Synchronization (3)

Dave Eckhardt
de0u@andrew.cmu.edu

Synchronization

- P2 (et seq.) partners
  - “Partner Registration Page” on web site
- Good things to talk about
  - How many late days?
  - Projects in other classes?
  - Auditing or pass/fail?
  - Prior experience
  - Class load

Outline

- Condition variables
  - Under the hood
  - The atomic-sleep problem
- Semaphores
- Monitors

Voluntary de-scheduling

- The Situation
  - You hold lock on shared resource
  - But it's not in “the right mode”
- Action sequence
  - Unlock shared resource
  - Go to sleep until resource changes state

What *not* to do

```
while (!reckoning) {
  mutex_lock(&scenario_lk);
  if ((date >= 1906-04-18) &&
    (hour >= 5))
    reckoning = true;
  else
    mutex_unlock(&scenario_lk);
}
wreak_general_havoc();
mutex_unlock(&scenario_lk);
```

Arguably Less Wrong

```
while (!reckoning) {
  mutex_lock(&scenario_lk);
  if ((date >= 1906-04-18) &&
    (hour >= 5))
    reckoning = true;
  else {
    mutex_unlock(&scenario_lk);
    sleep(1);
  }
}
wreak_general_havoc();
mutex_unlock(&scenario_lk);
```

Something is missing

- Mutex protects shared state
  - Good
- How can we sleep for the ***right*** duration?
  - Get an expert to tell us!

Once more, with feeling!

```
mutex_lock(&scenario_lk);
while (cvar = wait_on()) {
  cond_wait(&scenario_lk, &cvar);
}
wreak_general_havoc(); /* locked! */
mutex_unlock(&scenario_lk);
```

wait_on()?

```
if (y < 1906)
  return (&new_year);
else if (m < 4)
  return (&new_month);
else if (d < 18)
  return (&new_day);
else if (h < 5)
  return (&new_hour);
else
  return (0);
```

What wakes us up?

```
for (y = 1900; y < 2000; y++)
  for (m = 1; m <= 12; m++)
    for (d = 1; d <= days(month); d++)
      for (h = 0; h < 24; h++)
        ...
        cond_signal(&new_hour);
      cond_signal(&new_day);
    cond_signal(&new_month);
  cond_signal(&new_year);
```

Condition Variable Design

- Basic Requirements
  - Keep track of threads asleep “for a while”
  - Allow notifier thread to wake sleeping thread(s)
  - Must be thread-safe

Why *two* parameters?

```
condition_wait(mutex, cvar);
```

- Lock required to access/modify the shared state
- Whoever awakens you will need to hold that lock
  - You'd better give it up.
- When you wake up, you will need to hold it
  - “Natural” for condition_wait() to un-lock/re-lock
- But there's something more subtle

Condition Variable Implementation

- mutex
  - multiple threads can condition_wait() at once
- “queue” - of sleeping processes
  - FIFO or more exotic

Condition Variable Implementation

```
cond_wait(mutex, cvar)
{
  lock(cvar->mutex);
  enq(cvar->queue, my_thread_id());
  unlock(mutex);
  ATOMICALLY {
    unlock(cvar->mutex);
    pause_thread();
  }
}
```

- What is this “ATOMICALLY” stuff?

Pathological execution sequence

cond_wait(m, c);	*cond_signal(c);*
`enq(c->que, me);`	
`unlock(m);`	
`unlock(c->m);`	
	`lock(c->m);`
	`id = deq(c->que);`
	`thr_wake(id);`
	`unlock(c->m);`
`thr_sleep();`	

Achieving condition_wait() Atomicity

- Disable interrupts (if you are a kernel)
- Rely on OS to implement condition variables
  - (yuck?)
- Have a “better” sleep()/wait() interface

Semaphore Concept

- Integer: number of free instances of a resource
- Thread blocks until it is allocated an instance
- wait(), aka P(), aka proberen("wait")
  - wait until value > 0
  - decrement value
- signal(), aka V(), aka verhogen("increment")
  - increment value
- Just one small issue...
  - wait() and signal() ***must be atomic***

“Mutex-style” Semaphore

```
semaphore m = 1;
do {
  wait(m); /* mutex_lock() */
  ..critical section...
  signal(m); /* mutex_unlock() */
  ...remainder section...
} while (1);
```

“Condition-style” Semaphore

Thread 0	*Thread 1*
	`wait(c);`
`result = 42;`	
`signal(c);`	
	`use(result);`

“Condition with Memory”

Semaphores ***retain memory*** of signal() events

“full/empty bit”

Thread 0	***Thread 1***
`result = 42;`	
`signal(c);`	
	`wait(c);`
	`use(result);`

Semaphore vs. Mutex/Condition

- Good news
  - Semaphore is a higher-level construct
  - Integrates mutual exclusion, waiting
  - Avoids mistakes common in mutex/condition API
    - Lost signal()
    - Reversing signal() and wait()
    - ...

Semaphore vs. Mutex/Condition

- Bad news
  - Semaphore is a higher-level construct
  - Integrates mutual exclusion, waiting
    - Some semaphores are “mutex-like”
    - Some semaphores are “condition-like”
    - How's a poor library to know?

Semaphores - 31 Flavors

- Binary semaphore
  - It counts, but only from 0 to 1!
    - “Available” / “Not available”
  - Consider this a hint to the implementor...
    - “Think mutex!”
- Non-blocking semaphore
  - wait(semaphore, timeout);
- Deadlock-avoidance semaphore
  - #include <deadlock.lecture>

My Personal Opinion

- One ***simple, intuitive*** synchronization object
- In 31 performance-enhancing flavors!!!
- “The nice thing about standards is that you have so many to choose from.”
  - Andrew S. Tanenbaum

Semaphore Wait: The Inside Story

```
wait(semaphore s) {
  ACQUIRE EXCLUSIVE ACCESS
  --s->count;
  if (s->count < 0) {
    enqueue(s->queue, my_id());
    ATOMICALLY
      RELEASE EXCLUSIVE ACCESS
      thread_pause()
    } else {
      RELEASE EXCLUSIVE ACCESS
    }
}
```

Semaphore Signal - The Inside Story

```
signal(semaphore s) {
  ACQUIRE EXCLUSIVE ACCESS
  ++s->count;
  if (s->count <= 0) {
    tid = dequeue(s->queue);
    thread_wakeup(tid);
}
RELEASE EXCLUSIVE ACCESS
```

- What's all the shouting?
  - An exclusion algoritm much like a mutex
  - OS-assisted atomic de-scheduling

Monitor

- Basic concept
  - Semaphore eliminate some mutex/condition mistakes
  - Still some common errors
    - Swapping “signal()” & “wait()”
    - Accidentally omitting one
- Monitor: higher-level abstraction
  - Module of high-level language procedures
    - All access some shared state
  - ***Compiler*** adds synchronization code
    - Thread in any procedure blocks ***all*** thread entries

Monitor “commerce”

```
int cash_in_till[N_STORES] = { 0 };
int wallet[N_CUSTOMERS] = { 0 } ;

boolean buy(int cust, store, price) {
  if (wallet[cust] >= price) {
    cash_in_till[store] += price;
    wallet[cust] -= price;
    return (true);
  } else
    return (false);
}
```

Monitors – What about waiting?

- Automatic mutal exclusion is nice...
  - ...but it is too strong
- Sometimes one thread needs to wait for another
  - Automatic mutual exclusion forbids this
  - Must leave monitor, re-enter - ***when?***
- Have we heard this “when” question before?

Monitor condition variables

- Similar to condition variables we've seen
- condition_wait(cvar)
  - Only one parameter
  - Mutex-to-drop is implicit
    - (the “monitor mutex”)
- signal() policy question - which thread to run?
  - Signalling thread? Signalled thread?
  - Or: signal() ***exits monitor*** as side effect

Summary

- Two fundamental operations
  - Mutual exclusion for must-be-atomic sequences
  - Atomic de-scheduling (and then wakeup)
- Mutex style
  - Two objects for two core operations
- Semaphores, Monitors
  - ***Same core ideas inside***

Summary

- What you should know
  - Issues/goals
  - Underlying techniques
  - How environment/application design matters
- All done with synchronization?
  - Only one minor issue left
    - Deadlock