Lecture #8: ~~Semaphores~~ Shared objects, Monitors, Condition Variables, and Bounded buffer

****
Review -- 1 min
****

- Hardware support for synchronization
- Building higher-level synchronization programming
- abstractions on top of hardware support (e.g., Lock)

****
Outline - 1 min
****

~~Definition of semaphore~~
~~Example of programming w. semaphore~~
~~Semaphore expresses 2 types of synchronization~~
- ~~mutex (like lock)~~
- ~~synchronization (wait for some event)~~

~~Simple implementation (time permitting)~~

Two kinds of synchronization
Monitor = lock + c.v. + shared state = shared object
Simple implementation

****
Preview - 1 min
****
How to program with shared objects

****
Lecture - 32 min
****

1. Motivation

writing concurrent programs hard – coordinate updates to shared memory

synchronization – coordinating multiple concurrent activities that are using shared state

Question: what are the right synchronization abstractions to make it easy to build concurrent programs?

Answer will necessarily be a compromise :
- between making it easy to modify shared variables any time you want and controlling when you can modify shared variables.
- between really flexible primitives that can be used in a lot of different ways and simple primitives that can only be used one way (but are more difficult to misuse)

Rules will seem a bit strange – why one definition and not another?
- no absolute answer
- history has shown that they are reasonably good – if you follow these definitions, you will find writing correct code easier.
- for now just take them as a given; use it for a while; then, if you can come up with something better, be my guest!

2. Shared object abstraction

[[PICTURE -- shared state, methods operating on shared state

-- example -- bounded buffer/producer consumer queue
-- methods: add(), remove()
-- state: linked list (or array or ...), fullCount, ...
-- Accessed by several threads --> **must synchronize access]]**

3. 2 "types" of synchronization

Convenient to break synchronization into two cases

(1) **Mutual exclusion** – only allow one thread to access a given set of shared state at a time

E.g., bounded buffer

How do we do it?
Each shared object has lock and shared state variables

Public methods acquire the lock before reading/writing member state variables

(2) **Scheduling constraints –** wait for some other thread to do something

E.g., bounded buffer....

General problem
e.g., wait for other thread to finish, wait for other thread to produce work, wait for other thread to consume work, wait for other thread to accept a connection, wait for other thread to get bytes off disk, …

How do we do it?
Need new synchronization primitive "Wait until X"

~~4. Definition of Semaphores~~

~~like a generalized lock~~
~~first defined by Dijkstra in late 60's~~
~~originally main synchronization primitive in Unix (now others available)~~

~~**semaphore** – has a non-negative integer value and supports the following two operations:~~
~~semaphore->P() – an atomic operation that waits for the semaphore to become positive; then decrements it by 1~~
~~semaphore->V() – an atomic operation that increments the semaphore by 1, waking up a waiting P if any~~

~~Like integers, except:~~
1) ~~No negative values~~
2) ~~Only operations are P() and V() – can't read or write the value (except to set it initially)~~
3) ~~operations must be atomic – two P's that occur together can't decrement the value below zero. Similarly, thread going to sleep in P won't miss wakeup from V, even if they both happen at about the same time~~

~~**binary semaphore** – instead of an integer value, has a boolean value.~~
~~P waits until value is 1, then sets it to 0~~
~~V sets value to 1, waking up a waiting P if any~~

~~5. Two uses of semaphores~~

~~5.1 mutual exclusion~~

~~When semaphores are used for mutual exclusion, the semaphore has an initial value of 1, and P() is called before the critical section, and V() is called after the critical section~~

~~semaphore = new Semaphore(1);~~
~~…~~
~~semaphore->P();~~
~~// critical section goes here~~
~~semaphore->V();~~

~~5.2 scheduling constraints~~

~~semaphores can be used to describe general scheduling constraints – e.g. they provide a way to wait for something~~

~~usually in this case (but not always) the initial value for the semaphore is 0~~

~~Example: Wait for another thread to get done processing a request~~

~~*********************************~~
~~Admin - 3 min~~
~~*********************************~~

~~*********************************~~

~~Lecture - 30 min~~

~~*********************************~~

~~6. Producer-consumer with bounded buffer~~

~~6.1 problem definition~~

~~producer puts things into a shared buffer~~
~~consumer takes them out~~

~~need synchronization for coordinating producer and consumer~~

~~e.g. cpp | cc1 | cc2 | as~~
~~e.g., read/write network/disk (e.g., web server reads from disk, sends to network while your web client reads from network and draws to screen)~~

~~Don't want producer and consumer to operate in lock-step, so put a fixed sized **buffer** between them.~~
~~Synchronization – producer must wait if buffer is full; consumer must wait if buffer is empty~~

~~e.g. coke machine~~
~~producer is delivery person~~
~~consumer is students and faculty~~

~~Notice: *shared object* (coke machine) *separate from threads* (delivery person, students, faculty). Shared object coordinates activity of threads.~~
~~Common confusion on project – try to do the synchronization within the threads' code. No, the synchronization happens within the shared objects. "Let the shared objects do the work."~~

~~Solution uses semaphores for both mutex and scheduling~~

~~6.2 Correctness constraints for solution~~

~~***Synchronization problems have semaphores represent 2 types of constraint***~~

- ~~***mutual exclusions***~~
- ~~***wait for some event***~~

~~***When you start working on a synchronization problem, first define the mutual exclusion constraints, then ask "when does a thread wait", and create a separate synchronization variable representing each constraint***~~

~~QUESTION: what are the constraints for bounded buffer?~~
~~1) only one thread can manipulate buffer queue at a time~~
~~*mutual exclusion*~~
~~2) consumer must wait for producer to fill buffers if none full~~
~~*scheduling constraint*~~
~~3) producer must wait for consumer to empty buffers if all full~~
~~*scheduling constraint*~~

~~Use a separate semaphore for each constraint~~

```
~~Semaphore mutex;~~
~~Semaphore fullBuffers; // consumer's constr~~
~~                         // if 0 no coke~~
~~Semaphore emptyBuffers; // producer's constr.~~
~~           // if 0, nowhere to put more coke~~
```

~~6.3 Solution~~

```
~~Class CokeMachine{~~

~~Semaphore new mutex(1);// no one using machine~~
~~Semaphore new fullBuffers(0); // initally no coke!~~
~~Semaphore new emptyBuffers(numBuffers);~~
~~         // initially # empty slots~~
~~         // semaphore used to count how many~~
~~         // resources there are~~

~~Produce(Coke *coke){~~
~~  emptyBuffers.P();   // check if there is space~~
~~                      // for more coke~~
~~  mutex.P();          // make sure no one else~~
~~                      // using machine~~
```

```
  ~~put 1 coke in machine~~

  ~~mutex.V();              // OK for others to use~~
                            ~~// machine~~
  ~~fullBuffers.V();        // tell consumers there is~~
                            ~~// now a coke in machine~~
~~}~~


~~Coke *Consume(){~~
  ~~fullBuffers.P();  // check if there's a coke~~
  ~~mutex.P();        // make sure no one else~~
                    ~~// using the machine~~
  ~~coke = take a coke out~~
  ~~mutex.V();        // next person's turn~~
  ~~emptyBuffers.V(); // tell producer we're~~
                    ~~// ready for more~~
  ~~return coke;~~
~~}~~
~~}~~
```

~~6.4 Questions~~

~~Why does producer P and V different semaphores than consumer?~~

~~Is order of Ps important?~~

~~Is order of V's important?~~

~~What if we have 2 producers or 2 consumers? Do we need to change anything?~~

~~7. implementing semaphores~~

~~last time: implement locks by turning off interrupts (or test&set)~~

~~Question: how would you implement semaphores? (let's solve problem with the "turning off interrupts" technique:~~

~~Here was lock code:~~

```
~~member variables:~~
    ~~int value~~
    ~~queue *queue;~~

~~Lock::Lock()~~
    ~~value = FREE;~~
    ~~queue = new Queue();~~

~~Lock::Acquire()~~
    ~~disable interrupts~~
    ~~if (value == BUSY)~~
        ~~put thread's TCB on queue of threads~~
        ~~waiting for lock~~
        ~~switch~~
    ~~else~~
        ~~value = BUSY~~
    ~~enable interrupts~~

~~Lock::Release()~~
    ~~disable interrupts~~
    ~~if anyone on wait queue{~~
        ~~take a waiting thread's TCB off queue~~
        ~~put it on ready queue~~
    ~~else~~
        ~~value = FREE;~~
    ~~enable interrupts~~
```

~~Fill in the semaphore code:~~

```
~~Member variables:~~

~~Semaphore::Semaphore()   // constructor~~



~~Semaphore::P()~~
~~//~~
~~// Thread that calls P() should wait for the~~
~~// semaphore to become positive and then~~
~~// decrement it by 1~~
~~//~~



~~Semaphore::V()~~
~~//~~
~~// A thread that calls V() should increment~~
~~// the semaphore by 1, waking up a thread~~
~~// waiting in P() if any~~
~~//~~
```

~~8. Problems with semaphores/Motivation for monitors~~

~~Semaphores a huge step up -- just think of trying to do bounded buffer problem with just loads and stores~~
~~(busy waiting?)~~

~~**3 problems with semaphores**~~
~~**Problem** 1 -- semaphores are dual purpose -- mutex, scheduling constraints~~
~~→ hard to read code~~
~~→ hard to get code right (initial values; order of P() for different semaphores, …)~~

~~**Problem 2** -- Semaphores have "hidden" internal state~~
~~**Problem** 3 -- careful interleaving of "synchronization" and "mutex" semaphores~~

~~→ waiting for a condition is independent of mutex locks (to examine shared variables)~~
~~→ either cleverly define condition to map exactly to semaphore semantics (e.g., "12 buffers so initialize semaphore to 12" what if you don't know ahead of time how many buffers?) OR clever code (interleaving mutex V() with check condition P()) OR both~~

~~idea of monitor -- separate these concerns: use locks for mutex and condition variables for scheduling constraints~~

~~philosophy -- think about Join() example with producer/consumer. Just one line of code to make it work with semaphores, but need to think a bit to convince self it really works -- relying on semaphore to do both mutex (via atomicity) and condition. What happens when you change the code later to, say, give different priorities to different consumers?~~

9. Monitor definition

monitor – a lock and zero or more condition variables for managing concurrent access to shared data

monitor = shared object -- I'll use these terms interchangeably

NOTE: Historically monitors were first a programming language construct, where the monitor lock is automatically acquired on calling any procedure in a C++ class. (Java does something like this – you can specify that certain routines are *synchronized*) Book tends to describe it this way.

But you don't need this – monitors are also a set of programming *conventions* that you should follow when doing thread programming in C or C++ or Javacript or … (or Modula c.f. Birrell): explicit calls to locks and condition variables

I will teach the "manual" version of monitors (and require that you do things manually on the projects) because I want to make sure it is clear what is going on and why.

9.1 Lock

The **lock** provides mutual exclusion to the shared data

Lock::Acquire() -- wait until lock is free, then grab it
Lock::Release() – unlock; wake up anyone waiting in Acquire

Rules for using a lock
- Always acquire before accessing shared data structure
- Always release after finishing with shared data
- Lock is initially free

Simple example: a synchronized list

```
class Queue{
  public:
   add(Item *item);
   Item *remove();
private:
          Lock mutex;
          List list;
}

Queue::add(Item *item){
  mutex.Acquire();          // lock before using shared data
  list.add(item);           // ok to access shared data
  mutex.Release()           // unlock after done w. shared data
}

Item *Queue::remove(){
  Item *ret;

  lock.Acquire();           // lock before using shared data
  if (list.notEmpty()) {    // something on queue remove it
     ret = list.remove();
  }
  else{
       ret = NULL;
  }
  lock.Release();           // unlock after done
  return ret;
}
```

QUESTION: Why "ret"?

Aside:
If you have exceptions (as in Java), another variation is:

```
Foo(){
  try{
    lock.lock();
    …
    return item;
  }
  finally{
    lock.unlock();
  }
```

9.2 2.2 Condition variables

How do we change Queue::remove() to wait until something is on the queue? How do we change Queue::add() to bound number of items in queue (e.g., wait until there is room?)

Logically, want to transition to *waiting* state inside of critical section, but if hold lock when transition to *waiting*, other threads won't be able to get in to add things to queue, to reenable the waiting thread

~~(Recall that for semaphores, we had essentially this problem and we solved it by cleverly doing our "accounting" for synchronization before we grabbed the lock for mutex. This type of subtle reasoning in programs worries me.)~~

Key idea with condition variables: make it possible to transition to *waiting* inside critical section, by **atomically** releasing lock at same time we transition to *waiting*

Condition variable: a queue of threads waiting for something **inside** a critical section

3 operations
Wait() – release lock; transition to *waiting*; reaquire lock

- ♦ releasing lock and transition to *waiting* are atomic

Signal() – wake up a waiter, if any
Broadcast() – wake up all waiters

RULE: must hold lock when doing condition variable operations

In lecture, I'll follow convention: require lock as parameter to condition variable operations. Get in the habit; other systems don't always require this

Some will tell you you can do signal outside of lock. IGNORE THEM. This is only a (small) performance optimization, and it is likely to lead you to write incorrect code.

A synchronized queue with condition variables

```
class Queue{
  ...
    static const int MAX;
  private:
    Lock mutex;
    Cond moreStuff;
    Cond moreRoom;
    List list;
}

Queue::add(Item *item){
 mutex.Acquire();
 while(list.count == Queue::MAX){
    moreRoom.wait(&mutex);
 }
 list.insert(item);
 assert(list.count <= Queue::MAX);
 moreStuff.signal(&mutex);
 mutex.Release();
}

Queue::remove(){
 mutex.Acquire();
 while (list.count == 0){
      moreStuff.wait(&lock); // release lock; go to sleep; require
  }
  ret = list.remove();
  assert(ret != NULL);
  moreRoom.signal(&mutex);
  mutex.Release();
  return ret;
}
```

9.3 Mesa/Hansen v. Hoare monitors

Need to be careful about precise defn of signal and wait

Mesa/Hansen-style: (most real operating systems)
Signaler keeps lock, processor
Waiter simply put on ready queue, with no special priority.
(In other words, waiter may have to **wait** to re-acquire lock)

Hoare-style: (most textbooks)
Signaler gives up lock and CPU to waiter; waiter runs immediately
Waiter gives up lock, processor back to signaler, when it exits critical section or if it waits again

Code above for synchronized queuing happens to work with either style, but for many programs it matters which you are using.

With Hoare-style, can change "while" in RemoveFromQueue to "if" because the waiter only gets woken up if item on the list.
With Mesa-style, waiter may need to wait again after being woken up b/c some other thread may have acquired the lock and removed the item before the original waiting thread gets to the front of the ready queue.

This means that as a general principle, you **always** need to check the condition after the wait, with mesa-style monitors (e.g., use a "while" instead of an "if")

Answer: Hansen
Why (simple): That's what systems have
Why (deeper): That's what is better/right (IMHO)
(1) That's what systems have
(2) more modular -- safety property is local
(3) more flexible

code written to work under Hansen works under Hoare, but not vice versa

(4) spurious wakeups

real implementations (e.g.,, Java, Posix) say that "cond::wait()" can return if (a) cond::signal() is called, (b) cond::broadcast() is called, or (c) other, implementation-specific situations

Always use while(...){cv.wait(*lock);}

Admin – 3 min

Project

10. Implementing CV

Simple uniprocessor implementation:

```
class Cond{
private:
    Queue waiting;

public:
void Cond::Wait(Lock *lock){
    disable interrupts;
    readyList->remove(current TCB);
    waiting.add(current TCB);
    lock->release();
    switch();
    enable interrupts;
    lock->Acquire();
}

void Cond::Signal(Lock *lock){
    disable interrupts;
    if(waiting.notEmpty()){
        TCB enabled = waiting.remove();
        readyList->add(enabled);
    }
    enable interrupts;
}

void Cond::broadcast(Lock *lock){
    disable interrupts;
    while(waiting.notEmpty()){
        TCB enabled = waiting.remove();
        readyList->add(enabled);
    }
    enable interrupts;
}
```

****

Summary - 1 min

****

2 types of synchronization

mutual exclusion
sheduling/waiting
~~semaphore can be used for both (is this good?)~~

~~Semaphore operations~~
~~P()~~
~~V()~~
~~Note: you can't ask the value of a semaphore – only can do P() and V()~~

~~Semaphore built on same hardware primitives as lock using essentially same techniques~~

Monitor = shared object = lock + [CV]* + state