Atividade Prática 6

Paula Raissa Silva

1 Comunicação entre Processos

1.1 pipe

```
#include <unistd.h>
int pipe(int fds[2]);
```

Um *pipe* em C estabelece um canal de comunicação entre dois processos, um de cada lado do *pipe*. Um processo escreve o dado de um lado, e o outro processo faz a leitura do outro lado. Assim, é estabelecido uma comunicação direta entre eles.

São necessários dois componentes para estabelecer um *pipe*: uma chamada para escrita (*write()*) e outra chamada para a leitura da informação a ser passada (*read()*). O fluxo de leitura e escrita é controlado pelo *file descriptor*, que é um vetor inteiro de duas posições. Cada posição desse vetor irá definir a entrada-padrão e a saída-padrão do *pipe*. O primeiro elemento do vetor ($fd[0]$) define a leitura (saída) de dados, e o segundo elemento ($fd[1]$) define a escrita de dados no pipe (entrada).

Pipes se comportam como uma estrutura de dados fila *FIFO* (First in First out). O tamanho de leitura e gravação não precisa ser o mesmo. Portanto, podemos escrever 512 bytes, mas podemos ler apenas 1 byte.

Alguns pontos interessantes sobre o uso de pipes:

- Se mais de um processo ou thread está escrevendo no pipe ao mesmo tempo, o kernel assegura que as escritas serão atômicas, ou seja, não haverá interrupções de escritas, desde que o número de bytes escritos seja menor ou igual PIPE_BUF. Esta definição está dentro do ficheiro */usr/include/linux/limits.h*.
- Se o pipe estiver vazio e chamarmos a chamada de sistema *read*, as leituras no *pipe* retornarão EOF (valor de retorno 0) se nenhum processo tiver o final de gravação aberto.
- A capacidade de um pipe de reter informação é limitada. Desde a versão 2.6.11 do Linux o limite de dados que um pipe pode armazenar é 65536 bytes. Mas, uma vez que uma aplicação for bem desenhada e que o lado de leitura do pipe consiga ler assim que o dado está disponível, não haverá problemas com esta limitação.
- Pipes podem somente ser usados por processos filhos, ou seja, processos pai com processos filhos, netos e etc. Existe um tipo de pipe que pode ser usado por processos diferentes chamado FIFO.

1.1.1 dup() and dup2()

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

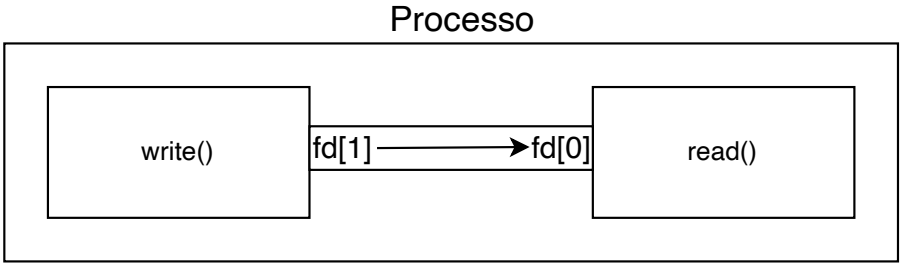

Figura 1: Pipe

A chamada de sistema dup() cria uma cópia de um *file descriptor*. Tal chamada usa o descritor não utilizado de número mais baixo para o novo descritor. Se a cópia for criada com êxito, os descritores de arquivo original e de cópia poderão ser usados alternadamente. Ambos se referem à mesma descrição de arquivo aberto e, portanto, compartilham sinalizadores de deslocamento e status de arquivo.

```
#include <unistd.h>
int dup2(int oldfd, int newfd);
```

Os ficheiros são geralmente manipulados após serem abertos usando a chamada de sistema *open*. Em caso de sucesso, open retorna um novo descritor de arquivo associado ao arquivo recém-aberto. Em sistemas baseados em Unix, o sistema operativo mantém uma lista de ficheiros abertos para cada programa em execução, chamada de tabela de ficheiros. Cada entrada é representada usando o inteiro do tipo *int*. Esses inteiros são chamados de descritores de arquivo nesses sistemas, e muitas chamadas de sistema tomam os valores do descritor de arquivo como parâmetros.

Cada programa em execução tem três descritores de ficheiro abertos por padrão quando o processo é criado, a menos que eles optem por fechá-los explicitamente. A chamada de sistema dup2() é semelhante a dup(), mas a diferença básica entre elas é que, em vez de usar o descritor de arquivo não utilizado de menor número, ele usa o número do descritor especificado pelo usuário. Se o descritor da *newfd* foi aberto anteriormente, ele é fechado antes de ser reutilizado. Se *oldfd* não for um descritor de arquivo válido, a chamada falhará e *newfd* não será fechado. Se *oldfd* for um descritor de arquivo válido e *newfd* tiver o mesmo valor que *oldfd*, *dup2()* não executará nada e retornará *newfd*.

Considerando que o objetivo do *dup2()* é duplicar o file descriptor para um número definido, pode-se ter um processo filho que está a colocar um *pipe* no lugar do file descriptor 0 que é o *stdin* e outro pipe no lugar do *stdout*. Isto só é possível faz quando o processo filho vai chamar *exec()* em seguida, para que o novo executável "herde"os pipes como stdin e stdout. Se o processo filho for continuar a executar sem chamar *exec()*, pode-se usar os file descriptors do *pipe* para comunicar com o processo pai.

1.2 Socketpair

```
#include <sys/types.h>
#include <sys/socket.h>
int socketpair(int domain, int type, int protocol,
               int socket_vector[2]);
```

Para que um processo se comunique com o outro de forma bidirecional é necessário ter duas instâncias de *pipe* para que o canal de escrita do pipe A se conecte ao canal de leitura do

pipe B e vice e versa. Em termos de código para realizar essa implementação é necessário criar duas instâncias de pipe e inserir como argumento ambos os handles para que a comunicação bidirecional seja possível. O *socketpair* é uma alternativa para obter esse comportamento de forma simplificada.

O *socketpair* é uma chamada de sistema capaz de conectar dois sockets de forma simplificada sem a necessidade de toda a inicialização de uma conexão tcp ou udp, devolvendo dois sockets conectados que podem ser usados para realizar a comunicação entre si de forma bidirecional.

Esse é um dos melhores métodos de IPC, pois permite a comunicação entre processos na mesma máquina e em máquinas fisicamente separadas, também é possível se comunicar com outras tecnologias baseado em um protocolo padrão, além disso, permite outras utilidades como comunicação entre threads para evitar concorrências, criação de padrões arquiteturais como cliente/servidor, bem como a criação da biblioteca de comunicação conhecida como *zeromq*. Porém com toda essa facilidade, gera-se um grande problema quando precisa-se trafegar os dados em uma rede pública, quando feito dessa forma os dados estão sendo expostos, mas existem formas de protegê-los.

O socketpair é uma boa alternativa para o uso de pipes pois facilita o modo de comunicação entre processos caso haja necessidade que os processos se comuniquem entre si, ou seja, de forma bidirecional. Devido a sua facilidade de criação do par de sockets, isso o torna relativamente fácil de usar. Portanto caso houver a necessidade de usar pipes, se possível prefira o socketpair por garantir a comunicação entre ambas as direções.

1.3 mmap

```
#include <sys/mman.h>
void *mmap(void *addr, size_t length, int prot,
           int flags, int fd, off_t offset);

int munmap(void *addr, size_t length);
```

O *mmap* cria um novo mapeamento de memória virtual, que pode ser de dois tipos: Mapeamento de arquivo, onde é necessário um arquivo para poder mapear essa memória, e o mapeamento anônimo, nesse caso não é necessário um arquivo para realizar o mapeamento. O espaço de endereçamento consiste em várias páginas e cada página pode ser mapeada em algum recurso. Podemos criar esse mapeamento para os recursos que queremos usar. O mapeamento pode ser de dois tipos privado ou compartilhado, para que esse recurso possa ser usado na forma de IPC, precisa ser do tipo compartilhado, sendo esse o modo que será abordado nos exercícios. Se dois processos utilizarem o mesmo arquivo para mapear essa memória com o tipo compartilhado, um processo pode enxergar o que o outro está a alterar, o que permite a troca de mensagem entre eles.

Para destruir um mapeamento realizado, é utilizado a chamada de sistema *munmap*, que libera essa porção de memória alocada.

2 Sinais

Sinal é uma notificação que avisa um determinado processo que um evento ocorreu. Um sinal é considerado uma interrupção por software, similar a interrupção via hardware, onde quando há um evento, o fluxo do programa é alterado, normalmente chamando uma função que

foi registrada para ser invocada quando esse sinal acontecer. Sinal pode ser considerado um IPC porém não transmite dados, e são assíncronos. Quando um processo o recebe, interrompe o processamento atual para atender o evento, ou seja, assim que um evento é recebido, o processamento é imediato, como boa prática os handlers registrados para os sinais devem possui uma rotina muito pequena para o tratamento desse sinal, para que possa retornar rapidamente para o ponto onde foi interrompido.

Signal pode ser considerado um IPC porém não transmite dados, e são assíncronos, mas quando um processo o recebe, interrompe o processamento atual para atender o evento, ou seja, assim que um evento é recebido, o processamento é imediato, como boa prática os handlers registrados para os sinais devem possui uma rotina muito pequena para o tratamento desse sinal, para que possa retornar rapidamente para o ponto onde foi interrompido.

Para realizar um registro de uma callback faz-se o uso da system call signal que recebe dois argumentos o SIG[TIPO] que é o evento, e a callback que será executado quando houver o evento, onde a callback deve respeitar a assinatura do sighandler, que recebe um argumento do tipo int e não retorna nada.

```
#include <signal.h>
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

Com Signal é possível emitir o evento para si mesmo através da system call

```
#include <signal.h>
int raise(int sig);
```

ou para um processo externo, nesse caso é necessário conhecer o pid do processo no qual se quer enviar

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);
```

Para fazer com que o programa somente processe mediante a um evento, pode-se usar a system call pause() que permite que o programa entre em em modo *sleep* até que um sinal seja recebido para assim acordar e processar.

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

3 Resolução dos Exercícios

Ver os ficheiros, em anexo, ficha6_parte1.zip e ficha6_parte2.zip. Nos ficheiros *.zip* estão os comentários com explicações para os exercícios resolvidos.