Gestione dei File in UNIX

(Parte II)

Le funzioni stat, fstat, lstat

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

int stat (const char *pathname, struct stat *buffstat);
int fstat (int filedes, struct stat *buffstat);
int lstat (const char *pathname, struct stat *buffstat);
```

- Consentono di indagare sullo stato di un file, individuato tramite ***pathname*** (stat, lstat) oppure ***filedes*** (fstat);
- Restituiscono 0 in caso di successo, -1 altrimenti;
- ***pathname*** è il pathname (assoluto o relativo) del file, ***filedes*** il suo descrittore;
- ***buffstat*** è l'indirizzo di una struttura dati che serve a contenere le informazioni di stato restituite al termine della chiamata.

stat, fstat, lstat (cont.)

La definizione della struttura dati stat si trova nell'header <sys/stat.h> e può dipendere dall'implementazione:

```
struct stat {
   mode_t   st_mode;   /* modo del file (16 bit)  */
   ino_t    st_ino;    /* numero di inodo */
   dev_t    st_dev;    /* numero del dispositivo (filesystem) */
   nlinks_t st_nlink;  /* numero di link al file */
   uid_t    st_uid;    /* user ID del proprietario */
   gid_t    st_gid;    /* group ID del proprietario */
   off_t    st_size    /* dimensione in byte del file */
   time_t   st_atime   /* ultimo accesso ai dati */
   time_t   st_mtime   /* ultima modifica ai dati */
   time_t   st_ctime   /* ultima modifica all'inodo */
   ...
}
```

La maschera di modo

La maschera di modo per la creazione dei file è una stringa di bit che permette di definire i permessi di default per i file successivamente creati.

La maschera di modo può essere definita mediante un numero ottale di tre cifre: la prima cifra definisce i permessi per il proprietario del file, la seconda per il gruppo e la terza per il resto degli utenti.

Il valore (in ottale) ***perm_val*** del permesso per il file si ottiene dal relativo valore ottale della maschera ***mask_val*** in base al calcolo seguente:

$$perm\_val = \begin{cases} 7 - mask\_val & \text{per le directory} \\ 6 - mask\_val & \text{per gli altri tipi di file} \end{cases}$$

La maschera di modo (cont.)

User permissions	*Group permissions*	*Other permissions*	
r w x	r w x	r w x	*Tipo*
$u_1u_2u_3$	$g_1g_2g_3$	$o_1o_2o_3$	*Bit*
$4u_1+2u_2+u_3$	$4g_1+2g_2+g_3$	$4o_1+2o_2+o_3$	*Valore ottale*

Esempio:

Se vogliamo che i file creati che non corrispondono a directory abbiano di default i permessi in lettura e scrittura per il proprietario ed in sola lettura il gruppo ed il resto degli utenti, allora:

stringa di permessi voluta di default per i file creati	rw - r- - r- -
stringa ottale corripondente	644
stringa ottale da assegnare alla maschera di modo	022

La maschera di modo (cont.)

La maschera di modo utente è una maschera di modo associata alla shell di default di un utente, mediante uno dei file di inizia-lizzazione della shell.

Per visualizzare (o ridefinire) il valore della maschera di modo associata alla shell in uso si può utilizzate il comando umask.

Il permessi effettivi di un file creato mediante le chiamate open o creat si ottengono effettuando l'AND bit a bit tra la stringa re-lativa ai permessi definiti mediante la maschera e quelli specificati attraverso l'argomento mode nella chiamata.

Un processo può cambiare il valore corrente della maschera di modo chiamando la funzione umask.

La funzione umask

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

mode_t  umask(mode_t  cmask);
```

- permette di definire la nuova maschera di modo ***cmask*** per il processo chiamante;
- l'argomento ***cmask*** si forma combinando le costanti simboliche relative ai permessi mediante l'operatore "|";
- restituisce il precedente valore della maschera.

Le funzioni chmod, fchmod

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

int  chmod (const char *pathname, mode_t  mode);
int  fchmod ( int filedes, mode_t  mode);
```

- permettono di cambiare il modo di accesso per un file esistente secondo quanto specificato da ***mode***;
- restituiscono 0 in caso di successo, -1 altrimenti;
- chmod opera sul file individuato da ***pathname***, fchmod su un file aperto con descrittore ***filedes***;
- l'argomento ***mode*** è specificato attraverso l'uso delle costanti simboliche di modo, congiunte con l'operatore "|".

Le funzioni chown, fchown, lchown

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

int  chown (const char *pathname, uid_t  owner, gid_t  group);
int  fchown ( int filedes, uid_t  owner, gid_t  group);
int  lchown (const char *pathname, uid_t  owner, gid_t  group);
```

- permettono di cambiare lo user ID ed il group ID per un file esistente, come specificato da ***owner*** e da ***group***;
- restituiscono 0 in caso di successo, -1 altrimenti;
- chmod e fchmod operano sul file individuato da ***pathname***, lchmod su un file aperto con descrittore ***filedes***;
- se il file è un link simbolico, lchown cambia la proprietà del link stesso, non del file cui si riferisce.

Le funzioni truncate, ftruncate

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

int  truncate (const char *pathname, off_t  length);
int  ftruncate ( int  filedes, off_t  length);
```

- permettono di modificare la dimensione di un file esistente al valore in byte fornito da ***length***;
- restituiscono 0 in caso di successo, -1 altrimenti;
- truncate opera sul file individuato da ***pathname***, ftruncate su un file aperto con descrittore ***filedes***;
- se il file ha una dimensione maggiore di ***length***, i dati dopo ***length*** non sono più accessibili;
- se il file ha una dimensione minore di ***length***, i dati compresi tra la vecchia e la nuova fine del file vengono letti come zeri.

Le funzioni link, unlink

```
#include <unistd.h>

int  link (const char *existingpath, const char  *newpath);
int  unlink( const char *existingpath);
```

- link permette di creare il nuovo nome di file (path assoluto o relativo) ***newpath*** per il file individuato da ***existingpath***;
- unlink cancella il nome di file ***existingpath***;
- restituiscono 0 in caso di successo, -1 altrimenti;
- Se ***newpath*** già esiste, link restituisce un errore;
- molte implementazioni richiedono che ***existingpath*** e ***newpath*** siano relativi allo stesso filesystem;
- se ***existingpath*** è un link simbolico, unlink ha effetto su di esso, non sul file referenziato.

Le funzioni symlink, readlink

```
#include <unistd.h>

int  symlink (const char *actualpath, const char  *sympath);
int  readlink( const char *pathname , char * buf, int  bufsize );
```

- symlink crea il nuovo link simbolico ***sympath*** che punta a ***actualpath***;
- symlink restituisce 0 in caso di successo, -1 altrimenti;
- I path ***actualpath*** e ***sympath*** possono essere relativi a differenti filesystem, e ***actualpath*** può non esistere prima della chiamata;

- readlink permette di leggere il contenuto del link individuato da ***pathname***, memorizzandolo all'indirizzo ***buf***;
- restituisce il numero di byte letti in caso di successo, -1 altrimenti;
- combina le azioni di open, read e close.

La funzione utime

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

int  utime( const char *pathname , const struct utimbuf *times);
```

- permette di cambiare i tempi di utimo accesso e/o di ultima modifica del file di nome ***pathname***;
- restituisce 0 in caso di successo, -1 altrimenti;
- ***times*** è l'indirizzo di una struttura dati in cui vengono memorizzate le informazioni necessarie alla chiamata;
- La struttura è definita nell'header <utime.h> come segue:

```
struct utimbuf {
  time_t   actime;     /* tempo di ultimo accesso */
  time_t   modtime   /* tempo di ultima modifica */
}
```

La funzione utime (cont.)

Il tipo di operazione effettuato da utime, ed i privilegi necessari per eseguirla, dipendono dal fatto che *times* sia un puntatore nullo o meno:

- se *times* =NULL, i tempi di ultimo accesso ed ultima modifica sono entrambi posti uguali al tempo corrente;

- se *times* !=NULL, i tempi di ultimo accesso ed ultima modifica sono posti uguali ai valori memorizzati nella struttura cui punta *times*. In tal caso è necessario che il processo chiamante abbia i privilegi di root, oppure che il suo UID effettivo sia uguale all' UID del proprietario del file.

Le funzioni mkdir, rmdir

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

int mkdir (const char *pathname, mode_t  mode);
```

```
#include <unistd.h>

int rmdir (const char *pathname);
```

- mkdir crea una nuova directory di nome ***pathname***, contenente solo i record relativi alla directory corrente (.) ed alla directory genitore (..);
- rmdir rimuove il link ***pathname*** relativo ad una directory vuota;
- entrambe restituiscono 0 in caso di successo, -1 altrimenti.

Lettura delle directory

Per una directory i permessi hanno un significato diverso che per gli altri tipi di file:

- un permesso in lettura indica che il listato dei file contenuti nella directory può essere visualizzato;
- un permesso in scrittura indica che un file può essere creato o rimosso dalla directory, a condizione che si possa accedere alla directory stessa (traversare);
- un permesso in esecuzione indica che si può accedere alla directory.

La presenza dei permessi wx per una directory consente ad un processo utente solo di creare/eliminare un file (link) mediante funzioni opportune, non di manipolare direttamente i dati relativi ad una directory. Questa operazione, per salvaguardare l'integrità del file system, è concessa solo al kernel.

Lettura delle directory (cont.)

Il formato di una directory dipende dall'implementazione, così per ottenere la portabilità per quei programmi che devono leggere directory, POSIX definisce il seguente insieme di routine:

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

DIR *opendir (const char *pathname);
struct dirent *readdir (DIR *dp);
void rewinddir (DIR *dp);
int closedir (DIR *dp);
```

- La struttura DIR è utilizzata da opendir, readdirr rewinddir e closedir per gestire le informazioni relative alla directory da leggere, individuata da *pathname*;
- La struttura dirent (directory entry) serve a memorizzare le informazioni contenute nei record della directory da leggere.

Lettura delle directory (cont.)

- La struttura dirent è definita nell'header <dirent.h> e dipende dall'implementazione, ma contiene almeno i membri relativi al nome del file ed al relativo inodo;
- opendir restituisce un puntatore alla struttura DIR (il puntatore nullo in caso di errore), utilizzato dalle rimanenti tre funzioni, ed effettua le operazioni necessarie affinchè readdir possa leggere il primo record della directory;
- readdir legge un record della directory, memorizzandone i campi nella struttura dirent. Restituisce il puntatore a dirent in caso di successo, il puntatore nullo in caso di EOF o di errore;
- rewinddir ridefinisce la posizione nella directory al suo inizio;
- closedir chiude la directory prima aperta con opendir. Restituisce 0 in caso di successo, -1 altrimenti

Le funzioni chdir e fchdir

```
#include <unistd.h>

int  chdir (const char *pathname);
int  fchdir( int  filedes);
```

- permettono di cambiare la directory corrente di lavoro (CWD) per il processo chiamante;
- chdir cambia la directory corrente di lavoro in ***pathname***;
- fchdir specifica la nuova directory di lavoro attraverso il descrittore di file ***filedes***;
- entrambe restituiscono 0 in caso di successo, -1 altrimenti;
- poichè la directory corrente di lavoro è un attributo di un processo, le chiamate chdir e fchdir non hanno alcun effetto sui processi genitore.

La funzione getcwd

Le informazioni relative alla directory di lavoro corrente di cui il kernel tiene traccia per ogni processo consistono nel nome-file e nell'inodo della CWD. Il kernel non conserva il path name completo della CWD.
Per permettere ad un processo di conoscere il pathname assoluto della propria CWD si può ricorrere alla funzione getcwd:

```
#include <unistd.h>

char *getcwd (char *buff  size_t  size);
```

- memorizza il pathname assoluto della directory di lavoro corrente per il processo nel buffer di indirizzo *buff* ;
- la dimensione *size* del buffer deve essere tale da poter accoglie re il pathname assoluto della CWD, terminato dal byte nullo;
- restituisce *buff* in caso di successo, il puntatore nullo altrimenti

Le funzioni sync e fsync

Molte implementazioni UNIX hanno un buffer di cache nel kernel per le operazioni I/O: quando si effettua una write per un file, i dati sono generalmente copiati dal kernel nella cache, in attesa di una successiva scrittura su disco.
Le funzioni sync ed fsync assicurano la consistenza del file system sul disco con i contenuti del buffer di cache:

```
#include <unistd.h>

void  *sync (void);
int  fsync (int  filedes);
```

- sync mette in coda tutti i buffer modificati per la scrittura, non attende per l'effettivo completamento dell'output e ritorna;
- fsync opera solo sul file individuato da *filedes*, attendendo per il completamento della sua scrittura su disco;
- entrambe restituiscono 0 in caso di successo, -1 altrimenti.

Riferimenti

- W.R Stevens – Advanced Programming in the UNIX Environment - Addison Wesley, 1992