# Criando Pipes em C

Para criar um pipe simples com C, utilizamos a chamada do sistema pipe(). Essa chamada necessita de um único argumento, que é um array de dois inteiros e, se a chamada for bem-sucedida, o array conterá dois novos descritores de arquivos que serão usadas para o pipeline. Depois de criar um pipe, o processo normalmente gera um novo processo (lembre-se de que o filho herda os descritores de arquivos abertos do pai).

```
CHAMADA DE SISTEMA: pipe();

PROTÓTIPO: int pipe( int fd[2] );
  RETORNA: 0 se sucesso
           -1 se erro: errno = EMFILE (sem descritores livres)
                               EMFILE (tablea de arquivos do sistema está cheia)
                               EFAULT (o array passado como parâmetro é inválido)

NOTAS: fd[0] é configurado como somente leitura, fd[1] como somente escrita
```

O primeiro inteiro no array (elemento 0) é configurado e aberto para leitura, enquanto o segundo inteiro (elemento 1) está configurado e aberto para gravação. Visualmente falando, a saída de fd1 se torna a entrada para fd0. Todos os dados que passam pelo canal se movem através do kernel.

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

main()
{
        int     fd[2];

        pipe(fd);
        .
        .
}
```

Lembre-se de que um nome de vetor em C é convertido em um ponteiro para seu primeiro elemento. Acima, `fd` é equivalente a `&fd[0]`. Uma vez estabelecido o pipeline, utilizamos o `fork` para criarmos um processo filho:

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

main()
{
        int     fd[2];
        pid_t   childpid;

        pipe(fd);

        if((childpid = fork()) == -1)
        {
                perror("fork");
                exit(1);
        }
        .
        .
}
```

Se o pai quiser receber dados do filho, ele deve fechar fd1, e o filho deve fechar fd0. Se o pai quiser enviar dados para o filho, deve fechar fd0, e o filho deve fechar fd1. Como os descritores são compartilhados entre o pai e a filho, devemos sempre fechar o fim do pipe que não iremos utilizar. Tecnicamente falando, o fim de arquivo (EOF) nunca será retornado se as extremidades desnecessárias do pipe não forem explicitamente fechadas.

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

        main()
        {
                int     fd[2];
                pid_t   childpid;

                pipe(fd);

                if((childpid = fork()) == -1)
                {
                        perror("fork");
                        exit(1);
                }

                if(childpid == 0)
                {
                        /* Processo filho fecha seu lado de entrada do pipe*/
                        close(fd[0]);
                }
                else
                {
                        /* Processo pai fecha a saida do pipe */
                        close(fd[1]);
                }
                .
                .
        }
```

Como dito anteriormente, uma vez que o pipeline tenha sido estabelecido, os descritores de arquivo podem ser tratados como descritores para arquivos normais.

```
/*****************************************************************************
 Retirado de "Linux Programmer's Guide - Chapter 6"
 (C)opyright 1994-1995, Scott Burkett
 *****************************************************************************
 MODULE: pipe.c
 *****************************************************************************/

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

int main(void)
{
        int     fd[2], nbytes;
        pid_t   childpid;
        char    string[] = "Hello, world!\n";
        char    readbuffer[80];

        pipe(fd);

        if((childpid = fork()) == -1)
        {
                perror("fork");
                exit(1);
        }

        if(childpid == 0)
```

```
        {
                /* Child process closes up input side of pipe */
                close(fd[0]);

                /* Send "string" through the output side of pipe */
                write(fd[1], string, (strlen(string)+1));
                exit(0);
        }
        else
        {
                /* Parent process closes up output side of pipe */
                close(fd[1]);

                /* Read in a string from the pipe */
                nbytes = read(fd[0], readbuffer, sizeof(readbuffer));
                printf("Received string: %s", readbuffer);
        }

        return(0);
}
```

Frequentemente, os descritores no filho são duplicados na entrada padrão ou saída. O filho pode então chamar exec() para executar outro programa, que herda os fluxos padrão. Vamos dar uma olhada na chamada de sistema dup():

```
 CHAMADA DE SISTEMA: dup();

  PROTÓTIPO: int dup( int oldfd );
    RETORNA: novo descrito se sucesso
             -1 se erro: errno = EBADF (oldfd não é um descritor válido)
                                 EBADF (newfd está fora do limite)
                                 EMFILE (número de descritores abertos pelo processo excedido)

  NOTA: o descritor antigo não é fechado! Os dois fazem referência ao novo arquivo.
```

Embora o descritor antigo e o descritor recém-criado possam ser usados de forma intercambiável, normalmente fechamos um dos fluxos padrão primeiro. A chamada do sistema dup() usa o menor descritor não utilizado para o novo.

Considere:

```
        .
        .
        childpid = fork();

        if(childpid == 0)
        {
                /* Fecha a entrada padrao do filho */
                close(0);

                /* Duplica a entrada do pipe na entrada padrao. Isso substitui
                a entrada padrao pelo pipe */
                dup(fd[0]);
                execlp("sort", "sort", NULL);
                .
        }
```

Como o descritor de arquivos 0 (stdin) foi fechado, a chamada para dup() duplicou o descritor de entrada do pipe (fd0) em sua entrada padrão. Nós então fazemos uma chamada para execlp() (ou execvp()), para sobrepor o segmento de texto (código) do filho com o do programa sort. Como programas recém-executados herdam fluxos padrão de seus chamadores, ele realmente herda o lado de entrada do pipe como sua entrada padrão! Agora, qualquer coisa que o processo pai original envia para o pipe, vai para o processo sort.

Existe outra chamada de sistema, dup2(), que pode ser usada também. Esta chamada particular foi criada com a Versão 7 do UNIX e agora é requerida pelo padrão POSIX.

```
CHAMADA DE SISTEMA: dup2();

PROTÓTIPO: int dup2( int newfd, int oldfd );
  RETORNA: novo descritor se sucesso
          -1 se erro: errno =  EBADF (oldfd não é um descritor válido)
                               EBADF (newfd está fora do limite)
                               EMFILE (número de descritores abertos pelo processo excedido)

NOTES: o descritor antigo é fechado com dup2()!
```

Com esta chamada em particular, fechamos o descritor antigo e duplicamos no novo, em uma chamada de sistema. Além disso, é garantido que essa chamada seja atômica, o que essencialmente significa que nunca será ser interrompida por um sinal. Com o chamada original do sistema dup(), os programadores tinham que executar uma operação close() antes de chamá-lo. Isso resulta em duas chamadas de sistema, com um pequeno grau de vulnerabilidade no curto período de tempo decorrido entre eles. Se um sinal chegasse durante essa breve período, a duplicação do descritor iria falhar. Claro, dup2() resolve esse problema para nós.

Considere:

```
.
.
childpid = fork();

if(childpid == 0)
{
        /* Fecha stdin, e duplica a saída do pipe para stdin */
        dup2(fd[0], 0);
        execlp("sort", "sort", NULL);
        .
        .
}
```