I.-C. Lin, Assistant Professor. Textbook: Operating System Principles 7ed

CHAPTER 6: SYNCHRONIZATION

Module 6: Synchronization

- Background
- The Critical-Section Problem
- Peterson's Solution
- Synchronization Hardware
- Semaphores
- Classic Problems of Synchronization
- Monitors
- Synchronization Examples
- Atomic Transactions

Background

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

- 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 count that keeps track of the number of full buffers. Initially, count 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 and put in nextProduced
                while (count == BUFFER_SIZE)
                        ; // do nothing
                buffer [in] = nextProduced;
                in = (in + 1) % BUFFER_SIZE;
                count++;
    }
```

Consumer

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

Race Condition

- count++ could be implemented as

  ```
  register1 = count
  register1 = register1 + 1
  count = register1
  ```

- count-- could be implemented as

  ```
  register2 = count
  register2 = register2 - 1
  count = register2
  ```

Race Condition (cont.)

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

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

- **Race condition**
  - The situation where several processes access and manipulate shared data concurrently. The final value of the shared data depends upon which process finishes last.

- To prevent race conditions, concurrent processes must be **synchronized**.

- *n* processes all competing to use some shared data
- Each process has a code segment, called *critical section*, in which the shared data is accessed.
- Problem – ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section.

Solution to Critical-Section Problem

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.

Solution to Critical-Section Problem

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

Initial Attempts to Solve The Problem

- Only 2 processes, P_0 and P_1
- General structure of process P_i (other process P_j)

```
do {
        entry section
                critical section
        exit section
                remainder section
} while (1);
```

- Processes may share some common variables to synchronize their actions.

Algorithm 1

- Shared variables:
  - **int turn**;
    initially **turn = 0**
  - **turn == i** ⇒ P_i can enter its critical section
- Process P_i

```
do {
        while (turn != i) ;
                critical section
        turn = j;
                remainder section
} while (1);
```

- Satisfies mutual exclusion, but not progress

Algorithm 2

- Shared variables
  - **boolean flag[2]**;
    initially **flag [0] = flag [1] = false.**
  - **flag [i] = true** ⇒ P_i ready to enter its critical section
- Process P_i

```
do {
        flag[i] := true;
        while (flag[j]) ;
    critical section
        flag [i] = false;
            remainder section
} while (1);
```

- Satisfies mutual exclusion, but not progress requirement.

Peterson's Solution

- Two process solution
- Assume that the LOAD and STORE instructions are atomic; that is, cannot be interrupted.
- 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);
```

Synchronization Hardware

- Many systems provide hardware support for critical section code
- 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-interruptable
  - Either test memory word and set value
  - Or swap contents of two memory words

TestAndndSet 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);
```

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);
```

Semaphore

- Synchronization tool that does not require busy waiting
- Semaphore *S* – integer variable
- Two standard operations modify S: wait() and signal()
  - Originally called P() and V()
- Less complicated

Semaphore

- Can only be accessed via two indivisible (atomic) operations
  - wait (S) {

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

  - signal (S) {

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

Semaphore as General Synchronization Tool

- Counting semaphore – integer value can range over an unrestricted domain
- Binary semaphore – integer value can range only between 0 and 1; can be simpler to implement
  - Also known as mutex locks
- Can implement a counting semaphore S as a binary semaphore
- Provides mutual exclusion
  - Semaphore S; // initialized to 1
  - wait (S);

```
        Critical Section
    signal (S);
```

Semaphore Implementation

- Must guarantee that no two processes can execute wait () and signal () on the same semaphore at the same time
- Thus, implementation becomes the critical section problem where the wait and signal code are placed in the crtical section.
  - Could now have busy waiting in critical section implementation
    - But implementation code is short
    - Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution.

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.

Semaphore Implementation with no Busy waiting (Cont.)

- Implementation of wait:

```
wait (S){
        value--;
        if (value < 0) {
                add this process to waiting queue
          block(); }
}
```

- Implementation of signal:

```
Signal (S){
         value++;
          if (value <= 0) {
                  remove a process P from the waiting queue
         wakeup(P); }
 }
```

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

P_0	P_1
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

- 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

    wait (empty);
    wait (mutex);

        //  add the item 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

    signal (mutex);
    signal (empty);

        //  consume the removed item

} 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.
- Shared Data
  - Data set
  - Semaphore mutex initialized to 1.
  - Semaphore wrt initialized to 1.
  - Integer readcount initialized to 0.

Readers-Writers Problem (Cont.)

- The structure of a writer process

```
do  {
    wait (wrt) ;

     //    writing is performed

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

Readers-Writers Problem (Cont.)

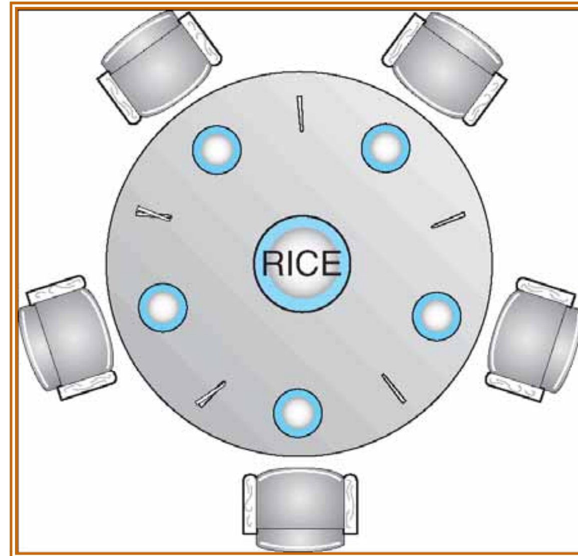
- The structure of a reader process

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

         // reading is performed

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

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) ;
```

Problems with Semaphores

- Correct use of semaphore operations:
  - signal (mutex) …. wait (mutex)
  - wait (mutex) … wait (mutex)
  - Omitting of wait (mutex) or signal (mutex) (or both)

Monitors

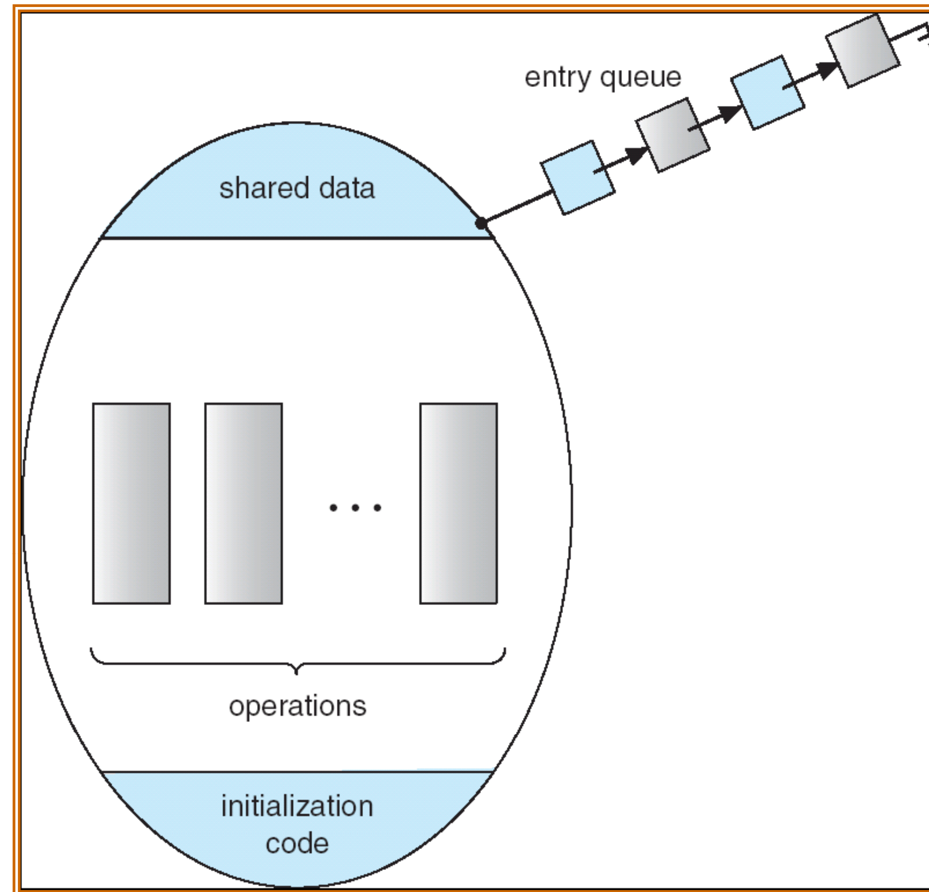
- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- Only one process may be active within the monitor at a time

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

   procedure Pn (…) {……}

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

Schematic View of a Monitor

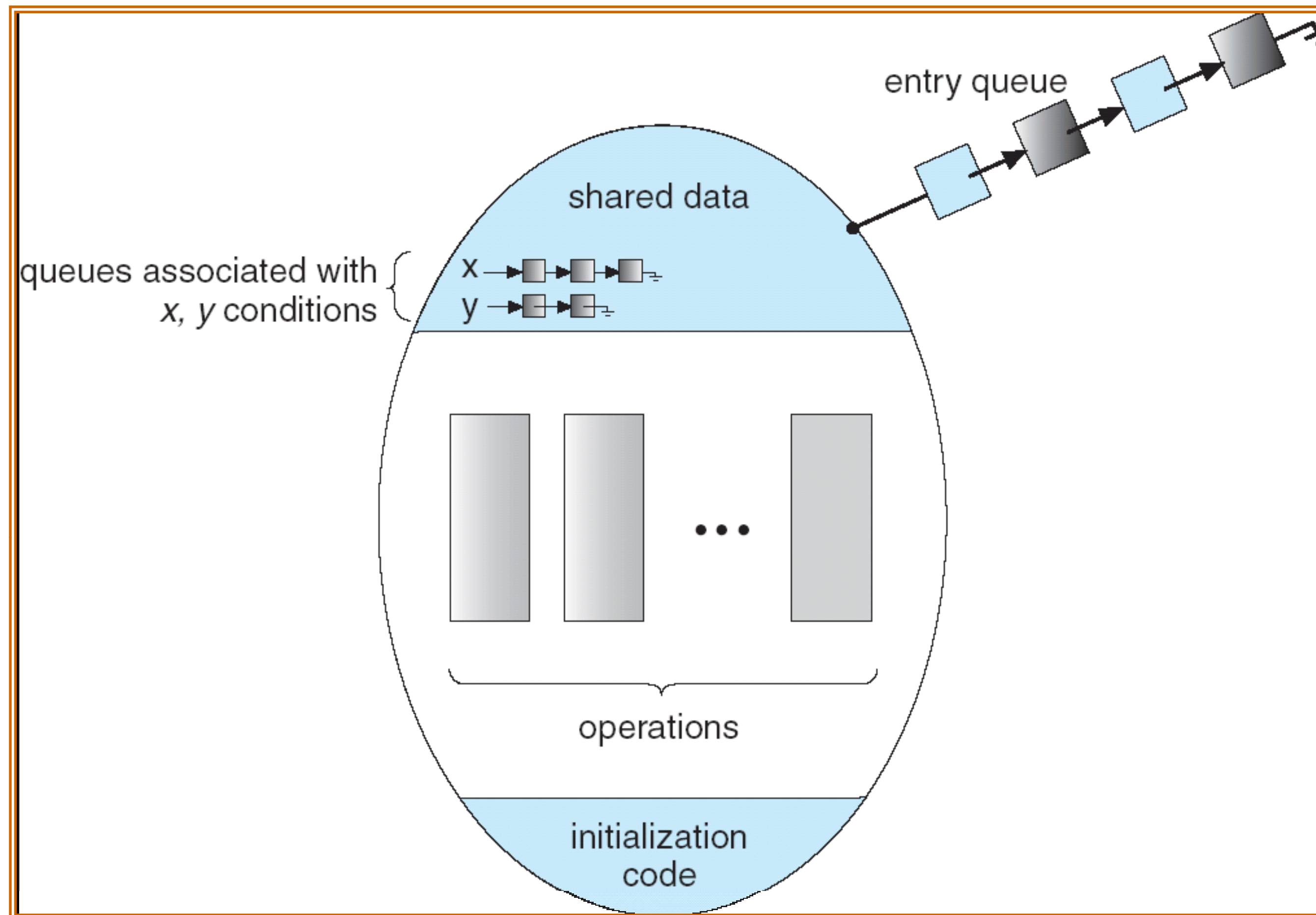


Condition Variables

- condition x, y;

- Two operations on a condition variable:
  - x.wait () – a process that invokes the operation is suspended.
  - x.signal () – resumes one of processes (if any) invoked x.wait ()

Monitor with Condition Variables



Condition Variables with Semaphores

```
mutex = 1; next=0; next_count=0;  x_sem = 0; x_count = 0;
```

```
Monitor Procedure F
{
mutex = 1;
wait(mutex);
Body of F
if(next_count > 0)
    signal(next);
else
    signal(mutex);
}
```

```
x.wait()
{
x_count++;
If(next_count > 0)
    signal(next);
else
    signal(mutex);
wait(x_sem);
xcount--;
}
```

```
x.signal()
{
next_count++;
signal(x_sem);
wait(next);
next_count--;
}
```

Condition Variables with Semaphores

```
Monitor Example {
    Proc1() {
        P1A;
        x.wait();
        P1B;
    }

    Proc2() {
        P2A;
        x.signal();
        P2B;
    }
```

Proc1		Proc2	
	wait(mutex)		
P1A			
			wait(mutex) {suspend}
x.wait()			
	signal(mutex)		
	wait(x_sem) {suspend}		
		P2A	
		x.signal()	
			signal(x_sem)
			wait(next) {suspend}
P1B			
	signal(next)		
		P2B	
			signal(mutex)

Solution to Dining Philosophers

dp.pickup (i)

 …..

Eat

….

dp.putdown(i)

Solution to Dining Philosophers

```
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);
```

Solution to Dining Philosophers (cont)

```
    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;
    }
}
```

- *Conditional-wait* construct: **x.wait(c);**
  - **c** – integer expression evaluated when the **wait** operation is executed.
  - value of **c** (a *priority number*) stored with the name of the process that is suspended.
  - when **x.signal** is executed, process with smallest associated priority number is resumed next.

- Solaris
- Windows XP
- Linux
- Pthreads

Solaris Synchronization

- Implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing

- Uses adaptive mutexes for efficiency when protecting data from short code segments
  - Spinlock <-> block

- Uses condition variables and readers-writers locks when longer sections of code need access to data
- Uses turnstiles to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock
  - A queue structure according to a priority inheritance protocol

- Uses interrupt masks to protect access to global resources on uniprocessor systems
- Uses spinlocks on multiprocessor systems
- Also provides dispatcher objects which may act as either mutexes and semaphores
  - Signaled, nonsignaled
- Dispatcher objects may also provide events
  - An event acts much like a condition variable

- Linux:
  - disables interrupts to implement short critical sections

- Linux provides:
  - semaphores
  - spin locks

Pthreads Synchronization

- Pthreads API is OS-independent
- It provides:
  - mutex locks
  - condition variables

- Non-portable extensions include:
  - read-write locks
  - spin locks

- Transaction
  - a collection of instructions that performs a single logical function.
- How to preserve atomicity despite the possibility of failures with in a computer system?
- committed : complete its execution successfully.
- Aborted: cease due to some logical error.

Log-based Recovery

- Write-ahead logging
  - Each log record describes a single operation of a transaction write.
  - (Transaction Name, Data Item Name, Old Value, New Value).
  - <Ti starts>……..<Ti commits>
  - Penalty : two physical writes are required for every logical write request.

  - Undo and redo

- To avoid the time-consuming searching process, checkpoints are used.

- A system periodically performs check points .
  - Output all log records onto stable storages.
  - Output all modified data to the stable storage.
  - Output a log record <checkpoint> onto stable storage.

Serializability

- Serializability
  - concurrent execution of transactions = transactions executed serially
  - A serial schedule: a schedule where each transaction is executed atomically.
- The execution result of a nonserial schedule is not necessarily incorrect.

- *Oi* and *Oj* *conflict*
  - access the same data item and at least one of these operations is a write operation.

Serializability

T0	T1
Read(A)	
Write(A)	
Read(B)	
Write(B)	
	Read(A)
	Write(A)
	Read(B)
	Write(B)

T0	T1
Read(A)	
Write(A)	
	Read(A)
	Write(A)
Read(B)	
Write(B)	
	Read(B)
	Write(B)

Conflict serializable

Serializability (cont.)

- If O_i and O_j are operations of different transactions and do not conflict, we can swap O_i, O_j for a new schedule S'.

- Timestamp protocol
  - A fixed timestamp TS(T_i): system clock or logical counter.
  - W-timestamp(Q): the largest timestamp of write(Q) (successfully)
  - R-timestamp(Q): the largest timestamp of read(Q) (successfully)

  - For read, if Ts(Ti) < W-timstamp(Q): read is rejected, Ti roll back
  - For write, if Ts(Ti) < R-timestamp(Q): write is rejected, Ti roll back
  - For write, if Ts(Ti) < W-timestamp(Q): write is rejected, Ti roll back

END OF CHAPTER 6