2013-05-19 CS34



CS 134:
Operating Systems
Locks and Low-Level Synchronization

Overview

Overview

Beyond Looking

Low-Level Synchronization

Non-Blooking Synchronization

Availing Looks

Beyond Locking

Low-Level Synchronization

Non-Blocking Synchronization

Avoiding Locks

Message-Based Interprocess Communication

An alternative to communication via shared memory + locks.

- Analogous to sending message by mail, or package by sea
- Provides virtual communications medium
- ► Requires two basic operations:
 - ▶ send message(destination, message)
 - receive_message(sender, message)

Class Exercise

send_message and receive_message seem vaguely defined

- What details are missing?
- What are the options?



Some missing things:

- How to deal with process that has multiple messages waiting?
- Is there a way to receive all mail at once?
- Who passes messages?
- Are messages picked up like mail or interrupting like phone calls?
- Should we store messages? How many? What to do when we can't deliver? Wait or discard?
- What can be in a message? Bits? FDs? Memory pages?
- Does receiver need to know sender?
- How reliable is the mail?
- Does a receiver know who the sender is?
- Is there a permissions system?

Some options:

- We can have queues of messages, priority queues, stacks, etc.
- We can store no messages, only 1 message, maybe n messages.

Beyond Locking

Messaging—Design Questions

Questions include:

- ▶ Is a "connection" set up between the two processes?
 - ▶ If so, is the link unidirectional or bidirectional?
- ▶ How do processes find the "addresses" of their friends?
- ► Can many processes send to the same destination?
- ▶ Does the sender wait until the receiver receives the message?
- Does the receiver always know who sent the message?
- Can the receiver restrict who can talk to it?
- Is the capacity of the receiver's mailbox fixed? (and if so, what are the limits?)
- Can messages be lost?
- Can messages vary in size or is the size fixed?
- Do messages contain typed data?
- Is the recipient guaranteed to be on the same machine?



Example: Unix-Domain Sockets with UDP

Sockets call message sources and destinations "ports"

- ► Textual address (actually a valid filename!)
- Numeric port number

Other properties:

- ▶ Is a "connection" set up between the two processes?
 - No ("connectionless datagrams")
- Can a process have more than one port open/listening?
 - Yes
- How do processes find the addresses of their friends?
 - Prior knowledge (well-known ports)
 - ► Port inheritance from parent process



Beyond Locking

Example: Unix-Domain Sockets with UDP

Properties (continued):

- Can many processes send to the same destination?
 - Yes—Messages arrive in unspecified order
- Can many processes receive at the same destination?
 - No
- Does the sender wait until the receiver receives the message?
 - No if mailbox has space for message
 - Yes if mailbox is full.
- Does the receiver always know who sent the message?
 - Usually
- Can the receiver restrict who can talk to it?
 - Only by receiving messages and discarding undesirable ones.



Beyond Locking

Example: Unix-Domain Sockets with UDP

Properties (continued):

- What is the capacity of the receiver's mailbox?
 - Approximately 32 KB of data.
- Do messages arrive in order?
 - Messages from the same sender arrive in order.
 - Messages from different senders might not be temporally ordered
- Can messages be lost?
 - Not under OS X, BSD, Linux or Solaris.
- Can messages vary in size or is the size fixed?
 - Yes, size can vary, up to a limit.
- Do messages contain typed data?
 - Usually no, just bytes
 - But can send open file descriptors!!



On macasanae arrive in nerter?

Can messages be lost? Can messages very in size or is the size fixed:

Example: Unix-Domain Sockets with UDP

Properties (continued):

- What happens if the receiver dies?
 - Messages already delivered to the receiver's mailbox will be (silently) lost.
 - ► Future delivery attempts fail with an error.
- ▶ Is the recipient guaranteed to be on the same machine?
 - Yes.



Unix-Domain UDP Sockets—Class Exercise



Could you implement locks using messaging?

Unix-Domain UDP Sockets—Class Exercise

Could you implement messaging where sender waits for reception?

Could you implement messaging that allows multiple receivers?



Beyond Locking

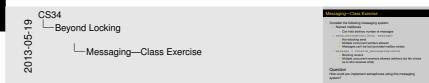
Messaging—Class Exercise

Consider the following messaging system:

- Named mailboxes
 - Can hold arbitrary number of messages
- ▶ send message (mailbox, message)
 - Non-blocking send
 - Multiple concurrent senders allowed
 - Messages can't be lost (provided mailbox exists)
- message = receive_message(mailbox)
 - Blocking receive
 - Multiple concurrent receivers allowed (arbitrary but fair choice as to who receives what)

Question

How could you implement semaphores using this messaging system?



Atomic Synchronization Instructions

Modern processors often provide help with synchronization issues.

- Atomic—Provide a read-op-write cycle.
- Simple—just protecting access to one memory word



Test & Set

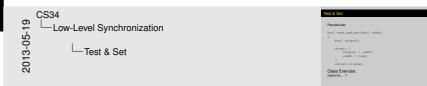
Pseudocode:

```
bool test_and_set(bool *addr)
{
    bool origval;

atomic {
        origval = *addr;
        *addr = true;
}
    return origval;
```

Class Exercise:

Useful for...?



Have them write a spin lock & then show how busy-waiting is bad.

Swap

Pseudocode:

```
int swap(int *addr, int newval)
{
    int orgival;

    atomic {
        origval = *addr;
        *addr = newval;
    }
    return origval;
}
```

Class Exercise:

Useful for...?

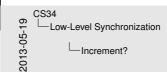
Can you write increment?

Limitations...?



Increment?

```
Try:
void atomic_add(int *i, int delta)
    int v = *i;
                                       // Line 1
    for (;;) {
        int w = swap(*i, v + delta); // Line 2
        if (w == v)
                                       // Line 3
            break;
                                       // Line 4
        v = w;
```





The problem here is that we are assuming that what we get from w is the most recently incremented value from another process, so we can add delta to that "most recent" value and have a correct new value. But consider the following sequence:

- 1. A reads v_1 in line 1
- 2. B increments v to $v_1 + 1$
- 3. A swaps in line 2, seeing & setting $v_1 + 1$
- 4. B increments v to $v_1 + 2$
- 5. B increments v to $v_1 + 3$
- 6. A assigns in line 4, setting $v_2 = v_1 + 1$
- 7. A swaps in line 2, setting v to $v_1 + 2$
- 8. A will now set v to $v_1 + 3$, which is wrong!

The Fundamental Problem?

Class Exercise:

Identify the fundamental problem that prevents us from writing atomic_add correctly.



The difficulty is that we're replacing $\star i$ with v even if $\star i$ has changed in the meantime. We need a way to say "replace $\star i$ only if it still has the value I think it has." As a bonus, it would be good to (a) know whether the value changed, and (b) know what the old value was.

Compare & Swap

```
Pseudocode:
```

```
int compare_and_swap(int *addr, int expectedval,
 int newval)
    int origval;
    atomic {
        origval = *addr;
        if (origval == expectedval)
             *addr = newval;
    return origval;
```

Class Exercise:

Useful for...?

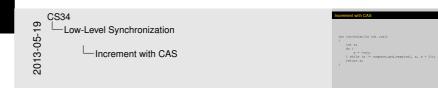
Can you write increment? Limitations...?

CS34
Low-Level Synchronization
Compare & Swap



Increment with CAS

```
int inc(volatile int *val)
{
    int x;
    do {
        x = *val;
    } while (x != compare_and_swap(val, x, x + 1));
    return x;
}
```



Ordinary Stack Code (Unsynchronized)

```
void push(item value)
                             bool trypop(item *valueptr)
  struct stacknode *newnode;
                               item value;
                               struct stacknode *oldtop;
  newnode = malloc(...);
                               if (top == NULL)
  newnode->value = value;
                                 return false;
  newnode->next = top;
                               oldtop = top;
  top = newnode;
                               top = top->next;
                               *valueptr = oldtop->value;
                               free (oldtop);
                               return true;
```



Lots of problems here. If two people push, a node will be lost. If two pop, they might both get the same value (and double-free).

Non-Blocking Stack Code

```
void push(item value)
                             bool trypop(item *valueptr)
  struct stacknode *newnode;
                               item value;
  struct stacknode *oldtop;
                               struct stacknode *oldtop;
                               struct stacknode *newtop;
  newnode = malloc(...);
                               do
  newnode->value = value;
                                 oldtop = top;
  do {
                                 if (top == NULL)
    oldtop = top;
                                   return false;
    newnode->next = oldtop;
                                 newtop = oldtop->next;
  } while (cas(&top, oldtop,
                                 while (cas(&top, oldtop,
             newnode)
                                         newtop)
            == oldtop);
                                        == oldtop);
                               *valueptr = oldtop->value;
                               free (oldtop);
                               return true;
```



This almost works. But note that it depends on only loading <code>top</code> once, and otherwise only using <code>oldtop</code>, to make sure pointer accesses are consistent. It's also critical that in <code>trypop</code>, we don't try to access <code>oldtop->value</code> until after we are sure we own the node; otherwise somebody else might have freed it first. Finally, in a system where freeing memory might return it to the (segfaultable) pool, we might segfault when we follow <code>oldtop->next</code>.

But there's a more subtle bug. Suppose that after we assign to newtop in trypop, somebody else successfully pops a value (oldtop), frees it, pops another, then pushes two such that the second reuses oldtop. Now the CAS will work, but what we have in newtop isn't necessarily valid! The only cure is to ensure that no free happens until we're sure oldtop isn't going to be used in a CAS—perhaps by letting all other CPUs run first.

Load Linked / Store Conditional

Pseudocode:

```
int load linked(int *addr)
                             bool store conditional (
                               int *addr, newval)
  int origval;
  atomic {
                               atomic {
    origval = *addr;
                                  switch (
    mem watch (addr);
                                  watch result(addr)) {
                                    case UNCHANGED:
  return origval;
                                      *addr = newval;
                                      return true;
                                    case CHANGED:
                                      return false;
                                    case WASNT WATCHING:
                                      return false;
                                  stop_watching(addr);
```



Can you write increment? Answer: yes, because you can implement CAS with this.

But II/sc is limited, because often only one memory location can be watched at a time. So if many II are used at once, all but one might break. And in any case, there is no guarantee of fairness.

Which Processors Have What...

Instructions to perform *simple* changes in atomic read-*op*-write cycle.

```
m68k Compare and Swap (cas)

SPARC Compare and Swap (cas)

x86 Compare and Exchange (cmpxchgl)

MIPS Load-Linked/Store Conditional (ll/sc)
(R4000 upwards)

PowerPC Load Word & Reserve/Store Word Conditional (lwarx/stwcx)
```



Which Processors Have What...

Instructions to perform *simple* changes in atomic read-*op*-write cycle.

```
m68k Compare and Swap (cas)
```

SPARC Compare and Swap (cas)

x86 Compare and Exchange (cmpxchgl)

MIPS Load-Linked/Store Conditional (11/sc) (R4000 upwards)

PowerPC Load Word & Reserve/Store Word Conditional (lwarx/stwcx)

System/161 No hardware synchronization (MIPS R2000/R3000)



Which Processors Have What...

Instructions to perform *simple* changes in atomic read-*op*-write cycle.

```
m68k Compare and Swap (cas)
```

SPARC Compare and Swap (cas)

x86 Compare and Exchange (cmpxchgl)

MIPS Load-Linked/Store Conditional (11/sc) (R4000 upwards)

PowerPC Load Word & Reserve/Store Word Conditional (lwarx/stwcx)

System/161 No hardware synchronization (MIPS R2000/R3000)

Which primitives can we simulate and how?



What Do You Want?



What do you want?

Avoiding Locks & Slow Synchronization



When don't we need synchronization?

Bernstein's Conditions

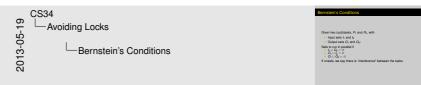
Given two (sub)tasks, P_1 and P_2 , with

- ▶ Input sets I_1 and I_2
- ▶ Output sets O_1 and O_2 :

Safe to run in parallel if

- $\begin{array}{c}
 I_1 \cup O_2 = \varnothing \\
 O_1 \cup I_2 = \varnothing
 \end{array}$
- $O_1 \cup \bar{O}_2 = \emptyset$

If unsafe, we say there is "interference" between the tasks.



A.J. Bernstein, IEEE Transactions on Electronic Computers, October 1966.