Lecture 9: Conditional Variables

Yiying Zhang
Anouncement

• HW1 solution released

• Midterm review next Friday
Implementing Locks with a queue

- If cannot hold lock, give up CPU (move to block queue)
- Use a guard on the lock itself

```c
struct lock {
    int held = 0;
    queue Q;
}

void acquire (lock) {
    if (lock->held == 0) {
        lock->held = 1;
        return;
    }
    put current thread on lock->Q;
    go to sleep;
}

void release (lock) {
    if (lock->Q is empty)
        lock->held = 0;
    if (lock->Q is not empty)
        move a waiting thread to the ready queue;
}
```
Implementing Locks with a queue

- If cannot hold lock, give up CPU (move to block queue)
- Use a guard on the lock itself

```c
struct lock {
    int held = 0;
    int guard = 0;
    queue Q;
}

void acquire (lock) {
    while (test-and-set(lock\rightarrow guard)) ;
    if (lock\rightarrow held == 0) {
        lock\rightarrow held = 1;
        lock\rightarrow guard = 0;
        return;
    }
    put current thread on lock\rightarrow Q;
    lock\rightarrow guard = 0;
    go to sleep;
}

void release (lock) {
    while (test-and-set(lock\rightarrow guard)) ;
    if (lock\rightarrow Q is empty) {
        lock\rightarrow held = 0;
    } else {  // lock\rightarrow Q is not empty
        move a waiting thread to the ready queue;
        lock\rightarrow guard = 0;
    }
}
```
Implementing Locks with a queue

- If cannot hold lock, give up CPU (move to block queue)
- Use a guard on the lock itself

```c
struct lock {
    int held = 0;
    int guard = 0;
    queue Q;
}

void acquire (lock) {
    disable interrupts;
    while (test-and-set(lock->guard)) ;
    if (lock->held == 0) {
        lock->held = 1;
        lock->guard = 0;
        enable interrupts;
    }
    put current thread on lock->Q;
    lock->guard = 0;
    go to sleep;
    enable interrupts;
}

void release (lock) {
    disable interrupts;
    while (test-and-set(lock->guard)) ;
    if (lock->Q is empty) {
        lock->held = 0;
    } else {
        move a waiting thread to the ready queue;
        lock->guard = 0;
        enable interrupts;
    }
}
Deep Thinking

• Why is this busy waiting (the while loop) not a concern?
  ♦ What’s our critical section here?

• Can we remove the disable/enable interrupts?
  ♦ With interrupts, when a process that gets the guard (pass the while loop) get context switched out, all other wait processes on other cores will be busy waiting
Synchronization Primitives

- Lock is useful when implementing critical sections

- Critical sections’ semantic is to mutually exclude

- Mutual exclusion does not solve all the synchronization problems

- Sometimes, we need other semantics
  - E.g., wait for shared resources to become available
  - E.g., allowing multiple threads to generate/get (different) shared resources
  - E.g., use certain conditions to decide when to enter CS
Producer & Consumer Problem

- **Producer**: creates copies of a resource
- **Consumer**: uses up (destroys) copies of a resource.
- **Buffers**: fixed size, used to hold resource produced by producer before consumed by consumer

```
Producer   N = 4
            | 2 empty slots
            v 2 occupied slots
Consumer
```

CSE 120 – Lecture 6 – Semaphores
Producer & Consumer Problem

• Producer and consumer execute at different rates
  ♦ No serialization of one behind the other
  ♦ There can be multiple producers and multiple consumers
  ♦ Tasks are independent (easier to think about)
  ♦ The buffer set allows each to run without explicit handoff

• Synchronization: ensuring concurrent producers & consumers access the buffer in a correct way
  ♦ What’s a “correct way”?

• Happens inside OS all the time (e.g., I/Os)
Producer & Consumer – simple attempt, what’s wrong?

Producer

while (1) {
    produce an item;
    insert(item to pool);
}

Consumer

While (1) {
    remove(item from pool);
    consume the item;
}
Producer & Consumer – simple attempt, what’s wrong?

Producer

while (1) {

    produce an item;

    wait until pool is not full

    insert(item to pool);

}

Consumer

While (1) {

    wait until pool is not empty

    remove(item from pool);

    consume the item;

}
Producer & Consumer – simple attempt, what’s wrong?

Producer
while (1) {
    produce an item;

    while (pool is full) {
        ;
    }
    Intercepted by another producer thread!
    insert(item to pool);
}

Consumer
While (1) {
    while (pool is empty) {
        ;
        Intercepted by another consumer thread!
        remove(item from pool);
    }
    use the item for some work;
}
int buffer[MAX];
int fill = 0;
int use = 0;

insert (int value) {
    buffer[fill] = value;
    fill = (fill + 1) % MAX
}

int remove() {
    int tmp = buffer[use]
    use = (use + 1) % MAX
    return tmp;
}

Need to protect shared resource (critical section)!
Producer & Consumer – lock?

Producer
while (1) {
    produce an item;
    acquire(lock);
    while (pool is full)
        ;
    insert(item to pool);
    release(lock);
}

Consumer
While (1) {
    acquire(lock);
    while (pool is empty)
        ;
    remove(item from pool);
    release(lock);
    use the item for some work;
}
Often times, we have to wait for shared resources

- Busy waiting is a bad idea
- Checking resources itself needs to be in critical section!
- Busying waiting inside CS even worse!
  - No one else can check!

→ Need a more powerful sync. primitive!
→ Want the primitive to check & wait
Higher-Level Synchronization

• We want synchronization mechanisms that
  ♦ Provide semantics beyond mutual exclusion

• Locks and condition variables
  ♦ Lock alone is not flexible enough
  ♦ Need some mechanism to check conditions

• Monitor

• Semaphore (next lecture)
  ♦ Binary (mutex) and counting
Conditional Variables

• An explicit queue that threads can put themselves on when some state of execution (i.e., some condition) is not as desired (by waiting on the condition)
  ◆ Also called wait (Java, C++), sleep (Nachos, C#)
• Some other thread, when it changes said state, can then wake one (or more) of those waiting threads and thus allow them to continue (by signaling on the condition)
  ◆ Wake up one: wake (Nachos, C#), notify (Java), notify_one (C++)
  ◆ Wake up all: wakeAll (Nachos, C#), notifyAll (Java), notify_all (C++)
Conditional Variables

- Used in conjunction with locks
- Used inside critical section to wait for certain conditions

Contrast with Semaphore:
- Has no counting bundled
- More intuitive to many people

Usage
- On creation, specify which lock it is associated with
Conditional Variables

- Wait (condition)
  - Block on "condition"

- Signal (condition)
  - Wakeup one or more threads blocked on "condition"
  - Signal is no-op if none blocked

- Associated with a lock

Queue of waiting Process trying to Enter CSes protected by lock L

Queues associated with x, y condition

Lock: L
Condition variables: x(L) y(L)

Shared data
operations
Producer & Consumer -- use condition variables

**Producer**

```c
while (1) {
    produce an item;
    acquire(mutex);
    if (pool is Full) {
        wait(NotFULL);
    }
    insert(item);
    release(mutex);
}
```

**Consumer**

```c
While (1) {
    acquire(mutex);
    if (pool is Empty) {
        wait(NotEMPTY);
    }
    remove(item);
    release(mutex);
}
```
Producer & Consumer -- use condition variables

Producer

while (1) {

produce an item;

acquire(mutex);
if (pool is Full) {

wait(NotFULL);

wait(NotFULL);
}

insert(item);

signal(NotEMPTY);
release(mutex);

}

Consumer

While (1) {

acquire(mutex);
if (pool is Empty) {

wait(NotEMPTY);

}

remove(item);

signal(NotFULL);
release(mutex);

use the item for some work;

}
Producer & Consumer -- use condition variables

Producer

while (1) {

produce an item;

acquire(mutex);
if (pool is Full) {
wait(NotFULL);
}
record if pool was empty;
insert(item);

if (pool was empty)
signal(NotEMPT); release(mutex);
}

Consumer

While (1) {
acquire(mutex);
if (pool is Empty) {
wait(NotEMPTY);
}
record if pool was full;
remove(item);

if (pool was Full)
signal(NotFULL);
release(mutex);
use the item for some work;
}
Producer & Consumer -- use condition variables

Producer

while (1) {

produce an item;

acquire(mutex);
if (pool is Full) {
  release(mutex);
  wait(NotFULL);
  acquire(mutex);
}
record if pool was empty;
insert(item);
if (pool was empty)
  signal(NotEMPTY);
release(mutex);
 }

Consumer

While (1) {

acquire(mutex);
if (pool is Empty) {
  release(mutex);
  wait(NotEMPTY);
  acquire(mutex);
}
record if pool was full;
remove(item);
if (pool was Full)
  signal(NotFULL);
release(mutex);
use the item for some work;
}

Put me To sleep

If anyone is sleeping, wake it up
Producer & Consumer -- use condition variables

**Producer**

```c
while (1) {

    produce an item;

    acquire(mutex);
    if (pool is Full) {
        wait(NotFULL);
    }

    record if pool was empty;
    insert(item);

    if (pool was empty)
        signal(NotEMPTY);

    release(mutex);
}
```

**Consumer**

```c
While (1) {

    acquire(mutex);
    if (pool is Empty) {
        wait(NotEMPTY);
    }

    record if pool was full;
    remove(item);

    if (pool was Full)
        signal(NotFULL);
    release(mutex);

    use the item for some work;
}
```

The simplification implies NotFull is tied to mutex.
• One problem – what happens on wakeup?
  ♦ Only one thing can be inside critical section
  ♦ But wakeup implies both signaler and waiter may be in critical section, who should go on?
Signal Semantics

- signal() places a waiter on the ready queue, but signaler continues inside lock
  - Known as “Mesa” style
  - Easy to implement
  - Another early-time semantics is Hoare style (signaler gives up lock, waiter runs immediately)

- What can happen when the awaken thread gets a chance to run?
  - E.g. pool is full, producer 1 waits; consumer signals it; p1 in ready queue; consumer release(lock); p2 comes along…
**Producer & Consumer – CV problem?**

### Producer

```c
while (1) {
    produce an item;

    acquire(mutex);
    if (pool is Full) {
        release(mutex);
        wait(NotFULL);
        acquire(mutex);
    }
    record if pool was empty;
    insert(item);
    if (pool was empty)
        signal(NotEMPTY);

    release(mutex);
}
```

### Consumer

```c
While (1) {
    acquire(mutex);
    if (pool is Empty) {
        release(mutex);
        wait(NotEMPTY);
        acquire(mutex);
    }
    record if pool was full;
    remove(item);

    if (pool was Full)
        signal(NotFULL);
    release(mutex);

    use the item for some work;
}
```

**Put me To sleep**

If anyone is sleeping, wake it up
Signal Semantics

• What can happen when the awaken thread gets a chance to run?
  ♦ E.g. pool is full, producer 1 waits; consumer signals it; p1 in ready queue; consumer release(lock); p2 comes along…

• Condition not necessarily true when waiter runs again
  ♦ Returning from wait() is only a hint that something changed
  ♦ Must recheck conditional case
Producer & Consumer – use condition variables – how to fix?

Producer

while (1) {

produce an item;

acquire(mutex);
while (pool is Full) {
    wait(NotFULL);
}
record if pool was empty;
insert(item);

if (pool was empty)
    signal(NotEMPTY);
release(mutex);
}

Consumer

While (1) {

acquire(mutex);
while (pool is Empty) {
    wait(NotEMPTY);
}
record if pool was full;
remove(item);

if (pool was Full)
    signal(NotFULL);
release(mutex);
use the item for some work;
}

Is this busy waiting?
Be Careful About Pitfalls: CVs Cannot Be “Tested”

- Do not use a CV as a predicate
- Need to use a separate flag

```c
acquire(lock);
...
while (CV != true) {
    wait(CV);
}
...
release(lock);
```

```c
acquire(lock);
...
while (flag != true) {
    wait(CV);
}
...
release(lock);
```
Be Careful About Pitfalls: CVs Require Holding Lock

- Do not release the lock before using the CV
  - Using a CV requires a thread to hold the lock
- Purpose of a CV is to enable threads to block while in a critical section
Be Careful About Pitfalls: Need Lock When Testing Flag

- Testing a condition needs to be done while holding the lock
- It is a shared variable that can lead to race conditions

```plaintext
... if (check-condition) {
    acquire(lock);
    wait(CV);
    release(lock);
}
...

acquire(lock);
...
if (check-condition) {
    wait(CV);
}
...
release(lock);
```
Monitors

- A monitor is a programming language construct that controls access to shared data
  - Synchronization code added by compiler, enforced at runtime
- A monitor is a module that encapsulates
  - Shared data structures
  - Procedures that operate on the shared data structures
  - Synchronization between concurrent threads that invoke the procedures
- A monitor protects its data from unstructured access
- It guarantees that threads accessing its data through its procedures interact only in legitimate ways
- If curious, read more in backup slides
Monitors

- A monitor is a programming language construct that controls access to shared data
  - Synchronization code added by compiler, enforced at runtime
- A monitor is a module that encapsulates
  - Shared data structures
  - Procedures that operate on the shared data structures
  - Synchronization between concurrent threads that invoke the procedures
- A monitor protects its data from unstructured access
- It guarantees that threads accessing its data through its procedures interact only in legitimate ways
Monitor Semantics

- A monitor guarantees mutual exclusion
  - Only one thread can execute any monitor procedure at any time (the thread is “in the monitor”)
  - If a second thread invokes a monitor procedure when a first thread is already executing one, it blocks
    - So the monitor has to have a wait queue…
  - Can have a condition variable inside a monitor
Account Example

Monitor `account`
{
    double balance;
    double `withdraw(amount)` {
        balance = balance - amount;
        return balance;
    }
}

`withdraw(amount)`
balance = balance - amount;
return balance;

Threads block waiting to get into monitor

When first thread exits, another can enter. Which one is undefined.

♦ Hey, that was easy!
♦ But what if a thread wants to wait inside the monitor?
   » Such as “mutex(empty)” by reader in bounded buffer?
Monitors, Monitor Invariants and Condition Variables

- A **monitor invariant** is a **safety property** associated with the monitor, expressed over the monitored variables. It holds whenever a thread enters or exits the monitor.
- A **condition variable** is associated with a **condition** needed for a thread to make progress once it is in the monitor.

Monitor M {
  ...
  monitored variables
  Condition c;

  void enterMonitor (...)
  {
    if (extra property not true) wait(c); waits outside of the monitor's mutex
    do what you have to do
    if (extra property true) signal(c); brings in one thread waiting on condition
  }
}
Monitors and Java

- A lock and condition variable are in every Java object
  - Later added explicit classes for locks or condition variables
- Every object is/has a monitor
  - At most one thread can be inside an object’s monitor
  - A thread enters an object’s monitor by
    - Executing a method declared `synchronized`
      - Can mix synchronized/unsynchronized methods in same class
    - Executing the body of a `synchronized` statement
      - Supports finer-grained locking than an entire method
      - Identical to the Modula-2 “LOCK (m) DO” construct
  - The compiler generates code to acquire the object’s lock at the start of the method and release it just before returning
    - The lock itself is implicit, programmers do not worry about it
Monitors and Java

• Every object can be treated as a condition variable
  ♦ Half of Object’s methods are for synchronization!

• Take a look at the Java Object class:
  ♦ Object.wait(*) is wait (Condition.sleep in Nachos)
  ♦ Object.notify() is signal (Condition.wake)
  ♦ Object.notifyAll() is broadcast (Condition.wakeAll)
Modern Languages

• Modern languages provide some form of locks and condition variables for synchronization and coordination
  ♦ C, C++, C#, Java, Go, Rust, …
  ♦ Most common form of synchronization you will encounter

• Typically locks are explicit
  ♦ Programmers have to use acquire and release explicitly
    » C++ and Rust have “release on return” language semantics
    » A half-way monitor implementation…
  ♦ Even Java eventually added separate classes (Lock, Condition) for flexibility