Lecture Notes on Operating Systems

Mythili Vutukuru, Department of Computer Science and Engineering, IIT Bombay

Synchronization in xv6

1. Locks and (equivalents of) condition variables

- Locks (i.e., spinlocks) in xv6 are implemented using the `xchg` atomic instruction. The function to acquire a lock (line 1574) disables all interrupts, and the function that releases the lock (line 1602) re-enables them.
- xv6 provides `sleep` (line 2803) and `wakeup` (line 2864) functions, that are equivalent to the wait and signal functions of a condition variable. The sleep and wakeup functions must be invoked with a spinlock that ensures that the sleep and wakeup procedures are completed atomically. A process that wishes to block on a condition calls `sleep(chan, lock)`, where `chan` is any opaque handle, and `lock` is any lock being used by the code that calls sleep/wakeup. The sleep function acquires a specific lock that protects the scheduler's process list (`ptable.lock`) and releases the lock provided to it, so that it can make changes to the state of the process and invoke the scheduler (line 2825). Note that the function checks that the lock provided to it is not this `ptable.lock` to avoid locking it twice. All calls to the scheduler should be made with this `ptable.lock` held, so that the scheduler's updating of the process list can happen without race conditions.
- Once the sleep function calls the scheduler (line 2825), it is context switched out. The control returns back to this line (of calling the scheduler) once the process has been woken up and context switched in by the scheduler again, with the `ptable.lock` held. At that time, the sleep function releases this special lock, reacquires the original lock, and returns back in the woken up process. Note that the sleep function holds at least one of the two locks—the lock given to it by the caller, or the `ptable.lock`—at any point of time, so that no other process can run wakeup in the duration between this process deciding to sleep and actually sleeping, because a process will need to lock one or both of these locks while calling wakeup. (Understand why lines 2818 and 2819 can't be flipped.)
- A process that makes the condition true will invoke `wakeup(chan)` while it holds the lock. The wakeup call makes all waiting processes runnable, so it is indeed equivalent to the signal broadcast function of `pthreads` API. Note that a call to wakeup does not actually context switch to the woken up processes. Once wakeup makes the processes runnable, these processes can actually run when the scheduler is invoked at a later time.
- xv6 does not export the sleep and wakeup functionality to userspace processes, but makes use of it internally at several places within its code. For example, the implementation of pipes (sheet 65) clearly maps to the producer-consumer problem with bounded buffer, and the implementation uses locks and condition variables to synchronize between the pipe writer and pipe reader processes.

2. Context switching and the process table lock

- Let us revisit the CPU scheduler code on sheet 27, with the knowledge of locking in xv6. Review the `scheduler`, `sched`, and `swtch` functions. Focus on the acquisition and release of the lock that protects the process table data structure.

- Recall that a process that wishes to relinquish the CPU calls the function `sched`. This function triggers a context switch, and when the process is switched back in at a later time, it resumes execution again in `sched` itself. Thus a call to `sched` freezes the execution of a process temporarily. When does a process relinquish its CPU in this fashion? (i) When a timer interrupt occurs and it is deemed that the process has run for too long, the trap function calls `yield`, which in turn calls `sched` (line 2776). (ii) When a process terminates itself using `exit`, it calls `sched` one last time to give up the CPU (line 2641). (iii) When a process has to block for an event and sleep, it calls `sched` to give up the CPU (line 2825). The function `sched` simply checks various conditions, and calls `swtch` to switch to the scheduler thread.

- Any function that calls `sched` must do so with the `ptable.lock` held. This lock is held all during the context switch. During a context switch from P1 to P2, P1 locks `ptable.lock`, calls `sched`, which switches to the scheduler thread, which again switches to process P2. When process P2 returns from `sched`, it releases the lock. For example, you can see the lock and release calls before and after the call to `sched` in `yield` and `sleep`. There is no call to release the lock in `exit` because a terminated process is not expected to run again.

- Note that the function `forkret` (line 2783) releases the lock for a process that is executed for the first time, since a new process does not return in `sched`. The new process subsequently goes to `trapret`, and returns from trap like the parent. The context structure on the kernel stack of the new process is built this way by `allocproc`. The primary reason for a new process to go to forkret before returning from trap is to release this lock.

- Typically, a process that locks also does the corresponding unlock, except during a context switch when a lock is acquired by one process and released by the other.

- Note that all interrupts are disabled when any lock is held in xv6, so all interrupts are disabled during a context switch. If the scheduler finds no process to run, it periodically releases `ptable.lock`, re-enables interrupts, and checks for a runnable process again.

- With a knowledge of how scheduling works, it may be worth revisiting the sleep and wakeup functions (sheet 28), especially noting the subtleties around locks. The `sleep` function (line 2803) must eventually call `sched` to give up the CPU, so it must acquire `ptable.lock`. The function first checks that the lock held already is not `ptable.lock` to avoid deadlocks, and releases the lock given to it after acquiring `ptable.lock`. Is it OK to release the lock given to sleep before actually calling `sched` and going to sleep? Yes it is, because wakeup also requires `ptable.lock`, so there is no way a wakeup call can run while `ptable.lock` is held. Is it OK to release the lock given to sleep before acquiring `ptable.lock`? No, it is not, as wakeup may be invoked in the interim when no lock is held.

- The `wait` and `exit` system calls (sheet 26) are another example of the usage of `sleep` and `wakeup` functionality within the kernel. When a parent calls `wait`, the wait function (line 2653) acquires `ptable.lock`, and looks over all processes to find any of its zombie children. If none is found, it calls sleep to block until a child dies. Note that the lock provided to sleep in this case is also `ptable.lock`, so sleep must not attempt to re-lock it again. When a child calls `exit` (line 2604), it acquires `ptable.lock`, and wakes up its parent. Note that the exit function does not actually free up the memory of the process. Instead, the process simply marks itself as a zombie, and relinquishes the CPU by calling `sched`. When the parent wakes up in wait, it does the job of cleaning up its zombie child, and frees up the memory of the process. The wait and exit system calls provide a good use case of the sleep and wakeup functions.