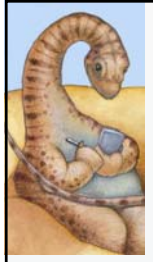
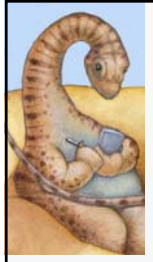
Chapter 6: Synchronization



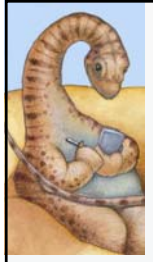
Module 6: Synchronization

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



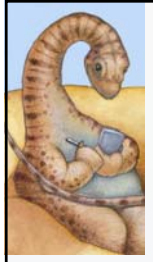
Background

- 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.



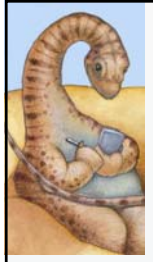
Producer

```
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

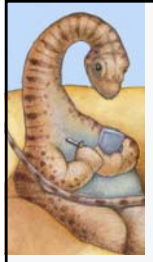
```
register1 = count
register1 = register1 + 1
count = register1
```

- count-- could be implemented as

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

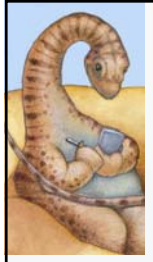
- Consider this execution interleaving with “count = 5” initially:

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}



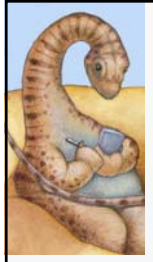
Requirements to Critical-Section Problem Solutions

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



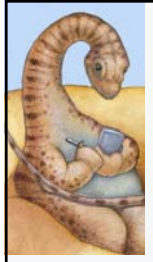
Preemptive and Nonpreemptive Kernel

- Preemptive kernel
  - Allow a kernel-mode process to be preempted
  - Subjected to race condition
  - Suitable for real-time programming, more responsive
  - Linux 2.6
  - Solaris, IRIX
- Nonpreemptive kernel
  - A kernel-mode process will run until it exists kernel mode, blocks, or voluntarily yields
  - Windows XP, 2000
  - Linux < 2.6



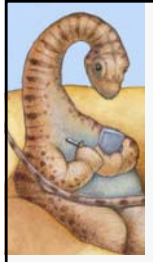
Peterson's Solution

- Software solution
- Because of the way modern computer arch perform basic machine-language instructions, such as load and store, this solution may not work correctly.



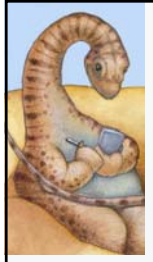
Peterson's Solution (Cont.)

- 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);
```

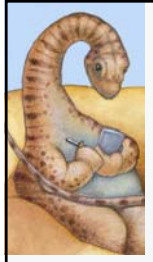


Do we really need all this?

- Gain access to the shared variable and lock out the other process.

```
bool locked = FALSE;

P0, P1:
        while (locked);
        locked = TRUE;
           CRITICAL SECTION
        lock = FALSE;
           REMAINDER SECTION
```

violates mutual exclusion (both test, set to TRUE and enter critical section)



Do we really need all this? (Cont.)

- Take turns using the shared variable.

```
turn = 0;
```

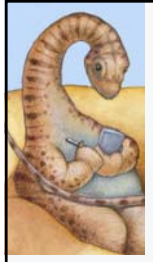
P0:

```
while (turn == 1);
  CRITICAL SECTION
turn = 1;
  REMAINDER SECTION
```

P1:

```
while (turn == 0);
  CRITICAL SECTION
turn = 0;
 REMAINDER SECTION;
```

violates progress condition (due to strict alternation)



Do we really need all this? (Cont.)

- Announce intensions and check to see if the other process is using the variable.

```
bool flag[2];
```

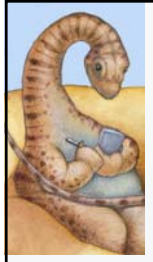
P0:

```
while (flag[1]);
flag[0] = TRUE;
   CRITICAL SECTION
flag[0] = FALSE;
```

P1:

```
while (flag[0]) do;
flag[1] = TRUE;
   CRITICAL SECTION
flag[1] = FALSE;
```

violates mutual exclusion (both test and find it FALSE, both set flags to TRUE and enter critical section)



Do we really need all this? (Cont.)

- Announce intensions and check to see if the other process is using the variable.

```
bool flag[2];
flag[0] = flag[1] = FALSE;
```

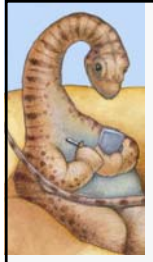
P0:

```
flag[0] = TRUE;
while (flag[1]);
   CRITICAL SECTION
flag[0] = FALSE;
```

P1:

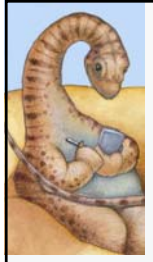
```
flag[1] = TRUE;
while (flag[0]) do;
   CRITICAL SECTION
flag[1] = FALSE;
```

violates progress condition (when both set flags to TRUE)



Multi-Process Mutual Exclusion

- n processes, numbered 0...n-1
- Bakery algorithm: when entering the critical section
  - Take a number and wait
  - Can't guarantee that numbers are distinct
    - Process with lowest process-id goes first

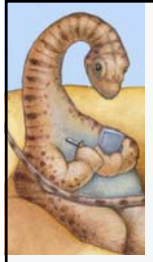


Bakery Algorithm

```
bool choosing[n];          // initially false
int number[n];             // initially 0
/* process i */
choosing[i] = true;
number[i] = max(number[0], ..., number[n]) + 1;
choosing[i] = false;
for(j = 0; j != i; j++) {
        while(choosing[j]);              // busy wait
        while(number[j] && number[j] <= number[i]);
}
for(j = i + 1; j < n; j++) {
        while(choosing[j]);
        while(number[j] && number[j] < number[i]);
}
        CRITICAL SECTION
number[i] = 0;
        REMAINDER SECTION
```

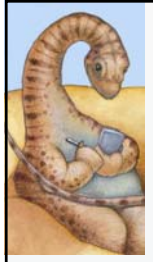
Pick a number

Busy wait



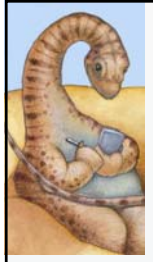
Problems with Software Solutions

- Solutions are complex and awkward
- Too much time spent in the busy wait



Synchronization Hardware

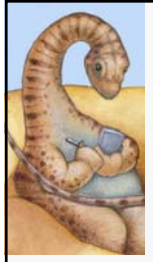
- Many systems provide hardware support for critical section code
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
- Modern machines provide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



TestAndndSet Instruction

- Definition:

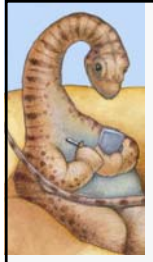
```
bool TestAndSet (bool *target)
{
    bool rv = *target;
    *target = TRUE;
    return rv:
}
```



Solution using TestAndSet

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

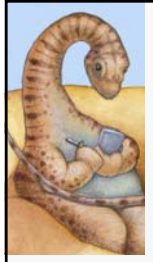
```
do {
  while ( TestAndSet (&lock ))
            ;   /* do nothing
     CRITICAL SECTION
  lock = FALSE;
     REMAINDER SECTION
 } while ( TRUE);
```



Swap Instruction

- Definition:

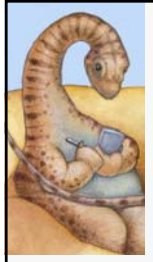
```
void Swap (bool *a, bool *b)
{
    bool temp = *a;
    *a = *b;
    *b = temp:
}
```



Solution using Swap

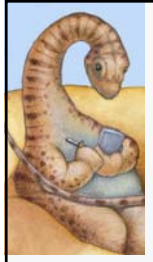
- Shared Boolean variable lock initialized to FALSE; Each process has a local Boolean variable key.
- Solution:

```
do {
    key = TRUE;
    while ( key == TRUE)
        Swap (&lock, &key );
      CRITICAL SECTION
    lock = FALSE;
      REMAINDER SECTION
} while ( TRUE);
```



So far

- Solution 1: Disable interrupts
  - big hammer
- Solution 2: busy wait with shared variables
  - 2a: Peterson's solution
    - complex but no hardware support
    - works on virtually all uniprocessors
  - 2b: TestAndSet
    - simpler but assumes TestAndSet (or equiv) instruction
    - works even on multiprocessors
  - Can be generalized to > 2 threads



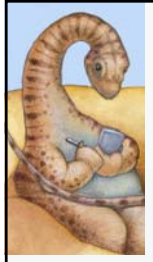
Semaphore (Dijkstra1965)

- Slightly more abstract primitives
- A semaphore is a non-negative integer variable with two atomic operations: wait() and signal()
  - Originally called P() and V(), a.k.a., Passeren and Verhogen
- Can only be accessed via two indivisible (atomic) operations
  - wait (S) decrements S by 1 if possible, otherwise it waits until S becomes positive, then decrements it; i.e. (idea only, not actual implementation),

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

  - signal (S) increments S by one; i.e. (idea only, not actual implementation),

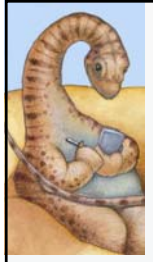
```
signal (S) {
   S++;
}
```



Semaphore as General Synchronization Tool

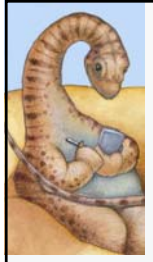
- Counting semaphore – integer value can range over an unrestricted domain
- Binary semaphore – integer value can range only between 0 and 1; can be simpler to implement
  - Also known as mutex locks
- Can implement a counting semaphore S as a binary semaphore
- Provides mutual exclusion

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



Semaphore Implementation

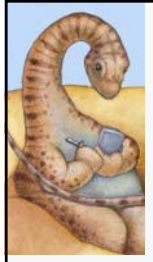
- Must guarantee that no two processes can execute wait () and signal () on the same semaphore at the same time
- Thus, implementation becomes the critical section problem where the wait and signal code are placed in the critical section.
  - Could now have busy waiting in critical section implementation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



Implementing Semaphores

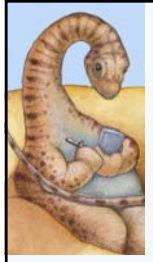
- Since semaphores is actually a critical section problem, it may be implemented by various approaches.
- For example, TestAndSet may be used to implement a binary semaphore as following

```
void wait(S) {
        while(TestAndSet(&S));
}

void signal(S) {
        S = 0;
}
```



Are we done yet?

- Not quite!
- The shared variable solutions require busy waiting
  - bad: this can be inefficient - the locked out thread needs to be running
  - worse: can deadlock if the waiting process has higher priority in the scheduler
- We'd usually prefer a blocking solution

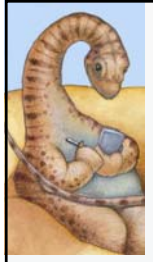


Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated waiting queue.
- Each semaphore has two data items:

```
typedef struct {
    int value;
    struct process *list;  // pointer to next record in the list
} semaphore;
```

- Two operations:
  - block – place the process invoking the operation on the appropriate waiting queue associated with the semaphore.
  - wakeup – remove one of processes in the waiting queue and place it in the ready queue.



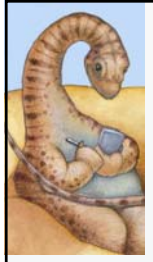
Semaphore Implementation with no Busy waiting (Cont.)

- Implementation of wait:

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

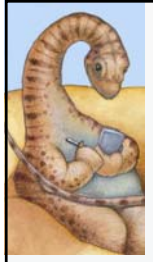
- Implementation of signal:

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



Semaphore Implementation with no Busy waiting (Cont.)

- No two process can execute wait() and signal() on the same semaphore at the same time – a critical section problem (semaphore is critical section).
  - Can solve this by simply disabling interrupt on uniprocessors.
- Note that this implementation does not completely eliminate busy waiting.
  - But remove busy waiting from the entire section, i.e., entire wait() or signal(), to the critical sections of wait() or signal(), i.e., semaphore itself.
  - Thus, the critical section is almost never occupied, and busy waiting occurs rarely, and then for only a short time

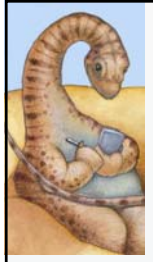


Deadlock and Starvation

- 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

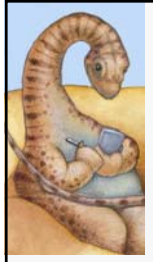
P_0	P_1
wait (S);	wait (Q);
wait (Q);	wait (S);
.	.
.	.
.	.
signal (S);	signal (Q);
signal (Q);	signal (S);

- Starvation – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.
  - E.g., adding and removing processes from the semaphore queue in LIFO order



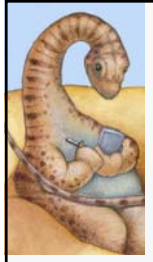
Classical Problems of Synchronization

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem



Bounded-Buffer Problem

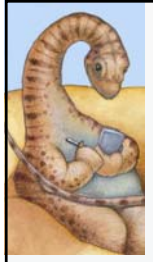
- *N* buffers, each can hold one item
- Semaphore mutex initialized to the value 1
  - used to provide mutual exclusion for accesses to the buffer pool
- Semaphore full initialized to the value 0
  - used to count number of full buffers
- Semaphore empty initialized to the value N.
  - used to count number of empty buffers



Bounded Buffer Problem (Cont.)

- The structure of the producer process

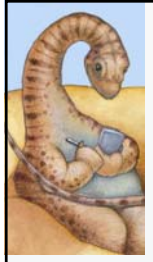
```
do {

        //   produce an item

    wait (empty);          // wait if there is no empty buffer
    wait (mutex);

       //  add the item to the  buffer

     signal (mutex);
     signal (full);          // one more full buffer
  } while (true);
```



Bounded Buffer Problem (Cont.)

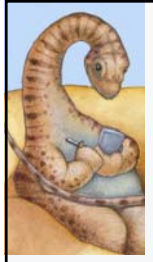
- The structure of the consumer process

```
do {
    wait (full);                // wait if there is no full buffer
    wait (mutex);

       //  remove an item from  buffer

     signal (mutex);
     signal (empty);        // one more empty buffer

       //  consume the removed item

} while (true);
```



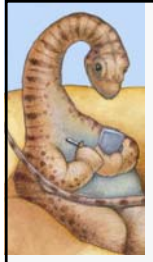
Readers-Writers Problem

- 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.
  - Has many variations, all involving priorities
    - Requires that no reader will be kept waiting unless a writer has already obtained write permission
    - Once a writer is ready, no new readers may start reading
  - Both may result in starvation.



Readers-Writers Problem (Cont.)

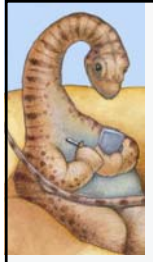
- Solution for the first reader-writer problem
- Shared Data
  - Data set
  - Semaphore mutex initialized to 1.
    - To ensure mutual exclusion access to readcount
  - Semaphore wrt initialized to 1.
  - Integer readcount initialized to 0.
    - To keep track of how many processes are reading the object



Readers-Writers Problem (Cont.)

- The structure of a writer process

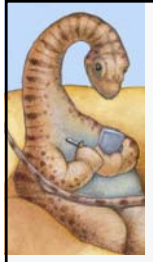
```
do {
    wait (wrt) ;

      //    writing is performed

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



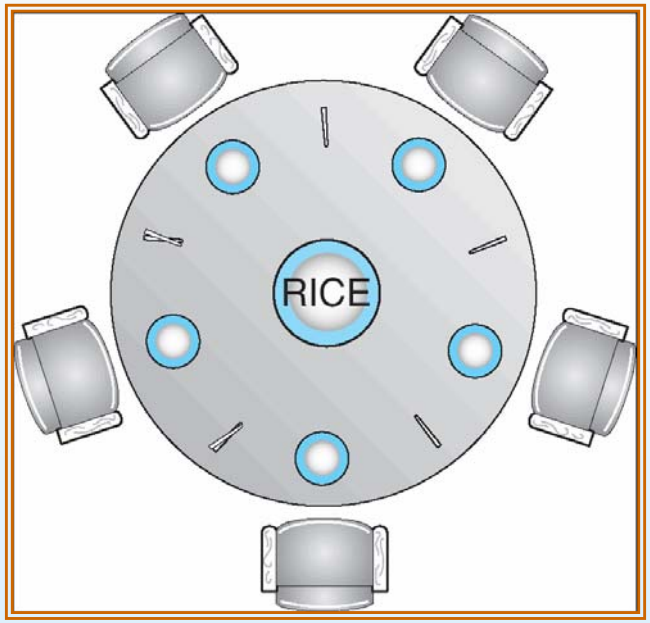
Readers-Writers Problem (Cont.)

- The structure of a reader process

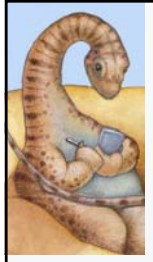
```
do  {
        wait (mutex) ;
        readcount ++ ;
        if (readercount == 1)  wait (wrt) ;
        signal (mutex)

            // reading is performed

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



Dining-Philosophers Problem



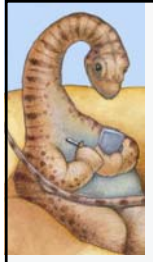
- Shared data
  - Bowl of rice (data set)
  - Semaphore chopstick [5] initialized to 1
- A simple representation of the need to allocate several resources among several processes in a deadlock-free and starvation-free manner.



Dining-Philosophers Problem (Cont.)

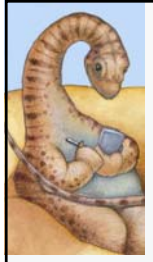
- The structure of Philosopher *i*:

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

          //  eat

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

          //  think

} while (true) ;
```



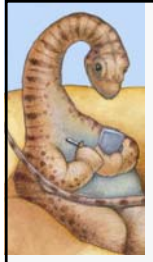
Dining-Philosophers Problem (Cont.)

- No two neighbors are eating simultaneously
  - Deadlock may occur
- Possible remedies
  - 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.
    - 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 pick up her right chopstick and then her left chopstick.



Problems with Semaphores

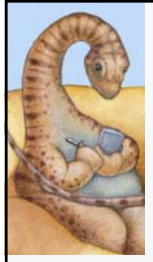
- Still too low-level
- Semaphore operations are scattered through code
- wait() and signal() are used in pairs: how do you know which pairs match?
  - Require correct use of semaphore operations:
    - signal (mutex) …CRITICAL SECTION… wait (mutex)
    - wait (mutex) …CRITICAL SECTION… wait (mutex)
    - Omitting of wait (mutex) or signal (mutex) (or both)
  - These errors may be discovered only if several processes are simultaneously active in their critical sections.
    - This situation may not always be reproducible – hard to debug



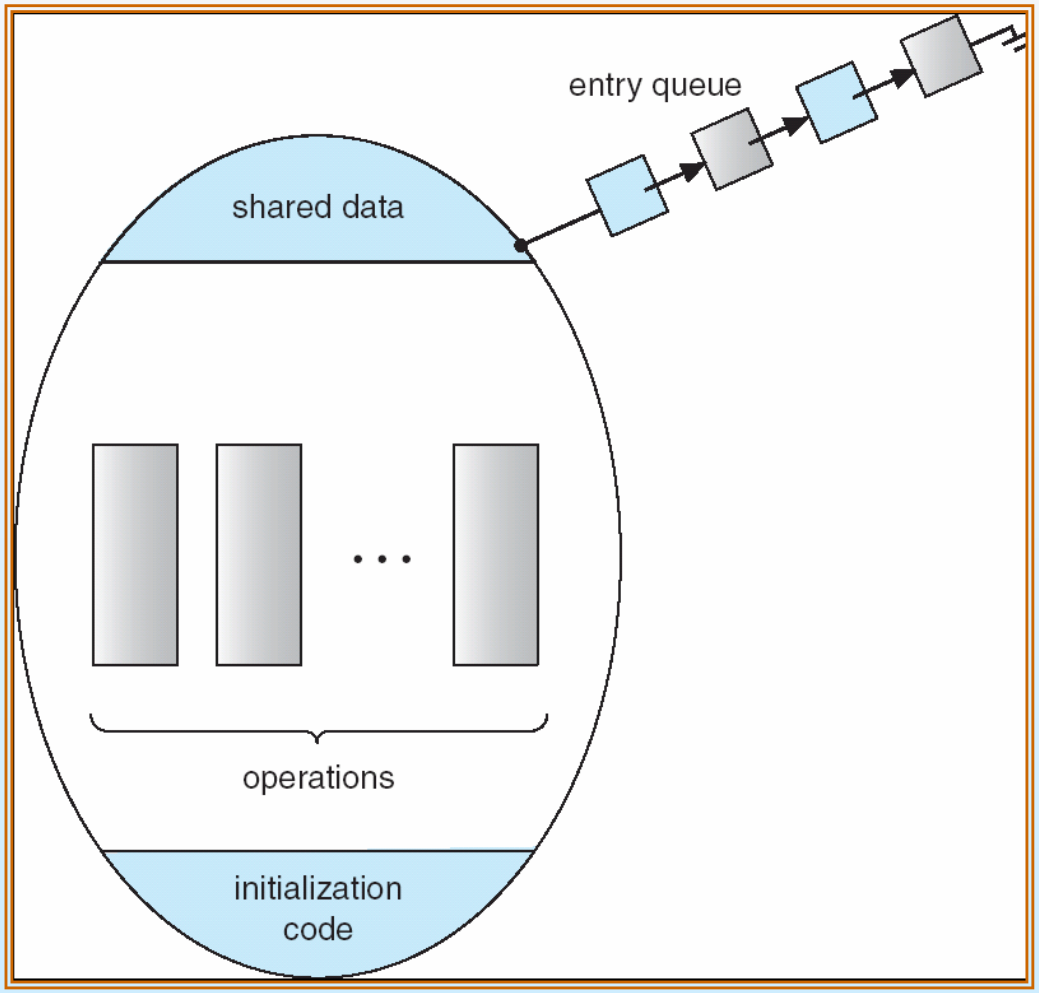
Monitors

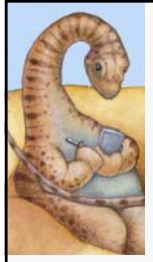
- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- Only one process may be active within the monitor at a time
  - The programmer does not need to code this synchronization constraint explicitly.

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

    procedure Pn (…) {……}

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



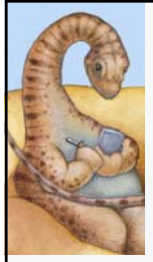
Schematic view of a Monitor



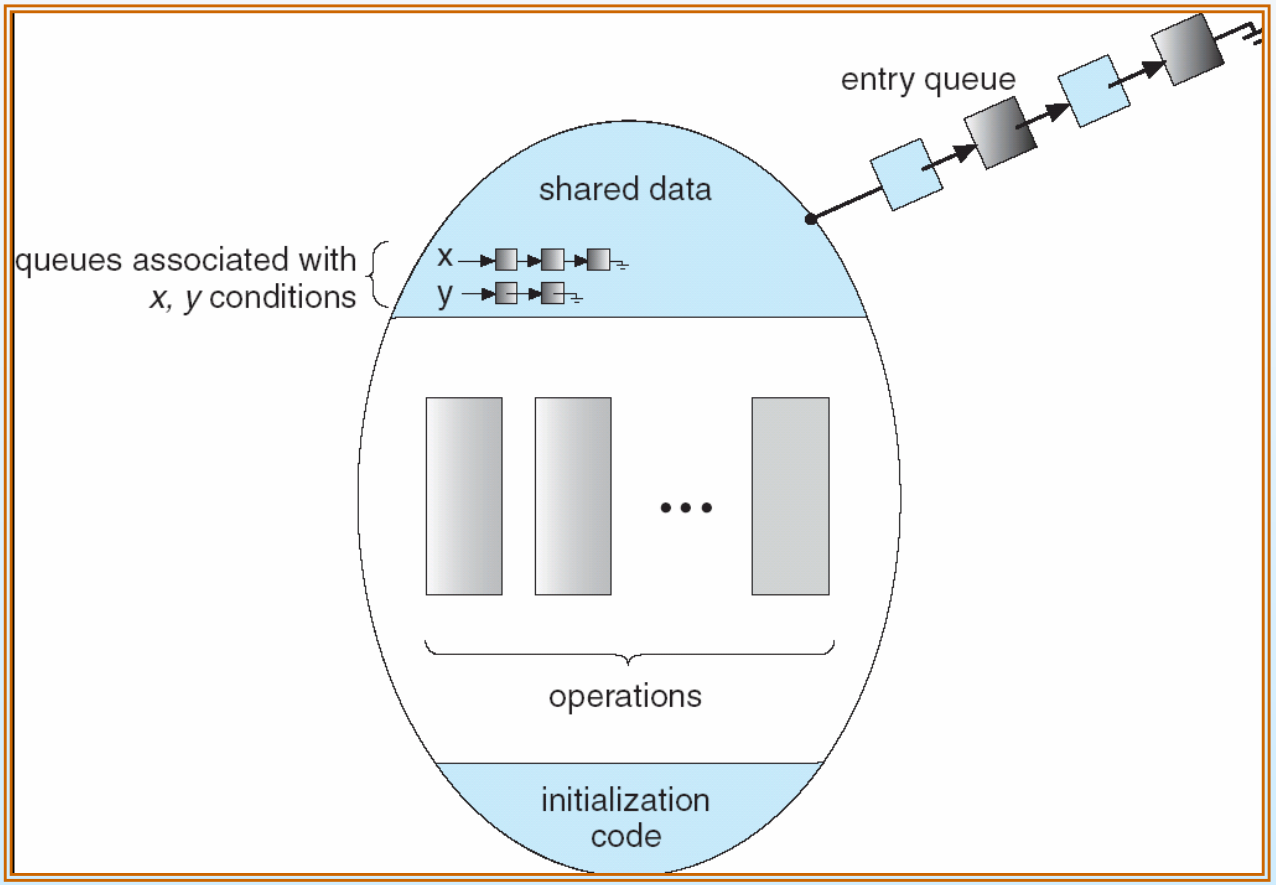


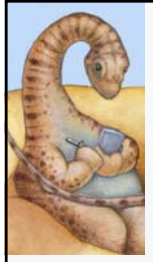
Condition Variables

- condition x, y;
  - Can be used by programmers who need tailor-made synchronization scheme
- Two operations on a condition variable:
  - x.wait () – a process that invokes the operation is suspended.
  - x.signal () – resumes one of suspended processes (if any) that invoked x.wait ().
    - If no process is suspended, then the signal () operation has no effect.
      - The state of x is the same as if the operation had never been executed.
    - Contrast this with the signal () operation associated with semaphores, which always affects the state of the semaphore.



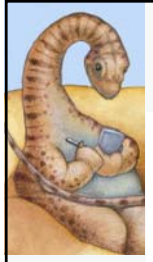
Monitor with Condition Variables



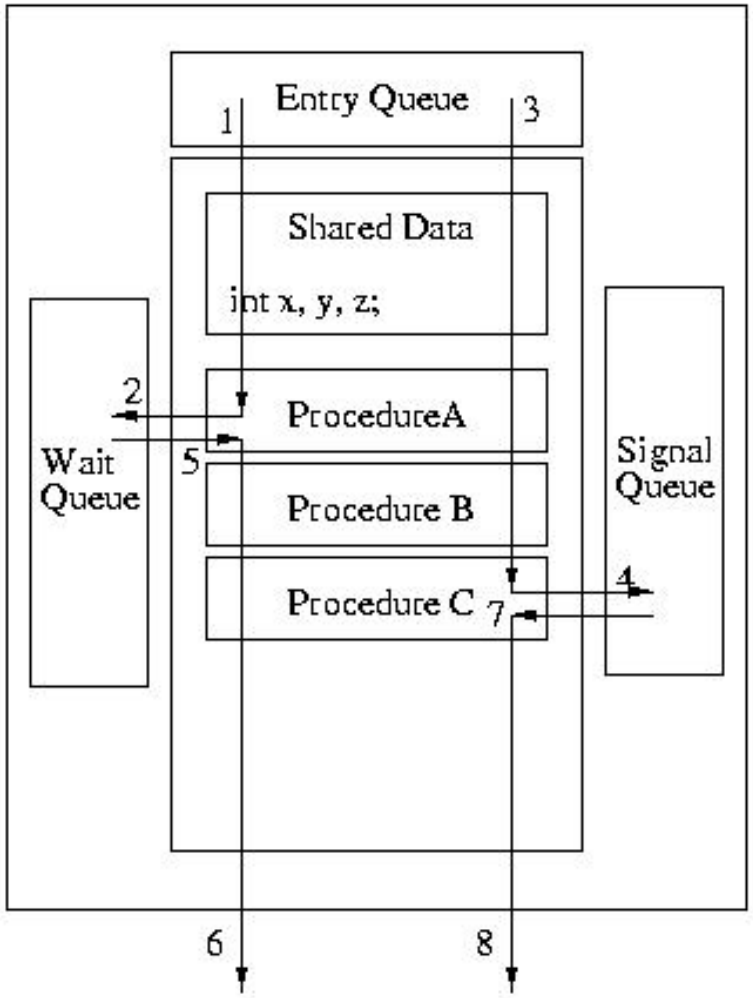


Condition Variables

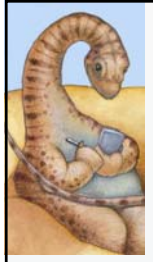
- Suppose Q associated with x is resumed when P invokes x.signal(). Two possibilities exist.
  - Note that only one process is active within the monitor
- Signal and wait: P either waits until Q leaves the monitor or waits for another condition.
- Signal and continue: Q either waits until P leaves the monitor or waits for another condition.



Hoare Semantics

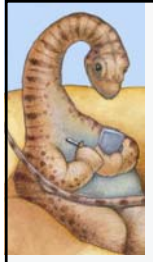


1. ThreadA enters the monitor
2. ThreadA waits for a resource
3. ThreadB enters the monitor
4. ThreadB signals on the resource and enters the signal queue
5. ThreadA re-enters the monitor
6. ThreadA leaves the monitor
7. ThreadB re-enters the monitor
8. ThreadB leaves the monitor
9. Another process can be admitted form the entry queue



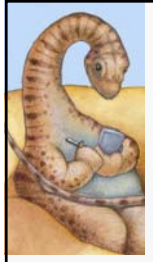
Solution to Dining Philosophers

```
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)

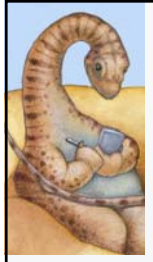
```
void test (int i) {
        if ( (state[(i + 4) % 5] != EATING) &&
        (state[i] == HUNGRY) &&
        (state[(i + 1) % 5] != EATING) ) {
            state[i] = EATING ;
             self[i].signal() ;
         }
 }

 initialization_code() {
       for (int i = 0; i < 5; i++)
          state[i] = THINKING;
}
}
```



Solution to Dining Philosophers (cont)

- The distribution of the chopsticks is controlled by the monitor dp.
- Philosopher i must invoke the operations pickup() and putdown() in the following sequence:

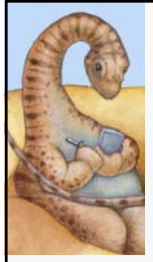
```
dp.pickup(i);
…
eat
…
dp.putdown(i);
```



Implementing a Monitor Using Semaphores

- A semaphore mutex is provided for the number of process allowed in monitor (initialized to 1, for example).
- A process must execute wait(mutex) before entering the monitor and must execute signal(mutex) after leaving the monitor.
- Another semaphore next is introduced on which the signaling processes may suspend themselves (initialized to 0).
  - To implement signal and wait semantic
- An integer next_count is used to count the number of processes suspended on next.
- Thus each operation F will be coded as

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



Implementing a Monitor Using Semaphores (Cont.)

- For each condition x, a semaphore x_sem and an integer variable x_count (both initialized to 0) will be used.
- x.wait():

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

- X.signal():

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

End of Chapter 6