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.

THE RACE CONDITION

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.

- Frequente nei sistemi operativi multitasking, sia per dati in user space sia per strutture in kernel.
- Estremamente pericolosa: porta al malfunzionamento dei processi cooperanti, o anche (nel caso delle strutture in kernel space) dell'intero sistema.
- Difficile da individuare e riprodurre: dipende da informazioni astratte dai processi (decisioni dello scheduler, carico del sistema, utilizzo della memoria, numero di processori, . . .)

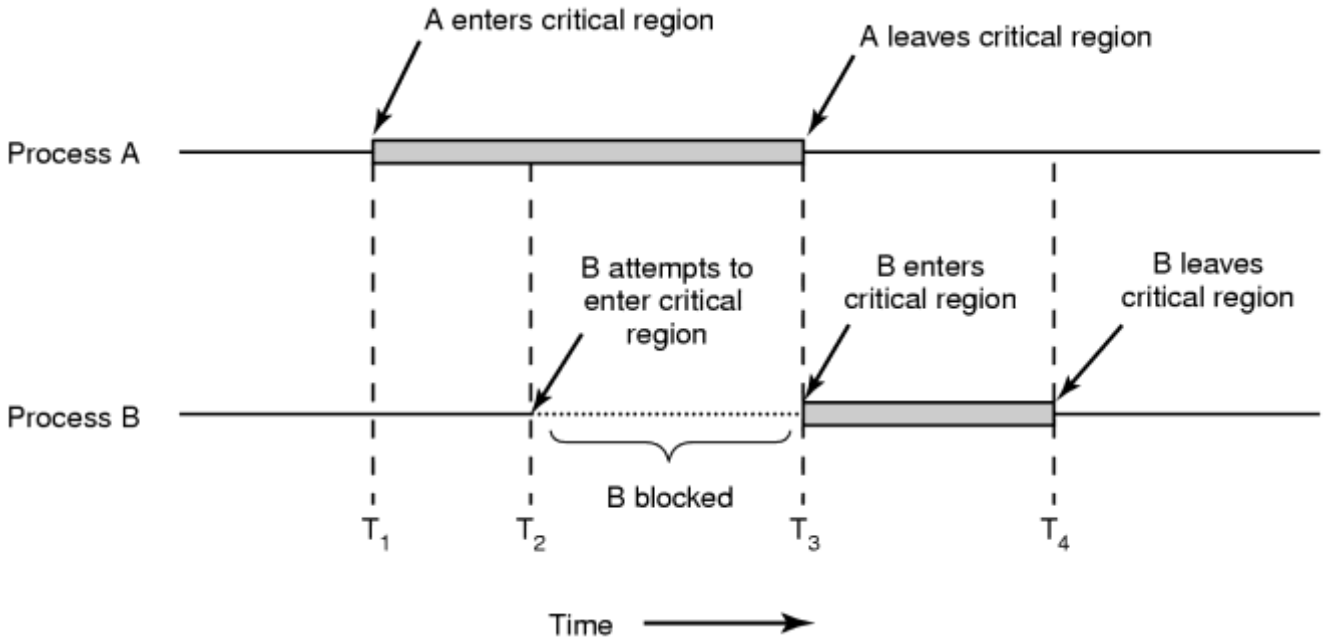
To prevent race conditions, concurrent processes must be **synchronized**.

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 – to avoid a race condition, it is necessary to 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 (1/2)

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 (2/2)

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_i and P_j

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);  no-op  /* 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_j is ready to enter its critical section, it cannot do so, even though P_i is in its remainder section.

→ **inadatto per processi con differenze di velocità**

– è un esempio di **busy wait**: attesa attiva di un evento (es: testare il valore di una variabile).
  - semplice da implementare
  - può portare a consumi inaccettabili di cpu
  - in genere, da evitare, ma a volte è preferibile (es. in caso di attese molto brevi)

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);
```

It is wrong.

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.

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

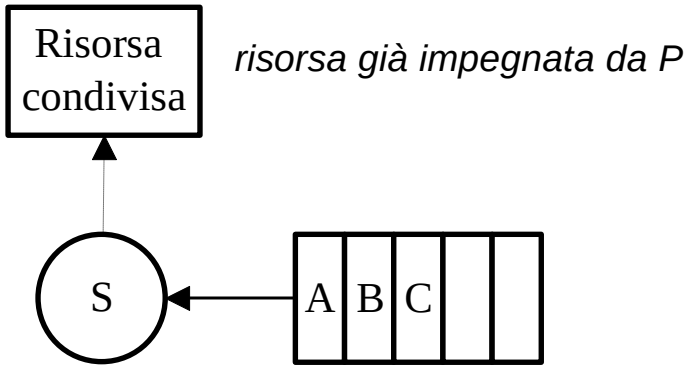
- Synchronization tool that does not require busy waiting.
- Semaphore *S* – integer variable
- can only be accessed via two indivisible (atomic) operations

wait (*S*): down(S): attendi finchè S è maggiore di 0; quindi decrementa S

```
while S≤ 0 do no-op;
S--;
```

signal (*S*): up(S): incrementa S

```
S++;
```



coda dei processi in attesa

Normalmente, l'attesa è implementata spostando il processo in stato di wait, mentre la up(S) mette uno dei processi eventualmente in attesa nello stato di ready.

I nomi originali erano **P** (proberen, testare) e **V** (verhogen, incrementare)

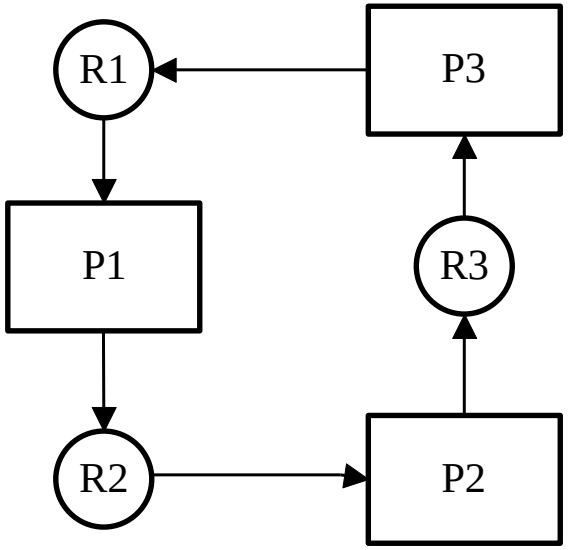
COUNTING SEMAPHORES

- *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. In informatica, per starvation (termine inglese che tradotto letteralmente significa inedia) si intende l'impossibilità, da parte di un processo pronto all'esecuzione, di ottenere le risorse hardware di processamento di cui necessita per essere eseguito.

  Un esempio tipico è il non riuscire ad ottenere il controllo della CPU da parte di processi con priorità molto bassa, qualora vengano usati algoritmi di scheduling a priorità. Può capitare, infatti, che venga continuamente sottomesso al sistema un processo con priorità più alta. Un aneddoto è la storia del processo con bassa priorità, scoperto quando fu necessario fermare il sistema sull'IBM 7094 al MIT nel 1973: era stato sottomesso nel 1967e fino ad allora non era ancora stato eseguito.

MONITOR

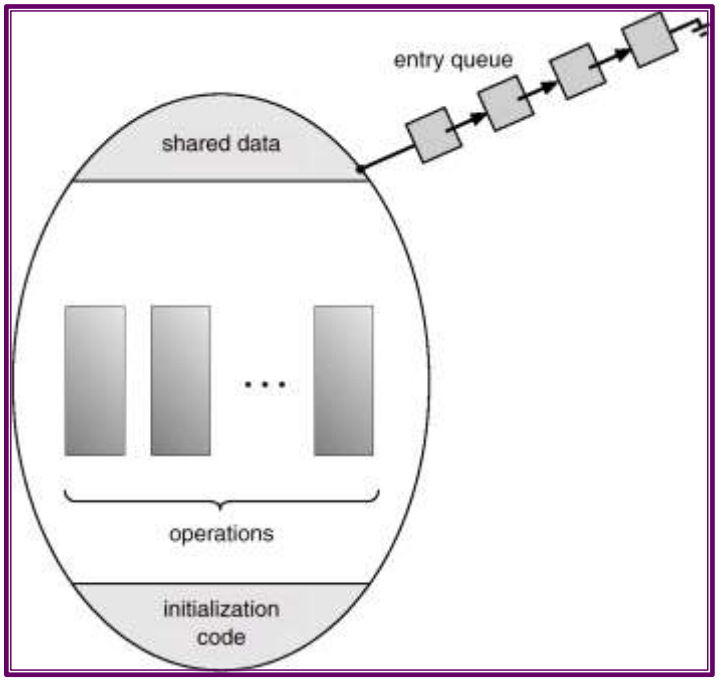
High-level synchronization construct that allows the safe sharing of an abstract data type among concurrent processes.

- Un monitor è un tipo di dato astratto che fornisce funzionalità di mutua esclusione
  - collezione di dati privati e funzioni/ procedure per accedervi.
  - i processi possono chiamare le procedure ma non accedere alle variabili locali.
  - un solo processo alla volta può eseguire codice di un monitor
- Il programmatore raccoglie quindi i dati condivisi e tutte le sezioni critiche relative in un monitor; questo risolve il problema della mutua esclusione
- Implementati dal compilatore con dei costrutti per mutua esclusione

```
monitor monitor-name
{shared variable declarations
        procedure body P1 (…)
        {       . . .
                }
        procedure body P2 (…)
        {       . . .
                }
        procedure body Pn (…)
        {       . . .
                }

                {
                initialization code
                }
}
```

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

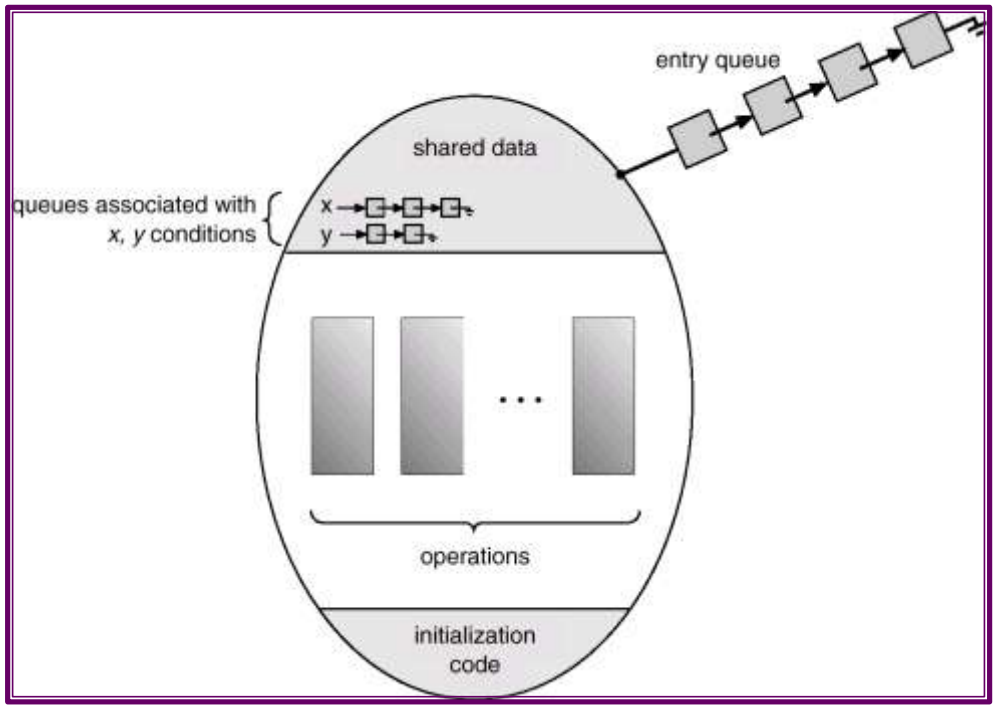
  ```
  x.wait();
  ```

  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();
  ```

MONITOR


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