Chapter 5 Scheduling

Any operating system is likely to run with more processes than the computer hasprocessors, 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 theoperating system multiplex multiple virtual processors on a single physical processor.This chapter how xv6 multiplexes a processor among several processes.

Multiplexing

Xv6 adopts this multiplexing approach. When a process is waiting for disk request, xv6 puts it to sleep, and schedules another process to run. Furthermore, xv6 using timer interrupts to force a process to stop running on a processor after a fixed-amount of time (100 msec), so that it can schedule another process on the processor.This multiplexing creates the illusion that each process has its own CPU, just as xv6used the memory allocator and hardware page tables to create the illusion that eachprocess has its own memory.

Implementing multiplexing has a few challenges. First, how to switch from oneprocess to another? Xv6 uses the standard mechanism of context switching; althoughthe idea is simple, the code to implement is typically among the most opaque code inan operating system. Second, how to do context switching transparently? Xv6 uses thestandard technique of using the timer interrupt handler to drive context switches.Third, many CPUs may be switching among processes concurrently, and a locking planis necessary to avoid races. Fourth, when a process has exited its memory and otherresources must be freed, but it cannot do all of this itself because (for example) it can'tfree 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 processreading on a pipe may need to wait for some other process to write the pipe. Ratherthan make the waiting process waste CPU by repeatedly checking whether the desiredevent has happened, xv6 allows a process to give up the CPU and sleep waiting for anevent, and allows another process to wake the first process up. Care is needed to avoidraces that result in the loss of event notifications. As an example of these problemsand their solution, this chapter examines the implementation of pipes.

Code: Context switching

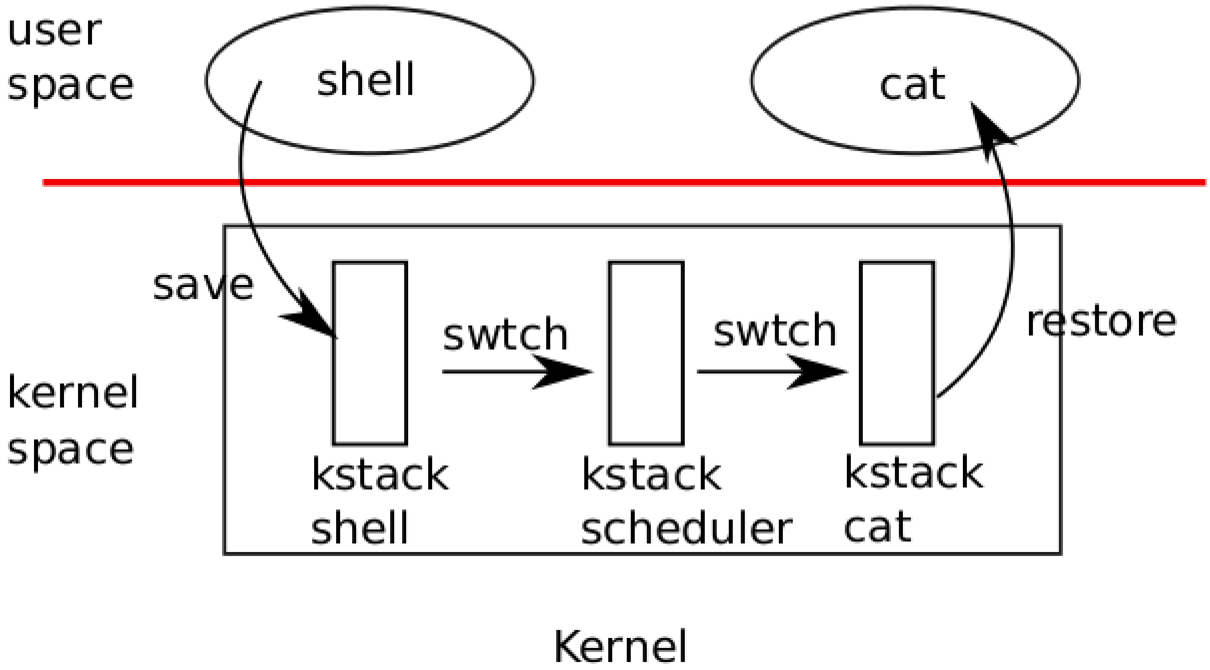

Figure 5-1. Switching from one user process to another. In this example, xv6 runs with one CPU (andthus one scheduler thread).

As shown in Figure 5-1, to switch between processes, xv6 performs two kinds of context switches at a low level: from a process’s kernel thread to the current CPU’sscheduler 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 auser-kernel transition (system call or interrupt), a context switch to the scheduler, acontext switch to a new process’s kernel thread, and a trap return. In this section we’llexample 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 Chapter2. 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 previouslysaved registers ofthe new thread; the fact that %esp and %eip are saved and restored means that theCPU 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 andstruct context *new. It pushes the current CPU register onto the stack and saves thestack 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 processback in. We saw in Chapter 3 that one possibility at the end of each interrupt is thattrap calls yield. Yield in turn calls sched, which calls swtch to save the currentcontext in proc->context and switch to the scheduler context previously saved incpu->scheduler (2516).

Swtch (2702) starts by loading its arguments off the stack into the registers %eaxand %edx (2709-2710); swtch must do this before it changes the stack pointer and can nolonger access the arguments via %esp. Then swtch pushes the register state, creating acontext 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 (2713-2716); it saves the last implicitly as thestruct context* written to *old (2719). There is one more important register: theprogram counter %eip was saved by the call instruction that invoked swtch and ison the stack just above %ebp. Having saved the old context, swtch is ready to restorethe new one. It moves the pointer to the new context into the stack pointer (2720).The new stack has the same form as the old one that swtch just left—the new stackwas 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 thenreturns (2723-2727). Because swtch has changed the stack pointer, the values restoredand 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-CPUscheduler context. That context had been saved by scheduler’s call to swtch (2478).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’skernel stack.

Code: Scheduling

The last section looked at the low-level details of swtch; now let’s take swtch as agiven and examine the conventions involved in switching from process to schedulerand 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 (2522) follows this convention, as do sleepand exit, which we will examine later. Sched double-checks those conditions (2507-2512) and then an implication of those conditions: since a lock is held, the CPU shouldbe running with interrupts disabled. Finally, sched calls swtch to save the currentcontext 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(2478). The scheduler continues the for loop, finds a process to run, switches to it, andthe cycle repeats.

We just saw that xv6 holds ptable.lock across calls to swtch: the caller ofswtch must already hold the lock, and control of the lock passes to the switched-tocode. This convention is unusual with locks; the typical convention is the thread thatacquires a lock is also responsible of releasing the lock, which makes it easier to reasonabout correctness. For context switching is necessary to break the typical conventionbecause ptable.lock protects invariants on the process’s state and context fieldsthat are not true while executing in swtch. One example of a problem that could ariseif ptable.lock were not held during swtch: a different CPU might decide to run theprocess after yield had set its state to RUNNABLE, but before swtch caused it to stopusing 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 thesame 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 wouldobserve the following simple pattern: (2478), (2516), (2478), (2516), and so on. The procedures in which this stylized switching between two threads happens are sometimes referred to as coroutines; in this example, sched and scheduler are co-routines ofeach other.

There is one case when the scheduler’s swtch to a new process does not end upin sched. We saw this case in Chapter 2: when a new process is first scheduled, it begins at forkret (2533). Forkret exists only to honor this convention by releasing theptable.lock; otherwise, the new process could start at trapret.

Scheduler (2458) 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 wasrunning a process could ever perform a context switch or any process-related systemcall, and in particular could never mark a process as RUNNABLE so as to break theidling CPU out of its scheduling loop. The reason to enable interrupts periodically onan idling CPU is that there might be no RUNNABLE process because processes (e.g., theshell) are waiting for I/O; if the scheduler left interrupts disabled all the time, the I/Owould never arrive.

The scheduler loops over the process table looking for a runnable process, onethat has p->state == RUNNABLE. Once it finds a process, it sets the per-CPU currentprocess variable proc, switches to the process's page table with switchuvm, marks theprocess as RUNNING, and then calls swtch to start running it (2472-2478).

One way to think about the structure of the scheduling code is that it arranges toenforce a set of invariants about each process, and holds ptable.lock whenever thoseinvariants are not true. One invariant is that if a process is RUNNING, things must beset 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. theyaren'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 procmust refer to the process's proc[] slot. Another invariant is that if a process isRUNNABLE, things must be set up so that an idle CPU's scheduler can run it; thismeans that p->context must hold the process's kernel thread variables, that no CPUis executing on the process's kernel stack, that no CPU's %cr3 refers to the process'spage table, and that no CPU's proc refers to the process.

Maintaining the above invariants is the reason why xv6 acquires ptable.lock inone thread (often in yield) and releases the lock in a different thread (the schedulerthread or another next kernel thread). Once the code has started to modify a runningprocess's state to make it RUNNABLE, it must hold the lock until it has finished restoringthe 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 arunnable process to RUNNING, the lock cannot be released until the kernel thread iscompletely running (after the swtch, e.g. in yield).

ptable.lock protects other things as well: allocation of process IDs and freeprocess table slots, the interplay between exit and wait, the machinery to avoid lostwakeups (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 forclarity and perhaps for performance.

Sleep and wakeup

Locks help CPUs and processes avoid interfering with each other, and schedulinghelps processes share a CPU, but so far we have no abstractions that make it easy forprocesses to communicate. Sleep and wakeup fill that void, allowing one process tosleep waiting for an event and another process to wake it up once the event has happened. Sleep and wakeup are often called sequence coordination or conditionalsynchronization 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.This queue is similar to the one used by the IDE driver to synchronize a processorand device driver (see Chapter 3), but abstracts all IDE-specific code away. The queueallows one process to send a nonzero pointer to another process. Assuming there isonly 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 thequeue. Recv loops until the queue is non-empty and takes the pointer out. When runin different processes, send and recv both edit q->ptr, but send only writes to thepointer when it is zero and recv only writes to the pointer when it is nonzero, so theydo not step on each other.

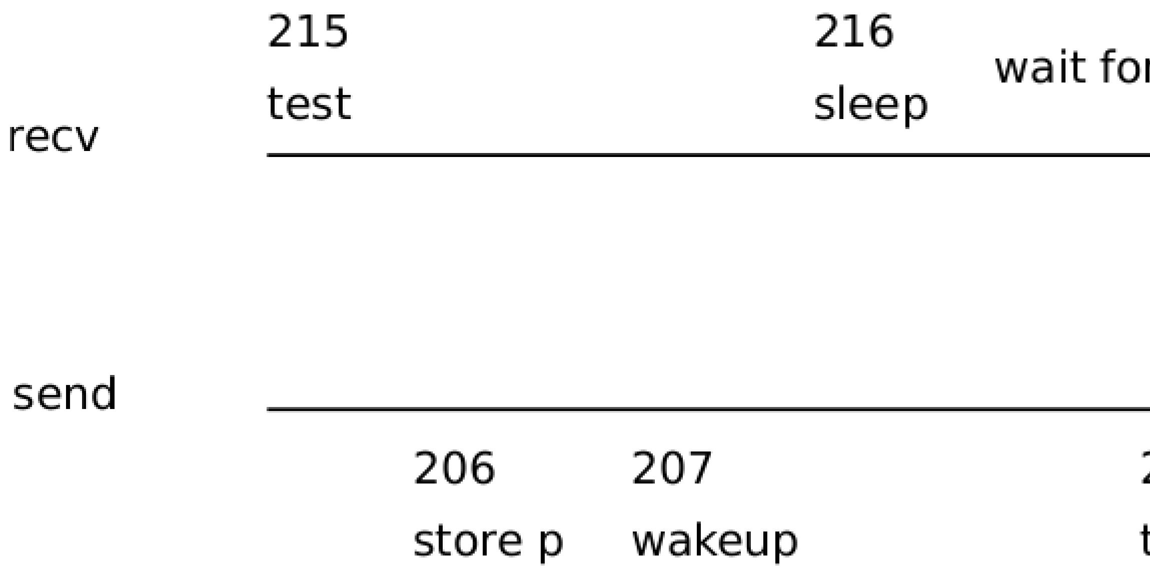

Figure 5-2. Example lost wakeup problem

The implementation above may be correct, but it is expensive. If the sendersends 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 putsthe calling process to sleep, releasing the CPU for other work. Wakeup(chan) wakesall 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:

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

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 turnsout not to be straightforward to design sleep and wakeup with this interface withoutsuffering from what is known as the ‘‘lost wake up’’ problem (see Figure 5-2). Supposethat recv finds that q->ptr == 0 on line 215 and decides to call sleep. Before recvcan sleep (e.g., its processor received an interrupt and the processor is running the interrupt handler, temporarily delaying the call to sleep), send runs on another CPU:it changes q->ptr to be nonzero and calls wakeup, which finds no processes sleepingand thus does nothing. Now recv continues executing at line 216: it calls sleep andgoes 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 inthe 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. One incorrect way ofprotecting the invariant would be to modify the code for recv as follows:

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

void*
send(struct q *q, void *p)
{
  acquire(&q->lock);
```

```
  while(q->ptr != 0)
    ;
  q->ptr = p;
  wakeup(q);
  release(&q->lock);
}

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

  acquire(&q->lock);
  while((p = q->ptr) == 0)
    sleep(q);
  q->ptr = 0;
  release(&q->lock);
  return p;
}
```

This solution protects the invariant because when going calling sleep the process stillholds the q->lock, and send acquires that lock before calling wakeup. sleep will notmiss the wakeup. However, this solution has a deadlock: when recv goes to sleep itholds on to the lock q->lock, and the sender will block when trying to acquire thatlock.

This incorrect implementation makes clear that to protect the invariant, we mustchange the interface of sleep. Sleep must take as argument the lock that sleep canrelease 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 beforereturning. We would like to be able to have the following code:

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

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

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

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

```
  q->ptr = 0;

  release(&q->lock);

  return p;

}
```

The fact that recv holds q->lock prevents send from trying to wake it up between recv’s check of q->ptr and its call to sleep. Of course, the receiving processhad better not hold q->lock while it is sleeping, since that would prevent the senderfrom waking it up, and lead to deadlock. So what we want is for sleep to atomicallyrelease 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 isto have sleep mark the current process as SLEEPING and then call sched to releasethe processor; wakeup looks for a process sleeping on the given pointer and marks itas RUNNABLE.

Sleep (2553) begins with a few sanity checks: there must be a current process(2555) and sleep must have been passed a lock (2558-2559). Then sleep acquires ptable.lock (2568). 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 otherprocess (in the example, one running send) could start a call wakeup(chan). Now thatsleep holds ptable.lock, it is safe to release lk: some other process may start a callto wakeup(chan), but wakeup will not run until it can acquire ptable.lock, so itmust wait until sleep has finished putting the process to sleep, keeping the wakeupfrom missing the sleep.

There is a minor complication: if lk is equal to &ptable.lock, then sleep woulddeadlock 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 (2567). For example, wait (2403) calls sleep with &ptable.lock

Now that sleep holds ptable.lock and no others, it can put the process to sleepby recording the sleep channel, changing the process state, and calling sched (2573-2575).

At some point later, a process will call wakeup(chan). Wakeup (2603) acquires ptable.lock and calls wakeup1, which does the real work. It is important that wakeuphold the ptable.lock both because it is manipulating process states and because, aswe 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 awakeup when it already holds the ptable.lock; we will see an example of this later.Wakeup1 (2603) loops over the process table. When it finds a process in state SLEEPINGwith a matching chan, it changes that process’s state to RUNNABLE. The next time thescheduler runs, it will see that the process is ready to be run.

Wakeup must always be called while holding a lock that prevents observation ofwhatever the wakeup condition is; in the example above that lock is q->lock. Thecomplete argument for why the sleeping process won’t miss a wakeup is that at alltimes from before it checks the condition until after it is asleep, it holds either the lockon the condition or the ptable.lock or both. Since wakeup executes while holdingboth of those locks, the wakeup must execute either before the potential sleeper checksthe 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 wake-up will wake them all up.

One of them will run first and acquire the lock that sleepwas 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. Forthis reason sleep is always called inside a loop that checks the condition.

Callers of sleep and wakeup can use any mutually convenient number as thechannel; in practice xv6 often uses the address of a kernel data structure involved inthe waiting, such as a disk buffer. No harm is done if two uses of sleep/wakeup acci-dentally 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 itis 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 theyare interacting with).

Code: Pipes

The simple queue we used earlier in this chapter was a toy, but xv6 contains two realqueues that uses sleep and wakeup to synchronize readers and writers. One is in theIDE driver: processes add a disk requests to a queue and then calls sleep. The interrupt handler uses wakeup to alert the process that its request has completed.

An more complex example is the implementation of pipes. We saw the interfacefor pipes in Chapter 0: bytes written to one end of a pipe are copied in an in-kernelbuffer 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.D

Each pipe is represented by a struct pipe, which contains a lock and a databuffer. The fields nread and nwrite count the number of bytes read from and writtento 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 topiperead and pipewrite happen simultaneously on two different CPUs.

Pipewrite (6080) begins by acquiring the pipe’s lock, which protects the counts,the data, and their associated invariants. Piperead (6101) then tries to acquire the locktoo, but cannot. It spins in acquire (1474) 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 (6094). During this loop, it could happen that thebuffer fills (6086). In this case, pipewrite calls wakeup to alert any sleeping readers tothe fact that there is data waiting in the buffer and then sleeps on &p->nwrite to waitfor a reader to take some bytes out of the buffer. Sleep releases p->lock as part ofputting pipewrite’s process to sleep.

Now that p->lock is available, piperead manages to acquire it and start runningin earnest: it finds that p->nread != p->nwrite (6106) (pipewrite went to sleep because p->nwrite == p->nread+PIPESIZE (6086)) so it falls through to the for loop,copies data out of the pipe (6113-6117), and increments nread by the number of bytescopied. That many bytes are now available for writing, so piperead calls wakeup (6118)to wake any sleeping writers before it returns to its caller. Wakeup finds a processsleeping on &p->nwrite, the process that was running pipewrite but stopped whenthe buffer filled. It marks that process as RUNNABLE.

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

Code: Wait, exit, and kill

Sleep and wakeup can be used in many kinds of situations involving a condition thatcan be checked needs to be waited for. As we saw in Chapter 0, a parent process cancall 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 tolearn of the exit. The parent is then responsible for freeing the memory associatedwith the process and preparing the struct proc for reuse. Each process structurekeeps 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 surethat some process cleans up after the child when it exits. All the process structures areprotected by ptable.lock.

Wait begins by acquiring ptable.lock. Then it scans the process table lookingfor 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 (2439) 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 (2376).This may look premature, since exit has not marked the current process as a ZOMBIEyet, but it is safe: although the parent is now marked as RUNNABLE, the loop in waitcannot 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(2388). Before exit reschedules, it reparents all of the exiting process's children, passingthem to the initproc (2378-2385). Finally, exit calls sched to relinquish the CPU.

Now the scheduler can choose to run the exiting process's parent, which is asleepin wait (2439). The call to sleep returns holding ptable.lock; wait rescans the process table and finds the exited child with state == ZOMBIE. (2382). It records the child'spid and then cleans up the struct proc, freeing the memory associated with the process (2418-2426).

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 thechild runs exit, its stack sits in the memory allocated as p->kstack and it uses itsown pagetable. They can only be freed after the child process has finished running forthe 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.

Exit allows an application to terminate itself; kill (2625) allows an application toterminate another process. Implementing kill has two challenges: 1) the to-be-killedprocess may be running on another processor and it must switch off its stack to itsprocessor's scheduler before xv6 can terminate it; 2) the to-be-killed may be in sleep,holding kernel resources. To address these challenges, kill does very little: it runsthrough the process table and sets p->killed for the process to be killed and, if it issleeping, it wakes it up. If the to-be-killed process is running another processor, it willenter the kernel at some point soon: either because it calls a system call or an interrupt occurs (e.g., the timer interrupt). When the to-be-killed process leaves the kernelagain, trap checks if p->killed is set, and then the process calls exit, terminating itself.

If the to-be-killed process is in sleep, the call to wakeup will cause the to-be-killed process to run and return from sleep. This is potentially dangerous because theprocess returns from sleep, even though the condition is waiting for may not be true.Xv6 is carefully programmed to use a while loop around each call to sleep, and testsin that while loop if p->killed is set, and, if so, returns to its caller. The caller mustalso check if p->killed is set, and must return to its caller if set, and so on. Eventually the process unwinds its stack to trap, and trap will check p->killed. If it is set,the process calls exit, terminating itself. We see an example of the checking of p->killed in a while loop around sleep in the implementation of pipes (6087).

There is a while loop that doesn't check for p->killed. The ide driver (3979) immediately invokes sleep again. The reason is that it is guaranteed to be woken up because it is waiting for a disk interrupt. And, if it doesn't wait for the disk interrupt,xv6 may get confused. If a second process calls iderw before the outstanding interrupt happens, then ideintr will wake up that second process, instead of the process that was originally waiting for the interrupt. That second process will believe it hasreceived the data it was waiting on, but it has received the data that the first processwas reading!

Real world

The xv6 scheduler implements a simple scheduling policy, which runs each process in turn. This policy is called round robin. Real operating systems implementmore sophisticated policies that, for example, allow processes to have priorities. Theidea is that a runnable high-priority process will be preferred by the scheduler over arunnable low-priority thread. These policies can become complex quickly becausethere are often competing goals: for example, the operating might also want to guarantee fairness and high-throughput. In addition, complex policies may lead to unintended interactions such as priority inversion and convoys. Priority inversion canhappen when a low-priority and high-priority process share a lock, which when acquired by the low-priority process can cause the high-priority process to not run. Along convoy can form when many high-priority processes are waiting for a low-priority process that acquires a shared lock; once a convoy has formed they can persist forlong period of time. To avoid these kinds of problems additional mechanisms are necessary in sophisticated schedulers.

Sleep and wakeup are a simple and effective synchronization method, but thereare 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 sleepsimply 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’smsleep takes the same approach. Plan 9’s sleep uses a callback function that runswith the scheduling lock held just before going to sleep; the function serves as a lastminute check of the sleep condition, to avoid missed wakeups. The Linux kernel’ssleep uses an explicit process queue instead of a wait channel; the queue has its owninternal lock.

Scanning the entire process list in wakeup for processes with a matching chan isinefficient. A better solution is to replace the chan in both sleep and wakeup with adata structure that holds a list of processes sleeping on that structure. Plan 9’s sleepand 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 sleepand 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 duringsleep.

The implementation of wakeup wakes up all processes that are waiting on a par-ticular channel, and it might be the case that many processes are waiting for that particular channel. The operating system will schedules all these processes and they willrace to check the sleep condition. Processes that behave in this way are sometimescalled a thundering herd, and it is best avoided. Most condition variables have twoprimitives for wakeup: signal, which wakes up one process, and broadcast, whichwakes up all processes waiting.

Semaphores are another common coordination mechanism. A semaphore is aninteger value with two operations, increment and decrement (or up and down). It isaways possible to increment a semaphore, but the semaphore value is not allowed todrop below zero: a decrement of a zero semaphore sleeps until another process increments the semaphore, and then those two operations cancel out. The integer valuetypically corresponds to a real count, such as the number of bytes available in a pipebuffer or the number of zombie children that a process has. Using an explicit count aspart of the abstraction avoids the ‘‘missed wakeup’’ problem: there is an explicit countof the number of wakeups that have occurred. The count also avoids the spuriouswakeup and thundering herd problems.

Terminating processes and cleaning them up introduces much complexity in xv6.In most operating systems it is even more complex, because, for example, the to-be-killed process may be deep inside the kernel sleeping, and unwinding its stack requiresmuch careful programming. Many operating system unwind the stack using explicitmechanisms for exception handling, such as longjmp. Furthermore, there are otherevents that can cause a sleeping process to be woken up, even though the events it iswaiting for has not happened yet. For example, when a process is sleeping, anotherprocess may send a it.signalto process will return from the interrupted system callwith the value -1 and with the error code set to EINTR. The application can check forthese values and decide what to do. Xv6 doesn’t support signals and this complexitydoesn’t arise.

Exercises

1. Sleep has to check lk != &ptable.lock to avoid a deadlock (2567-2570). It could eliminate the special case by replacing

```
2. if(lk != &ptable.lock){

3.   acquire(&ptable.lock);

4.   release(lk);

5. }
```

   with

```
release(lk);

acquire(&ptable.lock);
```

   Doing this would break sleep. How?

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

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