NOVA FACULDADE DE CIÊNCIAS E TECNOLOGIA DEPARTAMENTO DE INFORMÁTICA

Fundamentos de Sistemas de Operação

LEI - 2023/2024

Vitor Duarte
Mª. Cecília Gomes

1

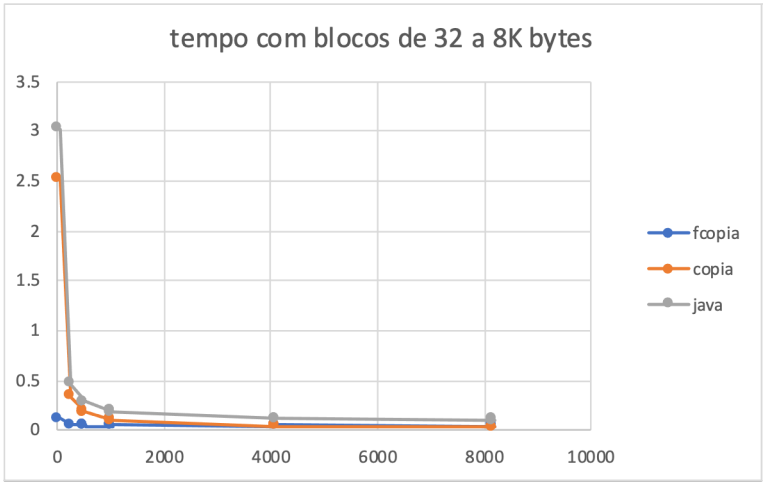
Aula 3

- Lab 1: conclusões
- Processos e ficheiros
  - API para ficheiros e criação de processos
- OSTEP: cap. 4, 5.1, 39.0 - 39.5
- (também de AC: Dive Into Systems cap. 13)

NOVA FACULDADE DE CIÊNCIAS E TECNOLOGIA DEPARTAMENTO DE INFORMÁTICA

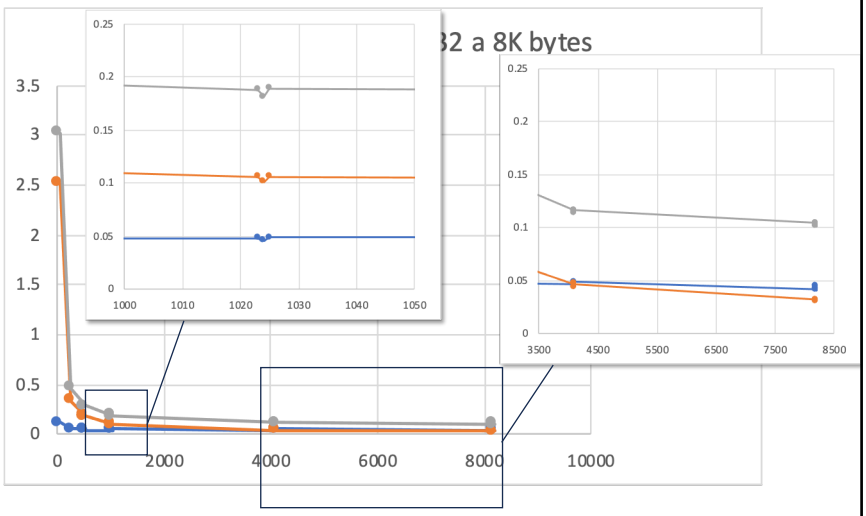
2

Lab 1 – conclusões



3

Lab 1 – conclusões



4

Lab 1 – conclusões

- Nº chamadas READ + WRITE (strace –c)
  - ficheiro de 20MB = 20 971 520 bytes

buffer	fcopia	copia	java
1	10 242	41 943 042	41 943 108
1024	10 242	40 962	41 022
4096	10 242	10 242	10 301

  - libc: usou para FILE buffers de 4K
  - Java: FileInputStream e FileOutputStream não usam buffers

copia.c : *syscalls*

```
...
  int fin = open( argv[2], O_RDONLY );
  if ( fin < 0 ) aborta( "Erro no primeiro ficheiro" );

  int fout = creat( argv[3], 0666 );
  // equivale a open(argv[3], O_CREAT|O_WRONLY|O_TRUNC, 0666 )
  if ( fout < 0 ) aborta( "Erro no segundo ficheiro" );

  while( (nr = read( fin, buf, bsize )) > 0 ) {
    nw = write( fout, buf, nr );
    if ( nw < nr ) aborta( "Erro na escrita" );
  }
  if ( nr < 0 ) aborta( "Erro na leitura" );
  close(if);
  close(of);
...
                    // 0666 = RW for user, group and others
                    // 0110110110b = rw-rw-rw
```

Chamadas SO sobre ficheiros

- **`int open(char *path, int flags, [int mode])`**
  - O SO resolve *path* com base no CWD; verifica se acesso é possível/permitido
  - O SO cria estruturas de dados em memória e traz do disco para a RAM os meta-dados do ficheiro (inode) ou cria o ficheiro
  - Devolve o descritor do ficheiro aberto (*file descritor*) -- índice na tabela de canais do processo, com cursor a 0
  - Isto permite aceder depois de forma eficiente.
- **`int close( int fd )`**
  - O SO é informado que não vai haver mais leituras ou escritas no ficheiro
  - Eventualmente escreve o que falta e liberta o descritor do ficheiro
  - Se já não é precisa a informação sobre o ficheiro, liberta memória.
- NOTA: qualquer chamada ao sistema devolve -1 se houve erro e deixa na var global **`errno`** o número do erro

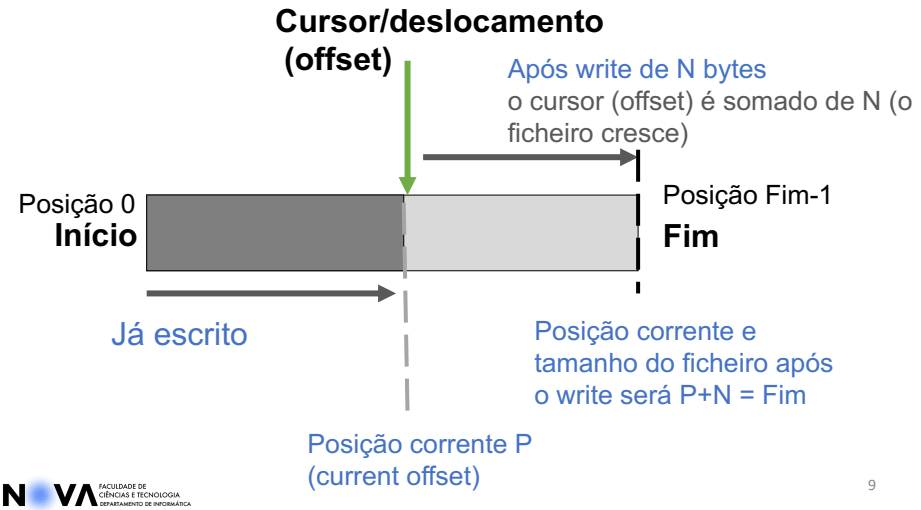
open – flags e mode

- Flags é um conjunto de valores que codifica o acesso pedido:
  - um de O_RDONLY, O_WRONLY, O_RDWR
  - e opcionalmente O_APPEND, O_TRUNC, O_CREAT, ...
  - **`open(path,O_CREAT|O_TRUNC|O_WRONLY, mode)`** é o mesmo que **`create(path, mode)`**
- Quando O_CREAT, em *mode* indica-se o conjunto de modos de acesso permitidos:
  - read, write execute, para cada categoria de utilizadores: o dono, alguém do grupo e outros utilizadores
  - exemplo visto do comando ls:

```
fso@linuxfso:~$ ls -l
total 64
-rwxr-xr-x 1 fso fso 16000 Sep 18 14:52 c
-rw-r--r-- 1 fso fso   212 Sep 18 14:52 c.c
drwxr-xr-x 2 fso fso  4096 Sep 10 19:47 Desktop
-rw-r--r-- 1 fso fso   610 Sep 18 14:36 foo
drwx------ 2 fso fso  4096 Sep 10 20:13 projects
```

Escrita sequencial no ficheiro

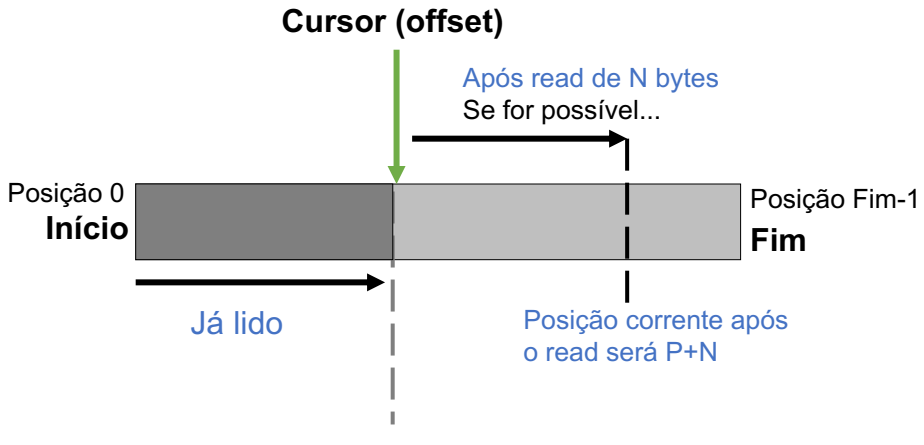
- `int write( int fd, void *data, int n )`



9

Leitura sequencial do ficheiro

- `int read( int fd, void *data, int n )`



10

Helloworld.c

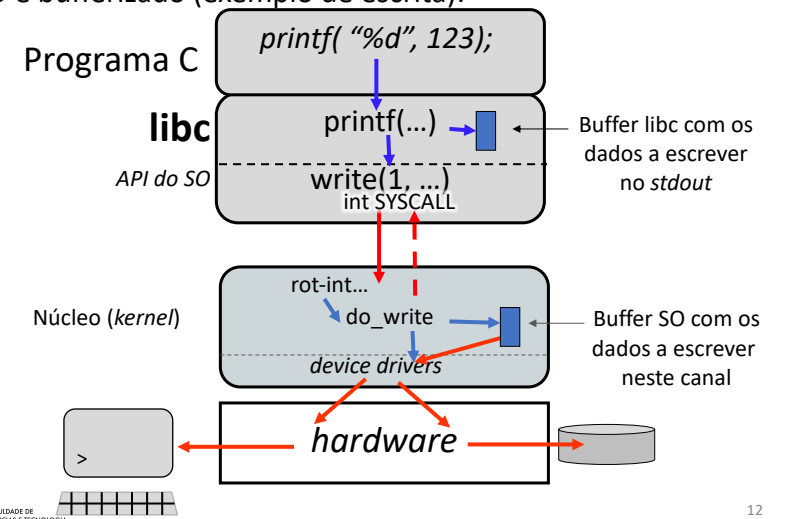
```
#include <stdio.h>
int main( int argc, char*argv[] ) {
   printf("Hello world!\n");
   return 0;
}
```

Dentro da libc:

```
printf(...)
     fprintf( stdout, ... )
          ...
          write( 1, ... ) // interface POSIX
               int 0x80   // no Linux
```


Escrita - exemplo

- IO é bufferizado (exemplo de escrita):



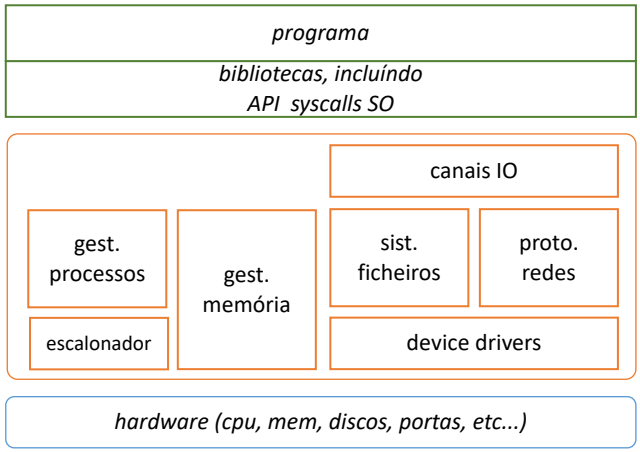
Exemplos de políticas de IO

- As linguagens e o SO permitem escrever nos discos, terminais e outros dispositivos
- Quando é que os dados chegam realmente ao dispositivo depende das politicas que procuram optimizar o IO
  - Nas linguagens, como C: após as conversões necessárias, os bytes ficam num buffer do runtime da linguagem
    - até buffer encher ou, se canal é para ecrã, até '\n'
    - ou até se chamar fflush()
  - Dentro do SO, os bytes são acrescentados ao buffer do canal
    - até este buffer encher ou, se canal em modo terminal, até '\n'
    - ou até se chamar fsync
    - ou até passar um intervalo de tempo, ...
- Nas leituras, a linguagem ou o SO, podem ler mais do que é pedido pelo programador

Métodos de acesso a ficheiros

- Existe sempre uma posição corrente (deslocamento); se o ficheiro tem N bytes, essa posição pode ir de 0 a N-1
- Quando se abre um ficheiro, a posição no ficheiro é 0
- **Acesso Sequencial**: A leitura ou escrita de B bytes, soma B à posição corrente (em princípio)
- Os bytes na zona da posição corrente são mantidos num buffer em memória (janela para o ficheiro)
- **Acesso "Direto"**: Por alteração da posição corrente
  - Pode existir uma operação que permita mudar a posição corrente sem ler ou escrever.
  - Ex: `lseek`, permite alterar ou saber a posição corrente
  - (outros mecanismos, mais tarde...)

Principais componentes dos sistemas



a realidade é bem mais complicada!

Atributos de um processo no SO

- Descritor de processo ou *Process Control Block* (PCB) ou *Task Structure:*
  - Identificador (*process identifier - PID*)
  - Identificador do utilizador (UID)
  - Localização da imagem do programa em memória
    - Código, dados e pilha (e.g. tabela de páginas)
  - Estado do CPU (quando não executa)
    - Conteúdo dos registos
  - Estado do I/O
    - Tabela de descritores canais de I/O: ficheiros abertos e posição em que vai a leitura / escrita
  - Estado de execução (exemplo: READY, RUNNING, BLOCKED, ZOMBIE, ...)
- O gestor de processos gere um conjunto destes PCB (por exemplo num vetor) com todos os processos existentes

Estrutura de dados no SO

Exemplo xv6 (do livro):

```
struct proc {
    char *mem; // or page table
    uint sz;
    char *kstack;
    enum proc_state state;
    int pid;
    struct proc *parent;
    void *chan;
    int killed;
    struct file *ofile[NOFILE];
    struct inode *cwd; //or PWD
    struct context context;
    struct trapframe *tf;
 };
```

```
struct context {
    int eip; int esp; int ebx;
    int ecx; int edx; int esi;
    int edi; int ebp;   };

enum proc_state { UNUSED,
    EMBRYO, SLEEPING, RUNNABLE,
    RUNNING, ZOMBIE  };

struct file {
  enum { FD_NONE, FD_PIPE,
           FD_INODE}type;
  int ref; // reference count
  char readable;
  char writable;
  struct pipe *pipe;
  struct inode *ip;
  uint off;
};
```


open e nomes absolutos vs relativos

- Começa por / → nome absoluto ou completo
- Exemplo: **`/bar/foo/bar.txt`**

(não há confusão com /foo/bar.txt)

- Não começa por / → nome relativo, usa CWD
- Exemplos: se CWD=/bar, os nomes seguintes são equivalentes

`foo/bar.txt`

`./foo/bar.txt`

`../bar/foo/bar.txt`

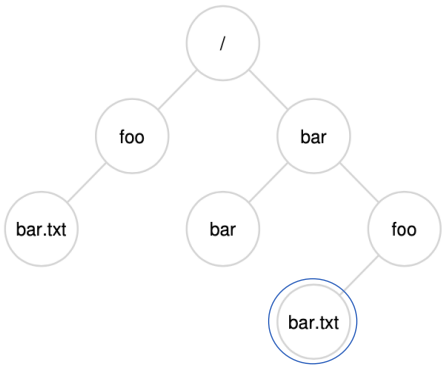
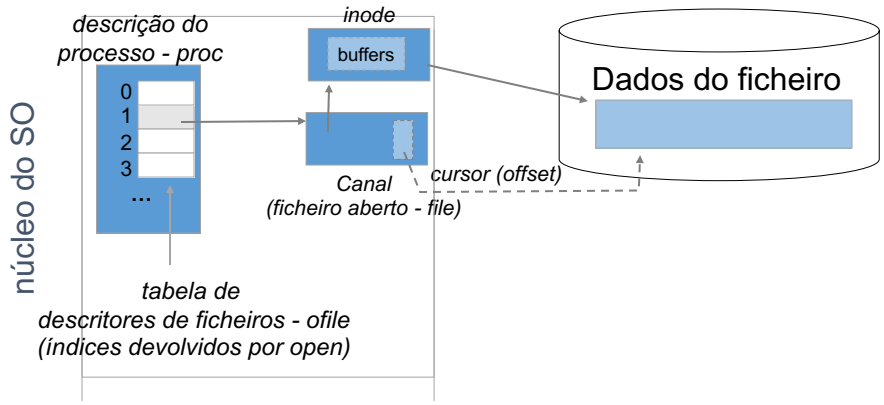


Figure 39.1: **An Example Directory Tree**

Canais de I/O

- Um processo acede a um ficheiro pedindo ao SO um canal de acesso (open). Exemplo:



19

Variáveis do ambiente

- Para além dos atributos do processo, este recebe ainda variáveis de ambiente
- Definem um ambiente "livre" controlável pelo utilizador (pode conter qualquer coisa na forma **`nome=valor`**)
  - exemplos:
    - HOME – *home directory* (diretoria raiz do utilizador)
    - PATH – lista de diretorias (caminhos) onde procurar ficheiros executáveis
    - LANG – língua preferida pelo utilizador
- São definidas:
  - automaticamente no login (HOME, USER, …)
  - ou definidas num ficheiro de inicializações
  - ou definidas pelo utilizador!

20

Variáveis do ambiente (2)

- A lista de variáveis (*Environment list*) é passada ao programa C (em Posix/Unix) através de:
  - Uma variável global: `char *environ[]`
  - Também pode ser um argumento do main:

```
int main(int argc, char *argv[], char *envp[])
```

- Deve ser consultada usando a função da biblioteca:

```
char *getenv( char *name )
```

(norma do C, independente do SO)

Executar um novo programa

- Pedir a criação de um processo:
  - o SO verifica todas as permissões e…
  - cria as estruturas para gerir o processo (PCB)
  - inicia todos os seus atributos e vars ambiente
  - inicia o mapa de memória com o programa (p. ex. a partir de um ficheiro executável)
  - inicia o CPU, transferindo o controlo para o processo criado (IP, SP,…)
- Queremos algo como:

CriarProcesso(ficheiro-exec, canais-iniciais, atributos-proc, vars-ambiente,…etc) ??

Norma da linguagem C

- `int system( char *cmd )`
- na realidade isto faz executar num shell o comando indicado (como se ao terminal) herdando ambiente e atributos do processo corrente
- fica bloqueada até que o comando termine

- continuamos sem saber como se pede ao SO um novo processo concorrente ou com diferentes atributos

Java

```
ProcessBuilder pb = new ProcessBuilder();
pb.command( "myexec" );
pb.environment().put( ... )
pb.directory( ... );
pb.redirectOutput( ... );
. . .
Process newproc = pb.start();
     // ask SO to create the new process
     // ...implementation OS dependent...
```

MS-Windows

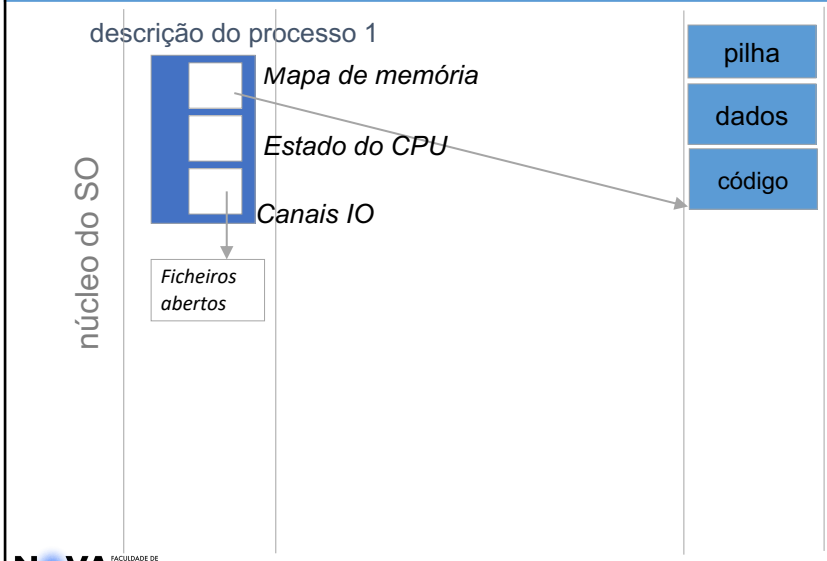
- MS-Windows C#/.net – abordagem como no Java
- MS-Windows (Win32), exemplo de uma das chamadas:

```
BOOL CreateProcess( LPCSTR AppName,
                    LPTSTR CommLine,
                    LPSECURITY_ATTRIBUTES ProcAttr,
                    LPSECURITY_ATTRIBUTES ThAttr,
                    BOOL InheritHandles,
                    DWORD CreatFlags,
                    LPVOID Environment,
                    LPCSTR CurrDirectory,
                    LPSTARTUPINFO StartupInf,
                    LPPROCESS_INFORMATION ProcInfo )
```


Executar novo programa no Unix

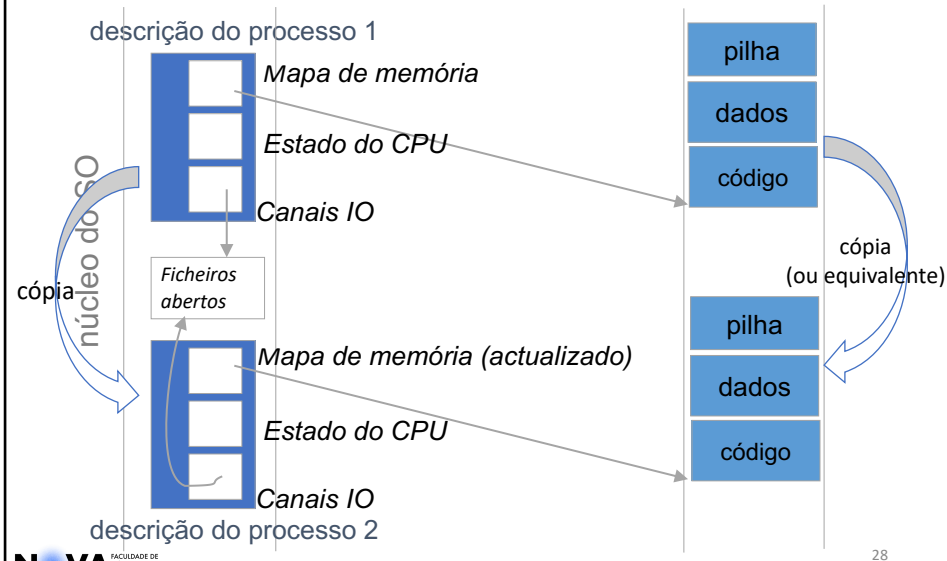
- Abordagem UNIX, envolve duas fases:
- **Fork:** clonar o processo corrente (criar um novo processo)
- **Exec**: substituir a imagem em memória (carregar novo programa)

Processo e sua memória (simplificado)



27

Processos após fork em processo 1



28

Clonagem de processo

- Fork cria um novo processo
  - A sua descrição é idêntica ao processo original
  - Mesmo mapa de memória e CPU, mesmo estado de execução (READY), mesmo UID e permissões, mesmos canais IO, mesma diretoria corrente,...
  - Novo PID, memória virtual cópia da original
- A chamada ao sistema vai retornar para dois processos (original e clone), devolvendo valores diferentes
- O novo processo fica pronto a executar como qualquer outro

Interface Posix/Unix

`int fork(void)`

- cria um novo processo (filho) cópia do original (pai)
  - incluindo: programa, atributos e informação sobre canais abertos (o *pid* é diferente, claro!)
- devolve o *pid* do filho ao pai
  - devolve -1 se erro
- o filho, sendo uma cópia do pai (tem o mesmo IP), começa a executar na instrução imediatamente a seguir ao `fork()`!
  - o SO também tem de devolver um resultado do `fork` no novo processo: devolve zero!