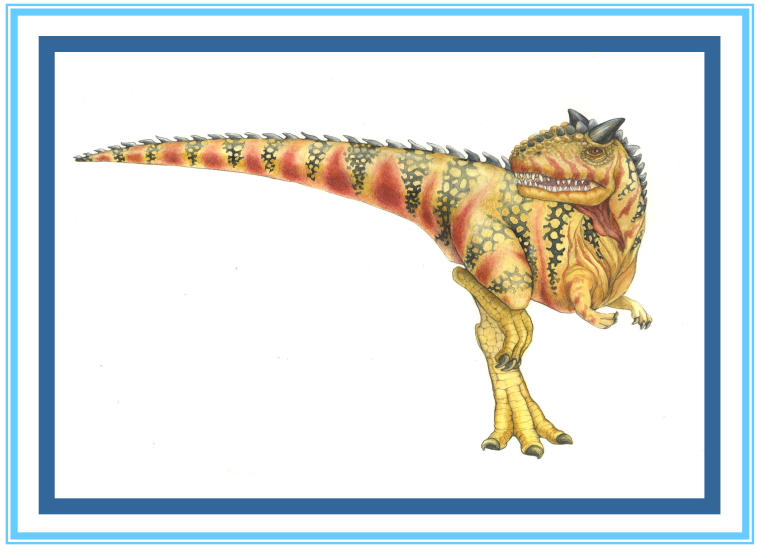
Chapter 6: Process Synchronization



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 msg initialized to the value 0   // message available
- Semaphore buf initialized to the value N   // buffer available

Bounded Buffer Problem (Cont.)

- The structure of the producer process

```
do
{

    //produce an item in nextp
    wait (buf);
    wait (mutex);

   //  add the item to the  buffer
    signal (mutex);
    signal (msg);
} while (TRUE);
```

Bounded Buffer Problem (Cont.)

- The structure of the consumer process

```
do {
        wait (msg);
        wait (mutex);
         //  remove an item from  buffer to nextc
        signal (mutex);
        signal (buf);
          //  consume the 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
- Several variations of how readers and writers are treated – all involve priorities
- Shared Data
  - Data set
  - Semaphore mutex initialized to 1
  - Semaphore wrt initialized to 1
  - Integer readcount initialized to 0

Readers-Writers Problem (Cont.)

- The structure of a writer process

```
do {
        wait (wrt) ;

        // writing is performed

        signal (wrt) ;
   } while (TRUE);
```

Readers-Writers Problem (Cont.)

- 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);
```

Readers-Writers Problem Variations

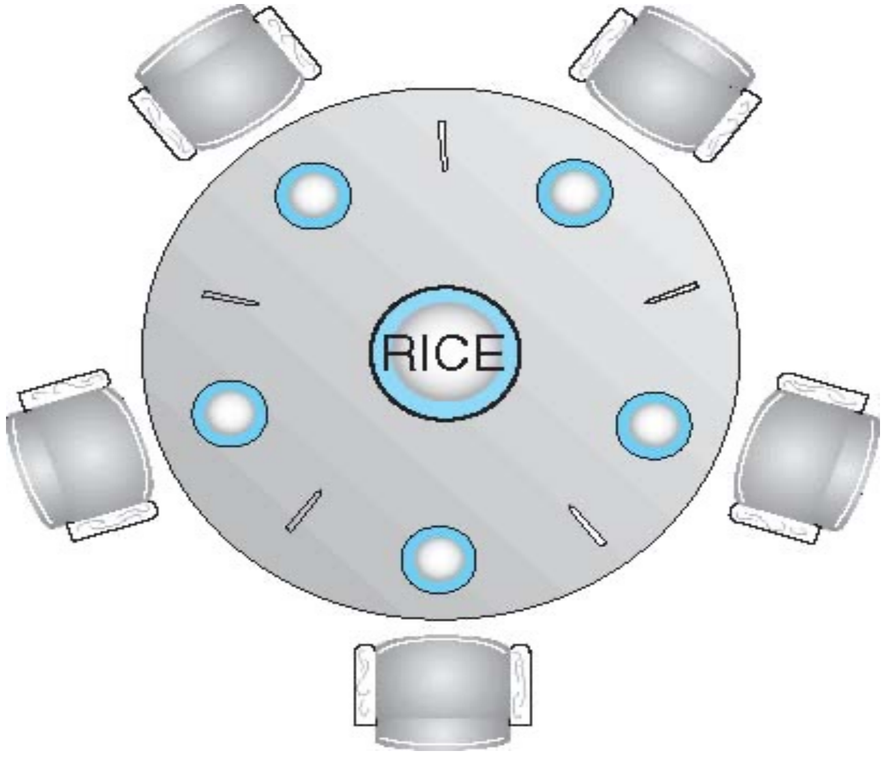
- *First* variation – no reader kept waiting unless writer has permission to use shared object

- *Second* variation – once writer is ready, it performs write asap

- Both may have starvation leading to even more variations

- Problem is solved on some systems by kernel providing reader-writer locks

Dining-Philosophers Problem



- Philosophers spend their lives 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?

Problems with Semaphores

- Incorrect use of semaphore operations:
  - signal (mutex) …. wait (mutex)
  - wait (mutex) … wait (mutex)
  - Omitting of wait (mutex) or signal (mutex) (or both)
- <u>Deadlock</u> and starvation

Monitors

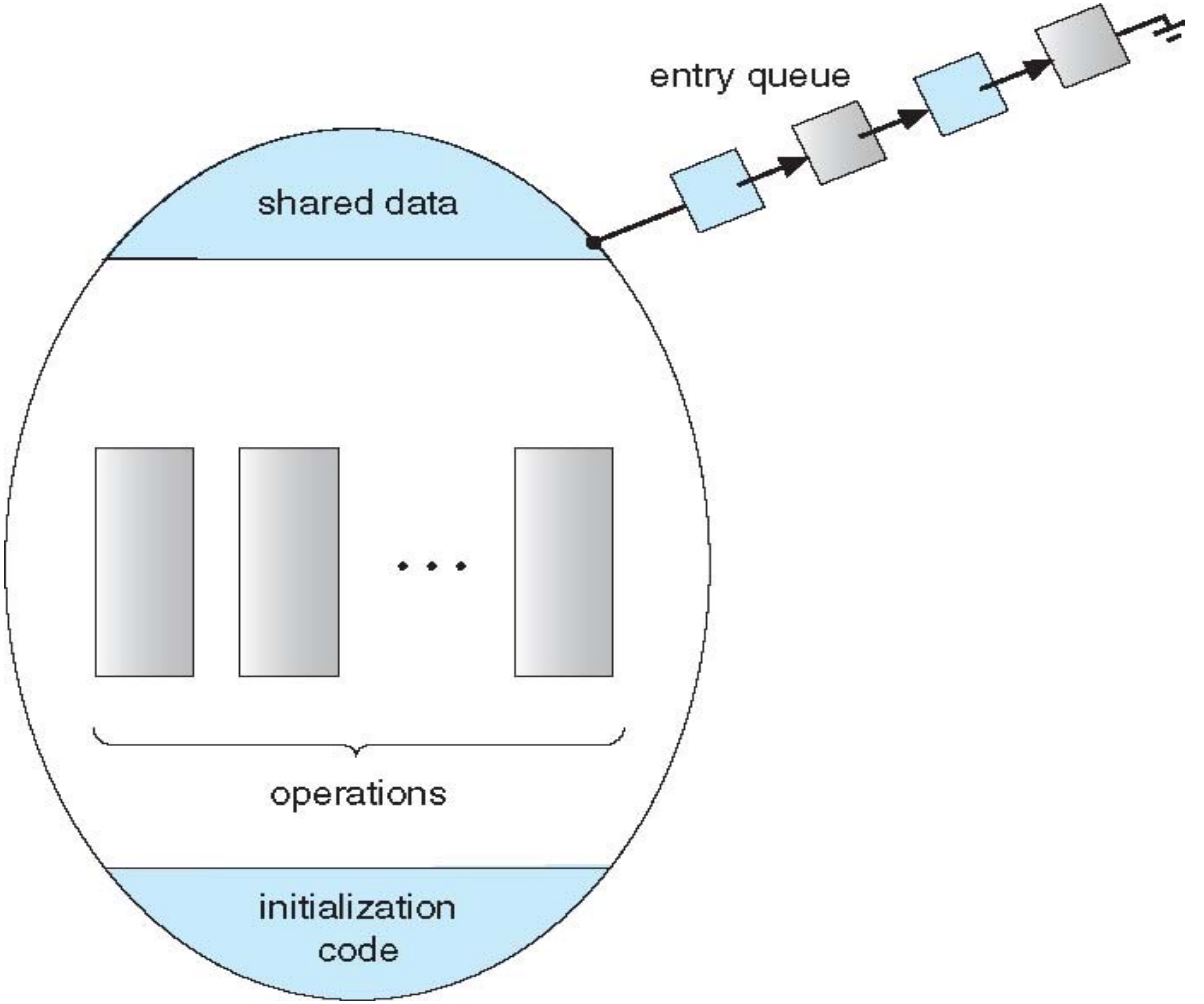
- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- *Abstract data type*, internal variables only accessible by code within the procedure
- Only one process may be active within the monitor at a time
- But not powerful enough to model some synchronization schemes

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

    procedure Pn (…) {……}

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

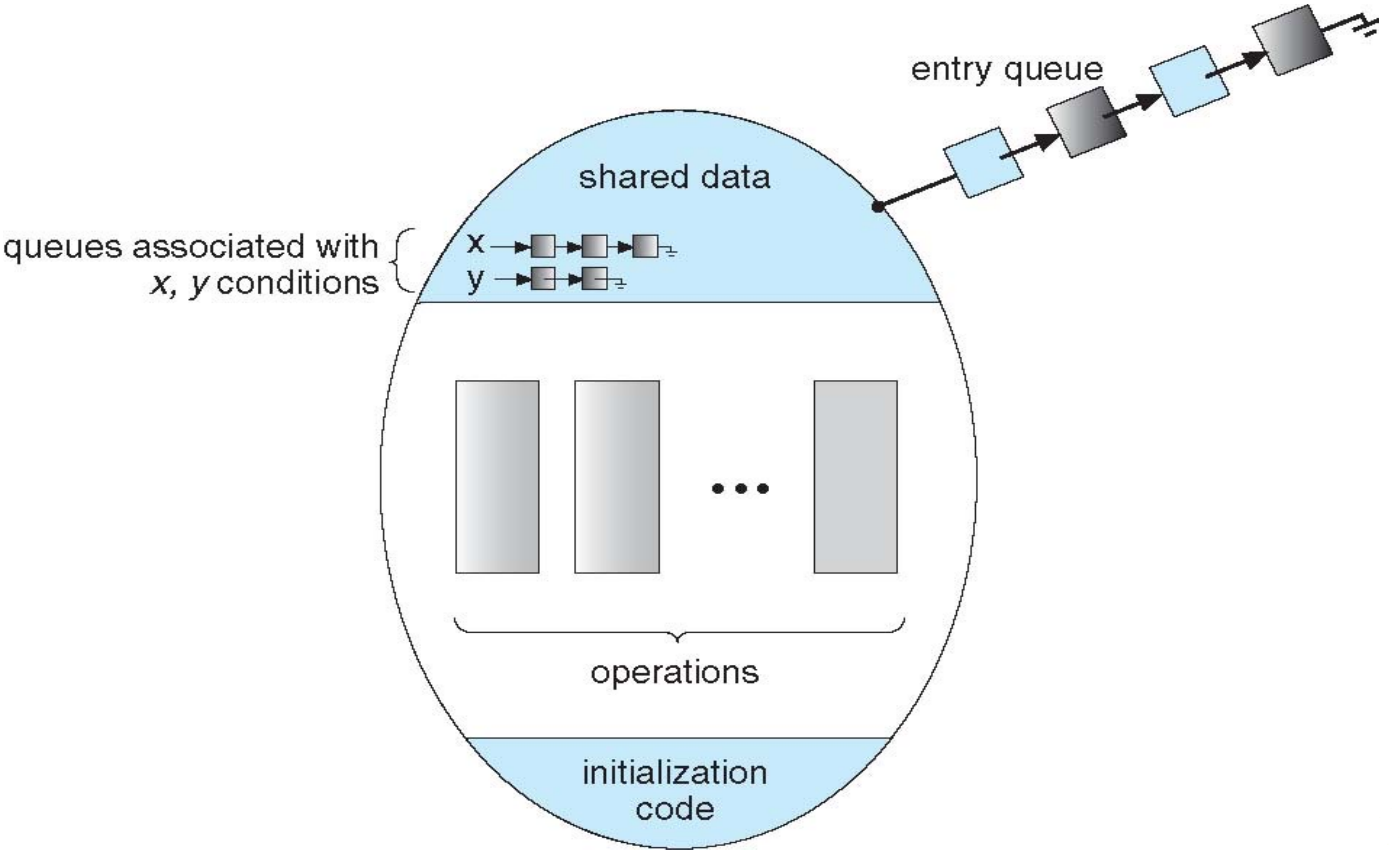
Schematic view of a Monitor



Condition Variables

- condition x, y;
- Two operations 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 process P invokes x.signal (), with Q in x.wait () state, what should happen next?
  - If Q is resumed, then P must wait
- Options include
  - **Signal and wait** – P waits until Q leaves monitor or waits for another condition
  - **Signal and continue** – Q waits until P leaves the monitor or waits for another condition
  - Both have pros and cons – language implementer can decide
  - Implemented in C#, Java

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

Monitor Implementation Using Semaphores

- **Signal and wait** – P waits until Q leaves monitor or waits for another condition
- Variables

```
semaphore mutex;  // (initially  = 1)
semaphore next;   // (initially  = 0)
int next_count = 0;
```

- Each procedure ***F*** will be replaced by

```
wait(mutex);
    …
      body of F;

    …
if (next_count > 0)
   signal(next)
else
   signal(mutex);
```

- Mutual exclusion within a monitor is ensured

Monitor 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--;
```

Monitor Implementation (Cont.)

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

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

Resuming Processes within a Monitor

- If several processes queued on condition x, and x.signal() executed, which should be resumed?
- FCFS frequently 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

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;
	}
}
```

Pthreads Synchronization

- Pthreads API is OS-independent
- It provides:
  - mutex locks
  - condition variables
- Non-portable extensions include:
  - read-write locks
  - spinlocks