CS307 Operating Systems

Process Synchronization

Fan Wu
Department of Computer Science and Engineering
Shanghai Jiao Tong University
Spring 2020

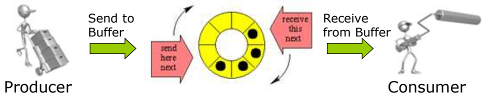
Background

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes

---

Producer-Consumer Problem

- Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process
    - *unbounded-buffer* places no practical limit on the size of the buffer
    - *bounded-buffer* assumes that there is a fixed buffer size



Bounded-Buffer – Shared-Memory Solution

- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
   . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

Bounded-Buffer – Shared-Memory Solution

Producer

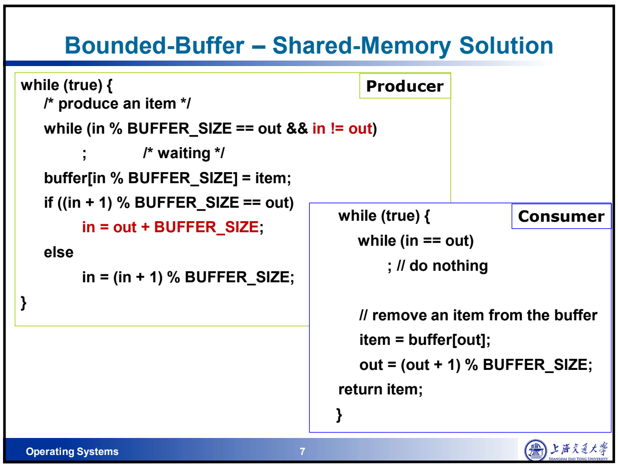
```
while (true) {
  /* Produce an item */
  while (((in + 1) % BUFFER_SIZE)  == out)
     ;   /* do nothing -- no free buffers */
     buffer[in] = item;
     in = (in + 1) % BUFFER_SIZE;
}
```

Consumer

```
while (true) {
    while (in == out)
           ; // do nothing
      // remove an item from the buffer
      item = buffer[out];
      out = (out + 1) % BUFFER_SIZE;
         return item;
   }
```

Bounded-Buffer – Shared-Memory Solution

- Weakness:
    - The solution allows only BUFFER_SIZE-1 elements at the same time
    - Busy waiting
- Can you:
    - Rewrite the previous processes to allow BUFFER_SIZE items in the buffer at the same time



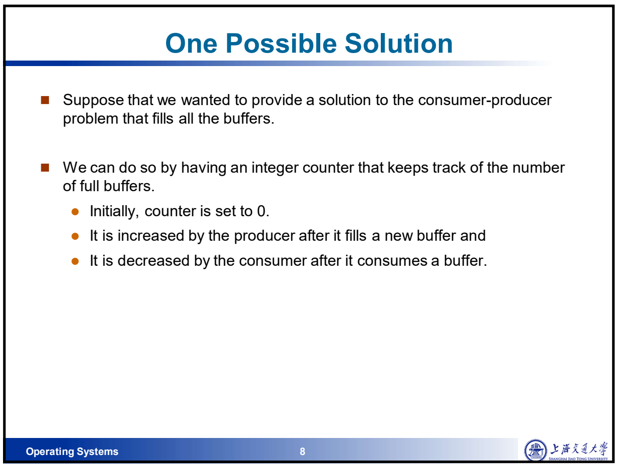
Bounded-Buffer – Shared-Memory Solution

Producer

```
while (true) {
     /* produce an item */
     while (in % BUFFER_SIZE == out && in != out)
          ;            /* waiting */
     buffer[in % BUFFER_SIZE] = item;
     if ((in + 1) % BUFFER_SIZE == out)
          in = out + BUFFER_SIZE;
     else
          in = (in + 1) % BUFFER_SIZE;
}
```

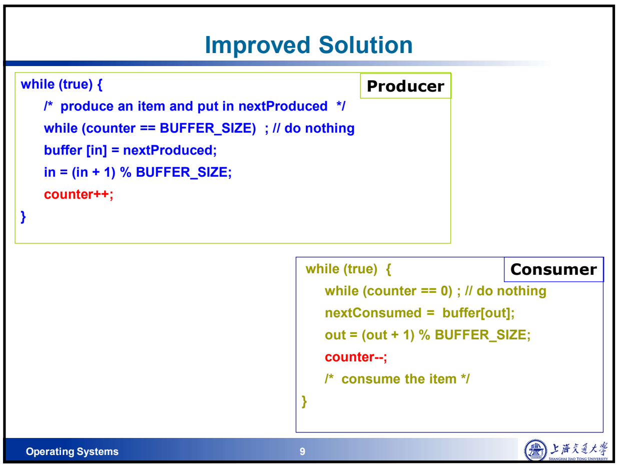
Consumer

```
while (true) {
     while (in == out)
          ; // do nothing

     // remove an item from the buffer
     item = buffer[out];
     out = (out + 1) % BUFFER_SIZE;
return item;
}
```



One Possible Solution

- 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 increased by the producer after it fills a new buffer and
    - It is decreased by the consumer after it consumes a buffer.



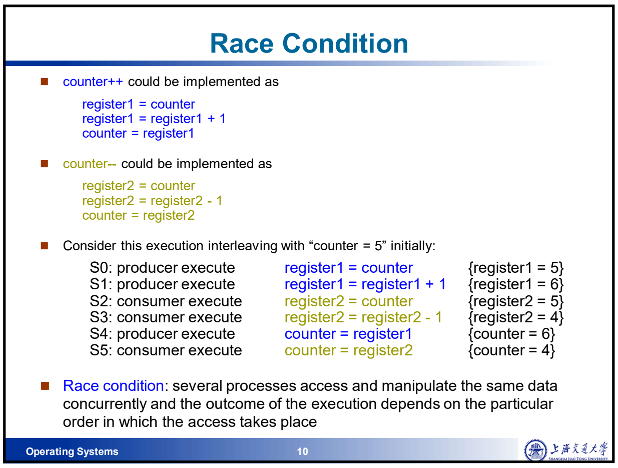
Improved Solution

Producer

```
while (true) {
     /*  produce an item and put in nextProduced  */
     while (counter == BUFFER_SIZE)  ; // do nothing
     buffer [in] = nextProduced;
     in = (in + 1) % BUFFER_SIZE;
     counter++;
}
```

Consumer

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



Race Condition

- counter++ could be implemented as

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

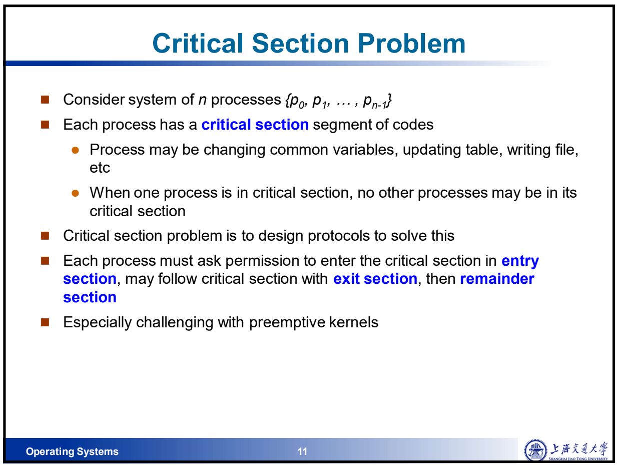
- counter-- could be implemented as

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

- Consider this execution interleaving with "counter = 5" initially:

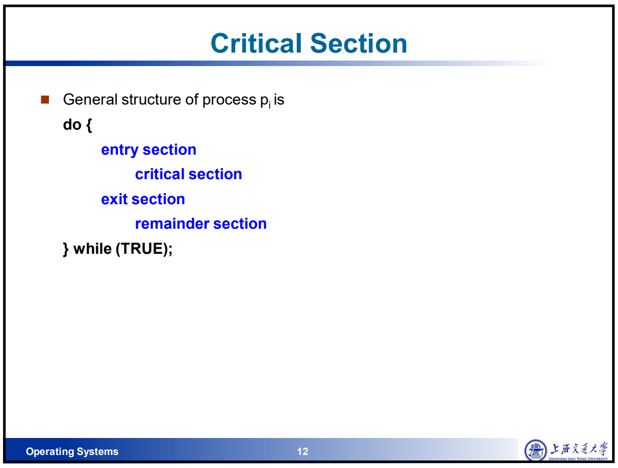
S0: producer execute	register1 = counter	{register1 = 5}
S1: producer execute	register1 = register1 + 1	{register1 = 6}
S2: consumer execute	register2 = counter	{register2 = 5}
S3: consumer execute	register2 = register2 - 1	{register2 = 4}
S4: producer execute	counter = register1	{counter = 6}
S5: consumer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

- Race condition: several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place



Critical Section Problem

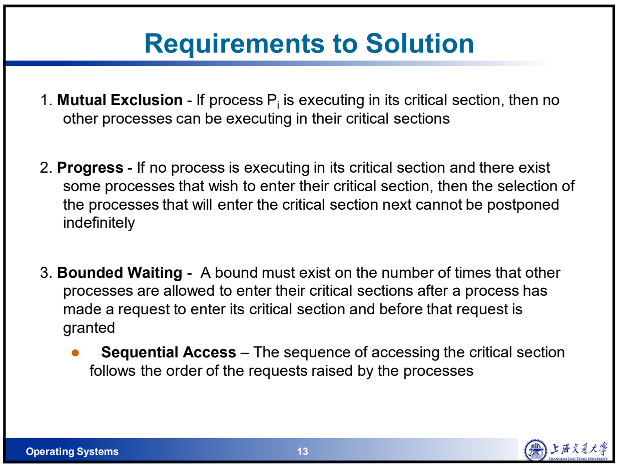
- Consider system of *n* processes $\{p_0, p_1, \ldots, p_{n-1}\}$
- Each process has a **critical section** segment of codes
    - Process may be changing common variables, updating table, writing file, etc
    - When one process is in critical section, no other processes may be in its critical section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



Critical Section

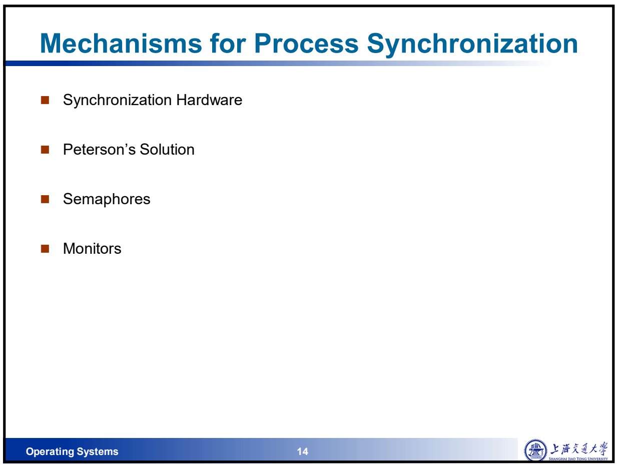
- General structure of process p_i is

```
do {
     entry section
          critical section
     exit section
          remainder section
} while (TRUE);
```



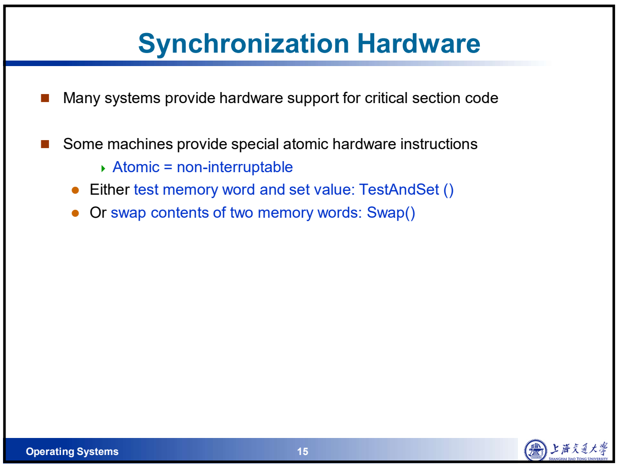
Requirements to Solution

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
   - **Sequential Access** – The sequence of accessing the critical section follows the order of the requests raised by the processes



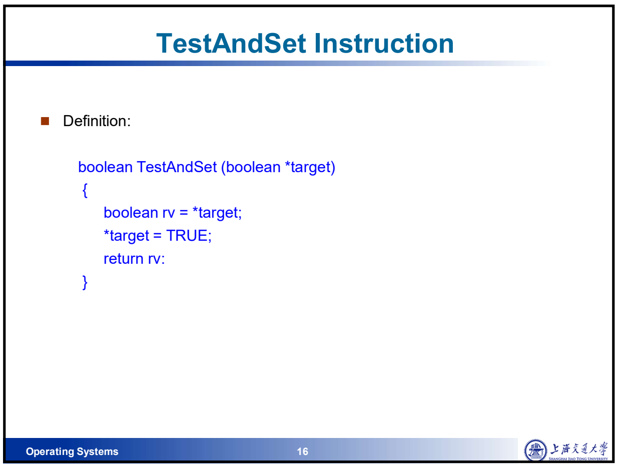
Mechanisms for Process Synchronization

- Synchronization Hardware
- Peterson's Solution
- Semaphores
- Monitors



Synchronization Hardware

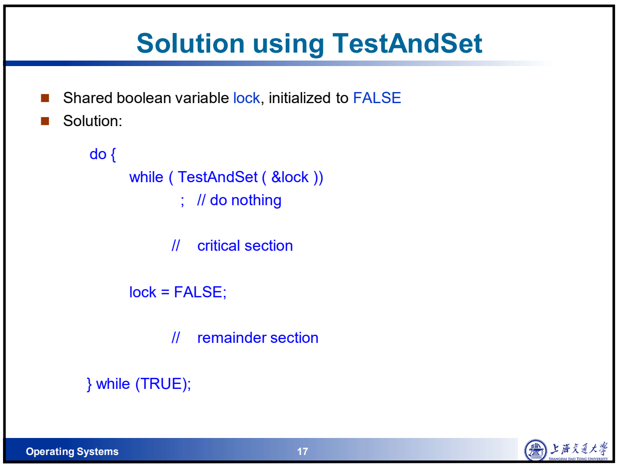
- Many systems provide hardware support for critical section code
- Some machines provide special atomic hardware instructions
    - Atomic = non-interruptable
  - Either test memory word and set value: TestAndSet ()
  - Or swap contents of two memory words: Swap()



TestAndSet Instruction

- Definition:

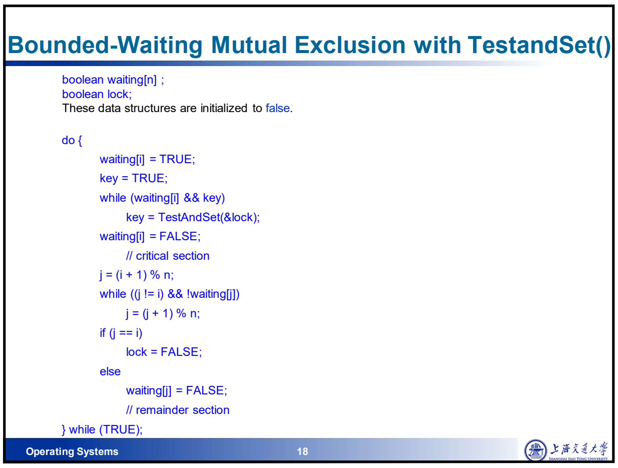
```
boolean TestAndSet (boolean *target)
 {
     boolean rv = *target;
     *target = TRUE;
     return rv:
 }
```



Solution using TestAndSet

- Shared boolean variable lock, initialized to FALSE
- Solution:

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

              //    critical section

        lock = FALSE;

              //    remainder section

} while (TRUE);
```

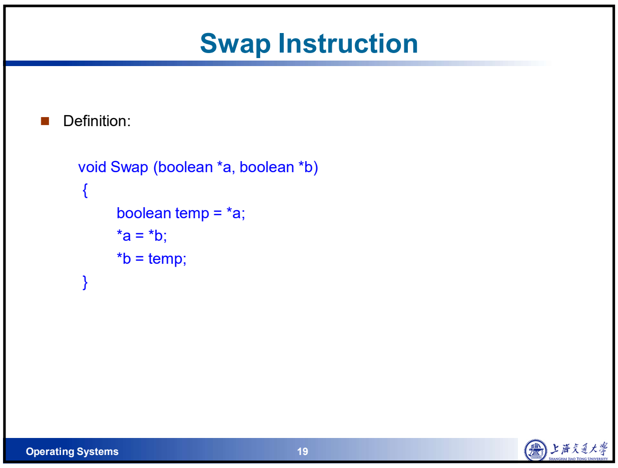


Bounded-Waiting Mutual Exclusion with TestandSet()

```
boolean waiting[n] ;
boolean lock;
```

These data structures are initialized to false.

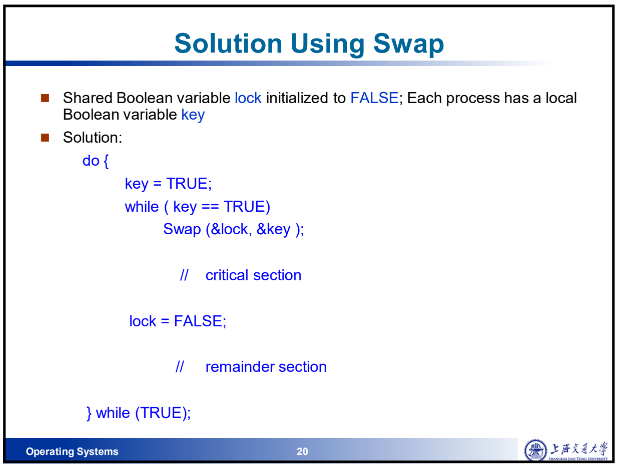
```
do {
        waiting[i] = TRUE;
        key = TRUE;
        while (waiting[i] && key)
              key = TestAndSet(&lock);
        waiting[i] = FALSE;
              // critical section
        j = (i + 1) % n;
        while ((j != i) && !waiting[j])
              j = (j + 1) % n;
        if (j == i)
              lock = FALSE;
        else
              waiting[j] = FALSE;
              // remainder section
} while (TRUE);
```



Swap Instruction

- Definition:

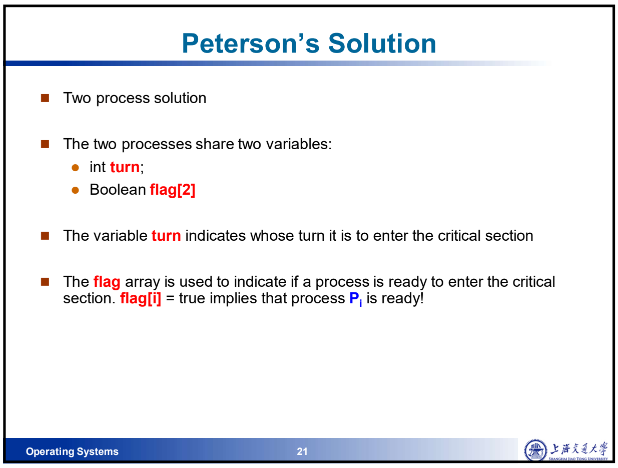
```
void Swap (boolean *a, boolean *b)
{
        boolean temp = *a;
        *a = *b;
        *b = temp;
}
```



Solution Using Swap

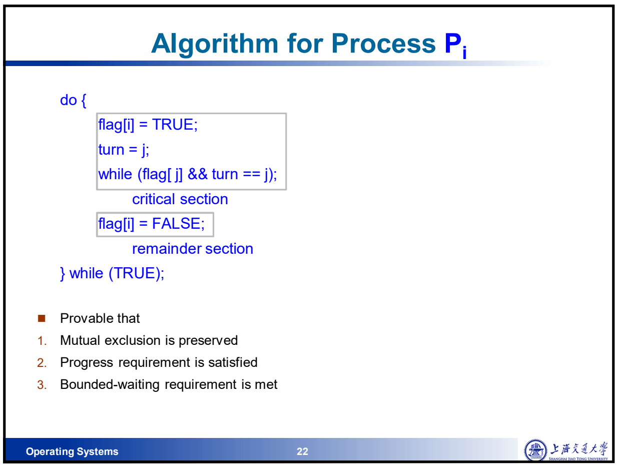
- Shared Boolean variable lock initialized to FALSE; Each process has a local Boolean variable key
- Solution:

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

                //    critical section

        lock = FALSE;

                //    remainder section

} while (TRUE);
```



Peterson's Solution

- Two process solution
- The two processes share two variables:
  - int **turn**;
  - Boolean **flag[2]**
- The variable **turn** indicates whose turn it is to enter the critical section
- The **flag** array is used to indicate if a process is ready to enter the critical section. **flag[i]** = true implies that process P_i is ready!

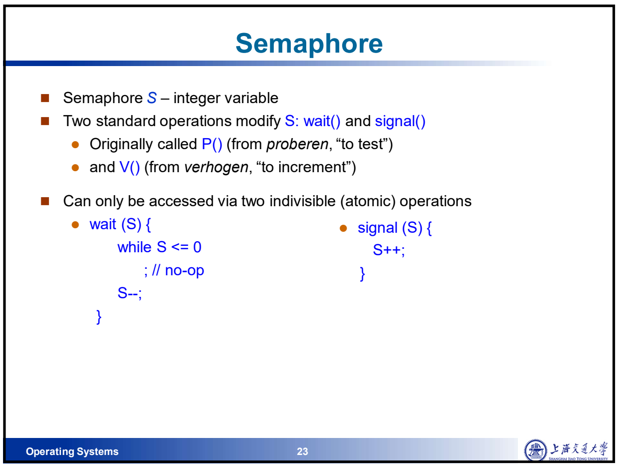


Algorithm for Process P_i

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

- Provable that

1. Mutual exclusion is preserved
2. Progress requirement is satisfied
3. Bounded-waiting requirement is met

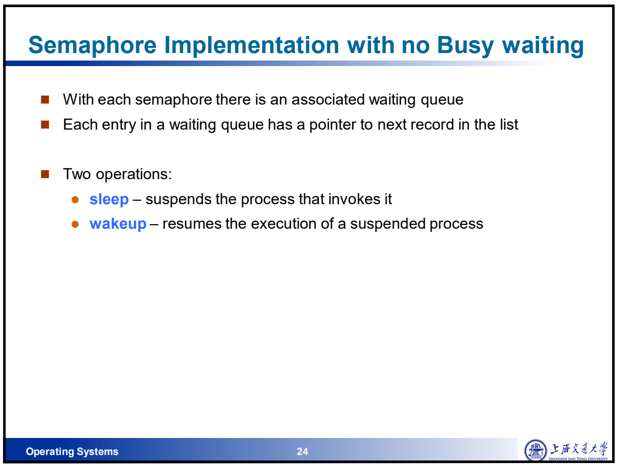


Semaphore

- Semaphore *S* – integer variable
- Two standard operations modify S: wait() and signal()
  - Originally called P() (from *proberen*, “to test”)
  - and V() (from *verhogen*, “to increment”)
- Can only be accessed via two indivisible (atomic) operations

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

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



Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated waiting queue
- Each entry in a waiting queue has a pointer to next record in the list
- Two operations:
  - **sleep** – suspends the process that invokes it
  - **wakeup** – resumes the execution of a suspended process

Semaphore Implementation with no Busy Waiting

- Implementation of wait:

```
wait(semaphore *S) {
        S->value--;
        if (S->value < 0) {
                add this process to S->list;
                sleep();
        }
}
```

- Implementation of signal:

```
signal(semaphore *S) {
        S->value++;
        if (S->value <= 0 && S->list != NULL) {
                remove a process P from S->list;
                wakeup(P);
        }
}
```

- Semaphore stucture

```
typedef struct {
        int value;
        struct process *list;
} semaphore;
```

Semaphore as General Synchronization Tool

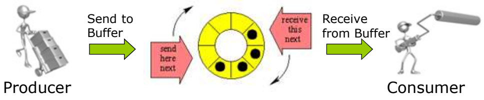
- **Binary** semaphore – integer value can range only between 0 and 1
  - Also known as **mutex locks**
- **Counting** semaphore – integer value can range over an unrestricted domain
- Can implement a counting semaphore S as a binary semaphore
- Provides mutual exclusion

```
Semaphore mutex;    //  initialized to 1
do {
     wait (mutex);
          // Critical Section
     signal (mutex);
          // remainder section
} while (TRUE);
```

Classical Problems of Synchronization

- Classical problems used to test newly-proposed synchronization schemes
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem
  - Sleeping Barber Problem
  - Baboons Crossing Problem
  - Search-Insert-Delete Problem

Bounded-Buffer Problem



Improved Solution

Producer

```
while (true) {
     /*  produce an item and put in nextProduced  */
     while (counter == BUFFER_SIZE)  ; // do nothing
     buffer [in] = nextProduced;
     in = (in + 1) % BUFFER_SIZE;
     wait (mutex);
     counter++;
     signal (mutex);
}
```

Consumer

```
while (true)  {
     while (counter == 0) ; // do nothing
     nextConsumed =  buffer[out];
     out = (out + 1) % BUFFER_SIZE;
     wait (mutex);
     counter--;
     signal (mutex);
     /*  consume the item */
}
```

Bounded-Buffer Problem

- *N* buffer slots, each can hold one item
- Semaphore full initialized to the value 0
- Semaphore empty initialized to the value N

Bounded Buffer Problem (Cont.)

- Producer process

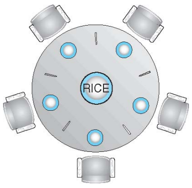
```
do  {
    //   produce an item

    wait (empty);
    buffer [in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    signal (full);

} while (TRUE);
```

- Consumer process

```
do {
    wait (full);
    nextConsumed =  buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    signal (empty);

    //  consume the item

} while (TRUE);
```

Dining-Philosophers Problem

- Philosophers spend their lives thinking and eating
- Don't interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
  - Need both to eat, then release both when done
- In the case of 5 philosophers
  - Shared data
    - Chopsticks
    - Semaphore chopstick [5] initialized to 1



Dining-Philosophers Problem Algorithm

- 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);
```

- What is the problem with this algorithm?

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 defined:
  - allow multiple readers to read at the same time
  - Only one single writer can access the shared data at each point of time
- Shared Data
  - Semaphore wrt initialized to 1: ensure mutual modification to the data set and mutual-exclusion of reading and writing
  - Integer readcount initialized to 0
  - Semaphore mutex initialized to 1: ensure mutual access to readcount

---

Readers-Writers Problem (Cont.)

- Writer process

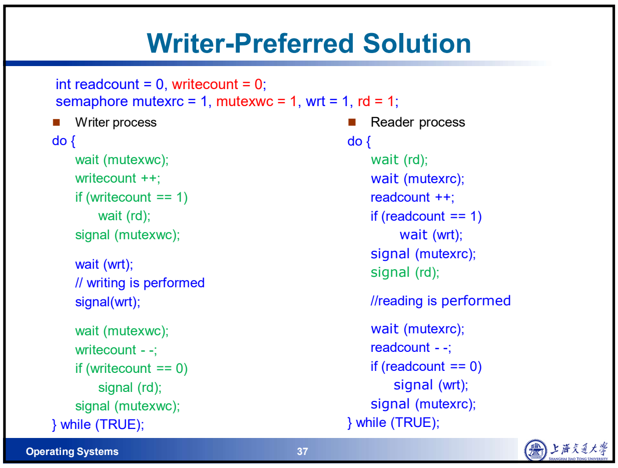
```
do {
      wait (wrt) ;

      //    writing is performed

      signal (wrt) ;
} while (TRUE);
```

- Reader process

```
do {
       wait (mutex) ;
       readcount ++ ;
       if (readcount == 1)
            wait (wrt) ;
       signal (mutex)

       // reading is performed

       wait (mutex) ;
       readcount  - - ;
       if (readcount   == 0)
              signal (wrt) ;
       signal (mutex) ;
} while (TRUE);
```

Readers-Writers Problem Variations

- *Reader-Preferred Solution* – no reader kept waiting unless writer has permission to use shared object
  - no reader should wait for other readers to finish simply because a writer is waiting
- *Writer-Preferred Solution* – once writer is ready, it performs write asap
  - if a writer is waiting to access the object, no new readers may start reading



Writer-Preferred Solution

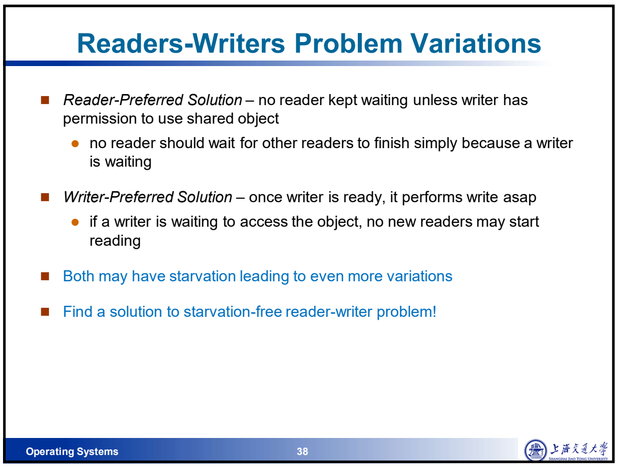
```
int readcount = 0, writecount = 0;
semaphore mutexrc = 1, mutexwc = 1, wrt = 1, rd = 1;
```

- Writer process

```
do {
    wait (mutexwc);
    writecount ++;
    if (writecount == 1)
        wait (rd);
    signal (mutexwc);

    wait (wrt);
    // writing is performed
    signal(wrt);

    wait (mutexwc);
    writecount - -;
    if (writecount == 0)
        signal (rd);
    signal (mutexwc);
} while (TRUE);
```

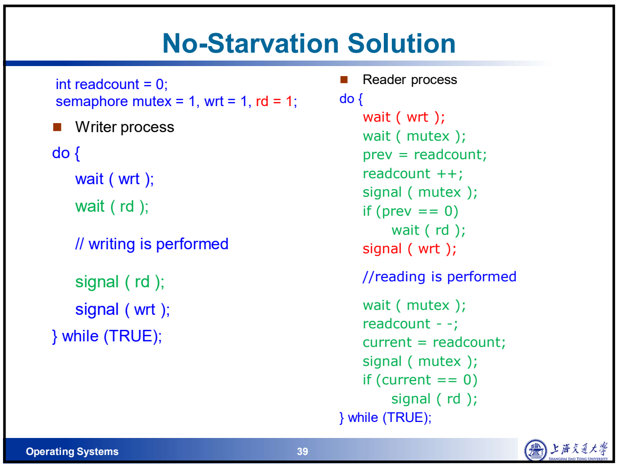
- Reader process

```
do {
    wait (rd);
    wait (mutexrc);
    readcount ++;
    if (readcount == 1)
        wait (wrt);
    signal (mutexrc);
    signal (rd);

    //reading is performed

    wait (mutexrc);
    readcount - -;
    if (readcount == 0)
        signal (wrt);
    signal (mutexrc);
} while (TRUE);
```



Readers-Writers Problem Variations

- *Reader-Preferred Solution* – no reader kept waiting unless writer has permission to use shared object
  - no reader should wait for other readers to finish simply because a writer is waiting
- *Writer-Preferred Solution* – once writer is ready, it performs write asap
  - if a writer is waiting to access the object, no new readers may start reading
- Both may have starvation leading to even more variations
- Find a solution to starvation-free reader-writer problem!



No-Starvation Solution

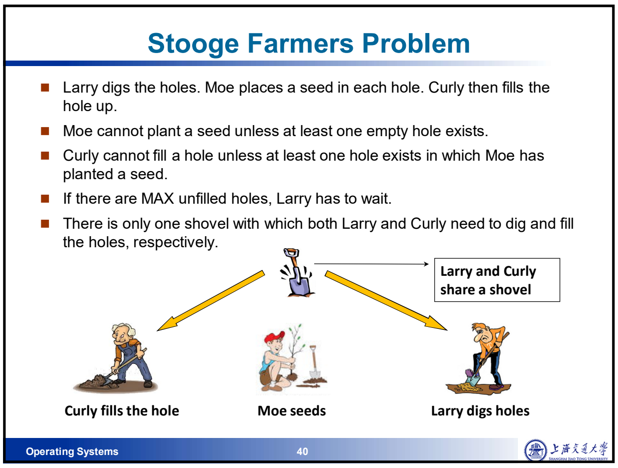
```
int readcount = 0;
semaphore mutex = 1, wrt = 1, rd = 1;
```

- Writer process

```
do {
    wait ( wrt );
    wait ( rd );

    // writing is performed

    signal ( rd );
    signal ( wrt );
} while (TRUE);
```

- Reader process

```
do {
    wait ( wrt );
    wait ( mutex );
    prev = readcount;
    readcount ++;
    signal ( mutex );
    if (prev == 0)
        wait ( rd );
    signal ( wrt );

    //reading is performed

    wait ( mutex );
    readcount - -;
    current = readcount;
    signal ( mutex );
    if (current == 0)
        signal ( rd );
} while (TRUE);
```



Stooge Farmers Problem

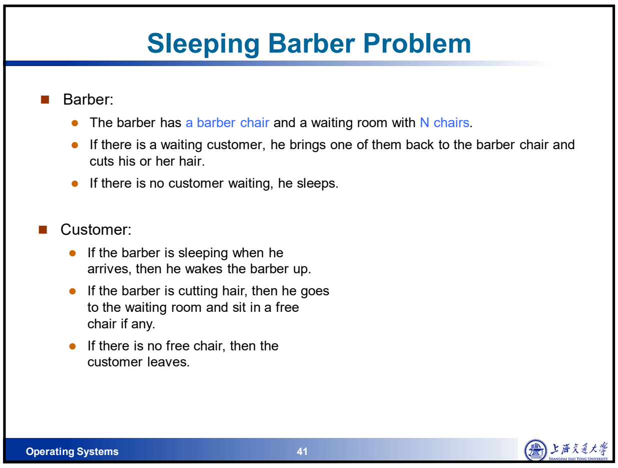
- Larry digs the holes. Moe places a seed in each hole. Curly then fills the hole up.
- Moe cannot plant a seed unless at least one empty hole exists.
- Curly cannot fill a hole unless at least one hole exists in which Moe has planted a seed.
- If there are MAX unfilled holes, Larry has to wait.
- There is only one shovel with which both Larry and Curly need to dig and fill the holes, respectively.

Larry and Curly share a shovel

Curly fills the hole

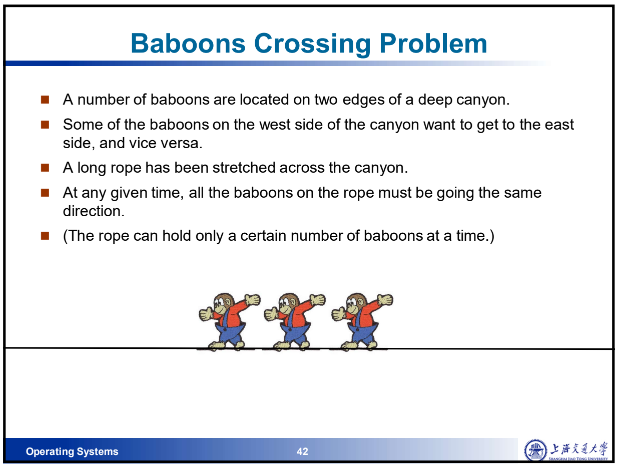
Moe seeds

Larry digs holes



Sleeping Barber Problem

- Barber:
  - The barber has a barber chair and a waiting room with N chairs.
  - If there is a waiting customer, he brings one of them back to the barber chair and cuts his or her hair.
  - If there is no customer waiting, he sleeps.
- Customer:
  - If the barber is sleeping when he arrives, then he wakes the barber up.
  - If the barber is cutting hair, then he goes to the waiting room and sit in a free chair if any.
  - If there is no free chair, then the customer leaves.



Baboons Crossing Problem

- A number of baboons are located on two edges of a deep canyon.
- Some of the baboons on the west side of the canyon want to get to the east side, and vice versa.
- A long rope has been stretched across the canyon.
- At any given time, all the baboons on the rope must be going the same direction.
- (The rope can hold only a certain number of baboons at a time.)

Solution to Baboons Crossing Problem

```
int waitEast=waitWest=0;
semaphore eastWard=westWard=mutexEast=mutexWest=mutex=1;
```

- #EastWard:

```
wait(mutex);
wait(westWard);
wait(mutexEast);
waitEast = waitEast + 1;
if(waitEast == 1)
      wait(eastWard);
signal(mutexEast);
signal(westWard);
signal(mutex)
//cross eastWard
wait(mutexEast);
waitEast = waitEast – 1;
if(waitEast == 0)
      signal(eastWard);
signal(mutexEast);
```

- #WestWard:

```
wait(mutex)
wait(eastWard);
wait(mutexWest);
waitWest = waitWest + 1;
if(waitWest == 1)
      wait(westWard);
signal(mutexWest);
signal(eastWard);
signal(mutex);
//cross westWard
wait(mutexWest);
waitWest = waitWest – 1;
if(waitWest == 0)
      signal(westWard);
signal(mutexWest);
```

Search-Insert-Delete Problem

- *Searchers* access the list without changing it. Any number of concurrent searchers can be accessing the structure safely.
- *Inserters* have the ability to add new elements to the end of the structure. Only one inserter can access the structure at any given time, but can work concurrently with any number of searchers.
- *Deleters* can remove items from any position in the structure. Any deleter demands exclusive access to the structure.

Problems with Semaphores

- Incorrect use of semaphore operations:
  - signal (mutex) …. wait (mutex)
  - wait (mutex) … wait (mutex)
  - Omitting of wait (mutex) or signal (mutex) (or both)
- Deadlock and starvation

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
  ① wait (S);         ② wait (Q);
  ③ wait (Q);         ④ wait (S);
        .                   .
        .                   .
    signal (S);         signal (Q);
    signal (Q);         signal (S);
```

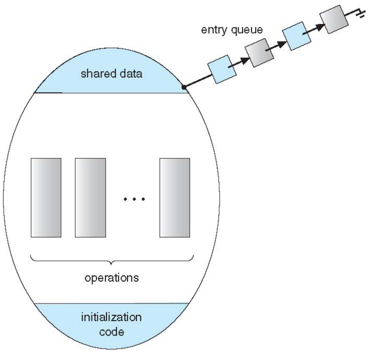
- **Starvation** – indefinite blocking
  - A process may never be removed from the semaphore queue in which it is suspended

Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- *Abstract data type*, internal variables only accessible by code within the procedure
- Only one process may be active within the monitor at a time

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

   procedure Pn (…) {……}

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

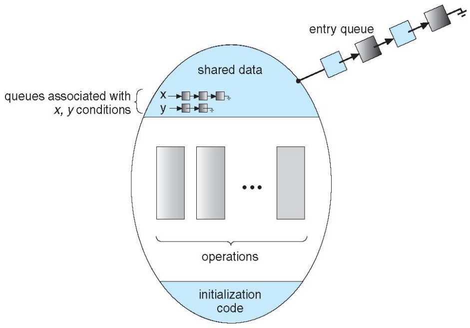
Schematic view of a Monitor



Condition Variables

- The monitor construct is not powerful enough to model some synchronization schemes
  - Need additional synchronization schemes: condition construct
- condition x;
- Two operations on a condition variable:
  - x.wait () – a process that invokes the operation is suspended until x.signal ()
  - x.signal () – resumes one of processes (if any) that invoked x.wait ()
    - If no x.wait () on the variable, then it has no effect on the variable
    - Different from that of wait() on the semaphore

Monitor with Condition Variables



Condition Variables Choices

- If process P invokes x.signal (), with Q in x.wait () state, what should happen next?
  - If Q is resumed, then P must wait
- Options include
  - **Signal and wait** – P waits until Q leaves monitor or waits for another condition
  - **Signal and continue** – Q waits until P leaves the monitor or waits for another condition
  - Both have pros and cons – language implementer can decide
  - P leaves the monitor immediately after executing signal, Q is resumed

Solution to Dining Philosophers

```
monitor DiningPhilosophers
  {
    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);
     }

      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.)

- Each philosopher *i* invokes the operations pickup() and putdown() in the following sequence:

```
DiningPhilosophers.pickup (i);

   EAT

DiningPhilosophers.putdown (i);
```

- No deadlock, but starvation is possible

Monitor Implementation Using Semaphores

- Variables

```
semaphore mutex;  // (initially  = 1)
semaphore next;   // (initially  = 0)
int next_count = 0;
```

- Each procedure ***F*** will be replaced by

```
wait(mutex);
    …
    body of F;
    …
if (next_count > 0)
  signal(next)
else
  signal(mutex);
```

- Mutual exclusion within a monitor is ensured

Monitor Implementation – Condition Variables

- For each condition variable ***x***, we have:

```
semaphore x_sem; // (initially = 0)
int x_count = 0;
```

- The operation x.wait can be implemented as:

```
x-count++;
if (next_count > 0)
    signal(next);
else
    signal(mutex);
wait(x_sem);
x-count--;
```

Monitor Implementation (Cont.)

- The operation x.signal can be implemented as:

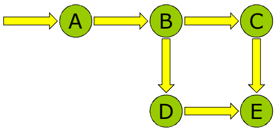
```
if (x-count > 0) {
    next_count++;
    signal(x_sem);
    wait(next);
    next_count--;
}
```

Homework

- Reading
  - Chapter 6
- Exercise
  - See course website

Pop Quiz

- Given a set of five processes: A, B, C, D, and E, write pseudo-codes for each process to synchronize the order in which they are executed, as shown in the following graph:



- That is, process A must finish executing before B starts, process B must finish before C or D start, and process C and D must finish before process E starts.