit (MMU) then uses p as an offset into a page table. The page table entry contains an associated frame number f. (We often have a valid bit in the page table entry to signify whether the frame is loaded in memory or not.)

(*b*) **Disadvantages** [Slide 82] The lecture notes mentioned three disadvantages of paging:

(*1*) The OS needs to keep a page table per process.

(*2*) We get internal fragmentation (up to a max of a frame size, typically 4K, so not normally a huge problem nowadays).

(*3*) The page table represents an additional overhead on context switching.

Another disadvantage (allowed) is that without a TLB paging implies that every "actual" memory reference now requires two memory accesses! (Hence the next part of the question!)

(*c*) **TLB** [Slide 84] The TLB is a fast piece of associative memory, i.e. it's a little table where the key entries are checked *simultaneously*. It's expensive, and hence small (maybe 64 entries) and mirrors part of the page table. As before, each address consists of a page number p and an offset o. Now the TLB is searched first with p. If there's a hit, we use the page table entry from the TLB and access memory. The point is here that we do only a TLB search and a memory access to perform an "actual" memory reference. If there isn't a hit then we have to look in the page table as before (and hence have two memory accesses *and* the TLB search). The black art is making sure that the TLB is an effective cache of the page table, i.e. the hit rate is high.

(*d*) [This is not in the notes, but I went through it in the lecture, and it's pretty obvious from the previous part.] The equation for the average memory access time (AvMAT) is as follows:

$$\text{AvMAT} = \text{HR} * (\text{TLBST} + \text{MAT}) + (1 - \text{HR}) * (\text{TLBST} + 2 * \text{MAT})$$

where HR is the hit ratio (as a probability), TLBST is the TLB search time, and MAT is the memory access time.

Thus the problem boils down to solving the following equation:

$$120 = \text{HR} * (20 + 80) + (1 - \text{HR}) * (20 + 80 + 80)$$

Thus HR = 0.75, or 75%.

(*e*) **Multi-level page tables** [Slides 85–87] Nowadays the logical address space can be big, e.g. 2^{32} or 2^{64}. Thus if a page is 4K, i.e. 12 bits for an offset, then we have 20 bits for the page, i.e. 2^{20} entries in the page table, which means that the page table is *huge* (4MB)! The solution is to page the page table.

Thus imagine the 64 bit machine, with pages of size 4K. The essential fact is that the **page table ought to fit one page**. Thus as the frames will be 4KB we use 12 bits for the offset. In a one-level paging system this would leave the page table with up to 2^{52} entries. We could then move to a two-level paging system, where the inner page table could contain 2^{10} 4-byte entries, which would leave the outer page table will 2^{42} entries. Thus we need to break the outer page table, and so create a second outer page, i.e. the logical address is of the form (p_1, p_2, p_3, o) where p_1 is 32 bits, p_2 and p_3 are 10 bits and o is 12 bits. Of course, the outer page table is still too big, and we have to page it.

The problem is that we'll end up with 6 or 7 (depending how it is arranged) levels of paging – i.e. potential 6 or 7 memory accesses to translate a logical address, which is prohibitively expensive.