# Seconda Esercitazione

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<br>• **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<br>• 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.**<br>• Raccoglie lo stato di terminazione di un figlio<br>• 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<br>• **NON** crea processi figli |

# Esercitazione 2 - Obiettivi

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

👉**Ai fini del bonus occorre svolgere l'esercizio 2.1**

Gli esercizi 2.2. e 2.3 non determinano l'attribuzione del bonus ma sono **fortemente raccomandati**!

# Esercizio 2.1 (1/2)

Si realizzi un programma concorrente che analizza le consegne effettuate da una piccola azienda di logistica che ha **F** fattorini. Il programma dovrà prevedere la seguente interfaccia:

```
./analisi_consegne N F
```

- **N** è un intero positivo che rappresenta il numero totale di consegne effettuate in un certo giorno;
- **F** è un intero positivo che rappresenta il numero di fattorini

Il processo padre P0 deve inizializzare in modo casuale un array di **N** interi con valori compresi nell'intervallo [0,**F-1**] (estremi inclusi). Ogni elemento dell'array rappresenta una consegna e il suo valore indica il fattorino che l'ha eseguita.

Ogni valore rappresenta una consegna effettuata da un fattorino. Esempio:

| 1 | 1 | 2 | 1 | 2 | 0 | 1 |
|---|---|---|---|---|---|---|

- In questa giornata, il fattorino 1 ha fatto quattro consegne, il 2 ne ha fatte due, il fattorino 0 ne ha fatta una.

# Esercizio 2.1 (2/2)

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

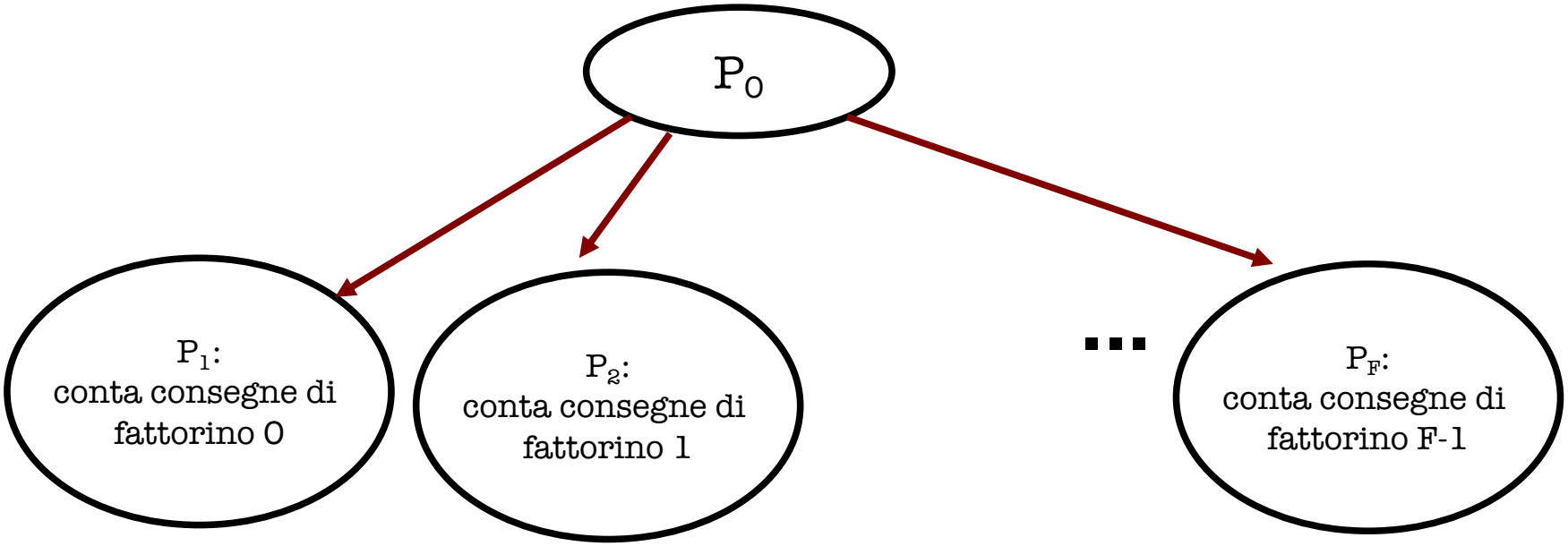
Successivamente creerà **F processi** figli (**uno per ogni fattorino**): $P_1$, $P_2$, ..., $P_F$.

Ogni figlio $\mathbf{P_i}$ avrà il compito di contare il numero di consegne effettuate dal fattorino **i**-esimo.

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** del fattorino e il numero di consegne che ha fatto (valore calcolato dal processo $P_i$)

# Gerarchia


$P_0$
$P_1$:
conta consegne di
fattorino 0
$P_2$:
conta consegne di
fattorino 1
...
$P_F$:
conta consegne di
fattorino F-1

# Richiami e suggerimenti

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

```
#include <stdlib.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 (pid e i)?
  - 👉 Ripassare exit&wait
  - 👉 Il padre deve ricordarsi a quale **i** corrisponde il pid di ogni figlio

# Esercizio 2.2 (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 2.2 (2/2)

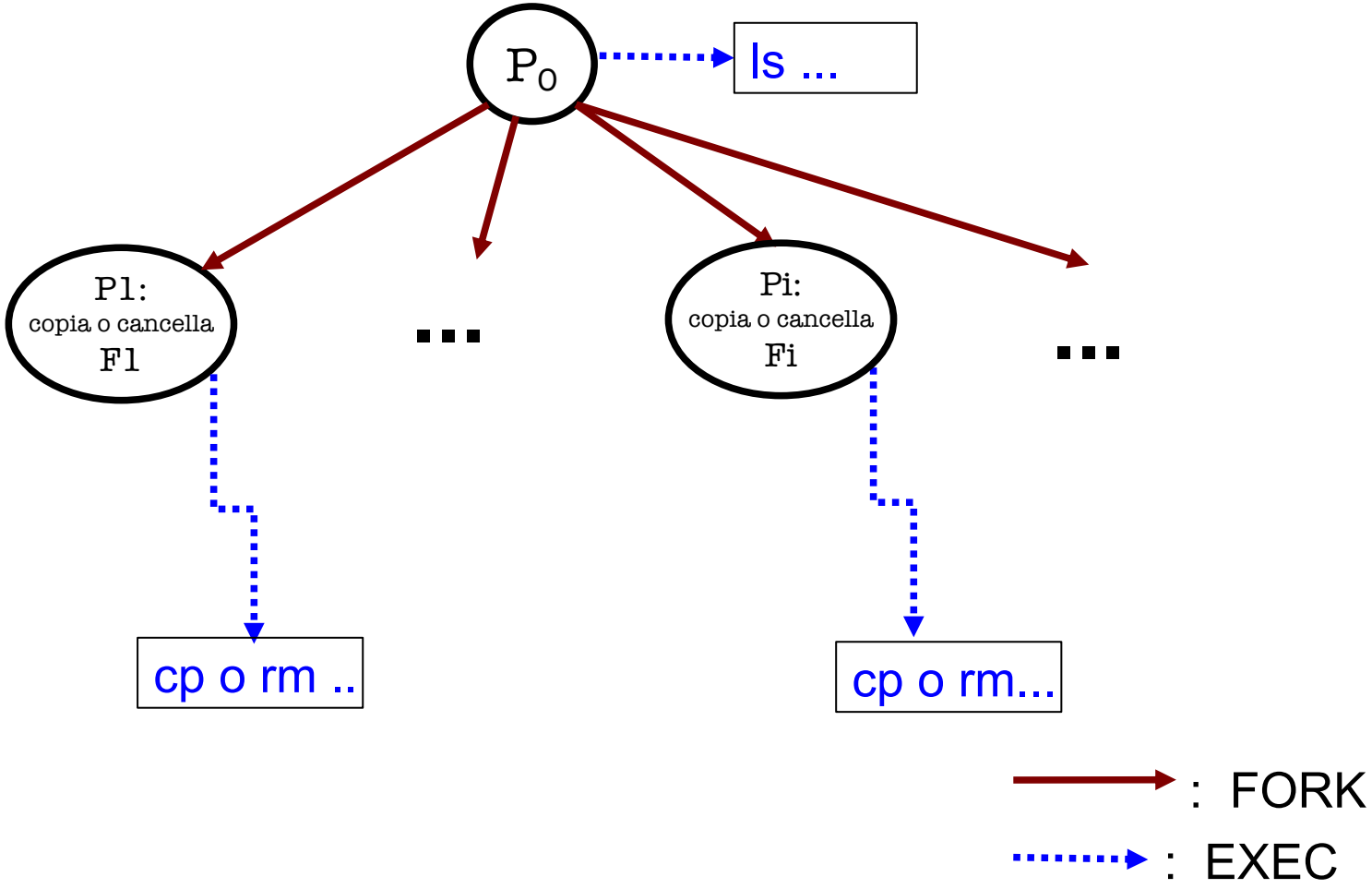
Il comportamento di ogni **processo figlio PI** dipende dal valore del proprio pid:

- 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 2.3 (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.

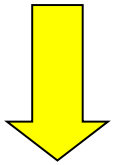
# Esercizio 2.3 (2/2)

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

- uno dei figli/nipoti si incaricherà di **copiare** fileI nella directory **`dir_2`** (usare il comando **`cp`**)
- un altro figlio/nipote (DISTINTO dal precedente) dovrà **rinominare** il file FileI 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**

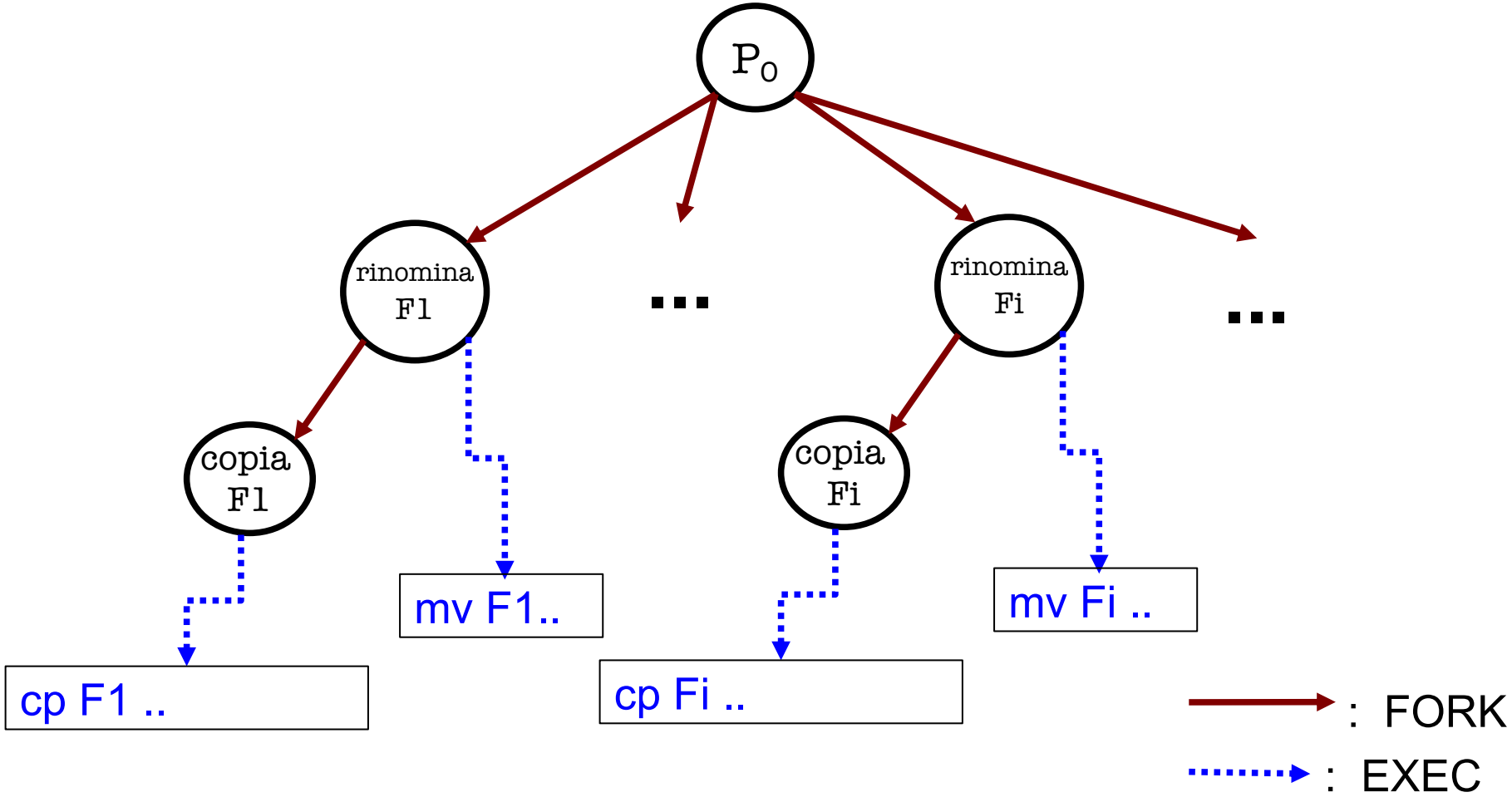
# Schema di generazione

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 PO 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



# FAQ 1:

Qual è il path assoluto del comando X?

esiste il comando shell **which**. Esempio:

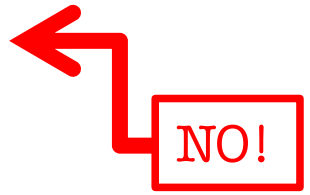
```
anna@cloud$ which gcc
/usr/bin/gcc

anna@cloud$ which ls
/bin/ls
```

# FAQ 2:

```
execl("/bin/cp",arg1,..,(char*)0);
```

Non riesco a invocare cp...




- execl prevede la seguente sintassi:

```
execl("/bin/cp",
      "cp", arg1, ...,(char*)0);
```

Path (assoluto o relativo) per raggiungere il comando

Stringa COMPLETA di invocazione del comando