Chapter 5

Scheduling

Any operating system is likely to run with more processes than the computer has processors, and so some plan is needed to time share the processors between the processes. An ideal plan is transparent to user processes. A common approach is to provide each process with the illusion that it has its own virtual processor, and have the operating system multiplex multiple virtual processors on a single physical processor.

Xv6 adopts this approach. If two processes want to run on a single CPU, xv6 multiplexes them, switching many times per second between executing one and the other. This multiplexing creates the illusion that each process has its own CPU, just as xv6 used the memory allocator and hardware segmentation to create the illusion that each process has its own memory.

Implementing multiplexing has a few challenges. First, how to switch from one process to another? Xv6 uses the standard mechanism of context switching; although the idea is simple, the code to implement is typically among the most opaque code in an operating system. Second, how to do context switching transparently? Xv6 uses the standard technique of using the timer interrupt handler to drive context switches. Third, many CPUs may be switching among processes concurrently, and a locking plan is necessary to avoid races. Fourth, when a process has exited its memory and other resources must be freed, but it cannot do all of this itself because (for example) it can't free its own kernel stack while still using it. Xv6 tries to solve these problems as simply as possible, but nevertheless the resulting code is tricky.

xv6 must provide ways for processes to coordinate among themselves. For example, a parent process may need to wait for one of its children to exit, or a process reading on a pipe may need to wait for some other process to write the pipe. Rather than make the waiting process waste CPU by repeatedly checking whether the desired event has happened, xv6 allows a process to give up the CPU and sleep waiting for an event, and allows another process to wake the first process up. Care is needed to avoid races that result in the loss of event notifications. As an example of these problems and their solution, this chapter examines the implementation of pipes.

Code: Context switching

At a low level, xv6 performs two kinds of context switches: from a process's kernel thread to the current CPU's scheduler thread, and from the scheduler thread to a process's kernel thread. xv6 never directly switches from one user-space process to another; this happens by way of a user-kernel transition (system call or interrupt), a context switch to the scheduler, a context switch to a new process's kernel thread, and a

trap return. In this section we'll example the mechanics of switching between a kernel thread and a scheduler thread.

Every xv6 process has its own kernel stack and register set, as we saw in Chapter 2. Each CPU has a separate scheduler thread for use when it is executing the scheduler rather than any process's kernel thread. Switching from one thread to another involves saving the old thread's CPU registers, and restoring previously-saved registers of the new thread; the fact that `esp` and `eip` are saved and restored means that the CPU will switch stacks and switch what code it is executing.

`swtch` doesn't directly know about threads; it just saves and restores register sets, called contexts. When it is time for the process to give up the CPU, the process's kernel thread will call `swtch` to save its own context and return to the scheduler context. Each context is represented by a `struct context*`, a pointer to a structure stored on the kernel stack involved. `Swtch` takes two arguments: `struct context **old` and `struct context *new`. It pushes the current CPU register onto the stack and saves the stack pointer in `*old`. Then `swtch` copies `new` to `esp`, pops previously saved registers, and returns.

Instead of following the scheduler into `swtch`, let's instead follow our user process back in. We saw in Chapter 3 that one possibility at the end of each interrupt is that `trap` calls `yield`. `Yield` in turn calls `sched`, which calls `swtch` to save the current context in `proc->context` and switch to the scheduler context previously saved in `cpu->scheduler` (2166).

`Swtch` (2352) starts by loading its arguments off the stack into the registers `%eax` and `%edx` (2359-2360); `swtch` must do this before it changes the stack pointer and can no longer access the arguments via `%esp`. Then `swtch` pushes the register state, creating a context structure on the current stack. Only the callee-save registers need to be saved; the convention on the x86 is that these are `%ebp`, `%ebx`, `%esi`, `%ebp`, and `%esp`. `Swtch` pushes the first four explicitly (2363-2366); it saves the last implicitly as the `struct context*` written to `*old` (2369). There is one more important register: the program counter `%eip` was saved by the `call` instruction that invoked `swtch` and is on the stack just above `%ebp`. Having saved the old context, `swtch` is ready to restore the new one. It moves the pointer to the new context into the stack pointer (2370). The new stack has the same form as the old one that `swtch` just left—the new stack *was* the old one in a previous call to `swtch`—so `swtch` can invert the sequence to restore the new context. It pops the values for `%edi`, `%esi`, `%ebx`, and `%ebp` and then returns (2373-2377). Because `swtch` has changed the stack pointer, the values restored and the instruction address returned to are the ones from the new context.

In our example, `sched` called `swtch` to switch to `cpu->scheduler`, the per-CPU scheduler context. That context had been saved by `scheduler`'s call to `swtch` (2128). When the `swtch` we have been tracing returns, it returns not to `sched` but to `scheduler`, and its stack pointer points at the current CPU's scheduler stack, not `initproc`'s kernel stack.

Code: Scheduling

The last section looked at the low-level details of `swtch`; now let's take `swtch` as a

given and examine the conventions involved in switching from process to scheduler and back to process. A process that wants to give up the CPU must acquire the process table lock `ptable.lock`, release any other locks it is holding, update its own state (`proc->state`), and then call `sched`. `Yield` (2172) follows this convention, as do `sleep` and `exit`, which we will examine later. `Sched` double-checks those conditions (2157-2162) and then an implication of those conditions: since a lock is held, the CPU should be running with interrupts disabled. Finally, `sched` calls `swtch` to save the current context in `proc->context` and switch to the scheduler context in `cpu->scheduler`. `Swtch` returns on the scheduler's stack as though `scheduler`'s `swtch` had returned (2128). The scheduler continues the `for` loop, finds a process to run, switches to it, and the cycle repeats.

We just saw that xv6 holds `ptable.lock` across calls to `swtch`: the caller of `swtch` must already hold the lock, and control of the lock passes to the switched-to code. This convention is unusual with locks; the typical convention is the thread that acquires a lock is also responsible of releasing the lock, which makes it easier to reason about correctness. For context switching is necessary to break the typical convention because `ptable.lock` protects invariants on the process's `state` and `context` fields that are not true while executing in `swtch`. One example of a problem that could arise if `ptable.lock` were not held during `swtch`: a different CPU might decide to run the process after `yield` had set its state to `RUNNABLE`, but before `swtch` caused it to stop using its own kernel stack. The result would be two CPUs running on the same stack, which cannot be right.

A kernel thread always gives up its processor in `sched` and always switches to the same location in the scheduler, which (almost) always switches to a process in `sched`. Thus, if one were to print out the line numbers where xv6 switches threads, one would observe the following simple pattern: (2128), (2166), (2128), (2166), and so on. The procedures in which this stylized switching between two threads happens are sometimes referred to as co-routines; in this example, `sched` and `scheduler` are co-routines of each other.

There is one case when the scheduler's `swtch` to a new process does not end up in `sched`. We saw this case in Chapter 2: when a new process is first scheduled, it begins at `forkret` (2183). `Forkret` exists only to honor this convention by releasing the `ptable.lock`; otherwise, the new process could start at `trapret`.

`Scheduler` (2108) runs a simple loop: find a process to run, run it until it stops, repeat. `scheduler` holds `ptable.lock` for most of its actions, but releases the lock (and explicitly enables interrupts) once in each iteration of its outer loop. This is important for the special case in which this CPU is idle (can find no `RUNNABLE` process). If an idling scheduler looped with the lock continuously held, no other CPU that was running a process could ever perform a context switch or any process-related system call, and in particular could never mark a process as `RUNNABLE` so as to break the idling CPU out of its scheduling loop. The reason to enable interrupts periodically on an idling CPU is that there might be no `RUNNABLE` process because processes (e.g., the shell) are waiting for I/O; if the scheduler left interrupts disabled all the time, the I/O would never arrive.

The scheduler loops over the process table looking for a runnable process, one

that has p->state == RUNNABLE. Once it finds a process, it sets the per-CPU current process variable proc, switches to the process's page table with switchuvm, marks the process as RUNNING, and then calls swtch to start running it (2122-2128).

One way to think about the structure of the scheduling code is that it arranges to enforce a set of invariants about each process, and holds ptable.lock whenever those invariants are not true. One invariant is that if a process is RUNNING, things must be set up so that a timer interrupt's yield can correctly switch away from the process; this means that the CPU registers must hold the process's register values (i.e. they aren't actually in a context), cr3 must refer to the process's pagetable, esp must refer to the process's kernel stack so that swtch can push registers correctly, and proc must refer to the process's proc[] slot. Another invariant is that if a process is RUNNABLE, things must be set up so that an idle CPU's scheduler can run it; this means that p->context must hold the process's kernel thread variables, that no CPU is executing on the process's kernel stack, that no CPU's cr3 refers to the process's page table, and that no CPU's proc refers to the process.

Maintaining the above invariants is the reason why xv6 acquires ptable.lock in one thread (often in yield) and releases the lock in a different thread (the scheduler thread or another next kernel thread). Once the code has started to modify a running process's state to make it RUNNABLE, it must hold the lock until it has finished restoring the invariants: the earliest correct release point is after scheduler stops using the process's page table and clears proc. Similarly, once scheduler starts to convert a runnable process to RUNNING, the lock cannot be released until the kernel thread is completely running (after the swtch, e.g. in yield).

ptable.lock protects other things as well: allocation of process IDs and free process table slots, the interplay between exit and wait, the machinery to avoid lost wakeups (see next section), and probably other things too. It might be worth thinking about whether the different functions of ptable.lock could be split up, certainly for clarity and perhaps for performance.

Sleep and wakeup

Locks help CPUs and processes avoid interfering with each other, and scheduling helps processes share a CPU, but so far we have no abstractions that make it easy for processes to communicate. Sleep and wakeup fill that void, allowing one process to sleep waiting for an event and another process to wake it up once the event has happened. Sleep and wakeup are often called sequence coordination mechanisms, and there are many other such mechanisms in the operating systems literature.

To illustrate what we mean, let's consider a simple producer/consumer queue. The queue allows one process to send a nonzero pointer to another process. Assuming there is only one sender and one receiver and they execute on different CPUs, this implementation is correct:

```
100     struct q {
101       void *ptr;
102     };
103
104     void*
105     send(struct q *q, void *p)
106     {
107       while(q->ptr != 0)
108         ;
109       q->ptr = p;
110     }
111
112     void*
113     recv(struct q *q)
114     {
115       void *p;
116
117       while((p = q->ptr) == 0)
118         ;
119       q->ptr = 0;
120       return p;
121     }
```

`Send` loops until the queue is empty (`ptr == 0`) and then puts the pointer `p` in the queue. `Recv` loops until the queue is non-empty and takes the pointer out. When run in different processes, `send` and `recv` both edit `q->ptr`, but `send` only writes to the pointer when it is zero and `recv` only writes to the pointer when it is nonzero, so they do not step on each other.

The implementation above may be correct, but it is very expensive. If the sender sends rarely, the receiver will spend most of its time spinning in the `while` loop hoping for a pointer. The receiver's CPU could find more productive work if there were a way for the receiver to be notified when the `send` had delivered a pointer.

Let's imagine a pair of calls, `sleep` and `wakeup`, that work as follows. `Sleep(chan)` sleeps on the arbitrary value `chan`, called the wait channel. `Sleep` puts the calling process to sleep, releasing the CPU for other work. `Wakeup(chan)` wakes all processes sleeping on `chan` (if any), causing their `sleep` calls to return. If no processes are waiting on `chan`, `wakeup` does nothing. We can change the queue implementation to use `sleep` and `wakeup`:

```
201     void*
202     send(struct q *q, void *p)
203     {
204       while(q->ptr != 0)
205         ;
206       q->ptr = p;
207       wakeup(q);  /* wake recv */
208     }
209
210     void*
211     recv(struct q *q)
212     {
213       void *p;
```

```
214
215       while((p = q->ptr) == 0)
216         sleep(q);
217       q->ptr = 0;
218       return p;
219     }
```

`recv` now gives up the CPU instead of spinning, which is nice. However, it turns out not to be straightforward to design `sleep` and `wakeup` with this interface without suffering from what is known as the "lost wake up" problem. Suppose that `recv` finds that `q->ptr == 0` on line 215 and decides to call `sleep`. Before `recv` can sleep, `send` runs on another CPU: it changes `q->ptr` to be nonzero and calls `wakeup`, which finds no processes sleeping and thus does nothing. Now `recv` continues executing at line 216: it calls `sleep` and goes to sleep. This causes a problem: `recv` is asleep waiting for a pointer that has already arrived. The next `send` will sleep waiting for `recv` to consume the pointer in the queue, at which point the system will be deadlocked.

The root of this problem is that the invariant that `recv` only sleeps when `q->ptr == 0` is violated by `send` running at just the wrong moment. To protect this invariant, we introduce a lock, which `sleep` releases only after the calling process is asleep; this avoids the missed wakeup in the example above. Once the calling process is awake again `sleep` reacquires the lock before returning. We would like to be able to have the following code:

```
300     struct q {
301       struct spinlock lock;
302       void *ptr;
303     };
304
305     void*
306     send(struct q *q, void *p)
307     {
308       acquire(&q->lock);
309       while(q->ptr != 0)
310         ;
311       q->ptr = p;
312       wakeup(q);
313       release(&q->lock);
314     }
315
316     void*
317     recv(struct q *q)
318     {
319       void *p;
320
321       acquire(&q->lock);
322       while((p = q->ptr) == 0)
323         sleep(q, &q->lock);
324       q->ptr = 0;
325       release(&q->lock);
326       return p;
327     }
```

The fact that `recv` holds `q->lock` prevents `send` from trying to wake it up be-

tween recv's check of q->ptr and its call to sleep. Of course, the receiving process had better not hold q->lock while it is sleeping, since that would prevent the sender from waking it up, and lead to deadlock. So what we want is for sleep to atomically release q->lock and put the receiving process to sleep.

A complete sender/receiver implementation would also sleep in send when waiting for a receiver to consume the value from a previous send.

Code: Sleep and wakeup

Let's look at the implementation of sleep and wakeup in xv6. The basic idea is to have sleep mark the current process as SLEEPING and then call sched to release the processor; wakeup looks for a process sleeping on the given pointer and marks it as RUNNABLE.

Sleep (2203) begins with a few sanity checks: there must be a current process (2205) and sleep must have been passed a lock (2208-2209). Then sleep acquires ptable.lock (2218). Now the process going to sleep holds both ptable.lock and lk. Holding lk was necessary in the caller (in the example, recv): it ensured that no other process (in the example, one running send) could start a call wakeup(chan). Now that sleep holds ptable.lock, it is safe to release lk: some other process may start a call to wakeup(chan), but wakeup will not run until it can acquire ptable.lock, so it must wait until sleep has finished putting the process to sleep, keeping the wakeup from missing the sleep.

There is a minor complication: if lk is equal to &ptable.lock, then sleep would deadlock trying to acquire it as &ptable.lock and then release it as lk. In this case, sleep considers the acquire and release to cancel each other out and skips them entirely (2217).

Now that sleep holds ptable.lock and no others, it can put the process to sleep by recording the sleep channel, changing the process state, and calling sched (2223-2225).

At some point later, a process will call wakeup(chan). Wakeup (2253) acquires ptable.lock and calls wakeup1, which does the real work. It is important that wakeup hold the ptable.lock both because it is manipulating process states and because, as we just saw, ptable.lock makes sure that sleep and wakeup do not miss each other. Wakeup1 is a separate function because sometimes the scheduler needs to execute a wakeup when it already holds the ptable.lock; we will see an example of this later. Wakeup1 (2253) loops over the process table. When it finds a process in state SLEEPING with a matching chan, it changes that process's state to RUNNABLE. The next time the scheduler runs, it will see that the process is ready to be run.

wakeup must always be called while holding a lock that prevents observation of whatever the wakeup condition is; in the example above that lock is q->lock. The complete argument for why the sleeping process won't miss a wakeup is that at all times from before it checks the condition until after it is asleep, it holds either the lock on the condition or the ptable.lock or both. Since wakeup executes while holding both of those locks, the wakeup must execute either before the potential sleeper checks the condition, or after the potential sleeper has completed putting itself to sleep.

It is sometimes the case that multiple processes are sleeping on the same channel;

for example, more than one process trying to read from a pipe. A single call to `wakeup` will wake them all up. One of them will run first and acquire the lock that `sleep` was called with, and (in the case of pipes) read whatever data is waiting in the pipe. The other processes will find that, despite being woken up, there is no data to be read. From their point of view the wakeup was "spurious," and they must sleep again. For this reason sleep is always called inside a loop that checks the condition.

Callers of sleep and wakeup can use any mutually convenient number as the channel; in practice xv6 often uses the address of a kernel data structure involved in the waiting, such as a disk buffer. No harm is done if two uses of sleep/wakeup accidentally choose the same channel: they will see spurious wakeups, but looping as described above will tolerate this problem. Much of the charm of sleep/wakeup is that it is both lightweight (no need to create special data structures to act as sleep channels) and provides a layer of indirection (callers need not know what specific process they are interacting with).

Code: Pipes

The simple queue we used earlier in this Chapter was a toy, but xv6 contains a real queue that uses `sleep` and `wakeup` to synchronize readers and writers. That queue is the implementation of pipes. We saw the interface for pipes in Chapter 0: bytes written to one end of a pipe are copied in an in-kernel buffer and then can be read out of the other end of the pipe. Future chapters will examine the file system support surrounding pipes, but let's look now at the implementations of `pipewrite` and `piperead`.

Each pipe is represented by a `struct pipe`, which contains a `lock` and a `data` buffer. The fields `nread` and `nwrite` count the number of bytes read from and written to the buffer. The buffer wraps around: the next byte written after `buf[PIPESIZE-1]` is `buf[0]`, but the counts do not wrap. This convention lets the implementation distinguish a full buffer (`nwrite == nread+PIPESIZE`) from an empty buffer `nwrite == nread`), but it means that indexing into the buffer must use `buf[nread % PIPESIZE]` instead of just `buf[nread]` (and similarly for `nwrite`). Let's suppose that calls to `piperead` and `pipewrite` happen simultaneously on two different CPUs.

`Pipewrite` (5680) begins by acquiring the pipe's lock, which protects the counts, the data, and their associated invariants. `Piperead` (5701) then tries to acquire the lock too, but cannot. It spins in `acquire` (1573) waiting for the lock. While `piperead` waits, `pipewrite` loops over the bytes being written—`addr[0]`, `addr[1]`, ..., `addr[n-1]`—adding each to the pipe in turn (5694). During this loop, it could happen that the buffer fills (5686). In this case, `pipewrite` calls `wakeup` to alert any sleeping readers to the fact that there is data waiting in the buffer and then sleeps on `&p->nwrite` to wait for a reader to take some bytes out of the buffer. `Sleep` releases `p->lock` as part of putting `pipewrite`'s process to sleep.

Now that `p->lock` is available, `piperead` manages to acquire it and start running in earnest: it finds that `p->nread != p->nwrite` (5706) (`pipewrite` went to sleep because `p->nwrite == p->nread+PIPESIZE` (5686)) so it falls through to the `for` loop, copies data out of the pipe (5713-5717), and increments `nread` by the number of bytes

copied. That many bytes are now available for writing, so `piperead` calls `wakeup` (5718) to wake any sleeping writers before it returns to its caller. `Wakeup` finds a process sleeping on `&p->nwrite`, the process that was running `pipewrite` but stopped when the buffer filled. It marks that process as `RUNNABLE`.

The pipe code uses separate sleep channels for reader and writer (`p->nread` and `p->nwrite`); this might make the system more efficient in the unlikely event that there are lots of readers and writers waiting for the same pipe. The pipe code sleeps inside a loop checking the sleep condition; if there are multiple readers or writers, all but the first process to wake up will see the condition is still false and sleep again.

Code: Wait and exit

`Sleep` and `wakeup` can be used in many kinds of situations involving a condition that can be checked needs to be waited for. As we saw in Chapter 0, a parent process can call `wait` to wait for a child to exit. In xv6, when a child exits, it does not die immediately. Instead, it switches to the `ZOMBIE` process state until the parent calls `wait` to learn of the exit. The parent is then responsible for freeing the memory associated with the process and preparing the `struct proc` for reuse. Each process structure keeps a pointer to its parent in `p->parent`. If the parent exits before the child, the initial process `init` adopts the child and waits for it. This step is necessary to make sure that some process cleans up after the child when it exits. All the process structures are protected by `ptable.lock`.

`Wait` begins by acquiring `ptable.lock`. Then it scans the process table looking for children. If `wait` finds that the current process has children but that none of them have exited, it calls `sleep` to wait for one of the children to exit (2089) and loops. Here, the lock being released in `sleep` is `ptable.lock`, the special case we saw above.

`Exit` acquires `ptable.lock` and then wakes the current process's parent (2026). This may look premature, since `exit` has not marked the current process as a `ZOMBIE` yet, but it is safe: although the parent is now marked as `RUNNABLE`, the loop in `wait` cannot run until `exit` releases `ptable.lock` by calling `sched` to enter the scheduler, so `wait` can't look at the exiting process until after the state has been set to `ZOMBIE` (2038). Before exit reschedules, it reparents all of the exiting process's children, passing them to the `initproc` (2028-2035). Finally, `exit` calls `sched` to relinquish the CPU.

Now the scheduler can choose to run the exiting process's parent, which is asleep in `wait` (2089). The call to `sleep` returns holding `ptable.lock`; `wait` rescans the process table and finds the exited child with `state == ZOMBIE`. (2032). It records the child's `pid` and then cleans up the `struct proc`, freeing the memory associated with the process (2068-2076).

The child process could have done most of the cleanup during `exit`, but it is important that the parent process be the one to free `p->kstack` and `p->pgdir`: when the child runs `exit`, its stack sits in the memory allocated as `p->kstack` and it uses its own pagetable. They can only be freed after the child process has finished running for the last time by calling `swtch` (via `sched`). This is one reason that the scheduler procedure runs on its own stack rather than on the stack of the thread that called `sched`.

Scheduling concerns

XXX checking p->killed

XXX thundering herd

Real world

`Sleep` and `wakeup` are a simple and effective synchronization method, but there are many others. The first challenge in all of them is to avoid the "missed wakeups" problem we saw at the beginning of the chapter. The original Unix kernel's `sleep` simply disabled interrupts, which sufficed because Unix ran on a single-CPU system. Because xv6 runs on multiprocessors, it adds an explicit lock to `sleep`. FreeBSD's `msleep` takes the same approach. Plan 9's `sleep` uses a callback function that runs with the scheduling lock held just before going to sleep; the function serves as a last minute check of the sleep condition, to avoid missed wakeups. The Linux kernel's `sleep` uses an explicit process queue instead of a wait channel; the queue has its own internal lock. (XXX Looking at the code that seems not to be enough; what's going on?)

Scanning the entire process list in `wakeup` for processes with a matching `chan` is inefficient. A better solution is to replace the `chan` in both `sleep` and `wakeup` with a data structure that holds a list of processes sleeping on that structure. Plan 9's `sleep` and `wakeup` call that structure a rendezvous point or `Rendez`. Many thread libraries refer to the same structure as a condition variable; in that context, the operations `sleep` and `wakeup` are called `wait` and `signal`. All of these mechanisms share the same flavor: the sleep condition is protected by some kind of lock dropped atomically during sleep.

Semaphores are another common coordination mechanism. A semaphore is an integer value with two operations, increment and decrement (or up and down). It is aways possible to increment a semaphore, but the semaphore value is not allowed to drop below zero: a decrement of a zero semaphore sleeps until another process increments the semaphore, and then those two operations cancel out. The integer value typically corresponds to a real count, such as the number of bytes available in a pipe buffer or the number of zombie children that a process has. Using an explicit count as part of the abstraction avoids the "missed wakeup" problem: there is an explicit count of the number of wakeups that have occurred. The count also avoids the spurious wakeup and thundering herd problems inherent in condition variables.

Exercises:

Sleep has to check lk != &ptable.lock to avoid a deadlock (2217-2220). It could eliminate the special case by replacing

```
if(lk != &ptable.lock){
  acquire(&ptable.lock);
  release(lk);
}
```

with

```
release(lk);
acquire(&ptable.lock);
```

Doing this would break `sleep`. How?

Most process cleanup could be done by either `exit` or `wait`, but we saw above that `exit` must not free `p->stack`. It turns out that `exit` must be the one to close the open files. Why? The answer involves pipes.

Implement semaphores in xv6. You can use mutexes but do not use sleep and wakeup. Replace the uses of sleep and wakeup in xv6 with semaphores. Judge the result.

Additional reading:

cox and mullender, semaphores.

pike et al, sleep and wakeup