Project 2 Grading Rubric

Tyler Allen

April 9, 2018

Every grade on canvas has a comment next to it about why points were taken off. I will discuss the major issues here. There was a slight curve applied to the grades because some issues affected nearly everybody. Note that these were problems with explicit requirements listed in the handout.

I will only respond to questions about individual mistakes in-person during office hours.

Most Common Issues:

- Orphan Threads: What happens when a process does not join a thread? This is called an *orphaned* thread. Some modern user thread implementations (pthreads) will issue an error if this happens unless the thread is *detatched*. In our case, we did not have a specified notion of unattached/attached. How you handle this was left to interpretation - but crashing the OS is not one of them.

  Note: With this, threads should clean up the shared address space if they are the last reference. In most cases, there were checks in place so that the main thread *did not* clean up the process address space if there was a remaining thread, but did not address the fact that a thread was remaining alive. Of course, if did clean up the address space then the behavior was undefined.

- File Handles: File handles are typically associated with the parent and shallow copied for children. This is loosely related to the "shared memory" concept.

- Address Space Locks: Now that processes share an address space, what happens when both try to grow their address space at the same time? When does this happen? Primarily in `malloc` in xv6 - we want to grow the heap. The xv6 `malloc` actually has two issues. The foundational syscall behind `malloc` is `sbrk`, which calls `growproc`. If two threads sharing an address space call malloc at the same time, there is a race condition within `growproc` and behavior is undefined. The `malloc` implementation itself is also not thread-safe, using global state variables to track the list of allocated memory. This creates yet another race condition. Note that it is *not* sufficient to just lock `malloc`; the in-kernel race condition would apply to *any* memory allocator, not just `malloc`.

  – The best solution to this used a lock *for each parent process*. This method only causes threads that share an address space to wait on the lock, not all running processes. This was not required; we only required (1) of the two described locks to be implemented in a working form.

- User Lock Safety: Most projects correctly used the atomic transactions to perform the lock update. However, standard library lock implementations typically provide a form of memory barrier around locks to prevent dangerous over-optimization. This memory barrier is in the form of the `__sync_synchronize()` compiler builtin function. This barrier ensures all memory transactions are committed to memory prior to the call. This ensures that lock updates stuck in the cache are forced to hit memory and be seen by the lock-acquiring thread. *Without this*, every lock and variable updated across-threads must be marked with the keyword *volatile* to ensure that modifications to these values reach main memory. See spinlock.h for corrected usage. The `sti` functions were *not* to be used, there is no system-crashing lock-interrupt issue in userspace.

- When you update process states, you need to grab the process table lock. See `fork()`.