Lecture C5: ~~Semaphores~~ Shared objects, Monitors, Condition Variables, and Bounded buffer

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

- Hardware support for synchronization
- abstractions on top of hardware support (e.g., Lock)
- Shared objects

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

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

****
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 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

Lecture

10. Programming strategy:

(See "Programming with threads" handout for more details)

Goal: Systematic ("cookbook") way to write *easy to read and understand* and *correct* multi-threaded programs

10.1 General approach

1. Decompose problem into objects

object oriented style of programming – encapsulate shared state and synchronization variables inside of objects

Note:

(1) Shared objects are separate from threads

(2) Shared object encapsulates code, synchronization variables, and state variables

Warning: most examples in the book are lazy and talk about "thread 1's code" and "thread 2's code", etc. This is b/c most of the "classic" problems were studied before OO programming was widespread, and the textbooks have not caught up

Hint: don't manipulate synchronization variables or shared state variables in the code associated with a thread, do it with the code associated with a shared object.

Point of possible confusion – in Java, Thread is a class, so Threads are objects. An object of a type that inherits from Thread or implements runnable should **never** have a member variable that is a Lock or Condition; it should **never** say synchronized{}. Why? A thread's state is by definition thread-local state.

Each thread tends to have a "main" loop that accesses shared objects *but the thread object does not include locks or condition variables in its state, and the thread's main loop code does not directly access locks or cv's.*

Locks and CVs are encapsulated in the shared objects.

Why?

(1) Locks are for synchronizing across multiple threads. Doesn't make sense for one thread to "own" a lock!
(2) Encapsulation – details of synchronization are internal details of a shared object. Caller should not know about these details.

"Let the shared objects do the work."

1A. Identify units of concurrency. Make each a thread with a go() method. Write down the actions a thread takes at a high level.

1b. Identify shared chunks of state. Make each shared *thing* an object. Identify the methods on those objects – the high-level actions made by threads on these objects.

1C. Write down the high-level main loop of each thread.

Advice: stay high level here. Don't worry about synchronization yet. Let the objects do the work for you.

Separate threads from objects. The code associated with a thread should not access shared state directly (and so there should be no access to locks/condition variables in the "main" procedure for the thread.) Shared state *and synchronization* should be encapsulated in shared objects.

Now, for each object:

2. Write down the synchronization constraints on the solution. Identify the type of each constraint: *mutual exclusion* or *scheduling*

3. Create a lock or condition variable corresponding to each constraint

4. Write the methods, using locks and condition variables for coordination

10.2 Coding standards/style

These are **required standards** in class. See the handout for details!

I taught m/t coding the standard way...
-- I explained locks give mutual exclusion...
-- I explained how condition variables work; how they are related to the shared state; Hoare v. Hansen, ...

Fall 2001 midterm:

- Every program with incorrect semantic behavior violated at least one rule
- >90% of programs that violated at least one rule were "obviously" semantically incorrect (that is, I could see the bug within seconds of looking at the program; there may have been additional bugs…)

- o All that violate one rule are *wrong* – they are harder to read, understand, maintain, …
- o Since I've declared "violating rule is *wrong*", huge reduction in bugs in exams and projects

Passion for these rules goes deeper.
I learned m/t coding the standard way...

These two experiences + this is really important --> I am a zealot...

The rules: (See handout)

1. Always do things the same way

2. Always use monitors (condition variables + locks)

   Almost always more clear than semaphores + "always do things the same way"

3. Always hold lock when operating on a condition variable

   You signal on a condition variable because you just got done manipulating shared state. You proceed when some condition about a shared state becomes true. Condition variables are useless without shared state and shared state is useless without holding a lock.

4. Always grab lock at beginning of procedure and release it right before return

   - Simplifies reading your code ("always do things the same way")
   - If you find yourself wanting to release lock in middle of a procedure, 99% of time code would be more clear if you split it into two procedures

5. Always use

```
  while(predicateOnStateVariables(...) ==
true/false){
        condition->wait(&lock);
  }
not
   if(...){…
```

(Where `PredicateOnStateVariables(...)` looks at the state variables of the current object to decide if it is OK to proceed.)

`While` works any time `if` does, and it works in situations when `if` doesn't. By rule 1, you should do things the same way every time.

`If` breaks modularity

When you always use ***while***, you are given incredible freedom about where you put the signal()'s. In fact, signal() becomes a *hint* -- you can add more signals to a correct program in arbitrary places and it remains a correct program!
→ Can determine correctness of signal calls and wait calls locally

6. (Almost) never `sleep()`

Never use sleep() to wait for another thread to do something. The correct way to wait for a condition to become true is to wait() on a condition variable.

sleep() is only appropriate when there is a particular real-world moment in time when you want to perform some action. If you catch yourself writing {\tt while(some condition)\{sleep();\}}, treat this is a big red flag that you are probably making a mistake.

I'm sure there are valid exceptions to all of the above rules, but they are few and far between. And the benefit you get by occasionally breaking the rules is unlikely to make up for the cost in your effort, extra debugging and maintenance cost, and loss of modularity.

10.3 Java rules

In some years, we use Java for the project. Java is a modern language with supports for threads from day 1. This is mostly good news. 2 issues:

(1) For production use: Support for some dangerous/undesirable constructs/styles of programming
(2) For teaching: "too much" support for multi-threading → someone can write code that invokes synchronization with our without knowing what's going on

→ Coding standards for this class
(J1) Do not use synchronized blocks within method

This is a specific incarnation of rule (4) above "Always grab locks at beginning and release at the end"

The following is forbidden:

```
Foo(){
        …
        synchronized(this){
                …
        }
        …
}
```

Instead, move the synchronized block into its own method.

(J2) Cleanly separate *Threads* from *shared objects*

Classes that define Threads (e.g., that extend Thread or implement Runnable) should include per-thread state. They should not include shared state. They should not include locks or condition variables.

The model is threads operate on shared state (picture).

(J3) *For this class* the *synchronized* keyword is forbidden. Instead, explicitly allocate and invoke locks and condition variables.

The purpose of this rule is to make it easier to teach and learn how to think about synchronization.

Example (correct):

```
class Foo{
        SimpleLock lock;
        Condition c1;
        Condition c2;

        public Foo(){
                lock = new SimpleLock();
                c1 = lock.newCondition();
                c2 = lock.newCondition();
                …
        }

        public void doSomething(…){
                try{
                        lock.lock();
                        …
                        while(…){
                                c1.awaitUninterruptably();
                        }
                        …
                        c2.signal();
                }
                finally{
                        lock.unlock();
                }
        }
}
```

Example (acceptable):

```
class Foo{
        SimpleLock lock;
        Condition c1;
        Condition c2;

        public Foo(){
                lock = new SimpleLock();
                c1 = lock.newCondition();
                c2 = lock.newCondition();
                …
        }

        public void doSomething(…){
                lock.lock();
                …
                while(…){
                        c1.awaitUninterruptably();
                }
                …
                c2.signal();
                lock.unlock();
        }
}
```

Example (forbidden for this class; often correct in real world):

```
class Foo{

        public Foo(){
                …
        }

        public synchronized void doSomething(…){
                …
                while(…){
                        this.wait();
                }
                …
                this.signal();
        }

}
```

(Note that once you leave this class the above style can be used when an object needs one lock and one condition variable; if you need two condition variables, fall back on the manual version as in this class.)

10.4 D. Example/Basic template:

(1,2) Always use condition variables for code you write.
Be able to *understand* code written in semaphores. But the coding standard your manager (me) is enforcing for this group is condition variables for synchronization

```
class Foo{

private:
// Synchronization variables
Lock mutex;
Cond condition1;
Cond condition2;
…
```

```
    // State variables
    …

    public:
    Foo::foo()
    {
     /*
* (#4) Always, grab mutex at start of procedure, release at
* end (or at any return!!!). Reasoning: if there is a logical
* set of actions to do when you hold a mutex, that logical
* set of actions should be expressed as a procedure, right?
*/
      mutex->acquire(){
        Assert(invariants hold – shared variables in consistent state)
          …
           invariants may or may not hold; shared variables may be
           in inconsistent state



          …
     // (#5)always "while" never "if"
           while(shared variables in some state){
                assert(invariants hold)
                // (#3) Always hold lock when operating on C.V.
                condition1->wait(&mutex)
                assert(invariants hold)
           }
          …
           invariants may or may not hold; shared variables may be
           in inconsistent state
          …
          … // (#3) Always hold lock when operating on C.V.
          …condition2->signal(&mutex);
          …condition1->signal(&mutex);
          …
          Assert(invarients hold)
     }mutex->release()
    }
```

11. }; // Class

12.

13. Rule (#6) (Almost) never sleep()

Sleep(time) puts the current thread on a waiting queue at the timer – only use it to wait until a specific time, not to wait for an event of a different sort
Hint: sleep should never be in a while(…){sleep}
Problems with using sleep:

1) no atomic release/reacquire lock
2) really inefficient (example – cascading sleeps in Aname)
3) not logical

Warning: on the project and on exams, improper use of sleep will be regarded as strong evidence that you have no idea how to write multi-threaded programs and will affect your grade accordingly.
(I make this a point of emphasis b/c this error is so common in past years and easy to avoid.)

Aside: Double checked locking is broken example...

****
Summary - 1 min
****

Monitors represent the logic of the program. Wait if necessary, signal if change something so waiter might need to wake up.

```
mutex->lock
while (need to wait)
        cv->wait();
mutex->unlock

mutex->lock
do something so no need to wait
cv->signal();
mutex->unlock
```

14. 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;
}
```

****
Lecture - 20 min
****

15. Readers/Writers

15.1 Motivation

Shared database (for example, bank balances, or airline seats)

Two classes of users:
Readers – never modify database
Writers – read and modify data

Using a single mutex lock would be overly restrictive.
Instead, want:
    many readers at same time
    only one writer at same time

15.2 Constraints

Notice: for every constraint, there is a synchronization variable.
This time different types for different purposes.

1) Reader can access database when no writers (Condition okToRead)
2) Writers can access database when no readers or writers (condition okToWrite)
3) Only one thread manipulates shared variables at a time (mutex)

15.3 Solution

Basic structure

```
Database::read()
    check in -- wait until no writers
    access database
    check out – wake up waiting writer

Database::write()
```

```
        check in -- wait until no readers or writers
        access database
        check out – wake up waiting readers or writers
```

State variables:

```
    AR = 0;  // # active readers
    AW = 0; // # active writers
    WR = 0; // # waiting readers
    WW = 0; // # waiting writers

    Condition okToRead = NIL;
    Condition okToWrite = NIL;
    Lock lock = FREE;
```

Code:

```
    Database::read(){
        startRead();  // first, check self into the system
        Access Data
        doneRead();  // Check self out of system
    }

    Database::startRead(){
        lock.Acquire();
        while((AW + WW) > 0){
            WR++;
            okToRead.Wait(&lock);
            WR--;
        }
        AR++;
        lock.Release();
    }

    Database::doneRead(){
        lock.Acquire();
        AR--;
        if(AR == 0 && WW > 0){ // if no other readers still
           okToWrite.Signal(); // active, wake up writer
        }
        lock.Release();
    }
```

```
Database::write(){  // symmetrical
        startWrite();   // check in
        accessData
        doneWrite();  // check out
}

Database::startWrite(){
        lock.Acquire();
        while((AW + AR) > 0){ // check if safe to write
                                // if any readers or writers, wait
                WW++;
                okToWrite->Wait(&lock);
                WW--;
        }
        AW++;
        lock.Release();
}

Database::doneWrite(){
        lock.Acquire();
        AW--;
        if(WW > 0){
                okToWrite->Signal(); // give priority to writers
        }
        else if (WR > 0){
                okToRead->Broadcast();
        }
        lock.Release();
}
```

Question

1) Can readers starve?
2) Why does checkRead need a while?
3) Suppose we had a large DB with many records, and we want many users to access it at once. Probably want to allow two different people to update their bank balances at the same time, right? What are issues?

16. Example: Sleeping Barber (Midterm 2002)

The shop has a barber, a barber chair, and a waiting room with NCHAIRS chairs. If there are no customers present, the barber sits in the barber chair and falls asleep. When a customer arrives, he wakes the sleeping barber. If an additional customer arrives while the barber is cutting hair, he sits in a waiting room chair if one is available. If no chairs are available, he leaves the shop. When the barber finishes cutting a customer's hair, he tells the customer to leave; then, if there are any customers in the waiting room he announces that the next customer can sit down. Customers in the waiting room get their hair cut in FIFO order.

The barber shop can be modeled as 2 shared objects, a BarberChair with the methods napInChair(), wakeBarber(), sitInChair(), cutHair(), and tellCustomerDone(). The BarberChair must have a state variable with the following states: EMPTY, BARBER_IN_CHAIR, LONG_HAIR_CUSTOMER_IN_CHAIR, SHORT_HAIR_CUSTOMER_IN_CHAIR. Note that neither a customer or barber should sit down until the previous customer is out of the chair (state == EMPTY). Note that cutHair() must not return until the customer is sitting in the chair (LONG_HAIR_CUSTOMER_IN_CHAIR). And note that a customer should not get out of the chair (e.g., return from sit in chair) until his hair is cut (SHORT_HAIR_CUSTOMER_IN_CHAIR). The barber should only get in the chair (BARBER_IN_CHAIR) if no customers are waiting. **You may need additional state variables.**

The WaitingRoom has the methods enter() which immediately returns WR_FULL if the waiting room is full or (immediately or eventually) returns MY_TURN when it is the caller's turn to get his hair cut, and it has the method callNextCustomer() which returns WR_BUSY or WR_EMPTY depending on if there is a customer in the waiting room or not. Customers are served in FIFO order.

Thus, each customer thread executes the code:

```
Customer(WaitingRoom *wr, BarberChair *bc)
{
    status = wr->custEnter();
    if(status == WR_FULL){
        return;
    }
    bc->wakeBarber();
    bc->sitInChair();  //  Wait for chair to be EMPTY
                        // Make state LONG_HAIR_CUSTOMER_IN_CHAIR
                       // Wait until SHORT_HAIR_CUSTOMER_IN_CHAIR
                       // then make chair  EMPTY and return
    return;
}
```

The barber thread executes the code:

```
Barber(WaitingRoom *wr, BarberChair *bc)
{
    while(1){          // A barber's work is never done
        status = wr->callNextCustomer();
        if(status == WR_EMPTY){
            bc->napInChair(); // Set state to BARBER_IN_CHAIR; return with state EMPTY
        }
        bc->cutHair(); // Block until LONG_HAIR_CUSTOMER_IN_CHAIR;
                       // Return with SHORT_HAIR_CUSTOMER_IN_CHAIR
        bc->waitCustomerDepart(); // Return when EMPTY
    }
```

}

Write the code for the WaitingRoom class and the BarberChair class. Use locks and condition variables for synchronization and follow the coding standards specified in the handout.

Hint and requirement reminder: remember to start by asking for each method "when can a thread wait?" and writing down a synchronization variable for **each** such situation.

List the member variables of class **WaitingRoom** including their type, their name, and their initial value

Type	Name	Initial Value (if applicable)
mutex	*lock*	
cond	*canGo*	
int	*nfull*	*0*
int	*ticketAvail*	*0*
int	*ticketTurn*	*-1*

```
int WaitingRoom::custEnter()
    lock.acquire();
    int ret;
    if(nfull == NCHAIRS){
        ret = WR_FULL;
    }
    else{
        ret = MY_TURN;
        myTicket = ticketAvail++;
        nfull++;
        while(myTicket > ticketTurn){
                canGo.wait(&lock);
        }
        nfull--;
    }
    lock.release();
    return ret;

int WaitingRoom::callNextCustomer()
    lock.acquire();
    if(nfull == 0){
        ret = EMPTY;
    }
    else{
        ret = BUSY;
        ticketTurn++;
        canGo.broadcast();
    }
    lock.release();
    return ret;
```

List the member variables of class **BarberChair** including their type, their name, and their initial value

Type	Name	Initial Value (if applicable)
mutex	*lock*	
cond	*custUp*	
cond	*barberGetUp*	
cond	*sitDown*	
cond	*seatFree*	
cond	*cutDone*	
int	*state*	*EMPTY*
int	*custWalkedIn*	*0*

```
void BarberChair::napInChair()
    lock.acquire();
    if(state == EMPTY){ // Cust could arrive before I sit down
      state = BARBER_IN_CHAIR;

      while(custWalkedIn == 0){
         barberGetUp.wait(&lock);
      }
      state = EMPTY
      seatFree.signal(&lock);
    }
    lock.release();

void BarberChair::wakeBarber()
    lock.acquire();
    custWalkedIn = 1;
    barberGetUp.signal(&lock);
    lock.release()

void BarberChair::sitInChair()
    lock.acquire()
    while(state != EMPTY){
      seatFree.wait(&lock);
    }
    custWalkedIn = 0;
    state = LONG_HAIR_CUSTOMER_IN_CHAIR;
    sitDown.signal(&lock);
    while(state != SHORT_HAIR_CUSTOMER_IN_CHAIR){
      cutDone.wait(&lock);
    }
    state = EMPTY;
    custUp.signal(&lock);
    lock.release();
}


void BarberChair::cutHair()
    lock.acquire();
    while(state != LONG_HAIR_CUSTOMER_IN_CHAIR){
        sitDown.wait(&lock);
    }
    state = SHORT_HAIR_CUSTOMER_IN_CHAIR;
    cutDone.signal(&lock);
    lock.release();
```

```
void BarberChair::waitCustomerDepart()
    lock.acquire();
    while(state != EMPTY){  // NOTE: No other cust can arrive until I call call_next_cust()
      custUp.wait(&lock);
    }

    lock.release();
```

17. Semaphores v. Condition variables

Illustrate the difference by considering: can we build monitors out of semaphores? After all, semaphores provide atomic operations and queuing.

Does this work:

```
Wait(){ semaphore->P() }
Signal{ semaphore->V()}
```

No: Condition variables only work inside a lock. If try to use semaphores inside a lock, have to watch for deadlock.

Does this work:

```
Wait(Lock *lock){
    lock->Release();
    semaphore->P();
    lock->Acquire();
}

Signal(){
    semaphore->V();
}
```

Condition variables have no history, but semaphores do have history.

What if thread signals and no one is waiting?
    → No Op
What if thread later waits?
    → Thread waits.

What if thread V's and no one is waiting?
    Increment
What if thread later does P
    Decrement and continue

In other words, P+V are commutative – result is the same no mater what order they occur. Condition variables are not commutative. That's why they must be in a critical section – need to access state variables to do their job.

Does this fix the problem?

```
Signal(){
    if semaphore queue is not empty
        semaphore->V();
}
```

For one, not legal to look at contents of seemaphore queue.
Also, race condition – signaller can slip in after lock is released and before wait. Then waiter never wakes up

Need to release lock and go to sleep atomically.

Is it possible to implement condition variables using semaphores?
Yes, but exercise left to the reader!

****

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