C211 – Operating Systems

Tutorial: Processes and Threads

– Answers –

1. Which of the following instructions should only be allowed in kernel mode, and why?
   (a) Disable all interrupts
   (b) Read the time of day clock
   (c) Change the memory map
   (d) Set the time of day

   (a), (c) and (d)

2. If a multithreaded process forks, a problem occurs if the child gets copies of all the parent's threads. Suppose that one of the original threads was waiting for keyboard input. Now two threads are waiting for keyboard input, one in each process. Does this problem ever occur in single-threaded processes?

   No. If a single-threaded process is blocked on the keyboard, it cannot fork.

3. What is the biggest advantage of implementing threads in user space? What is the biggest disadvantage?

   The biggest advantage is efficiency. No traps to the kernel are needed to switch threads. The ability of having their own scheduler can also be an important advantage for certain applications. The biggest disadvantage is that if one thread blocks, the entire process blocks.

4. If in a multithreaded web server the only way to read from a file is the normal blocking `read()` system call, do you think user-level threads or kernel-level threads are being used? Why?

   A worker thread will block when it has to read a Web page from the disk. If user-level threads are being used, this action will block the entire process, destroying the value of multithreading. Thus it is essential that kernel threads are used to permit some threads to block without affecting the others.

5. Why would a thread ever voluntarily give the CPU by calling `thread_yield()`? After all, since there is no periodic clock interrupts, it may never get the CPU back.

   Threads in a process cooperate. They are not hostile to one another. If yielding is needed for the good of the application, then a thread will yield. After all, it is usually the same programmer who writes the code for all of them.

6. The register set is a per-thread rather than a per-process item. Why? After all, the machine has only one set of registers.

   When a thread is stopped, it has values in the registers. They must be saved, just as when the process is stopped the registers must be saved. Multiprogramming threads is no different than multiprogramming processes, so each thread needs its own register save area.

7. In a system with threads, is there one stack per thread or one stack per process when user-level threads are used? What about when kernel threads are used? Explain.

Each thread calls procedures on its own, so it must have its own stack for the local variables, return addresses, and so on. This is equally true for user-level threads as for kernel-level threads.

8. In this problem you are to compare reading a file using a single-threaded file server and a multithreaded server, running on a single-CPU machine. It takes 15 msec to get a request for work, dispatch it, and do the rest of the necessary processing, assuming that the data needed are in the block cache. If a disk operation is needed, as is the case one-third of the time, an additional 75 msec is required, during which time the thread sleeps. For this problem, assume that thread switching time is negligible. How many requests/sec can the server handle if it is single-threaded? If it is multithreaded?

   *In the single-threaded case, the cache hits take 15 msec and cache misses take 90 msec. The weighted average is 2/3*15 + 1/3*90. Thus the mean request takes 40 msec and the server can do 25 req/s. For a multithreaded server, if there are at least 6 threads and we assume a cooperative (non-preemptive) scheduler, all the waiting for the disk can be overlapped, so every request takes 15 msec, and the server can handle 66 2/3 requests per second. On the other hand, if we assume a preemptive round-robin scheduler (with a small quantum), we can perform the following analysis. Each request needs on average 15ms CPU time and 2/3*0 + 1/3*75 = 25ms I/O time. The probability that all n threads are sleeping is $(25/40)^n = (5/8)^n$. Thus the CPU utilisation is $1 - (5/8)^n$; during 1000ms, the CPU handles $(1 - (5/8)^n) * 1000/15$ requests. For $n = 1$ we get 25 req/s, as expected. For $n = 2$ we get 40.62 req/s, for $n = 6$ we get 62.69 req/s, etc.*

9. Would an algorithm that performs several independent CPU-intensive calculations concurrently (e.g., matrix multiplication) be more efficient if it used threads, or if it did not use threads? Why is this a hard question to answer?

   The answer to this question depends on the implementation of threads and if the system is a uniprocessor system or a multiprocessor system. If the system is a multiprocessor system, the implementation using threads is more efficient if each thread could execute on a separate processor. However, if the system is a uniprocessor system, the implementation using threads would be less efficient due to the additional overhead incurred by spawning threads and switching contexts between the threads.

10. IPC mechanisms

    (a) What happens when a signal is received by a process? *The default action for most signals is to terminate the process, unless the process has installed a handler for that signal. Note that SIGKILL and SIGSTOP cannot be ignored/handled.*

    (b) When two processes communicate through a pipe, the kernel allocates a buffer (of size 65536 bytes in Linux) for the pipe. What happens when the process at the write-end of the pipe attempts to send additional bytes on a full pipe?

    The process at the write-end of the pipe blocks until data is read at the other end.

    (c) What happens when the process at the write-end of the pipe attempts to send additional bytes and the process at the read-end has already closed the file descriptor associated with the read-end of the pipe?

    The process receives the SIGPIPE signal.

    (d) The process at the write-end of the pipe wants to transmit a linked list data structure (with one integer field and a "next" pointer) over a pipe. How can it do this?

    The communication channel provided by a pipe is a stream of bytes. As a result, the linked list has to be serialized before transmission. In particular, the pointer values cannot be transmitted directly, since the addresses at the write-end process are likely to be invalid at the read-end process.

    (e) When would it be better for two processes to communicate via shared memory instead of pipes? What about the other way around?

    Communicating via shared memory is often faster because there is no kernel intervention after the

shared memory is established; it is also more flexible, since communication can be bi-directional. However, if the communication between the two processes is uni-directional, with the standard output of one process acting as the standard input of the other, then pipes may be preferable, because their communication is synchronized by the kernel, while with shared memory processes would need to implement their own synchronization mechanism.