1.0 FILE DESCRIPTOR

Quando si tratta l'argomento di I/O non bufferizzato, ci riferiamo sempre al **file descriptor**, che sono degli interi non negativi (partono da 0). Ogni volta che all'interno di un processo viene aperto un file, il kernel gli assegna un file descriptor. Tutte le system call elencate in precedenza, esclusa la open, prenderanno in input il file descriptor del file e non il nome del file stesso. A differenza del normale utilizzo dei file in C che abbiamo sempre visto, in cui quando veniva invocata la fopen ci veniva restituito un puntatore al file, in questo caso ci viene quindi restituito un file descriptor, che è diverso. Il numero massimo di file che possono essere aperti al giorno d'oggi in un file sono 63.

Ogni nuovo processo apre 3 file standard di default: input, output, error e vi si riferisce ad essi con 3 file descriptor:

- **0** (STDIN_FILENO)
- **1** (STDOUT_FILENO)
- **2** (STDERR_FILENO)

Tenendo conto che all'apertura di un processo vengono generati di default questi file descriptor, è chiaro, il primo file che apriamo all'interno di un processo avrà come indice il 3.

1.1 OPEN

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *pathname, int oflag, ... /*, mode_t mode */);
```

Permette di aprire un file. Per apertura s'intende il venirsi restituito un file descriptor se l'operazione è andata bene. Altrimenti viene restituito -1. *mode_t mode* non è obbligatorio. *fd* restituito è il più piccolo numero non usato come *fd*.

L'argomento **oflag** è formato dall'OR di uno o più dei seguenti flag di stato:

- **Una ed una sola costante tra:**
  - *O_RDONLY ; O_WRONLY ; ORDWR*
- **Una qualunque tra (sono opzionali)**:
  - *O_APPEND*: tutto ciò che verrà scritto sarà posto alla fine.
  - *O_CREAT*: usato quando si usa open per creare un file.
  - *O_EXCL*: messo in OR con O_CREAT per segnalare errore se il file già esiste. Se il file non esiste non vengono segnalati errori.
  - *O_TRUNC*: se il file già esiste, aperto in write oppure read-write tronca la sua lunghezza a 0, cioè elimina qualsiasi contenuto precedente del file.
  - *O_SYNC (SVR4)*: se si sta aprendo in write, fa completare prima I/O.
  - *O_NOCTTY, O_NONBLOCK*

L'argomento **mode** viene utilizzato quando si crea un nuovo file utilizzando O_CREAT per specificare i permessi di accesso del nuovo file che si sta creando. Se il file già esiste questo argomento viene ignorato. I permessi si assegnano in questo modo:

mode	Description	
S_ISUID	set-user-ID on execution	4000
S_ISGID	set-group-ID on execution	2000
S_ISVTX	saved-text (sticky bit)	1000
S_IRWXU	read, write, and execute by user (owner)	0700
S_IRUSR	read by user (owner)	0400
S_IWUSR	write by user (owner)	0200
S_IXUSR	execute by user (owner)	0100
S_IRWXG	read, write, and execute by group	0070
S_IRGRP	read by group	0040
S_IWGRP	write by group	0020
S_IXGRP	execute by group	0010
S_IRWXO	read, write, and execute by other (world)	0007
S_IROTH	read by other (world)	0004
S_IWOTH	write by other (world)	0002
S_IXOTH	execute by other (world)	0001

Ci sono delle costanti come si vede dall'immagine. Concentriamoci sulla seconda tripletta: S_IRUSR indica il permesso di lettura per lo user, S_IWUSR permessi di scrittura per lo user e così via. La tripletta successiva è riferita al gruppo e l'ultima tripletta per other. Tutto ciò perché i permessi di un file sono organizzati in tre triplette: tripletta di permessi per l'utente, tripletta di permessi per il gruppo, tripletta di permessi per other.
Alla destra di ogni modalità vi è associato il contenuto reale di ogni costante: per esempio S_IRUSR è composto da 4 cifre ottali (in totale 12 bit).

Se per esempio mettiamo in OR S_IRUSR, S_IWUSR otteniamo:
000 100 000 000 || 000 010 000 000 = 000 110 000 000 → 0600, cioè permessi di lettura e scrittura per l'utente.
0644 indica che il proprietario può leggere e scrivere, mentre gruppo e other possono solo leggere.

Vediamo un esempio pratico →

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

int main(void)
{
  int    fd;

   fd=open("FILE",O_RDONLY);
   exit(0);
}
```

In questo caso, fd varrà 3 e subito dopo viene fatto exit. All'uscita del processo, tutti i file vengono chiusi.

In questo caso aggiungiamo dei flag messi in OR.
In particolare, vogliamo creare (O_CREAT) un file in wronly (O_WRONLY) e se per caso il file già esiste allora viene segnalato un errore (O_EXCL). Poiché stiamo creando un file, mettiamo anche i permessi: in questo caso con 0600 l'utente può leggere e scrivere, mentre gruppo e other non possono fare nulla. In caso di errore fd varrà -1. Se per caso si decidesse di mettere il file O_CREAT senza mettere O_EXCL e il file già esiste, quello che succede è che il file non viene ricreato ma si apre quello esistente, e i permessi che sono stati inseriti non vengono considerati, bensì vengono tenuti i precedenti.

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

int main(void)
{
  int    fd;
  fd=open("FILE1",O_CREAT|O_EXCL|O_WRONLY,0600);
   exit(0);
}
```

1.2 CREAT

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int create(const char *pathname, mode_t mode);
```

Viene citata solo per motivi storici. Quello che fa è creare un file dal nome pathname con i permessi descritti in mode. Se va tutto bene, viene restituito il fd del file aperto come write-only, -1 altrimenti.

Questo significa che questa chiamata è esattamente uguale a: **open(*pathname*, O_WRONLY | O_CREAT | O_TRUNC, *mode*)**.

Con la creat non posso aprire un file in modalità read-write, per cui questa non viene mai usata.

1.3 UMASK

```
#include <sys/types.h>
#include <sys/stat.h>
mode_t umask(mode_t cmask);
```

Questa system call serve per impostare la maschera di creazione per l'accesso ad un file. Il valore di ritorno è la maschera di creazione precedente (non restituisce un valore di errore).

Una maschera di creazione di un file permette di disabilitare alcuni permessi in maniera tale che erroneamente l'utente li setti. Viene usato ogni volta che il processo crea un nuovo file o directory, secondo il seguente criterio:

- Setta la maschera di creazione (cmask).
- Comando: umask
- Alla creazione del file viene fatto l'AND tra la maschera negata e il mode della creazione del file.

Dovunque ci sia un 1 nella maschera, tale permesso non verrà dato ad un file/directory.

Mettendo come maschera 0777, tutti i nuovi file non potranno avere alcun permesso. Con umask 0000, qualunque sia il permesso scelto dall'utente, verrà applicato senza restrizione.

Vediamo un esempio →

Se mettiamo umask(0), che corrisponde a 0000, con la creat del file *foo,* non verrà disabilitato alcun permesso che l'utente ha inserito nella creat. Dunque nella pratica il file foo avrà permessi 0666.
Con umask(S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH) che corrisponde a umask(0066), vengono disabilitati lettura e scrittura per gruppo e other, a prescindere dal fatto che l'utente li inserisca nei permessi alla creazione del file. Infatti, se creiamo un file *bar*, avremo che i suoi permessi sono 0600 in quanto quelli di gruppo e other sono disabilitati.

```
#include     <sys/types.h>
#include     <sys/stat.h>
#include     <fcntl.h>
                    /*esempio di utilizzo di umask*/
int main(void)
{
  umask(0);
  if(creat("foo",S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH)<0)
       printf("creat error for foo \n");

   umask(S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH);   /* 0066 */

if (creat("bar",S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH)<0)
       printf("creat error for bar \n");
   exit(0);
}
```

1.4 CLOSE

```
#include <unistd.h>
int close (int filedes);
```

Banalmente chiude il file con file descriptor filedes. Restituisce 0 se la procedura va a buon fine, -1 altrimenti. Quando un processo termina, tutti i file aperti vengono automaticamente chiusi dal kernel.

1.5 OFFSET

Quando si apre un file si vorrebbe scrivere o leggere dentro. Sorge il problema di dove iniziare a leggere o dove iniziare a scrivere. Questo è il concetto di **offset**. Normalmente, ogni file aperto ha assegnato un **current offset** (intero >0) che misura in numero di byte la posizione nel file. Operazioni come open e creat settano il current offset all'inizio del file a meno che O_APPEND sia specificato. Operazioni come read e write partono dal current offset e causano un incremento pari al numero di byte letti o scritti. Nonostante tutto ciò si ha la possibilità di potersi spostare liberamente all'interno di un file.

1.5.1 LSEEK

```
#include <sys/types.h>
#include <unistd.h>
off_t lseek(int filedes, off_t offset, int whence);
```

Questa è una system call che ci permette di decidere dove posizionarsi all'interno del file per leggere o scrivere. Restituisce il nuovo offset se OK (possiamo vederlo come intero), -1 altrimenti. Il primo parametro della system call è il file descriptor del file già aperto (si noti che non ci si riferisce più attraverso il nome).

L'argomento **whence** può assumere uno dei seguenti tre valori:

- **SEEK_SET**: ci si sposta del valore di offset a partire dall'inizio.
- **SEEK_CUR**: ci si sposta del valore di offset (positivo o negativo) a partire dalla posizione corrente.
- **SEEK_END**: ci si sposta del valore di offset (positivo o negativo) a partire dalla fine (taglia del file).

Quando si scrive all'interno di un file di testo per esempio "Ciao", e lo si salva all'interno di un file Ciao.txt, all'interno di questo file, a seconda di quale sia la rappresentazione dei set di caratteri c'è un determinato contenuto.

Quando si inserisce SEEK_END con una costante positiva, ci si sposta in avanti e solo se dopo questo spostamento c'è un'operazione scrittura allora tutti i caratteri precedenti tra la fine vecchia del file e il nuovo punto di inizio saranno riempiti da un carattere speciale: il carattere ASCII 0 che corrisponde a \0.

lseek permette dunque di settare il current offset oltre la fine dei dati esistenti nel file. Se vengono inseriti successivamente dei dati in tale posizione, si crea un buco. Una lettura nel buco restituirà byte con valore 0 (\0). Se lseek fallisce (restituisce -1), il valore del current offset rimane inalterato.

La chiamata di lseek non aumenta la taglia del file. Appena si scrive un nuovo carattere, la taglia aumenta di offset definita nella chiamata.

Con questo esempio di codice →

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

int main(void)
{
  int    fd,i;

   fd=open("FILE",O_RDONLY);
   i=lseek(fd,50,SEEK_CUR);
   exit(0);
}
```

Si apre un file di nome FILE in modalità lettura. Appena si apre un file il current offset è all'inizio. Successivamente c'è una lseek a cui passiamo il fd ricavato precedentemente. Con SEEK_CUR vogliamo spostarci in avanti a partire dalla posizione corrente, in questo caso di 50 byte. Dopo l'esecuzione della lseek, se si volesse stampare il valore di i, questo varrà 50.

In questo caso →

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


int main()
{
int fd, i;

fd= open("buco", O_CREAT|O_RDWR|O_TRUNC, 0644);
write(fd,"ABCDEF",6);
lseek(fd,10,SEEK_CUR);
write(fd,"GHILMN",6);

}
```

Si sta creando un file denominato *buco*, nel caso esista il valore precedente viene troncato. La modalità di apertura è in read-write e i permessi sono di lettura e scrittura per l'utente, lettura per group e other. Il current offset è a 0 in questo caso

Con la write si scrive all'interno del file che ha file descriptor fd, "ABCDEF". Con questa system call, il current offset attuale vale 6.

Con la lseek, rispetto alla posizione corrente, ci si sposta in avanti di 10 caratteri. Il current offset attuale è ancora 6, anche se virtualmente siamo alla posizione 16.

Con la successiva write, a partire dalla posizione 16, si scrive "GHILMN". In questo modo il nuovo current offset vale 22.

1.6 READ

```
#include <unistd.h>
ssize_t read (int filedes, void *buff, size_t nbytes);
```

Con la read, si legge dal file con file descriptor *filedes* un numero di nbyte e li mette in *buff*.

Il valore di ritorno è il numero di bytes letti, 0 se alla fine del file, -1 in caso di errore.

Naturalmente la lettura parte dal current offset, alla fine dell'operazione il current offset è incrementato del numero di byte letti.

Se il current offset è alla fine del file o anche dopo, viene restituito 0 e non vi è alcuna lettura.

In questo caso →

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

int main(void)
{
  int    fd,i;
  char   buf[20];

   fd=open("FILE",O_RDONLY);
   i=lseek(fd,50,SEEK_CUR);
   read(fd,buf,20);
   exit(0);
}
```

Viene aperto un file denominato *FILE* in modalità lettura. Si sposta il current offset in avanti di 50 byte. Vengono letti 20 caratteri e vengono memorizzati in buf. Alla fine di questa operazione il current offset vale 70.

1.7 WRITE

```
#include <unistd.h>
ssize_t write (int filedes, void *buff, size_t nbytes);
```

Con la write, si scrivono *nbyte* presi dal *buff* sul file con file descriptor *filedes*.

Vengono restituiti il numero di byte scritti se l'operazione va a buon fine, -1 in caso di errore.

La posizione da cui si comincia a scrivere è current offset. Alla fine della scrittura current offset è incrementato di *nbytes* e se tale scrittura ha causato un aumento della lunghezza del file anche tale parametro viene aggiornato.

Se viene richiesto di scrivere più byte rispetto allo spazio a disposizione (limite fisico), solo lo spazio disponibile è occupato e viene restituito il numero effettivo di byte scritti.

Se filedes è stato con O_APPEND allora current offset è settato alla fine del file in ogni operazione di write.

In questo esempio →

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

int main(void)
{
  int    fd,i,n;
  char   buf[20];

   fd=open("FILE",O_RDWR);
   i=lseek(fd,50,SEEK_CUR);
   n=read(fd,buf,20);
   write(1,buf,n);
   exit(0);
}
```

Viene aperto un file di nome *FILE* in modalità read-write. Ci posizioniamo alla posizione 50 rispetto all'inizio. Vengono letti al massimo 20 caratteri e vengono salvati in buf. La variabile n può valere al più 20.

La write scrive su standard output gli n caratteri che sono stati messi all'interno di buf.

1.8 EFFICIENZA DI I/0

Le operazioni di lettura e scrittura sono molto onerose. La CPU è in grado di eseguire molte istruzioni al secondo che operano sulla memoria interna. Quando l'operazione avviene nella memoria di massa (operazione I/O), il processo viene swappato dal processore, entra in una coda speciale per effettuare tale operazione e poi rientra nel processore per continuare l'esecuzione del codice. Ogni volta che si accede al disco è di conseguenza un'operazione molto lenta in confronto ad un accesso alla memoria centrale.

In questo programma →

```
#define BUFFSIZE 8192

int main(void)
{
  int   n;
  char buf[BUFFSIZE];

  while((n = read(STDIN_FILENO,buf,BUFFSIZE))>0)
  if (write(STDOUT_FILENO,buf, n) != n)
  printf("write error");
   if (n < 0)   printf("read error");
  exit(0);
}
```

Viene definita una costante BUFFSIZE con valore 8192. Si dichiara un array di dimensione BUFFSIZE. Parte poi il ciclo: fintantoché il valore di ritorno di una read è maggiore di 0, scrivi ciò che hai letto da qualche altra parte. La read legge da standard input (tastiera) 8192 caratteri e li memorizza in buf. Successivamente vengono scritti su standard output gli n caratteri letti da buf.

Questo esempio è stato mostrato per questo motivo →

Se si legge un carattere alla volta, il tempo di esecuzione in termini di CPU e sistema è di 117 secondi (2 minuti) che è molto tempo. Più caratteri si leggono alla volta, più il tempo di esecuzione si dimezza in ordine di potenze di 2.
Superati 8192 (migliore performance) non avviene più una diminuzione ma avviene un piccolo aumento del tempo, questo perché c'è un limite dell'hardware che non dipende più da quanto spesso la CPU chiede i dati, ma da quanto tempo l'hard disk ci mette a rispondere con i dati.

BUFFSIZE	User CPU (seconds)	System CPU (seconds)	Clock time (seconds)	Number of loops
1	20.03	117.50	138.73	516,581,760
2	9.69	58.76	68.60	258,290,880
4	4.60	36.47	41.27	129,145,440
8	2.47	15.44	18.38	64,572,720
16	1.07	7.93	9.38	32,286,360
32	0.56	4.51	8.82	16,143,180
64	0.34	2.72	8.66	8,071,590
128	0.34	1.84	8.69	4,035,795
256	0.15	1.30	8.69	2,017,898
512	0.09	0.95	8.63	1,008,949
1,024	0.02	0.78	8.58	504,475
2,048	0.04	0.66	8.68	252,238
4,096	0.03	0.58	8.62	126,119
8,192	0.00	0.54	8.52	63,060
16,384	0.01	0.56	8.69	31,530
32,768	0.00	0.56	8.51	15,765
65,536	0.01	0.56	9.12	7,883
131,072	0.00	0.58	9.08	3,942
262,144	0.00	0.60	8.70	1,971
524,288	0.01	0.58	8.58	986

1.9 CONDIVISIONE DI FILE

C'è la possibilità per più processi di aprire lo stesso file. Supponiamo di avere il seguente scenario:
2 processi aprono lo stesso file, ognuno si posiziona alla fine e scrive (in 2 passi):

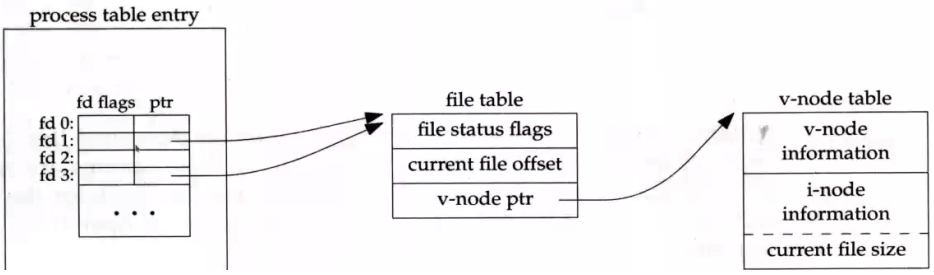
- lseek(fd, 0, SEEK_END);
- write(fd,buff, 100);

Cosa succede se i due processi partono contemporaneamente? Potrebbe accadere che si sovrascrive il contenuto di uno dei due processi. Per risolvere il problema, bisogna aprire il file con il flag O_APPEND in modo che questo fa posizionare l'offset alla fine, prima di ogni write. In questo modo l'operazione diventa atomica (passi che o sono eseguiti tutti insieme o non ne è eseguito nessuno).

1.9.1 DUP & DUP2

```
#include <unistd.h>
int dup(int filedes);
int dup2(int filedes, int filedes2);
```

Consentono di assegnare un altro file descriptor ad un file che già ne possedeva uno, cioè *filedes*.
Restituiscono entrambe il nuovo fd se l'operazione va a buon fine.
Ciò significa che quando si lancia in esecuzione dup(3) succede che viene creato un nuovo file descriptor cioè 4 che sarà una copia di 3: entrambi punteranno allo stesso file.
La differenza tra dup e dup2 è che con la dup si copia *filedes* sul primo file descriptor libero che c'è nella tabella dei file aperti dal processo. Con dup2 si assegna al file avente già file descriptor *filedes* anche il file descriptor *filedes2*.
Se filedes2 è gia open esso è prima chiuso e poi è assegnato a filedes. Se filedes2=filedes viene restituito direttamente filedes2.
L'effetto di dup/dup2 è il seguente:



O uso fd1 o fd3 è la stessa identica cosa (ad esempio hanno lo stesso current file offset). Questo è diverso dall'avere 2 processi distinti che operano sullo stesso file.