6.033 Lec 4
2/18/2014

Bounded Buffer

Recap:

Last time saw how we can isolate two running programs on the same computer from each other by creating a *virtual memory* abstraction, where two processes cannot reference each other's memory.

(Each program running on a separate CPU).

This is achieved via an MMU, that translates *virtual addresses* into *physical addresses* by performing lookups in a *page table.*

Saw how the *kernel* acts as a mediator: it is responsible for handling references to invalid address, and for setting up the page tables for running programs.

The *kernel* also provides system calls that allow programs to access various operating systems abstractions, e.g., the file system instead of raw disk.

System calls and *invalid page exceptions* are both examples of interrupts: calls from the program into the OS.

(Show diagram)

In this class, going to see how two programs can communicate.

Several different ways we might implement this via a new operating system abstraction called a *bounded buffer*.

Programs never interact directly with each other, but just with OS via *send* and *receive* calls. Diagram:

For now, focus on one way calling (could implement two way with two of these).

Note that this is a way to provide an RPC mechanism between the two processes, like we learned about in the first lecture!

Aside: the kernel doesn't strictly have to pass messages. A different
design would be to arrange the page tables of two processes to share
one common page of memory. The two processes can then implement message
passing via this shared page, as we will describe below.

Abstraction: buffer storing N messages

API:

send(m): add m to the buffer
m <— receive() : get m from the buffer

If no messages, receiver may need to wait, or *block*
Buffer is finite so send may also need to block for space to become free

Why do we need more than one entry?
Help deal with bursty senders and receivers
E.g., if lots of messages arrive, sender and queue them up

Note that this is a lot like the unix abstraction of *pipes*, which you'll learn more about in the hands on and in recitation Thursday.

What is tricky about designing a data structure that implements this abstraction?
Concurrency!

Suppose our two communicating programs both try to access buffer at the same time.
send() and receive() are interacting with some shared structure
(show diagram)

If we aren't careful, we might get some unexpected result, e.g., reading a message that has already been sent, or that has never been written, or….

Concurrency is a major source of complexity in systems, and one we will return repeatedly in this class.

Ok, so lets see how we can design a buffer that correctly implements our API.

Allocate a *bounded buffer* of fixed size,
e.g., buf(5) stores 5 messages
two variables: in : total messages written to buffer
out: total messages read from buffer
When is it OK to write:
when (in - out) < 5 (less than 5 pending messages)
When is it OK to read:
when out < in (more than 0 pending messages)
Where should we write the next message to:
can't just write to *out,* since out may be much larger that buffer size
what we want to do is to cause next write to wrap around after we have written 5th slot
use a modulus
out mod 5 = next slot to write

Same logic works for in:
in mod 5 = next slot to read

Example:
in = 28, out = 26 ==> two messages waiting
28 mod 5 = 3, so two messages are in slots 1 and 2

Ok, so lets look at some code:

[slide: send()]

loop waits for space to send ("busy-wait")
perform write before increment

bb is just a pointer to an instance, so can have many of them

[slide: receive()]
busy wait for message to read
perform read before increment

<u>Does this work when running concurrently on two CPUs?</u>
(yes — but this is surprising — usually we will need to do more to achieve concurrency)

<u>Also, not as easy as it looked — suppose we swap increment and buffer write — does that work?</u>

No! reader might sneak in and read slot before we have written to it!

Must increment in/out before access (for both reader and writer)

Ok, so is this all there is to designing a bounded buffer?

Some limitations:

1) Only 1 sender and receiver
2) Each sender/receiver has its own CPU (busy wait prevents anyone else from running!)

How to fix:

1) Rest of today's lecture
2) Tomorrow's lecture

Multiple senders:

suppose we want:

A: send(bb, m1)
B: send(bb, m2)

Concurrently

What is the expected behavior?

Possible spec: m1 and m2 both in buffer, don't care about their order

Suppose they both run at the same time:

(show slide)

Outcome: in = 1, buf[0] = m2, m1 was lost

This is a "race", since both are running through the code at the same time, and one of them will "win"

Other races: with just one slot, A and B can both think there's space and overwrite the oldest entry.

Races are tricky: we had convinced ourself code on slide was right, but didn't imagine case with two concurrent writers

Hard to find: won't always manifest itself because it is timing dependent
E.g., Therac-25 bug

Why did this work with concurrent sender and receiver? Only one CPU manipulating in / out at a time

Let's see if we can achieve the same with only multiple senders?

idea: add "locks" that only allow one CPU to be in a piece of code at a time

API: acquire(l)
release(l)

Lock can be "locked" or "unlocked"

Acquire: (show slide)
Release (show slide)

If 2 cpus try to acquire a lock at the same time, one will succeed, the other will block

Now let's see how we can add locks to send()

(show slide)

Associate a lock with each bb
Acquire before access
Release after access
Only one send() executing at a time
If code was correct with one send(), this one should be too!

Note that you have to get locking right — e.g., (show slide)
What happens if we acquire after the if statement?

Concurrent senders might both think they can write!

[demo: lockdemo]
Two side-by-side terminals

vi lockdemo.c on the left, initially with acquire/release.
shell on the right.

Code has one BB (array) and two CPUs, each appending three messages.
Every second, we will run the two CPUs and print out contents of BB.
Array contents reset between each run

Q: what will array[] hold?
A: make lockdemo && ./lockdemo
various correct answers, depends on interleaving.

Q: what will array[] hold if we don't have locks?
A: comment out acquire() and release().
make lockdemo && ./lockdemo
sometimes correct output: races are often hard to reproduce in practice!
too few elements: both wrote same array[n], both set n to new value.
zero elements: both wrote same array[n], then increment in order.

Q: what will array[] hold if we put locks around each statement in append()?
A: put acquire() and release() around each of the two statements.
make lockdemo && ./lockdemo
always the right number of elements (increments happen in order).
sometimes zero slots when both write to the same array[n].

Why is locking each line of code not enough?
Refer back to the instruction sequences for two concurrent send()s.

Locks create *atomic actions*, sequences of code that are executed in their entirety, one at a time

Grouping increment's instructions ensures increment happens correctly.
Not enough to ensure send() will use the correct slot in bb.

How big should an atomic action big? When is it big enough?

Often helpful to think of locks as protecting invariants.
Invariant for bb: refer back to data structure definition.
Can release lock when invariant holds: other CPUs can see state.
Not OK to release lock when invariant is broken -- must fix it up first!
E.g., in the middle of send(), already put msg, but haven't updated in.
Lock prevents other CPUs from seeing an inconsistent state.

More complex example: file system.
[slide: file system without concurrency]
How do we make it safe for multiple CPUs to call move()?

Locking design #1: one lock ("coarse-grained" locking).
[slide: coarse-grained locking]
+: Likely correct, just like on a single CPU.
-: No concurrency inside FS, even if accessing different directories.
Can lead to low performance on many CPUs.

Locking design #2: per-directory locks ("fine-grained" locking).
[slide: fine-grained locking]
+: More concurrency, can lead to higher performance.
-: Hard to get correct! Best to start coarse-grained, refine as needed.

Problem: in the middle of move(), file doesn't appear in either directory.
[slide: file not in dir1 or dir2]
Invariant involves both dir1 and dir2, so need link+unlink to be atomic.

Locking design #3: holding both locks together.
[slide: holding two locks]
+: Solves the atomicity problem.
-: Deadlock.
[slide: deadlock]

Locking design #4: solving deadlock.
Plan: look for all places where multiple locks are held.
Ensure that in each case, locks are acquired in the same order.
No locking cycles => no deadlock.
[slide: solving deadlock]
+: Finally might work.
-: Painful, because it requires global reasoning about all locks.
E.g., can you call printf()? Does it acquire some lock?
Good news: deadlocks are not subtle once they occur!
Program stops exactly at the place where deadlock occurred.

How to implement acquire() and release()?

```
acquire(l):     ## strawman design
  while l != 0:
    do nothing
  l <- 1

release(l):
  l <- 0
```

Race condition similar to send(), with two CPUs:
Both CPUs run acquire(l).
Both see l==0.
Both set l<-1.
Both start executing code after the acquire().
Checking l!=0 and setting l=1 should be atomic!
Chicken-and-egg problem?

Hardware support: atomic instruction that combines both operations!
Luckily, most processors provide a solution (often many possible solutions).
One such atomic instruction on x86:

```
  XCHG reg, addr   ## atomic exchange
    temp <- Mem[addr]
    Mem[addr] <- reg
    reg <- temp
```

If XCHG is atomic, we can implement acquire.

```
acquire(l):
  do:
    r <- 1
    XCHG r, l
  while r==1
```

(if l is 0, after XCHG, r will be 0, and loop will terminate
otherwise, someone else is holding l, and need to keep trying
when we succeed we will have atomically installed 1 into l, so others
will block)

How does hardware make XCHG atomic?

[Recent x86 processors, simplified.]

There's a controller responsible for managing access to memory.

Can partition the overall system memory across multiple controllers.

Requirement: each memory location is the responsibility of one controller.

To perform an atomic operation, a CPU must obtain permission from controller.

Controller ensures only one processor has permission at a time.

Processors request permission by sending messages to controller.

What if two messages arrive simultaneously at the controller on two links?

Message router services incoming messages sequentially.

One CPU will lose, and will have to retry.

(Show concl)