Process Synchronization

Operating Systems (ECEg-4181)

Mequanent Argaw Muluneh

Wednesday, April 29, 2020

Outline

- ❖ The Critical Section Problem
- ❖ Peterson’s Solution
- ❖ Synchronization Hardware
- ❖ Mutex Locks
- ❖ Semaphores
- ❖ Classic Problems of Synchronization
- ❖ Monitors

Objectives

- 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 or in parallel.
  - The scheduler switches rapidly to provide concurrent execution.
  - Thus, a process may only partially complete its execution before another process is scheduled.
- 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 fills **all** the buffers.

Background ...

Producer-Consumer: Producer

❖ We can do so by having an integer **`counter`** that keeps track of the number of full buffers. Initially, **`counter`** 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.

This was **[** `(in + 1) % BUFFER_SIZE == out` **]** previously.

```
while (true)
 {
   /* produce an item in next produced */
   while (counter == BUFFER_SIZE)
       ;    /* do nothing */
   buffer[in] = next_produced;
   in = (in + 1) % BUFFER_SIZE;
   counter++;
}
```

Background ...

Producer-Consumer: Consumer

This was [`in == out`] previously.

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

Background ...

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 "counter = 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}

Background ...

Race Condition ...

- A situation like above, where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a **race condition**.
- To guard against the race condition above, we need to ensure that only one process at a time can be manipulating the variable `counter`.
- To make such a guarantee, we require that the processes be *synchronized* in some way.

The Critical Section Problem

- Consider system of ***n*** processes {p_0, p_1, … p_{n-1}} each process having **critical section (CS)** segment of code
  - Process may be changing common variables, updating a table, writing a file, and so on.
  - When one process executes its critical section, no other process is allowed to execute its critical section.
- The critical-section problem is to design a protocol that the processes can use to cooperate.

The Critical Section …

- Each process must request permission by its ***entry section*** to enter its critical section.
- The critical section may be followed by an ***exit section***. The remaining code is the ***remainder section***.
- General structure of a typical process P_i

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

The Critical Section ...

Solution to Critical Section Problem

- A solution to the critical-section problem *must* satisfy the following *three* requirements:

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.

- **Assumption**: each process is executing at a nonzero speed. However, no assumption concerning the relative speed of the *n* processes.

The Critical Section …

Critical Section Handling in OS

- ***Two*** general approaches are used to handle critical sections in operating systems:
  - **Preemptive kernels** – allow a process to be preempted while it is running in kernel mode.
  - **Non-preemptive kernels**– do not allow a process running in kernel mode to be preempted. It runs until it exits kernel mode, blocks, or voluntarily yields control of the CPU.
    - Hence, it is free from race conditions on kernel data structures, as only one process is active in the kernel at a time.

Peterson's Solution

- Petrson's Solution is a good algorithmic description of solving the critical section (CS) problem for two processes.
- Assume that the **`load`** and **`store`** machine-language instructions are atomic; i.e. 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!

Peterson’s Solution …

The structure of process P_i in Peterson’s solution.

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

Peterson's Solution ...

- Provable that the three critical section requirements are met in Peterson's solution:
  - Mutual exclusion is preserved since P_i enters CS only if:
    - either **`flag[j] = false`** or **`turn = i`**
  - If P_j resets **`flag[j]`** to **`true`**, it must also set **`turn`** to **`i`**. Thus, since P_i does not change the value of the variable **`turn`** while executing the while statement, P_i will enter the critical section (progress) after at most one entry by P_j (bounded waiting).

Synchronization Hardware

- Many systems provide hardware support for implementing the critical section code.
- All solutions to be discussed below ranging from hardware to software-APIs are based on idea of **locking.**
  - Protecting critical regions via locks.
- Single-processor environments could disable interrupts while modifying a shared variable to solve the CS problem.
  - Currently running code would execute without preemption.
  - Disabling interrupts is not feasible on multiprocessor systems.
- Modern machines provide special atomic (non-interruptible) hardware instructions. These instructions allow us to:
  - Either test memory word and set value
  - Or compare and swap contents of two memory words atomically.

Synchronization Hardware ...

Solution to Critical Section Problem Using Locks

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

Synchronization Hardware …

Test_and_Set Instruciton

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`".

Synchronization Hardware …

Solution Using Test_and_Set()

- The shared boolean variable **`lock`**, is initialized to **`false`**.
- Solution:

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

Do you think that all the above three requirements are satisfied?

Synchronization Hardware ...

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 value of the passed parameter "`new_value`" to the variable "`value`" but only if "`value`" =="`expected`". That is, the swap takes place only under this condition.

Synchronization Hardware …

Solution Using Compare_and_Swap

- The shared integer "**lock**" is initialized to 0;
- Solution:

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

Do you think that all the above three requirements are satisfied?

Synchronization Hardware …

Bounded_Waiting Mutual Exclusion with Test_and_Set

Both test_and_set() and compare_and_swap() algorithms satisfy mutual exclusion but not bounded-waiting. test_and_set() is improved as shown on the right to meet the bounded-waiting requirement. Both **`waiting[i]`** and **`key`** are 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;
    /* remainder section */
} while (true);
```

Mutex Locks

- The hardware-based solutions to the CS problem discussed above are ***complicated*** and generally inaccessible to application programmers.
- OS designers build software tools to solve critical section problem.
- The simplest of these tools is the **mutex lock**.
- Protect a CS since a process must first **`acquire()`** a lock before entering the CS and then **`release()`** the lock when it exiting the CS.
  - Mutex lock uses a boolean variable **`available`** whose value indicates if lock is available or not.
- Calls to **`acquire()`** and **`release()`** must be performed atomically.
  - Often implemented using one of the above hardware mechanisms.
- The main disadvantage of the implementation given here is that it requires **busy waiting.**
  - This lock is therefore called a **spinlock** since a process "spins" while waiting for the lock to become available.

Mutex Locks …

`acquire()` and `release()`

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

Definition of **`acquire()`**

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

Definition of **`release()`**

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

Solution to the critical-section problem using mutex locks.
  ```

Semaphores

- Semaphore is a robust tool that can behave similarly to a mutex lock but can also provide more sophisticated ways for processes to synchronize their activities.
- A **semaphore** S is an integer variable that, apart from initialization, is accessed only through two standard ***atomic*** operations: **`wait()`** and **`signal()`**.
- The definition of **`wait()`** operation

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

- The definition of **`signal()`** operation

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

Semaphores ...

Semaphore Usage

- Operating systems often distinguish between counting and binary semaphores.
- The value of a **counting semaphore** can range over an unrestricted domain.
- The value of a **binary semaphore** can range only between 0 and 1.
  - Thus, binary semaphores behave similarly to mutex locks.
- 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;
```

Semaphores ...

Semaphore Implementation

- 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.
  - We could now have **busy waiting** in critical section implementation
    - But implementation code is short.
    - Little busy waiting if critical section is rarely occupied.
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution.

Semaphores ...

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.

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

Semaphores ...

Implementation With No Busy Waiting ...

```
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);
    }
}
```

Semaphores ...

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 and accessed by processes P0 and P1.

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

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

Classical Problems of Synchronization …

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

Classical Problems of Sync …

Bounded-Buffer Problem …

❖ Structure of the producer process

```
do {
        ...
      /* produce an item in
         next_produced */
        ...
       wait(empty);
       wait(mutex);
         ...
       /* add next_produced
          to the buffer */
         ...
       signal(mutex);
       signal(full);
    } while (true);
```

❖ Structure of the consumer process

```
Do {
       wait(full);
       wait(mutex);
          ...
    /* remove an item from
       buffer to
       next_consumed */
          ...
       signal(mutex);
       signal(empty);
          ...
   /* consume the item
   in next consumed */
          ...
    } while (true);
```

Classical Problems of Sync ...

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 and only one single writer can access the shared data at a time.
- There are several variations of how readers and writers are considered all involving some form of priorities.
- Shared data set includes:
  - Semaphore **`rw_mutex`** initialized to 1
  - Semaphore **`mutex`** initialized to 1
  - Integer **`read_count`** initialized to 0

Classical Problems of Sync ...

Readers-Writers Problem ...

- The structure of a writer process

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

- The structure of a reader process

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

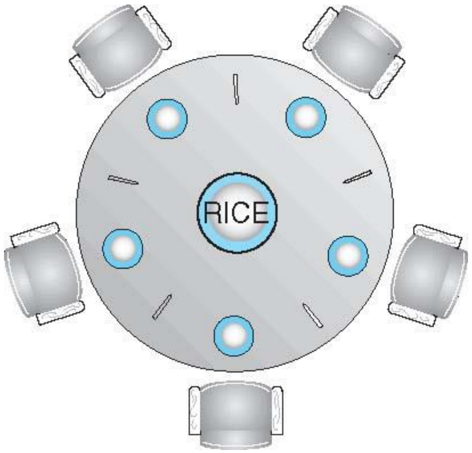
Classical Problems of Sync …

Readers-Writers Problem Variations

- ***First*** variation – no reader is kept waiting unless writer has already obtained permission to use shared object.
- ***Second*** variation – once writer is ready, it performs the write as soon as possible.
- Both may have starvation leading to even more variations.

Classical Problems of Sync …

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.

Classical Problems of Sync …

Dining-Philosophers Problem …

- Deadlock handling
  - Allow at most 4 philosophers to be sitting simultaneously at the table.
  - Allow a philosopher to pick up the forks 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.

Classical Problems of Sync …

Problems with Semaphores

- ❖ Using semaphores incorrectly can result in timing errors that are difficult to detect.
- ❖ These errors happen only if particular execution sequences take place and these sequences do not always occur.
- ❖ Some incorrect use of semaphore operations:
    - ❖ signal (mutex) …. wait (mutex)
    - ❖ wait (mutex) … wait (mutex)
    - ❖ Omitting of wait (mutex) or signal (mutex) (or both)
- ❖ **Deadlock and starvation are possible and mutual exclusion can be violated.**

Monitors

- Monitor is a high-level abstraction that provides a convenient and effective mechanism for process synchronization.
- *Abstract data type*, internal variables are only accessible by code within the procedure.
- Only one process is active within the monitor at a time.

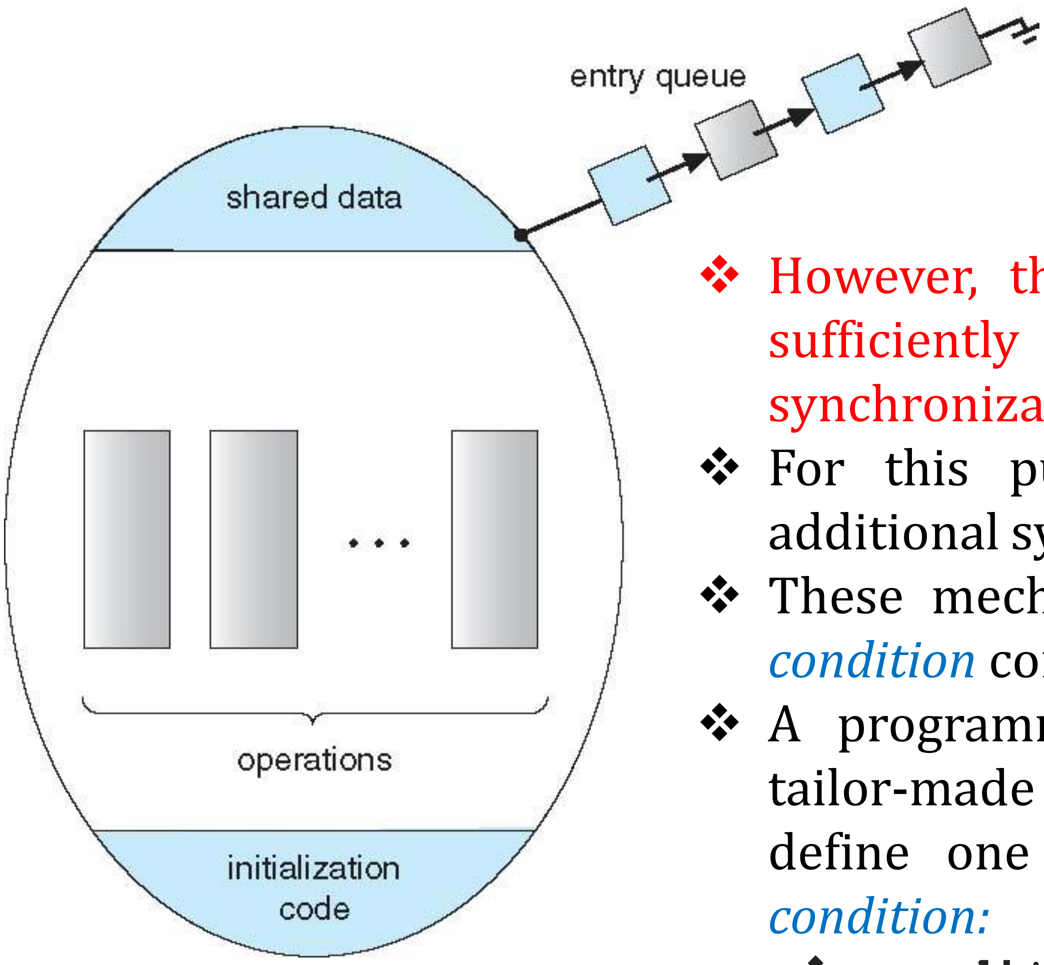
```
monitor monitor-name
{
  // shared variable declarations
  procedure P1 (…) { …. }

  procedure Pn (…) {……}

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

Monitors …

Schematic View of a Monitor



- However, this monitor construct is not sufficiently powerful to model some synchronization schemes.
- For this purpose, we need to define additional synchronization mechanisms.
- These mechanisms are provided by the *condition* construct.
- A programmer who needs to write a tailor-made synchronization scheme can define one or more variables of type *condition:*
  - **`condition x, y;`**

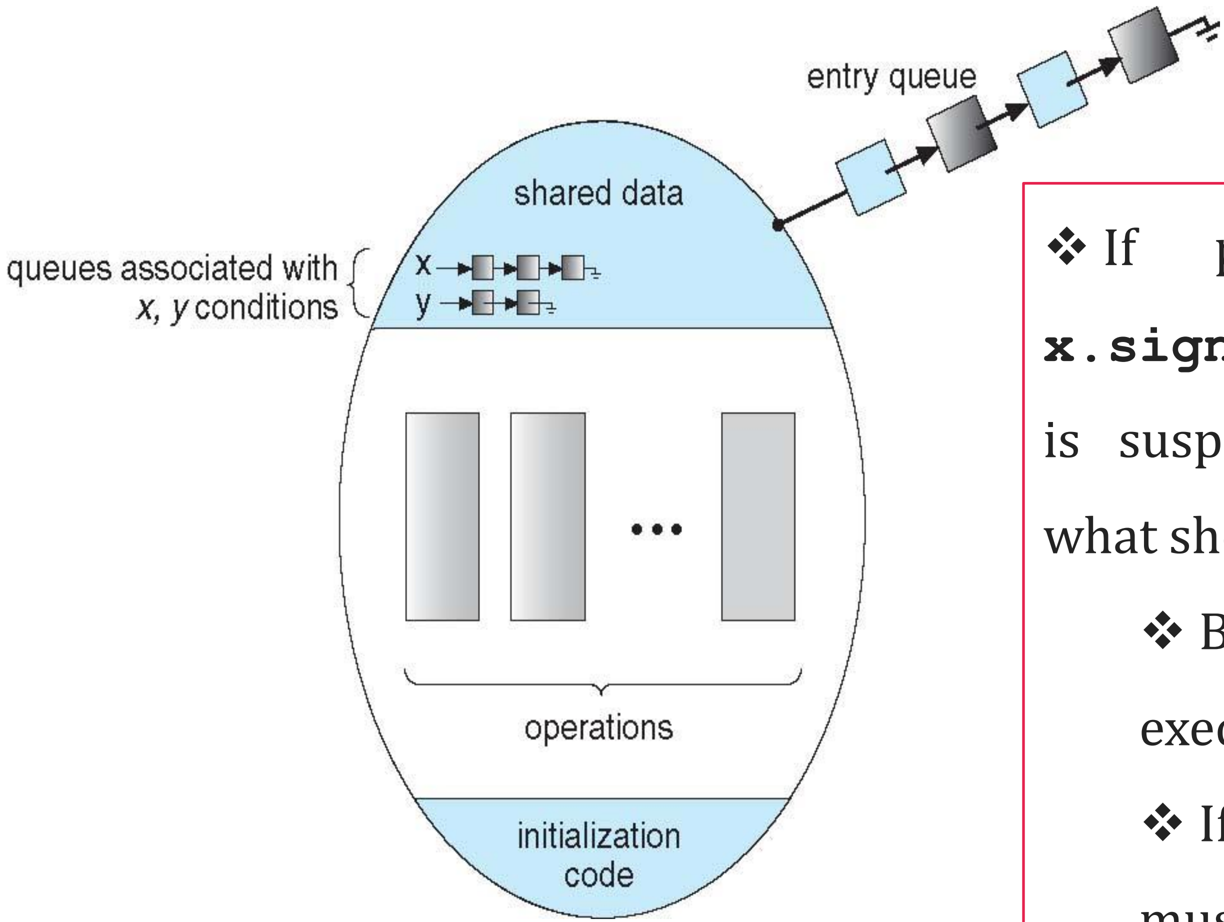
Monitors ...

Condition Variables

- Two operations are allowed on a condition variable:
  - **`x.wait()`** – a process that invokes the operation is 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.

Monitors ...

Monitor with Condition Variables



- ❖ 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 parallel.
  - ❖ If Q is resumed, then P must wait.

Monitors …

Condition Variables Choices

- Two possibilities exist
  - **Signal and wait** – P waits until Q either leaves the monitor or it waits for another condition.
  - **Signal and continue** – Q waits until P either leaves the monitor or it waits for another condition.
- Both options have pros and cons – language implementer can decide.
- Monitors implemented in Concurrent Pascal compromise between the two choices: when P executes signal, it immediately leaves the monitor & Q is resumed.

Monitors ...

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);
    }
```

Monitors …

Monitor Solution to Dining Philosophers …

```
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;
   }
}
```

Monitors …

Monitor Solution to Dining-Philosophers …

- 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 which leads a philosopher to death.

Monitors …

Resuming Process within a Monitor

- If several processes queued on a condition x, and x.signal() is executed, which process should be resumed?
- FCFS is used frequently even if it is not adequate.
- **conditional-wait** construct of the form x.wait(c)
  - Where c is **priority number**
  - Process with lowest number (highest priority) is scheduled next.

Monitors ...

Single Resource Allocation

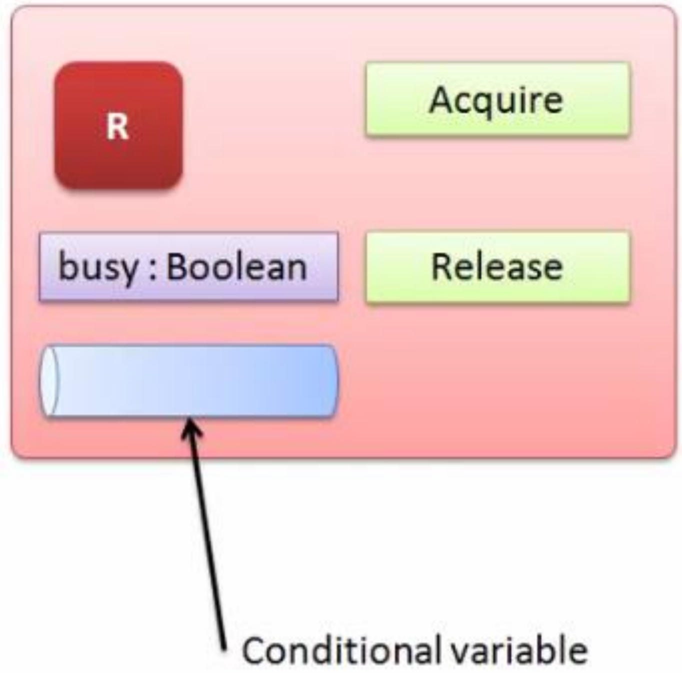
- ❖ Allocate a single resource among competing processes using priority numbers that specify the maximum time a process plans to use the resource.

```
R.acquire(t);
     ...
  access the resurce;
     ...
 R.release;
```

- ❖ Where R is an instance of type `ResourceAllocator`

Monitors …

A Monitor to Allocate Single Resource




```
monitor ResourceAllocator
{
  boolean busy;
  condition x;
  void acquire(int time) {
          if (busy)
            x.wait(time);
          busy = TRUE;
  }
  void release() {
          busy = FALSE;
          x.signal();
  }
initialization code() {
   busy = FALSE;
  }
}
```

Reference: Silberschatz et al., Operating System Concepts, Ninth Edition, 2013.

Questions???