**Session No-17**

**Process 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.

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.

**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 each process has its own memory

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 inan operating system.

Second, how to do context switching transparently?

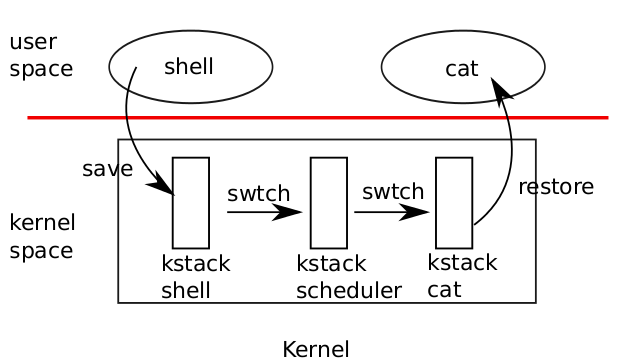
Xv6 uses the standard technique of using the timer interrupt handler to drive context switches.

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, xv6 allows a process to give up the CPU and sleep waiting for ane vent, 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**



Switching from one user process to another. In this example, xv6 runs with one CPU (and thus one scheduler thread).

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’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.

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.

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 [\*\*](https://pekopeko11.sakura.ne.jp/unix_v6/xv6-book/en/Scheduling.html" \l "id1)old and struct context [\*](https://pekopeko11.sakura.ne.jp/unix_v6/xv6-book/en/Scheduling.html" \l "id3)new. It pushes the current CPU register onto the stack and saves the stack pointer in [\*](https://pekopeko11.sakura.ne.jp/unix_v6/xv6-book/en/Scheduling.html" \l "id5)old. Then swtch copies new to %esp, pops previously saved registers, and returns.

starts by loading its arguments off the stack into the registers %eax and %edx ;

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.

it saves the last implicitly as thestruct context\* written to [\*](https://pekopeko11.sakura.ne.jp/unix_v6/xv6-book/en/Scheduling.html" \l "id7)old ().

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 ().

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 .

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 () follows this convention, as do sleep and exit, which we will examine later. Sched double-checks those conditions () 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(). 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 ariseif 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,

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,

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

. Fork ret exists only to honor this convention by releasing the ptable. lock; otherwise, the new process could start at trapret.

Scheduler 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 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/O would never arrive.

The scheduler loops over the process table looking for a runn able 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

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.

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.

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 process device driver (see Chapter 3), but abstracts all IDE-specific code away. The queue allows one process to send a nonzero pointer to another process.

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 runin 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.

Figure 5-2. Example lost wakeup problem

The implementation above may be correct, but it is 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 with out suffering from what is known as the ‘‘lost wake up’’ problem (see Figure 5-2). Suppose that recv finds that q->ptr == 0 on line 215 and decides to call sleep. Before recvcan 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 dead locked.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 of protecting the invariant would be to modify the code for recv as follows:

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);

324 q**->**ptr **=** 0;

325 **release**(**&**q**->**lock);

326 **return** p;

327 }

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.

Sleep must take as argument the lock that sleep can release 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:

400 **struct** q {

401 **struct** spinlock lock;

402 **void** **\***ptr;

403 };

404

405 **void\***

406 **send**(**struct** q **\***q, **void** **\***p)

407 {

408 acquire(**&**q**->**lock);

409 **while**(q**->**ptr **!=** 0)

410 ;

411 q**->**ptr **=** p;

412 **wakeup**(q);

413 **release**(**&**q**->**lock);

414 }

415

416 **void\***

417 recv(**struct** q **\***q)

418 {

419 **void** **\***p;

420

421 **acquire**(**&**q**->**lock);

422 **while**((p **=** q**->**ptr) **==** 0)

423 sleep(q, **&**q**->**lock);

424 q**->**ptr **=** 0;

425 **release**(**&**q**->**lock);

426 **return** p;

427 }

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 process shad 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.

**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.

There is a minor complication: if l k is equal to &ptable.lock, then sleep would deadlock trying to acquire it as &ptable.lock and then release it as l k.

In this case, sleep considers the acquire and release to cancel each other out and skips them entirely calls sleep with &ptable.lock

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

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

Wakeup 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, 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; 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.

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

**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 implement more sophisticated policies that, for example, allow processes to have priorities. The idea is that a runnable high-priority process will be preferred by the scheduler over arunnable low-priority thread. These policies can become complex quickly because there 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 can happen 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 for long 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 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’sm sleep 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.

The implementation of wakeup wakes up all processes that are waiting on a particular 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 will race to check the sleep condition. Processes that behave in this way are some times called a thundering herd, and it is best avoided. Most condition variables have two primitives for wakeup: signal, which wakes up one process, and broadcast, which wakes up all processes waiting.

Semaphores are another common coordination mechanism. A semaphore is an integer value with two operations, increment and decrement (or up and down). It is a ways 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

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 requires much careful programming. Many operating system unwind the stack using explicit mechanisms for exception handling, such as long jmp. Furthermore, there are other events that can cause a sleeping process to be woken up, even though the events it is waiting for has not happened yet. For example, when a process is sleeping, another process may send a it. signal to process will return from the interrupted system call with the value -1 and with the error code set to EINTR. The application can check for these values and decide what to do. Xv6 doesn’t support signals and this complexity doesn’t arise.