ECE30021/ITP30002 Operating System

Synchronization

(OSC: Ch. 6)

This lecture note is taken from the instructor's resource of Operating System Concept, 9/e and then partly edited/revised by Shin Hong.

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 fills ***all*** the buffers. 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.

Naïve Concurrent Producer-Consumer

- Bounded Buffer as Circular Queue in Multithreaded Executions

```
Producer(item):

  while (counter == BUFSIZE) ;

  buffer[in] = item;
  in = (in + 1) % BUFSIZE;
  counter++;
```

```
Consumer():

  while (counter == 0) ;

  item = buffer[out];
  out = (out + 1) % BUFFER_SIZE;

  counter--;
  return item ;
```

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:

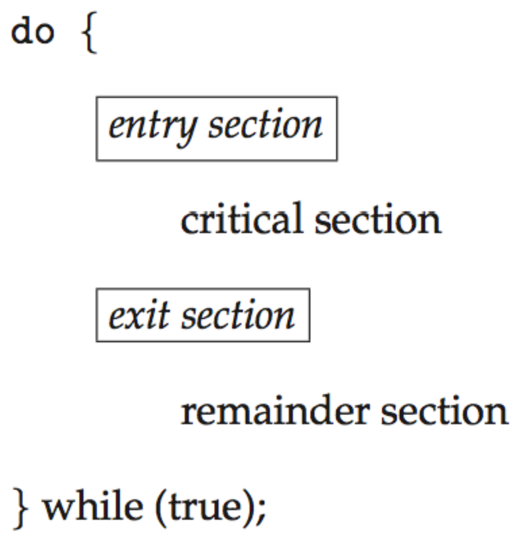
S0: producer execute `register1 = counter`	{register1 = 5}
S1: producer execute `register1 = register1 + 1`	{register1 = 6}
S2: consumer execute `register2 = counter`	{register2 = 5}
S3: consumer execute `register2 = register2 - 1`	{register2 = 4}
S4: producer execute `counter = register1`	{counter =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

- Race condition

multiple processes access (i.e., read and/or write) the same data concurrently, and the outcome depends on the order in which the accesses take place

Critical Section Policy

- Consider a system of ***n*** processes {p_0, p_1, … p_{n-1}}
- Each process has a segment of code as **critical section**
  - A 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 at the same time
- ***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**



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

Mechanism for Implementing Critical-Section

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

Critical-Section Handling in OS

- Two approaches depending on if kernel is preemptive or non-preemptive
  - Preemptive kernel allows preemption of process when running in kernel mode
  - Non-preemptive kernel runs until exits kernel mode, blocks, or voluntarily yields CPU
    - ~~Essentially free of race conditions in kernel mode~~

Peterson's Solution

- Software-based synchronization mechanism
  - 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;`**
    - to indicates whose turn it is to enter the critical section
  - **`Boolean flag[2]`**
    - to indicate if a process is ready to enter the critical section
      - **`flag[i] = `*`true`*** implies that process $\mathbf{P_i}$ is ready!

Algorithm

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

```
thread_0 { //_tid = 0
  …
  flag[_tid] = 1 ;
  turn = !_tid ;
  while (flag[!_tid] &&
         turn == !_tid);

  /*critical section*/

  flag[_tid] = 0;
  …
}
```

```
thread_1 { //_tid = 1
 …
  flag[_tid] = 1 ;
  turn = !_tid ;
  while (flag[!_tid] &&
         turn == !_tid);

  /*critical section*/

  flag[_tid] = 0 ;
  …
}
```

An Example of Wrong Mutual Exclusion

```
char cnt=0,x=0,y=0,z=0;

void process() {
     char me=_pid +1; /* me is 1 or 2*/
again:
        x = me;
        If (y ==0 || y== me) ;
        else goto again;

        z =me;
        If (x == me) ;
        else goto again;

        y=me;
        If(z==me);
        else goto again;

        /* enter critical section */
        cnt++;
        assert( cnt ==1);
        cnt --;
        goto again;
}
```

Software locks

Critical section

Mutual Exclusion Algorithm

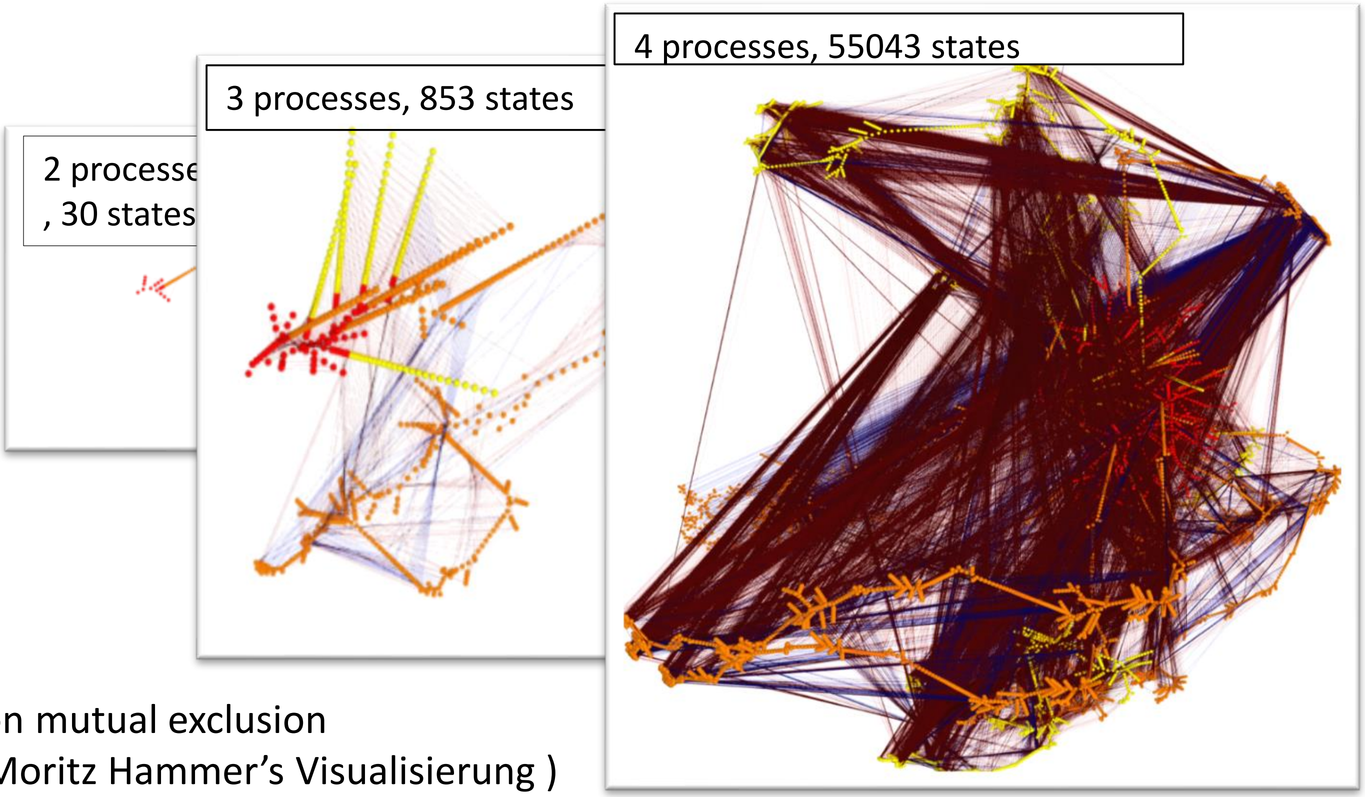
```
Process 0                              Process 1
                                       x = 2
                                       If(y==0 || y ==2)
                                       z = 2
                                       If(x==2)
x = 1
If(y==0 || y == 1)
                                       y=2
                                       If (z==2)
                                       cnt++
z = 1
If(x == 1)
y = 1
If(z == 1)
cnt++
```

Violation detected !!!

Counter Example

High Complexity of Peterson's algorithm

- The number of execution paths is exponential to the number of threads and the lengths due to non-deterministic thread scheduling
  - Testing technique for sequential programs do not properly work



Ex. Peterson mutual exclusion
(From Dr. Moritz Hammer's Visualisierung)

Synchronization Hardware

- Many systems provide hardware support for implementing the critical section code.
- All solutions below based on 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
    - Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
  - **Atomic** = non-interruptible
  - Test-and-set or Compare-and-swap

Solution to Critical-section Problem Using Locks

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

Test-and-set Instruction

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

volatile int m = 0

void
CriticalSection ()
{
  while (test_and_set(&m) == 1) ; // lock(&m) ;
  /* critical section */
  m = 0 ;                          // unlock(&m) ;
}
```

Compare-and-swap Instruction

```
int
compare_and_swap (int *var, int expected, int val)
{
  atomic {
    int t = *var ;
    if (t == expected)
      *var = val ;
  }
  return t:
}

volatile int lock = 0

void
CriticalSection ()
{
  while (compare_and_swap(&lock, 0, 1) == 1) ;

  /* critical section */

  lock = 0 ;
}
```

Mutex Locks

- OS designers build software tools to solve critical section problem
  - Previous solutions are complicated and generally inaccessible to application programmers

- Mutex is a locking variable on which a thread acquires and releases
  - A Boolean variable (i.e., lock variable) indicating if the lock is available or not
  - Protect a critical section by first **`acquire(m)`** on a lock **`m`** and then **`release(m)`** the lock
    - acquire(m) will be block when another thread already passes acquire(m) and does not release(m) yet
  - Usually implemented via hardware atomic instructions
  - Requires **busy waiting**

Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex) for process to synchronize their activities.
- Semaphore **`S`**
  - An integer variable (i.e., counter)
  - A waiting queue
  - Two atomic operations, **`wait()`** and **`signal()`** (i.e., also called **`P()`** and **`V()`**)
- **`wait(S)`:**

  1. decrease S (i.e., the counter) by 1

  2.a. when S is positive or zero, return immediately

  2.b. when S is negative, block by putting the thread to the waiting queue
- **`signal(S)`:**

  1. Increase S by 1

  1.a. when S is positive, return immediately

  1.b. when S is zero and negative,
  unblock a waiting thread in the queue

Semaphore Usage

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
  - Similar with a **mutex lock**
  - Can implement a counting semaphore ***S*** with a binary semaphore

- Ex. 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;
```

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

```
        P0                      P1

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

- **Starvation – indefinite blocking**
  - A process may never be removed from the semaphore queue in which it is suspended
- **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* buffers, each can hold one item

- Semaphore **`mutex`** initialized to the value 1
- Semaphore **`filledbuffer`** initialized to the value 0
- Semaphore **`emptybuffer`** initialized to the value *n*

Bounded Buffer Problem (Cont.)

Producer process

```
do {
       ...
      /* produce an item
       ...
    wait(emptybuffer);
    wait(mutex);
       ...
    /* add to the buffer */
       ...
    signal(mutex);
    signal(filledbuffer);
 } while (true);
```

Consumer process

```
do {
     wait(filledbuffer);
     wait(mutex);
     ...
     /* remove an item from buffer */
     ...
     signal(mutex);
     signal(emptybuffer);
     ...
     /* consume the 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
- Several variations of how readers and writers are considered – all involve some form of priorities
- Shared Data
  - Data set
  - Semaphore **`rw_mutex`** initialized to 1
  - Semaphore **`mutex`** initialized to 1
  - Integer **`read_count`** initialized to 0

Readers-Writers Problem (Cont.)

Writer:

```
do {
   wait(rw_mutex);

   /* writing is performed */


   signal(rw_mutex);

} while (true);
```

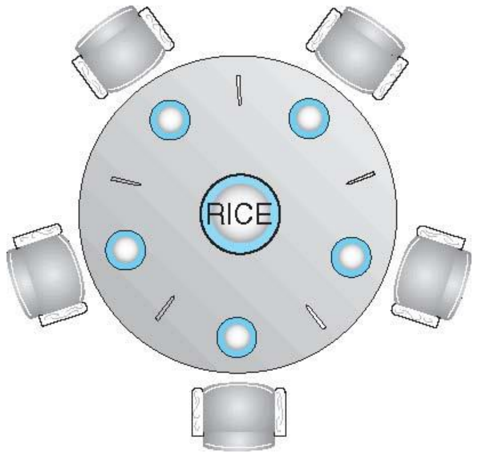
Reader:

```
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);
```

Readers-Writers Problem Variations

- ***First*** variation – no reader kept waiting unless writer has permission to use shared object
- ***Second*** variation – once writer is ready, it performs the write ASAP
- Both may have starvation leading to even more variations
- Problem is solved on some systems by kernel providing readers-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);
```

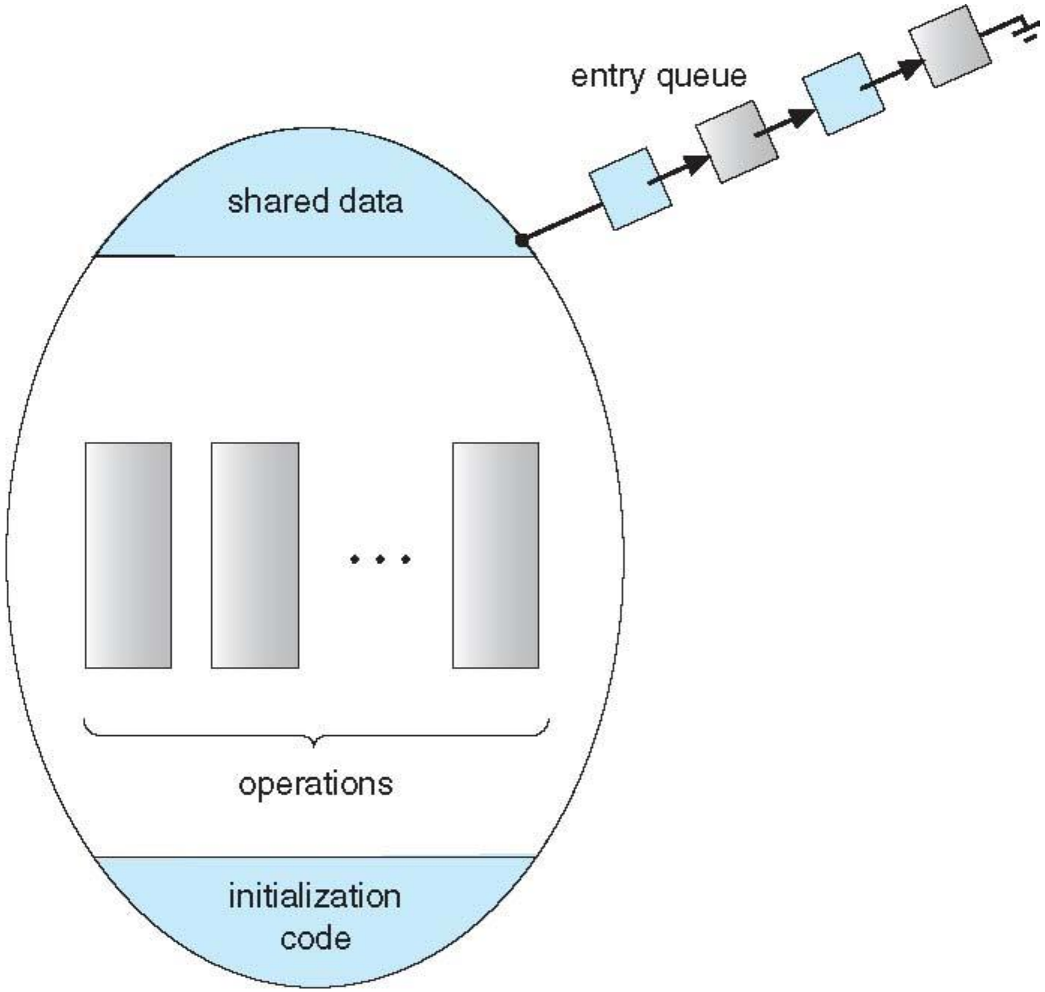
- What is the problem with this algorithm?

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
- But not powerful enough to model some synchronization schemes

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

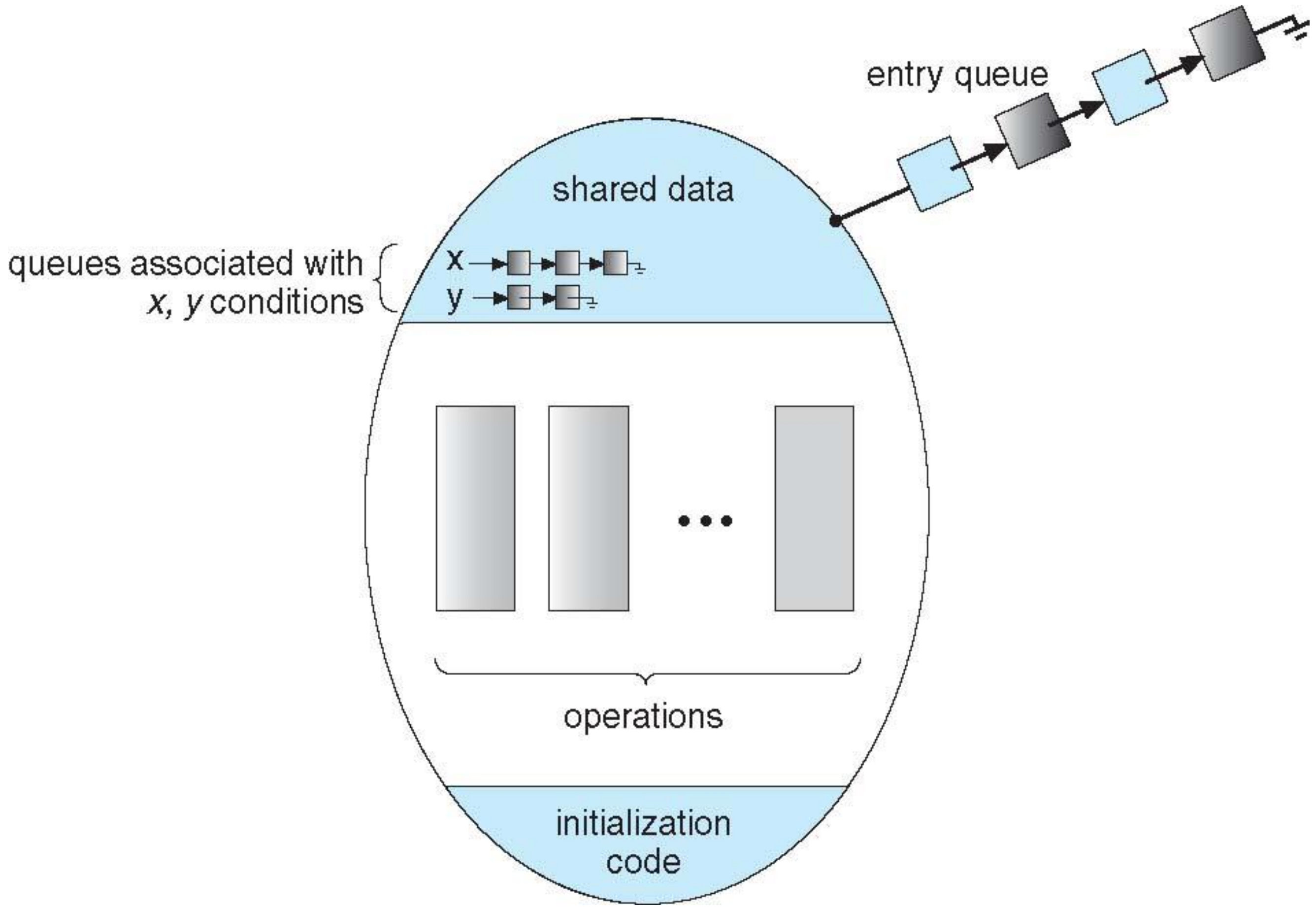
Schematic view of a Monitor



Condition Variables

- `condition x, y;`
- 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

Monitor with Condition Variables



Condition Variables Choices

- If proc P invokes `x.signal()` and proc 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
- Options include
  - **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 have pros and cons – language implementer can decide
  - Monitors implemented in Concurrent Pascal compromise
    - P executing signal immediately leaves the monitor, Q is resumed
  - Implemented in other languages including Mesa, C#, Java

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

```
monitor DiningPhilosophers {
  enum { THINKING, HUNGRY, EATING }
    state [5] ;
  condition self [5];

  pickup (int i) {
    state[i] = HUNGRY;
    tryeat(i);
    if (state[i] != EATING)
       self[i].wait;
  }

  putdown (int i) {
    state[i] = THINKING;
     // test left and right neighbors
    tryeat((i + 4) % 5);
    tryeat((i + 1) % 5);
  }

  tryeat (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

- 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