Sistemas Operativos: Introdução

February 23, 2012

Sumário

Chamadas ao Sistema

Lab 2

Organização dum SO

Arranque dum Sistema Operativo

Sumário

Chamadas ao Sistema

Lab 2

Organização dum SO

Arranque dum Sistema Operativo

Chamadas ao Sistema (*System Calls*)

Problema: Como se acede aos serviços do sistema operativo?

Solução: Através de chamadas ao sistema (ou através de programas utilitários):

- Chamadas ao sistema são a interface programática (API) dum SO.
- Para cada tipo de serviço, o SO oferece um conjunto de chamadas ao sistema.
- Do ponto de vista do programador, fazer uma chamada ao sistema consiste em invocar uma função (`/usr/share/man/man2`).

Chamadas ao Sistema: Gestão de Ficheiros

Operação	Chamada ao Sistema em Unix
Criar	`open()` (`creat()` – obsoleta)
Lêr	`read()`
Escrever	`write()`
Reposicionar cabeça	`lseek()`
Lêr atributos	`fstat()`, `lstat()`, `stat()`
Alterar atributos	`chmod()`, `chown()` ...
Mapear na Memória	`mmap()`, `munmap`

- Em Unix:
  - **tem-se** que invocar `open()` antes de aceder a um ficheiro;
  - **deve-se** invocar `close()` quando não se pretende aceder mais ao ficheiro.

Chamadas ao Sistema vs. Funções

Pergunta: Por que razão uma chamada ao sistema não é uma simples função?

Resposta: Porque o SO reside numa área de memória (*kernel space*) inacessível à generalidade das aplicações (que residem em *user space*):

- evita-se que uma aplicação aceda duma forma arbitrária ao SO (e aos recursos que ele gere):

  só o pode fazer através das chamadas ao sistema
- protege-se o SO das aplicações e as aplicações umas das outras

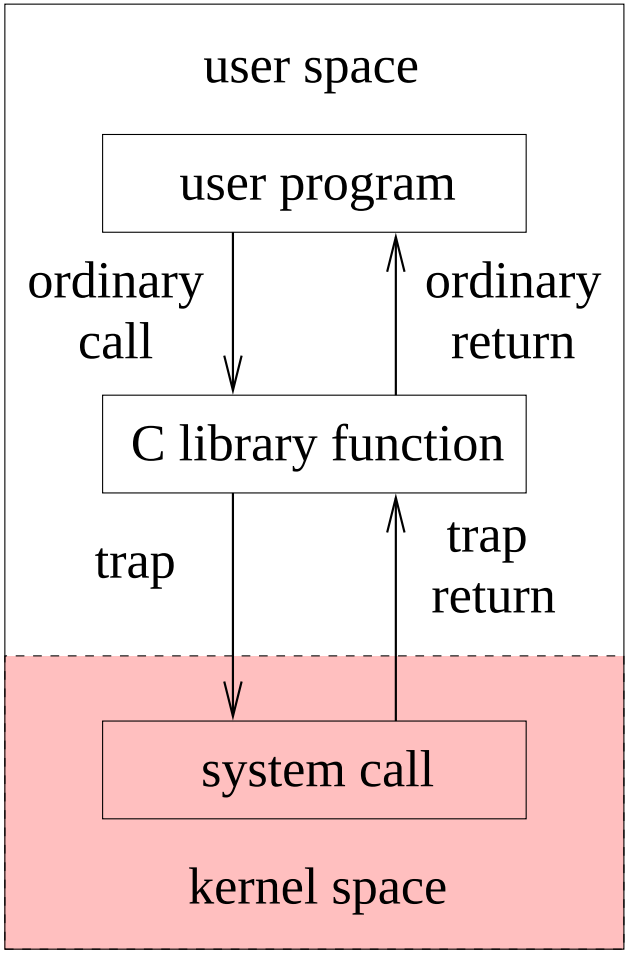
Níveis de Privilégio do CPU

- CPUs de concepção moderna suportam pelo menos 2 níveis de privilégio (a arquitectura IA32 suporta 4, mas Linux usa apenas 2):

  user level nível de execução da generalidade das aplicações:
  - não permite aceder ao *kernel space*;
  - não permite executar certas instruções do CPU.

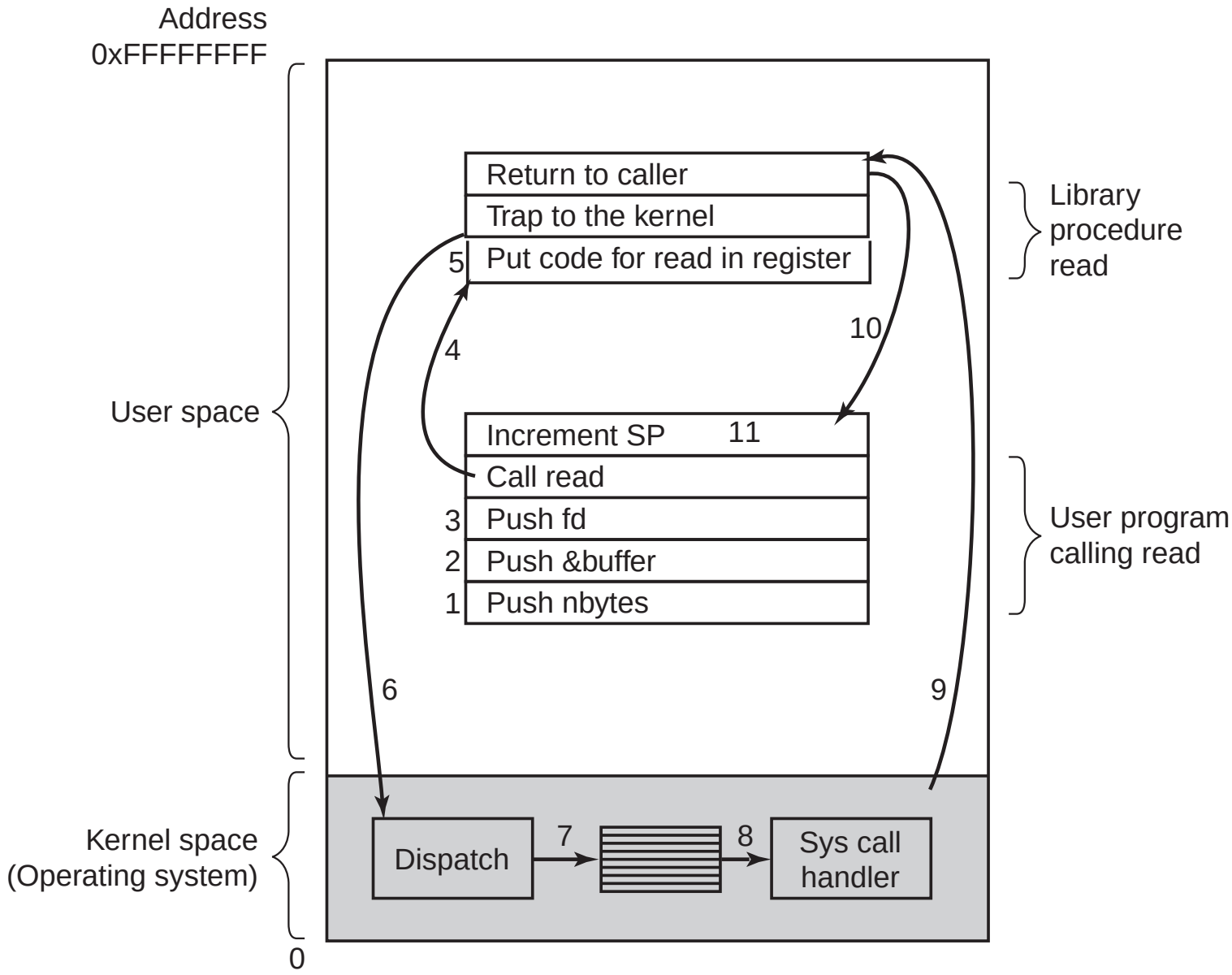
  kernel level nível de execução do SO - sem restrições.
- A comutação entre níveis de privilégio é feita de forma automática:
  - quando se executa determinadas instruções;
  - quando o CPU responde a uma interrupção.
- Toda a segurança dum SO é construída sobre este mecanismo.

Implementação das Chamadas ao Sistema



- Usa instruções especiais oferecidas pelo HW (*call gates* ou *sw interrupts*, no caso da arquitectura IA32), que comutam automáticamente de nível de privilégio.
- Para o programador, é como se invocasse uma função da biblioteca de C.

ssize_t read(int fd, void *buf, size_t count)



Passos na Execução de `read()`

1, 2, 3 `push` dos argumentos para a *stack*;

4 chamada da função `read` da biblioteca C;

5 inicialização do registo com o # da chamada ao sistema;

6 mudança de modo de execução do CPU;

7 despacho para o *handler* apropriado;

8 execução do *handler*;

9 **possível** retorno para a função da biblioteca C;

10 retorno da função `read` da biblioteca C;

11 ajuste da *stack*.

Sumário

Chamadas ao Sistema

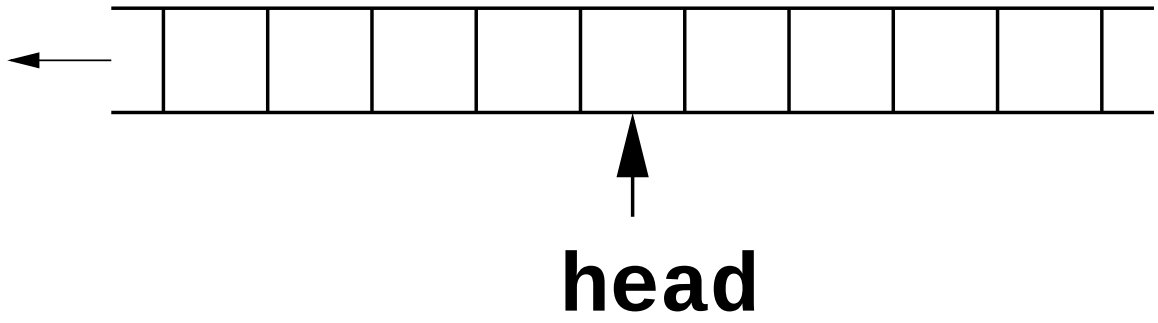
Lab 2

Organização dum SO

Arranque dum Sistema Operativo

Modelo Acesso a Ficheiros

- *Sequencial* (fita magnética)



  - para aceder a um *byte*/registo tem-se que aceder a todos os que o precedem;
  - não é possível saltar para a frente ou para trás.
- *Directo (random)*
  - permite posicionar a cabeça de leitura/escrita em qualquer *byte*/registo do ficheiro;
  - nem todos os dispositivos de E/S suportam este tipo de acesso:
    - por exemplo, porta série.

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()` não garante que o SO escreve todos os *bytes*.
  - Deveria usar um ciclo

Sumário

Chamadas ao Sistema

Lab 2

Organização dum SO

Arranque dum Sistema Operativo

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.

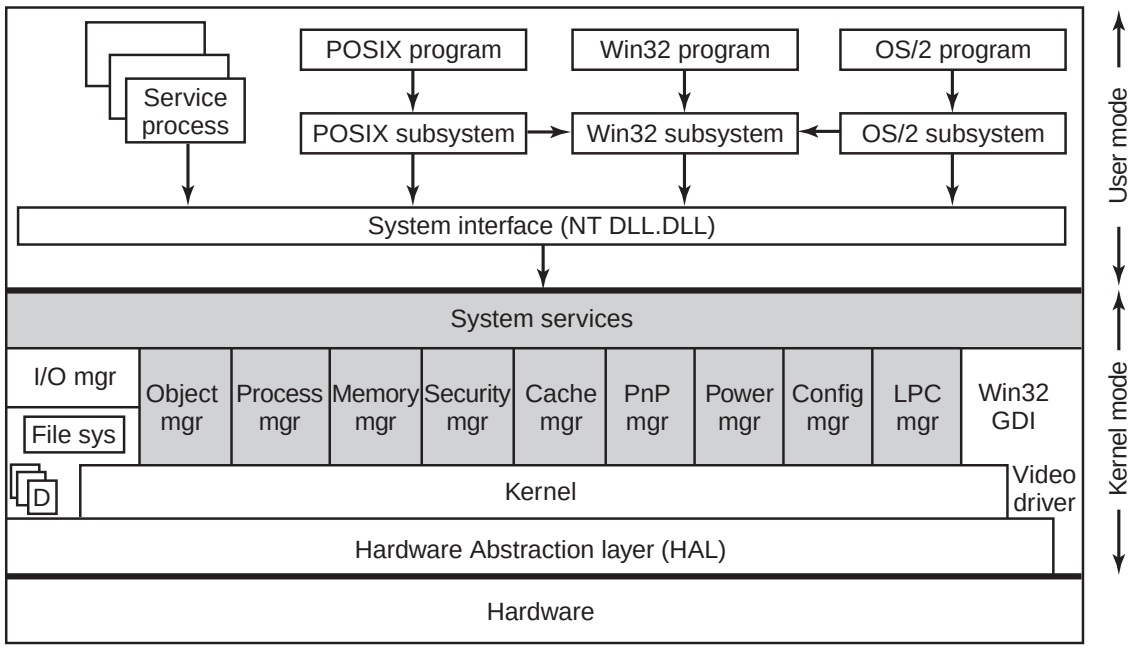
Implementação do *Kernel* dum SO

- 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:

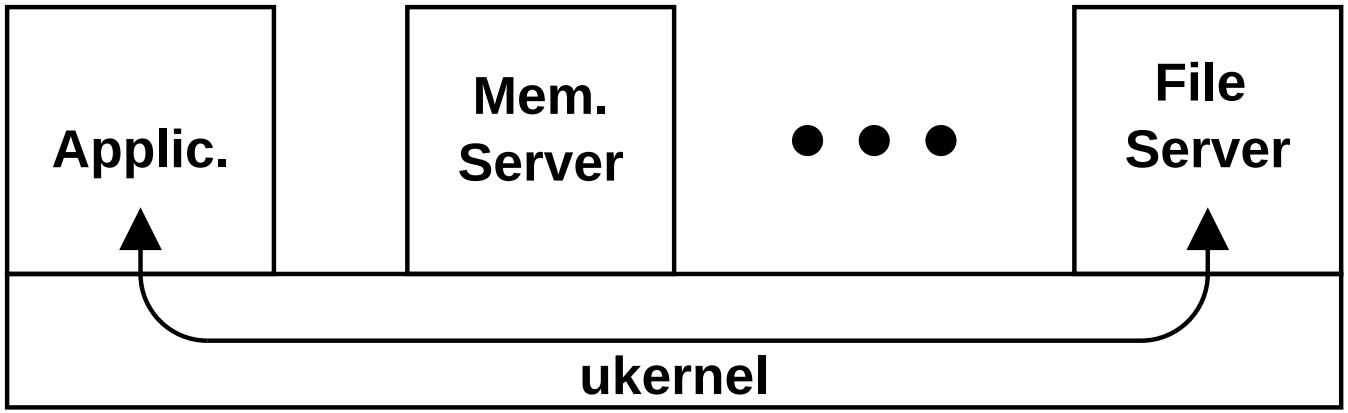
  Monolítica , p.ex. Linux e
  Micro-Kernel , p.ex., QNX

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

Chamadas ao Sistema

Lab 2

Organização dum SO

Arranque dum Sistema Operativo

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/processos genéricos:

  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 vários *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*

- Obviamente, cada um dos outros *bootloader* tem que conhecer a localização do *bootloader* seguinte na cadeia.

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