CMPSC 473: Fall 2009: Solutions to Quiz 1, 2

Bhuvan Urgaonkar

October 20, 2009

1. *Assuming 4kB to save information about one function call, what is the largest integer whose factorial can be calculated by a recursive program whose address space is 4MB in size? Ignore space occupied by code segment.*

   $$factorial(n) = n * factorial(n-1), n > 1$$

   $factorial(1) = 1$

   The stack would only be able to hold 1,000 function frames. So the largest integer would be 1,000.

2. *An event is synchronous if it can only happen at a pulse of the clock driving the CPU. Which of the following is synchronous? interrupt arrival, beginning of interrupt handling, trap occurrence, beginning of trap handling.*

   Only the arrival of an interrupt is asynchronous (it can arrive at any time from outside the CPU and is not generated at a clock unlike a trap).

3. *Assume a program with 1M instructions. Of these 40% refer to a data item or an instruction in memory. Assuming one cycle per instruction fetched to the CPU and ignoring hardware caches, how many cycles does the program need to finish if there is no TLB?*

   Instructions that refer to a data item or insturction in memory would require an extra instruction to fetch the virtual-to-physical mapping from memory. These would take 1M*0.4*2=0.8M cycles. The remaining instructions need 0.6M cycles. So the total number of cycles would be 0.8+0.6=1.4M.

4. *Think of two ways a kernel could detect a malfunctioning timer interrupt mechanism.*

   First way: Run a piece of code with a predictable relationship with the number of instructions it would execute within a given amount of time. Then compare this count with that expected based on the time reported by the timer.

   Second way: Send messages to an external entity with access to a functional timer and compare the time elapsed between these with that reported by our timer.

5. *We discussed the need to flush the TLB upon context switch on certain processors. What would you change about the TLB for it not to have to be flushed upon a context switch?*

   We would use what are called *tagged TLBs*, where each translation entry within the TLB also has bits to indicate which process (i.e., address space) that entry corresponds to.

6. *What do you think happens when a data item needed by a process is not found in main memory? I am looking for an answer in terms of the events/abstractions we have studied so far.*

   A trap is generated which causes the OS to run and fetch these missing memory contents needed by the process from a secondary storage device (typically a disk) into main memory. This kind of trap is called a *page fault* and we will study it when we get to memory management.

7. *Recall user-level threads. Why is it safe to let a user switch the CPU? (Hint: what abstraction that we have studied ensures this safety and who enforces it?)*

   It is safe to do so because of the notions of context switching and virtual memory. A user-level piece of code may fill up the registers with any value (e.g., using something like `setjmp`), but the OS guarantees that this will not affect the register contents seen by any other process. A good example is the program counter register. A process may fill any address into this register, but it would be a virtual address *within its own address space* and would be safely translated into a physical memory address assigned to this process.

8. *Process scheduling is usually described as selection from a queue of processes. For each of these scheduling methods, describe the selection method, the insertion method (if required), further properties of the queue (if required), and whether a simple queue should be replaced by a more sophisticated data structure.*

   Round-robin time-sharing: Selection method just requires finding the next process in a circular linked list.

   Shortest-job-first: Consider two data structures. If the queue is maintained as a linked list, selection requires a scan to find the shortest job. If it is maintained as a heap (not to be confused with a heap within an address space) or a binary tree, selection merely picks the "top" of this data structure; the data structure must be updated after each scheduling or arrival/departure of a process (log(n) work where n is the number of processes).

   Multilevel feedback methods in general: Selection is usually from the most important queue. Work has to be done at each scheduler invocation as well as arrival/departure of a process to move processes across these queues depending on the priority promotion/demotion policy.

9. *So far we have said that all registers including the PC need to be saved when context switching out a process. However, one can typically get away without*

saving the PC (meaning it gets saved in some other way). Where do you think it gets saved? Who (what entity) saves it?

It gets saved on the process stack! Remember, it contains the user-mode instruction executed just before calling the kernel routine responsible for (initiating) the context switch. So we exepct to find it in the process stack once we context switch this process back in. It would be saved by the kernel (specifically some instruction within the context switching routine).

10. *Which of process/K-level thread/U-level thread would you pick to design an application whose constituent activities: Are mostly CPU-bound; Are mostly CPU-bound and need to meet timeliness guarantees; Do a lot of blocking I/O?*

    For an application whose activities are mostly CPU-bound, we would choose user-level threads since they would block only occasionally and hence are likely to use their CPU cycles well. Context switching between these threads would pose less overheads than if we chose other options.

    If these have timeliness guarantees, we would choose kernel-level threads, since user-level threads can only be context switched at rather coarse time granularities (since the user-level scheduler relies on a signal-based software timer to run).

    If there is a lot of blocking I/O, we would prefer kernel-level threads.

    Kernel-level threads are more efficient than processes, which is why we pick them over processes in the last two cases.

11. *A process on an average runs for time T before blocking for I/O. A context switch takes time S. For round robin scheduling with quantum Q, give a formula for CPU efficiency.*

    CPU efficiency is the ratio of cycles (time) used for running processes (i.e., time spent doing useful work) divided by the total number of cycles over a long enough period of time. I will solve this for some of the cases and you should attempt the rest on your own.

    First, note that there are two ways in which a process may get preempted (i.e., context switched out): (i) because the quantum expires, which happens once every Q time units, and (ii) it issues a blocking I/O call, which happens once every T time units.

    If $Q = \inf$, only (ii) applies. That is, the system has to do a context switch once every T time units. Since a context switch consumes S time units, the CPU efficiency is:

    $$\frac{T}{T+S}.$$

    $S < Q < T$. To make things easy, assume that T is an integral multiple of Q: $T = nQ$, where $n > 1$. Then we have $(n+1)$ context switches over a period of length $n(Q+S)$ time units (n due to quantum expirations and 1 due to an I/O call). The overhead during this period is $(n+1)S$ (total time spent context

switching). Therefore CPU efficiency is:

$$\frac{n(Q+S)-(n+1)S}{nQ}.$$