which checks that the device really has finished sending, and hands the device the next buffered output character. Thus if a process writes multiple bytes to the console, typically the first byte will be sent by `uartputc`'s call to `uartstart`, and the remaining buffered bytes will be sent by `uartstart` calls from `uartintr` as transmit complete interrupts arrive.

A general pattern to note is the decoupling of device activity from process activity via buffering and interrupts. The console driver can process input even when no process is waiting to read it; a subsequent read will see the input. Similarly, processes can send output without having to wait for the device. This decoupling can increase performance by allowing processes to execute concurrently with device I/O, and is particularly important when the device is slow (as with the UART) or needs immediate attention (as with echoing typed characters). This idea is sometimes called *I/O concurrency*.

5.3 Concurrency in drivers

You may have noticed calls to `acquire` in consoleread and in `consoleintr`. These calls acquire a lock, which protects the console driver's data structures from concurrent access. There are three concurrency dangers here: two processes on different CPUs might call `consoleread` at the same time; the hardware might ask a CPU to deliver a console (really UART) interrupt while that CPU is already executing inside `consoleread`; and the hardware might deliver a console interrupt on a different CPU while `consoleread` is executing. Chapter 6 explores how locks help in these scenarios.

Another way in which concurrency requires care in drivers is that one process may be waiting for input from a device, but the interrupt signaling arrival of the input may arrive when a different process (or no process at all) is running. Thus interrupt handlers are not allowed to think about the process or code that they have interrupted. For example, an interrupt handler cannot safely call `copyout` with the current process's page table. Interrupt handlers typically do relatively little work (e.g., just copy the input data to a buffer), and wake up top-half code to do the rest.

5.4 Timer interrupts

Xv6 uses timer interrupts to maintain its clock and to enable it to switch among compute-bound processes; the `yield` calls in `usertrap` and `kerneltrap` cause this switching. Timer interrupts come from clock hardware attached to each RISC-V CPU. Xv6 programs this clock hardware to interrupt each CPU periodically.

RISC-V requires that timer interrupts be taken in machine mode, not supervisor mode. RISC-V machine mode executes without paging, and with a separate set of control registers, so it's not practical to run ordinary xv6 kernel code in machine mode. As a result, xv6 handles timer interrupts completely separately from the trap mechanism laid out above.

Code executed in machine mode in `start.c`, before `main`, sets up to receive timer interrupts (kernel/start.c:57). Part of the job is to program the CLINT hardware (core-local interruptor) to generate an interrupt after a certain delay. Another part is to set up a scratch area, analogous to the

trapframe, to help the timer interrupt handler save registers and the address of the CLINT registers. Finally, `start` sets `mtvec` to `timervec` and enables timer interrupts.

A timer interrupt can occur at any point when user or kernel code is executing; there's no way for the kernel to disable timer interrupts during critical operations. Thus the timer interrupt handler must do its job in a way guaranteed not to disturb interrupted kernel code. The basic strategy is for the handler to ask the RISC-V to raise a "software interrupt" and immediately return. The RISC-V delivers software interrupts to the kernel with the ordinary trap mechanism, and allows the kernel to disable them. The code to handle the software interrupt generated by a timer interrupt can be seen in `devintr` (kernel/trap.c:204).

The machine-mode timer interrupt vector is `timervec` (kernel/kernelvec.S:93). It saves a few registers in the scratch area prepared by `start`, tells the CLINT when to generate the next timer interrupt, asks the RISC-V to raise a software interrupt, restores registers, and returns. There's no C code in the timer interrupt handler.

5.5 Real world

Xv6 allows device and timer interrupts while executing in the kernel, as well as when executing user programs. Timer interrupts force a thread switch (a call to `yield`) from the timer interrupt handler, even when executing in the kernel. The ability to time-slice the CPU fairly among kernel threads is useful if kernel threads sometimes spend a lot of time computing, without returning to user space. However, the need for kernel code to be mindful that it might be suspended (due to a timer interrupt) and later resume on a different CPU is the source of some complexity in xv6. The kernel could be made somewhat simpler if device and timer interrupts only occurred while executing user code.

Supporting all the devices on a typical computer in its full glory is much work, because there are many devices, the devices have many features, and the protocol between device and driver can be complex and poorly documented. In many operating systems, the drivers account for more code than the core kernel.

The UART driver retrieves data a byte at a time by reading the UART control registers; this pattern is called *programmed I/O*, since software is driving the data movement. Programmed I/O is simple, but too slow to be used at high data rates. Devices that need to move lots of data at high speed typically use *direct memory access (DMA)*. DMA device hardware directly writes incoming data to RAM, and reads outgoing data from RAM. Modern disk and network devices use DMA. A driver for a DMA device would prepare data in RAM, and then use a single write to a control register to tell the device to process the prepared data.

Interrupts make sense when a device needs attention at unpredictable times, and not too often. But interrupts have high CPU overhead. Thus high speed devices, such networks and disk controllers, use tricks that reduce the need for interrupts. One trick is to raise a single interrupt for a whole batch of incoming or outgoing requests. Another trick is for the driver to disable interrupts entirely, and to check the device periodically to see if it needs attention. This technique is called *polling*. Polling makes sense if the device performs operations very quickly, but it wastes CPU time if the device is mostly idle. Some drivers dynamically switch between polling and interrupts

depending on the current device load.

The UART driver copies incoming data first to a buffer in the kernel, and then to user space. This makes sense at low data rates, but such a double copy can significantly reduce performance for devices that generate or consume data very quickly. Some operating systems are able to directly move data between user-space buffers and device hardware, often with DMA.

5.6 Exercises

1. Modify `uart.c` to not use interrupts at all. You may need to modify `console.c` as well.

2. Add a driver for an Ethernet card.

Chapter 6

Locking

Most kernels, including xv6, interleave the execution of multiple activities. One source of interleaving is multiprocessor hardware: computers with multiple CPUs executing independently, such as xv6's RISC-V. These multiple CPUs share physical RAM, and xv6 exploits the sharing to maintain data structures that all CPUs read and write. This sharing raises the possibility of one CPU reading a data structure while another CPU is mid-way through updating it, or even multiple CPUs updating the same data simultaneously; without careful design such parallel access is likely to yield incorrect results or a broken data structure. Even on a uniprocessor, the kernel may switch the CPU among a number of threads, causing their execution to be interleaved. Finally, a device interrupt handler that modifies the same data as some interruptible code could damage the data if the interrupt occurs at just the wrong time. The word *concurrency* refers to situations in which multiple instruction streams are interleaved, due to multiprocessor parallelism, thread switching, or interrupts.

Kernels are full of concurrently-accessed data. For example, two CPUs could simultaneously call `kalloc`, thereby concurrently popping from the head of the free list. Kernel designers like to allow for lots of concurrency, since it can yield increased performance though parallelism, and increased responsiveness. However, as a result kernel designers spend a lot of effort convincing themselves of correctness despite such concurrency. There are many ways to arrive at correct code, some easier to reason about than others. Strategies aimed at correctness under concurrency, and abstractions that support them, are called *concurrency control* techniques.

Xv6 uses a number of concurrency control techniques, depending on the situation; many more are possible. This chapter focuses on a widely used technique: the *lock*. A lock provides mutual exclusion, ensuring that only one CPU at a time can hold the lock. If the programmer associates a lock with each shared data item, and the code always holds the associated lock when using an item, then the item will be used by only one CPU at a time. In this situation, we say that the lock protects the data item. Although locks are an easy-to-understand concurrency control mechanism, the downside of locks is that they can kill performance, because they serialize concurrent operations.

The rest of this chapter explains why xv6 needs locks, how xv6 implements them, and how it uses them.

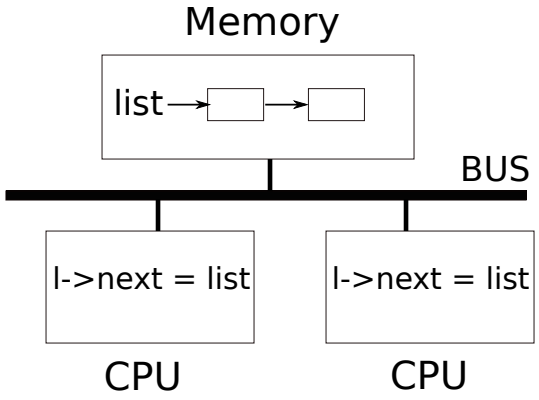

Figure 6.1: Simplified SMP architecture

6.1 Race conditions

As an example of why we need locks, consider two processes calling `wait` on two different CPUs. `wait` frees the child's memory. Thus on each CPU, the kernel will call `kfree` to free the children's pages. The kernel allocator maintains a linked list: `kalloc()` (kernel/kalloc.c:69) pops a page of memory from a list of free pages, and `kfree()` (kernel/kalloc.c:47) pushes a page onto the free list. For best performance, we might hope that the `kfree`s of the two parent processes would execute in parallel without either having to wait for the other, but this would not be correct given xv6's `kfree` implementation.

Figure 6.1 illustrates the setting in more detail: the linked list is in memory that is shared by the two CPUs, which manipulate the linked list using load and store instructions. (In reality, the processors have caches, but conceptually multiprocessor systems behave as if there were a single, shared memory.) If there were no concurrent requests, you might implement a list `push` operation as follows:

```
1     struct element {
2       int data;
3       struct element *next;
4     };
5
6     struct element *list = 0;
7
8     void
9     push(int data)
10    {
11      struct element *l;
12
13      l = malloc(sizeof *l);
14      l->data = data;
15      l->next = list;
16      list = l;
```

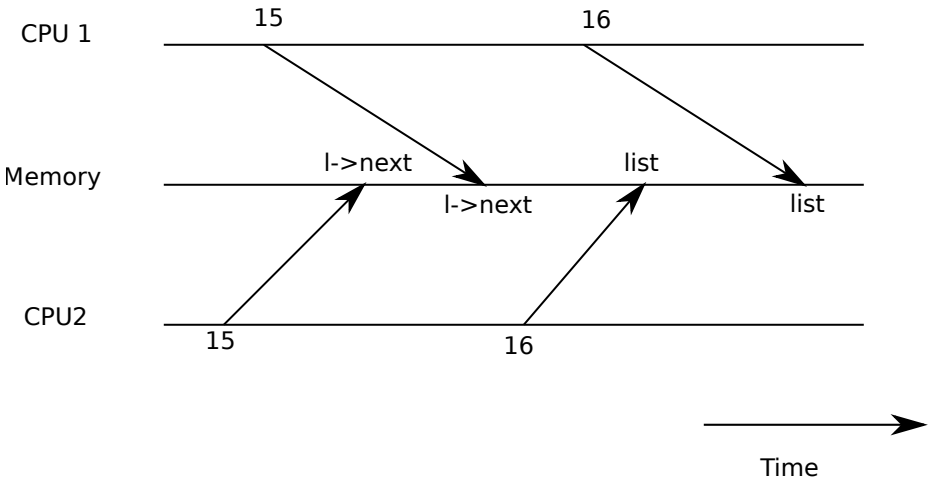

Figure 6.2: Example race

```
17      }
```

This implementation is correct if executed in isolation. However, the code is not correct if more than one copy executes concurrently. If two CPUs execute `push` at the same time, both might execute line 15 as shown in Fig 6.1, before either executes line 16, which results in an incorrect outcome as illustrated by Figure 6.2. There would then be two list elements with `next` set to the former value of `list`. When the two assignments to `list` happen at line 16, the second one will overwrite the first; the element involved in the first assignment will be lost.

The lost update at line 16 is an example of a *race condition*. A race condition is a situation in which a memory location is accessed concurrently, and at least one access is a write. A race is often a sign of a bug, either a lost update (if the accesses are writes) or a read of an incompletely-updated data structure. The outcome of a race depends on the exact timing of the two CPUs involved and how their memory operations are ordered by the memory system, which can make race-induced errors difficult to reproduce and debug. For example, adding print statements while debugging `push` might change the timing of the execution enough to make the race disappear.

The usual way to avoid races is to use a lock. Locks ensure *mutual exclusion*, so that only one CPU at a time can execute the sensitive lines of `push`; this makes the scenario above impossible. The correctly locked version of the above code adds just a few lines (highlighted in yellow):

```
6     struct element *list = 0;
7     struct lock listlock;
8
9     void
10    push(int data)
11    {
12      struct element *l;
13      l = malloc(sizeof *l);
14      l->data = data;
15
```

```
16      acquire(&listlock);
17      l->next = list;
18      list = l;
19      release(&listlock);
20    }
```

The sequence of instructions between `acquire` and `release` is often called a *critical section*. The lock is typically said to be protecting `list`.

When we say that a lock protects data, we really mean that the lock protects some collection of invariants that apply to the data. Invariants are properties of data structures that are maintained across operations. Typically, an operation's correct behavior depends on the invariants being true when the operation begins. The operation may temporarily violate the invariants but must reestablish them before finishing. For example, in the linked list case, the invariant is that `list` points at the first element in the list and that each element's `next` field points at the next element. The implementation of `push` violates this invariant temporarily: in line 17, `l` points to the next list element, but `list` does not point at `l` yet (reestablished at line 18). The race condition we examined above happened because a second CPU executed code that depended on the list invariants while they were (temporarily) violated. Proper use of a lock ensures that only one CPU at a time can operate on the data structure in the critical section, so that no CPU will execute a data structure operation when the data structure's invariants do not hold.

You can think of a lock as *serializing* concurrent critical sections so that they run one at a time, and thus preserve invariants (assuming the critical sections are correct in isolation). You can also think of critical sections guarded by the same lock as being atomic with respect to each other, so that each sees only the complete set of changes from earlier critical sections, and never sees partially-completed updates.

Although correct use of locks can make incorrect code correct, locks limit performance. For example, if two processes call `kfree` concurrently, the locks will serialize the two calls, and we obtain no benefit from running them on different CPUs. We say that multiple processes *conflict* if they want the same lock at the same time, or that the lock experiences *contention*. A major challenge in kernel design is to avoid lock contention. Xv6 does little of that, but sophisticated kernels organize data structures and algorithms specifically to avoid lock contention. In the list example, a kernel may maintain a free list per CPU and only touch another CPU's free list if the CPU's list is empty and it must steal memory from another CPU. Other use cases may require more complicated designs.

The placement of locks is also important for performance. For example, it would be correct to move `acquire` earlier in `push`: it is fine to move the call to `acquire` up to before line 13. This may reduce performance because then the calls to `malloc` are also serialized. The section "Using locks" below provides some guidelines for where to insert `acquire` and `release` invocations.

6.2 Code: Locks

Xv6 has two types of locks: spinlocks and sleep-locks. We'll start with spinlocks. Xv6 represents a spinlock as a `struct spinlock` (kernel/spinlock.h:2). The important field in the structure is

`locked`, a word that is zero when the lock is available and non-zero when it is held. Logically, xv6 should acquire a lock by executing code like

```
21    void
22    acquire(struct spinlock *lk) // does not work!
23    {
24      for(;;) {
25        if(lk->locked == 0) {
26          lk->locked = 1;
27          break;
28        }
29      }
30    }
```

Unfortunately, this implementation does not guarantee mutual exclusion on a multiprocessor. It could happen that two CPUs simultaneously reach line 25, see that `lk->locked` is zero, and then both grab the lock by executing line 26. At this point, two different CPUs hold the lock, which violates the mutual exclusion property. What we need is a way to make lines 25 and 26 execute as an *atomic* (i.e., indivisible) step.

Because locks are widely used, multi-core processors usually provide instructions that implement an atomic version of lines 25 and 26. On the RISC-V this instruction is `amoswap r, a`. `amoswap` reads the value at the memory address `a`, writes the contents of register `r` to that address, and puts the value it read into `r`. That is, it swaps the contents of the register and the memory address. It performs this sequence atomically, using special hardware to prevent any other CPU from using the memory address between the read and the write.

Xv6's `acquire` (kernel/spinlock.c:22) uses the portable C library call `__sync_lock_test_and_set`, which boils down to the `amoswap` instruction; the return value is the old (swapped) contents of `lk->locked`. The `acquire` function wraps the swap in a loop, retrying (spinning) until it has acquired the lock. Each iteration swaps one into `lk->locked` and checks the previous value; if the previous value is zero, then we've acquired the lock, and the swap will have set `lk->locked` to one. If the previous value is one, then some other CPU holds the lock, and the fact that we atomically swapped one into `lk->locked` didn't change its value.

Once the lock is acquired, `acquire` records, for debugging, the CPU that acquired the lock. The `lk->cpu` field is protected by the lock and must only be changed while holding the lock.

The function `release` (kernel/spinlock.c:47) is the opposite of `acquire`: it clears the `lk->cpu` field and then releases the lock. Conceptually, the release just requires assigning zero to `lk->locked`. The C standard allows compilers to implement an assignment with multiple store instructions, so a C assignment might be non-atomic with respect to concurrent code. Instead, `release` uses the C library function `__sync_lock_release` that performs an atomic assignment. This function also boils down to a RISC-V `amoswap` instruction.

6.3 Code: Using locks

Xv6 uses locks in many places to avoid race conditions. As described above, `kalloc` (kernel/kalloc.c:69) and `kfree` (kernel/kalloc.c:47) form a good example. Try Exercises 1 and 2 to see what happens if those functions omit the locks. You'll likely find that it's difficult to trigger incorrect behavior, suggesting that it's hard to reliably test whether code is free from locking errors and races. It is not unlikely that xv6 has some races.

A hard part about using locks is deciding how many locks to use and which data and invariants each lock should protect. There are a few basic principles. First, any time a variable can be written by one CPU at the same time that another CPU can read or write it, a lock should be used to keep the two operations from overlapping. Second, remember that locks protect invariants: if an invariant involves multiple memory locations, typically all of them need to be protected by a single lock to ensure the invariant is maintained.

The rules above say when locks are necessary but say nothing about when locks are unnecessary, and it is important for efficiency not to lock too much, because locks reduce parallelism. If parallelism isn't important, then one could arrange to have only a single thread and not worry about locks. A simple kernel can do this on a multiprocessor by having a single lock that must be acquired on entering the kernel and released on exiting the kernel (though system calls such as pipe reads or `wait` would pose a problem). Many uniprocessor operating systems have been converted to run on multiprocessors using this approach, sometimes called a "big kernel lock," but the approach sacrifices parallelism: only one CPU can execute in the kernel at a time. If the kernel does any heavy computation, it would be more efficient to use a larger set of more fine-grained locks, so that the kernel could execute on multiple CPUs simultaneously.

As an example of coarse-grained locking, xv6's `kalloc.c` allocator has a single free list protected by a single lock. If multiple processes on different CPUs try to allocate pages at the same time, each will have to wait for its turn by spinning in `acquire`. Spinning reduces performance, since it's not useful work. If contention for the lock wasted a significant fraction of CPU time, perhaps performance could be improved by changing the allocator design to have multiple free lists, each with its own lock, to allow truly parallel allocation.

As an example of fine-grained locking, xv6 has a separate lock for each file, so that processes that manipulate different files can often proceed without waiting for each other's locks. The file locking scheme could be made even more fine-grained if one wanted to allow processes to simultaneously write different areas of the same file. Ultimately lock granularity decisions need to be driven by performance measurements as well as complexity considerations.

As subsequent chapters explain each part of xv6, they will mention examples of xv6's use of locks to deal with concurrency. As a preview, Figure 6.3 lists all of the locks in xv6.

6.4 Deadlock and lock ordering

If a code path through the kernel must hold several locks at the same time, it is important that all code paths acquire those locks in the same order. If they don't, there is a risk of *deadlock*. Let's say two code paths in xv6 need locks A and B, but code path 1 acquires locks in the order A then