System calls e Linux

(Stevens)
Capitolo 8: Process Control

System Calls e Funzioni di libreria

- Tutti i sistemi operativi provvedono service points attraverso i quali i programmi richiedono servizi dal kernel.
- Unix prevede un ben definito e limitato numero di entry points direttamente nel kernel, chiamati ***system calls***.
- Questi entry points in Unix sono direttamente accessibili al programmatore C.
  - A differenza di sistemi operativi più antiquati in cui erano accessibili solo attraverso routines in liguaggio macchina.
- La tecnica usata e' questa:
  - per ogni system call esiste una funzione con lo stesso nome nella libreria standard di C.
  - La chiamata a questa funzione C invocherà l'appropriato servizio kernel usando qualsiasi tecnica sia appropriata in quello specifico sistema.

System Calls e Funzioni di libreria (II)

- Dal punto di vista del programmatore sia system calls che funzioni di libreria appaiono come normali funzioni C.
  - Mentre però sarà sempre possibile riscrivere le funzioni di libreria, ovviamente non sarà possibile rimpiazzare o riscrivere le system-calls.
- In <sys/types.h> sono presenti alcune definizioni di tipi di dati dipendenti dalla macchina usati dal sistema.
- Questi tipi di dati sono detti ***primitive system data types***
  - la maggior parte di questi tipi di dati finisce in ***_t***
  - Ad es. ***off_t***, ***dev_t***, etc.

vi

- Due “modi” di lavoro: insert mode e command mode
- “**vi programma.c**” permette di entrare in vi, editando il file programma.c.
- Si entra in *command mode,* dando il comando ***i*** si va in *insert mode*.
- Dando ***escape*** (o ctrl-c) si esce dall’insert mode e si va in command mode.
- Per salvare il file che si stà editando il comando (da command mode) e’ **:w** oppure ***:w nomefile***.
- ***:wq*** salva il file ed esce dall’editor.
- ***:q*** esce dall’editor (se il file corrente non e’ stato modificato
- ***:q!*** esce dall’editor senza aggiornare le modifiche.
- ***:r nomefile*** inserisce nel testo il file ***nomefile*** nella riga sotto il cursore.

vi e gcc

- ***x*** cancella un carattere.
- ***r c*** rimpiazza con un qualsiasi carattere ***c*** il carattere su cui e' posizionato il cursore.
- ***R*** rimpiazza tutti i caratteri fino ad **escape**.
- ***dw*** cancella una parola davanti al cursore.
- ***db*** cancella una parola dietro al cursore.
- ***dd*** cancella una linea.
- Per compilare un programma C useremo il compilatore gcc.
- ***gcc nomefile.c*** compila ***nomefile.c*** e se la compilazione ha successo salva l'eseguibile nel file ***a.out***.
- ***gcc -o nomefile nomefile.c*** compila ***nomefile.c*** **e** salva l'eseguibile come ***nomefile***.

Identificatori di processi

- Ogni processo ha un identificatore univoco: un intero positivo.
- Esistono system call che permettono ad un processo di conoscere il suo pid ed anche il pid del padre.

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

pid_t getpid (void); process ID del processo chiamante
pid_t getppid (void); process ID del padre del processo
    chiamante
```

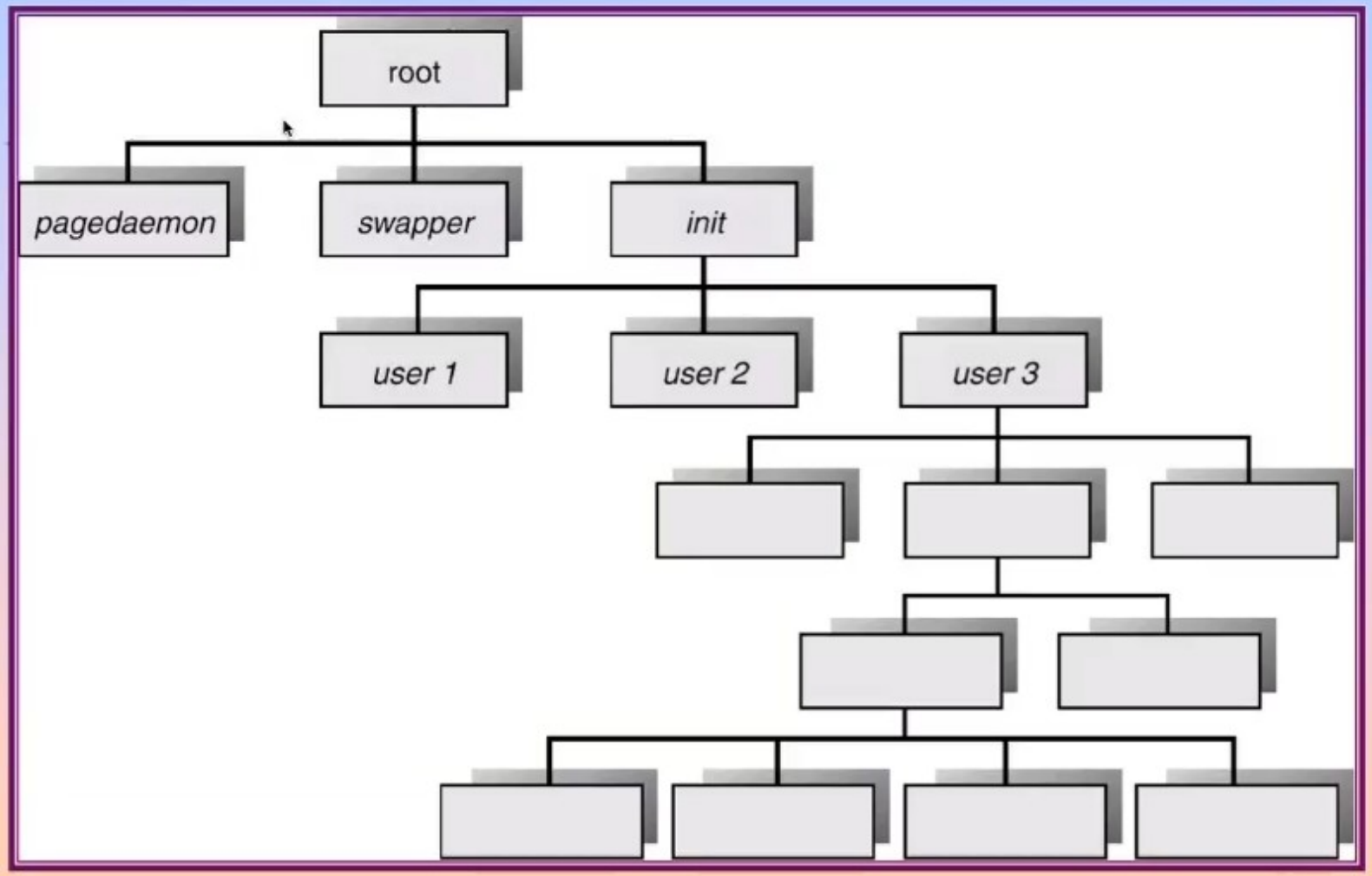
Identificatori di processi (II)

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

int main(void)
    {
    printf("pid del processo = %d\n", getpid());
    printf("pid del padre del processo = %d\n", getppid());

    return (0);
}
```

Albero di processi in un sistema UNIX



Creazione di nuovi processi

- L'unico modo per creare nuovi processi è attraverso una chiamata a sistema ***fork*** da parte di un processo già esistente.
- Quando viene chiamata, la fork genera un nuovo processo detto figlio.
- Dalla fork si ritorna due volte:
  - il valore restituito al processo figlio e' 0,
  - il valore restituito al padre è il pid del figlio,
    - ✓ un processo può avere più figli e non c'e' nessuna system call che permette al padre di recuperare il pid dei suoi figli.
- Figlio e padre continuano ad eseguire concorrentemente le istruzioni che seguono la chiamata di fork.

Creazione di nuovi processi (II)

- Il figlio è una ***copia*** del padre.
- condividono dati, stack e heap,
  - il kernel protegge questi dati settando i permessi come read-only.
- Solo se uno dei processi tenta di modificare una di queste strutture dati questa viene fisicamente copiata e duplicata (COW - Copy On Write).
- E' l'algoritmo di schedulazione dello specifico SO che deciderà se eseguire prima il processo figlio o quello padre.

fork

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

pid_t fork(void);
```

Restituisce: 0 nel figlio,
pid del figlio nel padre
-1 in caso di errore

Creazione di un processo in Unix

```
# include <stdio.h>
void main (int argc, char *argv[ ])
{ int pid;
    /* genera un nuovo processo */
    pid = fork ();
    if  (pid < 0)     { /* errore */
                      fprintf(stderr, "errore");
                      exit(-1);
                      }
          else        if (pid == 0)        { /* processo figlio */
                                             execlp ("/bin/ls", "ls", NULL);
                                             }
                                   else      { /* processo genitore */
                                             wait (NULL);
                                             printf("Il figlio ha terminato");
                                             exit(0);
                                             }
}
```

Funzioni wait e waitpid

- Quando un processo termina il kernel manda al padre il segnale SIGCHLD.
- Il padre può ignorare il segnale (default) oppure lanciare una funzione (signal handler).
- In ogni caso il padre può chiedere informazioni sullo stato di uscita del figlio.
- Questo è fatto chiamando le funzioni ***wait*** e ***waitpid***.

wait

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

pid_t wait (int *statloc);
```

- Chiamata da un processo padre ottiene in statloc lo stato di terminazione di un figlio
- Restituisce:
  - PID se OK,
  - -1 in caso di errore.

waitpid

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

pid_t waitpid (pid_t pid, int *statloc, int options);
```

- Chiamata da un processo padre chiede lo stato di terminazione in statloc del figlio specificato in pid.
- Il processo padre si blocca in attesa o meno secondo il contenuto di options.

- Restituisce:
  - PID se OK,
  - 0 oppure -1 in caso di errore

Argomento pid in waitpid

- In waitpid l'interpretazione dell' argomento ***pid*** dipende dal suo valore:
  - pid == -1: aspetta per un qualunque processo figlio (analogo di wait),
  - pid > 0: aspetta il figlio che ha process id uguale al valore di pid,
  - pid == 0: aspetta un qualsiasi figlio che ha process group id uguale a quello del processo che ha chiamato waitpid,
  - pid < -1: aspetta un qualsiasi figlio che ha process group id uguale valore assoluto di pid.
- Con waitpid si avrà errore se il processo o il process group non esistono.
- L'argomento ***options*** serve a controllare ulteriormente waitpid.
- Può essere 0 (nessuna opzione) oppure ad esempio uguale a WNOHANG, nel qual caso waitpid non bloccherà il chiamante se il figlio specificato da pid non è immediatamente disponibile ma restituirà 0).

wait e waitpid

- in generale con la funzione wait:
  - il processo si blocca in attesa (se tutti i figli stanno girando)
  - ritorna immediatamente con lo stato di un figlio
  - ritorna immediatamente con un errore (se non ha figli).
- Un processo puo' chiamare wait quando riceve SIGCHLD, in questo caso ritorna immediatamente con lo stato del figlio appena terminato.
- waitpid può scegliere quale figlio aspettare (1 argomento).
- wait può bloccare il processo chiamante (se non ha figli che hanno terminato), mentre waitpid ha una opzione (WNOHANG) per non farlo bloccare e ritornare immediatamente.
- Quindi waitpid (al contrario di wait) ci permette di aspettare uno specifico processo e provvede una versione non-blocking di wait.

System calls di *exec*

- ***fork*** di solito è usata per creare un nuovo processo (*processo figlio*).
- Il figlio potrebbe dover eseguire un programma diverso da quello del padre,
  - potrà farlo chiamando una delle funzioni di ***exec***.
- Dopo la chiamata ad una funzione di exec, lo spazio indirizzi del figlio è completamente rimpiazzato dal nuovo programma.
  - non è però cambiato il pid del figlio.

System calls di *exec* (II)

- L'unico modo per creare un processo è attraverso la system call ***fork***.
- L'unico modo per eseguire un nuovo programma o comando è attraverso una delle system call di ***exec***.
- Una chiamata di ***exec*** reinizializza un processo: il text segment (segmento istruzioni) e lo user data segment (segmento dati utente) cambiano (viene eseguito un nuovo programma)
- Il segmento dati di sistema rimane invariato.

exec

```
#include <unistd.h>

int execl (const char *path, const char *arg0, ... /* (char *) 0 */);
int execv (const char *path, char *const argv[ ]);
int execle (const char *path, const char * arg0, ... /*(char *) 0, char
                                                    *const envp[ ] */);
int execve (const char *path, char *const argv[ ], char *const
                                                    envp[]);
int execlp (const char *file, const char * arg0, ... /*(char *) 0 */);
int execvp (const char *file, char *const argv[ ]);
```

- Restituiscono: -1 in caso di errore
  non ritornano se OK.

exec (II)

- Quando un processo chiama una delle exec, quel processo è completamente rimpiazzato dall'esecuzione del nuovo programma.
- Il suo PID non cambia,
  - perchè non viene creato un nuovo processo.
- ***exec*** rimpiazza il processo corrente (text, dati, heap, stack, etc.) con un nuovo programma.

exec: differenze

- Le lettere finali dele 6 funzioni sono un aiuto a ricordare gli argomenti delle funzioni stesse.
- La lettera **l** indica che la funzione avrà una **lista di argomenti**,
- invece la lettera **v** indica un **vettore argv[]**.
- Quindi le funzioni ***execlp***, ***execl***, ed ***execle*** richiederanno che gli argomenti sulla linea di comando per il nuovo programma siano specificati come argomenti separati della funzione.
  - L'ultimo argomento sarà un puntatore nullo,
    - ✓ da notare il cast a (char *) 0.
- Per le altre tre funzioni sarà necessario passare l'indirizzo di un array di puntatori agli argomenti sulla linea di comando del nuovo programma.

exec: differenze (II)

- La lettera **p** (***execlp***, ***execvp***), indica che la funzione prende come argomento un nome di file ed utilizza la variabile di ambiente **PATH** per cercare la posizione nel file system del file eseguibile.
  - La variabile **PATH** contiene una lista di directories separate da ":".
    - ✓ Ad es. PATH = /bin : /usr/bin : /usr/local/bin : .
- Se non vi è **p** allora sarà indicato il pathname completo del file.
- La lettera **e** (***execle***, ***execve***) indica che la funzione prende un **envp[]** array invece di usare l'environment (variabili di ambiente) correnti.
- In alcune versioni di UNIX solo una di queste funzioni (***execve***) è una system call
  - le altre sono solo funzioni di libreria che successivamente chiamano ***execve***.

Terminazione

Un processo può terminare:

involontariamente:

- tentativi di azioni illegali
- interruzione mediante segnale

salvataggio dell'immagine nel file core

volontariamente:

- chiamata alla funzione exit()
- esecuzione dell'ultima istruzione

Terminazione (II)

- Per ogni processo in esecuzione, il kernel esegue il codice del processo e determina lo stato di terminazione.
- Se normale, lo stato è l'argomento di **exit.**
- altrimenti il kernel genera uno stato di terminazione che indica il motivo "anormale“.

- In entrambi i casi il padre del processo ottiene questo stato da **wait** o **waitpid**

exit ()

#include <stdlib.h>

void exit (int status);

- la system call exit prevede un parametro (status) mediante il quale il S.O. può comunicare al padre del processo che termina informazioni sul suo stato di terminazione (ad es., l'esito della sua esecuzione).
- chiude tutti i descrittori di file di un processo, dealloca le aree codice, dati e stack, e quindi fa terminare il processo
- è sempre una chiamata senza ritorno

wait & exit ()

Rilevazione dello stato di terminazione tramite **wait**:

- in caso di terminazione di un figlio, la variabile status raccoglie il suo stato di terminazione;
- nell'ipotesi che lo stato sia un intero a 16 bit: se il byte meno significativo di status è zero, il più significativo rappresenta lo stato di terminazione (terminazione volontaria).
- in caso contrario, il byte meno significativo di status descrive il segnale che ha terminato il figlio (terminazione involontaria).

wait & exit: Esempio

```
main()
{
int pid, status;
pid=fork();
if (pid==0)
    {printf("figlio"); exit(0);}
  else {pid=wait(&status);
        printf("terminato processo figlio n.%d", pid);
        if ((char)status==0)
                printf("term. volontaria con stato %d", status>>8);
            else printf("terminazione involontaria per segnale %d\n",
                                                    (char)status);}
}
```

wait & exit: Esempio (II)

Rilevazione dello stato:

È necessario conoscere la rappresentazione di status:

lo standard Posix.1 prevede delle macro (definite nell'header file <sys/wait.h> per l'analisi dello stato di terminazione.

In particolare:

- WIFEXITED(status): restituisce vero, se il processo figlio è terminato volontariamente: in questo caso la macro WEXITSTATUS(status) restituisce lo stato di terminazione.
- WIFSIGNALED(status): restituisce vero, se il processo figlio è terminato involontariamente: in questo caso la macro WTERMSIG(status) restituisce il numero dell'interruzione SW che ha causato la terminazione.

wait & exit: Esempio (III)

```
#include <sys/wait.h>
main()
{int pid, status;
pid=fork();
if (pid==0)
        {printf("figlio");
        exit(0); }
else { pid=wait(&status);
       if (WIFEXITED(status))
          printf("Terminazione volontaria di %d con stato %d\n", pid,
                                                    WEXITSTATUS(status));
          else if (WIFSIGNALED(status)) printf("terminazione involontaria per
                                                segnale %d\n", WTERMSIG(status));
           }
}
```

Cosa succede quando un processo termina?

- Se un figlio termina prima del padre, allora il padre può ottenere lo stato di terminazione del figlio con wait (o waitpid).
- Se un padre termina prima del figlio, allora il processo init diventa il nuovo padre.
- Se un figlio termina prima del padre ma il padre non recupera il suo stato tramite una wait (o waitpid), il figlio diventa uno zombie.
- Uno zombie non ha aree codice, dati o pila allocate, quindi non usa molte risorse di sistema ma continua ad avere un PCB nella Process Table (di grandezza fissa). Quindi la presenza di molti processi zombie potrebbe costringere l'amministratore ad intervenire

Primitive di controllo dei processi UNIX

- con la system call di **exit** viene chiuso il ciclo delle primitive di controllo dei processi UNIX
  - ***fork***: creazione nuovi processi
  - ***exec***: esecuzione nuovi programmi
  - ***wait / waitpid***: attesa fine processo
  - ***exit***: fine processo

system

#include <stdlib.h>

int system (const char *cmdstring);

- Serve ad eseguire un comando shell dall'interno di un programma.
  - Ad esempio: ***system("date > file");***
- Viene spesso implementata utilizzando le system call ***fork***, ***exec*** e ***waitpid***.