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 has provides this plan. If two different processes are competing for a single CPU, xv6 multiplexes them, switching many times per second between executing one and the other. Xv6 uses multiplexing to create 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 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 to force context switch in the timer interrupt handler, Third, may processes may be switching concurrently, and a locking plan is necessary to avoid races. Fourth, when a process completed its execution, it shouldn't be multiplexed with other processes, but cleaning a process is not easy; it cannot clean up itself since that requires that it runs. Xv6 tries to solve these problems as straightforward as possible, but nevertheless the resulting code is tricky.

Once there are multiple processes executing, xv6 must also provide some way for them to coordinate among themselves. Often it is necessary for one process to wait for another to perform some action. Rather than make the waiting process waste CPU by repeatedly checking whether that action has happened, xv6 allows a process to sleep waiting for an event and allows another process to wake the first process. Because processes run in parallel, there is a risk of losing a wake up. As an example of these problems and their solution, this chapter examines the implementation of pipes.

Code: Scheduler

Chapter 2 breezed through the scheduler on the way to user space. Let's take a closer look at it. Each processor runs `mpmain` at boot time; the last thing `mpmain` does is call `scheduler` (1263).

`Scheduler` (1908) runs a simple loop: find a process to run, run it until it stops, repeat. At the beginning of the loop, `scheduler` enables interrupts with an explicit `sti` (1914), so that if a hardware interrupt is waiting to be handled, the scheduler's CPU will handle it before continuing. Then 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 `cp`, updates the user segments with `usegment`, marks the process as `RUNNING`, and then calls `swtch` to start running it (1922-1928).

Code: Context switching

Every xv6 process has its own kernel stack and register set, as we saw in Chapter 2. Each CPU has its own kernel stack to use when running the scheduler. `Swtch` saves the scheduler's context—it's stack and registers—and switches to the chosen process's context. When it is time for the process to give up the CPU, it 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 stack involved. `Swtch` takes two arguments `struct context **old` and `struct context *new`; it saves the current context, storing a pointer to it in `*old` and then restores the context described by `new`.

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 `cp->context` and switch to the scheduler context previously saved in `c->context` (1967).

`Swtch` (2202) starts by loading its arguments off the stack into the registers `%eax` and `%edx` (2209-2210); `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 (2213-2216); it saves the last implicitly as the `struct context*` written to `*old` (2219). 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 (2220). 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 (2223-2227). Because `swtch` has changed the stack pointer, the values restored and the address returned to are the ones from the new context.

In our example, `sched`'s called `swtch` to switch to `c->context`, the per-CPU scheduler context. That new context had been saved by `scheduler`'s call to `swtch` (1928). When the `swtch` we have been tracing returns, it returns not to `sched` but to `scheduler`, and its stack pointer points at the 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. The convention in xv6 is that 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 (cp->state), and then call sched. Yield (1973) follows this convention, as do sleep and exit, which we will examine later. Sched double checks those conditions (1957-1962) and then an implication: since a lock is held, the CPU should be running with interrupts disabled. Finally, sched calls swtch to save the current context in cp->context and switch to the scheduler context in c->context. Swtch returns on the scheduler's stack as though scheduler's swtch had returned (1928). 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 the state and context fields in each process structure. Without the lock, it could happen that a process decided to yield, set its state to RUNNABLE, and then before it could swtch to give up the CPU, a different CPU would try to run it using swtch. This other CPU's call to swtch would use a stale context, the one from the last time the process was started, causing time to appear to move backward. It would also cause two CPUs to be executing on the same stack. Both are incorrect.

To avoid this problem, xv6 follows the convention that the thread that releases a processor acquires the ptable.lock lock and the thread that receives that processor next releases the lock. To make this convention clear, a thread gives up its processor always in sched, switches always to the same location in the scheduler thread, which returns a processor always in sched. Thus, if one were to print out the line numbers where xv6 switches threads, one would observe the following simple pattern: (1928), (1967), (1928), (1967), 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 (1984). Forkret exists only to honor this convention by releasing the ptable.lock; otherwise, the new process could start at trapret.

Sleep and wakeup

Locks help CPUs and processes avoid interfering with each other, and scheduling help 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.

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:

```
struct q {
  void *ptr;
};

void*
send(struct q *q, void *p)
{
  while(q->ptr != 0)
    ;
  q->ptr = p;
}

void*
recv(struct q *q)
{
  void *p;

  while((p = q->ptr) == 0)
    ;
  q->ptr = 0;
  return p;
}
```

`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. `Sleep` and `wakeup` provide such a mechanism. `Sleep(chan)` sleeps on the pointer `chan`, called the wait channel, which may be any kind of pointer; it is used only as an identifying address and is not dereferenced. `Sleep` puts the calling process to sleep, releasing the CPU for other work. It does not return until the process is awake again. `Wakeup(chan)` wakes all the processes sleeping on `chan` (if any), causing their `sleep` calls to return. We can change the queue implementation to use `sleep` and `wakeup`:

```
void*
send(struct q *q, void *p)
{
  while(q->ptr != 0)
    ;
  q->ptr = p;
  wakeup(q);  /* wake recv */
}
```

```
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    }
```

This code is more efficient but no longer correct, because it suffers 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. 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. The following code is correct and makes efficient use of the CPU when `recv` must wait:

```
struct q {
  struct spinlock lock;
  void *ptr;
};

void*
send(struct q *q, void *p)
{
  lock(&q->lock);
  while(q->ptr != 0)
    ;
  q->ptr = p;
  wakeup(q);
  unlock(&q->lock);
}

void*
recv(struct q *q)
{
  void *p;

  lock(&q->lock);
  while((p = q->ptr) == 0)
    sleep(q, &q->lock);
  q->ptr = 0;
  unlock(&q->lock);
```

```
    return p;
  }
```

A complete 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` (2003) begins with a few sanity checks: there must be a current process (2005-2006) and `sleep` must have been passed a lock (2008-2009). Then `sleep` acquires `ptable.lock` (2018). 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` is done, 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 (2017).

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` (2023-2025).

At some point later, a process will call `wakeup(chan)`. `Wakeup` (2053) 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` (2053) 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.

There is another complication: spurious wakeups.

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 (5230) begins by acquiring the pipe's lock, which protects the counts, the data, and their associated invariants. Piperead (5251) then tries to acquire the lock too, but cannot. It spins in acquire (1373) 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 (5244). During this loop, it could happen that the buffer fills (5236). 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 (5256) (pipewrite went to sleep because p->nwrite == p->nread+PIPESIZE (5236)) so it falls through to the for loop, copies data out of the pipe (5263-5267), and increments nread by the number of bytes copied. That many bytes are now available for writing, so piperead calls wakeup (5268) 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.

Let's suppose that the scheduler on the other CPU has decided to run some other process, so pipewrite does not start running again immediately. Instead, piperead returns to its caller, who then calls piperead again. Let's also suppose that the first piperead consumed all the data from the pipe buffer, so now p->nread == p->nwrite. Piperead sleeps on &p->nread to await more data (5261). Once the process calling piperead is asleep, the CPU can run pipewrite's process, causing sleep to return (5242). Pipewrite finishes its loop, copying the remainder of its data into the buffer (5244). Before returning, pipewrite calls wakeup in case there are any readers waiting for the new data (5246). There is one, the piperead we just left. It continues running (pipe.c/piperead-sleep/) and copies the new data out of the pipe.

Code: Wait and exit

Sleep and wakeup do not have to be used for implementing queues. They work for any condition that can be checked in code and 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 (2188) 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 (2126). 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` (2138). Before exit reschedules, it reparents all of the exiting process's children, passing them to the `initproc` (2128-2135). 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` (2188). The call to `sleep` returns holding `ptable.lock`; `wait` rescans the process table and finds the exited child with `state == ZOMBIE`. (2132). It records the child's `pid` and then cleans up the `struct proc`, freeing the memory associated with the process (2168-2175).

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`: when the child runs `exit`, its stack sits in the memory allocated as `p->kstack`. The stack can only be freed once the child process has called `swtch` (via `sched`) and moved off it. This reason is the main one that the scheduler procedure runs on its own stack, and that xv6 organizes `sched` and `scheduler` as co-routines. Xv6 couldn't invoke the procedure `scheduler` directly from the child, because that procedure would then be running on a stack that might be removed by the parent process calling `wait`.

Scheduling concerns

XXX spurious wakeups

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` disabled interrupts. This sufficed because Unix ran on a single-CPU system. Because xv6 runs on multiprocessors, it added 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 (2017-2020). It could eliminate the special case by replacing

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

with

```
release(lk);
acquire(&ptable.lock);
 .P2
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