CS 134:
Operating Systems
Better Synchronization

CS 134:
Operating Systems
Better Synchronization

Overview

Overview

Aside: Attending a Conference

More Low-Level Synchronization

Higher-Level Primitives
atomic
yield

Avoiding Locks

How to Attend a Conference

OSDI is next week

How to get the most out of it?

How to Attend a Conference

OSDI is next week

How to get the most out of it?

Tech Sessions

- Program is posted at `https://www.usenix.org/conference/osdi12/tech-schedule/osdi-12-program`
  - Mouse over title to get abstract
  - Key for full-text versions will be sent this week
- Not required to attend all sessions
  - But should be over 50%
  - . . . and interest should be in 75%
- Use in-session time wisely
  - Treat it like class or colloquium
  - If you’re a scribe, take careful notes—you’re the only one!
- Wireless will be available

Tech Sessions

- Program is posted at `https://www.usenix.org/conference/osdi12/tech-schedule/osdi-12-program`
  - Mouse over title to get abstract
  - Key for full-text versions will be sent this week
- Not required to attend all sessions
  - But should be over 50%
  - . . . and interest should be in 75%
- Use in-session time wisely
  - Treat it like class or colloquium
  - If you’re a scribe, take careful notes—you’re the only one!
- Wireless will be available

Poster Sessions

- Two sessions Monday & Tuesday evenings
- Often best source of information about cutting-edge research
- Budget your time wisely
- Spend time getting detail on posters that interest you
- Finger food will be provided

Poster Sessions

- Two sessions Monday & Tuesday evenings
- Often best source of information about cutting-edge research
- Budget your time wisely
- Spend time getting detail on posters that interest you
- Finger food will be provided

BOFs

- Late evenings
- Choose wisely—often not terribly informative

BOFs
- Late evenings
- Choose wisely—often not terribly informative

The “Hallway Track”

- Often considered most important part of a conference
- Takes place at breaks, at lunch, poster sessions, etc.
- Chance to learn more, get to know useful people
- Get up your gumption and talk to a stranger!
  - Choose small groups (2-3)
    - Should have at least one younger person
  - OK to talk to anyone who’s alone
  - I will introduce you to anybody I’m talking to
    - Don’t join if large group (limits exposure)
    - Don’t cling (limits variety)
  - Good chance to quiz people with interesting papers/posters

The “Hallway Track”
- Often considered most important part of a conference
- Takes place at breaks, at lunch, poster sessions, etc.
- Chance to learn more, get to know useful people
- Get up your gumption and talk to a stranger!
  - Choose small groups (2-3)
    - Should have at least one younger person
  - OK to talk to anyone who’s alone
  - I will introduce you to anybody I’m talking to
    - Don’t join if large group (limits exposure)
    - Don’t cling (limits variety)
  - Good chance to quiz people with interesting papers/posters

Where We Were. . .

Last time we looked at Test-and-Set, Swap, and Compare-and-Swap

T&S is good for locking; Swap isn't good for much of anything. C&S can be used for lock-free synchronization—if you're very careful!

Where We Were. . .

Last time we looked at Test-and-Set, Swap, and Compare-and-Swap

T&S is good for locking; Swap isn't good for much of anything. C&S can be used for lock-free synchronization—if you're very careful!

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?

Higher-Level Primitives

The idea of wanting to do things atomically seems like a good one...

Higher-Level Primitives

The idea of wanting to do things atomically seems like a good one...

Higher-Level Primitives

The idea of wanting to do things atomically seems like a good one. . .

```
atomic {
    yourBalance = yourbalance - 100;
    myBalance = myBalance + 100.00;
}
```


Higher-Level Primitives

The idea of wanting to do things atomically seems like a good one. . .

atomic {
    yourBalance = yourbalance - 100;
    myBalance = myBalance + 100.00;
}

Recap: 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);
  }
}
```


Recap: Bounded Buffer with Locks/CVs

Bounded Buffer with `atomic`

```
item_queue buffer;        // the buffer itself

void producer()                      void consumer()
{                                    {
  item made_item;                      item usable_item;

  for ( ; ; ) {                        for ( ; ; ) {
    made_item = make_item();             atomic {
    atomic {                               while (isEmpty(buffer))
      while (isFull(buffer))                 ;
        ;                                  usable_item = get_item(buffer);
      put_item(buffer, made_item);       }
    }                                    use_item(usable_item);
  }                                    }
}                                    }
```


Bounded Buffer with atomic

The problem with this implementation is it deadlocks because of the loop inside the `atomic` block.

Bounded Buffer with `atomic`

```
item_queue buffer;        // the buffer itself
```

```
void producer()
{
  item made_item;

  for ( ; ; ) {
    made_item = make_item();
    atomic {
      while (isFull(buffer))
        retry();
      put_item(buffer, made_item);
    }
  }
}
```

```
void consumer()
{
  item usable_item;

  for ( ; ; ) {
    atomic {
      while (isEmpty(buffer))
        retry();
      usable_item = get_item(buffer);
    }
    use_item(usable_item);
  }
}
```


Bounded Buffer with atomic

Alternative Bounded Buffer with `atomic`

```
item_queue buffer;        // the buffer itself
```

```
void producer()
{
  item made_item;

  for ( ; ; ) {
    made_item = make_item();
    atomic (!isFull(buffer)) {


      put_item(buffer, made_item);
    }
  }
}
```

```
void consumer()
{
  item usable_item;

  for ( ; ; ) {
    atomic (!isEmpty(buffer)) {


      usable_item = get_item(buffer);
    }
    use_item(usable_item);
  }
}
```


Alternative Bounded Buffer with atomic

Discussion

What's good/bad/poorly specified?

How is it implemented?

Discussion
What's good/bad/poorly specified?
How is it implemented?

How is this implemented? (It's sometimes done as a global lock.)

If you forget `atomic` in one thread, things break.

Retry isn't needed if there are no conditionals (but are there conditionals in `get_item`?).

Alternative: rollback.

A Gentler Time

Cooperative multitasking: scheduler runs at thread's request

Net effect: everything is atomic (except for interrupts)

A Gentler Time

Cooperative multitasking: scheduler runs at thread's request

Net effect: everything is atomic (except for interrupts)

Bounded Buffer with `atomic`

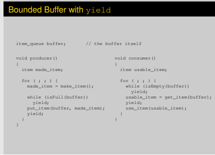
```
item_queue buffer;        // the buffer itself

void producer()
{
  item made_item;

  for ( ; ; ) {
    made_item = make_item();
    atomic {
      while (isFull(buffer))
        ;
      put_item(buffer, made_item);
    }
  }
}
```

```
void consumer()
{
  item usable_item;

  for ( ; ; ) {
    atomic {
      while (isEmpty(buffer))
        ;
      usable_item = get_item(buffer);
    }
    use_item(usable_item);
  }
}
```

Bounded Buffer with `yield`

```
item_queue buffer;        // the buffer itself

void producer()
{
  item made_item;

  for ( ; ; ) {
    made_item = make_item();

    while (isFull(buffer))
      yield;
    put_item(buffer, made_item);
    yield;
  }
}
```

```
void consumer()
{
  item usable_item;

  for ( ; ; ) {
    while (isEmpty(buffer))
      yield;
    usable_item = get_item(buffer);
    yield;
    use_item(usable_item);
  }
}
```

Avoiding Locks & Slowness of Synchronization

When *don't* we need synchronization?

Avoiding Locks & Slowness of 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.