FACULDADE DE CIÊNCIAS E TECNOLOGIA
DEPARTAMENTO DE INFORMÁTICA

Fundamentos de Sistemas de Operação

LEI - 2023/2024

Vitor Duarte
Mª. Cecília Gomes

1

Aula 15

Parte I

- Gestão de memória: otimizações
- OSTEP: cap. 22.9-22.11

Parte II

- Introdução do Sistema de ficheiros
- OSTEP: cap. 36.7, 39.0-39.12

FACULDADE DE CIÊNCIAS E TECNOLOGIA
DEPARTAMENTO DE INFORMÁTICA

2

Variantes: tentar reduzir swap outs

- Despejar uma página escrita (*dirty*) vai obrigar a swap-out, o que é mais demorado e usa mais disco de swap
- Variante: ter preferência pelas não escritas
  - Primeiro só escolhe se frame não acedida e não escrita, mas memoriza a primeira não acedida e escrita
  - Se não encontrar nenhuma não escrita, usa a memorizada

Variantes: tentar otimizar IO

- O carregamento de cada página *on-demand* pode não ser o mais eficiente
  - no início de um novo programa vamos ter muitos *page-faults*
  - o tamanho da página pode não ser o mais eficiente para leituras ou escritas nos discos
  - quando é preciso libertar *frames*, pode ser necessário várias
- Variante: tentar agrupar páginas para fazer IO com blocos maiores
  - Nos *page-in*, se é provável precisar das páginas seguintes, fazer prefetching
  - Tentar agrupar páginas a fazer swap para num pedido ao disco escrever tudo

Reservatório (pool) de páginas livres

- O SO procura ter uma reserva de “frames” livres:
  - O SO procura garantir um nível mínimo de frames livres (watermark)
  - Quando é necessário libertar frames, o SO atribui frames livres da reserva e, em “background”, vai libertando outras frames para manter a reserva
  - Permite que o swap-out decorra enquanto executa os processos
- Todas as “frames” vão primeiro para a pool
  - o conteúdo é mantido e só altera quando for necessário usar → posta a zeros ou lida de swap ou de ficheiro
  - se a página que estava numa frame volta a ser necessária, reutiliza a frame que já tem o conteúdo da página (evita swap-in)

O n° de faltas de página

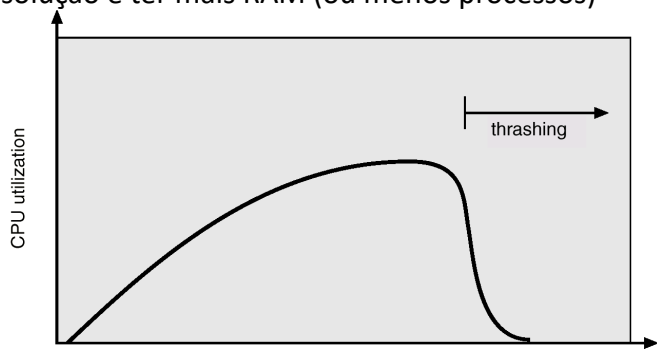
- O nº de faltas de página está relacionado com o nº de páginas atribuídas
- Se um processo tem todas as páginas que necessita (working set) em RAM não incorre em falta de páginas
- À medida que vai tendo menos páginas (devido a outros processos), ou o working set muda, o desempenho degrada-se
- No limite, o processo tende a estar sempre no estado BLOCKED à espera de ter uma página em memória.

Thrashing na gestão de memória

- Se vários processos não têm páginas "suficientes", o ritmo de faltas de página aumenta muito (é necessário mais memória que a disponível).
- Isto é caracterizado por:
  - Baixa taxa de ocupação do CPU.
  - Grande número de operações de I/O sobre o disco de paginação.
  - Poucas "frames" livres; espaço de swap muito ocupado
- **Thrashing** ≡ os processos estão praticamente sempre à espera que o SO carregue páginas (page in) e transfira páginas para o disco (page out).

Thrashing

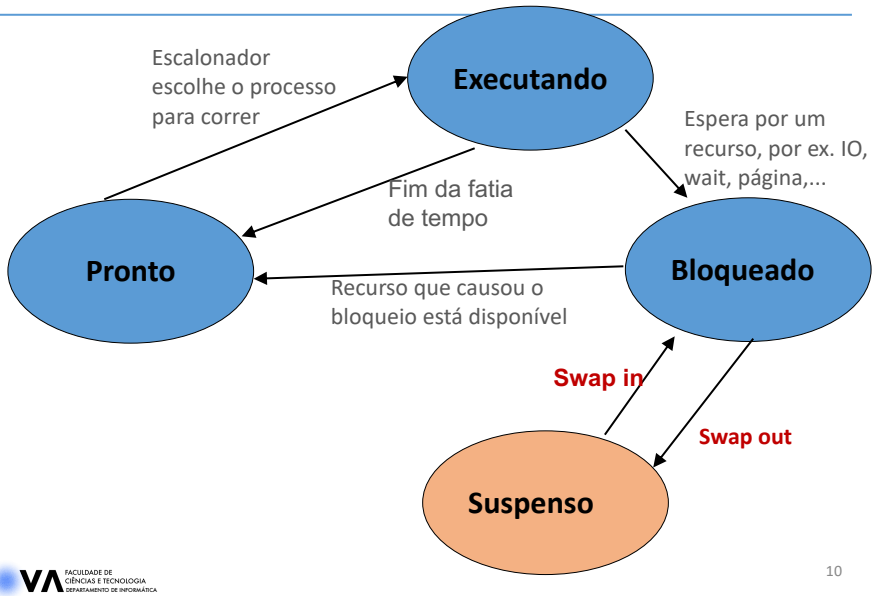
- À medida que o número de processos aumenta, (ou os working sets aumentem) o número de páginas para cada um diminui
  - o multiprocessamento diminui
- A solução é ter mais RAM (ou menos processos)



Process Swapping

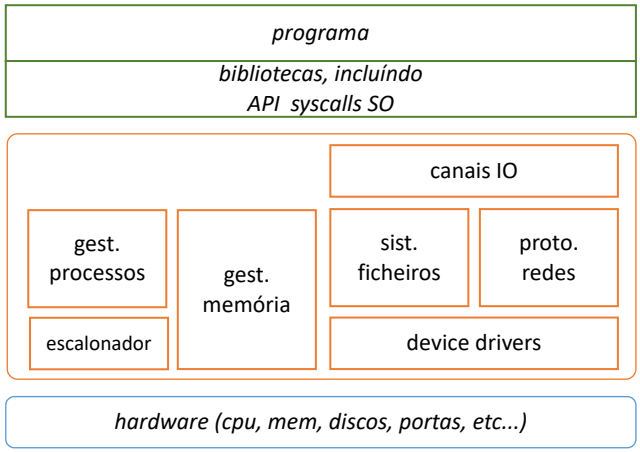
- Em situações críticas de escassez de páginas, o SO pode decidir suspender processos
  - P.ex. o(s) que provocam mais *swapping*, retirando-lhe todas as páginas em RAM (process swap out)
  - No Linux, em casos drásticos, **termina** (kill) um ou mais processos
- As frames libertadas são usadas pelos outros processos
  - Diminui o multiprocessamento para melhorar o uso do CPU, reduzir *trashing* e assim terminar mais rapidamente alguns dos processos
- Desaparecida a situação de escassez de páginas, o processo suspenso volta a ter páginas em RAM (swap in)

Process Swapping



11

Principais componentes dos sistemas



a realidade é bem mais complicada!

12

Gestão do Sistema de ficheiros

- Sistema para gestão da informação guardada (normalmente) de forma persistente
  - Exemplo típico: informação gravada num disco!
  - Tem também de gerir o espaço livre nesses dispositivos periféricos
- Tem de identificar e permitir recuperar a informação:
  - Associa identificação ou nome: "ficheiro", "/home/user/f.txt", etc.
  - Indica o dispositivo de suporte (e respetivo device-driver) e endereços dentro desse dispositivo
  - Outra meta-informação: dimensão, dono, permissões, ...
- Oferece API de chamadas ao sistema com modelo descritor e canal (ou *stream*)
  - Operações típicas: open, read, write, close, mais algumas extensões para acomodar certas classes de periféricos
  - Outras interfaces para outras abstrações do SO com nome, como periféricos genéricos e outras para objetos temporários (p.ex. pipes , semáforos, memória partilhada, ...)

Tipos de dispositivos

- Orientado ao bloco
  - dada a sua capacidade os endereços dentro do periférico referem-se a grandes blocos de bytes. Exemplo: os discos (setores de 512, 1024, ...)
  - podemos endereçar estes blocos (cada um tem um endereço)
  - O IO mais eficiente pode ser com grupos desses blocos. Exemplo: nos discos é típico os melhores desempenhos serem ler ou escrever vários setores contíguos
- Orientado ao byte (ou *streams*)
  - periféricos que permitem IO de sequências de bytes de/para periféricos
  - têm nome no sistema de ficheiros mas a leitura ou escrita é sempre em *stream* e sem possibilidade de endereçar bytes específicos. Exemplos: teclado, rato, portas série, ...

Utilização dos periféricos de bloco

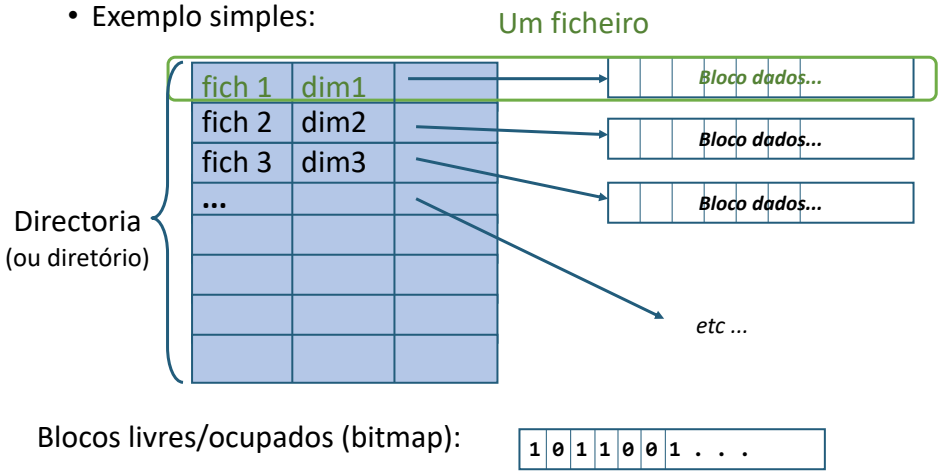
- Os processos normalmente só podem usar os blocos através da abstração de **ficheiro** do SF que o gere.
  - O nome do ficheiro identifica o dispositivo e os blocos atribuídos para o seu conteúdo
  - No open, o SF converte o nome num objeto interno ao SF que identifica o dispositivo e os blocos com os dados do ficheiro
  - Nota: alguns processos podem necessitar de acesso direto a uma interface de blocos (p.e. verificar consistência do SF, ...)
- As leituras e escritas são mapeadas em blocos geridos pelo SO e lidos/escritos no respectivo dispositivo
  - um *device-driver* esconde do SF os detalhes de cada periférico, oferecendo uma API para ler e escrever cada bloco do periférico
- Sistema de memória virtual usa uma interface de blocos interna ao *kernel* para ler e escrever conteúdos de páginas nesses dispositivos

Como fazer um SF?

- Como guardar os nomes e respetiva informação associada?
  - Onde está a informação pretendida? Que dispositivo ou abstração do SO?
- Como gerir (eficientemente) o espaço em disco?
  - Que blocos estão ocupados/livres?
- Respostas:
  - Estruturas de dados mantém a identificação (nome) e respectiva informação associada (para os ficheiros, directorias, etc): meta-dados e dados
  - Outras estruturas de dados descrevem a ocupação do disco, para a sua gestão
  - Estas ED têm de existir em memória do kernel durante a sua utilização e estar gravas em disco para persistirem (num formato próprio)

Um sistema de ficheiros muito simples

- Exemplo simples:



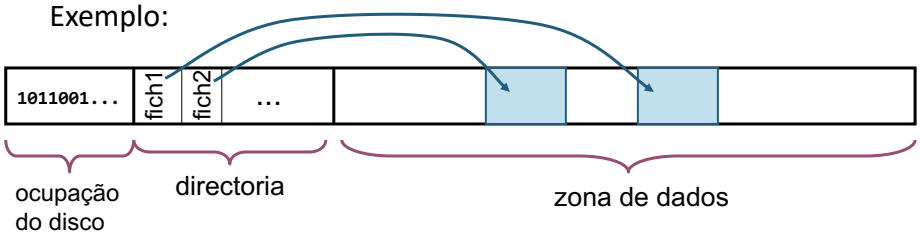
Blocos livres/ocupados (bitmap):

17

Um sistema de ficheiros no disco

- **Volume** = parte de um disco ou agrupamento de discos, usado como o suporte para guardar um SF
- As estruturas de dados têm de ser guardadas no volume a que dizem respeito, num formato conhecido -> **formatação** do volume
- Só o SO manipula estas estruturas!

Exemplo:



18

Observações

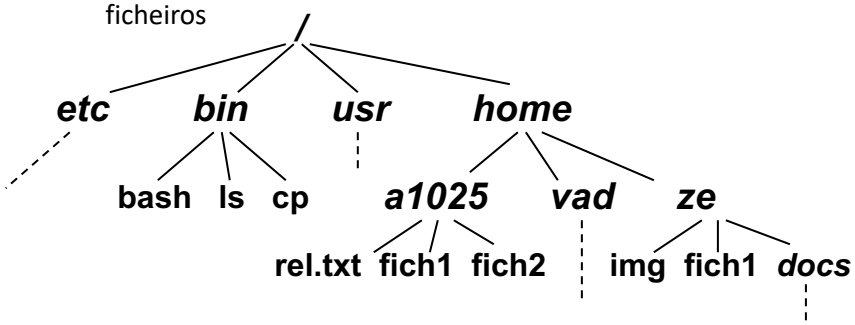
- Nem todo o espaço em disco é para dados
- O acesso a um ficheiro exige ver primeiro a diretoria
- Se a diretoria desaparece ou tem erros perdem-se ficheiros
  - Qualquer erro nas estruturas de gestão podem levar a perder/corromper ficheiros ou espaço livre
- Como atribuir vários blocos a um ficheiro? Como crescer/diminuir?
  - No exemplo anterior é muito limitado
  - Alternativas: blocos contíguos? Lista de blocos? Fragmentação dos ficheiros no disco é problema?
- Onde guardar metadados (dono, permissões, etc)?
  - No exemplo anterior só tem dimensão
  - Alternativas: aumentando a diretoria? Estrutura separada?

No UNIX...

- Um ficheiro é uma sequência de bytes (*stream*) com um nome (em disco ou num outro dispositivo)
- Os processos acedem via canais de I/O
  - identificados por descritores
  - abrir ficheiro: associar-lhe buffers e um cursor (offset)
- Os ficheiros/canais são abstrações genéricas:
  - Dados em disco, em memória, em disquete, em CD, noutras máquinas, portas série, etc...
- Para o processo (programador) é uma abstração com uma API comum
  - *(nem sempre ...)*

Visão do utilizador: espaço de nomes

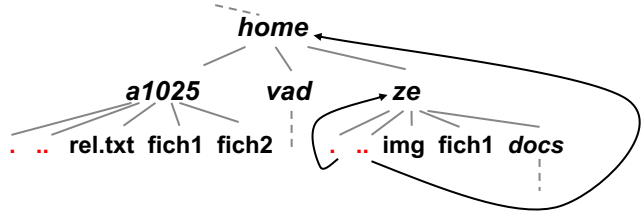
- Espaço de nomes hierárquico
- Cada nome (nó) pode ser uma diretoria, um ficheiro em disco ou noutro dispositivo...
  - Cada diretoria contém uma lista de outras diretorias e/ou ficheiros



21

Nomes de ficheiros e diretorias (2)

- Cada diretoria tem sempre duas outras:
  - . (a própria) .. (a superior)
- Qualquer nome iniciado por '.' pode não ser mostrado pelo comando ls
- Cada processo tem uma diretoria de trabalho corrente (CWD) para podermos usar nomes relativos:
  - Exemplo: se a diretoria corrente é /home/ze
    - /home/ze/fich1 = ./fich1 = fich1
    - /home/a1025/fich1 = ../a1025/fich1



22

Um ficheiro em disco

- Nome, numa diretoria
- Conteúdo é um conjunto de dados (sequência de bytes) de tamanho arbitrário
  - Pode estar espalhado por blocos no disco
- Meta-dados (descreve o ficheiro):
  - Índices dos blocos contendo os dados
  - Tamanho
  - Utilizador dono e grupo
  - Permissões de acesso
    - (ou modos de acesso)
  - etc…

i-node

- Idem para diretorias ou outros nós do SF

Obtendo informação (stat)

```
int stat(char *path, struct stat *info)
int fstat(int fd, struct stat *info)
```

Coloca em *info* informação obtida tipicamente do *inode* :

```
struct stat {
    dev_t  st_dev;  /* ID of device */
    ino_t  st_ino;  /* inode number */
    mode_t st_mode; /* protection */
    uid_t  st_uid;  /* user ID of owner */
    off_t  st_size; /* total size, in bytes */
    time_t st_mtime;/* time of last modific.*/
    nlink_t st_nlink;/* number of hard links */
    . . .
};
```

Criação/abertura/fecho de um ficheiro

`int open(char *path, int flags)`

verifica path, lê para memória o i-node, verifica permissões

se tudo bem, cria canal/descritor que devolve

`int creat(char *filename, int mode)` ou

`int open(char *filename,int flags,int mode)`

onde `flags` inclui **`O_CREAT`**

se poder cria o nome na diretoria e depois faz o open anterior

`int close( int fd )`

liberta descritor, se último a usar liberta canal (offset), se último a usar, liberta i-node

se em escrita garante 1º que conteúdo no buffer e alterações ao i-node são escritas em disco

Leituras / escritas / alterar posição

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

copia data para buffer do canal, atualiza cursor e i-node

se "necessário" escreve para disco

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

pede leitura de n bytes para data

se bytes no buffer, copia, se não pede ao driver (disco) o bloco respetivo tendo em conta o cursor, para o buffer (bloqueia processo)

`int fsync( int fd )`

pede escrita do que está em buffer para disco

`int lseek( int fd, int offset, int whence )`

altera cursor do canal

Diretorias

- São como "ficheiros especiais"
  - As permissões associadas têm um significado ligeiramente diferente: x = direitos de entrar na dir (aceder a nomes debaixo da diretoria)
- São lidas por funções próprias:
  - **`opendir`**, **`readdir`**, **`closedir`**
  - Interpretam o formato da diretoria do sistema de ficheiros
- São alteradas por chamadas próprias

  exemplos: **`creat`** p/ficheiros (ou `open` com `O_CREAT`)
  `unlink` p/ficheiros (apagar nome desta diretoria)
  `rename` (alterar nome)
  `mkdir`, **`rmdir`**, etc

Funções de acesso a directorias

```
DIR *opendir(char *dname);
struct dirent *readdir(DIR *dp);
```

- ler uma entrada (de cada vez)
- devolve NULL quando atinge o fim

- O tipo DIR é interno à interface que mantém informação sobre a directoria, como uma lista de entradas 'dirent'

```
struct dirent {
       inot_t d_ino;       /*i-node*/
       char d_name[NAME_MAX+1];
}
```

Exemplo: listar diretoria

```
int main(int argc, char *argv[]) {
   DIR *dp = opendir(".");  // cwd
   if (dp==NULL) abort();
   struct dirent *d;
   while ((d = readdir(dp)) != NULL) {
     printf("%lu %s\n",
             (unsigned long)d->d_ino,d->d_name);
   }
   closedir(dp);
   return 0;
}
```