AXO

Architetture dei Calcolatori e Sistema Operativo

programmazione di sistema

Sistema Operativo (SO)

Il Sistema Operativo (SO) è un insieme di programmi (moduli software) che svolgono funzioni di servizio nel calcolatore.

- costituisce la parte essenziale del cosiddetto **software di sistema** (o di base) in quanto non svolge funzioni applicative, ma ne costituisce un supporto

Lo scopo del sistema operativo è quello di:

- mettere a disposizione dell'utente una macchina virtuale in grado di eseguire comandi dati dall'utente, utilizzando una macchina reale, di livello inferiore
- mettere a disposizione del software applicativo (e quindi del programmatore) un insieme di servizi di sistema, invocabili tramite chiamate di sistema (system call)
- controllare le risorse fisiche del sistema e creare parallelismo

SO come creatore di parallelismo – I

- Nel modello di esecuzione sequenziale, l'esecuzione di N programmi avviene in sequenza, attivando lo *i-esimo* programma solo dopo avere terminata l'esecuzione dello (*i – 1*)-*esimo* programma.
  - questo garantisce che non possano esistere interferenze nell'esecuzione dei vari programmi

esecuzione sequenziale vs esigenze dei sistemi di calcolo

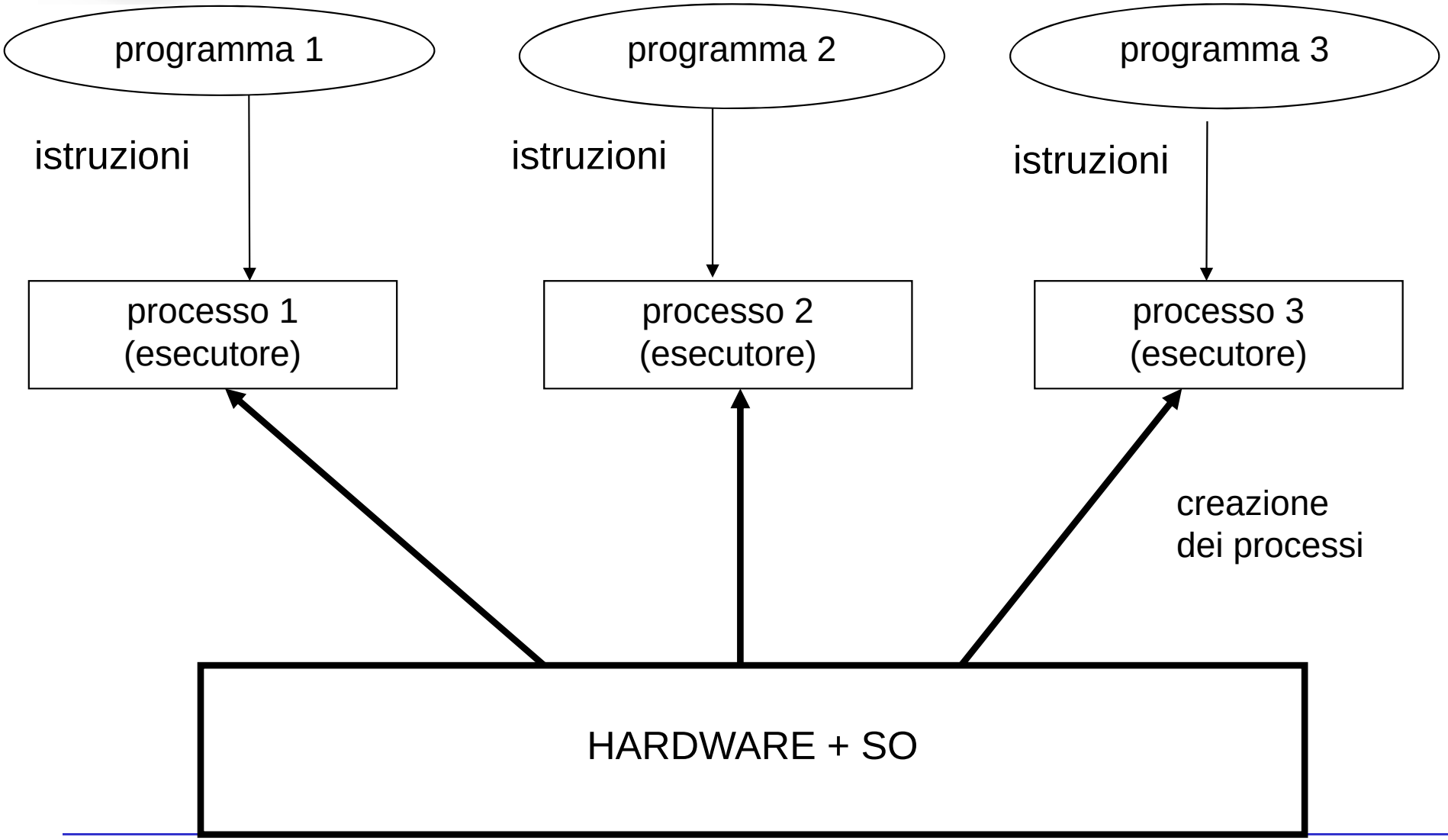
Necessità di parallelismo

- applicazioni distribuite
- calcolatore multiutente
- applicazioni multiple del singolo utente

SO come creatore di parallelismo – II

- *Modello di esecuzione parallela:*
  - garanzia di soddisfacimento di due obiettivi contrastanti
    - parallelismo
    - virtualizzazione del sistema di calcolo (esecuzione senza interferenze)
  - gli obiettivi vengono soddisfatti tramite la creazione dinamica di tanti ESECUTORI quanti sono i programmi da eseguire in parallelo
    - gli esecutori creati dinamicamente vengono chiamati processi
    - un processo è un esecutore completo
- Il Sistema Operativo è in grado di creare processi indipendenti e mette a disposizione del programmatore delle chiamate di sistema (system call) che permettono di creare ed eliminare processi

processi dal punto di vista del programmatore



virtualizzazione delle risorse di calcolo

- Il sistema operativo gestisce eventuali conflitti di accesso contemporaneo alle risorse “reali” condivise, creando delle risorse *virtuali* e accodando i processi richiedenti.

- Accesso a periferiche condivise da più processi:
  - il codice eseguibile di un programma non può contenere istruzioni che accedano direttamente a periferica, ma deve utilizzare certe *system call* che svolgono servizi di sistema opportuni
    - p. es. vedi implementazione delle librerie di Ingresso / Uscita del linguaggio C

programmazione di sistema e chiamate di sistema

- Con programmazione di sistema si intendono le tecniche utilizzate nella scrittura di programmi utente che interagiscono strettamente con il sistema operativo e che utilizzano i servizi (*system call*) messi a disposizione da quest'ultimo
- Categorie principali di chiamate di sistema:
  - gestione di processi
  - gestione di file
  - gestione di cartelle e file system
  - segnalazione
  - protezione
  - gestione di ora e data
  - e altre ancora ... (correntemente Linux ne ha 322)

La programmazione di sistema

primitive per la gestione dei processi

aspetti generali relativi ai processi

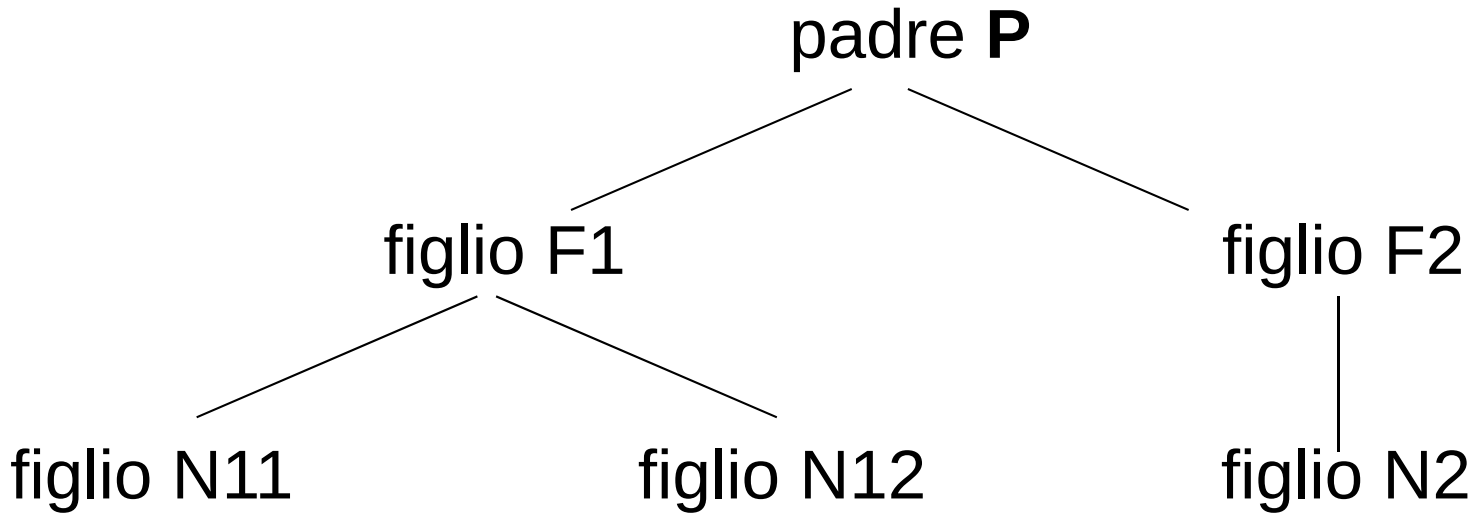
- Ciascun processo è identificato in modo univoco da un PID (Process IDentifier), di solito un numero intero positivo non nullo.
- Tutti i processi sono creati da altri processi e quindi hanno un processo *padre* (unica eccezione: il processo "init", primo processo creato all'avviamento del SO, che non ha un processo padre).
- Dal punto di vista del programmatore di sistema, la memoria di lavoro associata a un processo può essere vista come costituita da tre segmenti fondamentali (poi raffinabili in più segmenti):
  - **segmento codice** (*text segment*): contiene l'eseguibile del programma
  - **segmento dati** (*data segment*): contiene tutte le variabili del programma
    - globali e statiche, locali allocate in pila ,e variabili dinamiche create tramite `malloc()`
  - **segmento di sistema** (*system data segment*): contiene i dati non gestiti esplicitamente dal programma in esecuzione, ma dal SO (p. es. la "tabella dei file aperti" e i descrittori di socket)

operazioni sui processi – I

- Le chiamate di sistema principali (*system call*), dette anche *primitive*, per la gestione dei processi consentono di:
  - generare un processo figlio (*child* o anche *slave*) copia del processo padre (*parent* o anche *master process*) in esecuzione
  - terminare un processo *figlio* (restituendo un codice al processo *padre)*
  - attendere la terminazione di un processo *figlio*
  - sostituire il codice di un processo in esecuzione, cioè sostituire il programma eseguito da un processo

operazioni sui processi – II

- un processo figlio *F* può a sua volta generare un ulteriore processo figlio *N*
- si stabilisce così una gerarchia di processi



P è padre di F1 e F2

F1 è padre di N11 e N12

F2 è padre di N2

generazione di processi: fork ( )

La funzione *fork*:

- crea un processo *figlio* (o *child*) identico al processo padre (o *parent*)
- il figlio è una copia identica del padre all'istante della *fork*
- **vengono duplicati il segmento dati e quello di sistema**, quindi le variabili, i file aperti, e i descrittori di periferica utilizzati sono duplicati nel figlio
  - all'istante della *fork*, i segmenti dei processi padre e figlio contengono gli stessi valori (tranne che per il valore restituito dalla *fork* stessa, vedi dopo), però l'evoluzione indipendente dei due processi (può) modifica(re) i segmenti
- l'unica differenza tra figlio e padre è il valore restituito dalla *fork*:
  - **nel padre**, la funzione **restituisce il *pid* del processo figlio** appena generato, quindi il padre conosce il *pid* del figlio (rstituisce –1 in caso di errore (*fork* non eseguita)
  - **nel figlio** la *fork* **restituisce il valore 0**
- prototipo *fork*:

  `pid_t fork (void)` dove `pid_t` è un tipo predefinito

terminazione dei processi: exit ( )

- La funzione *exit*:
  - termina il processo corrente (ossia quello che esegue *exit*)
  - ma un processo può terminare anche in assenza di una exit esplicita (in tale caso c'è sempre una *exit* implicita forzata, come si vedrà)
  - prototipo *exit*:

```
void exit (int)
```

  - il codice di terminazione di *exit* (cioè l'intero passato come parametro) viene "restituito" al padre
  - se il processo che termina non ha più un processo padre (è già terminato) il valore viene restituito all'interprete comandi del SO
  - per l'uso del codice di terminazione, si veda anche la funzione *wait*

acquisizione del PID: getpid ( )

- La funzione *getpid*:
  - consente a un processo di conoscere il valore del proprio *pid*
  - prototipo *getpid*:

```
pid_t getpid (void)
```

esempio semplice

```
#include <stdio.h>
#include <sys/types.h>

void main (int argc, char * argv []) {
   pid_t pid;
   int retstatus = 0;

   pid =  fork ( );
   if (pid == 0) {   /* pid == 0, sono nel figlio */
        printf (“Sono il processo figlio.\n”);
        retstatus = 1;
        exit (retstatus);
   } else {          /* pid != 0, sono nel padre */
        printf (“Sono il processo padre.\n”);
        exit (retstatus);
   } /* end if */
} /* end main */
```

esempio 1- getpid

```
#include <stdio.h>
#include <sys/types.h>

void main ( ) {
   pid_t pid;
   printf (“Prima della fork: PID = %d\n”, getpid ( ));
   pid = fork ( );
   if (pid == 0) {     /* PROCESSO FIGLIO */
        printf (“FIGLIO: PID = %d\n”, getpid ( ));
        exit (0);
   } else {            /* PROCESSO PADRE */
        printf (“PADRE: PID = %d\n”, getpid ( ));
        printf (“PADRE: PID DEL FIGLIO = %d\n”, pid);
        exit (0);
   } /* end if */
} /* end main */
```

esempio 1- getpid - esecuzione

```
Prima della fork: PID = 3375
FIGLIO: PID = 3399
PADRE: PID = 3375
PADRE: PID DEL FIGLIO = 3399
```

Dal risultato dell'esecuzione si deduce che l'ordine di esecuzione dei processi è stato: prima figlio poi padre.

Nota bene:

- quando la *fork* è stata eseguita è stato creato il secondo processo, e l'esecuzione può proseguire o con il processo padre per primo oppure con il processo figlio per primo

esempio 2: generazione di due figli

```
void main ( ) {
    pid_t pid1, pid2;
    pid1 = fork ( );
    if (pid1 == 0) {        /* PRIMO PROCESSO FIGLIO */
          printf (“FIGLIO 1: PID = %d\n”, getpid ( ));
          printf (“FIGLIO 1: eseguo exit.\n”);
          exit (0);
    } else {                /* PROCESSO PADRE */
          pid2 = fork ( );
          if (pid2 == 0) { /* SECONDO PROCESSO FIGLIO */
                  printf (“FIGLIO 2: PID = %d\n”, getpid ( ));
                  printf (“FIGLIO 2: eseguo exit.\n”);
                  exit (0);
          } else {          /* PROCESSO PADRE */
                  printf (“PADRE: PID = %d\n”, getpid ( ));
                  printf (“PADRE: PID DEL FIGLIO 1 = %d\n”, pid1);
                  printf (“PADRE: PID DEL FIGLIO 2 = %d\n”, pid2);
                  exit (0);
          } /* end if */
    } /* end if */
} /*end if */
```

esempio 2: generazione di due figli

Nell'ipotesi che l'ordine di esecuzione sia figlio1, padre, figlio2, padre:

```
FIGLIO 1: PID = 4300
FIGLIO 1: eseguo exit.
FIGLIO 2: PID = 4335
FIGLIO 2: eseguo exit.
PADRE: PID = 3375
PADRE: PID DEL FIGLIO 1 = 4300
PADRE: PID DEL FIGLIO 2 = 4335
```

altro esempio di generazione di due figli

```
void main ( ) {
    pid_t pid;
    pid = fork ( );
    if (pid == 0) { /* PRIMO PROCESSO FIGLIO */
          printf (“(1) Sono il primo figlio con pid: = %d\n”, getpid ( ));
          exit (0);
    } else {                    /* PROCESSO PADRE */
          printf (“(2) Sono il processo padre.\n”);
          printf (“(3) Ho creato un primo figlio con pid: = %d\n”, pid);
          printf (“(4) Il mio pid e’: = %d\n”, getpid ( ));
          pid = fork ( );
          if (pid == 0) {      /* SECONDO PROCESSO FIGLIO */
                    printf (“(5) Sono il secondo figlio con pid: = %d\n”,
                              getpid ( ));
                    exit (0);
          } else {              /* PROCESSO PADRE */
                    printf (“(6) Sono il processo padre.\n”);
                    printf (“(7) Ho creato un secondo figlio con pid: = %d\n”,
                              pid);
                    exit (0);
          } /* end if */
     } /* end if */
} /* end main */
```

esecuzione

```
(2) Sono il processo padre.
(1) Sono il primo figlio con pid: = 695
(3) Ho creato un primo figlio con pid: = 695
(4) Il mio pid e': = 694
(6) Sono il processo padre.
(5) Sono il secondo figlio con pid: = 696
(7) Ho creato un secondo figlio con pid: 696
```

attesa della terminazione di un figlio e codice di terminazione restituito: wait e exit

- La funzione *wait*:
  - sospende l'esecuzione del processo padre che la esegue e attende la terminazione di un qualsiasi processo figlio
  - se il figlio termina prima che il padre esegua la *wait*, l'esecuzione della *wait* nel padre termina istantaneamente
  - prototipo *wait*:

```
pid_t wait (int *)
```

  - il valore restituito dalla funzione (di tipo pid_t) è il valore del pid del figlio terminato
  - il parametro passato per indirizzo assume il valore del codice di terminazione del figlio (cioè il valore del parametro della *exit* eseguita dal figlio per terminare)
    - in effetti il valore passato per indirizzo è il codice di terminazione del figlio moltiplicato per 256 (8 bit più significativi)

esempio

```
#include  . . . .
void main (int argc, char * argv []) {
   pid_t pid;
   int stato_exit, stato_wait;
   pid = fork ( );
   if (pid == 0) {
        printf (“Sono il processo figlio.\n”);
        printf (“Il mio pid e’: %d\n”, getpid ( ));
        stato_exit = 5;
        exit (stato_exit);
   } else {
        printf ("Ho generato il processo figlio con pid %d\n",pid);
        pid = wait (&stato_wait);
        printf ("E’ terminato il processo %d con esito %d\n",
                pid, stato_wait / 256);
   } /* end if */
} /* end if */
```

la *fork* genera un altro processo con PID = b

```
pid = fork();
if (pid == 0) {compito del figlio}
else {...
   pid = wait (&stato_wait);
   seguito del padre ...
};
```

PID = a

PID = a, pid = b

padre

figlio

PID = b, pid = 0

```
pid = fork();
if (pid == 0) {...}
else {...
   pid = wait (&stato_wait);
   il padre attende la
   terminazione del figlio ...};
```

```
pid = fork();
if (pid == 0) {
   compito del figlio
   exit (stato_exit)
}
else {...};
```

dopo la terminazione del processo figlio

termine del processo con PID = b

```
pid = fork();
if (pid == 0) {...}
else {...
   pid = wait (&stato_wait);
   dopo la terminazione del figlio
   il padre procede l'esecuzione ...};
```

funzione waitpid ( )

- La funzione waitpid**:**
  - sospende l'esecuzione del processo padre e attende la terminazione del processo figlio di cui viene **fornito** il pid
  - se il figlio termina prima che il padre esegua la *waitpid*, l'esecuzione di *waitpid* nel padre termina istantaneamente

- prototipo *waitpid*:

```
/* restituisce un pid_t */
pid_t waitpid (pid_t pid, int * status, int options)
```

nel padre:

- il valore resitituito assume il valore del pid del figlio terminato
- *status* assume il valore del codice di terminazione del processo figlio
- *options* specifica ulteriori opzioni (ipotizziamo > 0)

utilizzo della funzione *waitpid*

```
#include . . .
void main (int argc, char * argv[]) {
   pid_t pid, my_pid;
   int status;
   pid = fork ( );
   if (pid == 0) {

      /* CODICE DEL FIGLIO */

   } else {   /* pid != 0, sono nel padre */
      printf ("Ho generato il processo figlio con pid %d\n", pid);
      printf ("Attendo la terminazione del figlio con pid %d\n", pid);
      my_pid = waitpid (pid, &status, 1);
      printf ("E' terminato il processo %d con esito %d\n", my_pid, status);
   } /* end if */
} /* end if */
```

sostituzione del programma in esecuzione: exec ( )

- La funzione exec:
  - **sostituisce il segmento codice e il segmento dati** del processo corrente con il codice e i dati di un programma contenuto in un file eseguibile specificato
  - **il segmento di sistema non viene sostituito** (i file e i descrittori di periferica rimangono aperti e disponibili)
  - **il processo rimane lo stesso** e quindi mantiene lo stesso pid
  - la funzione **`exec`** passa dei parametri al programma che viene eseguito, tramite il meccanismo di passaggio dei parametri al main **`argc`** e **`argv`**

sintassi di *exec*

- Sintassi:

```
/* restituisce int */
int  execl (char * path_programma, char * arg0, char * arg1,
            ..., char * argn);
```

path_programma: path completo del file eseguibile del nuovo programma da lanciare

arg0, arg1, …, argn: puntatori alle stringhe che da passare come parametri al main del nuovo programma

arg0 deve essere il nome del programma

argn in chiamata deve essere il valore NULL

il valore restituito è:

- 0 se l'operazione è stata eseguita correttamente
- –1 se c'è stato un errore e l'operazione di sostituzione del codice è fallita

al momento dell'esecuzione del main del nuovo programma:

void main (int argc, char * argv [])

gli argomenti **arg0, arg1, …** vengono resi accessibili tramite l'array di puntatori **argv**

passaggio di parametri a *main*

- **argc**: contiene il numero dei parametri ricevuti
- **argv**: è un vettore di puntatori a stringhe, ognuna delle quali è un parametro
- per convenzione argv [0] contiene sempre il nome del programma in esecuzione

```
#include <stdio.h>
void main (int argc, char * argv []) {
   int i;
   printf (“Il valore di argc e’ %d \n \n”, argc);
   for (i = 0; i < argc; i++)  {
        printf (“Parametro %i = %s\n”, i, argv [i]);
   } /* end for */
} /* end main */
```

In esecuzione:

>prova1 131.175.23.1
Il valore di argc e’ 2

Parametro 0 = prova1
Parametro 1 = 131.175.23.1

utilizzo di *exec*

```
/* programma main1 */

#include <stdio.h>

void main (int argc, char * argv []) {
    int i;
    printf ("Programma main1 in
    esecuzione.\n");
    printf ("Ho ricevuto %d parametri.\n",
           argc);
    for (i = 0; i < argc; i++) {
          printf ("Il parametro %d e':
                  %s\n", argv [i]);
    } /* end for */
} /* end main */
```

```
/* programma exec1 */

#include <stdio.h>
#include <sys/types.h>

void main (int argc, char * argv []) {
    char P0[] = "main1";
    char P1[] = "parametro1";
    char P2[] = "parametro2";
    printf ("Programma exec1 in
             esecuzione.\n");
    execl ("/antola/esempi/main1", P0, P1,
          P2, NULL);
    printf ("Errore di exec.\n");
} /* end main */
```

esecuzione

```
$ ./exec1
Programma exec1 in esecuzione.


Programma main1 in esecuzione.
Ho ricevuto 3 parametri.
Il parametro 0 e':main1
Il parametro 1 e':parametro1
Il parametro 2 e':parametro2
```

utilizzo di *fork-exec* – I

- La sostituzione di codice non implica necessariamente la generazione di un figlio (vedi esempio precedente):
  - in questo caso, quando il programma che è stato lanciato in esecuzione tramite la **`execl`** termina, termina anche il processo che lo ha lanciato (sono lo stesso processo !!)

- È necessario creare un nuovo processo, che effettua la sostituzione di codice (utilizzo di fork- exec), quando è necessario "mantenere in vita" il processo di partenza, dopo l'esecuzione del codice sostituito:
  - spesso questo implica che il padre attenda la terminazione del programma lanciato con mutazione di codice

utilizzo di *fork-exec* – II

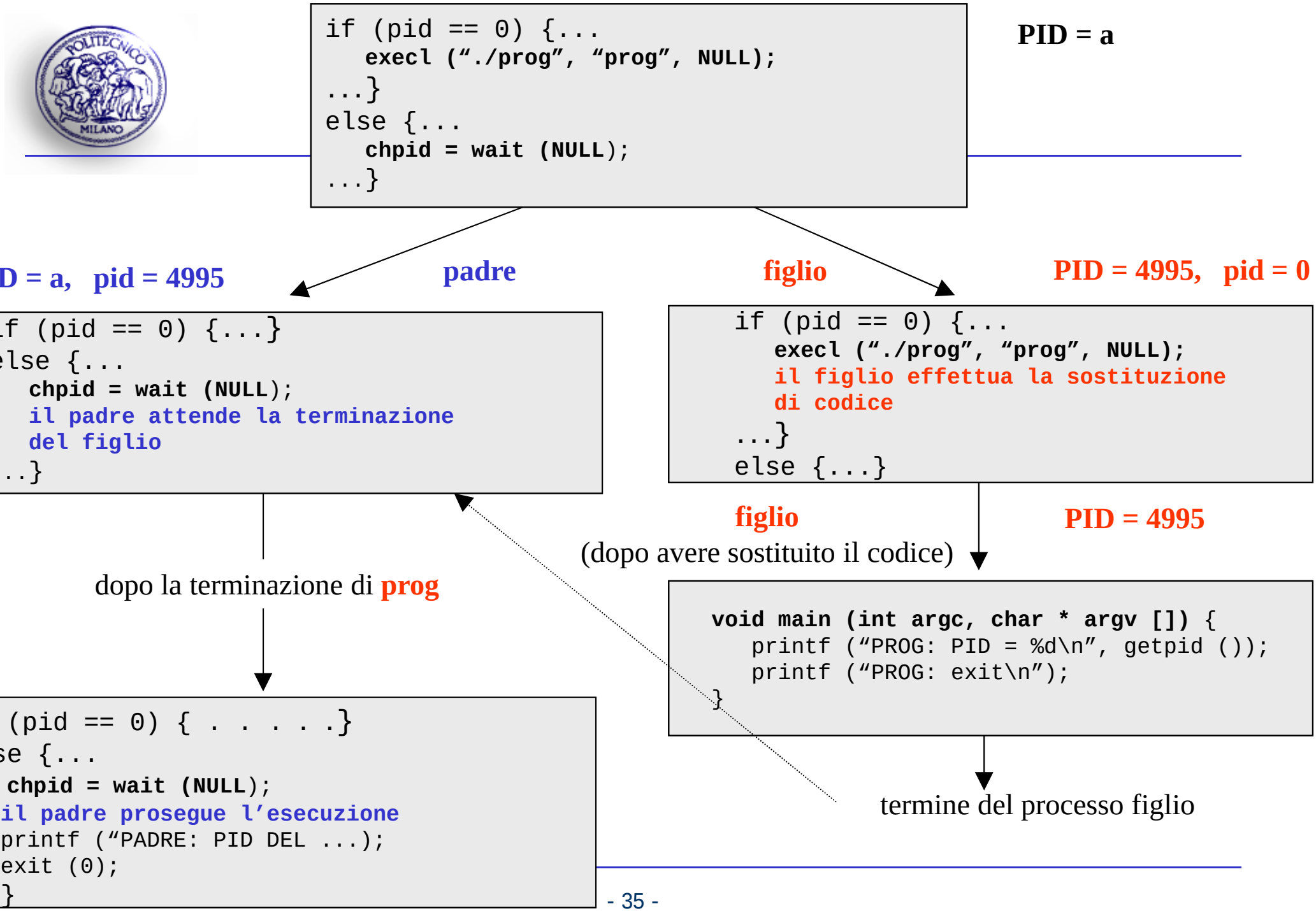
```
void main ( ) {
   pid_t pid, chpid;
   pid = fork ( );
   if (pid == 0) {
        /* PROCESSO FIGLIO*/
        printf (“FIGLIO: prima del cambio di codice.\n”);
        printf (“FIGLIO: PID = %d\n”, getpid ( ));
        execl (“./prog”, “prog”, NULL);
        printf (“FIGLIO: errore nel cambio di codice.\n”);
        exit (1);
   } else {
        /* PROCESSO PADRE */
        printf (“PADRE: wait.\n”);
        chpid = wait (NULL);
        printf (“PADRE: PID DEL FIGLIO = %d\n”, chpid);
        exit (0);
   } /* end if */
} /* end main */
```

utilizzo di *fork-exec* – III

```
void main (int argc, char * argv []) {
        printf (“PROG: PID = %d\n”, getpid ( ));
        printf (“PROG: exit.\n”);
} /* end main */
```

ESECUZIONE: nell’ipotesi che venga eseguito prima il padre e poi il figlio.

```
PADRE: wait.
FIGLIO: prima del cambio del codice.
FIGLIO: PID = 4995
PROG: PID = 4995
PROG: exit.
PADRE: PID DEL FIGLIO = 4995
```


if (pid == 0) {...
execl ("./prog", "prog", NULL);
...}
else {...
chpid = wait (NULL);
...}
PID = a
PID = a, pid = 4995
padre
figlio
PID = 4995, pid = 0
if (pid == 0) {...}
else {...
chpid = wait (NULL);
il padre attende la terminazione
del figlio
...}
if (pid == 0) {...
execl ("./prog", "prog", NULL);
il figlio effettua la sostituzione
di codice
...}
else {...}
figlio
(dopo avere sostituito il codice)
PID = 4995
dopo la terminazione di prog
void main (int argc, char * argv []) {
printf ("PROG: PID = %d\n", getpid ());
printf ("PROG: exit\n");
}
if (pid == 0) { . . . . .}
else {...
chpid = wait (NULL);
il padre prosegue l'esecuzione
printf ("PADRE: PID DEL ...);
exit (0);
...}
termine del processo figlio