

Acesso a Arquivos

Laboratório de Sistemas Operacionais

Prof. MSc. João Tavares



JESUÍTAS BRASIL



Somos infinitas possibilidades

Introdução

- Arquivo é um tipo de dados abstrato criado pelo SO
- Persistência de dados é um dos usos mais comuns de arquivos
 - Porém, o conceito de arquivo é mais geral que isso!
 - Nem sempre um arquivo possui uma representação física em um meio de armazenamento
- Diversas outras estruturas do SO Unix são expostas aos processos com uma interface de arquivos
 - Ex.:
 - * controle de dispositivos físicos;
 - * pipes, sockets;
 - * informações do kernel e sobre os processos

Conceito de arquivo

- Representa uma sequência de registros (bytes)
 - Ordenamento dos registros é significativo e preservado
 - Atributo offset marca posição corrente na sequência
- Acessada para leitura e/ou escrita de registros
 - Nem todos os tipos suportam escrita+leitura
 - Ponteiro de arquivo atualizado a cada acesso
- Reside fora do espaço de memória do processo
 - Sequência é acessada pelos processos através de chamadas de sistema disponibilizadas pelo SO
 - Descritor de arquivo → representação do arquivo dentro de um processo

Descritor do arquivo

- SO mantém internamente um array de registros descrevendo cada arquivo aberto pelo processo
 - Processo não tem acesso direto a essa estrutura
 - Processo utiliza um mecanismo indireto para referenciar seus arquivos abertos
- Descritor de arquivo → índice para o arquivo de interesse na tabela de arquivos abertos do processo
 - Número inteiro não negativo entre 0 e OPEN_MAX
 - * Não confundir com o tipo FILE do C!
 - Passado como parâmetro nas chamadas de sistema feitas pelo processo para manipular seus arquivos abertos

E/S usando descritores de arquivo

- Interface básica E/S (padrao POSIX)
 - `read()` → lê bytes e atualiza offset
 - `write()` → escreve bytes e atualiza offset
 - `close()` → avisa que o arquivo não será mais usado, permitindo que o SO libere recursos antes alocados
- Nessa API, operações de leitura e escrita não sofrem buferização dentro do processo
 - Cada `read()` e `write()` gera uma chamada de sistema
 - Eficiência?

E/S usando descritores de arquivo

- Dependendo do tipo de arquivo, outras operações podem estar disponíveis
 - `lseek()` → modifica offset (posição de trabalho) para arquivos que suportam acesso randômico
 - `ioctl()` → modifica aspectos de arquivos que representam dispositivos
 - `fcntl()` → controla diversos atributos e possibilita monitoramento de modificações

Função read()

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

- Descrição
 - Copia no máximo count bytes do arquivo aberto referenciado por fd para o buffer apontado por buf
 - Dados são lidos da posição corrente do arquivo (offset) que é então atualizada
 - Retorna o número de bytes lidos do arquivo
 - 0 → se atingiu o fim do arquivo
 - 1 → em caso de erro de E/S
- Não há verificação de buffer overflow!
- Pode retornar antes de ler count bytes!

Função write()

API

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

- Descrição
 - Copia até count bytes do buffer buf para o arquivo aberto referenciado por fd
 - Dados são escritos da posição corrente do arquivo (offset) que é atualizada
 - Retorna o número de bytes escritos no arquivo
 - 1 → em caso de erro de E/S
- write() não verifica ocorrência de buffer overflow!
 - Ex.: escreve lixo se count > informação no buffer

Função close()

API `int close(int fd);`

- Descrição
 - Fecha o descritor de arquivo informado
 - * O descritor após fechado não pode mais ser utilizado pelo processo
- É saudável fechar arquivos após utilizá-los!
 - Arquivo aberto consome recursos escassos do kernel
 - * SO limita número de arquivos simultaneamente abertos por processo
 - Quando um processo termina, SO fecha todos descritores de arquivos ainda abertos
 - * Muitos programas se aproveitam disso e não fecham explicitamente seus arquivos

Função lseek()

API `off_t lseek(int fd, off_t offset, int mode);`

- Descrição
 - Modifica posição corrente para leitura ou escrita de um arquivo (offset do arquivo)
 - Interpretação de offset depende do valor de mode:
 - * `SEEK_SET` → parâmetro substitui posição corrente
 - * `SEEK_CUR` → parâmetro somado à posição corrente
 - * `SEEK_END` → parâmetro somado o fim de arquivo
 - Retorna a nova posição corrente do arquivo em caso de sucesso ou -1 em caso de erro
- Ideal quando precisa-se saltar partes do arquivo
- Nem todos os tipos de arquivos suportam lseek()!

“Truques” com lseek()

- Como descobrir a posição corrente do arquivo?
→ *offset 0 a partir da posição corrente*
Ex.: `posatual = lseek(fs, 0, SEEK_CUR)`
- Como descobrir o tamanho do arquivo?
→ *offset 0 a partir do fim do arquivo*
Ex.: `tam= lseek(fs, 0, SEEK_END)`
- Se a posição corrente é movida além do fim do arquivo, SO preenche espaço vazio com zeros na próxima escrita
 - Ex.: `lseek(fs, 1024, SEEK_END)`

Exemplo com lseek()

- *Ex.: ex-lseek.c*

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

int main(){
    int fd = STDOUT_FILENO;
    char buf1[] = "abcdefghij";
    char buf2[] = "ABCDEFGHIJ";

    if (write(fd, buf1, 10) != 10) perror("escrita buf1");
    /* offset=10 */
    if (lseek(fd, 40, SEEK_SET) == -1) perror("lseek");
    /* offset=40 */
    if (write(fd, buf2, 10) != 10) perror("escrita buf2");
    /* offset=50 */

    return(0);
}
```

Exemplo com lseek()

- Compilar

```
$ gcc -Wall ex-lseek.c -o ex-lseek
```

- Executar redirecionando o descritor correspondente à saída padrão (1) para arquivo

```
$ ./ex-lseek > file.hole
```

- Utilizando a ferramenta od, examinar o conteúdo do arquivo gerado

```
$ od -c file.hole
```

```
0000000  a b c d e f g h i j \0 \0 \0 \0 \0 \0
0000020  \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
0000040  \0 \0 \0 \0 \0 \0 \0 \0 A B C D E F G H
0000060  I J
0000062
```

Obtendo descritores de arquivo

- Descritores de arquivo padrão definidos em `unistd.h`
 - `STDIN_FILENO` (0) → entrada padrão do processo
 - `STDOUT_FILENO` (1) → saída padrão
 - `STDERR_FILENO` (2) → saída de erro padrão
- Herança
 - Descritores de arquivo são herdados pelo processo filho em um `fork()`
- Criação via chamadas de sistema específicas
Ex.: `open()`, `pipe()`, `socket()`
- Duplicação de outro descritor existente
Ex.: `dup()`, `dup2()`

Função open()

API

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

- Descrição
 - Abre um arquivo para leitura ou escrita
 - * Opcionalmente, pode criar o arquivo se não existe
 - Parâmetros
 - * pathname → caminho, absoluto ou relativo, para o arquivo no sistema de arquivos
 - * flags → controla diversas opções do open
 - * mode → define as permissões que serão atribuídas ao arquivo, se estiver sendo criado
 - Retorna um descritor de arquivo para o arquivo aberto em caso de sucesso ou -1 em caso de erro

Função open()

- Valor de flags é uma combinação “ou” de constantes
 - Obrigatório → especificar Modo de Acesso ao Arquivo
 - * O_RDONLY, O_WRONLY ou O_RDWR
 - Flags opcionais (descritos na manpage do open):
 - * O_APPEND, O_CREAT, O_EXCL, O_TRUNC, O_NOCTTY, O_NONBLOCK ou O_SYNC
- Valor de mode só é relevante se O_CREAT foi indicado em flags e o arquivo ainda não existe
- O valor retornado pelo open() é o menor número não usado para descritor de arquivo atualmente no processo

Exemplos de uso de open()

- *Abrindo um arquivo e criando um arquivo*

```
int fd;  
fd= open(nome, O_CREAT | O_RDWR, S_IRUSR|S_IWUSR);  
if (fd == -1) perror("Falha no open()");
```

- * Arquivo será criado, se não existir, com permissão 0600
- * O processo solicita acesso em leitura+escrita ao arquivo

Exemplos de uso de open()

- *Abrindo um arquivo pré-existente*

```
int fd;  
fd= open(nome, O_RDONLY);  
if (fd == -1) perror("Falha no open()");
```

- * Operação falhará se o arquivo não existir
- * Processo solicita acesso somente em leitura ao arquivo

Exemplo com open() + close()

- *Ex.: ex-open-close.c*

```
/* includes omitidos, consulte as man pages */

int main(void) {
    int fd;
    char nome[50];

    printf("Nome do arquivo a criar:");
    scanf("%49s", nome);
    fd = open(nome,
               O_CREAT | O_RDWR,
               S_IRGRP | S_IWGRP | S_IRUSR | S_IWUSR);
    if (fd == -1) {
        perror("Problema na criação!");
        exit(EXIT_FAILURE);
    }
    printf("Arquivo Criado.\n");
    close (fd);
    return (0);
}
```

E/S de arquivo usando API Streams

- API Streams → parte do padrão ANSI C
- Oferece alternativas bufferizadas para as operações de acesso a arquivos (chamados STREAMS)
 - `fopen()`, `fread()`, `fwrite()`, `fclose()`, etc.
 - Implementadas sobre as operações do padrão POSIX
- Utiliza tipo `FILE*` para representar cada stream ao invés do descritor de arquivo diretamente
 - Conversão `FILE*` ↔ fd possível através das funções `fdopen()` e `fileno()`
- Mais informações → man stdio

Outras chamadas de sistema...

- Pesquise também sobre as seguintes chamadas:
 - `stat()`, `lstat()` e `fstat()`
para obter informações sobre arquivos
 - `opendir()`, `readdir()` e `closedir()`
para obter informações de diretórios
 - `chmod()` e `fchmod()`
para alterar permissões de arquivos
 - `chown()`, `lchown()` e `fchown()`
para alterar o proprietário de um arquivo
 - `creat()`
alternativa para criação de arquivos
 - `select()`, `poll()`
monitoramento de múltiplos descritores de arquivos

Leituras complementares

- STEVENS, W.R. Advanced Programming in the UNIX Environment. 2nd. Ed., Addison Wesley, 2005.
- Man pages
 - cada uma das funções abordadas
 - * **Em particular, a man page do open(2)**
 - Overview da API Streams → man stdio
- Livro: Advanced Linux Programming
 - * Em particular o Apendice B: Low Level I/O

<http://www.advancedlinuxprogramming.com/alp-folder>

Referências Bibliográficas

- Material originalmente elaborado por Prof. Cristiano Costa. Material autorizado e cedido pelo autor. Revisado e atualizado por Prof. Luciano Cavalheiro e posteriormente pelo Prof. João Tavares.