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 possiede **dati replicati** • **Restituisce** il PID 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** di un processo • **NON** crea processi figli

Esempio 1 - 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 1 - 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

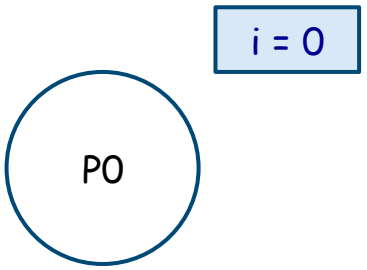
- Da ricordare:
  Una volta creato, ogni figlio esegue **concorrentemente** al padre e ai fratelli a partire dall'istruzione successiva alla fork() che l'ha creato.

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)



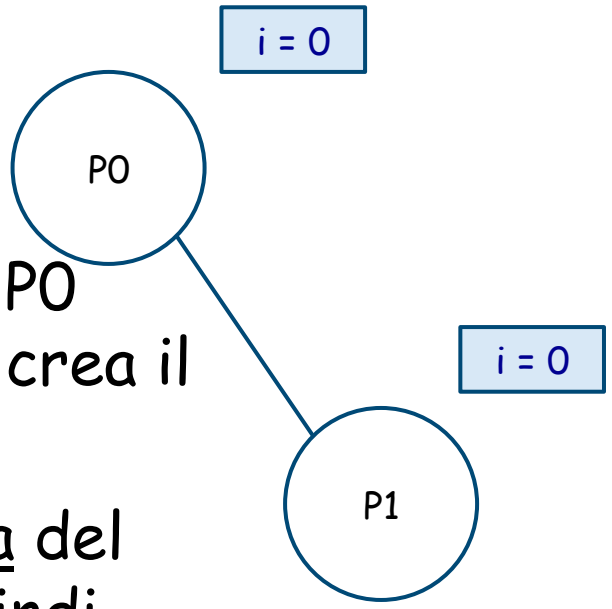
Il processo padre P0 viene creato e inizia la prima iterazione del for (i=0)

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**

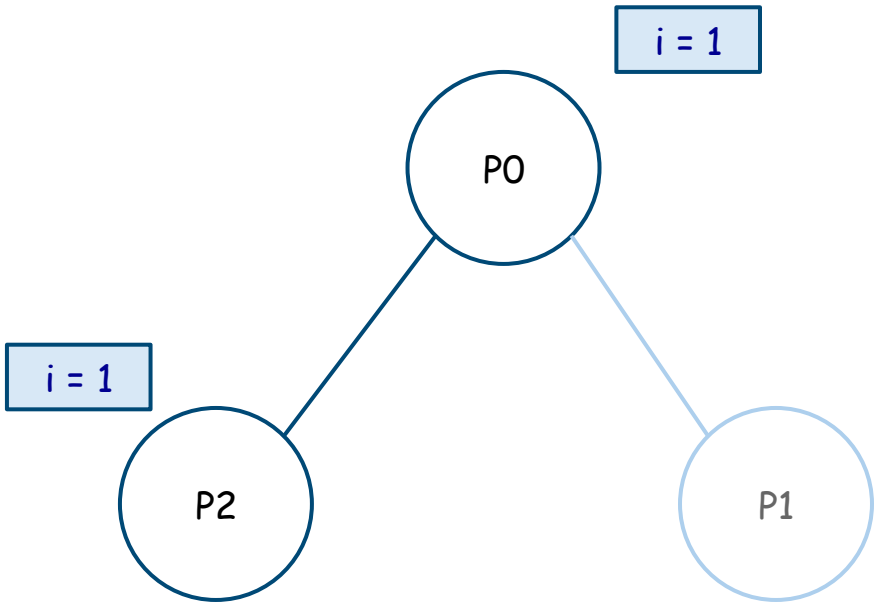
Processo P0

```
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

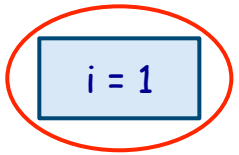
```
void main(int argc, char *argv[]) {
  int i, j, k, pid, status, n_children;
  char term;
  n_children = atoi(argv[1]);
  term = argv[2][0];
  for ( i=0; i<n_children; i++ ) {
    pid = fork();
    if ( pid == 0 ) { // Eseguito dai figli
      if ( term == '1' )    exit(0);
    }
    else if ( pid > 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);
    }
  }
}
```



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

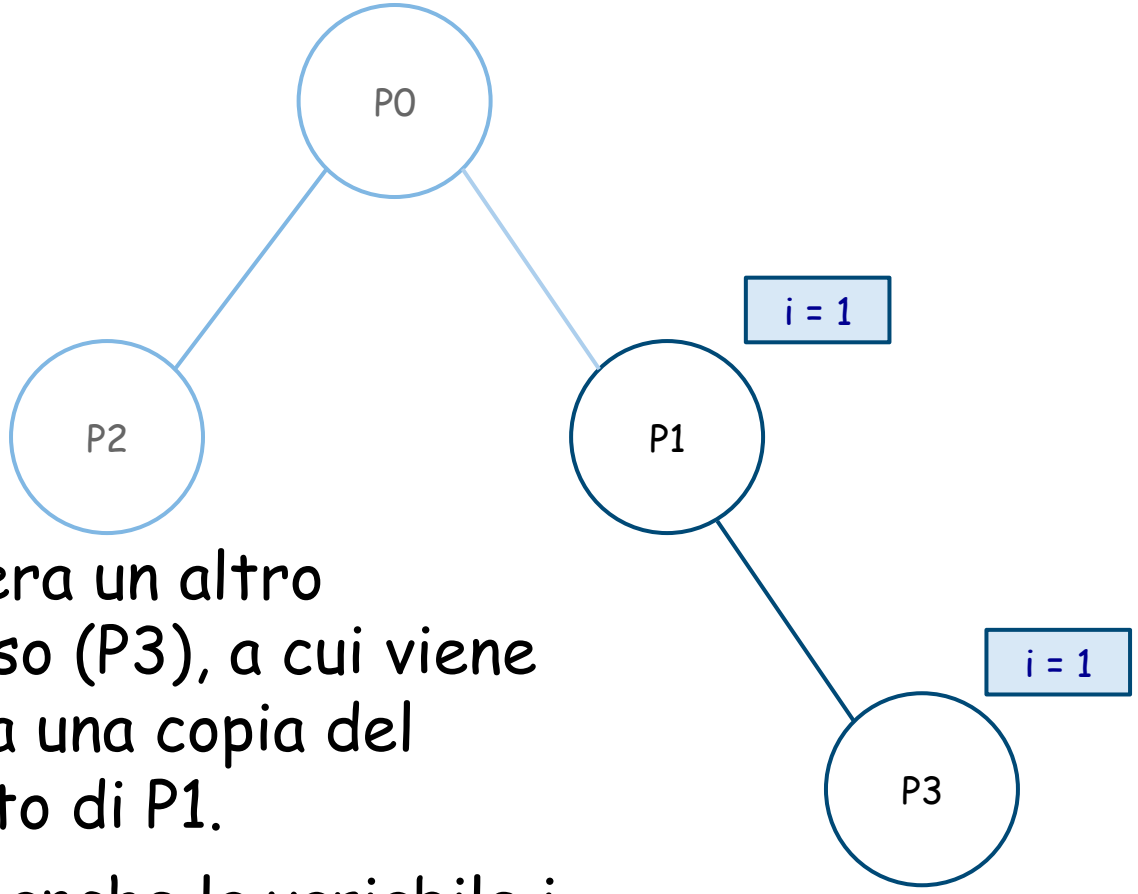
Processo P1

```
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 1

Estendere l'esempio 1:

- eliminando l'argomento term (ogni figlio fa sempre exit)
- Facendo in modo che il padre attenda la terminazione di ogni figlio, stampandone il pid e lo stato di terminazione.

```
./generatenew <N>
```

Esercizio 2

Scrivere un programma C con la seguente interfaccia:

```
./compilaEdEsegui <file1.c> <file2.c> … <fileN.c>
```

dove file1.c,...., fileN.c sono file sorgenti C.

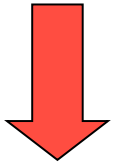
Il processo padre deve **generare 2*N processi** (figli e/o nipoti),

- 2 per ciascun sorgente; per ogni file,
- uno dei figli/nipoti si incaricherà di **compilare** il file,
- un altro figlio/nipote (DISTINTO dal precedente) di **metterne in esecuzione** l'eseguibile risultante.

Si generino i processi figli sequenzializzando il meno possibile le operazioni di compilazione ed esecuzione.

Vincoli di sincronizzazione

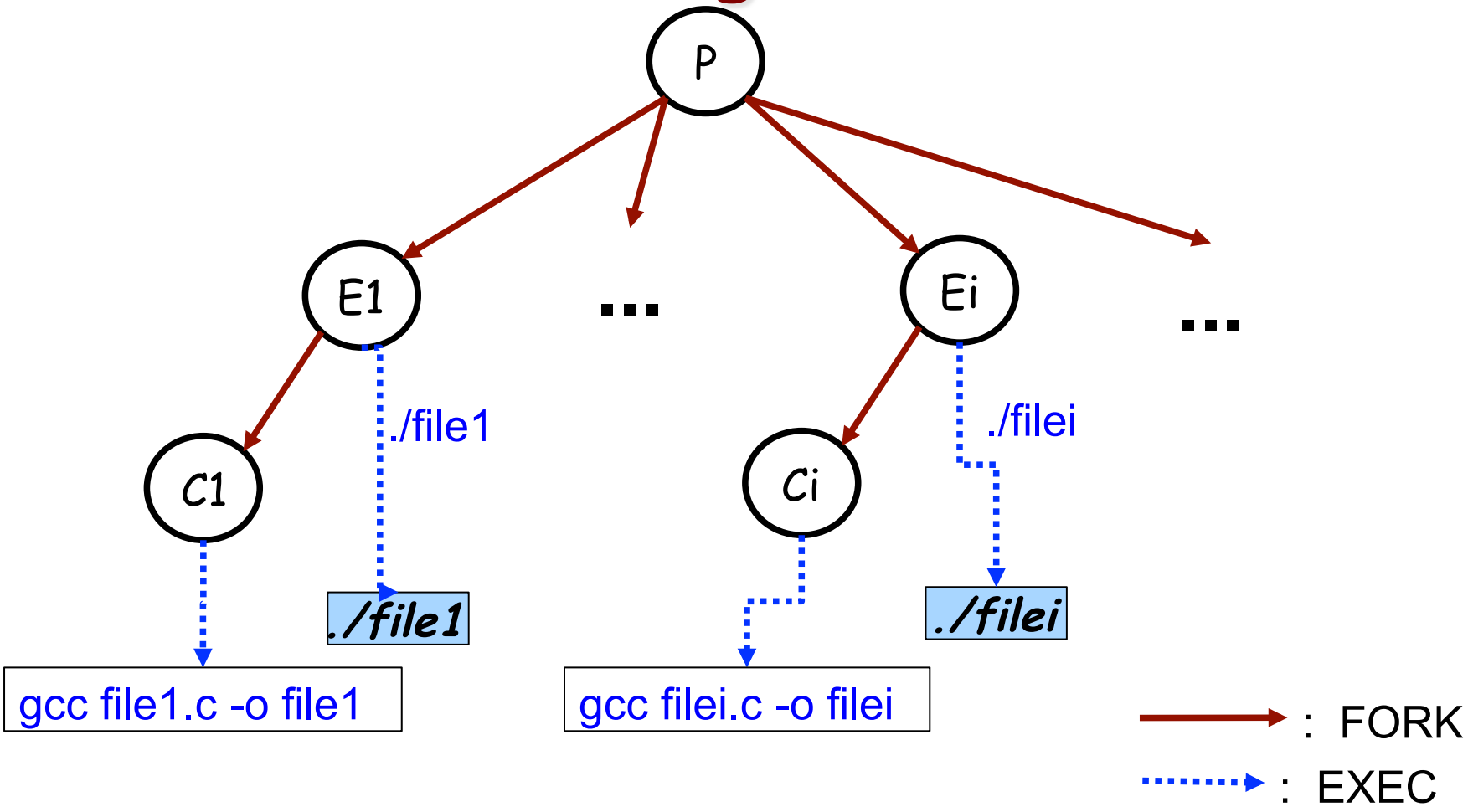
- I processi **compilatori** possono essere messi in esecuzione in maniera **concorrente**, ma...
- La **compilazione deve avvenire prima dell'esecuzione** --> il processo che esegue deve sincronizzarsi col processo che compila



una possibile soluzione

- il processo esecutore ATTENDE il termine dell'esecuzione del processo compilatore --> **relazione di gerarchia**

Schema di generazione



Schema di generazione

Dati gli strumenti visti fin'ora, 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 esecutori
- gli esecutori generano i compilatori e poi si mettono in attesa della loro terminazione.

Esercizio 3

Si realizzi un programma, che, utilizzando le system call del sistema operativo UNIX, soddisfi le seguenti specifiche.

Sintassi di invocazione:

```
-:$ ./eseguiComandi K COM1 COM2 ... COMN
```

Significato degli argomenti:

- **eseguiComandi** è il nome del file eseguibile associato al programma.
- **COM1,COM2,…,COMN** sono N stringhe che rappresentano il nome di un file (per semplicita`, si supponga che il direttorio di appartenenza del file COM sia nel PATH)
- **K** e` un valore intero positivo (minore di N)

Specifiche

Il processo iniziale (P0) deve **mettere in esecuzione** gli N comandi passati come argomenti, secondo la seguente logica:

- i primi **K** comandi passati come argomenti dovranno essere eseguiti **in parallelo** da **altrettanti figli** di P0
- Al termine dei primi K processi, i restanti **N-K** comandi dovranno essere eseguiti **in sequenza** da **altrettanti figli e/o nipoti** di P0