

Sviluppo di software di rete con l'utilizzo di sistemi UNIX

Gestione dei processi

Claudio Vicari

Process identifier: PID

- ➔ il PID è un numero intero, che identifica univocamente un processo nel sistema
- ➔ proprio per questo, viene usato anche nelle funzioni come `tmpnam`
- ➔ il processo `init` ha sempre il PID 1
- ➔ non tutti i processi sono anche file su disco: esistono processi del kernel, come lo `swapper`, o il processo per la gestione delle interruzioni

PID: funzioni

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

```
pid_t getpid();           // ottenere il PID
pid_t getppid();          // parent PID
```

```
uid_t get_uid();          // User ID
uid_t get_euid();         // effective UID
```

```
gid_t getgid();           // Group ID
gid_t getegid();          // effective Group ID
```

La funzione fork

pid_t fork();

- ➔ la funzione fork è l'unica primitiva di sistema per creare nuovi processi in un UNIX
- ➔ il nuovo processo creato è chiamato *processo figlio* (child), ed è una copia esatta del processo che chiama la funzione. Quindi, non condividono lo spazio di memoria!!
- ➔ anche se viene chiamata una sola volta, ritorna due volte:
 - nel processo figlio, restituisce il valore 0
 - nel processo padre, restituisce il PID del figlio

La funzione fork

Visualizza esempio

- ➔ notiamo cosa succede alle variabili, globali e della funzione
- ➔ notiamo come si può usare il valore di ritorno di fork per sapere se siamo nel parent o nel child
- ➔ controllare cosa succede se l'output è a video oppure su file!!

fork e condivisione dei file

- ➔ anche i file descriptor vengono duplicati dalla fork, quindi entrambi i processi possono continuare a scrivere
- ➔ però, l'offset nel file è condiviso! Se entrambi i processi continuano a scrivere nello stesso file, l'output risultante è mescolato...
- ➔ normalmente, le cose si gestiscono in uno di questi modi:
 - il parent aspetta la terminazione dei child, e poi scrive
 - il parent chiude i FD che non gli servono, e prosegue normalmente, mentre i child continuano sulla loro strada... (tipico nei server di rete)

Funzioni exit

- ➔ i processi hanno vari modi di uscire
- ➔ modi di terminazione normale, che forniscono un valore di ritorno al chiamante:
 - `return` dalla funzione `main`
 - usare la funzione `exit` o `_exit`. La prima chiama anche le funzioni registrate con `atexit()`
- ➔ terminazione anomala:
 - funzione `abort`
 - alla ricezione di alcuni segnali

Funzioni wait e waitpid

```
pid_t wait(int *status);  
pid_t waitpid(pid_t pid, int *status, int  
options);
```

- ➔ alla fine di un processo figlio, il padre riceve un segnale SIGCHLD
- ➔ questo gli dà l'opportunità di conoscere lo stato di uscita del figlio, ed agire di conseguenza
- ➔ è consigliato gestire il segnale e fare una `wait`, perché alcune risorse relative al processo vengono mantenute finché i dati non vengono raccolti da `wait`
- ➔ `waitpid` può specificare un processo preciso, oppure (con `pid=-1`), comportarsi come `wait`

Funzioni wait e waitpid

- ➔ `wait` restituisce il PID del processo terminato, o -1 in caso di errore
- ➔ `waitpid`, in più, può restituire 0:
 - quando nelle opzioni si specifica `WNOHANG`, per uscire immediatamente se non ci sono child già terminati
- ➔ ci sono varie macro a disposizione, per avere dettagli ulteriori:
 - `WIFEXITED(s)` ci dice se è terminato normalmente
 - `WEXITSTATUS(s)` per ottenere lo stato di uscita
 - altre macro per sapere se `SIGCHLD` è stato generato a causa di un segnale, e per sapere se il processo è stato solamente fermato (STOP)

esempio: funzione pr_exit

```
void pr_exit( int status ) {
    if(WIFEXITED(status))
        printf("normal term, status=%d\n",
            WEXITSTATUS(status));
    else if (WIFSIGNALED(status))
        printf("abnormal term, signal=%d%s\n",
            WTERMSIG(status),
#ifdef WCOREDUMP
            WCOREDUMP(status) ? " (core file
generated)" : "");
#else
            "" );
#endif
    else if (WIFSTOPPED(status))
        printf("child stopped, signal = %d\n",
            WSTOPSIG(status));
}
```

esercizi

- ➔ creare un processo figlio, provare a terminarlo:
 - normalmente
 - con SIGSEGV da un altro processo
 - facendone stop (con SIGSTOP o altri...)
- ➔ e vedere cosa succede, utilizzando `wait` e la funzione `pr_exit`

Funzioni exec

```
int execl(const char *path, const char *arg,  
...);
```

```
int execlp(const char *file, const char *arg,  
...);
```

```
int execlx(const char *path, const char *arg ,  
..., char * const envp[]);
```

```
int execv(const char *path, char *const argv[]);
```

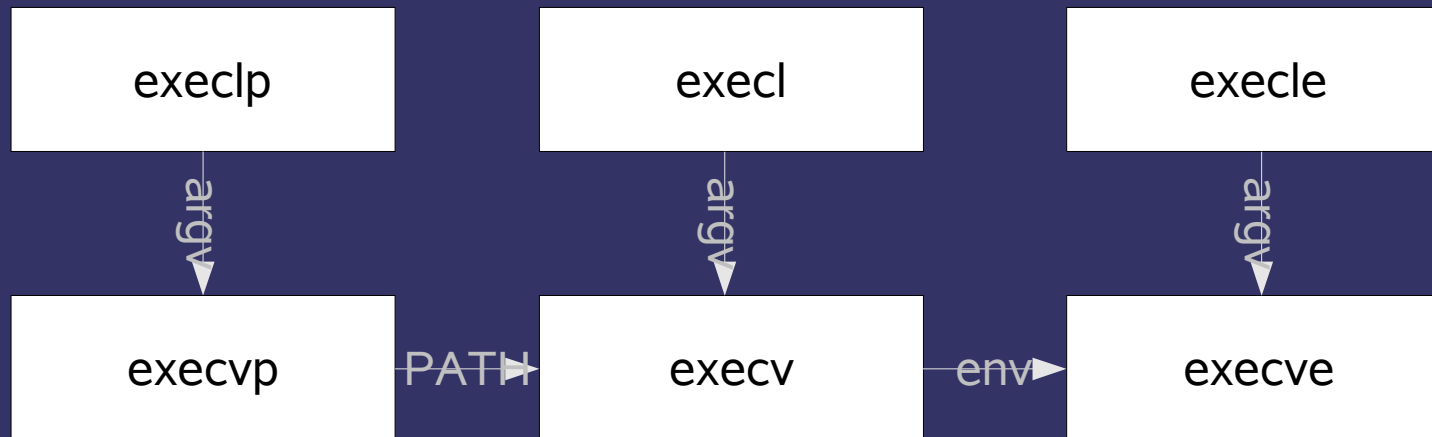
```
int execvp(const char *file, char *const argv[]);
```

```
int execve(const char *filename, char *const argv  
[], char *const envp[]);
```

Funzioni exec

- ➔ questa famiglia di funzioni sovrascrive il processo corrente, con un nuovo processo
- ➔ dal manuale:
 - tutte queste funzioni consentono di specificare un nome di file da eseguire, e una lista di argomenti da fornirgli
 - `execlp`, `execvp` si comportano come la shell, cercando l'eseguibile dalle directory specificate nella variabile d'ambiente `PATH`. Come la `bash`, possono eseguire anche file non eseguibili direttamente dal sistema
 - `execle`, `execve` permettono di specificare anche l'ambiente di esecuzione
- ➔ POSIX definisce almeno 4096 byte a disposizione per specificare gli argomenti

Funzioni exec



execve è la funzione di sistema!!

Esempio per exec

Visualizza esempio