

# *Fundamentos de Sistemas de Operação*

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

*Acesso concorrente a ficheiros:  
Semântica(s) de Partilha*

# *Alguns cenários...*

## □ **Cenário 1:** Dois processos numa relação pai-filho

- Aqui, assume-se que o processo-pai abriu um ficheiro e depois fez um `fork()`. Os dois processos leem e escrevem sobre esse ficheiro.
- **Notas:** este cenário apenas se distingue pelo comportamento particular do file pointer (`fp`): como é partilhado, as alterações de posição decorrentes de operações realizadas por um dos processos reflectem-se no outro...

## □ **Cenário 2:** Dois processos independentes

- Aqui, dois processos não-relacionados (i.e., um não descende via `fork()` do outro) leem e escrevem sobre um mesmo ficheiro (como tal, os fps são independentes).

# *Semântica de Partilha (1)*

## □ O que é?

- *Define a forma como o Sistema de Ficheiros (ou o SO, se este obriga todos os SFs a terem o mesmo comportamento) oferece/torna visível aos processos que partilham um ficheiro o resultado de uma operação realizada por um deles*

## □ Semântica de Partilha no UNIX

- *As escritas são atómicas. Isto quer dizer que se, e.g., dois processos fazem concorrentemente cada um o seu `write()` e as regiões a serem escritas se sobrepõem, total ou parcialmente, o resultado final é o mesmo que se obteria se um fizesse um `write()` primeiro e outro depois – i.e., na região sobreposta não é possível encontrar uma mistura dos dados de um com os de outro (a execução corresponde a uma serialização dos `write()`)*

# Semântica de Partilha (2)

## □ Semântica de Partilha no UNIX

(cont.)

- (...As escritas são atómicas) Isto quer dizer que se, e.g., um processo faz um `write()` e outros fazem concorrentemente `read()` e as regiões lidas se sobrepõem, total ou parcialmente, à escrita, o resultado final é o mesmo que se obteria se um fizesse o `write()` primeiro e os `read()` depois, ou vice-versa – i.e., na leitura ou se encontram os dados presentes antes do `write()`, ou depois (a execução corresponde a uma serialização do `write()` em relação aos `read()`)
- **Nota:** no Linux, em ext2 (pelo menos!) a serialização faz-se entre escritas, mas não entre escritas e leituras.

# Semântica de Partilha (3)

## □ Semântica de partilha POSIX: ATENÇÃO

- “*This volume of IEEE Std 1003.1-2001 does not specify the behaviour of concurrent writes to a file from multiple processes. Applications should use some form of concurrency control.*”

## □ Contudo...

- “*...after a write() to a regular file has successfully returned, any successful read() from each byte position in the file that was modified by the write shall return the data specified by that write() for that position until such byte positions are again modified, and any subsequent successful write() to the same byte positions in the file shall overwrite that file data.*”

(continua)

# *Semântica de Partilha (4)*

## □ *Semântica de partilha POSIX (e ainda...)*

- “*I/O is intended to be atomic to ordinary files and pipes and FIFOs. Atomic means that all the bytes from a single operation that started out together end up together, without interleaving from other I/O operations.*”,
- Nota: Retirado do Standard IEEE POSIX 1003.1-2001.

## □ *Outras semânticas (apenas como nota)*

- *Ficheiros imutáveis e SFs com suporte de versões*
- *Sessão*
- *Transacção*

# *Locking em Ficheiros* (1)

## □ *Trincos em ficheiros*

- São de dois tipos: partilhados, ou de leitura, e exclusivos, ou de escrita. Definem uma região trancada sobre um ficheiro especificando-a com dois file offsets: origem e comprimento.
- Se uma região está trancada para leitura por um dado processo, então qualquer outro trinco para leitura que cubra total ou parcialmente a região colocado por outro processo é aceite e “sobreposto” (stacked) sobre o anterior.
- Um processo só consegue colocar um trinco para escrita numa região se não há nenhum trinco colocado por outro processo a cobrir, total ou parcialmente, essa região.

# *Locking em Ficheiros (2)*

- *Trincos recomendados (advisory locks)*
  - *Implementados com a chamada `fcntl()`, necessitam que todos os processos os usem para serem efectivos. Se um processo acede ao(s) ficheiro partilhado sem usar a chamada `fcntl()`, o SF não verifica as operações e o funcionamento pode ser incorrecto.*
- *Trincos obrigatórios (mandatory locks)*
  - *Implementados com a chamada `fcntl()` sobre ficheiros marcados (nos bits de permissões) como configurados para “mandatory locking” e o volume (“disco”) foi montado com a opção para suportar “mandatory locking”.*

# *Locking em Ficheiros (3)*

## □ Quando o trinco “não é aceite”...

- No caso mais comum de lock, que é com opção de espera (`F_SETLKW`) o processo que tenta trancar a região é bloqueado até que o trinco possa ser colocado. Contudo, se o ficheiro foi aberto para utilização de operações não-bloqueantes (`O_NONBLOCK`), é retornado o erro **EAGAIN** (isto é, “tente novamente” ☺).
- Se foi usada a opção de “não esperar” (`F_SETLKW`) é retornado o erro “Resource temporarily unavailable”

# *Locking em Ficheiros (4)*

## □ O tipo “lock”

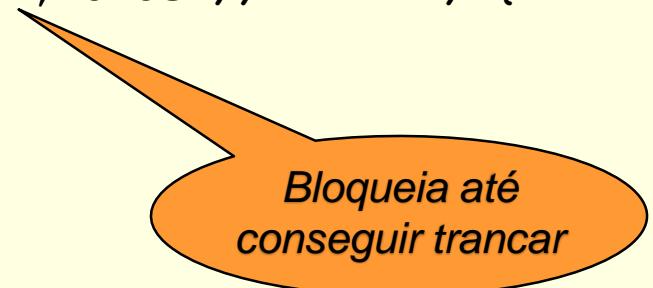
```
struct flock {  
    short l_type;  
    short l_whence;  
    __kernel_off_t l_start;  
    __kernel_off_t l_len;  
    __kernel_pid_t l_pid;  
    __ARCH_FLOCK_PAD  
};
```

Para que a dimensão total  
da estrutura seja eficiente  
na CPU cache

# *Locking em Ficheiros: exemplos (1)*

- Colocar um *write lock*, no *offset 0*, trancando *20 bytes*.

```
lock.l_type = F_WRLCK;  
lock.l_start = 0;  
lock.l_whence = SEEK_SET;  
lock.l_len = 20;  
lock.l_pid = getpid();  
  
if ( (retc = fcntl(fd, F_SETLKW, &lock)) == -1 ) {  
    perror("F_SETLKW:: ");  
    exit(1);  
}
```



Bloqueia até  
conseguir trancar

# *Locking em Ficheiros: exemplos (2)*

- Remover um *lock*, que vai do *offset 0*, ao *20*.

```
lock.l_type = F_UNLCK;
lock.l_start = 0;
lock.l_whence = SEEK_SET;
lock.l_len = 20;
lock.l_pid = getpid();

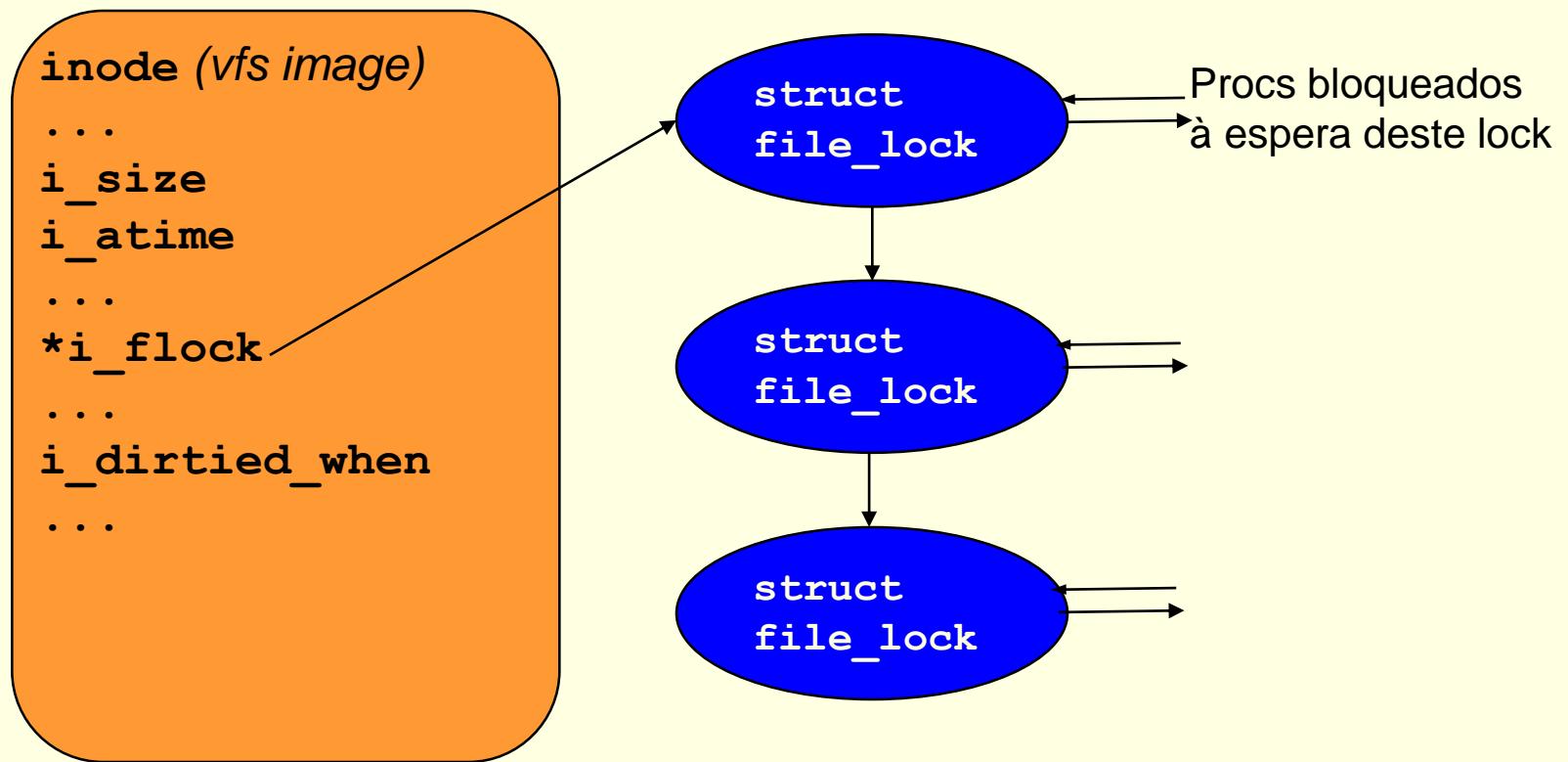
if ( (retc = fcntl(fd, F_SETLKW, &lock)) == -1 ) {
    perror("F_SETLKW:: ");
    exit(1);
}
```

# Demo: Locking em Ficheiros

- Um processo A,
  - Cria um ficheiro *F*, mete um lock e escreve 20 bytes.
  - Aguarda...
  - Retira o lock e termina
- Um processo B (versão 1 e versão 2)
  - Abre o ficheiro *F* em leitura, mete um lock... e
    - Versão 1: bloqueia, espera pelo unlock de A, e termina
    - Versão 2: retorna erro “Resource temporarily unavailable” e termina

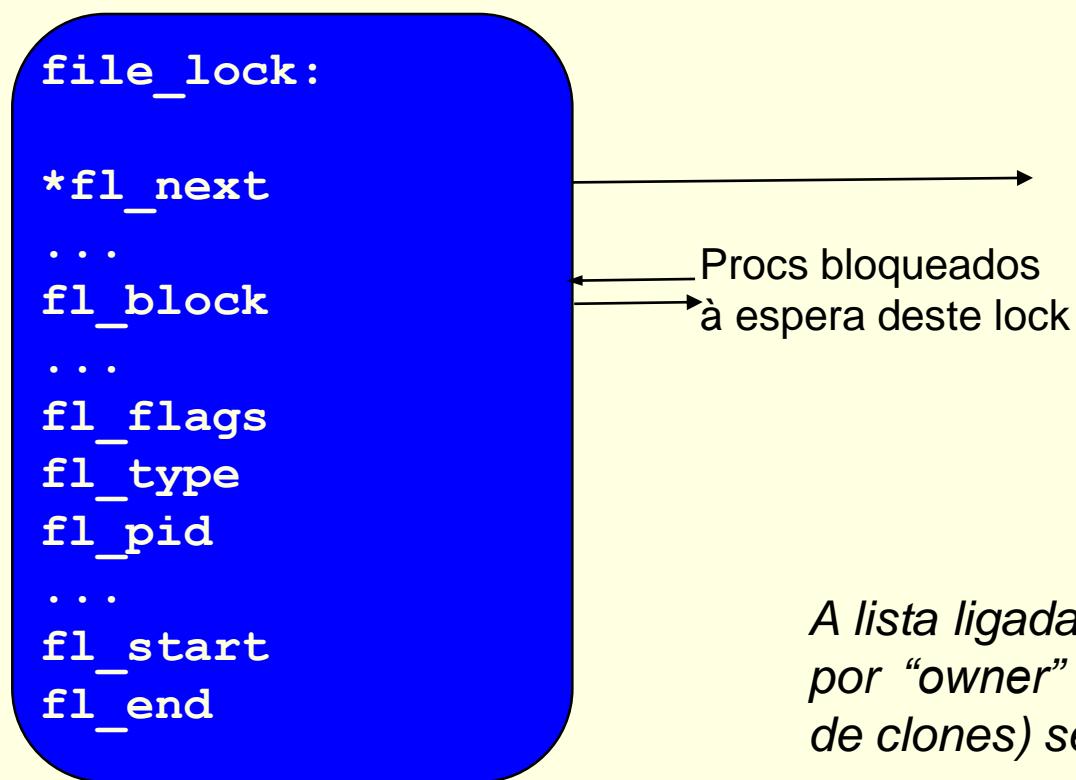
# *File locks no kernel* (1)

- A implementação dos *file locks* no kernel é “simples”:



# *File locks no kernel (2)*

## □ Estrutura do kernel file\_lock:



A lista ligada é mantida ordenada por “owner” (processo ou grupo de clones) seguido de start e end