

Fundamentos de Sistemas de Operação

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

*Modos de Acesso a ficheiros:
API e semânticas*

Modos de Acesso

□ Canónico

- O mais simples, e aquele que temos usado até agora: as leituras e escritas são “cached” pelo SO, regra geral aumentando em muito o desempenho. As leituras, beneficiam de *read-ahead* (*ler em avanço, antes da solicitação do utilizador*) e as escritas são dadas como completadas logo que a informação é copiada para a cache.

□ Assíncrono

- As leituras e escritas não são bloqueantes – prosseguem em *background* enquanto o processo (ou thread) continua a sua execução. Existem primitivas que permitem interrogar o SO sobre o estado da operação em curso, e que podem ser usadas quando o programador assim o entender... Há primitivas POSIX para este modo, mas o Linux tem as suas próprias, em alternativa.

Modos de Acesso

□ Síncrono

- Em leitura, idêntico ao canónico. Em escrita, o `write()` não termina enquanto os dados não estão efectivamente guardados em disco; diz-se que a operação é bloqueante. O `open()` inclui a flag `O_SYNC`

□ Directo

- As operações são realizadas sem usar a cache, i.e., decorrem entre o ficheiro e o espaço de memória do processo. O `open()` inclui a flag `O_DIRECT`

□ I/O mapeado em memória

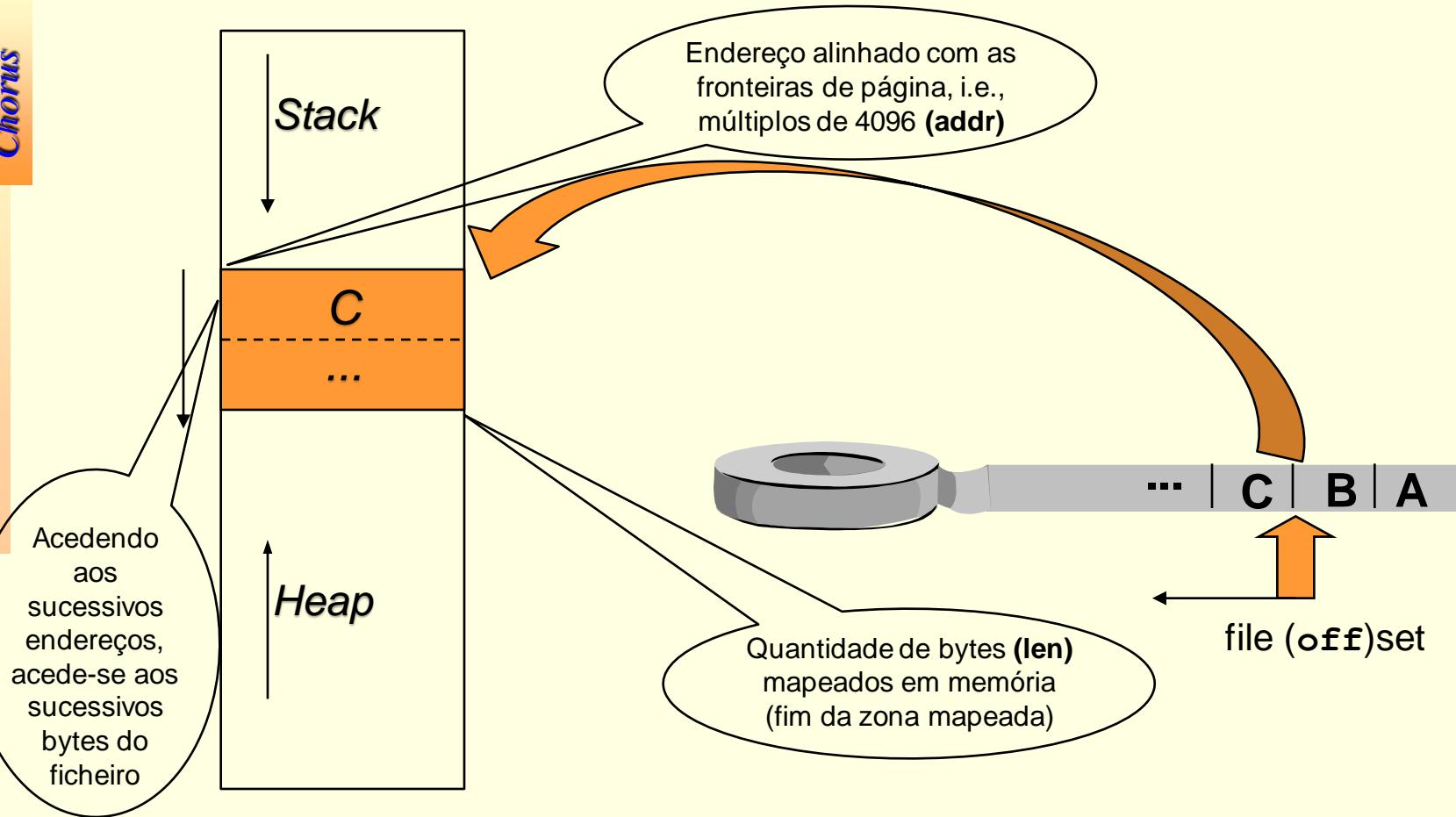
- Não são usadas as primitivas `read()` ou `write()`. Em vez disso, o ficheiro é mapeado numa zona de memória do processo e acedido como se fosse um vector (de bytes).

Fundamentos de Sistemas de Operação

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

*Modos de Acesso:
Ficheiros mapeados em memória*

Ficheiro mapeado em memória (1)



Ficheiro mapeado em memória (2)

- *A primitiva para mapear o ficheiro no EE*

```
void *mmap(void *addr, size_t len, int prot,  
           int flags, int fd, off_t off)
```

addr: endereço de memória no EE do utilizador onde queremos começar o mapeamento do ficheiro (alinhado a um múltiplo de 4096 na arquitectura x86) [Se **NULL** o SO escolhe ele mesmo o endereço]

len: dimensão da zona mapeada

off: início da zona do ficheiro a mapear

fd: file descriptor do ficheiro a mapear (obtido por **open()**)

Ficheiro mapeado em memória (3)

□ Pseudocódigo para ler o ficheiro no EE

Seja: **addr** o endereço de onde começa o mapeamento de um ficheiro de texto e consideremos que **off** foi colocado a zero (representa portanto o início do ficheiro) e que **len** é o comprimento do ficheiro; o seguinte código imprime o conteúdo do ficheiro:

```
char *start= (char *)addr; char *end= start+len;  
for (char *p= start; p < end; p++)  
    putchar(*p);
```

Demo: mmap de um pequeno ficheiro

```
int main(int argc, char *argv[])
{
    int fd, len;
    struct stat st;
    char *start;

    fd= open("ficheiro",O_RDONLY);
    fstat(fd, &st); len=st.st_size;
    if ( len > PAGESIZE ) return 1; //ficheiro maior que 1 pagina,
                                    // não queremos aqui tartar...

    start= (char *)mmap(NULL, len, PROT_READ, MAP_SHARED, fd, 0);

    for (char *end= start+len; start < end; start++) putchar(*start);
    return 0;
}
```

Ficheiro mapeado em memória (4)

- *A primitiva unmap*

```
int munmap(void *addr, size_t len);
```

addr: endereço da página de memória na qual queremos quebrar o mapa entre o *EE* e o ficheiro

len: dimensão da zona mapeada a remover do mapa

Ficheiro mapeado em memória (5)

□ Pseudocódigo com unmap para ler o ficheiro no EE

Consideremos a seguinte situação: o ficheiro é agora maior que o comprimento das páginas mapeadas – aqui, mapeamos apenas 1 página e o ficheiro tem 6000 bytes.

```
addr=(char *)mmap(NULL,4096,PROT_READ,MAP_SHARED,fd,0);
char *end= addr+4096;
for (char *p= addr; p < end; p++)    putchar(*p); //1os bytes
unmap(addr,4096);
remain=6000-4096;
addr=(char *)mmap(NULL,remain,PROT_READ,MAP_SHARED,fd,4096);
char *end= addr+remain;
for (char *p= addr; p < end; p++)    putchar(*p); //restantes
```

Ficheiro mapeado em memória (7)

- Para finalizar... outras *informações* (as mais importantes)
 - Flags das **Permissões** (se necessário, combinar com |)
 - **PROTECT_WRITE**, permite a escrita
 - **PROTECT_NONE**, marca a página como *inválida*
 - **PROTECT_EXECUTE**, permite colocar código e executá-lo
 - Modo de **Partilha** com outros processos que mapeiem a mesma zona do ficheiro
 - **MAP_SHARED**, as alterações são visíveis a outros processos e são escritas no ficheiro
 - **MAP_PRIVATE**, as alterações não são visíveis a outros processos nem escritas no ficheiro – o mapa funciona como espaço COW

(continua)

Ficheiro mapeado em memória (8)

□ Para finalizar... outras informações (continuação)

- *Flags para o modo de Partilha (a combinar com |)*
 - **MAP_ANONYMOUS**, a região mapeada não é suportada (backed) por um ficheiro. Deve colocar-se o `fd == -1` e o `offset == 0`. Pode usar-se para implementar uma região de memória inteiramente controlada por código do utilizador – e.g., um heap “privado”
 - **MAP_LOCKED**, a região mapeada não pode ser paged-out...

Ficheiro mapeado em memória (9)

□ Exercício: resolva o problema geral...

Considere a seguinte situação: seja PMAP número máximo, decidido pelo programa, de páginas a mapear. Para aceder a todo o ficheiro, três casos são possíveis:

- O comprimento do ficheiro é menor que PMAP
 - O comprimento do ficheiro é igual a PMAP
 - O comprimento do ficheiro é maior que PMAP
-
- Faça um programa que cubra as 3 situações....

Fundamentos de Sistemas de Operação

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

*Modos de Acesso:
I/O assíncrono*

I/O assincrono (1)

□ *Relembrando...*

Permite efectuar leituras e escritas não-bloqueantes. i.e., que prosseguem em background enquanto o processo (ou thread) continua a sua execução.

□ *Interface POSIX*

- **`aio_read()`, `aio_write()`**: permitem lançar a operação passando como argumento um AIO Control Block (aiocb)
- *Depois de lançada uma operação, usando o seu aiocb, pode saber-se com `aio_error()` se a operação ainda está em curso, foi cancelada, ou terminou. Nos dois últimos casos pode obter-se o estado da operação com `aio_return()`.*

I/O assincrono (2)

□ Casos de uso

Situações em que se pode usar AIO incluem servidores single-threaded em que é necessário atender pedidos de múltiplos clientes (e enquanto se faz I/O para satisfazer um cliente, tem de se continuar a atender e processar os outros).

□ Programação com AIO

O AIO exige uma forma de programar que é muito diferente da tradicional “de cima para baixo” na qual as instruções que aparecem “em baixo” (no papel ☺ ou no ecrã) são executadas depois das que aparecem “mais acima”.

Quando se usa AIO é vulgar “muito mais abaixo” no programa “perguntar” se uma operação desencadeada “mais acima” já acabou e, nesse caso, executar algum código...

I/O assíncrono (3)

□ Alternativa: threads

- O uso de *threads* aparece como uma alternativa mais simples – uma operação é lançada numa *thread* e, quando necessário, efectua-se o *join...* ou outra operação de sincronização.
- Em situações em que o número de *threads* poderia ser demasiado grande (*milhares*) o que causaria grande consumo de recursos (*TCBs*, escalonamento) pode recorrer-se a *pools* com um certo número de *threads* (*centena*) sendo que cada *thread* processa, ao longo do tempo, pedidos de diferentes clientes...

□ Servidor Web Apache: How it Works

“A single control process (the parent) is responsible for launching child processes. Each child process creates a fixed number of server threads as well as a listener thread which listens for connections and passes them to a server thread for processing when they arrive.”