C211 – Operating Systems
Tutorial: Processes and Threads

Lecturer: Peter Pietzuch ⟨prp@doc.ic.ac.uk⟩

1. 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?
2. What is the biggest advantage of implementing threads in user space? What is the biggest disadvantage?
3. 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?
4. 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.
5. The register set is a per-thread rather than a per-process item. Why? After all, the machine has only one set of registers.
6. 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.
7. 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?
8. 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?
9. IPC mechanisms
    (a) What happens when a signal is received by a process?
    (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?
    (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?
    (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?
    (e) When would it be better for two processes to communicate via shared memory instead of pipes? What about the other way around?