Come è fatta l'immagine di memoria di un processo:

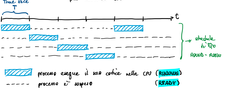
```
int main(int argc, char *argv[]) {
    Static char *s1="hello";
    int *p=malloc(sizeof(int));
    ---
    Free(p);
    Printf("...");
    ---
}
```

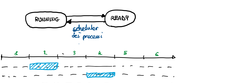
MULTIPROGRAMMAZIONE

Possibilità all'interno di un programma di avere processi multipli in esecuzione concorrente.

Come fa il so a gestire richieste di accesso alla CPU che arrivano da processi concorrenti?

Soluzione: **TIME-SHARING**: tutti i processi sono "aspettati" fino a quando non gli viene assegnato del tempo per l'uso della CPU per un intervallo temporale T (Time slice). Per ogni Time slice, un solo processo può essere in CPU

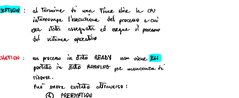




PREREQUISITO: al termine di una Time slice la CPU interrompe l'esecuzione del processo a cui gli viene assegnata ed esegue il processo del sistema operativo

CONDIZIONI: un processo in STATO READY non deve più potersi in STATO RUNNING per aumentare le risorse.
Per avere entrambe abbiamo:

1. PREEMPTIVE
2. scheduler con algoritmo di tipo Time Sharing



FUNZIONI PER LA GESTIONE DEI PROCESSI

fork() genera un nuovo processo

All'invocazione di fork() il sistema operativo

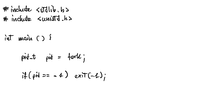
CREA UNA COPIA DEL PROCESSO

Dopo la copia il processo che ha invocato fork() e il nuovo processo generato vengono messi in STATO READY

ATTENZIONE: in questo momento i due processi possono essere diversi solo perché hanno PID diversi.

```
#include <unistd.h>

pid_t fork();
```

- -1 → errore
- 0 → restituito al nuovo processo creato (processo figlio)
- >0 → restituito al processo che ha invocato fork() (processo padre) e rappresenta il PID del processo creato.



Per distinguere il codice che va eseguito nel processo padre/figlio si usa la seguente struttura:

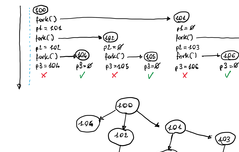
```
#include <stdio.h>
#include <unistd.h>

int main () {

    pid_t pid = fork();

    if (pid == 0) {
        /* codice del processo figlio
        printf("Sono il processo figlio\n");
    } else {
        /* codice del processo padre
        printf("Sono il processo padre\n");
    }
    return 0;
}
```

Eseguito solo dal processo figlio

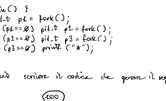
Eseguito solo dal processo padre

FORK VOLUTAPLE

```
int main() {
    pid_t pid = fork();
    pid_t pid = fork();
    pid_t pid3 = fork();
    if (pid3==0) printf("a");
}
```



Quale codice genera il seguente albero di processi?




```
int main() {
    pid_t pid = fork();
    if (pid==0) pid_t pid = fork();
    if (pid==0) pid_t pid3 = fork();
    if (pid3==0) printf("a");
}
```

Per convincersi scrivere il codice che genera il seguente albero:

```
pid_t getpid();      → restituisce il PID del processo corrente
pid_t getppid();     → " " " " " " padre
_exit(-);            → funziona come exit() ma non esegue l'eventuale gestore delle terminazione
```

SINCRONIZZAZIONE DEI PROCESSI

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

IMPORTANTE: non possiamo dire di conoscere in tempo 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**: blocca alla terminazione di un processo figlio

Quando invoco wait sono possibili due casi:

1. il figlio è ancora in esecuzione → attesa
2. " " " già terminato → ritorno immediato

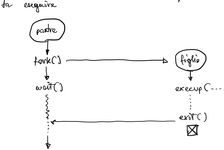
Quando un processo figlio termina, PRIMA che il padre invochi wait, questo si chiamerà "processo zombie".

Per leggere il valore di uscita del figlio:

```
WIFEXITED (status_ptr);      → se il processo è terminato con exit() - exit() o return()
                             → altrimenti
WEXITSTATUS (status_ptr);    → codice di terminazione restituito dal figlio
```

CARICAMENTO DI PROGRAMMI

```
#include <unistd.h>

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

- filename: percorso del codice del programma da eseguire
- argv[]: parametri del passare al programma.
- -1 in caso di errore

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