- IPC → *Inter-process Comunication*
  - Nome dado a coleção de primitivas utilizadas para comunicação entre processos
    - Popularizado com o Unix System V
- Arquivos regulares podem ser vistos como um mecanismo de IPC, porém com limitações
  - Eficiência → limitada pela velocidade de acesso ao meio de armazenamento
  - Capacidade → limitada pelo tamanho máximo de arquivo e espaço livre no meio de armazenamento
- Frequentemente, o uso do termo IPC, exclui as comunicações indiretas através de arquivos
  - Ex.: pipes, sinais, mensagems, mem. compartilhada

Pipes

- Pipe → canal unidirecional entre dois processos
  - Forma mais antiga de IPC no UNIX, juntamente com Sinais
- Pontos positivos
  - Permite a comunicação eficiente de qualquer volume de informação entre dois processos
  - Provê sincronização automática entre os processos
- Limitações
  - Unidirecional
  - Não há separação entre mensagens
  - Exige que os processos sejam relacionados por um antecessor comum, que é quem cria o pipe

Usos de Pipes

- Comunicação entre pai e filho
  - Pipe é criado por um processo que chama fork() e o utiliza para comunica-se com seu processo filho
- Comunicação entre processos "irmãos"
  - Pipe criado por um processo que chama fork() duas vezes, sendo usado para comunicação entre os dois processos filhos criados
  - Implementação do operador "|" no shell
    - Conecta a saída padrão de um processo com a entrada padrão de outro

Usos de Pipes

- Ex.:

  ```
  $ cat nomes | sort
  ```

- Ambos os comandos, cat e sort, executam concorrentemente
  - Processo cat escreve o conteúdo que é lido por sort
  - O shell é o ancestral comum que cria o pipe!
- O pipe automaticamente bufferiza o conteúdo
- Sincronização implícita
  - buffer cheio → kernel suspende o processo escritor
  - buffer vazio → kernel suspende o processo leitor

APIs para Pipes

- Baixo nível → descritores de arquivos
  - Após criado, pipe é manipulado pelas operações padrão read(), write() e close()
  - Utilizado com fork() ou fork()+exec()
    - Não há execução automática de outro programa
- Alto nível → stream (FILE*)
  - Pipe criado e terminado por chamadas diferenciadas (popen() e pclose()) e usado através da API de streams (fread(), fwrite(), etc.)
  - Mais fácil de usar, porém menos flexível
    - Programa executado pelo filho é especificado na criação do pipe, juntamente com a direção da comunicação

Função pipe()

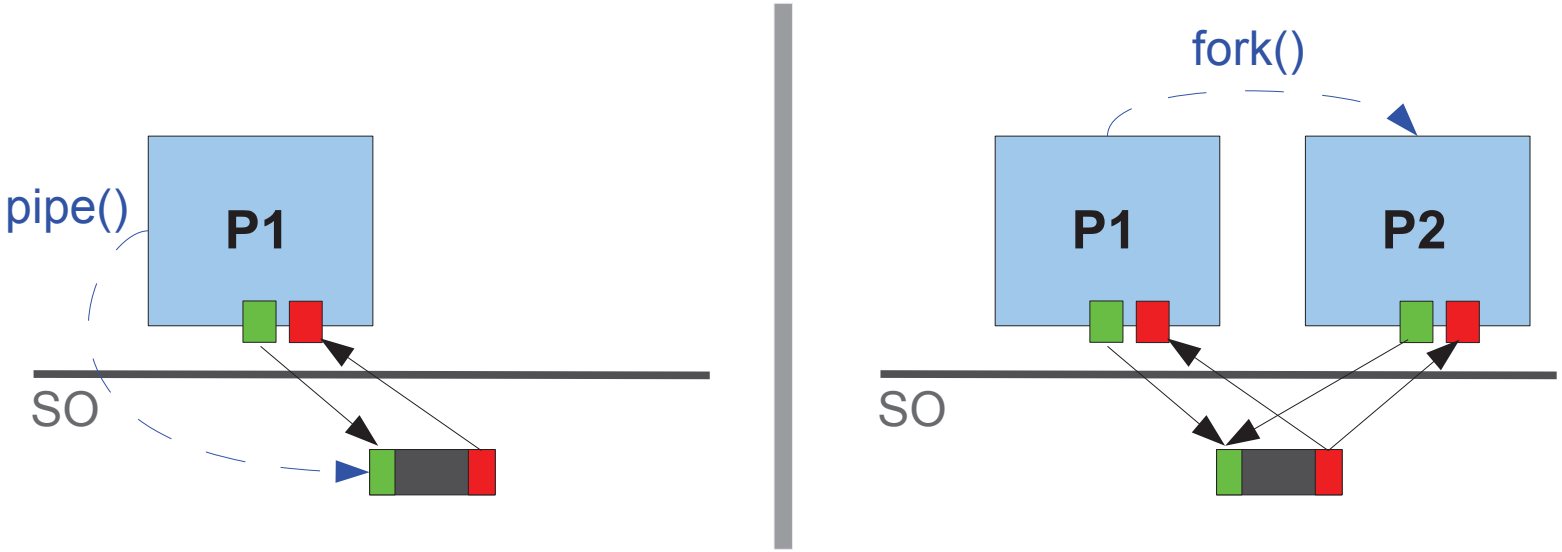
API

```
int pipe(int pfd[2]);
```

- Descrição
  - Cria um pipe, gravando em pfd os descritores de arquivo correspondentes as duas extremidades
    - pfd[0] → conterá a extremidade de leitura
    - pfd[1] → conterá a extremidade de escrita
  - Retorna 0 em caso de sucesso e -1 em caso de erro
- Dentro do SO, os descritores retornados estão conectados
  - O que for escrito para fd[1] poderá ser lido em fd[0]
- Normalmente, fork() é chamada após pipe()

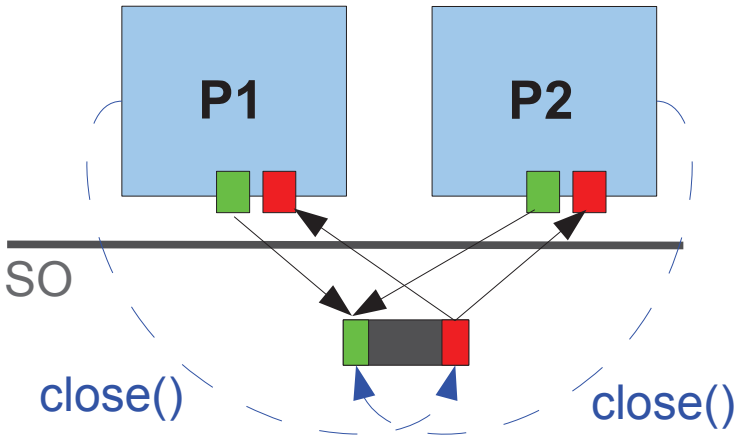
Criando um PIPE

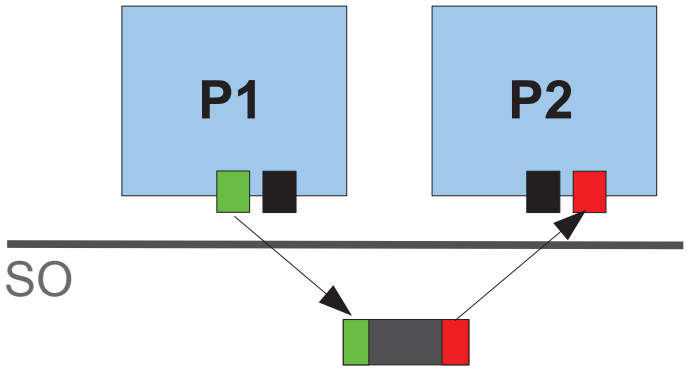
- Normalmente, pipe é aberto antes de um fork()
  - Descritores de arquivo do pipe serão duplicados e compartilhados entre pai e filho
  - Ambos os processos podem escrever no pipe ou ler do pipe



Criando um PIPE

- Que processo recebe é imprevisível!
- Para um comportamento novamente previsível:
  - um processo fecha sua entrada
  - o outro, sua saída
  - Resultado → um pipe simples novamente
- Quando escritor termina, fecha sua extremidade
  - Leitor recebe 0 (zero) no próximo read





Exemplo com pipe() e fork()

- *Ex.: ex-pipe-fork.c*

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

#define MAXLINE 4096

int main(void) {
    int     n, fd[2];
    char    line[MAXLINE];
    if (pipe(fd)<0) {perror("pipe()");exit(EXIT_FAILURE);}

    switch(fork()) { /* trat. erro parcialmente omitido */
        case -1:
             perror("fork()"); exit(EXIT_FAILURE);
        case 0:
            close(fd[0]); write(fd[1],"hello world\n",12);
        default:
            close(fd[1]); n = read(fd[0], line, MAXLINE);
            write(STDOUT_FILENO, line, n);
    }
    exit(EXIT_SUCCESS);
}
```

Protocolos sobre PIPEs

- Devido ao buffer, o conteúdo escrito no pipe pode não ser percebido imediatamente pelo leitor
  - Necessário incluir terminador NULL (\0) ou nova linha (\n) ou fechar a extremidade de leitura para forçar o SO a fazer o flush do conteúdo do buffer interno
- Para comunicação bidirecional cria-se dois pipes
- Não há preservação implícita da separação entre múltiplas mensagens escritas para um pipe
  - Aplicação deve definir um protocolo, de forma que leitora possa deduzir onde cada mensagem inicia e termina

Protocolos sobre PIPEs

- Exemplos de protocolos (esqueletos)
  - Predefinir que todas as mensagens terão tamanho fixo e sem ler a mesma quantidade de bytes
  - Enviar o tamanho a mensagem em bytes, antes de enviar a mensagem, seguida de \0 para forçar flush
  - Predefinir um marcador que será o separador de mensagens e executar a leitura até encontrar o próximo marcador
- Qualquer que seja a estratégia, é conveniente incluir após o termino da mensagem um caractere '\0' ou '\n' para forçar a entrega da mensagem.

Combinando pipe() e exec()

- Pipe funciona naturalmente após um fork() pois ambos os processos sabem os descritores do pipe
- E se um processo substitui seu programa com exec(), como ele saberá o descritor?
- **Estratégia** → re-atribuir descritores padrão!
  - Normalmente processos recebem de stdin(0) e escrevem para stdout(1)
  - Se, antes do exec(), o descritor do pipe (>=3) puder ser transformado...
    - em 0 → ao ler de sua entrada padrão, o novo programa estará lendo do pipe
    - em 1 → ao escrever para sua saída padrão, o novo programa estará escrevendo para o pipe

Combinando pipe() e exec()

- Ex.: considere o comando shell abaixo

```
$  ls | wc
```

- Informações previamente conhecidas
  - O ls normalmente escreve para sua saída padrão, ou seja, seu descritor de arquivo 1
  - O wc normalmente lê de sua entrada padrão, ou seja, de seu descritor de arquivo 0
  - O operador "|" representa um pipe conectando a saída do primeiro processo (1) à entrada do segundo (0)

Combinando pipe() e exec()

- Linhas gerais da implementação em C
  - Cria-se um pipe e chama-se o fork()
  - Processo pai...
    - Associa seu descritor 1 à extremidade de escrita do pipe
    - A seguir, chama exec() para carregar o programa "ls"
      - O ls escreve normalmente para sua saída padrão
  - Processo filho...
    - Associa seu descritor 0 à extremidade de leitura do pipe
    - A seguir, chama exec() para carregar o programa "wc"
      - O wc lê normalmente de sua entrada padrão
  - O que **ls** escrever aparecerá com entrada para **wc**!

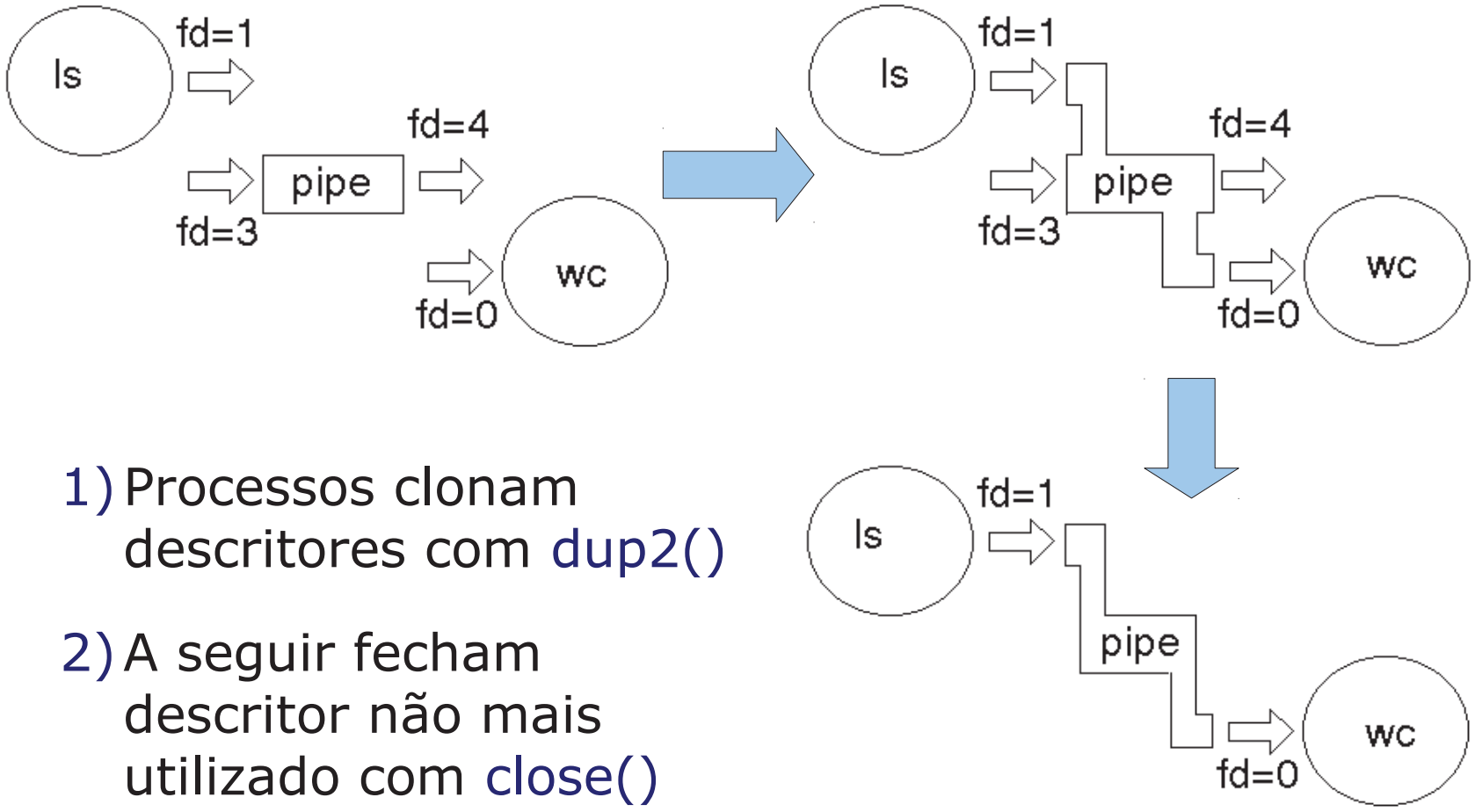
Funções dup() e dup2()

API

```
int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

- Descrição
  - Executam a clonagem de um descritor de arquivo
    - dup() → clona o descritor de arquivo informado para o primeiro descritor livre do processo (não usado)
    - dup2() → fecha o arquivo correspondente a newfd e a seguir, copia oldfd para newfd
  - Ambas retornam o novo descritor em caso de sucesso ou -1 em caso de erro
- Após o retorno dessas funções o descritor antigo o no recém criado clone referenciam exatamente o mesmo arquivo aberto do processo

Reposicionando Descritores



1) Processos clonam descritores com dup2()

2) A seguir fecham descritor não mais utilizado com close()

Exemplo com pipe() + exec()

- *Ex.: ex-pipe-exec.c*

```
/* includes omitidos, consulte as man pages */
/* tratamento de erro parcialmente omitido */
int main() {
    int pfd[2];
    if (pipe(pfd) != 0) perror("pipe()");
    else
        switch (fork()) {
        case 0:
            close(pfd[1]);
            dup2(pfd[0], STDIN_FILENO); close(pfd[0]);
            execlp("wc", "wc", NULL);
            perror("exec 'wc'"); break;
        default:
            close(pfd[0]);
            dup2(pfd[1], STDOUT_FILENO); close(pfd[1]);
            execlp("ls", "ls", NULL);
            perror("exec 'ls'"); break;
        }
    exit(EXIT_FAILURE);
}
```

FIFOs

- Arquivo especial do tipo ***F**irst-**I**n, **F**irst-**O**ut*
- Também chamados de Pipes nomeados
  - Uma vez abertos, funcionam exatamente como Pipes convencionais
- São identificados por um nome existente no sistema de arquivos
  - Permite criar canais de comunicação entre processos não relacionados
  - Podem ser persistentes: existem até serem removidos

- Por padrão, usam comunicação bloqueante
- Em vários Unix são unidirecionais
- Podem ser criados via shell com o comando mkfifo

Ex.:

```
$  mkfifo meuPipe
$  ls -la meuPipe
prw-r--r-- 1 john users 0 Oct 09 11:42 meuPipe
```

Tamanho zero!
Virtualmente não ocupa espaço

Arquivo especial do tipo **pipe**

Pode ser necessário ajustar as **permissões para leitura e escrita**

Criando FIFOs com mkfifo()

API

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

- Descrição
  - Cria um um arquivo especial fifo
  - Parâmetros
    - pathname → caminho, absoluto ou relativo, para o arquivo do fifo no sistema de arquivos
    - mode → permissões que serão atribuídas ao fifo
      - Equivalente ao terceiro parâmetro do open()
  - Retorna 0 em caso de sucesso ou -1 em caso de erro

Ex.:

```
if (mkfifo("/tmp/meuFifo",
           S_IRGRP|S_IWGRP|S_IRUSR|S_IWUSR) != 0)
   { /* tratamento de erro...  */ }
```

E/S com FIFOs

- Abre-se FIFO com a função open(), como um arquivo convencional
  - FIFO deve ter sido previamente criado (C ou shell)
  - escritores abrem somente-escrita
  - leitores abrem somente-leitura
- write() acrescenta informações ao final do FIFO
- read() lê próximos bytes do início do FIFO
- close() fecha o FIFO (não remove)
  - Para remover FIFO:
    - Em C, usa-se unlink()
    - No shell, pode-se usar o rm ou unlink

Sincronização, SIGPIPE e EOF

- Ausência de leitores
  - Se tentarmos escrever em um FIFO que não está atualmente aberto para leitura por nenhum outro processo, o sinal SIGPIPE é gerado
    - Tratamento padrão desse sinal é matar o processo!
- Ausência de escritores
  - Quando o último escritor de um FIFO fecha o canal, um fim de arquivo é gerado para o leitor do FIFO
- O mesmo vale para pipes
- open() sobre um FIFO normalmente é bloqueado até que haja pelo menos um leitor e um escritor

Leituras Complementares

- STEVENS, W.R. Advanced Programming in the UNIX Environment. 2nd. Ed., Addison Wesley, 2005.
- Man pages
  - Referentes a cada uma das funções abordadas
  - Overview de Pipes e FIFOs
    - man 7 pipe
    - man 7 fifo
- Livro: Advanced Linux Programming
  http://www.advancedlinuxprogramming.com/alp-folder
- Na web: System Software Unix IPC API
  http://jan.newmarch.name/ssw/ipc/unix.html