Lectures on Operating Systems (Mythili Vutukuru, IIT Bombay)

Lab: Synchronization in xv6

The goal of this lab is to understand the concepts of concurrency and synchronization in xv6.

Before you begin

- Download, install, and run the original xv6 OS code. You can use your regular desktop/laptop to run xv6; it runs on an x86 emulator called QEMU that emulates x86 hardware on your local machine. In the xv6 folder, run `make`, followed by `make qemu` or `make-qemu-nox`, to boot xv6 and open a shell.
- We have modified some xv6 files for this lab, and these patched files are provided as part of this lab's code. Before you begin the lab, copy the patched files into the main xv6 code directory. The modified files are as follows.
  - The files `defs.h`, `syscall.c`, `syscall.h`, `sysproc.c`, `user.h` and `usys.S` have been modified to add the new system calls of this lab.
  - The programs `test_counters.c`, `test_toggle.c`, `test_barrier.c`, `test_waitpid.c` and `test_sem.c` are user test programs to test the synchronization primitives you will develop in this lab.
  - The new synchronization primitives you need to implement are declared in `uspinlock.h`, `barrier.h`, and `semaphore.h`. These functions must be implemented in `proc.c`, `uspinlock.c`, and `barrier.c`, `semaphore.c` in the placeholders provided. Some of the header files and test programs may also need to be modified by you.
  - An updated Makefile has been provided, to include all the changes for this lab. You need not modify this file.

Part A: Userspace locks in xv6

In the first part of the lab, we will implement userspace locks in xv6, to provide mutual exclusion in userspace programs. However, xv6 processes do not share any memory, and therefore a need for mutual exclusion does not arise in general. As a proxy for shared memory on which synchronization primitives like locks can be tested, we have defined shared counters within the xv6 kernel, that are available for user space programs.

Within the patched code for this lab, there are NCOUNTER (defined to be 10 in the modified `sysproc.c`) integers that are available to all userspace programs. These counters can all be initialized to 0 with the system call `ucounter_init()`. Further, one can set and get values of a particular shared counter using the following system calls: `ucounter_set(idx, val)` sets the value of the

counter at index idx to the value val and ucounter_get(idx) returns the value of the counter at index idx. Note that the value of idx should be between 0 and NCOUNTER-1.

We will now implement userspace spinlocks to protect access to these shared counters. You must support NLOCK (defined to be 10 in uspinlock.h) spinlocks for use in userspace programs. You will define a structure of NLOCK userspace spinlocks, and implement the following system calls on them. All your code must be in uspinlock.c.

- uspinlock_init() initializes all the spinlocks to unlocked state.
- uspinlock_acquire(idx) will acquire the spinlock at index idx. This function must busily spin until the lock is acquired, and return 0 when the lock is successfully acquired.
- uspinlock_release(idx) will release the spinlock at index idx. This function must return 0 when the lock is successfully released.

While not exposed as a system call, you must also implement the function uspinlock_holding(idx), which must return 0 if the lock is free and 1 if the lock is held. You will find this function useful in implementing part B later. Do not worry about concurrent updates to the lock while this function is executing; it suffices to simply return the lock status.

You must use hardware atomic instructions to implement user level spinlocks, much like the kernel spinlock implementation of xv6. However, the userspace spinlock implementation is expected to be simpler, since you do not need to worry about disabling interrupts and such things with userspace spinlocks. The skeleton code for these system calls has already been provided to you in sysproc.c, and you only need to implement the core logic in uspinlock.c.

We have provided a test program test_counters.c for you to test your mutual exclusion functionality. This program forks a child, and the parent and child increment a shared counter concurrently multiple times. When you run this program in its current form, without any locks, you will see that an incorrect result is being displayed. You must add locks to this program to enable correct updates to the shared counter. After you implement locks and add calls to lock/unlock in the test program, you should be able to see that the shared counter is correctly updated as expected.

Part B: Userspace condition variables in xv6

In this part, you will implement condition variables for use in userspace programs. The following two system calls have been added in the patched code provided to you.

- ucv_sleep(chan, idx) puts the process to sleep on the channel chan. This call should be invoked with the userspace spinlock at index idx held. The code to parse the system call arguments has already been provided to you in sysproc.c, and you only need to implement the core logic of this system call in the function ucv_sleep defined in proc.c.
- ucv_wakeup (chan) wakes up all of the processes sleeping on the channel chan. This system call has already been implemented for you in sysproc.c, by simply invoking the kernel's wakeup function.

While the userspace wakeup function can directly invoke the kernel's wakeup function with a suitable channel argument, the ucv_sleep function cannot call the kernel's sleep function for the following reason: the kernel sleep function is invoked with a kernel spinlock held, while the userspace programs

will invoke sleep on the userspace condition variables with a userspace spinlock held. Therefore, you will need to implement the userspace sleep function yourself, along the lines of the kernel sleep function. Note that the process invoking this function must be put to sleep after releasing the userspace spinlock, and must reacquire the spinlock before returning back after waking up.

We have provided a simple test program `test_toggle.c` to test condition variable functionality. This program spawns a child, and the parent and child both print to the screen. You must use condition variables and modify this program in such a way that the parent and child alternate with each other in printing to the screen (starting with the child). After you correctly implement the `ucv_sleep` function, and add calls to sleep and wakeup to the test program, you should be able to achieve this desired behavior of execution toggling between the parent and child. Of course, you may also write other such test programs to test your condition variable implementation.

Part C: Waitpid system call

Implement the `waitpid` system call in xv6. This system call takes one integer argument: the PID of the child to reap. This system call must return the PID of the said child if it has successfully reaped the child, and must return -1 if the said child does not exist. Once a child is reaped with `waitpid`, the same child should not be returned by a subsequent `wait` system call. You must write your code in the placeholder provided in `proc.c`.

You have been provided a test program `test_waitpid.c` to test your code. The program forks a child and tries to reap it via `waitpid` and `wait`. If waitpid functionality is implemented correctly, the child will be reaped by the `waitpid` call and not by the subsequent `wait`.

Part D: Userspace barrier in xv6

In this part, you will implement a barrier in xv6. A barrier works as follows. The barrier is initialized with a count N, using the system call `barrier_init(N)`. Next, processes that wish to wait at the barrier invoke the system call `barrier_check()`. The first $N - 1$ calls to `barrier_check` must block, and the N-th call to this function must unblock all the processes that were waiting at the barrier. That is, all the N processes that wish to synchronize at the barrier must cross the barrier after all N of them have arrived. You must implement the core logic for these system calls in `barrier.c`.

You may assume that the barrier is used only once, so you need not worry about reinitializing it multiple times. You may also assume that all N processes will eventually arrive at the barrier.

You must implement the barrier synchronization logic using the `sleep` and `wakeup` primitives of the xv6 kernel, along with kernel spinlocks. Note that it does not make sense to use the userspace sleep/wakeup functionality or userspace spinlocks developed by you in this part of the assignment, because the code for these system calls must be written within the kernel using kernel synchronization primitives.

We have provided a test program `test_barrier.c` to test your barrier implementation. In this program, a parent and its two children arrive at the barrier at different times. With the skeleton code provided to you, all of them clear the barrier as soon as they enter. However, once you implement the barrier, you will find that the order of the print statements will change to reflect the fact that the processes block until all of them check into the barrier.

Part E: Semaphores in xv6

In this part, you will implement the functionality of semaphores in xv6. You will define an array of (NSEM = 10) semaphores in the kernel code, that are accessible across all user programs. User processes in xv6 will be able to refer to these shared semaphores by an index into the array, and perform up/down operations on them to synchronize with each other. Our patch adds three new system calls to the xv6 kernel:

- `sem_init(index,val)` initializes the semaphore counter at the specified `index` to the value `val`. Note that the index should be within the range [0, NSEM-1].
- `sem_up(index)` will increment the counter value of the semaphore at the specified index by one, and wakeup any one sleeping process.
- `sem_down(index)` will decrement the counter value of the semaphore at the specified index by one. If the value is negative, the process blocks and is context switched out, to be woken up by an up operation on the semaphore at a later point.

The scaffolding to implement these system calls has already been done or you, and you will need to only implement the system call logic in the files `semaphore.h` and `semaphore.c`. You must define the semaphore structure, and the array of NSEM semaphores, in the header file. You must also implement the functions that operate on the semaphores in the C file. Your implementation may need to invoke some variant of the `sleep` or `wakeup` functions of the xv6 kernel. For writing modified sleep/wakeup functions in xv6, you will need to modify the files `proc.c`, and `defs.h`.

Once you implement the three system calls, run the test program `test_sem` given to you to verify the correctness of your implementation. The test program spawns two children. When you run the test program before implementing semaphores, you will find that the children print to screen before the parent. However, once the semaphore logic has been correctly implemented by you in `semaphore.c` and `semaphore.h`, you will find that the parent prints to screen before the children, by virtue of the semaphore system calls. Do not modify this test program in any way, but simply use it to test your semaphore implementation.

Submission instructions

- For this lab, you will need to submit modified versions of the following files: `defs.h`, `proc.c`, `uspinlock.c`, `barrier.c`, `test_counters.c`, `test_toggle.c`, `semaphore.h`, `semaphore.c`.
- Place all the files you modified in a directory, with the directory name being your roll number (say, 12345678).
- Tar and gzip the directory using the command tar -zcvf 12345678.tar.gz 12345678 to produce a single compressed file of your submission directory. Submit this tar gzipped file on Moodle.