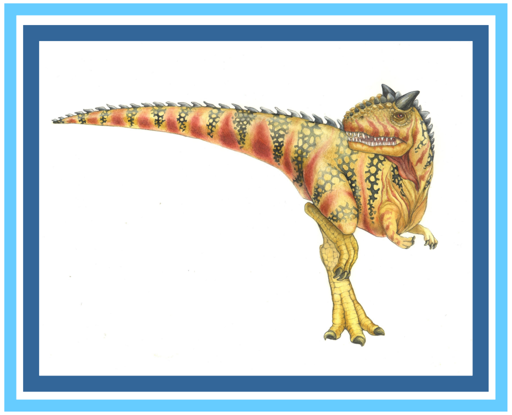
Chapter 5: Process Synchronization



Chapter 5: Process Synchronization

- Background
- The Critical-Section Problem
- Peterson’s Solution
- Synchronization Hardware
- Mutex Locks
- Semaphores
- Classic Problems of Synchronization
- Monitors
- Synchronization Examples
- Alternative Approaches

Objectives

- To present the concept of process synchronization.
- To introduce the critical-section problem, whose solutions can be used to ensure the consistency of shared data
- To present both software and hardware solutions of the critical-section problem
- To examine several classical process-synchronization problems
- To explore several tools that are used to solve process synchronization problems

Background

- Processes can execute concurrently
  - May be interrupted at any time, partially completing execution
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Illustration of the problem:
  Suppose that we wanted to provide a solution to the consumer-producer problem that **completely** fills the buffer. We can do so by having an integer `counter` that keeps track of the number of unconsumed elements. Initially, `counter` is set to 0. It is incremented by the producer after it produces a new element and is decremented by the consumer after it consumes an element.

Producer

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

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

Consumer

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

Race Condition

- `counter++` could be implemented as

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

- `counter--` could be implemented as

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

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

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

Critical Section Problem

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

Critical Section

- General structure of process ***P***$_i$

```
do {
    entry section

        critical section

    exit section

        remainder section

} while (true);
```

Solution to Critical-Section Problem

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, and only those processes that are not in their remainder section are considered.
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

Critical-Section Handling in OS

Two approaches depending on if kernel is preemptive or non-preemptive

- **Preemptive** – allows preemption of process when running in kernel mode
- **Non-preemptive** – runs until exits kernel mode, blocks, or voluntarily yields CPU
  - Essentially free of race conditions in kernel mode
- Preemptive kernel is more responsive (reduces the risk that one kernel-mode process occupies the CPU for a long time)
- Preemptive kernel is more suitable for real-time programming

Peterson's Solution

- Good algorithmic description of a solution to the problem
- 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`*** indicates that process $\mathbf{P_i}$ is ready!
- Assume that changes to variables *turn* and *flag* are **atomic**; that is, cannot be interrupted

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);
```

Peterson’s Solution (Cont.)

- Provable that all three CS requirements are met:

  1. Mutual exclusion is preserved

     `P`$_i$ enters CS only if:

     either **`flag[j] = false`** or **`turn = i`**

     `turn is either i or j but not both`

  2. Progress requirement is satisfied. The process in its remainder section will have its *flag* set to *false*; therefore, it cannot be selected. The other process will necessarily be selected if its *flag* is *true*.

  3. Bounded-waiting requirement is met. After executing its critical section, each process sets its *flag* to *false*, and if it tries to enter its critical section again, it will set *turn* to the other process number.

Synchronization Hardware

- Many systems provide hardware support for implementing the critical section code.
- All solutions below are based on the idea of **locking**
  - Protecting critical regions via locks
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems, because you have to pass a message to all processors
- Modern machines provide special atomic hardware instructions
    - **Atomic** = non-interruptible
  - Either test memory word and set value
  - Or swap contents of two memory words

Solution to Critical-section Problem Using Locks

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

test_and_set Instruction

Definition:

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

1. Executed atomically
2. Returns the original value of passed parameter
3. Set the new value of passed parameter to “TRUE”.

Solution using test_and_set()

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

```
do {
       while (test_and_set(&lock))
        ; /* do nothing */
            /* critical section */
    lock = false;
            /* remainder section */
 } while (true);
```

compare_and_swap Instruction

Definition:

```
int compare _and_swap(int *value, int expected, int new_value) {
    int temp = *value;

    if (*value == expected)
       *value = new_value;
 return temp;
}
```

1. Executed atomically
2. Returns the original value of passed parameter "value"
3. Set the variable "value" the value of the passed parameter "new_value" but only if "value" =="expected". That is, the swap takes place only under this condition.

Solution using compare_and_swap

- Shared integer "lock" initialized to 0;
- Solution:

```
do {
     while (compare_and_swap(&lock, 0, 1) != 0)
      ; /* do nothing */
   /* critical section */
 lock = 0;
   /* remainder section */
} while (true);
```

Bounded-waiting Mutual Exclusion with test_and_set

```
// Need two Boolean variables lock and waiting[n]
// Initialized to false
do {
   waiting[i] = true;
   key = true;
   while (waiting[i] && key)
      key = test_and_set(&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; //Change waiting[j] from true to false
   /* remainder section */
} while (true);
```

Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest is mutex lock
- Protect a critical section by first **`acquire()`** a lock then **`release()`** the lock
  - Boolean variable indicating if lock is available or not
- Calls to **`acquire()`** and **`release()`** must be atomic
  - Usually implemented via hardware atomic instructions

acquire() and release()

-
  ```
  acquire() {
      while (!available)
          ; /* busy wait */
      available = false;;
  }
  ```

-
  ```
  release() {
      available = true;
  }
  ```

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

Mutex Locks

- Solution in previous slide requires **busy waiting**
  - This lock therefore called a **spinlock**
  - Takes CPU time (process is in running or ready state) but avoids context switching
  - Could be more efficient if locks are expected to be held for a short time on a multiprocessor system
  - One process is in the critical section on one processor, while another process is spinning on another processor

Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for processes to synchronize their activities.
- Semaphore ***S*** – integer variable
- Can only be accessed via two indivisible (atomic) operations
  - **`wait()`** and **`signal()`**
    - Originally called **`P()`** and **`V()`**
    - Signal is called **`post in Pthreads`**
- Definition of the **`wait() operation`**

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

- Definition of the **`signal() operation`**

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

Semaphore Usage

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
  - Same as a **mutex lock**
- Can solve various synchronization problems
- Consider ***P**$_1$* and ***P**$_2$* that require ***S**$_1$* to happen before ***S**$_2$*

  Create a semaphore "`synch`" initialized to 0

```
P1:
    S1;
    signal(synch);
P2:
    wait(synch);
    S2;
```

Semaphore Implementation

- Must guarantee that no two processes can execute the **`wait()`** and **`signal()`** on the same semaphore at the same time
- Could have **busy waiting** in critical section implementation
  - Implementation code is short
  - Little busy waiting if critical section rarely occupied
- If application spends lots of time in critical sections, this is not a good solution

Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated wait list (could be implemented as a FIFO 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 in the appropriate waiting queue
  - **wakeup** – remove one of processes from the waiting queue and place it in the ready queue

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

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 is not guaranteed to be removed from the semaphore wait list in which it is suspended
- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
  - Solved via **priority-inheritance protocol**

Classical Problems of Synchronization

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

Bounded-Buffer Problem

- ***N-item*** buffer
- Semaphore **`mutex`** initialized to the value 1
  - To protect access to the in and out variables when there are multiple producers and consumers
  - Not needed if there is only one producer and one consumer
- Semaphore **`fullCnt`** initialized to the value 0
  - The number of produced but unconsumed items
- Semaphore **`emptyCnt`** initialized to the value n
  - The number of empty locations in the buffer
  - Empty could mean never used or used but consumed (can be reused)

Bounded Buffer Problem (Cont.)

- The structure of the producer process

```
do {
        ...
         /* produce an item in next_produced */
        ...
     wait(emptyCnt);
     wait(mutex);
         ...
         /* add next produced to the buffer */
         buffer[in] = next_produced;
         in = (in+1)% BUFFER_SIZE
         ...
     signal(mutex);
     signal(fullCnt);
 } while (true);
```

Bounded Buffer Problem (Cont.)

- The structure of the consumer process

```
do {
    wait(fullCnt);
    wait(mutex);
       ...
       /* remove an item from buffer to next_consumed */
    next_consumed = Buf[out];
    out = (out+1)% BUFFER_SIZE
       ...
    signal(mutex);
    signal(emptyCnt);
       ...
       /* consume the item in next consumed */
       ...
    } while (true);
```

How about this Solution?

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

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

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 writer can access the shared data at the same time
- Several variations of how readers and writers are considered – all involve some form of priorities.
- Shared Data
  - The buffer itself
  - Integer **`read_count`** initialized to 0, tracks the number of readers currently reading from the buffer
  - Semaphore **`rw_mutex,`** initialized to 1, ensures exclusive access to buffer by either one writer or multiple readers
  - Semaphore r_**`mutex,`** initialized to 1, guards the read_count variable shared by readers

Readers-Writers Problem (Cont.)

- The structure of a writer process

```
    do {
           wait(rw_mutex);
              ...
           /* writing is performed */
              ...
           signal(rw_mutex);
  } while (true);
```

Readers-Writers Problem (Cont.)

- The structure of a reader process

```
do {
        wait(r_mutex);
        read_count++;
        if (read_count == 1)
            wait(rw_mutex);
        signal(r_mutex);
         ...
        /* reading is performed */
         ...
        wait(r_mutex);
        read count--;
        if (read_count == 0)
            signal(rw_mutex);
        signal(r_mutex);
} while (true);
```

Readers-Writers Problem (Cont.)

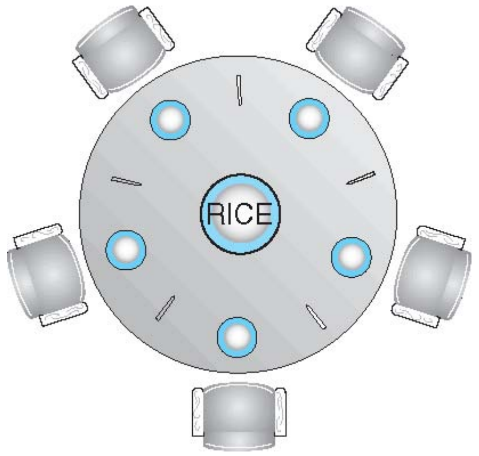
- The structure of a reader process

```
do {
       wait(r_mutex);
       read_count++;
       if (read_count == 1)//first reader

          wait(rw_mutex);

       signal(r_mutex);

        ...
       /* reading is performed */

        ...
       wait(r_mutex);
       read count--;
       if (read_count == 0) //last reader

          signal(rw_mutex);

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

Readers-Writers Problem Variations

- This is the ***First*** variation – no reader kept waiting unless writer has permission to use shared object
- In this variation, writers may starve but readers will never starve (assuming semaphores have FIFO queues)
- There are other variations
- Problem is solved on some systems by kernel providing **reader-writer locks**

Dining-Philosophers Problem



- Philosophers spend their lives alternating 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
    - Bowl of rice (data set)
    - 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?

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?
  - Deadlock if all five philosophers pick their left chopsticks at the same time

Dining-Philosophers Problem Algorithm (Cont.)

- Deadlock handling
  - Allow at most 4 philosophers to be sitting simultaneously at the table.
  - Allow a philosopher to pick up the chopsticks only if both are available (picking must be done in a critical section).
  - Use an asymmetric solution -- an odd-numbered philosopher picks up first the left chopstick and then the right chopstick. Even-numbered philosopher picks up first the right chopstick and then the left chopstick.

Problems with Semaphores

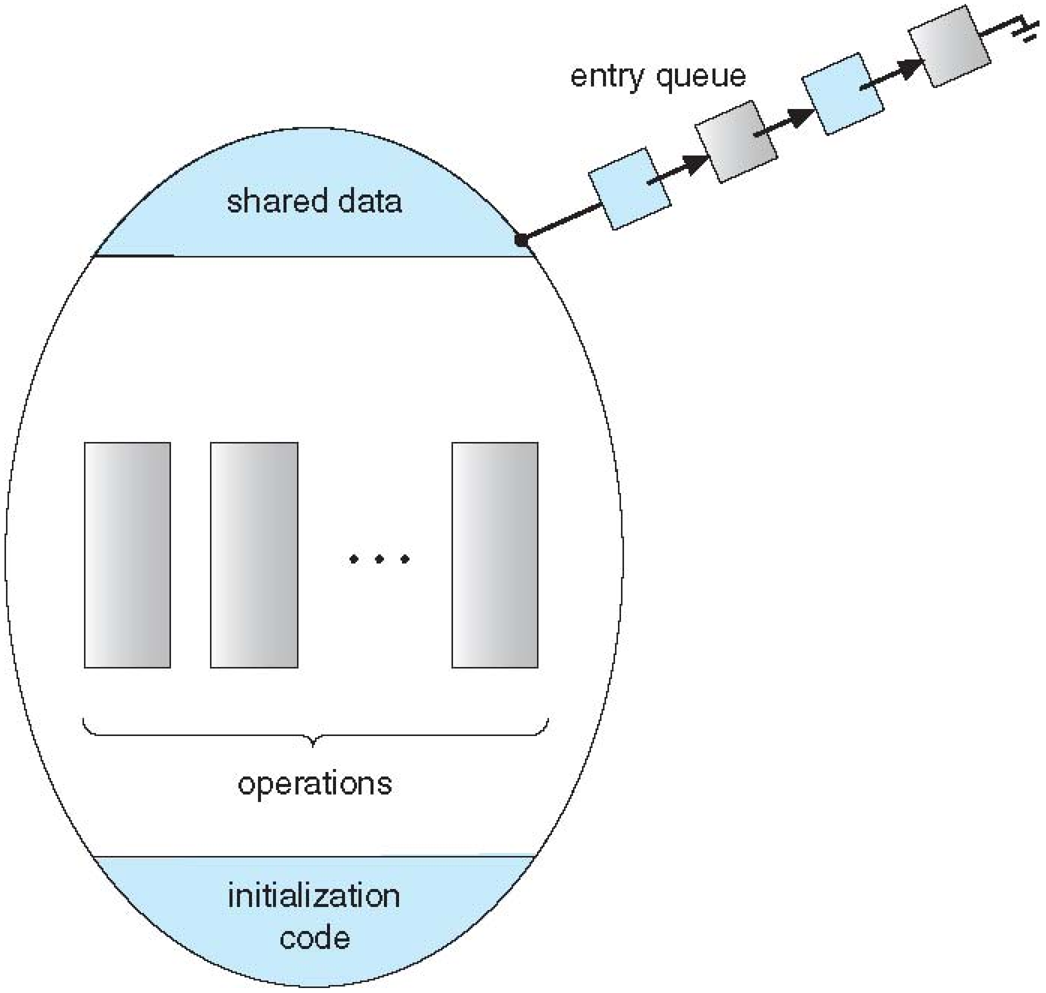
- Incorrect use of semaphore operations:
  - signal (mutex) …. wait (mutex)
    - Multiple processes may be in the CS at the same time
  - wait (mutex) … wait (mutex)
    - Deadlock
  - Omitting of wait (mutex) or signal (mutex) (or both)
    - mutual exclusion violation or deadlock

Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- *Abstract data type (ADT)*, internal variables only accessible by code within procedures
- Implemented in some languages including C# and Java
- Only one process may be active within the monitor at a time

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

  procedure Pn (…) {……}

  Initialization code (…) { … }
}
```

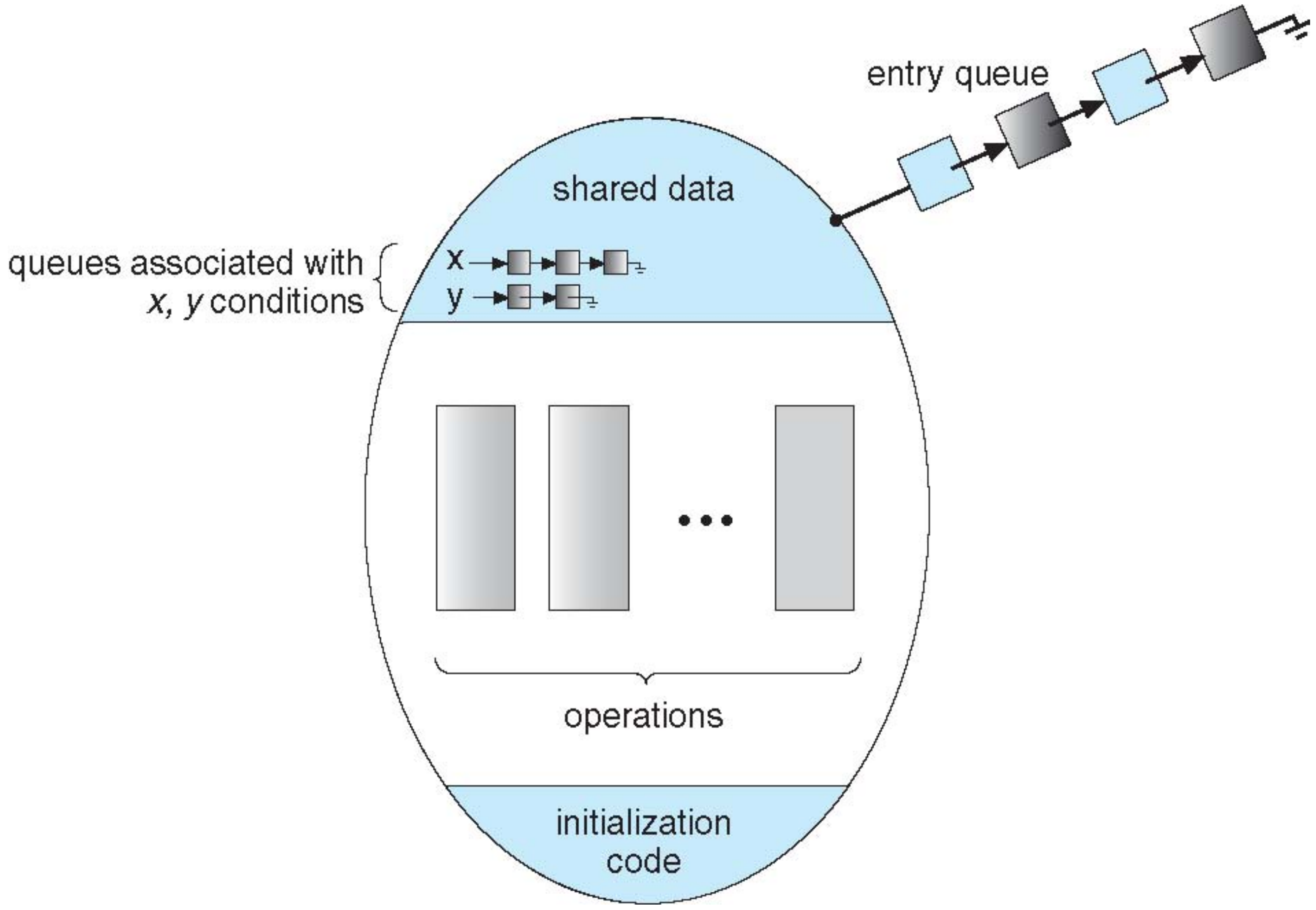
Schematic view of a Monitor



Condition Variables

- Monitors by themselves are not powerful enough to solve synchronization problems
  - Need additional synchronization mechanisms, such as *condition* variables
- **`condition x, y;`**
- Two operations are allowed on a condition variable **`x`**:
  - **`x.wait()`** – a process that invokes the operation is suspended until another process does **`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

Monitor with Condition Variables



Condition Variables Choices

- If process P invokes **`x.signal()`** and process Q is suspended in **`x.wait()`**, what should happen next?
  - Both Q and P cannot execute in the monitor at the same time. If Q is resumed, then P must wait
- Options include
  - **Signal and wait** – P waits until Q leaves the monitor
  - **Signal and continue** – Q waits until P leaves the monitor
  - Both have pros and cons – language implementer can decide

Monitor 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);
}
```

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.)

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

```
DiningPhilosophers.pickup(i);

     EAT

DiningPhilosophers.putdown(i);
```

- Is a deadlock possible? Is starvation possible?

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);
```

- Is a deadlock possible? Is starvation possible?
  - No deadlock, but starvation is possible

Synchronization Examples

- Windows
- Linux
- Pthreads

Windows Synchronization

- Inside kernel:
  - Uses interrupt masks to protect access to global resources on uniprocessor systems
  - Uses **spinlocks** on multiprocessor systems
    - Spinlocking-thread will never be preempted
- Outside kernel: provides **dispatcher objects,** including mutex locks, semaphores, events, and timers
  - **Event** acts much like a condition variable
  - **Timer** notifies one or more threads when time expires
  - Dispatcher objects either **signaled-state** (object available) or **non-signaled state** (thread will block)
    - Number of threads kernel notifies depends on object type (1 if mutex, all if event).

Linux Synchronization

- Linux:
  - Prior to kernel Version 2.6, disables interrupts to implement short critical sections
  - Version 2.6 and later, fully preemptive
- Linux provides:
  - Atomic integers
    - atomic_t with set, add, sub, inc operations
  - Spinlocks for short duration
  - Mutex locks, semaphores, and reader-writer locks for longer durations

Pthreads Synchronization

- Pthreads API is OS-independent
- It provides:
  - mutex locks
  - condition variables
  - Semaphores (POSIX SEM extension)
- Non-portable extensions include:
  - read-write locks
  - spinlocks

Pthreads Synchronization Examples

- Mutex Locks

```
#include <pthread.h>
pthread_mutex_t mutex;
pthread_mutex_init (&mutex, NULL); // create mutex lock
pthread_mutex_lock (&mutex); // acquire mutex lock
/* critical section */
pthread_mutex_unlock (&mutex); // release mutex lock
```

- Semaphores

```
#include <semaphore.h>
sem_t sem;
sem_init (&sem, 0, 1); // create semaphore and initialize it to 1
sem_wait (&sem); // acquire semaphore
/* critical section */
sem_post (&sem); // release semaphore
```

Alternative Approaches

- Transactional Memory
- OpenMP

Transactional Memory

- A **memory transaction** is a sequence of read-write operations to memory that are performed atomically.

```
 Replace
                  void update()
 {
acquire ();
/* read/write memory */
                            release ();
   }
 With
                  void update()
 {
atomic {
/* read/write memory */
                            }
   }
```

OpenMP

- OpenMP is a set of compiler directives and API that support parallel progamming.

```
void update(int value)
 {
      #pragma omp critical
      {
            count += value
      }
  }
```

The code contained within the **`#pragma omp critical`** directive is treated as a critical section and performed atomically.

End of Chapter 5