Seconda Esercitazione
Gruppo LZ

Gestione di processi in Unix

Primitive Fork, Wait, Exec

System call fondamentali

`fork`	• Generazione di un processo figlio, che **condivide il codice** con il padre e eredita **copia dei dati** del padre • Restituisce il PID (>0) del processo creato per il padre, 0 per il figlio, o un valore negativo in caso di errore
`exit`	• **Terminazione** di un processo • Accetta come parametro lo **stato di terminazione** (0-255). Per convenzione 0 indica un'uscita con **successo**, un valore non-zero indica uscita con **fallimento**.
`wait`	• Chiamata **bloccante**. • Raccoglie lo stato di terminazione di un figlio • Restituisce il PID del figlio terminato e permette di capire il **motivo della terminazione** (es. volontaria? con quale stato? Involontaria? A causa di quale segnale?)
`exec`	• **Sostituzione di codice (e dati)** del processo che l'invoca • **NON** crea processi figli

Esempio - fork e exit

Consideriamo un programma in cui il processo padre procede alla creazione di un numero N di figli

```
./generate <N> <term>
```

Dove :

- **`N`** è il numero di figli
- **`term`** è un flag [0,1]
  - se 1, ogni figlio fa exit()
  - altrimenti no.

Esempio - Il Codice

```
void main(int argc, char *argv[]) {
  int i, j, k, pid, status, n_children;
  char term;
  n_children = atoi(argv[1]);
  term = argv[2][0];
  for ( i=0; i<n_children; i++ ) {
    pid = fork();
    if ( pid == 0 ) { // Eseguito dai figli
      if ( term == '1' )    exit(0);
    }
    else if ( pid > 0 ) { // Eseguito dal padre
      printf("%d: child created with PID %d\n",
      getpid(), pid);
    }
    else {
      perror("Fork error:");
      exit(1);
    }
  }
}
```

Simulazione di Esecuzione (1/7)

Vediamo cosa succede durante l'esecuzione del programma

Assumiamo:

`N = 2` : Il padre genera due processi figli

`term = '0'` : I figli non chiamano exit

Da ricordare:

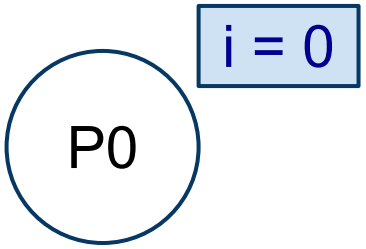
Una volta creato, ogni figlio esegue **concorrentemente** al padre e ai fratelli a partire dall'istruzione successiva alla fork() che l'ha creato.

Processo P0

```
void main(int argc, char *argv[]) {
  int i, j, k, pid, status, n_children;
  char term;
  n_children = atoi(argv[1]);
  term = argv[2][0];
→ for ( i=0; i<n_children; i++ ) {
    pid = fork();
    if ( pid == 0 ) { // Eseguito dai figli
      if ( term == '1' )    exit(0);
    }
    else if ( pid > 0 ) { // Eseguito dal padre
      printf("%d: child created with PID %d\n",
            getpid(), pid);
    }
    else {
      perror("Fork error:");
      exit(1);
    }
  }
}
```

i = 0

Simulazione di Esecuzione (2/7)



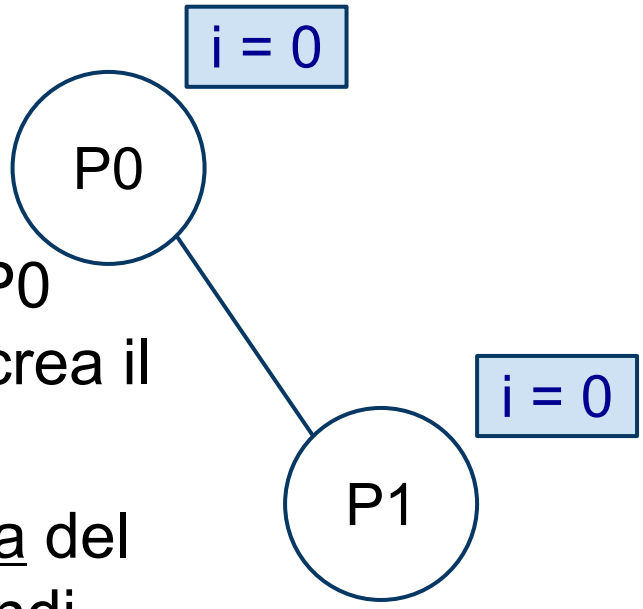
Il processo padre P0 viene creato e inizia la prima iterazione del for (i=0)

Processo P0

```
void main(int argc, char *argv[]) {
  int i, j, k, pid, status, n_children;
  char term;
  n_children = atoi(argv[1]);
  term = argv[2][0];
  for ( i=0; i<n_children; i++ ) {
    pid = fork();
    if ( pid == 0 ) { // Eseguito dai figli
      if ( term == '1' )    exit(0);
    }
    else if ( pid > 0 ) { // Eseguito dal padre
      printf("%d: child created with PID %d\n",
            getpid(), pid);
    }
    else {
      perror("Fork error:");
      exit(1);
    }
  }
}
```

i = 0

Simulazione di Esecuzione (3/7)



Il processo padre P0 esegue la fork() e crea il primo figlio P1.

P1 riceve una copia del contesto di P0, quindi anche una sua variabile `i` inizializzata a 0.

Continuiamo a concentrarci su **P0** (padre)

Per il momento trascuriamo **P1**, che intanto sta eseguendo...

Processo P0

```
void main(int argc, char *argv[]) {
  int i, j, k, pid, status, n_children;
  char term;
  n_children = atoi(argv[1]);
  term = argv[2][0];
  for ( i=0; i<n_children; i++ ) {
    pid = fork();
    if ( pid == 0 ) { // Eseguito dai figli
      if ( term == '1' )    exit(0);
    }
    else if ( pid > 0 ) { // Eseguito dal padre
      printf("%d: child created with PID %d\n",
             getpid(), pid);
    }
    else {
     perror("Fork error:");
     exit(1);
    }
  }
}
```

i = 0

La prima differenza tra i contesti di P0 e P1 è la variabile pid.

- P1: **pid=0**
- P0: **pid>0** (pid del figlio)

P0 esegue la **printf**

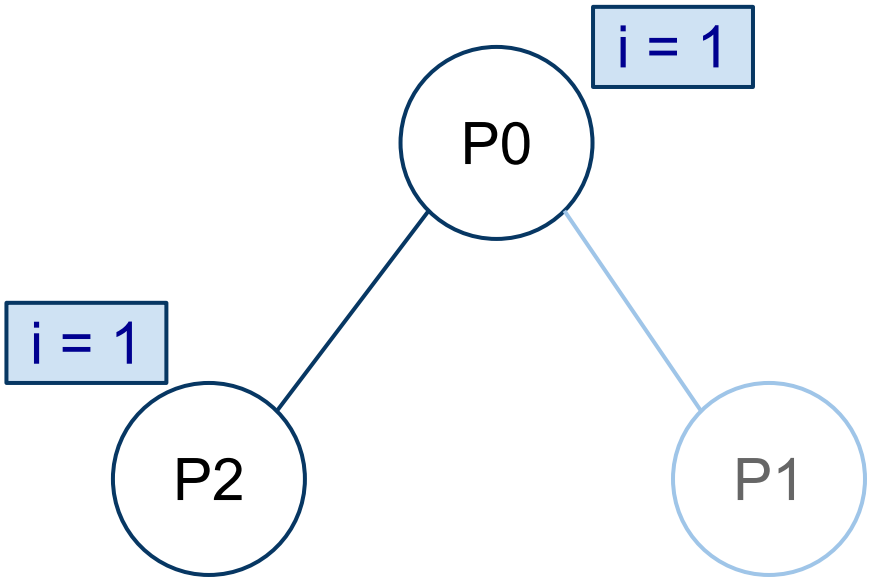
Processo P0

```
void main(int argc, char *argv[]) {
  int i, j, k, pid, status, n_children;
  char term;
  n_children = atoi(argv[1]);
  term = argv[2][0];
  for ( i=0; i<n_children; i++ ) {
    pid = fork();
    if ( pid == 0 ) { // Eseguito dai figli
      if ( term == '1' )    exit(0);
    }
    else if ( pid > 0 ) { // Eseguito dal padre
      printf("%d: child created with PID %d\n",
             getpid(), pid);
    }
    else {
      perror("Fork error:");
      exit(1);
    }
  }
}
```

i = 1

P0 continua l'esecuzione e ricomincia il ciclo for con **i=1**. Esegue ancora una fork

Simulazione di esecuzione (4/7)



La fork eseguita da P0 genera P2, che riceve una copia del contesto di P0. Quindi P2 riceve anche una variabile i inizializzata a 1.

Processo **P0**

```
void main(int argc, char *argv[]) {
  int i, j, k, pid, status, n_children;
  char term;
  n_children = atoi(argv[1]);
  term = argv[2][0];
  for ( i=0; i<n_children; i++ ) {
    pid = fork();
    if ( pid == 0 ) { // Eseguito dai figli
      if ( term == '1' )    exit(0);
    }
    else if ( pid > 0 ) { // Eseguito dal padre
      printf("%d: child created with PID %d\n",
             getpid(), pid);
    }
    else {
      perror("Fork error:");
      exit(1);
    }
  }
}
```

i = 1

P0 esegue ancora una **printf()**

Processo **P0**

```
void main(int argc, char *argv[]) {
  int i, j, k, pid, status, n_children;
  char term;
  n_children = atoi(argv[1]);
  term = argv[2][0];
→ for ( i=0; i<n_children; i++ ) {
    pid = fork();
    if ( pid == 0 ) { // Eseguito dai figli
      if ( term == '1' )    exit(0);
    }
    else if ( pid > 0 ) { // Eseguito dal padre
      printf("%d: child created with PID %d\n",
             getpid(), pid);
    }
    else {
      perror("Fork error:");
      exit(1);
    }
  }
}
```

i = 2

P0 ricomincia il ciclo for: i=2. Testa la condizione (2<2), esce dal for

Simulazione di esecuzione (5/7)

P0 a questo punto ha creato tutti i figli che doveva

MA

Cosa hanno fatto i suoi figli nel frattempo ?

Iniziamo da **P2**...
Ricordate: i processi figli non terminano subito dopo essere stati creati (term = '0')

Processo P2

```
void main(int argc, char *argv[]) {
  int i, j, k, pid, status, n_children;
  char term;
  n_children = atoi(argv[1]);
  term = argv[2][0];
  for ( i=0; i<n_children; i++ ) {
    pid = fork();
    if ( pid == 0 ) { // Eseguito dai figli
      if ( term == '1' )    exit(0);
    }
    else if ( pid > 0 ) { // Eseguito dal padre
      printf("%d: child created with PID %d\n",
             getpid(), pid);
    }
    else {
      perror("Fork error:");
      exit(1);
    }
  }
}
```

i = 1

P2 esegue il suo codice a partire da **`if(pid==0)`**

Processo P2

```
void main(int argc, char *argv[]) {
  int i, j, k, pid, status, n_children;
  char term;
  n_children = atoi(argv[1]);
  term = argv[2][0];
→ for ( i=0; i<n_children; i++ ) {
    pid = fork();
    if ( pid == 0 ) { // Eseguito dai figli
      if ( term == '1' )    exit(0);
    }
    else if ( pid > 0 ) { // Eseguito dal padre
      printf("%d: child created with PID %d\n",
             getpid(), pid);
    }
    else {
      perror("Fork error:");
      exit(1);
    }
  }
}
```

i = 2

P2 ricomincia il ciclo for: i=2. Testa la condizione (2<2), esce dal for e termina.

Simulazione di esecuzione (..continua)

Analizziamo il comportamento di **P1**….

Processo P1

```
void main(int argc, char *argv[]) {
  int i, j, k, pid, status, n_children;
  char term;
  n_children = atoi(argv[1]);
  term = argv[2][0];
  for ( i=0; i<n_children; i++ ) {
    pid = fork();
    if ( pid == 0 ) { // Eseguito dai figli
      if ( term == '1' )    exit(0);
    }
    else if ( pid > 0 ) { // Eseguito dal padre
      printf("%d: child created with PID %d\n",
             getpid(), pid);
    }
    else {
      perror("Fork error:");
      exit(1);
    }
  }
}
```

i = 0

P1 esegue il suo codice a partire da **`if(pid==0)`**

Processo P1

```
void main(int argc, char *argv[]) {
  int i, j, k, pid, status, n_children;
  char term;
  n_children = atoi(argv[1]);
  term = argv[2][0];
→ for ( i=0; i<n_children; i++ ) {
    pid = fork();
    if ( pid == 0 ) { // Eseguito dai figli
      if ( term == '1' )    exit(0);
    }
    else if ( pid > 0 ) { // Eseguito dal padre
      printf("%d: child created with PID %d\n",
             getpid(), pid);
    }
    else {
      perror("Fork error:");
      exit(1);
    }
  }
}
```

i = 1

Poichè la sua copia di i vale 0, P1 ricomincia il ciclo con i=1.

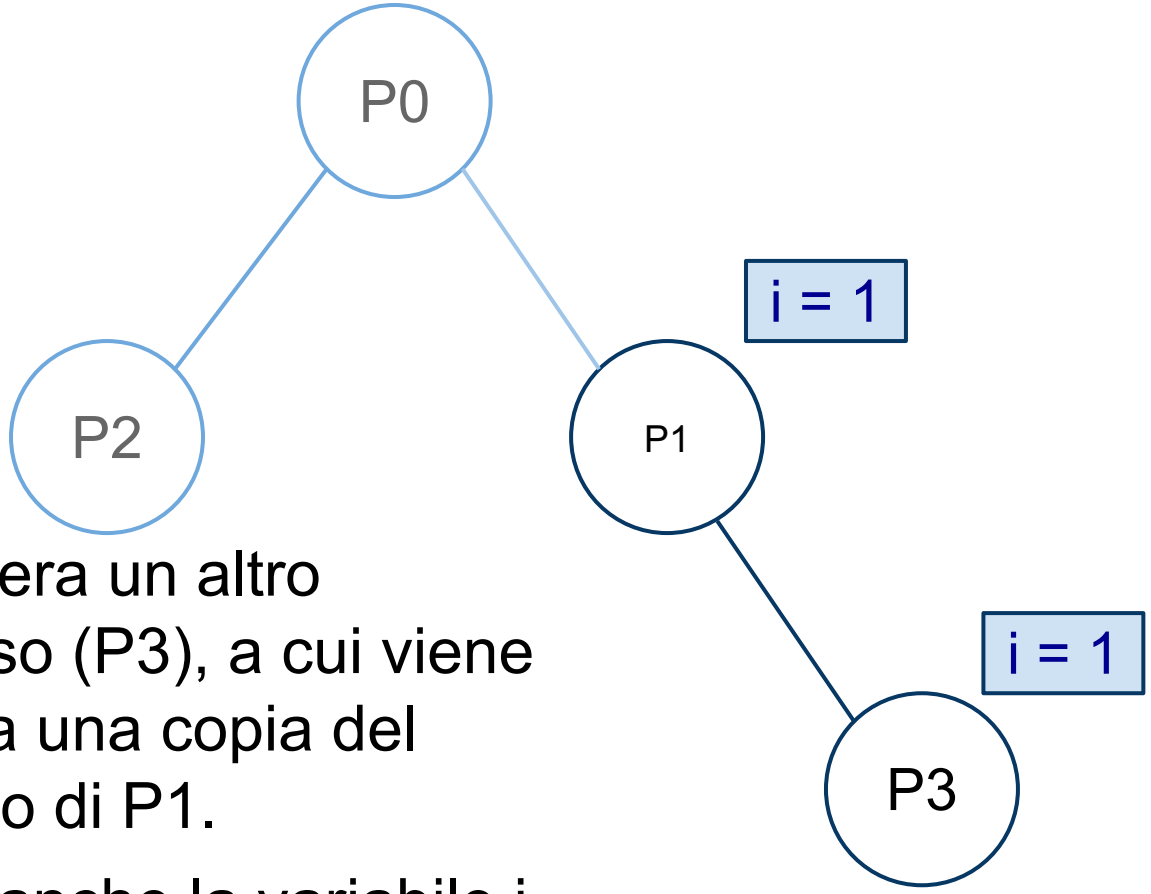
Processo P1

```
void main(int argc, char *argv[]) {
  int i, j, k, pid, status, n_children;
  char term;
  n_children = atoi(argv[1]);
  term = argv[2][0];
  for ( i=0; i<n_children; i++ ) {
    pid = fork();
    if ( pid == 0 ) { // Eseguito dai figli
      if ( term == '1' )    exit(0);
    }
    else if ( pid > 0 ) { // Eseguito dal padre
      printf("%d: child created with PID %d\n",
             getpid(), pid);
    }
    else {
      perror("Fork error:");
      exit(1);
    }
  }
}
```

i = 1

P1 esegue un'altra fork!

Simulazione di esecuzione (6/7)



P1 genera un altro processo (P3), a cui viene passata una copia del contesto di P1.

Quindi anche la variabile i inizializzata a 1

Morale

- Quando si usa la *system call* **fork()**, bisogna sempre tener presente che i dati del processo padre vengono duplicati nel processo figlio e che la sua esecuzione prosegue secondo quanto descritto nel codice (almeno inizialmente condiviso) del programma.

- Trascurare questo "dettaglio" può portare a comportamenti indesiderati

Esercitazione 2 - Obiettivi

- Utilizzo delle system call fondamentali:
  - **fork**
  - **exit**
  - **wait**
  - **exec**

👉**Ai fini del bonus occorre svolgere gli esercizi 1 e 2 (almeno uno dei due!)**

Gli esercizi 3 e 4 non determinano l'attribuzione del bonus ma sono **fortemente raccomandati**!

Esercizio 1 (1/2)

Si realizzi un programma concorrente per l'analisi del log di sistema di un ascensore. Il programma dovrà prevedere la seguente interfaccia:

```
./ascensore fermate ultimoPiano
```

- **`fermate`** è un intero positivo che rappresenta il numero totale di fermate effettuate dall'ascensore in una giornata.
- **`ultimoPiano`** è un intero positivo che indica il numero corrispondente all'ultimo piano

Il processo padre P0 deve inizializzare in modo casuale un array di interi di lunghezza pari a **`fermate`** i cui valori siano compresi nell'intervallo [0,**`ultimoPiano`**] estremi inclusi (0 rappresenta il piano terra). Ogni elemento dell'array rappresenta il numero del piano a cui si è fermato l'ascensore.

Esempio: Il sistema simula 10 fermate dell'ascensore. La prima viene effettuate al piano 1, la seconda al piano 0, la terza al piano 2, ecc…

1
0
2
4
0
5
4
4
0
3

Esercizio 1 (2/2)

Come prima cosa il processo P_0 stamperà a video l'array generato.

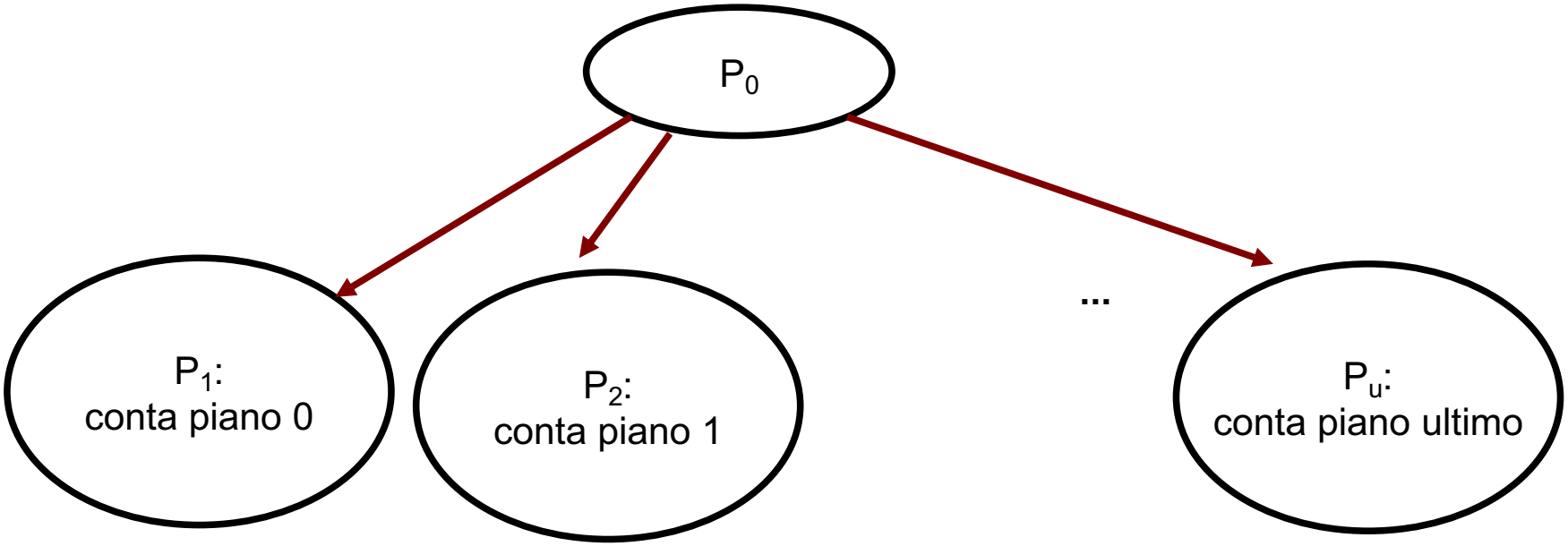
Successivamente creerà un numero di processi pari a `ultimoPiano+1`: **un processo figlio per ogni piano**.

Ogni figlio $\mathbf{P_i}$ avrà il compito di contare il numero di occorrenze del piano **i** nel log di sistema dell'ascensore.

Il valore ottenuto dovrà essere comunicato al padre contestualmente alla terminazione.

Il padre P_0, per ogni figlio P_i terminato, ne stamperà a video il **pid**, l'**indice i corrispondente al piano,** il numero di occorrenze (valore calcolato dal processo P_i)

Gerarchia



Richiami e suggerimenti

- Generazione numeri casuali: **rand()** e **srand()**:

```
#include <stdlib.h>
#include <time.h>
#define MAX 100
main()
{       int x; //numero da generare
        srand(time(NULL)); // inizializzazione generatore
        x=rand()%MAX; // x è un numero compreso tra 0 e 99
        printf("valore casuale: %d\n",x);
}
```

- Come può un figlio **trasferire un risultato al padre**? Come fa il padre ad acquisire ogni risultato ed associarlo a un particolare figlio?
  - 👉 Ripassare **exit** & **wait**
  - 👉 Il padre deve ricordarsi a quale **i** corrisponde il pid di ogni figlio

Esercizio 2

Si realizzi un programma concorrente con finalità analoghe a quelle dell'esercizio precedente. Il programma dovrà prevedere la seguente interfaccia:

```
./analisi_piano nomefile piano
```

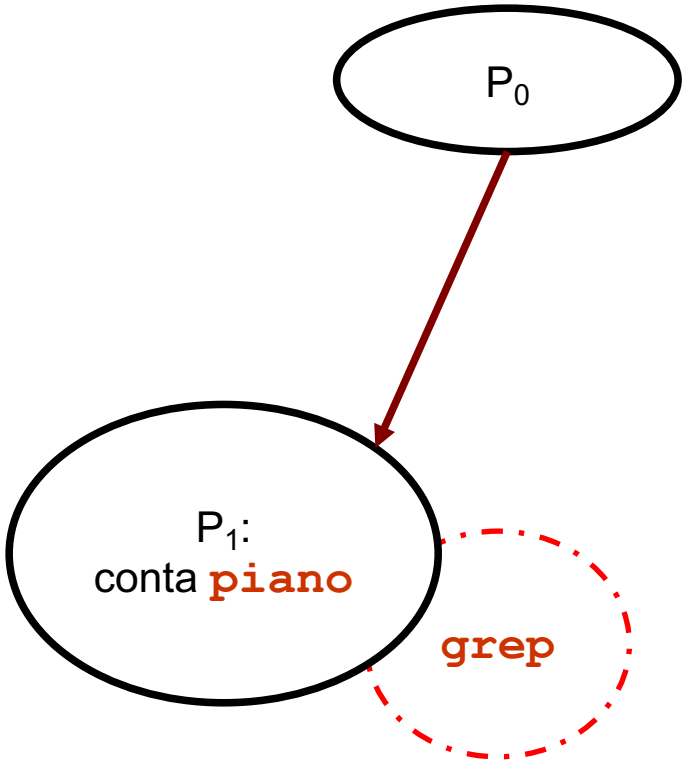
- **`nomefile`** è il nome assoluto di un file contenente il log di sistema dell'ascensore, i.e., dovrà riportare uno dopo l'altro, su righe diverse i piani a cui si è fermato l'ascensore (stesso contenuto dell'array generato randomicamente nell'esercizio precedente)
- **`piano`** è un intero positivo che indica un piano

Il processo padre P0 deve lanciare un unico processo figlio P1 deputato a contare le occorrenze di **`piano`** nel file **`nomefile`**

Il processo P1 deve eseguire tale operazione avvalendosi del comando **grep**. Si veda il man di grep per individuare l'opzione che permette di contare il numero di righe.

P0 deve attendere il completamento di P1 e stamparne a video lo stato di terminazione (volontaria/involontaria)

Gerarchia



Esercizio 3 (1/2)

Scrivere un programma C con la seguente interfaccia:

```
./ese22 dir_1 dir_2 file1 file2 … fileN
```

Dove:

- **`dir_1`** e **`dir_2`** sono nomi assoluti di directory (distinte ed entrambe esistenti).
- **`file1`**,...., **`fileN`** sono nomi relativi di file di testo contenuti nella directory **`dir_1`**;

Il processo padre deve **generare N processi figli (P1,..PN),** uno per ciascun file dato **`fileI`** (I=1..N)

Esercizio 3 (2/2)

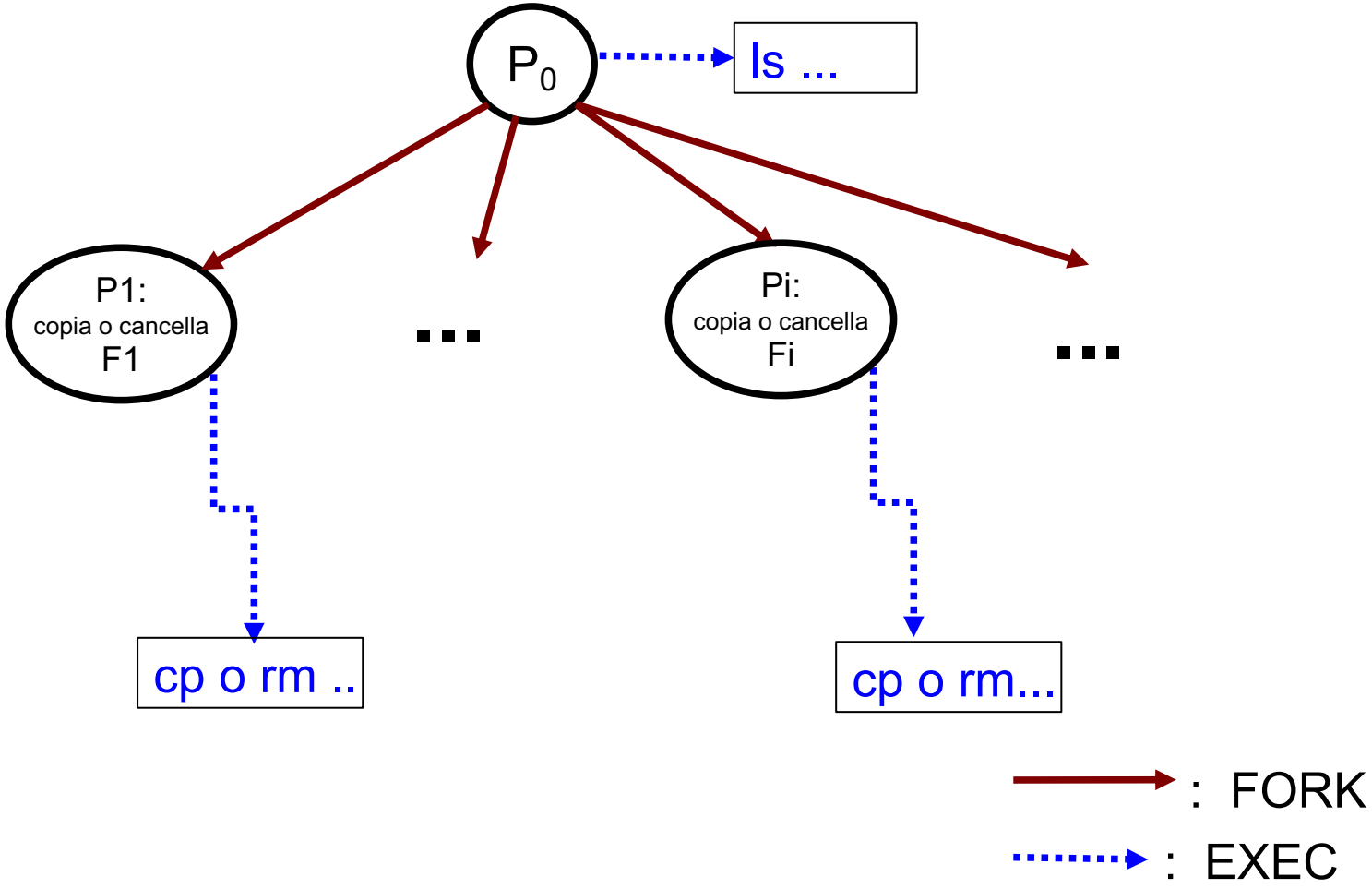
Il comportamento di ogni **processo figlio PI** dipende dal valore del proprio pid:

- se il pid di PI è **pari**, il figlio produce una copia del file **`fileI`** nella directory **`dir_2`** (usare il comando **`cp`**)
- se il pid di PI è **dispari**, il figlio cancella **`fileI`** dalla directory **`dir_1`** (usare il comando **`rm`**)

Il **processo padre** dovrà comportarsi come segue:

- una volta **terminati volontariamente tutti i figli**, dovrà stampare sullo standard output l'elenco di tutti i file contenuti nella directory **`dir_2.`** (usare il comando **`ls`**)
- Nel caso in cui almeno un figlio Pi terminasse **involontariamente**, il padre dovrà **stampare** un messaggio di errore contenente **il pid di Pi.**

Schema di generazione



Esercizio 4 (1/2)

Scrivere un programma C con la seguente interfaccia:

```
/ese23 dir_1 dir_2 file1 file2 … fileN
```

Dove:

- **`dir_1`** e **`dir_2`** sono nomi assoluti di directory (distinte ed entrambe esistenti).
- **`file1`**,...., **`fileN`** sono nomi relativi di file di testo contenuti nella directory **`dir_1`**;

Il processo padre (P0) deve **creare una gerarchia di 2*N processi** (figli e/o nipoti), 2 per ciascun file di testo.

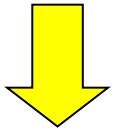
Esercizio 4 (2/2)

Per ogni **filel** (I=1,..N):

- uno dei figli/nipoti si incaricherà di **copiare** filel nella directory **`dir_2`** (usare il comando **`cp`**)
- un altro figlio/nipote (DISTINTO dal precedente) dovrà **rinominare** il file Filel con il proprio pid (usare il comando **`mv`**) all'interno della directory **dir_1**

Vincoli di sincronizzazione

- I processi figli possono essere messi in esecuzione in maniera tra loro **concorrente**,
- I processi **nipoti** possono essere messi in esecuzione in maniera tra loro **concorrente**, ma...
- La **copia** di fileI in dir_2 **deve avvenire prima della rinominazione** del file dalla directory dir_1 --> il processo che cancella deve sincronizzarsi col processo che copia



- ogni processo che deve eseguire mv ATTENDE il termine dell'esecuzione del corrispondente processo incaricato della copia --> **relazione di gerarchia**

Schema di generazione

Con gli strumenti visti finora, la sincronizzazione tra due processi può essere realizzata solo facendo in modo che il processo padre attenda il figlio.

Quindi:

- Il padre P0 genera i processi figli che devono rinominare i file
- ogni figlio genera un nipote dedicato alla copia e si mette in attesa della sua terminazione, per poi procedere con il mv.

Schema di generazione