Breve storia di UNIX

1969: Ken Thompson (dell'AT&T) sviluppa in Assembler la prima versione di UNIX, come alternativa al sistema MULTICS..

1973: Thompson e Ritchie riscrivono buona parte del sistema in C

1975: La University of California at Berkeley incomincia a sviluppare una versione alternativa di UNIX (BSD)

1986: Esce la versione 4.3BSD

1989: Esce la versione System V, Release 3

Metà ottanta: standard IEEE, POSIX

Fine ottanta: interesse commerciale in UNIX: OSF/1 (DEC), Xenix (Microsoft), SunOS, Solaris, AIX (IBM)

Caratteristiche di UNIX

UNIX possiede tutti gli elementi che caratterizzano i sistemi operativi moderni.

Multi-utenza: diversi utenti possono accedere contemporaneamente al sistema

Multi-programmazione: più processi possono essere attivi contemporaneamente

Portabilità: Più del 90% di UNIX è scritto in linguaggio C

Login e logout in UNIX

Quando si riceve l'autorizzazione ad accedere al sistema da parte dell'amministratore del sistema, si ottiene un nome simbolico (nome di login) ed un unico numero identificatore chiamato user-identificator (UID).

Queste informazioni sono inserite nel file `/etc/passwd` in cui sono inseriti anche i caratteri (criptati) della password.

Per accedere a UNIX è necessario farsi riconoscere mediante il proprio nome di login e la propria password. Il programma di login si preoccuperà di criptare le password digitata e confrontarla con quella contenuta nel file `/etc/passwd`

Esempio:

```
login: bianchi
Password: gJkk7Y78     /* non visibile */
$ _
```

Inizialmente la password è fornita dal system administrator.

Per cambiare la propria password UNIX mette a disposizione il comando `passwd`.

Esempio:

```
$ passwd
Changing password for bianchi.
Old password: gJkk7Y78            /* la password
New password: eQh6yUK9               non viene
Retype new password: eQh6yUK9        visualizzata */
Password changed
$ _
```

Per uscire dal sistema si possono usare i comandi `logout`, `exit`, o `^D`.

Lo Shell di UNIX

Dopo aver verificato la correttezza del login-name e della password, il programma `login` lancia l'interprete dei comandi (**shell**) specificato nel file `/etc/passwd` ed esce.

Lo shell (dopo una prima fase di inizializzazione) stampa un carattere di prompt (per esempio, $ o %) sullo schermo ed aspetta che l'utente scriva una linea di comando (terminata da un ritorno a capo) del tipo:

```
command arg_1 arg_2 .... arg_n
```

dove gli argomenti possono anche essere opzioni di esecuzione (specificate dal carattere −).

Esempi:

```
cp src dest
```

il comando invoca il comando `cp` (copy) con due argomenti. Il file `src` verrà copiato in un file che si chiama `dest`.

```
head -20 file
```

In questo caso il S.O. stampa sul video i primi 20 caratteri del file, invece dei 10 di default.

Attenzione, il comando

```
head 20 file
```

è perfettamente legale. Dice di stampare le prime 10 righe del file chiamato `20` e poi le prime 10 righe del file chiamato `file`.

Il comando `man`

Il manuale UNIX è accessibile "on-line" mediante il comando `man`.

La sua sintassi è:

```
man [chapter] word
```

visualliza le informazioni relative al comando `word` (nel capitolo specificato da `chapter`, nel caso in cui sia presente). Se si omette chapter, viene visualizzata solo la prima entry.

Un altro modo di invocare il comando è

```
man -k keyword
```

che visualizza tutte le voci del manuale che contengono la parola chiave `keyword`.

Il comando `ls`

Il comando `ls` visualizza la lista dei file contenenti nella directory corrente.

Opzioni:

- `a`: visualizza i file "nascosti" (quelli che incominciano con ".")
- `l`: visualizza informazioni più dettagliate come flag di protezione, proprietario, tempo ultima modifica
- `s`: visualizza il numero di blocchi occupati dal file (un blocco è tipicamente tra 1K e 4K)
- `F`: pone un carettere alla fine del nome del file per indicarne il tipo:
  - `*`: file eseguibile
  - `/`: directory
- `R`: visualizza ricorsivamente il contenuto delle directory e sottodirectory

Esempi:

```
$ ls
pippo    pluto.c  paperino.h

$ ls *.c
pluto.c

$ ls -l *.c
-rw-r--r--   1 rossi users  106    Jan 30   19:46    pluto.c

$ _
```

I comandi `more/page/head/tail`

Consentono la visualizzazione del contenuto di un file.

`more`: Consente di visualizzare un file una pagina alla volta.
Dopo la visualizzazione di una pagina, `more` visualizza
il messaggio "`-- More --`"
Per visualizzare la pagina successiva premere "spazio"
Per visualizzare la linea successiva premere "invio"
Per uscire premere "q"
Per ottenere un help premere "h"

`page:` Funziona come `more`, con la differenza che cancella lo schermo prima di visualizzare le pagine.

`head:` Visualizza le prime n linee di un file. Se n non è specificato il valore di default è 10.

`tail:` Visualizza le ultime n linee di un file. Se n non è specificato il valore di default è 10.

Esempi:

```
$ more prova.c
#include <stdio.h>
main()
{
printf("Ciao\n");
} /* end of program */

$ head -2 prova.c
#include <stdio.h>
main()

$ tail -1 prova.c
} /* end of program */
```

I comandi `cp/mv/rm`

`cp`: copia un file

<u>Esempio:</u>

```
$ cp prova1.c prova2.c
$ ls -lF
-rw-r--r--   1 rossi users  106    Jan 15  19:46  prova1.c
-rw-r--r--   1 rossi users  106    Apr 16   9:40  prova2.c
drwxr-xr-x   1 rossi users  100    Mar 18  12:41  programmi/
$ cp prova1.c programmi
$ ls programmi
prova1.c
```

`mv`: rinomina un file

<u>Esempio:</u>

```
$ mv prova1.c prova2.c
$ ls -lF
-rw-r--r--   1 rossi users  106    Apr 16   9:40  prova2.c
drwxr-xr-x   1 rossi users  100    Mar 18  12:41  programmi/
$ mv prova2.c programmi
$ ls
drwxr-xr-x   1 rossi users  100    Mar 18  12:41  programmi/
$ ls programmi
prova2.c
```

`rm`: cancella un file.

<u>Esempio:</u>

```
$ rm prova1.c
$ ls -lF
drwxr-xr-x   1 rossi users  100    Mar 18  12:41  programmi/
```

Comandi che operano su directory

cd: Cambia la directory corrente

Esempi:

```
$ cd                 .... si sposta sulla home directory
$ pwd                .... visualizza la directory corrente
/usr/rossi
$ cd corso
$ pwd
/usr/rossi/corso
$ cd ..              .... si sposta sulla directory "madre"
$ pwd
/usr/rossi
```

mkdir: Crea una directory

Esempio:

```
$ mkdir programmi
$ ls -lF
-rwxr-x---   1  rossi  users  30   Mar 21 19:29 pluto.c
drwxr-xr-x   2  rossi  users  512  Feb 15 12:16 programmi/
$ ls -aF programmi
./   ../
```

rmdir: Cancella una directory
La directory deve essere vuota.

Il compilatore C

Per compilare (e "linkare") un programma C si usa il comando `cc`.

Esempi:

```
$ cc prova.c                ... compilo il programma prova.c
$ ls
prova.c  a.out              ... il risultato è "a.out"
$ a.out                     ... eseguo il programma

$ cc -o prova prova.c       ... il codice eseguibile è "prova"
$ ls
prova.c  prova

$ cc -c prova.c             ... esegue solo la compilazione
$ ls
prova.c prova.o             ... il risultato è "prova.o"

$ cc -o prova prova.o       ... genero l'eseguibile
$ ls
prova.c prova.o prova

$ cc -c modulo1.c modulo2.c modulo3.c   ... compilo 3 moduli
$ ls
modulo1.c    modulo2.c    modulo3.c
modulo1.o    modulo2.o    modulo3.o
$ cc modulo1.o modulo2.o modulo3.o -o main1
$ _
```

Altre opzioni utili:

- `g`: genera informazioni utili al debugger (symbol table, etc.)
- `P`: esegue soltanto la fase di pre-processing e pone il risultato in un file con suffisso "`.i`"
- `O`: produce codice ottimizzato
- `w`: sopprime messaggi di warning

Shell di UNIX

Lo "shell" è un programma che si pone tra l'utente e il nucleo del sistema operativo. E' tipicamente memorizzato sotto la directory `/bin`.

Ci sono tre shell tipicamente in uso:

- lo shell Bourne (`/bin/sh`)
- lo shell Korn (`/bin/ksh`)
- lo shell C (`/bin/csh`)

Esiste anche una variante dello shell C che si chiama TC (`/bin/tcsh`).

Dopo l'operazione di login, il sistema operativo lancia la shell il cuni nome è memorizzato (dal system administrator) nel file `/etc/passwd`.

Per invocare una nuova shell basta digitarne il nome (e.g., `sh, ksh, tcsh`).

Per determinare la shell attualmente in uso si usa la variabile di sistema chiamata SHELL

Esempio:

```
$ echo $SHELL
/bin/tcsh
$ _
```

Per uscire dalla shell corrente si digita il comando `exit` o `logout` (o `^D`).

Quando una shell è invocata esegue le seguenti operazioni:

1) legge ed interpreta uno speciale file di startup
2) visualizza un prompt (`$`, `%`, etc.) ed attende i comandi dell'utente
3) se il comando è `exit` o `logout` (o `^D`) termina, altrimenti esegue il comando e torna al passo 2).

Redirezione dell'Input/Output

Molti comandi UNIX leggono dal file di input standard e scrivono sul file di output standard per default. Per esempio: il comando

```
sort
```

legge linee dal terminale, ordina le linee lessicograficamente, e scrive il risultato sullo schermo.

E' possibile redirezionare lo standard input e lo standard output mediante i caratteri "<", ">".

Esempi:

```
sort <in >out
```

fa sì che il comando `sort` prenda il suo input dal file in e scriva l'output sul file out.

Il comando

```
mail rossi@unive.it < lettera.txt
```

invia il file `lettera.txt` all'utente `rossi@unive.it`.

La sequenza

```
command >> fileName
```

appende l'output di `command` al file di nome `fileName`.

Pipeline

Si consideri il comando seguente

```
sort <in >temp; head -30 <temp; rm temp
```

Prima il comando `sort` ordina il file `in` e scrive il risultato su `temp`, poi si visualizzano le prime 30 righe di `temp`, e quindi si cancella il file `temp`.

Lo stesso effetto si ottiene con la sequenza:

```
sort <in | head -30
```

La barra verticale, chiamata simbolo di **pipe**, dice di prendere l'output dal comando `sort` e usarlo come input per il programma `head`.

Un insieme di comandi connessi dal simbolo di pipe è detto **pipeline**.

Esempio:

```
grep ter *.t | sort | head -20 | tail -5 >foo
```

Tutte le linee contenenti la stringa "`ter`" nei file con estensione `.t` sono visualizzate sullo schermo, dove sono ordinate dal comando `sort`. Successivamente, le prime 20 di queste sono selezionate da `head`, che le passa al comando `tail`, il quale scriverà le ultime 5 sul file `foo`.

Sostituzione di comandi

Un comando racchiuso da accenti gravi (`) è rimpiazzato dal suo standard output.

Esempi:

```
$ echo la data di oggi è `date`
la data di oggi è Mon Feb 3 00:41:55 CST 1992

$ who             ... visualizza gli utenti
bianchi  ttyp0    Jan 22   15:31
rossi    ttyp3    Jan 21   10:00
verdi    ttyp9    Jan 22   11:00

$ echo ci sono `who | wc -l` utenti connessi sul sistema
ci sono 3 utenti connessi sul sistema
```

Sequenze

E' possibile eseguire più di un comando su un'unica linea di comando semplicemente usando il simbolo ";".

Esempio:

```
$ date; pwd; ls
Mon Feb 3  00:11:10  CST 1992
/usr/rossi/programmi
prova1.c   prova2.c   prova3.c
$ _
```

Ciascun comando in una sequenza può essere rediretto individualmente:

Esempio:

```
$ date > date.txt; ls; pwd > pwd.txt
prova1.c   prova2.c   prova3.c   date.txt
$ more date.txt
Mon Feb 3  00:11:10  CST 1992
$ more pwd.txt
/usr/rossi/programmi
```

Sequenze condizionali

Ogni processo UNIX termina con un valore di ritorno. Per convenzione un valore di ritorno 0 indica che il processo è terminato correttamente, and un valore non-nullo indica fallimento.

I metacaratteri `&&` e `||` consentono di eseguire dei comandi sulla base del codice di terminazione.

- `&&`: il comando successivo è eseguito soltanto se il precedente è terminato correttamente (codice di ritorno 0)
- `||`: il comando successivo è eseguito soltanto se il precedente è terminato con fallimento (codice di ritorno ≠0)

Esempi:

```
$ cc myprog.c && a.out     ... esegue a.out solo se cc OK

$ cc myprog.c || echo Compilazione fallita
                           ... se cc non OK stampa messaggio
```

Gruppi di comandi

Due o più comandi possono essere raggruppati insieme mediante le parentesi tonde. Questo provoca l'attivazione di uno shell "figlio" (subshell).

I gruppi di comandi condividono lo stesso standard input e output e possono essere rediretti o messi in pipeline come se fossero un unico comando.

Esempi:

```
$ date; ls; pwd > out.txt      ...esegue una sequenza
Mon Feb 3  00:11:10  CST 1992  ...output da "date"
prova1.c   prova2.c   prova3.c ...output da "ls"
$ more out.txt                 ...only "pwd" è rediretto
/usr/rossi/programmi


$ (date; ls; pwd) > out.txt    ...raggruppa e redireziona
$ more out.txt                 ...tutto l'output è rediretto
Mon Feb 3  00:11:10  CST 1992
prova1.c   prova2.c   prova3.c
/usr/rossi/programmi
$ _
```

Esecuzioni in Background

UNIX, oltre ad essere un sistema time-sharing, è anche un sistema in multiprogrammazione. Ciascun utente può far girare molti programmi contemporaneamente, ognuno come processo separato.

Per fare questo (da shell) è sufficiente far seguire al comando il simbolo &.

Esempio:

Il comando

```
wc -l <a >b &
```

conta il numero di linee (opzione `-l`) nel file di input `a`, scrive il risultato sul file `b`, e fa tutto questo in background. Questo significa che non appena il comando è stato lanciato, lo shell è pronto ad accettare nuovi comandi mentre il processo `wc` sta girando.

Anche le pipeline possono essere lanciate in background.

Esempio:

Il comando:

```
sort <x | head &
```

Pipeline multiple possono essere lanciate in background simultaneamente.

Script di Shell

Lo shell di UNIX può anche essere visto come linguaggio di programmazione. E' possibile infatti creare dei programmi costituiti da comandi di shell (**script**).

Gli script possono anche avere parametri

Ci sono anche costrutti di controllo come l'if, il while, il for, e il case.

Quando si lancia uno script il nucleo determina lo shell appropriato esaminando la prima riga dello script, secondo le regole seguenti:

1. se la prima riga è semplicemente un "#" allora lo script viene interpretato dallo shell C
2. se la prima riga è del tipo `#!pathname`, allora il nucleo esegue il comando specificato nel pathname (es: `#!/bin/ksh`)
3. altrimenti lo script viene interpretato dallo shell Bourne

Il carattere # viene interpretato come simbolo di inizio commento, in tutto il resto del documento.

Esempio:

```
$ cat > script.ksh
#!/bin/ksh
# Questo è un esempio di script Korn
echo -n The date today is
date          # stampa la data
^D

$ chmod +x script.ksh          ...rendo il file eseguibile
$ script.csh
The date today is Sat Feb  1  10:30:00  CST  1992
$ _
```

Sottoshell

Quando si e entra all'interno di una login l sistema determina la shell iniziale (dal file `/etc/passwd`). Ciascun comando digitato viene eseguito da questa shell.

Ci sono comunque dei casi in cui la shell iniziale crea una nuova shell figlia (sottoshell).

Questi sono i casi più comuni:

1. quando si esegue un gruppo di comandi, come `(ls; pwd; date)`, la shell "madre" crea una sottoshell che ha il compito di eseguire i comandi. Nel caso in cui il comando non sia eseguito in background la shell madre va in stato di "sleep" finche' la sottoshell non termina.
2. quando si esegue uno script la shell madre crea una sottoshell per eseguirne i comandi.
3. quando si esegue un comando in background viene creata una sottoshell. La shell madre continua a girare in concorrenza con la sottoshell.

La sottoshell, come ogni altro processo UNIX, ha la sua directory corrente, e quindi il comando `cd` dato all'interno di una sottoshell non influenza la directory corrente della shell madre.

Esempio:

```
$ pwd
/usr/rossi
$ (cd /; pwd)
/
$ pwd
/usr/rossi
```

Variabili (1)

Lo shell supporta due tipi di variabili: locali e di ambiente. Ogni shell contiene due aree preposte a contenerle.

Una sottoshell eredita tutte le variabili di ambiente della shell madre ma non quelle locali.

La sintassi per assignare valori a variabili dipende dal tipo di shell. In generale, comunque, il modo di accesso è lo stesso.

Non è necessario dichiarare le variabili prima dell'uso.

Se `var` è una variabile `$var` rappresenta il contenuto di quella variabile

Ci sono alcune variabili predefinite, come per esempio:

`$HOME`	il percorso completo della home directory
`$PATH`	una lista di directory per eseguire comandi
`$MAIL`	il percorso della mailbox
`$USER`	l'identificatore di utente
`$SHELL`	il percorso completo della shell di login
`$TERM`	il tipo di terminale

Esempi:

```
$ echo HOME = $HOME, PATH = $PATH
HOME = /usr/rossi, PATH = :/usr/bin:/bin
$ echo MAIL=$MAIL
MAIL=/usr/spool/mail/rossi
```

Variabili (2)

Per creare una variabile è sufficiente assegnarle un valore.

Negli shell Bourne e Korn l'assegnazione avviene nel seguente modo:

nome=valore

Esempio:

```
$ var1=rossi
$ var2=verdi
$ echo $var1, $var2
rossi, verdi
$
```

Il comando `export` consente di trasformare una variabile local in una di ambiente.

Esempio:

```
$ nome=marco
$ cognome=rossi
$ echo $nome $cognome
Marco Rossi
$ export cognome
$ sh
$ echo $nome $cognome
Rossi
$ logout
$ echo $nome $cognome
Marco Rossi
$ _
```

Variabili (3)

Uno script può essere chiamato con degli argomenti in ingresso.

Ci sono alcune variabili "built-in" che hanno un significato particolare:

`$$`	l'identificatore di processo dello shell
`$0`	il nome dello script
`$1..$9`	$n si riferisce all'n.simo argomento (se esiste)
`$*`	la lista degli argomenti

Esempio:

```
$ more script.sh
echo il nome dello script e' $0
echo il 1o argomento e' $1
echo una lista degli argomenti e' $*
echo questo script crea un file chiamato $1.$$
date > $1.$$     # reindirizza la data
ls $1.$$         # visualizza la lista di file
$ script.sh marco paolo matteo
il nome dello script e' script.sh
il primo argomento e' marco
una lista degli argomenti e' marco paolo matteo
questo script crea un file chiamato marco.22451
$ marco.22451
$ _
```

Valori di terminazione e di uscita

Ogni processo UNIX termina con un valore di uscita. Per convenzione un valore uguale a 0 indica che il processo è terminato con successo, e un valore diverso da 0 indica fallimento.

Negli shell Bourne e Korn la variabile `$?` contiene il valore di uscita del precedente comando. Nello shell C si usa invece la variabil `$status`.

Esempio:

```
$ date
Mon Feb  3  10:47:25  CST  1992
$ echo $?
0
$ cc prog.c
Syntax Error!!!!
$echo $?
1
$ _
```

Per far sì che uno script ritorni un valore di uscita si usa il comando `exit`.

Esempio:

```
$ more script.sh
echo questo script ritorna un valore pari a 3
exit 3
$ script.sh
questo script ritorna un valore pari a 3
$ echo $?
3
$ _
```

File in UNIX

Un file UNIX è semplicemente una sequenza di byte (non è fatta alcuna distinzione tra file ASCII o binari).

Ad ogni file viene associato un nome simbolico tipicamente seguito da una estensione che serve ad identificarne il tipo (esempio, prog.c indica che prog è un programma C, prog.p che invece è un programma Pascal, etc.)

A ciascun file UNIX associa un stringa di 9 bit che contiene informazioni relative alla protezione del file. I primi 3 si riferiscono ai diritti di accesso del proprietario, i secondi 3 ai diritti dei compenenti il gruppo del proprietario, gli ultimi 3 si riferiscono ai diritti di tutti gli altri.

Esempi:

```
rwxr-----
```

vuold dire che il proprietario può solo leggere (`r`), scrivere (`w`) ed eseguire (`x`) il file, mentre gli appartenenti al suo gruppo possono soltanto leggerlo.

La sequenza

```
rwxrwxrwx
```

significa che ogni utente può accedere al file in lettura/scrittura/esecuzione.

Directory in UNIX

I file possono essere raggruppati in directory (o cataloghi, o direttori, etc.)....

In UNIX ci sono tre tipi di file:

I file ordinari: sono destinati a contenere informazioni generiche ad uso dell'utente

Le directory: contengono elenchi di nomi di file, e possono a loro volta contenere subdirectory

I file speciali: sono associati a dispositivi fisici (dischi, nastri, stampante, etc)

Il file system di UNIX è organizzato in maniera gerarchica. Le directory possono cioè essere messe in relazione tra loro secondo una struttura ad albero.

All'interno di una directory, insieme ai file ordinari, possono esistere altre sotto-directory, che possono avere al loro interno altre sotto-directory e così via.

L'intero file system ha origine da una directory predefinita chiamata **root** che costituisce la radice dell'albero.

I file ordinari e speciali costituiscono le foglie dell'albero, mentre i nodi intermendi rappresentano le directory.

Alcuni comandi per l'uso di directory:

`cd`	cambia la directory corrente
`mkdir`	crea una nuova directory
`rmdir`	cancella una directory
`pwd`	visualizza la directory corrente

Processi in UNIX

UNIX è orientato ad uno schema di programmazione concorrente basato sui concetti di **immagine** e di **processo**.

Un'immagine rappresenta lo stato corrente di esecuzione di una macchina virtuale e contiene un'immagine della memora, un insieme di registri virtuali, lo stato dei file aperti, etc.

Un processo è l'esecuzione di un'immagine, ossia una sequenza di transizioni di stato di una macchina viruale.

La parte dell'immagine di un processo corrispondente alla memoria è composta da un **testo** (il codice) protetto in scrittura e da un'**area dati** su cui sono consentiti accessi sia in lettura che in scrittura.

Da un punto di vista logico più processi possono evolvere contemporaneamente, e sono detti **attivi**. Su macchine dotate di un solo processore, comunque, un solo processo tra quelli attivi è in esecuzione in un certo istante.

Quando un processo è in esecuzione la sua immagine deve essere presente in memoria centrale.

Creazione di processi

La creazione di un processo avviene sempre e solo per duplicazione di un processo già esistente mediante la chiamata di sistema **`fork`**.
Fa eccezione il processo numero 0 (`init`), creato dal S.O. al momento dell'inizializzazione del sistema.

L'effetto di

```
pid = fork()
```

è quello di creare una copia esatta dell'immagine associata al processo in quell'istante e generare un nuovo processo come esecuzione di quell'immagine.

I due processi risultanti sono tra di loro in relazione gerarchica: il processo che ha eseguito la `fork` è il processo **padre** rispetto a quello che è stato creato che prende il nome di **figlio**.

I due processi (padre e figlio) si distinguono in base al loro identificatore di processo (pid).

Il figlio non condivide memoria con il padre (ne condivide però il codice) in quanto dal momento della sua creazione in poi le due immagini evolvono parallelamente nei rispettivi processi.

Poiché la creazione avviene per duplicazione, il processo figlio eredita dal padre l'ambiente di lavoro, cioè i file aperti, i privilegi, la directory di lavoro, etc.

Per permettere ad ogni processo di riconoscersi come padre o figlio la funzione `fork` restituisce:

al figlio il valore `pid=0`
al padre l'identificatore del figlio (`pid>0`)

Se l'operazione fallisce il processo padre riceve un codice di errore (<0).

Esempio:

```
pid = fork();
if (pid < 0) {
     /* la fork ha fallito .... */
}  else if (pid > 0) {
     /* il codice del padre va qui */
}  else {
     /* il codice del figlio va qui */
}
```

Ogni processo può conoscere il proprio pid invocando la funzione `getpid` (la funzione `getppid` restituisce invece il pid del processo padre).

La famiglia di funzioni **exec**

La duplicazione di un processo non è un meccanismo sufficiente per permettere l'esecuzione di programmi, perché il processo figlio esegue la stessa immagine del padre.

L'associazione di una diversa immagine di un processo è permessa dalla famiglia di funzioni `exec`:

```
n = execl(nome, arg1, arg2,..., argN, 0)
n = execv(nome, argv)
```

Queste funzioni determinano l'esecuzione, da parte del processo che le chiama, di un nuovo programma, il cui nome è contenuto in `nome`; `arg1, ... argN` sono i parametri con cui si chiama il programma
Per convenzione `arg1` contiene il nome del programma, e argN è seguiti da `0`.

L'unica differenza tra le diverse funzioni è il modo in cui i parametri sono passati. La funzione `execv` differisce dalla execl in quanto gli argomenti sono raccolti in una lista, accessibile tramite il puntatore `argv`.

Il nuovo programma si sostituisce in memoria all'immagine del processo che ha eseguito la `exec`. Più precisamente, il nuovo programma sostituisce definitivamente la parte codice e la parte dati dell'immagine precedente, mentre restano inalterati i file aperti, la directory corrente e le relazioni che il processo aveva con altri processi.

Di conseguenza, dopo aver eseguito la funzione `exec` il codice originale del processo non esiste più. L'unico caso in cui il codice originale può continuare la sua evoluzione è la mancata evoluzione della funzione a causa di errori.

Esempio:

```
pid = fork();                  /* crea un processo figlio */
if( pid == 0)                  /* se è il figlio */
     { exec("program",...);    /* esegue "program" */
       error(...);             /* ...a meno di errori */
     }
```

Terminazione di processi

Tramite la primitiva

```
pid = wait(&status)
```

un processo si sospende in attesa della terminazione di uno qualsiasi dei suoi figli; quando ciò avviene il processo padre viene risvegliato.

Il parametro di ritorno `pid` riporta l'identificatore del processo figlio terminato. Altre informazioni sulla terminazione del processo vengono riportate in `status`.

Un processo può terminare in modo sincrono chiamando la funzione `exit`. L'effetto di

```
exit(status)
```

è quello di terminare il processo che la esegue, distruggere la sua immagine, chiudere i file aperti e segnalare la terminazione al padre nel caso che esso sia sospeso su una `wait`.

Tramite il parametro status il figlio può trasmettere al padre informazioni sulla propria terminazione.

Le funzioni `fork`, `exec`, e `wait` permettono di realizzare un meccanismo analogo alla chiamata di sottoprogramma dei lingiaggi di programmazione

Esempio:

```
...
pid = fork();              /* crea un nuovo processo */
if(pid == 0)               /* se sei il figlio */
    exec(...);             /* cambia immagine */
proc_id=wait(&status);     /* il padre aspetta la
                              terminazione del figlio */
if(status == ...)          /* e ne analizza lo stato */
...
```

Comunicazione fra processi

I processi possono comunicare attraverso dei canali chiamati **pipe** (condotti). Una pipe corrisponde ad un file accessibile ad un estremo in lettura e all'estremo opposto in scrittura. I dati immessi nel canale con successive scritture risultano accodati nell'attesa che un altro processo le legga; la coda è gestita con criterio FIFO.

Le pipe forniscono anche una sincronizzazione tra processi. Quando un processo tenta di scrivere, può essere sospeso se la pipe è piena. Analogamente, un processo che tenti di scrivere può essere sospeso se la pipe è vuota.

Per creare una pipe un processo invoca la funzione **pipe**:

```
pipe(fd)
```

in cui `fd` è un vettore di due elementi interi: all'elemento `fd[0]` viene assegnato il descrittore del file aperto in lettura corrispondente al lato di uscita della pipe, mentre a `fd[1]` viene assegnato il descrittore del file aperto in scrittura corrispondente al lato di ingresso.

La pipe è subordinata alla organizzazione gerarchica dei processi: l'unico modo che hanno due processi per comunicare è di avere un antenato comune che abbia predisposto una pipe a questo fine

Esempio:

```
int files[2];
...
pipe(files);
if(fork() == 0)              /* crea il produttore */
     exec("producer",...);
if(fork == 0)                /* crea il consumatore */
     exec("cosumer",...);
i = wait(&status);           /* attende la terminazione
i = wait(&status);              di entrambi */
```

Interruzioni e segnali

UNIX permette di gestire gli eventi asincroni (le interruzioni) provenienti dal mondo esterno trasformandoli in **segnali** che vengono comunicati ai processi interressati.

Il meccanismo dei segnali gestisce in modo analogo una differente classe di eventi (chiamati trappole) che pur provenendo dall'interno del sistema sono gestiti alla stregua di eventi asincroni.

Sono definiti nel sistema alcuni segnali ognuno dei quali rappresenta un particolare evento

Esempi:

Nome	Numero	Core	Commento
SIGINT	2	no	interrupt da terminale
SIGFPE	8	sì	errore floating point
SIGBUS	10	sì	errore sul bus
SIGPIPE	13	no	scrittura su una pipe che nessuno può leggere

Il comportamento standard di un processo quando riceve un segnale comporta la terminazione del processo stesso. Per alcuni segnali la terminazione è preceduta dal salvataggio dell'immagine del processo in un file chiamato **core**.

La funzione signal

```
signal(sig_num, program)
```

permette di modificare il comportamento del processo a fronte di un particolare segnale (`sig_num`). Ad ogni segnale è associata una routine di servizio che viene attivata all'arrivo del segnale. Tale routine viene sostituita dal programma indicato nel parametro `program` permettendo all'utente di ridefinire il comportamento del sistema.

Per trasmettere un segnale ad un processo esiste la funzione **kill**:

```
kill(proc_id, sig_num)
```

che trasmette al processo il cui identificatore è `proc_id` il segnale `sig_num`.

Inoltre, tramite la funzione **pause**:

```
pause()
```

un processo può sospendersi in attesa di un qualsiasi segnale ad esso indirizzato. Quando il processo verrà risvegliato riprenderà l'esecuzione con la routine associata al segnale che ne ha causato il risveglio.

La primitiva

```
alarm(nsec)
```

trasmette un segnale SIGALRM dopo "nsecs" secondi al processo che l'ha lanciata.