Chapter 6
Synchronization

Images from Silberschatz

Processes

- Multiple processes accessing the same data
  - Could be threads

- Producer/Consumer
  - Section 3.4.1

```
while (true)
{
    /* produce an item and put in nextProduced */

    while(count == BUFFER_SIZE)
        ; // do nothing

    buffer[in] = nextProduced;
    in = (in +1) % BUFFER_SIZE;
    count++;
}
```

- These are two separate threads.
- What's the problem?

```
while (true)
{
    while(count == 0)
        ; // do nothing

    nextConsumed = buffer[out];
    out = (out +1) % BUFFER_SIZE;
    count--;

    /* use nextConsumed */
}
```

Race Condition

- How can count++ be executed?
- How can count-- be execute?
- Why is this a problem?
  - Why else is it a problem?
- Atomic

Critical Section Problem

- Critical Section
- Mutual Exclusion
- Progress
- Bounded Waiting
- Preemptive vs non-preemptive kernels

Peterson's Solution

- Assumptions:

```
while (true) {
        flag[i] = TRUE;
        turn = j;
        while ( flag[j] && turn == j);

            CRITICAL SECTION

        flag[i] = FALSE;

            REMAINDER SECTION

}
```

- Are the 3 properties preserved?
- How might we implement this?
  - Think about system calls....

Hardware support

- Implement this on the processor
  - Machine instructions

```
boolean TestAndSet (boolean *target)
{
    boolean rv = *target;
    *target = TRUE;
    return rv:
}
```

```
while (true) {
        while ( TestAndSet (&lock ))
                ;   /* do nothing

                //    critical section

        lock = FALSE;

                //      remainder section

}
```

More hardware solutions

- **xchng** on Intel chips
- TestAndSet is really xchng & test

```
void Swap (boolean *a, boolean *b)
{
        boolean temp = *a;
        *a = *b;
        *b = temp:
}
```

```
while (true)  {
        key = TRUE;
        while ( key == TRUE)
                Swap (&lock, &key );

                //    critical section

         lock = FALSE;

                //      remainder section

}
```

CompareAndSwap

- **cmpxchg** on Intel Itanium and Intel IA-32
- pthreads eventually calls this instruction for pthread_mutex_lock()
  - http://ftp.gnu.org/gnu/glibc/glibc-2.9.tar.gz
  - deep in the nptl directory
    - lowlevellock.h

```
do
{
    waiting[i] = TRUE;
    key = TRUE;
    while(waiting[i] && key)
    {
        key = TestAndSet(&lock);
    }
    waiting[i] = FALSE;

    // critical section
    j = (i + 1) % n;
    while((j != i) && !waiting[j])
    {
        j = (j + 1) %n;
    }

    if(j == i)
    {
        lock = FALSE;
    }
    else
    {
        waiting[j] = FALSE;
    }
    // non-critical section
}while(TRUE);
```

```
// initialize to FALSE
boolean waiting[n];
boolean lock;
```

Semaphore

- Counting
- Binary
  - ??
- Spin lock
- Problems?
  - solutions?
- What can we say about Critical Sections?

```
wait (S) {
        while S <= 0
            ; // no-op
        S--;
    }
signal (S) {
      S++;
    }
```

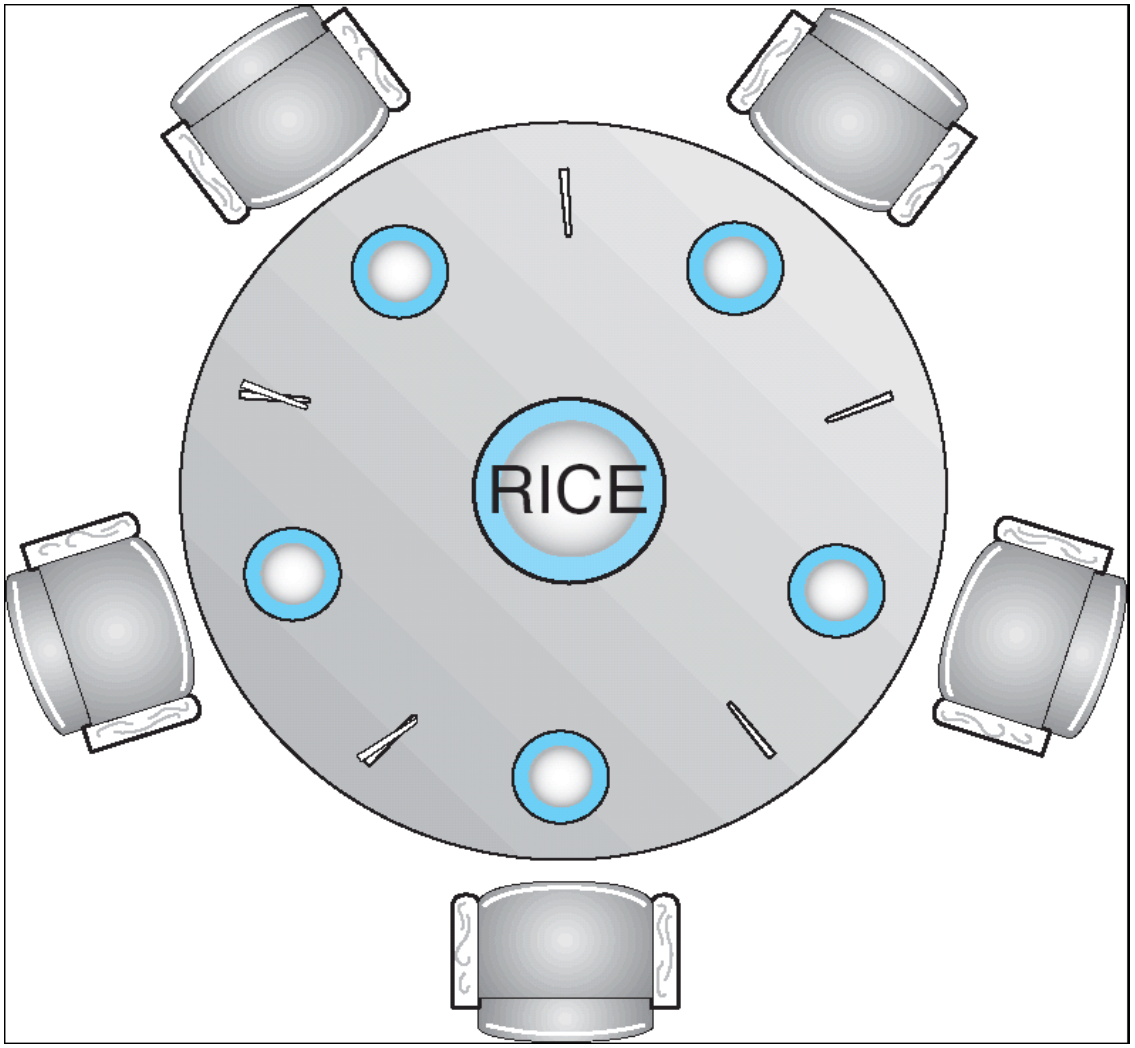
```
Semaphore S;    // initialized to 1
wait (S);
        Critical Section
signal (S);
```

Deadlock & Starvation

Classic Problems of Synchronization

- Used to test new synchronization methods
- Bounded Buffer
- Readers-Writers
- Dining Philosophers
  - or, why you should never eat at a table full of computer scientists

Dining Philosophers



Dining Philosophers Solution

- Using semaphores

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

            //  eat

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

            //  think

}
```

- Problems?

- Solutions?

Problems with Semaphores

- What can you think of?

- Why are these problems bad?
  - Really, really, really bad?
    - Evil even.

Monitors

- High level coding practice
  - *design pattern*
  - Sometimes part of the language
    - Java: *synchronized*
    - C#: *Monitor* class
    - C++ .NET: *Monitor* class
  - Sometimes you code it yourself
    - C
- Only one process can be in a monitor at a time

```
monitor monitor-name
{
   // shared variable declarations
   procedure P1 (…) { …. }
            …

   procedure Pn (…) {……}

    Initialization code ( ….) { … }
            …
   }
}
```

- Why is this useful?

Log-Based Recovery

- Ensure atomicity
  - In case of a crash
  - Databases
  - Long running computations
    - Weather simulations
    - Nuclear reaction simulations
- Write-ahead logging
  - Start
  - Commit
  - Undo
  - Redo
- Problems?

Checkpoints