CS 202 Lab 2 Report, March 9, 2019, by Tyson Loveless, 861118716

Lab 2:
xv6 Threads

In this lab, we explore issues surrounding the implementation of multi-threaded kernels. In particular, we extend xv6 to include kernel level threads to allow multi-threaded processes to execute. In conjunction with this extension, we implement various types of locks to deal with write exclusivity over code regions, and demonstrate our implementation through a simulated game of frisbee played between multiple threads.

Creating threads: the `clone` system call and thread library

To extend xv6 to allow for kernel threads, we first add a `clone` system call, which acts similarly to the `fork` system call, with minor changes:

(1) children share the address space of whomever they `clone` from
(2) file descriptors are shared instead of duplicated
(3) a per-thread stack should be allocated

The difficulty in the this part is setting up the thread's stack; in particular, the parent's stack must be copied, along with any additional storage that is allocated for local variables. To accomplish the stack copy, I we copy all addresses from `curproc->tf->esp` through `curproc->tf->ebp + shift`, where `shift` is the size of all local variables within the `clone` function. This ensures that locals allocated for when calling `clone` is copied over, and hence the correct return address will be given when returning from the clone and the parent. A simple test program [`test.c`] verifies that clones are being created as expected.

Lock implementations

Multithreaded processes suffer from race conditions when more than one thread attempts to write to a shared location. To avoid these conditions, locks are generally used. To allow for proper multithreaded programming, I implemented and tested four types of locks:

(1) a basic spinlock
(2) an array-based queuing lock
(3) seqlock
(4) MCS lock

The various lock implementations differ in their method and requirements from the system. For the spinlock, an atomic exchange operation is used (as in the given xv6 kernel code), and all threads spin on the same location. This causes significant overhead when many threads are competing for the lock. The other three locks attempt to mitigate the overhead from all spinning on the same location in different ways. The array-based lock (and MCS lock) spin on memory locations unique to each thread, so that when it is a given thread's turn to enter the critical region, there is no chatter on the lock. The seqlock differs, in that any reader can enter a critical region, but if a writer is going to enter, an internal condition variable (seq) must be set to the correct value. Typically, this is accomplished through incrementing seq when acquiring and releasing the lock; however, for this assignment I chose to set seq to the PID of the next thread in line.

The most difficult implementation was for the MCS lock, and outside help was sought, as xv6 does not include necessary instructions within the kernel to accomplish atomic fetch_and_store and compare_and_exchange. Thankfully, MIT's course includes a source file with the necessary atomic instructions. I adapted my MCS lock from MIT's source (which, incidentally, does not compile as is and required adaptation to work in the current version of xv6).

Frisbee

Each frisbee simulation used the various locks, with the uptime() taken just before and just after the game commences. The table below gives the average running time for each lock taken over 10 games each, with each column giving the number of threads and number of throws given as parameters.

#threads, #throws	**3, 3**	**10, 15**	**20, 40**
Spinlock	13.9	129.2	566.8
Array Lock	4	17	42
SeqLock	4	23.6	58.8
MCS Lock	8	34.6	71.4

The array lock and seqlock were very consistent across the runs, with only 1 or 2 variations (seemingly due to kernel interruptions). As expected, the spinlock wildly varied between runs, as contention for the shared lock caused significant chatter. MCS suffered from variation as well, which may be due to the complexity of the implementation, requiring several 'atomic' instructions during acquire and release.

List of Modified Files and Added Functions:

For providing the system call (`clone`):
- `defs.h`
- `sysproc.c`
- `user.h`

- `syscall.h`
- `syscall.c`
- `proc.h`
- `proc.c`
- `usys.S`

Files added:

- `test.c`
- `frisbee.c`
- `arr_frisbee.c`
- `seq_frisbee.c`
- `mcs_frisbee.c`
- `threads.h`
- `threads.c`