

```
#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)
```

```
{
    printf(stderr, "ERRORE, serve un parametro con il nome del file\n");
    exit(1);
}
```

```
f = fopen(argv[1], "r");
if(f==NULL)
```

```
{
    printf(stderr, "ERRORE, impossibile aprire il file %s\n", argv[1]);
    exit(1);
}
```

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



Processi

Controllo avanzato

Stefano Quer

Dipartimento di Automatica e Informatica

Politecnico di Torino

Fork e Exec

- ❖ La system call **fork** permette la duplicazione di un processo
- ❖ Esistono due principali applicazioni di tale meccanismo
 - Padre e figlio eseguono **sezioni diverse** di codice
 - Caso tipico dei server di rete: all'arrivo di una richiesta il server si duplica e il figlio gestisce la richiesta mentre il padre continua l'attesa
 - Padre e figlio eseguono codici **differenti**
 - Caso comune a tutte le shell
 - Richiede l'utilizzo della famiglia di comandi **exec**
 - Tale funzione viene utilizzata da moltissime altre system call

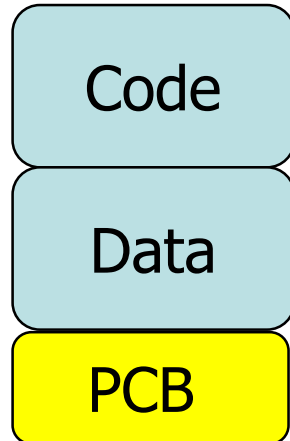
Sostituzione di un processo

- ❖ La system call **exec sostituisce** il processo con un nuovo programma
- ❖ Il nuovo programma incomincia l'esecuzione in maniera standard (dal main)
- ❖ In particolare la **exec**
 - Non crea un nuovo processo
 - Sostituisce l'immagine del processo corrente (i.e., il suo codice, i suoi dati, stack e heap) con quelli di un processo nuovo
 - Il PID del processo non cambia
 - fork → duplica un processo **esistente**
 - exec → esegue un **nuovo** programma

Spazio degli indirizzi

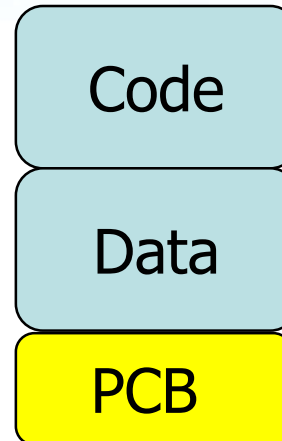
Fork:
crea nuovi processi

Process

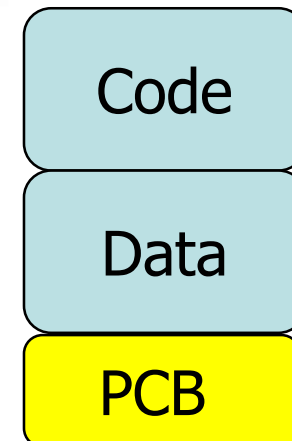


fork

Parent

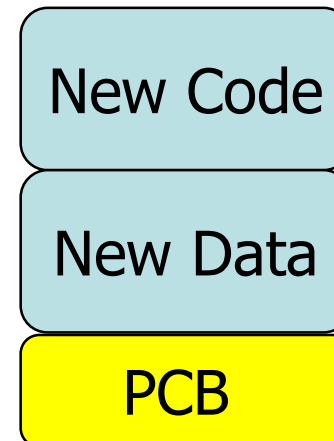


Child



exec

New Process



Exec:
esegue nuovi programmi

System call exec ()

Stevens, Rago,
Edizione 3, Pagina 249

- ❖ Esistono **6** versioni della system call exec
 - execl, execlp, execl
 - execv, execvp, execve

Tipo	Azione
l (list)	La funzione riceve una lista di argomenti
v (vector)	La funzione riceve un vettore di argomenti
p (path)	La funzione riceve solo il nome del file (non il suo path) e lo rintraccia tramite la variabile di ambiente PATH
e (environment)	La funzione riceve un vettore di environment che specifica le variabili di ambiente, invece di utilizzare l'environment corrente

Significato opposto
(p: logica negativa
e: logica positiva)

System call exec ()

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

```
int execl (char *path, char *arg0, ..., (char *)0);  
int execlp (char *name, char *arg0, ..., (char *)0);  
int execl_e (char *path, char *arg0, ..., (char *)0,  
             char *envp[]);
```

```
int execv (char *path, char *argv[]);  
int execvp (char *name, char *argv[]);  
int execve (char *path, char *argv[], char *envp[]);
```

❖ Valore di ritorno

- **Nessuno**, in caso di successo
- Il valore -1, in caso di errore

Il programma corrente
non esiste più

System call exec ()

❖ Parametri

➤ Il path (nome) del programma da eseguire

- Tale nome può sempre specificare il nome di un file oppure il nome di un file con il relativo path
- Nelle versioni "p" della exec è sufficiente (si dovrebbe) specificare il solo nome del file
 - Se il nome non contiene un path, questo è definito dalla variabile di sistema PATH (echo \$PATH)
 - Se il nome contiene un path, la versione "p" di exec coincide con quella non-"p"
- Nelle versioni non-"p" il nome dovrebbe includere il path
 - Se il path rimane ignoto l'eseguibile può risultare irraggiungibile

```
int execl (char *path, ...);  
int execlp(char *name, ...);
```


System call exec ()

➤ La sua lista di argomenti

- Nelle versioni "l" la exec riceve un elenco di parametri (come un main C)
 - Il primo argomento è il nome dell'eseguibile non l'eseguibile stesso
 - In pratica è la stringa argv[0] della sintassi C
 - I successivi sono i parametri dell'eseguibile
 - In pratica sono gli argv[i] con i>0 della sintassi C
- Nelle versioni "v" l'argomento è un vettore di puntatori agli argomenti stessi
 - In pratica è una matrice dinamica simile a **argv
 - Simile, non identica, perché è "NULL terminated"
 - Il valore argv[i]==NULL indica la fine degli argomenti

```
int execl (char *path, char *arg0, ..., (char *)0);  
int execv (char *path, char *argv[], ...);
```


System call exec ()

➤ Le (eventuali) variabili di ambiente

- Nelle versioni non-"e" le variabili di ambiente sono ereditate dal processo chiamante
- Nelle versioni "e" le variabili di ambiente sono specificate esplicitamente
 - Si indica una seconda matrice dinamica NULL-terminated, ovvero un vettore di puntatori a stringhe di caratteri
 - Tali stringhe specificano i valori delle variabili di ambiente desiderate (e.g., variabile=valore)

```
int execl (char *path, char *argv[]);  
int execve (char *path, char *argv[], char *envp[]);
```

Esempi

OK

whereis cp: /bin/cp

Nome "fittizio"

```
execl("/bin/cp", "cp", "./file1", "./file2", NULL);
```

OK

Terminazione diversa

```
execl("/bin/cp", "cp", "./file1", "./file2", (char*)0);
```

NO

Manca il path

```
execl("cp", "copioFile", "./file1", "./file2", (char*)0);
```

OK

Path di default (\$PATH)

```
execvp("cp", "copioFile", "./file1", "./file2", (char*)0);
```

Esempio

Scrivere un programma (**./pgrm**)
che richiama se stesso se riceve
come parametro 1 oppure 2

```
...
n = atoi (argv[1]);
switch (n) {
    case 1:
        printf("#1:PID=%d;PPID=%d\n", getpid(), getppid());
        sleep (n*10);
        execvp ("./pgrm", "./*grm", "2", (char *) 0);
        break;
    case 2:
        printf("#2:PID=%d;PPID=%d\n", getpid(), getppid());
        sleep (n*10);
        execvp ("./pgrm", "ilMioPgrm", "3", (char *) 0);
        break;
    default:
        printf("#3:PID=%d;PPID=%d\n", getpid(), getppid());
        sleep (n*10);
        break;
}
return (1);
```

Anche argv[0]

Il path è lo stesso
arg0 (il suo nome) cambia

Esempio

Run con n=1

IL PID non cambia

Comandi di shell (in blu)

```
> ./pgrm 1 &
[2] 2471
#1: PID=2471; PPID=2045
> ps -aux | grep 2471
quer 2471 0.0 0.0 4192 352 pts/2 S 19:29 0:00 ./pgrm 1
#2: PID=2471; PPID=2045
> ps -aux | grep 2471
quer 2471 0.0 0.0 4192 356 pts/2 S 19:29 0:00 ./Pgrm 2
#3: PID=2471; PPID=2045
> ps -aux | grep 2471
quer 2471 0.0 0.0 4192 356 pts/2 S 19:29 0:00 ilMioPgrm 3
[2]+  Exit 1 ./pgrm 1
```

Il nome cambia

System call exec ()

❖ execv[p]

➤ Utilizza come parametro un unico puntatore

- Il puntatore individua un vettore di puntatori ai parametri
- Il vettore va opportunamente inizializzato

```
char *cmd[] = {  
    "ls",  
    "-laR",  
    ".",  
    (char *) 0  
};
```

Il vettore di puntatori può dover essere creato **dinamicamente** (e.g., laboratorio)

L'ultimo argomento è indicato con un puntatore NULL

```
...  
execv ("/bin/ls", cmd);
```

System call exec ()

❖ exec[lv]e

➤ Specifica esplicitamente l'environment

- Puntatore a un vettore di puntatori
- Per le altre funzioni l'environment (del nuovo programma) è ereditato dal processo chiamante

```
char *env[] = {  
    "USER=unknown",  
    "PATH=/tmp",  
    NULL  
};  
...  
execle (path, arg0, ..., argn, NULL, env);  
...  
execve (path, argv, env);
```

Considerazioni

- ❖ Si osservi che durante la exec
 - Vengono mantenuti tutti i file descriptor esistenti (compresi stdin, stdout, stderr)
 - Questo comportamento è necessario per ereditare eventuali redirezioni impostate nei comandi di shell una volta eseguita la exec
- ❖ In molti SO
 - Solo una versione della exec (in genere la **execve**) è implementata come system call
 - Le altre versioni chiamano tale versione

Esercizio

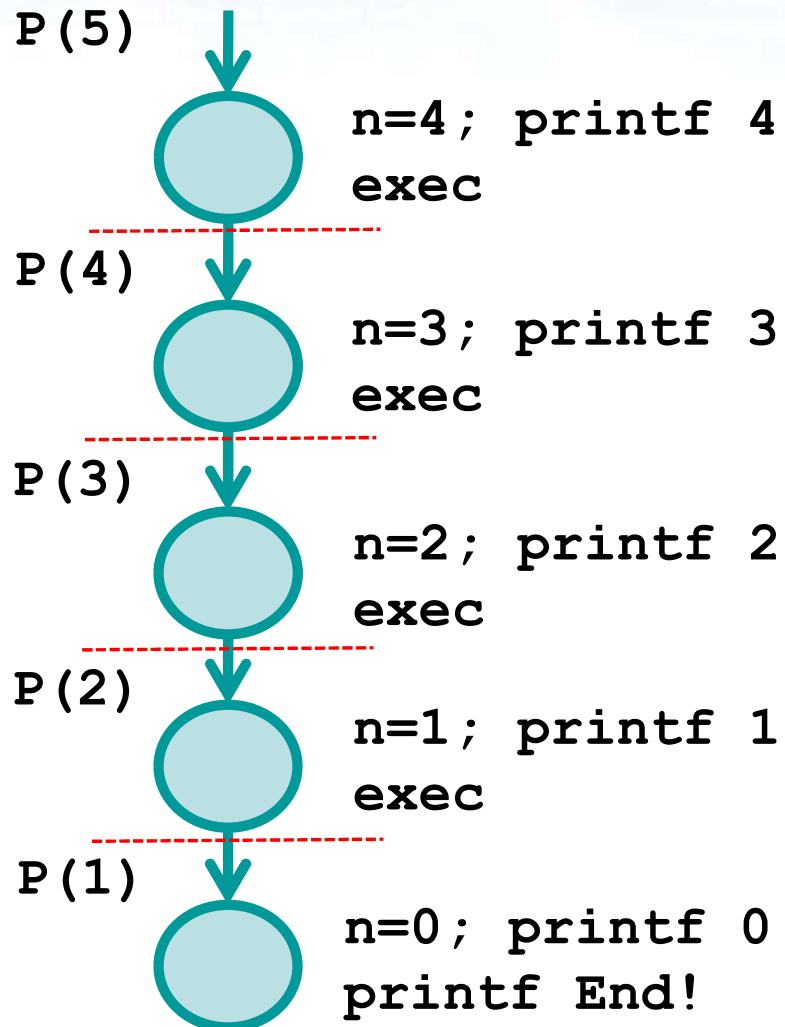
- ❖ Si riporti l'albero di generazione dei processi a seguito dell'esecuzione del seguente tratto di codice C
 - Si supponga che il programma venga eseguito con un unico parametro, il valore intero 5, sulla riga di comando
- ❖ Si indichi inoltre che cosa esso produce su video e per quale motivo

Esercizio

Run con n=5

```
#include <stdio.h>
...
#include <unistd.h>
int main (int argc, char ** argv) {
    char str[10];
    int n;
    n = atoi(argv[1]) - 1;
    printf ("%d\n", n);
    if (n>0) {
        sprintf (str, "%d", n);
        execl (argv[0], argv[0], str, NULL);
    }
    printf ("End!\n");
    return 1;
}
```

Soluzione



```
int main (int argc, char ** argv) {  
    char str[10];  
    int n;  
    n = atoi(argv[1]) - 1;  
    printf ("%d\n", n);  
    if (n>0) {  
        sprintf (str, "%d", n);  
        execl (argv[0], argv[0], str, NULL);  
    }  
    printf ("End!\n");  
    return 1;  
}
```

Output

```
4  
3  
2  
1  
0  
End!
```

Esercizio

- ❖ Si riporti l'albero di generazione dei processi a seguito dell'esecuzione del seguente tratto di codice C
- ❖ Si indichi inoltre che cosa esso produce su video e per quale motivo

Esercizio

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

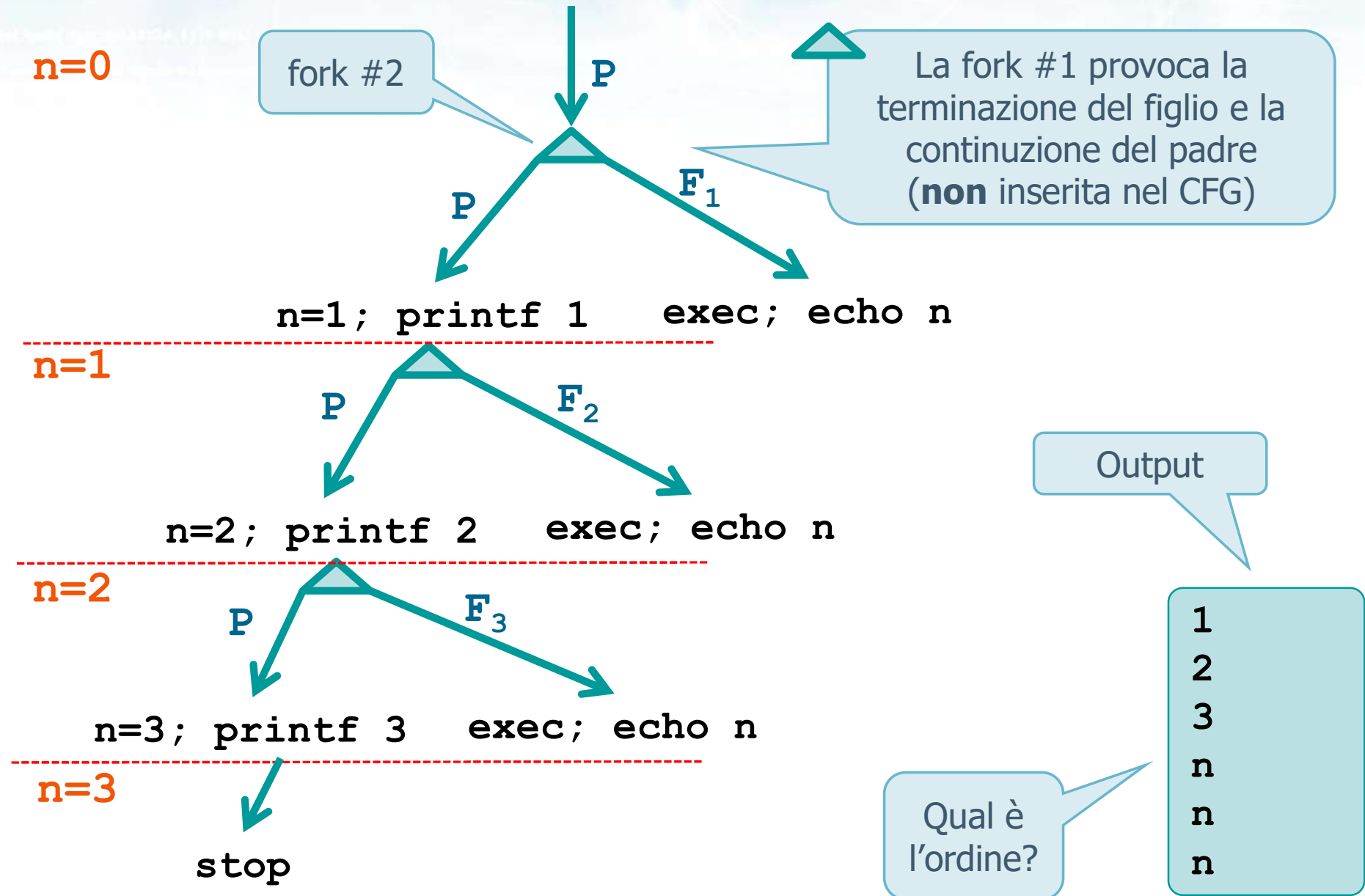
```
int main(){
    int n;
    n=0;
    while (n<3 && fork()){
        if (!fork())
            execlp ("echo", "n++", "n", NULL);
        n++;
        printf ("%d\n", n);
    }
    return (1);
}
```

fork #1
Se 0 siamo nel figlio; il
figlio termina subito

fork #2
Se 0 siamo nel figlio;
il figlio fa la exec

"printf" di shell

Soluzione



Scheletro di una shell UNIX

❖ Comando eseguito in foreground

➤ <comando>

```
while (TRUE) {  
    write_prompt ();  
    read_command (command, parameters);  
    if (fork() == 0)  
        /* Child: Execute Command */  
        execve (command, parameters);  
    else  
        /* Father: Wait Child */  
        wait (&status);  
}
```

Il programma cambia il processo no. Il padre rimane tale e può fare la wait.

Scheletro di una shell UNIX

❖ Comando eseguito in background

➤ `<comando> &`

```
while (TRUE) {  
    write_prompt ();  
    read_command (command, parameters);  
    if (fork() == 0)  
        /* Child: Execute Command */  
        execve (command, parameters);  
    #if 0  
    else  
        /* Padre: NON Attende */  
        wait (&status);  
    #endif  
}
```

Esecuzione di un comando

Stevens, Rago,
Edizione 3, Pagina 264

- ❖ Può essere conveniente eseguire una **stringa di comando** dall'interno di un programma in esecuzione
 - Ad esempio può essere utile inserire data o ora nel nome o nel contenuto di un file
- ❖ A tale scopo è nata la funzione **system**
 - Definita dallo standard ISO C e POSIX
 - Anche se definita dallo standard C, è fortemente implementation-dependent
 - Risulta sempre presente nei sistemi UNIX-like

System call system ()

```
#include <stdlib.h>

int system (const char *string);
```

❖ La system call **system**

- Passa il comando **string** all'ambiente host affinché questo lo esegua
 - In pratica, invoca il comando string all'interno di una shell
- Il controllo viene restituito al processo chiamante una volta che l'esecuzione del comando è terminata

System call system ()

❖ Parametri

- Il comando da eseguire

Essendo implementata con fork, exec e wait ha diverse condizioni di terminazione

❖ Valore di ritorno

- -1, se fallisce la fork o la waitpid usata per realizzarla
- 127, se fallisce la exec usata per realizzarla
- Il valore di terminazione della shell che esegue il comando (con formato specificato dalla waitpid)

```
int system (const char *string);
```

Esempi

```
...  
system ("ls -laR");  
...
```

Ridirezione ...
vedere sezione relativa
alle pipe

```
...  
system ("date");  
...  
system ("date > file");  
...
```

```
char str[L];  
...  
strcpy (str, "ls -la");  
system (str);  
...
```

Implementazione della system ()

❖ Le versioni iniziali di UNIX

- Implementavano la system call system tramite
 - fork, exec e wait
- Erano inefficienti a causa del polling

```
while ( (lastpid=wait(&status)) != pid  
        && lastpid!=-1 );
```

❖ Le versioni attuali

- Utilizzano solitamente le system call fork, exec e waitpid

Implementazione della system ()

Stevens, Rago,
Edizione 3, Pagina 266

```
int system (const char *cmd) {  
    pid_t pid;  
    int status;  
    if (cmd == NULL)  
        return(1);  
    if ( (pid = fork()) < 0) {  
        status = -1;  
    } else if (pid == 0) {  
        execl("/bin/sh", "sh", "-c", cmd, (char *) 0);  
        _exit(127);  
    } else {  
        while (waitpid (pid, &status, 0) < 0)  
            if (errno != EINTR) {  
                status = -1;  
                break;  
            }  
    }  
    return(status);  
}
```

Errore nella fork

La shell deve leggere da
riga di comando non da
stdin

Non exit (per non
fare fflush nel
figlio)

Interrupted
function call

Options:
WNOHANG

Esercizio

- ❖ Si riporti l'albero di generazione dei processi a seguito dell'esecuzione del seguente tratto di codice C
 - Si supponga che il programma venga eseguito con un unico parametro, il valore intero 4, sulla riga di comando
- ❖ Si indichi inoltre che cosa esso produce su video e per quale motivo

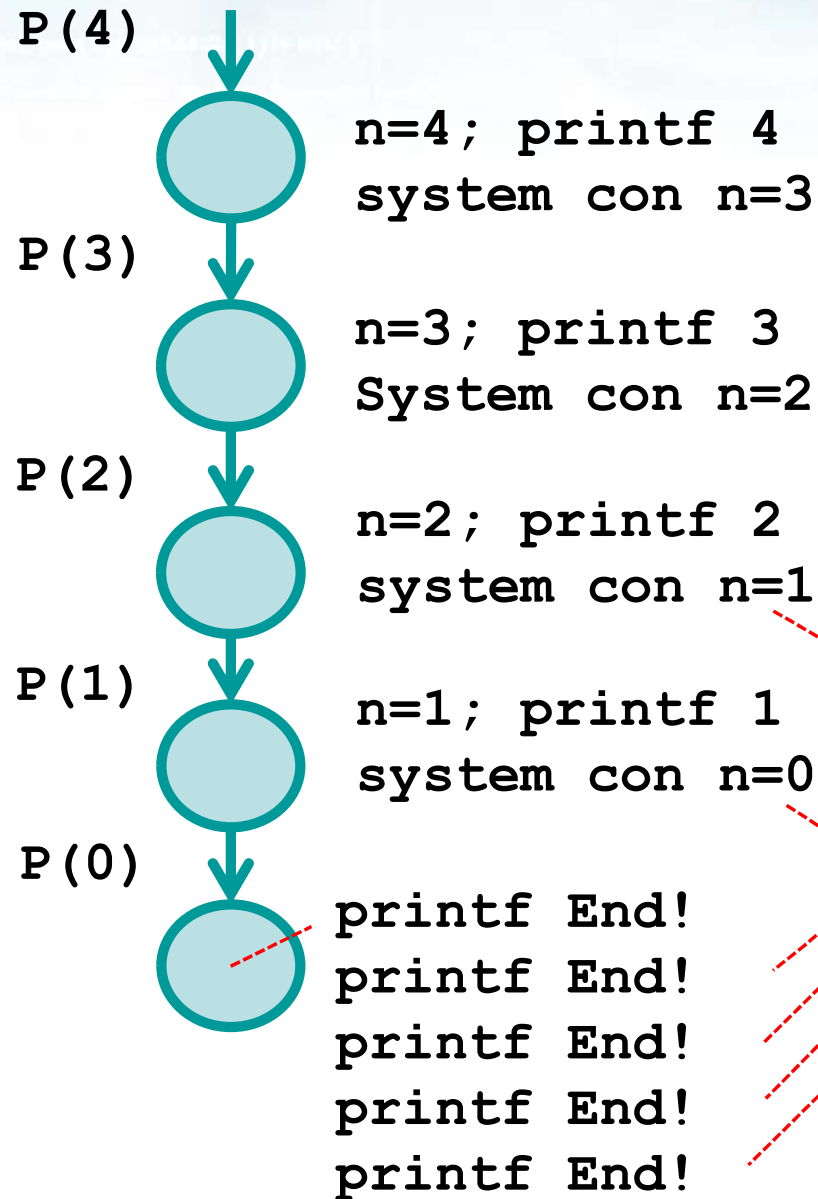
Esercizio

Run con n=4

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

int main(int argc, char ** argv){
    int n;
    char str[10];
    n = atoi (argv[1]);
    if (n>0) {
        printf ("%d\n", n);
        sprintf (str, "%s %d", argv[0], n-1);
        system (str);
    }
    printf("End!\n");
    return (1);
}
```

Soluzione



```
n = atoi (argv[1]);  
if (n>0) {  
    printf ("%d\n", n);  
    sprintf (str, "%s %d", argv[0], n-1);  
    system (str);  
}  
printf("End!\n");
```

Output

4
3
2
1
End.
End.
End.
End.
End.

Esercizio

- ❖ Si riporti l'albero di generazione dei processi a seguito dell'esecuzione del seguente tratto di codice C
- ❖ Si indichi inoltre che cosa esso produce su video e per quale motivo

Esercizio

```
#include ...
int main () {
    char str[100];
    int i;
    for (i=0; i<2; i++){
        if (fork()!=0) {
            sprintf (str, "echo system with i=%d", i);
            system (str);
        } else {
            if (fork()==0) {
                sprintf (str, "exec with i=%d", i);
                execlp ("echo", "myPgrm", str, NULL);
            }
        }
    }
    return (0);
}
```

100



i=1

i=2

● exec echo with i=%d

Qual è l'ordine?

```
system with i=0
system with i=1
exec with i=1
exec with i=0
system with i=1
exec with i=1
```