Lecture Notes for CS347: Operating Systems

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

6. Scheduling and Synchronization in xv6

6.1 Locks and (equivalents of) conditional 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 conditional variable. The sleep and wakeup functions must be invoked with a lock 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 releases the lock the process came with, and acquires a specific lock that protects the scheduler's process list (`ptable.lock`), 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, 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 while sleep is executing, because a process will 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 conditional variables to synchronize between the pipe writer and pipe reader processes.

6.2 Scheduler and context switching

- xv6 uses the following names for process states (line 2350): UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE. The `proc` structures are all stored in a linked list.
- Every CPU has a scheduler thread that calls the `scheduler` function (line 2708) at the start, and loops in it forever. The job of the scheduler function is to look through the list of processes, find a runnable process, set its state to RUNNING, and switch to the process.
- All actions on the list of processes are protected by `ptable.lock`. Any process that wishes to change this data structure must hold the lock, to avoid race conditions. What happens if the lock is not held? It is possible that two CPUs find a process to be runnable, and switch it to running state simultaneously, causing one process to run at two places. So, all actions of the `scheduler` function are always done with the lock held.
- Every process has a `context` structure on its stack (line 2343), that stores the values of the CPU registers that must be saved during a context switch. The registers that are saved as part of the context include the EIP (so that execution can resume at the instruction where it stopped), ESP (so that execution can continue on the stack where it left off), and a few other general purpose registers. To switch the CPU from the current running process P1 to another process P2, the registers of P1 are saved in its context on its stack, and the registers of P2 are reloaded from its context (that would have been saved in the past when P2 was switched out). Now, when P1 must be resumed again, its registers are reloaded from its context, and it can resume execution where it left off.
- The `swtch` function (line 2950) does the job of switching between two contexts, and old one and a new one. The step of pushing registers into the old stack is exactly similar to the step of restoring registers from the new stack, because the new stack was also created by `swtch` at an earlier time. The only time when the `context` structure is not pushed by `swtch` is when a process is created for the first time. For a new process, `allocproc` writes the context onto the new kernel stack (lines 2488-2491), which will be loaded into the CPU registers by `swtch` when the process executes for the first time. All new processes start at `forkret`, so `allocproc` writes this address into the EIP of the context it is creating. Except in this case of a new process, the EIP stored in context is always the address at which the running process invoked `swtch`, and it resumes at the same point at a later time. Note that `swtch` does not explicitly store the EIP to point to the address of the `swtch` statement, but the EIP is automatically stored on the stack as part of making the function call to `swtch`.
- In xv6, the scheduler runs as a separate thread with its own stack. Therefore, context switches happen from the kernel mode of a running process to the scheduler thread, and from the scheduler thread to the new process it has identified. (Other operating systems can switch directly between kernel modes of two processes as well.) As a result, the `swtch` function is called at two places: by the scheduler (line 2728) to switch from the scheduler thread to a new process, and by a running process that wishes to give up the CPU in the function `sched` (line 2766). A running process always gives up the CPU at this call to `swtch` in line 2766, and always

resumes execution at this point at a later time. The only exception is a process running for the first time, that never would have called `swtch` at line 2766, and hence never resumes from there.

- 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`, `sleep`, and `exit`. Note that the function forkret also releases the lock for a process that is executed for the first time, since a new process does not return in sched. 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.

6.3 Implementation of sleep, wakeup, wait, exit

- 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.

- 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. 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.