

```
#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 )
```



Sincronizzazione

I semafori

Stefano Quer

Dipartimento di Automatica e Informatica

Politecnico di Torino

Introduzione

- ❖ Le soluzioni
 - Software sono complesse per il programmatore
 - Hardware sono difficili da realizzare per il progettista
- ❖ I sistemi operativi forniscono primitive dette **semafori** che
 - Risultano più semplici da utilizzare
 - Sono più efficienti
 - Non si basano su implementazioni con busy waiting e quindi non sprecano risorse
 - Permettono di risolvere problemi più generali della sola ME

Introdotti da
Dijkstra nel 1965

Definizione

Oggetto comune, di
tipo intero, che funge
da contatore protetto

- ❖ Un semaforo **S** è
 - Una variabile intera condivisa
 - Protetta dal sistema operativo
 - Utilizzabile per inviare e ricevere segnali
- ❖ Le operazioni su **S** sono sempre eseguite in maniera atomica
 - L'atomicità è garantita dal sistema operativo
 - È impossibile per due processi eseguire operazioni contemporanee sullo stesso semaforo
 - Le istruzioni che manipolano un semaforo non sono mai eseguite in maniera inter-allacciata

Primitive di manipolazione

❖ Operazioni “standard” dato un semaforo S

➤ init (S, k)

- Definisce e inizializza il semaforo S al valore k

➤ wait (S)

sleep, down, P

- Permette (nella sezione di ingresso) di ottenere l'accesso della SC protetta dal semaforo S

➤ signal (S)

wakeup, up, V

- Permette (nella sezione di uscita) di uscire dalla SC protetta dal semaforo S

➤ destroy (S)

- Cancella (libera/free) il semaforo S

Non sono le “wait” e “signal” viste in passato

Primitive di manipolazione

❖ `init (S, k)`

K è un valore intero

➤ Definisce e inizializza il semaforo S al valore k

➤ Esistono due tipi di semafori

Noti come "mutex lock"
(mutex = MUTual EXclusion)

- Semafori binari

- Il valore di k e quello del semaforo in un istante qualsiasi è un intero uguale a 0 oppure a 1

- Semafori con conteggio

- Il valore di k e quello del semaforo in un istante qualsiasi è un intero nell'intervallo $[0, k]$ con $k > 1$

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

Implementazione
logica (di principio)

Operazione
atomica

Primitive di manipolazione

❖ wait (S)

- Se il valore di S è negativo o nullo **blocca** il processo chiamante (la risorsa non è disponibile)
 - Se S è negativo il suo valore assoluto $|S|$ indica il numero di P (o T) in attesa
- In ogni caso decrementa il valore di S

Nella versione logica S non diventa mai negativo

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

Operazione
atomica

Implementazione
logica (di principio)

Le implementazioni reali
non utilizzano busy waiting

Primitive di manipolazione

❖ wait (S)

- Originariamente era denominata P() dall'olandese "probeer te verlagen", i.e., "try to decrease"
- Da **non** confondere con la system call **wait** utilizzata per attendere un processo figlio

Nella versione logica S non diventa mai negativo

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

operazione
atomica

Implementazione
logica (di principio)

Le implementazioni reali
non utilizzano busy waiting

Primitive di manipolazione

❖ signal (S)

- Incrementa la variabile semaforica
 - Se S era negativo o nullo un qualche P (o T) risultava essere bloccato e ora potrà accedere
- Originariamente denominata V(), dall'olandese "verhogen", i.e., "to increment"
- Da **non** confondere con la system call **signal** utilizzata per istanziare un gestore di segnali

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

Implementazione
logica (di principio)

Operazione atomica
(registro=s;registro++;s=registro;)

Primitive di manipolazione

❖ destroy (S)

- Rilascia la memoria occupata dal semaforo S
 - Le implementazioni reali di un semaforo richiedono molto di più di una semplice variabile globale per definire un semaforo
- Presente nelle implementazioni reali, spesso non utilizzata negli esempi

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

Implementazione
logica (di principio)

Mutua esclusione con un semaforo

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

```
while (TRUE) {  
    wait (S);  
    SC  
    signal (S);  
    sezione non critica  
}
```

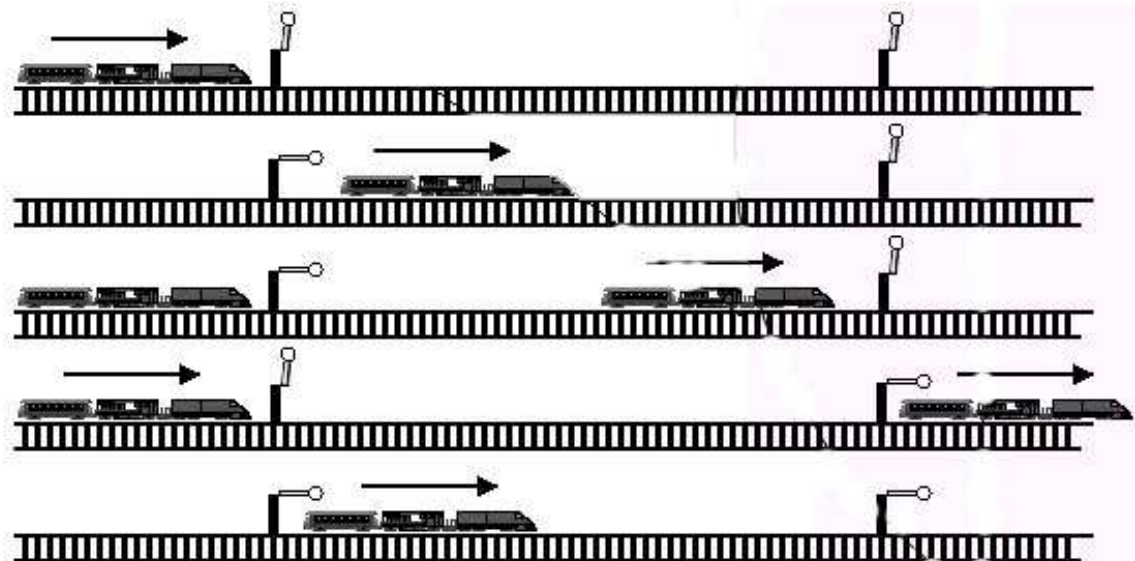
P_i / T_i

```
while (TRUE) {  
    wait (S);  
    SC  
    signal (S);  
    sezione non critica  
}
```

P_j / T_j

Ricordare:

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



Semaforo binario con N Thread

```
init (S, 1);
...
wait (S);
SC di Pi
signal (S);
```

P ₁ /T ₁	P ₂ /T ₂	P ₃ /T ₃	S	queue
			1	
wait			0	
SC	wait		0	P ₂ /T ₂
	blocked	wait	0	P ₂ /T ₂ , P ₃ /T ₃
		blocked	0	
signal			1	P ₂ /T ₂ , P ₃ /T ₃
	SC		0	P ₃ /T ₃
	signal		1	
		SC	0	
		signal	1	

Al più 1 P/T alla volta nella SC

Semaforo con conteggio con N Thread

```
init (S, 2);
...
wait (S);
SC di Pi
signal (S);
```

P ₁ /T ₁	P ₂ /T ₂	P ₃ /T ₃	S	queue
			2	
wait			1	
SC	wait		0	
	SC	wait	0	P ₃ /T ₃
		blocked	0	
signal			1	
		SC	0	
	signal		1	
		signal	2	

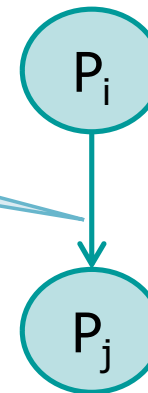
Al più 2 P/T alla volta nella SC

Uso dei semafori: Esempio 1

- ❖ Ottenere uno specifico ordine di esecuzione
 - P_i viene eseguito prima di P_j
 - P_j viene eseguito solo una volta terminato P_i

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

Arco di precedenza



```
SC di  $P_i$   
signal (S);
```

P_i / T_i

```
wait (S);  
SC di  $P_j$ 
```

P_j / T_j

Uso dei semafori: Esempio 2

- ❖ Sincronizzare due processi P_i e P_j in modo che
 - P_i attenda P_j in un preciso punto
 - P_j attenda P_i in un preciso punto


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

P_i / T_i

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

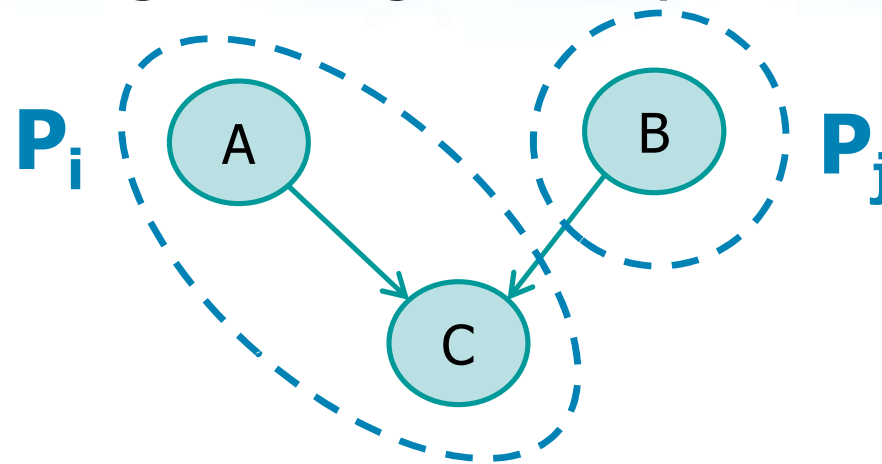
P_j / T_j

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



Uso dei semafori: Esempio 3

- ❖ Ottenere il seguente grafo di precedenza con **2** processi



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

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

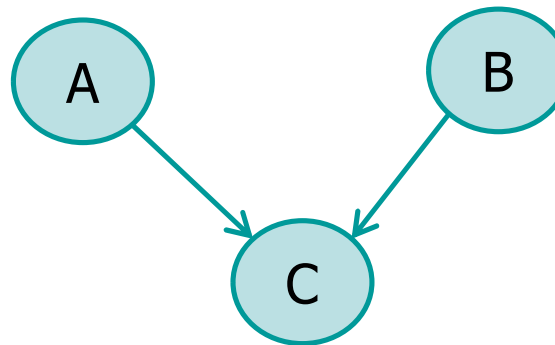
 P_i / T_i

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

 P_j / T_j

Uso dei semafori: Esempio 3

- ❖ Ottenere il seguente grafo di precedenza con **3** processi



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

A
`signal (S);`

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

B
`signal (S);`

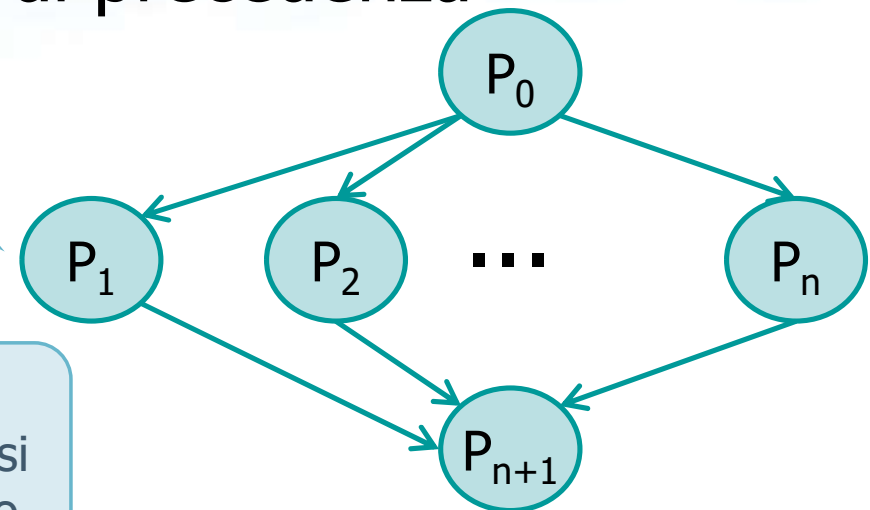
Uso dei semafori: Esempio 4

❖ Ottenere il seguente grafo di precedenza

Costrutto cobegin-coend
(begin-end concorrente)

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

Osservazione:
nessuno dei processi
concorrenti è ciclico



P_0 / T_0

```
...
i=1
while (i<=n) {
    signal (S1);
    i++;
}
...
```

P_i / T_i

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

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

```
...
i=1;
while (i<=n) {
    wait (S2);
    i++;
}
...
```

Errori nell'uso dei semafori: Esempio 1

Solo un P (tra N)
in SC

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

P_i / T_i

```
while (TRUE) {  
    ...  
    signal (S);  
    SC1  
    wait (S);  
    ...  
}
```

Entra nella SC e fa
entrare anche altri 2
processi

P_i / T_i

```
while (TRUE) {  
    ...  
    wait (S);  
    SC2  
    wait (S);  
    ...  
}
```

La wait in "eccesso"
blocca tutti i processi

P_i / T_i

```
while (TRUE) {  
    ...  
    signal (S);  
    SC3  
    signal (S);  
    ...  
}
```

La signal in "eccesso"
fa entrare tutti

Errori nell'uso dei semafori: Esempio 2

Acquisizione di
due risorse

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

P_1 / T_1

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

Accede all'HD e poi al DVD

P_2 / T_2

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

Accede al DVD e poi all'HD

Esercizio

- ❖ Siano dati i semafori e i processi indicati
 - Quale **ordine** di esecuzione è possibile?

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

...

P_1

```
while (1) {  
    wait (S1);  
    SC  $P_1$   
    signal (S2);  
}  
...
```

...

P_2

```
while (1) {  
    wait (S2);  
    SC  $P_2$   
    signal (S2);  
}  
...
```

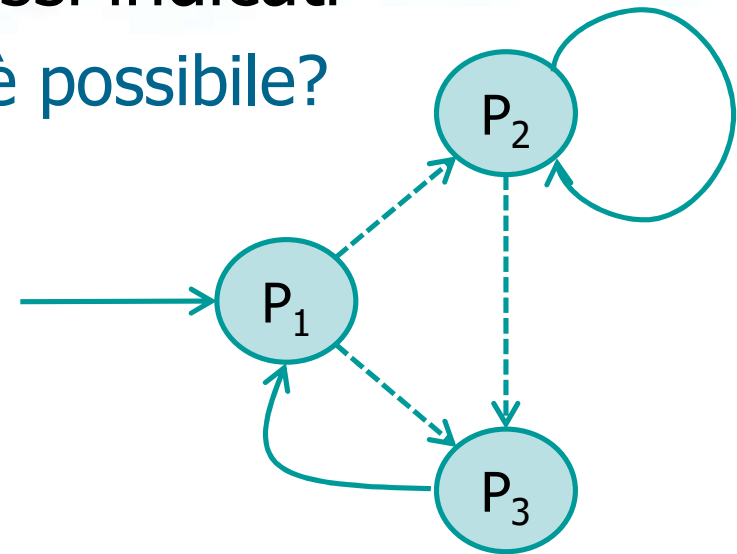
...

P_3

```
while (1) {  
    wait (S2);  
    SC  $P_3$   
    signal (S1);  
}  
...
```

Soluzione

- ❖ Siano dati i semafori e i processi indicati
 - Quale **ordine** di esecuzione è possibile?



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

...

P₁

```
while (1) {  
    wait (S1);  
    SC di P1  
    signal (S2);  
}  
...
```

...

P₂

```
while (1) {  
    wait (S2);  
    SC di P2  
    signal (S2);  
}  
...
```

...

P₃

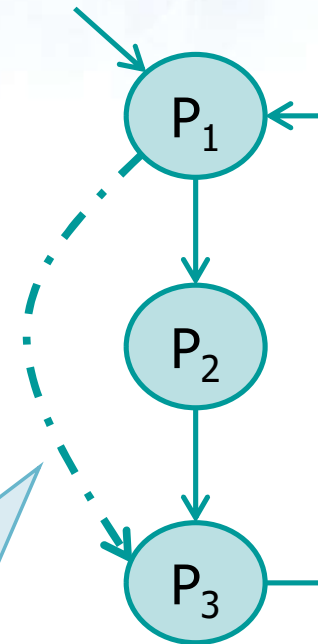
```
while (1) {  
    wait (S2);  
    SC di P3  
    signal (S1);  
}  
...
```

Esercizio

- ❖ Si realizzi mediante semafori il seguente grafo di precedenza
 - **Tutti** i processi devono essere **ciclici**

In questo modo non devono essere istanziati più volte

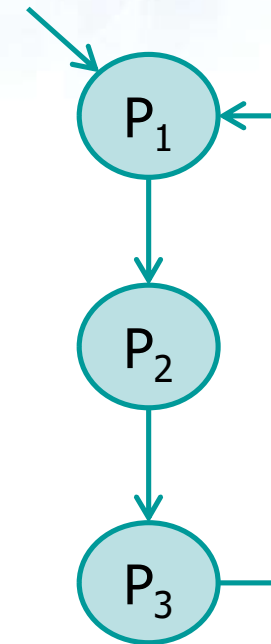
Per i grafi di precedenza vale la proprietà transitiva:
se P_2 segue P_1 e P_3 segue P_2
allora una eventuale precedenza da P_1 a P_3 può essere ignorata



Esercizio

- ❖ Si realizzi mediante semafori il seguente grafo di precedenza
 - **Tutti** i processi devono essere **ciclici**

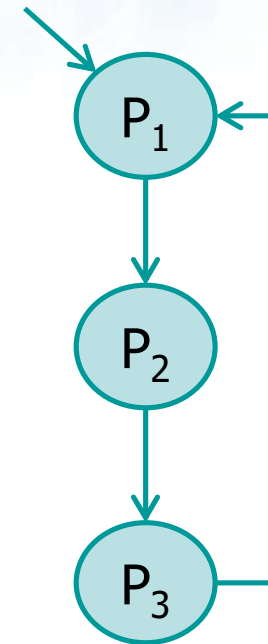
In questo modo non devono essere istanziati più volte



Soluzione

- ❖ Si realizzi mediante semafori il seguente grafo di precedenza
 - **Tutti** i processi devono essere **ciclici**

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



...

P₁

```
while (1) {  
    wait (S1);  
    SC di P1  
    signal (S2);  
}  
...
```

...

P₂

```
while (1) {  
    wait (S2);  
    SC di P2  
    signal (S3);  
}  
...
```

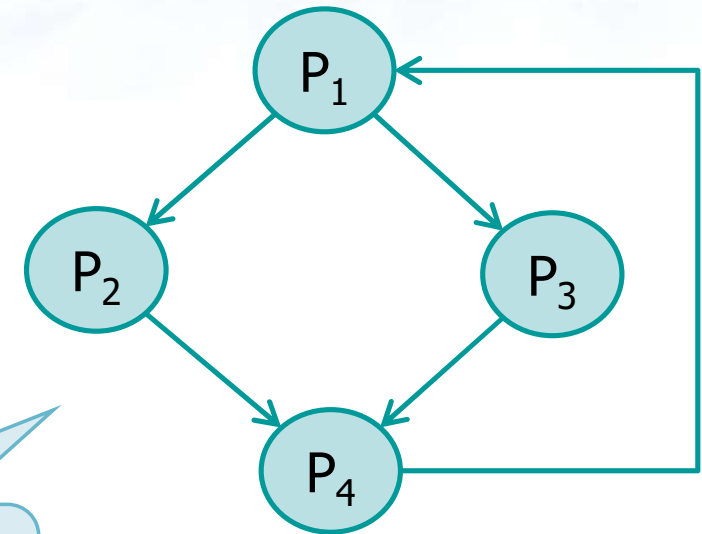
...

P₃

```
while (1) {  
    wait (S3);  
    SC di P3  
    signal (S1);  
}  
...
```

Esercizio

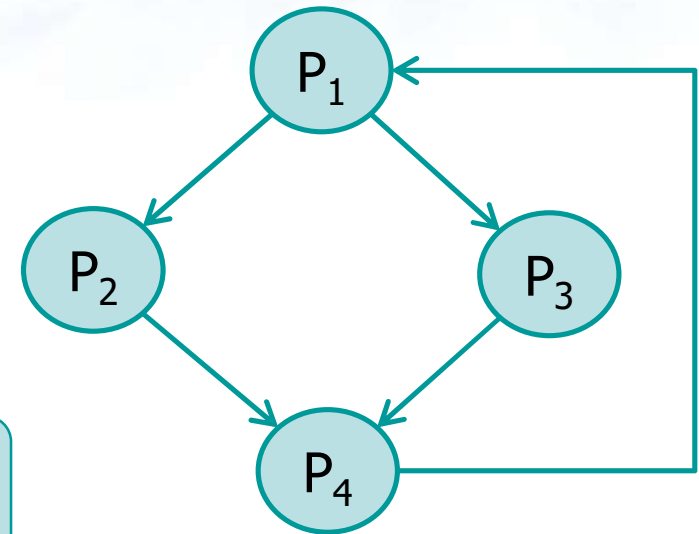
- ❖ Si realizzi mediante semafori il seguente grafo di precedenza
 - **Tutti** i processi devono essere **ciclici**



Vedere esempio 4
(pagina 17)

Soluzione **errata**

- ❖ Si realizzi mediante semafori il seguente grafo di precedenza
- **Tutti** i processi devono essere **ciclici**



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

```
while (1) { P2  
    wait (S2);  
    SC di P2  
    signal (S3);  
}
```

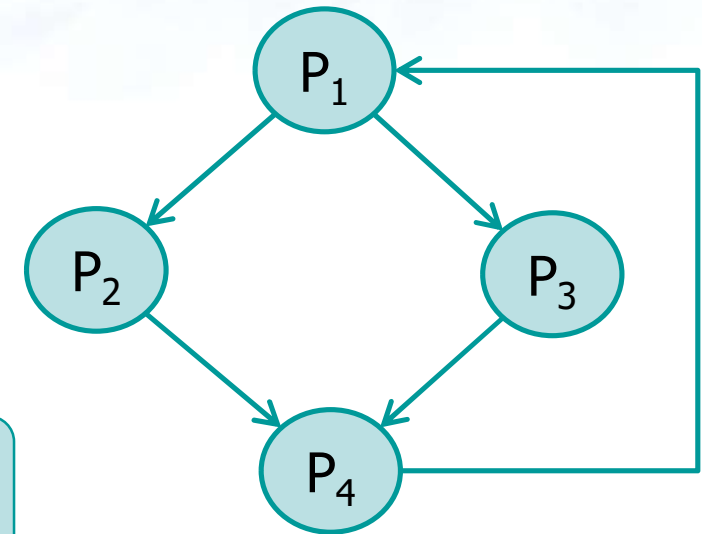
```
while (1) { P1  
    wait (S1);  
    SC di P1  
    signal (S2);  
    signal (S2);  
}
```

```
while (1) { P3  
    wait (S2);  
    SC di P3  
    signal (S3);  
}
```

```
while (1) { P4  
    wait (S3);  
    wait (S3);  
    SC di P4  
    signal (S1);  
}
```

Soluzione

- ❖ Si realizzi mediante semafori il seguente grafo di precedenza
 - Tutti i processi devono essere **ciclici**



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

```
while (1) {  
    wait (S1);  
    SC di P1  
    signal (S2);  
    signal (S3);  
}
```

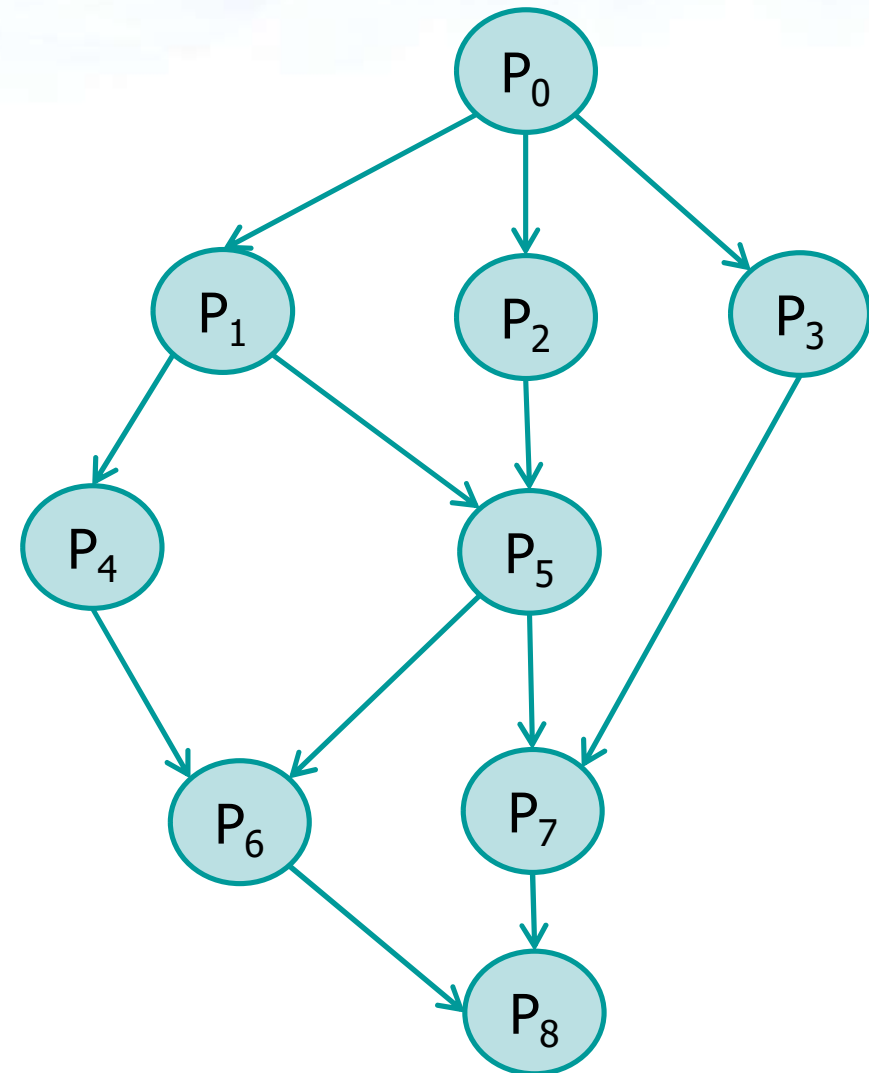
```
while (1) {  
    wait (S2);  
    SC di P2  
    signal (S4);  
}
```

```
while (1) {  
    wait (S3);  
    SC di P3  
    signal (S4);  
}
```

```
while (1) {  
    wait (S4);  
    wait (S4);  
    SC di P4  
    signal (S1);  
}
```

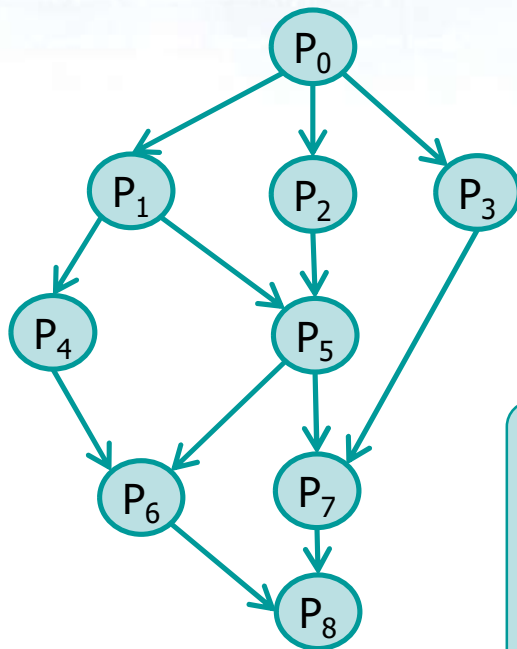
Esercizio

- ❖ Si realizzi mediante semafori il seguente grafo di precedenza
 - In processi **non** sono ciclici



Soluzione **errata**

Approccio con il
numero minimo di
semafori



P₀
SC
signal (S123) ;
signal (S123) ;
signal (S123) ;

P₁
wait (S123) ;
SC
signal (S45) ;
signal (S45) ;

P₂
wait (S123) ;
SC
signal (S45) ;

P₃
wait (S123) ;
SC
signal (S67) ;

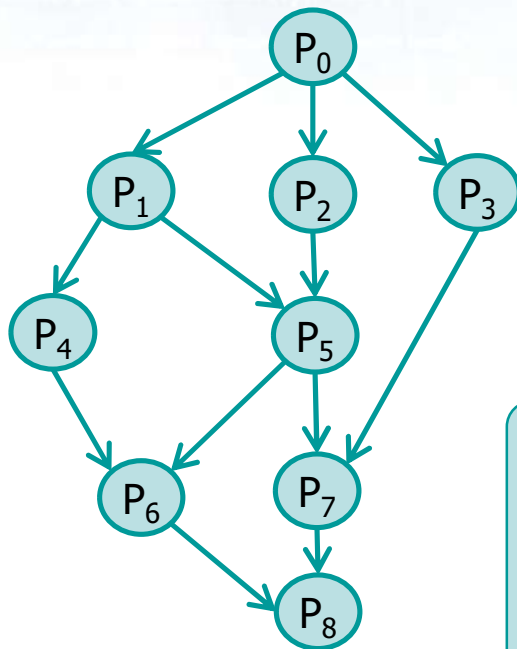
```
init (S123, 0) ;
init (S45, 0) ;
init (S67, 0) ;
...
```

P₄
wait (S45) ;
SC
signal (S67) ;

P₅
wait (S45) ;
wait (S45) ;
SC
signal (S67) ;

Soluzione

Approccio con il
numero minimo di
semafori



P₀
SC
signal (S123) ;
signal (S123) ;
signal (S123) ;

P₁
wait (S123) ;
SC
signal (S4) ;
signal (S5) ;

P₂
wait (S123) ;
SC
signal (S5) ;

P₃
wait (S123) ;
SC
signal (S7) ;

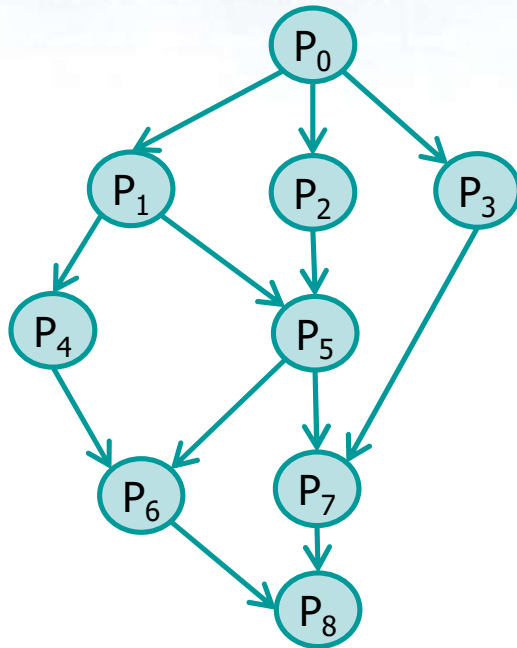
init (S123, 0) ;
init (S4, 0) ;
init (S5, 0) ;
...

P₄
wait (S4) ;
SC
signal (S6) ;

P₅
wait (S5) ;
wait (S5) ;
SC
signal (S6) ;
signal (S7) ;

...

Soluzione



P₆
wait(S6) ;
wait(S6) ;
SC
signal(S8) ;

P₇
wait(S7) ;
wait(S7) ;
SC
signal(S8) ;

P₈
wait(S8) ;
wait(S8) ;
SC

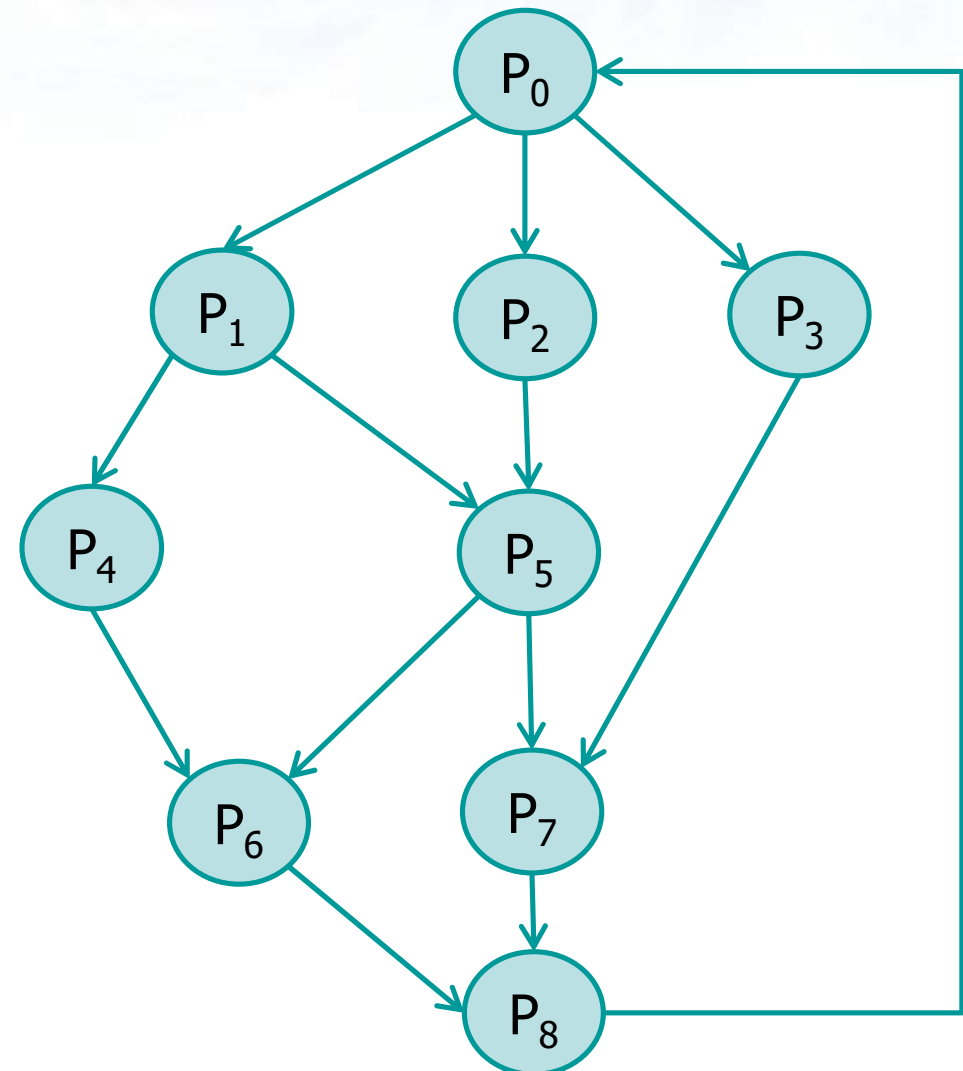
Se ognuno aspetta sul proprio semaforo la soluzione è corretta.

Però il numero di semafori può non essere minimo !!!

Esercizio

❖ Si realizzi mediante semafori il seguente grafo di precedenza

➤ I processi **sono** ciclici



Implementazione di un semaforo

- ❖ I semafori vanno implementati senza ricorrere all'attesa attiva **busy waiting** (**spin-lock**)
- ❖ Definiamo un semaforo come una struttura C munita di
 - Un contatore
 - Una lista (coda) di processi

```
typedef struct semaphore_s {  
    int cnt;                // Numero processi  
    process_t *tail;        // Lista processi (FIFO)  
} semaphore_t;
```

Implementazione di un semaforo

Attesa solo se
 $cnt < 0$

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

Init con $k \geq 0$

FIFO (queue and dequeue on the "tail")

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

cnt può
assumere
valori negativi

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

Ci sono P in
coda solo se
 $cnt \leq 0$

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

Tutti i P rimanenti vengono
estratti dalla coda

Implementazione di un semaforo

- ❖ L'implementazione reale permette a un semaforo di avere valori negativi
 - Il suo valore assoluto indica il numero di processi in coda sul semaforo
- ❖ La coda
 - Può essere implementata con un puntatore nel Process Control Block (PCB) dei processi
 - Può soddisfare le politiche che lo scheduler desidera (e.g., FIFO)
 - Ha un comportamento indipendente dai processi in attesa

Implementazioni reali

❖ Esistono diverse implementazioni

- **Semafori tramite pipe**
- **Semafori POSIX**
- **Pthread**
 - **Mutex** (Mutua esclusione)
- **Semafori Linux**
- **Mutex standard C11**
- ...

System call:
pthread_cond_init
pthread_cond_wait
pthread_cond_signal
pthread_cond_broadcast
pthread_cond_destroy

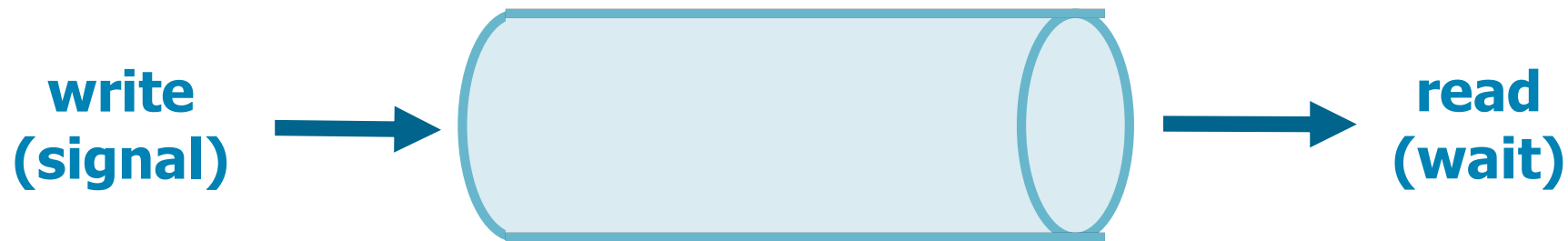
System call:
semget, semop, semctl
(in sys/sem.h) sono di
utilizzo complesso

System call:
mtx_init, mtx_destroy,
mtx_lock, mtx_trylock,
mtx_timedlock, mtx_unlock

Semafori tramite pipe

❖ Data una pipe

- Il contatore di un semaforo è realizzato tramite il concetto di **token**
- La **signal** è effettuata tramite una **write** di un token sulla pipe (non bloccante)
- La **wait** è effettuata tramite una **read** di un token dalla pipe (bloccante)



semaphore_init

```
#include <unistd.h>

void semaphore_init (int *S) {
    if (pipe (S) == -1) {
        printf ("Error");
        exit (-1);
    }
    return;
}
```

❖ Inizializza il semaforo S

➤ La variabile S va definita come variabile globale

- `int S[2];`
- `int *S = malloc (2 * sizeof (char));`

semaphore_signal

```
#include <unistd.h>

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

- ❖ Scrive un carattere (qualsiasi) sulla pipe
 - Si suppone di non eccedere la capacità massima della pipe
 - Per inizializzare il semaforo a un valore $k \neq 0$ è possibile effettuare k chiamate "iniziali"

semaphore_wait

```
#include <unistd.h>

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

- ❖ Legge un carattere dalla pipe (read bloccante)

semaphore_destroy

```
#include <unistd.h>

void semaphore_destroy (int *S) {
    close (S[0]);
    close (S[1]);
    return;
}
```

- ❖ Chiude entrambi gli estremi della pipe

Esempio

Utilizzo di una pipe
quale semaforo di
sincronizzazione tra P
padre e P figlio

Eventuale
inizializzazione del
semaforo al valore k

```
int main() {  
    int S[2];  
    pid_t pid;  
    semaphore_init (S);  
    pid = fork();  
  
    if (pid == 0) {  
        semaphore_wait (S);  
        printf("Wait done.\n");  
    } else {  
        printf("Sleep 3s.\n");  
        sleep (3);  
        semaphore_signal (S);  
        printf("Signal done.\n");  
    }  
    semaphore_destroy (S);  
  
    return 0;  
}
```

```
for (i=0; i<k; i++)  
    semaphore_signal (S);
```

Child

Parent

Semafori POSIX

❖ Ci sono due tipi di semafori POSIX

➤ Unnamed semaphores

- Implementati nella memoria interna del processo
- Sono utilizzabili nella sincronizzazione di thread all'interno dello stesso processo

➤ Named semaphores

- Implementati utilizzando la memoria condivisa, sono "process-shared semaphore"
- Sono generalmente utilizzati nella sincronizzazione tra processi
 - Il nome del semaforo permetta il suo utilizzo (sem_open) all'interno di processi diversi

Semafori POSIX

- ❖ Analizzeremo solo gli **unnamed** semaphores
 - L'implementazione è indipendente dal SO ed è definita nell'header file semaphore.h
 - Inserire nei file .c
 - `#include <semaphore.h>`
- ❖ Un semaforo è una variabile di tipo **sem_t**
 - Un semaforo può essere allocato staticamente o dinamicamente
 - `sem_t sem1, *sem2, ...;`
- ❖ Le funzioni di manipolazione
 - Sono denominate **sem_***
 - In caso di errore ritornano il valore -1

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  
);
```

Linux does not currently
support shared semaphores

- ❖ Inizializza il semaforo al valore **value**
- ❖ Il valore di **pshared** identifica il tipo del semaforo
 - Se uguale a 0, allora il semaforo è **locale** al processo corrente ("shared between threads")
 - Altrimenti, il semaforo può essere **condiviso** tra diversi processi ("shared between processes")

sem_wait

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

❖ Operazione di wait standard

- Se il semaforo è uguale a 0, blocca il chiamante sino a quando può decrementare il valore del semaforo

sem_trywait

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

- ❖ Operazione di wait senza blocco (non-blocking wait)
 - Se il semaforo ha un valore maggiore di 0, lo decrementa e ritorna 0
 - Se il semaforo è uguale a 0, ritorna -1 (invece di bloccare il chiamante come la wait)

sem_post

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

- ❖ Classica operazione signal
 - Incrementa il valore del semaforo

sem_getvalue

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

Warning: Il valore può cambiare prima di essere utilizzato

- ❖ Permette di esaminare il valore di un semaforo
 - Il valore del semaforo viene assegnato a *valP (i.e., valP è il puntatore all'intero che indica il valore del semaforo dopo la chiamata)
 - Se ci sono processi in attesa, a *valP si assegna 0 o un numero negativo il cui valore assoluto è uguale al numero di processi in attesa

sem_destroy

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

- ❖ Distrugge un semaforo creato precedentemente
- ❖ Può ritornare -1 se si cerca di distruggere un semaforo utilizzato da un altro processo

Esempio

Utilizzo delle
primitive `sem_*` per
la sincronizzazione

```
#include "semaphore.h"
```

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

```
... create threads ...
```

```
sem_destroy (&sem);
```

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

Semaforo statico

```
#include "semaphore.h"
```

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

```
... create threads ...
```

```
sem_destroy (sem);
```

Semaforo dinamico

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

Mutex Pthread

❖ Una possibile implementazione di semafori binari (mutex) è fornita dalla libreria Pthread

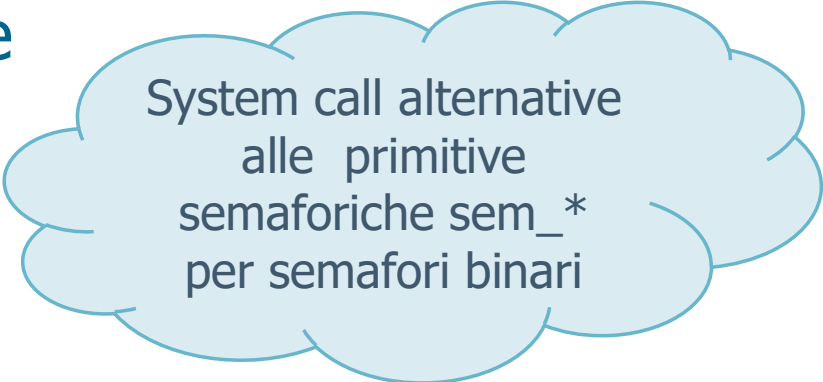
➤ Il tipo base di un mutex è **pthread_mutex_t**

➤ Funzioni base

- pthread_mutex_init
- pthread_mutex_lock
- pthread_mutex_trylock
- pthread_mutex_unlock
- pthread_mutex_destroy

➤ Occorre ricordarsi di inserire nei file **.c** la riga

- `#include <pthread.h>`



System call alternative
alle primitive
semaforiche `sem_*`
per semafori binari

pthread_mutex_init

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

- ❖ Crea un nuovo lock (mutex variable) **mutex**
 - Restituisce mutex al chiamante
 - attr specifica gli attributi di mutex (default=NULL)
- ❖ Valore di ritorno
 - Se l'operazione ha avuto successo 0
 - Il codice di errore altrimenti

pthread_mutex_lock

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

- ❖ Controlla il valore del lock **mutex** e
 - Blocca il chiamante se è già locked
 - Acquisisce il lock se non è locked
- ❖ Valore di ritorno
 - Se l'operazione ha avuto successo 0
 - Il codice di errore altrimenti

pthread_mutex_trylock

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

- ❖ Simile a `pthread_mutex_lock` ma nel caso il lock sia già stato acquisito ritorna senza bloccare il chiamante
- ❖ Valore di ritorno
 - Se il lock è stato acquisito 0
 - Se il mutex era posseduto da un altro thread, codice di errore EBUSY

pthread_mutex_unlock

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

- ❖ Rilascia il lock **mutex** al termine della SC
- ❖ Valore di ritorno
 - Se l'operazione ha avuto successo 0
 - Il codice di errore altrimenti

pthread_mutex_destroy

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

- ❖ Rilascia la memoria occupata dal lock **mutex**
- ❖ Tale lock non sarà più utilizzabile
- ❖ Valore di ritorno
 - Se l'operazione ha avuto successo 0
 - Il codice di errore altrimenti