Critical Section Protection (Locks, Continued)

Computer Science 432 — Lecture 12 — Duane Bailey

April 4, 2022



Announcements

- Today, in lab, review of Lab 5 solution
- Small Group Meetings, Microscopy Lab, Hopper Science, Ground Floor. Optional, but: why not?
- *
- Updated syllabus, on-line. •

O/S Conference topic contracts available. Return by Wednesday, April 6. Happy to discuss.



Critical Sections

- There is concurrency in xv6. For example, we have three processors.
- Sometimes you're interested in *exclusively* executing sections of code that are *critical*. Examples:
 - Modifying the free list in the kernel page allocator.
- Modifying the struct proc structure in proc.c One approach to managing access to critical sections is the use of *locks*.
 - A race condition.
 - Deadlock.
 - Livelock.
 - Starvation.
- Today, we'll look at lock design.

Failure to appropriately manage access to critical sections could lead to:



Sharing Memory Between Processors

- The memory in xv6 is shared among all cpu's/processors:
 - Every processor can see all of memory.
 - Every processor can execute the kernel.
 - Processes can freely move between processors, over time.
- RISC-V has a number of instructions that support *atomic access* to memory: Atomic operators, "AMOs".
 - Load-reserved, store-conditional operators, "LR/SC". *



Atomicity Operators

By definition, the atomicity operators:

- the same as executing the AMO before or after the other instructions.
- In short: the AMOs act like operations that happen directly in memory.
- Example: amoswap rd, rs1, rs2 (Similar to: csrrw rd, rs1, rs2, but involving memory.) rd is assigned value at memory pointed to bey rs1. memory at rs1 gets value in rs2.
- AMO guarantees require consistency protocols in the cache. ✤ L1\$ and L2\$ are dedicated to the CPU.

Are complex. You might attempt to do the same things with 2 or more instructions. Access memory. Since internal state of the CPU is not shared, the only concern is memory. Guaranteed atomicity. Concurrent instructions that involve AMOs guarantee the outcome is

* L3\$ is shared. This level of cache is responsible for managing atomic access to memory.





Load-Reserved/Store-Conditional

- The load-reserved (LR) and store-conditional (SC) operations work together:

 - SC stores a value to a memory location, but only if there is a reservation there.
 - Regular loads and stores clear the reservation at their addresses.

LR loads a value from memory location and places a "reservation" on that address



The Atomicity Axiom

Volume I: RISC-V Unprivileged ISA V20191213

A.3.3 Atomicity Axiom

Atomicity Axiom (for Aligned Atomics): If r and w are paired load and store operations generated by aligned LR and SC instructions in a hart h, s is a store to byte x, and r returns a value written by s, then s must precede w in the global memory order, and there can be no store from a hart other than h to byte x following s and preceding w in the global memory order.

The Progress Axiom

A.3.4 Progress Axiom

Progress Axiom: No memory operation may be preceded in the global memory order by an infinite sequence of other memory operations.



Implementing Compare-and-Swap with LR/SC

- An example atomic operation, built with LR/SC.
- compareAndSwap(a0,a1,a2) checks to see if *a0 == a1 and, if so, sets *a0 = a2 and returns 0 (success). Otherwise, returns 1.

```
cas:
    lr.w t0,
    bne t0, a1
    sc.w t0, a
    bnez t0, d
    li a0, 0
    jr ra
fail:
    li a0, 1
    jr ra
```

- # a0 holds address of memory location
- # a1 holds expected value
- # a2 holds desired value
- # a0 holds return value, 0 if successful, !0 otherwise

(a0)	#	Load original value.
1, fail	#	Doesn't match, so fail.
a2, (a0)	#	Try to update.
cas	#	Retry if store-conditional failed.
	#	Set return to success.
	#	Return.
	#	Set return to failure.

Return.

Figure 8.1: Sample code for compare-and-swap function using LR/SC.





Managing Critical Sections

- Suppose we have a number of *processes*, P(i), each sharing a critical section of code that must be exclusively executed by at most one process.
- Three properties are required for successful sharing of critical code:

 - on the number of processes that enter before P(i).

```
P(i):
for (;;) {
   { entry code}
   critical section
   { exit code }
   outside code
```

Mutual Exclusion. If process P(i) is in the critical section, no other process can be there too.

* **Progress.** If no process is in the critical section, and some wish to, those trying to enter must be responsible for determining who does, and it cannot be postponed indefinately.

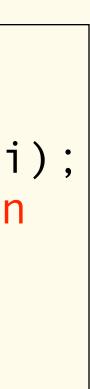
Bounded Waiting. If P(i) makes a request to enter a critical section, there must be bound



Two Process Approach #1.

- Suppose we have two processes (P(0) and P(1)). Assume that *i* is 0 or 1 and that j = !i.
- Keep a variable, turn, that determines which process should enter. What happens?
 - There is mutual exclusion.
 - We have **bounded waiting**.
 - But there is no guarantee of **progress**. * If turn = 0, but only P(1) wants to enter, it must wait until P(0) lets it, if ever.

```
P(i):
for (;;) {
   while (turn != i);
   critical section
   turn = j;
   outside code
```





Two Process Approach #2.

- Suppose we have two processes (P(0) and P(1)). Assume that *i* is 0 or 1 and that j = !i.
- Keep an array, flag[2], that keeps track of which process is in the critical section.
 - We fail to achieve mutual exclusion. Time 0: P(0) finds flag[1] is 0. P(1) finds flag[0] is 0. Time 1: P(0) sets flag[0] = 1 and P(1) sets flag[1] to 1. Time 2: Both processes enter the critical section.

 - We might have bounded waiting. It depends on timing.

Observation: The approach depends on perfect timing to keep critical section protected.

```
P(i):
for (;;) {
   while (flag[j]);
   flag[i] = 1;
   critical section
   flag[i] = 0;
   outside code
```

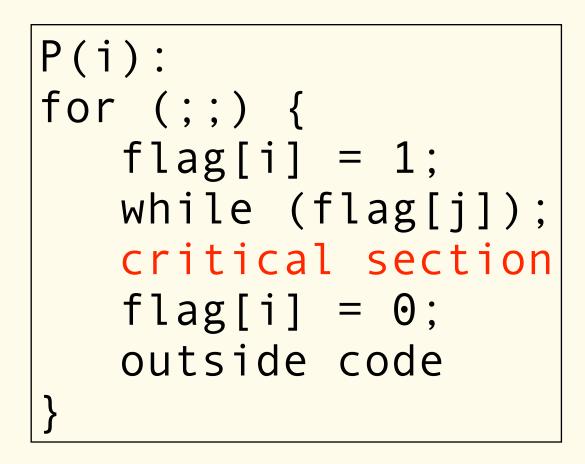
• We have **progress**. No process has to wait for decisions by another process in outside code.



Two Process Approach #3.

- Suppose we have two processes (P(0) and P(1)). Assume that *i* is 0 or 1 and that j = !i.
- Keep an array, flag[2], that keeps track of which process *wants to be in* the critical section.
 - Here, we have mutual exclusion.
 - We have **bounded waiting**.
 - But there is no guarantee of progress.
- *





Both processes could *want* to enter the critical section at the same time. Loops infinitely.

Observation: An example of *deadlock*. No process moves forward without drastic intervention.





Two Process Approach #4.

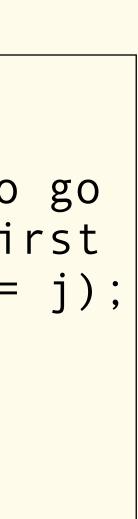
- ✤ Suppose we have two processes (P(0) and P(1)). Assume that *i* is 0 or 1 and that j = !i.
- Keep an array, flag[2], that keeps track of which process *wants to be in* the critical section. We also keep track of turn. Here, we have mutual exclusion.

Notice the only thing holding back a process is the loop.

- We have progress. Suppose P(i) is waiting for P(j) in c.s. As soon as process j leaves, i will enter.
- We have **bounded waiting**. P(i) could only held back once. After that, it takes a turn.

Observation: This is a working solution, but is not obvious. Protecting critical sections is hard. *

P(i): for (;;) { flag[i] = 1; // I want to go turn = j; // you go first while (flag[j] && turn == j); critical section flag[i] = 0;outside code





More than two processes.

The critical section problem for more than two processes is a bit harder: ✤ From the point-of-view of process P(i), there are several other processes, j. The flag variable keeps track of several states: *idle, want-in,* and *in-cs.* *

- ✤ To enter:
 - Typically, flag[i] == idle.
 - When a process tries to enter, it sets its own flag to want-in.
 - Eventually, j == i.
 - ✤ It sets flag[i] = *in-cs*.
 - ✤ It checks that it is the only process with flag[i] == in-cs.
- ✤ To leave:
 - We set turn to the next non-idle process, or turn+1. We set flag[i] = idle.

This approach achieves mutual exclusion, progress, and bounded waiting.

Starting at P(turn), it cycles around, it searches for first process j with flag[j] != idle.

If flag[turn] == idle or turn == i, we set turn = i and enter. Otherwise, try again.



Solution #1 using Hardware.

Suppose we have an *atomic* instruction int testAndSet(int* target) that... Temporarily saves the value stored at target. Sets the target memory value to 1. Returns the saved, prior value of the target memory location. Because it's atomic, no other instruction has access to target during this read&write operation.

Now the critical section problem is pretty simple. Declare a global integer, lock, initially 0.

```
P(i):
for (;;) {
   while (testAndSet(lock));
   critical section
   lock = 0;
   outside code
```



Solution #2 using Hardware.

- Suppose we have an *atomic* instruction void swap(int *a, int *b) that... Sets a temporary to the value at location a. Sets the location a to the value at location b. Sets the location b to the value in temporary. Again, because it's atomic, no other instruction has access to target during this swapping operation.
- Now the critical section problem is easy: Declare a global integer, lock, initially 0, and a local integer, key.

```
P(i):
for (;;) {
   key = 1;
   do {
     swap(lock, key);
   } while (key == 1);
   critical section
   lock = 0;
   outside code
```



Solution #3 using Hardware.

- Compares *a and b. It returns 1 (failure) if they differ. Otherwise, it sets *a, to c.
- We built this so it's atomic, no other instruction has access to target during this swapping operation.
- Now the critical section problem is easy: Declare a global integer, lock, initially 0, and a local integer, key.

```
Suppose we have an atomic instruction int compareAndSwap(int *a, int b, int c) that...
```

```
P(i):
for (;;) {
   while (compareAndSwap(lock,0,1));
   critical section
   lock = 0;
   outside code
```

