

# Tema4: mecanismes de comunicació, sincronització i exclusió mútua

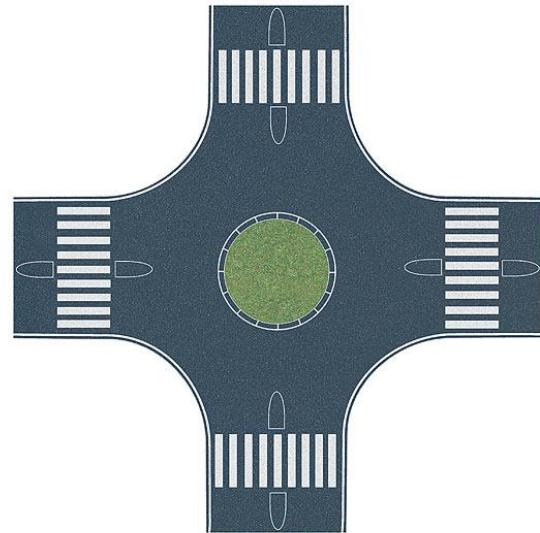
Xavi Canaleta  
La Salle, Universitat Ramon Llull  
[xavic@salleurl.edu](mailto:xavic@salleurl.edu)

# Índex

- ▶ 1. El problema de l'exclusió mútua i el deadlock
- ▶ 2. Solucions hardware
- ▶ 3. Solucions software
- ▶ 4. Comunicació entre processos.
  - ▶ 4.1 Pipes
  - ▶ 4.2 Pas de missatges
  - ▶ 4.3 Sockets
  - ▶ 4.4 Memòria compartida
- ▶ 5. Eines d'Exclusió Mútua i Sincronització
  - ▶ 5.1 Semàfors
  - ▶ 5.2 Monitors

# 1. Exclusió mútua i deadlock

- ▶ Exemple cruïlla de carrers
- ▶ Exemple caixer automàtic



# 1. Exclusió mútua i deadlock

var

    x:enter

fivar

procés A

...

tmpA:=x

[1a]

tmpA:=tmpA+1

[2a]

...

x:=tmpA

[3a]

fiprocés

procés B

...

tmpB = x

[1b]

tmpB:=tmpB-1

[2b]

...

x:=tmpB

[3b]

fiprocés

# 1. Exclusió mútua i deadlock

```
var
    buffer:array[0..MAX-1] de producte
    compt: enter
fivar
compt:=0

procés Productor()
var
    esc:enter
    C1:producte
fivar
    esc:=0
mentre CERT fer
    C1:=produir()
    mentre compt=MAX fimentre
        buffer[esc]:=C1
        esc:=(esc+1) mod MAX
        compt:=compt+1
fimentre
Fiprocés

procés Consumidor()
var
    lec:enter
    C2:producte
fivar
    lec:=0
mentre CERT fer
    mentre compt=0 fimentre
        C2:=buffer[lec]
        lec:=(lec+1) mod MAX
        compt:=compt-1
fimentre
fiprocés
```

# 1. Exclusió mútua i deadlock

```
var
    buffer:array[0..MAX-1] de producte
    compt: enter
fivar
compt:=0

procés Productor()
var
    esc:enter
    C1:producte
fivar
    esc:=0
mentre CERT fer
    C1:=produir()
    mentre compt=MAX fimentre
        buffer[esc]:=C1
        esc:=(esc+1) mod MAX
        compt:=compt+1
fimentre
Fiprocés

procés Consumidor()
var
    lec:enter
    C2:producte
fivar
    lec:=0
mentre CERT fer
    mentre compt=0 fimentre
        C2:=buffer[lec]
        lec:=(lec+1) mod MAX
        compt:=compt-1
fimentre
fiprocés
```

*podrien haver problemes*

“Que hem de respectar per que No peti i no ens restin -4 a l'examen! ” postulats de: DIJKSTRA

- + Hem de garantir que els postulats de l'exclusió mутua siguin
- 1. atomicitat (No divisible)
  - 2. Exclusió mutua
  - 3. Progressió
  - 4. Espua limitada.
- { Si no els respectem No estàbem!
- 4+1 (5) → No podem presuposar cap velocitat d'execució entre processos

- \* Exclusió Mútua → Hem de garantir que només 1 de tots els processos accedeix a aquell recurs que comparteixen entre aquests processos.
- \* Progressió → Si tiene un recurs compartit; no el necessita haig de permetre que altres processos accedeixin
- \* Espua limitada → Haig de garantir que si un procés vol accedir a un recurs ho farà amb un temps determinada

# Tema4: mecanismes de comunicació, sincronització i exclusió mútua

Xavi Canaleta  
La Salle, Universitat Ramon Llull  
[xavic@salleurl.edu](mailto:xavic@salleurl.edu)

anem a veure com arreglar els punts 2 i 3 del postulats de Dijkstra perquè tot sigui OK

- 1. El problema de l'exclusió mútua i el deadlock
- 2. Solucions hardware → *per la mà* *crear una instrucció nova a baix nivell (L'un conyaza...)*
- 3. Solucions software →
- 4. Comunicació entre processos.
  - 4.1 Pipes
  - 4.2 Pas de missatges
  - 4.3 Sockets
    - 4.4 Memòria compartida
- 5. Eines d'Exclusió Mútua i Sincronització
  - 5.1 Semàfors
  - 5.2 Monitors

# 2 Solucions Hardware

## 1. EI / DI : inhibir les interrupcions

## 2. INC / DEC :

\_mem = -1

A: inc \_mem  
jz B  
dec \_mem  
jmp A

B: //Regió Crítica Part que volem protegir  
dec \_mem Part que es comparteix

fa el mateix que C

C: inc \_mem incrementem \_mem  
jz B si el resultat del que acabem de fer a B anem a B per tocar la dada compartida  
dec \_mem decrementem \_mem  
jmp C → estem a un bucle

Si respecta p2 : p3 del postulat.  
No respectem p4 del postulat → Com sabem que Δ no entra  
2 cops i B no....?  
Resultat No OK.

Tan Δ com C són atòmiques, perquè estem a baix nivell.  
Putada → hem de programar això a baix nivell

### 3. TEST & SET

funció Test\_and\_Set (ref target: booleà) retorna booleà

var

    test: booleà

fivar

    test:=target

    target:=CERT

retorna test

fifunció

### Exemple:

var

    lock: booleà

fivar

    lock := FALS

repeteix

mentre Test\_and\_Set(lock) fimentre

        /\* Secció Critica \*/

        lock := FALSE

        /\* Secció Residual \*/

fins que FALS

## 4. SWAP:

Exemple:

```

var
    lock: boolean
fivar
proc rc()
    var
        key : booleà
    fivar
    repetir
        key := CERT
    repetir
        swap (lock, key)
    fins key = FALSE
    /* Secció Crítica */
    lock := FALSE
    /* Secció Residual */
    fins FALSE
fiproc

```

```

proc Swap (ref a, b: booleà)
    var
        temp: booleà
    fivar
        temp := a
        a := b
        b := temp
    fiproc

```

# 3 Solucions Software

Veurem evolució de arconseguir que tots els postulats es compleixin

## 3.1 Torns

var torn: enter fivar

torn = i

Pj: mentre(torn <> i) fer fimentre

/\* Secció Crítica \*/

torn := j

/\* Secció Residual A \*/

Pj: mentre(torn <> j) fer fimentre

/\* Secció Crítica \*/

torn := i

/\* Secció Residual B \*/

Si No es el meu torn me passo quan l'altre surti de la secció crítica m'assignarà el torn a mi.

## Postulats

2	3	4
✓	X	X

↳ Si l'altre passa No vol estir a la secció critica però hi està, l'altre m'hi nom

Sedrà el torn pagueu mar enriberà i ja un.

pagueu podria ser que algú li toqui està a la secció critica però que NO el necessiti!

## 3.2 Fets Consumats

**var** esta\_dins: **array**[1..2] **de booleà fivar**

Pi: **mentre**(esta\_dins[j]) **fimentre**

esta\_dins[i] := CERT

/\* secció crítica \*/

esta\_dins[i] := FALS

/\* secció residual \*/

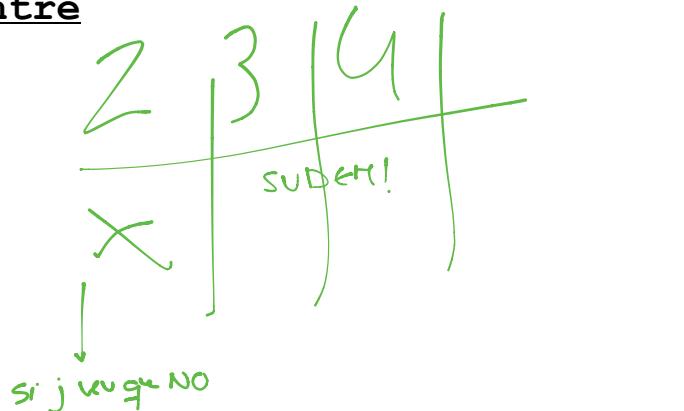
Pj: **mentre** (esta\_dins[i]) **fimentre** esta dins, i entra, si just entra/ la CPU

esta\_dins[j] := CERT

/\* secció crítica \*/

esta\_dins[j] := FALS

/\*secció residual \*/



flavors i també veo que NO a tè dins, no vost

els 2 processos estan a la secció crítica

### 3.3 Si us plau – L'Educat

var sol·licita: array[1..2] de booleà fivar

Pi: sol·licita[i] := CERT

mentre(sol·licita[j]) fimentre

/\* secció crítica \*/

sol·licita[i] := FALS

/\* secció residual \*/

Pi: sol·licita[j] := CERT

mentre(sol·licita[i]) fimentre

/\* secció crítica \*/

sol·licita[j] := FALS

/\* secció residual \*/

Espera circular... → Perge assignem els 2 proc a CERT, miren els 2 proc si l'altre està a cent, i si no, en pí que l'altre deixi d'estar cert per qd, mig dia que ja no està CERT perqüè està esperat



deadlock

Tot hi això p2 si el compleix p3 també! Però suspénja...

P4 No he  
podem assegurar

### 3.4 Si us plau més que mai – Més Educat

var sol·licita: array[1..2] de booleà fivar

Pi: sol·licita[i] := CERT

mentre(sol·licita[j]) fer

sol·licita[i] := FALS

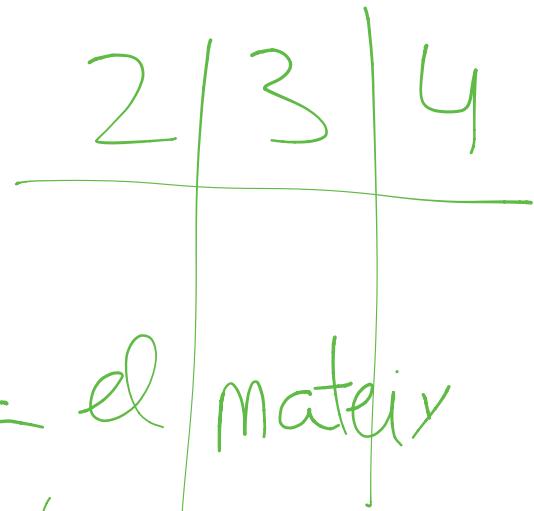
mentre(sol·licita[j]) fimentre

sol·licita[i] := CERT

fimentre

/\* secció crítica \*/

sol·licita[i] := FALS



Màxim, bixen prob de que es penji

### 3.4 Si us plau més que mai – Més Educat

```
Pj: sol·licita[j] := CERT
    mentre(sol·licita[i]) fer
        sol·licita[j] := FALS
        mentre(sol·licita[i]) fimentre
            sol·licita[j] := CERT
        fimentre
    /* secció crítica */
    sol·licita[j] := FALS
```

### 3.5 Decker

var sol·licita: array[1..2] de booleà

torn: enter

Mix de TIRNS amb l'Educet!

#### fivar

Procés i()

COMPLEIX TOT

sol·licita[i]:=CERT

mentre(sol·licita[j]) fer

si torn=j llavors

sol·licita[i] := FALS

mentre(torn=j) fimentre

sol·licita[i] := CERT

#### fisi

#### fimentre

/\* secció crítica \*/

sol·licita[i] := FALS torn := j

### 3.6 Peterson

var sol·licita: array[1..2] de booleà

torn: enter

fivar

procés i()

sol·licita[i] := CERT

torn := j

mentre((sol·licita[j]) i(torn=j)) fimentre

/\* secció crítica \*/

sol·licita[i] := FALS

fiprocés

Optimitzem  
el Decker

## 3.6 Peterson

Procés j ()

sol·licita[j] := CERT

torn := i

mentre((sol·licita[i]))i(torn=i))fimentre

/\* secció crítica \*/

sol·licita[j] := FALS

fiprocés

### **3.7 Algorismes per a n processos**

Algorisme d'Eisenberg – Mc Guire (1972)

Algorisme de Lamport (1974)

~~SEMDIFORS~~

→ es una variable compartida entre procesos que s'aplica.

init → inicial

wait → mira el valor del semáforo, si es menor o igual a 0  
si es menor a 0, agrega el PID del proceso, se lo guarda y el bloquea.

signal → Mira la wa del semáforo, si NO hi ha cap proc bloquejat, suma +1  
Si hi ha algun proc bloquejat, agafa el tr proceso de la wa i el passa a Ready



# Tema 4. Continguts

1. El problema de l'exclusió mútua i el dead-lock
2. Solucions hardware
3. Solucions software
4. Comunicació entre processos.
  - 4.1 Pipes
  - 4.2 Pas de missatges
  - 4.3 Sockets
5. Eines d'Exclusió Mútua i Sincronització
  - 5.1 Semàfors**
  - 5.2 Monitors

# 5.1. Semàfors

VARIABLE COMPARTIDA

## A) DEFINICIÓ I IMPLEMENTACIÓ

Dijkstra [1965]

```

tipus
    semàfor = registre
                compt: enter
                cua: cua processos bloquejats
                firegistre
fitipus

```

```

var
    s : semàfor
fivar

```

```

proc Init(var s:semafor; valor:enter)
    s.compt:=valor
fiproc

```

## 5.1. Semàfors

Wait → si  $n \geq \text{semàfor} > 0$  resta  
 $< 0$  s'apunta i s'autobloqueja

```

proc wait(var s:semafor)
    si s.compt > 0 llavors
        s.compt:=s.compt-1
    sino
        encuar(procés)
        bloquejar(procés)
    fisi
fiproc
    
```

```

proc signal(s:semàfor)
    si  $\neg s.cua.Buida()$  llavors
        desencuar(procés)
        desbloquejar(procés)
    sino
        s.compt:=s.compt+1
    fisi
fiproc
    
```

# 5.1. Semàfors

## B) UTILITZACIÓ

### 1. Exclusió mútua

```
var
  x:enter
fivar
```

```
x:=7
```

```
Proces A()
```

```
...
x:=x+3
...
```

```
fiproces
```

compartida

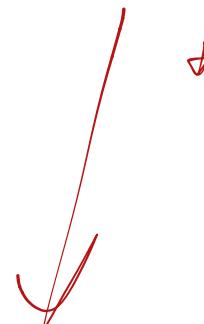
```
Proces B()
```

```
...
x:=x-5
...
```

```
fiproces
```

DIXÒ PÒT  
PONDRE DIFERENTS  
RESULTATS depenir  
del moment

↳ Com protegim la secció compartida?



# 5.1. Semàfors

## B) UTILITZACIÓ

### 1. Exclusió mútua

```

var
  x:enter
  s:semàfor alem semàfor
fivar
  
```

```

x:=7
Init(s,1) → l'iniciem a 1
  
```

Proces A()

$S=1 \leftarrow$  ...  
 Lo executem → wait(s)  
 $x:=x+3$   
 $S+1 = 2 \leftarrow$  signal(s)  
 ...

fiproces

Proces B()

...  
wait(s) → *imaginem que S=0, ambem a wait*  
 $x:=x-5$   
signal(s)  
 ...

fiproces

# 5.1. Semàfors

## B) UTILITZACIÓ

### 2. Sincronització

```
var
    x:enter
fivar

Proces A()
...
legeix(x)
...
fiproces
```

```
Proces B()
var
    r:real
fivar
...
r:=sqrt(x)
escriu (r)
...
fiproces
```

# 5.1. Semàfors

## B) UTILITZACIÓ

### 2. Sincronització

```

var
    x:enter
    s:semàfor
fivar
Init(s, 0)

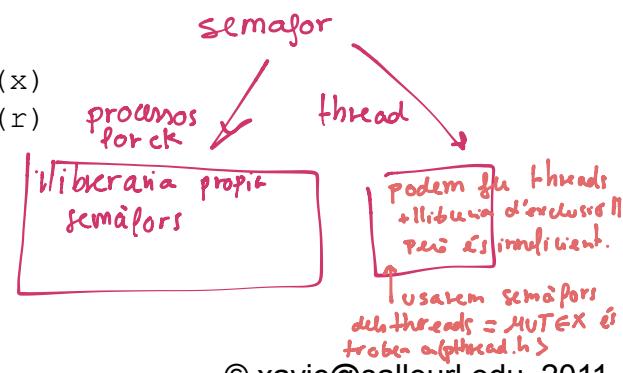
Proces A()
    ...
    lllegeix(x)
    signal(s)
    ...
fiproces

```

```

Proces B()
var
    r:real
fivar
    ...
    wait(s)
    r:=sqrt(x)
    escriu (r)
    ...
fiproces

```



# 5.1. Semàfors

## C) UTILITZACIÓ EN C

### Usar la llibreria pròpia semaphore.h

```
typedef struct
{
    int shmid;
} semaphore;
```

```
int SEM_constructor (semaphore * sem)

int SEM_init (const semaphore * sem, const int v)

int SEM_destructor (const semaphore * sem)

int SEM_wait (const semaphore * sem)

int SEM_signal (const semaphore * sem)
```

MUTEX NO SINCRONITZA CIÓ!!



## 5.1. Semàfors

### C) UTILITZACIÓ EN C

#### Exemple d'utilització semaphore.h

```
...
semaphore sem;
...
int main() {
    SEM_constructor(&sem); constructor
    SEM_init (&sem, 0); si usem exclusió mètua
    ...
    SEM_wait(&sem);
    // regió crítica a protegir
    SEM_signal(&sem);
    ...
    SEM_destructor (&sem);
```

↓  
~~Kempf~~

# Sistemes Operatius: threads i semàfors

© Xavi Canaleta, 2016

# Threads i semàfors

```
static int glob = 0;
```

```
static void * threadFunc (void *arg)
{
int loops = *((int *) arg);
int j;

for (j = 0; j < loops; j++) {
    glob++;
}
return NULL;
}
```



Nooo!!!

Pague No està protegida la variable `glob`  
posarem un `wait()` sobre `glob++` i un `signal()` sota `glob++`. Però com  
estem amb threads els semàfors són bullshit → 2 pàgines més  
avall hi ha la solució amb mutex! Que és la més mico

# Threads i semàfors

```

int main (int argc, char *argv[]){
pthread_t t1, t2;
int loops, s;

loops = (argc > 1) ? atoi (argv[1]) : 10000000;

enviem el mateix thrd
s = pthread_create (&t1, NULL, threadFunc, &loops);
s = pthread_create (&t2, NULL, threadFunc, &loops);

s = pthread_join (t1, NULL);
s = pthread_join (t2, NULL);

printf ("glob = %d\n", glob); si loops=10000 hauria de donar 20000
exit (EXIT_SUCCESS);

}

```

} loops = d que hem passat sinó passen res loops = 10000...

Però NO



# Threads i semàfors

```
static int glob = 0;
static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER; s'inicialitza

static void * threadFunc (void *arg)
{
int loops = *((int *) arg);
int j,s;

for (j = 0; j < loops; j++) {
    pthread_mutex_lock(&mtx);    seria un "wait"
        glob++;
    pthread_mutex_unlock(&mtx);    seria un "signal" } } si fossin memòtors
}
return NULL;
}
```



# Threads i semàfors

```
int main (int argc, char *argv[]){
pthread_t t1, t2;
int loops, s;

loops = (argc > 1) ? atoi (argv[1]) : 10000000;

pthread_create (&t1, NULL, threadFunc, &loops);
pthread_create (&t2, NULL, threadFunc, &loops);

pthread_join (t1, NULL);
pthread_join (t2, NULL);

pthread_mutex_destroy(&mtx); —> el matem.

printf ("glob = %d\n", glob);
exit (EXIT_SUCCESS);

}
```



# Tema 4. Continguts

1. El problema de l'exclusió mútua i el dead-lock
2. Solucions hardware
3. Solucions software
4. Comunicació entre processos.
  - 4.1 Pipes
  - 4.2 Pas de missatges**
  - 4.3 Sockets
5. Eines d'Exclusió Mútua i Sincronització
  - 5.1 Semàfors
  - 5.2 Monitors

# Comunicació entre processos

Per intercanviar informació entre processos, ho podem fer de tres mètodes:

1. Memòria compartida
2. Pas de missatges
3. Pipes
4. Sockets

## **Memòria compartida:**

- Compartició de variables.
- Responsabilitat de comunicació usuari
- El sistema operatiu ofereix la memòria compartida.

## **Pas de missatges:**

- Responsabilitat de comunicació del sistema operatiu.
- La sincronització està implícita.

## **Avantatges i inconvenients:**

- En la memòria compartida:

- no hi ha protecció sobre les dades.
- no hi ha sincronització.
- accés directe a les dades.
- facilitat de comprensió.

- En el pas de missatges:

- la transferència de dades (send/receive).
- sincronització és inherent.
- s'utilitzen espais de memòria privats.
- és més segur.
- més lent (fa còpies de les dades).

# Pas de missatges

- El sistema operatiu ens ofereix unes primitives bàsiques:
  - send()
  - receive()
- Opcions d'implementació:
  - A) Enllaç directe o indirecte.
  - B) Buffering automàtic o sense buffering.
  - C) Mida fixa o variable.

## A) Enllaç directe o indirecte

**A1. Enllaç directe:** cada procés que vulgui enviar o rebre haurà de nombrar explícitament l'emissor o receptor de mateix. Les funcions que tenim són: send (B, msg)  
receive (A, msg)

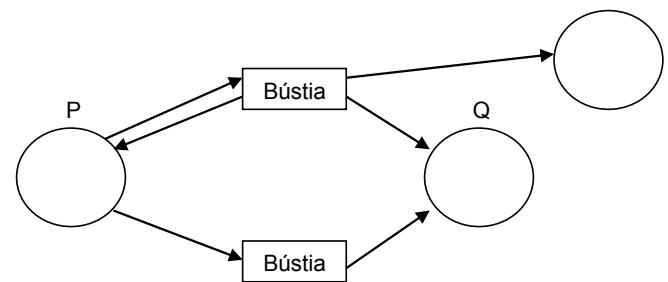
-envies missatge a algú especial

- S'estableix un enllaç automàticament, en cada parell de processos que es comuniquin.
- Un enllaç esta associat a 2 processos. Només poden establir un enllaç entre A i B.
- Els enllaços són bidireccionals.
- Si A canvia el seu nom, tots els que rebin d'ell han de canviar el nom amb que reben el missatge.

## A.2. Enllaç indirecte:

ho posa a una "bústia" i a veus qui ho rep

- Els missatges s'envien i reben de bústies (mailboxes) o de ports. Una bústia és un objecte on els processos poden deixar els missatges o llegir-los/escriure'ls.
- Enviem sempre a una bústia d'un procés, no al propi procés (la bústia pot pertànyer a un procés o al sistema operatiu).
- Cada bústia té un 'id', que és compartit pels processos.
- S'estableix un enllaç entre 2 o més processos. Si comparteixen una bústia poden tenir enllaços >2 processos.



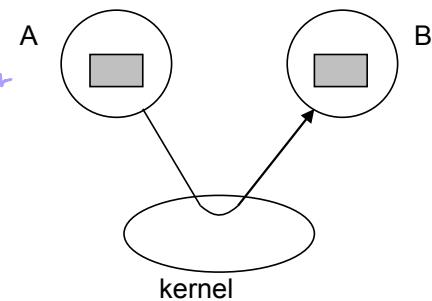
## B) Buffering explícit o automàtic

El Buffering és el número de missatges que es poden emmagatzemar temporalment en una bústia. Si hi ha buffering entremig dels processos, ens podem trobar amb les següents situacions:

### B1. Capacitat zero:

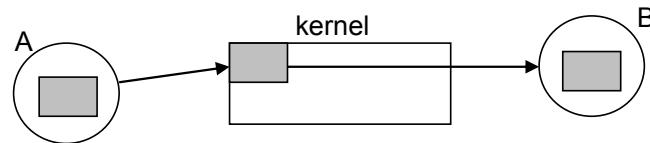
*No es guarda res, es com per sincronitzar*

- L'emissor espera al receptor; emissió i recepció del missatge són asíncrònies.
- A l'enviar un missatge hi ha una sincronització (*rendez\_vous*)
- 1 única còpia del missatge.
- Es bloqueja un procés quan no el rep ningú, o quan està esperant rebre i ningú l'envia.



## B2. Capacitat limitada (MAX missatges)

- L'emissor només es bloqueja quan la cua esta plena. Al fer send() no es bloqueja mentre número de missatges < MAX.
- Es fan 2 còpies del missatge. Una de l'origen al kernel i l'altre del kernel al destí.
- Es fa una copia en buffers (estructures) interns.



## B3. Capacitat il·limitada

- L'emissor fa sends i no es bloqueja mai.
- Es fan 2 còpies del missatge.

## C) Mida fixa o variable

### C1. Mida fixa:

- simplicitat d'implementació
- dificulta el treball del programador
- és el més usual

### C2. Mida Variable:

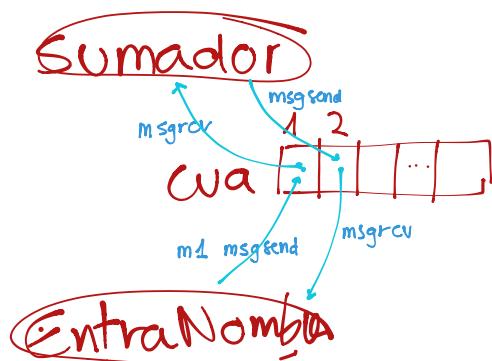
- d'implementació més complexa
- facilita el treball del programador
- problema: requereix gestió memòria

## Procediment de funcionament:

com si fos un num de bústia → podem usar-lo  
per enviar o recuperar msg

- Estructura pels missatges que es volen enviar i/o rebre. Ha de portar obligatoriament com **primer camp un long** per indicar un identificador del missatge. *(el camp del missatge serà un LONG per definir el id del missatge)*
- S'obté **una clau** a partir d'un fitxer existent qualsevol i d'un enter qualsevol. Tots els processos que vulguin compartir aquesta cua de missatges, hauran d'usar el mateix fitxer i el mateix enter. Crida a **ftok()**.
- Es **crea la cua de missatges** i s'obté un identificador per ella. Crida **msgget()**
- S'envia el missatge.** Crida a **msgsnd()**.
- Es rep un missatge.** Crida a **msgrcv()**.
- S'esborra** i es tanca la cua de missatges. Crida a **msgctl()**.

Ex: Tenim un procés que reb 2 num per teclat i els posa a una bústia  
Un altre procés que reb els 2 numeros i els suma i els torna a  
enviar al 1r procés, i aquell el mostre per pantalla.



```
//  
// Programa EntraNombres()  
//
```

```
#include <stdio.h>  
#include <errno.h>  
#include <sys/msg.h>  
  
typedef struct{  
    long idmis;  
    int n1,n2;  
}Missatge1;  
  
typedef struct{  
    long idmis;  
    int resultat;  
}Missatge2;
```

```

int main() {
key_t Clau;
int id_cua;
Missatge1 M1;
Missatge2 M2;
    fixar que només els interlocuts
    tenen num que només interlocuts
    sabent → NUM RANDOM
clau bústia
    Clau = ftok("EntraNombres.c", 12);
if (Clau==(key_t)-1) {
    printf("Error Clau\n\n");
    exit(-1);
}
    → accedir o crear (si no existeix)
id_cua = msgget(Clau, 0600|IPC_CREAT);
if(id_cua==-1) {
    printf("Error Creant cua\n\n");
    exit(-1);
}

```

```
M1.idmis=1; ← id de la bústia que volem
```

```
M1.n1=1;
```

```
while(M1.n1!=0) {
```

```
    printf("Entra num1:");
```

```
    scanf("%d", &M1.n1);
```

```
    printf("Entra num2:");
```

```
    scanf("%d", &M1.n2);
```

```
    msgsnd(id_cua, (struct msghdr *) &M1, sizeof(int)*2, IPC_NOWAIT);
```

```
    printf("enviant petició.....\n");
```

```
    if (M1.n1!=0) {
```

```
        msgsnd(id_cua, (struct msghdr *) &M2, sizeof(int), 2, 0);
```

```
        printf("%d+%d= %d\n\n", M1.n1, M1.n2, M2.resultat);
```

```
}
```

```
printf("Programa finalitzat\n\n");
```

```
msgctl (id_cua, IPC_RMID, (struct msqid_ds *)NULL);
```

```
}
```

*↳ eliminar cua*

*demarem nums*

*msg a enviar*

*mida msg a enviar  
SENSE comptar el long*

*→ si posem 0 → bloquem*

*L'envio msg a bústia i  
no m'espero a que la  
reballin*

*msg on ens guarda*

*mida*

*↓*

*Esperem*

*num bústia*

```
//Programa Sumador.c

//la primera part igual que la EntraNombres.c
mateixos structs!!!

M2.idmis=2;
for(;;){
    msgrcv(id_cua, (struct msgbuf *)&M1, sizeof(int)*2, 1, 0);
    printf("rebuts:%d,%d\n", M1.n1, M1.n2);
    if (M1.n1!=0) {
        M2.resultat=M1.n1+M1.n2;
        msgsnd(id_cua, (struct msgbuf *)&M2, sizeof(int), IPC_NOWAIT);
        printf("enviat = %d\n", M2.resultat);
    }
    else exit(1);
}
}
```

*//aqui NO eliminem la cuia perquè ja ho hem fet un cop!*