Sistemas Operativos: Introduction

Part II

February 19, 2020

Sumário

Sumário

Files

- Abstract/virtualize disk space
  - Actually, they abstract data source/sink
  - In the Unix worlld, they are use to virtualize other I/O devices
- Support 3 basic operations: read, write e execute (only if an executable program)
- Typically organized hierarchically using directories/folders:

  Files that contain other files
- Each file/directory has a user that is its owner
  - This information is used for access control

File System

- Files are organized in file systems
- Set a files
  - Determined mostly for their implementation, not so much for some higher-level relationship among the files
  - An OS may have serveral file systems
    - Check `mount`
- It is also used to denote the OS module that manages the file system
- To operate on a file it must have a name

File Names

Identifier is unic in the context of the file system to which it belongs. This is usually, implementation dependent.

Path name string that locates the file in the computer file system

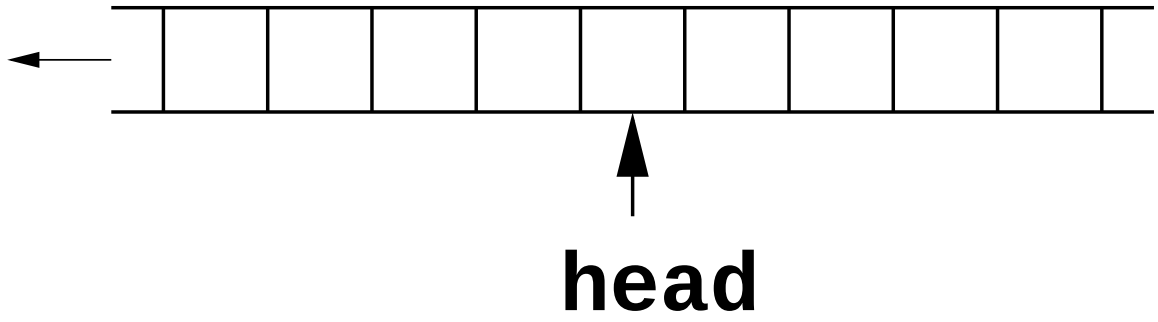
- **Absolute** location of the file wrt the filesystem root
- **Relative** location of the file wrt t present working directory

File descriptor a number that identifies an opened file in the context of the process that opened it

- The OS keeps for each process, a table with the files opened, and not yet closed, by that process

File Access Model

- Sequential (magnetic tape)



  - in order to access to a byte/register, process must access all bytes/registers that precede it
  - it is not possible to jump back and forwadnão é possível saltar para a frente ou para trás.
- Direct/random
  - supports positioning the reading/writing header in any byte/register of the file;
  - not all I/O devices support this kind of access
    - for example, the serial port does not.

System Calls

Issue: How do you access to the file system?

Solution: Using system calls (or utility programs):

- The system calls are the application program interface (API) of an OS
- For each type of service/abstraction the OS offers a set of system calls
- For a programmer, invoking a system call, is just like invoking one function (`/usr/share/man/man2`).

System calls: File Access

Operation	Unix system call
Create	`open()` (`creat()` – obsolete)
Read	`read()`
Write	`write()`
Reposition the head	`lseek()`
Reading attributes	`fstat()`, `lstat()`, `stat()`
Changing atributos	`chmod()`, `chown()` ...
Memory mapping	`mmap()`, `munmap`

- In Unix:
  - **must** call `open()` before file access;
    - OS must execute several actions so that subsequent operations on that file are faster
  - **must** call `close()` when you are done accessing a file
    - Allows the OS to release some resources

Protótipo das Chamadas ao Sistema

```
// To open a file that already exists
int open(const char *pathname, int flags);
// To create a file
int open(const char *pathname, int flags, mode_t mode);
// To close a file
int close(int fd);
// To read data from a file
ssize_t read(int fd, void *buf, size_t count);
// To write data to a file
ssize_t write(int fd, const void *buf, size_t count);
// To position the offset of the file,
//  so that the next read/write syscall will
//  start at that position
off_t lseek(int fd, off_t offset, int whence);
```

Chamadas ao Sistema para Ficheiros: Exemplo

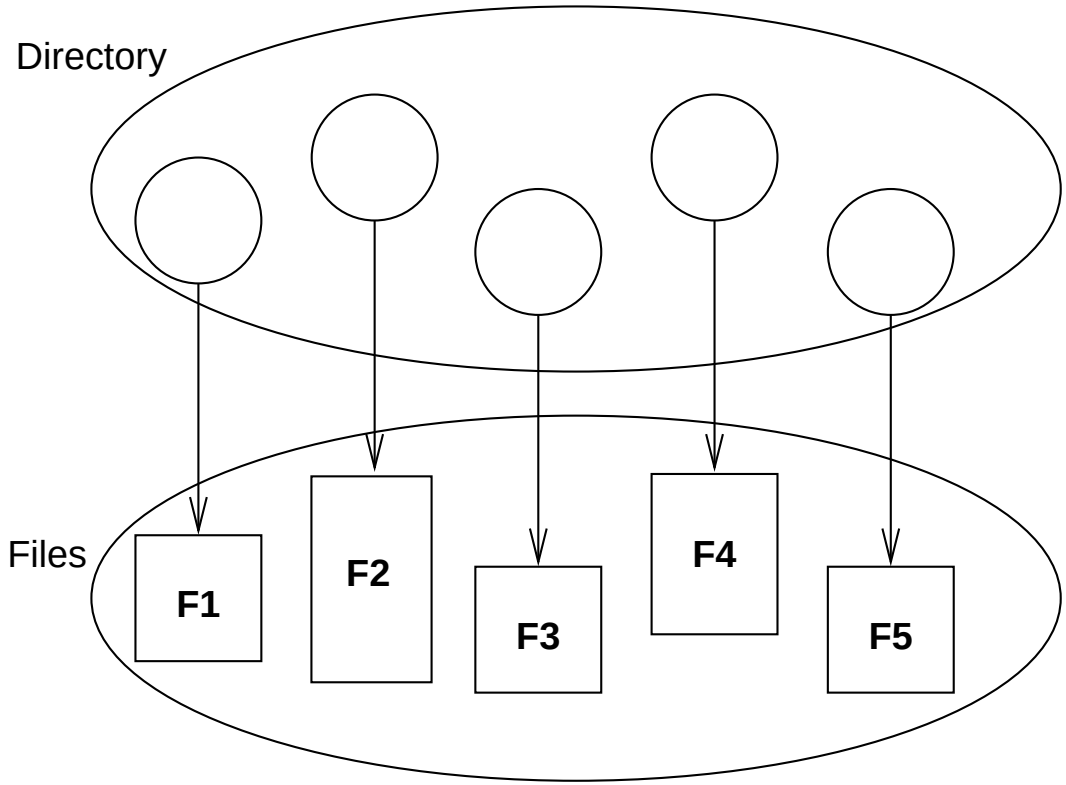
```
/* File display program. Minimal error checking */
#define BUF_SIZE 256

int main(int argc, char *argv[]) {
   int in_fd, rd_cnt, wr_cnt;
   char buf[BUF_SIZE];
   if (argc != 2)                                  /* incorrect number of args */
      exit(1);
   in_fd = open(argv[1], O_RDONLY);                /* open source file */
   if (in_fd < 0 )
      exit(2);                                     /* error in open */
   while (TRUE) {                                  /* loop until done, or an error */
      rd_cnt = read(in_fd, buf, BUF_SIZE);         /* read from source */
      if (rd_cnt <= 0)
         break;                                    /* end of file, or error */
      wr_cnt = write(STDOUT_FILENO, buf, rd_cnt); /* write block read */
      if (wr_cnt < 0)
         exit(4);                                  /* error writing */
   }
   close( in_fd);                                  /* close files */
   if( rd_cnt == 0 )                               /* no error on last read */
      exit(0);
   else                                            /* error on last read */
      exit(5);
}
```

- `write()` does not ensure that OS writes all bytes in its second argument
  - Should use a cycle

Sumário

Directories

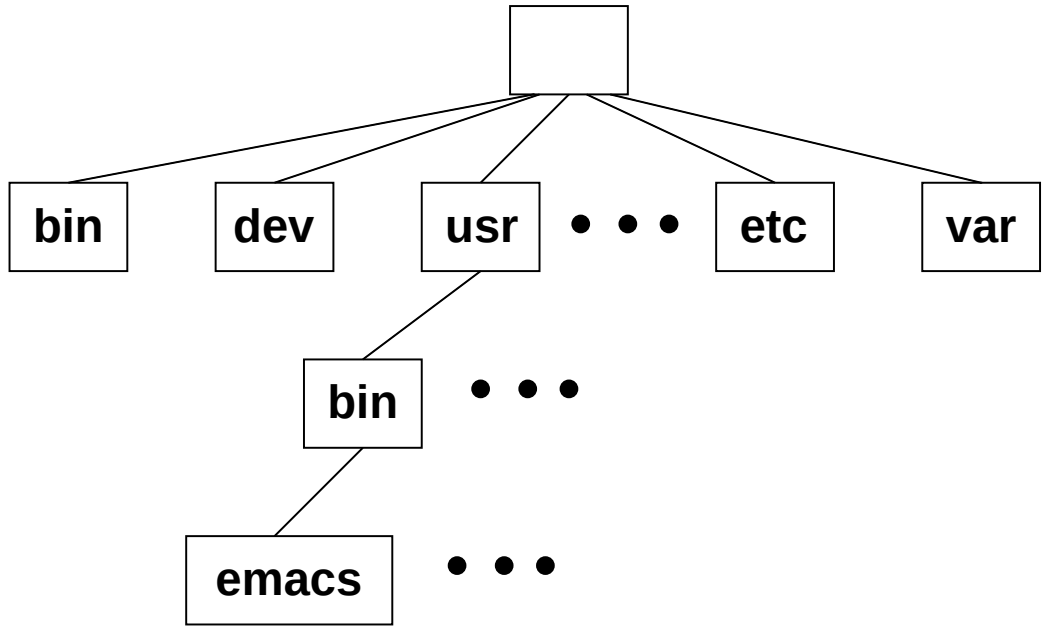
- Mapping of file names to their location on disk is done with the help of *diretories*:



- Both files and directories must be stored on disk
  - In Unix, directories are a special type of file

Diretórios Estruturados em Árvore

- Quase todos os sistemas de ficheiros suportam uma hierarquia de diretórios:



- O uso duma àrvore permite:
  - evitar conflitos de nomes;
  - agrupar ficheiros de alguma forma relacionados;
  - *resolver nomes* duma forma eficiente.

Nome (absoluto) inclui todos os *componentes* desde a raiz ( ) até ao ficheiro/diretório, separados por `/`. P.ex.:

`/usr/bin/emacs`

Diretório Corrente (.) e Nomes de Ficheiros

Problema Os nomes (pathnames) *absolutos* podem ser demasiado compridos e pouco convenientes

Solução Cada processo está associado a um diretório corrente (current/working diretory), que pode variar ao longo da sua vida (`chdir()`)

Nome relativo inclui todos os componentes desde o diretório corrente até ao ficheiro/diretório. P.ex.: `bin/emacs`.

- Nomes começando com um separador são absolutos
  - Em Unix, `chroot()` permite especificar o diretório raiz dum processo – usado para controlo de acesso.
- Componentes com significado especial:

  `.` representa o diretório corrente;
  `..` representa o diretório pai.

Operações sobre Diretórios

Operação	**Chamada ao Sistema em Unix**
Criar um diretório	`mkdir()`
Remover um diretório	`rmdir()`
Mudar o diretório corrente	`chdir()`
Mudar o diretório raiz	`chroot()`
Ler elementos dum diretório	`getdents()`
Criar um ficheiro	`open()`
Criar um nome novo	`link()`
Criar um *link* simbólico	`symlink()`
Remover um ficheiro	`unlink()`
Alterar o nome dum ficheiro	`rename()`

- A chamada ao sistema `getdents()` não é fácil de usar: é preferível usar funções da `libc` (`man readdir`).

Sumário

Organização dum SO

Núcleo/*kernel* A parte do sistema operativo que executa em modo priveligiado.

- Maior parte do código é *reactivo*, i.e. executado em reposta a:

  Chamadas ao sistema realizadas por processos de aplicação.

  Interrupções geradas por dispositivos de E/S.
- Outra parte, inclui processos/*threads* que executam em modo priveligiado

Biblioteca de chamadas ao sistema Que oferece a interface programática para aceder aos serviços do SO.

Processos de sistema Processos que não executam em modo priveligiado, mas que oferecem alguns dos serviços do SO.

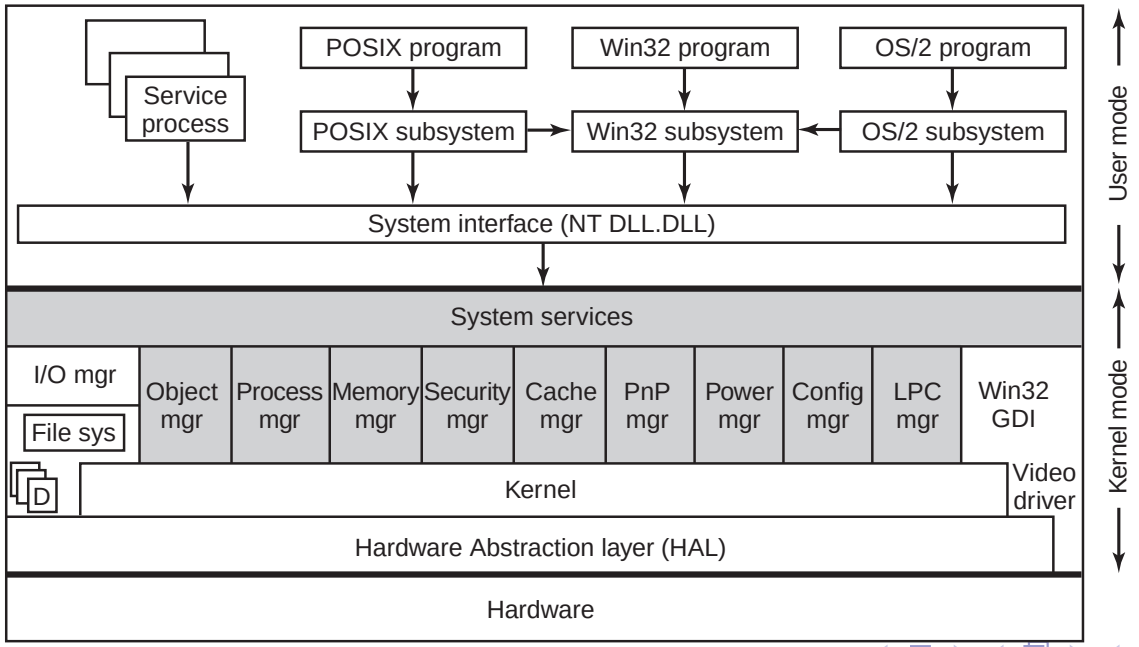
Implementação do *Kernel* dum SO

- O *kernel* dum SO consiste em código escrito para a ISA do processador:
  - Idealmente, o *kernel* deveria ser independente do processador, ou melhor da ISA;
  - Na prática, mais de 90% do *kernel* é escrito numa linguagem de *alto nível*, p.ex. C, e posteriormente compilado para a ISA do processador. O restante depende do ISA do processador, pelo que tem que ser reescrito para cada ISA suportada.
- Em termos de organização, há 2 particularmente comuns:

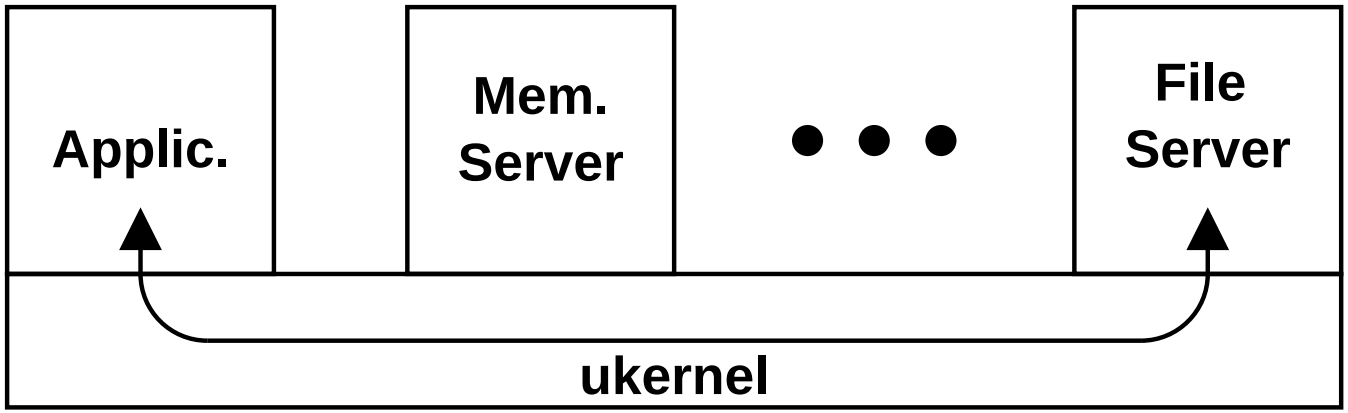
  Monolítica , p.ex. Linux e
  Micro-Kernel , p.ex., QNX

Implementação Monolítica

- Não faz uso de processos de sistema.
- Todos os serviços do SO são fornecidos pelo *kernel*
  - Implementados como um conjunto de funções
  - As estruturas de dados são implementadas em *kernel space*, sendo por isso partilhadas.
    - Há risco de *bugs* numa função do *kernel* causarem problemas noutras funções do *kernel*
- Em qualquer caso, o código está tipicamente organizado em módulos, como acontece p.ex. com o Windows 2000:



Implementação baseada em *Micro-Kernel*



- O *kernel* do SO oferece apenas funções essenciais para:
  - gestão de processos/*threads*
  - comunicação entre processos
  - gestão de memória (dependendo do *hardware*)
- A restante funcionalidade é implementada por processos de sistema.
  - Maior fiabilidade
    - Quanto menos código, menos *bugs*
    - É mais difícil que *bugs* num processo sejam propagados a outros processos.
  - Menor eficiência
    - A generalidade dos serviços requer a intervenção de pelo menos um processo adicional.

Sumário

Arranque dum Computador/*Boot(strapp)ing*

- Os pormenores desta fase são tipicamente dependentes do processador e do computador.
- Em qualquer caso pode-se identificar 3 fases/tarefas genérica s:

  Teste do *hardware* assim que o sistema é ligado
  - O código necessário reside em memória principal não volátil, fazendo parte do que se designa por **firmware**
  - O início deste programa encontra-se tipicamente no endereço da posição de memória com que o *program counter* é inicializado após *reset*/arranque do processador.

  Boot(strapp)loading durante a qual o *kernel* do SO é carregado na memória principal
  - Esta fase poderá não ocorrer, se o *kernel* estiver guardado em memória principal não volátil, como acontece tipicamente em sistemas embebidos mais simples

  Inicialização do SO durante a qual o SO detecta o *hardware* e é inicializado.

Boot(strapp)loading

- Este processo é tipicamente realizado por um programa, o *bootloader*, que está também guardado em memória principal não volátil, i.e. faz parte do *firmware*.
- Frequentemente, o espaço disponível em ROM é reduzido, e.g. nos PC, pelo que a carga de *kernel* exige pelo menos *2 bootloaders*:
  - Tipicamente, cada *bootloader* é usado apenas para carregar o *bootloader* seguinte.
- Em particular, o *bootloader* em *firmware* apenas carrega o *bootloader* seguinte.

Questão Onde se encontra esse *booloader*?

Resposta Normalmente nos primeiros “sectores” do dispositivo com o SO, seja ele um disco duro, um CD ou uma *USB-pen*

- De novo, o espaço disponível é frequentemente insuficiente, daí necessidade dum *bootloader* adicional

Inicialização do SO

- A última acção do último *bootloader* é passar o controlo para o *kernel* que acabou de carregar em memória RAM
- O *kernel* procede então à inicialização do SO, a qual tipicamente inclui os seguintes passos:
  - Detectar e inicializar o *hardware*, p.ex. controladores existentes
  - Inicializar as estruturas de dados do *kernel*
  - Iniciar os processos/*threads* ao nível do *kernel*
  - Inicia a execução de um ou mais processos de sistema (i.e. ao nível do utilizador)

Leitura Adicional

- Secções 2.1, 2.2, 2.3, 2.4 e 2.5 de
  José Alves Marques e outros, *Sistemas Operativos*,
  FCA - Editora Informática, 2009
- Secções 1.5 e 1.6 de
  Modern Operating Systems, 2nd Ed.
- Secções 2.3, 2.4, 2.5, 2.7 e 2.10 de
  Silberschatz e outros, *Operating System Concepts*,
  7th Ed.
- Outra documentação (transparências e enunciados dos TPs) na página da disciplina