# TECHNIQUES FOR AVOIDING RACE CONDITION - II

## *Techniques for avoiding Race Condition:*

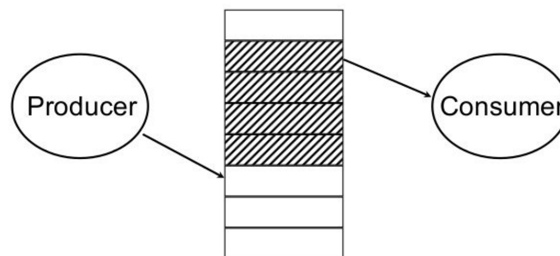
1.Unggr "cpf "Y cmgwr
2.Ugo cr j qtgu
3.O qpkvqtu
4.O guuci g"Rcuukpi

### Sleep and Wakeup:

Sleep and wakeup are system calls that blocks process instead of wasting CPU time when they are not allowed to enter their critical region. sleep is a system call that causes the caller to block, that is, be suspended until another process wakes it up. The wakeup call has one parameter, the process to be awakened.

### *Examples to use Sleep and Wakeup primitives:*

**Producer-consumer problem (Bounded Buffer):**
Two processes share a common, fixed-size buffer. One of them, the producer, puts information into the buffer, and the other one, the consumer, takes it out.




Trouble arises when

1. The producer wants to put a new data in the buffer, but buffer is already full.
   Solution: Producer goes to sleep and to be awakened when the consumer has removed data.
2. The consumer wants to remove data the buffer but buffer is already empty.
   Solution: Consumer goes to sleep until the producer puts some data in buffer and wakes consumer up.

```
#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 */
    }
}
```

Fig:The producer-consumer problem with a fatal race condition.

N → Size of Buffer

Count--> a variable to keep track of the no. of items in the buffer.

**Producers code:**

The producers code is first test to see if count is N. If it is, the producer will go to sleep ; if it is not the producer will add an item and increment count.

**Consumer code:**

It is similar as of producer. First test count to see if it is 0. If it is, go to sleep; if it nonzero remove an item and decrement the counter.

Each of the process also tests to see if the other should be awakened and if so wakes it up.

This approach sounds simple enough, but it leads to the same kinds of race conditions as we saw in the spooler directory.

1. The buffer is empty and the consumer has just read count to see if it is 0.
2. At that instant, the scheduler decides to stop running the consumer temporarily and start running the producer. (Consumer is interrupted and producer resumed)

3. The producer creates an item, puts it into the buffer, and increases count.
4. Because the buffer was empty prior to the last addition (count was just 0), the producer tries to wake up the consumer.
5. Unfortunately, the consumer is not yet logically asleep, so the **wakeup signal** is lost.
6. When the consumer next runs, it will test the value of count it previously read, find it to be 0, and go to sleep.
7. Sooner or later the producer will fill up the buffer and also go to sleep. Both will sleep forever.

The essence of the problem here is that a wakeup sent to a process that is not (yet) sleeping is lost. For temporary solution we can use wakeup waiting bit to prevent wakeup signal from getting lost, but it can't work for more processes.

### *Semaphore:*

In computer science, a semaphore is a protected variable or abstract data type that constitutes a classic method of controlling access by several processes to a common resource in a parallel programming environment . Synchronization tool that does not require busy waiting . **A semaphore is a special kind of integer variable which can be initialized and can be accessed only through two atomic operations. P and V. If S is the semaphore variable, then,**

| **P operation:** Wait for semaphore to become positive and then decrement<br>**P(S):**<br>**while(S<=0)**<br>**do no-op;**<br>**S=S-1** | **V Operation:** Increment semaphore by 1<br>**V(S):**<br>S=S+1; |
|---|---|

**Atomic operations:** When one process modifies the semaphore value, no other process can simultaneously modify that same semaphores value. In addition, in case of the P(S) operation the testing of the integer value of S (S<=0) and its possible modification (S=S-1), must also be executed without interruption.

**Semaphore operations:**
P or Down, or Wait: P stands for *proberen* for "to test
V or Up or Signal: Dutch words. **V** stands for *verhogen* ("increase")

wait(sem) -- decrement the semaphore value. if negative, suspend the process and place in queue. (Also referred to as *P(), down* in literature.)

signal(sem) -- increment the semaphore value, allow the first process in the queue to continue. (Also referred to as *V(), up* in literature.)

**Counting semaphore –** integer value can range over an unrestricted domain
**Binary semaphore** – integer value can range only between 0 and 1; can be simpler to implement Also known as mutex locks

**Provides mutual exclusion**
Semaphore S; // initialized to 1
wait (S);
Critical Section
signal (S);

**Semaphore implementation without busy waiting:**
**Implementation of wait:**

```
wait (S){
        value - -;
        if (value < 0) {
                add this process to waiting queue
                block(); }
        }
```

**Implementation of signal:**

```
Signal (S){
        value++;
        if (value <= 0) {
                remove a process P from the waiting queue
                wakeup(P); }
        }
```

```
#define N 100                       /* number of slots in the buffer */
typedef int semaphore;              /* semaphores are a special kind of int */
semaphore mutex = 1;                /* controls access to critical region */
semaphore empty = N;                /* counts empty buffer slots */
semaphore full = 0;                 /* counts full buffer slots */
void producer(void)
{
    int item;
    while (TRUE){                   /* TRUE is the constant 1 */
        item = produce_item();      /* generate something to put in buffer */
        down(&empty);               /* decrement empty count */
        down(&mutex);               /* enter critical region */
        insert_item(item);          /* put new item in buffer */
        up(&mutex);                 /* leave critical region */
        up(&full);                  /* increment count of full slots */
    }
}
```

```
void consumer(void)
{
    int item;
    while (TRUE){                    /* infinite loop */
        down(&full);                 /* decrement full count */
        down(&mutex);                /* enter critical region */
        item = remove_item();        /* take item from buffer */
        up(&mutex);                  /* leave critical region */
        up(&empty);                  /* increment count of empty slots */
        consume_item(item);          /* do something with the item */
    }
}
```

Fig: The producer-consumer problem using semaphores.

This solution uses three semaphore.

1. Full: For counting the number of slots that are full, initially 0
2. Empty: For counting the number of slots that are empty, initially equal to the no. of slots in the buffer.
3. Mutex: To make sure that the producer and consumer do not access the buffer at the same time, initially 1.

Here in this example seamphores are used in two different ways.

1.**For mutual Exclusion**: The mutex semaphore is for mutual exclusion. It is designed to guarantee that only one process at at time will be reading or writing the buffer and the associated variable.

2.**For synchronization:** The full and empty semaphores are needed to guarantee that certain certain event sequences do or do not occur. In this case, they ensure that producer stops running when the buffer is full and the consumer stops running when it is empty.

The above definition of the semaphore suffer the problem of busy wait. To overcome the need for busy waiting, we can modify the definition of the P and V operation of the semaphore. When a Process executes the P operation and finds that the semaphores value is not positive, it must wait. However, rather than busy waiting, the process can block itself. The block operation places into a waiting queue associated with the semaphore, and the state of the process is switched to the

**Advantages of semaphores:**

- Processes do not busy wait while waiting for resources. While waiting, they are in a "suspended" state, allowing the CPU to perform other chores.
- Works on (shared memory) multiprocessor systems.
- User controls synchronization.

**Disadvantages of semaphores:**

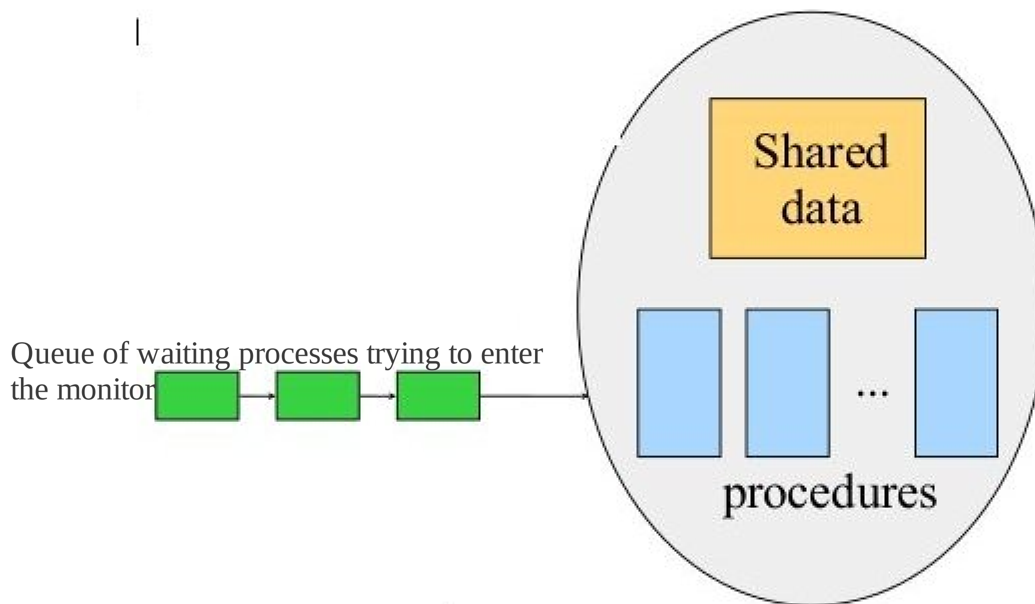
•can only be invoked by processes--not interrupt service routines because interrupt routines cannot block

•user controls synchronization--could mess up.

### ***Monitors:***

In concurrent programming, a **monitor** is an object or module intended to be used safely by more than one thread. The defining characteristic of a monitor is that its methods are executed with mutual exclusion. That is, at each point in time, at most one thread may be executing any of its methods. This mutual exclusion greatly simplifies reasoning about the implementation of monitors compared to reasoning about parallel code that updates a data structure.

Monitors also provide a mechanism for threads to temporarily give up exclusive access, in order to wait for some condition to be met, before regaining exclusive access and resuming their task. Monitors also have a mechanism for signaling other threads that such conditions have been met.




With semaphores IPC seems easy, but Suppose that the two downs in the producer's code were reversed in order, so mutex was decremented before empty instead of after it. If the buffer were completely full, the producer would block, with mutex set to 0. Consequently, the next time the consumer tried to access the buffer, it would do a down on mutex, now 0, and block too. Both processes would stay blocked forever and no more work would ever be done. This unfortunate situation is called a deadlock.

- A higher level synchronization primitive.
- A monitor is a collection of procedures, variables, and data structures that are all grouped together in a special kind of module or package.
- Processes may call the procedures in a monitor whenever they want to, but they cannot directly access the monitor's internal data structures from procedures declared outside the monitor.
- This rule, which is common in modern object-oriented languages such as Java, was relatively unusual for its time,
- Figure below illustrates a monitor written in an imaginary language, Pidgin Pascal.

```
monitor example
 integer i;
 condition c;
 procedure producer (x);
 .
 .
 .
 end;
 procedure consumer (x);
 .
 .
 .
 end;
end monitor;
```

**Fig: A monitor**

### *Message Passing:*

Message passing in computer science, is a form of communication used in parallel computing, object-oriented programming, and interprocess communication. In this model processes or objects can send and receive messages (comprising zero or more bytes, complex data structures, or even segments of code) to other processes. By waiting for messages, processes can also synchronize.

Message passing is a method of communication where messages are sent from a sender to one or more recipients. Forms of messages include **(remote) method invocation, signals, and data packets**. When designing a message passing system several choices are made:

- Whether messages are transferred reliably
- Whether messages are guaranteed to be delivered in order
- Whether messages are passed one-to-one, one-to-many (multicasting or broadcasting), or many-to-one (client–server).
- Whether communication is synchronous or asynchronous.

This method of interprocess communication uses two primitives, send and receive, which, like semaphores and unlike monitors, are system calls rather than language constructs. As such, they can easily be put into library procedures, such as

```
send(destination, &message);
```

and

```
receive(source, &message);
```

Synchronous message passing systems requires the sender and receiver to wait for each other to transfer the message

Asynchronous message passing systems deliver a message from sender to receiver, without waiting for the receiver to be ready.

Source : http://dayaramb.files.wordpress.com/2012/02/operating-system-pu.pdf