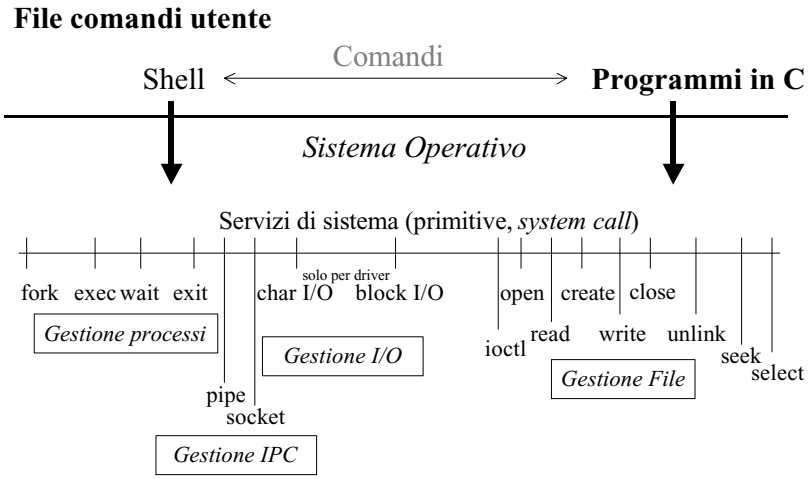
Programmazione di sistema in UNIX



Introduzione

Lo spazio di indirizzamento di un processo UNIX

Ogni processo opera in uno spazio di indirizzamento logico e privato

L'immagine di un processo è mappata in un address space ed è organizzata in due aree principali:

- area utente (codice, dati, stack);
- area di kernel (kernel stack, user area)

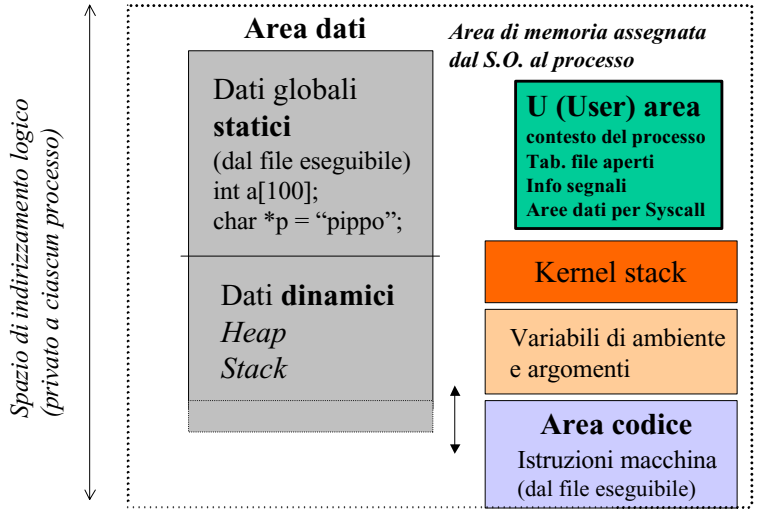
Entrambe possono essere trasferite sul dispositivo di swap (swappable)

Il kernel utilizza inoltre strutture dati residenti (non swappable) :

- process table
- text table (tabella dei codici correnti)

Introduzione

Immagine di un processo in UNIX



Il processo può riferirsi (generare indirizzi) solo a questa area

Introduzione

Argomenti di un programma

Un programma può accedere agli eventuali argomenti di invocazione attraverso i parametri della funzione principale **main**

```
/* mioprogramma.c */
main(int argc, char *argv[])
{
int i;
printf("Numero di argomenti (argc) = %d\n", argc);
for(i=0;i<argc;i++)
  printf("Argomento %d (argv[%d]) = %s\n",i,i,argv[i]);

printf("L'argomento di indice 0 è il nome del programma
eseguito\n");
}
```

Eseguendo il programma

```
$ mioprogramma 1 pippo pluto 4
```

viene visualizzato:

```
Numero di argomenti (argc) = 5
Argomento 0 (argv[0]) = mioprogramma
Argomento 1 (argv[1]) = 1
Argomento 2 (argv[2]) = pippo
Argomento 3 (argv[3]) = pluto
Argomento 4 (argv[4]) = 4
L'argomento di indice 0 è il nome del programma
eseguito
```


Variabili di ambiente

Sono accessibili attraverso :

- un terzo parametro char **envp della funzione principale **main** (o come variabile esterna `extern char **environ;`)
- mediante le funzioni di utilità getenv/putenv

```
/* mioprogramma.c */
main(int argc, char *argv[], char **envp)
{
int i;
printf("Numero di argomenti (argc) = %d\n", argc);
for(i=0;i<argc;i++)
  printf("Argomento %d (argv[%d]) = %s\n",i,i,argv[i]);

while (*envp != NULL)
      { printf("%s\n",*envp++);
}
```


Le primitive UNIX ritornano sempre un valore intero che esprime il successo (>=0) o il fallimento (-1 o comunque<0) della chiamata:

- la descrizione dell'errore viene resa disponibile nella variabile `errno` (non è necessario definirla)
- funzioni di utilità come `perror` o `strerror` permettono di visualizzare o di generare messaggi descrittivi dell'errore

```
if (syscall_N(…, …) < 0)
      {
      perror("Errore nella syscall_N");
      /* La descrizione dell'errore viene
      concatenata alla stringa argomento di
      perror */
      exit(-1);   /* terminazione del processo con
                  errore */
}
```


L'uso di file (e di dispositivi) in UNIX si basa su un protocollo di richiesta/rilascio della risorsa

- **Prologo** (richiesta della risorsa)
  - primitive `open` o `creat` (o altre per l'accesso a risorse che non siano file)
  - **Uso della risorsa**
    - primitive `read`, `write` o altre
- **Epilogo** (rilascio della risorsa)
  - primitiva `close`

Operazioni sui file

Apertura ed eventuale creazione di un file

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

int open(const char *pathname,int flags);
      oppure
fd=int open(const char *pathname,int flags,mode_t
mode);

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

- `pathname` è il nome/percorso del file da aprire o creare
- `flags` contiene il modo di accesso richiesto
  (uno tra `O_RDONLY, O_WRONLY, O_RDWR`) più altre eventuali opzioni in OR (ad es. O_WRONLY | O_CREAT |O_TRUNC per richiedere la creazione di un nuovo file o per azzerarlo se già esistente)
- `mode` indica i diritti di accesso del nuovo file (ad es. in ottale 0644)

Operazioni sui file

Se la invocazione di una primitiva `open` o `creat` ha successo, viene restituito al processo un valore intero >=0 che costituisce un file descriptor (fd) per quel file:

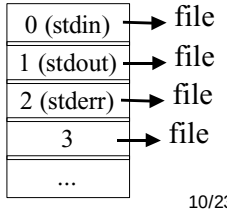
• il file descriptor va utilizzato per le successive operazioni su quel file da parte di quel processo (invece del nome/percorso, che viene utilizzato solo in `open/creat`)

• numeri successivi vengono associati ai nuovi file aperti (3, 4, ...)

Lo stesso comportamento vale anche per le primitive utilizzate per creare/accedere altri tipi di risorsa (`pipe, socket`)

I file descriptor sono indici per una tabella dei file aperti mantenuta per il processo nella sua User Area

i primi tre file descriptor sono predefiniti (non è necessario crearli ma vanno modificati per ottenere la redirezione)



Operazioni sui file

Duplicazione di un file descriptor

```
#include <unistd.h>

int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

Utilizzati per la redirezione e il piping

Il valore di ritorno di `dup` o il `newfd` di `dup2` possono essere utilizzati indifferentemente al posto del file descriptor originale

Chiusura di un file descriptor

```
#include <unistd.h>

int close(int fd);
```

La chiusura di un file descriptor consente di riutilizzarlo per un nuovo file. Se non vi sono altri file descriptor per quell'oggetto, le risorse associate (ad. es. l'I/O pointer) vengono effettivamente liberate.

Operazioni sui file

Lettura e scrittura da un file descriptor (file o altre risorse)

```
#include <unistd.h>

int read(int fd, void *buf, size_t count);
int write(int fd, void *buf, size_t count);
```

• **`read`** prova a leggere dall'oggetto a cui si riferisce `fd` fino a `count` byte, memorizzandoli a partire dalla locazione `buf`.

• **`write`** prova a scrivere sull'oggetto a cui si riferisce `fd` fino a `count` byte, letti a partire dalla locazione `buf`

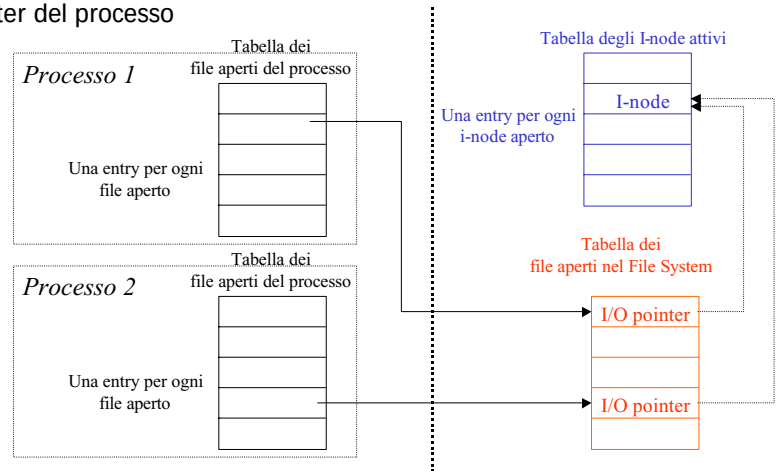
• La lettura e scrittura di un file avvengono a partire dalla posizione corrente del file indicata da un **I/O pointer**che viene modificato dalla operazione

• Le primitive ritornano il numero di byte effettivamente letti/scritti (una lettura di 0 byte significa che l'I/O pointer punta alla fine del file (EOF))

Operazioni sui file

Ogni processo ha la propria visione dei file aperti

- se più processi aprono lo stesso file, ciascun processo si riferisce ad un proprio I/O pointer distinto da quello degli altri
- le operazioni condotte da un processo su un file modificano il solo I/O pointer del processo



Operazioni sui file

Proprietà di `read` e `write`

- **`Sincronizzazione`**
  - **`read`** è normalmente sincrona: la primitiva attende la disponibilità dei dati richiesti a meno che non sia stato specificato il flag `O_NONBLOCK` in fase di apertura o creazione
  - **`write`** è normalmente semi-sincrona: la primitiva ritorna subito dopo aver scritto i dati in un buffer di kernel mentre l'effettiva scrittura sul disco viene portata a termine in modo asincrono (a meno che non sia stato specificato il flag `O_SYNC` in fase di apertura o creazione che comporta l'attesa della scrittura fisica sul disco)
- Atomicità
  - L'esecuzione di una singola primitiva `write` o `read` non è interrompibile mentre non è garantita l'atomicità di una sequenza di `read` e/o `write`
  - l'accesso esclusivo ad un file può essere ottenuto mediante `fcntl` oppure `flock`

Operazioni sui file

Esempi di lettura/scrittura : Copia di file (ver. 1)

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unixstd.h>
#define BUFSIZ 4096

main()
{
char    *f1= "filesorg"; char *f2= "/tmp/filedest";
int     infile, outfile; /* file descriptor */
int     nread;
char    buffer[BUFSIZ];

/* apertura file sorgente */
if ((infile=open(f1,O_RDONLY)) <0)
        { perror("Apertura f1"); exit(-1); }

/* creazione file destinazione */
if ((outfile=open(f2,O_WRONLY|O_CREAT|O_TRUNC, 0644)) <0)
        { perror("Creazione f2"); exit(-2);}

/* Ciclo di lettura/scrittura fino alla fine del file sorgente */
while((nread= read(infile, buffer, BUFSIZ)) >0)
        if(write(outfile, buffer, nread) != nread)
        { perror("Errore write"); exit(-3);}

close(infile); close(outfile);
exit(0);
}
```


Operazioni sui file

Copia di file (ver. 2 - uso argomenti)

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

#define PERM  0644

main(int argc, char *argv[])
{
int    infile, outfile; /* file descriptor */
int    nread;
char   buffer[BUFSIZ];

/* Controllo del numero degli argomenti */
if (argc != 3)
       { fprintf(stderr,"Uso: %s filesorg filedest\n",argv[0]);
       exit(-1); }
```

Operazioni sui file

Copia di file (ver. 2 - uso argomenti) (cont.)

```
/* apertura file sorgente */
if ((infile=open(argv[1],O_RDONLY)) <0)
       {fprintf(stderr,"Non posso aprire %s: %s\n", argv[1],
              strerror(errno)); exit(-2); }

/* creazione file destinazione */
if ((outfile=open(argv[2],O_WRONLY|O_CREAT|O_TRUNC, PERM)) <0)
       {fprintf(stderr,"Non posso creare %s: %s\n", argv[2],
                     strerror(errno)); exit(-3);}

/* Ciclo lettura/scrittura fino alla fine del file sorgente */
while((nread= read(infile, buffer, BUFSIZ)) >0)
       if(write(outfile, buffer, nread) != nread)
              { perror("Errore write"); exit(-4);}


close(infile); close(outfile);
exit(0);
}
```


Operazioni sui file

Copia di file (ver. 3 - richiede la redirezione dell' I/O)

```
/* copyredir.c */

#include <unistd.h>

#define BUFSIZ 4096

main()
{
int nread;
char buffer[BUFSIZ];

/* Ciclo di lettura/scrittura fino alla fine del file sorgente */

while((nread= read(0, buffer, BUFSIZ)) >0)
       if(write(1, buffer, nread) != nread)
              { perror("Errore write su stdout"); exit(-1);}


exit(0);
}
```

Esecuzione

```
$copyredir < filesorgente > filedestinazione
```


Operazioni sui file

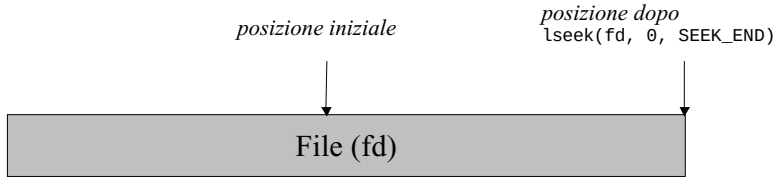
Riposizionamento non sequenziale dell' I/O pointer

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

off_t lseek(int fd, off_t offset, int whence);
```

Valori per il parametro whence

- SEEK_SET: l'I/O pointer punta ad `offset` byte dall'inizio del file;
- SEEK_CUR: l'I/O pointer punta a `offset` byte oltre la sua posizione corrente;
- SEEK_END: l'I/O pointer punta alla fine del file più `offset` byte;



Operazioni sui file

Informazioni su file (ordinari, speciali, direttori)

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

int stat(const char *filename, struct stat *buf);
int fstat(int fd, struct stat *buf);
```

Nella struttura `buf` vengono riportate le informazioni relative al file:

```
struct stat
   {
   dev_t         st_dev;     /* device */
   ino_t         st_ino;     /* inode */
   mode_t        st_mode;    /* protection */
   nlink_t       st_nlink;   /* number of hard links */
   uid_t         st_uid;     /* user ID of owner */
   gid_t         st_gid;     /* group ID of owner */
   dev_t         st_rdev;    /* device type (if inode device) */
   off_t         st_size;    /* total size, in bytes */
   unsigned long st_blksize; /* blocksize for filesystem I/O */
   unsigned long st_blocks;  /* number of blocks allocated */
   time_t        st_atime;   /* time of last access */
   time_t        st_mtime;   /* time of last modification */
   time_t        st_ctime;   /* time of last change */
   };
```

Operazioni sui file

Cancellazione di file

```
#include <unistd.h>

int unlink(const chat *filename);
```

Il file viene cancellato solo se:
- si tratta dell'ultimo link al file
- non vi sono altri processi che lo hanno aperto ; il file verrà effettivamente rimosso all'atto dell'ultima close

Nel caso si tratti di un link simbolico viene rimosso il link

Operazioni sui file

Controllo su file e dispositivi

```
#include <unistd.h>
#include <fcntl.h>

int fcntl(int fd, int cmd /*, arg */);
```

Vari comandi per il controllo sui file aperti: ad.es. gestione dei lock

```
#include <sys/ioctl.h>

int ioctl(int fd, int cmd /*, arg */);
```

Vari comandi per il controllo della modalità di funzionamento dei dispositivi

I comandi applicabili sono in genere determinati dalla tipologia del dispositivo anche se sono disponibili `ioctl` di carattere generale

`ioctl(fd, FIONREAD, &available)` in available viene restituito il numero di byte disponibili per la lettura

Operazioni sui file

Esempio di configurazione di un terminale su linea seriale

```
#include <unistd.h>
#include <sys/ioctl.h>
#include <sys/termios.h>


struct termios terminal;

...
/* Mancano i controlli sull'esito delle primitive */

fd = open("/dev/ttyS0", O_RDWR);

/* Ottiene la configurazione corrente della linea */
ioctl(fd, TCGETS, &terminal);

/* Modifica la configurazione per una comunicazione a 8 bit */
terminal.c_flag |= CS8 ;

/* Aggiorna la configurazione corrente della linea */
ioctl(fd, TCSETS, &terminal);

...
```