Agenda

- Assignment 2 out, due October 20
- Midterm on Wednesday
- Today's lecture
  - Classic Synchronization Problems
  - Textbook Reading: 6.6, 6.7, 6.8

Common Concurrency Mechanisms

- Semaphore – integer value used for signaling among processes.
  - Initialize, increment, decrement operations may be performed on a semaphore
  - Operations are atomic
  - Decrement operation may result in the blocking of a process
  - Increment operation may result in the unblock of a process
  - Binary Semaphore – takes only the values 0 and 1
- Mutex – similar to a binary semaphore
  - With the restriction that the process that locks the mutex must be the one to unlock it
  - Condition variable: a data type used to block a process/thread until a particular condition holds

Common Concurrency Mechanisms

- Monitors – Language specific synchronization
  - Provide a fundamental guarantee that only one process may be in a monitor at any time
  - In Java, the lock associated with an object is actually a monitor
- Implemented at the compiler/language level (hidden from the user)
  - The compiler must ensure that the property is preserved
  - Implementation of mutual exclusion is system dependent
  - Can be implemented with semaphores, locks, or other mechanisms
- The critical section is inside the monitor, to execute CS a thread
  - Enters monitor
    - Wait if there is already a thread in the monitor
  - Executes critical section
  - Leaves critical section
    - Once a thread leaves, next waiting thread can enter

Common Concurrency Mechanisms

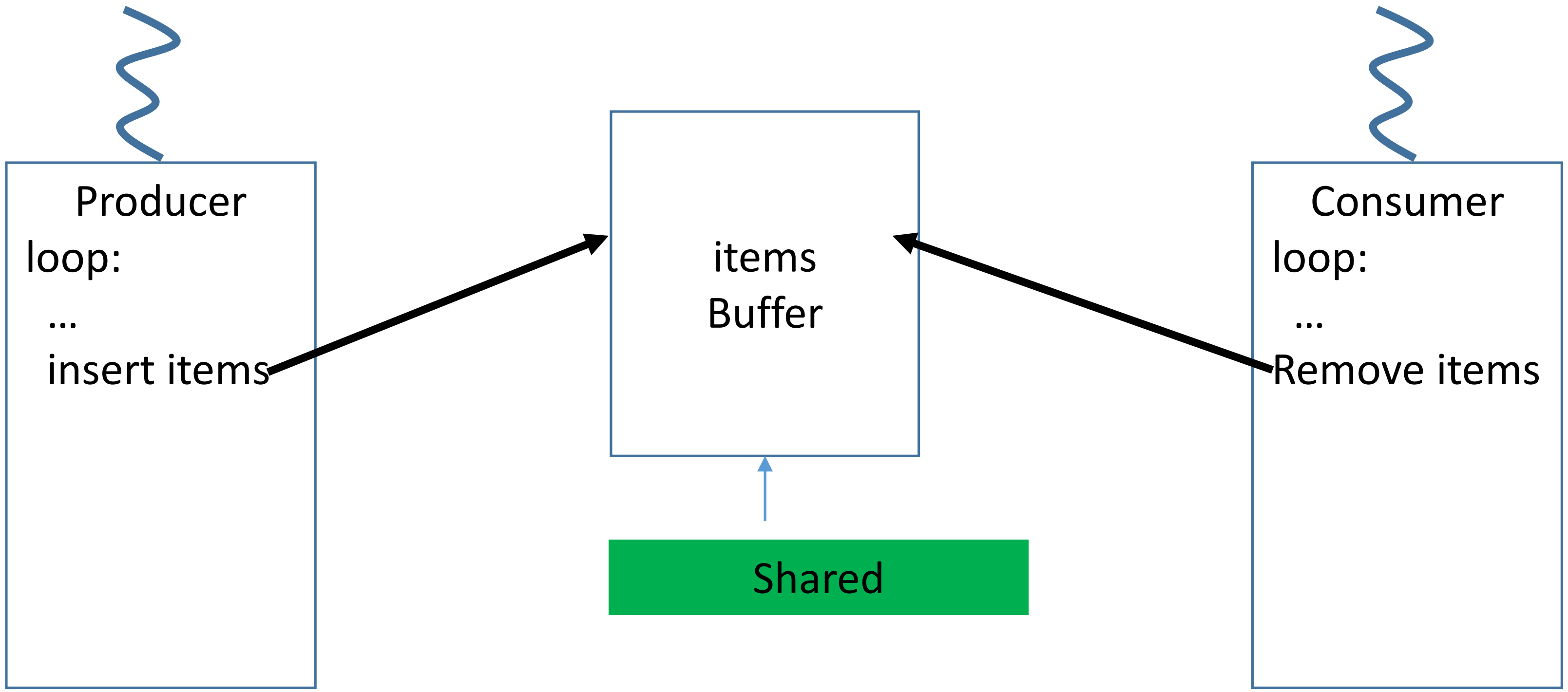
- Event Flags – a memory word used for synchronization.
  - Application code may associate a different event with each bit in a flag.
  - A thread can wait for either a single event or a combination of events by checking one or multiple bits in the corresponding flag.
  - The thread is blocked until all of the required bits are set (AND),
  - or until at least one of the bits is set (OR)
- Mailboxes/Messages – means for two processes to exchange information that may be used for synchronization
- Spinlocks – mechanism in which a process executes in an infinite loop waiting for the value of a lock variable to indicate availability

Classic Problems of Synchronization

- Producer Consumer (Bounded-Buffer) Problem
- Readers and Writers Problem
- Dining-Philosophers Problem

Producer-Consumer Revisited

Synchronization Challenge: Empty Buffer, Full Buffer, Shared Variables



Producer Consumer (Bounded-Buffer) Problem

```
// producers call this method
public void insert(E item) {

while (count == BUFFER_SIZE)
; // do nothing -- no free space

// add an element to the buffer
elements[in] = item;
in = (in + 1) % BUFFER_SIZE;
++count;
}
```

Wait until buffer is not full

```
// consumers call this method
public E remove() {
E item;

while (count == 0)
; // do nothing - nothing to consume

// remove an item from the buffer
item = elements[out];
out = (out + 1) % BUFFER_SIZE;
--count;

return item;
}
```

Wait until buffer is not empty

Producer Consumer – Better Solution

```java
// producers call this method
public void insert(E item) {

if (count == BUFFER_SIZE) sleep();

// add an element to the buffer
elements[in] = item;
in = (in + 1) % BUFFER_SIZE;
++count;
if (count == 1) wakeup(consumer);
}
```

sleep until buffer is not full

Send wake up to consumer thread

```java
// consumers call this method
public E remove() {
E item;
if (count == 0) sleep();

// remove an item from the buffer
item = elements[out];
out = (out + 1) % BUFFER_SIZE;
--count;
if (count == BUFFER_SIZE -1) wakeup(producer);
return item;
}
```

sleep until buffer is not empty

Send wake up to producer thread

Producer Consumer Using Semaphores

- *N* buffers, each can hold one item
- Semaphore mutex initialized to the value 1
- Semaphore full initialized to the value 0
- Semaphore empty initialized to the value N

Producer Consumer Using Semaphores

```
int BUFFER_SIZE = 5;

Semaphore mutex = new Semaphore(1);
Semaphore empty = Semaphore(BUFFER_SIZE);
Semaphore full = new Semaphore(0);


int count = 0;
int in =0, out=0;
E[] buffer = (E[]) new Object[BUFFER_SIZE];
```

Semaphore Wins:

- No need to increment/decrement count
- A single call handles boundary case (empty/full buffer)
- Use the same abstraction for various synchronization problems (Mutual exclusion, empty, full buffer)

```
// producer calls this method
public void insert(E item) {

empty.acquire();
mutex.acquire();

// add an item to the buffer
++count;
buffer[in] = item;
in = (in + 1) % BUFFER_SIZE;

mutex.release();
full.release();

}
```

Block until buffer is not full

Buffer is not empty

```
// consumer calls this method
public E remove() {

        full.acquire();
        mutex.acquire();

        // remove an item from the buffer
        --count;
        E item = buffer[out];
        out = (out + 1) % BUFFER_SIZE;

        mutex.release();
        empty.release();

return item;

}
```

Block until buffer is not empty

Buffer is not full

Producer Consumer Using Semaphores

```
int BUFFER_SIZE = 5;

Semaphore mutex = new Semaphore(1);
Semaphore empty = Semaphore(BUFFER_SIZE);
Semaphore full = new Semaphore(0);
```

Semaphore Case:

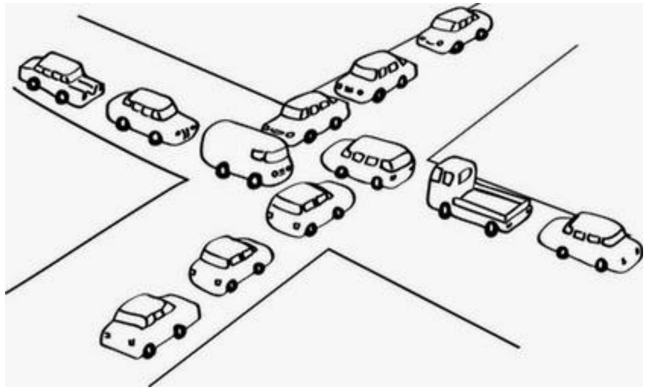
- Suppose we have a full buffer!

```
// producer calls this method
public void insert(E item) {

mutex.acquire();
empty.acquire();


// add an item to the buffer
++count;
buffer[in] = item;
in = (in + 1) % BUFFER_SIZE;

mutex.release();
full.release();

}
```

```
// consumer calls this method
public E remove() {

          full.acquire();
          mutex.acquire();

          // remove an item from the buffer
          --count;
          E item = buffer[out];
          out = (out + 1) % BUFFER_SIZE;

          mutex.release();
          empty.release();

return item;

}
```

Producer Consumer Using Semaphores

```java
int BUFFER_SIZE = 5;

Semaphore mutex = new Semaphore(1);
Semaphore empty = Semaphore(BUFFER_SIZE);
Semaphore full = new Semaphore(0);
```



Semaphore Problem:

- Suppose we have a full buffer!
- Deadlock situation!

```java
// producer calls this method
public void insert(E item) {

mutex.acquire();
empty.acquire();


// add an item to the buffer
++count;
buffer[in] = item;
in = (in + 1) % BUFFER_SIZE;

mutex.release();
full.release();

}
```

Block until buffer is not full

```java
// consumer calls this method
public E remove() {

        full.acquire();
        mutex.acquire();

        // remove an item from the buffer
        --count;
        E item = buffer[out];
        out = (out + 1) % BUFFER_SIZE;

        mutex.release();
        empty.release();

return item;

}
```

Block until CS is available

Producer Consumer Using Monitors

```
int count = 0;
int in =0, out=0;
E[] buffer = (E[]) new Object[BUFFER_SIZE];

condition notfull, notempty;

Monitor boundedbuffer;
```

Monitors Wins:

- Monitor enforces mutual exclusion
- Only synchronization is the responsibility of the programmer
- Once a monitor is correctly programmed, access to protected resources is correct from all processes

```
// producers call this method

public void insert(E item) {

while (count == BUFFER_SIZE) notfull.wait();

// add an element to the buffer

elements[in] = item;
in = (in + 1) % BUFFER_SIZE;
++count;

notempty.signal();

}
```

Wait until buffer is not full

Resume any waiting consumer

```
// consumers call this method

public E remove() {

E item;

while (count == 0) notempty.wait();

// remove an item from the buffer

item = elements[out];
out = (out + 1) % BUFFER_SIZE;
--count;

notfull.signal();

return item;

}
```

Wait until buffer is not empty

Resume any waiting producer

Readers-Writers Problem

- 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
- Synchronization problem
  - multiple readers can read at the same time
  - only one writer can access the shared data at the same time
  - If a writer is writing, no reader may read

Readers-Writers Problem

- Shared data
  - Library catalogue
  - Users read the catalog to locate a book - readers
  - Librarians are able to update the catalogue – writers
  - Writers cannot interfere with each other
  - Readers cannot read, while writing is in progress
- If we treat every access to the shared data as a CS
  - Users will be forced to read the catalogue one at a time
  - Unacceptable delays!
- Is it possible to treat this problem as a Producer Consumer problem?

Readers-Writers Problem – Solution 1

- Readers Have Priority
  - No reader is kept waiting unless a writer already locked the shared object
- Shared Data – using Semaphores
  - Data set
  - Semaphore mutex initialized to 1
  - Semaphore db initialized to 1
  - Integer readerCount initialized to 0

Readers-Writers Problem - Solution 1

- Interface for read-write locks

```
public interface ReadWriteLock
{
        public abstract void acquireReadLock(int readerNum);
        public abstract void acquireWriteLock(int writerNum);
        public abstract void releaseReadLock(int readerNum);
        public abstract void releaseWriteLock(int writerNum);
}
//Fig. 6.17
```

Database factory

```java
public class Database implements ReadWriteLock{

// the number of active readers
private int readerCount;

Semaphore mutex;  // controls access to readerCount
Semaphore db;     // controls access to the database

public Database() {
readerCount = 0;
mutex = new Semaphore(1);
db = new Semaphore(1);

}
```

Writer methods

```java
public void acquireWriteLock(int writerNum) {

        db.acquire();

}

public void releaseWriteLock(int writerNum) {

        db.release();

}
```

Resume waiting readers or writer

Reader methods

```java
public void acquireReadLock(int readerNum) {

   mutex.acquire();

   ++readerCount;

   // if I am the first reader tell all others

   // that the database is being read

   if (readerCount == 1) db.acquire();

   mutex.release();

}
```

First reader locks db

```java
public void releaseReadLock(int readerNum) {

   mutex.acquire();

   --readerCount;

   // if I am the last reader tell all others

   // that the database is no longer being read

  if (readerCount == 0) db.release();

  mutex.release();

}
```

Last reader unlock db

Readers-Writers Problem – Solution 1

- Readers Have Priority
  - No reader is kept waiting unless a writer already locked the shared object
- Problem - Writers are subject to starvation!
  - as long as there is at least one reader reading
  - it is possible for readers to retain control of the data area

Readers-Writers Problem – Solution 2

- Writers Have Priority
  - No new readers are allowed access to the data area once at least one writer showed interest to write

Readers-Writers Problem – Solution 2

- Writers Have Priority
  - No new readers are allowed access to the data area once at least one writer showed interest to write
- Shared Data – using Semaphores
  - Data set
  - Integer readerCount initialized to 0
  - Semaphore rMutex initialized to 1 - that controls the updating of readCount
  - Sempahore rdb initialized to 1 - inhibits all readers while there is at least one writer
  - Integer writerCount initialized to 0
  - Semaphore wMutex initialized to 1 – that controls the updating of writeCount
  - Semaphore wdb initialized to 1
  - Semaphore mutex initialized to 1 – readers queue

Writer methods

```
public void acquireWriteLock(int writerNum) {
        wMutex.acquire();
        writerCount++;
        if (writerCount == 1) rdb.acquire();
        wMutex.release();
        wdb.acquire()
}
public void releaseWriteLock(int writerNum) {
        wdb.release();
        wMutex.aqcuire();
        writerCount--;
        if (writerCount == 0) rdb.release();
        wMutex.release();
}
```

First writer locks db for reading

Last writer unlocks db

Resume waiting writer

Reader methods

```
public void acquireReadLock(int readerNum) {
mutex.acquire();
rdb.acquire();
   rMutex.acquire();
   ++readerCount;
   if (readerCount == 1) wdb.acquire();
   rMutex.release();
rdb.release();
mutex.release();
}

public void releaseReadLock(int readerNum) {
   rMutex.acquire();
   --readerCount;
  if (readerCount == 0) wdb.release();
  rMutex.release();
}
```

Reader queue on mutex before they queue on rdb, where only one reader will be queued

First reader locks db for writers

Last reader unlocks db for writers

Process Queues

Readers or Writers

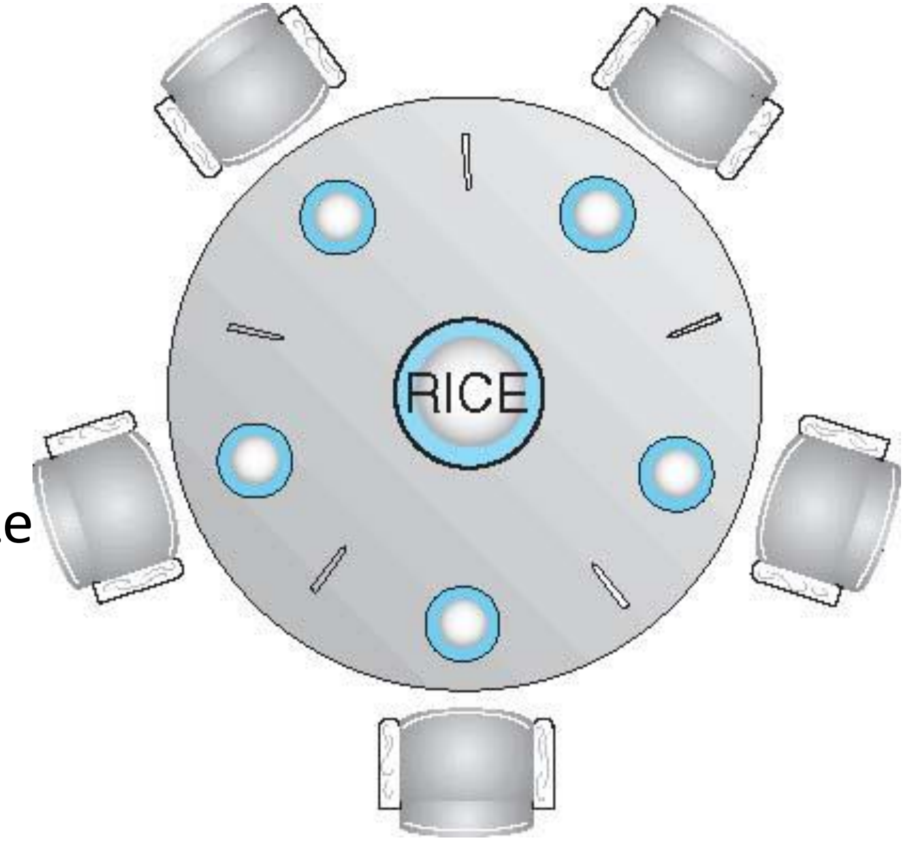
- Only readers:
  - `wdb` set
  - No queues
- Only writers:
  - `wdb` and `rdb` set
  - Writers queue on `wdb`

Readers and Writers

- Read first
  - `wdb` set by reader
  - `rdb` set by writer
  - All writers queue on `wdb`
  - One reader queues on `rdb`
  - Other readers queue on `mutex`
- Write first
  - `Wdb` set by writer
  - `rdb` set by writer
  - Writers queue on `wdb`
  - One reader queues on `rdb`
  - Other readers queue on `mutex`

Dining-Philosophers Problem

- Philosopher is either
  - Thinking
  - Eating
- Shared data
  - Bowl of rice (data set)
  - Semaphore chopStick [5] initialize



Next…

- Synchronization with Java
- Introduction to Deadlocks!