Operating Systems: Synchronization Tools and examples – Part II

Neerja Mhaskar

Department of Computing and Software, McMaster University, Canada

Acknowledgements: Material based on the textbook Operating Systems Concepts (Chapters 6 & 7)

Semaphore

- Synchronization tool that provides more sophisticated ways (other than Mutex locks) for processes to synchronize their activities.
- Semaphore ***S*** – integer variable (usually initialized with a +ve integer)
- Can only be accessed via two indivisible (**atomic**) operations
  - `wait()` and `signal()`
    - Originally called `P()` and `V()`

- **wait() operation:**

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

- **signal() operation:**

```
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
  - Behaves like a mutex lock
- Semaphores can solve various synchronization problems
  - Used as locks (binary semaphores)
  - To solve synchronization problems (may or may not access shared data)
  - Used to control access to a given resource consisting of a finite number of instances (counting semaphore)

Semphores – synchronization problems

- Consider two concurrently running processes $\boldsymbol{P_1}$ and $\boldsymbol{P_2}$ that require $\boldsymbol{S_1}$ to happen before $\boldsymbol{S_2}$ (here P1 and P2 may or may not access/modify shared variables.)
- **Solution:** Create a semaphore "**`synch`**" initialized to 0

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

Semaphore - Control access to resources

- Counting semaphores can be used to control access to a given resource consisting of a finite number of instances.
- The semaphore is initialized to the number of resources available.
- Each process that wishes to use a resource performs a **`wait()`** operation on the semaphore (thereby decrementing the count).
- When a process releases a resource, it performs a **`signal()`** operation (incrementing the count).
- When the count for the semaphore goes to 0, all resources are being used. After that, processes that wish to use a resource will block until the count becomes greater than 0.

Semaphores – No busy waiting

- Semaphores described so far suffer from *busy waiting*.
- **Solution:**
  - The process waiting on a semaphore is blocked, and removed from execution on the CPU
  - Later awakened when the semaphore is available.
  - As a result, each semaphore needs to maintain a list of processes that are blocked and waiting for it.
  - When the semaphore becomes available, one of the processes can be woken up and scheduled to execute on the CPU.

Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated waiting queue
- Each semaphore has two data items:
  - value (of type integer)
  - pointer to the list of processes waiting
- **`wait()`** and **`signal()`** operations are atomic.
- Two additional operations: **`block()`** and **`wakeup()`**
- **Semaphore Structure**

```
typedef struct{

int value;

struct process *list;

} semaphore;
```

Implementation with no Busy waiting (Cont.)

Wait Operation

```
wait(semaphore *S) {
   S->value--;
   if (S->value < 0) {
      add this process to S->list;
      block();
   }
}
```

block – places the process invoking the operation on the appropriate waiting queue

Signal Operation

```
signal(semaphore *S) {
   S->value++;
   if (S->value <= 0) {
      remove a process P from S->list;
      wakeup(P);
   }
}
```

wakeup – removes a processes in the waiting queue and places it in the ready queue

Semaphore Implementation with no Busy waiting

- Note that in this implementation, semaphore values may be negative
- If a semaphore value is negative, its magnitude is the number of processes waiting on that semaphore.
- Implementation of semaphores using waiting queues may result in **Deadlock and/or starvation**.

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

Semaphores in C

- Semaphores are not part of Pthread standards and instead they belong to the POSIX SEM extension.
- To use semaphores in C on a Linux machine include ‘semaphore.h’ header file:
    - `#include <semaphore.h>`
  - `sem_t` data type for semaphores.
  - semaphores created with `sem_init()` function.
    - `Sem_init()` - Takes 3 arguments:
      1. Pointer to the semaphore
      2. Flag indicating the level of sharing
      3. The semaphore’s initial value
  - `sem_wait();`
  - `sem_post();`

Semaphores in C

```
/*Declaring Semaphore*/

Sem_t sem;

/*Initialize Semaphore*/

if (sem_init(&sem, 0, n) !=0){

     printf("Error in initializing empty semaphore \n"

}

/* acquire the semaphore */

sem_wait(&sem);

/* critical section */

/* release the semaphore */

Sem_post(&sem);
```

sem_init arguments
1. pointer to the semaphore
2. flag indicating the level of sharing
3. The semaphore's initial value

Returns 0 when semaphore created with no errors, otherwise returns non zero value.

Classical Problems of Synchronization

- Bounded-Buffer Problem (Producer - Consumer Problem) (discussed in part I)
- Readers and Writers Problem
- Dining-Philosophers Problem

Bounded-Buffer Problem

- ***n*** buffers, each can hold one item
- Semaphore **`mutex`** initialized to the value 1
  - `mutex = 1 =>` indicates that `mutex` lock is available
  - `mutex = 0 =>` indicates that `mutex` lock is unavailable
- Semaphore **`full`** initialized to the value 0, as initially the number of filled slots in the buffers is zero.
- Semaphore **`empty`** initialized to the value n, as initially the number of empty slots in the buffer is 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);
```

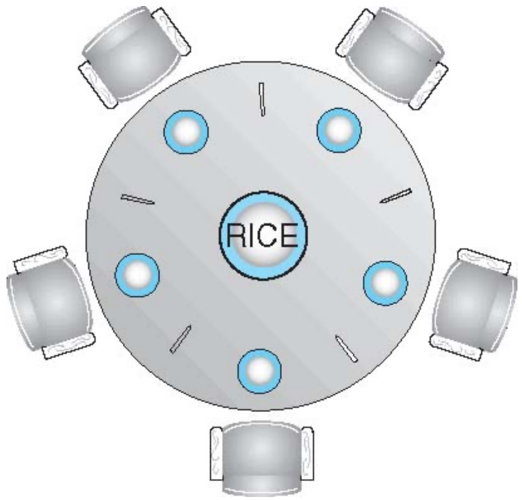
Bounded Buffer Problem (Cont.)

- 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);
```

Dining-Philosophers Problem

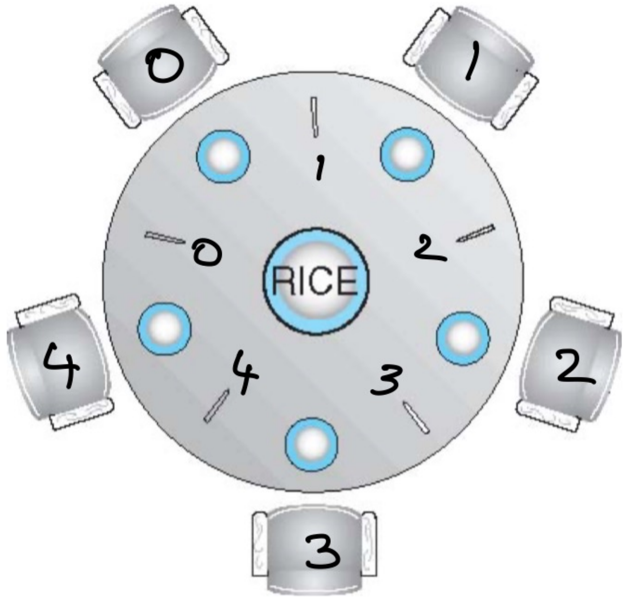
- Philosophers spend their lives alternating between **thinking** and **eating**
- Don’t interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from a bowl
  - Need both chopsticks to eat, then release both when done
- In the case of 5 philosophers
  - Shared data
    - Bowl of rice (data set) – assumed infinite
    - Semaphore chopstick [5] initialized to 1 (indicates its available)



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) %

                // think
} while (TRUE);
```



- What is the problem with this algorithm?

Dining-Philosophers Problem Algorithm

- Consider the following scenario in which **all** the philosophers are hungry and try to eat at the same time.
- Each philosopher grabs the chopstick to their ~~left~~ right (`chopstick[i]`).
- Each semaphore `chopstick[i]=0,` where `1<=i<=5`.
- When each philosopher tries to grab her ~~right~~ left chopstick (`chopstick[i+1]% 5`), she will be delayed forever. Thus resulting in a ***deadlock***.

Dining-Philosophers Problem Algorithm (Cont.)

- Deadlock can be handled by
  - Allowing at most 4 philosophers to be sitting simultaneously at the table.
  - Allow a philosopher to pick up the forks only if both are available (picking must be done in a critical section).
  - Use an asymmetric solution -- an odd-numbered philosopher picks up first the left chopstick and then the right chopstick. Even-numbered philosopher picks up first the right chopstick and then the left chopstick.

Problems with Semaphores

- Incorrect use of semaphore operations could result in mutual exclusion being violated, deadlock and/or starvation.
- Consider the following situation:
  - Processes share a Semaphore **`mutex,`** which is initialized to 1.
- Suppose that a process interchanges the order in which the `wait()` and `signal()` operations on the semaphore `mutex` are executed. That is, it executes

```
signal(mutex);
...
critical section
...
wait(mutex);
```

- **What is the problem with the above code?**

Problems with Semaphores

- Suppose that a process replaces `signal(mutex)` with `wait(mutex)`.

  That is, it executes

```
wait(mutex);

...

critical section

...

wait(mutex);
```

What is the problem here?

- Omitting of `wait(mutex)` or `signal(mutex)` (or both)
  - Deadlock and starvation are possible.

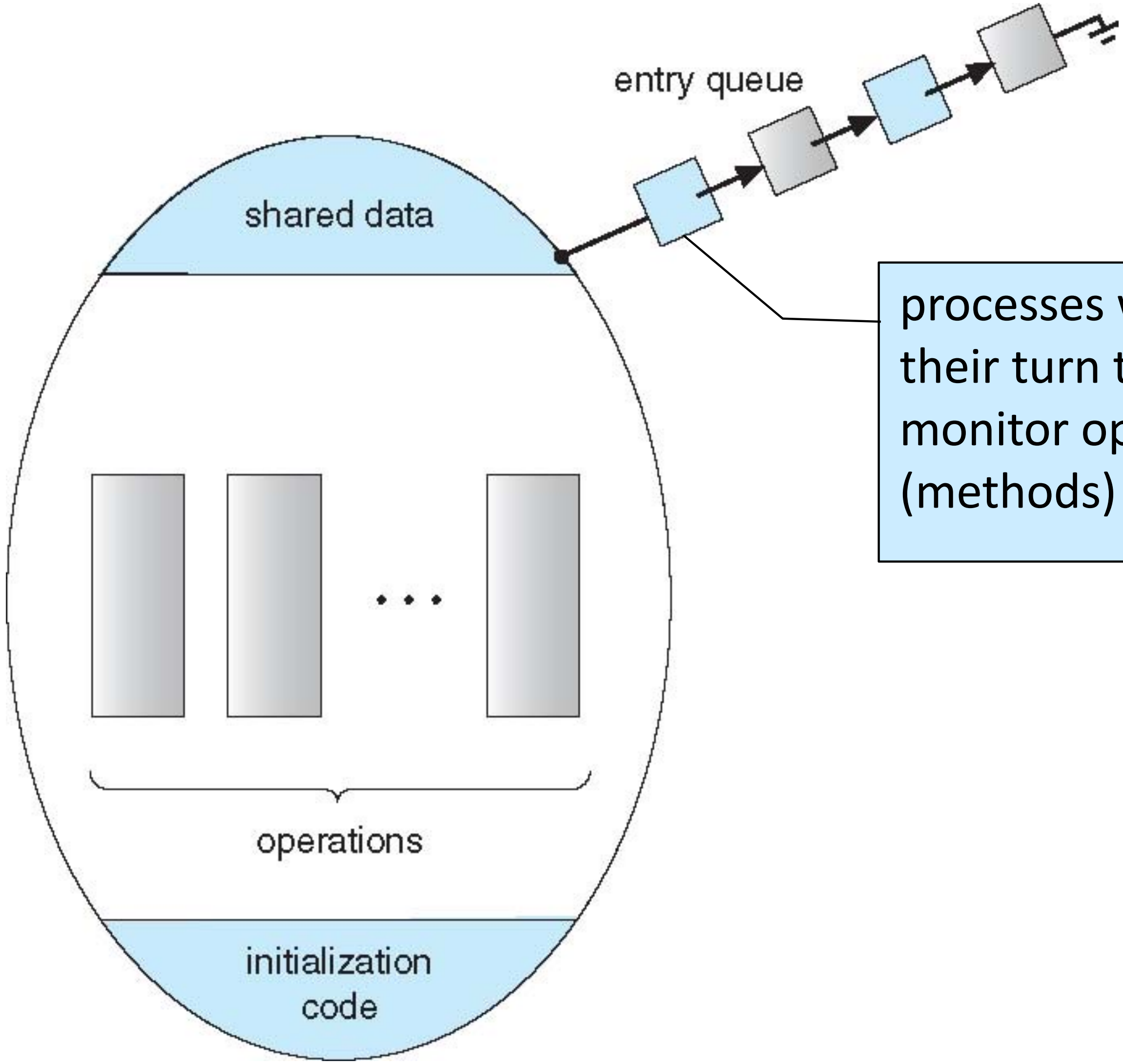
Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization.
- A monitor ADT has
  - Private data members
  - Public functions that are implicitly executed with mutual exclusion.
    - For each monitor instance there exists a mutual exclusion lock.
    - To enter the monitor, a process acquires the mutual exclusion lock
    - While exiting the monitor, a process releases the lock, and therefore the monitor, for other threads.
  - In addition, monitors may define wait conditions (condition variables) that can be used inside the monitor to synchronize the member functions.

Syntax of Monitors

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

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

Schematic view of a Monitor



processes waiting their turn to execute monitor operations (methods)

Monitor Example

```
monitor BankAccount {

      void BankAccount_init(){
            amount = 0;
      }
      void withdraw(int value){
            amount = amount - value;
      }
      void deposit(int value){
            amount = amount + value;
      }
      private: int amount;
};
```

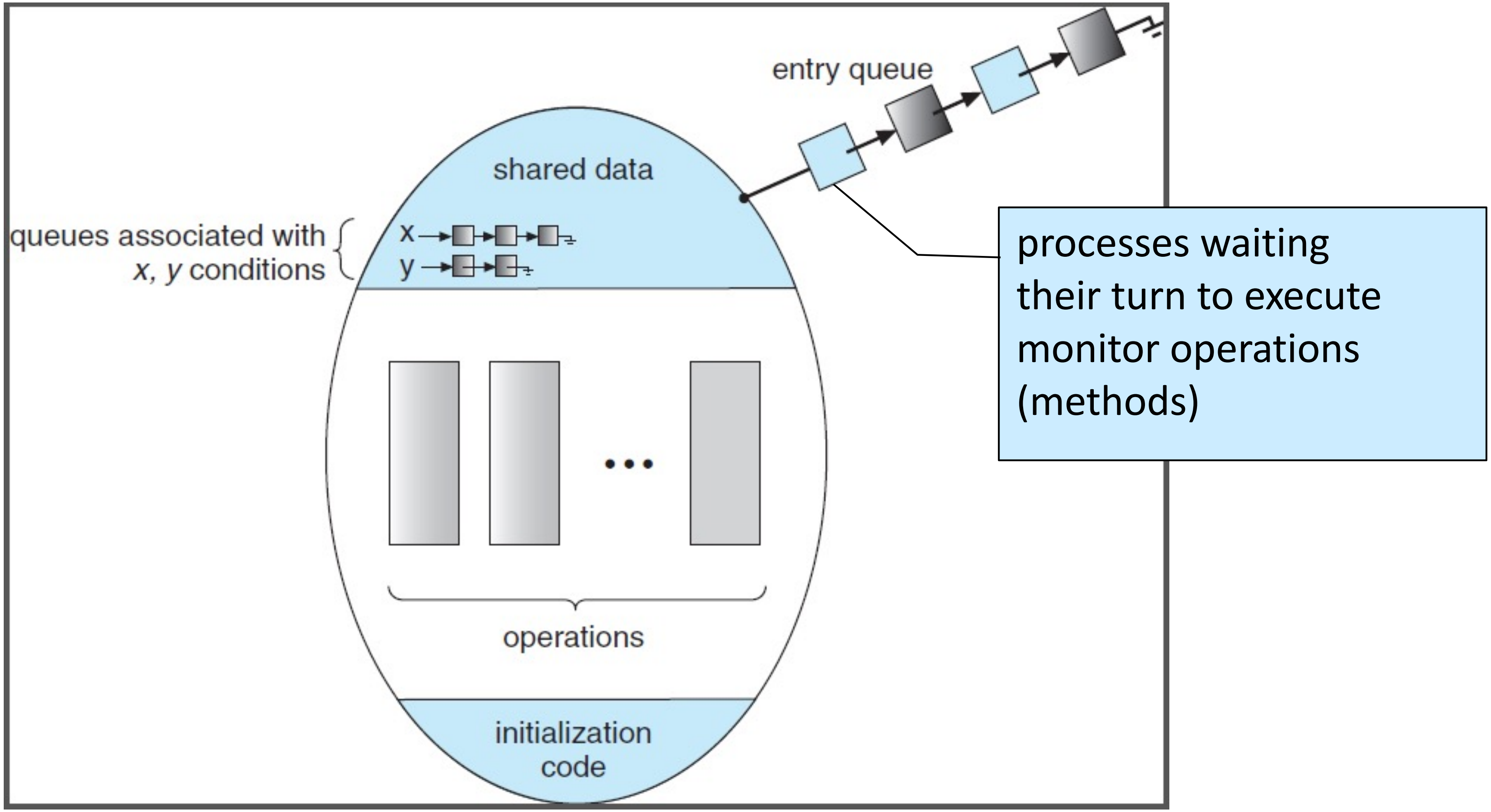
Monitor Example cont…

- How to ensure the following two conditions are met:
  - Deposits do not take place if amount >= 500
  - Withdrawals do not take place if amount <=0

Condition Variables

- Condition variables defined as: **`condition x, y;`**
- Condition variables have two legal atomic operations: **`wait()`** **`and signal()`**
- If x is a condition variable then the two operations can be accessed as: **`x.wait(), x.signal()`**
- **`wait()`** – blocks a process (until some other process calls signal on it), and adds it to a queue associated with that condition variable.
- **`signal()`** – wakes up exactly one process from the condition variable's queue of waiting processes. If no processes waiting then does nothing.

Monitor with Condition Variables



Monitor Example with condition variables

```
monitor BankAccount {

        void BankAccount_init(){
                amount = 0;
        }
        void withdraw(int value){
                if(amount <=0){c1.wait();}
                amount = amount - value;
                c2.signal();
        }
        void deposit(int value){
                if(amount >=500){c2.wait();}
                amount = amount + value;
                c1.signal();
        }
        int amount;
        condition c1, c2;
};
```

In this example we assume that the value deposited/withdrawn is the same, similar to the problem discussed in Lab 6

Condition Variables Choices

- If process P invokes **`x.signal()`**, and process 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

Resuming Processes within a Monitor

- If several processes queued on condition x, and **`x.signal()`** executed, which should be resumed?
  - FCFS (first come first serve) 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

Monitor 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