

```
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
```

```
#define MAXPAROLA 30
#define MAXRIGA 80
```

```
int main(int argc, char *argv[])
```

```
{
    int freq[MAXPAROLA]; /* vettore di contatori
    delle frequenze delle lunghezze delle parole */
    char riga[MAXRIGA];
    int i, inizio, lunghezza;
    FILE *f;
```

```
for(i=0; i<MAXPAROLA; i++)
    freq[i]=0;
```

```
if(argc != 2)
{
    fprintf(stderr, "ERRORE, serve un parametro con il nome del file\n");
    exit(1);
}
```

```
f = fopen(argv[1], "r");
if(f==NULL)
{
    fprintf(stderr, "ERRORE, impossibile aprire il file %s\n", argv[1]);
    exit(1);
}
```

```
while( fgets( riga, MAXRIGA, f ) != NULL )
```



# Synchronization

## Semaphores

Stefano Quer, Pietro Laface, and Stefano Scanzio

Dipartimento di Automatica e Informatica

Politecnico di Torino

[skenz.it/os](http://skenz.it/os)

[stefano.scanzio@polito.it](mailto:stefano.scanzio@polito.it)

# Introduction

- ❖ The previous solutions are not satisfactory, because
  - software solutions are complex to use from the point of view of the programmer
  - hardware solutions are difficult to implement from the point of view of the hardware designer
- ❖ OSs provide more appropriate primitives called **semaphores**
  - Introduced by Dijkstra in 1965
  - They are not based on busy waiting implementation, and therefore they do not waste resources

## Definition

❖ A semaphore **S** is a shared structure including

- A counter
- A waiting queue, managed by the kernel
- Both protected by a lock

```
typedef struct semaphore_tag {  
    char lock;                // Lock variable protects count  
                                // and queue management  
    int cnt;                  // Counter  
    process_t *head;          // Thread list  
} semaphore_t;
```

❖ Operations on **S** are **atomic**

- Atomicity is managed by the OS
- It is impossible for two threads to perform simultaneous operations on the same semaphore

## Definition

- ❖ A semaphore **S** is
  - An integer shared variable
  - Protected by the operating system
  - Usable for mutual exclusion and synchronization
- ❖ Operation on **S** are always executed in an atomic way
  - The atomicity is guaranteed by the operating system
  - It is impossible for two processes to execute concurrent operations on the same semaphore

Shared object, of integer type, which behaves as a shared counter

# Manipulation functions

## ❖ Typical operations on a semaphore S

### ➤ init (S, k)

- Defines and initializes the semaphore S to the value k

### ➤ wait (S)

sleep, down, P

- Allows (in the reservation code) to obtain the access of the CS protected by the semaphore S

### ➤ signal (S)

wakeup, up, V

- Allows (in the release code) to release the CS protected by the semaphore S

### ➤ destroy (S)

- Frees the semaphore S

They are not the "wait" and "signal" seen in the past



# Semaphore primitives

## ❖ `init(S, k)`

`k` is a counter

➤ Defines and initializes semaphore `S` to value `k`

➤ Two types of semaphores

- Binary semaphores

- The value of `k` is only **0** or **1**

known as "mutex lock"  
(mutex  $\equiv$  MUTual EXclusion)

- Counting semaphores

- The value of `k` is **non negative**

Logical implementation

```
init (S, k) {  
    alloc (S);  
    S=k;  
}
```

Atomic operation

# Semaphore primitives

## ❖ `wait (S)`

- If the counter value of **S** is negative or zero blocks the calling T/P
  - If S is negative, its absolute value |S| indicates the number of waiting threads
- The counter is decremented at each call

Logical implementation

```
wait (S) {  
    while (S <= 0);  
    S--;  
}
```

In the logical versions  
S is always positive

Real implementations do  
**not** use busy waiting

Atomic  
operation

Other possible (and equivalent)  
logical implementation

```
wait (S) {  
    if (S == 0) block();  
    else S--;  
}
```

# Semaphore primitives

## ❖ **wait (S)**

- Originally called **P ()** from the Dutch language "probeer te verlagen", i.e., "try to decrease"
- **Not** to be confused with the **wait** system call used to wait for a child process

Logical implementation

```
wait (S) {  
    while (S<=0);  
    S--;  
}
```

In the logical versions  
S is always positive

Real implementations do  
**not** use busy waiting

Atomic  
operation

Other possible (and equivalent)  
logical implementation

```
wait (S) {  
    if (S==0) block();  
    else S--;  
}
```



# Semaphore primitives

## ❖ **signal (S)**

- Increases the semaphore **s**
  - If **s** counter is negative or zero some T/P was blocked on the semaphore queue, and it can be wakeup
- Originally called **v()**, from the Dutch language "verhogen", i.e., "to increment"
- **Not to be confused** with system call **signal** that is used to declare a signal handler

Logical implementation

```
signal (S) {  
    S++;  
}
```

Other possible (and equivalent) logical implementation

```
signal (S) {  
    if (blocked())  
        wakeup();  
    else S++;  
}
```

Atomic operation  
(register=s; register++; s=register;)

# Semaphore primitives

## ❖ **destroy (S)**

### ➤ Release semaphore **S** memory

- Actual implementations of a semaphore require much more of a simple global variable to define a semaphore

### ➤ This function is often not used in the examples

```
destroy (S) {  
    free (S);  
}
```

Logical  
implementation

# Semaphore primitives

## ❖ The semaphore queue

- Is implemented in kernel space by means of a queue of Thread Control Blocks
- The kernel scheduler decides the queue management strategy (not necessarily FIFO)

# Mutual exclusion with semaphore

```
init (S, 1);
```

```
while (TRUE) {
    wait (S);
    CS
    signal (S);
    non critical section
}
```

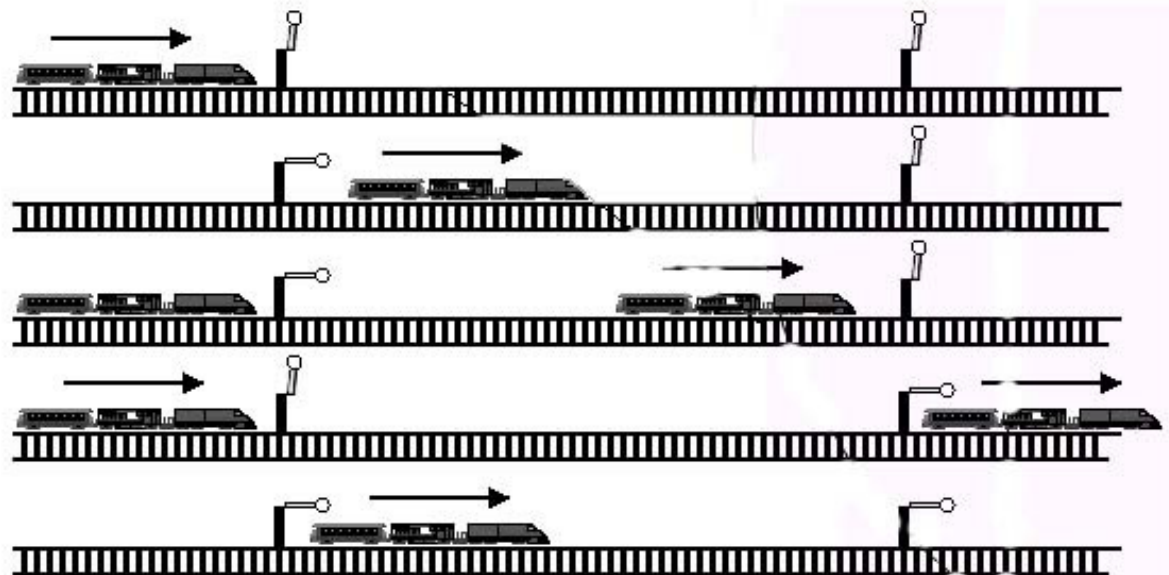
$P_i / T_i$

```
while (TRUE) {
    wait (S);
    CS
    signal (S);
    non critical section
}
```

$P_j / T_j$

## Remember:

```
wait (S) {
    if (S==0) block();
    else S--;
}
signal (S) {
    if (blocked())
        wakeup();
    else S++;
}
```



# Critical sections of N threads

```

init (S, 1);
...
wait (S);
CS
signal (S);
  
```

At most **one** T/P  
at a time in the  
critical section

T <sub>1</sub>	T <sub>2</sub>	T <sub>3</sub>	S	queue
			1	
wait			0	
CS <sub>1</sub>	wait		-1	T <sub>2</sub>
	blocked	wait	-2	T <sub>2</sub> , T <sub>3</sub>
		blocked	-2	T <sub>2</sub> , T <sub>3</sub>
signal			-2	T <sub>2</sub> , T <sub>3</sub>
	CS <sub>2</sub>	blocked	-1	T <sub>3</sub>
	signal		0	
		CS <sub>3</sub>	0	
		signal	1	



# Critical sections of N threads

```

init (S, 2);
...
wait (S);
CS
signal (S);
  
```

T <sub>1</sub>	T <sub>2</sub>	T <sub>3</sub>	S	queue
			2	
wait			1	
CS <sub>1</sub>	wait		0	
	CS <sub>2</sub>	wait	-1	T <sub>3</sub>
		blocked	-1	T <sub>3</sub>
signal			0	
		CS <sub>3</sub>	0	
	signal		1	
		signal	2	

Threads 1 and 2 in their CSs

Threads 2 and 3 in their CSs

At most **two** T/P at a time in the critical section

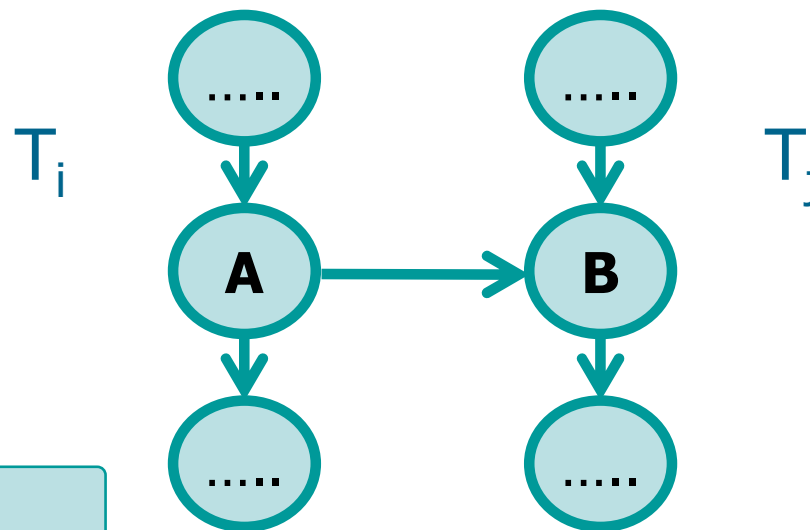
# Synchronization with semaphores

- ❖ The use of semaphores is not limited to the Critical Section access protocol
- ❖ Semaphores can be used to solve **any synchronization problem** using
  - An appropriate positioning of semaphores in the code
  - Possibly, more than one semaphore
  - Possibly, additional shared variables

# Pure synchronization: Example 1

❖ Obtain a specific order of execution

➤  $T_i$  executes code A before  $T_j$  executes code B



```
init (S, 0);
```

```
.....  
A;  
signal (S);  
.....
```

 $T_i$ 

```
.....  
wait (S);  
B;  
.....
```

 $T_j$

## Pure synchronization: Example 2

❖ Synchronize two T/P so that

- $T_j$  waits  $T_i$
- then,  $T_i$  waits  $T_j$
- It is a client-server schema

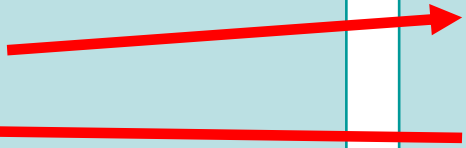
```
init (S1, 0);  
init (S2, 0);
```

$T_i / P_i$

```
while (TRUE) {  
    prepare data  
    signal (S1);  
    wait (S2);  
    get processed data  
}
```

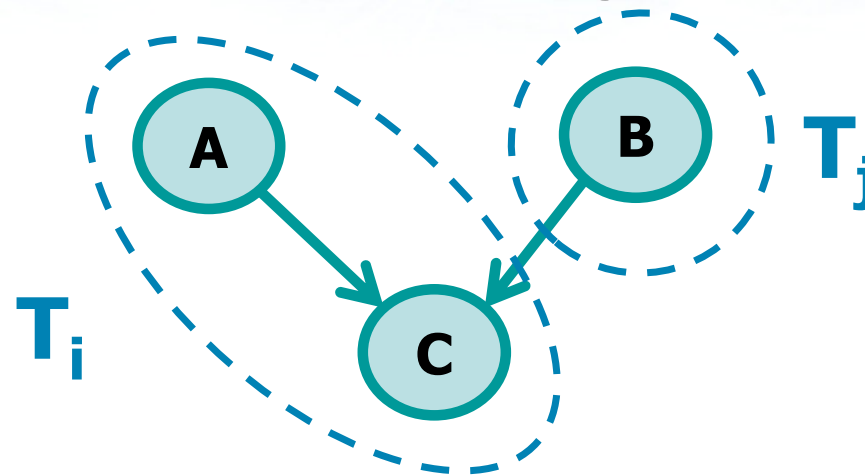
$T_j / P_j$

```
while (TRUE) {  
    wait (S1);  
    process data  
    signal (S2);  
    ...  
}
```



## Pure synchronization : Example 3

❖ Implement this precedence graph



```
init (S, 0);
```

```
A;  
wait (S);  
C;
```

 $T_i$ 

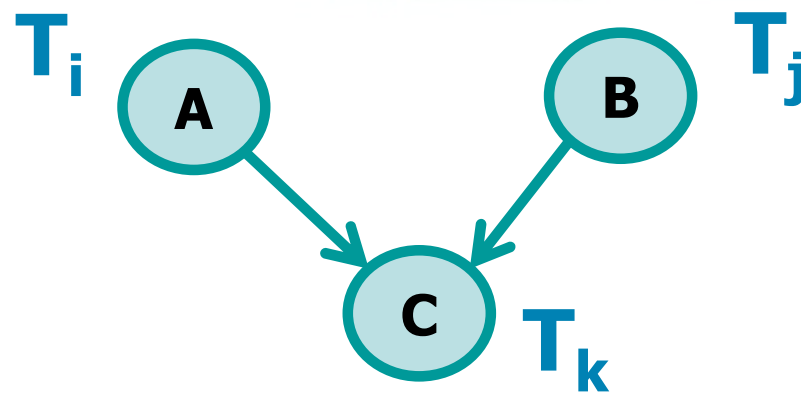
```
B;  
signal (S);
```

 $T_j$



## Pure synchronization : Example 3

❖ Other possible solution involving 3 P/T



```
init (S, 0);
```

```
A;  
signal (S);
```

```
wait (S);  
wait (S);  
C;
```

```
B;  
signal (S);
```

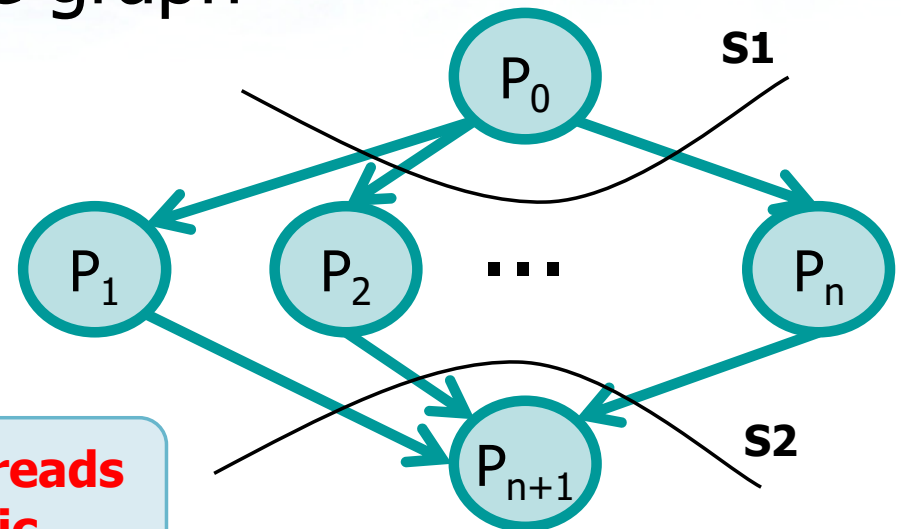
# Pure synchronization : Example 4

## ❖ Implement this precedence graph

cobegin-coend  
(concurrent begin-end)

```
init (S1, 0);
init (S2, 0);
```

**Note: These threads are not cyclic**



$P_0/T_0$

```
P0
for (i=1; i<=n; i++)
    signal (S1);
...
```

$P_i/T_i$

```
wait (S1);
Pi
signal (S2);
...
```

$P_{n+1}/T_{n+1}$

```
...
for (i=1; i<=n; i++)
    wait (S2);
Pn+1
```

# Errors using semaphores: Example 1

❖ Just **a single** thread is **incorrect**

```
init (S, 1);
```

$T_1$

```
while (TRUE) {  
    ...  
    signal (S); !!  
    CS1  
    wait (S); !!  
    ...  
}
```

$T_2$

```
while (TRUE) {  
    ...  
    wait (S);  
    CS2  
    signal (S);  
    ...  
}
```

$T_3$

```
while (TRUE) {  
    ...  
    wait (S);  
    CS3  
    signal (S);  
    ...  
}
```

Enters its CS and makes possible that the two other threads enter their CSs

## Errors using semaphores: Example 2

❖ Just **a single** thread is **incorrect**

```
init (S, 1);
```

$T_1$

```
while (TRUE) {  
    ...  
    wait (S);  
    CS1  
    wait (S);  !!  
    ...  
}
```

$T_2$

```
while (TRUE) {  
    ...  
    wait (S);  
    CS2  
    signal (S);  
    ...  
}
```

$T_3$

```
while (TRUE) {  
    ...  
    wait (S);  
    CS3  
    signal (S);  
    ...  
}
```

When the second wait is executed all thread are in deadlock

## Errors using semaphores: Example 3

❖ Just **a single** thread is **incorrect**

```
init (S, 1);
```

$T_1$

```
while (TRUE) {  
    ...  
    signal(S); !!  
    CS1  
    signal(S);  
    ...  
}
```

$T_2$

```
while (TRUE) {  
    ...  
    wait (S);  
    CS2  
    signal (S);  
    ...  
}
```

$T_3$

```
while (TRUE) {  
    ...  
    wait (S);  
    CS3  
    signal (S);  
    ...  
}
```

When the first signal is executed, two threads can enter their CSs.  
When the second signal is executed, all threads can enter their CSs.



## Errors using semaphores: Example 4

❖ Just **a single** thread is **incorrect**

```
init (S, 1);
```

$T_1$

```
while (TRUE) {  
    ...  
    wait (S);  
    CS1  
    !! no signal (S)  
    ...  
}
```

$T_2$

```
while (TRUE) {  
    ...  
    wait (S);  
    CS2  
    signal (S);  
    ...  
}
```

$T_3$

```
while (TRUE) {  
    ...  
    wait (S);  
    CS3  
    signal (S);  
    ...  
}
```

After  $T_1$  exit its CS, all threads will be in deadlock

If  $T_3$  is fast, all threads can enter their CSs

## Errors using semaphores: Example 5

❖ Just **a single** thread is **incorrect**

```
init (S, 1);
```

$T_1$

```
while (TRUE) {  
    ...  
    !! no wait(S);  
    CS1  
    signal (S);  
    ...  
}
```

$T_2$

```
while (TRUE) {  
    ...  
    wait (S);  
    CS2  
    signal (S);  
    ...  
}
```

$T_3$

```
while (TRUE) {  
    ...  
    wait (S);  
    CS3  
    signal (S);  
    ...  
}
```

If  $T_1$  is fast (i.e., it does two loops in the while cycle), all threads can enter their CSs

## Errors using semaphores: Example 6

Acquiring two  
resources

```
init (S, 1);  
init (Q, 1);
```

$T_1$

```
while (TRUE) {  
    ...  
    wait (S);  
    ... Use S  
    wait (Q);  
    ... Use S and Q  
    signal (Q);  
    signal (S);  
    ...  
}
```

Access to pen-drive, then to HD

$T_2$

```
while (TRUE) {  
    ...  
    wait (Q);  
    ... Use Q  
    wait (S);  
    ... Use Q and S  
    signal (S);  
    signal (Q);  
    ...  
}
```

Access to HD, then to pen-drive

## Exercise

- ❖ Given the code of these three threads
- Which is the possible execution order?

```
init (S1, 1);  
init (S2, 0);
```

...

$T_1$

```
while (1) {  
    wait (S1);  
     $T_1$  code  
    signal (S2);  
}  
...
```

...

$T_2$

```
while (1) {  
    wait (S2);  
     $T_2$  code  
    signal (S2);  
}  
...
```

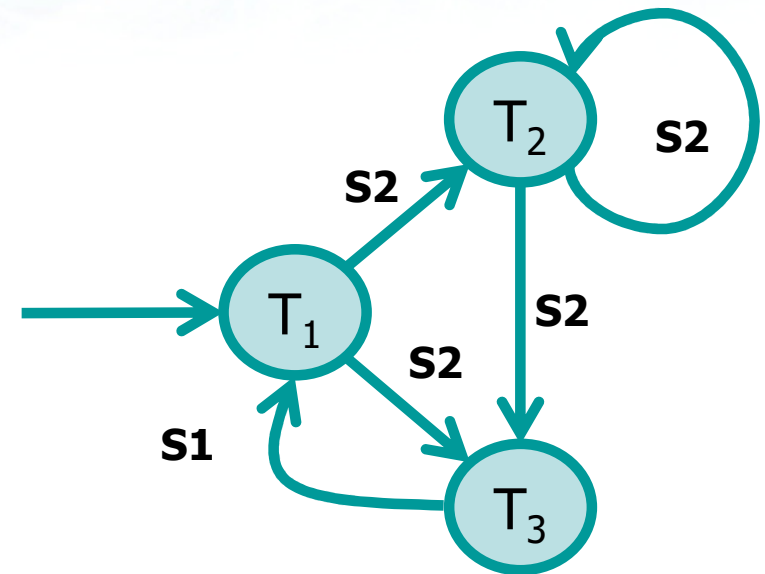
...

$T_3$

```
while (1) {  
    wait (S2);  
     $T_3$  code  
    signal (S1);  
}  
...
```

## Solution

❖ It is a peculiar synchronization example !!



```
init (S1, 1);  
init (S2, 0);
```

...

**T<sub>1</sub>**

```
while (1) {  
    wait (S1);  
    T1 code  
    signal (S2);  
}  
...
```

...

**T<sub>2</sub>**

```
while (1) {  
    wait (S2);  
    T2 code  
    signal (S2);  
}  
...
```

...

**T<sub>3</sub>**

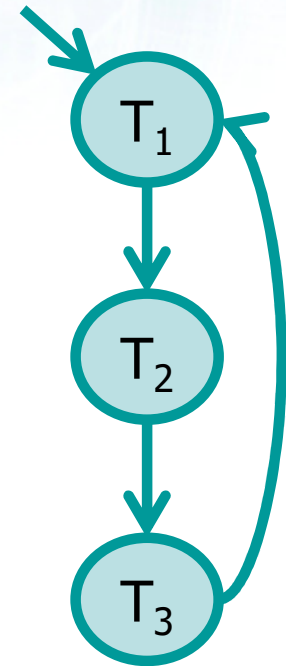
```
while (1) {  
    wait (S2);  
    T3 code  
    signal (S1);  
}  
...
```



## Exercise

- ❖ Implement this precedence graph using semaphores
  - **All T/P must be *cyclic***

This way they don't have to be instantiated several times



## Solution

❖ Implement this precedence graph using semaphores

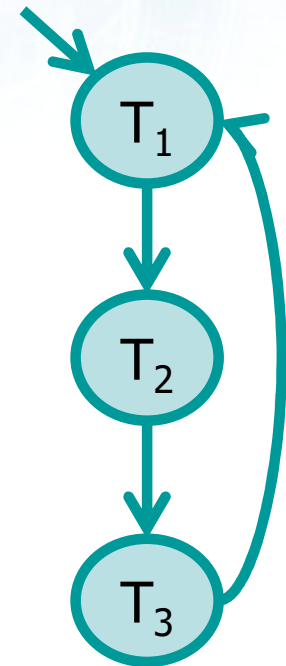
➤ All T/P must be **cyclic**

```
init (S1, 1);  
init (S2, 0);  
init (S3, 0);
```

...  $T_1$   
`while (1) {  
 wait (S1);  
  $T_1$  code  
 signal (S2);  
}`  
...

...  $T_2$   
`while (1) {  
 wait (S2);  
  $T_2$  code  
 signal (S3);  
}`  
...

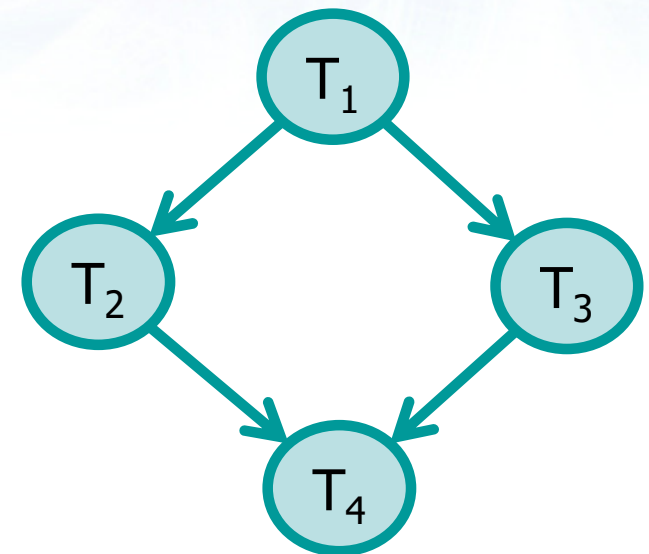
...  $T_3$   
`while (1) {  
 wait (S3);  
  $T_3$  code  
 signal (S1);  
}`  
...



## Exercise

❖ Implement this precedence graph using semaphores

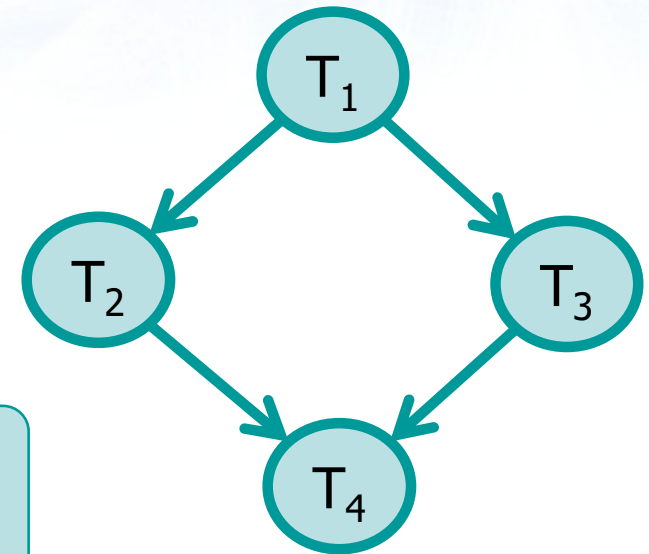
➤ T/P are not **cyclic**



## Solution

❖ Implement this precedence graph using semaphores

➤ T/P are not **cyclic**



```
init (S1, 0);  
init (S2, 0);
```

```
...  
wait (S1);  
T2 code  
signal (S2);  
...
```

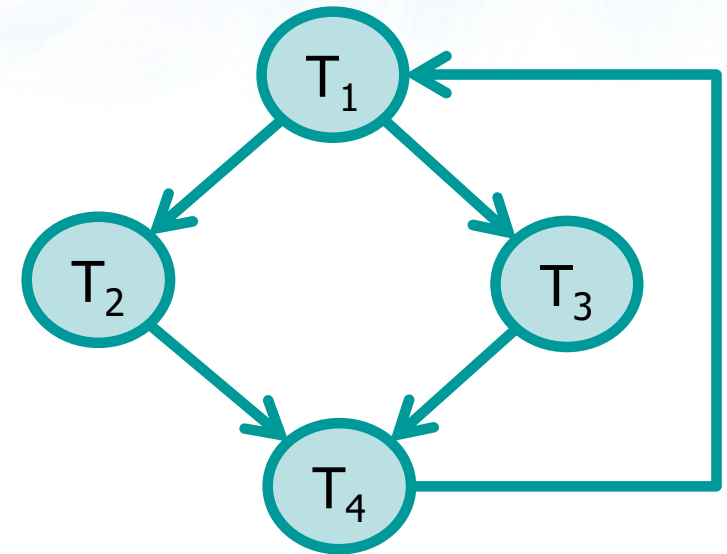
```
T1 code  
signal (S1);  
signal (S1);  
...
```

```
...  
wait (S1);  
T3 code  
signal (S2);  
...
```

```
...  
wait (S2);  
wait (S2);  
T4 code
```

## Exercise

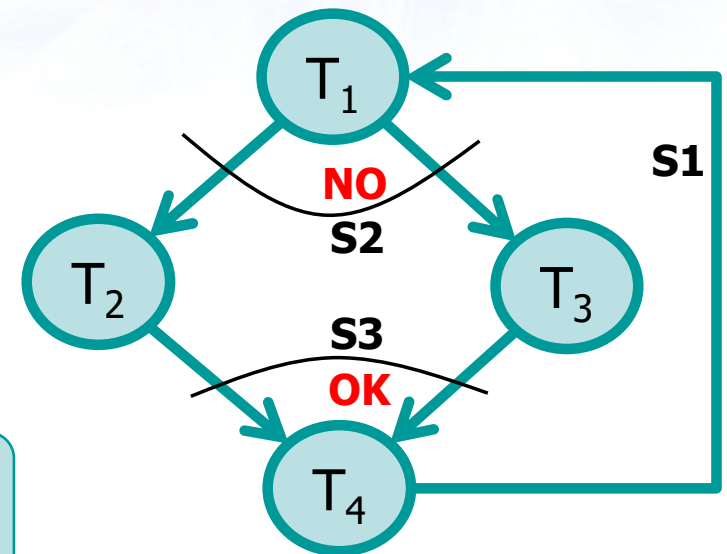
- ❖ Implement this precedence graph using semaphores
  - **All T/P must be cyclic**



# Erroneous solution

❖ Implement this precedence graph using semaphores

➤ All T/P must be **cyclic**



```

init (S1, 1);
init (S2, 0);
init (S3, 0);
  
```

```

while (1) {
    wait (S2);
    T2 code
    signal (S3);
}
  
```

```

while (1) {
    wait (S1);
    T1 code
    signal (S2);
    signal (S2);
}
  
```

```

while (1) {
    wait (S2);
    T3 code
    signal (S3);
}
  
```

```

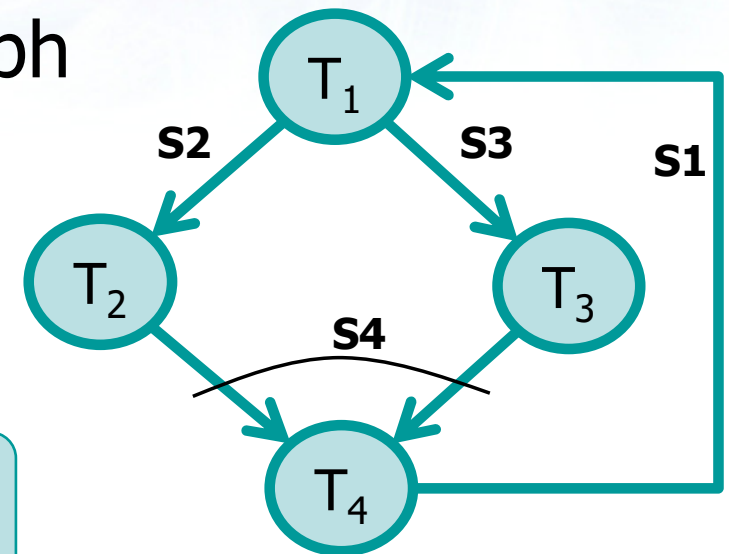
while (1) {
    wait (S3);
    wait (S3);
    T4 code
    signal (S1);
}
  
```



## Solution

❖ Implement this precedence graph using semaphores

➤ All T/P must be **cyclic**



```

init (S1, 1);
init (S2, 0);
init (S3, 0);
init (S4, 0);
  
```

```

while (1) {
    wait (S2);
    T2 code
    signal (S4);
}
  
```

```

while (1) {
    wait (S1);
    T1 code
    signal (S2);
    signal (S3);
}
  
```

```

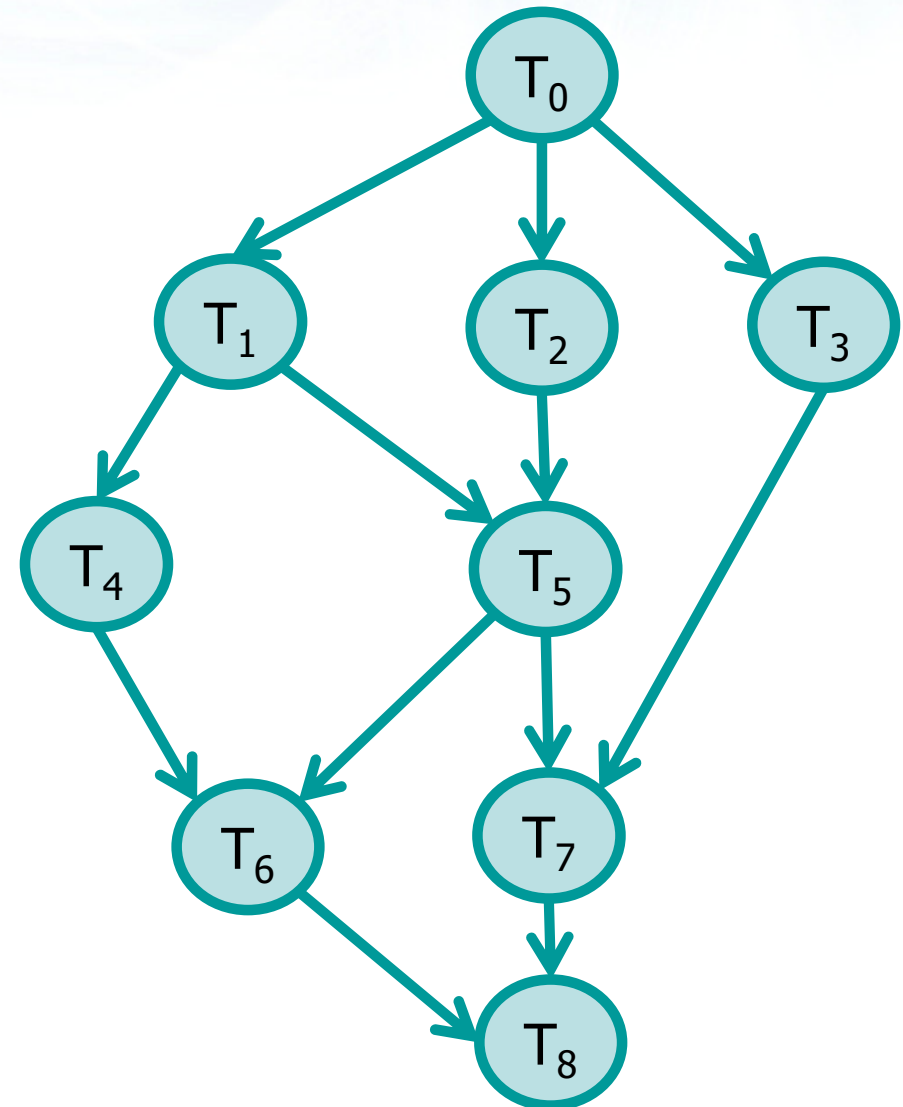
while (1) {
    wait (S3);
    T3 code
    signal (S4);
}
  
```

```

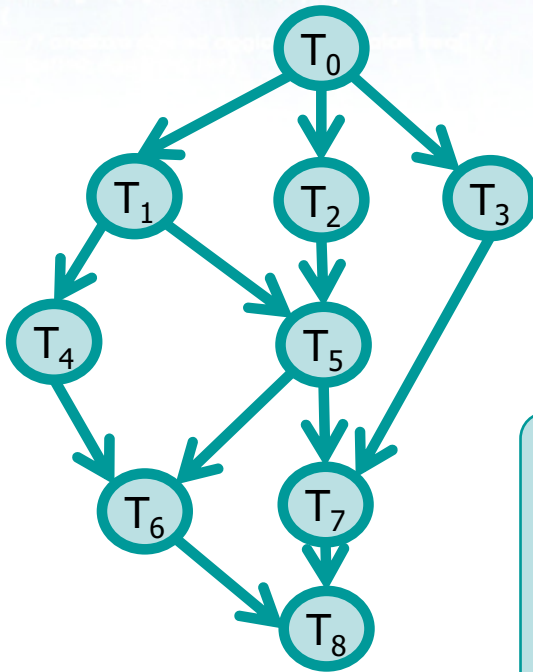
while (1) {
    wait (S4);
    wait (S4);
    T4 code
    signal (S1);
}
  
```

## Exercise

- ❖ Implement this precedence graph using semaphores
  - T/P are **not cyclic**



# Erroneous solution



$T_0$   
 $T_0$  code  
 signal (S1);  
 signal (S1);  
 signal (S1);

$T_1$   
 wait (S1);  
 $T_1$  code  
 signal (S2);  
 signal (S2);

$T_2$   
 wait (S1);  
 $T_2$  code  
 signal (S2);

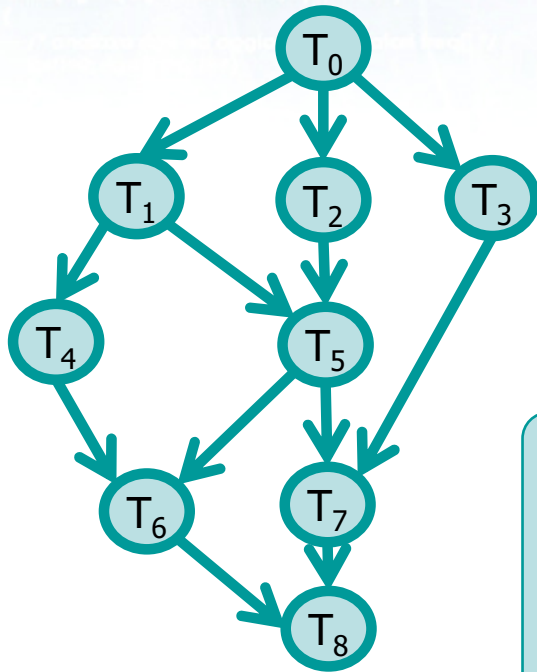
$T_3$   
 wait (S1);  
 $T_3$  code  
 ...

init (S1, 0);  
 init (S2, 0);  
 init (S3, 0);  
 ...

$T_4$   
 wait (S2);  
 $T_4$  code  
 ...

$T_5$   
 wait (S2);  
 wait (S2);  
 $T_5$  code  
 ...

## Solution



**T<sub>0</sub>**  
**T<sub>0</sub> code**  
**signal (S1);**  
**signal (S2);**  
**signal (S3);**

**T<sub>1</sub>**  
**wait (S1);**  
**T<sub>1</sub> code**  
**signal (S4);**  
**signal (S5);**

**T<sub>2</sub>**  
**wait (S2);**  
**T<sub>2</sub> code**  
**signal (S5);**

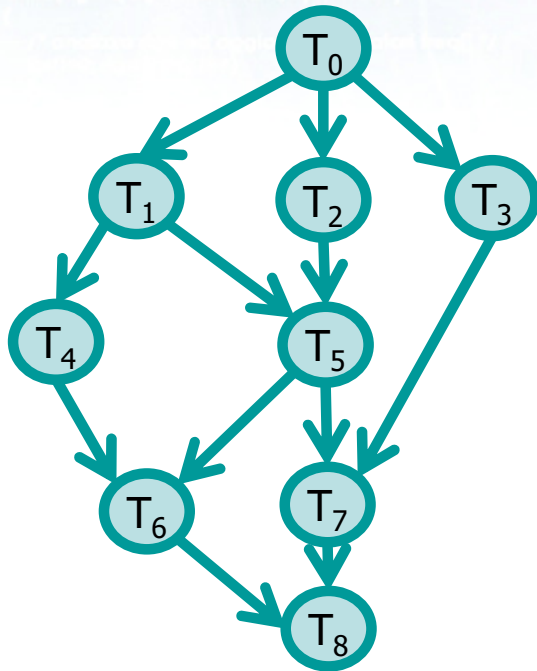
**T<sub>3</sub>**  
**wait (S3);**  
**T<sub>3</sub> code**  
**signal (S7);**

**init (S1, 0);**  
**init (S2, 0);**  
**init (S3, 0);**  
**...**

**T<sub>4</sub>**  
**wait (S4);**  
**T<sub>4</sub> code**  
**signal (S6);**

**T<sub>5</sub>**  
**wait (S5);**  
**wait (S5);**  
**T<sub>5</sub> code**  
**signal (S6);**  
**signal (S7);**

## Solution



$T_6$   
`wait (S6);`  
`wait (S6);`  
 $T_6$  code  
`signal (S8);`

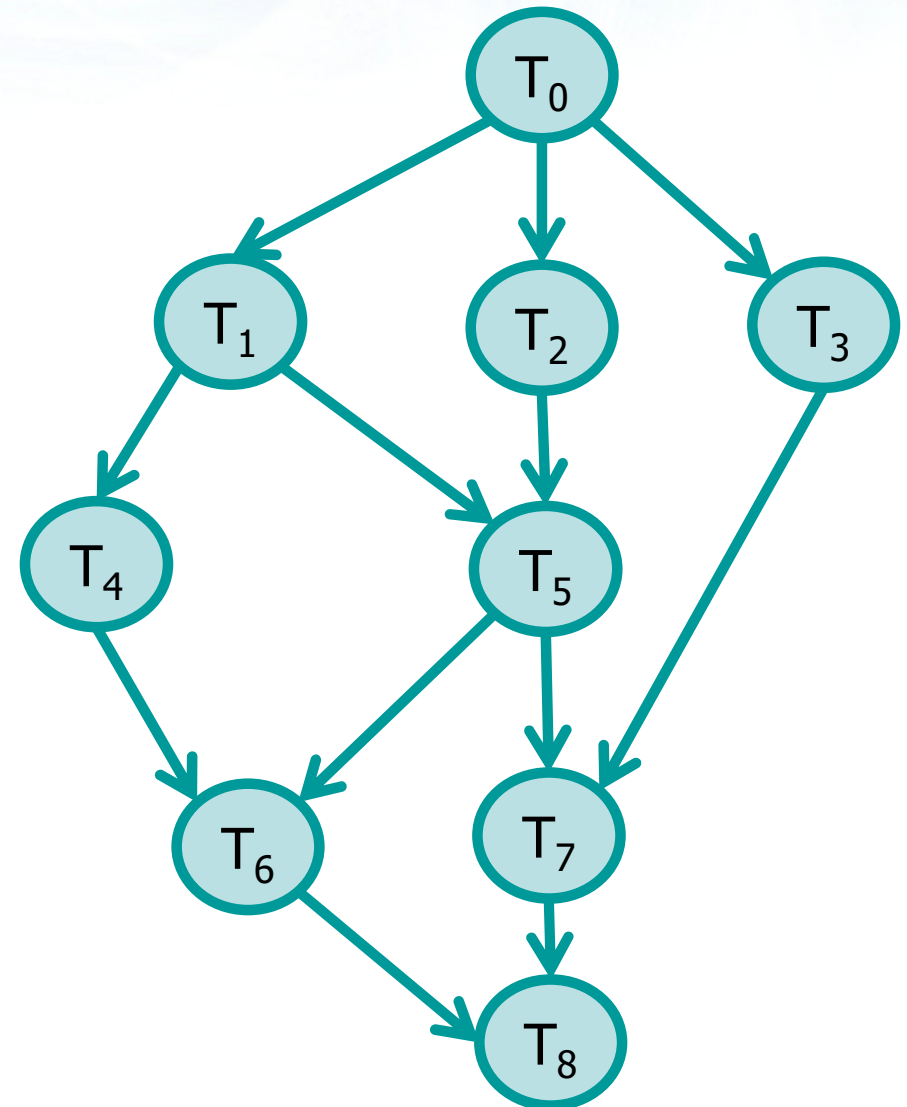
$T_7$   
`wait (S7);`  
`wait (S7);`  
 $T_7$  code  
`signal (S8);`

$T_8$   
`wait (S8);`  
`wait (S8);`  
 $T_8$  code

This solution is correct, but the number of semaphores is **not minimal**.

## Exercise

- ❖ Implement this precedence graph using semaphores
  - Version A: T/P are **not cyclic**, but use the **minimum number of semaphores**
  - Version B: T/P are **cyclic**





# Implementation of a semaphore

- ❖ Semaphores must be implemented without "active" **busy waiting** (**spin-lock**)
- ❖ We define a semaphore as a C structure with
  - A counter
  - A list (queue) of processes

```
typedef struct semaphore_s {  
    int cnt;                // Number of processes  
    process_t *head;        // List of processes  
} semaphore_t;
```

# Implementation of a semaphore

Wait only if  
 $cnt < 0$

```
init (semaphore_t *S, int k) {  
    alloc S;  
    S->cnt = k;  
    S->head = NULL;  
}
```

Init with  
 $k \geq 0$

```
wait (semaphore_t *S) {  
    S->cnt--;  
    if (S->cnt < 0) {  
        push P to S->head;  
        block P;  
    }  
}
```

cnt can assume  
negative values

```
signal (semaphore_t *S) {  
    S->cnt++;  
    if (S->cnt <= 0) {  
        pop P from S->head;  
        wakeup P;  
    }  
}
```

There are  
queued P only  
if  $cnt \leq 0$

```
destroy (semaphore_t *S) {  
    while (S->cnt <= 0) {  
        free P from S->head;  
        S->cnt++;  
    }  
}
```

All remaining P were  
extracted from the queue

# Implementation of a semaphore

- ❖ The real implementation allows a semaphore to have negative values
  - Its absolute value indicates the number of processes in the queue of the semaphore
- ❖ The queue
  - Can be implemented with a pointer in the Process Control Block (PCB) of the processes
  - It uses the policies defined by the scheduler (e.g., FIFO)

# Real implementations

## ❖ There are several semaphores implementations

### ➤ Semaphores by means of a pipe

### ➤ POSIX Pthread

- Condition variables
- **Semaphores**
  - The most important
- **Mutex** (for mutual exclusion)

System call:  
`pthread_cond_init`  
`pthread_cond_wait`  
`pthread_cond_signal`  
`pthread_cond_broadcast`  
`pthread_cond_destroy`

### ➤ Linux semaphores

System call:  
`semget`, `semop`, `semctl`  
(in `sys/sem.h`) they are  
complex to use

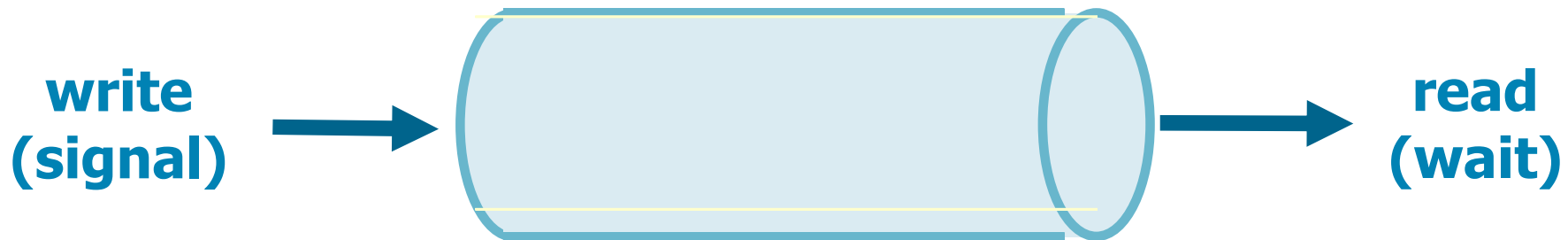
## ❖ Notice that semaphores are

- Global shared objects (see **`sem_init`**)
- They are allocated by a thread, but they are kernel objects

## Semaphore by means of a pipe

### ❖ Given a pipe

- The counter of a semaphore is achieved by means of **tokens**
- **Signal** implemented using the **write** system call to write a token on the pipe (non-blocking)
- **Wait** implemented using the **read** system call to read a token from the pipe (blocking)



## semaphoreInit (s)

```
#include <unistd.h>

void semaphoreInit (int *S, int k) {
    char ctr = 'X';
    int i;
    if (pipe (S) == -1) {
        printf ("Error"); exit (-1);
    }
    for(i=0; i<k; i++)
        if (write(S[1], &ctr, sizeof(char)) != 1) {
            printf ("Error"); exit (-1);
        }
    return;
}
```

Writes k characters, i.e., initializes the semaphore counter to k

### ❖ Semaphore initialization

- The variable S must be defined as a global variable
  - int S[2];
  - int \*S = malloc (2 \* sizeof (char));



## semaphoreSignal (s)

```
#include <unistd.h>

void semaphoreSignal (int *S) {
    char ctr = 'X';
    if (write(S[1], &ctr, sizeof(char)) != 1) {
        printf ("Error");
        exit (-1);
    }
    return;
}
```

Writes a single character,  
i.e., increments the  
semaphore counter k

- ❖ Writes a character (any) on a pipe
  - Suppose the number of writes (signals) before a read (wait) not exceed the dimension of the pipe

## semaphoreWait (s)

```
#include <unistd.h>

void semaphoreWait (int *S) {
    char ctr;
    if (read (S[0], &ctr, sizeof(char)) != 1) {
        printf ("Error");
        exit (-1);
    }
    return;
}
```

If the pipe is empty,  
read() waits

- ❖ Reads a character from a pipe (read is blocking )

# Example

Use of a pipe as a synchronization semaphore between P parent and P child

```
int main() {
    int S[2];
    pid_t pid;
    semaphoreInit (S, 0);
    pid = fork();
    // Check for correctness
    if (pid == 0) {                                // child
        semaphoreWait (S);
        printf("Wait done.\n");
    } else {                                       // parent
        printf("Sleep 3s.\n");
        sleep (3);
        semaphoreSignal (S);
        printf("Signal done.\n");
    }
    return 0;
}
```

# POSIX semaphores

## ❖ There are two types of POSIX semaphores

### ➤ Unnamed semaphores

- Implemented in the internal memory of the process
- They are used for the synchronization of threads within the same process

### ➤ Named semaphores

- Implemented using shared memory, they are "process-shared semaphore"
- They are generally used in the synchronization between processes
  - The name (sem\_open) allows their use in different processes

# POSIX semaphores

- ❖ We will analyze only **unnamed** semaphores
  - The implementation is independent from the OS, and it is defined in the semaphore.h header file
  - Insert in the .c file
    - `#include <semaphore.h>`
- ❖ The semaphore is a variable of type **sem\_t**
  - A semaphore can be allocated statically or dynamically
    - `sem_t *sem1, *sem2, ...;`
- ❖ Functions defined on semaphores
  - Are named **sem\_\***
  - Return -1 on error

System call:  
sem\_init  
sem\_wait  
sem\_trywait  
sem\_post  
sem\_getvalue  
sem\_destroy

## sem\_init ()

```
int sem_init (  
    sem_t *sem,  
    int pshared,  
    unsigned int value  
);
```

- ❖ Initializes the semaphore counter at value **value**
- ❖ The **pshared** value identifies the type of semaphore
  - If equal to 0, the semaphore is local to the **threads of current process**
  - Otherwise, the semaphore can be **shared between different processes** (parent that initializes the semaphore and its children)

Linux does not currently support shared semaphores



## sem\_wait ()

```
int sem_wait (  
    sem_t *sem  
);
```

### ❖ Standard wait

- If the semaphore is equal to 0, it blocks the caller until it can decrease the value of the semaphore

## sem\_trywait ()

```
int sem_trywait (  
    sem_t *sem  
);
```

### ❖ Non-blocking wait

- If the semaphore counter has a value greater than 0, perform the decrement, and returns 0
- If the semaphore is equal to 0, returns -1 (instead of blocking the caller as **sem\_wait** does)

## sem\_post ()

```
int sem_post (  
    sem_t *sem  
);
```

### ❖ Standard signal

- Increments the semaphore counter, or wakes up a blocked thread if present

## sem\_getvalue ()

```
int sem_getvalue (  
    sem_t *sem,  
    int *valP  
);
```

**Better not to use this function.** From Linux manual: "The value of the semaphore may already have changed by the time sem\_getvalue() returns."

❖ Allows obtaining the value of the semaphore counter

- The value is assigned to \*valP
- If there are waiting threads
  - 0 is assigned to \*valP (Linux)
  - or a negative number whose absolute value is equal to the number of processes waiting (POSIX)

## sem\_destroy ()

```
int sem_destroy (  
    sem_t *sem  
);
```

- ❖ Destroys the semaphore at the address pointed by sem
  - Destroying a semaphore that other threads are currently blocked on produces undefined behavior (on error, -1 is returned)
  - Using a semaphore that has been destroyed produces undefined results, until the semaphore has been reinitialized

# Example

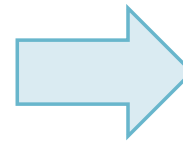
The use of the sem\_\*  
POSIX functions for  
synchronization

```
#include "semaphore.h"

sem_t sem;
sem_init (&sem, 0, 0);

... create threads ...

sem_destroy (&sem);
```



```
sem_wait (&sem);
... SC ...
sem_post (&sem);
```

Static semaphore

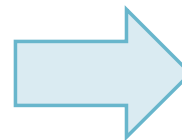
```
#include "semaphore.h"

sem_t *sem;
sem = (sem_t *)
      malloc(sizeof(sem_t));
sem_init (sem, 0, 0);

... create threads ...

sem_destroy (sem);
```

Dynamic  
semaphore



```
sem_wait (sem);
... SC ...
sem_post (sem);
```



## Pthread mutex

- ❖ Binary semaphores (mutex)
- ❖ A mutex is of type **pthread\_mutex\_t**
- ❖ System calls
  - pthread\_mutex\_init
  - pthread\_mutex\_lock
  - pthread\_mutex\_trylock
  - pthread\_mutex\_unlock
  - pthread\_mutex\_destroy

Alternative to sem\_xxxx primitives, mutex is less general than semaphores (i.e., they can assume only the two values 0 or 1)

## pthread\_mutex\_init ()

```
int pthread_mutex_init (  
    pthread_mutex_t *mutex,  
    const pthread_mutexattr_t *attr  
);
```

- ❖ Initializes the mutex referenced by `mutex` with attributes specified by `attr` (default=NULL)
- ❖ Return value
  - 0 on success
  - Error code otherwise

## pthread\_mutex\_lock ()

```
int pthread_mutex_lock (  
    pthread_mutex_t *mutex  
);
```

- ❖ Control the value of **mutex** and
  - Blocks the caller if the mutex is locked
  - Acquire the mutex lock if the mutex is unlocked
- ❖ Return value
  - 0 on success
  - Error code otherwise

## pthread\_mutex\_trylock ()

```
int pthread_mutex_trylock (  
    pthread_mutex_t *mutex  
);
```

- ❖ Similar to `pthread_mutex_lock`, but returns without blocking the caller if the mutex is locked
- ❖ Return value
  - 0 if the lock has been successfully acquired
  - **EBUSY** error if the mutex was already locked by another thread

# pthread\_mutex\_unlock ()

```
int pthread_mutex_unlock (  
    pthread_mutex_t *mutex  
);
```

- ❖ Release the **mutex** lock (typically at the end of a Critical Section)
- ❖ Return value
  - 0 on success
  - Error code otherwise

## pthread\_mutex\_destroy ()

```
int pthread_mutex_destroy (  
    pthread_mutex_t *mutex  
);
```

- ❖ Free **mutex** memory
- ❖ The mutex cannot be used any more
- ❖ Return value
  - 0 on success
  - Error code otherwise