

`uservec` restores that saved `tp` when entering the kernel from user space (`kernel/trampoline.S:70`). The compiler guarantees never to use the `tp` register. It would be more convenient if RISC-V allowed `xv6` to read the current hartid directly, but that is allowed only in machine mode, not in supervisor mode.

The return values of `cpuid` and `mycpu` are fragile: if the timer were to interrupt and cause the thread to yield and then move to a different CPU, a previously returned value would no longer be correct. To avoid this problem, `xv6` requires that callers disable interrupts, and only enable them after they finish using the returned `struct cpu`.

The function `myproc` (`kernel/proc.c:68`) returns the `struct proc` pointer for the process that is running on the current CPU. `myproc` disables interrupts, invokes `mycpu`, fetches the current process pointer (`c->proc`) out of the `struct cpu`, and then enables interrupts. The return value of `myproc` is safe to use even if interrupts are enabled: if a timer interrupt moves the calling process to a different CPU, its `struct proc` pointer will stay the same.

7.5 Sleep and wakeup

Scheduling and locks help conceal the existence of one process from another, but so far we have no abstractions that help processes intentionally interact. Many mechanisms have been invented to solve this problem. `Xv6` uses one called sleep and wakeup, which allow 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* or *conditional synchronization* mechanisms.

To illustrate, let's consider a synchronization mechanism called a *semaphore* [4] that coordinates producers and consumers. A semaphore maintains a count and provides two operations. The “V” operation (for the producer) increments the count. The “P” operation (for the consumer) waits until the count is non-zero, and then decrements it and returns. If there were only one producer thread and one consumer thread, and they executed on different CPUs, and the compiler didn't optimize too aggressively, this implementation would be correct:

```
100  struct semaphore {
101      struct spinlock lock;
102      int count;
103  };
104
105  void
106  V(struct semaphore *s)
107  {
108      acquire(&s->lock);
109      s->count += 1;
110      release(&s->lock);
111  }
112
113  void
114  P(struct semaphore *s)
115  {
```

```

116     while(s->count == 0)
117     ;
118     acquire(&s->lock);
119     s->count -= 1;
120     release(&s->lock);
121 }

```

The implementation above is expensive. If the producer acts rarely, the consumer will spend most of its time spinning in the `while` loop hoping for a non-zero count. The consumer's CPU could find more productive work than with *busy waiting* by repeatedly *polling* `s->count`. Avoiding busy waiting requires a way for the consumer to yield the CPU and resume only after `V` increments the count.

Here's a step in that direction, though as we will see it is not enough. 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 semaphore implementation to use `sleep` and `wakeup` (changes highlighted in yellow):

```

200 void
201 V(struct semaphore *s)
202 {
203     acquire(&s->lock);
204     s->count += 1;
205     wakeup(s);
206     release(&s->lock);
207 }
208
209 void
210 P(struct semaphore *s)
211 {
212     while(s->count == 0)
213         sleep(s);
214     acquire(&s->lock);
215     s->count -= 1;
216     release(&s->lock);
217 }

```

`P` 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 `P` finds that `s->count == 0` on line 212. While `P` is between lines 212 and 213, `V` runs on another CPU: it changes `s->count` to be nonzero and calls `wakeup`, which finds no processes sleeping and thus does nothing. Now `P` continues executing at line 213: it calls `sleep` and goes to sleep. This causes a problem: `P` is asleep waiting for a `V` call that has already happened. Unless we get lucky and the producer calls `V` again, the consumer will wait forever even though the count is non-zero.

The root of this problem is that the invariant that `P` only sleeps when `s->count == 0` is violated by `V` running at just the wrong moment. An incorrect way to protect the invariant would be to move the lock acquisition (highlighted in yellow below) in `P` so that its check of the count and its call to `sleep` are atomic:

```

300 void
301 V(struct semaphore *s)
302 {
303     acquire(&s->lock);
304     s->count += 1;
305     wakeup(s);
306     release(&s->lock);
307 }
308
309 void
310 P(struct semaphore *s)
311 {
312     acquire(&s->lock);
313     while(s->count == 0)
314         sleep(s);
315     s->count -= 1;
316     release(&s->lock);
317 }
```

One might hope that this version of `P` would avoid the lost wakeup because the lock prevents `V` from executing between lines 313 and 314. It does that, but it also deadlocks: `P` holds the lock while it sleeps, so `V` will block forever waiting for the lock.

We'll fix the preceding scheme by changing `sleep`'s interface: the caller must pass the *condition lock* to `sleep` so it can release the lock after the calling process is marked as asleep and waiting on the sleep channel. The lock will force a concurrent `V` to wait until `P` has finished putting itself to sleep, so that the wakeup will find the sleeping consumer and wake it up. Once the consumer is awake again `sleep` reacquires the lock before returning. Our new correct sleep/wakeup scheme is usable as follows (change highlighted in yellow):

```

400 void
401 V(struct semaphore *s)
402 {
403     acquire(&s->lock);
404     s->count += 1;
405     wakeup(s);
406     release(&s->lock);
407 }
408
409 void
410 P(struct semaphore *s)
411 {
412     acquire(&s->lock);
```

```

413     while(s->count == 0)
414         sleep(s, &s->lock);
415     s->count -= 1;
416     release(&s->lock);
417 }

```

The fact that `P` holds `s->lock` prevents `V` from trying to wake it up between `P`'s check of `c->count` and its call to `sleep`. Note, however, that we need `sleep` to atomically release `s->lock` and put the consuming process to sleep.

7.6 Code: Sleep and wakeup

Let's look at the implementation of `sleep` (kernel/proc.c:548) and `wakeup` (kernel/proc.c:582). The basic idea is to have `sleep` mark the current process as `SLEEPING` and then call `sched` to release the CPU; `wakeup` looks for a process sleeping on the given wait channel and marks it as `RUNNABLE`. Callers of `sleep` and `wakeup` can use any mutually convenient number as the channel. Xv6 often uses the address of a kernel data structure involved in the waiting.

`sleep` acquires `p->lock` (kernel/proc.c:559). Now the process going to sleep holds both `p->lock` and `lk`. Holding `lk` was necessary in the caller (in the example, `P`): it ensured that no other process (in the example, one running `V`) could start a call to `wakeup(chan)`. Now that `sleep` holds `p->lock`, it is safe to release `lk`: some other process may start a call to `wakeup(chan)`, but `wakeup` will wait to acquire `p->lock`, and thus will 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 the same lock as `p->lock`, then `sleep` would deadlock with itself if it tried to acquire `p->lock`. But if the process calling `sleep` already holds `p->lock`, it doesn't need to do anything more in order to avoiding missing a concurrent `wakeup`. This case arises when `wait` (kernel/proc.c:582) calls `sleep` with `p->lock`.

Now that `sleep` holds `p->lock` and no others, it can put the process to sleep by recording the sleep channel, changing the process state to `SLEEPING`, and calling `sched` (kernel/proc.c:564-567). In a moment it will be clear why it's critical that `p->lock` is not released (by `scheduler`) until after the process is marked `SLEEPING`.

At some point, a process will acquire the condition lock, set the condition that the sleeper is waiting for, and call `wakeup(chan)`. It's important that `wakeup` is called while holding the condition lock¹. `Wakeup` loops over the process table (kernel/proc.c:582). It acquires the `p->lock` of each process it inspects, both because it may manipulate that process's state and because `p->lock` ensures that `sleep` and `wakeup` do not miss each other. When `wakeup` 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.

Why do the locking rules for `sleep` and `wakeup` ensure a sleeping process won't miss a `wakeup`? The sleeping process holds either the condition lock or its own `p->lock` or both from a

¹Strictly speaking it is sufficient if `wakeup` merely follows the `acquire` (that is, one could call `wakeup` after the `release`).

point before it checks the condition to a point after it is marked `SLEEPING`. The process calling `wakeup` holds *both* of those locks in `wakeup`'s loop. Thus the waker either makes the condition true before the consuming thread checks the condition; or the waker's `wakeup` examines the sleeping thread strictly after it has been marked `SLEEPING`. Then `wakeup` will see the sleeping process and wake it up (unless something else wakes it up first).

It is sometimes the case that multiple processes are sleeping on the same channel; for example, more than one process reading 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.

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 which specific process they are interacting with).

7.7 Code: Pipes

A more complex example that uses `sleep` and `wakeup` to synchronize producers and consumers is `xv6`'s implementation of pipes. We saw the interface for pipes in Chapter 1: bytes written to one end of a pipe are copied to an in-kernel buffer and then can be read from the other end of the pipe. Future chapters will examine the file descriptor 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 total number of bytes read from and written to the buffer. The buffer wraps around: the next byte written after `buf[PIPE_SIZE-1]` is `buf[0]`. The counts do not wrap. This convention lets the implementation distinguish a full buffer (`nwrite == nread + PIPE_SIZE`) from an empty buffer (`nwrite == nread`), but it means that indexing into the buffer must use `buf[nread % PIPE_SIZE]` 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` (`kernel/pipe.c:77`) begins by acquiring the pipe's lock, which protects the counts, the data, and their associated invariants. `Piperead` (`kernel/pipe.c:103`) then tries to acquire the lock too, but cannot. It spins in `acquire` (`kernel/spinlock.c:22`) waiting for the lock. While `piperead` waits, `pipewrite` loops over the bytes being written (`addr[0..n-1]`), adding each to the pipe in turn (`kernel/pipe.c:95`). During this loop, it could happen that the buffer fills (`kernel/pipe.c:85`). 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 `&pi->nwrite` to wait for a reader to take some bytes out of the buffer. `sleep` releases `pi->lock` as part of putting `pipewrite`'s process to sleep.

Now that `pi->lock` is available, `piperead` manages to acquire it and enters its critical section: it finds that `pi->nread != pi->nwrite` (`kernel/pipe.c:110`) (`pipewrite` went to sleep be-

cause `pi->nwrite == pi->nread+PIPE_SIZE` (kernel/pipe.c:85)), so it falls through to the `for` loop, copies data out of the pipe (kernel/pipe.c:117), and increments `nread` by the number of bytes copied. That many bytes are now available for writing, so `piperead` calls `wakeup` (kernel/pipe.c:124) to wake any sleeping writers before it returns. `Wakeup` finds a process sleeping on `&pi->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 (`pi->nread` and `pi->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.

7.8 Code: Wait, exit, and kill

Sleep and wakeup can be used for many kinds of waiting. An interesting example, introduced in Chapter 1, is the interaction between a child's `exit` and its parent's `wait`. At the time of the child's death, the parent may already be sleeping in `wait`, or may be doing something else; in the latter case, a subsequent call to `wait` must observe the child's death, perhaps long after it calls `exit`. The way that xv6 records the child's demise until `wait` observes it is for `exit` to put the caller into the `ZOMBIE` state, where it stays until the parent's `wait` notices it, changes the child's state to `UNUSED`, copies the child's exit status, and returns the child's process ID to the parent. If the parent exits before the child, the parent gives the child to the `init` process, which perpetually calls `wait`; thus every child has a parent to clean up after it. The main implementation challenge is the possibility of races and deadlock between parent and child `wait` and `exit`, as well as `exit` and `exit`.

`Wait` uses the calling process's `p->lock` as the condition lock to avoid lost wakeups, and it acquires that lock at the start (kernel/proc.c:398). Then it scans the process table. If it finds a child in `ZOMBIE` state, it frees that child's resources and its `proc` structure, copies the child's exit status to the address supplied to `wait` (if it is not 0), and returns the child's process ID. If `wait` finds children but none have exited, it calls `sleep` to wait for one of them to exit (kernel/proc.c:445), then scans again. Here, the condition lock being released in `sleep` is the waiting process's `p->lock`, the special case mentioned above. Note that `wait` often holds two locks; that it acquires its own lock before trying to acquire any child's lock; and that thus all of xv6 must obey the same locking order (parent, then child) in order to avoid deadlock.

`Wait` looks at every process's `np->parent` to find its children. It uses `np->parent` without holding `np->lock`, which is a violation of the usual rule that shared variables must be protected by locks. It is possible that `np` is an ancestor of the current process, in which case acquiring `np->lock` could cause a deadlock since that would violate the order mentioned above. Examining `np->parent` without a lock seems safe in this case; a process's `parent` field is only changed by its parent, so if `np->parent==p` is true, the value can't change unless the current process changes it.

`Exit` (kernel/proc.c:333) records the exit status, frees some resources, gives any children to

the `init` process, wakes up the parent in case it is in `wait`, marks the caller as a zombie, and permanently yields the CPU. The final sequence is a little tricky. The exiting process must hold its parent's lock while it sets its state to `ZOMBIE` and wakes the parent up, since the parent's lock is the condition lock that guards against lost wakeups in `wait`. The child must also hold its own `p->lock`, since otherwise the parent might see it in state `ZOMBIE` and free it while it is still running. The lock acquisition order is important to avoid deadlock: since `wait` acquires the parent's lock before the child's lock, `exit` must use the same order.

`Exit` calls a specialized wakeup function, `wakeup1`, that wakes up only the parent, and only if it is sleeping in `wait` (`kernel/proc.c:598`). It may look incorrect for the child to wake up the parent before setting its state to `ZOMBIE`, but that is safe: although `wakeup1` may cause the parent to run, the loop in `wait` cannot examine the child until the child's `p->lock` is released by `scheduler`, so `wait` can't look at the exiting process until well after `exit` has set its state to `ZOMBIE` (`kernel/proc.c:386`).

While `exit` allows a process to terminate itself, `kill` (`kernel/proc.c:611`) lets one process request that another terminate. It would be too complex for `kill` to directly destroy the victim process, since the victim might be executing on another CPU, perhaps in the middle of a sensitive sequence of updates to kernel data structures. Thus `kill` does very little: it just sets the victim's `p->killed` and, if it is sleeping, wakes it up. Eventually the victim will enter or leave the kernel, at which point code in `usertrap` will call `exit` if `p->killed` is set. If the victim is running in user space, it will soon enter the kernel by making a system call or because the timer (or some other device) interrupts.

If the victim process is in `sleep`, `kill`'s call to `wakeup` will cause the victim to return from `sleep`. This is potentially dangerous because the condition being waiting for may not be true. However, xv6 calls to `sleep` are always wrapped in a `while` loop that re-tests the condition after `sleep` returns. Some calls to `sleep` also test `p->killed` in the loop, and abandon the current activity if it is set. This is only done when such abandonment would be correct. For example, the pipe read and write code returns if the killed flag is set; eventually the code will return back to `trap`, which will again check the flag and `exit`.

Some xv6 `sleep` loops do not check `p->killed` because the code is in the middle of a multi-step system call that should be atomic. The `virtio` driver (`kernel/virtio_disk.c:242`) is an example: it does not check `p->killed` because a disk operation may be one of a set of writes that are all needed in order for the file system to be left in a correct state. A process that is killed while waiting for disk I/O won't `exit` until it completes the current system call and `usertrap` sees the killed flag.

7.9 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 a runnable low-priority process. 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 unin-

tended 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 prevent the high-priority process from making progress. A long convoy of waiting processes can form when many high-priority processes are waiting for a low-priority process that acquires a shared lock; once a convoy has formed it can persist for long 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 “lost 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 lost wakeups. The Linux kernel’s `sleep` uses an explicit process queue, called a wait queue, instead of a wait channel; the queue has its own internal lock.

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, such as Linux’s wait queue. 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.

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 schedule all these processes and they will race to check the sleep condition. Processes that behave in this way are sometimes 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 waiting processes.

Semaphores are often used for synchronization. The count typically corresponds to something like 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 “lost 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.

Terminating processes and cleaning them up introduces much complexity in xv6. In most operating systems it is even more complex, because, for example, the victim process may be deep inside the kernel sleeping, and unwinding its stack requires much careful programming. Many operating systems unwind the stack using explicit mechanisms for exception handling, such as `longjmp`. Furthermore, there are other events that can cause a sleeping process to be woken up, even though the event it is waiting for has not happened yet. For example, when a Unix process is sleeping, another process may send a `signal` to it. In this case, the 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.

Xv6's support for `kill` is not entirely satisfactory: there are sleep loops which probably should check for `p->killed`. A related problem is that, even for sleep loops that check `p->killed`, there is a race between `sleep` and `kill`; the latter may set `p->killed` and try to wake up the victim just after the victim's loop checks `p->killed` but before it calls `sleep`. If this problem occurs, the victim won't notice the `p->killed` until the condition it is waiting for occurs. This may be quite a bit later (e.g., when the virtio driver returns a disk block that the victim is waiting for) or never (e.g., if the victim is waiting for input from the console, but the user doesn't type any input).

A real operating system would find free `proc` structures with an explicit free list in constant time instead of the linear-time search in `allocproc`; xv6 uses the linear scan for simplicity.

7.10 Exercises

1. Sleep has to check `lk != &p->lock` to avoid a deadlock (kernel/proc.c:558-561). Suppose the special case were eliminated by replacing

```
if(lk != &p->lock){
    acquire(&p->lock);
    release(lk);
}
```

with

```
release(lk);
acquire(&p->lock);
```

Doing this would break `sleep`. How?

2. Most process cleanup could be done by either `exit` or `wait`. It turns out that `exit` must be the one to close the open files. Why? The answer involves pipes.
3. Implement semaphores in xv6 without using `sleep` and `wakeup` (but it is OK to use spin locks). Replace the uses of `sleep` and `wakeup` in xv6 with semaphores. Judge the result.
4. Fix the race mentioned above between `kill` and `sleep`, so that a `kill` that occurs after the victim's sleep loop checks `p->killed` but before it calls `sleep` results in the victim abandoning the current system call.
5. Design a plan so that every sleep loop checks `p->killed` so that, for example, a process that is in the virtio driver can return quickly from the while loop if it is killed by another process.
6. Modify xv6 to use only one context switch when switching from one process's kernel thread to another, rather than switching through the scheduler thread. The yielding thread will need to select the next thread itself and call `swtch`. The challenges will be to prevent multiple cores from executing the same thread accidentally; to get the locking right; and to avoid deadlocks.

7. Modify xv6's `scheduler` to use the RISC-V `WFI` (wait for interrupt) instruction when no processes are runnable. Try to ensure that, any time there are runnable processes waiting to run, no cores are pausing in `WFI`.
8. The lock `p->lock` protects many invariants, and when looking at a particular piece of xv6 code that is protected by `p->lock`, it can be difficult to figure out which invariant is being enforced. Design a plan that is more clean by splitting `p->lock` into several locks.