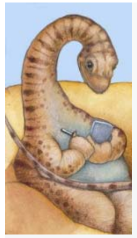
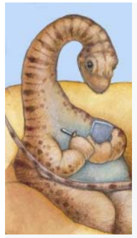
Chapter 6: Process Synchronization



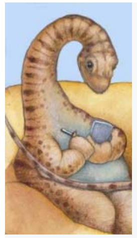
Module 6: Process Synchronization

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



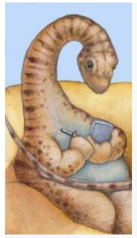
Background

- 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 producer-consumer 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.



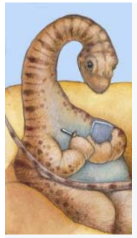
Producer

```
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

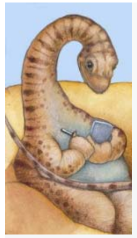
```
register1 = count
register1 = register1 + 1
count = register1
```

- count-- could be implemented as

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

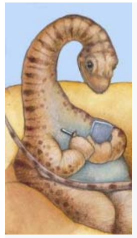
- Consider this execution interleaving with “count = 5” initially:

```
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}
```



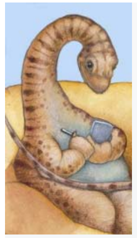
Solution to Critical-Section Problem

1. Mutual Exclusion - If process P_i is executing in its critical section ***with respect to a particular resource***, 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



Synchronization Hardware

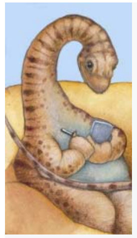
- Many systems provide hardware support for critical section code
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems that use this technique are not broadly scalable
- Modern machines provide special atomic hardware instructions
    - Atomic = non-interruptable
  - Either test memory word and set value ***in one atomic operation***
  - Or swap contents of two memory words ***in one atomic operation***



TestAndSet Instruction

- Definition of the functionality:

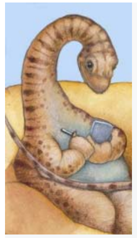
```
boolean TestAndSet (boolean *target)
 {
      boolean rv = *target;
      *target = TRUE;
      return rv:
 }
```



Solution using TestAndSet

- Shared boolean variable **lock**, initialized to false.
- Solution:

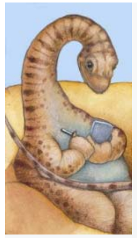
```
    do {
       while ( TestAndSet (&lock ))
                  ;   /* do nothing */

           //    critical section

       lock = FALSE;

           //      remainder section

    } while ( TRUE);
```



Swap Instruction

- Definition of the functionality:

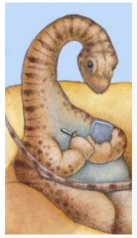
```
void Swap (boolean *a, boolean *b)
 {
      boolean temp = *a;
      *a = *b;
      *b = temp:
 }
```



Solution using Swap

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

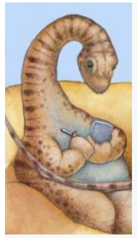
```
do {
      key = TRUE;
       while ( key == TRUE)
             Swap (&lock, &key );

               //    critical section

        lock = FALSE;

             //      remainder section

     } while ( TRUE);
```



Semaphore

- We need a 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() [Proberen(test) and Verhogen(increment) in Dutch]
- Less complicated than using TestAndSet and Swap
- A semaphore can only be accessed via two indivisible (atomic) operations
- Definition of the functionality of wait() and signal()

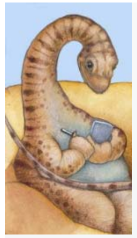
```
● wait (S) {
        while S <= 0
            ; // no-op
          S--;
   }
● signal (S) {
     S++;
  }
```



Semaphore as General Synchronization Tool

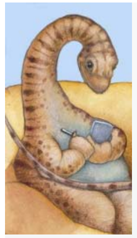
- Counting semaphore – integer value can range over an unrestricted domain
- Binary semaphore – integer value can range only between 0 and 1; can be simpler to implement
  - Also known as mutex locks
- Provides mutual exclusion
  - Semaphore S;   // initialized to 1
  - wait (S);

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



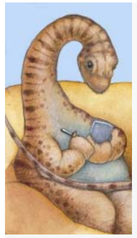
Semaphore Implementation

- The implementation of semaphores in the kernel must guarantee that no two processes can execute wait () and signal () on the same semaphore at the same time – i.e. that they are atomic
- The previous definitions for wait() and signal() required busy waiting over the integer values
- Busy waiting is almost never a good idea unless the system is structured so that it happens infrequently
- 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:
  - process control block for the waiting process
  - pointer to next entry in the queue
- 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.



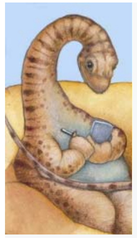
Semaphore Implementation with no Busy waiting (Cont.)

- 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);  }
}
```

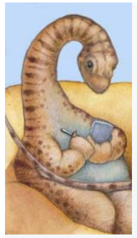


Deadlock and Starvation

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

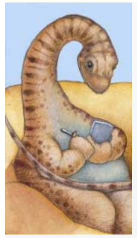
```
        P0                          P1
    wait (S);                   wait (Q);
    wait (Q);                   wait (S);
        .                           .
        .                           .
        .                           .
    signal  (S);                signal (Q);
    signal (Q);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

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



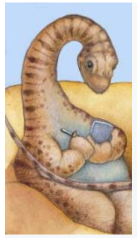
Classical Problems of Synchronization

- Bounded-Buffer Problem
- Readers and Writers Problem (in textbook)
- Dining-Philosophers Problem (in textbook)



Bounded-Buffer Problem

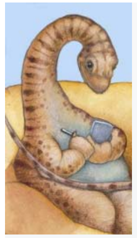
- *N* buffers, each can hold one item
- Semaphore mutex initialized to the value 1
- Semaphore full initialized to the value 0
- Semaphore empty initialized to the value N.



Bounded Buffer Problem (Cont.)

- The structure of the producer process

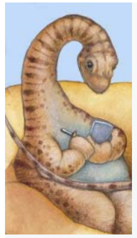
```
do {

        //   produce an item

    wait (empty);
    wait (mutex);

       //  add the item to the  buffer

     signal (mutex);
     signal (full);
  } while (true);
```



Bounded Buffer Problem (Cont.)

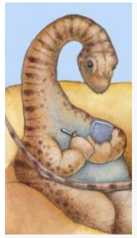
- The structure of the consumer process

```
do {
    wait (full);
    wait (mutex);

        //  remove an item from  buffer

     signal (mutex);
     signal (empty);

         //  consume the removed item

} while (true);
```



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)

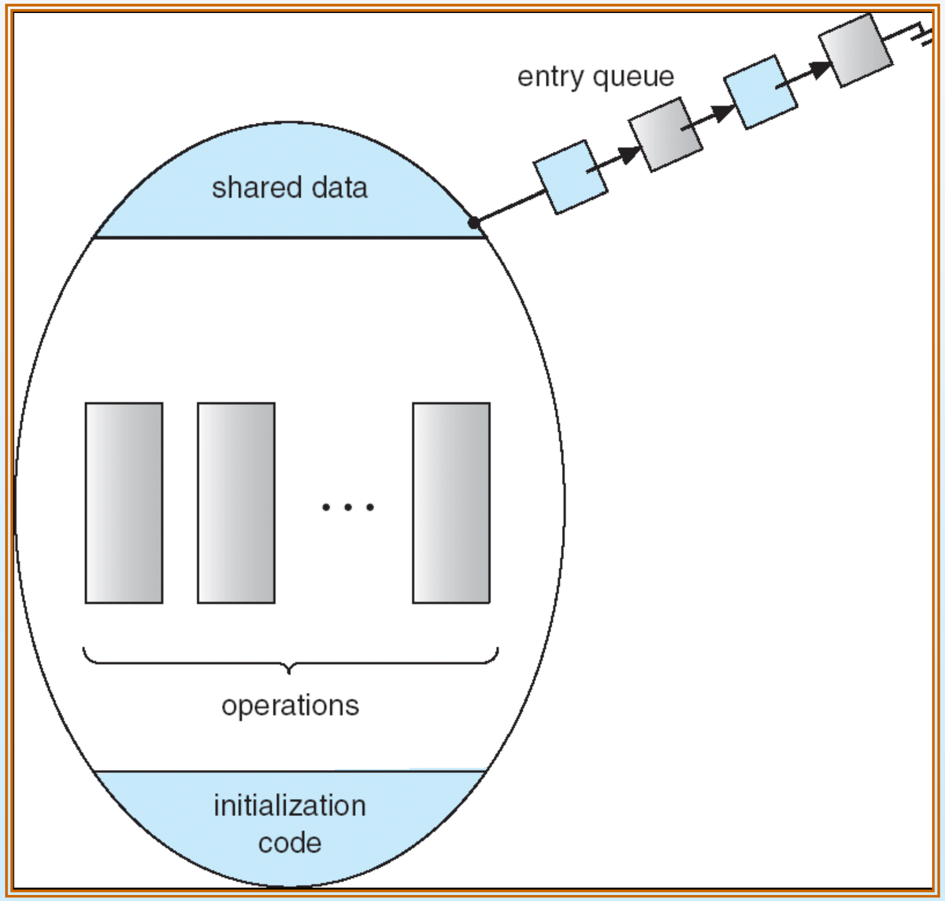


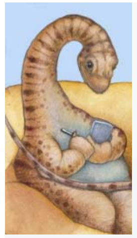
Monitors

- 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

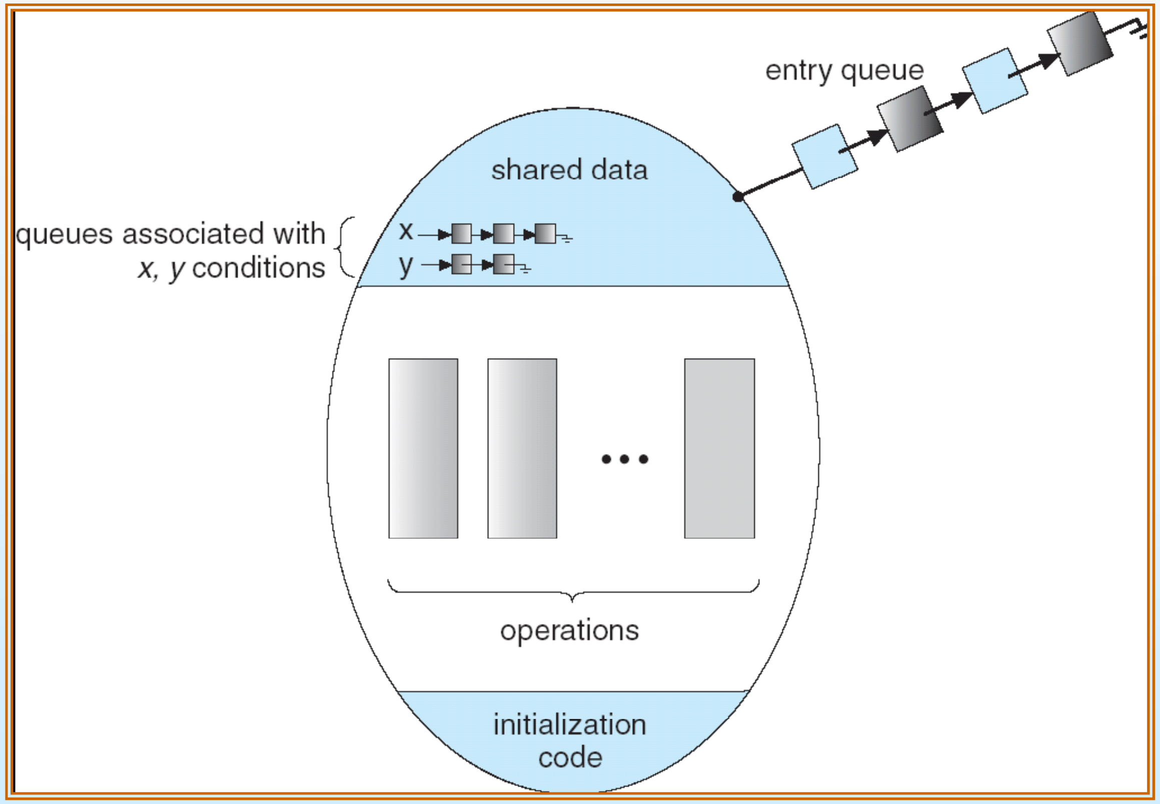


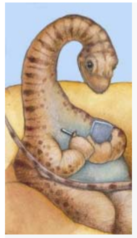


Condition Variables

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

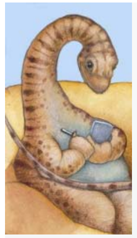
Monitor with Condition Variables





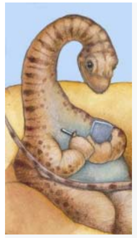
Linux Synchronization

- 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



Java Synchronization

- Monitors using the ***synchronized*** keyword
- Condition variables using wait(), notify() and notifyAll()