# Shared Memory Model

Alessio Vecchio
alessio.vecchio@unipi.it
Dip. di Ingegneria dell'Informazione
Università di Pisa

Based  on original slides by Silberschatz, Galvin, and Gagne
Operating System Concepts

# Overview

- The Critical-Section Problem
- Software Solutions
- Synchronization Hardware
- Semaphores
- Monitors
- Synchronization Examples

# Overview

- The Critical-Section Problem
- Software Solutions
- Synchronization Hardware
- Semaphores
- Monitors
- Synchronization Examples

# Producer-Consumer Problem

- The *Producer* process produces data that must processed by the *Consumer* process
- The inter-process communication occurs through a shared buffer (shared memory)
- Bounded Buffer Size
  - The Producer process cannot insert a new item if the buffer is full
  - The Consumer process cannot extract an item if the buffer is empty

# Producer-Consumer Problem

- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
        . . .
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
int counter = 0;
```

# Producer-Consumer Problem

- Producer process

```
item nextProduced;

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

# Producer-Consumer Problem

- Consumer process

```
item nextConsumed;

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

# Producer-Consumer Problem

- The statements

  **counter++;**
  **counter--;**

  must be performed atomically.

- Atomic operation means an operation that completes in its entirety without interruption.

# Producer-Consumer Problem

- The statement “counter++” may be implemented in machine language as:

  **register1 = counter**
  **register1 = register1 + 1**
  **counter = register1**

- The statement “counter- -” may be implemented as:

  **register2 = counter**
  **register2 = register2 – 1**
  **counter = register2**

# Producer-Consumer Problem

- If both the producer and consumer attempt to update the buffer concurrently, the assembly language statements may get interleaved.

- Interleaving depends upon how the producer and consumer processes are scheduled.

# Race Condition

- Assume **counter** is initially 5. One interleaving of statements is:

  producer: **register1 = counter** (*register1 = 5*)
  producer: **register1 = register1 + 1** (*register1 = 6*)

  consumer: **register2 = counter** (*register2 = 5*)
  consumer: **register2 = register2 – 1** (*register2 = 4*)

  producer: **counter = register1** (*counter = 6*)
  consumer: **counter = register2** (*counter = 4*)

- The value of **count** may be either 4 or 6, where the correct result should be 5.

# Race Condition

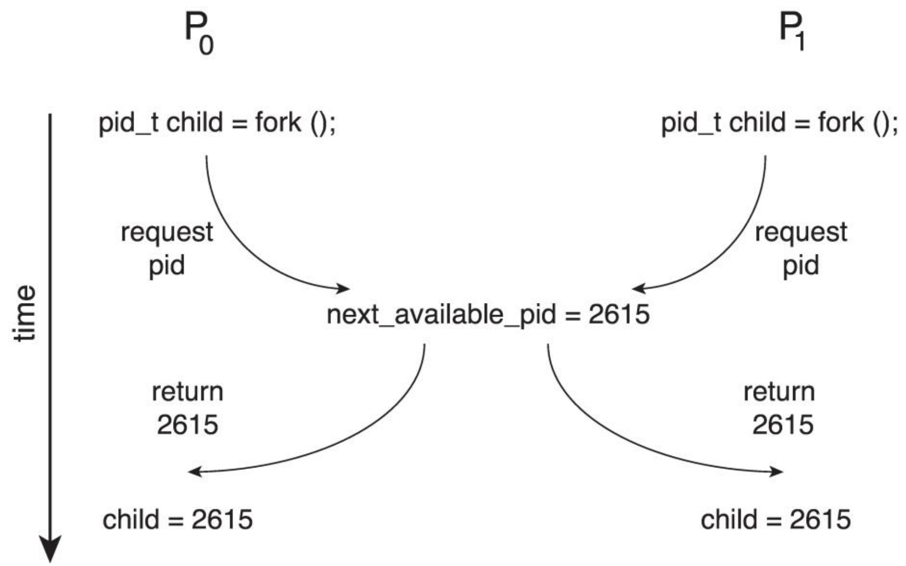
- Race condition
  - The situation where several processes access and manipulate shared data concurrently.
  - The final value of the shared data depends upon how instructions are interleaved.

- Show example about balance and num.Ops.

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

# Race Condition

- Processes $P_0$ and $P_1$ are creating child processes using the `fork()` system call
- Race condition on kernel variable `next_available_pid` which represents the next available process identifier (pid)




- Unless there is a mechanism to prevent $P_0$ and $P_1$ from accessing the variable `next_available_pid` the same pid could be assigned to two different processes!

# Critical Section Problem

- Consider system of ***n*** processes {$p_0, p_1, \ldots p_{n-1}$}
- 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 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**

# General Process Structure

- General structure of process $P_i$

**do** {

    entry section

    critical section

    exit section

    reminder section

} **while (true)**

# Critical-Section Problem (Cont.)

Requirements for 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 process 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

# Possible Solutions

- Software approaches
- Hardware solutions
  - Interrupt disabling
  - Special machine instructions
- Operating System Support
  - Semaphores
- Programming language Support
  - Monitor

# A Software Solution

```
boolean lock=false;
Process Pi {
        do {
                while (lock);          // do nothing
                lock=true;
                critical section
                lock=false;
                remainder section
        } while (true);
}
```

Does it work?

# Software Solution 1

- Two process solution
- Assume that the **`load`** and **`store`** machine-language instructions are atomic; that is, cannot be interrupted
- The two processes share one variable:
  - **`int turn`**;
- The variable **`turn`** indicates whose turn it is to enter the critical section

# Algorithm for Process $P_i$

```
do {

        turn = i;
        while (turn == j);

        /* critical section */

        turn = j;

        /* remainder section */

} while (true);
```

# Correctness of the Software Solution 1

- Mutual exclusion is preserved

  **$P_i$** enters critical section only if:

  **`turn = i`**

  and **`turn`** cannot be both 0 and 1 at the same time
- What about the Progress requirement?
- What about the Bounded-waiting requirement?

# Peterson's Solution

- Two process solution
- Assume that the **`load`** and **`store`** machine-language 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);
```

# Correctness of Peterson's Solution

- Provable that the three  CS requirement are met:
  1. Mutual exclusion is preserved

     $P_i$ enters CS only if:

     either **flag[j] = false** or **turn = i**
  2. Progress requirement is satisfied
  3. Bounded-waiting requirement is met

# Peterson's Solution and Modern Architecture

- Although useful for demonstrating an algorithm, Peterson's Solution is not guaranteed to work on modern architectures.
  - To improve performance, processors and/or compilers may reorder operations that have no dependencies
- Understanding why it will not work is useful for better understanding race conditions.
- For single-threaded this is ok as the result will always be the same.
- For multithreaded the reordering may produce inconsistent or unexpected results!

# Modern Architecture Example

- Two threads share the data:
```
boolean flag = false;
int x = 0;
```
- Thread 1 performs
```
while (!flag)
  ;
print x
```
- Thread 2 performs
```
x = 100;
flag = true
```
- What is the expected output?

  100

# Modern Architecture Example (Cont.)

- However, since the variables `flag` and `x` are independent of each other, the instructions:

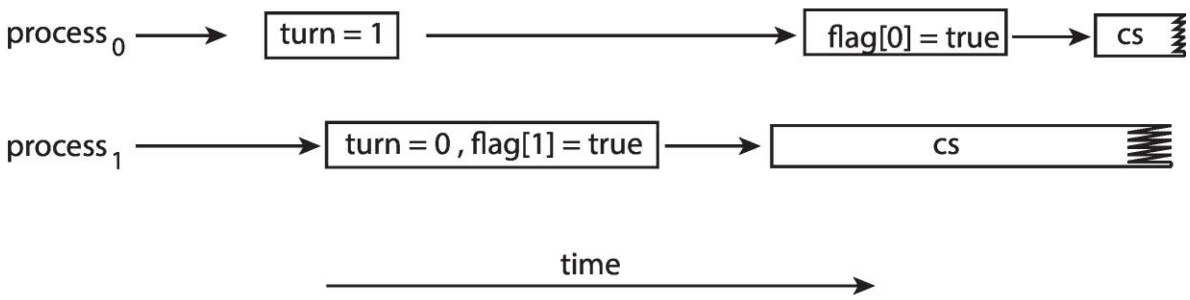
```
flag = true;
 x = 100;
```

for Thread 2 may be reordered

- If this occurs, the output may be 0!

# Peterson's Solution Revisited

- The effects of instruction reordering in Peterson's Solution




- This allows both processes to be in their critical section at the same time!
- To ensure that Peterson's solution will work correctly on modern computer architecture we must use **Memory Barrier**.

# Memory Barrier

- **Memory model** are the memory guarantees a computer architecture makes to application programs.
- Memory models may be either:
  - **Strongly ordered** – where a memory modification of one processor is immediately visible to all other processors.
  - **Weakly ordered** – where a memory modification of one processor may not be immediately visible to all other processors.
- A **memory barrier** is an instruction that forces any change in memory to be propagated (made visible) to all other processors.

# Memory Barrier Instructions

- When a memory barrier instruction is performed, the system ensures that all loads and stores are completed before any subsequent load or store operations are performed.
- Therefore, even if instructions were reordered, the memory barrier ensures that the store operations are completed in memory and visible to other processors before future load or store operations are performed.

# Memory Barrier Example

- Returning to previous example
- We could add a memory barrier to the following instructions to ensure Thread 1 outputs 100:
- Thread 1 now performs

```
while (!flag)
  memory_barrier();
print x
```

- Thread 2 now performs

```
x = 100;
memory_barrier();
flag = true
```

- For Thread 1 we are guaranteed that that the value of `flag` is loaded before the value of `x`.
- For Thread 2 we ensure that the assignment to `x` occurs before the assignment `flag`.

# Solution to Critical-section Problem Using Locks

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

# Overview

- The Critical-Section Problem
- Software Solutions
- **Synchronization Hardware**
- Semaphores
- Monitors
- Synchronization Examples

# Interrupt Disabling

```
do {
        disable interrupt;
        critical section
        enable interrupt;
        remainder section
  } while (true);
```

# Previous Solution

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

This solution does not guarantee the mutual exclusion because the test and set on lock are not atomic

# Synchronization Hardware

- Many systems provide hardware support for implementing the 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
- We will look at three forms of hardware support:

1. Hardware instructions
2. Atomic variables

# Hardware Instructions

- Special hardware instructions that allow us to either *test-and-modify* the content of a word, or two *swap* the contents of two words atomically (uninterruptedly.)
  - **Test-and-Set** instruction
  - **Compare-and-Swap** instruction

# The test_and_set Instruction

- Definition

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

- Properties
  - Executed atomically
  - Returns the original value of passed parameter
  - 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);
```

- Does it solve the critical-section problem?

# The 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;
  }
```

- Properties
  - Executed atomically
  - Returns the original value of passed parameter **`value`**
  - Set the variable **`value`** the value of the passed parameter **`new_value`** but only if **`*value == expected`** is true. That is, the swap takes place only under this condition.

# Solution using compare_and_swap

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

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

    /* critical section */

    lock = 0;

    /* remainder section */
}
```

- Does it solve the critical-section problem?

# Bounded-waiting with compare-and-swap

```
while (true) {
   waiting[i] = true;
   key = 1;
   while (waiting[i] && key == 1)
      key = compare_and_swap(&lock,0,1);
   waiting[i] = false;
   /* critical section */
   j = (i + 1) % n;
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;
   if (j == i)
      lock = 0;
   else
      waiting[j] = false;
   /* remainder section */
}
```

# Atomic Variables

- Typically, instructions such as compare-and-swap are used as building blocks for other synchronization tools.
- One tool is an **atomic variable** that provides *atomic* (uninterruptible) updates on basic data types such as integers and booleans.
- For example:
  - Let **`sequence`** be an atomic variable
  - Let **`increment()`** be operation on the atomic variable **`sequence`**
  - The Command:

```
increment(&sequence);
```

ensures **`sequence`** is incremented without interruption:

# Atomic Variables

- The **`increment()`** function can be implemented as follows:

```
void increment(atomic_int *v)
{
    int temp;
    do {
        temp = *v;
    }
    while (temp != (compare_and_swap(v,temp,temp+1));
}
```

# 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
  - Boolean variable indicating if lock is available or not
- Protect a critical section by
  - First **`acquire()`** a lock
  - Then **`release()`** the lock
- Calls to **`acquire()`** and **`release()`** must be **atomic**
  - Usually implemented via hardware atomic instructions such as compare-and-swap.
- But this solution requires **busy waiting**
  - This lock therefore called a **spinlock**

# Solution to CS Problem Using Mutex Locks

```
while (true) {
        acquire lock

        critical section

        release lock

        remainder section
}
```

# Semaphore

- Synchronization tool that does not require busy waiting
- Semaphore S – integer variable
- Can only be accessed via two indivisible (atomic) operations
  - wait() and signal()
  - Originally called P() and V()

# Semaphore

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

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

wait() and signal() must be atomic

# Semaphore as 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

# Semaphore as Mutex Tool

- Solution to the critical section problem
- Shared data:

```
semaphore mutex=1;
```

- Process *Pi:*

```
do {
   wait (mutex);
    /* critical section */
    signal (mutex);
   /* remainder section */
 } while (true);
```

# Semaphore Implementation

- Must guarantee that no two processes can execute the **`wait()`** and **`signal()`** on the same semaphore at the same time
- Thus, the 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

# 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

# Implementation with no Busy waiting (Cont.)

- Waiting queue

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

# Implementation with no Busy waiting (Cont.)

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

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

# Semaphore as a Synchronization Tool

- Execute *B* in $P_j$ only after *A* executed in $P_i$
- Use semaphore `flag` initialized to 0
- Code:

| | |
|---|---|
| $P_i$ | $P_j$ |
| … | … |
| ***A*** | ***wait(flag)*** |
| ***signal(flag)*** | ***B*** |

# 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

Producer Process

```
do {
     …
     <produce an item in nextp>
     …
     wait(empty);
     wait(mutex);
     …
     <add nextp to buffer>
     …
     signal(mutex);
     signal(full);

} while (true);
```

Consumer Process

```
do {
     wait(full)
     wait(mutex);
     …
     <remove item from buffer to nextc>
     …
     signal(mutex);
     signal(empty);
     …
     <consume item in nextc>
     …
} 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
- Variants
  - No new reader must wait when a writer is waiting for data access
  - No new reader can start reading when a writer is waiting for data access

# Readers-Writers Problem

- Shared Data
  - Data set
  - Integer readcount initialized to 0
  - Semaphore mutex initialized to 1
    - Mutual exclusion on readcount
  - Semaphore wrt initialized to 1
    - Mutual exclusion on the data set by writers

# Readers-Writers Problem

- The structure of a writer process

```
do {
    wait (wrt);
    // writing is performed
    signal (wrt);
} while (true);
```

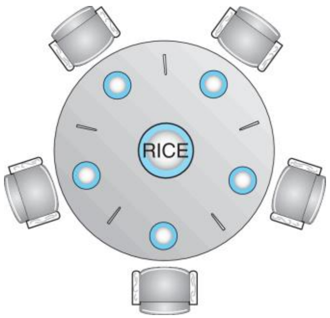
# Readers-Writers Problem

- The structure of a reader process

```
do {
     wait (mutex);
     readcount ++;
     if (readcount == 1)
                  wait (wrt);
     signal (mutex) ;
     // reading is performed
     wait (mutex) ;
     readcount --;
     if (readcount == 0)
                  signal (wrt);
     signal (mutex);
} while (true);
```

# Dining-Philosophers Problem

- N philosophers' sit at a round table with a bowel of rice in the middle.




- They spend their lives alternating thinking and eating.
- They do not  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, the shared data
    - Bowl of rice (data set)
    - Semaphore chopstick [5] initialized to 1

# Dining-Philosophers Problem Algorithm

- Semaphore Solution
- The structure of Philosopher i :

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

     /* eat for awhile */

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

     /* think for awhile */

}
```

- What is the problem with this algorithm?

# Dining-Philosophers Problem

- Deadlock
  - A deadlock occurs if all philosophers start eating simultaneously
- Possible solutions to avoid deadlocks
  - Only 4 philosophers can sit around the table
  - A philosopher can take his/her chopsticks only if they both are free
  - An odd philosopher takes the chopstick on its left first, and then the one on its right; an even philosopher takes the opposite approach.
- Starvation
  - Any solution must avoid that a philosopher may starve

# Problems with Semaphores

- Incorrect use of semaphore operations:
  - **`signal (mutex)`** ... **`wait (mutex)`**
  - **`wait (mutex)`** ... **`wait (mutex)`**
  - Omitting of **`wait (mutex)`** and/or **`signal (mutex)`**

# Overview

- The Critical-Section Problem
- Software Solutions
- Synchronization Hardware
- Semaphores
- **Monitors**
- Synchronization Examples

# Monitors

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

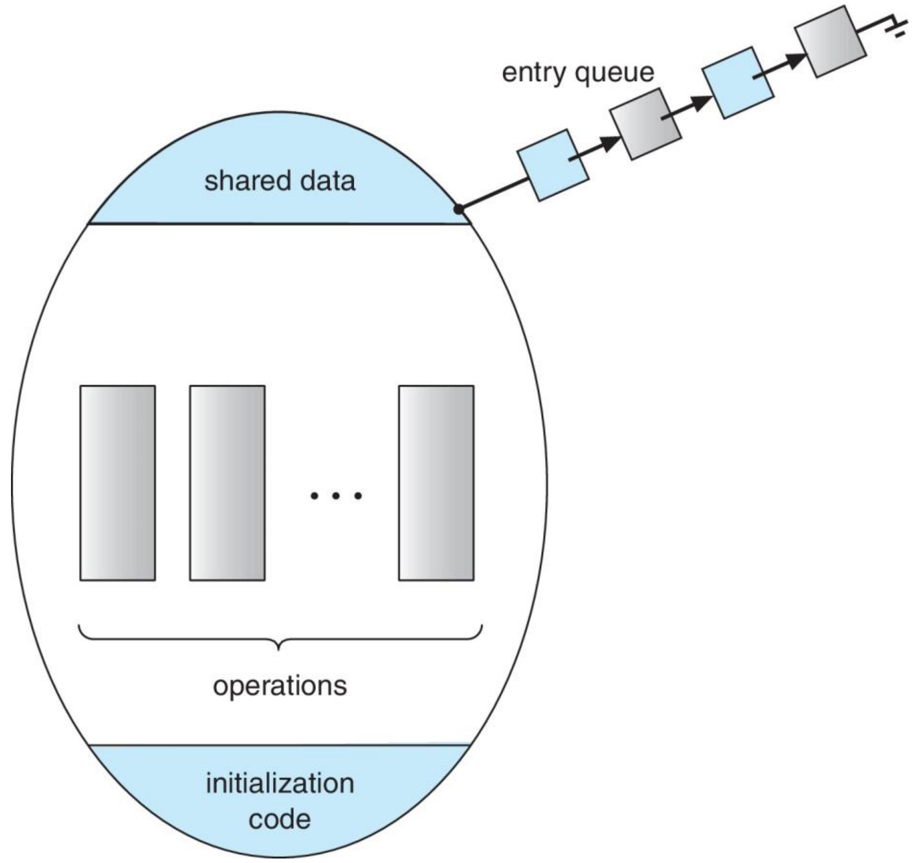
```
monitor monitor-name
{
  // shared variable declarations
  procedure P1 (…) { …. }

  procedure P2 (…) { …. }

  procedure Pn (…) {……}

  initialization code (…) { … }
}
```

# Schematic view of a Monitor



# Monitor Implementation Using Semaphores

- Variables

```
semaphore mutex
mutex = 1
```

- Each procedure ***P*** is replaced by

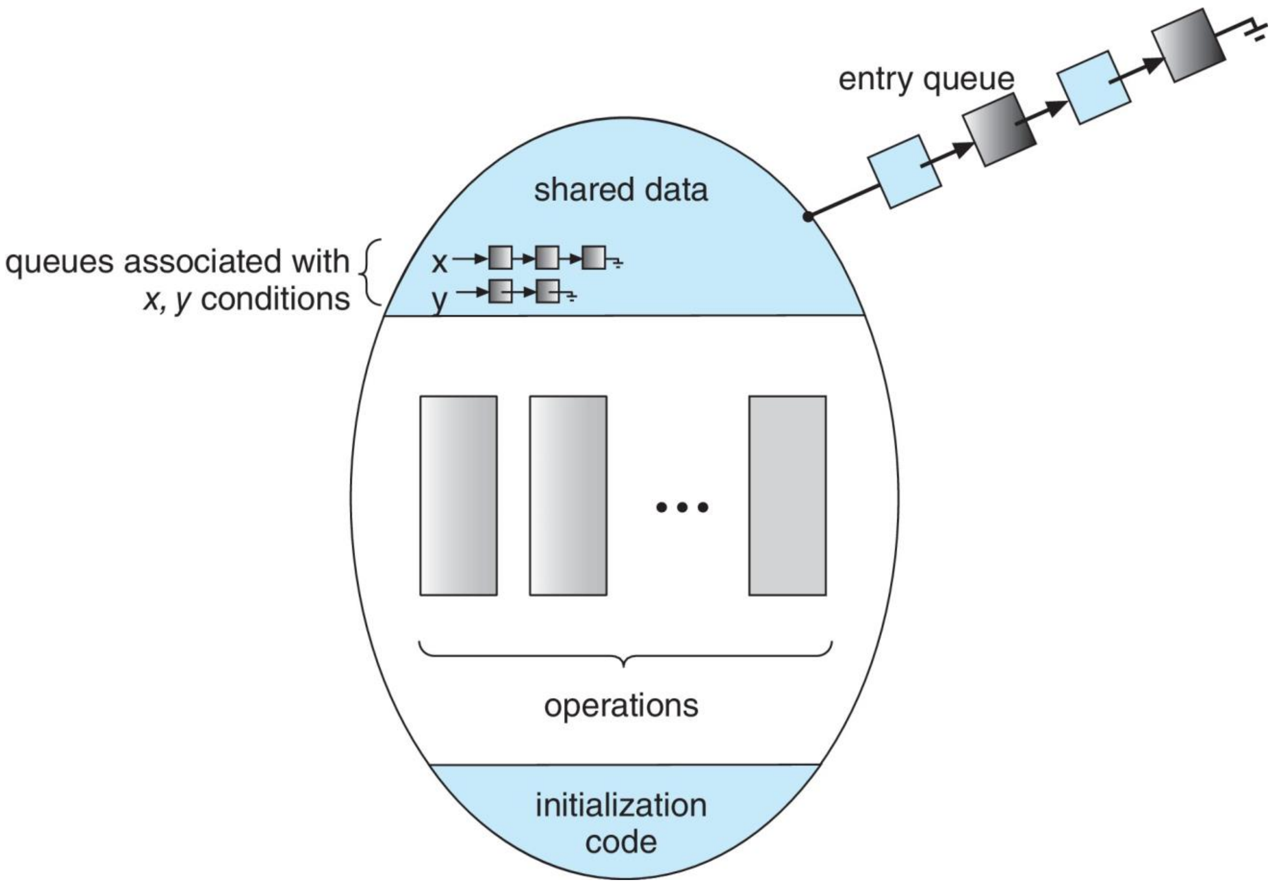
```
wait(mutex);
    ...
body of P;
    ...
signal(mutex);
```

- Mutual exclusion within a monitor is ensured

# Condition Variables

- **`condition x, y;`**
- Two operations are allowed on a condition variable:
  - **`x.wait()`** – a process that invokes the operation is always suspended until **`x.signal()`**
  - **`x.signal()`** – resumes one of processes (if any) that invoked **`x.wait()`**
    - If no **`x.wait()`** on the variable, then it has no effect on the variable

# Monitor with Condition Variables



# Usage of Condition Variable Example

- Consider $P_1$ and $P_2$ that that need to execute two statements $S_1$ and $S_2$ and the requirement that $S_1$ to happen before $S_2$
  - Create a monitor with two procedures $F_1$ and $F_2$ that are invoked by $P_1$ and $P_2$ respectively
  - One condition variable “x” initialized to 0
  - One Boolean variable “done”
  - **F1:**

```
S1;
done = true;
x.signal();
```

  - **F2:**

```
if done = false
   x.wait()
S2;
```

# Monitor Implementation Using Semaphores

- Variables

```
semaphore mutex;  // (initially  = 1)
semaphore next;   // (initially  = 0)
int next_count = 0; // number of processes waiting
                       inside the monitor
```

- Each function ***P*** will be replaced by

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

- Mutual exclusion within a monitor is ensured

# Implementation – Condition Variables

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

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

- The operation **`x.wait()`** can be implemented as:

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

# Implementation (Cont.)

- The operation **`x.signal()`** can be implemented as:

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

# 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);
```

- No deadlock, but starvation is possible

# Overview

- The Critical-Section Problem
- Software Solutions
- Synchronization Hardware
- Semaphores
- Monitors
- **Synchronization Examples**

# 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:
  - Semaphores
  - Atomic integers
  - Spinlocks
  - Reader-writer versions of both
- On single-CPU system, spinlocks replaced by enabling and disabling kernel preemption

# Linux Synchronization

- Atomic variables

  **`atomic_t`** is the type for atomic integer

- Consider the variables

```
atomic_t counter;
int value;
```

| *Atomic Operation* | *Effect* |
|---|---|
| atomic_set(&counter,5); | counter = 5 |
| atomic_add(10,&counter); | counter = counter + 10 |
| atomic_sub(4,&counter); | counter = counter - 4 |
| atomic_inc(&counter); | counter = counter + 1 |
| value = atomic_read(&counter); | value = 12 |

# Pthreads Synchronization

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

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