CS 571

Operating Systems

# Process Synchronization & Race Conditions

Angelos Stavrou, George Mason University

# Process Synchronization

- Race Conditions
- The Critical Section Problem
- Synchronization Hardware
- Semaphores
- Classical Problems of Synchronization
- Monitors

# Concurrent Access to Shared Data

Suppose that two processes A and B have access to a shared variable "Balance":

PROCESS A:
Balance = Balance - 100

PROCESS B:
Balance = Balance - 200

Further, assume that Process A and Process B are executing concurrently in a time-shared, multi-programmed system.

# Concurrent Access to Shared Data

- The statement " Balance = Balance – 100" is implemented by several machine level instructions such as:
  - A1. LOAD R1, BALANCE // load Balance from memory into Register 1 (R1)
  - A2. SUB R1, 100 // Subtract 100 from R1
  - A3. STORE BALANCE, R1 // Store R1's contents back to the memory location of Balance.
- Similarly, "Balance = Balance – 200" can be implemented by the following:
  - B1. LOAD R1, BALANCE
  - B2. SUB R1, 200
  - B3. STORE BALANCE, R1

# Race Conditions

**Observe:** In a *time-shared* or multi-processing system the *exact instruction execution order* cannot be predicted!

**Scenario 1:**

A1. LOAD R1, BALANCE

A2. SUB R1, 100

A3. STORE BALANCE, R1

Context Switch!

B1. LOAD R1, BALANCE

B2. SUB R1, 200

B3. STORE BALANCE, R1

Balance is effectively decreased by 300!

**Scenario 2:**

A1. LOAD R1, BALANCE

A2. SUB R1, 100

Context Switch!

B1. LOAD R1, BALANCE

B2. SUB R1, 200

B3. STORE BALANCE, R1

Context Switch!

A3. STORE BALANCE, R1

Balance is effectively decreased by 100!

# Race Conditions

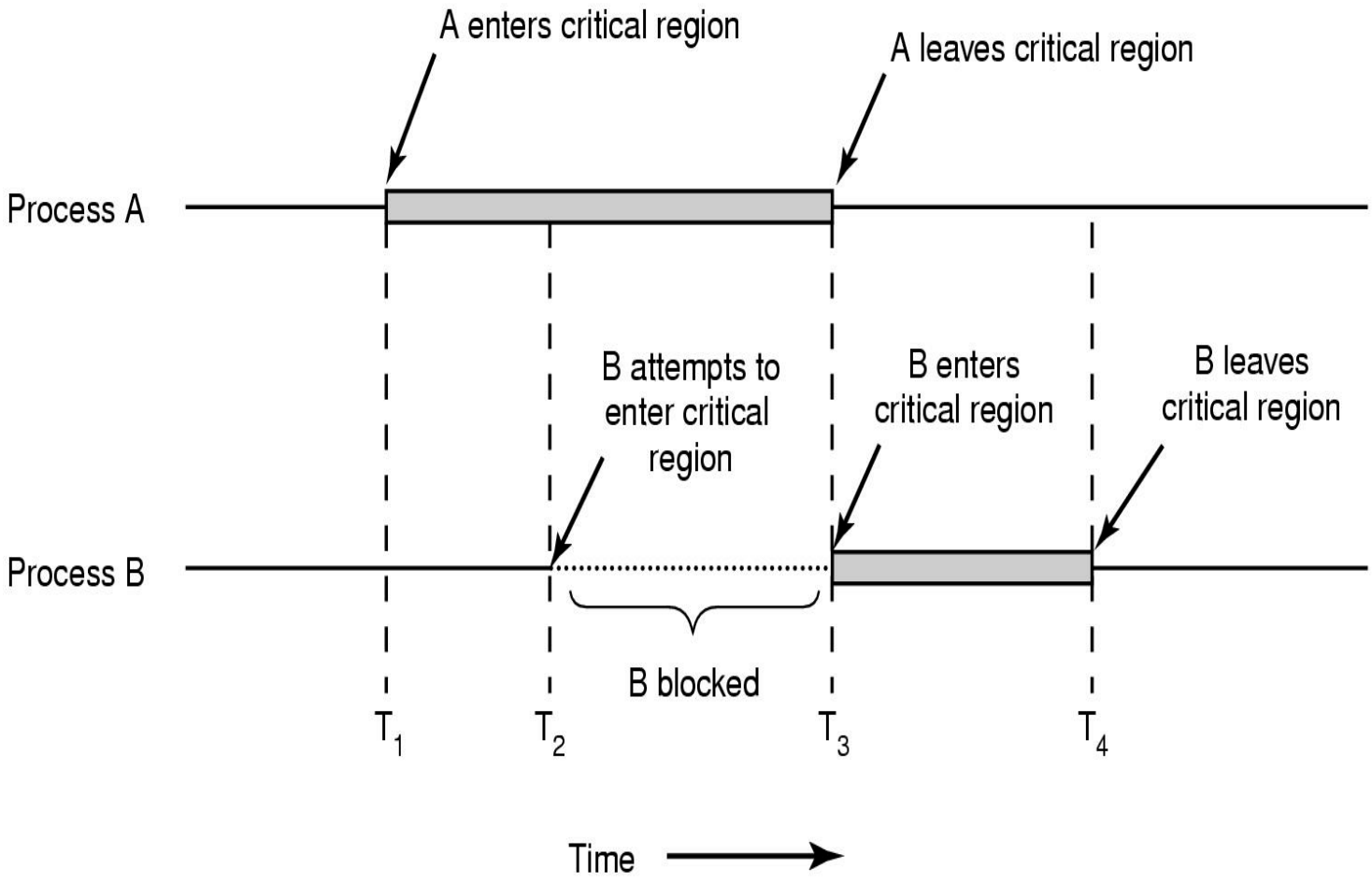
When multiple processes are accessing shared data without **access control** the final result depends on the execution order creating what we call **race conditions.**

- A serious problem for any concurrent system using shared variables!
- We need Access Control using code sections that are executed atomically.
- An **Atomic** operation is one that completes in its entirety without context switching (i.e. without interruption).

# The Critical-Section Problem

- Coordinate processes all competing to access shared data
- Each process has a code segment, called critical section (critical region), in which the shared data is accessed.
- Problem – ensure that when one process is executing in its critical section, no other process is allowed to execute in that critical section.
- The execution of the critical sections by the processes is **mutually exclusive.**
- **Critical section should be Short and with Bounded Waiting**

# Mutual Exclusion



# Solving Critical-Section Problem

**Any solution to the problem must satisfy four (4) conditions:**

## Mutual Exclusion:

- No two processes may be simultaneously inside the same critical section.

## Bounded Waiting:

- No process should have to wait forever to enter a critical section.

## Progress:

- No process running outside its critical region may block other processes.

## Arbitrary Speed:

- No assumption can be made about the relative speed of different processes (though all processes have a non-zero speed).

# General Structure of a Typical Process

```
while (1) {
        ....
          entry section
          critical section
          exit section
          remainder section
          }
```

- We will assume this structure when evaluating possible solutions to Critical Section Problem.
- In the entry section, the process requests "permission".
- We consider single-processor systems

# Getting Help from the Hardware

One solution supported by hardware may be to use interrupt capability:

```
do {
        DISABLE INTERRUPTS
            critical section;
        ENABLE INTERRUPTS
            remainder section
    } while (1);
```

- **Are we done?**

# Synchronization Hardware

- Many machines provide special hardware instructions that help to achieve mutual exclusion
- The TestAndSet (TAS) instruction tests and modifies the content of a memory word atomically
- TAS R1,LOCK
  reads the contents of the memory word LOCK into register R1, and stores a nonzero value (e.g. 1) at the memory word LOCK (again, atomically!)
  - Assume LOCK = 0
  - calling TAS R1, LOCK will set R1 to 0, and set LOCK to 1.
  - Assume LOCK = 1
  - calling TAS R1, LOCK will set R1 to 1, and set LOCK to 1.

# Mutual Exclusion with Test-and-Set

Initially, shared memory word LOCK = 0.

```
Process Pi
   while(1)
   {
        entry_section:
        TAS R1, LOCK
        CMP R1, #0   /* was LOCK 0? */
        JNE entry_section /* if not equal, jump to entry */
        critical section
        MOVE LOCK, #0   /* exit section */
        remainder section
    }
```

# Busy Waiting

**(a) Process 0.**

```
while (TRUE) {
    while (turn != 0)        /* loop */ ;
    critical_region( );
    turn = 1;
    noncritical_region( );
}
```

(a)

**(b) Process 1.**

```
while (TRUE) {
    while (turn != 1)        /* loop */ ;
    critical_region( );
    turn = 0;
    noncritical_region( );
}
```

(b)

**In both cases, be sure to note the semicolons terminating the while statements.**

# Busy Waiting

- This approach is based on busy waiting: if the critical section is being used, waiting processes loop continuously at the entry point.

- Disadvantages?

# Peterson's Solution

```
#define FALSE  0
#define TRUE   1
#define N      2                        /* number of processes */

int turn;                               /* whose turn is it? */
int interested[N];                      /* all values initially 0 (FALSE) */

void enter_region(int process);         /* process is 0 or 1 */
{
    int other;                          /* number of the other process */

    other = 1 – process;                /* the opposite of process */
    interested[process] = TRUE;         /* show that you are interested */
    turn = process;                     /* set flag */
    while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process)          /* process: who is leaving */
{
    interested[process] = FALSE;        /* indicate departure from critical region */
}
```

**Peterson's solution for achieving mutual exclusion.**

# Semaphores

Introduced by E.W. Dijkstra (in THE system in 1968)

Motivation: Avoid busy waiting by blocking a process execution until some condition is satisfied.

Semaphores support two operations:

**wait(semaphore)**: decrement, block until semaphore is open

- Also **P()**, after the Dutch word for test, or down()

**signal(semaphore):** increment, allow another thread to enter

- Also **V()** after the Dutch word for increment, or up()

# Semaphore Operations

Conceptually a semaphore has an integer value. This value is greater than or equal to 0.

wait(s):: wait/block until s.value > 0;

s.value-- ; /* Executed atomically! */

- A process executing the wait operation on a semaphore with value 0 is blocked until the semaphore's value becomes greater than 0.
  - No busy waiting
- signal(s):: s.value++; /* Executed atomically! */

# Semaphore Operations (cont.)

- If multiple processes are blocked on the same semaphore "s", only one of them will be awakened when another process performs signal(s) operation.
- Who will have priority?
- Carefully study **Section 6.5.2 of the textbook** to better understand how the semaphores are implemented at the kernel level.

# How Semaphores Work

- Associated with each semaphore is a queue of waiting processes

- When wait() is called by a thread:
  - If semaphore is open, thread continues
  - If semaphore is closed, thread blocks on queue

- Then signal() opens the semaphore:
  - If a thread is waiting on the queue, the thread is unblocked
  - If no threads are waiting on the queue, the signal is remembered for the next thread
    - In other words, signal() has "history" (c.f. condition vars later)
    - This "history" is a counter

# Critical Section Problem with Semaphores

**Shared data:**

```
semaphore mutex;  /* initially mutex = 1  */
```

**Process Pi:**

```
while(1) {
    wait(mutex);
        critical section
            signal(mutex);
        remainder section
}
```

# Re-visiting "Simultaneous Balance Update"

- **Shared data:**

  int Balance;

  semaphore mutex; // initially mutex = 1

- **Process A:**

  …….
  wait (mutex);
  Balance = Balance – 100;
  signal (mutex);
  ……

- **Process B:**

  …….
  wait (mutex);
  Balance = Balance – 200;
  signal (mutex);
  ……

# Semaphores as a Synchronization Mechanism

Semaphores provide a general process synchronization mechanism beyond the "critical section" problem.

Example problem: Execute B in Pj only after A executed in Pi

Use semaphore flag initialized to 0

Code:

| Pi : | Pj : |
|---|---|
| ⋮ | ⋮ |
| A | wait(flag) |
| signal(flag) | B |

# Readers-Writers Problem

- An object is shared among several threads
- Some threads only read the object, others only write it
- We can allow multiple readers
- But only one writer

How can we use semaphores to control access to the object to implement this protocol?

Use three variables

- int readcount – number of threads reading object
- Semaphore mutex – control access to readcount
- Semaphore w_or_r – exclusive writing or reading

# Readers-Writers Problem

```
// number of readers
int readcount = 0;
// mutual exclusion to readcount
Semaphore mutex = 1;
// exclusive writer or reader
Semaphore w_or_r = 1;

writer {
    wait(w_or_r); // lock out readers
    Write;
    signal(w_or_r); // up for grabs
}
```

```
reader {
    wait(mutex);       // lock readcount
    readcount += 1; // one more reader
    if (readcount == 1)
        wait(w_or_r); // synch w/ writers
    signal(mutex);   // unlock readcount
    Read;
    wait(mutex);       // lock readcount
    readcount -= 1; // one less reader
    if (readcount == 0)
        signal(w_or_r); // up for grabs
    signal(mutex);   // unlock readcount}
}
```

# Readers-Writers Problem

- If there is a writer:
  - First reader blocks on w_or_r
  - All other readers block on mutex
- Once a writer exits, all readers can fall through
  - Which reader gets to go first?
- The last reader to exit signals a waiting writer
- If readers and writers are waiting on w_or_r, and a writer exits, who goes first?
- Why doesn't a writer need to use mutex?

# Monitors

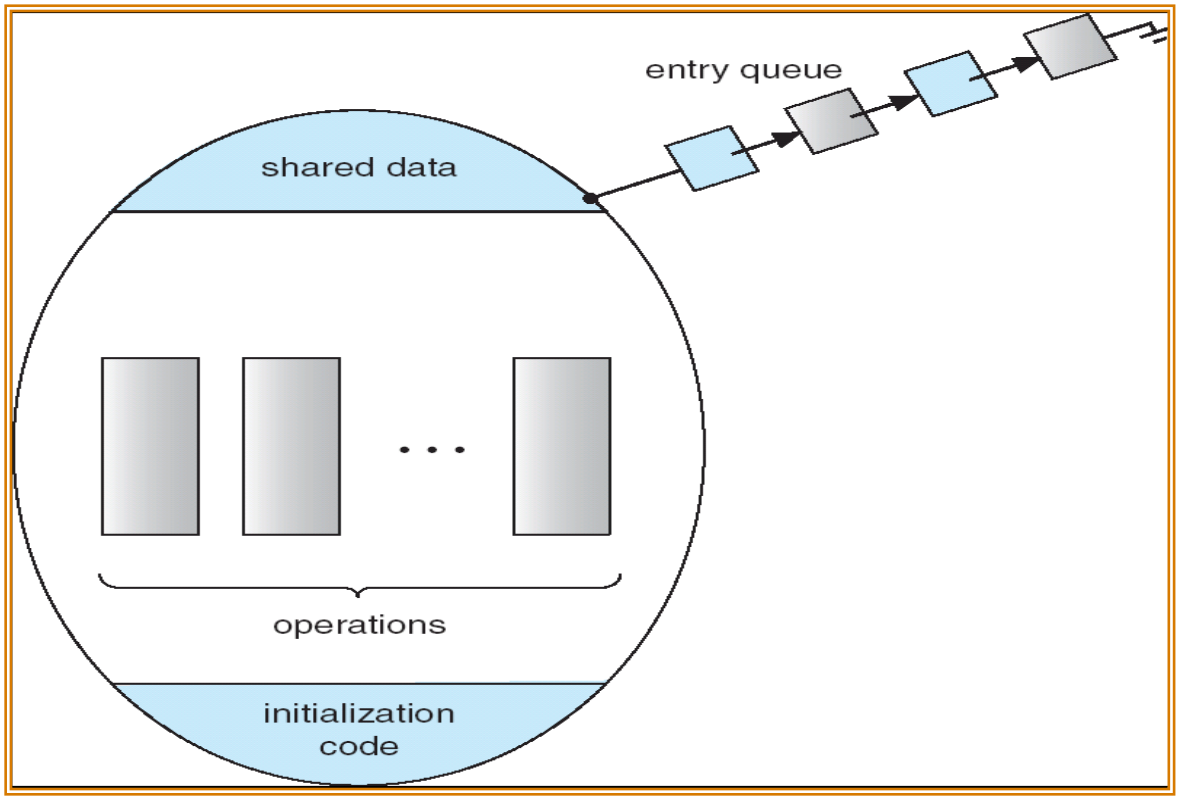
- A monitor is a programming language construct that controls access to shared data
  - Synchronization code added by compiler, enforced at runtime
  - Why is this an advantage?
- A monitor is a module that encapsulates
  - Shared data structures
  - Procedures that operate on shared data structures
  - Synchronization between concurrent procedure invocations
- A monitor protects its data from unstructured access
- It guarantees that threads accessing its data through its procedures interact only in legitimate ways

# Monitors

High-level synchronization construct that allows the safe sharing of an abstract data type among concurrent processes.

```
monitor monitor-name {
        shared variable declarations
        procedure body P1 (…) {
                              . . .
        }
        procedure body P2 (…) {
                              . . .
        }
        procedure body Pn (…) {
                                . . .
        }
        {
          initialization code
        }
}
```

# Schematic View of a Monitor




The monitor construct ensures that at most one process can be active within the monitor at a given time.

Shared data (local variables) of the monitor can be accessed only by local procedures.

# Monitors

To enable a process to wait within the monitor,

a **condition variable must be declared as condition**

Condition variable can only be used with the operations **wait** and **signal**.

The operation x.wait();
means that the process invoking this operation is suspended until another process invokes x.signal();

The x.signal operation resumes exactly one suspended process on condition variable x. If no process is suspended on condition variable x, then the signal operation has no effect.

**Wait and signal** operations of the monitors **are not the same** as **semaphore** wait and signal operations!

# Monitors

To enable a process to wait within the monitor,

a **condition variable must be declared as condition**

Condition variable can only be used with the operations **wait** and **signal**.

The operation x.wait();
means that the process invoking this operation is suspended until another process invokes x.signal();

The x.signal operation resumes exactly one suspended process on condition variable x. If no process is suspended on condition variable x, then the signal operation has no effect.

**Wait and signal** operations of the monitors **are not the same** as **semaphore** wait and signal operations!

# Conditional Variables

- Condition variables provide a mechanism to wait for events (a "rendezvous point")
  - Resource available, no more writers, etc.
- Condition variables support three operations:
  - **Wait** – release monitor lock, wait for C/V to be signaled
    - So condition variables have wait queues, too
  - **Signal** – wakeup one waiting thread
  - **Broadcast** – wakeup all waiting threads
- Note: Condition variables are not boolean objects
  - "if (condition_variable) then" … does not make sense
  - "if (num_resources == 0) then wait(resources_available)" does
  - An example will make this more clear

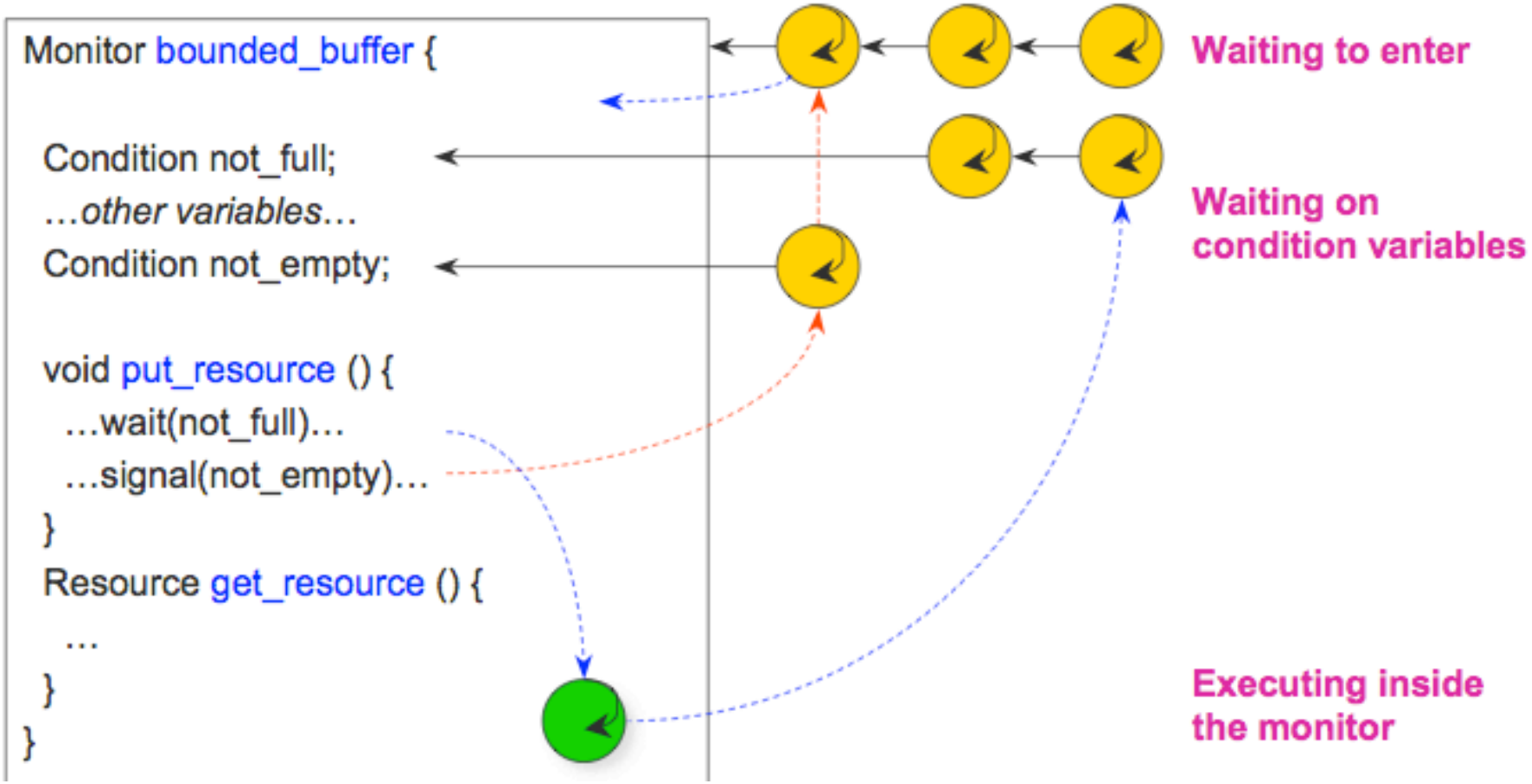
# Conditional Variables

```
Monitor bounded_buffer {
  Resource buffer[N];
  // Variables for indexing buffer
  Condition not_full, not_empty;

  void put_resource (Resource R) {
    while (buffer array is full)
        wait(not_full);
    Add R to buffer array;
    signal(not_empty);
  }
```

```
  Resource get_resource() {
    while (buffer array is empty)
        wait(not_empty);
    Get resource R from buffer array;
    signal(not_full);
    return R;
  }
} // end monitor
```

# Monitor Queues



# Monitor with Condition Variables

When a process P "signals" to wake up the process Q that was waiting on a condition, potentially both of them can be active.

However, monitor rules require that at most one process can be active within the monitor.
Who will go first?

- Signal-and-wait: P waits until Q leaves the monitor (or, until Q waits for another condition).
- Signal-and-continue: Q waits until P leaves the monitor (or, until P waits for another condition).
- Signal-and-leave: P has to leave the monitor after signaling

**The design decision is different for different programming languages**
**(CHECK JAVA and C)**

# Monitors vs Semaphores

- Which approach is more powerful?
  - Is there any synchronization problem that can be solved by semaphores but not monitors?
  - Is there any synchronization problem that can be solved by monitors but not semaphores?

# Monitors vs Semaphores (cont.)

**Semaphores (Disadvantages):**

Low-level:

- Easy to forget a Set or a Reset of the Semaphore

Scattered:

- Semaphore calls are scattered in the code
- Difficult and error-prone to debug (aliasing!).

# Summary

## Semaphores

- ✧wait()/signal() implement blocking mutual exclusion
- ✧Also used as atomic counters (counting semaphores)
- ✧Can be inconvenient to change and debug

## Monitors

Synchronizes execution within procedures that manipulate encapsulated data shared among procedures

- ✧Only one thread can execute within a monitor at a time
- ✧Relies upon high-level language support

## Condition variables

Used by threads as a synchronization point to wait for events Inside monitors, or outside with locks

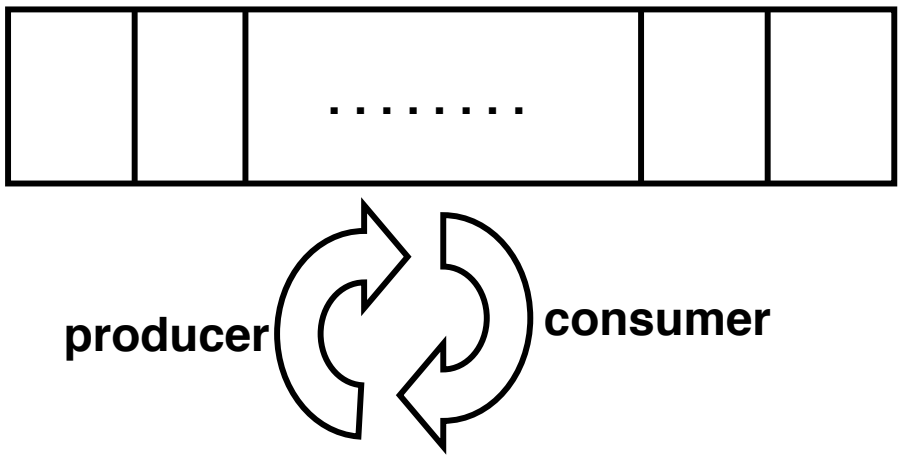
# Classical Problems of Synchronization

- Producer-Consumer Problem
- Readers-Writers Problem
- Dining-Philosophers Problem

The solutions will use only semaphores and monitors for synchronization. Busy waiting is best to be avoided.

# Producer-Consumer Problem

- The bounded-buffer producer-consumer problem assumes that there is a buffer of size n.
- The producer process puts items to the buffer area
- The consumer process consumes items from the buffer
- The producer and the consumer execute concurrently



# Producer-Consumer Problem (Cont.)

**The producer-consumer problem with a fatal race condition.**

```
#define N 100                                   /* number of slots in the buffer */
int count = 0;                                  /* number of items in the buffer */

void producer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          item = produce_item( );               /* generate next item */
          if (count == N) sleep( );             /* if buffer is full, go to sleep */
          insert_item(item);                    /* put item in buffer */
          count = count + 1;                    /* increment count of items in buffer */
          if (count == 1) wakeup(consumer);     /* was buffer empty? */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          if (count == 0) sleep( );             /* if buffer is empty, got to sleep */
          item = remove_item( );                /* take item out of buffer */
          count = count – 1;                    /* decrement count of items in buffer */
          if (count == N – 1) wakeup(producer); /* was buffer full? */
          consume_item(item);                   /* print item */
     }
}
```

# Producer-Consumer Problem (Cont.)

**Make sure that:**

- The producer and the consumer do not access the buffer area and related variables at the same time.
- No item is made available to the consumer if all the buffer slots are empty.
- No slot in the buffer is made available to the producer if all the buffer slots are full.

# Producer-Consumer Problem

**Shared data**

semaphore full, empty, mutex;

**Initially:**

full = 0 /* The number of full buffers */

empty = n /* The number of empty buffers */

mutex = 1 /* Semaphore controlling the access to the buffer pool */

# Producer Process

```
while (1) {
                    …
        produce an item in nextp
                    …
        wait(empty);
        wait(mutex);
                    …
        add nextp to buffer
                    …
        signal(mutex);
        signal(full);
};
```

# Consumer Process

```
while(1) {
          wait(full);
          wait(mutex);
                  …
          remove an item from buffer to nextc
                  …
          signal(mutex);
          signal(empty);
                  …
          consume the item in nextc
                  …
};
```

# Producer-Consumer Problem with Monitors

```
Monitor Producer-consumer
{
    condition full, empty;
    int count;
    void insert(int item); //the following slide
    int remove(); //the following slide
    void init() {
                    count = 0;
      }
}
```

# Producer-Consumer Problem with Monitors (Cont.)

```
void insert(int item)
   {
   if (count == N)   full.wait();
      insert_item(item);  // Add the new item
   count ++;
   if (count == 1)   empty.signal();
     }
int remove()
   { int m;
   if (count == 0)   empty.wait();
   m = remove_item(); // Retrieve one item
   count --;
   if (count == N -1)   full.signal();
      return m;
     }
```

# Producer-Consumer Problem with Monitors (Cont.)

```
void producer()      //Producer process
       {
       while (1) {
            item = Produce_Item();
            Producer-consumer.insert(item);
                  }
     }
void consumer()   //Consumer process
   {
       while (1) {
            item = Producer-consumer.remove(item);
            consume_item(item);
                  }
     }
```

# Readers-Writers Problem: Writer Process

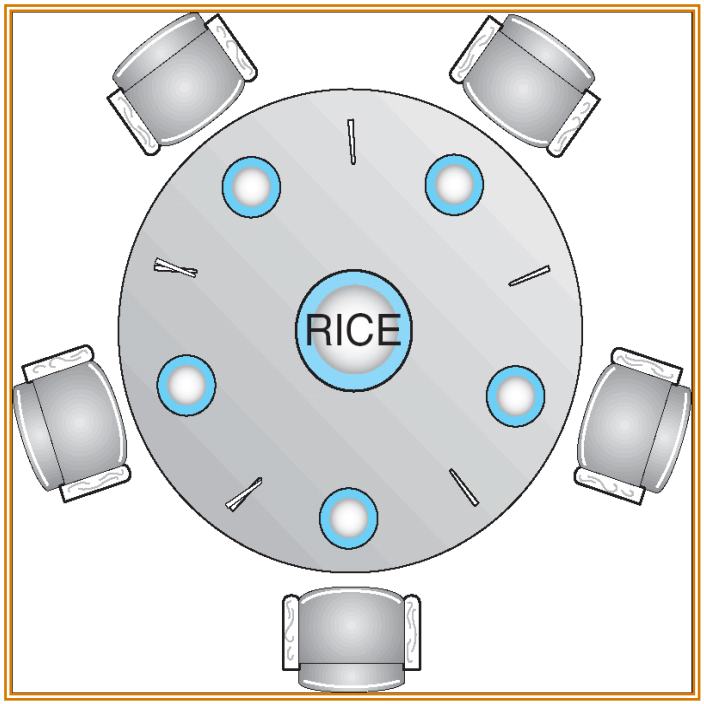
```
wait(wrt);
                    …
        writing is performed
                    …
signal(wrt);
```

# Readers-Writers Problem: Reader Process

```
wait(mutex);
readcount++;
if (readcount == 1)
                wait(wrt);
signal(mutex);
                  …
        reading is performed
                  …
wait(mutex);
readcount--;
if (readcount == 0)
        signal(wrt);
signal(mutex);
```

**Is this solution complete?**

# Dining-Philosophers Problem




Five philosophers share a common circular table. There are five chopsticks and a bowl of rice (in the middle). When a philosopher gets hungry, he tries to pick up the closest chopsticks.

A philosopher may pick up only one chopstick at a time, and cannot pick up a chopstick already in use. When done, he puts down both of his chopsticks, one after the other.

**Shared data**

semaphore chopstick[5];

**Initially** all semaphore values are 1

# Dining-Philosophers Problem

**Philosopher i:**

```
while (1) {
        wait(chopstick[i]);
        wait(chopstick[(i+1) % 5]);
                        …
                      eat
                        …
        signal(chopstick[i]);
        signal(chopstick[(i+1) % 5]);
                        …
                     think
                        …
      };
```

**Any Problems?**

# Dining-Philosophers Problem with Monitors

```
monitor dp
{
    enum {thinking, hungry, eating} state[5];
    condition self[5];
    void pickup(int i)       // following slides
    void putdown(int i)      // following slides
    void test(int i)         // following slides
    void init() {
          for (int i = 0; i < 5; i++)
                state[i] = thinking;
    }
}
```

# Dining-Philosophers Problem with Monitors

```
monitor dp
{
    enum {thinking, hungry, eating} state[5];
    condition self[5];
    void pickup(int i)      // following slides
    void                                ing slides
    void                                ing slides
    void
                state[i] = thinking;
    }
}
```

**Each philosopher will perform:**

**dp. pickup (i);**

**…eat..**

**dp. putdown(i);**

# Dining-Philosophers Problem with Monitors

```
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, wake them up
    // if possible
    test((i+4) % 5);
    test((i+1) % 5);
}
```

# Dining-Philosophers Problem with Monitors

```
void test(int i) {
    if ( (state[(i + 4) % 5] != eating) &&
      (state[i] == hungry) &&
      (state[(i + 1) % 5] != eating)) {
          state[i] = eating;
          self[i].signal();
    }
}
```

# Dining Philosophers problem (Cont.)

- Philosopher1 arrives and starts eating
- Philosopher2 arrives; he is suspended
- Philosopher3 arrives and starts eating
- Philosopher1 puts down the chopsticks, wakes up Philosopher2 (suspended once again)
- Philosopher1 re-arrives, and starts eating
- Philosopher3 puts down the chopsticks, wakes up Philosopher2 (suspended once again)
- Philosopher3 re-arrives, and starts eating
- ……

# Java "Monitors"

- Original Java definition included a "monitor-like" concurrency mechanism
  - When a method is defined as synchronized, calling the method requires owning the lock for the object.
  - If the lock is available when such a method is called, the calling thread becomes the owner of the lock and enters the method
  - If the lock is already owned (by another thread), the calling thread blocks and is put to an entry set
  - wait() and notify() methods are similar to the wait() and signal() statements we discussed for monitors

- Java provides support for semaphores, condition variables and mutex locks in java.util.concurrent package