Operating Systems

Lecture 11: Semaphore & Monitor

Department of Computer Science & Technology
Tsinghua University

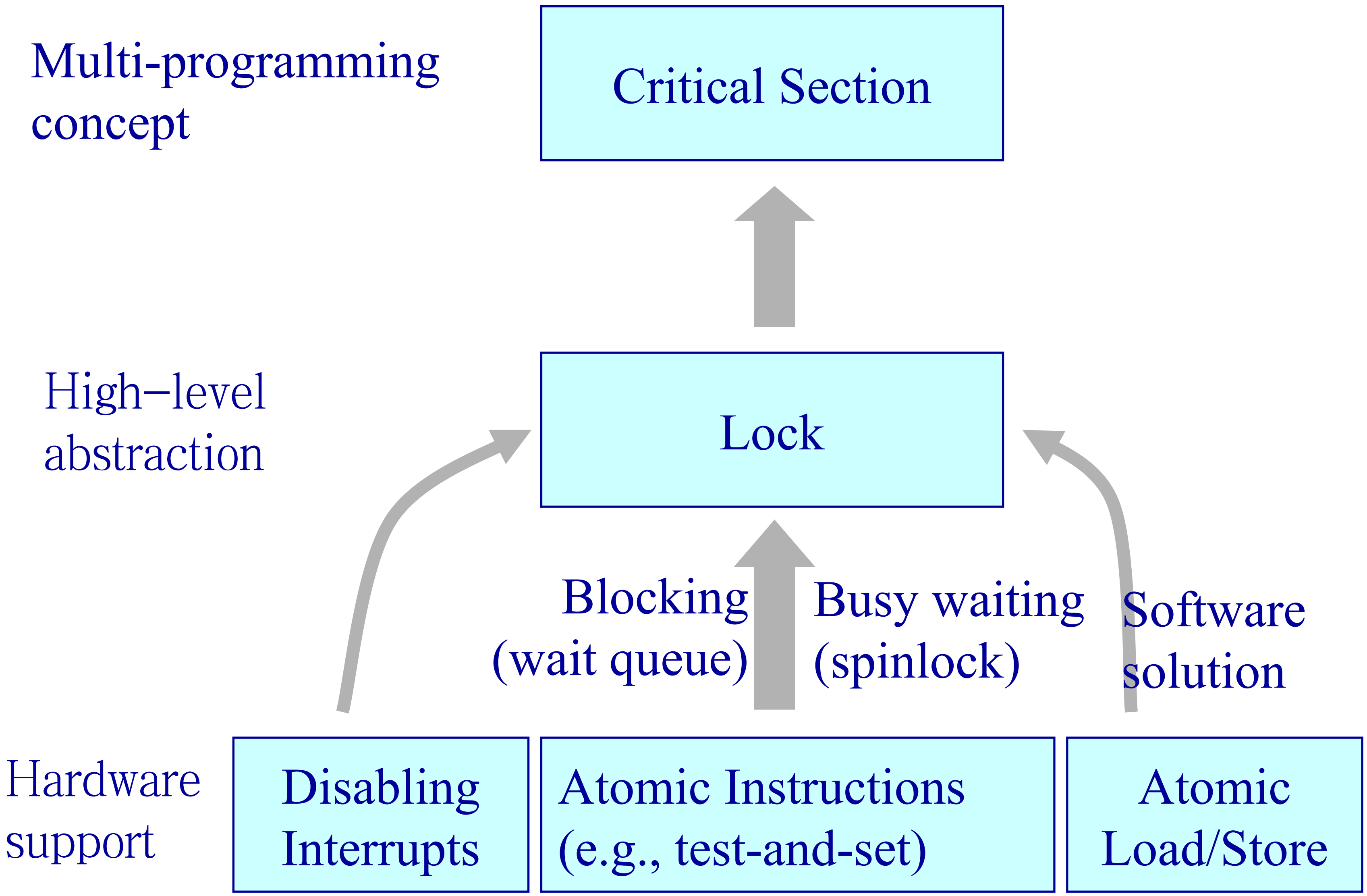
Outline

- Background
- Semaphore
- Using Semaphore
- Implementing Semaphore
- Monitor
- Classical Synchronization Problems

Recap for last week

- The concurrency problem: race condition
  - Big problem in concurrent multi-programming
- Synchronization
  - Coordinating execution of multiple threads that share common data
  - Include mutual exclusion and conditional synchronization
  - Mutual exclusion: only one thread can execute a critical section at a time
- Too difficult to get synchronization right?
  - Need high-level programming abstractions (e.g., Lock)
  - Build them from low-level hardware supports

Mutual Exclusion



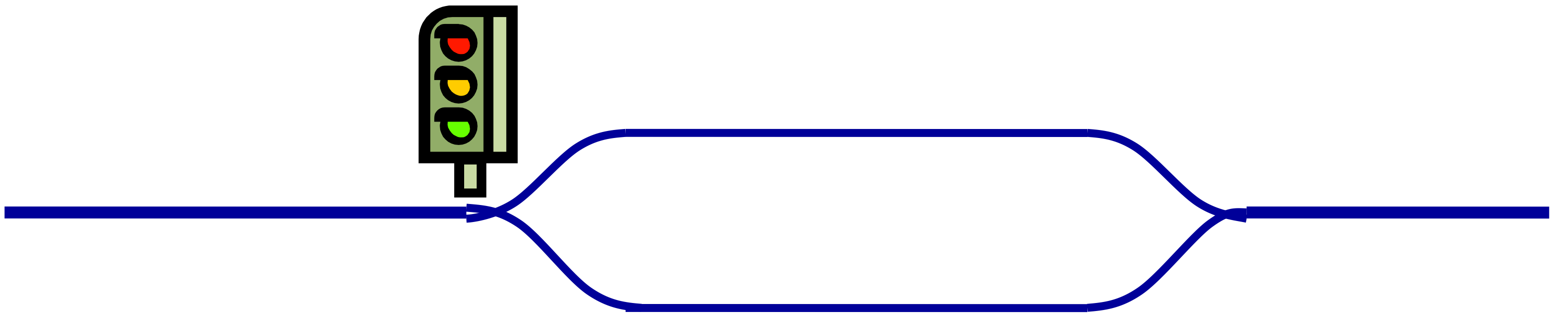
Outline

- Background
- Semaphore
- Using Semaphore
- Implementing Semaphore
- Monitor
- Classical Synchronization Problems

Another High-Level Abstraction: Semaphore

- Abstract data type
  - A integer (sem), with two atomic operations
  - P(): decreases sem by 1, if sem<0, then waits, otherwise continues
  - V(): increases sem by 1, if sem<=0,then wakes up a waiting P if any

- Semaphore from railway analogy
  - Here is a semaphore initialized to 2 for resource control:



Historical Perspective for Semaphores

- Introduced by Dijkstra in 1960s
  - V: Verhoog (Dutch for increase)
  - P: Prolaag (Dutch short for "Probeer te Verlagen", or try to decrease)
- Main synchronization primitives in early OSes
  - For example, original Unix
  - Much less used now (but still very important in computer science study)

Some Important Properties of Semaphores

- Semaphores are integers
- Semaphores are protected variables
  - After initialization, only way you can change the value of a semaphore is through P() and V()
  - Operations must be atomic
- P() can block, V() never blocks
- We assume a semaphore is "fair"
  - No thread that is blocked on P() remains blocked if V() is invoked infinitely often (on the same semaphore)
  - In practice, FIFO is mostly used

Spinlock can be in FIFO style?

More about Semaphores

- Two types of semaphores
  - Binary semaphores: can either be 0 or 1
  - General/Counting semaphores: can take any non-negative value
  - Both are as expressive (given one can implement the other)
- Semaphores can be used both for
  - Mutual exclusion
  - Conditional synchronization (scheduling constraints – one thread waiting for something to happen in another thread)

Outline

- Background
- Semaphore
- Using Semaphore
- Implementing Semaphore
- Monitor
- Classical Synchronization Problems

Using Semaphores for Mutual Exclusion

- Use a binary semaphore for mutual exclusion

```
mutex = new Semaphore(1);
```

```
mutex->P();
    …
Critical Section;
    …
mutex->V();
```

Semaphores for Conditional Synchronization

- Use a binary semaphore for scheduling constraints

```
condition = new Semaphore(0);
```

Thread A

```
…
condition->P();
    …
```

Thread B

```
…
condition->V();
    …
```

- P() is to wait, V() is to signal

Conditional Synchronization

- One thread waits for some other thread to do something
  - Like produced something, or consumed something, …
  - Mutual exclusion (locking) is not sufficient
- Example: the bounded buffer producer-consumer problem
  - One or more producers are generating data and placing these in a buffer
  - A single consumer is taking items out of the buffer one at time
  - Only one producer or consumer may access the buffer at any one time

Semaphores in Producer-Consumer Problem

- Correctness requirements
  - Only one thread manipulates the buffer at any time (mutual exclusion)
  - Consumer must wait for producer when buffer is empty (scheduling/synchronization constraint)
  - Producer must wait for the consumer when buffer is full (scheduling/synchronization constraint)
- Use a separate semaphore for each constraint
  - Binary semaphore mutex
  - General semaphore fullBuffers
  - General semaphore emptyBuffers

Producer-Consumer Problem using Semaphore

```
Class BoundedBuffer {
    mutex = new Semaphore(1);
    fullBuffers = new Semaphore(0);
    emptyBuffers = new Semaphore(n);
}
```

```
BoundedBuffer::Deposit(c) {
    emptyBuffers->P();
    mutex->P();
    Add c to the buffer;
    mutex->V();
    fullBuffers->V();
}
```

```
BoundedBuffer::Remove(c) {
    fullBuffers->P();
    mutex->P();
    Remove c from buffer;
    mutex->V();
    emptyBuffers->V();
}
```

- Does the order of P and V matter?

Outline

- Background
- Semaphore
- Using Semaphore
- Implementing Semaphore
- Monitor
- Classical Synchronization Problems

Implementing Semaphore

- Using hardware primitives
  - Disabling interrupts
  - Atomic instruction (test-and-set)
- Similar to locks
- Example: using disabling interrupts

```
classSemaphore {
int sem;
WaitQueue q;
}
```

```
Semaphore::P() {
   sem--;
  if (sem < 0) {
      Add this TCB to q;
      block(p);
   }
}
```

```
Semaphore::V() {
   sem++;
   if (sem<=0) {
      Remove a thread t from q;
      wakeup(t);
   }
}
```

P primitive: sem_wait

```
sem_wait (semaphore *S) {// Must be executed atomically
        S->value--;
        if (S->value < 0) {
                add this process to S->tlist;
                block();
        }
}
```

- int down_interruptible(struct semaphore *sem)
  - http://lxr.linux.no/linux+v3.3.6/kernel/semaphore.c#L75
- int down_killable(struct semaphore *sem)
  - http://lxr.linux.no/linux+v3.3.6/kernel/semaphore.c#L101
- static inline int __sched __down_common(struct semaphore *sem, long state, long timeout)
  - http://lxr.linux.no/linux+v3.3.6/kernel/semaphore.c#L204

V primitive: sem_wait

```
sem_signal (semaphore *S) {// Must be executed atomically
        S->value++;
        if (S->value <= 0) {
                remove thread t from S->tlist;
                wakeup(t);
        }
}
```

- void up(struct semaphore *sem)
  - http://lxr.linux.no/linux+v3.3.6/kernel/semaphore.c#L178
- static noinline void __sched __up(struct semaphore *sem)
  - http://lxr.linux.no/linux+v3.3.6/kernel/semaphore.c#L256
- int wake_up_process(struct task_struct *p)
  - http://lxr.linux.no/linux+v3.2/kernel/sched.c#L2929
- static int try_to_wake_up(struct task_struct *p, unsigned int state, int wake_flags)
  - http://lxr.linux.no/linux+v3.2/kernel/sched.c#L2821

The Problem with Semaphores

- Semaphores are used for dual purpose
  - Mutual exclusion and conditional synchronization
  - But waiting for condition is independent of mutual exclusion
- Difficult to read/develop code
  - Programmer needs to be clever about using semaphores
- Easy mistakes
  - Take a semaphore that is already held in same thread
  - Forget to release a semaphore
- Inadequate in dealing with deadlocks

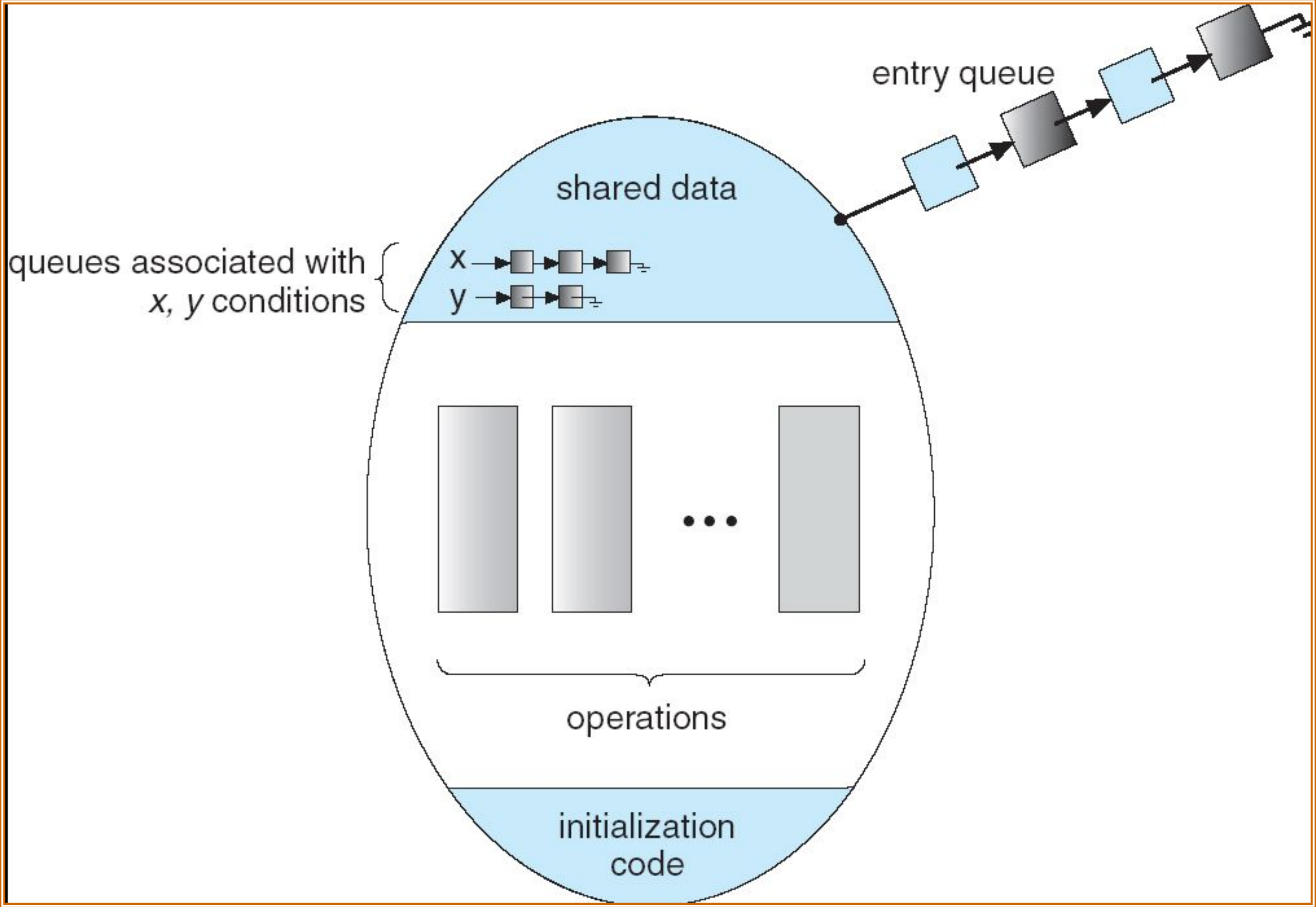
Outline

- Background
- Semaphore
- Using Semaphore
- Implementing Semaphore
- Monitor
- Classical Synchronization Problems

Introducing Monitor

- Purpose: separate the concerns of mutual exclusion and conditional synchronization
- What is a monitor?
  - One Lock: specify critical section
  - zero or more Condition variables: wait/signal inside critical section for managing concurrent access to shared data
- General Approach
  - Collect related shared data into an object/module
  - Define methods for accessing the shared data

Monitor with Condition Variables



Locks and Condition Variables

- Lock
  - Lock::Acquire() – wait until lock is free, then grab it
  - Lock::Release() – release the lock, wake up a waiter if any

- Condition Variable
  - Enable waiting inside a critical section
    - Allow threads to wait (sleep) inside a critical section
    - Does so by atomically releasing lock at time to go to sleep
  - Wait() operation
    - Release lock, go to sleep (block), re-acquire lock upon return
  - Signal() operation (or broadcast() operation)
    - Wake up a waiter (or all waiters), if any

Implementing Conditional Variables

- Implementation
  - Requires a per-condition variable queue to be maintained
  - Threads waiting for the condition wait for a signal()

```
Class Condition {
    int numWaiting = 0;
    WaitQueue q;
}
```

```
Condition::Wait(lock){
    numWaiting++;
    Add this TCB to q;
    lock->release();
    schedule();
    lock->acquire();
}
```

```
Condition::Signal(){
    if (numWaiting > 0) {
        Remove a thread t from q;
        wakeup(t);
        numWaiting--;
    }
}
```

Example Monitor: Producer-Consumer Problem

```
classBoundedBuffer {
    …
    Lock lock;
    int count = 0;
    Condition notFull, notEmpty;
}
```

```
BoundedBuffer::Deposit(c) {
    lock->Acquire();
    while (count == n)
        notFull.Wait(&lock);
    Add c to the buffer;
    count++;
    notEmpty.Signal();
    lock->Release();
}
```

```
BoundedBuffer::Remove(c) {
    lock->Acquire();
    while (count == 0)
        notEmpty.Wait(&lock);
    Remove c from buffer;
    count--;
    notFull.Signal();
    lock->Release();
}
```

Monitor: Two Styles

- Hansen-style (most real OSes, or Java, Mesa)

```
l.acquire()
…
x.wait()
```

T1 blocks

T2 starts

```
l.acquire()
…
x.signal()
…
l.release()
```

T2 finishes

```
…
l.release()
```

T1 resumes

- Hoare-style (most textbooks)

```
l.acquire()
…
x.wait()
```

T1 blocks

T2 starts

```
l.acquire()
…
x.signal()
```

T2 blocks

```
…
l.release()
```

T1 resumes

T1 finishes

T2 resumes

```
…
l.release()
```

Hansen Monitors versus Hoare Monitors

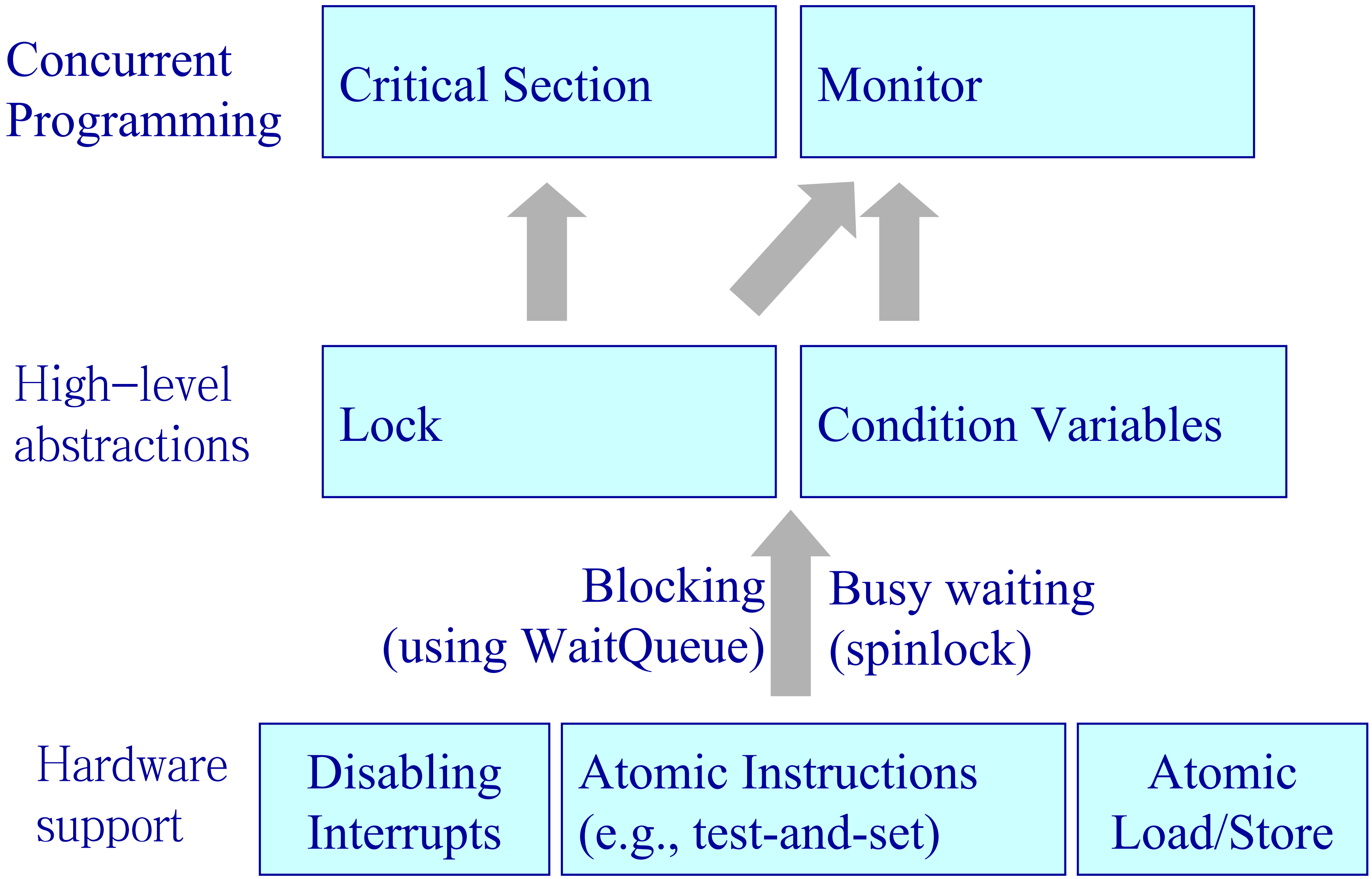
- Hansen-style
  - Signal is only a "hint" that the condition may be true
  - Need to check again
- Benefits
  - Efficient implementation

- Hoare-style
  - Cleaner, good for proofs
  - When a condition variable is signaled, it does not change
- But
  - Inefficient implementation

```
Hansen-style :Deposit(){
lock!acquire();
while (count == n) {
notFull.wait(&lock); }
Add  thing;
count++;
notEmpty.signal();
lock!release();
}
```

```
Hoare-style: Deposit(){
lock!acquire();
if (count == n) {
notFull.wait(&lock); }
Add thing;
count++;
notEmpty.signal();
lock!release();
}
```

Synchronization Summary



Concurrent Programming Summary

- Developing/debugging concurrent programs is hard
  - Non-deterministic interleaving of instructions
- Synchronization constructs
  - Locks: mutual exclusion
  - Condition variables: conditional synchronization
  - Other primitives: semaphores
- How can you use these constructs effectively?
  - Develop and follow strict programming style/strategy

Outline

- Background
- Semaphore
- Using Semaphore
- Implementing Semaphore
- Monitor
- Classical Synchronization Problems

Classical Synchronization Problems

- The bounded buffer producer-consumer problem
- The readers-writers problem
- The dining philosophers problem
- The sleeping barber problem

Readers/Writers: A Complete Example

- Motivation
  - Shared databases accesses
- Two types of users
  - Readers: Never modify data
  - Writers: read and modify data
- Problem constraints
  - Allow multiple readers at the same time, but only one writer at any time
  - Readers can access data when there are no writers
  - Writers can access data when there are no readers/writers
  - Only one thread can manipulate shared variables at any time

Readers/Writers: Using Semaphore

- A data set is shared among a number of concurrent processes
  - Readers – only read the data set; they do not perform any updates
  - Writers – can both read and write.

- Shared Data
  - Data set
  - Semaphore CountMutex initialized to 1.
  - Semaphore WriteMutex initialized to 1.
  - Integer Rcount initialized to 0.

Readers/Writers: Using Semaphore (Cont.)

Writer

```
sem_wait(WriteMutex);

   write;

sem_post(WriteMutex);
```

Reader

```
sem_wait(CountMutex);
  if (Rcount == 0)
     sem_wait (WriteMutex);
  ++Rcount;
sem_post(CountMutex);

read;

sem_wait(CountMutex);
  --Rcount;
  if (Rcount == 0)
    sem_post (WriteMutex);
sem_post(CountMutex)
```

Readers/Writers Problem: Writer Have Priority

```
/* program readersandwriters */
int  readcount, writecount;
semaphore x = 1, y = 1, z = 1, wsem = 1, rsem = 1;
void reader()
{
    while (true)
    {
     semWait (z);
          semWait (rsem);
               semWait (x);
                    readcount++;
                    if (readcount == 1)
                         semWait (wsem);
               semSignal (x);
          semSignal (rsem);
     semSignal (z);
     READUNIT();
     semWait (x);
          readcount--;
          if (readcount == 0)
               semSignal (wsem);
     semSignal (x);
    }
}
void writer ()
{
    while (true)
    {
     semWait (y);
          writecount++;
          if (writecount == 1)
               semWait (rsem);
     semSignal (y);
     semWait (wsem);
     WRITEUNIT();
     semSignal (wsem);
     semWait (y);
          writecount--;
          if (writecount == 0)
               semSignal (rsem);
     semSignal (y);
    }
}
void main()
{
    readcount = writecount = 0;
    parbegin (reader, writer);
}
```

Readers/Writer: Using Monitors

- Basic structure: two methods

```
Database::Read() {
    Wait until no writers;
    Access database;
    check out – wake up waiting writers;
}
```

```
Database::Write() {
    Wait until no readers/writers;
    Access database;
    check out – wake up waiting readers/writers;
}
```

- State variables

```
AR = 0;          // # of active readers
AW = 0;          // # of active writers
WR = 0;          // # of waiting readers
WW = 0;          // # of waiting writers
Condition okToRead;
Condition okToWrite;
Lock lock;
```

Solution Details: Readers

```
AR = 0;          // # of active readers
AW = 0;          // # of active writers
WR = 0;          // # of waiting readers
WW = 0;          // # of waiting writers
Condition okToRead;
Condition okToWrite;
Lock lock;
```

```
Public Database::Read() {
    StartRead();
    Access database;
    DoneRead();
}
```

```
Private Database::StartRead() {
    lock.Acquire();
    while ((AW+WW) > 0) {
        WR++;
        okToRead.wait(&lock);
        WR--;
    }
    AR++;
    lock.Release();
}
```

```
Private Database::DoneRead() {
lock.Acquire();
    AR--;
    if (AR ==0 && WW > 0) {
        okToWrite.signal();
    }
lock.Release();
}
```

Solution Details: Writers

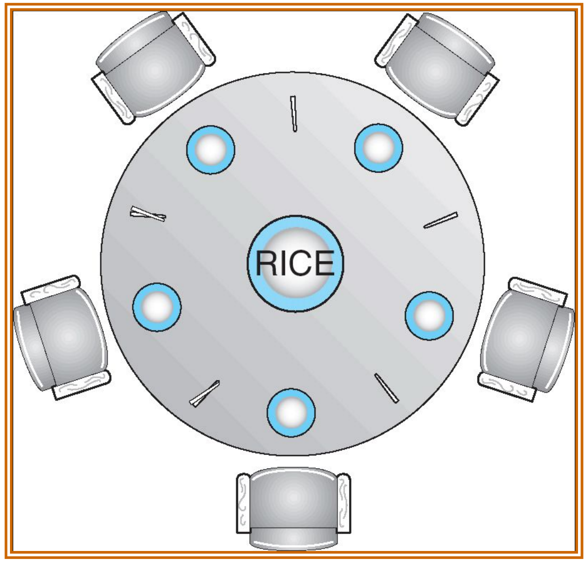
```
AR = 0;          // # of active readers
AW = 0;          // # of active writers
WR = 0;          // # of waiting readers
WW = 0;          // # of waiting writers
Condition okToRead;
Condition okToWrite;
Lock lock;
```

```
Public Database::Write() {
    StartWrite();
    Access database;
    DoneWrite();
}
```

```
Private Database::StartWrite() {
lock.Acquire();
    while ((AW+AR) > 0) {
        WW++;
        okToWrite.wait(&lock);
        WW--;
    }
    AW++;
lock.Release();
}
```

```
Private Database::DoneWrite() {
lock.Acquire();
    AW--;
    if (WW > 0) {
        okToWrite.signal();
    }
    else if (WR > 0) {
        okToRead.broadcast();
    }
lock.Release();
}
```

Dining-Philosophers Problem



- Shared data
  - Bowl of rice (data set)
  - Semaphore chopstick [5] initialized to 1

Dining-Philosophers Problem (Cont.)

- The structure of Philosopher *i*:

```
Do  {
      wait ( chopstick[i] );
      wait ( chopStick[ (i + 1) % 5] );

          //  eat

      signal ( chopstick[i] );
      signal (chopstick[ (i + 1) % 5] );

         //  think

} while (true) ;
```

Solution to Dining Philosophers

```
void test (int i) {
        if ( (state[(i + 4) % 5] != EATING) &&
        (state[i] == HUNGRY) &&
        (state[(i + 1) % 5] != EATING) ) {
            state[i] = EATING ;
             self[i].signal () ;
         }
    }

    initialization_code() {
        for (int i = 0; i < 5; i++)
        state[i] = THINKING;
}
}
```

Solution to Dining Philosophers (cont)

```
monitor DP
  {
    enum { THINKING; HUNGRY, EATING} state [5] ;
    condition self [5];

    void pickup (int i) {
           state[i] = HUNGRY;
           test(i);
           if (state[i] != EATING) self [i].wait;
    }

     void putdown (int i) {
           state[i] = THINKING;
                 // test left and right neighbors
            test((i + 4) % 5);
            test((i + 1) % 5);
      }
```

Sleeping Barber Problem

- There is one barber, and n chairs for waiting customers
- If there are no customers, then the barber sits in his chair and sleeps
- When a new customer arrives and the barber is sleeping, then he will wakeup the barber
- When a new customer arrives, and the barber is busy, then he will sit on the chairs if there is any available, otherwise (when all the chairs are full) he will leave.

Barber Shop Hints

Consider the following:

- Customer threads should invoke a function named getHairCut.
- If a customer thread arrives when the shop is full, it can invoke balk, which exits.
- Barber threads should invoke cutHair.
- When the barber invokes cutHair there should be exactly one thread invoking getHairCut concurrently.

Sleeping Barber Solution

```
int customers = 0;
mutex = Semaphore(1);
customer = Semaphore(0);
barber = Semaphore(0);


void barber (void){
    down(customer);
    up(barber);
    cutHair();
}
```

```
void customer (void){
    down(mutex);
        if (customers==n+1) {
            up(mutex);
            balk();
            }
        customers +=1;
    up(mutex);

    up(customer);
    down(barber);
    getHairCut();

    down(mutex);
        customers -=1;
    up(mutex);
}
```