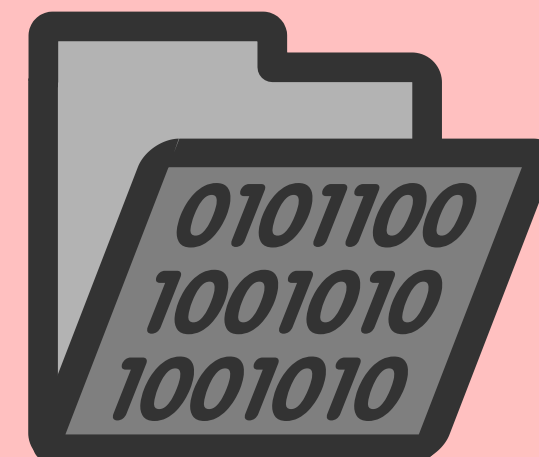


Comunicação entre Processos

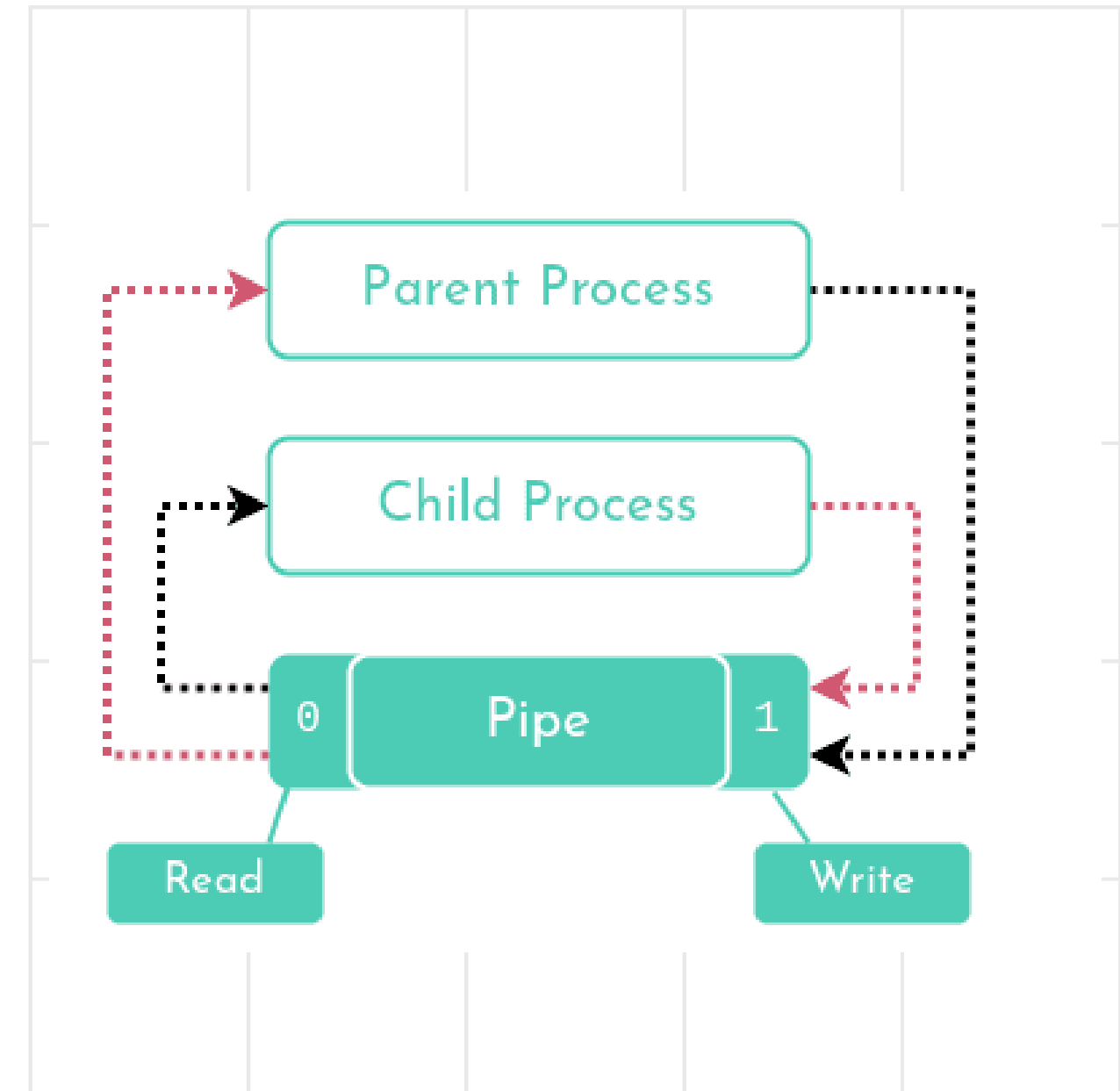
(usando a API do kernel e a Standard C Library - Parte I)



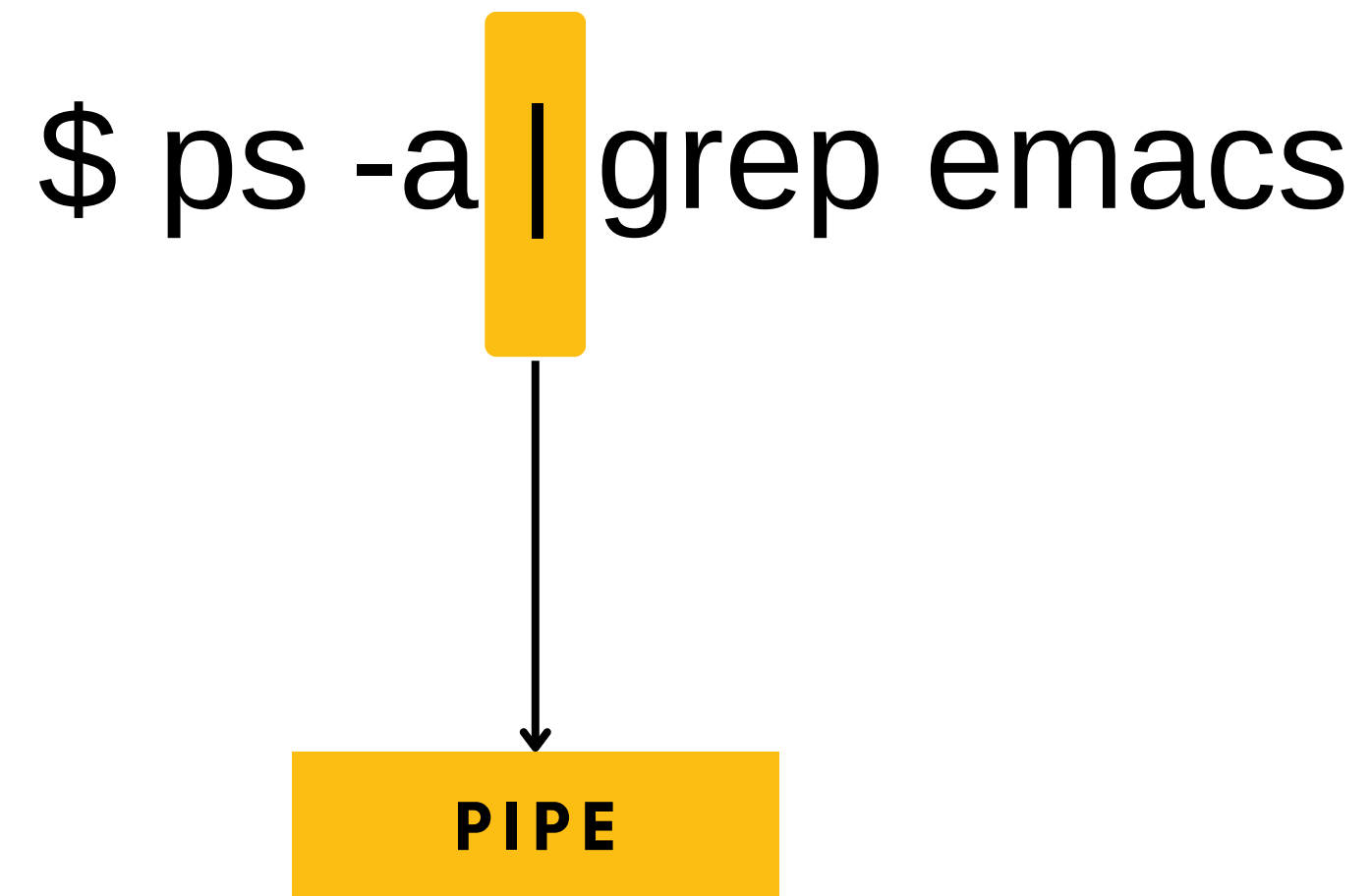
Bem-vindos à aula!

Agenda de Hoje

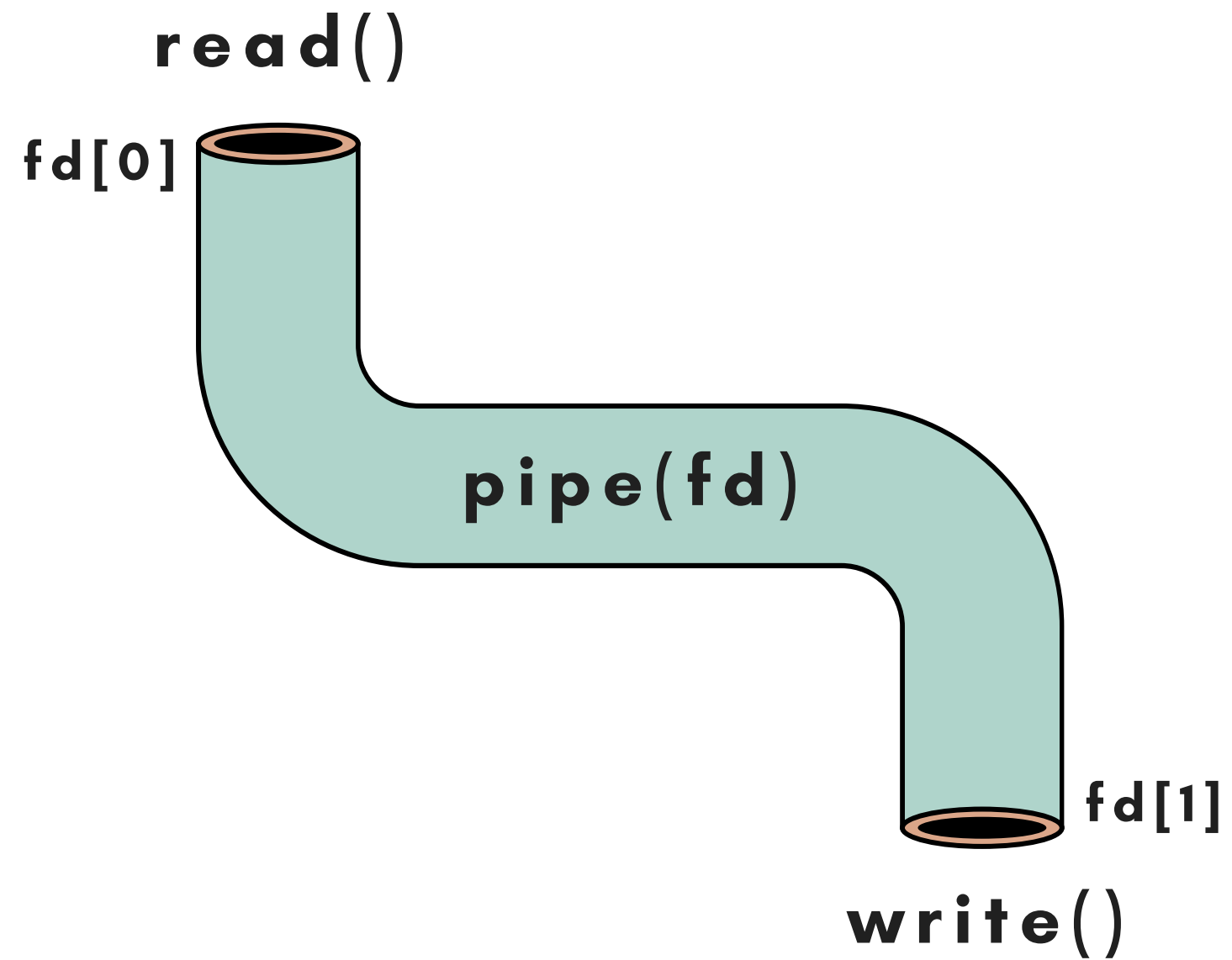
- Pipes
- Sockets
- Named pipes
- Redirecting standard output (dup and dup2)



pipes



Pipe em C



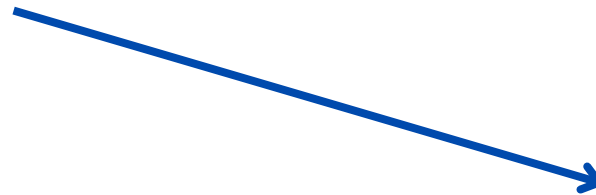
Pipe em C

Na prática, um pipe é um mecanismo de comunicação entre processos que funciona como um **buffer** de dados em memória. Aqui estão alguns pontos importantes sobre o funcionamento dos pipes:

- **Unidirecionalidade:** Um pipe é unidirecional, o que significa que os dados fluem em uma única direção. Um processo escreve no pipe e outro processo lê do pipe.
- **Acesso exclusivo:** Em um dado momento, um processo pode escrever no pipe enquanto outro processo pode ler do pipe. No entanto, múltiplos processos podem ler do mesmo pipe, mas a ordem de leitura pode não ser garantida.
- **Buffer de dados:** O pipe atua como um **buffer de dados temporário**. Quando o processo de escrita escreve dados no pipe, esses dados ficam armazenados no buffer até que o processo de leitura os leia.
- **Bloqueio:** Se o buffer do pipe estiver cheio, o processo de escrita será bloqueado até que o processo de leitura leia alguns dados e libere espaço no buffer. Da mesma forma, se o buffer estiver vazio, o processo de leitura será bloqueado até que o processo de escrita escreva alguns dados.

Pipe em C

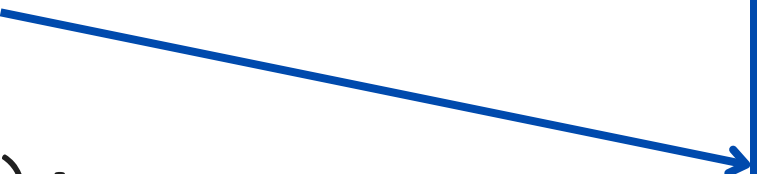
```
int main() {  
    int fd[2];  
    pid_t pid;  
    char buf;  
  
    if (pipe(fd) == -1) {  
        perror("pipe");  
        exit(EXIT_FAILURE);  
    }  
  
    pid = fork();  
    if (pid == -1) {  
        perror("fork");  
        exit(EXIT_FAILURE);  
    }  
}
```



Define um vetor de inteiros de tamanho 2, onde:
fd[0]: file descriptor para leitura
fd[1]: file descriptor para escrita

Pipe em C

```
int main() {  
    int fd[2];  
    pid_t pid;  
    char buf;  
  
    if (pipe(fd) == -1) {  
        perror("pipe");  
        exit(EXIT_FAILURE);  
    }  
  
    pid = fork();  
    if (pid == -1) {  
        perror("fork");  
        exit(EXIT_FAILURE);  
    }  
}
```



Um pipe é criado com a função pipe(), que cria um par de descritores: um para leitura e outro para escrita.

Pipe em C

```
int main() {  
    int fd[2];  
    pid_t pid;  
    char buf;  
  
    if (pipe(fd) == -1) {  
        perror("pipe");  
        exit(EXIT_FAILURE);  
    }  
  
    pid = fork();  
    if (pid == -1) {  
        perror("fork");  
        exit(EXIT_FAILURE);  
    }  
}
```



**fork() cria um novo processo.
O processo filho herda os
descritores do pipe.**

Pipe em C

```
if (pid == 0) { // Processo filho
    close(fd[1]);
    while (read(fd[0], &buf, 1) > 0) {
        write(STDOUT_FILENO, &buf, 1);
    }
    close(fd[0]);

} else {
    close(fd[0]);
    write(fd[1], "Olá, mundo!\n", 12);
    close(fd[1]);
    wait(NULL);
}

return 0;
}
```

Processo Filho:

- Fecha o descritor de escrita com `close(fd[1])`
- Lê a mensagem do pipe `read(fd[0])`
- Escreve a mensagem na saída padrão (`STDOUT_FILENO`).
- Fecha o fd de leitura para indicar que não precisa mais ler do pipe.

Pipe em C

```
if (pid == 0) { // Processo filho
    close(fd[1]);
    while (read(fd[0], &buf, 1) > 0) {
        write(STDOUT_FILENO, &buf, 1);
    }
    close(fd[0]);
```

```
} else { // Processo pai
    close(fd[0]);
    write(fd[1], "Olá, mundo!\n", 12);
    close(fd[1]);
    wait(NULL);
}
```

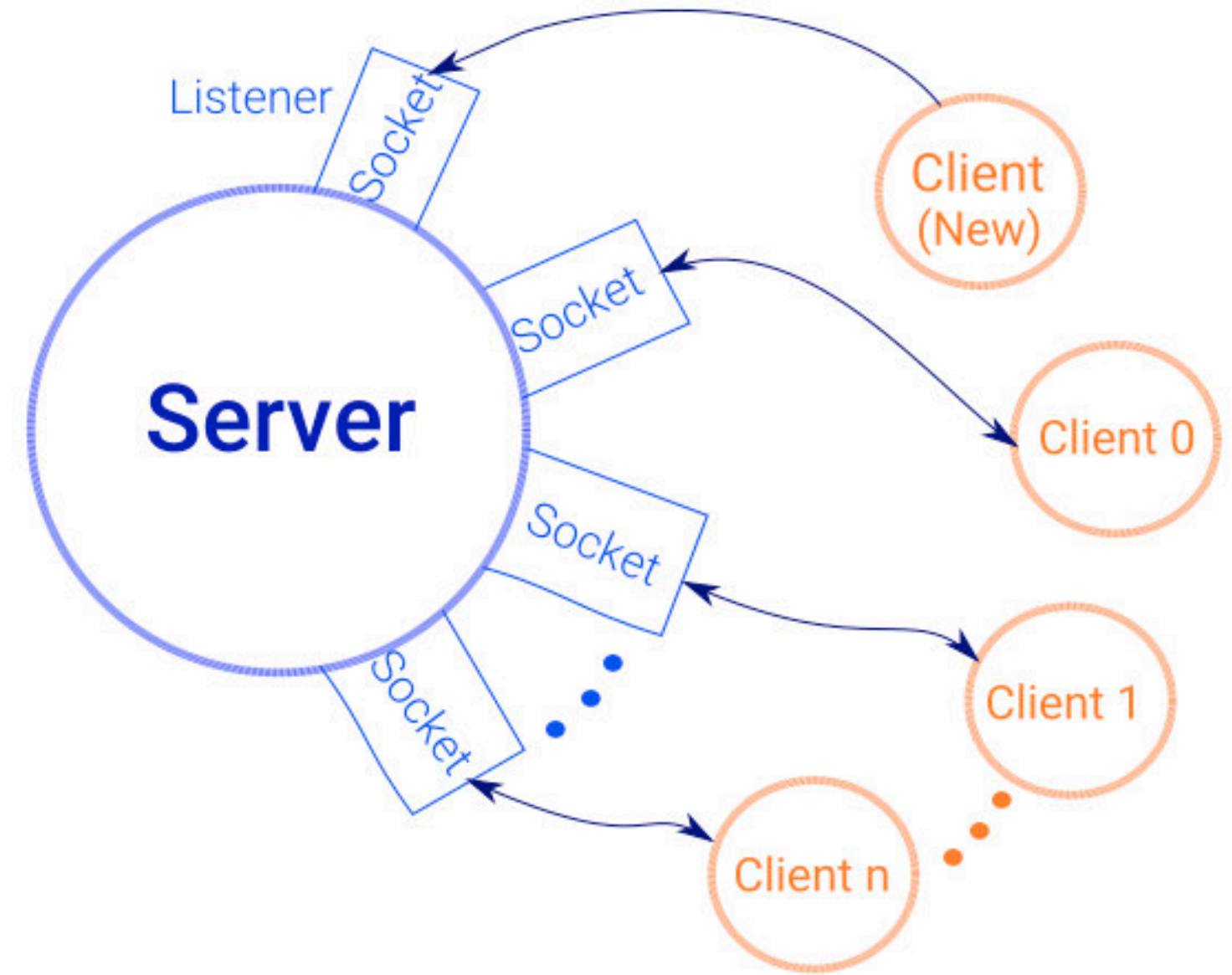
```
return 0;
```

```
}
```

Processo Pai:

- Fecha o descritor de leitura (fd[0])
- Escreve uma mensagem no pipe
- Fecha o descritor de escrita para enviar EOF ao filho
- Espera o processo filho terminar

sockets



Server and Clients

Sockets

- Definição: Sockets são pontos de comunicação que permitem a troca de dados entre processos, seja no mesmo sistema ou em sistemas diferentes.
- Tipo de Socket:
 - AF_UNIX ou AF_LOCAL: Usados para comunicação entre processos no mesmo sistema.
- Bidirecional: `socketpair` cria um par de sockets conectados que permitem comunicação bidirecional. Ambos os processos podem ler e escrever nos sockets.
- Exemplo: `socketpair(AF_UNIX, SOCK_STREAM, 0, sv)` cria dois sockets conectados `sv[0]` e `sv[1]`, ambos podem ser usados para leitura e escrita.

Sockets

Sintaxe:

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

Parâmetros:

- domain: Especifica o domínio do socket. Para socketpair(), geralmente é AF_UNIX ou AF_LOCAL.
- type: Especifica o tipo de socket. Pode ser SOCK_STREAM (para comunicação baseada em fluxo) ou SOCK_DGRAM (para comunicação baseada em datagramas).
- protocol: Normalmente definido como 0, pois o protocolo padrão é usado.
- sv: Um array de dois inteiros onde os file descriptors dos sockets criados serão armazenados.

Sockets

pipe

```
int fd[2];
pipe(fd);
if (fork() == 0) { // Processo filho
    close(fd[1]); // Fecha o
    // descritor de escrita
    char buf[100];
    read(fd[0], buf, sizeof(buf));
    printf("Filho leu: %s\n", buf);
    close(fd[0]);
} else { // Processo pai
    close(fd[0]); // Fecha o
    // descritor de leitura
    write(fd[1], "Olá, filho!", 12);
    close(fd[1]);
}
```

socketpair

```
int sd[2];
socketpair(AF_UNIX, SOCK_STREAM, 0,
sd);
if (fork() == 0) { // Filho
    close(sd[0]); // Fecha o fd do
    // pai
    char buf[100];
    read(sd[1], buf, sizeof(buf));
    printf("Filho leu: %s\n", buf);
    close(sd[1]);
} else { // Processo pai
    close(sd[1]); // Fecha o
    // descritor do filho
    write(sd[0], "Olá, filho!", 12);
    close(sd[0]);
}
```

Sockets

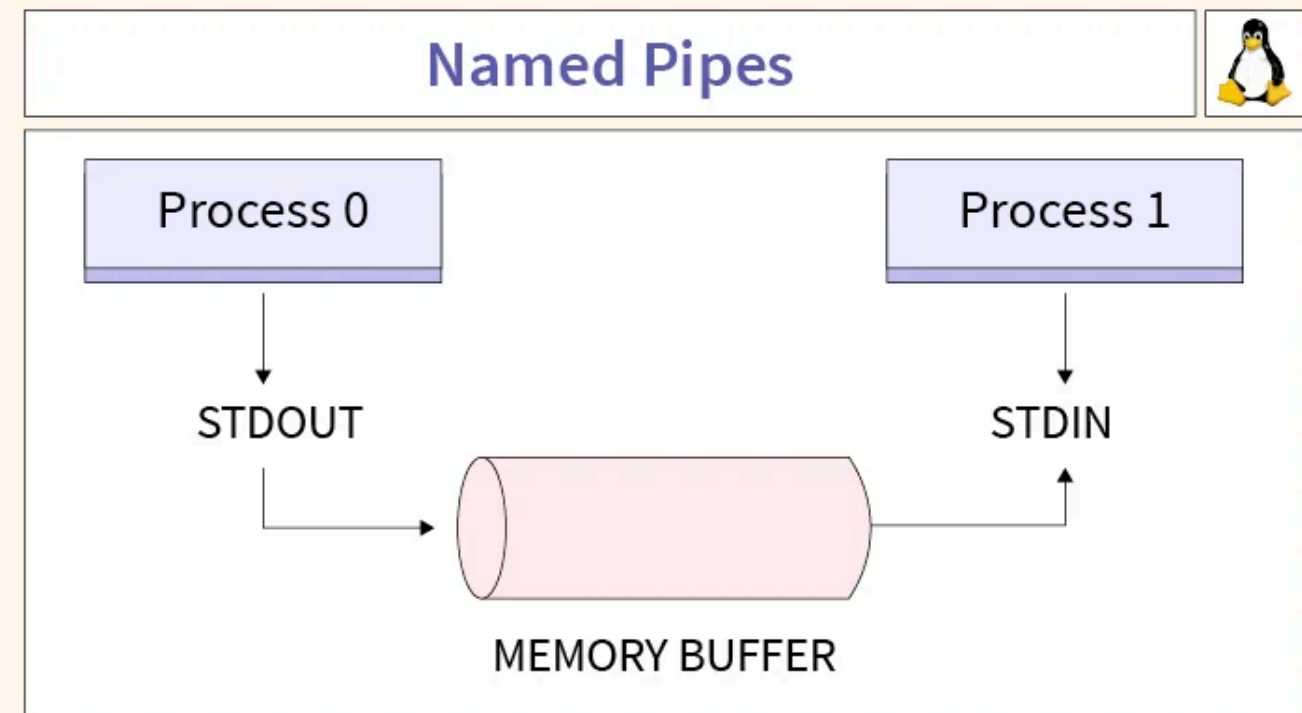
Em C existem vários tipos de sockets além de `socketpair()`. Aqui estão alguns dos principais tipos de sockets para comunicação interprocessual e em rede:

- `AF_INET/AF_INET6`: Para comunicação de rede (IPv4/IPv6).
- `AF_UNIX`: Para comunicação local entre processos.
- `AF_NETLINK`: Para comunicação entre o kernel e processos de espaço de usuário.
- `AF_TIPC`: Para comunicação em clusters de computadores.

Sintaxe:

```
#include <sys/types.h>
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

named pipes



Named Pipes

Também conhecidos como FIFOs (First In, First Out), são uma extensão do conceito de pipes tradicionais em sistemas Unix/Linux. Eles permitem a comunicação entre processos, mas diferentemente dos pipes anônimos, os named pipes têm um nome no sistema de ficheiros e podem persistir além da vida dos processos que os criaram.

Características:

1. Persistência:

- Named pipes existem como ficheiros no sistema de ficheiros e podem ser acessados por múltiplos processos, mesmo que os processos que os criaram já tenham terminado.

2. Comunicação Bidirecional:

- Assim como os pipes anônimos, os named pipes permitem a comunicação unidirecional ou bidirecional entre processos.

3. Criação e Uso:

- Named pipes são criados usando comandos como mkfifo no Unix.
- Podem ser abertos e usados como ficheiros regulares para leitura e escrita.

Named Pipes

```
#define PIPE_NAME "meu_pipe"

int main() {
    pid_t pid;
    char buf[100];

    mkfifo(PIPE_NAME, 0644);

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

    if (pid == 0) { // Processo filho
        int fd = open(PIPE_NAME, O_RDONLY);
        read(fd, buf, sizeof(buf));
        printf("Filho recebeu: %s\n", buf);
        close(fd);
    } else { // Processo pai
        int fd = open(PIPE_NAME, O_WRONLY);
        write(fd, "Olá, filho!", 12);
        close(fd);
        wait(NULL);
    }

    unlink(PIPE_NAME);

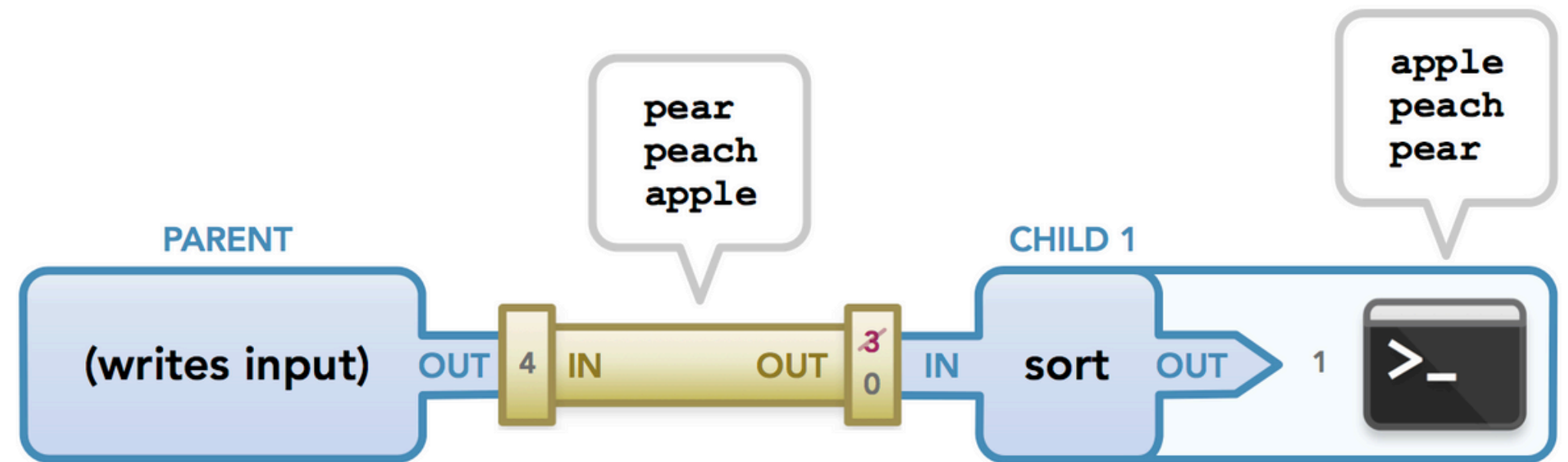
    return 0;
}
```

-->

Named Pipes

- Escrita no Named Pipe:
 - Quando múltiplos processos escrevem no mesmo named pipe, os dados são escritos de forma sequencial no pipe.
- Leitura do Named Pipe:
 - Um processo que lê do named pipe receberá os dados na ordem em que foram escritos.
 - Se múltiplos processos estiverem lendo do pipe, a ordem de leitura pode não ser garantida.
- Considerações:
 - Concorrência: Quando múltiplos processos escrevem no mesmo named pipe, o sistema operativo gerencia a concorrência, mas a ordem de escrita pode não ser garantida.
 - Sincronização: Em aplicações mais complexas, pode ser necessário usar mecanismos adicionais de sincronização (como semáforos) para garantir a ordem e integridade dos dados.

dup e dup2



dup e dup2

dup

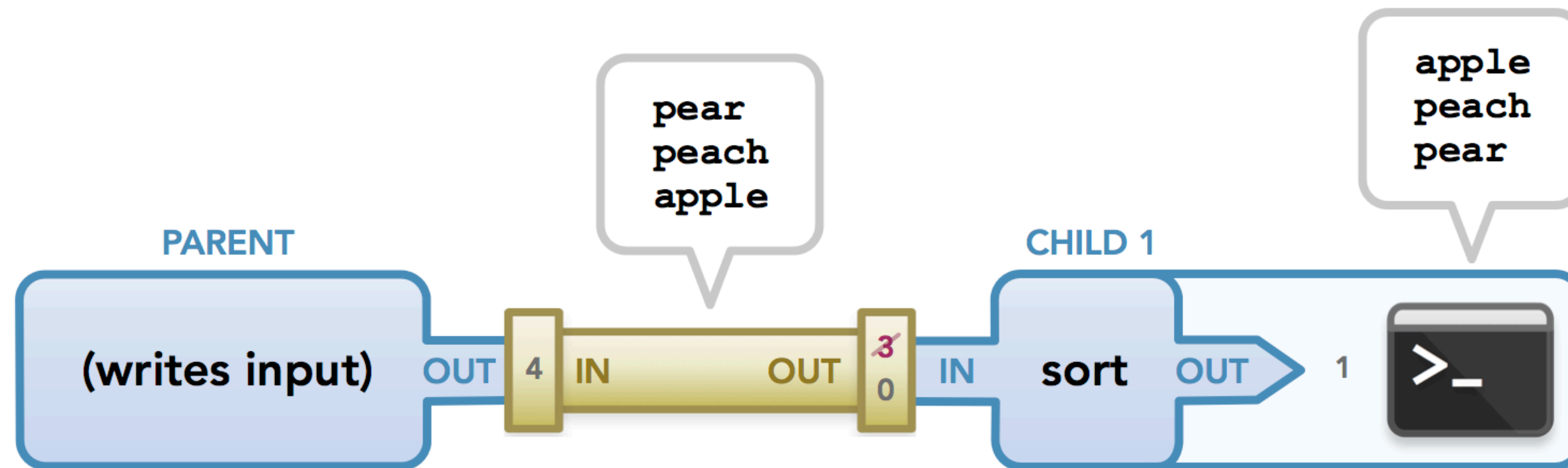
- Descrição: A função dup duplica um *file descriptor* e cria uma nova entrada na tabela de file descriptors que aponta para o mesmo ficheiro ou recurso.
- Sintaxe: `int dup(int oldfd);`
- Parâmetros:
 - oldfd: O file descriptor existente que será duplicado.
- Retorno: Retorna um novo descritor que é uma cópia do oldfd. Em caso de erro, retorna -1 e define errno.

dup e dup2

dup2

- Descrição: A função dup2 duplica um descritor, mas permite especificar o valor do novo descritor.
- Sintaxe: `int dup2(int oldfd, int newfd);`
- Parâmetros:
 - oldfd: O descritor existente que será duplicado.
 - newfd: O valor do novo descritor. Se newfd já estiver aberto, ele será fechado antes de ser reutilizado.
- Retorno: Retorna newfd em caso de sucesso. Em caso de erro, retorna -1 e define errno.

dup e dup2



```

int main(int argc, char *argv[]) {
    int fds[2];                // um array que vai conter dois descritores de ficheiros
    pipe(fds);                 // preenche fds com dois descritores de ficheiros
    pid_t pid = fork();        // cria um processo filho que é uma cópia do processo pai

    if (pid == 0) {            // se pid == 0, então estamos no processo filho
        dup2(fds[0], STDIN_FILENO); // fds[0] (a extremidade de leitura do pipe) passa os seus dados para o
        // descriptor de ficheiro 0
        close(fds[0]);          // o descriptor de ficheiro já não é necessário no filho, uma vez que o
        // stdin é uma cópia
        close(fds[1]);          // descriptor de ficheiro não utilizado no filho
        execlp("sort", "sort", NULL); // executa o comando sort
        exit(1); // se exec falhar, termina o processo filho
    }

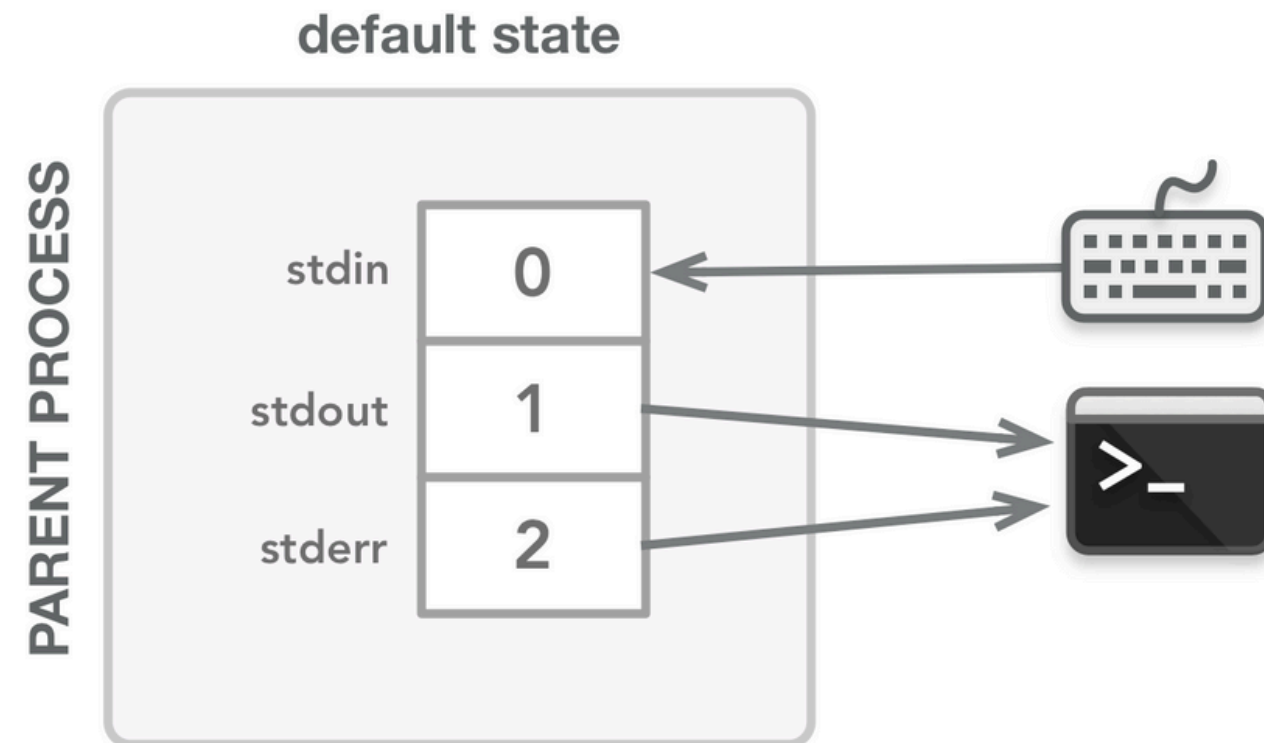
    // Processo pai
    close(fds[0]);              // descriptor de ficheiro não utilizado no pai
    const char *words[] = {"pera", "pêssego", "maçã"};
    // escreve a entrada no descriptor de ficheiro de escrita para que possa ser lida pelo filho:
    size_t numwords = sizeof(words)/sizeof(words[0]);
    for (size_t i = 0; i < numwords; i++) {
        dprintf(fds[1], "%s\n", words[i]);
    }

    // envia EOF para que o filho possa continuar (o filho bloqueia até que toda a entrada tenha sido
    // processada):
    close(fds[1]);

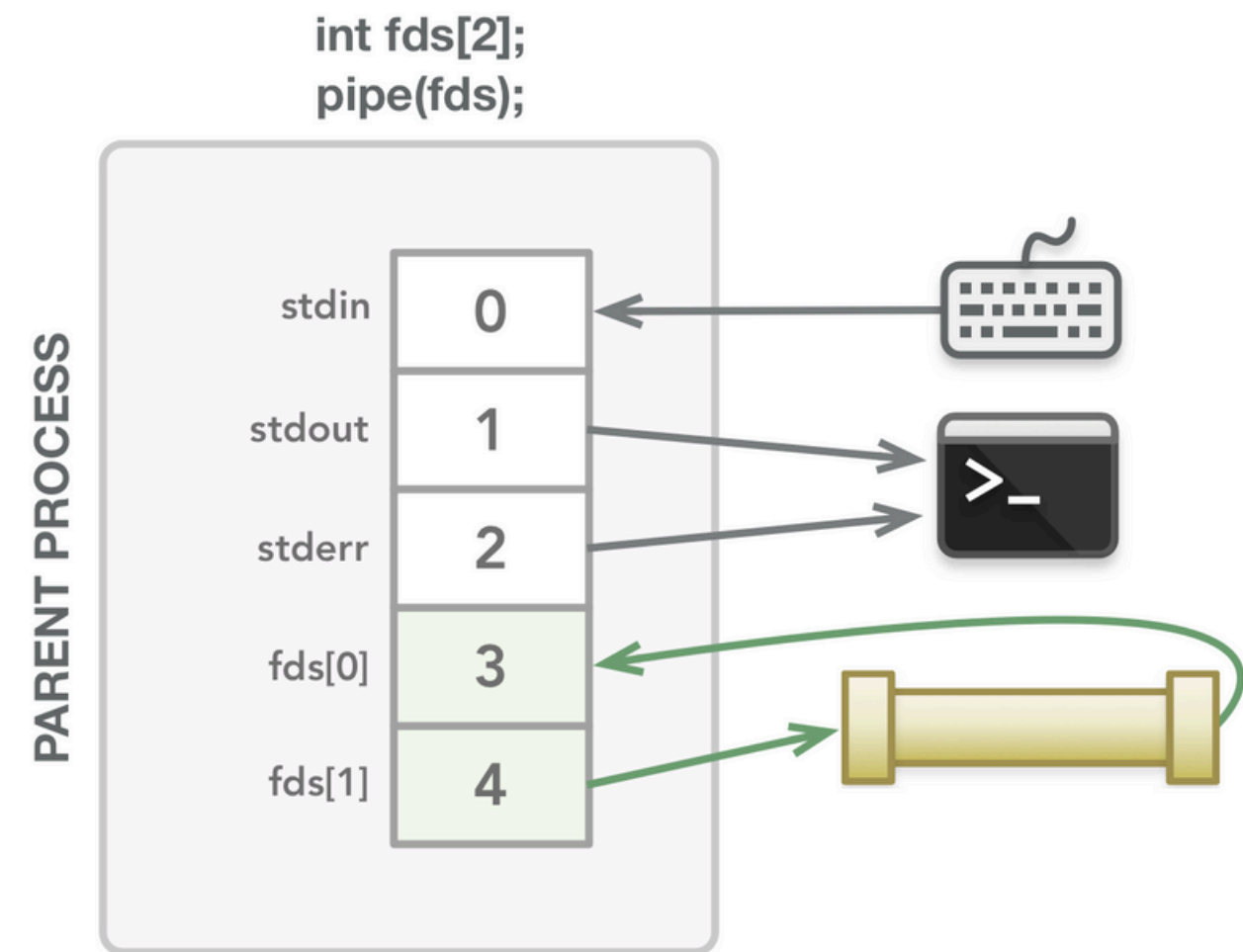
    wait(NULL); // espera que o filho termine antes de sair
    exit(EXIT_SUCCESS);
}

```


dup e dup2

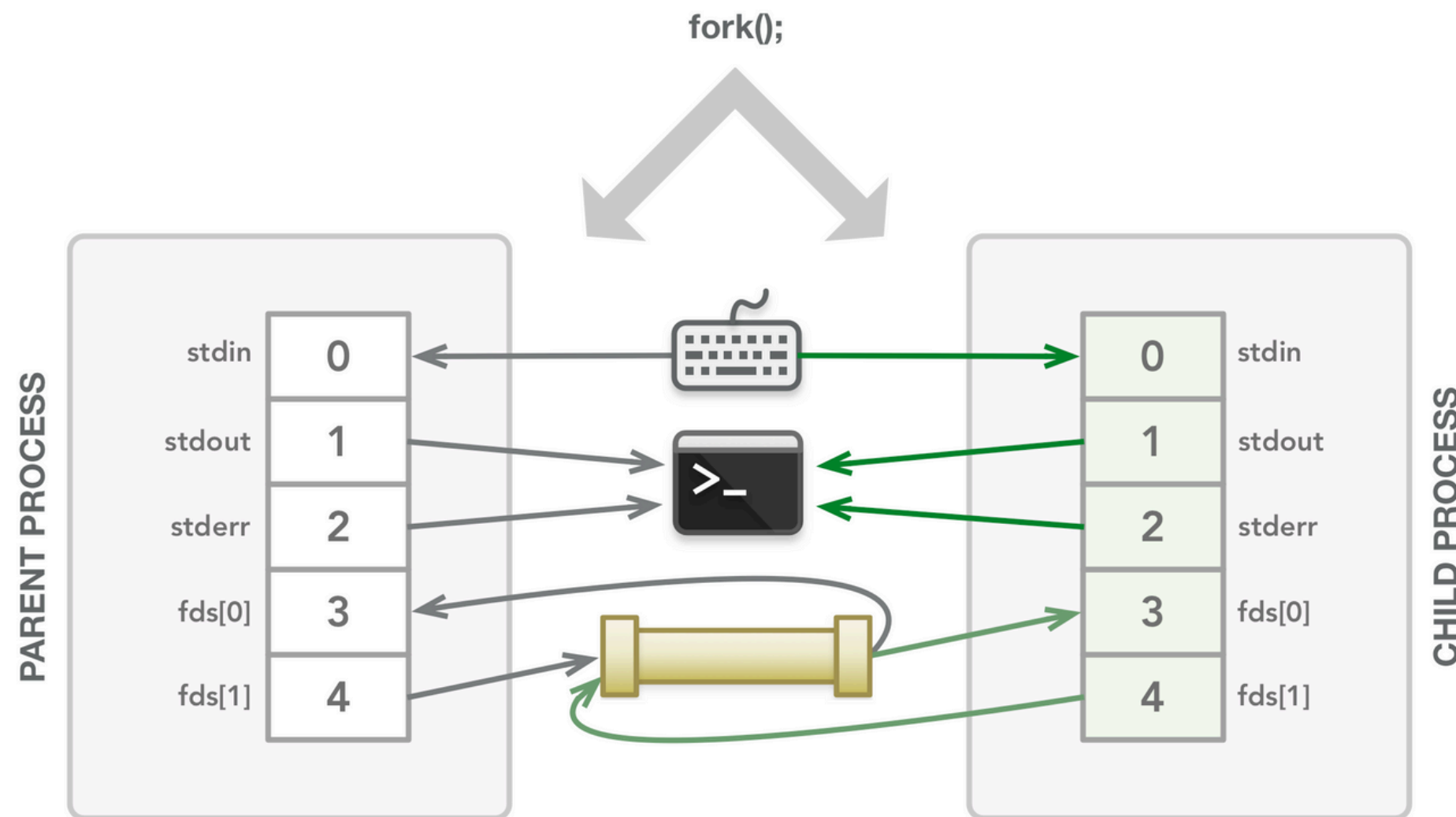


Quando o programa é iniciado, o processo principal é criado com os fluxos predefinidos configurados na sua tabela de descritores de ficheiros. As setas mostram o fluxo de dados: o stdin recebe a entrada do teclado, e o stdout e o stderr enviam a saída para o terminal.

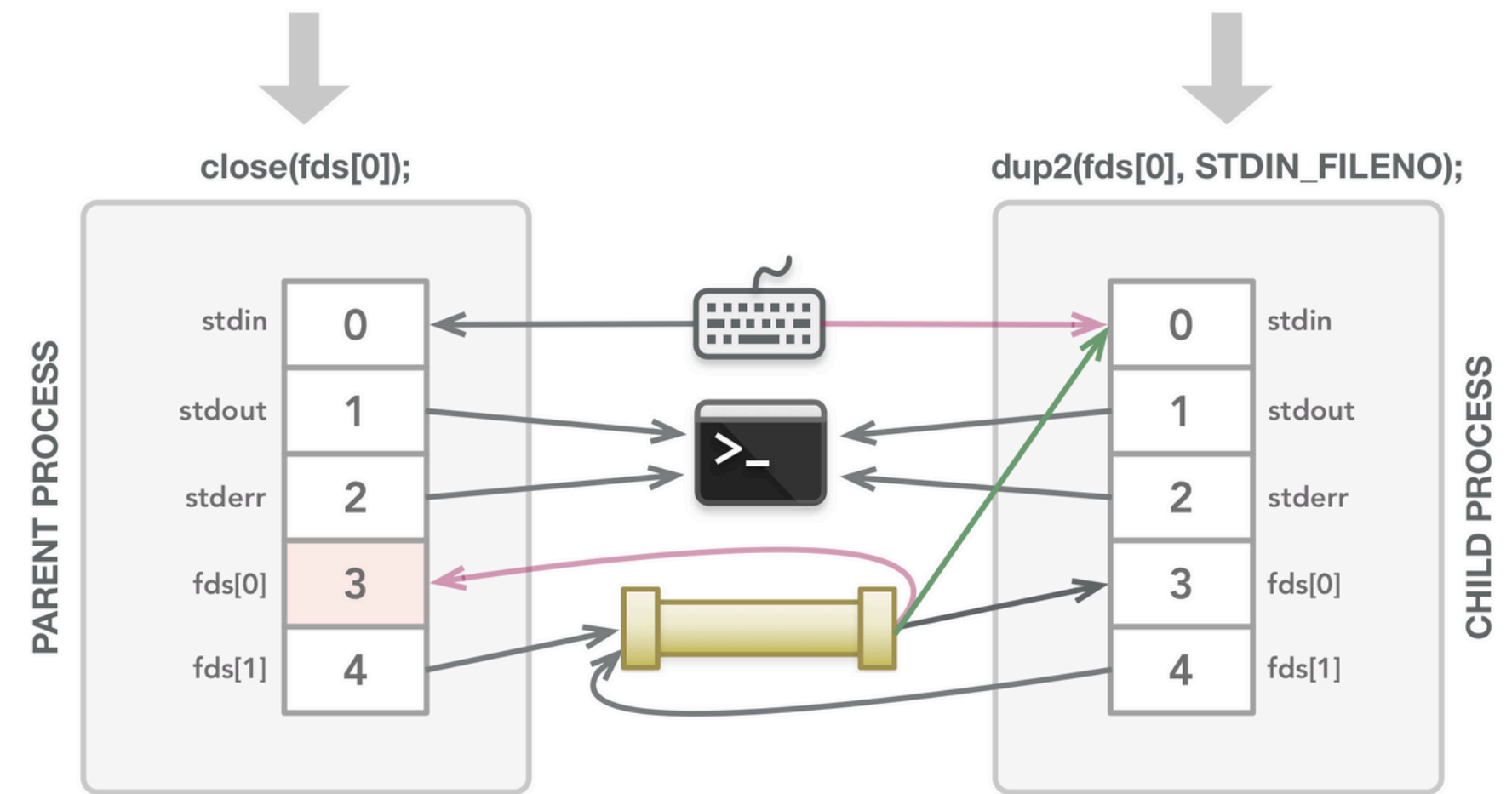


A chamada `pipe()` encontra os dois próximos descritores de ficheiros disponíveis e associa cada um à extremidade apropriada do pipe criado. Neste caso, um processo pode ler através do descritor 3 e escrever através do descritor 4.

dup e dup2

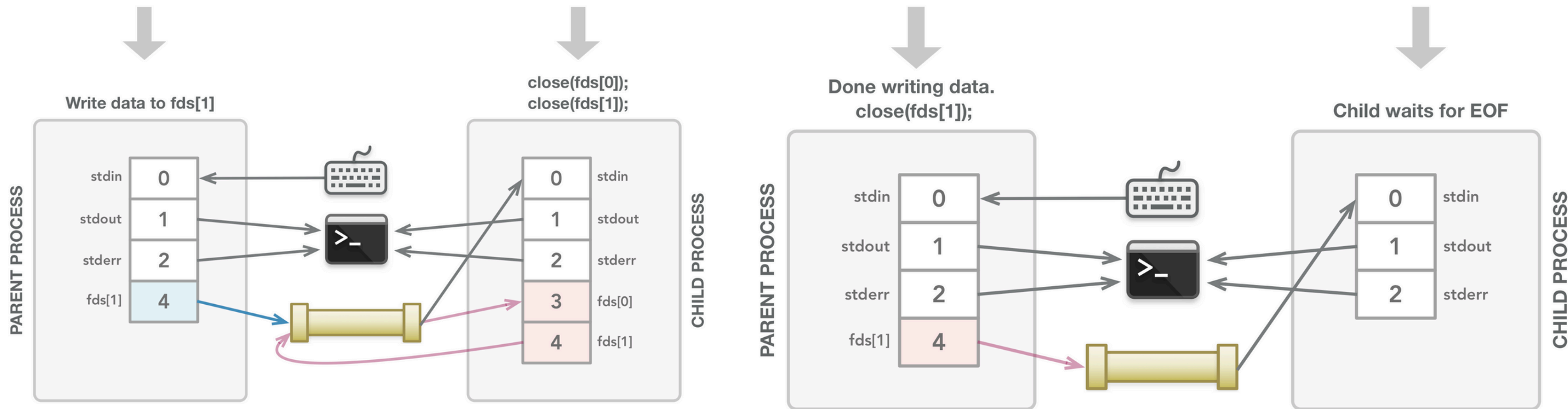


A chamada `fork()` cria o processo filho, que é uma cópia da memória e da tabela de descritores de ficheiros do processo pai naquele momento. Os ficheiros associados aos descritores de ficheiros do pai são os mesmos ficheiros associados aos descritores de ficheiros do filho.



O processo pai fecha o descritor de ficheiro que não necessita. O processo filho chama `dup2()` para que o seu `stdin` seja uma cópia de `fds[0]`, fechando primeiro o descritor de ficheiro `fds[0]`.

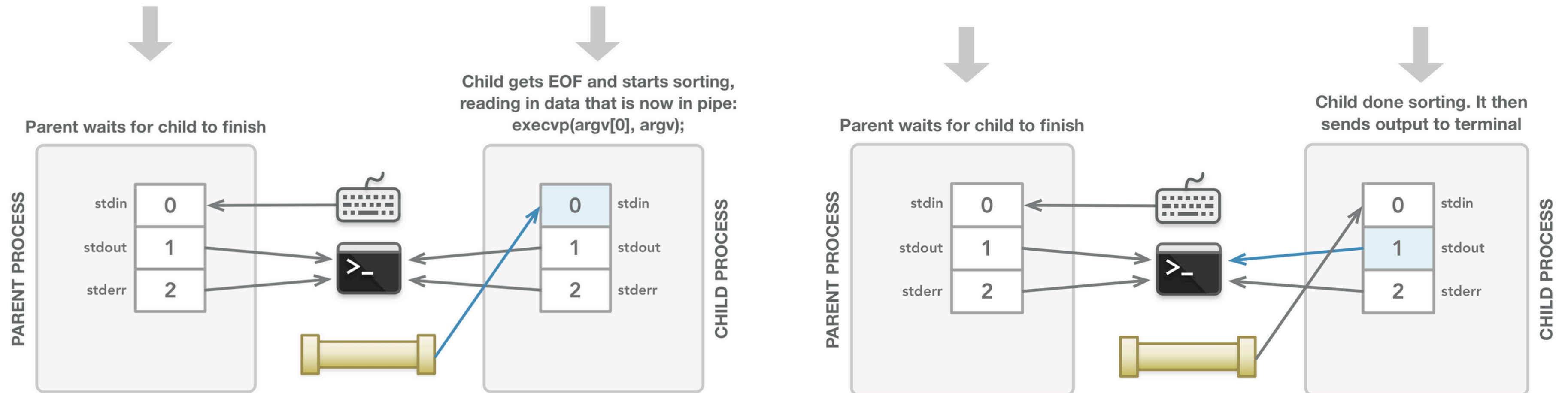
dup e dup2



O processo pai escreve dados na extremidade de escrita do pipe. O processo filho fecha os descritores de ficheiros que não necessita.

Após escrever todos os dados, o processo pai fecha `fds[1]` para informar o processo filho de que todos os dados foram enviados.

dup e dup2



O processo filho executa o comando na entrada recebida.

A saída ordenada é enviada para o terminal, e o processo filho envia um sinal ao terminar, permitindo então que o processo pai finalize.



Tente e Aprenda

Hora da Atividade

Ficha 6 – Exercícios 1, 2, 7, 8



**E por hoje
terminamos!**