Computer and Operating Systems (COS)

Lecture 9

Overview of the contents

- The critical-section problem
- Semaphores
- Monitors
- Deadlocks
- Synchronization problems

Process synchronization

We've already seen that processes can execute concurrently or in parallel.

CPU scheduler switches rapidly between processes to provide concurrent execution. This means that one process may only partially complete execution before another process is scheduled.

In parallel execution, multiple instruction streams (representing different processes) execute simultaneously on separate processing cores.

Concurrent or parallel execution can cause issues involving the integrity of data shared by several processes.

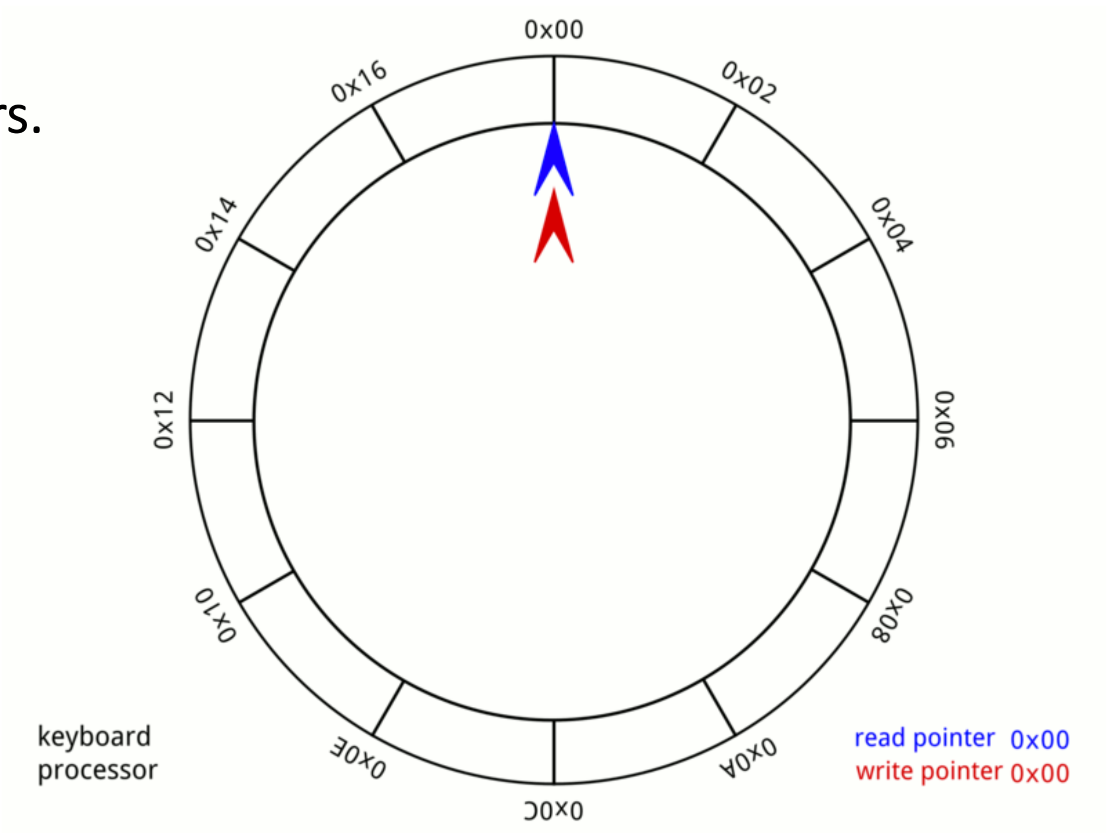
An example: Bounded buffer

A shared bounded buffer is implemented as a circular array with two logical pointers.

Write pointer in (red) points to the next free position in the buffer.

Read pointer out (blue) points to next full position in the buffer.

When in == out the buffer is empty, and the buffer is full when (in+1) == out.



An example: Bounded buffer

Producer: a process produces information

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

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

An example: Bounded buffer

Consumer: a process consumes information

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

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

Here the issue we have is called **Race Condition**

`counter++` will be performed as follows:

```
register1 = counter
register1 = register1 + 1
counter = register1
```

`Counter--` will be performed as follows:

```
register2 = counter
register2 = register2 - 1
counter = register2
```

Counter = 5 as a starting point

T_0: producer execute	`register1 = counter`	{register1 = 5}
T_1: producer execute	`register1 = register1 + 1`	{register1 = 6}
T_2: consumer execute	`register2 = counter`	{register2 = 5}
T_3: consumer execute	`register2 = register2 – 1`	{register2 = 4}
T_4: producer execute	`counter = register1`	{counter = 6}
T_5: consumer execute	`counter = register2`	{counter = 4}

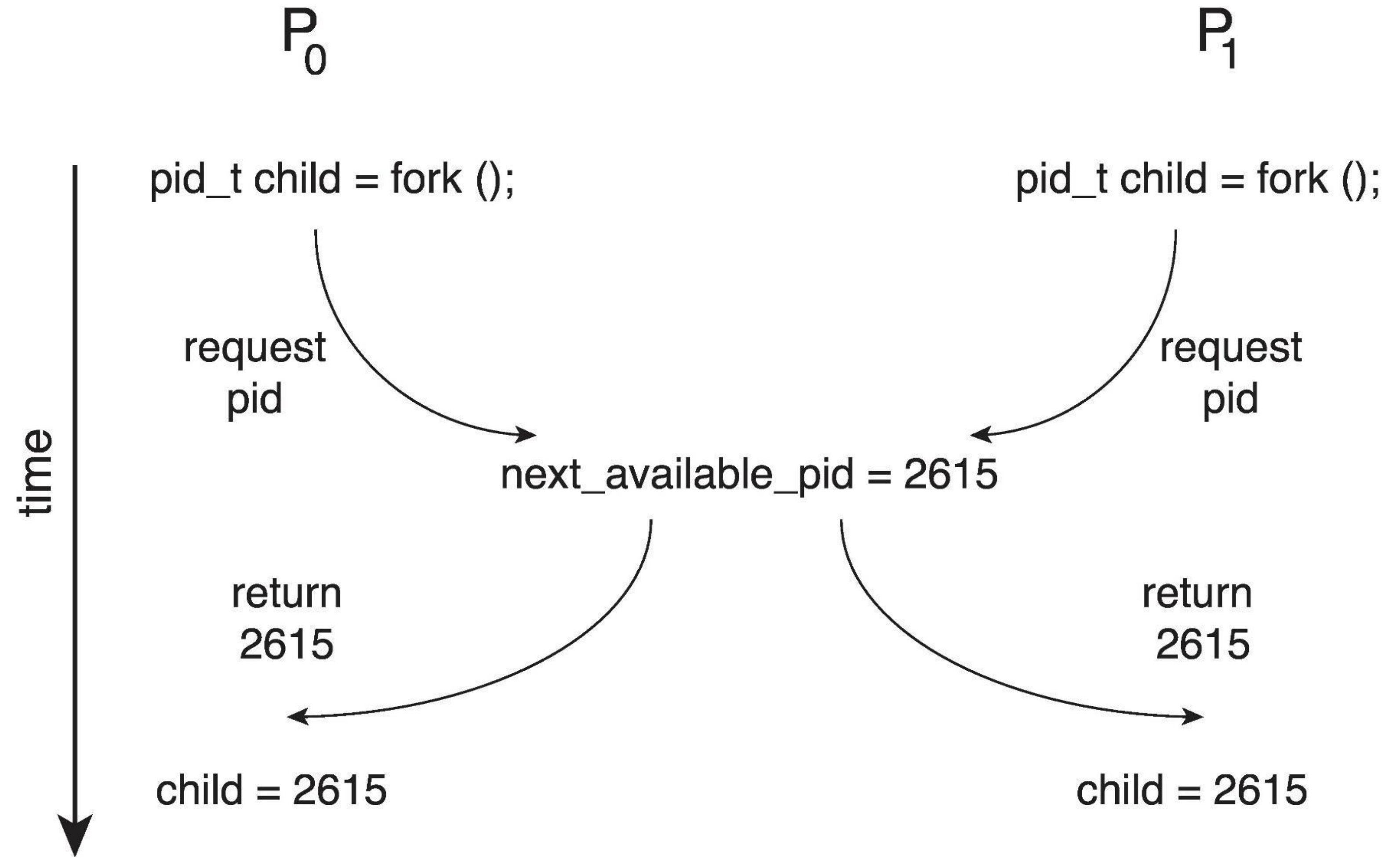
Then we can see that the counter has the value 4 after this process, which is wrong!

If T4 and T5 would have been performed in the reverse order, the value would have been 6, which is also wrong!

the correct result is of course 5.

We have a **race condition** because we allow the processes to manipulate the same variable in an environment that is **preemptive.**

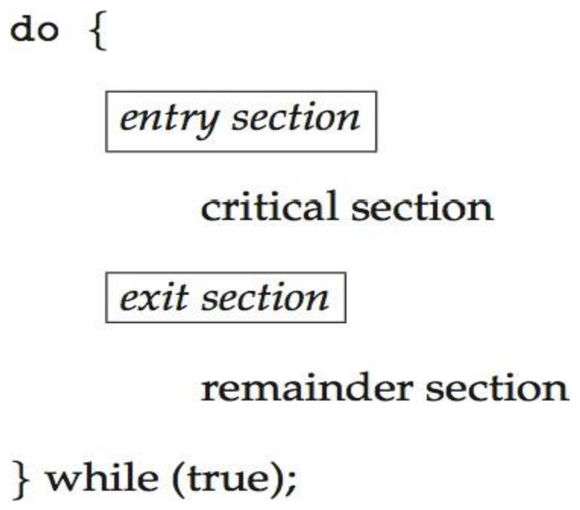
Another example of **Race Condition**



Here 2 processes start their child processes. If there is no synchronization, both processes may risk getting the same child process ID.

The solution here is **Process Synchronization**

<u>Critical-section problem:</u> Consider a system consisting of n processes $\{P_0, P_1, ..., P_{n-1}\}$. Each process has a segment of code, called a **<u>critical section</u>**, in which the process may be accessing and updating data that is shared with at least one other process. The **critical-section problem** is to design a protocol that the processes can use to synchronize their activities so as to cooperatively share data.



```
do {

    entry section

        critical section

    exit section

        remainder section

} while (true);
```

A solution to the critical-section problem must satisfy the following three requirements:

1. **Mutual Exclusion:** If a process Pi is in its critical section, then no other processes can be in their critical sections that manipulate the same variable as Pi.
2. **Progress:** If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely.
3. **Bounded Waiting:** There exists a bound, or limit, on the waiting time 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.
   <u>*Remember we are talking about critical sections that manipulate the same variables.*</u>

Semaphores

A **<u>Semaphore</u>** S is an integer variable that, apart from initialization, is accessed only through two standard **atomic** operations: wait() and signal().

Wait() is used when a process wants access to its critical section and is also called **spinlock** or **busy waiting.**

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

Signal() is used when a process wants to leave its critical section

```
signal (S) {
   S++;
}
```

Semaphores

An Example of semaphore usage

consider two concurrently running processes: P1 with a statement S1 and P2 with a statement S2. Suppose we require that S2 be executed only after S1 has completed. We can implement this scheme by letting P1 and P2 share a common **semaphore synch**, initialized to 0. In process P1, we insert the statements

```
S1;
signal(synch);
```

In process P2, we insert the statements

```
wait(synch);
S2;
```

Semaphore implementation

It is not always an advantage to use **busy waiting**, as the process spends its CPU time just waiting. Then, a semaphore queue can be used instead.

The semaphore is defined as a struct with a variable and a queue "list".

```
typedef struct{
  int value = 1;
  struct process *list;
} semaphore;
```

wait () method adds a process to a queue list if the semaphore value is negative

```
wait(semaphore *S) {
  S->value--;
  if (S->value < 0) {
    add this process to S->list; Only when S becomes negative processes are queued
    sleep();
  }
}
```

signal () method remove another process from the semaphore queue

```
signal(semaphore *S) {
  S->value++;
  if (S->value <= 0) {
    remove a process P from S->list; processes are queued as long as S is less than 0
    wakeup(P);
  }
}
```

The problem of semaphores

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

or…

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

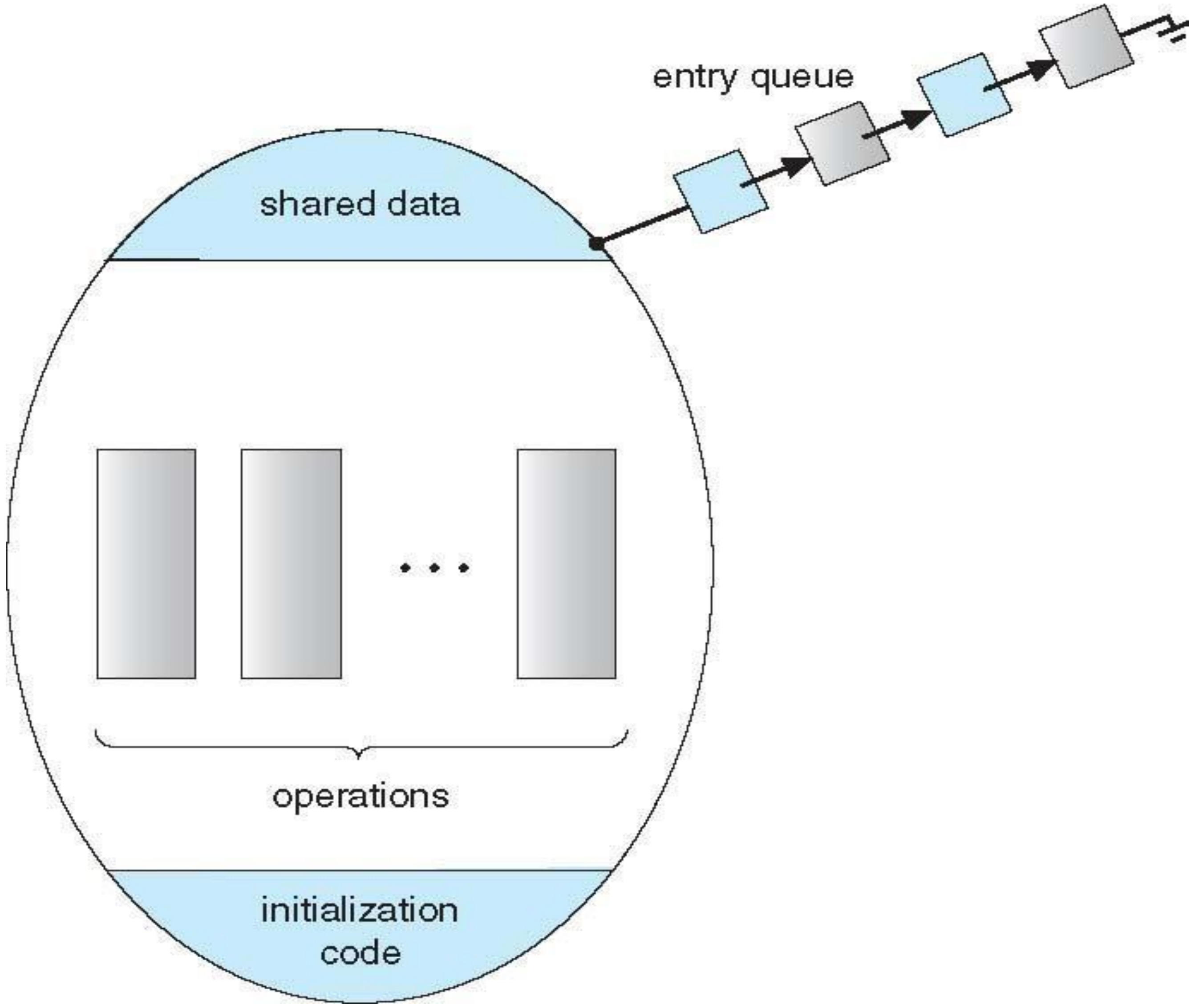
Monitors

```
monitor monitor_name
{
        /* shared variable declarations */

        function P1 (…) {
        …
        }
        .
        .
        .
        function Pn (…) {
        …
        }
        Initialization_code (…) {
        …
        }
}
```

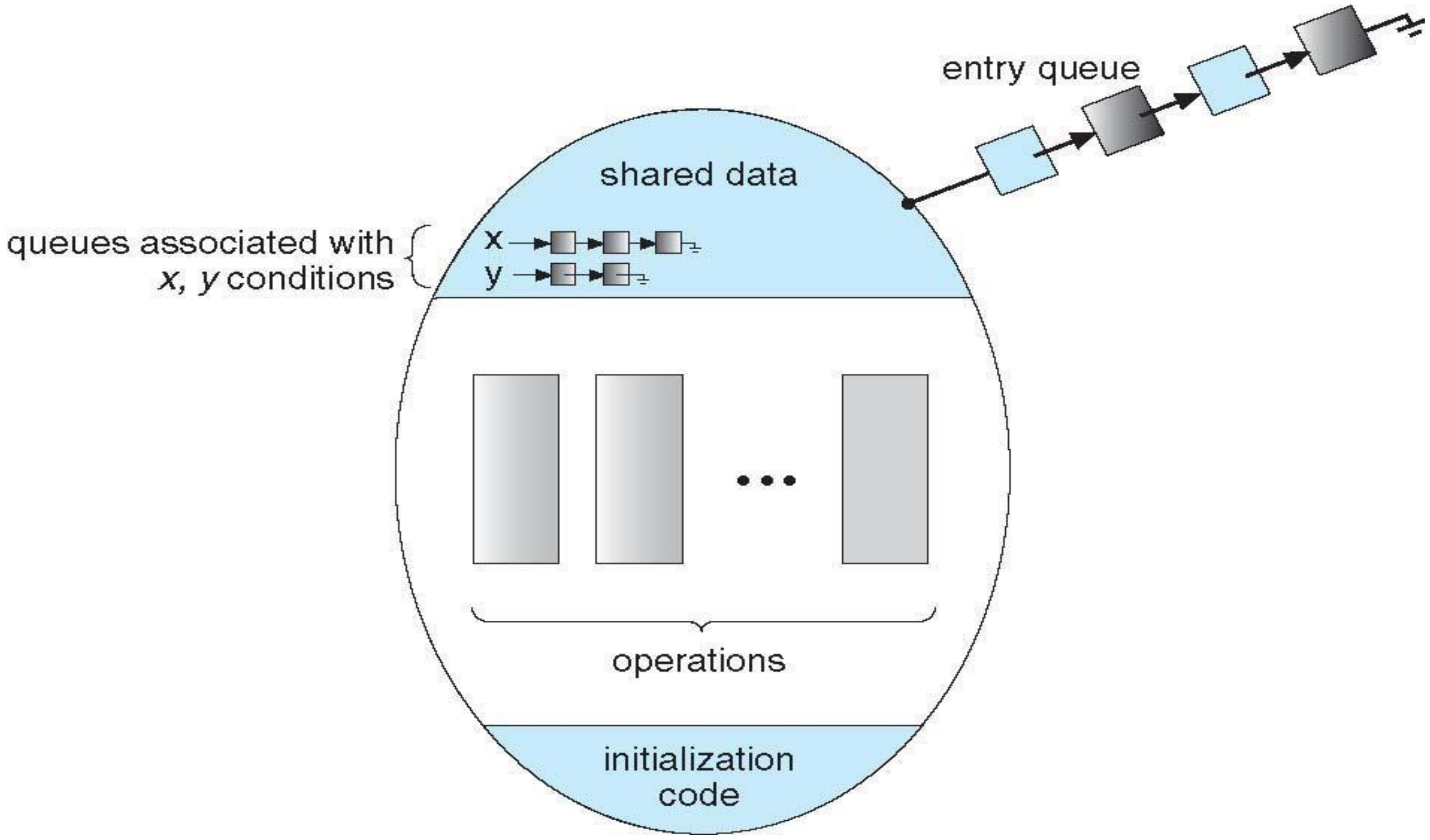
Monitors

Schematic structure



Monitors

Condition variable



x.wait() – a process calling this method is suspended until the condition is met by x.signal ()
x.signal() – activates a process (if any) that was suspended with x.wait ()

- if there are no processes waiting for this condition, then signal () has no effect

Deadlocks

The implementation of a semaphore with a waiting queue may result in a situation where two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes. The event in question is the execution of a signal() operation. When such a state is reached, these processes are said to be **deadlocked**.

```
    P0                P1
 wait(S);          wait(Q);
 wait(Q);          wait(S);
    .                 .
    .                 .
    .                 .
 signal(S);        signal(Q);
 signal(Q);        signal(S);
```

Synchronization examples

Several synchronization problems as examples of a large class of concurrency-control problems are listed here. These problems are used for testing nearly every newly proposed synchronization scheme.

- Bounded-Buffer Problem
- Readers-Writers Problem
- Dining-Philosophers Problem

Bounded-Buffer Problem

The shared data structures

```
int n;
semaphore mutex = 1;
semaphore empty = n;
semaphore full = 0;
```

We assume that the pool consists of n buffers, each capable of holding one item.

The mutex binary semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1.

The empty and full semaphores count the number of empty and full buffers. The semaphore empty is initialized to the value n; the semaphore full is initialized to the value 0.

Producer process

```
while (true) {
    ...
  /* produce an item in next_produced */
    ...
  wait(empty);
  wait(mutex);
    ...
  /* add next_produced to the buffer */
    ...
  signal(mutex);
  signal(full);
}
```

Consumer process

```
while (true){
  wait(full);
  wait(mutex);
    ...
  /* remove an item from buffer to next_consumed */
    ...
  signal(mutex);
  signal(empty);
    ...
  /* consume the item in next_consumed */
    ...
}
```

Readers-Writers Problem

The shared data structures:

semaphore rw_mutex = 1;
semaphore mutex = 1;
int read_count = 0;

Assume a database is to be shared among several concurrent processes. Some of these processes (**readers**) may want only to read the database, whereas others (**writers**) may want to update (that is, read and write) the database.

The binary semaphore **rw_mutex** is common to both reader and writer processes and is initialized to 1.
The **read_count** is an integer and initialized to 0. The read_count variable keeps track of how many processes are currently reading the object.
The binary semaphores **mutex** is used to ensure mutual exclusion when the variable read_count is updated and is initialized to 0.

The semaphore **rw_mutex** functions as a mutual exclusion semaphore for the writers. It is also used by the first or last reader that enters or exits the critical section. It is not used by readers that enter or exit while other readers are in their critical sections.

Writer process

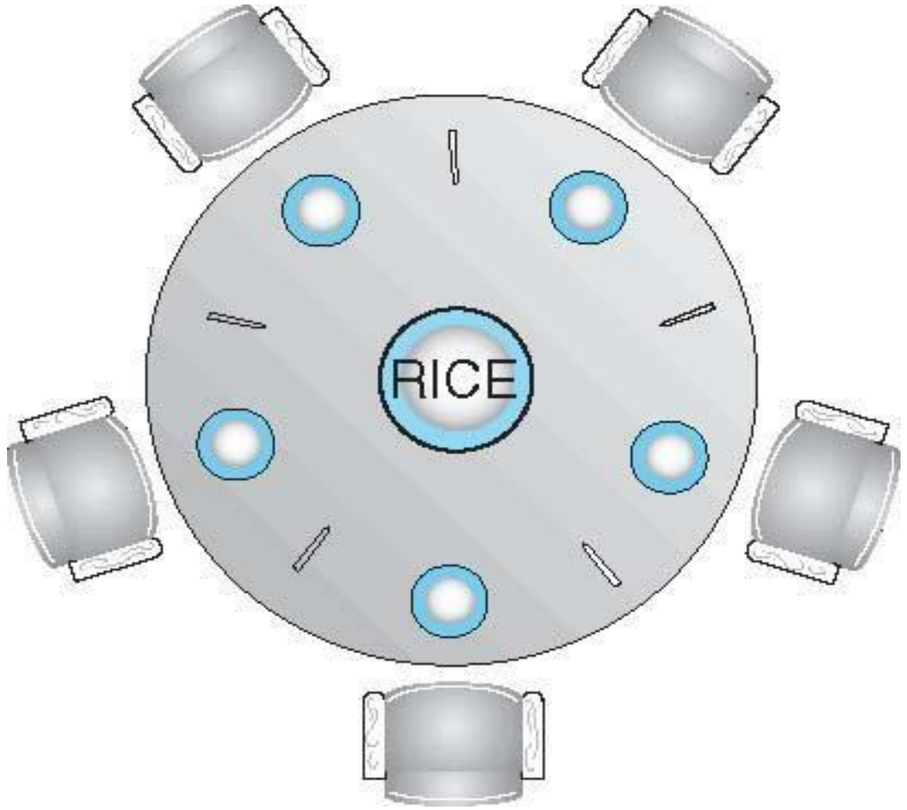
```
while (true) {
   wait(rw_mutex);
      ...
   /* writing is performed */
      ...
   signal(rw_mutex);
}
```

Reader process

```
while (true){
    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);
}
```

This variant gives readers precedence: as long as only one reader is in the system a writer cannot access

Dining-Philosophers Problem



Dining-Philosophers Problem

Semaphore Solution

The shared data structures:

semaphore chopstick [5] = 1;

One simple solution is to represent each chopstick with a semaphore. A philosopher tries to grab a chopstick by executing a wait() operation on that semaphore. She releases her chopsticks by executing the signal() operation on the appropriate semaphores.

All the elements of chopstick are initialized to 1.

Dining-Philosophers Problem
Semaphore Solution

Philosopher *i*

```
while (TRUE) {
        wait ( chopstick[i] );
        wait ( chopStick[ (i + 1) % 5] );
                        …
              /* eat for a while */
                        …
        signal ( chopstick[i] );
        signal (chopstick[ (i + 1) % 5] );
                        …
              /* think for a while */
                        …
}
```

Do we have a problem here?

Dining-Philosophers Problem

Monitor Solution

```
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);
      }
```

Dining-Philosophers Problem

Monitor Solution

```
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;
    }
}
```

Dining-Philosophers Problem
Monitor Solution

Each philosopher accesses the monitor and invoke the operations pickup() and putdown() in the following sequence:

```
DiningPhilosophers.pickup (i);
…
EAT
…
DiningPhilosophers.putdown (i);
```

This solution is deadlock-free, but starvation can occur.

Final notes

Monitors are good in the sense that they can prevent the programmer from making the typical copy-paste error.

But a monitor cannot do more than what you can do with semaphore.

Section 6.7.2 “Implementing a Monitor Using Semaphores” describes how to make a monitor with semaphores, you can read it if you want to go in depth here (it is not a requirement).

Several languages have the monitor concept built into them, e.g., Java… (when a method is declared synchronized, it usually has a monitor facility)