

Seconda Esercitazione

Gestione di processi in Unix
Primitive Fork, Wait, Exec

System call fondamentali

fork	<ul style="list-style-type: none">Generazione di un processo figlio, che condivide il codice con il padre e eredita copia dei dati del padreRestituisce il PID (>0) del processo creato per il padre, 0 per il figlio, o un valore negativo in caso di errore
exit	<ul style="list-style-type: none">Terminazione di un processoAccetta come parametro lo stato di terminazione ($0\text{-}255$). Per convenzione 0 indica un'uscita con successo, un valore <i>non-zero</i> indica uscita con fallimento.
wait	<ul style="list-style-type: none">Chiamata bloccante.Raccoglie lo stato di terminazione di un figlioRestituisce 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	<ul style="list-style-type: none">Sostituzione di codice (e dati) del processo che l'invocaNON 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)

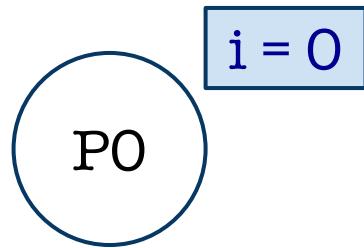
- Vediamo cosa succede durante l'esecuzione del programma
- Assumiamo:
N = 2 : Il padre genera due processi figli
term = '0' : I figli non chiamano exit
- Da ricordare:
Una volta creato, ogni figlio esegue **concorrentemente** al padre e ai fratelli a partire dall'istruzione successiva alla fork() che l'ha creato.

Processo PO

```
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)



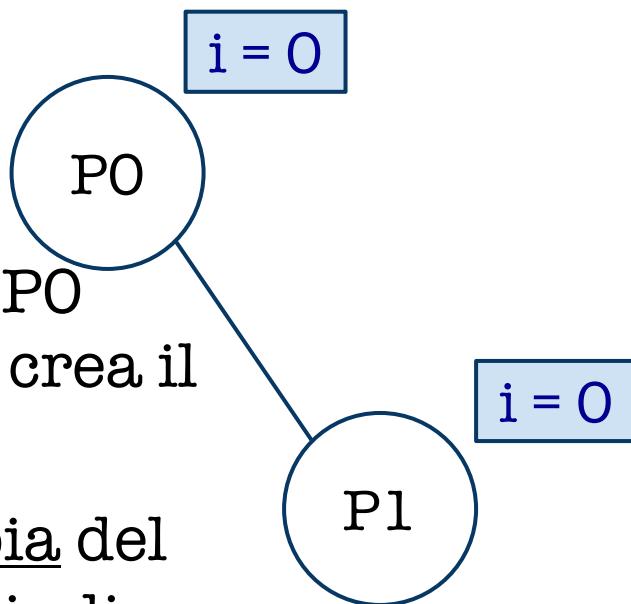
Il processo padre P0 viene
creato e inizia la prima
iterazione del for ($i=0$)

Processo PO

```
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 PO

```
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 PO e P1 è la variabile **pid**.

- **P1: pid=0**
 - **PO: pid>0** (pid del figlio)
- PO esegue la **printf**

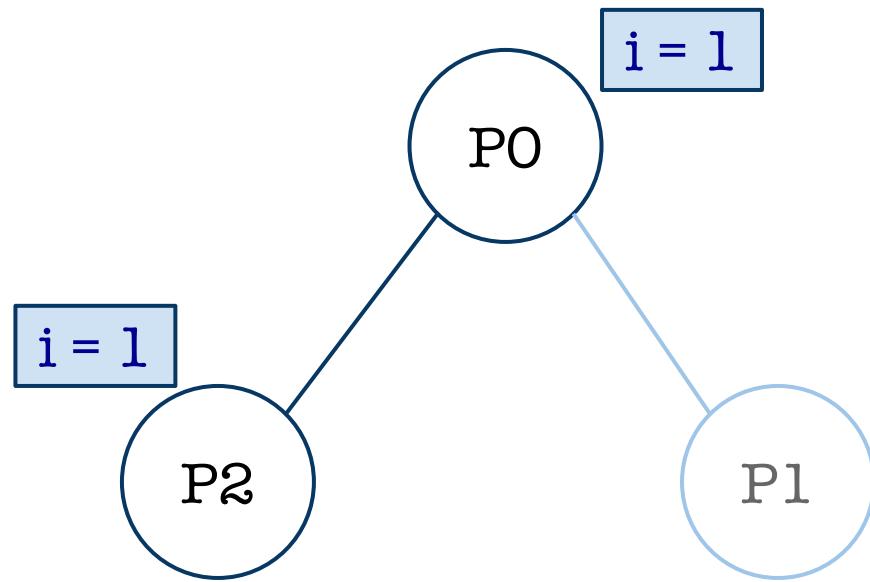
Processo PO

```
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

PO 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 PO

```
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

PO esegue ancora una
printf()

Processo PO

```
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

PO 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.

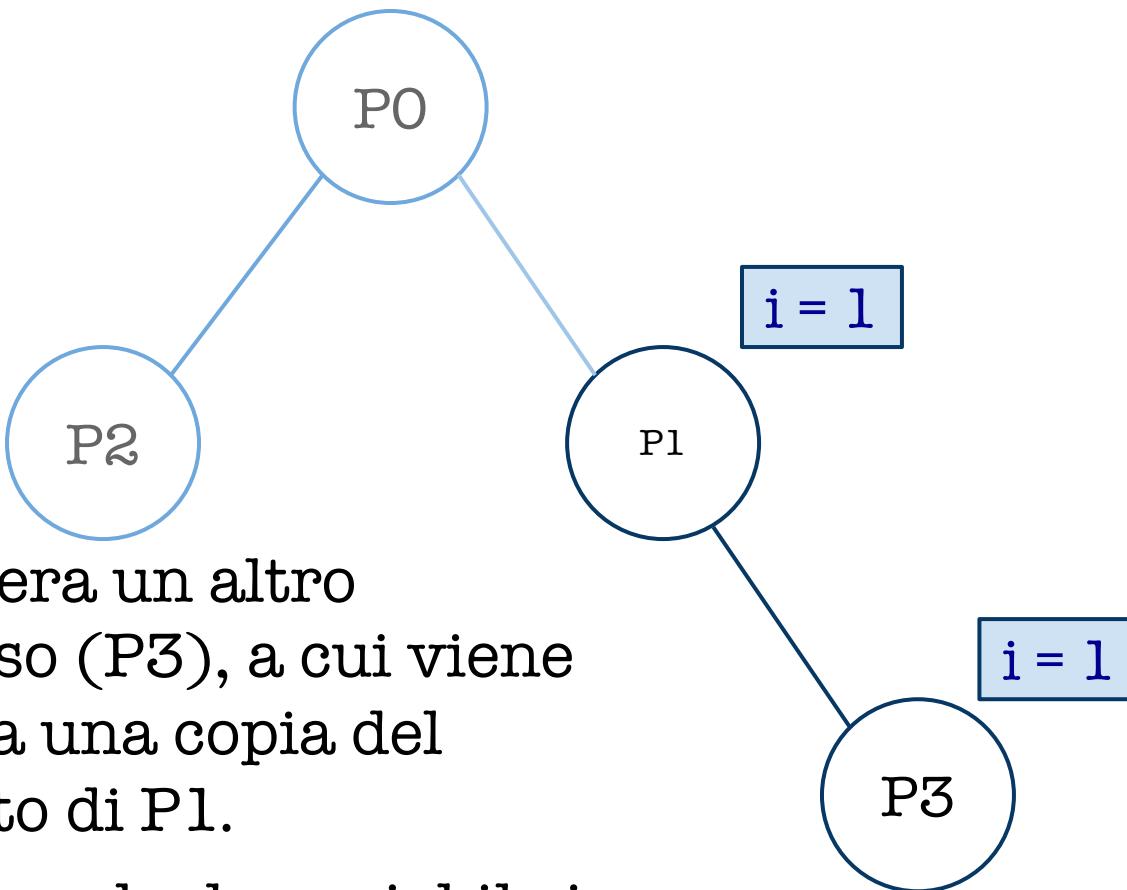
Processo P1

```
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
-

Esercizio 2.1 (1/2)

Si realizzi un programma concorrente che calcoli i voti da assegnare in pagella ad ogni studente di una classe. Il programma dovrà prevedere la seguente interfaccia:

./calcola_medi N V

Dove:

- **N** è un intero positivo che rappresenta il numero di studenti appartenenti alla classe;
- **V** è un intero positivo che rappresenta il numero di verifiche svolte dagli studenti nel periodo (es. quadrimestre) considerato.

Il processo padre **PO** deve **inizializzare con dati letti da standard input una matrice intera **M**** di dimensione **N x V**, dove ogni cella **M[i][j]** rappresenta il voto ottenuto dallo studente **i**-simo nella verifica **j**-sima.

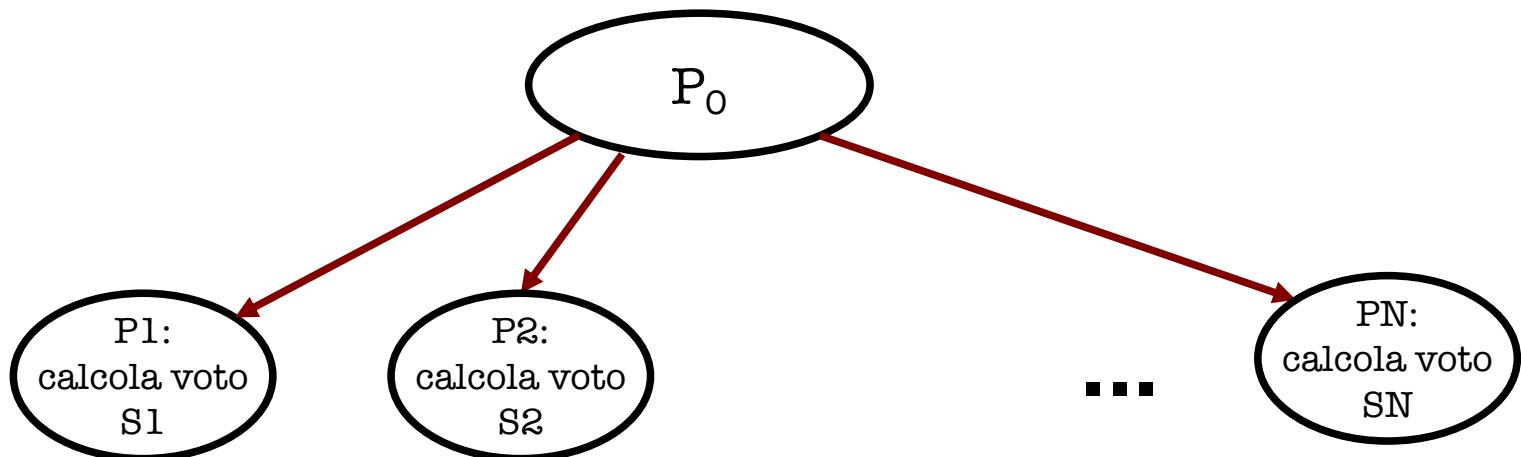
Esercizio 2.1 (2/2)

Successivamente il processo padre P_0 creerà N processi figli (uno per ogni studente) $P_1, P_2.. P_N$.

Ogni figlio P_i avrà il compito di **calcolare la media aritmetica dei voti riportati dallo studente S_i** in tutte le V verifiche; il valore ottenuto dovrà essere successivamente arrotondato all'intero più vicino e comunicato al padre contestualmente alla terminazione. Tale valore rappresenterà il voto finale dello studente S_i .

Il padre P_0 , per ogni figlio P_i terminato, ne stamperà a video il **pid**, l'**indice** i dello studente S_i (del quale P_i ha calcolato la media) ed il **voto finale** di S_i

Gerarchia



Esercizio 2.2 (1/2)

Scrivere un programma C con la seguente interfaccia:

```
./backup file1 file2 ... fileN dir_dest
```

Dove:

- **file1, ..., fileN** sono file di testo contenuti nella directory corrente;
- **dir_dest** è il nome assoluto di una directory esistente.

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

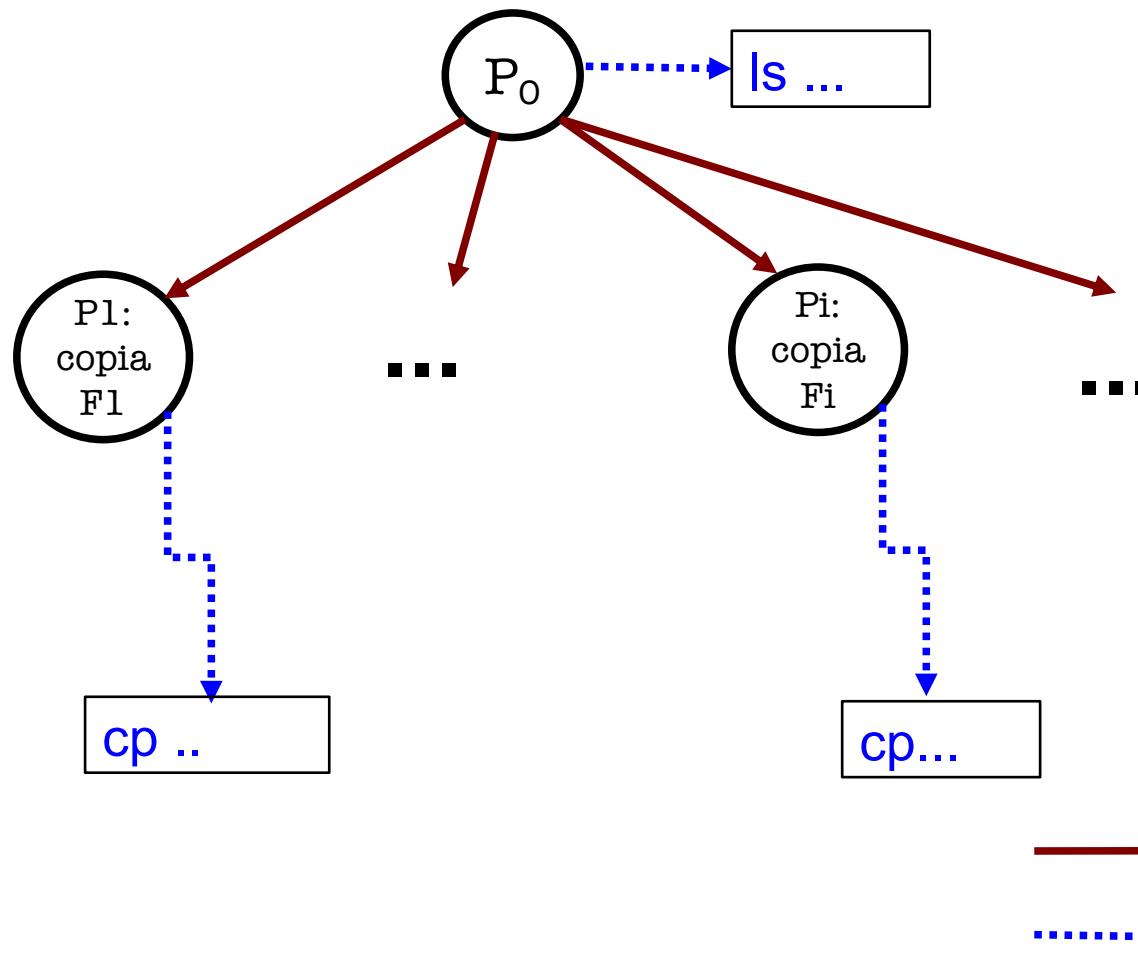
Ogni figlio P_i dovrà produrre una copia di nome **fileI.bak** del file **fileI** nella directory **dir_dest**. (usare il comando **cp**)

Esercizio 2.2 (2/2)

Il processo padre:

- una volta **terminati volontariamente tutti i figli**, dovrà stampare sullo standard output l'elenco di tutti i file contenuti nella directory **dir_dest**. (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 ed il nome del file (fileI)** assegnato al figlio terminato involontariamente.

Schema di generazione



Esercizio 2.3 (1/2)

Scrivere un programma C con la seguente interfaccia:

```
./sposta file1 file2 ... fileN dir_dest
```

Dove:

- **file1, ..., fileN** sono file di testo contenuti nella directory corrente;
- **dir_dest** è il nome assoluto di una directory esistente.

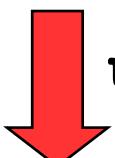
Il processo padre deve **generare 2 * N processi** (figli e/o nipoti), 2 per ciascun file di testo.

Esercizio 2.3 (2/2)

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

- uno dei figli/nipoti si incaricherà di **copiare** fileI nella directory **dir_dest**
- un altro figlio/nipote (DISTINTO dal precedente) dovrà cancellare il file FileI dalla directory corrente.

Vincoli di sincronizzazione

- I processi “**copiatori**” possono essere messi in esecuzione in maniera tra loro **concorrente**,
 - I processi “**cancellatori**” possono essere messi in esecuzione in maniera tra loro **concorrente**, **ma...**
 - La **copia** di fileI in dir_dest **deve avvenire prima della cancellazione** del file dalla directory corrente --> il processo che cancella deve sincronizzarsi col processo che copia
-  **una possibile soluzione**
- il processo cancellatore ATTENDE il termine dell'esecuzione del processo copiatore --> **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 cancellatori
- I cancellatori generano i copiatori e poi si mettono in attesa della loro terminazione.

Schema di generazione

