CREAZIONE PROCESSO

mercoledì 29 marzo 2023 16:38

Creazione di un processo

pid_t fork(void)

Descrizione	invoca la duplicazione del processo chiamante
Restituzione	1) nel chiamante: pid del figlio, -1 in caso di errore 2) nel figlio: 0

Su unix abbiamo una System_Call che ci permette di tirare su un nuovo processo - in particolare di attivare una nuova istanza di programma: Fork

fork() provoca la creazione di un nuovo processo. Il nuovo processo (processo figlio) è una copia esatta del processo chiamante (processo padre).

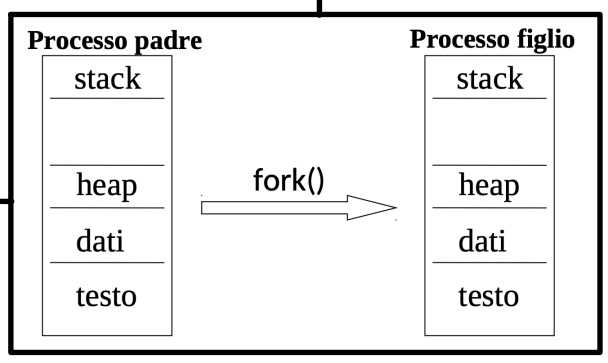
• Il processo figlio ha un ID processo univoco.

Questa fork è una funzione della libreria standard di sistema:

#include <unistd.h>

Quindi è il sistema operativo che genera il nuovo processo.

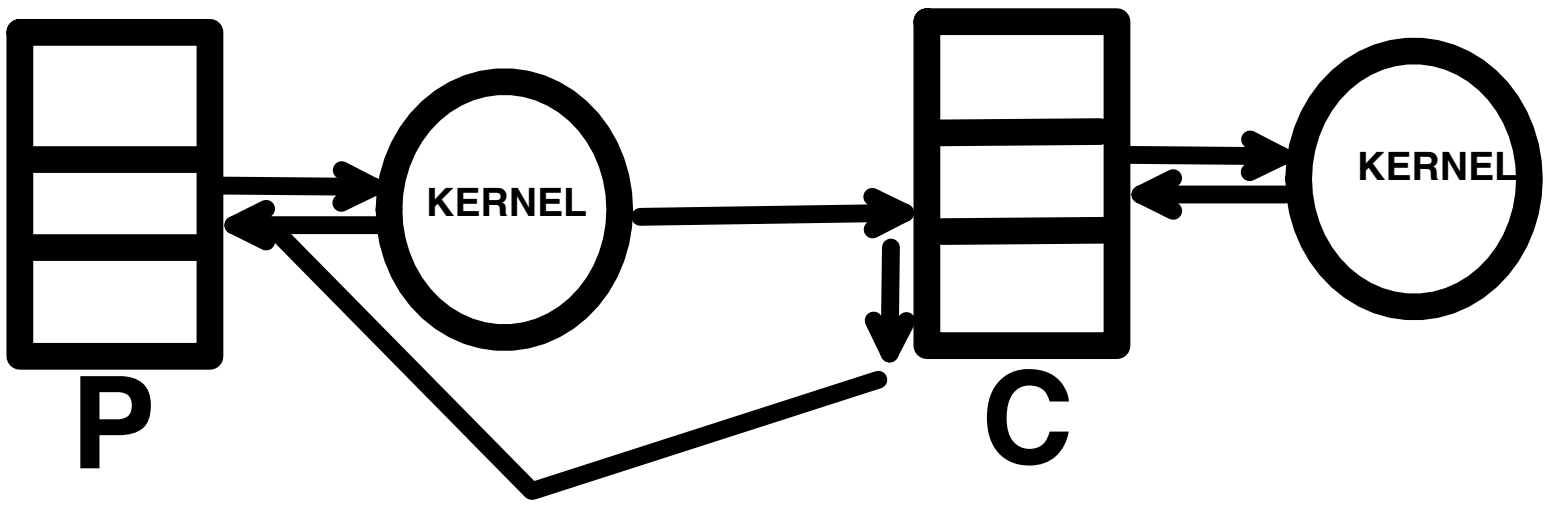
Nel campo informatico, fork() è il metodo principale per la creazione di processi su sistemi operativi UNIX. Questa funzione crea una nuova copia chiamata "figlio" del processo originale, ossia il genitore. Quando il processo padre si chiude o si arresta in modo anomalo per qualche motivo, termina anche il processo figlio.

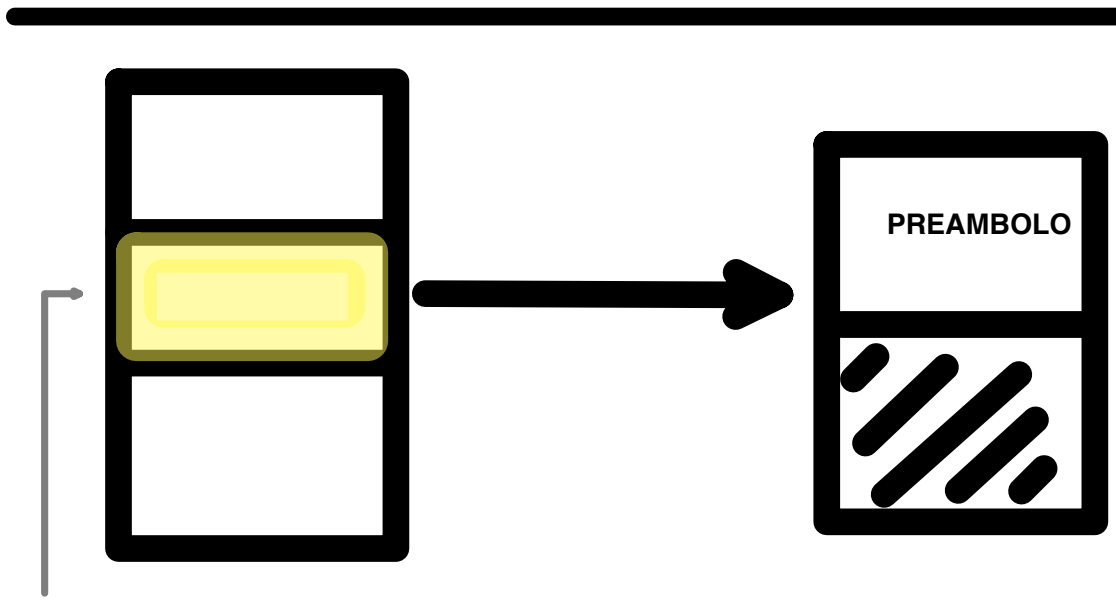


Questa funzione dello standard di sistema (System Call)non prende parametri.
Viene indicata la duplicazione del processo chiamante: quando un processo chiama una fork(), viene generato un altro processo che è esattamente il clone dell'istanza originale. Quindi nell'address space secondario che la funzione ha generato, abbiamo esattamente le stesse informazioni che avevamo prima di chiamare la fork().
Quando noi eseguiamo una fork in ambito UNIX, eseguiamo il clonaggio di un processo, ovvero del processo che chiama la fork() appunto.
Nel processo figlio dobbiamo capire qual è l'istruzione macchina che prende il controllo per prima quando la CPU verrà assegnata a questo processo.

Supponiamo che all'interno della zona .TEXT dell'address space del processo Padre ci sia il blocco di codice che implementa la system call FORK(), questa istruzione macchina porta il controllo fuori al kernel, poi il kernel restituirà il controllo all'istruzione macchina successiva. Nel momento in cui passiamo il controllo al kernel perché abbiamo chiamato fork() dobbiamo considerare che di processi ce ne saranno più di uno, ci sarà il parent e il child, il child con lo stesso address space del parent, quindi è ovvio e scontato pensare che ci sarà all'interno della zona .TEXT del child il blocco di codice che implementa la system call e infatti c'è proprio l'istruzione che - se eseguita - porta il controllo al kernel.
Quando il kernel prima o poi decide che nel Time-Sharing il controllo deve andare al processo figlio, in realtà in che punto di questo codice, contenuto nel .TEXT di questo address space, prenderemo il controllo?

Prenderemo il controllo all'istruzione macchina che ha dato il controllo al kernel nel parent.
I due processi riprendono esattamente la stessa istruzione macchina che è la successiva all'istruzione macchina che nel processo parent ha passato il controllo al kernel.
Quando noi generiamo un clone, esso non rifarà lo stesso lavoro del parent. Perché il parent fa delle cose e poi passa ad eseguire istruzioni macchina successiva rispetto all'istruzione macchina che passa il controllo al kernel.
La stessa cosa la farà anche il child e seppure child è un clone del parent non dobbiamo fare esattamente le stesse cose del genitore.





Entrambi i processi ripartono dall'istruzione successiva alla trap al kernel dovuta alla `fork()`

ORA ZOOM: Abbiamo il testo all'interno dell'address space, abbiamo il blocco di codice macchina che implementa la fork.

Il controllo viene passato al Kernel e abbiamo una coda di istruzioni macchina per prelevare il valore di ritorno di questa System_Call.

In realtà il parent comincia ad eseguire dall'istruzione macchina successiva all'istruzione che ha portato il controllo al kernel, quando il kernel ritorna il controllo.
La stessa cosa viene fatta nel child: anche il child esegue il blocco di istruzioni macchina, ovviamente non esegue la chiamata al Kernel che è stata eseguita dal parent, e noi riprendiamo esattamente dallo stesso punto perché siamo il clone.
La system call nel momento in cui viene generato il child, ha due ritorni: uno nel child ed uno nel parent.
La coda sotto viene eseguita sia nel parent che nel child! 2 RITORNI.
Ovviamente il preambolo sopra viene eseguito solo nel parent!
IL CHILD VIVE SOLO PERCHÉ L'ISTRUZIONE MACCHINA CHE PASSA IL CONTROLLO AL KERNEL È STATA ESEGUITA.

Nel parent ritorniamo il PID del figlio, infatti il valore di ritorno della funzione è proprio di tipo PID_T, nel caso non ci sono errori: se c'è un errore ritorniamo -1.
Nel figlio tipicamente il valore di ritorno è 0.
Nel codice, dopo la fork(), basta verificare se il valore di ritorno è un PID o è un codice 0.

Al completamento con successo, fork() restituisce un valore pari a 0 al processo figlio e restituisce il PID del processo figlio al processo padre.
In caso contrario, viene restituito un valore di -1 al processo padre, non viene creato alcun processo figlio e viene impostata la variabile globale
errno per indicare l'errore.

```
void main(int argc, char **argv){
     pid_t pid;   int status;
     pid = fork();
     if ( pid == 0 ){
               printf("processo figlio\n");
               exit(0);
     }
     else{
               printf("processo padre, attesa terminazione figlio\n");
               wait(&status);
     }
}
```

Se il PID==0, Allora stiamo eseguendo all'interno del CLONE child.

Terminazione su richiesta (definizione esplicita di un codice di uscita)

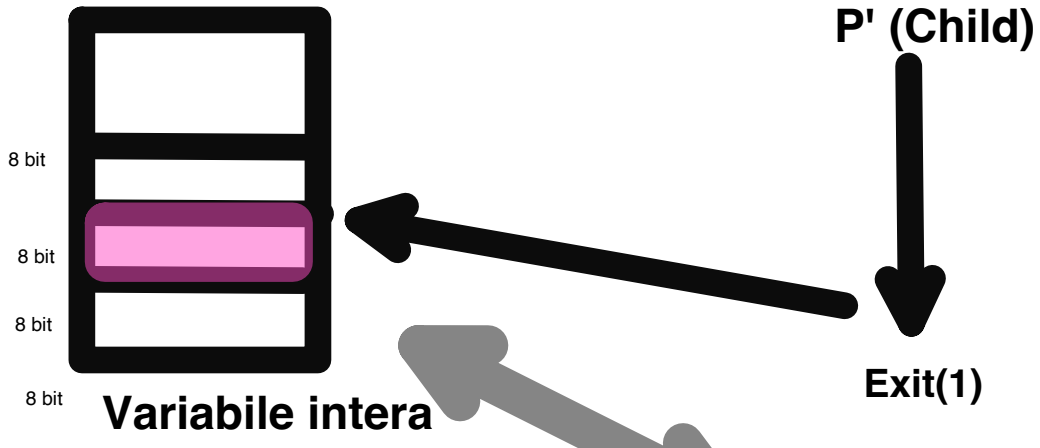
Con exit(0) il child sta entrando in uno stato zombie e non riprenderà più il controllo della cpu.

Se il PID è diverso da 0, andiamo nel ramo else e siamo il processo PADRE.

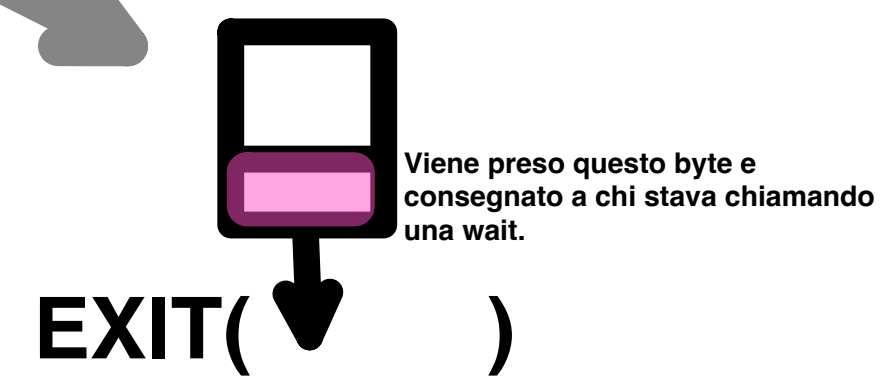
pid_t wait(int *status)	
Descrizione	invoca l'attesa di terminazione di un generico proc. figlio
Parametri	codice di uscita nei secondi 8 bit meno significativi puntata da status
Restituzione	-1 in caso di fallimento

NOI ABBIAMO un parent P che tramite una fork() genera un altro processo P', poi ognuno esegue attività differenti dall'altro.

P' non fa altro che tornare il valore 0 sulla exit quando chiama il Kernel, mentre invece il padre chiama una wait.
La wait è un servizio che permette ad un processo di chiamare il kernel per chiedere di essere messo in stato di sleep fin tanto che uno dei processi figli non ha terminato la sua esecuzione. Quindi P è in running, ma chiama il kernel per entrare in uno stato di wait. Appena qualcosa accade allora P andrà in Ready: ECCOLO LÍ. E poi quando il software del kernel ha deciso che è il mio turno io ripasso a running.
Il valore di ritorno della wait è un PID che mi dice qual è il valore del processo figlio per cui io mi sono risvegliato.
A Wait noi possiamo passare un puntatore ad intero, e quindi identifichiamo nell'address space una locazione di memoria che può ospitare un intero, dove il kernel ci va a restituire quello che è il codice di terminazione (per esempio 0) del processo child che ha terminato.
Quindi questo codice, chi ha chiamato la wait, eventualmente se lo troverà all'interno della variabile status di cui ha passato l'indirizzo: e quindi il main nell'esecuzione di questa applicazione nel processo parent, può andare a determinare qual è il codice di uscita del processo child (Scritto in status).
Noi sappiamo che la taglia di un intero è un certo numero di byte in memoria, questi codici di terminazione in realtà sono "SINGLE-BYTE":Quando noi passiamo un codice di terminazione al kernel. Attenzione: noi alla exit() passiamo un parametro, quello che succede al consegnare il parametro al secondo byte meno significativo, è sulla WAIT(), NON SU UNA EXIT(). Su exit noi consegnamo un codice numerico, in cui il kernel quando prende il controllo preleva questo codice
Abbiamo una variabile intera, se noi utilizziamo un pointer a questa variabile e chiamiamo una wait e da qualche parte il figlio P' ha consegnato una Exit con codice numerico zero, il software del sistema codice numerico e lo ritorna all'interno della variabile nel secondo byte. (Meno significativo)



Del parametro che passiamo ad exit, all'interno della variabile intera, viene considerata soltanto la parte meno significativa. Quando stai uscendo consegni un'informazione all'interno di una variabile e questa la passi all'interno della exit, ma di questa informazione l'unica cosa che conta è il byte meno significativo, perché questo è il valore effettivo che viene ad essere passato quando viene chiamata una wait a chi lo sta chiedendo. Quindi non possiamo consegnare codici di terminazione che vanno superiori a 255, 8 bit.



Quando si chiama una wait e si passa l'indirizzo di memoria di una variabile intera in cui si vogliono le informazioni a riguardo del child, è lì che il codice di terminazione viene scritto nel secondo byte ,meno significativo, ma gli altri 3 conserveranno altre informazioni.

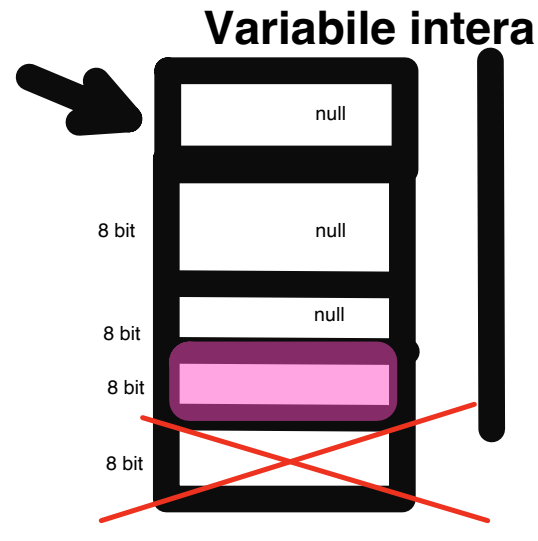
Il che implica dire che se noi veramente vogliamo scoprire qual è il codice di terminazione di questa applicazione P' che noi abbiamo generato tramite il clonaggio, dobbiamo riuscire ad estrarre questo secondo byte. Questo è quello che succede nell'esempio che abbiamo sotto:

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

int main(int a, char **b){

        pid_t pid;
        int exit_code;

        pid = fork();

        if (pid == 0){

                printf("child process simply exiting\n");
                exit(1);
        }

        sleep(7);
        printf("parent process simply goes waiting\n");
        wait(&exit_code);
        printf("parent process: child exited with code %d\n",exit_code>>8);
        pause();

}
```

Se il PID=0, sono il Child process e sto uscendo con il codice 1.

Se siamo nel parent questo IF non è verificato e ovviamente andiamo nelle

Exit_code>>8 da questo risultato, scarta l'ultimo record, ne aggiunge uno nuovo sopra e si legge l'informazione:



istruzioni successive.

Ci mettiamo in wait nel processo parent e chiaramente, quando riprendiamo il controllo, prendiamo exit_code e lo shiftiamo di 8 bit a destra.

Seguendo lo schema precedente, io sto mandando in output un intero, in cui ho scartato l'ultimo byte di sotto, perché ho shiftato, ho fatto entrare all'interno di questa struttura di intero quel valore (non all'interno della variabile) e lo sto mandando in output.
Poi andiamo in pausa, di fatto mi porta questa applicazione in stato di attesa.

```
DS/UNIX> ./a.out
child process simply exiting
parent process simply goes waiting
parent process: child exited with code 1
^C
```

Ora, supponiamo di avere un processo P attivo che esegue e chiama una wait per passare il controllo al kernel.
In questo caso il kernel dovrebbe verificare se uno dei processi figli di P ha completato la sua esecuzione!
Ma se P non ha lanciato processi CHILD?
Chiaramente questa WAIT ritornerà un codice di errore.
Il codice è -1, che andrà ad indicare che c'è stato un fallimento nel cercare di attendere qualcosa, ossia il completamento dei processi child, nel momento in cui è stata chiamata. Non c'erano processi CHILD.

ESEMPIO

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

#define NUM_FORKS 10

int main(int a, char ** b){

    int residual_forks = NUM_FORKS;

    another_fork:

    residual_forks--;
    if(fork()>0){
        pause();
    }
    else{
        if(residual_forks>0){   ;
            goto another_fork;
        }
    }
    pause();
}
```

Se il valore di ritorno della fork() è maggiore di 0, stiamo eseguendo nel processo padre, perché abbiamo detto che viene ritornato il PID del figlio.

Se nessuno emana segnalazioni io rimango in pausa indeterminata.

Se vado nel ramo else sono il processo figlio.

Il valore di residual_fork che il processo figlio si ritrova all'interno del suo address space è esattamente uguale al valore che avevamo settato prima di chiamare la fora, ossia è uguale a quello del processo padre (all'inizio).

POI D'ORA IN POI DECREMENTEREMO LA NOSTRA RESIDUAL_FORK, NON QUELLA DEL PADRE.

Eseguiamo nuovamente la fork e diventiamo nuovamente parent di un altro Child, e così via..

Il secondo Child si trova un residua fork decrementato due volte

```
OCESSES-AND-THREADS/UNIX> ./a.out &
[1] 23413
```

Questo è il codice generale del processo associato al programma appena compilato!

Ora andiamo a vedere i processi attualmente attivi nel sistema:

```
-THREADS/UNIX> ps --forest
  PID TTY          TIME CMD
13848 pts/1    00:00:01 bash
23413 pts/1    00:00:00  \_ a.out
23415 pts/1    00:00:00  |   \_ a.out
23416 pts/1    00:00:00  |       \_ a.out
23417 pts/1    00:00:00  |           \_ a.out
23418 pts/1    00:00:00  |               \_ a.out
23419 pts/1    00:00:00  |                   \_ a.out
23420 pts/1    00:00:00  |                       \_ a.out
23421 pts/1    00:00:00  |                           \_ a.out
23422 pts/1    00:00:00  |                               \_ a.out
23423 pts/1    00:00:00  |                                   \_ a.out
23424 pts/1    00:00:00  |                                       \_ a.out
23434 pts/1    00:00:00  \_ ps
```

Tutti questi processi sono attivi nel sistema, non stanno usando la CPU, HANNO CHIAMATO UNA SYSTEM CALL PAUSE(), e quest'ultima li ha messi in stato di attesa.

```
PLES/PROCESSES-AND-THREADS/UNIX> fg
./a.out
^C
```

```
PLES/PROCESSES-AND-THREADS/UNIX> ps --forest
  PID TTY          TIME CMD
13848 pts/1    00:00:01 bash
23477 pts/1    00:00:00  \_ ps
```

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

#define NUM_FORKS 10

int main(int a, char ** b){

        int residual_forks = NUM_FORKS;

        for(;residual_forks > 0 ; residual_forks--){
                if(fork()>0){
                        continue;
                }
                else{
                        break;
                }


        }
        pause();


}
```

```
PLES/PROCESSES-AND-THREADS/UNIX> ps --forest
  PID TTY          TIME CMD
13848 pts/1    00:00:01 bash
23511 pts/1    00:00:00  \_ a.out
23513 pts/1    00:00:00  |   \_ a.out
23514 pts/1    00:00:00  |   \_ a.out
23515 pts/1    00:00:00  |   \_ a.out
23516 pts/1    00:00:00  |   \_ a.out
23517 pts/1    00:00:00  |   \_ a.out
23518 pts/1    00:00:00  |   \_ a.out
23519 pts/1    00:00:00  |   \_ a.out
23520 pts/1    00:00:00  |   \_ a.out
23521 pts/1    00:00:00  |   \_ a.out
23522 pts/1    00:00:00  |   \_ a.out
23523 pts/1    00:00:00  \_ ps
```