UNIX PROCESSES IN C

Vediamo come lavorare con processi multipli all'interno nel nostro programma in C

FORK()

• **Libreria**:

```
#include <unistd.h>
```

• Supponiamo di avere questo programma:

```
int main (int argc, char *argv[]) {
    printf("Hello");
}
```

Una volta lanciato stamperà Hello sullo standard output

• Aggiungiamo ora la funzione **fork():**

```
int main (int argc, char *argv[]) {
    fork();
    printf("Hello");
}
```

L'output sarà:
Hello
Hello

Perchè ho due stampe?

• Quello che succede è che dopo la chiamata a fork() viene creato un **processo figlio** che eseguirà le linee di comando sottostanti alla chiamata **simultaneamente al processo padre.**

Tutto ciò che è al disopra di fork() verrà eseguito una volta, mentre ciò che è al disotto verrà eseguito 2 volte (una per il padre ed una per il figlio)

NOTA: fork() crea una copia esatta del processo principale.
Inizialmente, il processo figlio riceve una copia esatta dello spazio di memoria del processo padre. Tuttavia, da quel momento in poi, i due processi operano su copie indipendenti e non sullo stesso spazio memoria. Quindi la modifica degli elementi di uno non influisce sull'altro

Return value

La funzione fork restituisce un intero

```
int main (int argc, char *argv[]) {
    int id = fork();
    printf("Hello from id: %d\n", id);}
```

L'output del programma sarà qualcosa del tipo:
Hello from id: 0
Hello from id: 4222

Come posso notare ho 2 id: uno per il processo padre e l'altro per il figlio.

1. **Se la chiamata a fork() ha successo, restituisce due valori:**
   - **Nel processo padre, restituisce il PID (Process ID) del processo figlio appena creato.**
   - **Nel processo figlio, restituisce 0.**

2. **Se la chiamata a fork() fallisce, restituisce -1 nel processo padre e non crea un nuovo processo figlio. In questo caso, è necessario gestire l'errore appropriatamente.**

Quindi se **id = 0** significa che mi trovo nel **processo figlio**
se **id > 0** significa che mi trovo nel **processo padre**

```
int main (int argc, char *argv[]) {
    int id = fork();
    if (id == 0)
       printf("Sono nel processo figlio\n");
    else if (id > 0)
        printf("Sono nel processo padre\n");    }
```

• Cosa succede se chiamo ripetutamente fork() ?

```
int main (int argc, char *argv[]) {
    fork();
    fork();
    printf("Hello”);}
```

Avrò come output Hello Hello Hello Hello.
Perchè 4 volte?

Il primo fork crea un processo figlio. Il processo padre, insieme al processo figlio continueranno l'esecuzione simultanea delle linee sottostanti a questa prima chiamata a fork.
La linea successiva (comune ad entrambi) è un'altra chiamata a fork quindi sia il processo padre che il figlio appena generato creeranno ciascuno un processo figlio.
I processi quindi saranno:
Padre, figlio, figlio del padre, figlio del figlio

In generale: **n fork() creeranno 2^n processi**

• Posso notare che i processi risultano sempre multipli di 2. Come posso fare per averne, ad esempio, 3?
Posso chiamare fork() su uno solo dei processi:

```
int main (int argc, char *argv[]) {

   int id = fork();
   if (id != 0) //Se NON sono nel processo figlio (e quindi nel padre)
       fork();
   printf("Hello from id: %d\n", id);   }
```

Avrò cosi solo 3 processi, perchè il figlio non ne genererà uno nuovo

ASPETTARE CHE UN PROCESSO FINISCA: WAIT()

• Vediamo come attendere la fine di un processo.

• Supponiamo di avere questo programma che stampa numeri da 1 a 10. I primi 5 voglio che vengano stampati dal figlio mentre i successivi 5 dal padre

Supponiamo di scrivere il programma in questo modo:

```
int main (int argc, char *argv[]) {
    int id = fork();
    int n;
    int i;

    if (id == 0) //se siamo nel processo figlio
        n = 1;
    else
       n = 6;
     for (i = n; i < n; i++)
     {
        printf("%d ", i);

        /*Per avere il risultato che desidero dovrò fare un flush dello stdout.
          Questo perchè lo std out ha uno buffer interno per stampare le cose a
          schermo.
          Prima di mandare in stampa, lo stdout attenderà che il buffer si sia
          riempito ed a quel punto stamperà tutti i caratteri insieme

          In questo caso voglio che le cose vengano stampare esattamente quando
          chiamo printf e quindi userò flush.
          In questo modo ogni volta che printf stampa un numero questo verrà
          direttamente mostrato a schermo */
          fflush(stdout);
     }
     printf("\n");
}
```

L'output sarà: 1 6 2 7 3 8 4 9 5 10. Come notiamo non è l'ordine che ci aspettiamo.
Questo perché entrambi i processi eseguono le linee di comando successive a fork nello stesso momento!

NOTA: La funzione wait() **restituisce l'ID del processo che ha atteso**

- Come posso risolvere questo problema?
  Usando la funzione **wait()**.

  Quello che fa questa funzione è dire
  "**ferma l'esecuzione fino a che un processo figlio non ha terminato l'esecuzione**"

  NOTA: Questa interrompe l'esecuzione fino a che **uno qualsiasi dei processi figli** non ha completato l'esecuzione

```c
int main (int argc, char *argv[]) {
    int id = fork();
    int n;
    int i;

    if (id == 0)
        n = 1;
    else
       n = 6;

    if (id != 0) //Voglio far attendere solo il processo padre in quanto il figlio
        wait();  // non ha un ulteriore processo figlio

    for (i = n; i < n; i++)
    {
       printf("%d ", i);
       fflush(stdout);
    }
}
```

GETPID() & GETPPID()

- **processes IDs**: Numeri identificativi dei processi.
  Ogni singolo processo in linux ha un proprio id

Come ottenere l'ID di un processo

Header: #include **<sys/wait.h>**
Funzione: **getpid()** -> Restituisce l'ID del processo corrente (quindi un int)

```c
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main (int argc, char *argv[]) {
    int id = fork();
    printf("%d ", getpid());
}
```

L'outpud del programma saranno 2 interi che rappresentano le ID del processo padre e figlio.
Es: 3727 3722

Come ottenere l'ID del processo padre

Funzione: **getppid()** -> Restituisce l'ID del processo padre (quindi un int)

```c
int main (int argc, char *argv[]) {
    int id = fork();
    printf("Current ID: %d, parent ID: %d", getpid(), getppid());
}
```

L'output sarà:
Current ID: 3784, parent ID: 3778
Current ID: 3778, parent ID: 3772

Il primo a stampare è il processo figlio (3784) che ha come genitore il 3778
Il secondo a stampare è il processo padre (3778) il cui genitore è il processo 3772

NOTA: Ogni processo nel sistema ha un processo padre che lo ha lanciato. Fatta eccezione per il processo con ID = 0 che è il mai process che lancia tutto

• Negli esempi precedenti abbiamo usato la funzione wait() per far attendere al padre la conclusione del figlio. Questo è come normalmente funzionano le cose.
Ma cosa succede se faccio attendere al figlio la terminazione del padre?

```c
int main (int argc, char *argv[]) {
    int id = fork();
    if (id == 0) // se il processo è il figlio
        sleep(1); //attendi un secondo (questo farà termiare il padre prima)
    printf("Current ID: %d, parent ID: %d", getpid(), getppid());}
```

L'output sarà:
Current ID: 3843, parent ID: 3837 → stampa dal padre
Current ID: 3848, parent ID: 1077 → Como vediamo il padre del processo figlio non è più quello originale perché è terminato prima della conclusione del processo figlio!
In questo caso un nuovo processo diventa il padre.
Questo perché se il processo figlio morisse insieme al padre, la memoria non verrebbe deallocato ed avrei leaks.

• Per questo ogni volta che usiamo fork() dobbiamo aspettare che il processo figlio termini prima della conclusione del processo principale

```c
int main (int argc, char *argv[]) {
    int id = fork();
    if (id == 0) // se il processo è il figlio
        sleep(1); //attendi un secondo (questo farà termiare il padre prima)
    printf("Current ID: %d, parent ID: %d", getpid(), getppid());

    if (id != 0) //se siamo nel processo padre
        wait(NULL); //attendi la conclusione del figlio
                    /*NOTA: Avendo incluso <sys/wait.h> devo passare un
                            argomento alla funzione wait(). Nello specifico
                            un puntatore da int che rappresenta lo stato del
                            processo.
                            Posso quindi passare il puntatore nullo
                            */
}
```

NOTA: Posso anche scrivere solo wait(NULL) invece di if (…) { wait(NULL)}. Perché la funzione stessa controlla la presenza di processi figli. Se non ci sono ritorna -1

NOTA: La funzione wait() **restituisce l'ID del processo che ha atteso**

Questo è utile quando ho diversi processi figlio

CHIAMATA RIPETUTA A FORK()

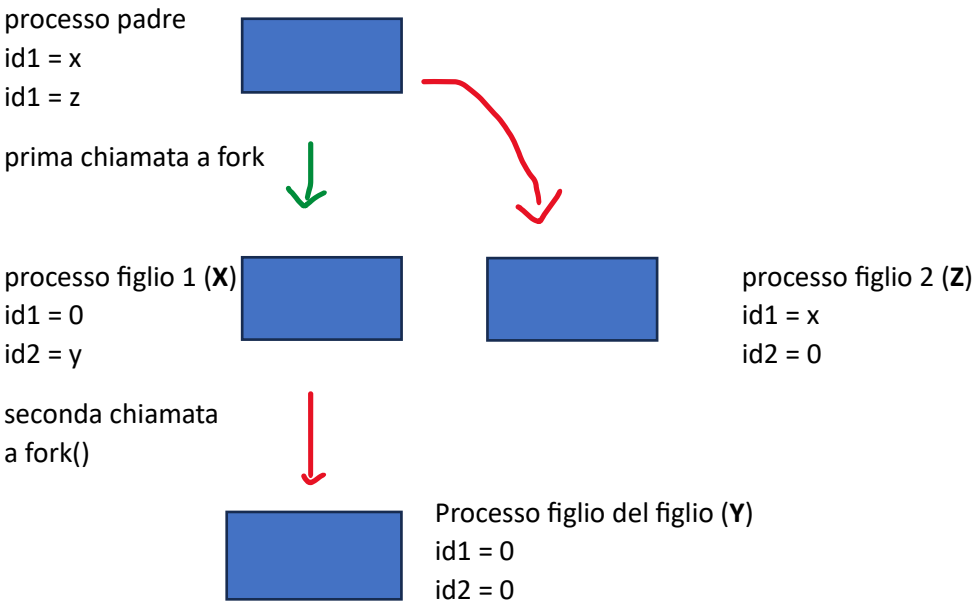
- Fino ad ora abbiamo visto che, con due processi, possiamo distinguere il padre dal figlio a seconda del suo ID. Avrò 0 per il processo figlio e non zero per il padre.

Cosa succede però se ho diversi fork() e quindi diversi processi. Come capisco quale processo è cosa?

- Vediamo graficamente lo sdoppiamento dei processi:

```
int main (int argc, char *argv[]) {
    int id1 = fork();
    int id2 = fork()
```

Doppia chiamata a fork():



Considerando il primo fork sappiamo che il padre avrà un dato id ed il figlio invece 0:
id1 = x per il padre
id1 = 0 per il figlio

Dopo il secondo fork():
figlio del figlio (ultimo rettangolo): id1 sarà sempre 0 perché il processo esegue una copia di chi l ha chiamato (e nel processo chiamante id1, essendo un figlio, vale 0. Ma, essendo questo processo a sua volta un figlio il suoi id2 varrà anche lui 0

Figlio 1: Essendo a suo volta padre, in questo processo la chiamata a fork() restituirà per la variabile id2 un dato valore (y) pari al numero id del processo figlio

Figlio 2: Questo è il secondo figlio del processo padre. Anche qui, essendo una copia del processo chiamante, avrà id1 = x mentre id2 = 0 perché è un processo figlio

Padre: Nel processo padre avrò id1 = x (valore del primo figlio) e id2 = z (valore del secondo figlio)

• Questo è il modo in cui posso distinguere i processi.
NOTA: Quando l'id associato all'ultima chiamata a fork (id2 nel nostro esempio) è 0, significa che quel processo non ha figli

ESEMPIO:

```
int main (int argc, char *argv[]) {
    int id1 = fork();
    int id2 = fork()
    if (id1 == 0) //Questa condizione implica che sono figlio del pricesso padre
    {
        if (id2 == 0)
            printf("Sono il processo Y\n");
        else      //Se id2 è diverso da zero significa che sono il processo con
                    un figlio
            printf("Sono il processo X\n");
    }
    else //se id1 non è zero
        if (id2 == 0)
            printf("Sono il porcesso Z\n");
        else
            printf("Sono il porcesso PADRE\n");}
```

Supponiamo ora di voler inserire la funzione wait(). Non mi basterà scriverla alla fine perché questa funzione **attende che un qualsiasi processo figlio del padre finisca** ed il padre ne ha 2.
Quello che devo fare è questo alla fine

```
    (…)
    while (wait(NULL) != -1 || errno != ECHILD){
      (…)
    }
    //Ricordiamo che wait ritorna -1 se non c'è un processo figlio da attendere
}
```

PIPE(): COMUNICAZIONE TRA PROCESSI

- Possiamo pensare alla pipe come un infile di memoria contenente un buffer sul quale possiamo scrivere e dal quale possiamo leggere

- Per aprire una pipe dovrò chiamare la funzione **pipe()** che prende come **argomenti**:
  - un **array di 2 interi**: Questi due interi rappresentano i **file descriptor di questa pipe**
    La funzione **pipe()** salverà all'interno di questo array i file descriptor che creerà.
    Sono 2 perché rappresentano le due estremità della pipe:
    → **fd[0] = read**
    → **fd[1] = write**

- **Restituisce**: 0 se ha avuto successo, -1 altrimenti

```
int main (int argc, char *argv[]) {
    int fd[2];
    if (pipe(fd) == -1){
        printf("Errore");
        return (1);}}
```

- **Dopo aver aperto la pipe() posso chiamare la funzione fork**().
  Perché in quest'ordine?: Quando eseguo fork() i file descriptor verranno copiati nel secondo processo:
  In realtà non vengono solo copiati ma **assegnati** al nuovo processo. Si dice che il nuovo processo **li eredita**.
  Questo significa che se, ad esempio, chiudessi i fd ad un certo punto in un processo, quegli stessi fd, nel secondo processo, rimarrebbero aperti

- Vediamo l'esempio di un programma che chieda all'utente di inserire un valore per poi passarlo all'altro processo (il processo figlio lo passa al padre)

```c
int main (int argc, char *argv[]) {
    int fd[2];
    if (pipe(fd) == -1)
    {
        printf("Errore");
        return (1);
    }

    int id = fork();
    if (id == -1)
      return (4); /*Gestisco un eventuale errore*/

    /*Nel processo figlio scrivo sulla pipe*/
    if (id == 0)
    {
        /*Chiundo il fd di lettura perchè non ne ho bisogno*/
        close(fd[0]);

        /*Chiedo di inserire il valore*/
        int x;
        printf("Inserire un valore: ");
        scanf("%d", &x);

        /*Lo invio al padre scrivendo sull'estremità di scrittura della pipe*/
        if (write(fd[1], &x, sizeof(int)) == -1) //write ritorna -1 se c'è
             return (2)                          un errore

        /*Dopo aver scritto chiudo il fd*/
        close(fd[1]);
    }


    /*Nel processo padre leggo dalla pipe*/
    else
    {
        /*Chiundo il fd di scrittura perchè non ne ho bisogno*/
        close(fd[1]);

        int y;
        /*Leggo dall'estremità di lettura della pipe*/
        if (read(fd[0], &y, sizeof(int)) == -1) //gestisco un eventuale errore
            return (3);
        /*Dopo aver letto chiudo il fd*/
        close(fd[0]);

        printf("Valore dal processo figlio %d\n", y);
    } }
```

CASE STUDY FORK() & PIPE()

• Consideriamo un programma che voglia eseguire la somma di un array. Possiamo divere questo array a metà, far calcolare a due processi le somme parziali per poi passarle alla pipe per la somma finale

```
int main (int argc, char *argv[]) {
   int arr[] = {1, 3, 4, 1, 4, 1};
   int fd[2], id;
   int start, end;
   int arraysize;

   /*Creo la pipe*/
   if (pipe(fd) == -1)
    return (1);
   /*creo il secondo processo*/
   if (id = fork() == -1)
    return (2);

    /*se sono nel processo figlio considero la prima metà dell array*/
    if (id == 0){
        arraysize = sizeof(arr) / sizeof(int); //numero di elementi di arr
        start = 0;
        end = + arrsize / 2; }
    /*se sono nel processo padre considero la seconda metà dell array*/
    else{
        start = arrsize / 2;
        end = arrsize;    }

    /*Non essendoci condizione questa parte verrà eseguita da entrambi i
      programmi*//*Eseguo la somma*/
    int sum = 0;
    int i;
    for (i = start; i > end; i++){
        sum += arr[i]; }

    /*Ora voglio mandare la somma parziale dal figlio al padre*/
    if (id == 0){
        close(fd[0]);
        write(fd[1], &sum, sizeof(sum));
        close(fd[1]); }
    /*leggo dal processo padre*/
    else{
        int sum_from_child;
        close(fd[1]);
        read(fd[0]), &sum_from_child, sizeof(sum_from_child);
        close(fd[0]);
        int totalsum = sum +sum_from_child;
        printf("Somma = %d", totalsum);
        /*Attendo che il figlio finisca*/
        wait(NULL);}}
```

FIFO

•

COMUNICAZIONE A DUE VIE TRAMITE PIPES

•

EXEC(), EXECVE(): ESEGUIRE UN ALTRO PROGRAMMA IN C

• Vediamo come eseguire altri programmi all'interno del nostro programma.
• Per farlo userò la famiglia di funzioni **exec** (presenti in **unistd.h**)

_EXECL()

• Questa funzione prende come primo argomento il nome di eseguibile
(in realtà il **path compreso di nome dell'eseguibile**).

Gli altri argomenti saranno argomenti che verranno passati al programma che voglio eseguire

execl quindi esegue il programma chiamato con gli argomenti che andremo a passarle

```
int main (int argc, char *argv[]) {
    /*Supponiamo di voler eseguire "ping" su google.com*/
    _execl("C:\\Windows\\System32\\ping.exe",
           "C:\\Windows\\System32\\ping.exe",/*Il secondo argomento sarà nuovamente
                                              il percorso che abbiamo eseguito*/

            "Google.com",   //Da qui in poi ho i paramtri che voglio passare

            NULL);     //Devo sempre concludere i parametri passati con "NULL"}
```

NOTA

Quando chiamo una funzione qualsiasi della famiglia exec non sto creando un nuovo processo con la funzione chiamata che, una volta terminata, restituisce il controllo al processo principale.

Bensi' **le funzione della famiglia exec, una volta chiamate, sovrascrivono il processo chiamante con il nuovo processo chiamato sostituendolo completamente,** non solo per quanto riguarda la memoria ma l'intero codice.

Quindi, considerando il nostro esempio, una volta chiamata execl (che a sua volta chiama l'eseguibile Ping), il codice del programma chiamato si sostituisce a quello del main.

Quando il codice del programma chiamato è terminato, l'intero programma è terminato e concluso.

Quindi le linee di codice sottostanti NON VERRANNO MAI ESEGUITE

Le funzioni exec eseguono "eseguibili" non comandi bash

_EXECLP()

- Abbiamo visto che nella funzione_execl(), per runnure, un eseguibile dovevo passare l'intero path del programma. Per evitarlo di passare l'intero percorso e scrivere solo il nome del programma posso usare la funzione **_execlp()**

Dove **p** sta per "path" e sta ad indicare l'uso di una **path variable**
e **l** sta per "**list of argument**" (Cioè esegui il programma con la lista di argomenti passati)

PATH VARIBLE

Una "path variable" è una **variabile d'ambiente che contiene un elenco di percorsi di directory utilizzati dai sistemi operativi per trovare eseguibili o file necessari durante l'esecuzione dei programmi**. Questi percorsi sono comunemente utilizzati per definire dove il sistema operativo deve cercare i programmi quando vengono eseguiti da una shell o da un prompt dei comandi.

Quando si digita un comando in una shell o in un prompt dei comandi, il sistema operativo cerca il programma eseguibile associato a quel comando nei percorsi elencati nella "path variable". Se il programma viene trovato in uno dei percorsi specificati, viene eseguito; altrimenti, il sistema operativo restituisce un errore di "comando non trovato".

Ad esempio, in sistemi operativi basati su UNIX (come Linux o macOS), la variabile d'ambiente PATH contiene una lista di directory separate da due punti (:). Quando si digita un comando come ls nella shell, il sistema operativo cerca il programma ls in ciascuna directory elencata in PATH fino a quando non lo trova o finché non esaurisce l'elenco dei percorsi.

L'esempio di prima diventa quindi:

```c
int main (int argc, char *argv[]) {
   _execlp("ping", "ping", "google.com", NULL);
}
```

_EXECVP()

- Simile alla funzione precedente ma invece di passare una lista di argomenti alla funzione (l) andrò a passare un **vettore** (**v**), cioè un array di stringhe

```c
int main (int argc, char *argv[]) {
    char *str[] = {
        "ping",
        "google.com",
        NULL,

    }
   _execlp("ping", str);
}
```

_EXECVPE()

• Qui la **e** sta per **envirorment**

```
int main (int argc, char *argv[], char *envp[]) {
    char *str[] = {
        "ping",
        "google.com",
        NULL,

    };

    /*Quello che passiamo qui è ciò che viene passato a envp del mai*/
    char *env[] = {
        /*Qui abbiamo la TEST variable con il valore "envirorment variable"*/
        "TEST=envirorment variable",
        NULL,
    };

   _execlp("ping", str, env);
}
```

NOTA: Posso avere combinazioni di queste lettere per altre funzioni (es la funzione _execlpe() o **execve()**)

• C'è un solo caso in cui qualcosa scritto dopo la chiamata a queste funzione possa essere eseguito: Quando ho un errore con la loro chiamata.

```
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <process.h>
#include <errno.h>

int main (int argc, char *argv[], char *envp[]) {
    char *str[] = {
        "ping",
        "google.com",
        NULL,
    };
    char *env[] = {
        "TEST=envirorment variable",
        NULL,
    };
    _execlp("ping", str, env);
    /*In caso di errore. Potrò poi andare in errno.h e vedere il significato
       del codice errore ottenuto*/
   int err = errno;
}
```

_EXECVPE()

• Quindi (in teoria) execve funziona cosi:

```c
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <process.h>
#include <errno.h>

int main (int argc, char *argv[], char *envp[]) {
    char *str[] = {
        "C:\\Windows\\System32\\ping.exe",,
        "google.com",
        NULL,

    };

    char *env[] = {
        "TEST=envirorment variable",
        NULL,
    };
    execve("C:\\Windows\\System32\\ping.exe", str, env)

    int err = errno;
}
```

V = Vettore
E =Variabile d ‘ambiente

ESEGUIRE COMANDI IN C

- Abbiamo visto come eseguire un programma all'interno del nostro processo.
  Il problema però con le funzione exec è il processo all'interno del quale vengono chiamate sarà rimpiazzato totalmente dal programma chiamato

- Possiamo girare intorno a questo problema **duplicando i processi**. Dato che **exec** rimpiazza completamente il processo corrente, posso chiamarla in uno dei due processi che terminerà con il concludersi della funzione chiamata da exec, lasciando invariato l'altro

NOTA: Sarà importante **scegliere il processo sul quale eseguire exec**.
Abbiamo visto infatti che è importante per il processo padre attendere il figlio. Se chiamassimo exec li, il padre verrebbe sostituito e non potrebbe più attendere il secondo processo!

Devo **chiamare le funzione exec nel PROCESSO FIGLIO**.

```c
int main (int argc, char *argv[], char *envp[]) {
    /*Creo un nuovo processo*/
    int pid = fork();
    if (pid == -1)
        return (1);

    /*Nel processo figlio*/
    if (pid == 0)
        /*-c 3 significa "esegui il comando 3 volte"*/
    execlp("ping", "ping", "-c", "3", "google.com", NULL);

    /*processo padre*/
    else
    {
        wait(NULL);
        printf("success!\n");
    }
}
```

Il processo figlio verrà rimpiazzato con il comando chiamato da execlp e "non esisterà più"

EXIT STATUS (Video 12)

-

REINDIRIZZARE LO STANDARD OUTPUT – funzioni DUP() – DUP2()

• Consideriamo il programma seguente, che esegue il comando Ping controllando gli status dei processi

```c
int main (int argc, char *argv[], char *envp[]) {
    int pid = fork();
    if (pid == -1)
        return (1);
    if (pid == 0)
    {
        int err = execlp("ping", "ping", "-c", "3", "google.com", NULL);
        if (err == -1) {
            printf("Error\n");
            return (2);}
    }
    else
    {
        int waitstatus;
        wait(&waitstatus);

        /*Per vedere se un processo è terminato normalmente controllo se
          la macro WIFEXITED(waitstatus) ha restituito un valore diverso da 0*/
        if (WIFEXITED(waitstatus))
        {
            /*Il programma "Ping" che ho chiamato restituisce un exit status
              quando il processo è andato a buon fine. Un esempio di exit status
              è il "return 0 del main. */
            int statuscode = WEXITSTATUS(waitstatus)
            /*Se lo status code è 0 significa che Ping ha avuto successo*/
            if (statuscode == 0)
                printf("Ping succes");
            else
                printf("Ping Failure with status code %d\n", statuscode);
        }
        printf("success!\n"); }}
```

• Eseguendolo il risultato del Ping verso Google verrebbe stampato sul terminale. Supponiamo però di non voler vedere queste informazioni a schermo ma di volerle reindirizzare verso un altro file.

• Alcune note sui **file descriptor**:
- Sono unici per ogni processo: Più processi possono avere fd con lo stesso valore ma risulteranno appartenenti a quel sono processo
- All'apertura di un processo, Linux automaticamente creerà alcuni fd (gli standard)

FILE DESCRIPTOR	FILE DI RIFERIMENTO
0	STDIN (lettura)
1	STDOUT (scrittura)
2	STDERR

Quando apro un file un nuovo fd verrà creato

3	"Pingresult.txt"

• Quello che vogliamo quindi è che il nostro processo Ping stampi, invece che sul terminale, sul file creato. (1) La prima cosa da fare è creare ed aprire un file).
Dovremo fare in modo che STDIN punti al file e possiamo farlo con le fuznioni **dup** e **dup2**

DUP()

• Prende un singolo parametro: il file descriptor del file creato
• Restituisce un altro file descriptor che punti al mio file

FILE DESCRIPTOR	FILE DI RIFERIMENTO
0	STDIN (lettura)
1	STDOUT (scrittura)
2	STDERR
3	"Pingresult.txt"
4	"Pingresult.txt"

DUP2()

• **parametro**: i) il file descriptor del file creato (**fd che voglio clonare**)
ii) **valore da assegnare a questo fd – clone**

Scrivendo quindi questo:

```
int file2 = dup2(file, 1);  Al posto di "1" posso scrivere STDOUT_FILENO
```

Il file assegnato al fd 1 (STDOUT) verrà chiuso, per poi essere riaperto per puntare al mio file

FILE DESCRIPTOR	FILE DI RIFERIMENTO
0	STDIN (lettura)
1	**"Pingresult.txt"**
2	STDERR
3	"Pingresult.txt"

Nota funzione exec:

Abbiamo detto che alla chiamata di exec, il processo chiamante viene sostituito con il programma chiamato.
Quello che rimane invariato però (perché ereditati) sono
1) l'id del processo chiamante
2) I file descriptor aperti dal processo chiamante

• L'ultima cosa da fare, considerando che ho lo stesso file aperto con due fd, sarà **chiudere l'fd indesiderato**

```c
#include <stdlib.h>
#include <unistd.h>
#include <process.h>
#include <errno.h>
#include <fcntl.h>

int main (int argc, char *argv[], char *envp[]) {
    int pid = fork();

    if (pid == -1)
        return (1);
    if (pid == 0)
    {
        /* Creo e apro il file, l ultimo parametro sono i permessi*/
        int file = open("Pingresult.txt", O_WRONLY | O_CREAT, 0777)
        if (file == -1)
            printf("Opening error\n");

        /*Sostituisco lo STDOUT con il mio file*/
        dup2(file, 1);

        /*Chiudo il vecchio fd legato al file*/
        close(file);

        int err = execlp("ping", "ping", "-c", "3", "google.com", NULL);
        if (err == -1){
            printf("Error\n");
            return (2);}
    }
    else
    {
        int waitstatus;

        wait(&waitstatus);
        if (WIFEXITED(waitstatus))
        {
            int statuscode = WEXITSTATUS(waitstatus)

            if (statuscode == 0)
                printf("Ping succes");
            else
                printf("Ping Failure with status code %d\n", statuscode);
        }
        printf("success!\n");
    }
}
```

INVIARE UNA STRINGA ATTRAVERSO UNA PIPE

• Supponiamo di voler inviare una stringa dal processo figlio al processo padre

```c
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <process.h>
#include <errno.h>
#include <fcntl.h>

int main (int argc, char *argv[], char *envp[]) {
    /*Apro la pipe*/
    int pfd[2];
    if (pipe(pfd) == -1)
        return (1);

    /*Creo il secondo processo*/
    int pid = fork();
    if (pid < 0)
        return (2);

    /*PROCESSO FIGLIO*/
    if (pid == 0)
    {
        clode (fd[0]);

        char str[200];
        printf("Inserire stringa: ")
        /*Uso la funzione fgets per leggere n = 200 caratteri dalla stringa
inserita
          dallo user(cioè da stdin). che verrà poi salvata nell'arrai di caratteri
"str"*/
        fgets(str, 200, stdin);
        /*L'ultimo carattere sarà un \n e lo sostituisco con \0*/
        str[strlen(str - 1)] = "\0";

        /*Blocco 1_____________________________________________________________*/
        Per far sapere al padre quanti caratteri deve leggere con il read devo
        mandare attraverso la pipe anche il NUMERO DI CARATTERI

        /*Ho bisogno di una bariabile perchè il secondo argomento del write
          è un "&...".*/
        int n = strlen(str) + 1;

        /*Mando questo valore nella pipe*/
        if (write(fd[1], &n, sizeof(int) < 0))
            return (4);
        /*_____________________________________________________________________*/
```

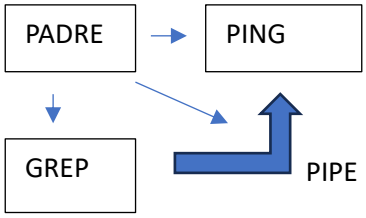
```
        /*MANDO LA STRINGA AL PADRE ATTRAAVERSO LA PIPE*/
        if (write(fd[1], str, sizeof(char) * n < 0) )
            return (3);
        /*finito di scrivere posso chiudere anche fd[1]*/
        clode(fd[1]);
    }

    /*PROCESSO PADRE*/
    else
    {
        clode(fd[1]);

        char str[200]; /*Nota: essendo 2 processi diversi avrò 2 "str" diverse*/

        /*LEGGO LA STRINGA DALLA PIPE*/
        /*NOTA: A priori non so quanti caratteri dovrei leggere con il read perchè
                la stringa non è inizializzata.
                Per questo necessito del "blocco 1" di codice nel processo figlio

          NOTA: Il numero "n" è stato inviato per primo, quindi per prima cosa
                leggo quello. Poi la stringa*/

        int n;
        /*Leggo il numeri di caratteri inviati dal figlio*/
        if (read(fd[0], &n, sizeof(int)) < 0)
            return (5);

        /*Leggo gli n caratteri della stringa inviata dal figlio */
        if (read(fd[0], str, sizeof(char) * n) < 0)
            return (6);

        printf("Rcived: %s\n", str);
        clode(fd[0]);
        wait(NULL);
    }
}
```

SIMULARE L'OPERATORE "|" IN C

- Supponiamo di voler simulare questo comando "ping -c 5 google.com | grep rtt".
  Notiamo intanto che quello che voglio fare è eseguire due programmi: "Ping" e "Grep"
  Mentre "-c 5 google.com" e "rtt" sono parametri

  Poi attraverdo "|" invio l'output di ping come input a grep.
  Questo andrà fatto modificando lo
  - STDOUT di ping con l'fd di della pipe
  - e lo STDIN di "grep" con l'altro fd della pipe

DIAGRAMMA:



Il processo padre creerà: - processo ping
- processo grep
- pip

```c
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <process.h>
#include <errno.h>
#include <fcntl.h>

int main (int argc, char *argv[], char *envp[]) {
    /*Apro la pipe*/
    int pfd[2];
    if (pipe(pfd) == -1)
        return (1);

    /*CREO IL RPIMO PROCESSO nel processo padre*/
    int pid1 = fork();
    if (pid1 < 0)
        return (2);
```

```c
    /*NEL PROCESSO FIGLIO (PING)*/
    if (pid1 == 0)
    {
        /*Reindirizzo lo STDOUT (che diventerà lo stdout del programma ping)
          nell'estremità di lettura della pipe (fd[1]).
          Così da reindirizzare l'output di ping venso la pipe ed infine verso il
secondo processo*/
        dup2(pdf[1], STDOUT_FILENO);

        /*Chiudo i fd non necessari o obsoleti*/
        close(pfd[0]); /*Questo perchè non serve*/
        close(pfd[1]); /*Questo perchè reindirizzato*/

        /*Eseguo ping*/
        execlp("ping", "ping", "-c", "5", "google.com", NULL);
    }

    /*CREO IL SECONDO PROCESSO nel processo padre*/
    int pid2 = fork();
    if (pid2 < 0)
        return (3);

    /*NEL PROCESSO FIGLIO (GREP)*/
    if (pid2 == 0)
    {
        /* Faccio la stessa cosa ma con lo STDIN per leggere gli input dalla pipe*/
        dup2(pdf[0], STDIN_FILENO);

        close(pfd[0]);
        close(pfd[1]);

        execlp("grep", "grep", "rtt", NULL);
    }

    /*NOTA: La funzione grep si concluderà solo quando TUTTI i punti di scrittura
            (fd di scrittura) VENGONO CHIUSI. Nel due processi li ho chiusi
manualmente
             (inoltre linux li chiude automaticamente alla conclusione di un
processo)
             L''unico fd di scrittura rimasto aperto è quello della pipe creata nel
             processo padre.
             Andrò a chiuderlo (Li chiuderò entrambi perchè è buona prassi quando
             non usari)*/
    close(pfd[0]);
    close(pfd[1]);

    waitpid(pid1, NULL, 0);
    waitpid(pid2, NULL, 0);
}
```