Lock	Description
bcache.lock	Protects allocation of block buffer cache entries
cons.lock	Serializes access to console hardware, avoids intermixed output
ftable.lock	Serializes allocation of a struct file in file table
icache.lock	Protects allocation of inode cache entries
vdisk_lock	Serializes access to disk hardware and queue of DMA descriptors
kmem.lock	Serializes allocation of memory
log.lock	Serializes operations on the transaction log
pipe's pi->lock	Serializes operations on each pipe
pid_lock	Serializes increments of next_pid
proc's p->lock	Serializes changes to process's state
tickslock	Serializes operations on the ticks counter
inode's ip->lock	Serializes operations on each inode and its content
buf's b->lock	Serializes operations on each block buffer

Figure 6.3: Locks in xv6

B, and the other path acquires them in the order B then A. Suppose thread T1 executes code path 1 and acquires lock A, and thread T2 executes code path 2 and acquires lock B. Next T1 will try to acquire lock B, and T2 will try to acquire lock A. Both acquires will block indefinitely, because in both cases the other thread holds the needed lock, and won't release it until its acquire returns. To avoid such deadlocks, all code paths must acquire locks in the same order. The need for a global lock acquisition order means that locks are effectively part of each function's specification: callers must invoke functions in a way that causes locks to be acquired in the agreed-on order.

Xv6 has many lock-order chains of length two involving per-process locks (the lock in each `struct proc`) due to the way that `sleep` works (see Chapter 7). For example, `consoleintr` (kernel/console.c:138) is the interrupt routine which handles typed characters. When a newline arrives, any process that is waiting for console input should be woken up. To do this, `consoleintr` holds `cons.lock` while calling `wakeup`, which acquires the waiting process's lock in order to wake it up. In consequence, the global deadlock-avoiding lock order includes the rule that `cons.lock` must be acquired before any process lock. The file-system code contains xv6's longest lock chains. For example, creating a file requires simultaneously holding a lock on the directory, a lock on the new file's inode, a lock on a disk block buffer, the disk driver's `vdisk_lock`, and the calling process's `p->lock`. To avoid deadlock, file-system code always acquires locks in the order mentioned in the previous sentence.

Honoring a global deadlock-avoiding order can be surprisingly difficult. Sometimes the lock order conflicts with logical program structure, e.g., perhaps code module M1 calls module M2, but the lock order requires that a lock in M2 be acquired before a lock in M1. Sometimes the identities of locks aren't known in advance, perhaps because one lock must be held in order to discover the identity of the lock to be acquired next. This kind of situation arises in the file system as it looks up successive components in a path name, and in the code for `wait` and `exit` as they search the table of processes looking for child processes. Finally, the danger of deadlock is often a constraint on

how fine-grained one can make a locking scheme, since more locks often means more opportunity for deadlock. The need to avoid deadlock is often a major factor in kernel implementation.

6.5 Locks and interrupt handlers

Some xv6 spinlocks protect data that is used by both threads and interrupt handlers. For example, the `clockintr` timer interrupt handler might increment `ticks` (kernel/trap.c:163) at about the same time that a kernel thread reads `ticks` in `sys_sleep` (kernel/sysproc.c:64). The lock `tickslock` serializes the two accesses.

The interaction of spinlocks and interrupts raises a potential danger. Suppose `sys_sleep` holds `tickslock`, and its CPU is interrupted by a timer interrupt. `clockintr` would try to acquire `tickslock`, see it was held, and wait for it to be released. In this situation, `tickslock` will never be released: only `sys_sleep` can release it, but `sys_sleep` will not continue running until `clockintr` returns. So the CPU will deadlock, and any code that needs either lock will also freeze.

To avoid this situation, if a spinlock is used by an interrupt handler, a CPU must never hold that lock with interrupts enabled. Xv6 is more conservative: when a CPU acquires any lock, xv6 always disables interrupts on that CPU. Interrupts may still occur on other CPUs, so an interrupt's `acquire` can wait for a thread to release a spinlock; just not on the same CPU.

xv6 re-enables interrupts when a CPU holds no spinlocks; it must do a little book-keeping to cope with nested critical sections. `acquire` calls `push_off` (kernel/spinlock.c:89) and `release` calls `pop_off` (kernel/spinlock.c:100) to track the nesting level of locks on the current CPU. When that count reaches zero, `pop_off` restores the interrupt enable state that existed at the start of the outermost critical section. The `intr_off` and `intr_on` functions execute RISC-V instructions to disable and enable interrupts, respectively.

It is important that `acquire` call `push_off` strictly before setting `lk->locked` (kernel/spinlock.c:28). If the two were reversed, there would be a brief window when the lock was held with interrupts enabled, and an unfortunately timed interrupt would deadlock the system. Similarly, it is important that `release` call `pop_off` only after releasing the lock (kernel/spinlock.c:66).

6.6 Instruction and memory ordering

It is natural to think of programs executing in the order in which source code statements appear. Many compilers and CPUs, however, execute code out of order to achieve higher performance. If an instruction takes many cycles to complete, a CPU may issue the instruction early so that it can overlap with other instructions and avoid CPU stalls. For example, a CPU may notice that in a serial sequence of instructions A and B are not dependent on each other. The CPU may start instruction B first, either because its inputs are ready before A's inputs, or in order to overlap execution of A and B. A compiler may perform a similar re-ordering by emitting instructions for one statement before the instructions for a statement that precedes it in the source.

Compilers and CPUs follow rules when they re-order to ensure that they don't change the results of correctly-written serial code. However, the rules do allow re-ordering that changes the

results of concurrent code, and can easily lead to incorrect behavior on multiprocessors [2, 3]. The CPU's ordering rules are called the *memory model*.

For example, in this code for `push`, it would be a disaster if the compiler or CPU moved the store corresponding to line 4 to a point after the `release` on line 6:

```
1       l = malloc(sizeof *l);
2       l->data = data;
3       acquire(&listlock);
4       l->next = list;
5       list = l;
6       release(&listlock);
```

If such a re-ordering occurred, there would be a window during which another CPU could acquire the lock and observe the updated `list`, but see an uninitialized `list->next`.

To tell the hardware and compiler not to perform such re-orderings, xv6 uses `__sync_synchronize()` in both `acquire` (kernel/spinlock.c:22) and `release` (kernel/spinlock.c:47). `__sync_synchronize()` is a *memory barrier*: it tells the compiler and CPU to not reorder loads or stores across the barrier. The barriers in xv6's `acquire` and `release` force order in almost all cases where it matters, since xv6 uses locks around accesses to shared data. Chapter 9 discusses a few exceptions.

6.7 Sleep locks

Sometimes xv6 needs to hold a lock for a long time. For example, the file system (Chapter 8) keeps a file locked while reading and writing its content on the disk, and these disk operations can take tens of milliseconds. Holding a spinlock that long would lead to waste if another process wanted to acquire it, since the acquiring process would waste CPU for a long time while spinning. Another drawback of spinlocks is that a process cannot yield the CPU while retaining a spinlock; we'd like to do this so that other processes can use the CPU while the process with the lock waits for the disk. Yielding while holding a spinlock is illegal because it might lead to deadlock if a second thread then tried to acquire the spinlock; since `acquire` doesn't yield the CPU, the second thread's spinning might prevent the first thread from running and releasing the lock. Yielding while holding a lock would also violate the requirement that interrupts must be off while a spinlock is held. Thus we'd like a type of lock that yields the CPU while waiting to acquire, and allows yields (and interrupts) while the lock is held.

Xv6 provides such locks in the form of *sleep-locks*. `acquiresleep` (kernel/sleeplock.c:22) yields the CPU while waiting, using techniques that will be explained in Chapter 7. At a high level, a sleep-lock has a `locked` field that is protected by a spinlock, and `acquiresleep` 's call to `sleep` atomically yields the CPU and releases the spinlock. The result is that other threads can execute while `acquiresleep` waits.

Because sleep-locks leave interrupts enabled, they cannot be used in interrupt handlers. Because `acquiresleep` may yield the CPU, sleep-locks cannot be used inside spinlock critical sections (though spinlocks can be used inside sleep-lock critical sections).

Spin-locks are best suited to short critical sections, since waiting for them wastes CPU time; sleep-locks work well for lengthy operations.

6.8 Real world

Programming with locks remains challenging despite years of research into concurrency primitives and parallelism. It is often best to conceal locks within higher-level constructs like synchronized queues, although xv6 does not do this. If you program with locks, it is wise to use a tool that attempts to identify race conditions, because it is easy to miss an invariant that requires a lock.

Most operating systems support POSIX threads (Pthreads), which allow a user process to have several threads running concurrently on different CPUs. Pthreads has support for user-level locks, barriers, etc. Supporting Pthreads requires support from the operating system. For example, it should be the case that if one pthread blocks in a system call, another pthread of the same process should be able to run on that CPU. As another example, if a pthread changes its process's address space (e.g., maps or unmaps memory), the kernel must arrange that other CPUs that run threads of the same process update their hardware page tables to reflect the change in the address space.

It is possible to implement locks without atomic instructions [8], but it is expensive, and most operating systems use atomic instructions.

Locks can be expensive if many CPUs try to acquire the same lock at the same time. If one CPU has a lock cached in its local cache, and another CPU must acquire the lock, then the atomic instruction to update the cache line that holds the lock must move the line from the one CPU's cache to the other CPU's cache, and perhaps invalidate any other copies of the cache line. Fetching a cache line from another CPU's cache can be orders of magnitude more expensive than fetching a line from a local cache.

To avoid the expenses associated with locks, many operating systems use lock-free data structures and algorithms [5, 10]. For example, it is possible to implement a linked list like the one in the beginning of the chapter that requires no locks during list searches, and one atomic instruction to insert an item in a list. Lock-free programming is more complicated, however, than programming locks; for example, one must worry about instruction and memory reordering. Programming with locks is already hard, so xv6 avoids the additional complexity of lock-free programming.

6.9 Exercises

1. Comment out the calls to `acquire` and `release` in `kalloc` (kernel/kalloc.c:69). This seems like it should cause problems for kernel code that calls `kalloc`; what symptoms do you expect to see? When you run xv6, do you see these symptoms? How about when running `usertests`? If you don't see a problem, why not? See if you can provoke a problem by inserting dummy loops into the critical section of `kalloc`.

2. Suppose that you instead commented out the locking in `kfree` (after restoring locking in `kalloc`). What might now go wrong? Is lack of locks in `kfree` less harmful than in `kalloc`?

3. If two CPUs call `kalloc` at the same time, one will have to wait for the other, which is bad for performance. Modify `kalloc.c` to have more parallelism, so that simultaneous calls to `kalloc` from different CPUs can proceed without waiting for each other.

4. Write a parallel program using POSIX threads, which is supported on most operating systems. For example, implement a parallel hash table and measure if the number of puts/gets scales with increasing number of cores.

5. Implement a subset of Pthreads in xv6. That is, implement a user-level thread library so that a user process can have more than 1 thread and arrange that these threads can run in parallel on different CPUs. Come up with a design that correctly handles a thread making a blocking system call and changing its shared address space.

Chapter 7

Scheduling

Any operating system is likely to run with more processes than the computer has CPUs, so a plan is needed to time-share the CPUs among the processes. Ideally the sharing would be transparent to user processes. A common approach is to provide each process with the illusion that it has its own virtual CPU by *multiplexing* the processes onto the hardware CPUs. This chapter explains how xv6 achieves this multiplexing.

7.1 Multiplexing

Xv6 multiplexes by switching each CPU from one process to another in two situations. First, xv6's `sleep` and `wakeup` mechanism switches when a process waits for device or pipe I/O to complete, or waits for a child to exit, or waits in the `sleep` system call. Second, xv6 periodically forces a switch to cope with processes that compute for long periods without sleeping. This multiplexing creates the illusion that each process has its own CPU, just as xv6 uses the memory allocator and hardware page tables to create the illusion that each process has its own memory.

Implementing multiplexing poses a few challenges. First, how to switch from one process to another? Although the idea of context switching is simple, the implementation is some of the most opaque code in xv6. Second, how to force switches in a way that is transparent to user processes? Xv6 uses the standard technique of driving context switches with timer interrupts. Third, many CPUs may be switching among processes concurrently, and a locking plan is necessary to avoid races. Fourth, a process's memory and other resources must be freed when the process exits, but it cannot do all of this itself because (for example) it can't free its own kernel stack while still using it. Fifth, each core of a multi-core machine must remember which process it is executing so that system calls affect the correct process's kernel state. Finally, `sleep` and `wakeup` allow a process to give up the CPU and sleep waiting for an event, and allows another process to wake the first process up. Care is needed to avoid races that result in the loss of wakeup notifications. Xv6 tries to solve these problems as simply as possible, but nevertheless the resulting code is tricky.

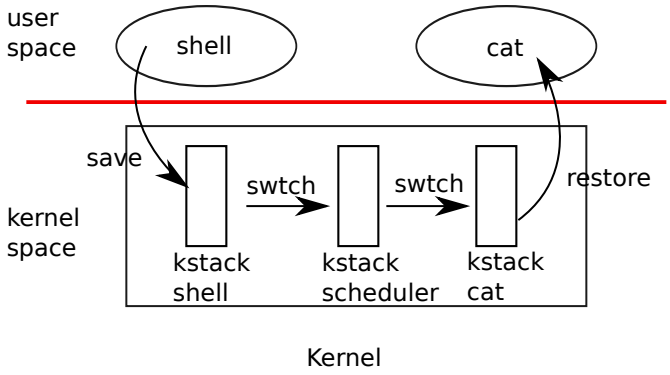

Figure 7.1: Switching from one user process to another. In this example, xv6 runs with one CPU (and thus one scheduler thread).

7.2 Code: Context switching

Figure 7.1 outlines the steps involved in switching from one user process to another: a user-kernel transition (system call or interrupt) to the old process's kernel thread, a context switch to the current CPU's scheduler thread, a context switch to a new process's kernel thread, and a trap return to the user-level process. The xv6 scheduler has a dedicated thread (saved registers and stack) per CPU because it is not safe for the scheduler execute on the old process's kernel stack: some other core might wake the process up and run it, and it would be a disaster to use the same stack on two different cores. In this section we'll examine the mechanics of switching between a kernel thread and a scheduler thread.

Switching from one thread to another involves saving the old thread's CPU registers, and restoring the previously-saved registers of the new thread; the fact that the stack pointer and program counter are saved and restored means that the CPU will switch stacks and switch what code it is executing.

The function `swtch` performs the saves and restores for a kernel thread switch. `swtch` doesn't directly know about threads; it just saves and restores register sets, called *contexts*. When it is time for a process to give up the CPU, the process's kernel thread calls `swtch` to save its own context and return to the scheduler context. Each context is contained in a `struct context` (kernel/proc.h:2), itself contained in a process's `struct proc` or a CPU's `struct cpu`. `Swtch` takes two arguments: `struct context *old` and `struct context *new`. It saves the current registers in `old`, loads registers from `new`, and returns.

Let's follow a process through `swtch` into the scheduler. We saw in Chapter 4 that one possibility at the end of an interrupt is that `usertrap` calls `yield`. `Yield` in turn calls `sched`, which calls `swtch` to save the current context in `p->context` and switch to the scheduler context previously saved in `cpu->scheduler` (kernel/proc.c:509).

`Swtch` (kernel/swtch.S:3) saves only callee-saved registers; caller-saved registers are saved on the stack (if needed) by the calling C code. `Swtch` knows the offset of each register's field in `struct context`. It does not save the program counter. Instead, `swtch` saves the `ra` register, which holds the return address from which `swtch` was called. Now `swtch` restores registers from

the new context, which holds register values saved by a previous `swtch`. When `swtch` returns, it returns to the instructions pointed to by the restored `ra` register, that is, the instruction from which the new thread previously called `swtch`. In addition, it returns on the new thread's stack.

In our example, `sched` called `swtch` to switch to `cpu->scheduler`, the per-CPU scheduler context. That context had been saved by `scheduler`'s call to `swtch` (kernel/proc.c:475). 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.

7.3 Code: Scheduling

The last section looked at the low-level details of `swtch`; now let's take `swtch` as a given and examine switching from one process's kernel thread through the scheduler to another process. The scheduler exists in the form of a special thread per CPU, each running the `scheduler` function. This function is in charge of choosing which process to run next. A process that wants to give up the CPU must acquire its own process lock `p->lock`, release any other locks it is holding, update its own state (`p->state`), and then call `sched`. `Yield` (kernel/proc.c:515) follows this convention, as do `sleep` and `exit`, which we will examine later. `Sched` double-checks those conditions (kernel/proc.c:499-504) and then an implication of those conditions: since a lock is held, interrupts should be disabled. Finally, `sched` calls `swtch` to save the current context in `p->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 `p->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; usually the thread that acquires a lock is also responsible for releasing the lock, which makes it easier to reason about correctness. For context switching it is necessary to break this convention because `p->lock` protects invariants on the process's `state` and `context` fields that are not true while executing in `swtch`. One example of a problem that could arise if `p->lock` were not held during `swtch`: a different CPU might decide to run the process after `yield` had set its state to `RUNNABLE`, but before `swtch` caused it to stop using its own kernel stack. The result would be two CPUs running on the same stack, which cannot be right.

A kernel thread always gives up its CPU in `sched` and always switches to the same location in the scheduler, which (almost) always switches to some kernel thread that previously called `sched`. Thus, if one were to print out the line numbers where xv6 switches threads, one would observe the following simple pattern: (kernel/proc.c:475), (kernel/proc.c:509), (kernel/proc.c:475), (kernel/proc.c:509), 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 of each other.

There is one case when the scheduler's call to `swtch` does not end up in `sched`. When a new process is first scheduled, it begins at `forkret` (kernel/proc.c:527). `Forkret` exists to release the `p->lock`; otherwise, the new process could start at `usertrapret`.

`Scheduler` (kernel/proc.c:457) runs a simple loop: find a process to run, run it until it yields,

repeat. 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 `c->proc`, marks the process as `RUNNING`, and then calls `swtch` to start running it (kernel/proc.c:470-475).

One way to think about the structure of the scheduling code is that it enforces a set of invariants about each process, and holds `p->lock` whenever those invariants are not true. One invariant is that if a process is `RUNNING`, a timer interrupt's `yield` must be able to safely switch away from the process; this means that the CPU registers must hold the process's register values (i.e. `swtch` hasn't moved them to a `context`), and `c->proc` must refer to the process. Another invariant is that if a process is `RUNNABLE`, it must be safe for an idle CPU's `scheduler` to run it; this means that `p->context` must hold the process's registers (i.e., they are not actually in the real registers), that no CPU is executing on the process's kernel stack, and that no CPU's `c->proc` refers to the process. Observe that these properties are often not true while `p->lock` is held.

Maintaining the above invariants is the reason why xv6 often acquires `p->lock` in one thread and releases it in other, for example acquiring in `yield` and releasing in `scheduler`. Once `yield` has started to modify a running process's state to make it `RUNNABLE`, the lock must remain held until the invariants are restored: the earliest correct release point is after `scheduler` (running on its own stack) clears `c->proc`. Similarly, once `scheduler` starts to convert a `RUNNABLE` process to `RUNNING`, the lock cannot be released until the kernel thread is completely running (after the `swtch`, for example in `yield`).

`p->lock` protects other things as well: the interplay between `exit` and `wait`, the machinery to avoid lost wakeups (see Section 7.5), and avoidance of races between a process exiting and other processes reading or writing its state (e.g., the `exit` system call looking at `p->pid` and setting `p->killed` (kernel/proc.c:611)). It might be worth thinking about whether the different functions of `p->lock` could be split up, for clarity and perhaps for performance.

7.4 Code: mycpu and myproc

Xv6 often needs a pointer to the current process's `proc` structure. On a uniprocessor one could have a global variable pointing to the current `proc`. This doesn't work on a multi-core machine, since each core executes a different process. The way to solve this problem is to exploit the fact that each core has its own set of registers; we can use one of those registers to help find per-core information.

Xv6 maintains a `struct cpu` for each CPU (kernel/proc.h:22), which records the process currently running on that CPU (if any), saved registers for the CPU's scheduler thread, and the count of nested spinlocks needed to manage interrupt disabling. The function `mycpu` (kernel/proc.c:60) returns a pointer to the current CPU's `struct cpu`. RISC-V numbers its CPUs, giving each a *hartid*. Xv6 ensures that each CPU's hartid is stored in that CPU's `tp` register while in the kernel. This allows `mycpu` to use `tp` to index an array of `cpu` structures to find the right one.

Ensuring that a CPU's `tp` always holds the CPU's hartid is a little involved. `mstart` sets the `tp` register early in the CPU's boot sequence, while still in machine mode (kernel/start.c:46). `usertrapret` saves `tp` in the trampoline page, because the user process might modify `tp`. Finally,