CS-323: OS Exercise Week 9 (Fall 2023)

November 15, 2023

Some parts of this exercise are intended to familiarize you with common threading aspects of modern systems, and go beyond the material presented in the lecture/OSTEP. You are encouraged to explore on your own by writing short programs and/or querying your favorite search engine. When in doubt, ask your TAs questions as well.

1 Threads vs Processes

(a) Why can multiple threads and processes be used for a single task?

Threads/processes allow computation in parallel and/or concurrently. Distinct programs always run in different processes. A single program may use multiple threads in one process or even run multiple processes.

For example, the Apache web server spawns a new process to handle each incoming connection. Then, each of these processes use multiple threads to handle concurrent HTTP requests. A good reason to use separate processes for each connection here is to keep them isolated: one connection cannot access another connection's data even if exploited. Using multiple threads in a single program allows for efficiency thanks to concurrency. Even if one thread is busy handling some request blocking for I/O, another thread can immediately start handling any new requests. If there was only a single thread assigned to the connection, it wouldn't be possible to start processing any new requests without fully completing the current one. This is an advantage due to the concurrency of separate threads, and works even on a single core CPU.

It's also possible to exploit parallelism. As an example, for a matrix multiplication, it is possible to split the matrices into tiles and multiply each one independently. Splitting these tile multiplication jobs between separate threads will allow the use of multiple cores on a CPU, since each thread can be scheduled on a different core. For K cores, this would make the multiplication rougly K times faster compared to a using a single thread!

(b) Threads share a lot of resources. List what is shared and what is kept private for each thread from the below list:
Virtual address space, open file descriptors, registers, stack pointer, child processes, signals and signal handlers, instruction pointer

Shared items	Private items
Virtual address space (e.g., global variables),	Instruction pointer
Open file descriptors	Registers
Child processes	Stack pointer
Signals and signal handlers	

Note: The only thing truly private to a thread are its registers (instruction and stack pointers are just special registers). While each thread gets its own stack space nothing stops other threads from accessing that memory area (although they should not). Also note that the instruction pointer and stack pointer are also registers.

(c) Why does each thread require its own stack?

Name: ____________________________

Each thread executes functions independently of each other, using its own stack for its own local variables. This is why each thread is given its own stack region, so that they can have their own execution context, separate from other threads'.

(d) What prevents threads from overwriting each other's stacks?

Nothing. Since all the threads in a single process live in the same address space, a t hread is able to read from and write to an address on another thread's stack. A proper application developer will not give another thread the address of its stack, and thus, will indirectly prevent access to it by other threads. As no other thread knows where the stacks of the other threads are, they stay hidden. Threads should work in a cooperative manner and not fight each other.

(e) Why is the heap shared between threads?

Often, threads in a single process will have to communicate through some shared data structures for sharing work or notifying each other of events etc. This is why the heap is shared between all threads (remember that they use the same address space, so it's a given). Separate threads can access the same data by simply using the same pointers. Note that properly using shared data will often require synchronization mechanisms between threads as studied in class.

(f) Why would you prefer creating multiple threads over spawning multiple processes for a parallel/-concurrent program?

- Data transfer using heap buffers or global variables is often much easier than using IPC (inter-process communication) mechanisms.
- Switching between threads (on the same processing core) can be faster than switching between processes, only registers need to be switched (no need to swap page tables and flush the TLB for example).
- Since all threads live in the same address space, synchronization without system calls can be possible depending on the use case with atomic instructions. This is much more efficient than doing synchronization with system calls.

(g) Why would you prefer using spawning multiple processes over creating multiple threads for a parallel/concurrent program?

- Security (isolation to be precise). Each process has its own address space. Any shared memory is explicitly setup, and restricted to specific addresses. e.g., the browser Chrome actually runs each tab as a separate process to prevent data leaks and exploits across tabs. This is why you can usually see dozens of `chrome.exes` in the task manager on Windows.

2 Synchronization with Atomics

(a) Should shared variables always be protected by locks? Specify if synchronization is necessary in the following scenarios, and where it is necessary if so:

In general, shared variables do not need to be protected unless the shared variable is being updated.

- Seven threads are using the value of the same constant global variable in one of their operations. Read-only access does not require any synchronization since none of the threads will be modifying the constant. No mechanisms are necessary.
- A program is adding two arrays of 100,000 values u and v, and producing another resulting array of 100,000 values w. The job is split between 10 threads: Thread 0 will add the first 10,000 values of u and v and write the results to the first 10,000 values of w in the range [0, 9999]. Thread 1 will do the same with values in the range [10000, 19999] and so on, until thread 9 which does the same with indices [90000, 99999].
No synchronization is necessary for reading from u and v. For writing to w, each thread writes to a separate part of the array and there is thus no need for synchronization once again.

- Instead of adding two arrays, we are now trying to sum up an array of a of 100,000,000 values. We split the work between 100 threads, where each will sum up 1,000,000 values into a partial sum r. The result r is then added to a final result variable s. s will contain the sum of the whole array once every thread is finished.
  Reading from a does not require synchronization, however adding the partial sum r to s requires s to be protected by a lock to guarantee no data races, since every thread needs to update s.

(b) The spinlock implementation shown in class is made possible thanks to atomic instructions, and does not work with the usual ones. Why do compilers not use atomic instructions as often as possible? For example, we could replace every increment with an atomic fetch-and-add instead of using three instructions in the form of `mov-add-mov`. Why is this not done in practice?

Atomic operations have an associated overhead, especially on modern multicore processors. This can involve hard memory barriers which prevent the the CPU hardware from reordering instructions, locking addresses from other cores to prevent access, or flushing caches into memory to force consistency with other cores. This extra runtime cost is best avoided when no synchronization is necessary. However, when usable, atomic instructions will still be significantly faster than synchronization involving system calls.

(c) Spinlocks can often be found in OS kernel code. However, they are considered an anti-pattern to be almost always avoided in user code, where sleepable locks should be preferred. Why is this the case?

Spinlocking wastes precious CPU cycles. The operating system does not know *what* a thread is executing, so useful work and spinning on a lock have the same priority. Especially in a high contention scenario where many threads are trying to access the same variable, a massive amount of CPU time will be wasted since many threads will spend their whole scheduling tick spinning and doing nothing else.

A sleepable lock using system calls will instead put the thread to sleep until the lock becomes available and it is woken up. While this has more overhead than a spinlock, there will be no CPU time wasted during waiting.

Spinlocks can still be faster than sleepable locks when contention is very low and you expect code to be able to acquire a lock immediately most of the time, but measuring and benchmarking is always recommended. They are used in such cases in kernel code, and also in code that cannot be put to sleep, such as interrupt handlers.