CS 134: Operating Systems

Locks and Low-Level Synchronization

CS 134:
Operating Systems
Locks and Low-Level Synchronization

Overview

Locks and Condition Variables

Beyond Locking

Avoiding Locks

Non-Blocking Synchronization

Avoiding Locks

Overview
Locks and Condition Variables
Beyond Locking
Avoiding Locks
Non-Blocking Synchronization
Avoiding Locks

Basic Operations

lock_acquire(lock) Simple mutual exclusion; locks out other threads

lock_release(lock) Release held lock

cv_wait(cond, lock) Atomically release **lock** and wait for signal on condition variable **cond**; reacquires **lock** before returning

cv_signal(cond, lock) Awaken thread (or all threads) waiting on (**cond**, **lock**)

- **lock** must be held
- **lock** not released
- Error if thread waiting on **cond** with some other lock
- Which thread selected if multiple waits?
- What behavior if no thread waiting?

Basic Operations

lock_acquire(lock) Simple mutual exclusion; locks out other threads

lock_release(lock) Release held lock

cv_wait(cond, lock) Atomically release **lock** and wait for signal on condition variable **cond**; reacquires **lock** before returning

cv_signal(cond, lock) Awaken thread (or all threads) waiting on (**cond**, **lock**)

- **lock** must be held
- **lock** not released
- Error if thread waiting on **cond** with some other lock
- Which thread selected if multiple waits?
- What behavior if no thread waiting?

It turns out that the best no-wait behavior is to discard the signal; that simplifies coding.
If multiple threads are waiting, it often makes sense to wake them all.

Bounded Buffer with Semaphores

```
enum { N = 128 };          // maximum capacity of the buffer
item_queue buffer;         // the buffer itself
struct sem *empty_slot;    // any free slots? (initialized to N)
struct sem *filled_slot;   // any filled slots? (initialized to 0)
struct sem *mutex;         // protection for the buffer (initialized to 1)
```

```
void producer()
{
  item made_item;

  for ( ; ; ) {
    made_item = make_item();
    P(empty_slot)
    P(mutex);
    put_item(buffer, made_item);
    V(mutex);
    V(filled_slot);
  }
}
```

```
void consumer()
{
  item usable_item;

  for ( ; ; ) {
    P(filled_slot);
    P(mutex);
    usable_item = get_item(buffer);
    V(mutex);
    V(empty_slot);
    use_item(usable_item);
  }
}
```

Bounded Buffer with Locks/CVs

```
item_queue buffer;        // the buffer itself
struct cv *has_space;     // any free slots?
struct cv *has_stuff;     // any filled slots?
struct lock *mutex;       // protection for the buffer
```

```
void producer()
{
  item made_item;

  for ( ; ; ) {
    made_item = make_item();
    lock_acquire(mutex);
    while (isFull(buffer))
        cv_wait(has_space, mutex);
    put_item(buffer, made_item);
    cv_signal(has_stuff, mutex);
    lock_release(mutex);
  }
}
```

```
void consumer()
{
  item usable_item;

  for ( ; ; ) {
    lock_acquire(mutex);
    while (isEmpty(buffer))
        cv_wait(has_stuff, mutex);
    usable_item = get_item(buffer);
    cv_signal(has_space, mutex);
    lock_release(mutex);
    use_item(usable_item);
  }
}
```

Readers–Writers Problem

Sometimes an object has

- Readers
  - Don't modify the object
  - Can *share* access with other readers
- Writers
  - May change the object
  - Cannot share access with others

You know this problem from 105! (In theory. . .)

Readers–Writers Problem
Sometimes an object has
Readers
Don't modify the object
Can *share* access with other readers
Writers
May change the object
Cannot share access with others
You know this problem from 105! (In theory. . .)

Readers/Writers with Locks & CVs

Form groups of 3-4 people. Between you, determine:

- The synchronization objects you'll need

Then, at the boards, everyone goes up to

- Declare `struct rwlock` (which might contain multiple locks) and initialization state
- Write `rwlock_readlock` & `rwlock_readunlock`
- Write `rwlock_writelock` & `rwlock_writeunlock`

Readers/Writers with Locks & CVs
Form groups of 3-4 people. Between you, determine:
- The synchronization objects you'll need
Then, at the boards, everyone goes up to
- Declare struct rwlock (which might contain multiple locks) and initialization state
- Write rwlock_readlock & rwlock_readunlock
- Write rwlock_writelock & rwlock_writeunlock

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?

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.

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?

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

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

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.

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.

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!!

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!!

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.

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 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?

Unix-Domain UDP Sockets—Class Exercise

Could you implement messaging where sender waits for reception?

Could you implement messaging that allows multiple receivers?

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?

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

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. . . ?

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. . . ?

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;
        v = w;                       // Line 4
    }
}
```


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 Fundamental Problem?
Class Exercise:
Identify the fundamental problem that prevents us from writing atomic_add correctly.

The difficulty is that we're replacing `*i` with `v` even if `*i` has changed in the meantime. We need a way to say "replace `*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. . . ?

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. . . ?

Increment with CAS is shown on next slide (not in handouts).

Increment with CAS

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


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)
{
  struct stacknode *newnode;

  newnode = malloc(...);

  newnode->value = value;
  newnode->next = top;

  top = newnode;
}
```

```
bool trypop(item *valueptr)
{
  item value;
  struct stacknode *oldtop;

  if (top == NULL)
    return false;

  oldtop = top;
  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)
{
  struct stacknode *newnode;
  struct stacknode *oldtop;

  newnode = malloc(...);

  newnode->value = value;
  do {
    oldtop = top;
    newnode->next = oldtop;
  } while (cas(&top, oldtop,
                newnode)
              == oldtop);
}
```

```
bool trypop(item *valueptr)
{
  item value;
  struct stacknode *oldtop;
  struct stacknode *newtop;

  do {
    oldtop = top;
    if (top == NULL)
      return false;
    newtop = oldtop->next;
  } while (cas(&top, oldtop,
                newtop)
              == oldtop);

  *valueptr = oldtop->value;
  free(oldtop);
  return true;
}
```


Non-Blocking Stack Code

This almost works. But note that it depends on only loading `top` once, and otherwise only using `oldtop`, to make sure pointer accesses are consistent. It's also critical that in `trypop`, we don't try to access `oldtop->value` 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 `oldtop->next`.

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)
{
  int origval;
  atomic {
    origval = *addr;
    mem_watch(addr);
  }
  return origval;
}
```

```
bool store_conditional(
  int *addr, newval)
{
  atomic {
    switch (
     watch_result(addr)) {
      case UNCHANGED:
        *addr = newval;
        return true;
      case CHANGED:
        return false;
      case WASNT_WATCHING:
        return false;
    }
    stop_watching(addr);
  }
}
```


Load Linked / Store Conditional
Pseudocode:
int load_linked(int *addr) { int origval; atomic { origval = *addr; mem_watch(addr); } return origval; }
bool store_conditional(int *addr, newval) { atomic { switch (watch_result(addr)) { case UNCHANGED: *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 ll/sc is limited, because often only one memory location can be watched at a time. So if many ll 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 (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 (`ll`/`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 (ll/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 (`ll`/`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?

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

Which primitives can we simulate and how?

What Do You Want?

What do you want?

What Do You Want?

What do you want?

Avoiding Locks & Slow Synchronization

When *don't* we need synchronization?

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

- $I_1 \cap O_2 = \varnothing$
- $O_1 \cap I_2 = \varnothing$
- $O_1 \cap O_2 = \varnothing$

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

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
$I_1 \cap O_2 = \varnothing$
$O_1 \cap I_2 = \varnothing$
$O_1 \cap O_2 = \varnothing$
If unsafe, we say there is "*interference*" between the tasks.

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