SHARED DATA ACCESS RISKS

Shared data

```
#define BUFFER_SIZE 10
typedef struct { . . . } item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
int counter = 0;
```

Producer process

```
item nextProduced;
while (1) {
            while (counter == BUFFER_SIZE); /* do nothing */
            buffer[in] = nextProduced;
            in = (in + 1) % BUFFER_SIZE;
            counter++;
           }
```

Consumer process

```
Item nextConsumed;
while (1) {
            while (counter == 0); /* do nothing */
            nextConsumed = buffer[out];
            out = (out + 1) % BUFFER_SIZE;
            counter--;
           }
```

SHARED DATA ACCESS RISKS

The statements

```
counter++;
counter--;
```

must be performed *atomically*.

Atomic operation means an operation that completes in its entirety without interruption.

The statement "**count ++**" may be implemented in machine language as:

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

The statement "**count --**" may be implemented as:

```
register2 = counter
register2 = register2 – 1
counter = register2
```

If both the producer and consumer attempt to update the buffer concurrently, the assembly language statements may get interleaved.

Interleaving depends upon how the producer and consumer processes are scheduled.

INTERLEAVING EFFECT

Assume **counter** is initially 5. One interleaving of statements is:

```
producer:   register1 = counter               (register1 = 5)
producer:   register1 = register1 + 1         (register1 = 6)
consumer:   register2 = counter               (register2 = 5)
consumer:   register2 = register2 – 1         (register2 = 4)
producer:   counter = register1               (counter = 6)
consumer:   counter = register2               (counter = 4)
```

The value of **count** may be either 4 or 6, where the correct result should be 5.

Race condition: The situation where several processes access – and manipulate shared data concurrently. The final value of the shared data depends upon which process finishes last.

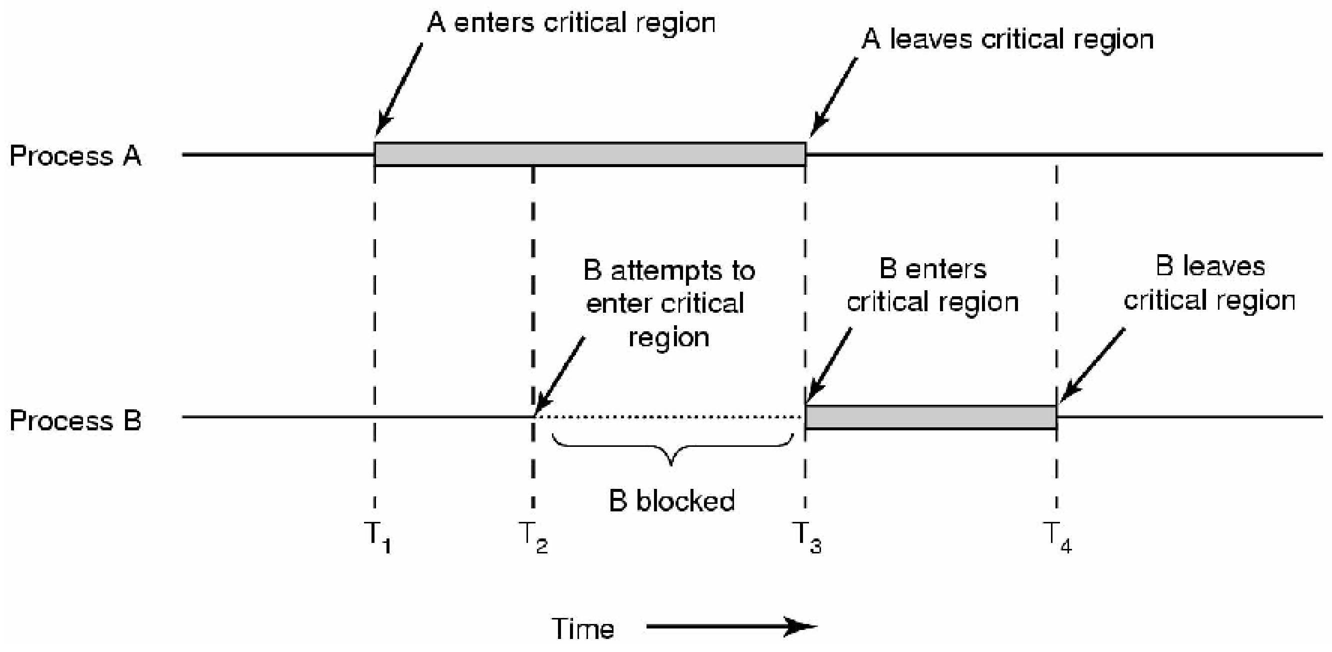
To prevent race conditions, concurrent processes must be **synchronized**.

THE CRITICAL-SECTION PROBLEM

- *n* processes all competing to use some shared data
- Each process has a code segment, called *critical section*, 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 its critical section.

SOLUTION TO CRITICAL-SECTION PROBLEM

1. **Mutual Exclusion**. If process P_i is executing in its critical section, then no other processes can be executing in their critical sections. Each process must request permission to enter its critical section.



Mutual exclusion using critical regions

SOLUTION TO CRITICAL-SECTION PROBLEM

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 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.

ATTEMPTS TO SOLVE THE CRITICAL-SECTION PROBLEM

Only 2 processes, P_0 and P_1

General structure of process P_i (other process P_j)

```
        do      {
                entry section
                        critical section
                exit section
                        reminder section
                } while (1);
```

Processes may share some common variables to synchronize their actions.

ALGORITHM 1

Shared variables:

```
    int turn;
```

initially **turn = 0**

turn = i $\Rightarrow$ P_i can enter its critical section

Process P_i

```
        do      {
                while (turn != i);    /* entry section */
                       critical section
                turn = j;             /* exit section */
                       reminder section
                } while (1);
```

Satisfies mutual exclusion, but not progress requirement, since it requires strict alternation of processes in executing the critical section.
In fact, if `turn == 0` and P_1 is ready to enter its critical section, it cannot do so, even though P_0 is in its remainder section.

ALGORITHM 2

Shared variables:

```
    boolean flag[2];
```

initially **flag [0] = flag [1] = false.**

flag [i] = true $\Rightarrow P_i$ ready to enter its critical section

Process P_i

```
        do      {
                flag[i] := true;        /* entry section */
                while (flag[j]);
                critical section
                flag [i] = false;       /* exit section */
                remainder section
                } while (1);
```

Satisfies mutual exclusion, but not progress requirement.

In fact, the interleaving of the entry sections leads to a looping forever (deadlock) state for both the processes. Switching the instructions in the entry section will only lead to mutual exclusion violation.

ALGORITHM 3

Combined shared variables of algorithms 1 and 2.

Process P_i

```
        do      {
                flag [i]:= true;                /* entry section */
                turn = j;
                while (flag [j] and turn = j);
                        critical section
                flag [i] = false;               /* exit section */
                        remainder section
                } while (1);
```

Meets all three requirements; solves the critical-section problem only for two processes.

BAKERY ALGORITHM
Critical section for n processes

Before entering its critical section, process receives a number. Holder of the smallest number enters the critical section.

If processes P_i and P_j receive the same number, if $i < j$, then P_i is served first; else P_j is served first.

The numbering scheme always generates numbers in increasing order of enumeration; i.e., 1,2,3,3,3,3,4,5...

SYNCHRONIZATION HARDWARE

Hardware features can solve the critical-section problem effectively.

☞ Test and modify the content of a word atomically.

```
boolean TestAndSet(boolean &target) {
              boolean rv = target;
              target = true;
              return rv;
                                      }
```

Shared data:

```
boolean lock = false;
```

Process P_i

```
       do     {
              while (TestAndSet(lock)) ;
                     critical section
              lock = false;
                     remainder section
              }
```

SYNCHRONIZATION HARDWARE

☞ Atomically swap two variables.

```
void Swap(boolean &a, boolean &b) {
               boolean temp = a;
               a = b;
               b = temp;
                                     }
```

Shared data (initialized to **false**):

```
boolean lock;
boolean waiting[n];
```

Process P_i

```
        do     {
               key = true;
               while (key == true)
                       Swap(lock,key);
                       critical section
                       lock = false;
                       remainder section
               }
```

SEMAPHORES

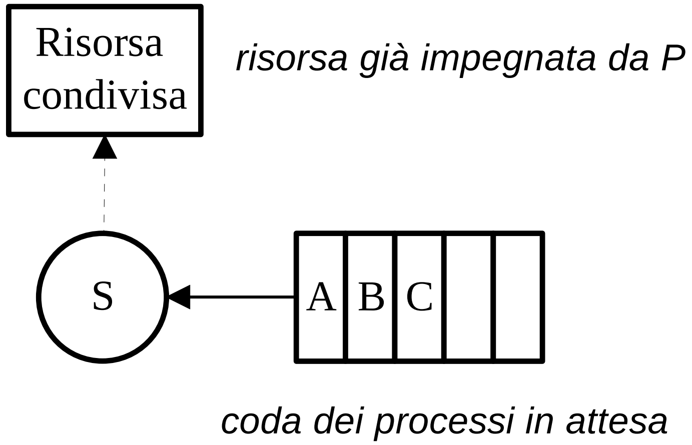
- Synchronization tool that does not require busy waiting.
- Semaphore *S* – integer variable
- can only be accessed via two indivisible (atomic) operations

wait (*S*):

```
   while S≤ 0 do no-op;
   S--;
```

signal (*S*):

```
  S++;
```



- *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.
- Can implement a counting semaphore *S* as a binary semaphore.

IMPLEMENTING *S* AS A BINARY SEMAPHORE

Data structures:

```
        binary-semaphore S1, S2;
        int C:
```

Initialization:

```
        S1 = 1
        S2 = 0
        C = initial value of semaphore S
```

wait operation

```
        wait(S1);
        C--;
        if (C < 0) {
                        signal(S1);
                        wait(S2);
                }
        signal(S1);
```

signal operation

```
        wait(S1);
        C ++;
        if (C <= 0)
                signal(S2);
        else
                signal(S1);
```

SEMAPHORE IMPLEMENTATION

Define a semaphore as a record

```
typedef struct {
               int value;
               struct process *L;

               } semaphore;
```

Assume two simple operations:

☞ **block** suspends the process that invokes it.

☞ **wakeup(*P*)** resumes the execution of a blocked process **P**.

Semaphore operations now defined as

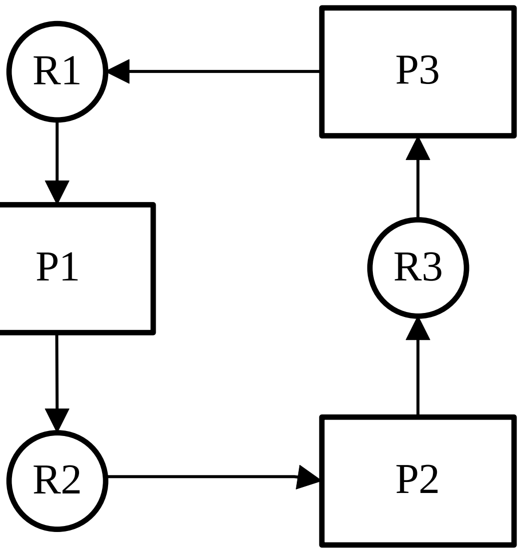
```
   wait(S):
        S.value--;
        if (S.value < 0) {
                       add this process to S.L;
                       block;
                       }

   signal(S):
        S.value++;
        if (S.value <= 0) {
                       remove a process P from S.L;
                       wakeup(P);
                       }
```

MUTUAL EXCLUSION RISKS

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.



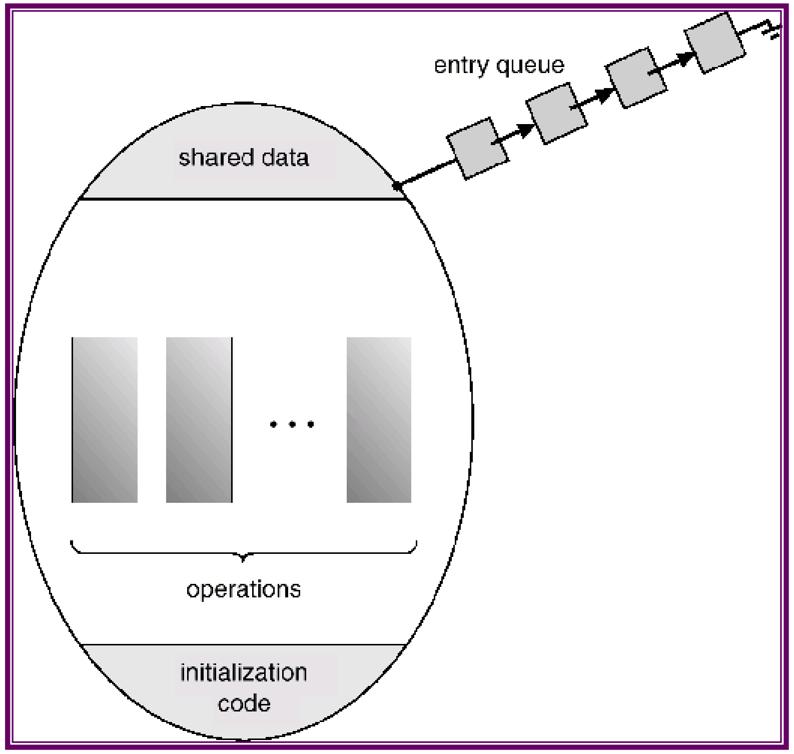
☞ **Starvation** – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.

MONITOR

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
            }
    }
```

MONITOR



MONITOR

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

```
        condition x, y;
```

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

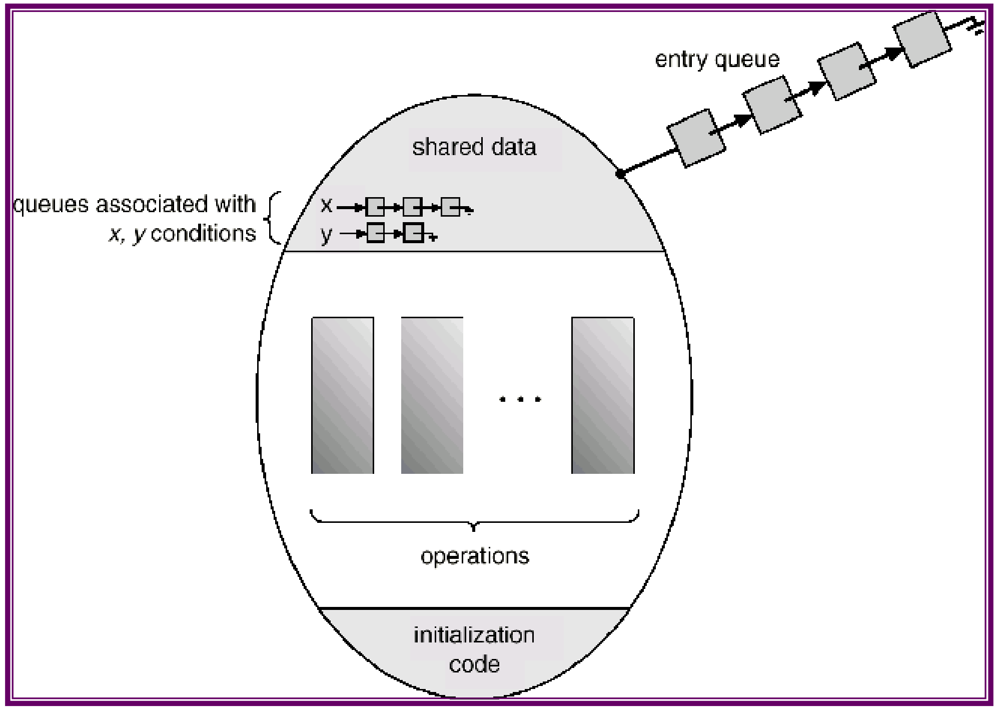
- The operation

  ```
       x.wait();
  ```

  means that the process invoking this operation is suspended until another process invokes

- The **x.signal** operation resumes exactly one suspended process. If no process is suspended, then the **signal** operation has no effect.

  ```
     x.signal();
  ```


entry queue
shared data
queues associated with x, y conditions
x
y
operations
initialization code