

Fundamentos de Sistemas de Operação

Unix Windows NT Netware Mac OS DOS/V/S Vax/VMS
Linux Solaris HP/UX AIX Mach Chorus

*A API de Processos:
Continuação*

Resumo de uma aula anterior...

- *O processo, conceito oferecido pelo SO,*
 - “É um contentor” que tem um CPU, uma memória e periféricos idênticos aos da “máquina real”. Cada programa corre nesta máquina virtual como se mais nada existisse...
 - O SO oferece um conjunto de funções para manipular processos
 - Criar: `fork()`
 - Terminar: `kill(pid,...)` e `_exit(valor)`
 - Obter PIDs: `getpid()` e `getppid()`
 - Esperar: `wait()`
 - Executar novo programa no processo: `execl()`, `execlp()`, `execv()`, `execve()`, `execvp()` **AINDA não abordado nas aulas...**

`fork()`: mais exemplos (1)

□ Esqueleto de código com um `fork()` e `if`

```
int main( int argc, char *argv[] )  
{  
    int pid;  
    ...  
    pid= fork();  
  
    if (pid > 0) {  
        // Código a executar pelo pai  
    } elseif (pid == 0) {  
        // Código a executar pelo filho  
    } else  
        // Código para tratar o erro do fork;  
    }  
  
    ...  
}
```

`fork()`: mais exemplos (2)

□ Esqueleto de código com um `fork()` e `switch`

```
int main( int argc, char *argv[] )  
{  
    int pid;  
    ...  
    pid= fork();  
  
    switch (pid) {  
        case  0: // Código a executar pelo filho  
        case -1: // Código para tratar o erro do fork  
        default: // Código a executar pelo pai  
    }  
    ...  
}
```

`fork()`: erro? Como?

- *Como com (quase) todas as chamadas ao sistema, a execução de um `fork()` pode falhar... quando?*
 - Quando o número máximo de processos definido por utilizador (considerando o utilizador que está a lançar o novo processo) é atingido, ou
 - Quando o número máximo de processos definido para o sistema é atingido
 - Quando há escassez de memória livre...

Fundamentos de Sistemas de Operação

Unix Windows NT Netware Mac OS DOS/V/S Vax/VMS
Linux Solaris HP/UX AIX Mach Chorus

*Programas, Processos e o SO:
Canais de Comunicação*

Canais de Comunicação

- Um processo necessita de comunicar; e por comunicar entende-se:
 - Transferir informação de/para periféricos
 - Exemplos: teclado, ecrã
 - Transferir informação de/para ficheiros
 - Transferir informação de/para outros processos (*IPC: Inter-Process Communication*)
 - Entre processos na mesma máquina: papel fundamental: SO
 - Entre processos em máquinas distintas: papel fundamental: protocolos de Redes de Computadores (**NÃO abordado nesta UC**)

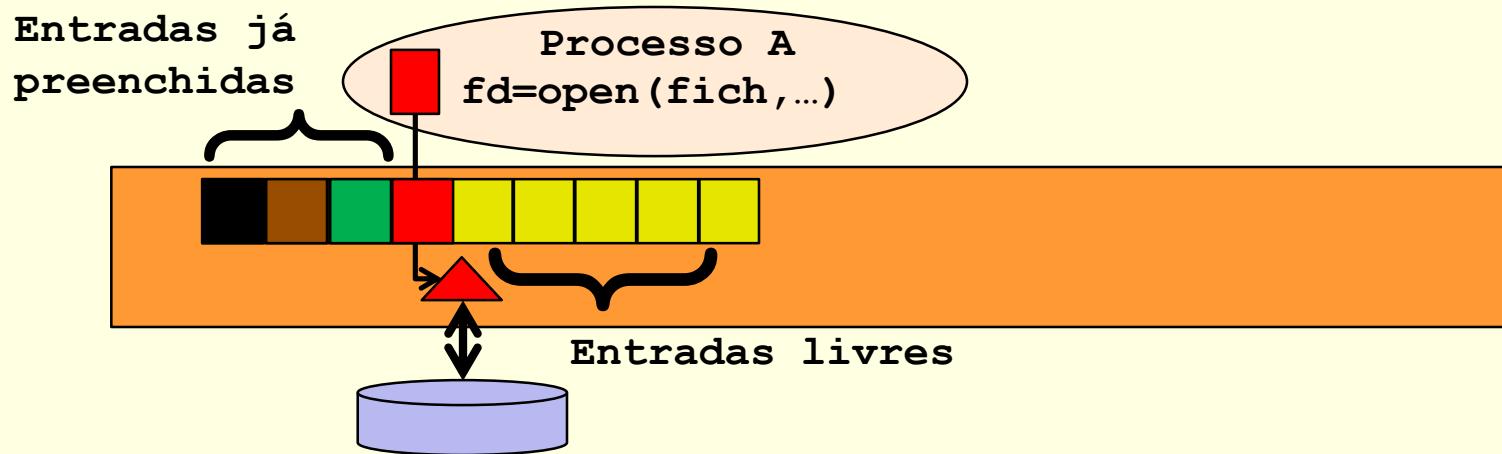
Canais de Comunicação em Unix (1)

- Nos SOs modernos o conceito de canal de comunicação que “liga” o processo ao “alvo” da transferência (periférico, ficheiro, etc.) suporta os 3 casos exactamente da mesma forma...
- Existem fundamentalmente duas formas de “estabelecer um canal de comunicação” para um “alvo”:
 - Abrir uma ligação para um “alvo” pré-existente, usando a função `open()` ou
 - Criar um novo alvo (usando a função apropriada, por ex. `creat()`, ou `pipe()`, etc.) e obtém uma ligação para este
 - Seja como for, a função usada retorna um número inteiro que se designa descriptor do ficheiro (file descriptor) ou número do canal (e até se chega por vezes a omitir a palavra “número”)

Canais de Comunicação em Unix (2)

Unix Windows NT Netware Mac OS DOS/VMS Vax/VMS Linux Solaris HP/UX AIX Mach Chorus

- No Unix, a cada processo está associado um vector, designado “tabela de descritores (de ficheiros)”. Quando o processo “abre” um “canal”, ou “ficheiro”, o SO procura no vector a entrada de menor índice que está livre, preenche essa entrada com uma referência ao “ficheiro” que se está a abrir e devolve, como retorno da função, o índice dessa entrada.

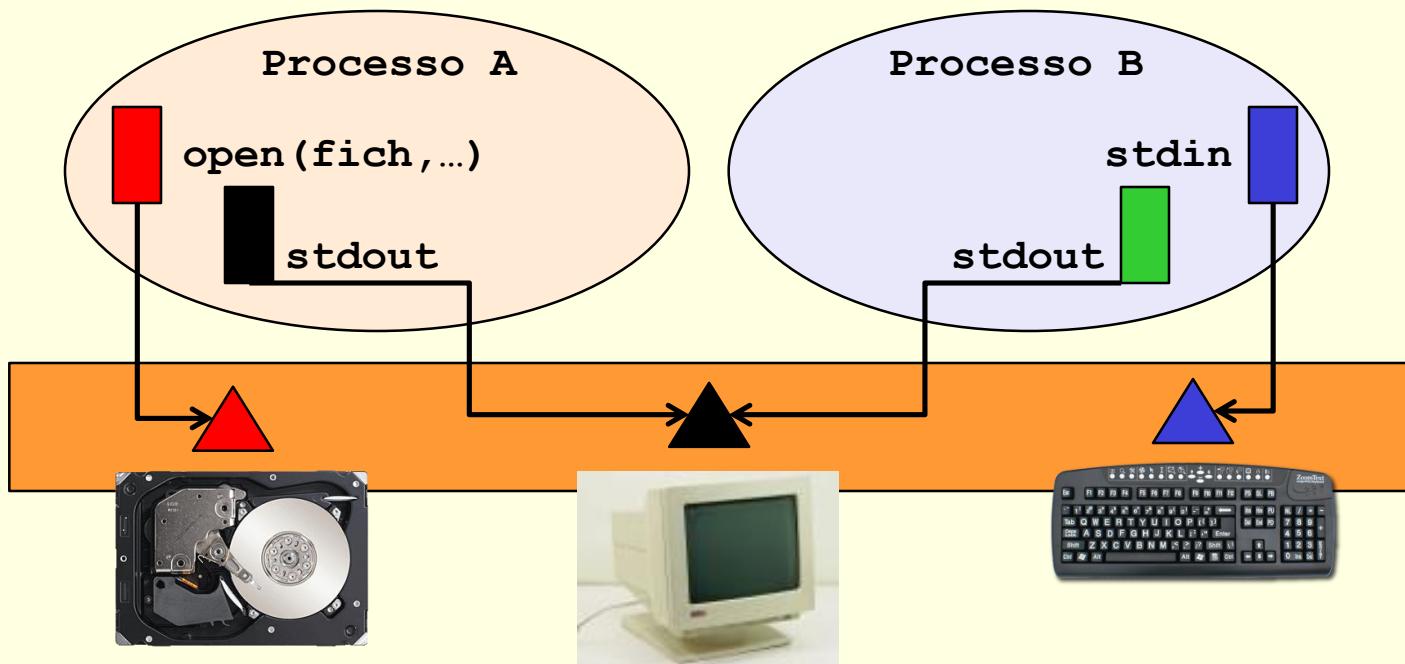


Canais de Comunicação em Unix (3)

- A partir do momento que se tem um descritor válido (i.e., que identifica um “alvo” apropriado - ficheiro, periférico, socket para comunicação numa rede, etc.)
 - Transfere-se informação de/para o “alvo” usando as primitivas `read()` e `write()` - afinal, as mesmas que já conhecemos para aceder a ficheiros...
 - Sinaliza-se o desejo de não querer transferir mais informação por via desse descritor de ficheiro, fechando-o com uma operação `close()`

Canais de Comunicação em Unix (4)

- Note-se que as mensagens escritas pelos processos A e B se “misturam” no mesmo ecrã... o que ilustra o que anteriormente referimos quando falamos, no `fork()`, da “clonagem” de canais...



Operações fundamentais de E/S: open ()

□ **int open(char *alvo, int opções)**

Onde *alvo* é o *identificador*, no sistema de ficheiros do computador, da entidade que queremos usar na comunicação... Por exemplo:

- Se queremos aceder a um ficheiro, o *alvo* pode ser "/home/pal/texto.txt"
- Se queremos aceder a um periférico, o *alvo* poderia ser "/dev/..." mas já sabemos que não podemos aceder directamente a periféricos

As opções servem para especificar muitas e variadas "coisas"... Por exemplo, podem ser usadas para especificar se apenas queremos usar o canal para ler (*O_RDONLY*), para ler e escrever (*O_RDWR*), ou só para escrever (*O_WRONLY*)

Operações fundamentais de E/S: open ()

- Exemplo 1: Queremos abrir o ficheiro “/home/pal/texto.txt” para leitura/escrita
 - `fd= open (“/home/pal/texto.txt”, O_RDWR)`
- Exemplo 2: Queremos abrir o ficheiro “texto.txt”, localizado na directória de trabalho do processo, somente para leitura
 - `fd= open (“texto.txt”, O_RDONLY)`
 - ou,
 - `fd= open (“./texto.txt”, O_RDONLY)`

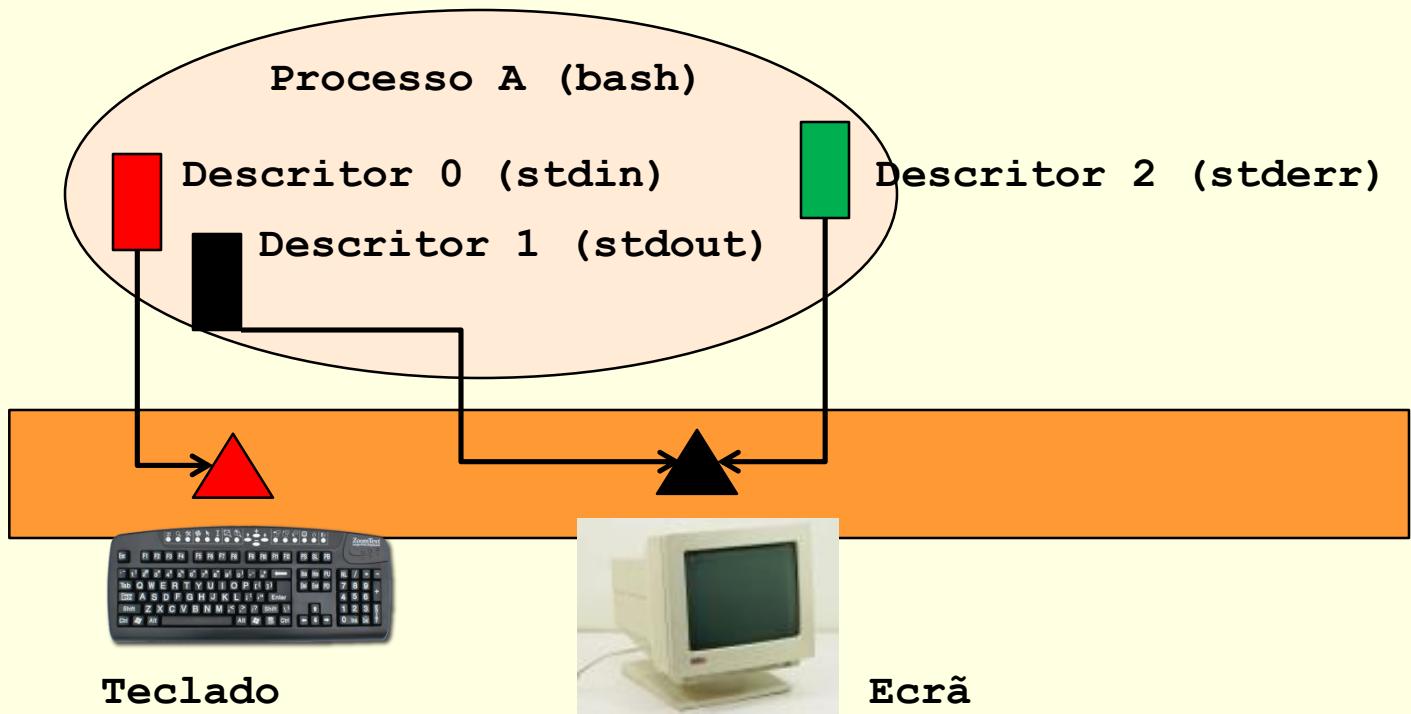
Operações fundamentais de E/S: open ()

- O **open** é uma chamada de sistema complexa porque permite fazer muitas “coisas” diferentes... Por exemplo,
 - permite criar um ficheiro:
`open("texto.txt", O_RDWR | O_CREAT, 0600)`

Já sabemos que a opção **O_RDWR** especifica que o canal deve ser aberto para ler e escrever, mas combinamos essa opção com **O_CREAT**, que indica que o ficheiro deve ser criado se já não existir, e ainda com **0600** que especifica que, no caso de ser criado, deve ficar com um conjunto de permissões que nos permitem apenas a nós, “seus criadores” aceder ao ficheiro – e mais ninguém. [As questões de controle de acessos/permissões serão apresentadas mais tarde ☺]

Canais standard em Unix

- No Unix, quando um utilizador efectua o login e “cai” numa shell, por omissão há 3 canais abertos:

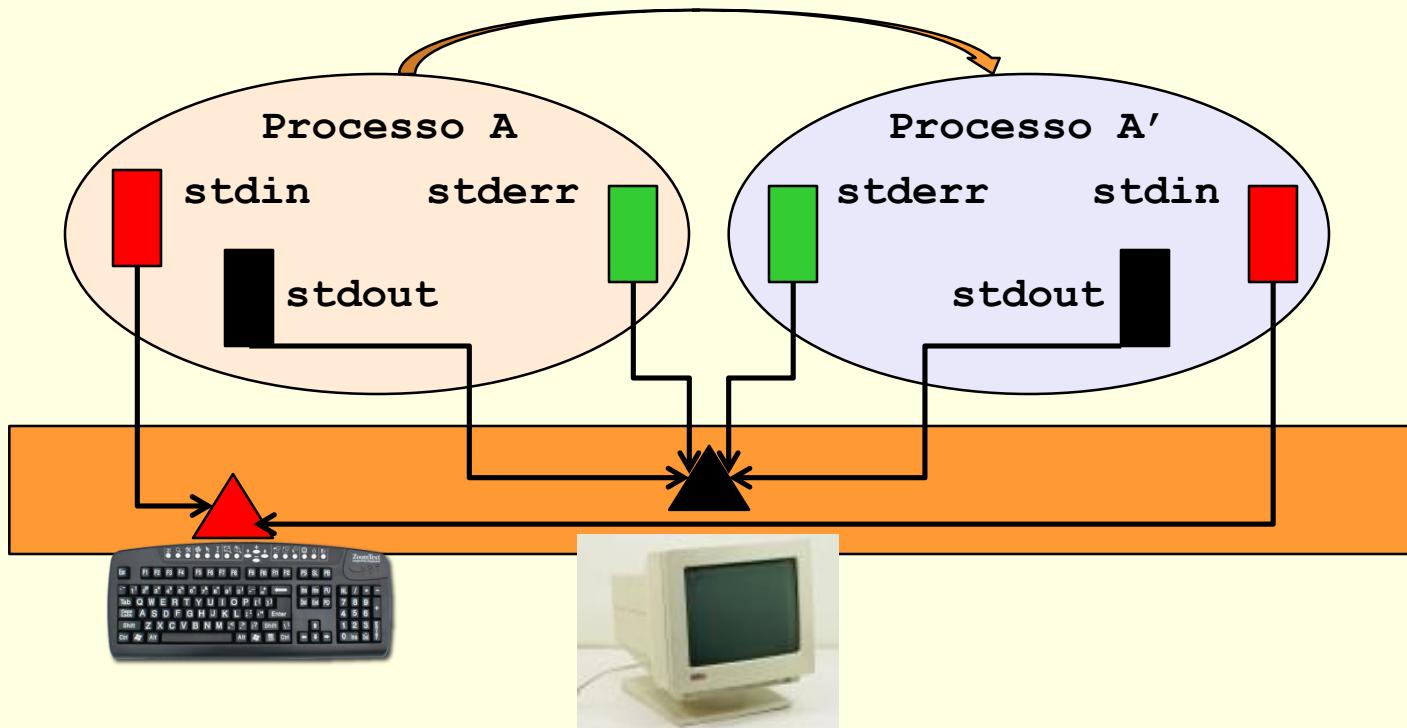


Canais standard em Unix e o open()

- *Na prática, isto quer dizer que:*
 - A *shell*, e qualquer programa dela lançado pode, por omissão, ler do descriptor 0 (e.g., teclado) e escrever nos descritores 1 (ecrã de saída) e 2 (ecrã, canal usado para mensagens de erro).
 - A execução de um novo `open()`, se efectuada com sucesso, retornará um descriptor com o número 3.
- *Em geral,*
 - O número de descriptor retornando por um `open()` é o menor inteiro que não está associado a um canal (aberto). Por exemplo,
 - Se num dado instante estão “em uso” os descritores 0, 1, 2 e 3, fecharmos o 2 e efectuarmos um novo `open()`, este retornará 2.

Canais standard em Unix e o fork()

- Num **fork()** os descritores do processo filho referenciam os mesmos canais - e dispositivos - que os do pai.



Canais standard em Unix: para pensar

- Num `fork()` os descritores do processo filho referenciam os mesmos canais - e dispositivos - que os do pai.
- Já viu, em demonstrações na aula, que os prints de pai e filho aparecem (geralmente) “misturados” no ecrã
- Questão: se os processos (pai/filho) leem ambos do teclado, o que é que, efectivamente, lê cada um?

Operações fundamentais de E/S: close ()

- **int close(int descriptor)**

Se o descritor “está aberto” (ou seja, está associado a um canal), então esta operação dissocia-os, deixando o descritor de poder ser usado para aceder a esse canal – e então qualquer tentativa subsequente de usar o descritor redonda em erro...

Se o descritor não “está aberto”, a operação falha com erro.

NOTA: quando se fecha um descritor que nos dá acesso a um “alvo” (i.e., ficheiro), isto não significa que não se pode continuar a transferir informação de/para o alvo – isso só é verdade se não houver mais nenhum descritor a referenciar esse “alvo” – Veja-se, p.ex., que por omissão, ambos os descritores 2 e 3 referenciam o ecrã.

Operações fundamentais de E/S: `read()`

□ `int read(int descr, void *buf, size_t sz)`

Se a execução da função tiver sucesso, a variável `buf` é preenchida com uma quantidade não superior a `sz` de bytes lidos do ficheiro...

- Se o `read()` retorna zero, então não há mais dados para ler [se o canal está associado a um ficheiro, estamos no fim-de-ficheiro]
- Se o `read()` retorna um valor `nb` positivo mas inferior a `sz`, então foram lidos `nb` bytes, mas não a totalidade, porque não havia dados suficientes para ler.
- Se o `read()` retorna o valor `sz`, então foi lida a quantidade exacta de bytes que pretendíamos ler.

Operações fundamentais de E/S: write()

□ **int write(int descr, void *buf, size_t sz)**

Se a execução da função tiver sucesso, o conteúdo da variável **buf** (uma quantidade não superior a **sz** de bytes) é transferido para o canal [escrito no ficheiro, se o canal está associado a um ficheiro]

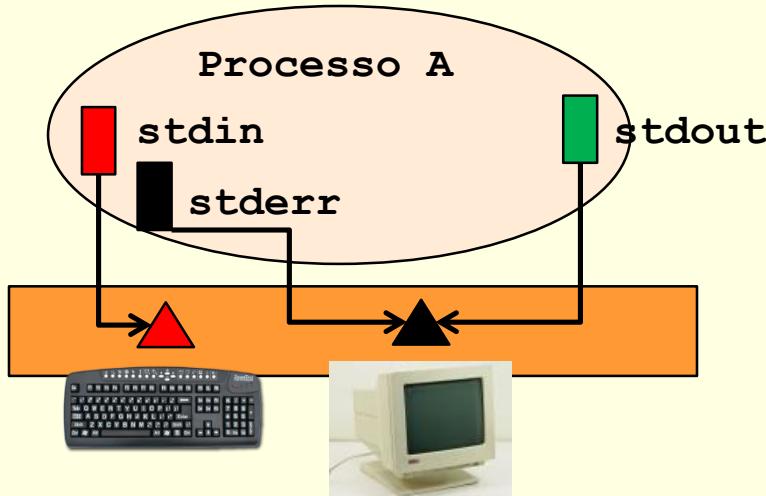
- Se o **write()** retorna zero, então nada foi escrito
- Se o **write()** retorna o valor **sz**, então foi escrita a quantidade exacta de bytes que pretendíamos.
- Se o **write()** retorna -1, houve um erro; consultar o código do erro, e o manual para conhecer a sua causa

Fundamentos de Sistemas de Operação

Unix Windows NT Netware Mac OS DOS/V/S Vax/VMS
Linux Solaris HP/UX AIX Mach Chorus

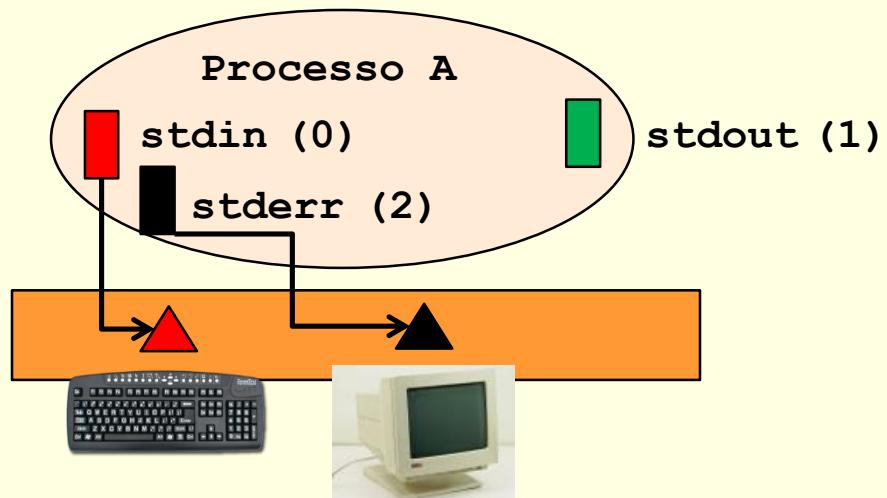
*Programas, Processos e o SO:
Redirecção de Canais de E/S*

Abertura e Fecho de Canais em Unix (1)

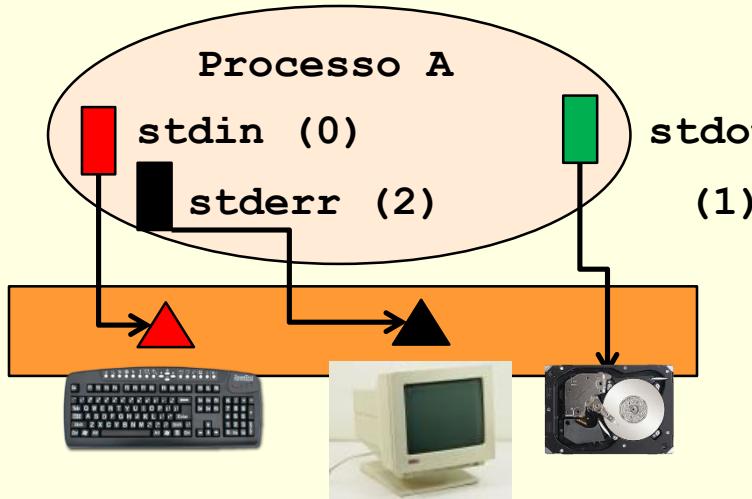


1: Estado inicial:
canais abertos por omissão,
ligados aos periféricos standard.

2: Após um `close(1)`
o descriptor 1 fica livre.



Abertura e Fecho de Canais em Unix (2)



Após o `open()` os *writes* no canal standard de saída, i.e., 1 (e logo os `printf`s) são escritos no ficheiro.

3: Após um `open("ficheiro", modo)`:
o canal 1 fica aberto novamente,
ligado ao "ficheiro"

```
#define MOD O_WRONLY|O_CREAT  
        O_TRUNC|0660

int main()  
{  
    ...                                // 1:  
  
    close(1);                            // 2:  
  
    open("ficheiro", MOD);   // 3:  
    printf("ola");  
    ...  
}
```

Demo

Unix Windows NT Netware Macos DOS/VSS Vax/VMS
Linux Solaris HP/UX AIX Mach Chorus

```
#include <stdio.h>

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main(int argc, char *argv[]) {

    printf("Aquilo que estou a escrever neste momento vai aparecer no ecrã\n" );

    close(1);
    open("dem-01.txt", O_WRONLY|O_CREAT, 0660);

    printf("Aquilo que estou a escrever agora vai aparecer no ficheiro dem-01.txt\n"
);

    return 0;
}
```

A Shell: algumas notas... (1)

□ Comandos *internos* vs. *externos*

- A shell tem um conjunto reduzido de comandos que estão implementados no próprio programa; os mais conhecidos são `cd` e `echo`
- Se um “comando” não é interno, então é um programa (instalado em `/bin` ou `/usr/bin`) [se é uma outra qualquer “aplicação”, por ex., `gcc`, poderá estar instalada num outro lugar qualquer]
- Para executar um “comando externo” a Shell
 - Faz um `fork()`
 - A shell “pai” pode ou não ficar bloqueada (ver pág. seguinte)
 - A shell filha faz `exec()` do programa

A Shell: algumas notas... (2)

□ Execução síncrona vs. assíncrona

- A shell pode, ao lançar um programa (ou “comando externo”) bloquear-se (*comportamento por omissão*)
`$ gcc -Wall teste.c` (enquanto o gcc não termina, o terminal está bloqueado – diz-se execução síncrona)
- Mas podemos evitar o bloqueamento
`$ gcc -Wall teste.c &` (o terminal continua livre e a compilação é executada – diz-se execução assíncrona)
- Apenas teremos uma situação “menos agradável” se a compilação produzir mensagens que nos “baralham” o ecrã

A shell e a redirecção de canais (1)

- Considero o seguinte comando: **\$ prog > ficheiro**
onde **prog** é um programa que escreve uma mensagem no canal de saída standard...
- O mecanismo de redirecção codificado na shell desencadeia a seguinte sucessão de acções:
 - Usando um **fork()** cria uma shell “filha”
 - A nova shell faz o **close(1)** e **open("ficheiro", modo)**
 - Depois faz o **execvp("prog", ...)**; como a nova shell tinha associado o canal 1 (**stdout**) ao ficheiro, quando o programa escrever no canal standard de saída, a escrita é efectuada no ficheiro
 - A shell “pai” espera com **wait()** que o **prog** termine

A shell e a redirecção de canais (2)

```
// Pseudo-código duma shell; exemplo de redirecção do canal 1 apenas!
// background= 1 se colocamos um & na linha
// redirigir2= 1 se colocamos um > na linha

int main()
{
    ...
    lerComando(); ...
    if (comandoExterno)
        if (fork())
            if (!background) wait(...); // shell pai espera pelo filho
        else { // shell filho
            if (redirigir2) {
                close(1); open(ficheiro, O_WRLNLY|O_CREATE|O_TRUNC|0660);
                execlp(programa,...);
            }
        }
    ...
}
```

Para pensar...

- Considero o seguinte comando: `$ prog < ficheiro`
onde `prog` é um programa que lê uma frase do canal de
entrada standard...
- Refine o pseudo-código anterior para suportar a redirecção de
canais de entrada.