

System call: controllo dei processi (creazione, terminazione, esecuzione)

Roberto De Virgilio

Sistemi operativi - 4 Dicembre 2017

Controllo processi

- **Identificatore di processo**

- ogni processo ha un identificatore univoco (intero non negativo)
- utilizzato spesso per creare altri identificatori e garantire unicità

- **Identifieri standard:**

- pid 0: Non assegnato o assegnato a un kernel process
- pid 1: Processo **init (/sbin/init)**
 - Invocato dal kernel al termine della procedura di bootstrap
 - Porta il sistema ad un certo stato (ad es. multiuser)

Controllo processi

- **Process id ed altri identificatori**

- `pid_t getpid();` // Process id of calling process
- `pid_t getppid();` // Process id of parent process
- `uid_t getuid();` // Real user id
- `uid_t geteuid();` // Effective user id
- `gid_t getgid();` // Real group id
- `gid_t getegid();` // Effective group id

Controllo processi - Creazione

- **System call: pid_t fork();**
 - crea un nuovo processo child, copiando completamente l'immagine di memoria del processo parent
 - *data, heap, stack* vengono copiati
 - il **codice** viene spesso condiviso
 - in alcuni casi, si esegue **copy-on-write**
 - sia il processo child che il processo parent continuano ad eseguire l'istruzione successiva alla **fork**
 - **fork** viene chiamata una volta, ma ritorna due volte
 - processo *child*: ritorna **0**
(E' possibile accedere al pid del parent tramite la system call **getppid**)
 - processo *parent*: ritorna il **process id** del processo child
 - **errore**: ritorna valore **negativo**

Controllo processi - Creazione

- **Esempio: fork1.c**

- Scopo di questo esempio è mostrare alcune caratteristiche della creazione di processi

```
#include    <stdio.h> /* standard I/O stream of C language */
#include    <sys/types.h> /* standard POSIX types */
#include    <unistd.h> /* miscellaneous symbolic constants and types */
#include    <stdlib.h>   /* standard of C language for performing general functions */

int  glob = 6;      /* external variable in initialized data */
char buf[] = "fork1 write to stdout\n";
```

Controllo processi - Creazione

- Esempio: fork1.c

```
int main(void) {  
    int var;      /* automatic variable on the stack */  
    pid_t pid;  
  
    var = 88;  
  
    /* STDOUT_FILENO is a file descriptor of LINUX system */  
  
    if (write(STDOUT_FILENO, buf, sizeof(buf)) != sizeof(buf)) {  
        printf("Cannot write!!\n");  
        exit(1);  
    }  
}
```

Controllo processi - Creazione

- Esempio: `fork1.c`
- **STDOUT_FILENO** è un file descrittore nel sistema LINUX.

`STDERR_FILENO`: File number of `stderr`; 2.

`STDIN_FILENO`: File number of `stdin`; 0.

`STDOUT_FILENO`: File number of `stdout`; 1.

- **write** è una funzione in `unistd.h` per scrivere *nbyte* di *buf* nel *file*.

```
ssize_t write(int file, const void *buf, size_t nbyte);
```

- **exit** è una funzione in `stdlib.h` per terminare il programma chiamante.

```
void exit(int status);
```

Controllo processi - Creazione

- Esempio: fork1.c

```
printf("before fork\n"); /* we don't flush stdout */

if ( (pid = fork()) < 0) {
    printf("Cannot fork!!\n");
    exit(1);
}

else
    if (pid == 0) {      /* child */
        glob++;           /* modify variables */
        var++;
    }
    else
        sleep(2);         /* parent */

printf("pid = %d, glob = %d, var = %d\n", getpid(), glob, var);

exit(0);
}
```

Controllo processi - Creazione

- Esempio: fork1.c
- sleep è una funzione in `unistd.h` per sospendere il processo chiamante un tempo in secondi indicato da input

```
unsigned sleep(unsigned seconds);
```

Controllo processi - Creazione

- **Esempio: fork1.c**
 - Scopo di questo esempio è mostrare alcune caratteristiche della creazione di processi
- **Nota:**
 - inizializza un paio di variabili
 - stampa utilizzando `write`
 - stampa utilizzando `printf`
 - chiama `fork` e crea un processo child
 - continua in modo diverso:
 - Il processo child modifica le variabili
 - Il processo parent dorme per due secondi
 - stampa il contenuto delle variabili, etc.

Controllo processi - Creazione

■ Esecuzione:

- in generale, l'ordine di esecuzione tra child e parent dipende dal meccanismo di scheduling
- il trucco utilizzato in questo programma (`sleep` per due secondi) non garantisce nulla (`sleep` non è bloccante); meglio usare `wait()`

■ Output:

```
$ gcc fork1.c -o test
```

```
$ ./test
```

```
fork1 write to stdout
```

```
before fork
```

```
pid = 432, glob = 7, var = 89 variabili child cambiate
```

```
pid = 431, glob = 6, var = 88 variabili parent immutate
```

Controllo processi - Creazione

- Relazione tra `fork` e funzioni di I/O

```
$ ./test > temp.out ; cat temp.out
```

```
fork1 write to stdout
```

```
before fork
```

```
pid = 432, glob = 7, var = 89      variabili child cambiate
```

```
before fork
```

```
pid = 431, glob = 6, var = 88      variabili parent immutate
```

- Spiegazione:

- `write` non è bufferizzata
- `printf` è bufferizzata

Controllo processi - Creazione

■ Le modalità di bufferizzazione

- La **bufferizzazione** è una delle caratteristiche principali dell'interfaccia degli stream; lo scopo è quello di ridurre al minimo il numero di system call (**read** o **write**) eseguite nelle operazioni di input/output
- I caratteri che vengono scritti su di uno stream normalmente vengono accumulati in un **buffer** e **poi** trasmessi in **blocco** tutte le volte che il buffer viene riempito, in maniera **asincrona** rispetto alla scrittura

Controllo processi - Creazione

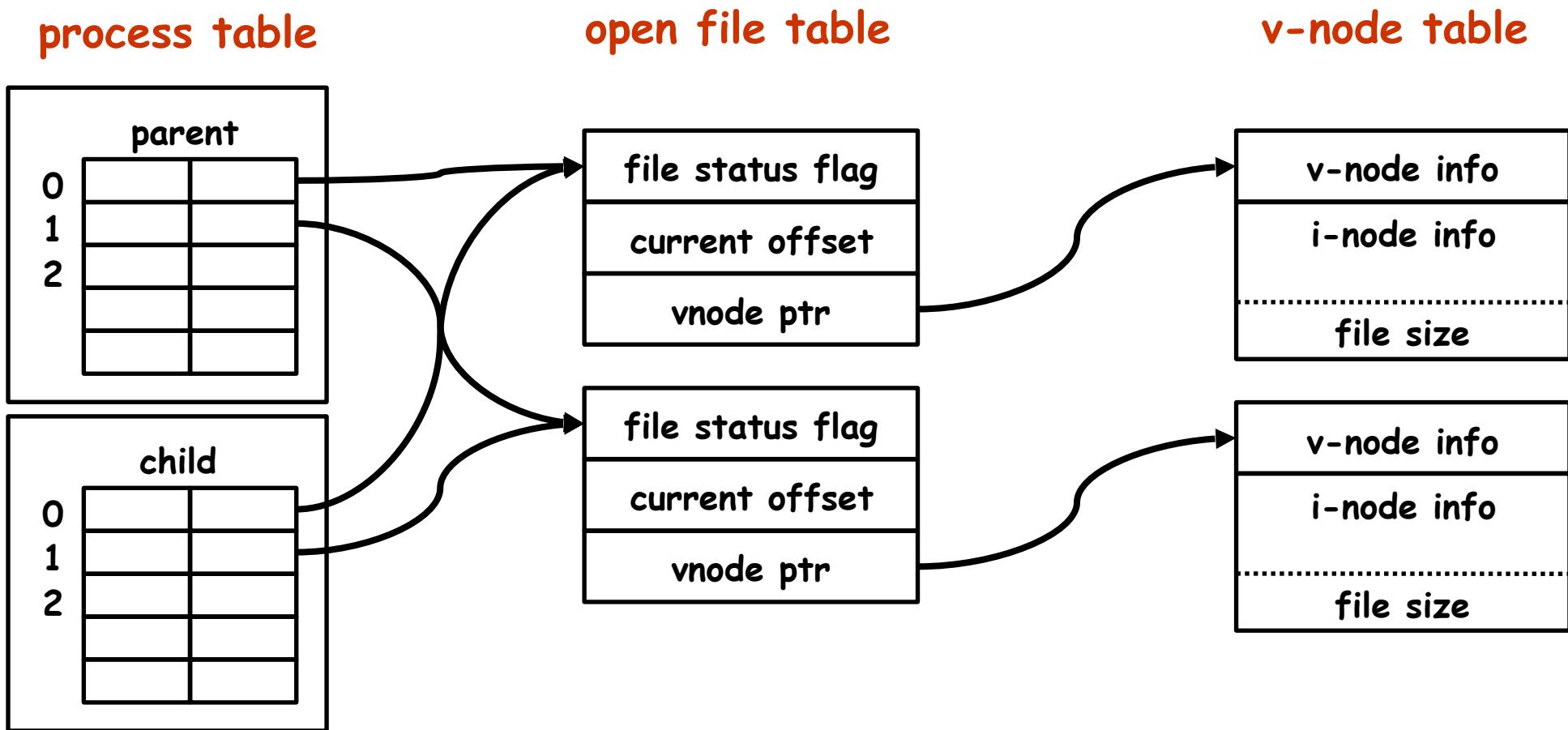
■ Le modalità di bufferizzazione

- lo standard definisce tre distinte modalità in cui può essere eseguita la bufferizzazione
 - **unbuffered**: in questo caso non c'è bufferizzazione ed i caratteri vengono trasmessi direttamente al file non appena possibile (effettuando immediatamente una **write**).
 - **line buffered**: in questo caso i caratteri vengono normalmente trasmessi al file in blocco ogni volta che viene incontrato un carattere di newline (il carattere ASCII **\n**).
 - **fully buffered**: in questo caso i caratteri vengono trasmessi da e verso il file in blocchi di dimensione opportuna.

Controllo processi - Creazione

■ Relazione tra `fork` e file aperti (redirezione)

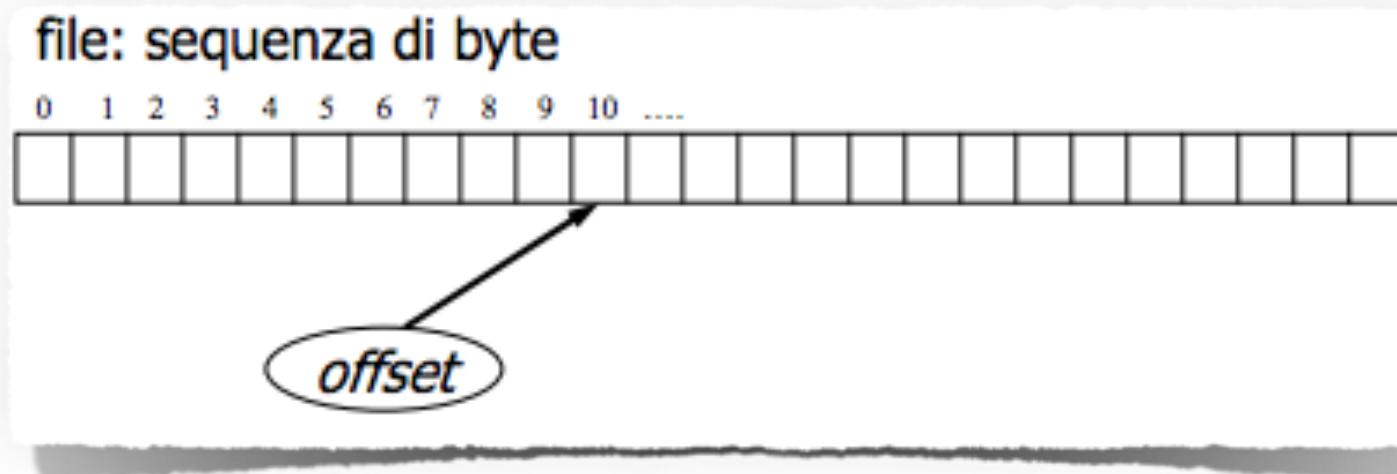
- una caratteristica della chiamata `fork` è che tutti i descrittori che sono aperti nel processo parent sono duplicati nel processo child



Controllo processi - Creazione

■ Relazione tra **fork** e file aperti (redirezione)

- è importante notare che parent e child condividono lo stesso file **offset**
- Ogni file aperto ha un offset (intero ≥ 0) che fornisce la posizione nel file, cioe' il numero di byte dall'inizio del file
- Operazioni di lettura e scrittura partono dall'offset corrente e causano un incremento pari al numero di byte letti o scritti



Controllo processi - Creazione

- **Relazione tra fork e file aperti (redirezione)**
 - è importante notare che parent e child condividono lo stesso file offset
 - esempio:
 - si consideri un processo che esegue **fork** e poi attende che il processo child termini (system call **wait**)
 - supponiamo che lo **stdout** sia rediretto ad un file, e che entrambi i processi scrivano su **stdout**
 - se parent e child non condividessero lo stesso offset, avremmo un problema:
 - il child scrive su **stdout** e aggiorna il proprio current offset
 - il parent sovrascrive **stdout** e aggiorna il proprio current offset

Controllo processi - Creazione

- **Come gestire i descrittori dopo una fork**

- il processo parent aspetta che il processo child termini
 - in questo caso, i file descriptor vengono lasciati immutati
 - eventuali modifiche ai file fatte dal processo child verranno riflesse nella file table entry e nella v-node entry del processo padre
- i processi parent e child sono indipendenti
 - in questo caso, ognuno chiuderà i descrittori non necessari e proseguirà opportunamente

Controllo processi - Creazione

- **Proprietà che il processo child eredita dal processo parent:**
 - real uid, real gid, effective uid, effective gid
 - gids supplementari
 - id del gruppo di processi
 - session ID
 - terminale di controllo
 - set-user-ID flag e set-group-ID flag
 - directory corrente
 - directory root
 - maschera di creazione file (umask)
 - maschera dei segnali
 - flag close-on-exec per tutti i descrittori aperti
 - environment

Controllo processi - Creazione

- Proprietà che il processo child non eredita dal processo parent:

- valore di ritorno di `fork`
- process ID
- process ID del processo parent
- file locks
- l'insieme dei segnali in attesa viene svuotato

Controllo processi - Creazione

■ Ragioni per il fallimento di `fork`

- il numero massimo di processi nel sistema è stato raggiunto
- il numero massimo di processi per user id è stato raggiunto

■ Utilizzo di `fork`

- quando un processo vuole duplicare se stesso in modo che parent e child eseguano parti diverse del codice
 - Network servers: il parent resta in attesa di richieste dalla rete; quando una richiesta arriva, viene assegnata ad un child, mentre il parent resta in attesa di richieste
- quando un processo vuole eseguire un programma diverso (utilizzo della chiamata `exec`)

Controllo processi - Creazione

- System call: `pid_t vfork();`
 - Stessa sequenza di chiamata e stessi valori di ritorno di `fork`
 - Semantica differente:
 - `vfork` crea un nuovo processo allo scopo di eseguire un nuovo programma con `exec()`
 - `vfork` non copia l'immagine di memoria del parent; i due processi condividono lo spazio di indirizzamento fino a quando il child esegue `exec()` o `exit()`
 - Guadagno di efficienza
 - `vfork` garantisce che il child esegua per primo:
 - continua ad eseguire fino a quando non esegue `exec()` o `exit()`
 - il parent continua solo dopo una di queste chiamate

Controllo processi - Creazione

- **Esempio: vfork1.c**

- Scopo di questo esempio è mostrare la differenza tra **vfork** e **fork**

```
#include    <stdio.h> /* standard I/O stream of C language */
#include    <sys/types.h> /* standard POSIX types */
#include    <unistd.h> /* miscellaneous symbolic constants and types */
#include    <stdlib.h>   /* standard of C language for performing general functions */

int  glob = 6;      /* external variable in initialized data */
```

Controllo processi - Creazione

- Esempio: vfork1.c

```
int main(void) {
    int var = 88;
    pid_t pid;

    printf("before vfork\n"); /* we don't flush stdio */

    if ((pid = vfork()) < 0){
        printf("Cannot vfork!!\n");
        exit(1);
    }

    else
        if (pid == 0) {          /* child */
            /* modify parent's variables */
            glob++;
            var++;
            _exit(0);           /* child terminates */
        }

    /* parent */
    printf("pid = %d, glob = %d, var = %d\n", getpid(), glob, var);
    exit(0);
}
```

Controllo processi - Creazione

- Esempio: `vfork1.c`
- `_exit` è una funzione in `unistd.h` per terminare il processo che ha invocato la funzione

```
void _exit(int status);
```

- provoca i seguenti effetti:
 - Ogni descrittore di file aperto dal processo chiamante, viene chiuso. I files condivisi con altri processi non vengono chiusi.
 - Ciascun figlio del processo corrente viene ereditato dal processo 1 (init, il padre di tutti i processi), sopravvivendo alla scomparsa del processo parent.
 - Il valore di status e' ritornato al padre del processo corrente che lo puo' rilevare tramite la funzione `wait()`.
 - Per convenzione un valore 0 di status indica una terminazione senza errori, mentre un valore diverso da 0 indica la presenza di una condizione di errore.

Controllo processi - Creazione

- **Esempio: vfork1.c**
 - Differenza tra **vfork** e **fork**
- **Nota:**
 - Incrementare le variabili nel child cambia i valori nell'immagine di memoria del parent
 - Il child chiama una versione speciale di **exit()**
 - Il parent non chiama **wait**, perché il child esegue prima
- **Output:**

```
$ gcc vfork1.c -o test
```

```
$ ./test
```

before vfork

pid = 432, glob = 7, var = 89 **variabili cambiate**

Controllo processi - Terminazione

- **Esistono tre modi per terminare normalmente:**
 - eseguire un **return** da main; equivalente a chiamare **exit()**
 - chiamare la funzione **exit**:
 - invocazione di tutti gli exit handlers che sono stati registrati
 - chiusura di tutti gli I/O stream standard
 - specificata in ANSI C; non completa per POSIX
 - chiamare la system call **_exit**
 - si occupa dei dettagli POSIX-specific;
 - chiamata da **exit**

Controllo processi - Terminazione

- **Esistono due modi per terminare in modo anormale:**
 - Quando un processo riceve certi segnali
 - Generati dal processo stesso
 - Da altri processi
 - Dal kernel
 - Chiamando **abort()**
 - Questo è un caso speciale del primo, in quanto genera il segnale **SIGABRT**
- **Terminazione normale/anormale**
 - Qualunque sia il modo di terminare del processo, l'effetto per il kernel è lo stesso:
 - rimozione della memoria utilizzata dal processo
 - chiusura dei descrittori aperti, etc.

Controllo processi - Terminazione

- **Exit status:**
 - Il valore che viene passato ad `exit` e `_exit` e che notifica il parent su come è terminato il processo (errori, ok, etc.)
- **Termination status:**
 - Il valore che viene generato dal kernel nel caso di una terminazione normale/anormale
 - Nel caso si parli di terminazione anormale, si specifica la ragione per questa terminazione anormale
- **Come ottenere questi valori?**
 - Tramite le funzioni `wait` e `waitpid` descritte in seguito

Controllo processi - Terminazione

- **Cosa succede se il parent termina prima del child?**
 - si vuole evitare che un processo divenga "orfano"
 - il processo child viene "adottato" dal processo **init** (pid 1)
 - meccanismo: quando un processo termina, si esamina la tabella dei processi per vedere se aveva figli; in caso affermativo, il parent pid viene posto uguale a 1
- **Cosa succede se il child termina prima del parent?**
 - se il child scompare, il parent non avrebbe più modo di ottenere informazioni sul termination/exit status del child
 - per questo motivo, alcune informazioni sul child vengono mantenute in memoria e il processo diventa uno **zombie**

Controllo processi - Terminazione

■ Status zombie

- vengono mantenute le informazioni che potrebbero essere richieste dal processo parent tramite `wait` e `waitpid`
 - Process ID
 - Termination status
 - Accounting information (tempo impiegato dal processo)
- il processo resterà uno zombie fino a quando il parent non eseguirà una delle system call `wait`

■ Child del processo init:

- non possono diventare zombie
- tutte le volte che un child di `init` termina, `init` esegue una chiamata `wait` e raccoglie eventuali informazioni
- in questo modo gli zombie vengono eliminati

Controllo processi - Terminazione

■ Notifica della terminazione di un figlio:

- Quando un processo termina (normalmente o no), il parent viene notificato tramite un segnale **SIGCHLD**
- Notifica asincrona
- Il parent:
 - Può ignorare il segnale (default)
 - Può fornire un signal handler (una funzione che viene chiamata quando il segnale viene lanciato)

■ System call:

```
pid_t wait(int *status);
```

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

- Utilizzate per ottenere informazioni sulla terminazione dei processi child

Controllo processi - Terminazione

- Quando un processo chiama `wait` o `waitpid`:
 - può bloccarsi
 - Se tutti i suoi child sono ancora in esecuzione
 - può ritornare immediatamente con il termination status di un child
 - Se un child ha terminato ed il suo termination status è in attesa di essere raccolto
 - può ritornare immediatamente con un errore
 - Se il processo non ha alcun child
- Nota:
 - se eseguiamo una system call `wait` poiché abbiamo ricevuto `SIGCHLD`, essa termina immediatamente
 - altrimenti può bloccarsi

Controllo processi - Terminazione

- **Significato degli argomenti:**

- **status** è un puntatore ad un intero; se diverso da **NULL**, il termination status viene messo nell'area di memoria puntata da questo argomento
- Il valore di ritorno è il process id del child che ha terminato

- **Differenza tra `wait` e `waitpid`:**

- **wait** può bloccare il chiamante fino a quando un qualsiasi child non termina;
- **waitpid** ha delle opzioni per evitare di bloccarsi
- **waitpid** può mettersi in attesa di uno specifico processo

Controllo processi - Terminazione

- Il contenuto del termination status dipende dall'implementazione:
 - bit per la terminazione normale, bit per l'exit status, etc.
- Standard POSIX:
 - Macro **WIFEXITED (status)**
 - Ritorna **true** se lo status corrisponde ad un child che ha terminato normalmente. In questo caso, possiamo usare:
 - Macro **WEXITSTATUS (status)**
 - Ritorna l'exit status specificato dal comando
 - Macro **WIFSIGNALED (status)**
 - Ritorna **true** se lo status corrisponde ad un child che ha terminato in modo anormale. In questo caso possiamo usare:

Controllo processi - Terminazione

■ Standard POSIX:

- Macro **WTERMSIG (status)**
 - Ritorna il numero di segnale che ha causato la terminazione
- Macro **WCOREDUMP (status)**
 - Ritorna **true** se un file core è stato generato
 - Non POSIX
- Macro **WIFSTOPPED (status)**
 - Argomento riguardante il job control
 - Ritorna **true** se il processo è stato stoppato, nel qual caso è possibile utilizzare:
- Macro **WSTOPSIG (status)**
 - Ritorna il numero di segnale che ha causato lo stop

Controllo processi - Terminazione

- **Esempio: prexit.c**
 - La funzione `pr_exit` utilizza le macro appena descritte per stampare una stringa che descrive il termination status

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

void pr_exit(int status)
{
    if (WIFEXITED(status))
        printf("normal termination, exit status = %d\n", WEXITSTATUS(status));
    else
        if (WIFSIGNALED(status))
            printf("abnormal termination, signal number = %d\n", WTERMSIG(status));
        else
            if (WIFSTOPPED(status))
                printf("child stopped, signal number = %d\n", WSTOPSIG(status));
}
```

Controllo processi - Terminazione

- **Esempio: prexit.c, wait1.c**
 - La funzione `pr_exit` utilizza le macro appena descritte per stampare una stringa che descrive il termination status
 - Il programma `main` utilizza questa funzione per dimostrare i valori differenti per il termination status

Controllo processi - Terminazione

- Esempio: prexit.c, wait1.c

```
int main(void) {
    pid_t pid;
    int status;

    if ((pid = fork()) < 0){
        printf("Cannot fork!!\n");
        exit(1);
    }
    else
        if (pid == 0) /* child */
            exit(7);

    if (wait(&status) != pid)      /* wait for child */
    {
        printf("Cannot wait!!\n");
        exit(1);
    }

    pr_exit(status);             /* and print its status */
```

Controllo processi - Terminazione

- Esempio: prexit.c, wait1.c

```
if ( (pid = fork()) < 0){
    printf("Cannot fork!!\n");
    exit(1);
}
else
    if (pid == 0)      /* child */
        abort();          /* generates SIGABRT */

if (wait(&status) != pid)      /* wait for child */
{
    printf("Cannot wait!!\n");
    exit(1);
}

pr_exit(status);              /* and print its status */
```

Controllo processi - Terminazione

- Esempio: prexit.c, wait1.c

```
if ( (pid = fork()) < 0){
    printf("Cannot fork!!\n");
    exit(1);
}
else
    if (pid == 0)          /* child */
        status /= 0;        /* divide by 0 generates SIGFPE */

    if (wait(&status) != pid) /* wait for child */
{
    printf("Cannot wait!!\n");
    exit(1);
}

pr_exit(status);           /* and print its status */

exit(0);
}
```

fatal arithmetic error

Controllo processi - Terminazione

- **Esempio: prexit.c, wait1.c**
 - La funzione `pr_exit` utilizza le macro appena descritte per stampare una stringa che descrive il termination status
 - Il programma `main` utilizza questa funzione per dimostrare i valori differenti per il termination status

▪ Output

```
$ gcc prexit.c wait1.c -o test
$ ./test
normal termination, exit status = 7
abnormal termination, signal number = 6
abnormal termination, signal number = 8
```

Controllo processi - Terminazione

- **System call:**

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

- Argomento **pid**:
 - **pid == -1** si comporta come **wait**
 - **pid > 0** attende la terminazione del child con process id corrispondente
 - **pid == 0** attende la terminazione di qualsiasi child con process group id uguale a quello del chiamante
 - **pid < -1** attende la terminazione di qualsiasi child con process group ID uguale a **-pid**

- **Options:**

- Il paramentro options puo' valere **0** o puo' essere in **OR** con i seguenti valori costanti:
 - **WNOHANG** specifica il ritorno immediato se i child non sono usciti
 - **WUNTRACED** specifica di ritornare anche se i child sono fermati (stop), e lo stato non deve venire riportato

Controllo processi - Terminazione

■ Esempio: fork2.c

- illustra come sia possibile evitare di stare in attesa di un processo figlio ed evitare che questo diventi uno zombie

```
#include <stdio.h> /* standard I/O stream of C language */
#include <sys/types.h> /* standard POSIX types */
#include <unistd.h> /* miscellaneous symbolic constants and types */
#include <stdlib.h>   /* standard of C language for performing general functions */
#include <sys/wait.h>

int main(void) {
    pid_t pid;

    if ((pid = fork()) < 0){
        printf("Cannot fork!!\n");
        exit(1);
    }
    else
        if (pid == 0) { /* first child */
            if ((pid = fork()) < 0){
                printf("Cannot fork!!\n");
                exit(1);
            }
            else
                if (pid > 0)
                    exit(0); /* parent from second fork == first child */
        }
}
```

Controllo processi - Terminazione

■ Esempio: fork2.c

```
/* We're the second child; our parent becomes init as soon
   as our real parent calls exit() in the statement above.
   Here's where we'd continue executing, knowing that when
   we're done, init will reap our status. */

    sleep(2);
    printf("second child, parent pid = %d\n", getppid());
    exit(0);
}

if (waitpid(pid, NULL, 0) != pid) /* wait for first child */
{
    printf("Cannot waitpid!!\n");
    exit(1);
}

/* We're the parent (the original process); we continue executing,
   knowing that we're not the parent of the second child. */

exit(0);
}
```

ritorna il process ID del child che termina

Controllo processi - Terminazione

- **Esempio: fork2.c**
 - illustra come sia possibile evitare di stare in attesa di un processo figlio ed evitare che questo diventi uno zombie
- **Nota:**
 - chiama fork due volte:
 - il parent effettua il primo **fork** e crea il primo child
 - il primo child effettua un secondo **fork** e crea il secondo child
 - il primo child termina
 - il second child viene adottato da **init**
 - il parent aspetta la terminazione del primo child e poi lavora in concorrenza
 - evitare i processi zombie ha il vantaggio di diminuire il numero di entry nella process table
- **Output**

```
$ gcc fork2.c -o test  
$ ./test  
second child, parent pid = 1
```

Controllo processi - Esecuzione

- **Quando un processo chiama una delle system call exec**
 - il processo viene rimpiazzato completamente da un nuovo programma (text, data, heap, stack vengono sostituiti)
 - il nuovo programma inizia a partire dalla sua funzione main
 - il process id non cambia

Controllo processi - Esecuzione

- Esistono sei versioni di exec:

```
int execl(char *pathname, char *arg0, ...);
```

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

```
int execle(char *pathname, char *arg0, ..., char* envp[]);
```

```
int execlp(char *filename, char *arg0, ...);
```

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

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

- Nota:

- Le funzioni execl(), execlp(), execle(), execv(), execvp() sono una interfaccia per la funzione **execve()**
- Normalmente una sola di queste è una system call, le altre sono chiamate di libreria
- **pathname** e' un programma eseguibile (binario) o uno script che dovrà sostituire il processo corrente.

Controllo processi - Esecuzione

- Nel nome della funzione chiamata sono contenuti dei caratteri che indicano sia il tipo di argomenti accettati e sia il tipo di comportamento dalla funzione stessa.
- carattere **1**, **execl()**, **execvp()**, **execle()** :
 - La funzione accetta una *lista* di argomenti.
 - Il primo argomento e' convenzionalmente il nome del programma che verra' eseguito.
 - All'ultimo argomento valido deve seguire un puntatore a NULL.

Controllo processi - Esecuzione

- Nel nome della funzione chiamata sono contenuti dei caratteri che indicano sia il tipo di argomenti accettati e sia il tipo di comportamento dalla funzione stessa.
- carattere **v**, **execv()**, **execvp()**, **execve()** :
 - La funzione accetta un **vettore** di argomenti.
 - Il primo elemento (**argv[0]**) deve contenere il nome del programma da invocare.
 - L'ultimo argomento valido deve essere seguito da un elemento contenente un puntatore a NULL.

Controllo processi - Esecuzione

- Nel nome della funzione chiamata sono contenuti dei caratteri che indicano sia il tipo di argomenti accettati e sia il tipo di comportamento dalla funzione stessa.
- carattere **e**, **execle()**, **execve()** :
 - La funzione accetta fra i suoi parametri un **array** di stringhe dell'ambiente (**environment**) che deve essere passato al **pathname** da eseguire.
 - Ogni stringa di ambiente e' del formato **parametro=valore** (es: TERM=vt100).
 - L'array **envp[]** e' terminato da un elemento a NULL.
 - Le funzioni con il nome non comprendenti la lettera **e**, fanno ereditare l'ambiente del processo attuale il quale e' contenuto nella variabile esterna **environ**, al programma individuato da **pathname**.

Controllo processi - Esecuzione

- Nel nome della funzione chiamata sono contenuti dei caratteri che indicano sia il tipo di argomenti accettati e sia il tipo di comportamento dalla funzione stessa.
- carattere **p**, **execp()**, **execvp()** :
 - In questo caso, se il *pathname* non contiene il carattere slash */*, *pathname* viene ricercato in base alla variabile di ambiente **PATH**.
 - Se la variabile **PATH** non e' specificata, pathname viene per default ricercato nelle directory **/bin** e **/usr/bin**

Controllo processi - Esecuzione

Funzione	pathname	filename	arg list	argv[]	environ	envp[]
exec1	●		●		●	
execlp		●	●		●	
execle	●		●			●
execv	●			●	●	
execvp		●		●	●	
execve	●			●		●
lettere		p	I	v		e

Controllo processi - Esecuzione

- **Cosa viene ereditato da exec?**

- process ID e parent process ID
- real uid e real gid
- supplementary gid
- process group ID
- session ID
- terminale di controllo
- current working directory
- root directory
- maschera creazione file (umask)
- file locks
- maschera dei segnali
- segnali in attesa

Controllo processi - Esecuzione

- **Cosa non viene ereditato da exec?**
 - effective user id e effective group id
 - vengono settati in base ai valori dei bit di protezione
- **Cosa succede ai file aperti?**
 - Dipende dal flag close-on-exec che è associato ad ogni descrittore
 - Se close-on-exec è true, vengono chiusi
 - Altrimenti, vengono lasciati aperti (comportamento di default)

Controllo processi - Esecuzione

- **Esempio: echoall.c**

- il programma **echoall** stampa gli argomenti e le variabili di ambiente

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

int main(int argc, char *argv[]) {
    int i;
    char **ptr;
    extern char **environ;

    for (i = 0; i < argc; i++) /* echo all command-line args */
        printf("argv[%d]: %s\n", i, argv[i]);

    for (ptr = environ; *ptr != 0; ptr++) /* and all env strings */
        printf("%s\n", *ptr);

    exit(0);
}
```

Controllo processi - Esecuzione

- **Esempio: exec1.c, echoall.c**

- L'esempio effettua due **fork**, eseguendo il programma **echoall** che stampa gli argomenti e le variabili di ambiente
- Nota: i due processi sono in concorrenza

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

char *env_init[] = { "USER=unknown", "PATH=/tmp", NULL };
```

Controllo processi - Esecuzione

- Esempio: exec1.c, echoall.c

```
int main(void) {
    pid_t pid;

    if ((pid = fork()) < 0){
        printf("Cannot fork!!\n");
        exit(1);
    }
    else if (pid == 0) { /* specify pathname, specify environment */
        if (execle("./echoall",
                    "echoall", "myarg1",
                    "MY ARG2",
                    (char *) 0,
                    env_init) < 0){
            printf("Cannot execle!!\n");
            exit(1);
        }
    }
}
```

Controllo processi - Esecuzione

- Esempio: exec1.c, echoall.c

```
if (waitpid(pid, NULL, 0) < 0){
    printf("Cannot waitpid!!\n");
    exit(1);
}

if ( (pid = fork()) < 0){
    printf("Cannot fork!!\n");
    exit(1);
}
else if (pid == 0) { /* specify filename, inherit environment */
    if (execl("./echoall",
              "echoall",
              "only 1 arg",
              (char *) 0) < 0){
        printf("Cannot execl!!\n");
        exit(1);
    }
}
exit(0);
```

Controllo processi - Esecuzione

- **Output:**

```
$ gcc echoall.c -o echoall  
$ gcc exec1.c -o test  
$ ./test
```

```
argv[0]: echoall  
argv[1]: myarg1  
argv[2]: MY ARG2  
USER=unknown  
PATH=/tmp  
argv[0]: echoall  
argv[1]: only 1 arg  
USER=rdevirgilio  
HOME=/Users/rdevirgilio  
LOGNAME=rdevirgilio  
...  
...
```

Controllo processi - Esecuzione

- **Funzione: int system(char* command) ;**
 - Esegue un comando, aspettando la sua terminazione
 - Standard e system:
 - Funzione definita in ANSI C, ma il suo modo di operare è fortemente dipendente dal sistema
 - Non è definita in POSIX.1, perché non è un'interfaccia al sistema operativo
 - Definita in POSIX.2

Controllo processi - Esecuzione

- **Esempio di implementazione: system1.c**

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

int main (void) {
    char command[50];
    strcpy( command, "ls -l" );
    system(command);
    return(0);
}
```

Controllo processi - Esecuzione

- **Output:**

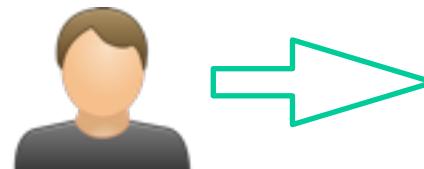
```
$ gcc system1.c -o test  
$ ./test
```

```
total 88  
-rw-----@ 1 rdevirgilio staff    432  3 Nov 00:23 echoall.c  
-rw-----@ 1 rdevirgilio staff   1225  3 Nov 00:23 exec1.c  
-rw-----@ 1 rdevirgilio staff    972  2 Nov 17:00 fork1.c  
-rw-----@ 1 rdevirgilio staff   1316  2 Nov 23:29 fork2.c  
-rw-----@ 1 rdevirgilio staff    441  2 Nov 23:15 prexit.c  
-rw-----@ 1 rdevirgilio staff    179  3 Nov 00:40 system1.c  
-rwxr-xr-x 1 rdevirgilio staff  8620  3 Nov 00:40 test  
-rw-----@ 1 rdevirgilio staff    787  2 Nov 22:42 vfork1.c  
-rw-----@ 1 rdevirgilio staff   1330  2 Nov 23:11 wait1.c
```

Controllo processi - Proprietà

- **real user, effettive user:**

- supponiamo di scrivere un programma per il gioco degli scacchi che tiene conto delle vittorie degli utenti ('player') e quindi scrive in un file di testo il punteggio dei giocatori.
- Il file dei punteggi non può essere scritto dagli user, altrimenti sarebbe troppo facile imbrogliare. Pertanto, questo file avrà come owner un user fittizio, diciamo 'scacchi' e sarà leggibile da tutti ma editabile solo e soltanto dall'user 'scacchi'.



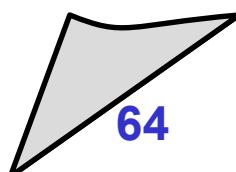
player



player



scacchi



Controllo processi - Proprietà

- **real user, effettive user:**

- Ora però si presenta il problema per il programma 'scacchi' di scrivere i punteggi. Poichè il programma, il cui owner è 'scacchi', viene eseguito dagli user 'player' è impossibile per il programma scrivere il punteggio di un 'player' quando il 'player' ha finito una partita poichè ogni programma acquisisce i privilegi di chi lo ha eseguito.
- La soluzione a questo problema è far acquisire al programma i diritti del suo owner 'scacchi' anche se eseguito da 'player'
- Perciò, 'player' è **real-user** mentre 'scacchi' è **effective-user**.

Controllo processi - Proprietà

■ privileged and unprivileged process:

- per verificare i permessi attribuiti ai processi, i tradizionali sistemi UNIX implementano i processi i due categorie: **privileged processes** (effective user ID = 0, quindi invocati da superuser o da root), e **unprivileged processes** (effective user ID diverso da 0).
- Linux tradizionalmente suddivide i privilegi in **capabilities** che possono essere abilitate o meno

■ some process-related capabilities:

- **CAP_IPC_LOCK**: Allow the process to lock memory.
- **CAP_SETUID**: Allow the process to make arbitrary manipulations of process UIDs and create forged UID when passing socket credentials via UNIX domain sockets
- **CAP_SETGID**: Same, but then for GIDs
- **CAP_KILL**: Bypass permission checks for sending signals
- **CAP_SYS_NICE**: This capability governs several permissions/abilities, namely to allow the process to change the nice value of itself and other processes

Controllo processi - Proprietà

- **System call:**

```
int setuid(uid_t uid);  
int setgid(gid_t gid);
```

- “sets the effective user (group) ID of the calling process”
- Esistono delle regole per cambiare questi id:
 - Se il processo ha privilegi da superutente (quindi abilitato in **CAP_SETUID**), la funzione **setuid** cambia real user ID con **uid**
 - Se il processo non ha privilegi da superutente e **uid** è un a real user id, la funzione **setuid** cambia **uid** in un effective user id
 - Se nessuna di queste condizioni è vera, **errno** è settato uguale a EPERM e viene ritornato un errore
- Per quanto riguarda group id, il discorso è identico

Controllo processi - Proprietà

- **int _POSIX_SAVED_IDS:**

```
int setuid(uid_t uid);  
int setgid(gid_t gid);
```

- "*Under Linux, setuid() is implemented like the POSIX version with the **_POSIX_SAVED_IDS** feature. This allows a set-user-ID (other than root) program to drop all of its user privileges, do some unprivileged work, and then reengage the original effective user ID in a secure manner*"

Controllo processi - Proprietà

■ Esempio di implementazione: the Linux id command

```
ID(1)                               User Commands                         ID(1)

NAME
    id - print real and effective user and group IDs

SYNOPSIS
    id [OPTION]... [USERNAME]

DESCRIPTION
    Print user and group information for the specified USERNAME, or (when USERNAME omitted) for the
    current user.

    -a      ignore, for compatibility with other versions

    -Z, --context
            print only the security context of the current user

    -g, --group
            print only the effective group ID

    -G, --groups
            print all group IDs

    -n, --name
            print a name instead of a number, for -ugG

    -r, --real
            print the real ID instead of the effective ID, with -ugG

    -u, --user
            print only the effective user ID

    --help display this help and exit

    --version
            output version information and exit
```

Controllo processi - Proprietà

- Esempio di implementazione: `set_ug.c`

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
int main(void) {
    int current_uid = getuid();
    printf("My UID is: %d. My GID is: %dn", current_uid, getgid());
    system("/usr/bin/id");
    if (setuid(0)) {
        perror("setuid");
        return 1;
    }
    //I am now root!
    printf("My UID is: %d. My GID is: %d\n", getuid(), getgid());
    system("/usr/bin/id");
    //Time to drop back to regular user privileges
    setuid(current_uid);
    printf("My UID is: %d. My GID is: %d\n", getuid(), getgid());
    system("/usr/bin/id");
    return 0;
}
```

Controllo processi - Proprietà

- Esempio di implementazione: set_ug.c

- Output:

```
$ gcc set_ug.c -o test
$ ls -al test
-rwxr-xr-x  1 rdevirgilio  staff  8664  3 Nov 09:33 test

$ ./test
uid=501(rdevirgilio) gid=20(staff) groups=20(staff),
12(everyone),61(localaccounts),79(_appserverusr),80(admin),
81(_appserveradm),98(_lpadmin),701(com.apple.sharepoint.group.
1),33(_appstore),100(_lpoperator),204(_developer),
395(com.apple.access_ftp),398(com.apple.access_screensharing),
399(com.apple.access_ssh)
```

setuid: Operation not permitted

Controllo processi - Proprietà

- Esempio di implementazione: `set_ug.c`

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
int main(void) {
    int current_uid = getuid();
    printf("My UID is: %d. My GID is: %dn", current_uid, getgid());
    system("/usr/bin/id");
    if (setuid(0)) {
        perror("setuid");
        return 1;
    }
    //I am now root!
    printf("My UID is: %d. My GID is: %d\n", getuid(), getgid());
    system("/usr/bin/id");
    //Time to drop back to regular user privileges
    setuid(current_uid);
    printf("My UID is: %d. My GID is: %d\n", getuid(), getgid());
    system("/usr/bin/id");
    return 0;
}
```

`setuid(0)` ha fallito: il processo non ha i permessi per accedere come root

Controllo processi - Proprietà

- Esempio di implementazione: `set_ug.c`

- Output:

```
$ su - root  
$ Password:  
$ chown root test  
$ chmod +s test  
$ ls -al test  
-rwsr-sr-x 1 root staff 8664 3 Nov 09:45 test  
  
$ ./test
```

s : set user or group ID on execution

Controllo processi - Proprietà

- Esempio di implementazione: `set_ug.c`

- Output:

```
$ ./test
uid=501(rdevirgilio) gid=20(staff) groups=20(staff),12(everyone),61(localaccounts),
79(_appserverusr),80(admin),81(_appserveradm),98(_lpadmin),
701(com.apple.sharepoint.group.1),33(_appstore),100(_lpoperator),204(_developer),
395(com.apple.access_ftp),398(com.apple.access_screensharing),
399(com.apple.access_ssh)
```

My UID is: 501. My GID is: 20

My UID is: 0. My GID is: 20

```
uid=0(root) gid=0(wheel) egid=20(staff) groups=0(wheel),1(daemon),2(kmem),3(sys),
4(tty),5(operator),8(procview),9(procmod),12(everyone),20(staff),29(certusers),
61(localaccounts),80(admin),33(_appstore),98(_lpadmin),100(_lpoperator),
204(_developer),395(com.apple.access_ftp),398(com.apple.access_screensharing),
399(com.apple.access_ssh),701(com.apple.sharepoint.group.1)
```

My UID is: 501. My GID is: 20

```
uid=501(rdevirgilio) gid=20(staff) groups=20(staff),12(everyone),61(localaccounts),
79(_appserverusr),80(admin),81(_appserveradm),98(_lpadmin),
701(com.apple.sharepoint.group.1),33(_appstore),100(_lpoperator),204(_developer),
395(com.apple.access_ftp),398(com.apple.access_screensharing),
399(com.apple.access_ssh)
```

Controllo processi - Proprietà

■ Nota

- Se l'utente è root bisogna fare attenzione: **setuid()** verifica l'effective user ID del chiamante e se questo è un superuser, tutti i processi legati a quel user ID sono configurati con l'iid passato alla funzione (quindi si perderanno alcuni privilegi).
- Dopo la chiamata di **setuid()**, sarà poi impossibile per il programma ripristinare i privilegi di root.
- Tale programma che ha consapevolmente perso i privilegi di root (assumendo l'identità di un utente "non privilegiato" non potrà riutilizzare **setuid()** per ripristinare i privilegi di root.
- Per evitare ciò si utilizzano le funzioni

```
int seteuid(uid_t euid);
```

```
int setreuid(uid_t ruid, uid_t euid);
```

Controllo processi - Proprietà

- Esempio di implementazione: game.c

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

/* Remember the effective and real UIDs. */

static uid_t euid, ruid;

/* Restore the effective UID to its original value. */

void do_setuid (void) {
    int status;

#ifdef _POSIX_SAVED_IDS
    status = seteuid (euid);
#else
    status = setreuid (ruid, euid);
#endif
    if (status < 0) {
        fprintf (stderr, "Couldn't set uid.\n");
        exit (status);
    }
}
```

Controllo processi - Proprietà

- Esempio di implementazione: game.c

```
/* Set the effective UID to the real UID. */

void undo_setuid (void){
    int status;

#ifndef _POSIX_SAVED_IDS
    status = seteuid (ruid);
#else
    status = setreuid (euid, ruid);
#endif
    if (status < 0) {
        fprintf (stderr, "Couldn't set uid.\n");
        exit (status);
    }
}
```

Controllo processi - Proprietà

- Esempio di implementazione: game.c

```
/* Record the score. */

int record_score (int score){
    FILE *stream;
    char *myname;

    /* Open the scores file. */
    do_setuid ();
    stream = fopen ("./gioco.txt", "a");
    undo_setuid ();

    /* Write the score to the file. */
    if (stream)
    {
        myname = getlogin();
        if (score < 0)
            fprintf (stream, "%10s: Couldn't lift.\n", myname);
        else
            fprintf (stream, "%10s: %d feet.\n", myname, score);
        fclose (stream);
        return 0;
    }
    else
        return -1;
}
```

Controllo processi - Proprietà

- Esempio di implementazione: game.c

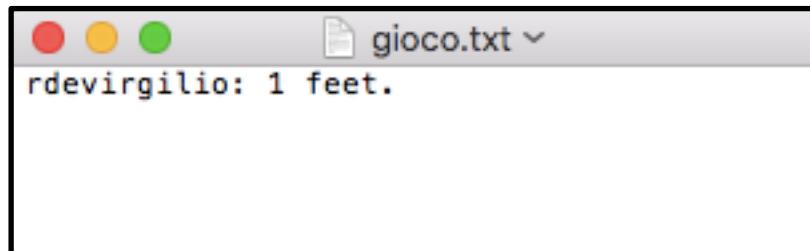
```
/* Main program. */

int main (void) {
    /* Remember the real and effective user IDs. */
    ruid = getuid ();
    euid = geteuid ();
    undo_setuid ();

    /* Do the game and record the score. */
    record_score(1);
}
```

- Output:

```
$ gcc game.c -o test
$ ./test
```



Controllo processi – Race condition

- **Ripasso:**

- una race condition avviene quando processi multipli cercano di accedere a dati condivisi e il risultato finale dipende dall'ordine di esecuzione dei processi

- **fork:**

- L'utilizzo di una funzione **fork** dà origine a problemi di race condition
 - In alcuni casi, il risultato di una esecuzione può dipendere:
 - da chi parte per primo fra parent e child
 - dal carico del sistema
 - dall'algoritmo di scheduling
 - Non è possibile fare previsioni

Controllo processi – Race condition

- **Esempio di race condition: fork2.c**
 - Se il sistema è carico, il secondo figlio può uscire da `sleep` prima che il primo figlio riesca a fare `exit`
- **Risoluzioni del problema?**
 1. fare `waitpid` sul process id del processo padre?
NO! Si può fare `wait` solo sui processi figli
 2. utilizzare un meccanismo di polling:

```
while (getppid() != 1)
    sleep(1)
```

NO! Funziona, ma spreca le risorse del sistema

- utilizzare un meccanismo di sincronizzazione più potente
(segnali, pipe, etc)

Controllo processi – Race condition

■ Esempio: tellwait1.c

- Un programma che stampa due stringhe, ad opera di due processi diversi
- L'output dipende dall'ordine in cui i processi vengono eseguiti

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

int main(void) {
    pid_t pid;

    if ((pid = fork()) < 0) {
        printf("Cannot fork! !\n");
        exit(1);
    }
    else
        if (pid == 0) printf("output from child\n");
        else   printf("output from parent\n");

    exit(0);
}
```

Controllo processi – Race condition

- **Esempio: tellwait1.c**

- Un programma che stampa due stringhe, ad opera di due processi diversi
- L'output dipende dall'ordine in cui i processi vengono eseguiti

- **Esempi di output:**

```
$ gcc tellwait1.c -o test
```

```
$ ./test  
output from child  
output from parent
```

```
$ ./test  
ooouuttpuutt ffrroomm cphairlednt
```

```
$ ./test  
outoutput from child  
put from parent
```



```
Current conditions at Pescara, Italy (IBP) 42.26N 014.12E 11M (IBP)
Last updated Feb 10, 2012 - 02:50 PM EST / 2012-02-10 1950 UTC
Temperature: 1 C
Relative Humidity: 80%
Wind: from the W (270 degrees) at 15 MPH (13 KT) gusting to 45 KPH
Weather: light snow grains
Sky conditions: overcast
Su Mo Tu We Th Fr Sa      Su Mo Tu We Th Fr Sa
feb 29e 30 31 01 02 03 04   mar 04 05 06 07 08 09 10
  05 06 07 08 09 10e 11    11e 12 13 14 15 16 17
  12 13 14 15 16 17 18    18 19 20 21 22 23 24
  19 20 21e 22e 23 24 25   25 26 27 28 29 30 31
mar 26 27 28 29 01 02 03   apr 01e 02 03 04 05 06e 07

silvio@Stain ~ $ cd Video
silvio@Stain ~ /Video $ movgrab http://vimeo.com/27998081

Formats available for this Movie: flv
Selected format: flv
Progress: 61.47% 15.4M of 25.1M 693.6K/s
```

