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.

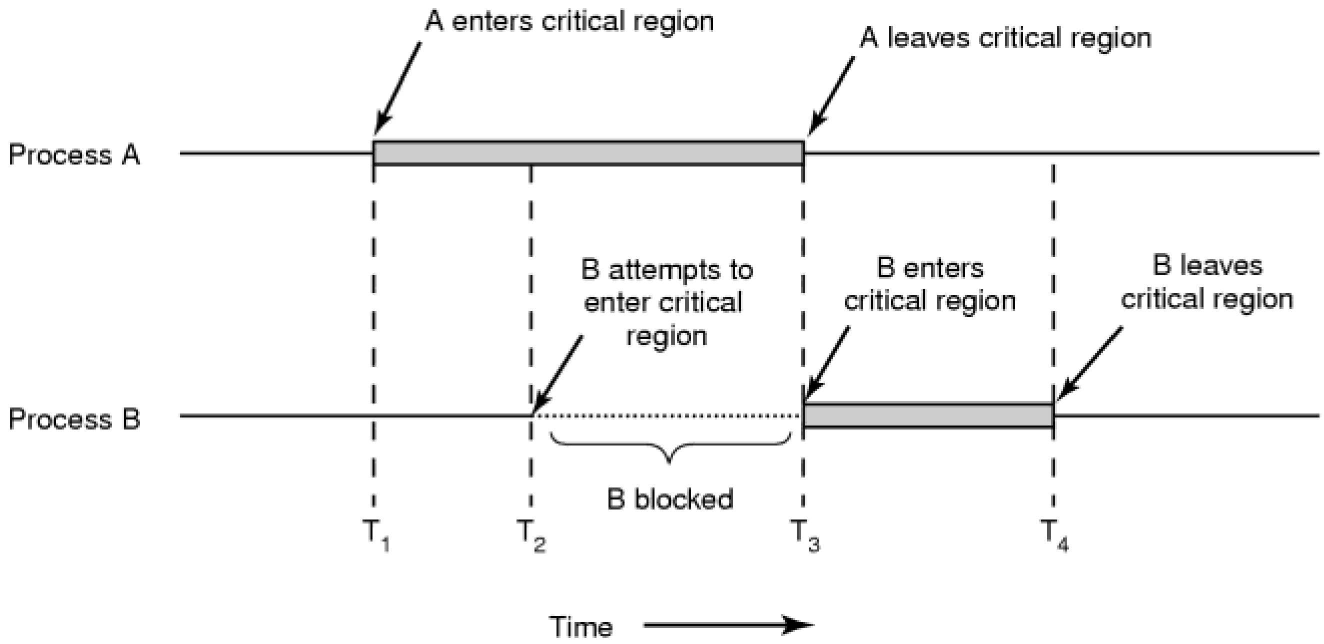
To prevent race conditions, concurrent processes must be **synchronized**.

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.

SOLVING THE CRITICAL-SECTION PROBLEM BY SOFTWARE

... is difficult and inefficient, even in the case of only 2 processes;

... requires an ordering algorithm known as ***bakery algorithm***;

... makes the processes access their critical sections in a strict FCFS basis.

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

Only hardware features can solve the critical-section problem effectively.

Adding a special instruction to the instruction set.

- ☞ Test and modify the content of a word atomically.

- ☞ Atomically swap two variables.

SEMAPHORES

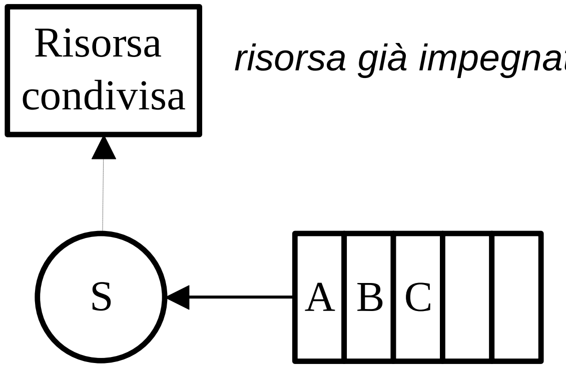
- Synchronization tool that does not require busy waiting.
- Semaphore *S* – integer variable
- can only be accessed via two indivisible (atomic) operations

wait (*S*):

```
while S≤ 0 do no-op;
S--;
```

signal (*S*):

```
S++;
```



risorsa già impegnata da P

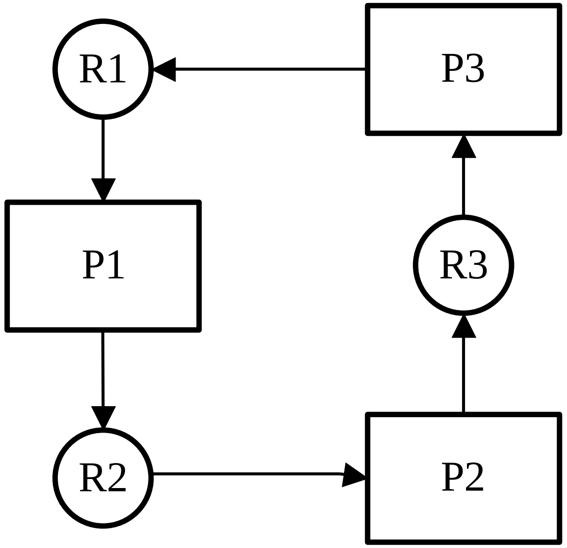
coda dei processi in attesa

- *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.

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.



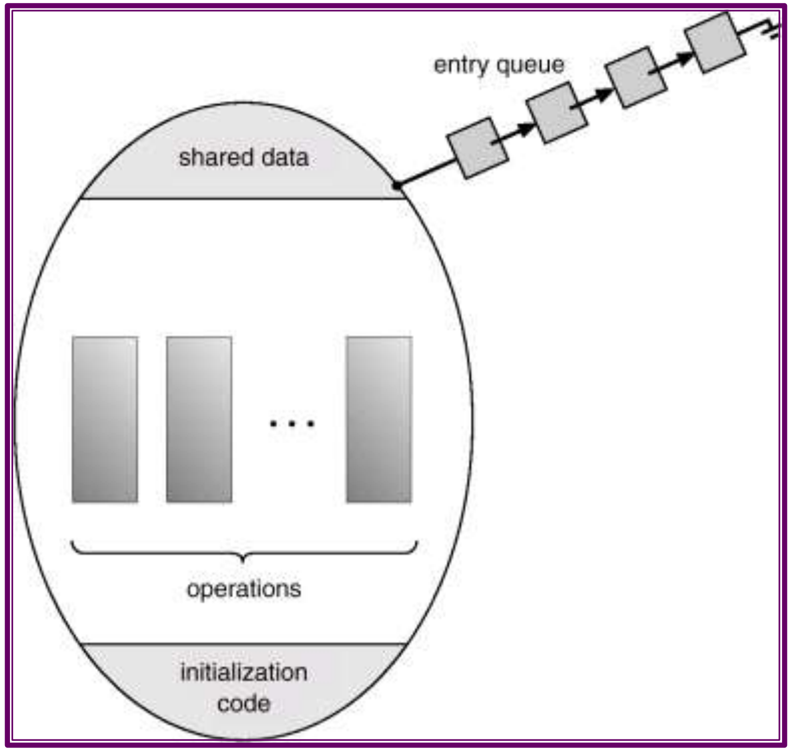
- **Starvation** – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.

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();
  ```