Modelo computacional

Nesta secção iremos abordar um conjunto básico de primitivas oferecidas pelo Linux para lidar com ficheiros, criar novos processos e suportar operações fundamentais de um Shell, nomeadamente redireccionamento de I/O standard e criação de pipelines de comandos.

Os detalhes das funções (argumentos, erros, ficheiros de include necessários, etc.) não serão aqui apresentados, recomendando-se a leitura dos manuais do Linux (`man <service_name>` ou `man 2 <service_name>`).

Em geral, em caso de erro as funções retornam -1, afetando a variável global `errno` com o código do erro.

A função `perror`,invocada de imediato, apresenta no *standard error* a descrição do erro.

Operações básicas sobre ficheiros

```
int open(const char *pathname, int flags [, mode_t mode ]);
```
Abre ou cria um ficheiro. Em caso de sucesso retorna o descritor de acesso ao ficheiro

```
int read(int fd, void *buf, size_t count);
```
Copia bytes de um ficheiro dado o descriptor. Retorna o número de bytes lidos

```
int write(int fd, void *buf, size_t count);
```
Escreve bytes para um ficheiro dado o descriptor. Retorna o número de bytes escritos.

```
int close(int fd);
```
Fecha o descritor passado por parâmetro

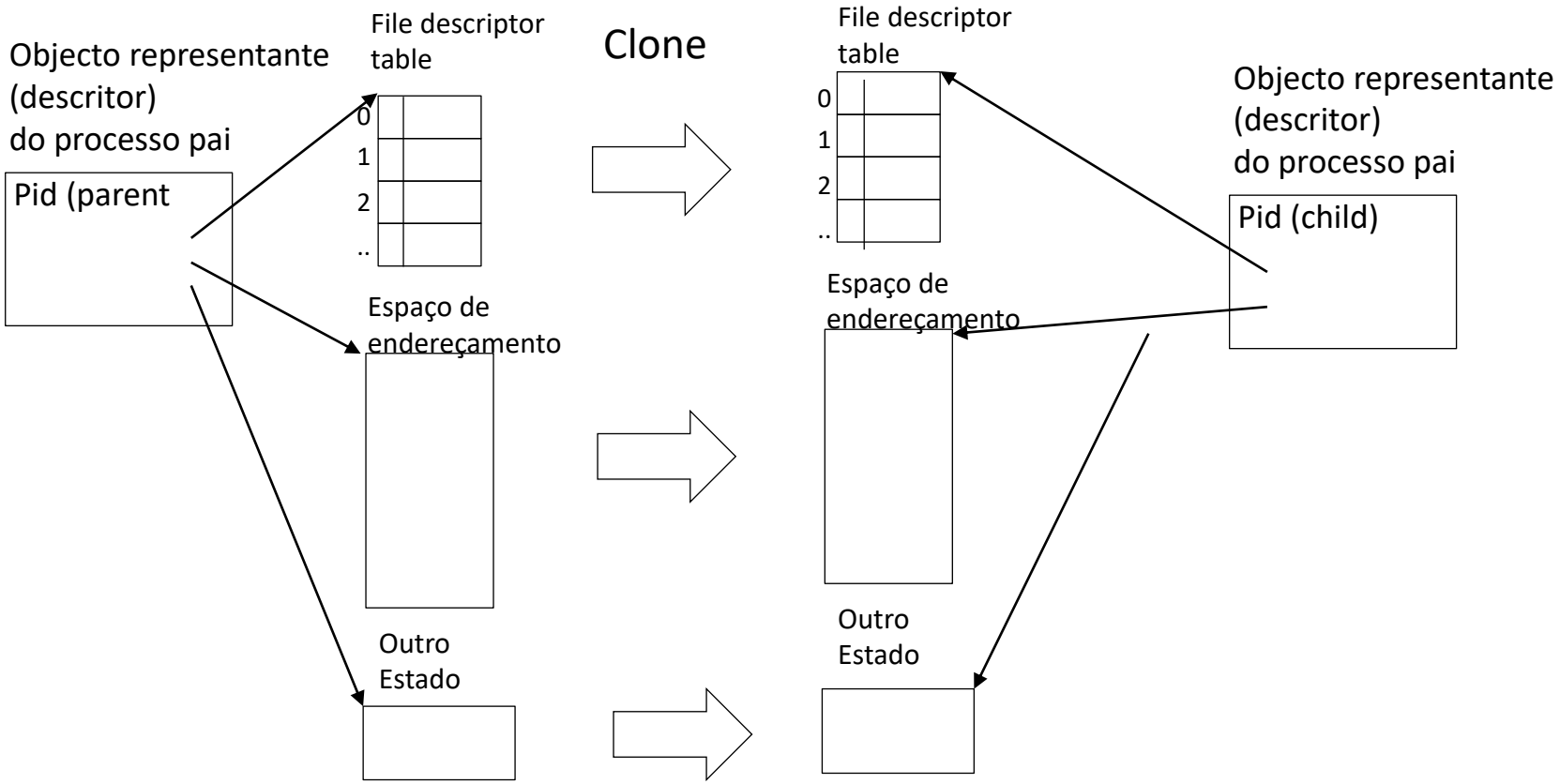
```
int dup(int fd);
```
Duplica o descritor passado por parâmetro na primeira entrada livre da tabela de descritores do processos. Ambos passam o referir o mesmo file object de acesso ao ficheiro (ver slide 'Descritores: herança e partilha'

```
int dup2(int fd_orig, int fd_dst);
```
O mesmo que dup, mas o descritor fd_orig é duplicado na entrada indicada em fd_dst. Caso exista o descritor destino, este é previamente fechado

Criação de processos 1 - fork

`pid_t` **`fork`**`(void);`

Cria um novo processo como um "clone" do processo criador. O novo processo executa-se a partir do retorno do fork. Retorna 0 para o filho e para o pai o pid do filho.



Criação de processos 2 – wait, waitpid

```
pid_t wait(int *wstatus);

pid_t waitpid(pid_t pid, int *wstatus, int options);
```

bloqueia o processo invocante até à terminação de um filho específico (`waitpid`) ou de qualquer filho (`wait`, `waitpid`). Informação sobre o estado de terminação do filho é depositada no inteiro referido por `wstatus`, se diferente de NULL

Retorna o *pid* do processo terminado.

Criação de processos 3 - exec

```
int execl(const char *pathname, const char *arg, ...  /* (char  *) NULL */);
int execlp(const char *file, const char *arg, ...  /* (char  *) NULL */);

int execv(const char *pathname, char *const argv[]);
int execvp(const char *file, char *const argv[]);
```

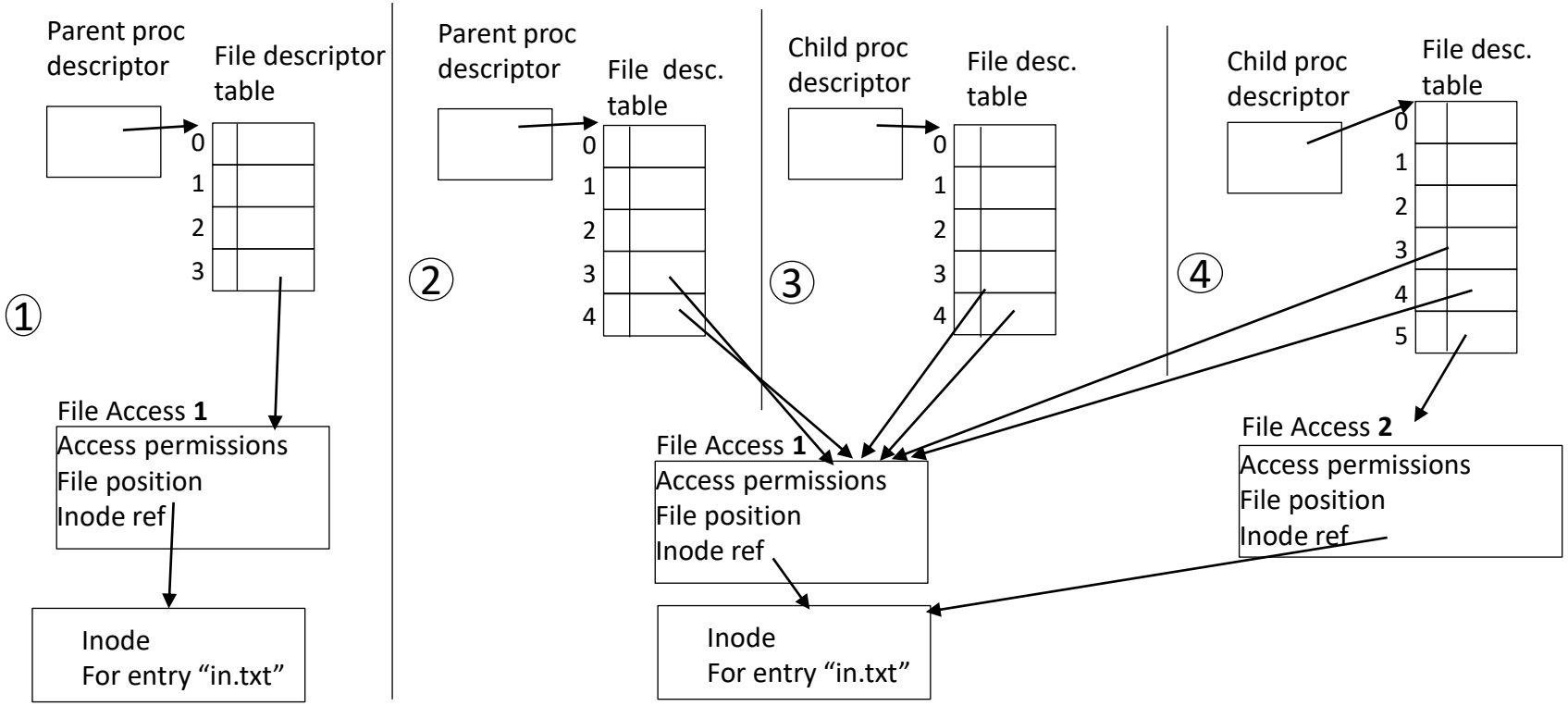
Modifica a imagem (programa a executar) do processo corrente. Usado em conjunto com `fork` para executar um comando num novo processo e `wait` (`waitpid`) para esperar pela sua terminação, como mostra o excerto de código abaixo, para executar o comando cat

```
int child_pid;
if ((child_pid  == fork()) == -1) {
      perror(“error on fork”);
}
else if (child_proc == 0) { // child process
      execlp("cat", "cat", “file.txt”, NULL);
      perror("error launching cat");
      exit(1);
}
else {  // parent
      waitpid(child_proc, NULL, 0);
}
```

Partilha de descritores

```
int fd1 = open("in.txt", O_RDONLY);          ①
int fd2 = dup(fd1);                          ②

if (fork() == 0) { // child                  ③
    fd1 = open("in.txt", O_RDONLY);          ④
}
```



Redireccionamento de I/O

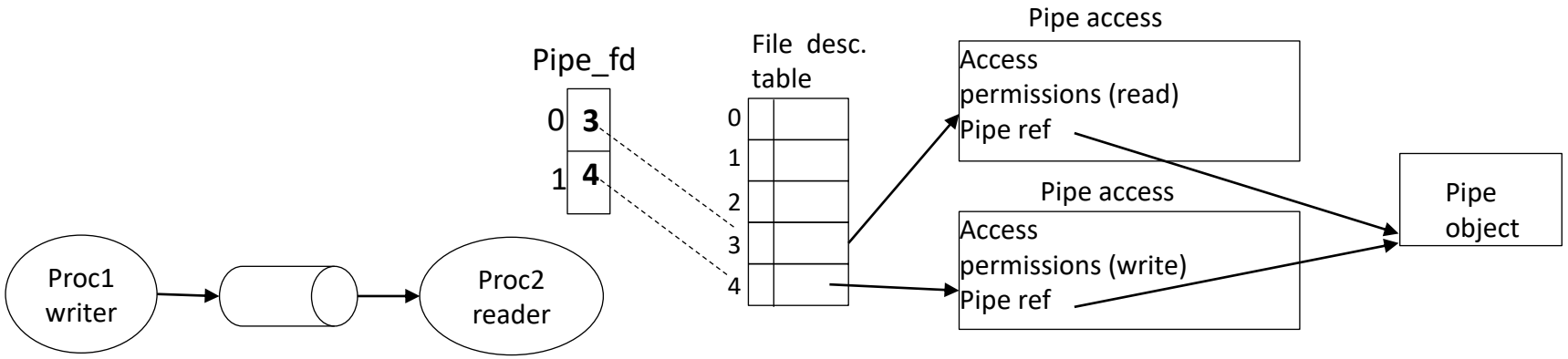
O excerto de código abaixo executa o comando `cat file.txt > out.txt` e espera pela sua terminação.

```
int child_pid;
if ((child_pid  == fork()) == -1) {
      perror(“error on fork”);
}
else if (child_proc == 0) { // child process
      int fd = open(“out.txt”, O_CREAT | O_WRONLY | O_TRUNC, 0644);
      if (fd > 0) {
          dup2(fd, STDOUT_FILENO /* 1 */);
          close(fd); // not needed anymore
          execlp("cat", "cat", “file.txt”, NULL);
      }
      perror("error launching cat");
      exit(1);
}
else {  // parent
      waitpid(child_proc, NULL, 0);
}
```

Comunicação através de pipes anónimos

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

Comunicação unidireccional entre processos hierarquicamente relacionados (pais e filhos ou entre 'irmãos'). A função de sistema pipe preenche o *array* passado com dois descritores de ficheiros: leitura do `pipe` (elemento de índice 0) e escrita no `pipe` (elemento de índice 1) . As leituras e escritas são realizadas usando as operações de transferência sobre ficheiros (`read`, `write`). A comunicação é devidamente sincronizada, pelo que a operação `write` pode bloquear o processo (thread) invocante, no caso do `pipe` se encontrar cheio e a operação `read` pode bloquear o processo (thread) invocante no caso do `pipe` se encontrar vazio. No caso de não existirem escritores (todos os descritores de escrita no `pipe` existentes terem sido fechados), e o pipe se encontra vazio a operação `read` retorna 0.
Até certa dimensão. O pipe garante que as escritas são atómicas até à dimensão especificada na constante PIPE_BUF (limits.h). Potencialmente isso permite usar um pipe com vários escritores e leitores, mas na utilização em pipelines de comandos a comunicação é sempre de 1 escritor para 1 leitor.



Sinais

Sinais são um mecanismo para enviar notificações assíncronas de diferentes tipos a processos (para alguns tipos com informação adicional). Podem ser vistos como interrupções por software. Por omissão, o envio de um sinal a um processo provoca a sua terminação, mas este comportamento pode ser alterando definindo um handler especifico para o sinal pretendido (usando a função signal). Sinais podem ser enviados através da função kill. Alguns sinais (os que representam exceções) são síncronos (Ex: `SIGSEGV`). O comando kill permite enviar via Shell sinais a processos. Já estão definidas constantes para representar handlers especiais que ignoram sinais (`SIG_IGN`) ou o tratamento por omissão (`SIG_DFL`). Para uma utilização fiável a API de sinais é mais complexa, por exemplo permitindo o bloqueio de certos sinais na execução de troços críticos de código.

```
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
int kill(pid_t pid, int sig);
```

Terminação condicional
na ocorrência de SIGINT

```
void sigint_handler(int signal) {
    if (…..)
        exit(1);
}

int main() {
     signal(SIGINT, sigint_handler);
     // …….
}
```

Tipo	Descrição
SIGINT	Sinal enviado pelo kernel via ^C
SIGCHLD	Enviado pelo kernel na terminação de processo filho
SIGKILL	Terminação (obrigatória)
SIGSEGV	Enviado pelo kernel na violação de espaço de endereçamento
SIGALRM	Notificação associada a uma temporização