Module 3
Process Synchronization

Ms.Hitha Paulson
Dept Of Computer Science

Background

- Processes can execute concurrently
    - May be interrupted at any time, partially completing execution
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes

Background

- Illustration of the problem:

  Suppose that we wanted to provide a solution to the consumer-producer problem that fills ***all*** the buffers. We can do so by having an integer **counter** that keeps track of the number of full buffers. Initially, **counter** is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

Producer

```
while (true) {
      /* produce an item in next produced 
*/

      while (counter == BUFFER_SIZE) ;
            /* do nothing */
      buffer[in] = next_produced;
      in = (in + 1) % BUFFER_SIZE;
      counter++;
}
```

Consumer

```
while (true) {
     while (counter == 0) ; /* do nothing */
     next_consumed = buffer[out];
     out = (out + 1) % BUFFER_SIZE;
        counter--;
     /* consume the item in next consumed */
}
```

Race Condition

- **`counter++`** could be implemented as

```
register1 = counter
register1 = register1 + 1
counter = register1
```

- **`counter--`** could be implemented as

```
register2 = counter
register2 = register2 - 1
counter = register2
```

- Consider this execution interleaving with “count = 5” initially:

```
S0: producer execute register1 = counter          {register1 = 5}
S1: producer execute register1 = register1 + 1    {register1 = 6}
S2: consumer execute register2 = counter          {register2 = 5}
S3: consumer execute register2 = register2 – 1    {register2 = 4}
S4: producer execute counter = register1          {counter = 6 }
S5: consumer execute counter = register2          {counter = 4}
```

Critical Section Problem

- Consider system of *n* processes $\{p_0, p_1, \ldots p_{n-1}\}$
- Each process has **critical section** segment of code
  - **Process may be changing common variables, updating table, writing file, etc**
  - **When one process in critical section, no other may be in its critical section**
- *Critical section problem* is to design protocol to solve this
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

Critical Section

- General structure of process P_i

```
do {

    entry section

        critical section

    exit section

        remainder section

} while (true);
```

Solution to Critical-Section Problem

A solution to the critical section problem must satisfy the following three re quirements

1. **Mutual Exclusion** – If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
2. **Progress** – If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
3. **Bounded Waiting** – A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
   - Assume that each process executes at a nonzero speed
   - No assumption concerning **relative speed** of the n processes

Two Process Solutions pi,pj : j=i-1

```
do {

    while (turn !=i);
    critical section
    turn = j;

    remainder section
 } while (true);
```

P_i

```
do {

            while (turn !=j);
            critical section
            turn = i;
            remainder section
     } while (true);
```

P_j

Here the problem is even if a process not wishes to be in its critical section, it gets a turn and blocks other processes to enter to their critical section

Algorithm 1

Two Process Solutions pi,pj : j=i-1

- Process P_i

```
repeat
flag[i] := true;
while (flag[j]);
critical section
flag [i] := false;
remainder section
until false;
```

- Process *Pj*
- **repeat**

```
flag[j]:= true;
while (flag[i]);
critical section
flag [j] := false;
remainder section
until false;
```

- Satisfies mutual exclusion, but not progress requirement.

Problem is both the processes flag m ay be true and both of them may be i n a waiting state

Algorithm 2

Two Process Solutions pi,pj : j=i-1

Peterson's Solution

```
do {
     flag[i] = true;
     turn = j;
     while (flag[j] && turn = = j);
             critical section
     flag[i] = false;
             remainder section
 } while (true);
```

```
do {
     flag[j] = true;
     turn = i;
     while (flag[i] && turn = = i);
             critical section
     flag[j] = false;
             remainder section
 } while (true);
```

- Provable that the three CS requirement are met:
  1. Mutual exclusion is preserved

     P_i enters CS only if:

     either `flag[j] = false` or `turn = i`
  2. Progress requirement is satisfied
  3. Bounded-waiting requirement is met

Algorithm 3

Multiple Process Solution
Bakery Algorithm

Critical section for n processes

- Before entering its critical section, process receives a number. Holder of the smallest number enters the critical section.
- If processes P_i and P_j receive the same number, if $i < j$, then P_i is served first; else P_j is served first.
- Similar to a token system in Bakery

Bakery Algorithm

```
repeat
   choosing[i] := true;
   number[i] := max(number[0], number[1], …, number [n – 1])+1;
   choosing[i] := false;
   for (j := 0; j<n; j++)
{
            while choosing[j];
            while (number[j] ≠ 0 and (number[j],j) < (number[i], i));
}
     critical section

number[i] := 0;

       remainder section
until false;
```

((a,b) < (c,d)) if a < c or if a = c and b < d

Structure of process Pi in Bakery Algorithm

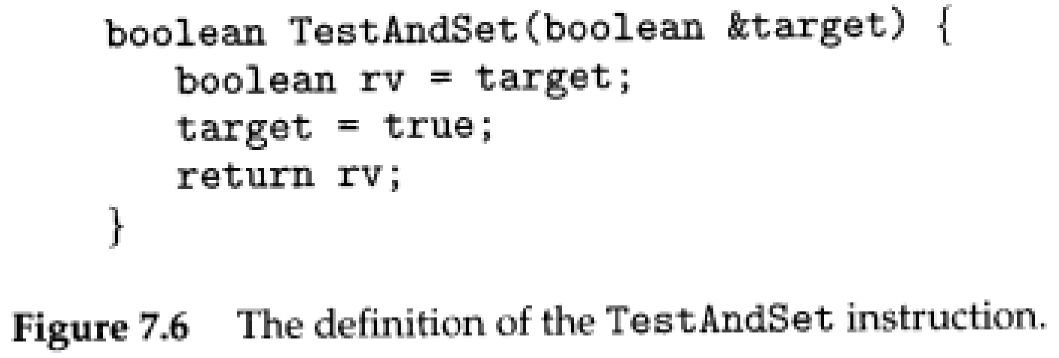
Synchronization Hardware

- Many systems provide hardware support for implementing the critical section code.
- All solutions below based on idea of locking
  - Protecting critical regions via locks
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
    - Atomic = non-interruptible
  - Either testandset Or swap instruction

Solution to Critical-section Problem Using Locks

```
do {
   acquire lock
         critical section
   release lock
         remainder section
} while (TRUE);
```

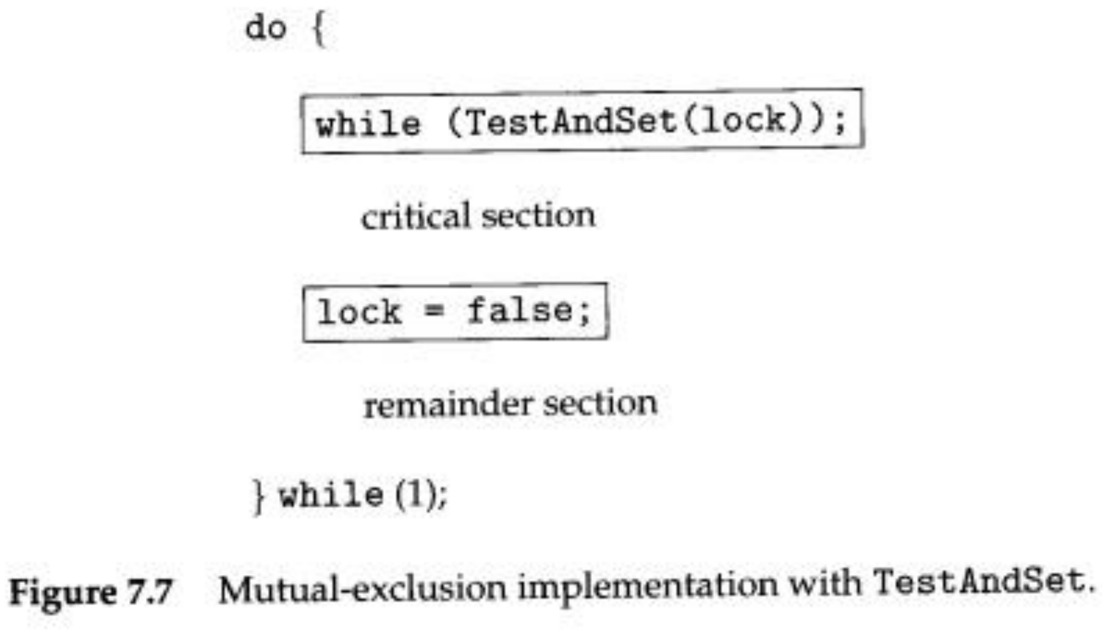
Synchronization Hardware



```
boolean TestAndSet(boolean &target) {
   boolean rv = target;
   target = true;
   return rv;
}
```

Figure 7.6 The definition of the TestAndSet instruction.

```
lock=false            // Global
```



```
do {

   while (TestAndSet(lock));

      critical section

   lock = false;

      remainder section

} while (1);
```

Figure 7.7 Mutual-exclusion implementation with TestAndSet.

Synchronization Hardware

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

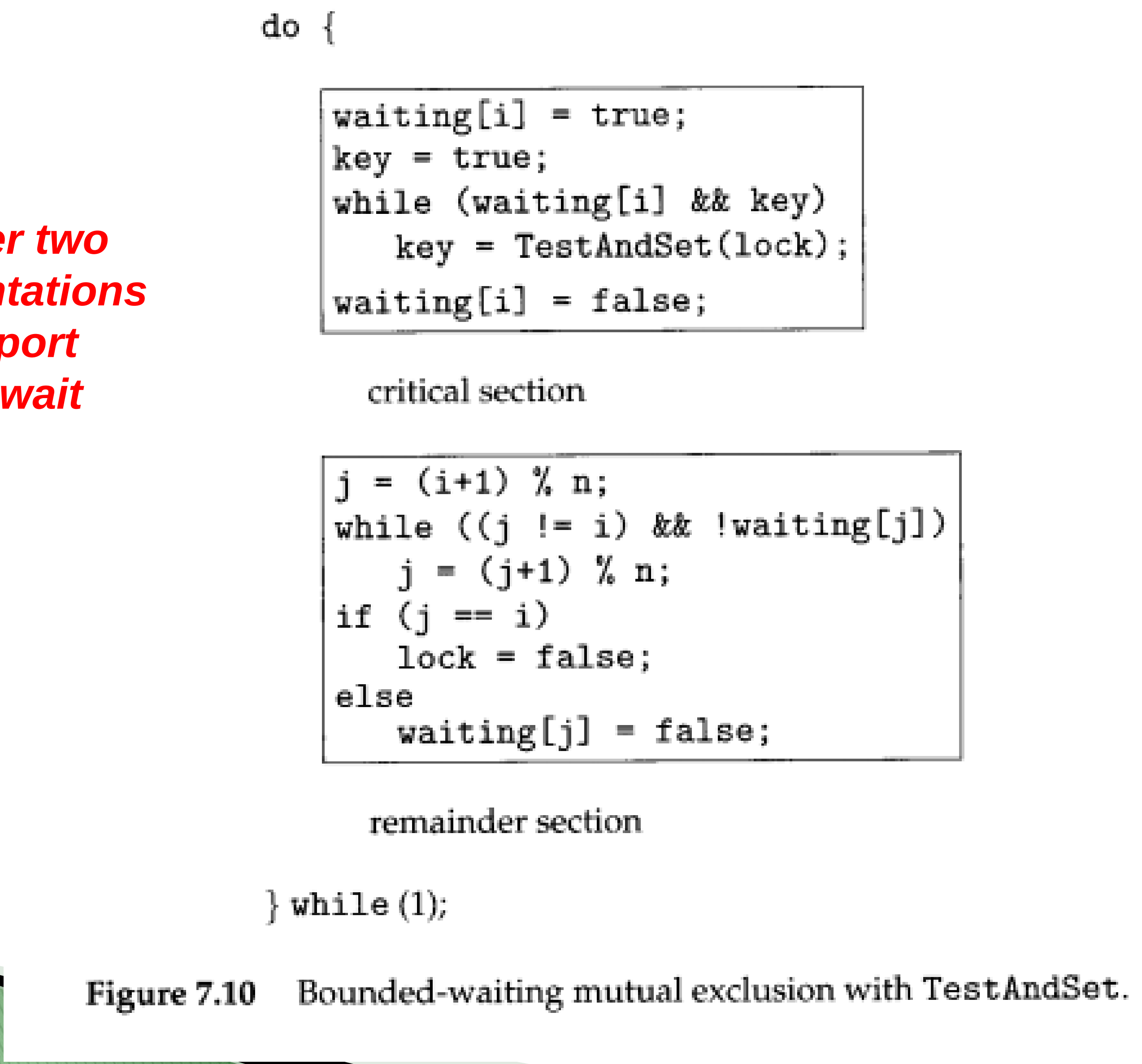
Figure 7.8 The definition of the Swap instruction.

```
do {

     key = true;
     while (key == true)
       Swap(lock,key);

        critical section

     lock = false;

        remainder section

} while (1);
```

Figure 7.9 Mutual-exclusion implementation with the Swap instruction.

Synchronization Hardware

The earlier two implementations wont support bounded wait



```
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;

        remainder section

} while (1);
```

Figure 7.10 Bounded-waiting mutual exclusion with TestAndSet.

Semaphore

- Synchronization tool that provides more sophisticated ways for process to synchronize their activities.
- Semaphore *S* is an integer variable that apart from initialization can only be accessed through two indivisible (atomic) operations
    - **`wait()`** and **`signal()`**
        - Originally called **`P()`** and **`V()`**

Semaphore

- Definition of the `wait() operation`

```
wait(S)
{
    while (S <= 0)
        ; // busy wait
    S--;
}
```

- Definition of the `signal() operation`

```
signal(S)
 {
    S++;
}
```

Semaphore Usage

- Counting semaphore – integer value can range over an unrestricted domain
- Binary semaphore – integer value can range only between 0 and 1
- Can solve various synchronization problems

- Consider P_1 and P_2 that require S_1 to happen before S_2

  Create a semaphore "**synch**" initialized to 0

```
P1:
    S1;
    signal(synch);

P2:
    wait(synch);
    S2;
```

Semaphore Implementation

- The main disadvantage of earlier mutual exclusion solutions are their busy waiting.
- Continual looping is a real problem.
- Even though this spinlocks wont make context switches they are expected to be held for short time.

Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated waiting queue
- Each entry in a waiting queue has two data items:
  - value (of type integer)
  - pointer to next record in the list
- Two operations:
  - block – place the process invoking the operation on the appropriate waiting queue
  - wakeup – remove one of processes in the waiting queue and place it in the ready queue
-
```
typedef struct{
  int value;
  struct process *L;
  } semaphore;
```

Implementation with no Busy waiting (Cont.)

The `wait` semaphore operation can now be defined as

```
void wait(semaphore S) {
        S.value--;
        if (S.value < 0) {
              add this process to S.L;
              block();
        }
}
```

The `signal` semaphore operation can now be defined as

```
void signal(semaphore S) {
        S.value++;
        if (S.value <= 0) {
              remove a process P from S.L;
              wakeup(P);
        }
}
```

Semaphore Implementation with no Busy waiting

- Under the classical definition semaphores cant be negative.
- This implementation makes semaphore negative and its magnitude is the number of processes waiting on that semaphore.

Semaphore Implementation

- Must guarantee that no two processes can execute the **`wait()`** and **`signal()`** on the same semaphore at the same time
- Thus, the implementation becomes the critical section problem where the **`wait`** and **`signal`** code are placed in the critical section
  - Could now have busy waiting in critical section implementation
    - But implementation code is short

Deadlock and Starvation

- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let S and Q be two semaphores initialized to 1

```
        P0                          P1
    wait(S);                    wait(Q);
    wait(Q);                    wait(S);
     ...                         ...
    signal(S);                  signal(Q);
     signal(Q);                  signal(S);
```

- **Starvation** – **indefinite blocking**
    - A process may never be removed from the semaphore queue in which it is suspended

Classical Problems of Synchronization

- Classical problems used to test newly-proposed synchronization schemes
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem

Bounded-Buffer Problem

- *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

Bounded Buffer Problem (Cont.)

- The structure of the producer process

```
do {
     ...
    /* produce an item in next_produced */
     ...
   wait(empty);
   wait(mutex);
      ...
    /* add next produced to the buffer */
      ...
   signal(mutex);
   signal(full);
} while (true);
```

Bounded Buffer Problem (Cont.)

- The structure of the consumer process

```
Do {
   wait(full);
   wait(mutex);
      ...
  /* remove an item from buffer to next_consumed */
      ...
   signal(mutex);
   signal(empty);
      ...
   /* consume the item in next consumed */
      ...
} while (true);
```

Readers-Writers Problem

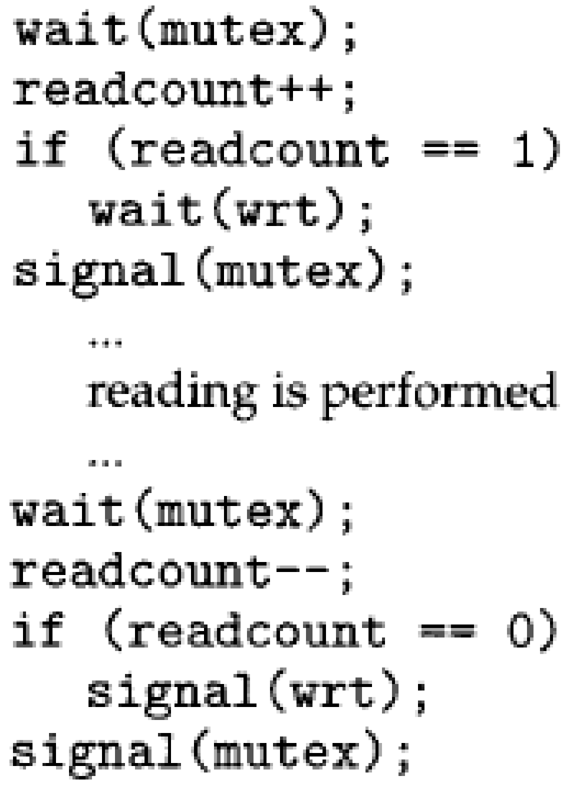
- A data set is shared among a number of concurrent processes
  - Readers – only read the data set; they do ***not*** perform any updates
  - Writers – can both read and write
- Problem – allow multiple readers to read at the same time
  - Only one single writer can access the shared data at the same time
- Mutex=1 wrt = 1 Readcount =0

Readers-Writers Problem

```
wait(wrt);
        ...
     writing is performed
        ...
signal(wrt);
```

Figure 7.14 The structure of a writer process.

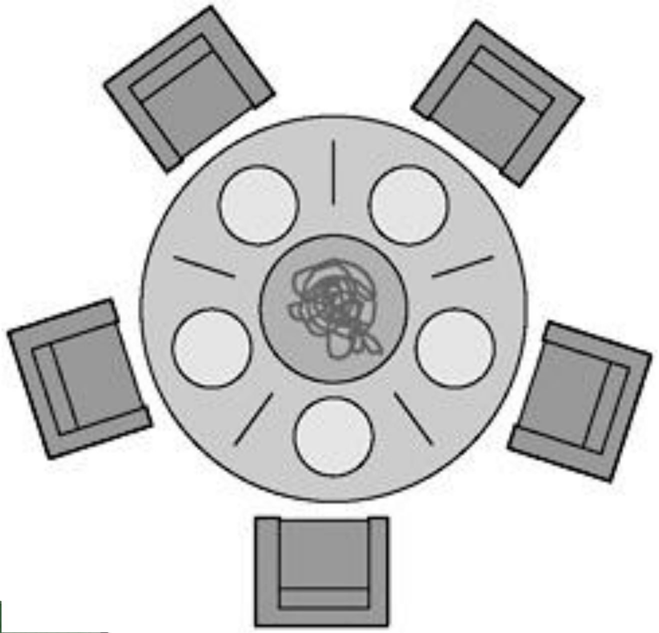
Readers–Writers Problem



```
wait(mutex);
readcount++;
if (readcount == 1)
   wait(wrt);
signal(mutex);
   ...
   reading is performed
   ...
wait(mutex);
readcount--;
if (readcount == 0)
   signal(wrt);
signal(mutex);
```

Figure 7.15 The structure of a reader process.

The Dining Philosophers Problem



```
do {
   wait(chopstick[i]);
   wait(chopstick[(i+1) % 5]);
      ...
      eat
      ...
   signal(chopstick[i]);
   signal(chopstick[(i+1) % 5]);
      ...
      think
      ...
} while (1);
```

Figure 7.17 The structure of philosopher *i*.

The Dining Philosophers Problem

ensures freedom from deadlocks.

- Allow at most four philosophers to be sitting simultaneously at the table.
- Allow a philosopher to pick up her chopsticks only if both chopsticks are available (to do this she must pick them up in a critical section).

- Use an asymmetric solution; that is, an odd philosopher picks up first her left chopstick and then her right chopstick, whereas an even philosopher picks up her right chopstick and then her left chopstick.