EECS 111:

System Software

Lecture: Process Synchronization

Prof. Mohammad Al Faruque

The Henry Samueli School of Engineering
Electrical Engineering & Computer Science
University of California Irvine (UCI)

Chapter 5: Process Synchronization

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

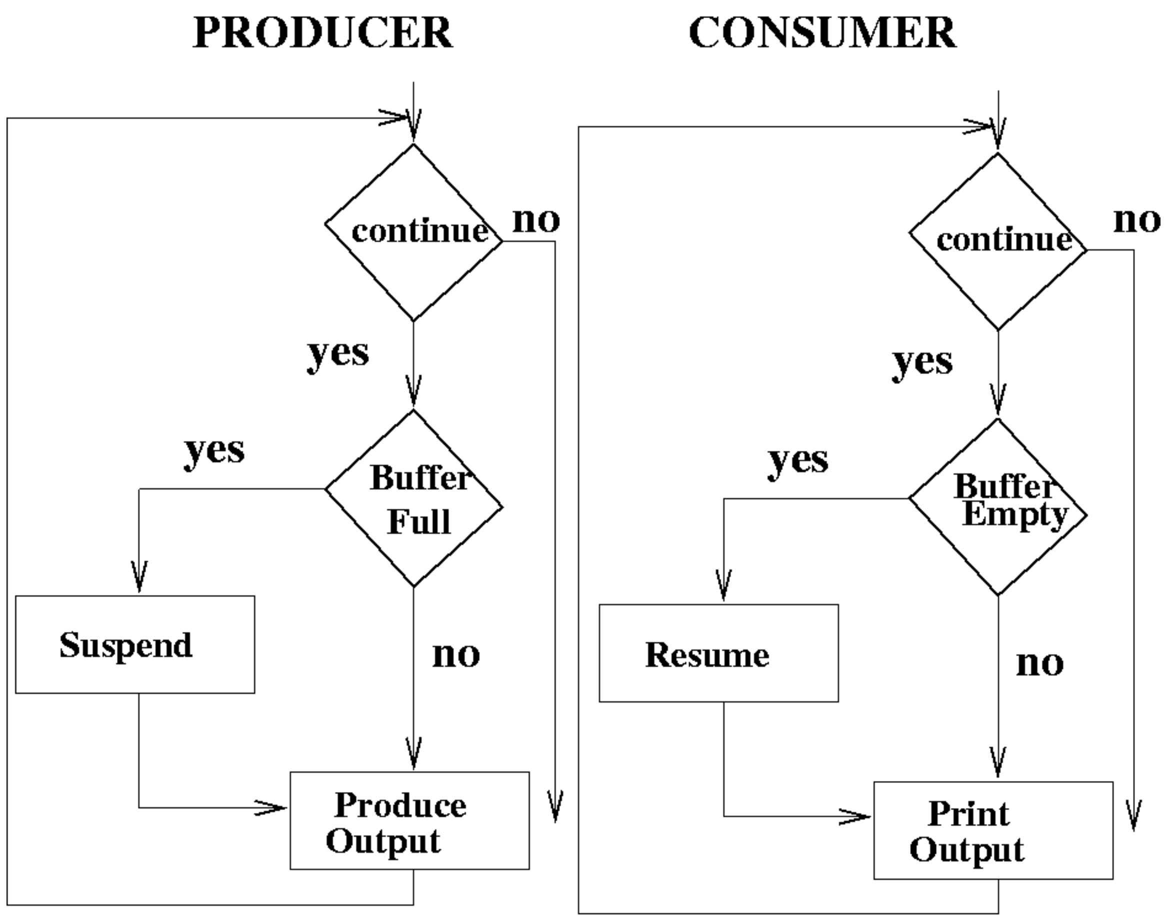
Background

- Processes can execute concurrently and/or in parallel
  - 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

Producer-Consumer Problem

- **Paradigm for cooperating processes;**
  - **producer process produces information that is consumed by a consumer process.**
- **We need buffer of items that can be filled by producer and emptied by consumer.**
  - Unbounded-buffer places no practical limit on the size of the buffer. Consumer may wait, producer never waits.
  - Bounded-buffer assumes that there is a fixed buffer size. Consumer waits for new item, producer waits if buffer is full.
  - **Producer and Consumer must synchronize.**

Producer-Consumer Problem



•

Bounded Buffer using IPC (messaging)

❑ Producer

```
message next produced;
while (true) {
     /* produce an item in next produced */
    send(next produced);
}
```

❑ Consumer

```
message next consumed;
while (true) {
    receive(next consumed);

    /* consume the item in next consumed */
}
```

Bounded-Buffer – Shared-Memory Solution

- **Shared data → This is a circular array**

```
#define BUFFER_SIZE 10
typedef struct {
   . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

- **Solution is correct, but can only use BUFFER_SIZE-1 elements**

Bounded-Buffer – Producer

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

Bounded Buffer – Consumer

```
item next consumed;
while (true) {
      while (in == out)
            ; /* do nothing */
      next consumed = buffer[out];
      out = (out + 1) % BUFFER SIZE;

      /* consume the item in next consumed */
}
```

- What will happen if producer process and the consumer process want to access the shared buffer concurrently?

Bounded-buffer - Shared Memory Solution

- **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.

Producer With Counter - Problem

```
while (true) {
       /* produce an item in next produced
*/

       while (counter == BUFFER SIZE) ;
             /* do nothing */
       buffer[in] = next produced;
       in = (in + 1) % BUFFER SIZE;
       counter++;
}
```

Consumer With Counter - Problem

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

Race Condition - Problem

- `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

Critical Section Problem

- Consider system of *n* processes $\{p_0, p_1, \ldots p_{n-1}\}$
- Each process has **critical section** segment of code
  - 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
- ***Critical section problem*** is to design a 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**

Critical Section

- General structure of process p_i is

```
do {
    [entry section]

        critical section

    [exit section]

        remainder section

} while (true);
```

Solution to Critical-Section Problem → 3 Requirements to fulfill

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 and 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**

Solution to Critical-Section Problem → 3 Requirements to fulfill

- **Two approaches depending on → if kernel is preemptive or non-preemptive**
  - **Preemptive – allows preemption of process when running in kernel mode**
    - Still good as it is more responsive
    - Most important today is for real-time systems
  - **Non-preemptive – runs until exits kernel mode, blocks, or voluntarily yields CPU**
    - Essentially free of race conditions in kernel mode

Peterson's Solution

- Good algorithmic description of solving the problem
  - Two process solution
  - Assume that the load and store instructions are atomic; → that is, 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 Pi is ready!

Algorithm for Process Pi

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

- ❑ **Provable that →**

1. **Mutual exclusion is preserved**
2. **Progress requirement is satisfied**
3. **Bounded-waiting requirement is met**

Do it yourself.

Chapter 5: Process Synchronization

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

Supporting Synchronization

Programs	*Shared Programs*
Higher-level API	*Locks Semaphores Monitors Send/Receive CCregions*
Hardware	*Load/Store Disable Ints Test&Set Comp&Swap*

- **We are going to implement various synchronization primitives using atomic operations**
  - **Everything is pretty painful if only atomic primitives are load and store**
  - **Need to provide inherent support for synchronization at the hardware level**
  - **Need to provide primitives useful at software/user level**

Synchronization Hardware

- **Many systems provide hardware support for 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
  - **Either test memory word and set value**
  - **Or swap contents of two memory words**

Solution to Critical-section Problem Using Locks

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

Test_and_Set Instruction

Definition:

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

Solution using Test_and_Set()

Shared boolean variable lock, initialized to FALSE

Solution:

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

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;
}
```

Solution using compare_and_swap

Shared Boolean variable lock initialized to FALSE; Each process has a local Boolean variable key

Solution:

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

Bounded-waiting Mutual Exclusion with Test_and_Set

```
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);
```

Prove that all 3 requirements are met!!!!

Supporting Synchronization

Programs	***Shared Programs***
Higher-level API	***Locks Semaphores Monitors Send/Receive CCregions***
Hardware	***Load/Store Disable Ints Test&Set Comp&Swap***

Mutex Locks

- **Previous solutions are complicated and generally inaccessible to application programmers**
- **OS designers build software tools to solve critical section problem**
- **Simplest is mutex lock**
- **Product critical regions with it by first `acquire()` a lock then `release()` it**
  - **Boolean variable indicating if lock is available or not**
- **Calls to `acquire()` and `release()` must be atomic**
  - **Usually implemented via hardware atomic instructions (may be with test_and_set or compare_and_swap type instructions)**
- **But this solution requires busy waiting**
  - This lock therefore called a **spinlock**

acquire() and release()

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

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

Semaphore

- Synchronization tool that does not require busy waiting
- Semaphore *S* – integer variable
- Two standard operations modify S: wait() and signal()
  - Originally called `P()` and `V()`
- Less complicated
- Can only be accessed via two indivisible (atomic) operations

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

Semaphore Usage

- **Counting semaphore – integer value can range over an unrestricted domain**
- **Binary semaphore – integer value can range only between 0 and 1**
  - **Then a mutex lock ☺**
- **Can implement a counting semaphore *S* to control access to a given resource consisting of a finite number of instances**
- **Consider P_1 and P_2 that require S_1 to happen before S_2**

```
P1:
    S1;
    signal(synch);
P2:
    wait(synch);
    S2;
```

Semaphore Implementation

- **Must guarantee that no two processes can execute `wait ()` and `signal ()` on the same semaphore at the same time**
- **Thus, implementation becomes the critical section problem where the wait and signal code are placed in the critical section**
  - **Could now have busy waiting in critical section implementation**
    - But implementation code is short
    - Little busy waiting if critical section rarely occupied
- **Note that applications may spend lots of time in critical sections and therefore this is not a good solution**

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 → pointer to a list of PCBs
- 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**

Semaphore Implementation with no Busy waiting (Cont.)

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

Block () and wakeup (p) are the basic system calls

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**

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 → semaphore in LIFO
- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
  - Solved via priority-inheritance protocol

Process Synchronization

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

Classical Problems of Synchronization

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

Bounded-Buffer Problem

- ***n*** **buffers, each can hold one item**
    1. Semaphore mutex initialized to the value 1
    2. Semaphore full initialized to the value 0
    3. Semaphore empty initialized to the value n

Bounded Buffer Problem (Cont.)

The 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);
```

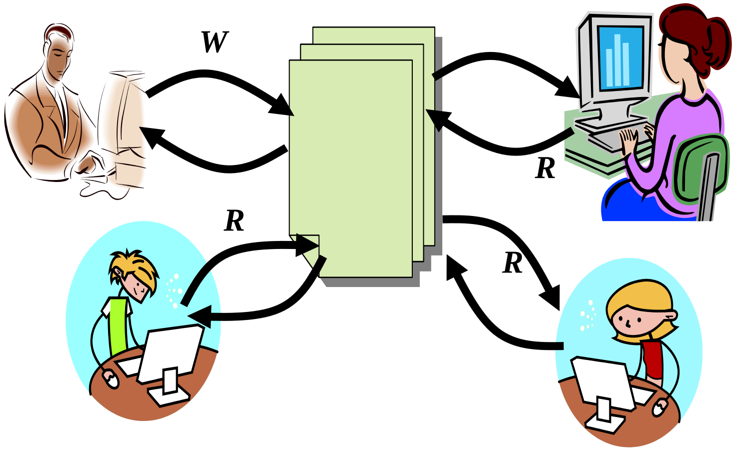
Bounded Buffer Problem (Cont.)

The 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);
```

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



Readers-Writers Problem Variations

- **First variation – <u>no reader kept waiting unless writer has permission already to use shared object</u>**
- **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**

Readers-Writers Solution- First Variation

- **Shared Data**
  - **Semaphore `rw_mutex` initialized to 1 →**
    - common to both reader and writer processes
    - Also used by first or last reader that enters or exits the critical section
  - **Semaphore `mutex` initialized to 1 →**
    - this is used to ensure to ensure mutual exclusion when variable read_count is updated
  - **Integer `read_count` initialized to 0**

Readers-Writers Problem (Cont.)

The structure of a writer process

```
do {
   wait(rw_mutex);

       ...
   /* writing is performed */

       ...

   signal(rw_mutex);

} while (true);
```

First Reader-writers solution

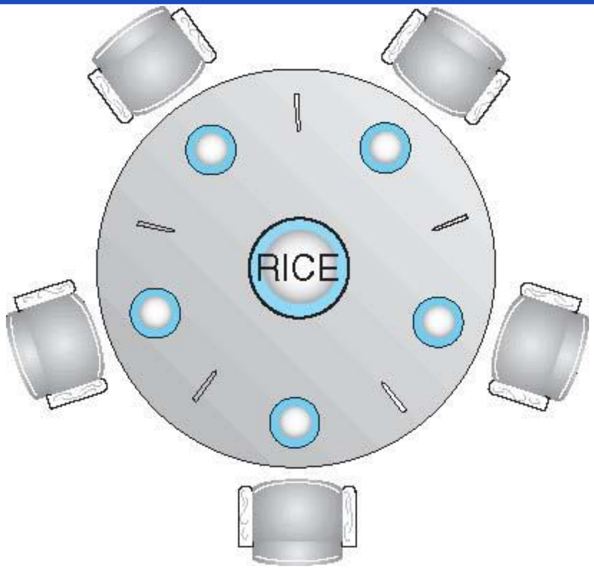
Readers-Writers Problem (Cont.)

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);
```

First Reader-writers solution

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?**

Higher Level Synchronization

- **Timing errors are still possible with semaphores**
  - **Example 1**

    signal (*mutex*);

    ...

    critical region

    ...

    wait (*mutex*);
  - **Example 2**

    wait(*mutex*);

    ...

    critical region

    ...

    wait (*mutex*);
  - **Example 3**

    wait(*mutex*);

    ...

    critical region

    ...

    Forgot to signal
- **Deadlock and starvation**

Motivation for Other Sync. Constructs

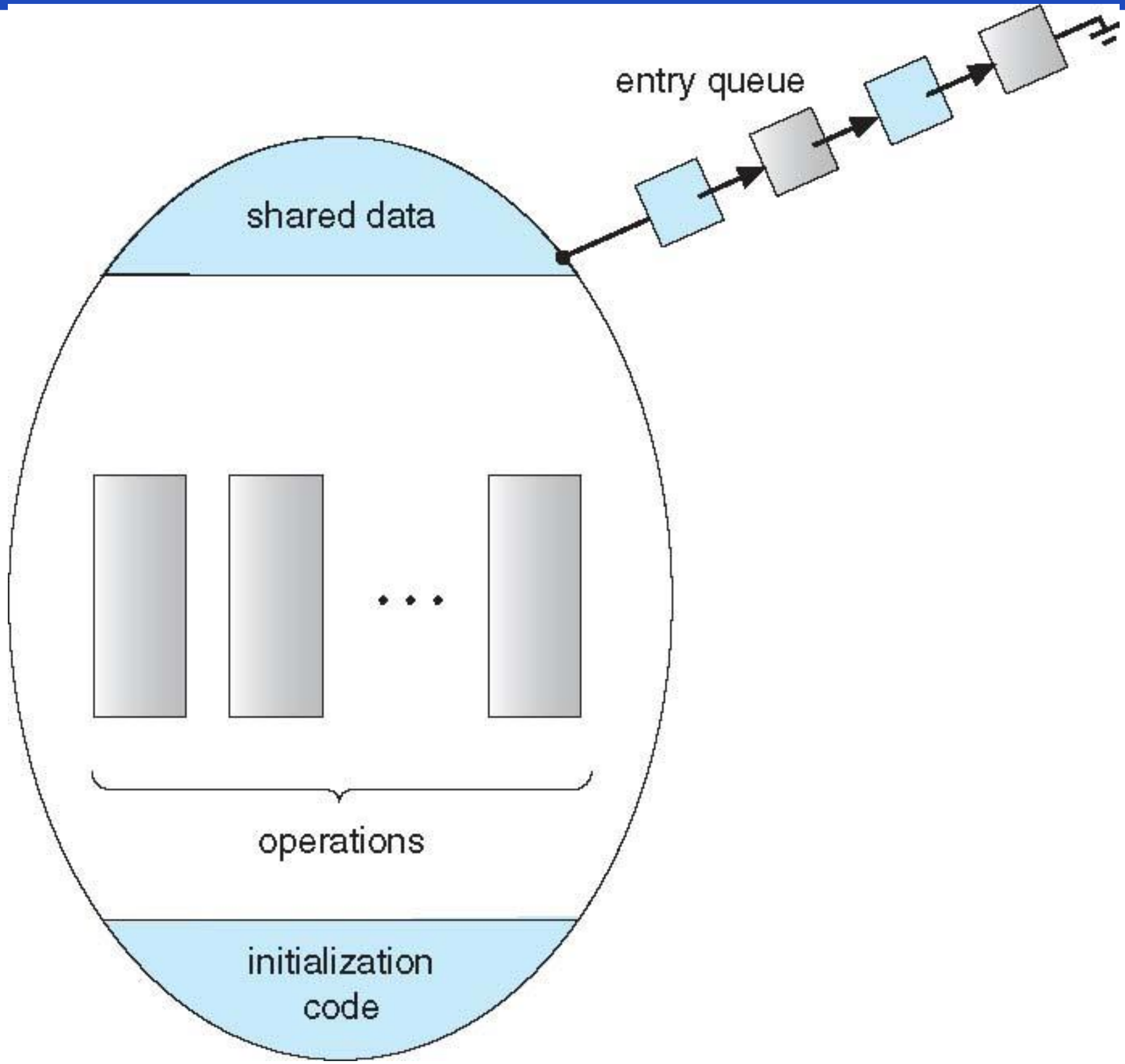
- **Semaphores are a huge step up from loads and stores**
  - **Problem is that semaphores are dual purpose:**
    - They are used for both **mutex** and **scheduling constraints**
- **Idea: allow manipulation of a shared variable only when condition (if any) is met – *conditional critical region***
- **Idea : Use *locks* for mutual exclusion and *condition variables* for scheduling constraints**
  - ***Monitor:* a lock (for mutual exclusion) and zero or more condition variables (for scheduling constraints) to manage concurrent access to shared data**
    - Some languages like Java provide this natively

Monitors

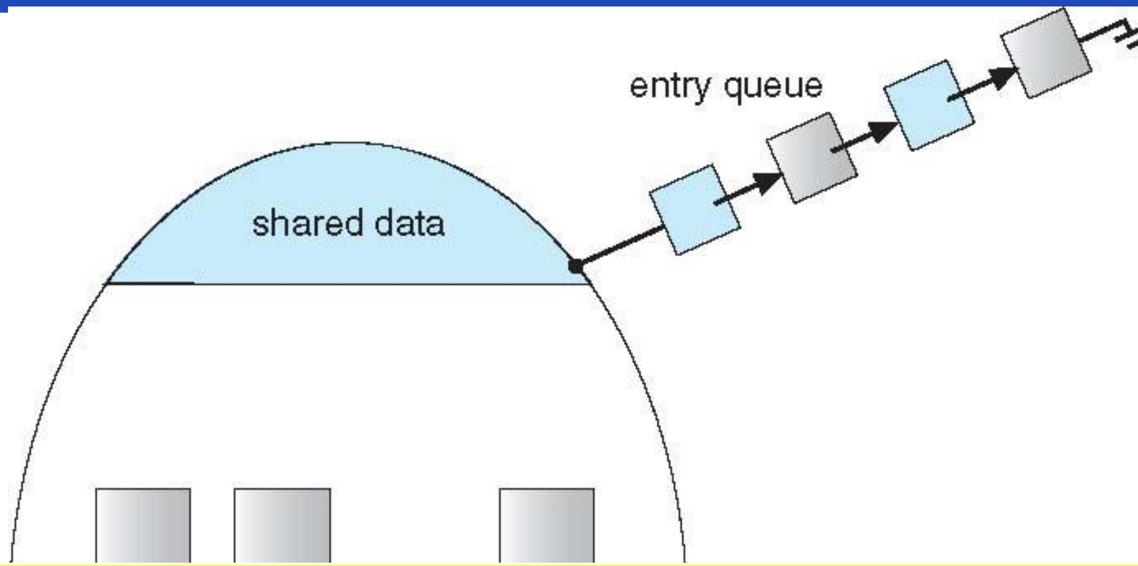
- **A high-level abstraction that provides a convenient and effective mechanism for process synchronization**
- ***Abstract data type*, internal variables only accessible by code within the procedure**
- **Only one process may be active within the monitor at a time**

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

        procedure Pn (…) {……}

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

Schematic view of a Monitor



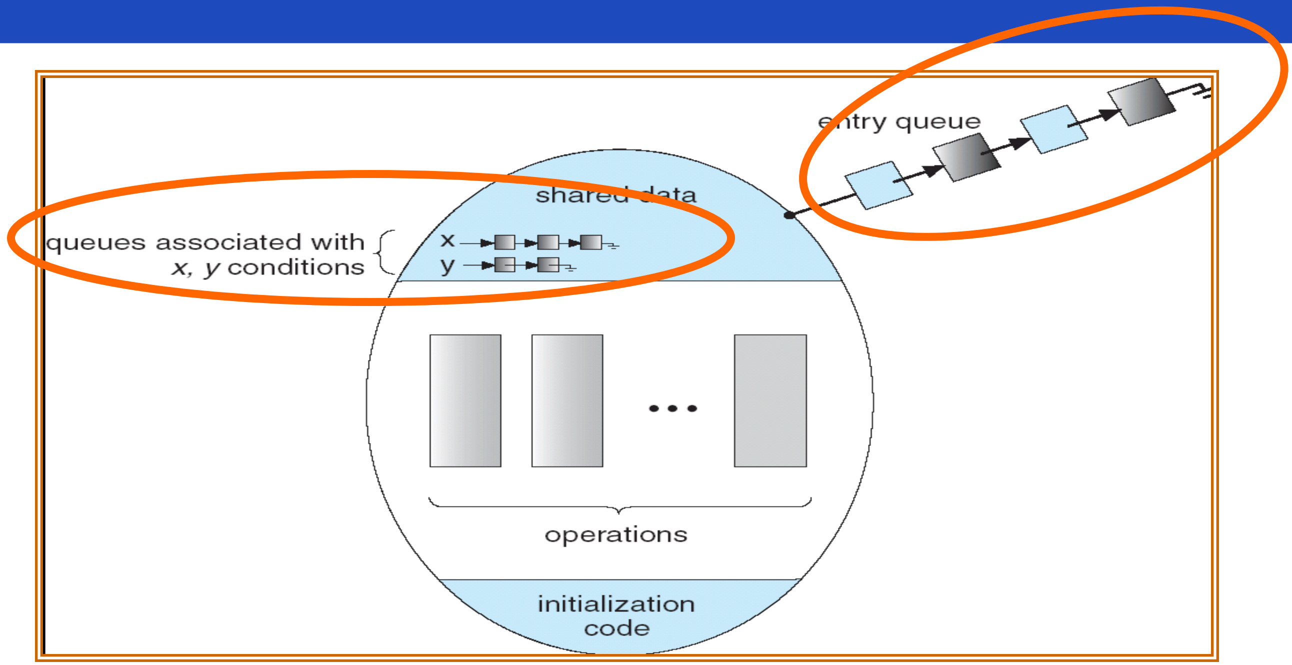
Schematic view of a Monitor



- ❑ **Monitor constructs ensure that only one process is active at a time within the monitor ADT**

- ❑ **The programmer does not need to code this synchronization constraints explicitly**

Monitor with Condition Variables



- **Lock:** **the lock provides mutual exclusion to shared data**
  - **Always acquire before accessing shared data structure**
  - **Always release after finishing with shared data**
  - **Lock initially free**
- **Condition Variable:** **a queue of threads waiting for something *inside* a critical section**
  - **Key idea:** **make it possible to go to sleep inside critical section by atomically releasing lock at time we go to sleep**

Monitors with condition variables

- **To allow a process to wait within the monitor, a condition variable must be declared, as:**

  var *x,y*: *condition*

  - Condition variable can only be used within the operations *wait* and *signal*. Queue is associated with condition variable.
    - 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. If no process is suspended, then the signal operation has no effect.

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 → Try yourself**
- ***http://vip.cs.utsa.edu/simulators/run/sp.html***

References

Part of the contents of this lecture has been adapted from the book Abraham Silberschatz, Peter B. Galvin, Greg Gagne: "Operating System Concept ", Publisher : Wiley; 9 edition (December 17, 2012), ISBN-13: 978-1118063330

Slides also contain lecture materials from John Kubiatowicz (Berkeley), John Ousterhout (Stanford), Nalini (UCI), Rainer (UCI), and others

Some slides adapted from http://www-inst.eecs.berkeley.edu/~cs162/ Copyright © 2010 UCB

Thank you for your attention