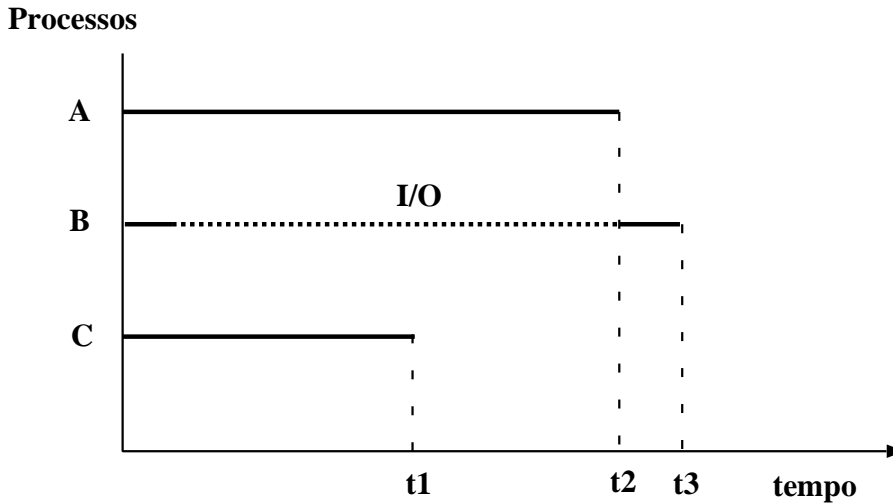


Processos

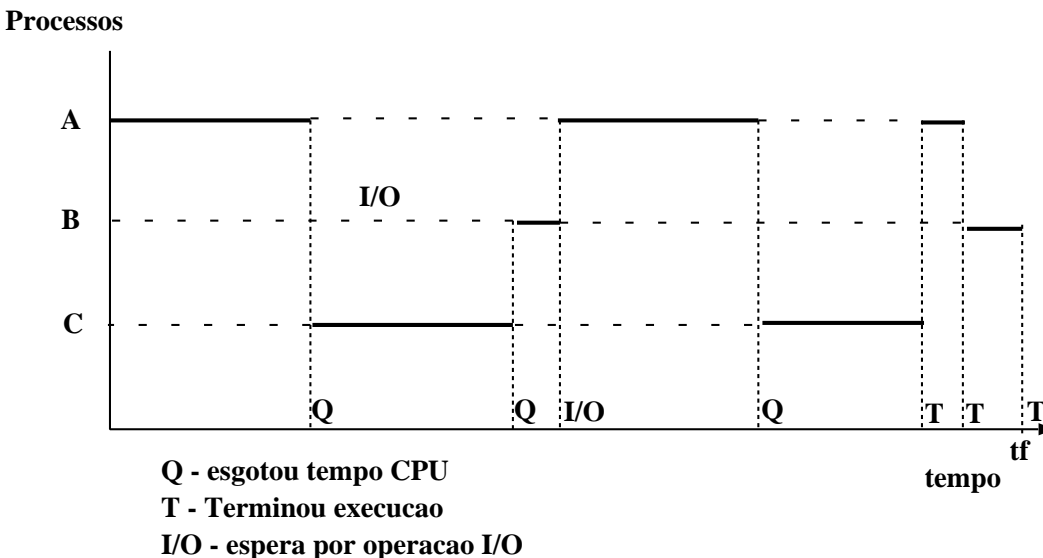
- Um SO executa uma multiplicidade de programas, em batch ou time-sharing, que se designam por:
processos ou **tarefas** (processes/tasks/jobs).
- **Processo** – é um programa em execução.
A execução de um processo é sequencial, no sentido em que num dado instante, apenas uma instrução do processo é executada.
- Um processo não tem apenas associado a si, o código de um programa, inclui também:
 - **program-counter** – que indica a próxima instrução a executar.
 - **pilha de execução** (stack) – com valores temporários (parâmetros de funções, endereços de retorno, etc.)
 - **região de dados** – com os valores das variáveis globais.

Execução de processos

Consideremos 3 processos cuja traçagem de execução, se executados um de cada vez até terminarem, se ilustra na figura:



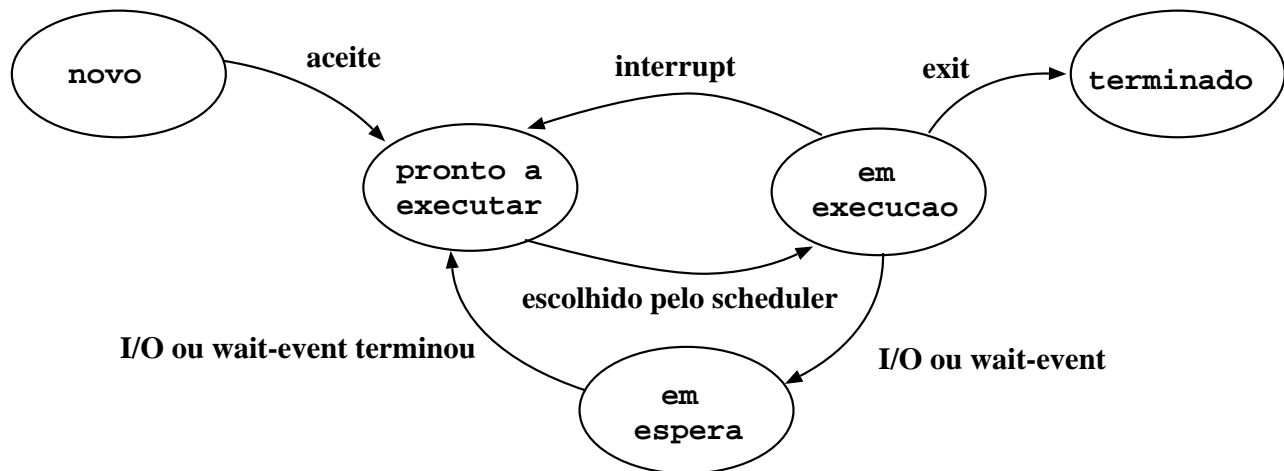
Suponha agora (2a. figura) que os processos são executados em time-sharing:



Observe que o tempo de execução, será menor quando ocorre time-sharing (i.e. $t_f < (t_1 + t_2 + t_3)$).

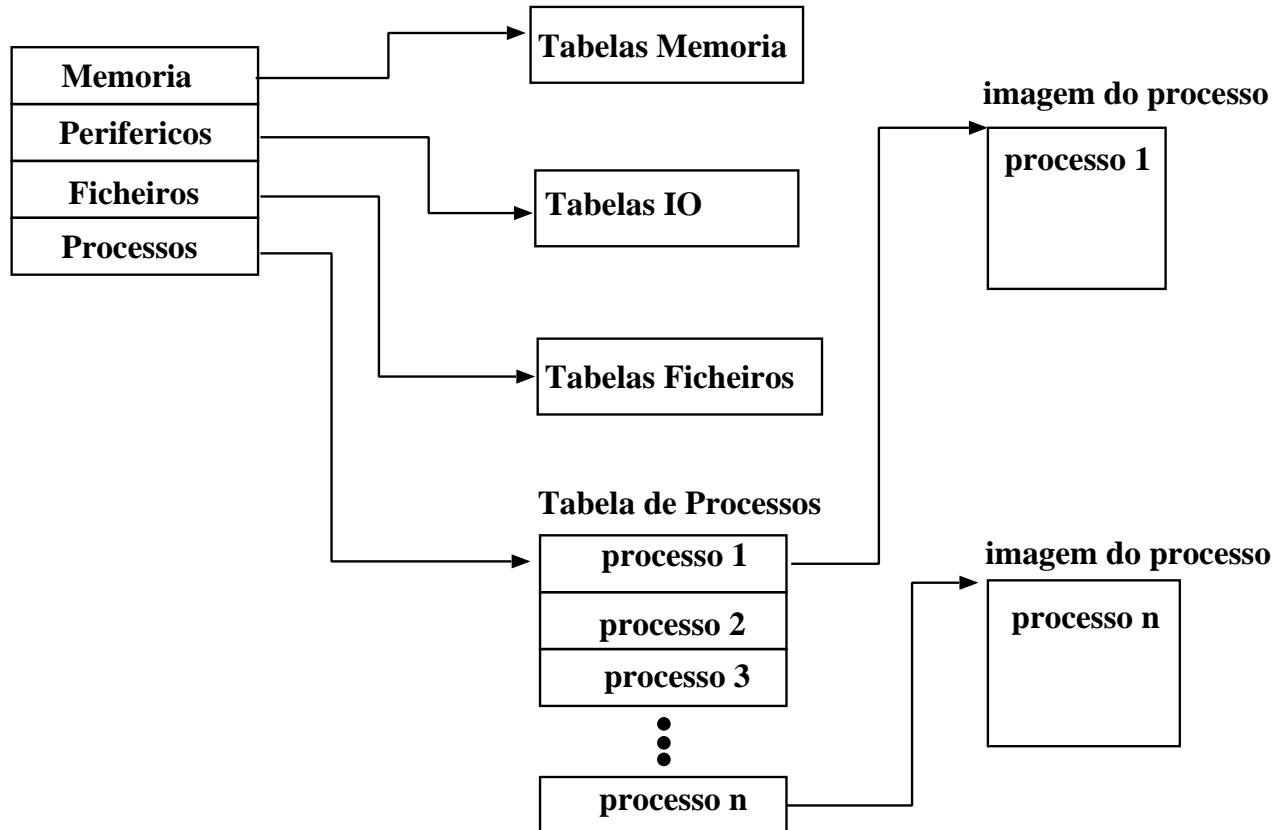
Estados de um Processo

- Um processo varia de estado durante a sua execução:
 - **novo** – o processo está a ser criado.
 - **em execução** – activo no CPU.
 - **em espera** – o processo está à espera de um evento externo.
 - **pronto a executar** – o processo está à espera de vez de CPU.
 - **terminado** – o processo terminou a execução.
- Transições possíveis entre os estados de um processo:



Estruturas de controlo do SO

O SO constroi e mantém tabelas com informação sobre cada entidade, processo e recurso do sistema, que gere.



- **tabelas de memória** – guardam informação sobre a memória principal e secundária associada a processos, assim como atributos de protecção e informação para gerir memória virtual.
- **tabelas de I/O** – permitem que o SO saiba se um dado periférico está livre ou não, qual a operação em curso num dado periférico e qual a zona de memória associada.
- **tabelas de ficheiros** – têm informação sobre os ficheiros existentes, a sua localização na memória secundária, o seu estado corrente e outros atributos.

Tabelas e atributos de processos

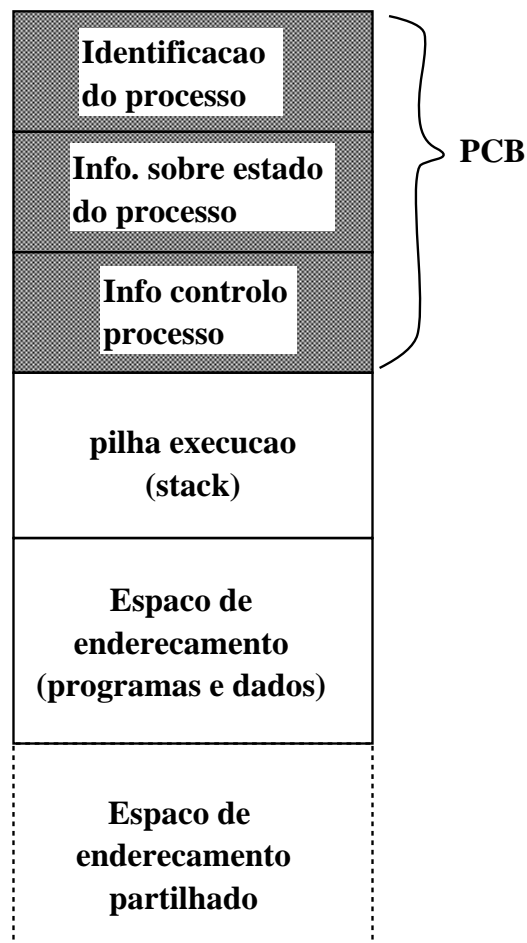
- **tabelas de processos** – permitem que o SO saiba localizar os processos correntes e quais os atributos que lhe estão associados.
- Associado a um processo (a sua imagem) temos:
 - um **espaço de endereçamento** que determina a execução do processo (memória onde estão partes do código executável e dados usados pelo processo).
 - um **contexto do processo** que define o ambiente de execução e determina o estado do processo. Estes atributos são normalmente referidos por **PCB** (Process Control Block).
- Informação típica num PCB:
 - **identificação do processo:**
 - *identificador do processo* (PID)
 - *identificador do processo que criou este* (processo-pai – PPID)
 - *identificador do utilizador dono do processo* (UID).
 - **estado do processo:**
 - *program counter*: próxima instrução a executar.
 - *registos do cpu*: acessíveis ao utilizador.
 - *status info*: inclui flags sobre interrupts (activos/inactivos), modo de execução (kernel/utilizador).
 - *códigos de condição*: resultados de operações lógicas e aritméticas (overflow...).
 - *apontador para a stack*. A stack é usada para guardar parâmetros quando da chamada de funções.

PCB (continuação)

- **informação de controlo:**

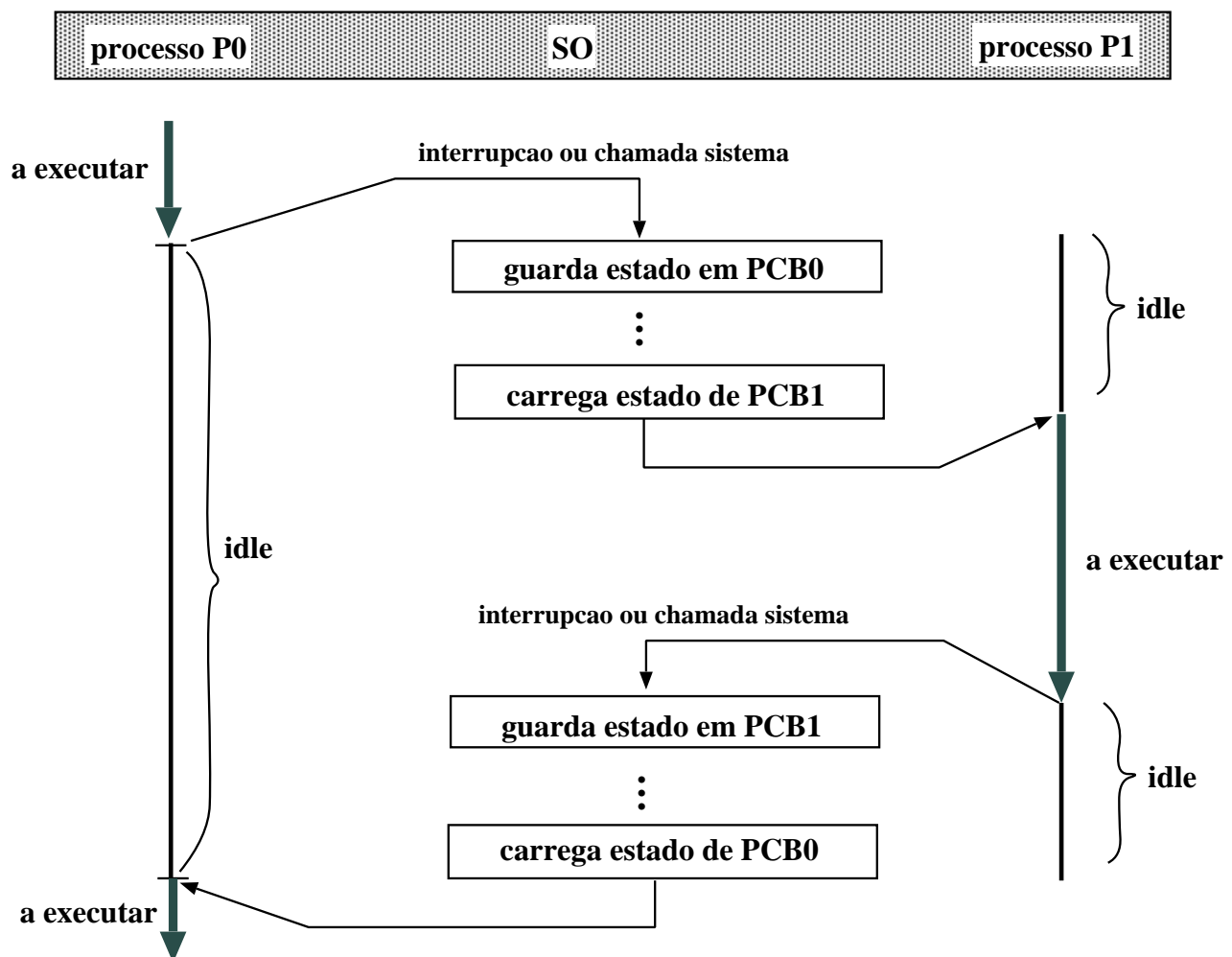
- *informação de scheduling*: estado do processo (a executar, pronto a executar, em espera, etc.), prioridade do processo.
- *privilégios do processo*
- *info. para gestão de memória*: páginas e segmentos de memória associados ao processo.
- *recursos associados*: ficheiros abertos, etc.

A estrutura da imagem de um processo na memória (virtual) está ilustrada na figura:



Troca de contexto (processos)

- Sempre que o CPU comuta de um processo para outro tem de guardar o estado do processo que sai (para poder ser retomado posteriormente) e carregar o estado do processo que entra.
- O estado do processo é guardado no PCB do processo.
- A troca de contexto é um *overhead*, i.e. o sistema não está a fazer nada de útil enquanto processa a troca.
- Ilustração do que ocorre numa troca de contexto:



Criação de processos

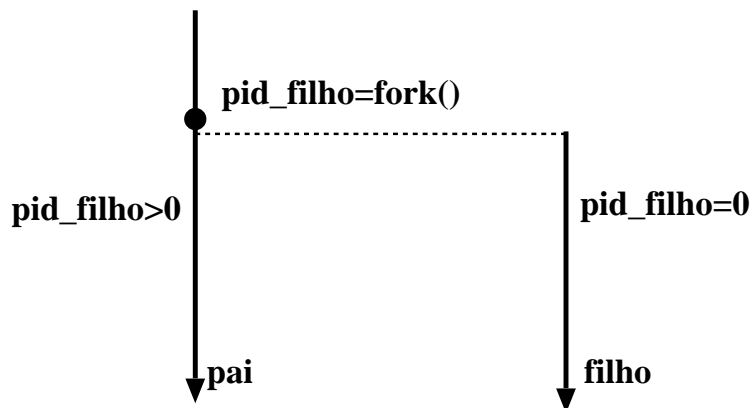
- Um processo pode criar novos processos, *processos-filho*, que por sua vez podem criar novos processos.
- **recursos:** o processo-filho é uma cópia da imagem do processo-pai e pode partilhar os recursos deste.
- **execução:**
 - Pai e filho executam concorrentemente, ou
 - Pai espera que os filhos terminem.
- **espaço de endereçamento:**
 - Filho é uma cópia do pai, executa o mesmo programa, ou
 - Filho pode executar um programa diferente.
- **em UNIX:**
 - `fork` – cria um novo processo.
 - `execve` – usado a seguir ao **fork** para substituir no espaço de memória do processo o programa a executar.

Criação de processos em UNIX

Faz-se através da função de sistema:

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

- **fork** retorna 0 ao processo-filho e retorna o PID do filho ao processo-pai. Caso não seja possível criar o novo processo, retorna -1.



Como distinguir a execução do processo-filho da do processo-pai?

```
if ( (pid_filho=fork())==0) {
    <código-processo-filho>
}
else {
    <código-processo-pai>
}
<código possivelmente comum>
```

Diferenças entre pai e filho:

- diferentes PID e PPID
- filho não herda locks de ficheiros que pertencem ao pai.

Exemplo do uso de fork()

```
#include <stdio.h>
main()
{
    int i;

    if (fork() == 0)    /* filho */
        for (i=0; i<5; i++) {
            printf("Filho: %d\n", i);
            sleep(2);
        }
    else                /* pai */
        for (i=0; i<5; i++) {
            printf("Pai: %d\n", i);
            sleep(3);
        }
}
```

execução:

Pai: 0
Filho: 0
Filho: 1
Pai: 1
Filho: 2
Pai: 2
Filho: 3
Filho: 4
Pai: 3
Pai: 4

→ o processo-filho executa o mesmo programa que o processo-pai, mas partes diferentes!

→ pai e filho executam concorrentemente.

Novos processos para execução de programas

É comum pretender-se que o processo-filho execute um programa diferente daquele que o criou.

As funções de sistema:

```
int execl(char *path, char *com, char *arg, ...);
int execlp(char *file, char *com, char *arg, ...);

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

diferem na passagem dos argumentos para o programa a executar.

→ Estas funções permitem substituir o programa que faz a chamada por um outro programa executável.

→ Modificam o segmento de dados e texto do processo, mas não alteram o seu contexto. Isto é, mantêm os mesmos identificadores de recursos e ficheiros abertos que o processo possuía antes da execução do `exec()`.

Exemplo: fork() + exec()

Programa que cria um processo para executar ls -l:

```
main()
{
    int filhoID, estado;
    char path[] = ``/bin/ls``;
    char cmd[] = ``ls``;
    char arg[] = ``-l``;

    if ( (filhoID= fork()) == -1) {
        printf(``Erro no fork``);
        exit(1);
    }
    else if (filhoID==0) {
        if (execl(path, cmd, arg, NULL)<0) {
            printf(``O exec falhou``);
            exit(1);
        }
    }
    else if (filhoID != wait(&estado))
        printf(``sinal antes do filho terminar``);
    exit(0);
}
```

Identificação do processo

Os processos são identificados através de um inteiro único, atribuído na criação do processo.

- Identificador de um processo (PID), obtido por:

```
pid_t getpid(void);
```

- Identificador do processo-pai:

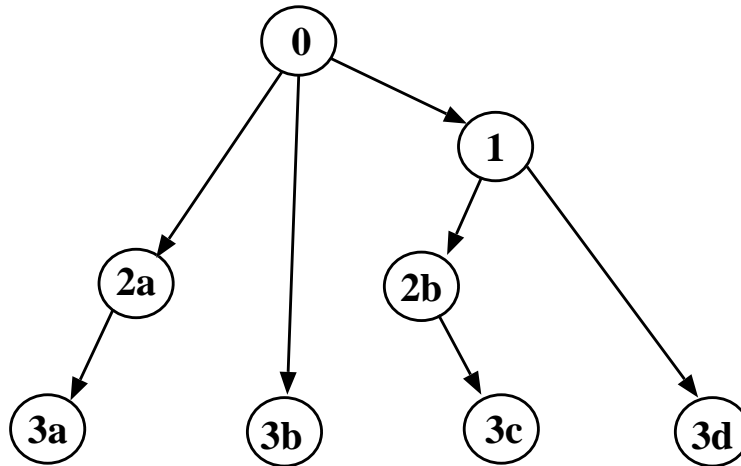
```
pid_t getppid(void);
```

- Cada processo tem um “dono” (owner), que tem privilégios sobre o processo. Pode ser determinado por:

```
uid_t getuid(void);
```

fornece o UID efectivo que permite determinar os privilégios do processo no acesso a recursos.

Como criar uma árvore de processos?



```
main()  
{  
    int i, n=4;  
    ...  
  
    for (i= 1; i<n; i++)  
        if ((filhoID=fork()) == -1)  
            break;  
    fprintf(stderr, ``proc(\\%ld) com pai(\\%ld)\\n``,  
            (long)getpid(), (long)getppid());  
}
```

Funções: `exit()` e `wait()`

- **Terminar a execução de um processo:**

```
void exit(int estado)
```

retorna o controlo ao processo-pai, indicando-lhe através da variável `estado` como é que o processo-filho terminou:

`estado == 0` – terminou normal.

`estado ≠ 0` – terminou com erro (valor é o código do erro).

- **Esperar que um processo-filho termine:**

```
pid_t wait(int *estado)
```

o processo que executa a função, suspende à espera que um dos processos-filho termine, ou até receber ele mesmo um sinal para terminar.

A função `wait()` retorna o identificador do processo que terminou e a forma como terminou (através da variável `estado`).

- **Esperar que um processo-filho específico termine:**

```
pid_t waitpid(pid_t pid, int *estado, int opções)
```

o processo suspende à espera que o processo `pid` termine. Se o valor de `pid` for `-1`, esta função equivale à função `wait()`.

Exemplo com exit() e wait()

```
#include <sys/types.h>
#include <sys/wait.h>
#include <stdlib.h>

main()
{
    pid_t filhoID;
    int estado;

    if ((filhoID= fork()) == -1) {
        printf(``O fork falhou\n``);
        exit(1);
    }
    if (filhoID==0)
        printf(``Sou o filho com pid=%ld\n``,(long)getpid());
    else if (wait(&estado) != filhoID)
        printf(``Um sinal interrompeu o wait()\n``);
    else
        printf(``Sou o pai com pid=%ld\n``,(long)getpid());
    exit(0);
}
```


I/O em Unix

Em C é habitual usar-se a biblioteca `stdio` para efectuar operações de I/O (e.g. `printf()`, `scanf()`, etc.).

As funções desta biblioteca são de *alto-nível* e realizam 3 operações importantes:

- *buffering* automático. Usam um buffer interno invisível ao programador que permite a leitura de muitos bytes de cada vez.
- conversões de input e output. Por exemplo, conversão da representação de um inteiro no seu valor numérico.
- formatação automática de input e output.

Contudo, nem sempre estas operações são desejadas. Quando se faz I/O directamente para um periférico, e.g. `tape`, é necessário ter maior controlo, nomeadamente sobre o tamanho dos buffers.

O Unix fornece uma interface *baixo-nível* para I/O, na qual um ficheiro é referenciado por um *descriptor de ficheiro* que mais não é do que um inteiro.

Descritores de ficheiros

Ao criar-se um ficheiro novo ou abrir-se um ficheiro existente, o que se obtém é um descriptor do ficheiro, que identifica o ficheiro em operações subsequentes.

Descritores 0, 1 e 2 são pré-definidos e correspondem a `stdin`, `stdout` e `stderr`.

→ **abertura e criação de ficheiros:**

```
int open(char *path, int flags, mode_t mode);  
int creat(char *path, mode_t mode);
```

flags: `O_RDONLY`, `O_WRONLY`, `O_CREAT`, ...

mode: define permissões de acesso.

A função `creat` é equivalente à função `open` com as flags definidas por: `O_CREAT | O_WRONLY | O_TRUNC`.

→ **fecho de um descriptor:**

```
int close(int fd);
```

Descritores de ficheiros (cont.)

→ **leitura e escrita:**

```
ssize_t read(int fd, void *buffer, size_t count);
```

lê `count` bytes do ficheiro cujo descriptor é `fd` para o `buffer`.
Devolve o número de bytes lidos ou `-1` se ocorrer um erro.

```
ssize_t write(int fd, void *buffer, size_t count);
```

escreve para o ficheiro cujo descriptor é `fd`, `count` bytes a partir de `buffer`. A função retorna o número de bytes escritos, ou `-1` se ocorrerem erros.

→ **acesso aleatório:**

```
off_t lseek(int fd, off_t offset, int whence);
```

posiciona a “cabeça de leitura” associada ao descriptor `fd` numa posição definida por `offset`.

em que `offset` representa o deslocamento a fazer sobre a posição corrente no ficheiro e `whence` define como interpretar o `offset`. Valores para `whence`:

`SEEK_SET` - posição corrente passa para `offset` bytes.

`SEEK_CUR` - adicionar `offset` à posição corrente.

`SEEK_END` - subtrair `offset` à posição corrente.

`offset = lseek(fd, 0, SEEK_CUR)` – determina a posição corrente

`new_offset = lseek(fd, 1024, SEEK_SET)` – posição corrente fica no byte 1024.

Exemplo: junção de dois ficheiros

Para simplificar não se inclui código de teste a situações de erro.

```
#define FLAGS O_WRONLY|O_CREAT|O_APPEND

main(int argc, char *argv[])
{
    int n, from, to;
    char buf[1024];

    from= open(argv[1], O_RDONLY);
    to= open(argv[2], FLAGS, 0644);

    while ((n= read(from, buf, sizeof(buf))) >0)
        write(to, buf, n);

    close(from);
    close(to);
    exit(0);
}
```

Supondo que o ficheiro se chama `junta.c` e o executável `junta`, então poderíamos fazer:

```
% junta fichA fichB
```

o que resultaria em acrescentar ao ficheiro `fichB` o conteúdo do ficheiro `fichA`.

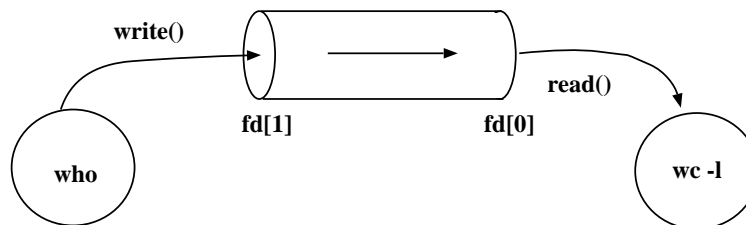
Canos (Pipes)

As pipes são um mecanismo de comunicação unidirecional entre dois ou mais processos.

Um exemplo do uso de pipes na shell é:

```
% who | wc -l
```

O programa `who` gera uma linha por utilizador e envia-a ao programa `wc` que vai contar o número de linhas recebidas.



→ **Criação de pipes:**

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

instancia dois descriptors de um ficheiro:

- `fd[0]` - descriptor associado à extremidade de leitura.
- `fd[1]` - descriptor associado à extremidade de escrita.

Retorna 0 se for bem sucedida e -1 caso ocorram erros.

→ Se um processo tentar ler de uma pipe vazia, mas cuja extremidade de escrita está aberta, o processo adormece até que exista input disponível.

Exemplo do uso de pipes

As pipes são habitualmente usadas para comunicação entre processo-pai e processo-filho (um deles escreve e o outro lê).

Um programa em que o processo-filho envia ao processo-pai uma mensagem:

```
#define READ  0
#define WRITE 1
char *msg= ``Ola papa!``;
main() {
    int fd[2], msgSize=strlen(msg)+1;
    char buf[100];

    pipe(fd);
    if (fork()==0) {
        close(fd[READ]);
        write(fd[WRITE],msg,msgSize);
        close(fd[WRITE]);
    }
    else {
        close(fd[WRITE]);
        msgSize= read(fd[READ],buf,100);
        printf(``Nbytes:%d, Msg:%s\n``,msgSize,buf);
        close(fd[READ]);
    }
    exit(0);
}
```

→ Comunicação bi-direccional? ⇒ 2 pipes.

Somar os elementos de uma matriz: N processos e pipes

```
/* cria N processos, cada um calcula a soma de 1 linha
   de 1 matriz e escreve na pipe o resultado. O pai
   lê as somas parciais e calcula a soma global */
#include <stdio.h>
#include <stdlib.h>
#define N      3
#define READ  0
#define WRITE 1

main() {
    int a[N][N]={{1,1,1},{2,2,2},{3,3,3}};
    int i, somaLin, somaGlo=0, fd[2];

    if (pipe(fd) == -1)
        msg_erro("`pipe() falhou!`");
    for (i=0; i<N; i++)
        if(fork() == 0) {                /* filho */
            somaLin= soma_vetor(a[i]);
            close(fd[READ]);
            write(fd[WRITE], &somaLin, sizeof(int));
            exit(0);
        }
    close(fd[WRITE]);                    /* pai */
    for (i=0; i<N; i++) {
        read(fd[READ], &somaLin, sizeof(int));
        somaGlo += somaLin;
    }
    close(fd[READ]); // ATENÇÃO!!! close somente 1 vez!!!
    printf("`soma da matriz: %d\n`", somaGlo);
}

int soma_vetor(int v[]) {
    int i, soma=0;
    for (i=0; i<N; i++) soma += v[i];
    return soma;
}
```

Duplicação de descritores

Existem situações em que se torna necessário duplicar descritores de ficheiros, nomeadamente quando o ficheiro representa uma *pipe* e quisermos construir uma *pipeline* entre processos.

→ **criação de uma cópia de um descriptor:**

```
int dup(int oldfd)
```

determina o menor descriptor disponível e coloca-o a apontar para o mesmo ficheiro que `oldfd`.

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

fecha `newfd` se estiver activo e coloca-o a apontar para o mesmo ficheiro que `oldfd`.

Os descritores original e cópia partilham o mesmo apontador para o ficheiro assim como o modo de acesso.

As funções retornam o novo descriptor ou -1 se ocorrer erro.

Vejamos um exemplo para redireccionar o standard-output de um processo ao standard-input do outro (e.g. pipes da shell).

Exemplo com dup2()

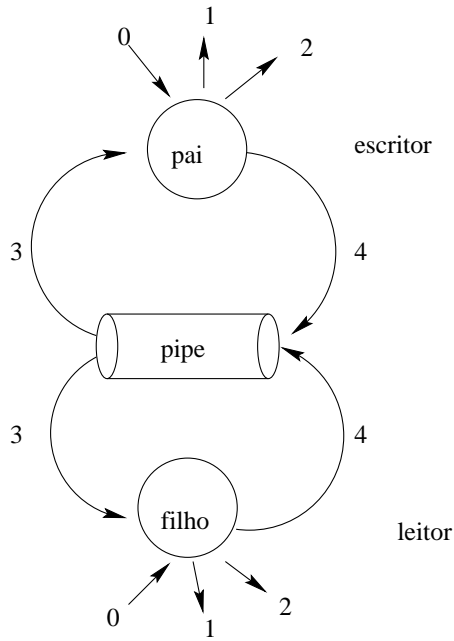
Programa que executa 2 programas dados, ligando o stdout do primeiro ao stdin do segundo (e.g. ligador who wc).

```
#define Read 0
#define Write 1
main(int argc, char *argv[])
{
    int fd[2];

    pipe(fd);
    if (fork() == 0) { /* filho */
        close(fd[Write]);
        dup2(fd[Read], 0);
        close(fd[Read]);
        execlp(argv[2], argv[2], NULL);
        perror(``ligação não sucedida``);
    }
    else { /* pai */
        close(fd[Read]);
        dup2(fd[Write], 1);
        close(fd[Write]);
        execlp(argv[1], argv[1], NULL);
        perror(``ligação não sucedida``);
    }
}
```

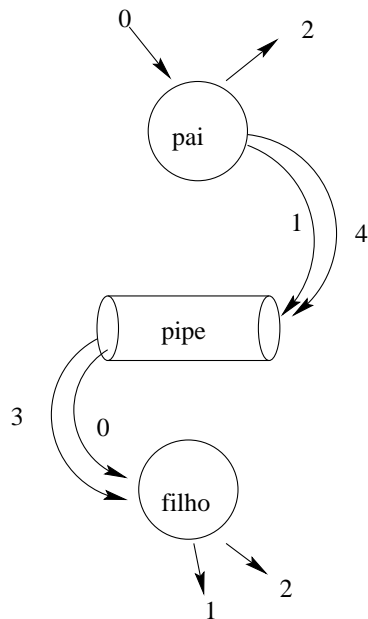
O exemplo em detalhe (1)

A: logo apos o fork()



	pai	filho
0	stdin	stdin
1	stdout	stdout
2	stderr	stderr
3	pipe_read	pipe_read
4	pipe_write	pipe_write

B: apos os dois processos executarem dup2()

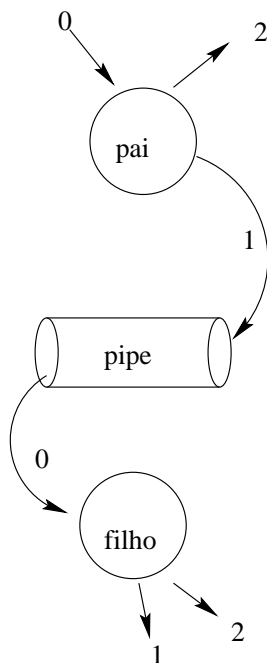


	pai	filho
0	stdin	pipe_read
1	pipe_write	stdout
2	stderr	stderr
3		pipe_read
4	pipe_write	

O exemplo em detalhe (2)

A: logo apos o fork()

C: apos o close() e antes do execlp()



	pai	filho
0	stdin	pipe_read
1	pipe_write	stdout
2	stderr	stderr

- o processo-pai executa o comando em argv[1] e escreve o resultado na pipe.
- o processo-filho le da pipe e executa o comando argv[2]

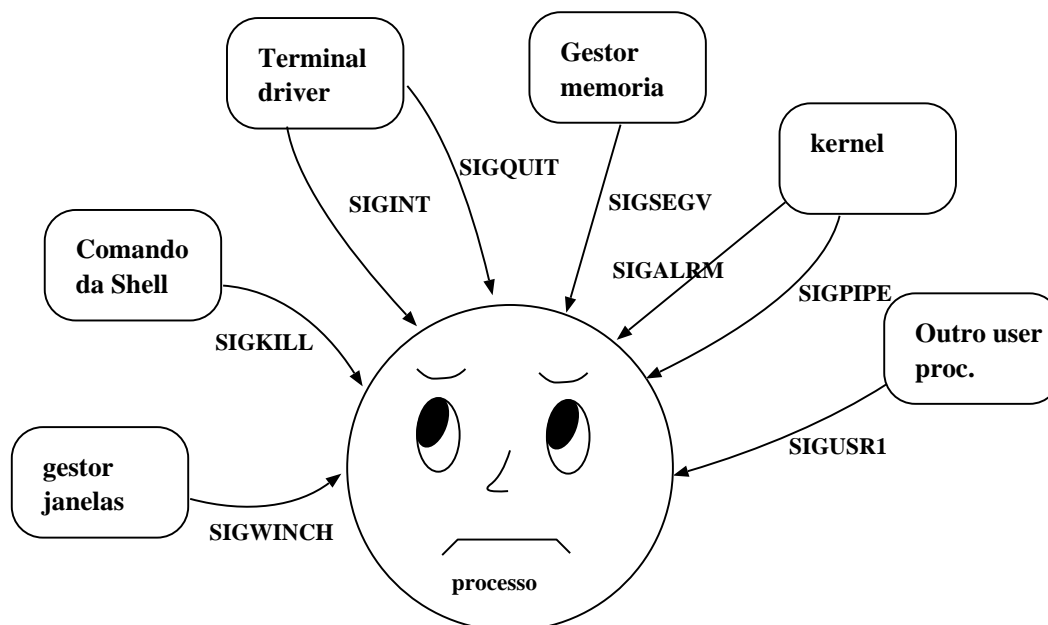
Exercício: modifique o programa anterior de forma a criar comunicação bidirecional entre os dois processos.

Sinais: eventos assíncronos

- Um sinal é um evento assíncrono que pode ser enviado a um processo, avisando-o de que algo de “inesperado” ou “anormal” aconteceu.
- *Evento Assíncrono* significa que pode ocorrer a qualquer momento.
- Tipos de sinais: (vêr `/usr/include/signal.h`)

Nome	Acção Default	Descrição/Causa
SIGINT	Termina proc.	Interrupção gerada pelo teclado
SIGQUIT	Imagem core	Abortar a execução (por teclado)
SIGKILL	Termina proc.	Resultado de um <code>kill -9</code>
SIGSEGV	Imagem core	Ref. inválida à memória
SIGPIPE	Termina proc.	Escrita na pipe sem ninguém para lêr
SIGALRM	Termina proc.	Sinal gerado por um despertador
SIGFPE	Imagem core	Excepção floating-point (divisão por zero)
SIGUSR1	Termina proc.	Sinal definido pelo utilizador

- De onde vêm os sinais?



Como é que um processo responde a um sinal?

Um processo pode escolher como responder à ocorrência de um sinal. Pode:

- *ignorar o sinal*: e.g. um processo pode proteger-se e ignorar sinais de interrupção.
- *apanhar o sinal*, executar uma função (signal-handler) em modo kernel e depois continuar a sua execução.
- aceitar a acção *default* do sinal, que na maioria dos casos termina o processo.

• Envio explícito de um sinal

```
int kill(pid_t pid, int sig);
```

envia o sinal definido por `sig` ao processo identificado por `pid`.

Um processo só pode enviar sinais aos processos com o mesmo UID.

Apanhar ou ignorar sinais

```
void *signal(int signum, void *handler())
```

onde `signum` define uma acção ou um sinal que quando ocorrer é apanhado e tratado pela função `handler()`. A função retorna o endereço da função que estava activa para o sinal indicado.

- A função `handler()` pode ser definida pelo utilizador ou uma das seguintes: `SIG_IGN` – para ignorar o sinal; ou `SIG_DFL` – para repôr o default para o sinal.

Outras funções ligadas a sinais

- Um processo pode requerer ao SO para que seja gerado um *timeout* enquanto faz outras tarefas. I.e. pede que lhe seja enviado um SIGALRM ao fim de um certo tempo (nsecs segundos):

```
int alarm(unsigned int nsecs);
```

- Um processo pode voluntariamente adormecer durante um certo tempo:

```
int sleep(unsigned int nsecs);
```

- Um processo também pode suspender voluntariamente a execução até receber um sinal:

```
int pause();
```

- **O que fazer com sinais?**

- Ignorar a acção de um dado sinal (ver os exemplos).
- Reconfiguração dinâmica. Um programa que leia dados de configuração de um ficheiro, normalmente fá-lo no início da execução, e caso ocorra alteração no ficheiro de configuração é necessário recomeçar o programa. Com um handler especial isso não seria necessário.

```
void le_fich_configuracao(int sig) {
    int fd=open(`myconfig`, O_RDONLY);
    /* ... leitura ...*/
    close(fd);
    signal(SIGHUP, le_fich_configuracao);
}
main() {
    le_fich_configuracao();
    while (1) { ... }
}
```

Exemplo: apanhar o sinal SIGINT.

O que fazer com sinais? (cont.)

- Libertar memória antes de terminar. Ao apanhar o SIGINT chama uma função que liberta segmentos de memória partilhada, ou remove ficheiros temporários.
- Ligar e desligar debugging.
- Programa que se protege contra o sinal SIGINT, ie contra o sinal control-c.

```
#include <signal.h>

main()
{
    int (*f0)();

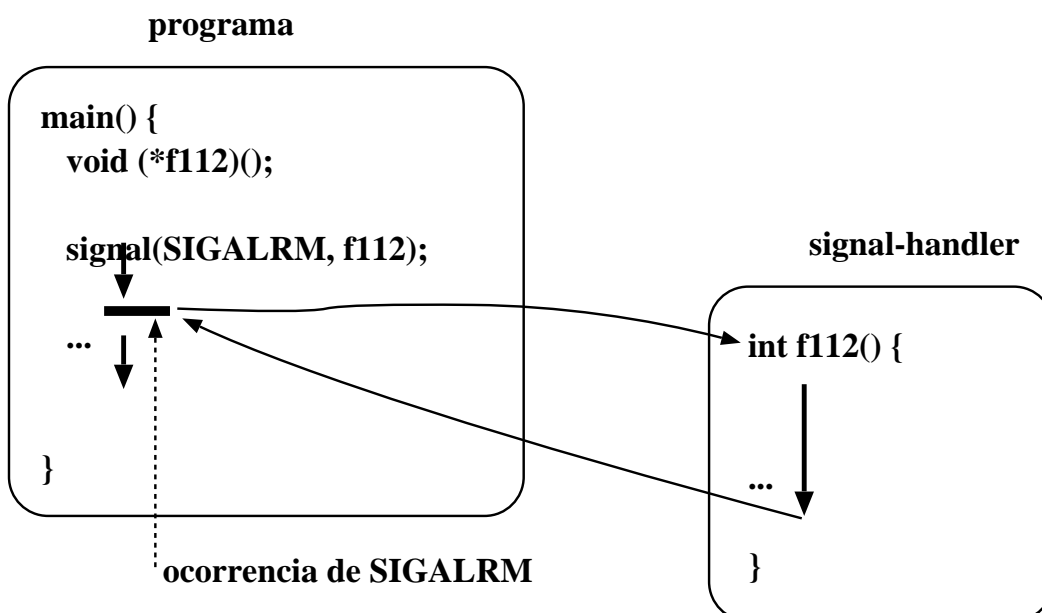
    printf(``Control-c activo\n'');
    sleep(3);
    f0= signal(SIGINT, SIG_IGN);
    printf(``Control-c inactivo\n'');
    sleep(3);
    signal(SIGINT, f0);
    printf(``Control-c activo\n'');
    sleep(3);
    printf(``Adeusinho!\n'');
}
```

Exemplo: apanhar o sinal SIGALRM.

```
#include <signal.h>
#include <unistd.h>

int flag=0;
void f112();

main()
{
    signal(SIGALRM, f112);
    alarm(5);
    printf(``Esperando...\n``);
    while (!flag)
        pause();
    printf(``terminei.\n``);
}
void f112()
{
    printf(``Recebi sinal de alarme\n``);
    flag=1;
}
```



Exemplo: cadeia de N processos

Escrever um programa que crie uma cadeia de N processos tal que o 1o processo da cadeia cria o 2o, o 2o cria o 3o, etc. Cada processo deve escrever o seu PID e o PID do processo-pai.

```
#define is_child(P)  (P == 0)

void cadeia_procs()
{
    pid_t pai, new_proc;
    int i;

    for (i=1; i<=N; i++) {

        pai = getpid();
        new_proc = fork();

        if (is_child(new_proc))
            printf("FILHO %d: PID=%d\tPPID=%d\n", i, getpid(), pai);
        else
            break;
    }
    return;
}
```

Exemplo: sincronizar processos

Escrever um programa que crie um processo filho para calcular o número de fibonacci de ordem 10, enviando o resultado ao processo pai por um sinal. O processo pai deve esperar que o processo filho termine, apanhar o sinal enviado e escrevê-lo. **ATENÇÃO:** este programa não funciona se o fibonacci devolver valores maiores do que 256, por causa do tipo de WEXITSTATUS!!

```
void sinc_fib10()
{
    pid_t new_proc;
    int status;

    new_proc = fork();
    if (is_child(new_proc))
        exit(fibonacci(10));
    else {
        wait(&status);
        printf("PAI: fib(10) = %d\n", WEXITSTATUS(status));
    }
    return;
}

long fibonacci(int n)
{
    if (n==0) return 1;
    if (n==1) return 1;
    return fibonacci(n-1) + fibonacci(n-2);
}
```

Exemplo: paginação do output

Escreva um programa que crie um processo filho para paginar no ecrã as mensagens do processo pai. O processo pai deve enviar ao processo filho a seguinte sequência de linhas:

```
Linha 1
...
Linha 100
```

O processo filho deve executar o programa `more` para paginar no ecrã a sequência de linhas recebida do processo pai. Como o paginador `more` precisa de ter acesso exclusivo ao terminal, o processo pai não deve terminar antes do processo filho.

```
void paginacao() {
    pid_t new_proc;
    int fd[2];

    pipe(fd);
    if (is_parent(fork())) {
        char buf[100];
        int n;
        close(fd[READ]);
        for (n=1; n<=100; n++) {
            sprintf(buf, "Linha %d\n", n);
            write(fd[WRITE], buf, strlen(buf));
        }
        close(fd[WRITE]);
        wait(NULL);
    } else { // filho
        close(fd[WRITE]);
        dup2(fd[READ], STDIN_FILENO);
        close(fd[READ]);
        execlp("more", "more", NULL);
        printf("FILHO: falha no execlp!\n");
    }
    return;
}
```

Exemplo: sincronização por sinais

Escrever um programa que crie um processo filho e ambos, pai e filho, devem imprimir alternadamente linhas, de modo a obter uma sequência alternada:

```
Filho: 1
Pai: 2
Filho: 3
...
Pai: 20
```

Solução 1:

```
void my_handler2() { signal(SIGUSR1, my_handler2);}
void sinc_sinais() {
    pid_t new_pid;
    int n=0;

    signal(SIGUSR1, my_handler2);
    if ((new_pid = fork())==0) { // se filho
        new_pid = getppid();
        n = 1;
        while (n <= 20) {
            sleep(1);
            printf(`Filho: %d\n`, n);
            n += 2;
            kill(new_pid, SIGUSR1);
            pause();
        }
    } else
        while (n <= 20) {
            pause();
            n += 2;
            printf(`Pai: %d\n`, n);
            kill(new_pid, SIGUSR1);
        }
}
```

Exemplo: sincronização por sinais

Solução 2:

```
#define LIMITE 20
int ctr=1; // var. global

void my_handler(int sinal) {
    ctr+=2;
    signal(SIGUSR1, my_handler2);
    if ( sinal==SIGKILL) exit(0);
}

void sinc_sinais(void) {
    int newpid,n2;
    n2=ctr=0;
    signal(SIGUSR1,my_handler2);
    if ((newpid=fork())==0) {
        ctr=1;
        while (ctr<LIMITE) {
            printf("Filho %d\n",ctr);
            n2=ctr;
            kill(getppid(),SIGUSR1);
            // esperar que o outro processo intervenha
            if(ctr==n2) pause();
        }
    } else {
        while (ctr<LIMITE) {
            // esperar que o outro processo intervenha
            if(ctr==n2) pause();
            printf("Pai %d\n",ctr);
            n2=ctr;
            kill(newpid,SIGUSR1);
        }
        kill(newpid,SIGKILL);
    }
}
```

Exemplo: pipe de processos (exame Julho 2002)

Escreva um programa que tire partido do uso de *pipes* para implementar comunicação entre processos. O programa deverá implementar um cenário em que o processo principal cria NP processos filho ($NP > 0$) para escreverem alternadamente linhas de *output* de modo a obterem uma sequência semelhante à do exemplo que se segue:

```
Filho 1: 1
Filho 2: 2
Filho 3: 3
Filho 1: 4
Filho 2: 5
Filho 3: 6
Filho 1: 7
Filho 2: 8
Filho 3: 9
Filho 1: 10
```

O primeiro filho a ser criado inicia a sequência a que se seguem os restantes filhos pela ordem que foram criados. O processo decorre repetidamente até que um dos filhos atinga um determinado valor VAL ($VAL > 0$). Attingido esse valor (10 no exemplo), todos os processos filhos devem terminar e o processo pai só deve terminar assim que todos os filhos terminem. Assuma que NP e VAL são constantes do sistema.

Exemplo: pipe de processos (cont.)

```
#include <sys/types.h>
#include <signal.h>
#include <unistd.h>
#define READ 0
#define WRITE 1
#define NP 3
#define VAL 10

main() {
    pid_t proc[NP];
    int fd[NP][2], np, val, i;

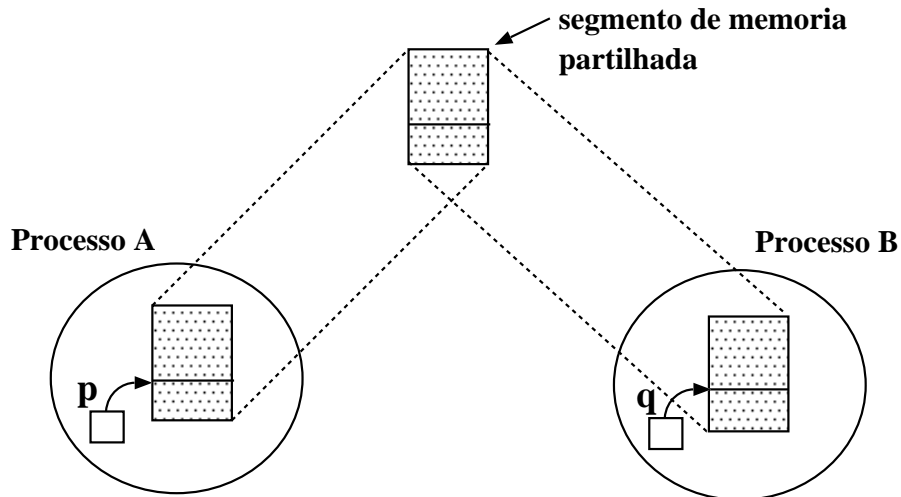
    for (np=0; np<NP; np++)
        pipe(fd[np]);

    for (np=0; np<NP; np++) {
        if ((proc[np] = fork()) == 0) {
            // processos do meio
            for (i=0; i<NP; i++) {
                if (i != np) close(fd[i][WRITE]);
                if (i != (np-1+NP)%NP) close(fd[i][READ]);
            }
            // primeiro processo
            if (np == 0) {
                val = 1;
                printf("Filho 1: 1\n");
                write(fd[np][WRITE], &val, sizeof(int));
            }
            while(read(fd[(np-1+NP)%NP][READ], &val, sizeof(int))) {
                if (val == VAL) break;
                val++;
                printf("Filho %d: %d\n", np + 1, val);
                write(fd[np][WRITE], &val, sizeof(int));
            }
        }
    }
}
```

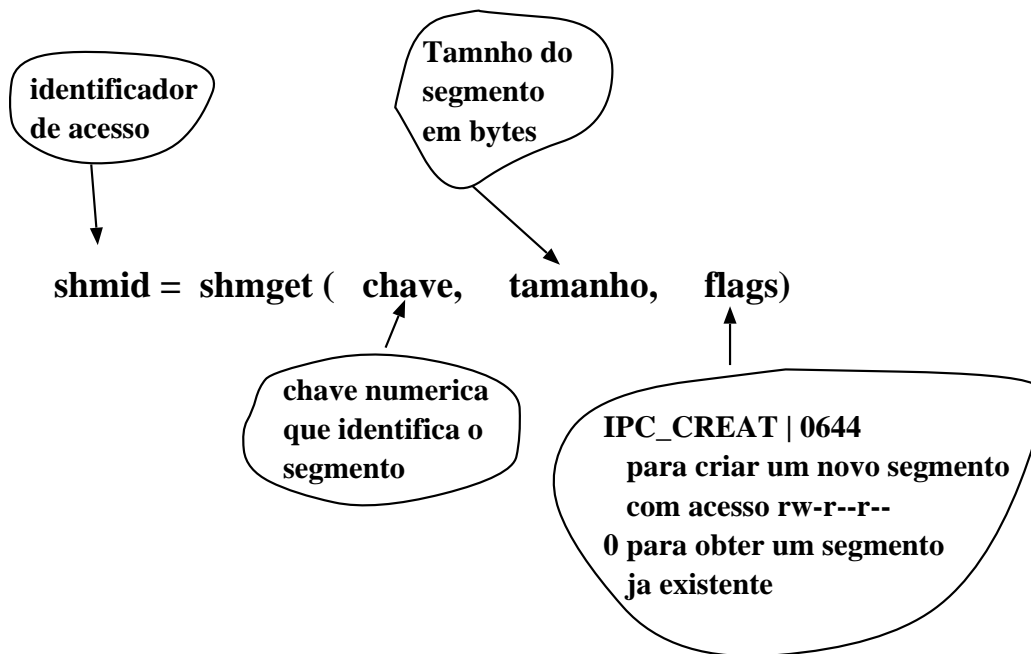
```
    // todos procs filho
    close(fd[np][WRITE]);
    close(fd[(np-1+NP)%NP][READ]);
    exit(0);
}
}
// pai fecha todas as extremidades
for (np=0; np<NP; np++) {
    close(fd[np][WRITE]);
    close(fd[np][READ]);
}
// pai espera pelos filhos
for (np=0; np<NP; np++)
    waitpid(proc[np],0,0);
    exit(0);
}
```


Memória partilhada em Unix SysV

A forma mais geral de comunicação entre processos é através de memória partilhada.

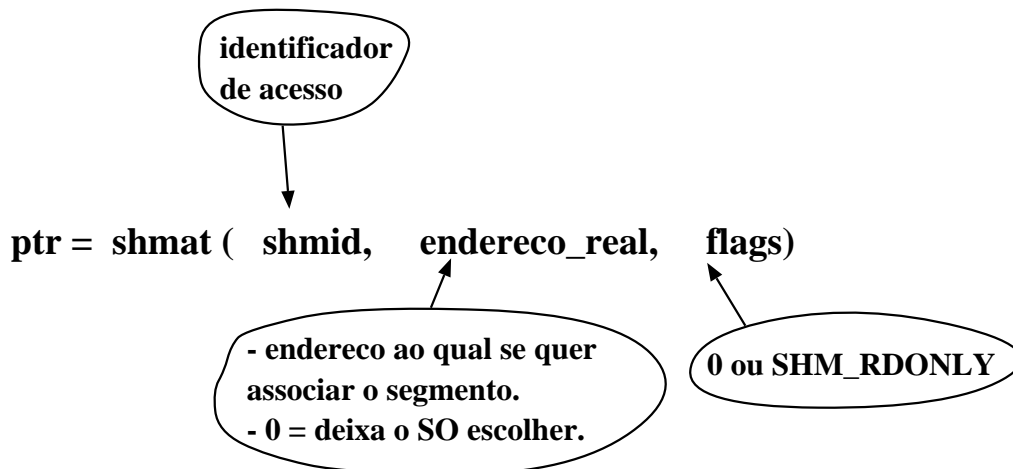


→ Criação de um segmento de memória partilhada:



Memória partilhada (cont.)

→ Ligar um segmento de memória partilhada a um endereço virtual no espaço de endereçamento do processo:



→ Desligar o segmento de memória partilhada do espaço do processo:

`int shmdt(ptr)` – o segmento deixa de ficar associado ao endereço local ao processo, `ptr`.

→ Remover o segmento de memória partilhada:

`int shmctl(shmid, cmd, buffer)` – caso `cmd` seja `IPC_RMID`, remove o segmento de memória partilhada do sistema.

→ O número de segmentos de memória partilhada que é possível criar é limitado. O SO fornece dois comandos para lidar com segmentos:

- `ipcs` – vêr que segmentos estão a ser usados;
- `ipcrm` – remover um segmento.

Exemplo: shmget() e shmat()

Exemplo de 2 programas (um servidor e outro cliente) que comunicam através de memória partilhada.

```
#include <sys/ipc.h>
#include <sys/shm.h>
#define SHMSZ 27
main() {
    char c, *shm, *s;
    int chave= 5678, shmid;

    shmid= shmget(chave, SHMSZ, (IPC_CREAT|0666));
    shm= (char *)shmat(shmid, NULL, 0);

    s= shm; /* escreve info em memória */
    for (c='a'; c<='z'; c++)
        *s++= c;
    *s= '\0';

    /* espera até que outro proc.
       altere o 1o. char em memória */

    while (*shm != '*')
        sleep(1);
    shmdt(shmid); /* liberta segmento */
    exit(0);
}
```

Exemplo shmget() e shmat() (cont.)

Programa para ler da memória partilhada:

```
#include <sys/ipc.h>
#include <sys/shm.h>
#define SHMSZ 27
main()
{
    char c, *shm, *s;
    int chave= 5678, shmid;

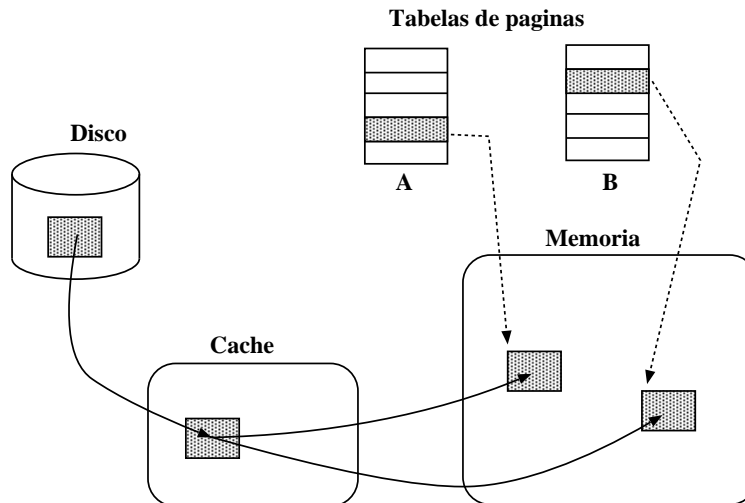
    shmid= shmget(chave, SHMSZ, 0666);
    shm= (char *)shmat(shmid, NULL, 0);

    for (s=shm; *s!='\0'; s++) /* lê da memória partilhada*/
        putchar(*s);
    putchar('\n');
    *shm='*'; /* alterar o 1o. caracter em memória */
    exit(0);
}
```

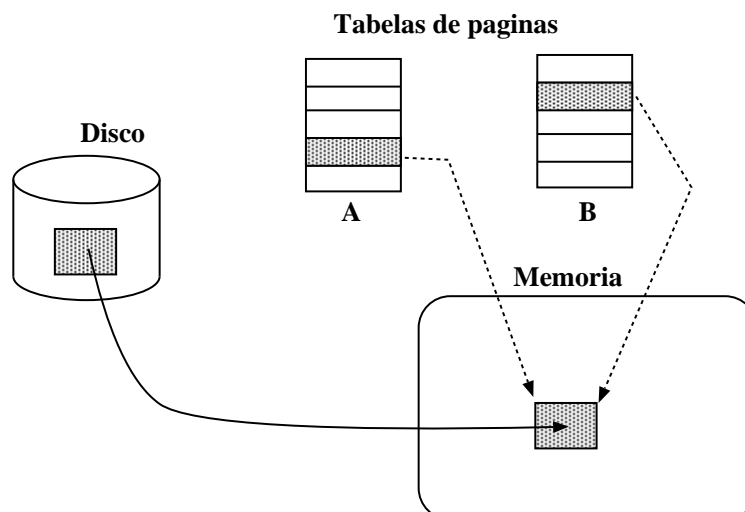
Mapeamento de Memória em Ficheiros

- Modo tradicional de acesso a ficheiros em Unix: open + (read ou write ou lseek) + close.

Dois processos mapeiam no seu espaço uma cópia da mesma página de um ficheiro:



Dois processos mapeiam a mesma página no seu espaço:



Esta página pode ser mapeada como partilhada ou como privada. Neste caso não há garantia de atomicidade na escrita e leitura do ficheiro (como acontece com read e write).

Função mmap()

`aptr = mmap(endereço, tam, prot, flags, fd, offset)`

onde,

- `aptr` é o endereço onde está colocado o mapeamento.
- `endereço` sugere um endereço em memória para o mapeamento. 0 ou NULL: o sistema escolhe.
- `tam` é o tamanho em bytes.
- `prot` é PROT_READ ou PROT_WRITE
- `flags`: MAP_SHARED
- `fd` é o descriptor do ficheiro (é necessário abrir o ficheiro antes!)
- `offset` deslocamento no ficheiro onde começar o mapeamento.

Atenção aos passos para cada processo:

- obter o descriptor do ficheiro
- fazer o `mmap` que retorna um apontador
- lê e escreve sobre o ficheiro através do apontador.

Exemplo com mmap(): rank-sort

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/mman.h>
#define N      6
#define SIZE  4*1024*sizeof(int)
int  a[N];
int *b;

main() {
    int childpid, status, fd, i;

    printf("Vetor A:\n");
    for(i=0; i<N; i++) {
        a[i]= N-i-1; printf("%d ",a[i]);
    }
    printf("\n");

    fd= open("tmp.mmap",O_RDWR); /* abre o ficheiro          */
    lseek(fd,SIZE,SEEK_SET);     /* truque para garantir */
    write(fd,"",1);             /* tamanho do fich é SIZE */

    b=(int *)mmap(0, (N+1)*sizeof(int),
        PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    for(i=0; i< N; i++)
        if ((childpid = fork()) == 0)
            putInPlace(i);
    for(i=0; i<N; i++)
        wait(&status);
    printf("Vetor B:\n");
    for(i=0; i<N; i++)
        printf("%d ",b[i]);
    printf("\n");
}
```

Exemplo com mmap (cont.)

```
void putInPlace(int i) {
    int t, j, rank;

    t= a[i];
    rank= 0;
    for (j=0; j<N; j++)
        if (t>a[j]) rank++;
    b[rank]= t;
    exit(0);
}
```


Semáforos em Unix

→ Criar um vetor de semáforos:

```
semid = semget ( chave, nsems, flag);
```

onde:

- `semid` identificador de acesso ao vetor de semáforos.
- `chave` identificador global que identifica este vetor de semáforos; se a chave for conhecida por outro processo, este pode com a execução de um novo `semget ()` e mesma chave, ganhar acesso aos mesmos semáforos.
- `nsems` número de semáforos a criar no vetor;
- `flag` condicionam o modo de uso do semáforo:
`IPC_CREAT | 0644`, cria novo com permissões `0644`; `flag = NULL`, para obter um vetor já existente.

→ Remover um vetor de semáforos:

```
semctl ( semid, semnum, comando);
```

com `comando = IPC_RMID`. Outros comandos permitem usar esta função com outro significado.

Operações sobre Semáforos em Unix

```
status= semop ( semid, semops, nsemops );
```

- executa operações sobre semáforos determinados
- `semid` identificador de acesso ao vetor de semáforos.
- `nsemops` número de operações.
- `semops`: apontador para um vetor de `nsemops` estruturas que definem as operações sobre os semáforos determinados;

Estrutura de cada elemento do vetor de operações:

```
struct sembuf {
    short semNum; /*num. sem. afectado por semOp*/
    short semOp; /*operação sobre semáforo */
    short SemFlag;
}
```

- Valores `semOp < 0` implicam espera do processo (se resultado igual a zero) e lock do semáforo; valores `semOp > 0` libertam o semáforo.

→ Semáforos binários (tomam apenas valores 0 e 1):

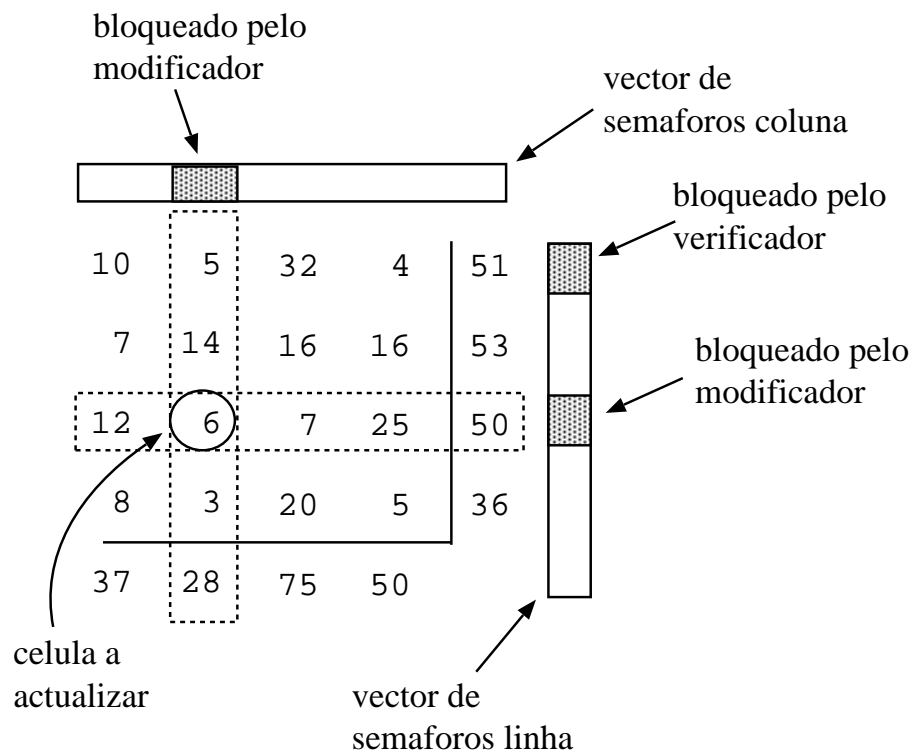
```
#define UP(sid,n) { \
    struct sembuf up={n,1,0}; \
    semop(sid, &up, 1); \
}
#define DOWN(sid,n) { \
    struct sembuf down={n,-1,0}; \
    semop(sid, &down, 1); \
}
#define INIT(sid,n) UP(sid,n)
```

Exemplo com "shm" e "sem": Folha de Cálculo

Consideremos uma espécie de folha de cálculo representada por uma matriz de N linhas e M colunas, em que cada elemento da última linha é a soma dos elementos da mesma coluna e cada elemento da última coluna é a soma dos elementos da mesma linha.

Suponhamos que temos dois processos:

- um *modificador*: periodicamente modifica, de forma aleatória, o valor de uma célula da matriz e atualiza os totais na linha e coluna que contêm a célula.
- um *verificador*: periodicamente escreve a matriz, e verifica se os totais estão corretos.



Folha de Cálculo (cont.)

Potenciais dificuldades: modificador e verificador podem estar a usar valores da mesma linha ou coluna e ocasionar problemas de concorrência.

MODIFICADOR:

```
escolhe aleatoriamente 1a célula e 1 valor
entrar na zona crítica
  modifica célula com novo valor
  atualiza total na linha
  atualiza total na coluna
sair da zona crítica
```

VERIFICADOR:

```
para todas as linhas {
  entrar na zona crítica
  calcula soma de todas células na linha
  se (soma != total na folha)
    escreve mensagem
  sair da zona crítica
}
para todas as colunas {
  entrar na zona crítica
  calcula soma de todas células na coluna
  se (soma != total na folha)
    escreve mensagem
  sair da zona crítica
}
```

Folha de Cálculo (cont.)

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>

/* CONSTANTES */
#define NLINS 8
#define NCOLS 8
#define SKEY 123
#define SSIZE NLINS*NCOLS*sizeof(int)

/* MACROS */
#define CELL(s,r,c) (*(s)+((r)*NCOLS)+(c))

/* operações sobre semáforos */
#define UP(sid,n) { \
    struct sembuf up={n,1,0}; \
    semop(sid, &up, 1); \
}
#define DOWN(sid,n) { \
    struct sembuf down={n,-1,0}; \
    semop(sid, &down, 1); \
}
#define INIT(sid,n) UP(sid,n)

/* VARIÁVEIS GLOBAIS */
int totalLin, totalCol; /* totais na linha e coluna */
int linSems, colSems; /* id vetores de semaforos */

/* PROTOTIPOS DAS FUNCOES */
void gera_nova_entrada(int *);
void escreve_e_verifica(int *);
int inicia_sems(key_t, int);
```

Folha de Cálculo (cont.)

```
int main()
{
    int id, lin, col, *folha, i=0, j=0;

    setbuf(stdout, NULL); /* evita buffering */
    totalLin= NLINS-1;
    totalCol= NCOLS-1;
    /* seg. mem. partilhada para a matriz */
    id= shmget(SKEY, SSIZE, IPC_CREAT|0600);
    folha= (int *) shmat(id, 0, 0);
    for (lin=0; lin < NLINS; lin++) /* celulas a zero */
        for (col=0; col<NCOLS; col++)
            CELL(folha, lin, col)=0;
                                /* cria e inicia vecs de sems */
    linSems= inicia_sems(SKEY, NLINS);
    colSems= inicia_sems(SKEY+1, NCOLS);
    if (fork()) { /* pai escreve e verifica*/
        for (;;) escreve_e_verifica(folha);
    }
    else { /* filho gera valores */
        for (;;) gera_nova_entrada(folha);
        exit(0);
    }
    wait(0);
    semctl(linSems, 0, IPC_RMID); /* liberta sems*/
    semctl(colSems, 0, IPC_RMID);
    shmdt(folha); /* liberta shm */
    shmctl(id, IPC_RMID, 0);
    exit(0);
}
```

Folha de Cálculo (cont.)

```
int inicia_sems(key_t k, int n)
{
    int semid, i;

    if ((semid=semget(k,n,0))!=-1) /* se ja existem */
        semctl(semid,0,IPC_RMID); /* liberta-os */
                                   /* cria novos */
    if ((semid=semget(k,n,IPC_CREAT|0600))!=-1)
        for (i=0; i<n; i++) /* inicia sems do vetor */
            INIT(semid,i);
    return semid;
}

void gera_nova_entrada(int *s)
{
    int lin, col, old, new;

    /* escolhe aleatoriamente celula e novo valor */
    lin= rand() % (NLINS-1);
    col= rand() % (NCOLS-1);
    new= rand() % 1000;

                                   /* tenta entrar na zona critica */
    DOWN(linSems, lin);
    DOWN(colSems, col);
    old= CELL(s, lin, col); /* atualiza celula e totais */
    CELL(s, lin, col)= new;
    CELL(s, lin, totalCol) += (new-old);
    CELL(s, totalLin, col) += (new-old);
    UP(colSems, col); /* sai da zona critica */
    UP(linSems, lin);
    usleep(5000);
}
```

Folha de Cálculo (cont.)

```
void escreve_e_verifica(int *s)
{
    int lin, col, soma, totalErrs;
    static int ctr= 0;

    totalErrs=0;
    ctr++;
    for (lin=0; lin<NLINS; lin++) {
        soma= 0;
        DOWN(linSems, lin);
        for (col=0; col<NCOLS; col++) {
            if (col != totalCol)
                soma += CELL(s, lin, col);
            printf("%5d", CELL(s,lin,col));
        }
        if (lin!= totalLin)
            totalErrs += (soma != CELL(s, lin, totalCol));
        UP(linSems, lin);
        printf("\n");
    }
    for (col=0; col<totalCol; col++) {
        soma=0;
        DOWN(colSems,col);
        for (lin=0; lin<totalLin; lin++)
            soma += CELL(s,lin, col);
        totalErrs += (soma != CELL(s, totalLin, col));
        UP(colSems, col);
    }
    if (totalErrs)
        printf("\nFolhaCalculo n(%d) falhou\n",ctr);
    if ((ctr % 20) == 0)
        printf("\nFolhaCalculo n(%d) processada\n",ctr);
    printf("\n-----\n");
    sleep(2);
}
```