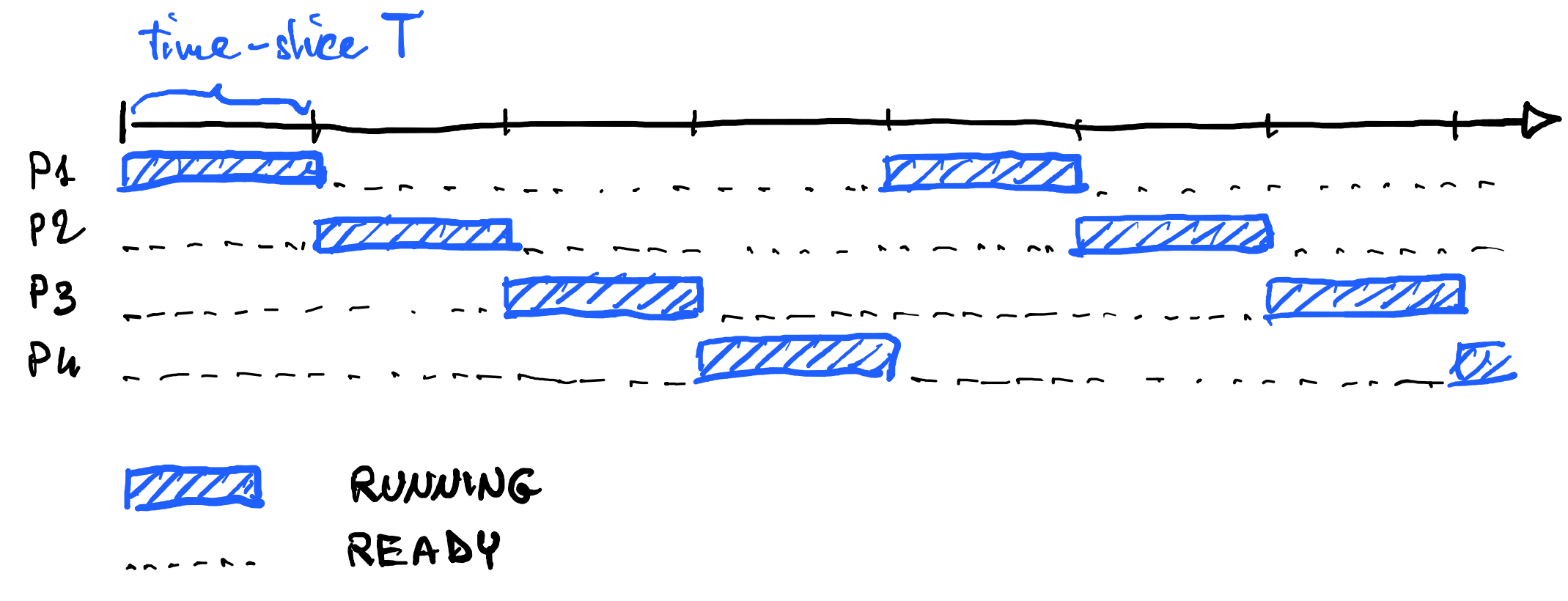
SC - 7/5/2025

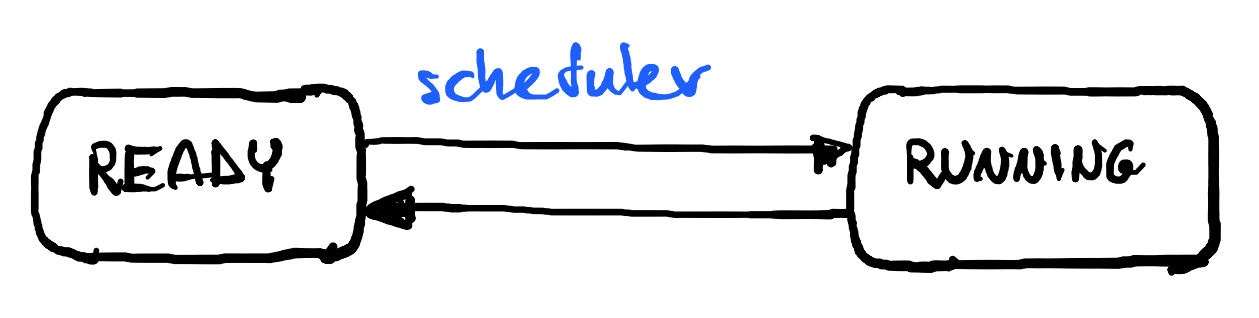
mercoledì 7 maggio 2025 10:09

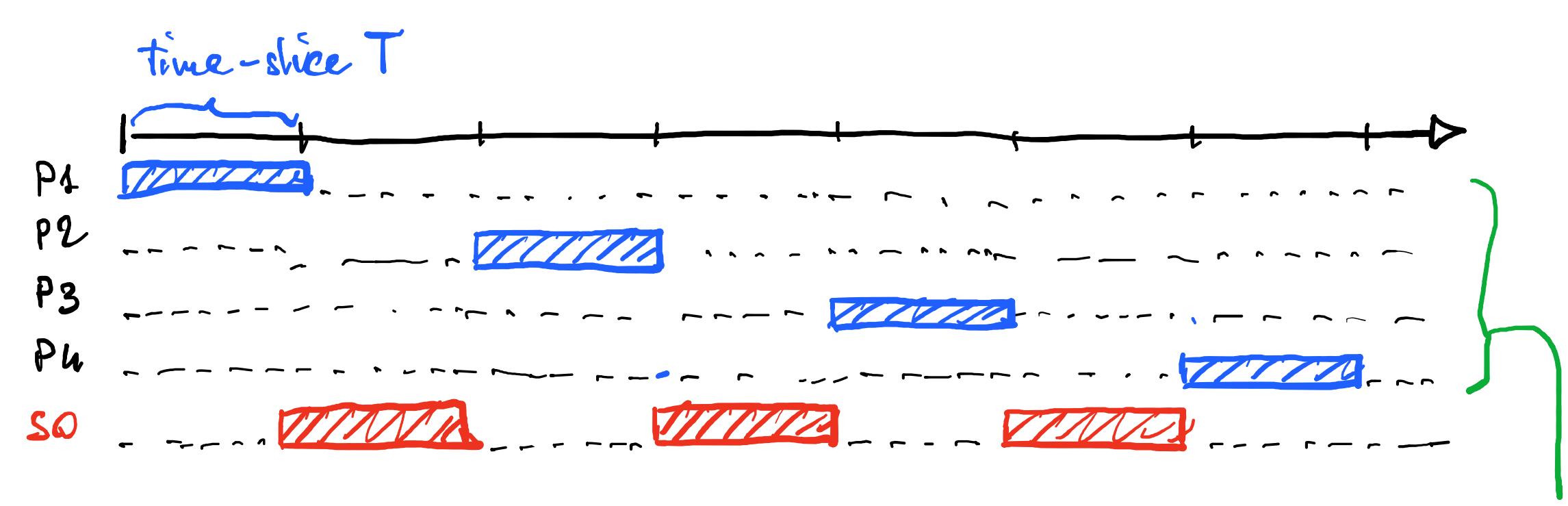
MULTIPROGRAMMAZIONE : possibilità all'interno di un programma di avere processi multipli in esecuzione concorrente

Come gestisce il SO le richieste concorrenti di uso della CPU da parte di processi distinti?

Soluzione: **TIME-SHARING** : Tutti i processi sono "sospesi" fino a quando non gli viene assegnato l'uso della CPU per un intervallo temporale T (time-slice). Per ogni time slice un solo processo può usare la CPU



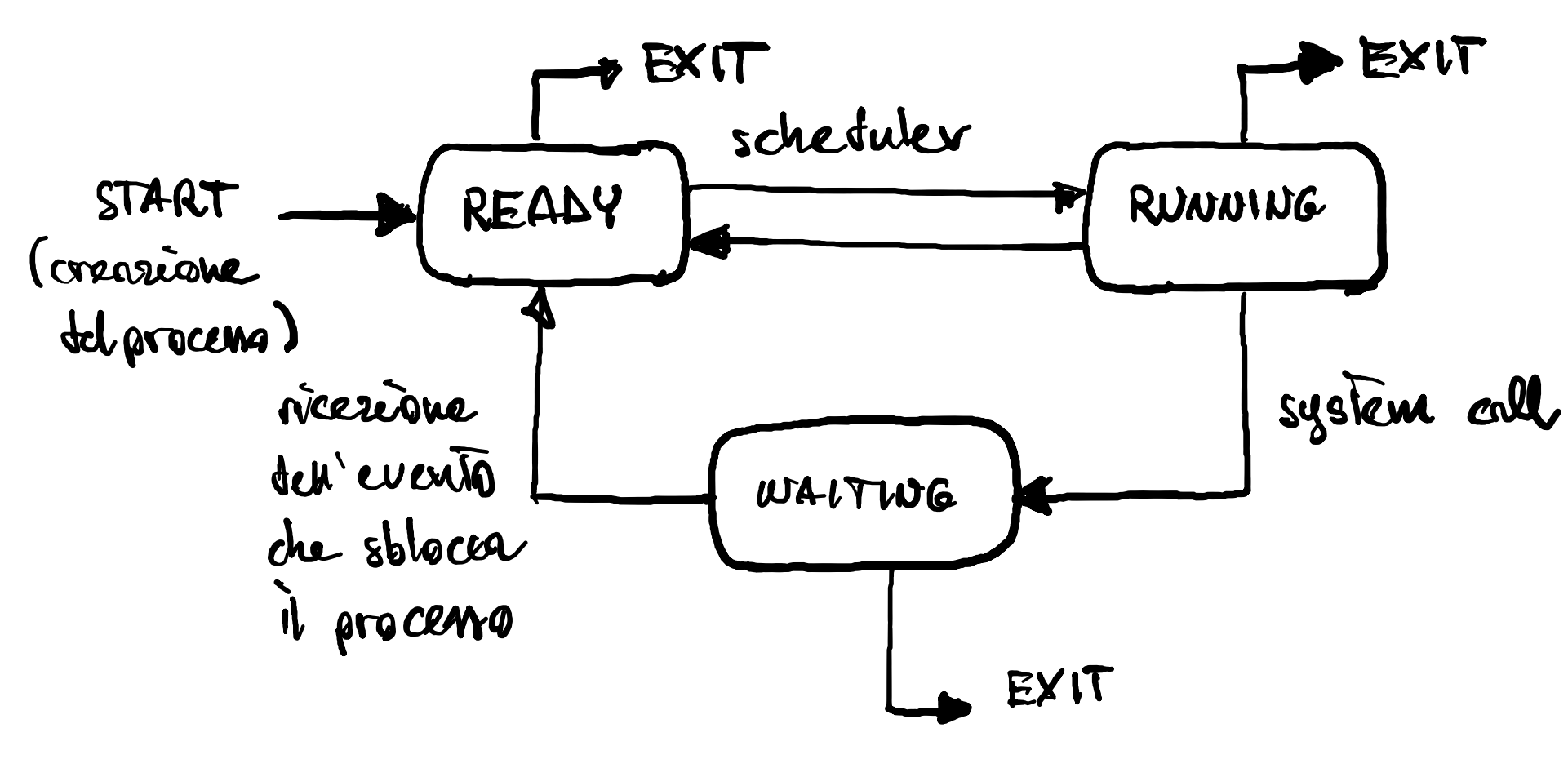


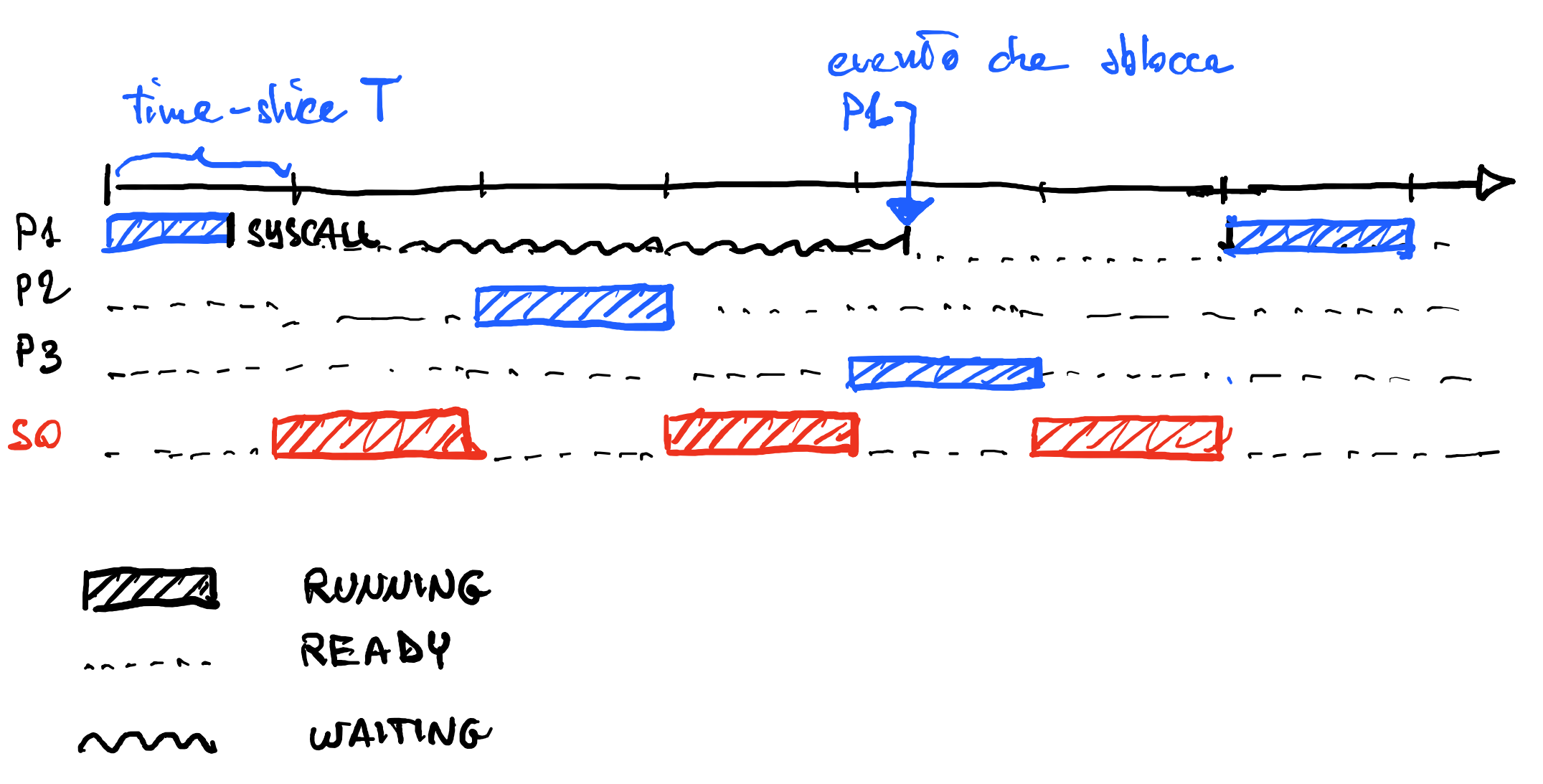


PREEMPTION : al termine della time slice che gli è stata assegnata il processo viene interrotto in modo forzato per restituire il controllo al SO

Esempio di scheduler ROUND ROBIN

STARVATION: un processo in stato READY non viene MAI portato in stato RUNNING per mancanza di risorse.
Può essere evitata grazie a:
① PREEMPTION
② scheduler con algoritmo di fair time-sharing





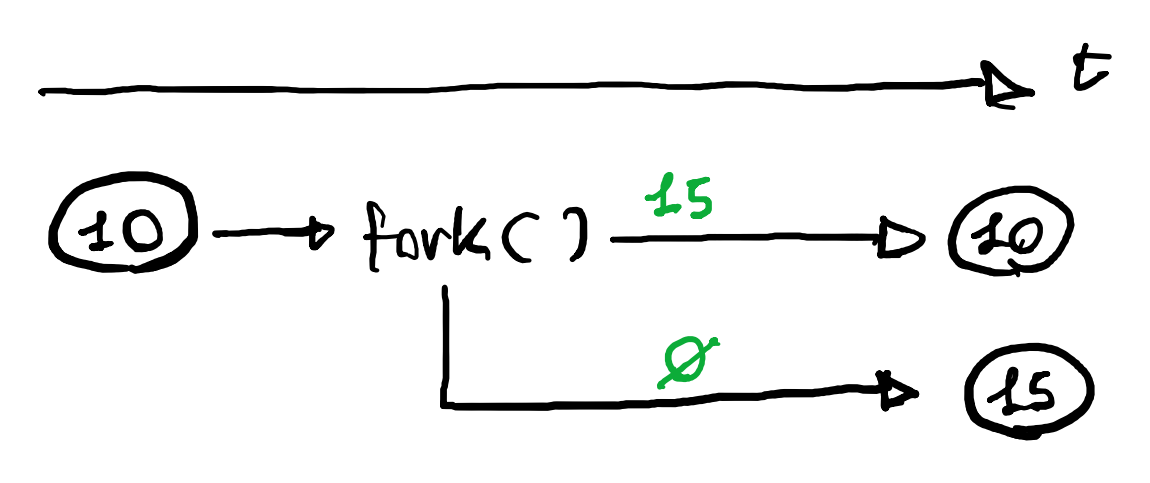
FUNZIONI PER LA GESTIONE DEI PROCESSI

fork() : genera un nuovo processo

All'invocazione di fork() il sistema operativo genera una copia del processo. Dopo la copia il processo che ha invocato fork ed il nuovo processo generato vengono messi in stato READY
ATTENZIONE: in questo momento i due processi possono essere distinti SOLO perché hanno PID diversi.

```
# include <unistd.h>

pid_T fork();
```

- -1 in caso di errore
- 0 restituito al nuovo processo creato (processo figlio)
- \>0 restituito al processo che ha invocato fork (processo padre) e rappresenta il PID del nuovo processo creato



Per distinguere il codice che va eseguito nel processo padre/figlio si usa una struttura del genere:

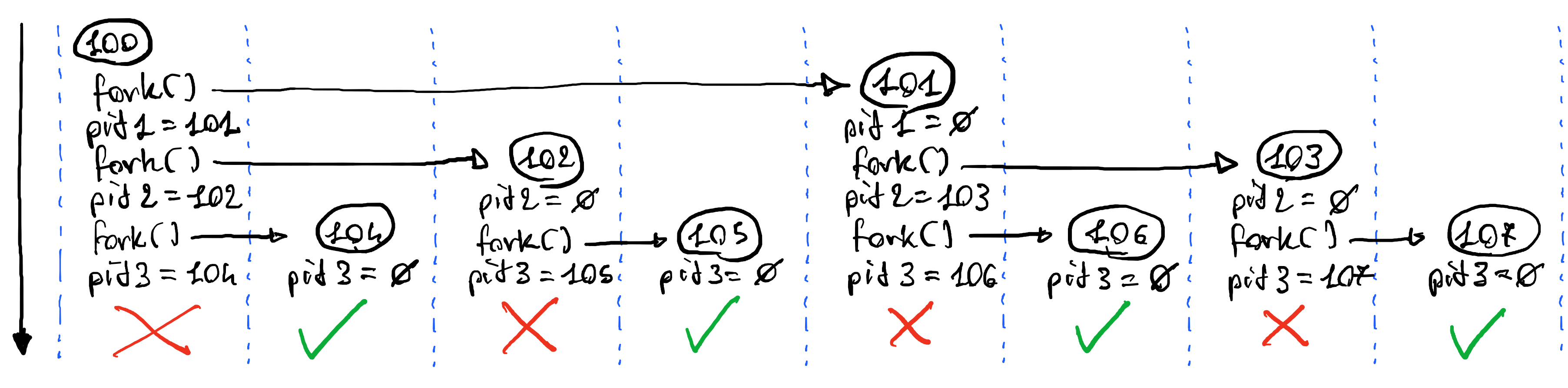
```
# include <stdlib.h>
# include <unistd.h>

int main () {
    pid_t pid = fork();
    if (pid == -1) {
        perror ("Errore in fork \n");
        exit (-1);
    }

    if ( pid == 0 ) {
        // Codice processo figlio
        printf (" Io sono il processo figlio \n");
    } else {
        // Codice processo padre
        printf (" Io sono il processo padre \n");
    }
    return 0;
}
```

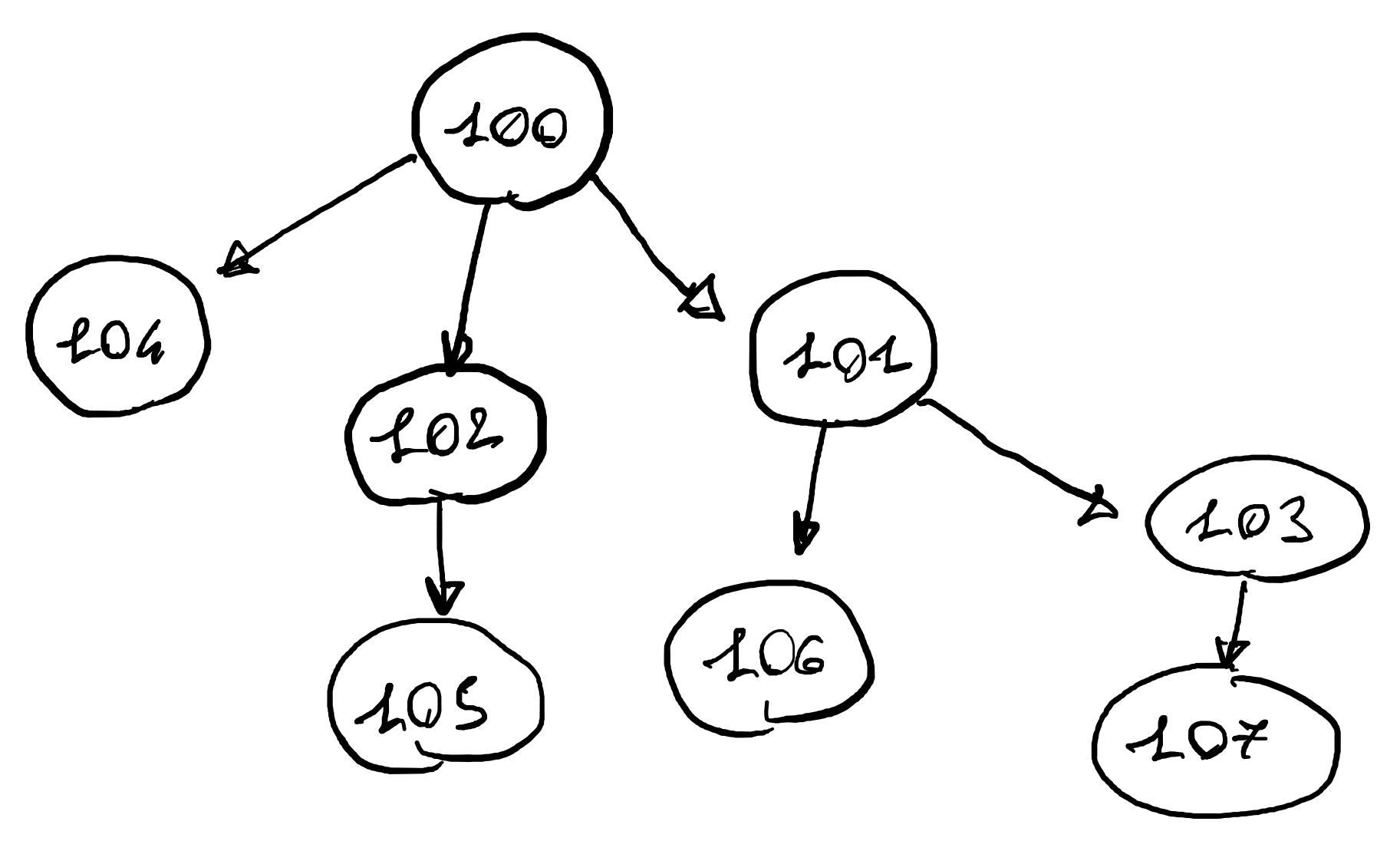
ESEGUITA SOLO DAL FIGLIO

ESEGUITA SOLO DAL PADRE

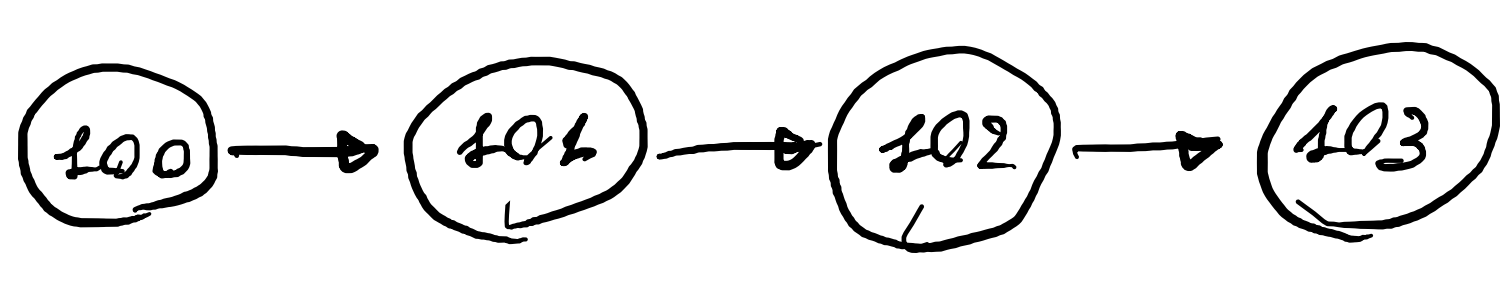
FORK MULTIPLE

```
int main() {
    pid_T pid 1 = fork();
    pid_T pid 2 = fork();
    pid_T pid 3 = fork();
    if (pid3 == 0) printf (" * \n");
}
```



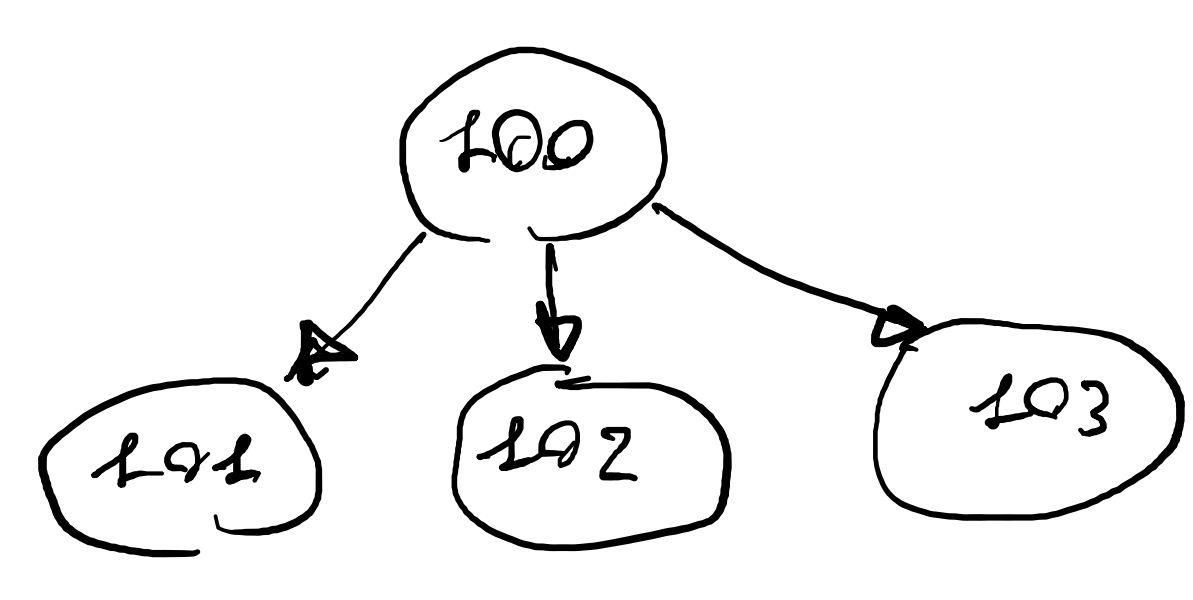


Che codice dovrei usare per ottenere



```
int main() {
    pid_T pid 1 = fork();
    if (pid 1 == 0) pid_T pid 2 = fork();
    if (pid2 == 0) pid_T pid 3 = fork();
    if (pid3 == 0) printf (" * \n");
}
```

Per esercizio scrivere il codice che genera



pid_T getpid(); → restituisce PID del processo corrente

pid_t getppid(); → " " " " padre

_exit(...); → come exit() ma NON esegue l'eventuale gestore della terminazione

SINCRONIZZAZIONE DEI PROCESSI (TERMINAZIONE)

Il processo padre può voler sapere quando i suoi processi figli terminano

> IMPORTANTE: non possiamo MAI fare alcuna assunzione sui tempi e l'ordine con cui verranno eseguite le istruzioni di due processi concorrenti

```
# include < sys/types.h >
# include < sys/wait.h >

pid_T wait ( int * status_ptr );
```

wait restituisce il PID del processo figlio che è terminato

wait è bloccante fino alla terminazione di un processo figlio.

Quando invoco wait possono verificarsi due casi distinti:

① Il figlio è ancora in funzione → attesa
② Il figlio è già terminato → ritorno immediato

Quando un processo figlio termina PRIMA che il padre invochi wait, questo è chiamato ZOMBIE

Per leggere il valore di ritorno del figlio:

WIFEXITED (status);
- 1 se il processo è terminato con exit(), _exit() o return()
- 0 altrimenti

WEXITSTATUS (status); codice di terminazione restituito dal figlio

CARICAMENTO DI PROGRAMMI

```
#include <unistd.h>

int execvp (const char * filename, char * argv[]);
```

-1 in caso di errore

percorso del file contente il programma da eseguire

parametri da passare al programma

execvp sostituisce integralmente l'immagine di memoria del processo che l'ha invocata con quella del programma da eseguire