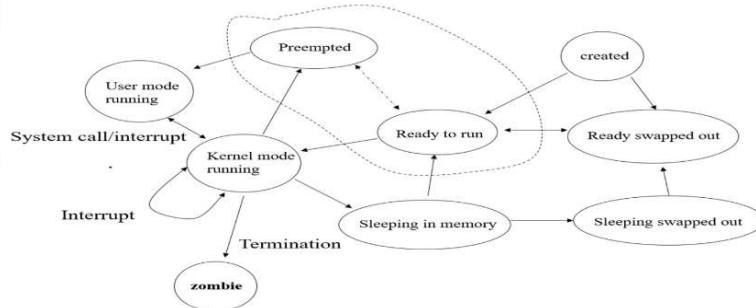


Processi e thread 3 Unix + fork

martedì 14 ottobre 2025 10:17

Dopo questa introduzione alla gestione dei processi, vediamo ora lo schema dei stati in UNIX :



La parte running viene suddivisa in due classi : user e kernel. Per quanto riguarda in vece lo stato di terminazione (che porta il processo in modalità zombie), si ha che applicazione è ancora all'interno del sistema , ma non può esser schedulato. Un esempio di un processo zombie si ha quanto un processo chiama la syscall EXIT() passando un valore di terminazione, aggiornando il PCB . Tornando all'immagine , vediamo ora quella di un processo UNIX il quale è composto da :

1. Contesto utente

- Testo dati stack utente e memoria condivisa
- Quindi si parla di address space

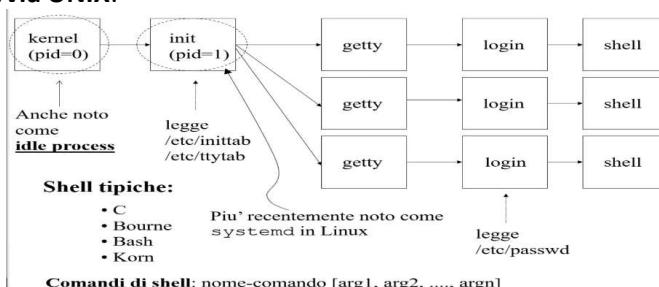
2. Contesto registri

- Info che riguardano lo stato della CPU (IP o PC), stack pointer e registri generali
- Quindi si parla di stato della cpu

3. Contesto sistema

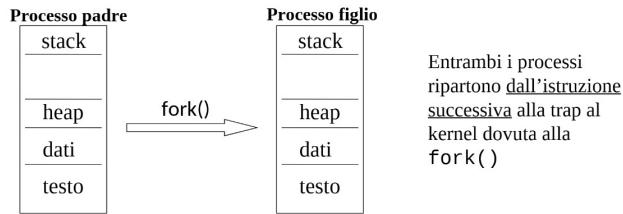
- Metadati aggiuntivi (entry tabella dei processi), user area , tabella di indirizzamento (memoria virtuale) e stack in modo kernel
- Qui si parla di informazioni addizionali

Vediamo ora come è fatta la tabella dei processi in UNIX: composta dallo stato del processo , **identificatore dell'utente** (reale ed effettivo) , identificatore di processi (pid, id genitore), descrizione dell'evento , **affinità processore** (CPU utilizzabile per svolgere un processo) , priorità, **segnali** (gestione eventi esterni), monitoraggio e lo stato della memoria (swap in/out); Per quanto riguarda la user area (seconda area stack del sistema) si hanno i seguenti campi : identificatore d'utente (effettivo / reale), array per gestione segnali , parametri di i/o (indirizzi buffer), valori di monitoraggio (valore ritorno syscall) e tabella descrittori dei file . **Dopo questa introduzione , vediamo ora come si avvia UNIX:**



Quindi il primo processo che viene caricato è il kernel , il quale ha valore 0 (**PID-> process id**) , il quale lancia altro processo **init** che ha codice numerico 1, il quale a sua volta legge file e lancia altri processi (getty-> get terminal) , passando per il login (leggendo user e pwd) da due file ed infine si arriva a lanciare la shell . **Vediamo ora come attivare un processo in UNIX:**

<code>pid_t fork(void)</code>	
Descrizione	invoca la duplicazione del processo chiamante
Restituzione	1) nel chiamante: pid del figlio, -1 in caso di errore 2) nel figlio: 0



Questa system call permette di tirare su un nuovo processo -> attivo nuova istanza di programma .

Quindi ci saranno 2 address space da gestire . Questa funzione permette di duplicare il processo chiamante , nel quale si ha nuova istanza che è clone di quello generante (stessi dati) . La ripresa dell'esecuzione riprende dall'istruzione successiva a quella dove viene invocato la fork, andando ad invocare le istruzioni della coda. È possibile sincronizzare il padre con la creazione del figlio?? Si .

Vediamo come :

<code>pid_t wait(int *status)</code>	
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

Vediamo ora in dettaglio :

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

#define NUM_FORKS 10

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

    int residual_forks = NUM_FORKS;

    another_fork:

    //decremento num_fork
    residual_forks--;
    //chiamo fork -> sto nel parent
    if(fork()>0){
        //pausa indeterminata
        pause();
    }
    else{
        //sono il processo figlio
        //residual fork del figlio è uguale a quello del padre
        if(residual_forks>0){
            //vado al codice identificato dalla label
            goto another_fork;
        }
    }
    pause();
    //tutti i processi entrano in pausa
}
```

Il quale programma compilato e facendo vedere il PID del processo con

```
gcc cascade-fork.c | ./a.out &
```

Ritorna un valore numerico

```

PID TTY      TIME CMD
9188 pts/1  00:00:00 bash
9310 pts/1  00:00:00  \_ a.out
9311 pts/1  00:00:00  |  \_ a.out
9312 pts/1  00:00:00  |  |  \_ a.out
9313 pts/1  00:00:00  |  |  |  \_ a.out
9314 pts/1  00:00:00  |  |  |  |  \_ a.out
9315 pts/1  00:00:00  |  |  |  |  |  \_ a.out
9316 pts/1  00:00:00  |  |  |  |  |  |  \_ a.out
9317 pts/1  00:00:00  |  |  |  |  |  |  |  \_ a.out
9318 pts/1  00:00:00  |  |  |  |  |  |  |  |  \_ a.out
9319 pts/1  00:00:00  |  |  |  |  |  |  |  |  |  \_ a.out
9320 pts/1  00:00:00  |  |  |  |  |  |  |  |  |  |  \_ a.out
9322 pts/1  00:00:00  |  |  |  |  |  |  |  |  |  |  |  \_ ps

```

Eseguendo ps --forest si ha effettivamente il risultato aspettato : ho creato 10 processi ed ogni volta che creo un processo, il processo creante diventa il padre, mentre quello creato diventa il figlio, i quali sono messi tutti in pause

Vediamo ora una modifica ovvero ho un solo padre e tanti figli :

```

#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();
}

```

Il quale compilato e mandato in esecuzione ed eseguendo ps --forest si arriva al seguente risultato

```

PID TTY      TIME CMD
10062 pts/3  00:00:00 bash
10160 pts/3  00:00:00  \_ a.out
10161 pts/3  00:00:00  |  \_ a.out
10162 pts/3  00:00:00  |  |  \_ a.out
10163 pts/3  00:00:00  |  |  |  \_ a.out
10164 pts/3  00:00:00  |  |  |  |  \_ a.out
10165 pts/3  00:00:00  |  |  |  |  |  \_ a.out
10166 pts/3  00:00:00  |  |  |  |  |  |  \_ a.out
10167 pts/3  00:00:00  |  |  |  |  |  |  |  \_ a.out
10168 pts/3  00:00:00  |  |  |  |  |  |  |  |  \_ a.out
10169 pts/3  00:00:00  |  |  |  |  |  |  |  |  |  \_ a.out
10170 pts/3  00:00:00  |  |  |  |  |  |  |  |  |  |  \_ a.out
10171 pts/3  00:00:00  |  |  |  |  |  |  |  |  |  |  |  \_ ps

```

Torniamo al discorso della sincronizzazione (vedendo che legame c'è tra la wait del padre e l'exit del figlio:

```

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);
    }
}

```

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

In questo esempio il padre che genera il figlio con la fork, si mette in attesa indefinita , **finché uno dei figli non termina la sua esecuzione**: quindi il processo padre si trova dallo stato di run a wait/blocked per poi tornare a ready quando il figlio termina. Nota : **wait() ritorna il pid del processo che mi ha svegliato**. Il parametro della wait è un puntatore al codice di terminazione del

processo child che ha terminato, il quale parametro viene messo nel secondo byte meno significativo. In dettaglio (scopriamo il codice di terminazione dell'applicazione clonata) :

```
#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== -1)
    {
        printf("%s", "errore nella fork\n");
        exit(-1);
    }

    if (pid == 0){

        sleep(4);
        printf("child process exiting\n");
        exit(1);
    }

    printf("parent process goes waiting\n");
    pid = wait(&exit_code);
    if(pid == -1){
        printf("wait error\n");
        exit(EXIT_FAILURE);
    }
    printf("parent process: child exited with code %d\n",exit_code>>8);
}
}
```

Il quale output è il seguente :

```
parent process goes waiting
child process exiting
parent process: child exited with code 1
```

Vediamo ora altre funzioni :

```
SYNOPSIS
#include <sys/types.h>
#include <unistd.h>
pid_t getpid(void);           Proprio PID
pid_t getppid(void);          Parent PID

SYNOPSIS
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);      PID del processo da attendere
pid_t waitpid(pid_t pid, int *status, int options);  Parametrizzazione
                                                               dell'esecuzione
```

Ed ora un esempio :

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

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

    pid_t pid, ppid;
    int exit_code;

    if (a < 2){
        printf("invalid number of arguments\n");
        exit(EXIT_FAILURE);
    }
    pid = fork();

    if (pid == 0){

        ppid = strtol(b[1],NULL,10);
        printf("reference process id is %d\n",ppid);
        redo:
        printf("\nchild process querying parent id is
%d\n",getpid());
        if (getppid() == ppid){
            printf("now my parent is systemd - I'm exiting\n");
            exit(0);
        }
        printf("going to sleep\n");
        fflush(stdout);
        sleep(3);
        goto redo;
    }

    sleep(1);
    exit(0);
}
```