Fundamentos de Sistemas de Operação

Unix Windows NT Netware MacOS DOS/VS Vax/VMS
Linux Solaris HP/UX AIX Mach
Chorus

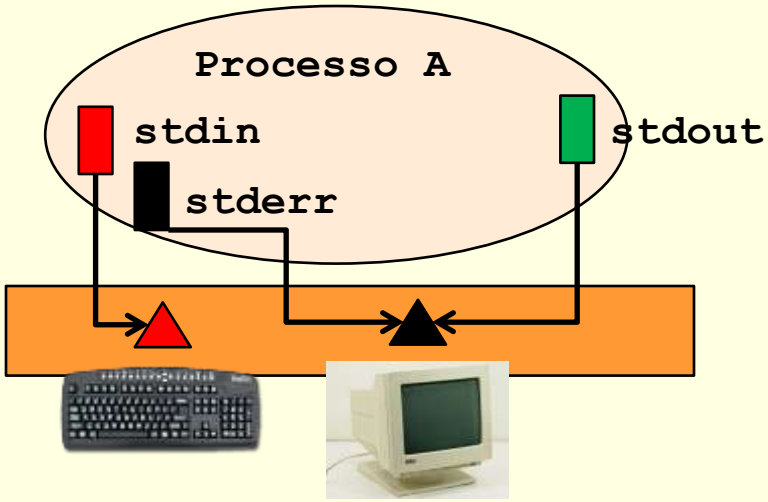
Programas, Processos e o SO:

Redirecção de Canais de E/S

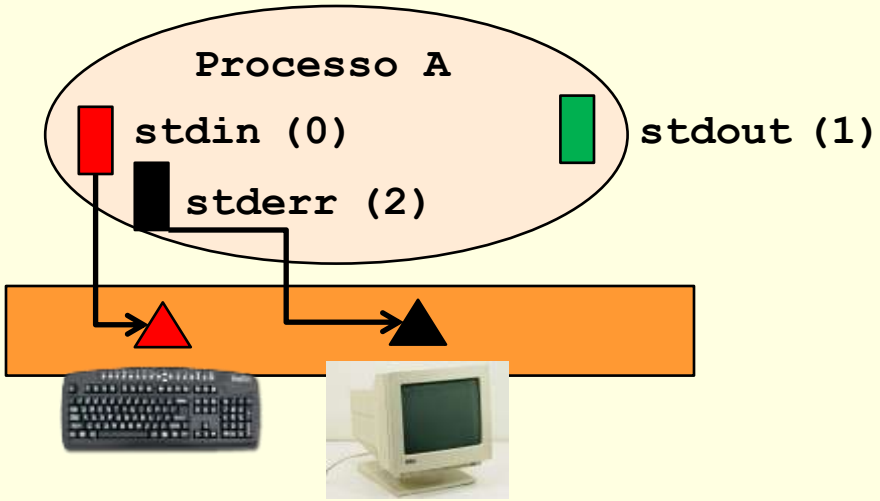
Resumo da aula anterior…

- *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.*
  - *Quando um processo invoca um* **`fork()`** *cria um processo-filho que corre o mesmo programa e herda o espaço de endereçamento e estado do pai: valores de variáveis, variáveis de ambiente, canais abertos, etc. Difere do pai no PID, PPID, e valor de retorno do* **`fork()`**.
  - *Quando um processo invoca um* **`exec()`**, *parte do seu espaço de endereçamento (stack, heap, código, variáveis globais) perde-se, sobreposto com o novo executável; sem alteração mantêm-se os canais abertos e as variáveis de ambiente.*

Abertura e Fecho de Canais em Unix (1)

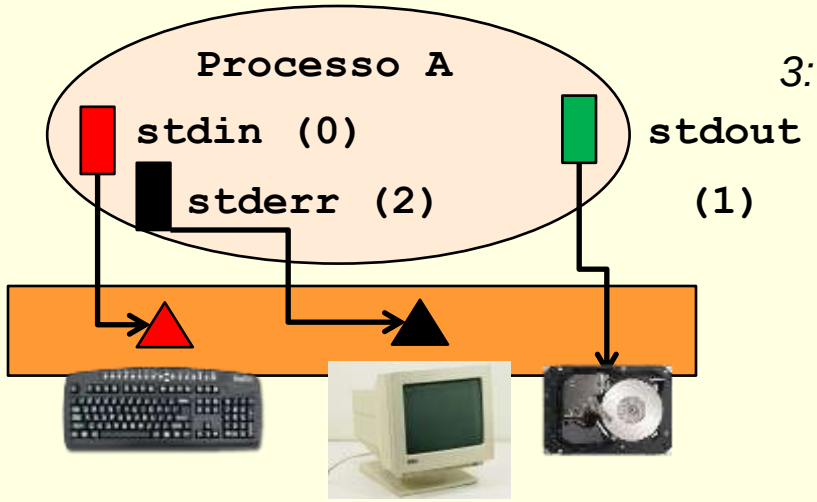


1: Estado inicial:
canais abertos por omissão,
ligados aos periféricos standard.



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

Abertura e Fecho de Canais em Unix (2)



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

```
#define MOD O_WRONLY|O_CREATE\
            O_TRUNC|0660

int main()
{
  ...                          // 1:

  close(1);                    // 2:

  open("ficheiro", MOD);       // 3:
  printf("ola");
  ...
}
```

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

A shell e a redirecção de canais (1)

- *Considere 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* **`execlp("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!

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_WROLNY|O_CREATE|O_TRUNC|0660);
          execlp(programa,...);
      }
    }
  ...
}
```

Para pensar…

- *Considere 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.*

Fundamentos de Sistemas de Operação

Unix Windows NT Netware MacOS DOS/VS Vax/VMS
Linux Solaris HP/UX AIX Mach
Chorus

Programas, Processos e o SO:

Comunicação entre processos: I

Comunicar usando um ficheiro (1)


```
int main()
{ ...
  p=fork();
  ...
  if (p) {                                          // pai escreve
    fd=open("ficheiro", O_WRONLY);
    write(fd, "ola", 3);
  } else {                                          // filho lê
    fd=open("ficheiro", O_RDONLY);
    read(fd, buf, 3);
  }
  ...
}
```

- *Como garantir que o filho só lê depois do pai escrever?*

Comunicar usando um ficheiro (2)


```
int main()
{ ...
  p=fork();
  if (!p) {                                   // filho escreve
    fd=open("ficheiro", O_WRONLY...);
    write(fd, "ola", 3);
  } else {                                    // pai
    wait(NULL);                               // espera...
    fd=open("ficheiro", O_RDONLY...);
    read(fd, buf, 3);                         // ... depois lê
  }
}
```

- *Assim garantimos que o pai só lê depois do filho escrever…*
- *Mas só corre um de cada vez ☹ … má utilização de recursos… CPUs desaproveitados… programa demora mais…*

Comunicação vs. Sincronização

- *Há comunicação entre processos quando*
  - *Há transferência de informação (bytes) entre os espaços de endereçamento dos processos: há emissor(es) e receptor(es).*
- *Há sincronização entre processos quando*
  - *Um processo assinala a outro(s) a ocorrência de um dado evento; por ex., um processo espera que outro termine.*
- *Note-se que há uma certa dualidade entre as duas*
  - *quando um processo espera por uma mensagem de outro, há simultaneamente sincronização e comunicação…*

Pipes (1)

- *Um pipe ("tubo") é uma abstracção do SO que permite a (um, dois ou mais…) processos comunicarem unidireccionalmente*
  - *Numa "extremidade" um descritor para leitura e na "outra extremidade" um descritor para escrita permite a troca de informação (fluxo de bytes)...*

Pipes (2)

- **`int pipe(int fd[2])`**

Se a execução da função tiver sucesso, o conteúdo do vector de dois inteiros **`fd`** *é tal que* **`fd[0]`** *identifica o canal de leitura do pipe enquanto* **`fd[1]`** *identifica o canal de escrita no pipe.*

```
int fd[2];

if ( pipe(fd) == -1) abort();
...
// fd[0] é o canal para ler do pipe
// fd[1] é o canal para escrever no pipe
```

Semântica das operações de E/S sobre pipes (1)

- **`int read(int fd, void *buf, size_t sz)`**

Um **`read()`**, *quando executado sobre um pipe, comporta-se por vezes de forma diferente do que quando executado sobre um ficheiro:*

- *Se não há dados para ler, o processo fica bloqueado, à espera… [no caso do ficheiro retornava zero]*
- *Se a quantidade de dados disponível é inferior à pedida retorna a quantidade existente [tal como no caso do ficheiro]*
- ***Caso especial:*** *se o canal de escrita está fechado e o pipe está vazio, a leitura não bloqueia, retorna zero. É assim que o escritor assinala que não mais quer comunicar…*

Semântica das operações de E/S sobre pipes (2)

- **`int write(int fd, void *buf, size_t sz)`**

Um **`write()`**, *quando executado sobre um pipe, comporta-se por vezes de forma diferente do que quando executado sobre um ficheiro.*

O SO define uma capacidade máxima para o pipe armazenar (ainda que temporariamente) dados

- *Se* **`sz`** *é menor do que a capacidade máxima, mas o espaço disponível no pipe não chega para armazenar os* **`sz`** *bytes, o escritor bloqueia até que um leitor tenha lido o suficiente para que caibam os* **`sz`** *bytes, duma só vez…*
- *Se o valor* **`sz`** *é maior do que a capacidade máxima, apenas uma parte dos dados são escritos (a que couber), e o escritor bloqueia até que um leitor tenha lido alguns bytes… nessa altura são escritos mais alguns…*

Comunicação usando um pipe (1)


```
int main()
{ ...
  if ( pipe(fd) == -1 ) abort();          // cria pipe e abre fd's
  pid=fork();
  if (pid) {
    write(fd[1], "ola", 3);               // pai escreve
    ...
  } else {
    read(fd[0], buf, 3);                  // filho lê
    ...
  }
}
```

- *Pai e filho executam concorrentemente ☺…*
- *O SO sincroniza os processos de forma indirecta, através das operações* **`read()`** *e* **`write()`** *…*

Comunicação usando um pipe (2)


```
int main()
{
  int pid, fd[2];

  if ( pipe(fd) == -1 ) abort();   // cria pipe e abre fd's para R e W
  pid=fork();
  if (pid) {
    close(fd[0]);                  // pai vai W, não precisa de R
    write(fd[1], "ola", 3);        // pai escreve
  } else {
      close(fd[1]);                // filho vai R, não precisa de W
      read(fd[0], buf, 3);         // filho lê
  }
}
```

- *Código praticamente completo! ☺…*