

Fork nei sistemi Unix-like (istruzioni)

(Ambiente di riferimento: linguaggio C con compilatore gcc)

In Linux, e in generale nei sistemi detti Unix-like, ogni processo che svolga delle istruzioni puo' generare ulteriori processi detti processi figli (child process)

Un **processo figlio** (child) dunque è un processo creato da un altro processo, il processo **padre**.

Ogni processo è identificato dall'OS da un codice identificativo detto **PID**.

Ogni processo ha un padre tranne il processo di partenza, **init**, con PID 1.

E' possibile acquisire il PID del processo in esecuzione con la funzione di sistema **getpid()**.

Per creare un processo figlio il processo padre invoca la funzione **fork()**. La funzione fork crea un processo figlio identico al padre e **duplica** tutti i segmenti del padre nel figlio.

La funzione fork ritorna un valore intero (spesso definito anche come tipo predefinito pid_t) molto importante per la gestione del codice da far eseguire nei due processi:

- il parametro restituito al processo **padre** è il **PID** del processo **figlio** assegnato da OS ed è un valore superiore a 0
- il parametro restituito al processo **figlio** è il valore **0**
- la restituzione di un valore **negativo** indica che si è verificato un errore nell'esecuzione della fork.

E' evidente l'importanza di questo valore di ritorno: infatti per impostare questa tecnica di programmazione è fondamentale distinguere il processo padre dal processo figlio per realizzare le diverse sezioni di codice da far eseguire ai due processi.

Un processo padre puo' generare piu' processi figli con più chiamate fork().

Utilizziamo ora un primo esempio.

Realizziamo un programma che visualizza il PID di un processo, invochiamo la fork e distinguiamo, attraverso il parametro di ritorno della fork, il processo padre ed il processo figlio.

Chiamiamo il sorgente fork.c

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

```
int main()
{
    int pid=getpid();
    printf("processo in esecuzione prima della fork %d \n",pid);
    pid=fork();

    if (pid == 0)
    {
        /*** codice eseguito dal figlio
        getpid() restituisce il PID del figlio, getpid() restituisce il PID del padre
        ***/
        printf("processo figlio in esecuzione \n");
        printf("PID padre : %d PID figlio: %d \n",getppid(), getpid());
    }
    else if (pid > 0)
    {
        /***codice eseguito dal padre
        getpid() restituisce il PID del padre, pid che ha assegnato il parametro di ritorno della
```

```

fork() restituisce il PID del figlio
***/
printf("processo padre in esecuzione \n");
printf("PID padre : %d  PID figlio: %d \n",getpid(), pid);

}
else
{
// codice eseguito dal padre in caso di errore
printf("Si e' verificato un errore nella chiamata a fork.\n");
}
}

```

Ora mandiamo in esecuzione il programma con il **compilatore gcc**.
 Posizioniamoci da terminale sulla directory in cui abbiamo salvato il sorgente e digitiamo il comando:

```
gcc -o fork fork.c
```

per compilare il file fork. C ed ottenere l'eseguibile fork.

Ora è possibile mandare in esecuzione il programma fork digitando

```
./fork
```

L'esecuzione sarà qualcosa del tipo

```

processo in esecuzione prima della fork 3186
processo padre in esecuzione
PID padre : 3186  PID figlio: 3187
processo figlio in esecuzione
PID padre : 3186  PID figlio: 3187

```

Tuttavia notiamo che, se mandiamo di nuovo in esecuzione il programma potremmo ottenere una sequenza esecutiva diversa, ad esempio

```

processo in esecuzione prima della fork 3191
processo padre in esecuzione
processo figlio in esecuzione
PID padre : 3191  PID figlio: 3192
PID padre : 3191  PID figlio: 3192

```

Cio' perchè l'esecuzione di un programma parallelo è legata agli algoritmi di schedulazione dell'OS, alle politiche di assegnazione della CPU e al numero di core presenti, quindi l'output può essere diverso ad ogni esecuzione.

Un esercizio che può evidenziare bene l'esecuzione parallela dei processi è questo: scriviamo un listato che, con una for di 10 passi mandi in stampa una variabile nel processo padre e un'altra nel processo figlio.

```
#include<stdio.h>
```

```

#include<stdlib.h>
#include<unistd.h>
#include<sys/types.h>

int main()
{
    int pid=getpid();
    printf("processo in esecuzione prima della fork %d \n",pid);
    pid=fork();

    if (pid == 0)
    {
        /*** codice eseguito dal figlio

        ***/
        int i;
        for (i=0;i<10; i++)
            printf("processo figlio passo: %d \n",i );
        sleep ( 2);
    }
    else if (pid > 0)
    {
        int j;
        for (j=0;j<10; j++)
            printf("processo padre passo: %d \n",j );
        sleep ( 2);
    }
    else
    {
        // codice eseguito dal padre in caso di errore
        printf("Si e' verificato un errore nella chiamata a fork.\n");
    }
}

```

Se mandiamo più volte in esecuzione questo programma possiamo vedere che i passi si alternano in modo diverso , con output legato alle reali esecuzioni nelle diverse situazioni.

Facciamo ora un altro esempio per studiare gli effetti della clonazione dei segmenti di memoria tra processo padre e processo figlio.

Creiamo un sorgente simile al precedente, ma inseriamo la dichiarazione di una variabile intera globale, i, inizializzata a 0. Proviamo a incrementare questa variabile sia nel processo padre che nel processo figlio e visualizziamone il valore.

```

#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/wait.h>
int i=0;
int main()
{
    int pid=getpid();

```

```

printf("processo in esecuzione prima della fork %d \n",pid);
pid=fork();

if (pid == 0)
{
    /*** codice eseguito dal figlio
    incremento di i e visualizzazione
    ***/
    printf("processo figlio in esecuzione \n");
    printf("PID padre : %d PID figlio: %d \n",getppid(), getpid());
    i++;
    printf("valore di i secondo il figlio %d \n",i);
    sleep(2);
}
else if (pid > 0)
{
    /***codice eseguito dal padre
    incremento di i e visualizzazione
    ***/
    printf("processo padre in esecuzione \n");
    printf("PID padre : %d PID figlio: %d \n",getpid(), pid);
    sleep (3);
    i++;
    printf(" valore di i secondo il padre %d \n",i);
}
else
{
    // codice eseguito dal padre in caso di errore
    printf("Si e' verificato un errore nella chiamata a fork.\n");
}

printf(" valore di i in uscita %d \n",i);
}

```

In output si avrà qualcosa del tipo

```

processo in esecuzione prima della fork 5275
processo padre in esecuzione
PID padre : 5275 PID figlio: 5276
processo figlio in esecuzione
PID padre : 5275 PID figlio: 5276
valore di i secondo il figlio 1
valore di i in uscita 1
valore di i secondo il padre 1
valore di i in uscita 1

```

Come si vede nell'output il valore della variabile i è rimasto 1. Infatti i segmenti di memoria utilizzati dai singoli processi sono completamente separati, le modifiche sulle variabili, come l'incremento di i, non hanno alcun effetto sul valore che le stesse variabili hanno negli altri processi.

La terminazione di un processo avviene mediante la chiamata della funzione **exit()**.

Se un processo padre termina prima dei propri processi figli genera una situazione di processi orfani. Un **processo orfano** è un processo privo di padre che viene immediatamente adottato da init (PID 1). Questo è il tipico caso dei programmi **demoni**, che rimangono residenti in memoria, ma vengono svincolati dal padre che li ha generati.

Ecco di seguito un esempio di sorgente con un processo orfano

```
include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/wait.h>

int main()
{

    pid_t pidpadre=getpid(); //acquisisco il pid del processo padre e lo stampo
    printf("pid del padre in esecuzione prima della fork %d \n",pidpadre);
    pid_t pid=fork();
    if (pid > 0)
    {

        /**codice eseguito dal padre che termina con exit dopo una pausa ***/
        sleep(1);
        exit(0);
    }
    else if (pid==0)
    {
        while (getppid()==pidpadre)
        {
            printf("padre ancora in esecuzione \n");
            sleep(1);
        }
    }
    printf("padre terminato \n figlio affidato a init con PID %d", getppid());
}
```

Quando un processo figlio termina con la funzione **exit()** i valori di uscita del processo vengono salvati nel descrittore di processo, in particolare la ragione della terminazione e il valore dello stato di uscita, e tali valori possono essere recuperato dal processo padre per determinare l'esito dell'esecuzione del figlio.

Per fare questo, il processo padre deve sospendere la propria attività : deve infatti sincronizzarsi col figlio, attendendo la sua terminazione con l'invocazione della funzione **wait()**.

Se viene invocata la **wait()** il processo padre si blocca fino alla prima terminazione di un processo figlio, e la **wait** restituisce come parametro il PID del processo figlio appena terminato.

Se si vuole evitare il blocco del processo padre si puo' utilizzare la chiamata **waitpid ()** che permette di specificare quale processo si vuole terminare passandone il PID come parametro.

Se il processo chiamante invece non ha figli, la chiamata fallisce, ritornando un parametro negativo.

Se il processo padre non recupera l'exit status del figlio, quest'ultimo diventa un **processo zombie**. Infatti solo a seguito di tale chiamata il PID e il PCB vengono liberati per poter essere riutilizzati da altri processi.

Fin quando non avviene tale chiamata il processo rimane nello stato di **zombie**.

Un processo zombie, o processo defunto è dunque un processo che, nonostante abbia terminato la propria esecuzione, possiede ancora un PID ed un PCB (process control block).

Quando un processo termina, tutta la memoria e le risorse ad esso associate vengono liberate così da poter essere utilizzate da altri processi, ma il PCB del processo resta nella tabella dei processi (process table), per consentire al processo padre di leggerne il valore di uscita eseguendo la chiamata di sistema wait(). Solo a seguito della wait() il processo zombie viene definitivamente rimosso. In seguito alla rimozione del processo zombie, i relativi PID e process control block possono essere riutilizzati.

Quando il processo padre non invoca la chiamata di sistema wait, il processo zombie continua a rimanere nella tabella dei processi: in alcuni casi ciò viene fatto di proposito, ad esempio quando il processo padre, creando un nuovo processo figlio, vuole assicurarsi che questo abbia un PID diverso da un figlio creato in precedenza appena terminato.

In realtà alcuni sistemi Unix-like prevedono un rilascio implicito, ma ciò non è assicurato per tutti gli OS pertanto l'unico sistema di sicuro rilascio rimane l'invocazione della funzione wait().

Il problema creato dall'esistenza dei processi zombie non nasce naturalmente dal consumo di memoria, in quanto, appunto, il processo ha terminato la sua esecuzione e non ha più risorse allocate, quanto piuttosto dall'esaurimento dei PID disponibili e il riempimento della tabella dei processi quando il sistema lavora a carichi importanti

Ecco un listato di terminazione senza zombie

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/wait.h>
```

```
int main()
{
    int stato,pid,pidwait;
    pid=fork();
```

```
if (pid > 0)
{
    /**codice eseguito dal padre ***/
    printf("processo padre in esecuzione \n");
    printf("PID padre : %d PID figlio: %d \n",getpid(), pid);
    pidwait=wait(&stato);
}
```

```
else if (pid == 0)
{
    /** codice eseguito dal figlio ***/
    printf("processo figlio in esecuzione \n");
    printf("PID padre : %d PID figlio: %d \n",getppid(), getpid());
    sleep ( 2);
    exit(1);
```

```

    }
    printf("programma terminato senza zombie \n");
}

```

Ecco invece un listato che lascia uno zombie: basterà togliere la funzione wait()

```

#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/wait.h>

int main()
{
    int pid=fork();

    if (pid > 0)
    {
        /**codice eseguito dal padre ***/
        printf("processo padre in esecuzione \n");
        printf("PID padre : %d PID figlio: %d \n",getpid(), pid);
    }

    else if (pid == 0)
    {
        /** codice eseguito dal figlio ***/
        printf("processo figlio in esecuzione \n");
        printf("PID padre : %d PID figlio: %d \n",getppid(), getpid());
        exit(1);
    }
    printf("programma terminato con zombie \n");
}

```

E' possibile **individuare i processi zombie esistenti** invocando da terminale il comando ps
ps aux | grep defunct

Per eliminare un processo zombie occorre eliminare il suo processo padre , inviando un segnale di tipo kill; in tal modo , il processo zombie diviene un **processo orfano** che viene adottato da init che esegue periodicamente le chiamate di sistema wait() e waitpid() eliminando così i propri processi figli zombie. Per individuare il PID del padre da killare è possibile sempre sfruttare il comando ps -def | grep PIDfiglio, per poi eseguire il kill del PID evidenziato.

La funzione fork è lo strumento tipico utilizzato per mandare in crash un sistema, con la cosiddetta **bomba fork** un attacco che agisce creando velocemente un gran numero di processi , così da saturare lo spazio disponibile tabella dei processi che viene mantenuta dall' OS. Questo impedisce di avviare ulteriori programmi perchè, eventuali spazi che si possono creare, vengono immediatamente occupati dalle istanze della bomba in attesa. Inoltre la bomba fork rallenta il

sistema, utilizzando ampiamente i tempi di cpu e gli spazi di memoria, rendone di fatto impossibile l'utilizzo.

Ecco il listato di una bomba fork

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#
int main()
{
    while(1)
        fork();
}
```

Una volta che una bomba fork è stata attivata su un sistema, può essere impossibile ripristinarne la normale operatività senza forzarne un riavvio dal momento che l'unica soluzione ad una bomba fork è quella di distruggerne tutte le istanze.