Operações usando descritores de arquivos

As principais funções operando sobre descritores de arquivos estão declaradas nos arquivos de cabeçalho `fcntl.h` e `unistd.h`.

Abrindo e fechando descritores

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open (const char *filename, int flags [, mode_t mode])
```

Cria e retorna um descritor para o arquivo indicado, ou -1 em caso de erro.

O argumento mode é usado para definir as permissões de acesso ao arquivo quando ele é criado. Os flags permitem controlar a forma de abertura do arquivo. Deve-se selecionar e combinar os flags desejados usando o operador "|" (OR binário). Os principais flags são:

- `O_RDONLY` : o arquivo é aberto para leitura.
- `O_WRONLY` : o arquivo é aberto para escrita.
- `O_RDWR` : o arquivo é aberto para leitura e escrita.
- `O_CREAT` : se o arquivo ainda não existe, deve ser criado usando as permissões definidas no parâmetro mode.
- `O_EXCL` : se `O_CREAT` e `O_EXCL` forem selecionados, a operação falha se o arquivo especificado já existe.
- `O_NOCTTY` : se o nome do arquivo corresponde a um terminal, o processo não deve ser associado ao terminal.
- `O_TRUNC` : ao abrir o arquivo em escrita, trunca seu tamanho para zero.
- `O_SHLOCK` : obtém uma trava compartilhada (*shared lock*) sobre o arquivo, de forma atômica.
- `O_EXLOCK` : obtém uma trava exclusiva sobre o arquivo, de forma atômica.
- `O_APPEND` : abre o arquivo em modo concatenação (*append*). É a única forma segura de fazer concatenações em arquivos compartilhados.
- `O_NONBLOCK` : habilita o modo não-bloqueante de leitura e escrita: a função `read` retorna imediatamente com erro se não houver entrada disponível; a função `write` retorna imediatamente com erro se não puder escrever imediatamente; a função `open` retorna imediatamente se não puder abrir ou criar o arquivo de imediato.
- `O_ASYNC` : habilita o modo de leitura assíncrono: serão gerados sinais `SIGIO` ao processo quando entradas estiverem disponíveis para leitura no arquivo.
- `O_FSYNC` ou `O_SYNC` : habilita o modo de escrita síncrono: cada chamada à função `write` só retorna quando os dados estiverem efetivamente no disco. Acrescenta segurança às operações de escrita, mas implica em queda de desempenho.

Ver também as funções `open64`, `creat` e `creat64`.

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

Fecha o arquivo correspondente ao descritor. Implica em descartar entradas não lidas, liberar travas e liberar o descritor para outros usos.

Exemplo: abertura de arquivo para leitura:

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

```
#include <fcntl.h>

int main (int argc, char *argv[])
{
   int fd ; /* file descriptor */

   printf ("Abrindo o arquivo x para leitura...\n") ;
   fd = open ("x", O_RDONLY) ;
   if ( fd < 0 )
   {
      perror ("Erro ao abrir arquivo x") ;
      exit (1) ;
   }
   printf ("Abri o arquivo x !\n") ;
   close (fd) ;
   exit (0) ;
}
```

Exemplo: abertura de arquivo para leitura e escrita; caso o arquivo não exista ele é criado, com permissões `0644`:

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

int main (int argc, char *argv[])
{
   int fd ; /* file descriptor */

   printf ("Abrindo o arquivo y para leitura...\n") ;
   fd = open ("y", O_RDWR | O_CREAT, 0640) ;
   if ( fd < 0 )
   {
      perror ("Erro ao abrir/criar arquivo y") ;
      exit (1) ;
   }
   printf ("Abri o arquivo y !\n") ;
   close (fd) ;
   exit (0) ;
}
```

Lendo e escrevendo dados

```
ssize_t
```

Tipo de dado usado para representar o número de blocos a ler ou escrever em uma operação `read` ou `write`.

```
#include <unistd.h>
ssize_t read (int filedes, void *buffer, size_t size)
```

Lê até `size` bytes do arquivo indicado pelo descritor `filedes`, armazenando o resultado em buffer. Retorna o número de bytes lidos, zero (EOF) ou -1 (erro). Ver também `pread` e `pread64`.

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

```
ssize_t write (int filedes, const void *buffer, size_t size)
```

Escreve até `size` bytes de dados contidos em buffer no arquivo indicado por `filedes`. Retorna o número de bytes escritos ou -1 (erro). Assim que a operação retorna, os dados estão disponíveis para leitura, mas não estão necessariamente no disco. Ver também `pwrite` e `pwrite64`.

Para leituras/escritas de grandes volumes de dados usando múltiplos buffers, torna-se mais apropriado usar as funções `readv` e `writev`.

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

Permite mudar a posição do ponteiro do arquivo indicado por `filedes`. O valor do deslocamento `offset` depende do modo de operação indicado por `whence`, que pode ser: relativo ao início do arquivo (SEEK_SET), à posição corrente (SEEK_CUR) ou ao final do arquivo (SEEK_END). Ver também `lseek64`.

Exemplo: abrir um arquivo em modo de leitura, ler seus primeiros 256 bytes e escreve-los na saída padrão:

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

#define SIZE 256

int main (int argc, char *argv[])
{
   int fd ; /* file descriptor */
   char buffer[SIZE] ;
   ssize_t bytesLidos, bytesEscritos ;

   /* abrir arquivo em leitura */
   fd = open ("x", O_RDONLY) ;
   if ( fd < 0 )
   {
      perror ("Erro ao abrir o arquivo x") ;
      exit (1) ;
   }

   /* ler SIZE bytes do arquivo */
   bytesLidos = read (fd, &buffer, SIZE) ;

   if ( bytesLidos < 0)
   {
      perror ("Erro na leitura de x") ;
      exit (1) ;
   }

   /* escrever os bytes lidos no terminal (stdout) */
   bytesEscritos = write (STDOUT_FILENO, &buffer, bytesLidos) ;

   if ( bytesEscritos < 0)
   {
      perror ("Erro na escrita em stdout") ;
      exit (1) ;
   }
```

```
   close (fd) ;
   exit (0) ;
}
```

Exemplo: abrir um arquivo em modo de leitura, ler seus últimos 256 bytes e escreve-los na saída padrão:

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

#define SIZE 256

int main (int argc, char *argv[])
{
   int fd ; /* file descriptor */
   char buffer[SIZE] ;
   ssize_t bytesLidos, bytesEscritos ;

   /* abrir arquivo em leitura */
   fd = open ("x", O_RDONLY) ;
   if ( fd < 0 )
   {
      perror ("Erro ao abrir o arquivo x") ;
      exit (1) ;
   }

   /* posicionar a SIZE bytes do final do arquivo */
   lseek (fd, -SIZE, SEEK_END) ;

   /* ler SIZE bytes do arquivo */
   bytesLidos = read (fd, &buffer, SIZE) ;

   if ( bytesLidos < 0)
   {
      perror ("Erro na leitura de x") ;
      exit (1) ;
   }

   /* escrever os bytes lidos no terminal (stdout) */
   bytesEscritos = write (STDOUT_FILENO, &buffer, bytesLidos) ;

   if ( bytesEscritos < 0)
   {
      perror ("Erro na escrita em stdout") ;
      exit (1) ;
   }

   close (fd) ;
   exit (0) ;
}
```

Interagindo com vários descritores

Algumas vezes um processo pode ter de esperar por dados vindos de diversas fontes. Um exemplo disso seria um processo funcionando como servidor para outros processos através de *pipes* ou *sockets*. Não é possível usar a função `read` para essa finalidade, pois ela irá aguardar em apenas um descritor por vez, ignorando os demais. Pode-se usar o modo não-bloqueante e varrer continuamente todos os descritores de interesse, mas isso não é eficiente. A função `select` permite aguardar até que a entrada ou saída em qualquer descritor de um conjunto de descritores esteja pronta (disponível). Também é possível estabelecer um prazo de espera (*time-out*).

Conjuntos de descritores são definidos pelo tipo `fd_set` e são manipulados pelas seguintes macros:

- `int FD_SETSIZE`: tamanho máximo do conjunto de descritores.
- `void FD_ZERO (fd_set *set)`: inicializa um conjunto como vazio.
- `void FD_SET (int filedes, fd_set *set)`: adiciona o descritor `filedes` ao conjunto `set`.
- `void FD_CLR (int filedes, fd_set *set)`: remove `filedes` do conjunto `set`.
- `int FD_ISSET (int filedes, const fd_set *set)`: testa se `filedes` está no conjunto `set`.

```
#include <sys/select.h>
int select (int nfds, fd_set *read_fds, fd_set *write_fds,
            fd_set *except_fds, struct timeval *timeout)
```

A função select bloqueia o processo até que ocorra alguma atividade nos primeiros `nfds` descritores indicados pelos conjuntos `read_fds`, `write_fds` e `except_fds`, ou que o prazo indicado por `timeout` tenha expirado.

- `read_fds`: descritores a esperar que estejam prontos para leitura, ou *null*.
- `write_fds`: descritore a esperar que estejam prontos para escrita, ou *null*.
- `except_fds`: descritores a esperar por condições excepcionais (vide sockets), ou *null*.
- `timeout`: estrutura do tipo `struct timeval` (vide timers) ou *null*.

Em caso de sucesso, a função retorna o número total de descritores prontos em todos os conjuntos. Cada conjunto é ajustado para conter apenas seus descritores prontos. Um retorno com 0 indica expiração de prazo e -1 indica erro. A recepção de um sinal fará com que select retorne imediatamente com um erro EINTR. Obs: A função `poll` também permite efetuar esperas em múltiplos descritores, mas não está na norma POSIX básica. Além disso, muitos sistemas a implementam `poll` usando `select`.

Sincronizando buffers

Em certos sistemas pode ser necessários assegurar que os dados escritos por processo sejam gravados no disco assim que possível, evitando que permaneçam muito tempo em buffers de memória.

```
#include <unistd.h>
int sync (void)
```

Força que os buffers de todos os descritores do sistema sejam escritos em disco.

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

Aguarda até que todo o buffer de escrita associado a `filedes` seja escrito no disco. Isso inclui os dados e meta-dados do arquivo.

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

```
int fdatasync (int fildes)
```

Aguarda até que todos os dados do arquivo tenham sido escritos em disco. Meta-dados não são considerados, portanto esta operação é mais rápida que `fsync`.

Obs: As operações de entrada/saída assíncronas (`aio_read`, `aio_write`, etc) somente foram incluídas no núcleo Linux 2.6, não estando disponíveis no núcleo Linux 2.4.

Controlando descritores

```
#include <unistd.h>
#include <fcntl.h>
int fcntl (int filedes, int command, ...)
```

Executa a operação indicada por command no descritor `filedes`. Alguns comandos podem necessitar de parâmetros adicionais, que são detalhados em sua documentação específica. Os principais comandos são apresentados brevemente a seguir:

- F_DUPFD: duplica o descritor (retorna outro descritor apontando para o mesmo arquivo aberto).
- F_GETFD: obtém os flags associados ao descritor.
- F_SETFD: ajusta os flags associados ao descritor.
- F_GETFL: obtém os flags associados ao arquivo aberto.
- F_SETFL: ajusta os flags associados ao arquivo aberto.
- F_GETLK: obtém uma trava no arquivo.
- F_SETLK: ajusta ou limpa uma trava.
- F_SETLKW: como F_SETLK, mas espera a conclusão do comando.

Duplicando descritores

Ao duplicar um descritor, cria-se um novo descritor que aponta para o mesmo arquivo aberto, embora seus modos de operação (*flags*) possam ser distintos. O uso mais freqüente da duplicação de descritores é a implementação da redireção de entrada ou saída de um processo.

```
#include <unistd.h>
int dup (int old)
```

Copia o descritor `old` no primeiro descritor disponível. Equivale a `fcntl (old, F_DUPFD, 0)`.

```
#include <unistd.h>
int dup2 (int old, int new)
```

Copia o descritor `old` no descritor new. Caso já esteja aberto, new é fechado antes. Equivale a `close (new); fcntl (old, F_DUPFD, new)`, mas efetuado de forma atômica.

Eis um exemplo de como usar dup2 para redirecionar a saída padrão de um processo para um arquivo, após um `fork`:

```
...
pid = fork ();
if (pid == 0)
{
  char *filename;
```

```
  char *program;
  int file;
  ...
  file = open (filename, O_RDONLY);
  dup2 (file, STDIN_FILENO);
  close (file);
  execv (program, NULL);
}
...
```

Travas em arquivos

As travas (*locks*) permitem a processos cooperantes trabalhar sobre os mesmos arquivos sem conflitos ou erros. Há basicamente dois tipos de travas:

- *Exclusive* ou *write locks* dão acesso exclusivo em escrita a uma parte do arquivo. Enquanto um *write lock* estiver ativo, nenhum outro processo pode travar aquela parte do arquivo.
- *Shared* ou *read locks* impedem outros processos de criar *write locks* sobre aquela parte do arquivo, mas permitem a existência de outros *read locks*.

As funções `read` e `write` não testam *locks*. Cabe ao programador implementar o controle de travas para impedir acessos conflitantes a partes de arquivos. Isso pode ser feito através das funções `flock` (para arquivos inteiros) e `fcntl` (para partes de arquivos).

Também é importante observar que *locks* são associados a processos, portanto um processo só pode ter um tipo de trava sobre um arquivo. Além disso, todas as suas travas são liberadas quando o processo fecha o arquivo ou encerra sua execução.

```
#include <sys/file.h>
int flock (int filedes, int operation)
```

Aplica ou remove uma trava sobre o arquivo indicado por `filedes`. O parâmetro operation pode ser LOCK_SH (*shared lock*), LOCK_EX (*exclusive lock*) ou LOCK_UN (*remove lock*). Além disso, esse parâmetro pode ser combinado com LOCK_NB (através de um OR) para obter operações não-bloqueantes. As operações sobre travas usando `fcntl` são mais complexas e estão detalhadas no manual da GLibC.

Exemplo: obter uma trava exclusiva sobre o arquivo cujo descritor é `fd`:

```
if (flock (fd, LOCK_EX) < 0)
{
   perror ("Erro ao obter trava") ;
   exit (1) ;
}
```