KOÇ UNIVERSITY

Synchronization-IV Monitors

Didem Unat

Lecture 11

COMP304 - Operating Systems (OS)

Semaphores

- We want to be able to write more complex constructs so need a language to do so. We define semaphores which we assume are atomic operations.
- Semaphores are more general synchronization tools
  - Operating System Primitive
  - Two standard atomic operations modify semaphore variable S: wait() and signal()

```
WAIT ( S ):

    while   ( S  <=  0 );
    S  =  S  -  1;
```

```
SIGNAL ( S ):

    S  =  S  +  1;
```

- As given here, these are not atomic as written in "macro code". We define these operations, however, to be atomic (Protected by a hardware lock.)

Semaphore as a general synchronization tool

- Provides mutual exclusion

```
Semaphore S =  1;    //  initialized to 1 or initialized to # of resources
wait (S);
    Critical Section
signal (S);
```

- Counting semaphore – integer value can range over an unrestricted domain
  - For example: resources in the hardware: semaphore is initialized to number of resources
- Binary semaphore – integer value can range only between 0 and 1; can be simpler to implement
  - This is the same as mutex locks

Semaphore as a general synchronization tool

- Semaphores can be used to force synchronization (precedence) if the **preceeder** does a signal at the end, and the **follower** does wait at beginning.

For example, here we want P1 to execute before P2.

- 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

Blocking Semaphores

```
typedef struct {
    int    value;
    struct process *list; /* list of processes waiting on  S  */
} SEMAPHORE;
```

```
SEMAPHORE s;
wait(s) {
      s.value = s.value  -  1;
      if (  s.value  <  0 ) {
         add this process to s.list;
          block ();
      }
}
```

```
SEMAPHORE s;
signal(s)  {
      s.value = s.value  +  1;
      if (  s.value  <=  0 )  {
         remove a process P from s.L;
         wakeup(P);
      }
}
```

Block – place the process invoking the operation on the appropriate waiting queue if semaphore is not available

Wakeup – Wakes up one of the blocked processes upon getting a signal and places the process to ready queue

Semaphores vs Locks

- Processes that are blocked at the level of program logic are placed on queues, rather than busy-waiting

- Busy-waiting may be used for the mutual exclusion
  - But these are very short critical sections

- Unlike locks, counting semaphores can take an integer value representing total number of resources

Problems with Semaphores (and Locks)

- Semaphores are shared global variables
  - Can be accessed from anywhere
- Used for both critical sections (mutual exclusion) and for coordination (scheduling or ordering execution)
- Incorrect use of semaphore operations
  - Call signal first and later on call wait
    - signal(mutex) …. wait(mutex)
  - Call wait after another wait
    - wait(mutex) …. wait(mutex)
  - Omitting of wait or signal
- Thus, they are prone to bugs
- To deal with such issues,
  - Introduce a high-level synchronization construct - **monitors**

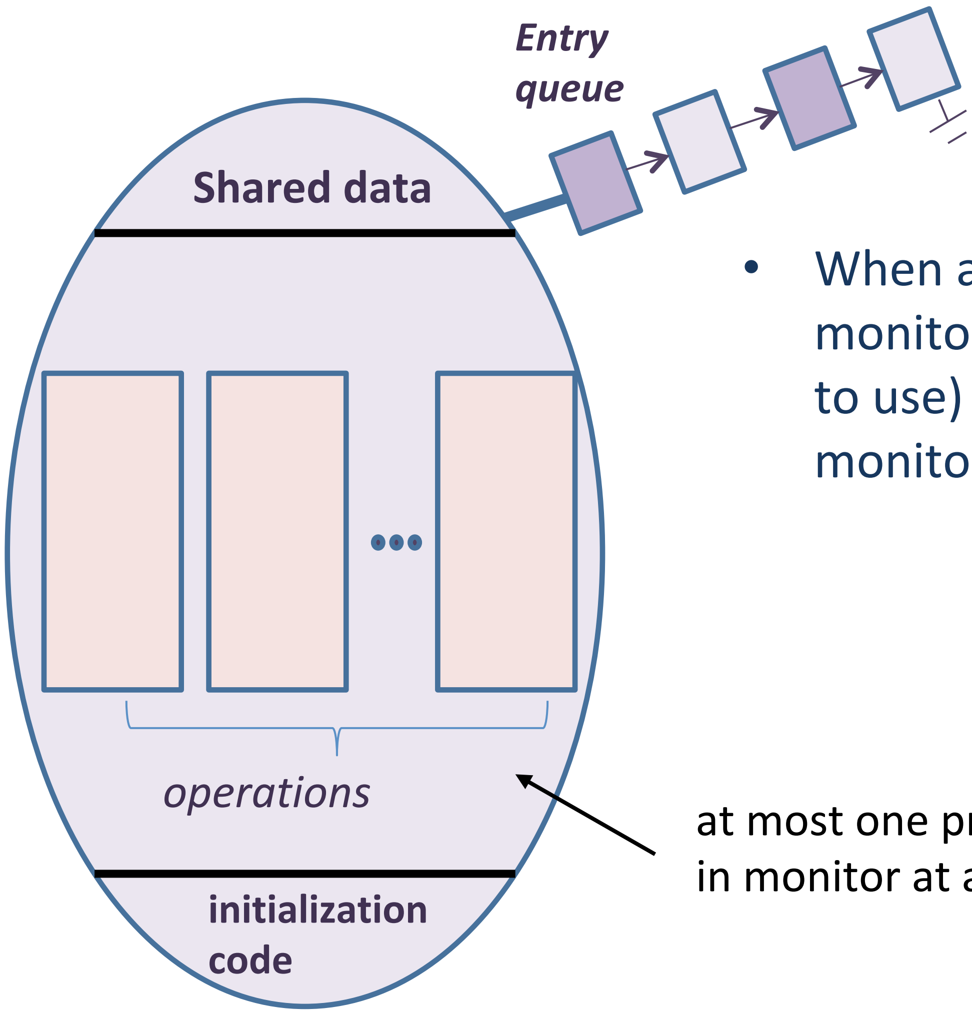
Monitors

- A monitor is a **programming language construct** that supports controlled access to shared data
  - First developed in Concurrent Pascal
  - It resembles an object-oriented approach for synchronization
- *A monitor encapsulates*
  - shared data structures
  - procedures that operate on the shared data (protects shared data)
  - synchronization between concurrent processes that invoke those procedures

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

    procedure Pn (…) {……}

   initialization(…) { … }

}
```

Monitor construct ensures that only one process at a time can be active within the monitor

Schematic View of a Monitor



- When a process is using the monitor, the others (that request to use) wait in the queue of the monitor

at most one process in monitor at a time

Example: Shared Balance

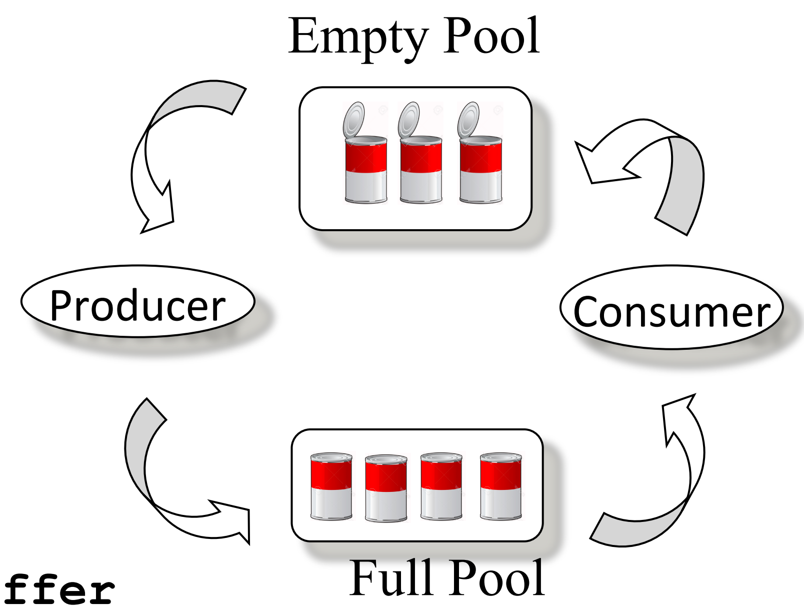
```
monitor sharedBalance {

    double balance;

    void deposit(double amount) {
       balance += amount;
    }
    void withdraw(double amount) {
       balance -= amount;
    }
    . . .
}
```

- Balance is a **shared** variable
- Deposit and withdraw are **procedures**
- Processes do not directly read/write into the shared variables but access them via these procedures

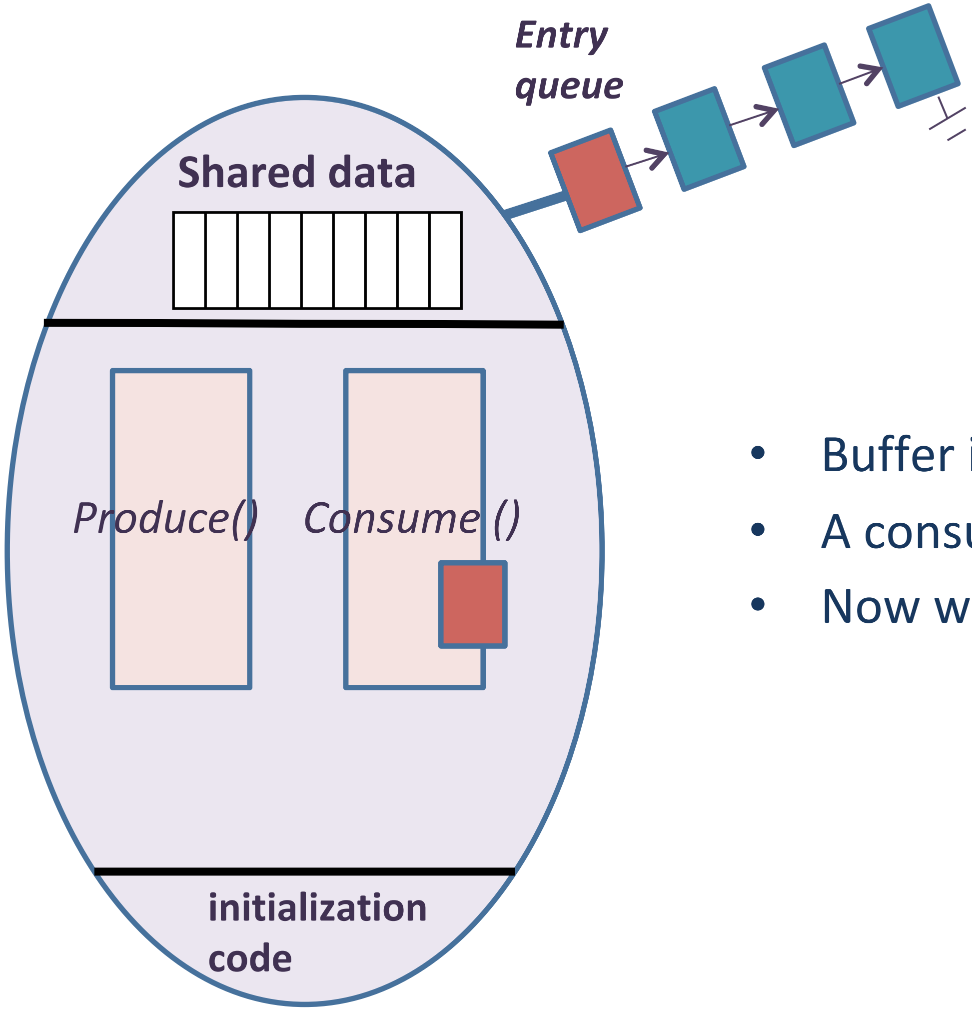
Producer-Consumer Problem

- Buffer size: buffer can hold **n** items

```
monitor ProducerConsumer {

    item buffer[N];

    void produce(item x) {
      //add an item to the buffer
    }
    void consume(item *x) {
      //remove an item from the buffer
    }
    . . .
}
```

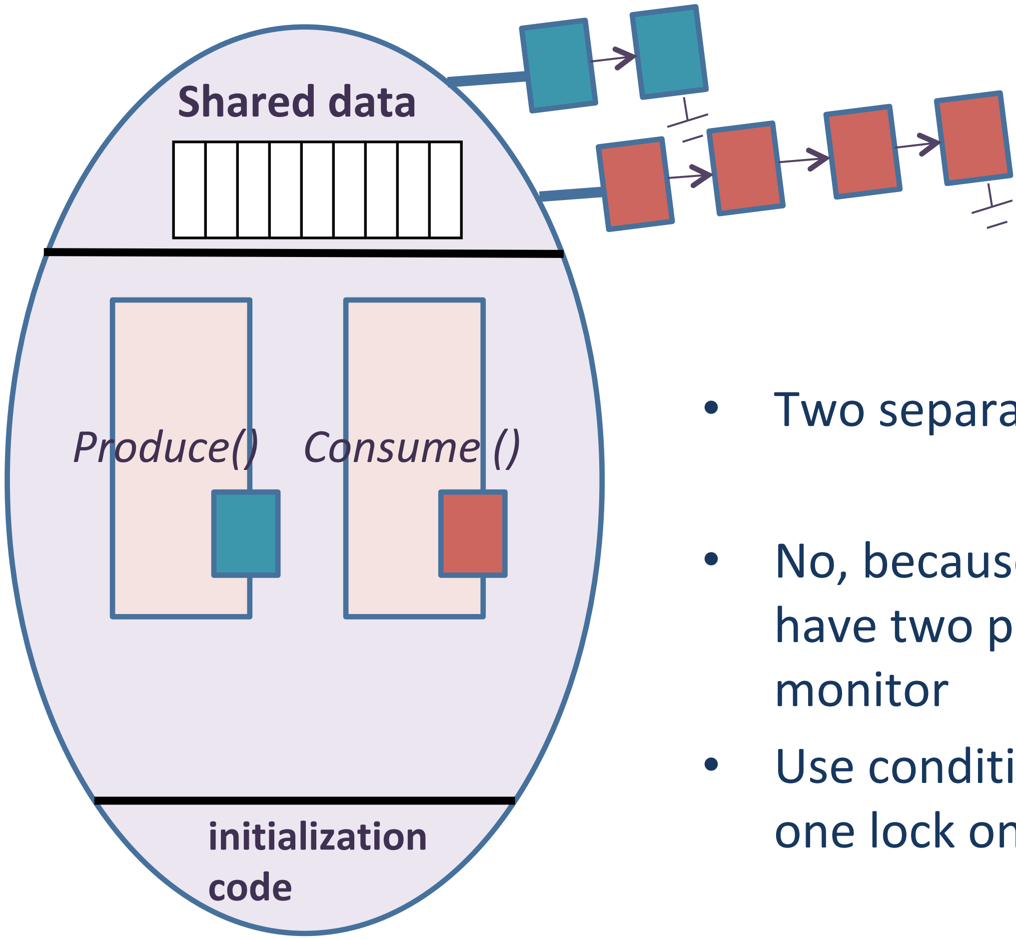


Example: Producer-Consumer Problem



- Buffer is empty
- A consumer (red) is in the queue
- Now what?

Example: Producer-Consumer Problem



- Two separate queues?
- No, because then we would have two processes in the monitor
- Use condition variables sharing one lock on the queue

Conditional Variables

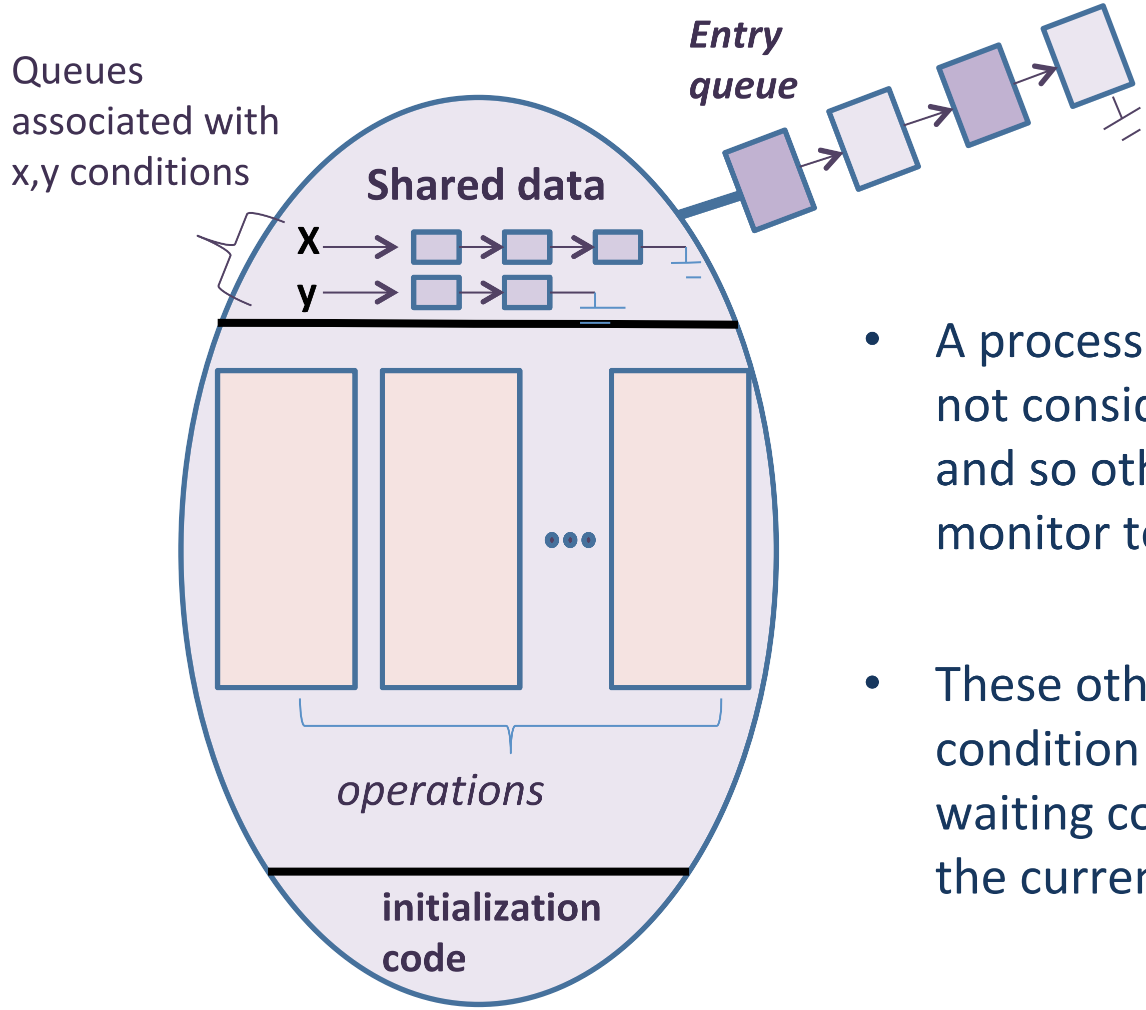
- Conceptually a condition variable is a queue of processes, associated with a monitor on which a process may wait for some condition to become true

- Sometimes called a rendezvous point
  - To allow a process to wait within the monitor, a **condition** variable must be declared, as

    condition c;

- Three operations on condition variables ‘c’
  - wait(c)
    - The calling process is suspended until another process invokes it
  - signal(c)
    - wake up at most one waiting process
    - if no waiting processes, signal has no effect
      - this is different than semaphores: no history!
  - broadcast(c)
    - wake up all waiting processes

Monitor with Condition Variables



- A process waiting for a condition is not considered to occupy the monitor, and so other processes may enter the monitor to change the monitor's state

- These other processes may signal the condition variable to indicate that waiting condition has become true in the current state

Producer and Consumer with Monitors

```
Monitor ProducerConsumer {
  item buffer[N];
  condition not_full, not_empty;

  produce(item x) {
    if (array "buffer" is full)// determined by a count
        wait(not_full);
    insert "x" in array "buffer"
    signal(not_empty);
  }

  consume(item *x) {
    if (array "buffer" is empty) //determined maybe by a count
         wait(not_empty);
    *x = get item from array "buffer"
    signal(not_full);

  }

}
```

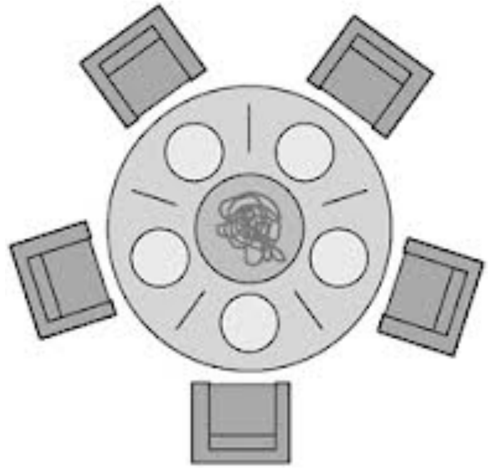
Producer and Consumer with Monitors

```
Monitor ProducerConsumer {
  item buffer[N];
  condition not_full, not_empty;
                                                          EnterMonitor
  produce(item x) {
    if (array "buffer" is full)// determined by a count
        wait(not_full);
    insert "x" in array "buffer"
    signal(not_empty);                                   ExitMonitor
  }

  consume(item *x) {
    if (array "buffer" is empty) //determined maybe by a count
        wait(not_empty);
    *x = get item from array "buffer"
    signal(not_full);

  }
}
```

- Monitor inserts the lock operations before at the entry of produce/ consume and releases the lock at the exit of produce/consume
- That's why we don't need a separate mutex lock for multiple producers (or consumers)

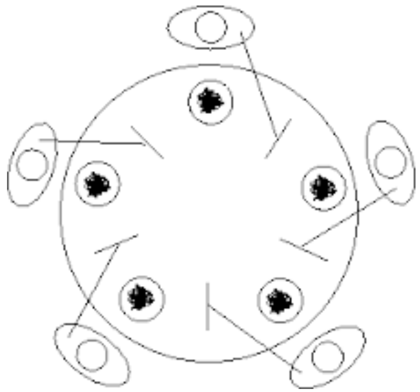
Dining Philosophers Problem

- 5 philosophers with 5 chopsticks sit around a circular table.
  - They each want to eat at random times
  - Must pick up the 2 chopsticks to eat
  - Pick one chopstick at a time
- While a philosopher is thinking, she drops the chopsticks on the table



Dining Philosophers Problem

- Shared data
  semaphore chopstick[5]; Initially all values are 1



Deadlock may occur!

Philosopher *i* :

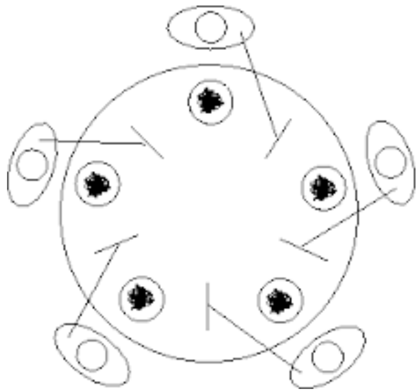
```
while (true) {
  wait(chopstick[i])
  wait(chopstick[(i+1) % 5])

  //  eat

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

  //  think
}
```

Dining Philosophers Problem

- Shared data
  semaphore chopstick[5]; Initially all values are 1

Philosopher *i* :

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

  //  eat

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

  //  think
}
```

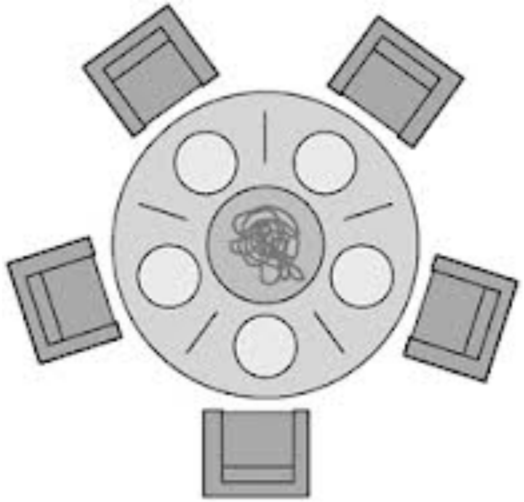


Starvation may occur!

Several Solutions

- Allow pickup only if both chopsticks are available

- Odd # philosophers always pick up left chopstick first, even # philosophers always pick up right chopstick first



A deadlock-free solution does not necessarily eliminate the possibility of starvation.

Monitor Solution to Dining Philosophers

- This implements a deadlock-free solutions with a restriction that a philosopher may pick up chopsticks only if both of them are available
- Is this solution starvation free?

```
monitor DP
 {
   enum { THINKING, HUNGRY, EATING) state [5] ;
   condition self [5];

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

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

Monitor 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;
   }
}
```

- If my neighbors are not in eating state and I am hungry, then I can eat!
- All philosophers' states are set to Thinking initially

Solution to DP Problem (cont.)

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

```
dp.pickup (i)

   EAT

dp.putdown (i)

   THINK
```

Language Support

- Java, C# and several other languages have support for Monitors
  - Here is a Java example

```
// To make a method synchronized, simply add the synchronized
// keyword to its declaration:

public class SynchronizedCounter {
    private int c = 0;
    public synchronized void increment() {
        c++;
    }
    public synchronized void decrement() {
        c--;
    }
    public synchronized int value() {
        return c;
    }
}
```

Summary of Monitors

- A **monitor** is a synchronization construct that allows processes
  - To have both mutual exclusion and
  - The ability to wait (block) for a certain condition to become true.
- Monitors also have a mechanism for signaling other processes that their condition has been met.
- A monitor consists of a mutex object and **condition variables**, procedures to access them

Summary of Synchronization

- Mutual exclusion is required to ensure no two concurrent processes are in their critical section at the same time
  - To prevent race conditions (corruption of shared data)
  - First identified and solved by Dijkstra in 1965
- Hardware solutions
  - Test-and-Test
  - Compare-and-Swap
- Software solutions (only provide higher-level abstractions to their hardware solutions)
  - Locks (busy-wait)
  - Semaphores (blocking, counting semaphores)
  - Monitors (high level language constructs)

OS Support for Synchronization

- Windows
  - Uses spinlocks on multiprocessor systems.
  - Also provides dispatcher objects which may act as mutexes and semaphores.
  - Dispatcher objects may also provide events. An event acts much like a condition variable.
- Linux
  - Disables interrupts to implement short critical sections
  - Provides semaphores and spinlocks
  - Pthreads: provides mutex locks and condition variables

Reading

- Read Chapter 6
- Wikipedia Article on Monitors
  - http://en.wikipedia.org/wiki/Monitor_%28synchronization%29

- Acknowledgments
  - These slides are adapted from
    - Öznur Özkasap (Koç University)
    - Operating System and Concepts (9th edition) Wiley
    - Jerry Breecher (Worcester Polytechnic Institute)
    - Mark Zbikowski and Gary Kimura (Univ. of Washington)