# OPERATING SYSTEMS: COMMUNICATION AND SYNCHRONIZATION AMONG PROCESSES

Threads and communication and synchronization mechanisms

# To remember…

| Before classes | Class | After class |
|---|---|---|

Prepare the prerequisites.

Study the material associated with the **bibliography**: slides alone are not enough.
Please ask questions (especially after study).

Exercising skills:
- ▸ Perform all **exercises**.
- ▸ Carrying out the **practice notebooks** and **the practical exercises** progressively.

# Recommended reading

## Base

1. **Carretero 2020:**
   1. Cap. 6
2. **Carretero 2007:**
   1. Cap. 6.1 and 6.2

## Suggested

1. **Tanenbaum 2006:**
   1. (es) Chap. 5
   2. (en) Chap. 5
2. **Stallings 2005:**
   1. 5.1, 5.2 and 5.3
3. **Silberschatz 2006:**
   1. 6.1, 6.2, 6.5 and 6.6

# Contents

- Introduction (definitions):
  - Concurrent processes.
  - Concurrency, communication and synchronization.
  - Critical section and Race conditions.
  - Mutual exclusion and critical section.
- Synchronization mechanisms (I):
  - Initial basic primitives.
  - Semaphores.
- Classic concurrency problems (I):
  - Producer-consumer
  - Reader-writers

- **Synchronization mechanisms of threads (II)**
  - **Semaphores**
    - **System calls for semaphores.**
    - **Classic concurrency problems.**
  - **Mutex and condition variables**
    - **System calls for mutex.**
    - **Classic concurrency problems.**

- Case study: concurrent server development

# Contents

- Introduction (definitions):
  - Concurrent processes.
  - Concurrency, communication and synchronization.
  - Critical section and Race conditions.
  - Mutual exclusion and critical section.
- Synchronization mechanisms (I):
  - Initial basic primitives.
  - Semaphores.
- Classic concurrency problems (I):
  - Producer-consumer
  - Reader-writers

- **Synchronization mechanisms of threads (II)**
  - **Semaphores**
    - **System calls for semaphores.**
    - Classic concurrency problems.
  - **Mutex and condition variables**
    - System calls for mutex.
    - Classic concurrency problems.

- Case study: concurrent server development

# POSIX Semaphores

- Synchronization mechanism for processes and/or threads on the same machine.

  ```
  #include <semaphore.h>
  ```
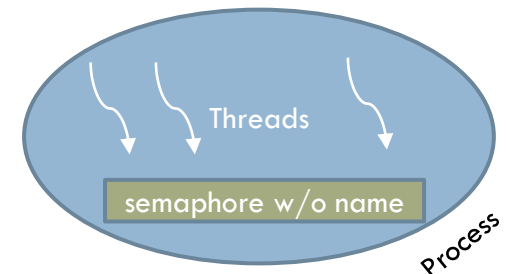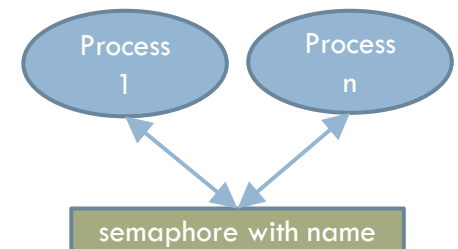
- Two types of POSIX semaphores:

  - **Semaphores with name:**
    - It can be used by different processes that know the name.
    - No shared memory required.
    - `sem_t *semaphore; // named`

  - **Semaphores without name:**
    - They can be used only by the processes that create them (and their threads) or by processes that have a shared memory area.
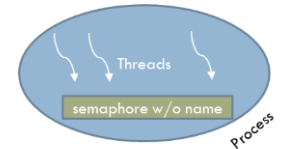    - `sem_t  semaphore; // no named`

Process 1

Process n

semaphore with name

Threads

semaphore w/o name

Process

ARCOS @ UC3M
Sistemas Operativos – Hilos y sincronización

# POSIX Semaphores

- int **sem_init**(sem_t *sem, int shared, int val);
  - Initializing an **unnamed** semaphore
- int **sem_destroy**(sem_t *sem);
  - Destroy an **unnamed** semaphore

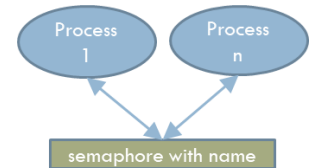- sem_t ***sem_open**(char *name, int flag, mode_t mode, int val);
  - Creates (or opens) a **named** semaphore.
- int **sem_close**(sem_t *sem);
  - Closes a **named** semaphore.
- int **sem_unlink**(char *name);
  - Remove a **named** semaphore.

- int **sem_wait**(sem_t *sem);
  - Performs the wait operation on the semaphore.
- int **sem_trywait** (sem_t *sem)
  - Attempts to wait but if it needs to block the process -> it does not block and gives -1
- int **sem_post**(sem_t *sem);
  - Performs the signal operation on the semaphore.

ARCOS @ UC3M
Sistemas Operativos – Hilos y sincronización

# Semaphores operations

```
sem_wait(s) {
    s = s - 1;
    if (s < 0) {
        <Blocking
         the process>
    }
}
```

```
sem_post(s) {
    s = s + 1;
    if (s <= 0)
        <Unblock a
         process blocked
         by a wait
         operation>
    }
}
```

# Contents

- Introduction (definitions):
  - Concurrent processes.
  - Concurrency, communication and synchronization.
  - Critical section and Race conditions.
  - Mutual exclusion and critical section.
- Synchronization mechanisms (I):
  - Initial basic primitives.
  - Semaphores.
- Classic concurrency problems (I):
  - Producer-consumer
  - Reader-writers

- ## Synchronization mechanisms of threads (II)
  - ### Semaphores
    - System calls for semaphores.
    - **Classic concurrency problems.**
  - ### Mutex and condition variables
    - System calls for mutex.
    - Classic concurrency problems.

- Case study: concurrent server development

# Critical sections with semaphores

- ☐ The semaphore must have an initial value of 1:

```
sem_wait(s);  /* enter in the critical section */
   < critical section >
sem_post(s);  /* leave the critical section */
```

- ☐ Example:
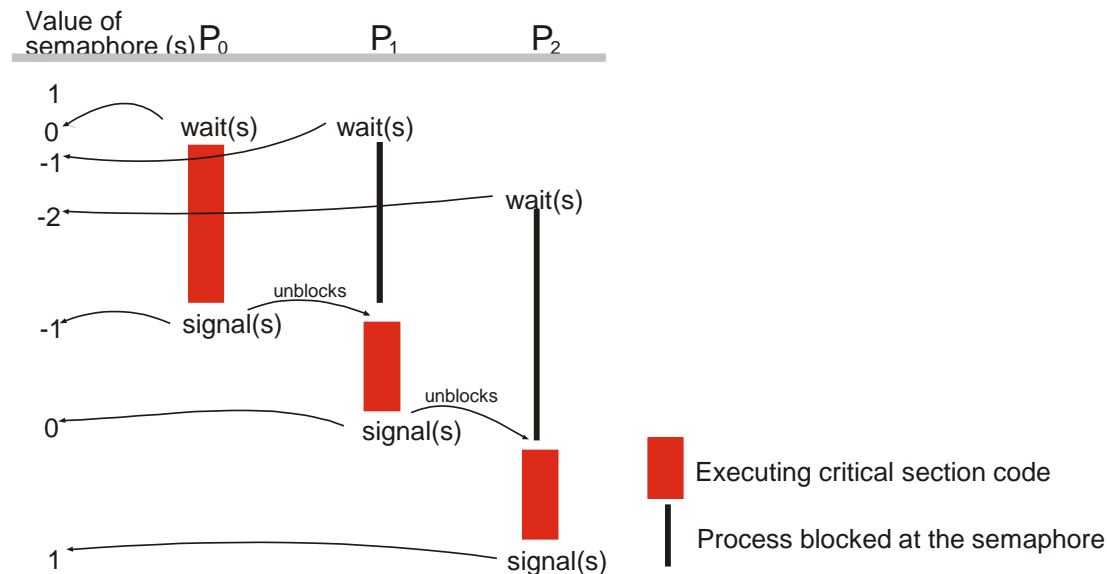


Value of semaphore (s)

Executing critical section code

Process blocked at the semaphore

# Producer-consumer with bounded buffered
## Semaphores without name

```c
/* buffer size*/
#define    MAX_BUFFER  1024
int buffer[MAX_BUFFER];    /* búfer común */
sem_t mutex;               /* critical section */
sem_t elements;            /* elements in buffer*/
sem_t huecos;              /* spaces in the buffer */
```

```c
void Producer(void)
{
    int pos = 0;
    int dato, i;

    for (i=0; i<DATA_TO_PRODUCE; i++)
    {
        dato = i;
        sem_wait(&huecos);
        sem_wait(&mutex);
        buffer[pos] = i;
        pos = (pos + 1) % MAX_BUFFER;
        sem_post(&mutex);
        sem_post(&elements);
    }
    pthread_exit(0);
}
```

```c
void Consumer ( void )
{
    int pos = 0;
    int dato, i;

    for (i=0; i<DATA_TO_PRODUCE; i++)
    {
        sem_wait(&elements);
        sem_wait(&mutex);
        dato = buffer[pos];
        pos = (pos + 1) % MAX_BUFFER;
        sem_post(&mutex);
        sem_post(&huecos);
        /* consume data */
    }
    pthread_exit(0);
}
```

# Producer-consumer with bounded buffered
## Semaphores without name (1/4)

```
//...
#include <semaphore.h>

/* buffer size */
#define MAX_BUFFER      1024
/* datos a producir */
#define DATA_TO_PRODUCE  100000

sem_t mutex;            /* critical section */
sem_t elements;         /* elements in the buffer */
sem_t huecos;           /* spaces in the buffer */
int buffer[MAX_BUFFER]; /* shared buffer */

int main ( int argc, char*argv[] )
{
    pthread_t th1, th2;

    /* initialize the semaphores */
    sem_init(&mutex, 0, 1);
    sem_init(&elements, 0, 0);
    sem_init(&huecos, 0, MAX_BUFFER);

    /* create threads */
    pthread_create(&th1, NULL, Producer, NULL);
    pthread_create(&th2, NULL, Consumer, NULL);

    /* wait for its completion */
    pthread_join(th1, NULL);
    pthread_join(th2, NULL);

    sem_destroy(&mutex);
    sem_destroy(&huecos);
    sem_destroy(&elements);
    return (0);
}

void Producer(void)
{
    int pos = 0;
    int dato, i;

    for (i=0; i<DATA_TO_PRODUCE; i++)
    {
        dato = i;
        sem_wait(&huecos);
        sem_wait(&mutex);
        buffer[pos] = i;
        pos = (pos + 1) % MAX_BUFFER;
        sem_post(&mutex);
        sem_post(&elements);
    }
    pthread_exit(0);
}

void Consumer ( void )
{
    int pos = 0;
    int dato, i;

    for (i=0; i<DATA_TO_PRODUCE; i++)
    {
        sem_wait(&elements);
        sem_wait(&mutex);
        dato = buffer[pos];
        pos = (pos + 1) % MAX_BUFFER;
        sem_post(&mutex);
        sem_post(&huecos);
        /* Consume data */
    }
    pthread_exit(0);
}
```
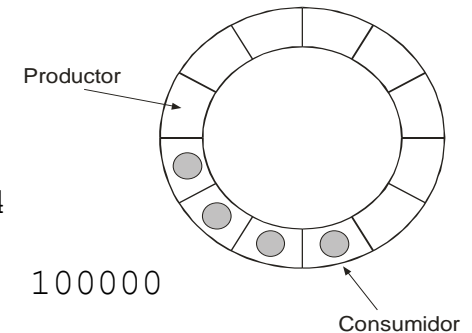
```
// ...
#include <semaphore.h>

/* buffer size*/
#define MAX_BUFFER  1024
/* data to produce */
#define DATA_TO_PRODUCE  100000


sem_t mutex;      /* critical section */
sem_t elements;  /* eltos in the buffer */
sem_t huecos;    /* spaces in the buffer */
int buffer[MAX_BUFFER]; /* shared buffer */


int main ( int argc, char *argv[])
{
    pthread_t th1, th2;

    /* initialize the semaphores */
    sem_init(&mutex,      0, 1);
    sem_init(&elements, 0, 0);
    sem_init(&huecos,     0, MAX_BUFFER);
```

Productor

Consumidor

# Producer-consumer with bounded buffered
## Semaphores without name (2/4)

```c
//...
#include <semaphore.h>

/* buffer size */
#define MAX_BUFFER      1024
/* datos a producir */
#define DATA_TO_PRODUCE 100000

sem_t mutex;            /* critical section */
sem_t elements;        /* elements in the buffer */
sem_t huecos;          /* spaces in the buffer */
int buffer[MAX_BUFFER]; /* shared buffer */

int main ( int argc, char*argv[] )
{
    pthread_t th1, th2;

    /* initialize the semaphores */
    sem_init(&mutex, 0, 1);
    sem_init(&elements, 0, 0);
    sem_init(&huecos, 0, MAX_BUFFER);

    /* create threads */
    pthread_create(&th1, NULL, Producer, NULL);
    pthread_create(&th2, NULL, Consumer, NULL);

    /* wait for its completion */
    pthread_join(th1, NULL);
    pthread_join(th2, NULL);

    sem_destroy(&mutex);
    sem_destroy(&huecos);
    sem_destroy(&elements);
    return (0);
}

void Producer(void)
{
    int pos = 0;
    int dato, i;

    for (i=0; i<DATA_TO_PRODUCE; i++)
    {
        dato = i;
        sem_wait(&huecos);
        sem_wait(&mutex);
        buffer[pos] = i;
        pos = (pos + 1) % MAX_BUFFER;
        sem_post(&mutex);
        sem_post(&elements);
    }
    pthread_exit(0);
}

void Consumer ( void )
{
    int pos = 0;
    int dato, i;

    for (i=0; i<DATA_TO_PRODUCE; i++)
    {
        sem_wait(&elements);
        sem_wait(&mutex);
        dato = buffer[pos];
        pos = (pos + 1) % MAX_BUFFER;
        sem_post(&mutex);
        sem_post(&huecos);
        /* Consume data */
    }
    pthread_exit(0);
}
```
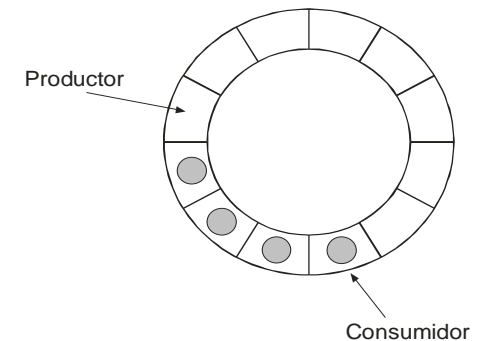


Productor

Consumidor

```c
/* create threads */
pthread_create(&th1, NULL, Producer, NULL);
pthread_create(&th2, NULL, Consumer, NULL);

/* wait for its completion */
pthread_join(th1, NULL);
pthread_join(th2, NULL);

sem_destroy(&mutex);
sem_destroy(&huecos);
sem_destroy(&elements);
return (0);
}
```

# Producer-consumer with bounded buffered
## Semaphores without name (3/4)

```c
//...
#include <semaphore.h>

/* buffer size */
#define MAX_BUFFER      1024
/* datos a producir */
#define DATA_TO_PRODUCE  100000

sem_t mutex;         /* critical section */
sem_t elements;      /* elements in the buffer */
sem_t huecos;        /* spaces in the buffer */
int buffer[MAX_BUFFER]; /* shared buffer */

int main ( int argc, char*argv[] )
{
    pthread_t th1, th2;

    /* initialize the semaphores */
    sem_init(&mutex, 0, 1);
    sem_init(&elements, 0, 0);
    sem_init(&huecos, 0, MAX_BUFFER);

    /* create threads */
    pthread_create(&th1, NULL, Producer, NULL);
    pthread_create(&th2, NULL, Consumer, NULL);

    /* wait for its completion */
    pthread_join(th1, NULL);
    pthread_join(th2, NULL);

    sem_destroy(&mutex);
    sem_destroy(&huecos);
    sem_destroy(&elements);
    return (0);
}

void Producer(void)
{
    int pos = 0;
    int dato, i;

    for (i=0; i<DATA_TO_PRODUCE; i++)
    {
        dato = i;
        sem_wait(&huecos);
        sem_wait(&mutex);
        buffer[pos] = i;
        pos = (pos + 1) % MAX_BUFFER;
        sem_post(&mutex);
        sem_post(&elements);
    }
    pthread_exit(0);
}

void Consumer ( void )
{
    int pos = 0;
    int dato, i;

    for (i=0; i<DATA_TO_PRODUCE; i++)
    {
        sem_wait(&elements);
        sem_wait(&mutex);
        dato = buffer[pos];
        pos = (pos + 1) % MAX_BUFFER;
        sem_post(&mutex);
        sem_post(&huecos);
        /* Consume data */
    }
    pthread_exit(0);
}
```

```c
void Producer(void)
{
    int pos = 0;
    int dato;
    int i;

    for (i=0; i<DATA_TO_PRODUCE; i++)
    {
        dato = i; /* produce... */
        sem_wait(&huecos);
        sem_wait(&mutex);
        buffer[pos] = i;
        pos = (pos + 1) % MAX_BUFFER;
        sem_post(&mutex);
        sem_post(&elements);

    }

    pthread_exit(0);
}
```

# Producer-consumer with bounded buffered
## Semaphores without name (4/4)

```
//...
#include <semaphore.h>

/* buffer size */
#define MAX_BUFFER       1024
/* datos a producir */
#define DATA_TO_PRODUCE  100000

sem_t mutex;            /* critical section */
sem_t elements;         /* elements in the buffer */
sem_t huecos;           /* spaces in the buffer */
int buffer[MAX_BUFFER]; /* shared buffer */

int main ( int argc, char*argv[] )
{
    pthread_t th1, th2;

    /* initialize the semaphores */
    sem_init(&mutex, 0, 1);
    sem_init(&elements, 0, 0);
    sem_init(&huecos, 0, MAX_BUFFER);

    /* create threads */
    pthread_create(&th1, NULL, Producer, NULL);
    pthread_create(&th2, NULL, Consumer, NULL);

    /* wait for its completion */
    pthread_join(th1, NULL);
    pthread_join(th2, NULL);

    sem_destroy(&mutex);
    sem_destroy(&huecos);
    sem_destroy(&elements);
    return (0);
}

void Producer(void)
{
    int pos = 0;
    int dato, i;

    for (i=0; i<DATA_TO_PRODUCE; i++)
    {
        dato = i;
        sem_wait(&huecos);
        sem_wait(&mutex);
        buffer[pos] = i;
        pos = (pos + 1) % MAX_BUFFER;
        sem_post(&mutex);
        sem_post(&elements);
    }
    pthread_exit(0);
}

void Consumer ( void )
{
    int pos = 0;
    int dato, i;

    for (i=0; i<DATA_TO_PRODUCE; i++)
    {
        sem_wait(&elements);
        sem_wait(&mutex);
        dato = buffer[pos];
        pos = (pos + 1) % MAX_BUFFER;
        sem_post(&mutex);
        sem_post(&huecos);
        /* Consume data */
    }
    pthread_exit(0);
}
```

```
void Consumer ( void )
{
    int pos = 0;
    int dato;
    int i;


    for (i=0; i<DATA_TO_PRODUCE; i++)
    {

        sem_wait(&elements);
        sem_wait(&mutex);
        dato = buffer[pos];
        pos = (pos + 1) % MAX_BUFFER;
        sem_post(&mutex);
        sem_post(&huecos);
        /* consume data */

    }

    pthread_exit(0);
}
```

# Reader-writer problem

- Problem that arises when you have:
  - A shared storage area.
  - Multiple processes read information.
  - Multiple processes write information.

- Conditions:
  - Any number of readers can read from the data zone concurrently: multiple readers possible at the same time.
  - Only one writer can modify the information at a time.
  - During a writing no reader can read
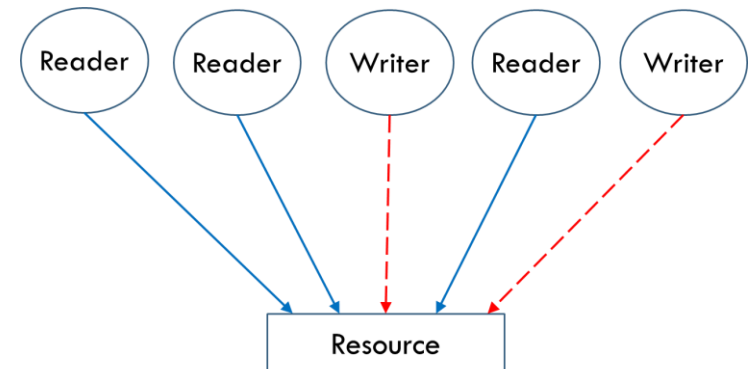
# Readers and writers
## Semaphores without name

```c
int dato = 5;         /* resource */
int n_readers = 0;    /* num. of readers */
sem_t sem_reader;     /* control access to n_readers */
sem_t mutex;          /* control access to dato */
```

```c
void Reader(void)
{
   sem_wait(&sem_reader);
   n_readers = n_readers + 1;
   if (n_readers == 1)
       sem_wait(&mutex);
   sem_post(&sem_reader);

   printf("%d\n", dato);

   sem_wait(&sem_reader);
   n_readers = n_readers - 1;
   if (n_readers == 0)
       sem_post(&mutex);
   sem_post(&sem_reader);
   pthread_exit(0);
}
```

```c
void Writer(void)
{
    sem_wait(&mutex);
    dato = dato + 2;
    sem_post(&mutex);

    pthread_exit(0);
}
```

# Readers and writers
## Semaphores without name

```
int dato = 5;          /* resource */
int n_readers = 0;     /* num. of readers */
sem_t sem_reader;      /* control access to n_readers */
sem_t mutex;           /* control access to dato */

int main ( int argc, char *argv[] )
{
    pthread_t th1, th2, th3, th4;
    sem_init(&mutex, 0, 1);
    sem_init(&sem_reader, 0, 1);

    pthread_create(&th1, NULL, Reader, NULL);
    pthread_create(&th2, NULL, Writer, NULL);
    pthread_create(&th3, NULL, Reader, NULL);
    pthread_create(&th4, NULL, Writer, NULL);
    pthread_join(th1, NULL);
    pthread_join(th2, NULL);
    pthread_join(th3, NULL);
    pthread_join(th4, NULL);

    sem_destroy(&mutex);
    sem_destroy(&sem_reader);

    return 0;
}
```

# Reader-writers (readers have priority)
## Semaphores without name

Alejandro Calderón Mateos

```
SHARED MEMORY:

int nreaders; semaphore s_read=1; semaphore s_write=1;
```

```
READER:

for(;;) {
  sem_wait(s_read);
  nreaders++;
  if (nreaders==1)
      sem_wait(s_write);
  sem_signal(s_read);

  perform_read();

  sem_wait(s_read);
  nreaders--;
  if (nreaders==0)
      sem_signal(s_write);
  sem_signal(s_read);
}
```

```
WRITER:

for(;;) {
  sem_wait(s_write);
  perform_write();
  sem_signal(s_write);
}
```

# Reader-writers (writers have priority)
## Semaphores without name

Alejandro Calderón Mateos

**SHARED MEMORY:**

```
int nreaders, nescr = 0;   semaphore lect, s_write = 1;
semaphore x, y, z = 1;
```

**READER:**

```
for(;;) {
    sem_wait(z);
    sem_wait(lect);
    sem_wait(x);
        nreaders++;
        if (nreaders==1)
            sem_wait(s_write);
    sem_signal(x);
    sem_signal(lect);
    sem_signal(z);
    // doReading();
    sem_wait(x);
        nreaders--;
        if (nreaders==0)
            sem_signal(s_write);
    sem_signal(x);
}
```

**WRITER:**

```
for(;;) {
    sem_wait(y);
        nescr++;
        if (nescr==1)
            sem_wait(lect);
    sem_signal(y);
    sem_wait(s_write);
        // doWriting();
    sem_signal(s_write);
    sem_wait(y);
        nescr--;
        if (nescr==0)
            sem_signal(s_readt);
    sem_signal(y);
}
```

REVIEW

# Semaphores with name
## Naming

□ Allow synchronization of different processes without using shared memory.

□ The name of a semaphore is a string of characters (with the same restrictions as a filename).

- If the name (path) is relative, only the process that creates it and its children can access the semaphore.

- If the name is absolute (starts with "/") the semaphore can be shared by any process that knows its name and has permissions.

□ Common mechanism for creating semaphores shared by parents and children

- "Unnamed" does not apply -> processes do NOT share memory.

ARCOS @ UC3M
Sistemas Operativos – Hilos y sincronización

# Semaphores with name
## Creation and use

☐ To create:

```
sem_t *sem_open(char *name, int flag, mode_t mode,int val);
```

- Flag = O_CREAT will create it.
- Flag: O_CREAT | O_EXECL will create it if it does not exist. -1 in case it exists.
- Mode: access permissions;
- Val: initial value of the semaphore (>=0);

☐ To use:

```
sem_t *sem_open(char *name, int flag);
```

- With flag 0. If it does not exist, it returns -1.

☐ Important:

- All processes must know "name" and use the same name.

# Readers and writers
## semaphores with name

```c
int main ( int argc, char *argv[] )
{
    int i, n= 5; pid_t pid;

    /* Creates the named semaphore */
    if ((mutex=sem_open("/tmp/sem_1", O_CREAT, 0644, 1))==(sem_t *)-1)
        { perror("Cannot create a semaphore: "); exit(1); }
    if((sem_reader=sem_open("/tmp/sem_2", O_CREAT, 0644, 1))==(sem_t *)-1)
        { perror("Cannot create a semaphore: "); exit(1); }

    /* Creates the processes */
    for (i = 1; i< atoi(argv[1]); ++i)
    {
      pid = fork();
      if (pid ==-1)
          { perror("The process cannot be created: "); exit(-1);}
      if (pid==0)
          { reader(getpid()); break; }
      else writer(pid);
    }
    sem_close(mutex);
    sem_close(sem_reader);
    sem_unlink("/tmp/sem_1");
    sem_unlink("/tmp/sem_2");

    return 0;

}
```

# Readers and writers
## semaphores with name

```
int dato = 5;           /* resource */
int n_readers = 0;      /* num. of readers */
sem_t *sem_reader;
sem_t *mutex;
```

```
void reader (int pid)
{
  sem_wait(sem_reader);
  n_readers = n_readers + 1;
  if (n_readers == 1)
      sem_wait(mutex);
  sem_post(sem_reader);

  printf("reader %d  dato: %d\n",
          pid, dato);

  sem_wait(sem_reader);
  n_readers = n_readers - 1;
  if (n_readers == 0)
      sem_post(mutex);
  sem_post(sem_reader);
}
```

```
void writer (int pid)
{
    sem_wait(mutex);
    dato = dato + 2;

    printf("writer %d  dato: %d\n",
            pid, dato);

    sem_post(mutex);
}
```
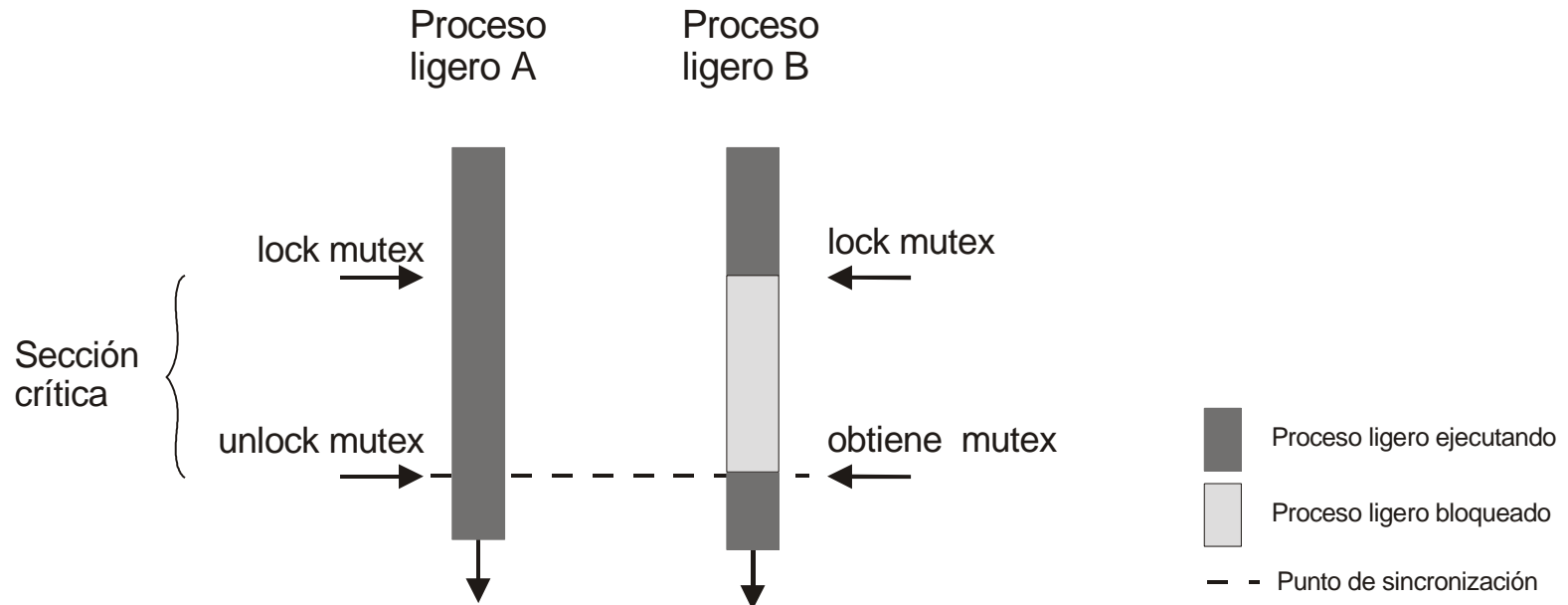
# Contents

- Introduction (definitions):
  - Concurrent processes.
  - Concurrency, communication and synchronization.
  - Critical section and Race conditions.
  - Mutual exclusion and critical section.
- Synchronization mechanisms (I):
  - Initial basic primitives.
  - Semaphores.
- Classic concurrency problems (I):
  - Producer-consumer
  - Reader-writers

- ## Synchronization mechanisms of threads (II)
  - ### Semaphores
    - System calls for semaphores.
    - Classic concurrency problems.
  - ### **Mutex and condition variables**
    - System calls for mutex.
    - Classic concurrency problems.

- Case study: concurrent server development

# Mutex and conditional variables

- A mutex is a synchronization mechanism suitable for threads.

- It is a binary semaphore with 2 atomic operations:
  - **lock(m)** Lock the mutex and if the mutex is already locked the process is suspended.
  - **unlock(m)** Unlocks the mutex and if there are suspended processes in the mutex then it unblock one.

- TIP: The `unlock` operation must be performed by the thread that executed `lock`
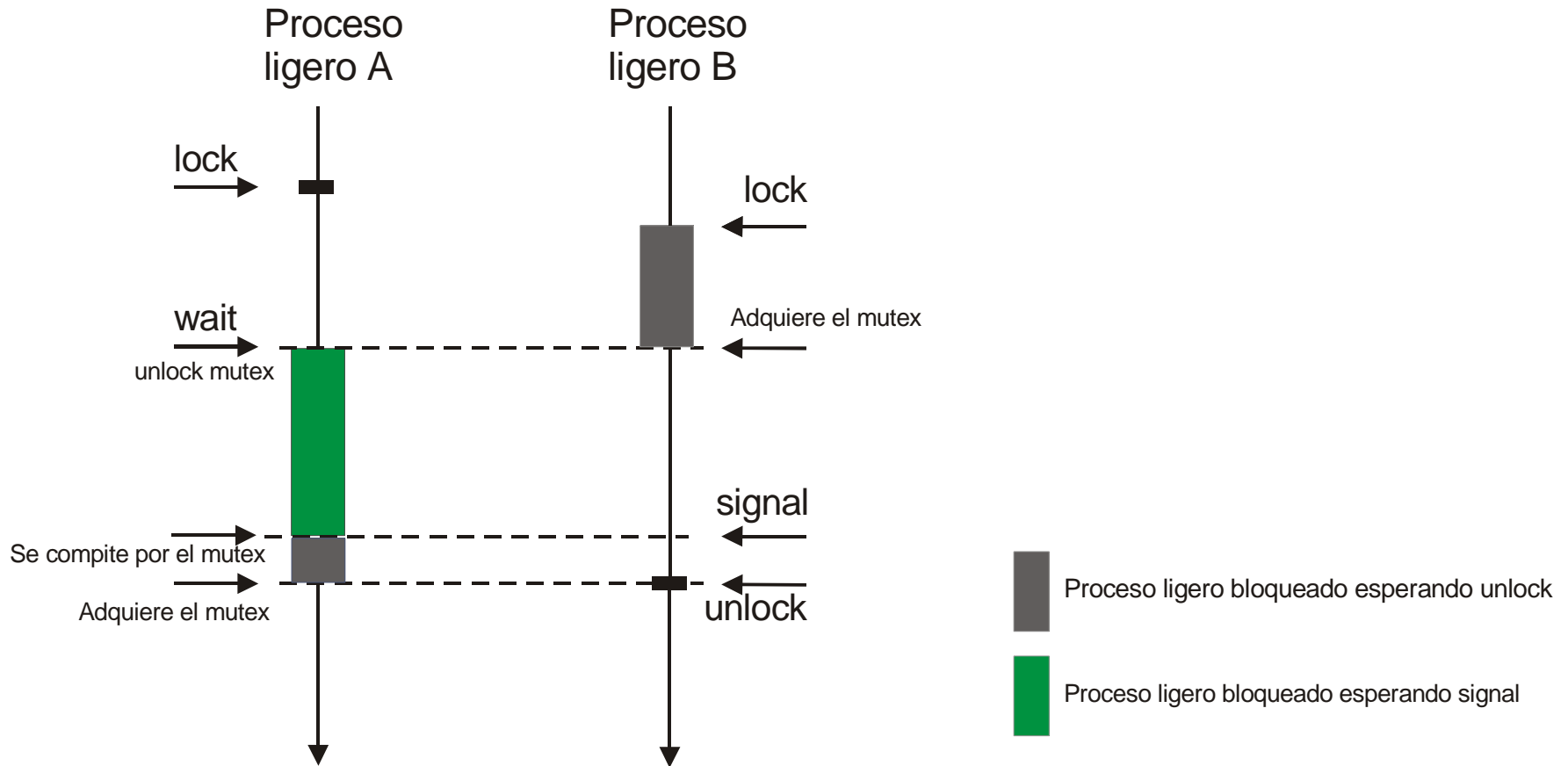
# Critical sections with mutex

```
lock(m);      /* enter in the critical section */
 < critical section >
unlock(m);    /* leave the critical section */
```



Proceso ligero A     Proceso ligero B

lock mutex     lock mutex

Sección crítica

unlock mutex     obtiene mutex

Proceso ligero ejecutando

Proceso ligero bloqueado

Punto de sincronización

ARCOS @ UC3M
Sistemas Operativos – Hilos y sincronización

# Conditional variables

- Synchronization variables associated with a mutex

- Two atomic operations:

  - **`wait:`**     Blocks the thread that executes it and ejects it from the mutex

  - **`signal:`** Unblocks one or more suspended processes in the conditional variable and the waking process competes again to lock the mutex

- Convenient to execute between `lock` and `unlock`

# Conditional variables

Proceso ligero A

Proceso ligero B

lock

lock

wait

Adquiere el mutex

unlock mutex

signal

Se compite por el mutex

Adquiere el mutex

unlock

Proceso ligero bloqueado esperando unlock

Proceso ligero bloqueado esperando signal

# Use of mutex and conditional variables

☐ **Thread A**

```
lock(mutex);   /* access to the resource */
check the data structures;
while (busy resource)
        wait(condition, mutex);
mark the resource as busy;
unlock(mutex);
```
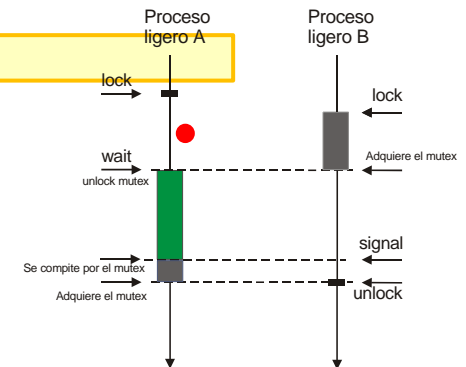
☐ **Thread B**

```
lock(mutex);   /* access to the resource */
mark the resource as free;
signal(condition);
unlock(mutex);
```
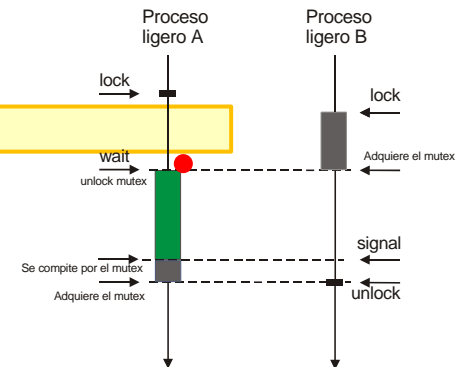
# Use of mutex and conditional variables

- ## Thread A

  ```
  lock(mutex);   /* access to the resource */
  check the data structures;
  while (busy resource)
          wait(condition, mutex);
  mark the resource as busy;
  unlock(mutex);
  ```

- ## Thread B

  ```
  lock(mutex);   /* access to the resource */
  mark the resource as free;
  signal(condition);
  unlock(mutex);
  ```

# Use of mutex and conditional variables

- ☐ **Thread A**

  ```
   lock(mutex);   /* access to the resource */
   check the data structures;
   while (busy resource)
          wait(condition, mutex);
   mark the resource as busy;
   unlock(mutex);
  ```
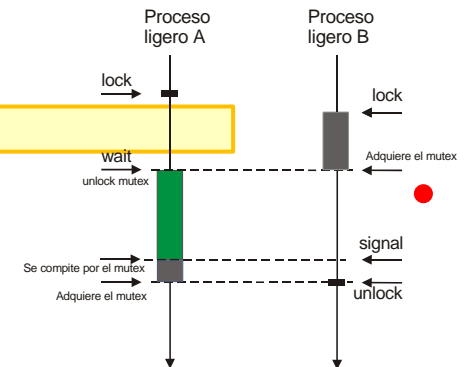
- ☐ **Thread B**

  ```
   lock(mutex);   /* access to the resource */
   mark the resource as free;
   signal(condition);
   unlock(mutex);
  ```

# Use of mutex and conditional variables

- ☐ **Thread A**

  **lock**(mutex);   /* access to the resource */
  check the data structures;
  while (busy resource)
          **wait**(condition, mutex);
  mark the resource as busy;
  **unlock**(mutex);

- ☐ **Thread B**

  **lock**(mutex);   /* access to the resource */
  mark the resource as free;
  **signal**(condition);
  **unlock**(mutex);

# Use of mutex and conditional variables

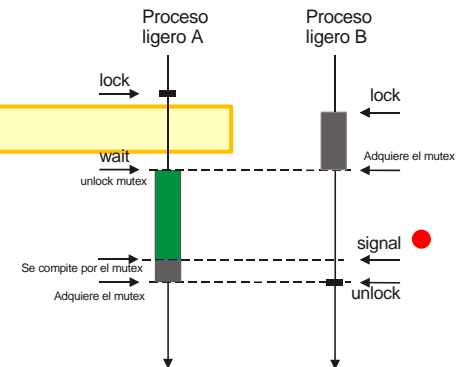☐ **Thread A**

```
lock(mutex);   /* access to the resource */
check the data structures;
while (busy resource)
        wait(condition, mutex);
mark the resource as busy;
unlock(mutex);
```

☐ **Thread B**

```
lock(mutex);   /* access to the resource */
mark the resource as free;
signal(condition);
unlock(mutex);
```

# Use of mutex and conditional variables
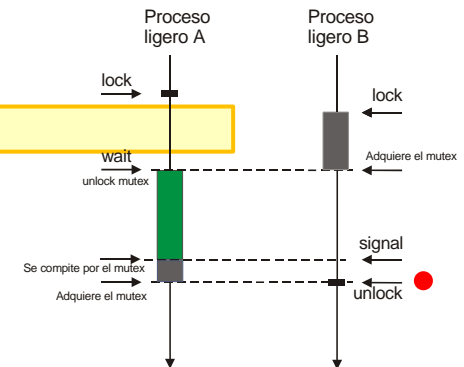
□ **Thread A**

```
lock(mutex);   /* access to the resource */
check the data structures;
while (busy resource)
        wait(condition, mutex);
mark the resource as busy;
unlock(mutex);
```

□ **Thread B**

```
lock(mutex);   /* access to the resource */
mark the resource as free;
signal(condition);
unlock(mutex);
```

# Use of mutex and conditional variables

- Thread A

```
lock(mutex);   /* access to the resource */
check the data structures;
while (busy resource)
        wait(condition, mutex);
mark the resource as busy;
unlock(mutex);
```

- Thread B

```
lock(mutex);   /* access to the resource */
mark the resource as free;
signal(condition);
unlock(mutex);
```
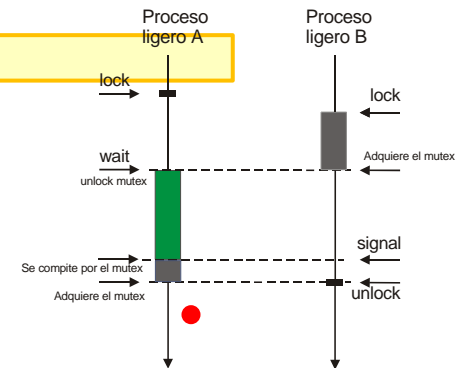
# Use of mutex and conditional variables

- □ Thread A

```
lock(mutex);   /* access to the resource */
check the data structures;
while (busy resource)
        wait(condition, mutex);
mark the resource as busy;
unlock(mutex);
```

Important to use `while` to re-evaluate condition

- □ Thread B

```
lock(mutex);    /* access to the resource */
mark the resource as free;
signal(condition);
unlock(mutex);
```

# Use of mutex and conditional variables

- Thread A

```
lock(mutex);   /* access to the resource */
check the data structures;
while (busy resource)
        wait(condition, mutex);
mark the resource as busy;
unlock(mutex);
```

- Thread B

```
lock(mutex);   /* access to the resource */
mark the resource as free;
signal(condition);
unlock(mutex);
```

# Use of mutex and conditional variables

☐ **Thread A**

```
lock(mutex);   /* access to the resource */
check the data structures;
while (busy resource)
        wait(condition, mutex);
mark the resource as busy;
unlock(mutex);
```

☐ **Thread B**

```
lock(mutex);    /* access to the resource */
mark the resource as free;
signal(condition);
unlock(mutex);
```

# Use of mutex and conditional variables

□ **Thread A**

```
lock(mutex);   /* access to the resource */
check the data structures;
while (busy resource)
        wait(condition, mutex);
mark the resource as busy;
unlock(mutex);
```

□ **Thread B**

```
lock(mutex);   /* access to the resource */
mark the resource as free;
signal(condition);
unlock(mutex);
```

> • **A signal before the wait is "lost".**
> • Herefore, the Boolean condition (free/busy resource) is important.

# Contents

- Introduction (definitions):
  - Concurrent processes.
  - Concurrency, communication and synchronization.
  - Critical section and Race conditions.
  - Mutual exclusion and critical section.
- Synchronization mechanisms (I):
  - Initial basic primitives.
  - Semaphores.
- Classic concurrency problems (I):
  - Producer-consumer
  - Reader-writers

- **Synchronization mechanisms of threads (II)**
  - **Semaphores**
    - System calls for semaphores.
    - Classic concurrency problems.
  - **Mutex and condition variables**
    - **System calls for mutex.**
    - Classic concurrency problems.

- Case study: concurrent server development

ARCOS @ UC3M
Sistemas Operativos – Hilos y sincronización

# POSIX Services

```
int pthread_mutex_init   ( pthread_mutex_t *mutex,
                           pthread_mutexattr_t * attr );
```
- Initializes a mutex.

```
int pthread_mutex_destroy ( pthread_mutex_t *mutex );
```
- Destroy a mutex.

```
int pthread_mutex_lock    ( pthread_mutex_t *mutex );
```
- Try to obtain the `mutex`.
- locks the thread if the `mutex` is acquired by another thread.

```
int pthread_mutex_unlock ( pthread_mutex_t *mutex );
```
- Unlock the `mutex`.

# POSIX Services

```
int pthread_cond_init ( pthread_cond_t*cond,
                        pthread_condattr_t*attr );
```
- Initializes a conditional variable.

```
int pthread_cond_destroy ( pthread_cond_t *cond );
```
- Destroys a conditional variable.

```
int pthread_cond_signal  ( pthread_cond_t *cond );
```
- At least one of the threads suspended in `cond` is reactivated.
- No effect if there is no thread waiting (different from semaphores).

```
int pthread_cond_wait    ( pthread_cond_t*cond,
                           pthread_mutex_t*mutex );
```
- At the same time suspends the calling thread and releases `mutex`.
- When another thread calls ..._cond_signal on `cond` the thread wakes up and competes again for the `mutex`.

# POSIX Services

```
int pthread_cond_broadcast ( pthread_cond_t *cond );
```

- All suspended threads in the conditional variable `cond` are reactivated.
- It has no effect if no thread is waiting.

# Contents

- **Synchronization mechanisms of threads (II)**
  - **Semaphores**
    - **System calls for semaphores.**
    - **Classic concurrency problems.**
  - **Mutex and condition variables**
    - **System calls for mutex.**
    - **Classic concurrency problems.**

# Producer-consumer with mutex

```
int main ( int argc, char *argv[] )
{
    pthread_t th1, th2;

    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&no_full, NULL);
    pthread_cond_init(&no_empty, NULL);

    pthread_create(&th1, NULL, Producer, NULL);
    pthread_create(&th2, NULL, Consumer, NULL);
    pthread_join(th1, NULL);
    pthread_join(th2, NULL);

    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&no_full);
    pthread_cond_destroy(&no_empty);
    return 0;
}
```

# Producer-consumer with mutex

```
#define MAX_BUFFER        1024    /* buffer size */
#define DATA_TO_PRODUCE  100000  /* data to be produced */

pthread_mutex_t mutex;              /* shared buffer access mutex */
pthread_cond_t no_full;             /* controls the filling of the buffer */
pthread_cond_t no_empty;            /* controls the emptying of the buffer */
int n_elements;                     /* number of elements in the buffer */
int buffer[MAX_BUFFER];             /* common buffer */
```

```
void Producer ( void )
{
  int dato, i ,pos = 0;

  for(i=0; i<DATA_TO_PRODUCE; i++)
  {
    dato = i;
    pthread_mutex_lock(&mutex);
    while (n_elements == MAX_BUFFER)
          pthread_cond_wait(&no_full,
                                &mutex);
    buffer[pos] = i;
    pos = (pos + 1) % MAX_BUFFER;
    n_elements ++;
    pthread_cond_signal(&no_empty);
    pthread_mutex_unlock(&mutex);
  }
  pthread_exit(0);
}
```

```
void Consumer(void)
{
  int dato, i ,pos = 0;

  for(i=0; i<DATA_TO_PRODUCE; i++)
  {
    pthread_mutex_lock(&mutex);
    while (n_elements == 0)
          pthread_cond_wait(&no_empty,
                                &mutex);
    dato = buffer[pos];
    pos = (pos + 1) % MAX_BUFFER;
    n_elements --;
    pthread_cond_signal(&no_full);
    pthread_mutex_unlock(&mutex);
    printf("Consume %d \n", dato);
  }
  pthread_exit(0);
}
```

# Producer-consumer with mutex

```c
#define MAX_BUFFER      1024    /* buffer size */
#define DATA_TO_PRODUCE 100000  /* data to be produced */

pthread_mutex_t mutex;          /* shared buffer access mutex */
pthread_cond_t no_full;         /* controls the filling of the buffer */
pthread_cond_t no_empty;        /* controls the emptying of the buffer */
int n_elements;                 /* number of elements in the buffer */
int buffer[MAX_BUFFER];         /* common buffer */
```

```c
void Producer ( void )
{
  int dato, i ,pos = 0;

  for(i=0; i<DATA_TO_PRODUCE; i++)
  {
    dato = i;
    pthread_mutex_lock(&mutex);
    while (n_elements == MAX_BUFFER)
          pthread_cond_wait(&no_full,
                              &mutex);
    buffer[pos] = i;
    pos = (pos + 1) % MAX_BUFFER;
    n_elements ++;
    pthread_cond_signal(&no_empty);
    pthread_mutex_unlock(&mutex);
  }
  pthread_exit(0);
}
```

```c
void Consumer(void)
{
  int dato, i ,pos = 0;

  for(i=0; i<DATA_TO_PRODUCE; i++)
  {
    pthread_mutex_lock(&mutex);
    while (n_elements == 0)
          pthread_cond_wait(&no_empty,
                              &mutex);
    dato = buffer[pos];
    pos = (pos + 1) % MAX_BUFFER;
    n_elements --;
    pthread_cond_signal(&no_full);
    pthread_mutex_unlock(&mutex);
    printf("Consume %d \n", dato);
  }
  pthread_exit(0);
}
```

# Reader-writers with mutex

```
int main ( int argc, char *argv[] )
{
    pthread_t th1, th2, th3, th4;

    pthread_mutex_init(&mutex, NULL);
    pthread_mutex_init(&mutex_readers, NULL);

    pthread_create(&th1, NULL, Reader, NULL);
    pthread_create(&th2, NULL, Writer, NULL);
    pthread_create(&th3, NULL, Reader, NULL);
    pthread_create(&th4, NULL, Writer, NULL);
    pthread_join(th1, NULL);
    pthread_join(th2, NULL);
    pthread_join(th3, NULL);
    pthread_join(th4, NULL);

    pthread_mutex_destroy(&mutex);
    pthread_mutex_destroy(&mutex_readers);
    return 0;

}
```

# Reader-writers with mutex

```c
int dato = 5;                    /* resource */
int n_readers = 0;               /* number of readers*/
pthread_mutex_t mutex;           /* control access to dato */
pthread_mutex_t mutex_readers;   /* control access to n_readers */
```

```c
void *Reader( void *arg )
{
  pthread_mutex_lock(&mutex_readers);
  n_readers++;
  if (n_readers == 1)
      pthread_mutex_lock(&mutex);
  pthread_mutex_unlock(&mutex_readers);

  /* read dato */
  printf("%d\n", dato);

  pthread_mutex_lock(&mutex_readers);
  n_readers--;
  if (n_readers == 0)
      pthread_mutex_unlock(&mutex);
  pthread_mutex_unlock(&mutex_readers);

  pthread_exit(0);
}
```

```c
void *Writer ( void *arg )
{
    pthread_mutex_lock(&mutex);

    /* modify the resource */
    dato = dato + 2;

    pthread_mutex_unlock(&mutex);

    pthread_exit(0);
}
```

# OPERATING SYSTEMS: COMMUNICATION AND SYNCHRONIZATION AMONG PROCESSES

Threads and communication and synchronization mechanisms