1- Introdução

- O que é um SO?
- Questão: Virtualizar recursos
  - Virtualizar o CPU
  - Virtualizar a memória
  - Concorrência
  - Persistência
- Objetivos na criação de um SO
- A importância do UNIX e o surgimento do LINUX

O que é um SO?

- Um programa executa instruções
- Milhões de vezes por segundo (1 milissegundo = 1/1000 seg., 1 micro-segundo = 1/1000 000) um processador:
  - Vais buscar instrução à memória;
    - Descodifica-a;
    - Executa-a;
  - Vai buscar a próxima instrução;
- Mas…
  - Enquanto um programa é executado, acontecem muitas outras coisas.
  - O software responsável por permitir a execução de determinado programa é o **Sistema Operativo**.

Virtualizar Recursos

THE CRUX OF THE PROBLEM:
HOW TO VIRTUALIZE RESOURCES

One central question we will answer in this book is quite simple: how does the operating system virtualize resources? This is the crux of our problem. *Why* the OS does this is not the main question, as the answer should be obvious: it makes the system easier to use. Thus, we focus on the *how*: what mechanisms and policies are implemented by the OS to attain virtualization? How does the OS do so efficiently? What hardware support is needed?

We will use the "crux of the problem", in shaded boxes such as this one, as a way to call out specific problems we are trying to solve in building an operating system. Thus, within a note on a particular topic, you may find one or more *cruces* (yes, this is the proper plural) which highlight the problem. The details within the chapter, of course, present the solution, or at least the basic parameters of a solution.

A QUESTÃO: COMO VIRTUALIZAR RECURSOS?

A questão central que se procura responder neste curso é simples: Como é que um sistema operativo (SO) virtualiza recursos? É essa a raiz do problema. Por que é que o SO o faz não é a principal questão. Mas a resposta é óbvia: torna a utilização do sistema mais simples. Focamo-nos no como: Que mecanismos e políticas são implementadas no SO para obter a virtualização? Como é que um SO o faz de forma tão eficiente? Que suporte do *hardware* é necessário?

Virtualizar o CPU

- O SO, com alguma ajuda do *hardware*, é responsável por criar a ilusão de que o sistema tem um grande número de CPUs que processam e executam os diferentes programas em simultâneo – em Linux, **processos**.
  - Qual programa correr? Quando? Em que circunstâncias executar um programa e não outro?

    [Qual processo correr? Quando? Em que circunstâncias executar um processo e não outro?]
- **CPU = Gestor de recursos**

Virtualizar o CPU

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <sys/time.h>
4   #include <assert.h>
5   #include "common.h"
6
7   int
8   main(int argc, char *argv[])
9   {
10      if (argc != 2) {
11          fprintf(stderr, "usage: cpu <string>\n");
12          exit(1);
13      }
14      char *str = argv[1];
15      while (1) {
16          Spin(1);
17          printf("%s\n", str);
18      }
19      return 0;
20  }
```

```
void Spin(int howlong) {
   double t = GetTime();
   while ((GetTime() - t) < (double)howlong)
                ; // do nothing in loop
}
```

Exige um argumento na entrada: O nome do programa e uma string que vai ser impressa no ciclo.

Repetidamente verifica o tempo e retorna assim que tiver passado 1 segundo...

```
prompt> gcc -o cpu cpu.c -Wall
prompt> ./cpu "A"
A
A
A
A
^C
prompt>
```

```
prompt> ./cpu A & ; ./cpu B & ; ./cpu C & ; ./cpu D &
[1] 7353
[2] 7354
[3] 7355
[4] 7356
A
B
D
C
A
B
D
C
A
C
B
D
...
```

Virtualizar memória

- O modelo físico da memória é simples: Vetor de *bytes*.
  - Para ler a memória, é necessário especificar o endereço (**address**).
    - Tem-se o acesso aos dados guardados nesse endereço
  - Para escrever na memória, é necessário especificar os dados e o endereço onde se pretende escrever os dados.
- A memória é acedida ao longo da execução do programa.
  - Programa guarda as estruturas de dados na memória e acede a esses dados ao longo das diferentes instruções.
  - Cada instrução é lida da memória.

- **Cada processo acede ao seu espaço privado de memória – address space.**

Virtualizar memória

```
1   #include <unistd.h>
2   #include <stdio.h>
3   #include <stdlib.h>
4   #include "common.h"
5
6   int
7   main(int argc, char *argv[])
8   {
9       int *p = malloc(sizeof(int));                    // a1
10      assert(p != NULL);
11      printf("(%d) address pointed to by p: %p\n",
12              getpid(), p);                            // a2
13      *p = 0;                                          // a3
14      while (1) {
15          Spin(1);
16          *p = *p + 1;
17          printf("(%d) p: %d\n", getpid(), *p);       // a4
18      }
19      return 0;
20  }
```

pid = process id

```
prompt> ./mem
(2134) address pointed to by p: 0x200000
(2134) p: 1
(2134) p: 2
(2134) p: 3
(2134) p: 4
(2134) p: 5
^C
```

```
prompt> ./mem &; ./mem &
[1] 24113
[2] 24114
(24113) address pointed to by p: 0x200000
(24114) address pointed to by p: 0x200000
(24113) p: 1
(24114) p: 1
(24114) p: 2
(24113) p: 2
(24113) p: 3
(24114) p: 3
(24113) p: 4
(24114) p: 4
...
```

Concorrência

- SO tem de resolver as questões que surgem do facto de executar muitos programas em simultâneo.
- Há que saber utilizar a concorrência na programação:
  - Modelo de tarefas (*threads*):
    - Uma tarefa é um trabalho executado pelo SO que partilha um mesmo espaço de memória.
    - Podem existir muitas *threads* associadas a um processo.
- **É preciso garantir que o acesso a recursos partilhados é controlado.**

Concorrência

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include "common.h"
4
5   volatile int counter = 0;
6   int loops;
7
8   void *worker(void *arg) {
9       int i;
10      for (i = 0; i < loops; i++) {
11          counter++;
12      }
13      return NULL;
14  }
15
16  int
17  main(int argc, char *argv[])
18  {
19      if (argc != 2) {
20          fprintf(stderr, "usage: threads <value>\n");
21          exit(1);
22      }
23      loops = atoi(argv[1]);
24      pthread_t p1, p2;
25      printf("Initial value : %d\n", counter);
26
27      Pthread_create(&p1, NULL, worker, NULL);
28      Pthread_create(&p2, NULL, worker, NULL);
29      Pthread_join(p1, NULL);
30      Pthread_join(p2, NULL);
31      printf("Final value   : %d\n", counter);
32      return 0;
33  }
```

O contador é inicializado a zero.

Este comando corresponde a execução de 3 instruções...

O “loops” recebe o valor introduzido pelo utilizador quando inicia o programa!

Concorrência

```
prompt> gcc -o thread thread.c -Wall -pthread
prompt> ./thread 1000
Initial value : 0
Final value   : 2000


prompt> ./thread 100000
Initial value : 0
Final value   : 143012    // huh??
prompt> ./thread 100000
Initial value : 0
Final value   : 137298    // what the??
```

Persistência

- O software de um SO que gere um disco é designado por sistema de ficheiros.
  - É responsável por guardar ficheiros que o utilizador cria de forma eficiente e segura.
- O SO não cria um disco virtual privado para cada aplicação.
  - Assume que os utilizadores pretendem partilhar a informação, ou seja, os ficheiros.
    - Exemplo:
      - Escrevemos programa C, utilizando o gedit main.c
      - Compilamos o programa utilizando gcc -o main main.c
      - Executamos o programa compilado ./main

      [O ficheiro main.c é utilizado para criar o main que depois é utilizado na execução.]

Persistência

```
1   #include <stdio.h>
2   #include <unistd.h>
3   #include <assert.h>
4   #include <fcntl.h>
5   #include <sys/types.h>
6
7   int
8   main(int argc, char *argv[])
9   {
10      int fd = open("/tmp/file", O_WRONLY | O_CREAT | O_TRUNC, S_IRWXU);
11      assert(fd > -1);
12      int rc = write(fd, "hello world\n", 13);
13      assert(rc == 13);
14      close(fd);
15      return 0;
16  }
```

Persistência

- Para realizar esta tarefa, são executadas três chamadas ao sistema operativo:
  - *open()*
    - Abre e cria o ficheiro;
  - *write()*
    - Escreve dados no ficheiro;
  - *close()*
    - Fecha o ficheiro indicando que o programa não vai escrever mais dados.
- Estas **chamadas sistema** são tratadas pelo SO designada por sistema de ficheiros.
- Escrever ou não escrever as atualizações de um ficheiro no disco é também um tema de discussão.

Objetivos de um SO

- O trabalho executado por um SO:
  - Usar recursos, como a CPU, memória ou disco e virtualiza-as.
  - Executar tarefas de forma concorrente.
  - Guardar informação de forma persistente.
- Alguns dos objetivos:
  - **Desempenho (*performance*)**
    - Um bom desempenho significa que a virtualização, a concorrência e persistência funcionam.
    - Mas há que reduzir a sobrecarga que esse desempenho provoca:
      - Tempo e memória;
  - **Proteção (*protection*)**
    - Entre aplicações e entre o SO e aplicações
      - Isolamento de processos (***isolation***) uns dos outros e garantir a proteção do SO.

Objetivos de um SO

- **Confiança (reliability)**
  - Não devem ter falhas
  - Se um SO falha, todas as aplicações que correm um sistema também falham!
- **Eficiência energética**
- **Segurança**
- **Mobilidade**

UNIX

Inserir no glossário em **UNIX**

ASIDE: **THE IMPORTANCE OF UNIX**

It is difficult to overstate the importance of UNIX in the history of operating systems. Influenced by earlier systems (in particular, the famous **Multics** system from MIT), UNIX brought together many great ideas and made a system that was both simple and powerful.

Underlying the original "Bell Labs" UNIX was the unifying principle of building small powerful programs that could be connected together to form larger workflows. The **shell**, where you type commands, provided primitives such as **pipes** to enable such meta-level programming, and thus it became easy to string together programs to accomplish a bigger task. For example, to find lines of a text file that have the word "foo" in them, and then to count how many such lines exist, you would type: `grep foo file.txt|wc -l`, thus using the `grep` and `wc` (word count) programs to achieve your task.

The UNIX environment was friendly for programmers and developers alike, also providing a compiler for the new **C programming language**. Making it easy for programmers to write their own programs, as well as share them, made UNIX enormously popular. And it probably helped a lot that the authors gave out copies for free to anyone who asked, an early form of **open-source software**.

Also of critical importance was the accessibility and readability of the code. Having a beautiful, small kernel written in C invited others to play with the kernel, adding new and cool features. For example, an enterprising group at Berkeley, led by **Bill Joy**, made a wonderful distribution (the **Berkeley Systems Distribution**, or **BSD**) which had some advanced virtual memory, file system, and networking subsystems. Joy later co-founded **Sun Microsystems**.

Unfortunately, the spread of UNIX was slowed a bit as companies tried to assert ownership and profit from it, an unfortunate (but common) result of lawyers getting involved. Many companies had their own variants: **SunOS** from Sun Microsystems, **AIX** from IBM, **HPUX** (a.k.a. "H-Pucks") from HP, and **IRIX** from SGI. The legal wrangling among AT&T/Bell Labs and these other players cast a dark cloud over UNIX, and many wondered if it would survive, especially as Windows was introduced and took over much of the PC market...

LINUX

Inserir no glossário em **LINUX**

ASIDE: **AND THEN CAME LINUX**

Fortunately for UNIX, a young Finnish hacker named **Linus Torvalds** decided to write his own version of UNIX which borrowed heavily on the principles and ideas behind the original system, but not from the code base, thus avoiding issues of legality. He enlisted help from many others around the world, and soon **Linux** was born (as well as the modern open-source software movement).

As the internet era came into place, most companies (such as Google, Amazon, Facebook, and others) chose to run Linux, as it was free and could be readily modified to suit their needs; indeed, it is hard to imagine the success of these new companies had such a system not existed. As smart phones became a dominant user-facing platform, Linux found a stronghold there too (via Android), for many of the same reasons. And Steve Jobs took his UNIX-based **NeXTStep** operating environment with him to Apple, thus making UNIX popular on desktops (though many users of Apple technology are probably not even aware of this fact). And thus UNIX lives on, more important today than ever before. The computing gods, if you believe in them, should be thanked for this wonderful outcome.

Questões

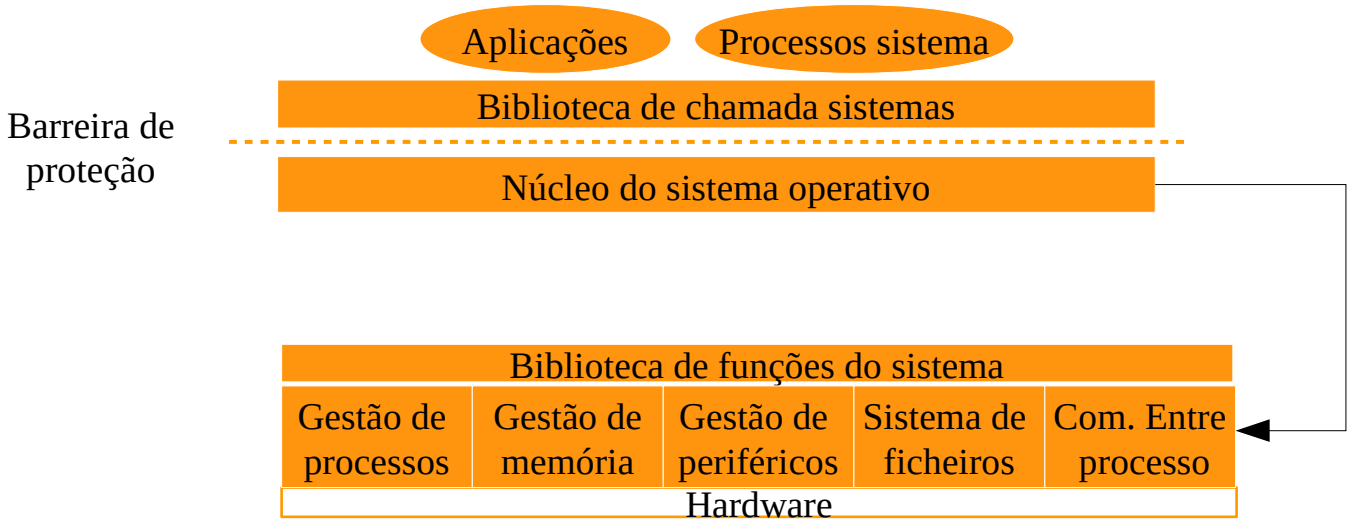
- Explique o que é a virtualização do cpu e a virtualização da memória?
- Será que um SO poderia funcionar sem o conceito da virtualização do cpu? O que é que acontecia?
- O que é que significa o conceito de persistência?
- Se o SO operativo não fosse responsável pela persistência o que poderia acontecer?
- Existe forma de executar mais que um programa partilhando recursos (por exemplo variáveis)? Qual é o nome associado a esse tipo de processo?
- O que pode acontecer se não tomarmos cuidado na gestão dos recursos quando existe mais do que um programa a executar no mesmo espaço de endereços (i.e no mesmo espaço de memória)?
- Quais são os objetivos de um sistema operativo?

2-Organização de um SO

- Composição de um SO
  - Núcleo
    - Confinamento do Núcleo
    - Modo núcleo vs. Modo utilizador
    - Exceções e Interrupções
  - Chamadas sistema
  - Processos sistema
- Arquitetura do SO
  - Windows
  - Android

Composição de um SO

- Um sistema operativo é composto por três entidades:
  - O Núcleo
  - As bibliotecas de chamada sistema (system calls)
  - Os processos sistema



Núcleo

- Implementa os mecanismos de base do sistema operativo
  - Gestão de processos
  - Gestão de memória
  - Gestão dos ficheiros
  - Gestão de E/S
  - Comunicação entre processos
  - Outras:
    - Interface gráfica
    - Gestão dos utilizadores
    - Gestão de energia

Confinamento do Núcleo

- A segurança do SO advém da existência de um isolamento.
  - Entre processos dos utilizadores.
  - Entre processos dos utilizadores e o núcleo do SO.
- O isolamento é garantido pela gestão da memória.
  - Só permite que determinadas posições de memória sejam acessíveis por um utilizador.
- Modo utilizador versus modo de núcleo.
  - A gestão da memória é realizada pelo núcleo.
  - Certas instruções necessitam de aceder a espaços de memória controlados pelo núcleo (por exemplo, acesso a uma tabela que localiza ficheiro no disco)
  - Existe dos modos de execução do processador controlados pelo ***hardware***:
    - *Modo núcleo*
    - *Modo utilizador*

Modo núcleo vs. Modo utilizador

- O modo utilizador restringe:
  - O acesso a determinadas posições de memória.
  - A certas instruções que podem ultrapassar o isolamento:
    - Interação direta com os periféricos.
- O modo núcleo não tem restrições.
  - É no modo núcleo que o SO se executa.

Interrupções ou exceções

- A passagem do modo utilizador para modo núcleo faz-se através de interrupções ou exceções.
  - Interrupções são exceções assíncronas – provocadas por evento externo ao processo.
  - As exceções propriamente ditas são síncronas – provocadas por evento associado ao próprio processo.
- Uma interrupção ou exceção provoca:
  - Uma mudança do modo de utilizador para modo núcleo do SO
  - A execução de uma rotina de tratamento da interrupção ou exceção, previamente definida para cada um dos tipo de interrupção ou execção.
  - O estado do processo em execução é salvaguardado antes de se executar a rotina de tratamento.
  - O retorno ao modo utilizador é automaticamente efetuado pela instrução de retorno da interrupção ou exceção (RTI)

Chamadas sistema

- As chamadas sistema são implementadas por uma função que invoca a exceção (**trap**) que:
  - Transfere o controlo para o núcleo.
  - Coloca o processador no modo de execução núcleo.
  - No núcleo é feita a escolha do código apropriado que implementa chamada sistema em causa.
  - No final da chamada, a instrução de retorno de interrupção, devolve o controlo para a função de biblioteca que, por sua vez, retorna para o código do utilizador.
- Vantagens das chamadas sistema:
  - O facto de o código das aplicações não ter acesso às estruturas de dados mantidas pelo núcleo.
  - O facto de o código das aplicações não ter acesso a instruções que permitiriam contornar os mecanismos de proteção que garantem o funcionamento seguro do sistema.

Processos sistema

- Os processos sistema executam funções que podem ser delegadas em processos autónomos de forma a reduzir a complexidade do núcleo e aumentar a sua robustez.
- Os processos executam-se em modo utilizador mas…
  - Pertencem ao sistema operativo pelo que têm privilégios de administrador (***superuser***).
  - Para invocar funções do sistema têm de o fazer através de chamadas sistema (menos eficiente do que se a execução do processo fosse diretamente do núcleo).

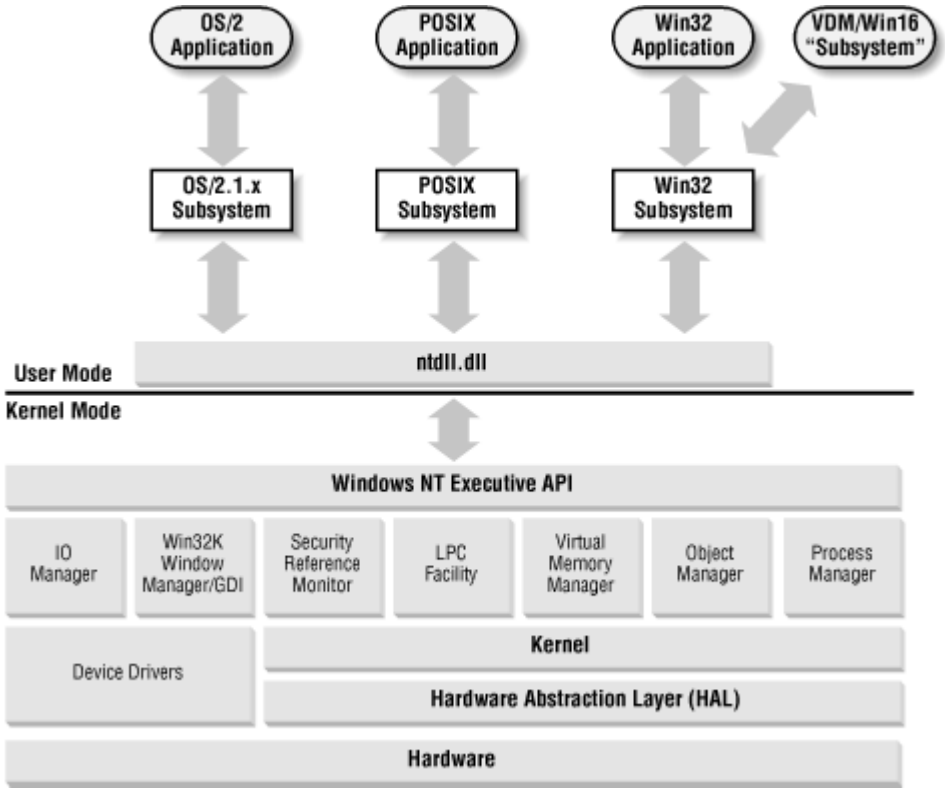
Arquitetura do núcleo

- Aspetos de grande importância no desenho de um SO:
  - Sistemas monolíticos:
    - Um programa com estruturas de dados globais e, eventualmente organizados por módulos.
    - A introdução de novos periféricos é problemática pois tem de se adicionar ao núcleo novos gestores de periféricos (**device drivers**).
    - Introduz um problema de robustez e segurança, pois um erro no gestor de periférico pode afetar todo o sistema.
  - Organização do núcleo em camadas:
    - Generaliza o conceito de proteção entre modo núcleo e modo utilizador.
      - Divide o núcleo em camadas que implementam diferentes funcionalidades.
      - A camada externa invoca funções das camadas internas através de mecanismos semelhantes às das chamada sistema.

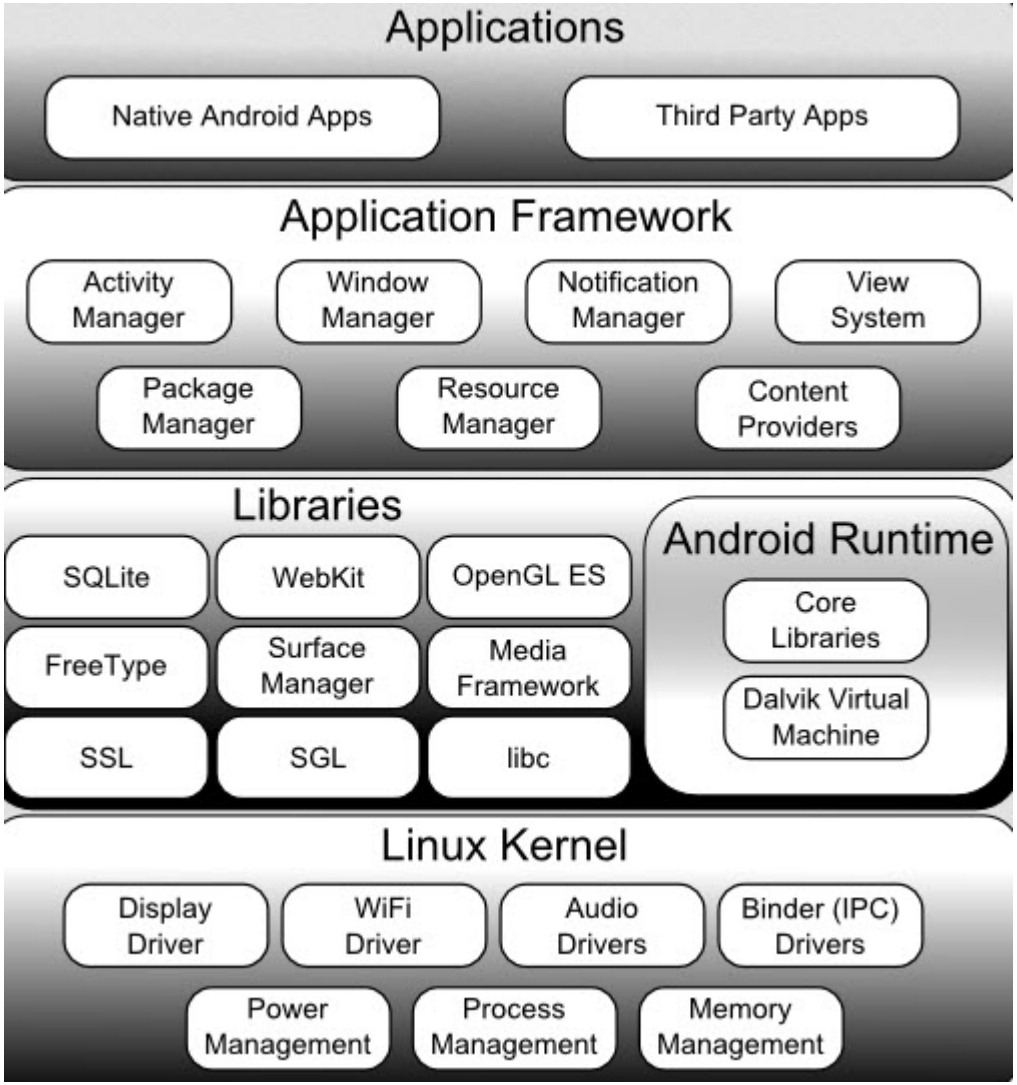
Arquitetura do núcleo

- Organização do núcleo em camadas
  - Garante mais isolamento mas há uma perda de desempenho devido à sobrecarga de processamento introduzida por estas camadas.
- A maior parte dos SO mativeram uma estrutura monolítica dado que o impacto no desempenho foi considerado demasiado penalizante.
- Sistemas micronúcleos
  - Oferecem serviços básicos de gestão de processos, memória e comunicação entre processos.
  - A restante funcionalidade é deslocada para um conjunto de processos sistema executados fora do núcleo (processos servidores).
  - Aumenta flexibilidade do sistema, e a organização é mais segura e robusta.
  - O mecanismo de intercomunicação entre os componentes do sistema operativos e entre estes e o núcleo é muito mais lento quando comparado com o sistema monolítico.

Arquitetura do Windows



Arquitetura do Android



Questões

- O que são chamadas sistema?
- O que é o confinamento do núcleo de um sistema operativo?
- Como é feita a mudança entre o modo núcleo e o modo utilizador? É feita por hardware ou por software? Porquê?
- Por que razão existem os processos sistema? Diferem dos processos de um utilizador em que medida?
- Qual é a diferença entre o que se designa por exceção e por interrupção?
- As chamadas sistemas provocam uma exceção ou uma interrupção? Porquê?
- Por que razão as chamadas sistemas provocam uma exceção (ou **trap**)?
- Quais são as vantagens da existência de chamadas sistema para o SO?
- Suponhamos a seguinte situação: “Em casa, apenas o António é que mexe na secretária onde guarda papel, lápis e canetas. Quando a Maria quer papel, pede ao António. Ele seleciona o papel e entrega-lhe. Quando o João quer um lápis, o António seleciona o lápis e entrega-lhe. Etc. Sempre que o Ricardo quer uma caneta, é também o António a entregar-lhe.” Como enquadra esta situação na gestão de recursos de SO? O António corresponde à execução em modo utilizador ou em modo núcleo? O que aconteceria se todos os recursos fossem acedidos diretamente por cada um dos utilizadores?

3-Processos

- O que é um processo?
- Problema
- Definição de um processo
- Estados de um processo

O que é um processo?

- Um processo é um programa em execução
  - **Programa é a entidade passiva:** sequência de instruções, que manipula valores de localizações (variáveis, parâmetros e retorno de funções).
    - É um objeto sem vida que corresponde a um conjunto de instruções (com alguns dados estáticos) à espera de ser executado.
  - **Processador:** dispositivo com capacidade de executar instruções.
  - **Processo é a entidade ativa:** programa em execução, designado por *instância* do programa.
  - **Um processo é identificado pelo seu PID:** Número que identifica univocamente o processo

    - PID é do tipo *pid_t (inteiro)*

    - *O seu valor máximo pode ser alterado:* */proc/sys/kernel/pid_max*

O problema

THE CRUX OF THE PROBLEM:
HOW TO PROVIDE THE ILLUSION OF MANY CPUS?
Although there are only a few physical CPUs available, how can the OS provide the illusion of a nearly-endless supply of said CPUs?

COMO DAR A IDEIA DE QUE EXISTEM MUITOS CPUS?
Embora existam apenas alguns CPU disponíveis, como se pode dar a ilusão de existirem infindáveis CPUs disponíveis?

Definição de um processo

- Um processo tem um estado interno, o estado da máquina – ***machine state*.**
  - O que pode o programa ler ou atualizar enquanto está a ser executado?
  - Que partes da máquina são importantes para a execução do programa?
    - Espaço de endereçamento (***address space***)
      - É necessária memória onde o programa tem acesso para ler e aceder a dados
      - É na memória onde o código do programa está guardado.
    - Registos (***registers***)
      - Muitas instruções utilizam os registos para a sua execução.
    - Contador (***program counter***)
      - Informação guardada num registo especial indicando em que instrução o programa está a ser executado (também designado **instruction pointer**)
    - Pilha (***stack***)
      - Guarda parâmetros de funções, variáveis locais e endereços de retorno.
    - Lista de ficheiros abertos no disco (**persistent storage device**)
      - Informação E/S que inclui lista de ficheiros abertos pelo processo.

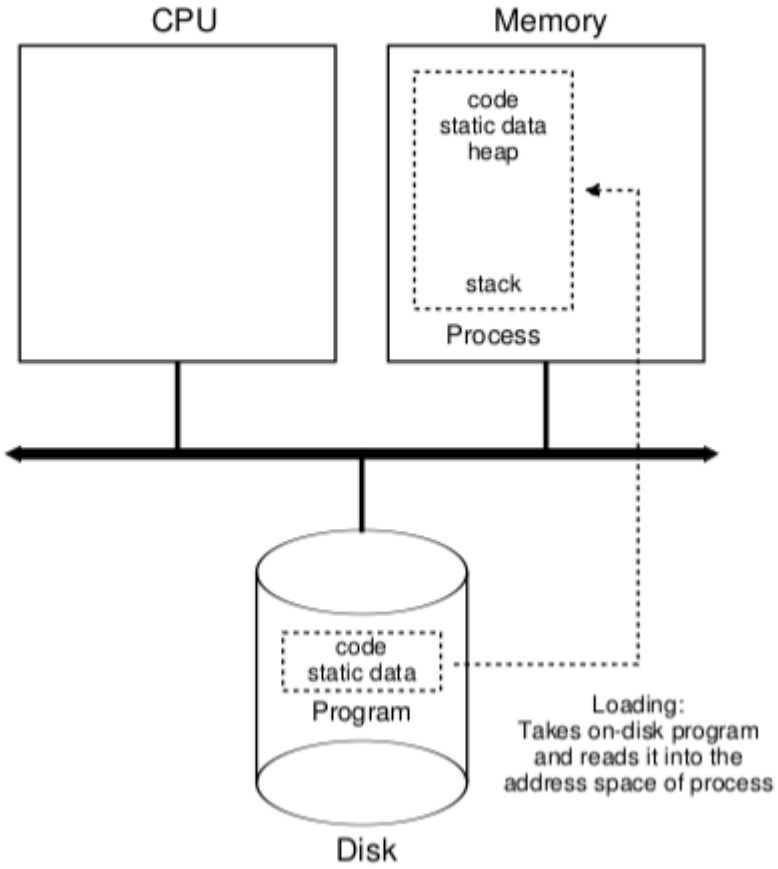
Processos API

- A interface para a programação de processos - ***Application Programming Interface*** ou ***API***
  - Conjunto de comandos que permitem interagir com os processos.
  - A janela de comandos do Windows e do Linux permitem essa interação.
- O que pode ser incluído numa interface do sistema operativo?
  - **Criar (*Create*)**: Um sistema operativo deve incluir um método para criar novos processos.
  - **Eliminar (*Destroy*)** : Deve existir também um comando para eliminar processos. Alguns terminam sozinhos, outros será útil terminá-los.
  - **Esperar (*Wait*)**: Por vezes é útil esperar que um processo termine.
  - **Outros controlos**: Por exemplo, suspender ou reiniciar um processo.
  - **Estado (*Status*)**: Comandos que permitem identificar o estado do processo, se está em execução, há quanto tempo, etc.

Carregamento de um processo

- Trazer para memória o programa - carregar o código e os dados estáticos, isto é as variáveis inicializada- no **espaço de endereço – address space,** do processo.
  - Em SO antigos, o processo de carregamento era feito antes de iniciar a execução do programa – de forma **ansiosa - *eagery*.**
  - Atualmente é feito pouco a pouco, consoante as necessidades – **de forma relaxada - *lazily***.
- Reservar espaço para a ***pilha - stack***.
  - São guardadas informações sobre o estado de variáveis utilizadas pelo programa (por exemplo, os parâmetros associados às funções que são chamadas)
- Criar um espaço para os dados num ***amontoado - heap***, onde estes são guardados à medida que o programa é executado – para reservar espaço para os dados, em C utiliza-se o comando **malloc()** [executar *man malloc* na janela de comandos]

Carregamento de um processo



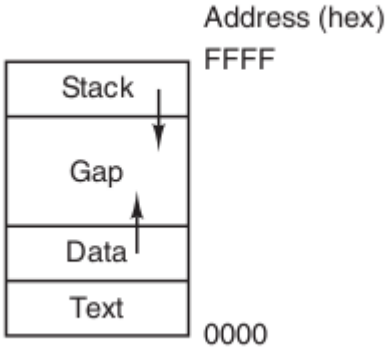


Figure 4.1: **Loading: From Program To Process**

Estados de um processo

- Nem sempre um processo está a ser executado.
  - Tem um estado “running”
  - O CPU determina se é esse processo que é executado.
    - Um programa em execução significa que o código é lido linha a linha (Em *C*, inicializado a partir do *main()).*
  - *O processo é bloqueado sempre que se inicia um processo de Entrada/Saída (exemplo, introduzir dados pelo utilizador).*

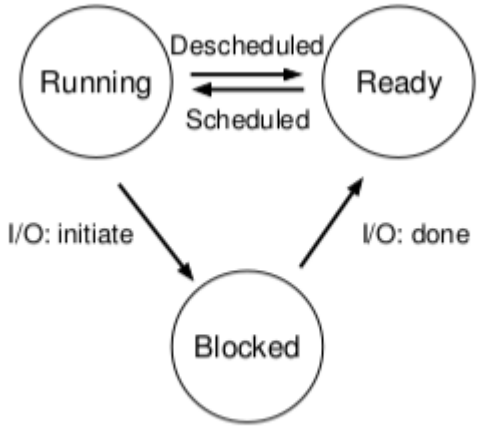
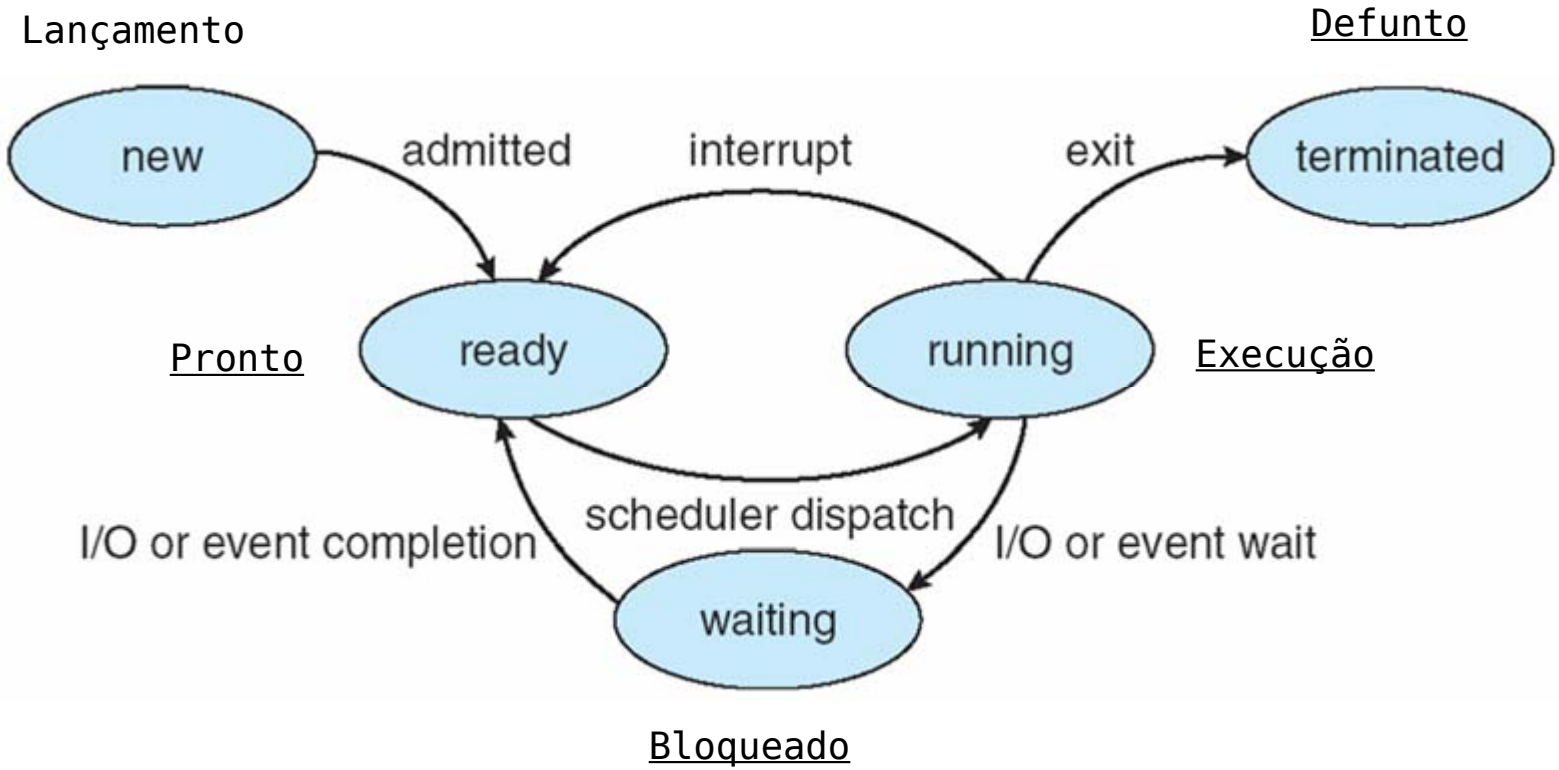


Figure 4.2: **Process: State Transitions**

Estados de um processo



Scheduler dispatch: Despacho seleciona para execução

Interrupt: Despacho suspende execução

I/O, event waiting: Aguarda conclusão de outro processo, p.ex. Leitura de ficheiro

I/O, event completion: Outro processo sinaliza conclusão

Estados de um processo

- Execução de processos que utilizam o CPU

Time	$Process_0$	$Process_1$	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	
4	Running	Ready	$Process_0$ now done
5	–	Running	
6	–	Running	
7	–	Running	
8	–	Running	$Process_1$ now done

- Execução de processos que utilizam o CPU e E/S

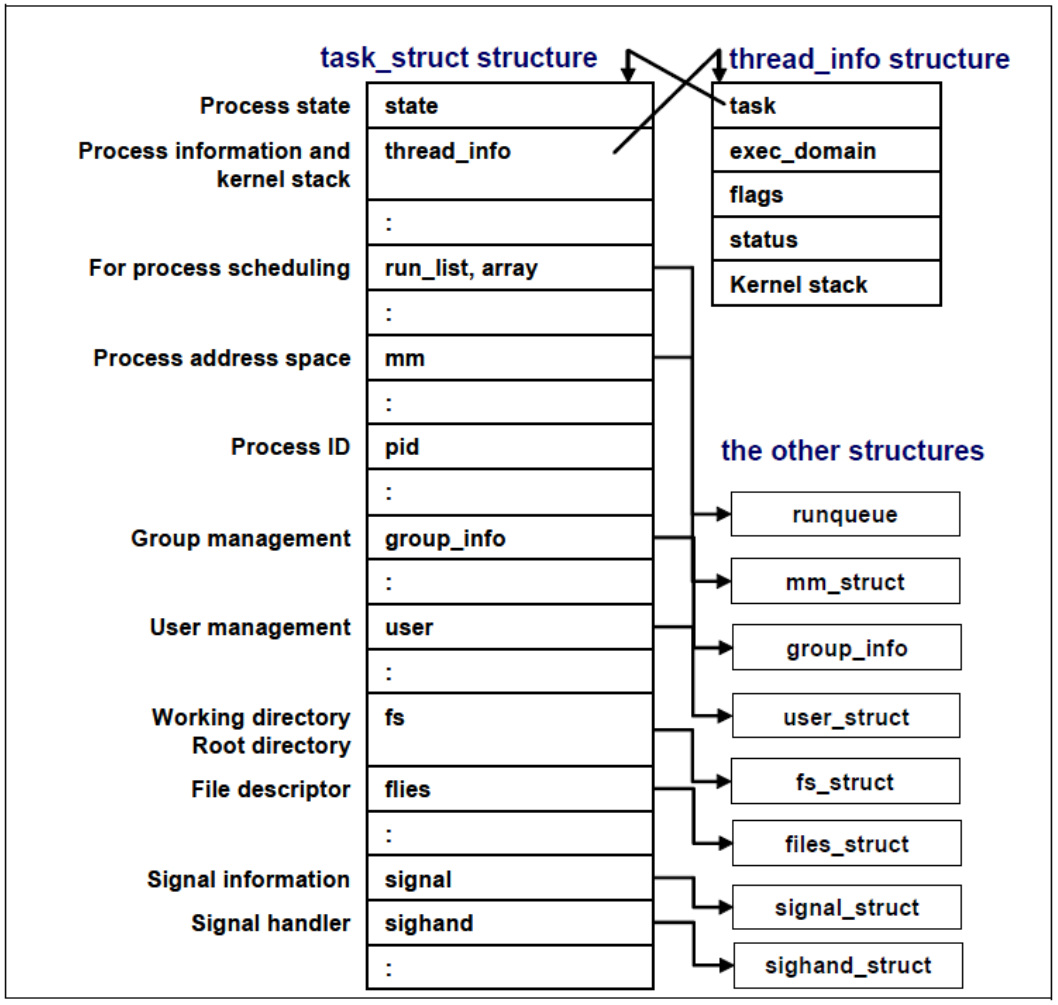
Time	$Process_0$	$Process_1$	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	$Process_0$ initiates I/O
4	Blocked	Running	$Process_0$ is blocked,
5	Blocked	Running	so $Process_1$ runs
6	Blocked	Running	
7	Ready	Running	I/O done
8	Ready	Running	$Process_1$ now done
9	Running	–	
10	Running	–	$Process_0$ now done

Estrutura de Dados de um Processo

- Um SO é um programa e tem, como qualquer programa, estruturas de dados que registam os elementos mais relevantes de informação.
  - Um SO guarda:
    - uma **lista de processos** de todos os processos no estado ***ready***.
    - Quais os processos que estão em execução.
    - Quais os processos que estão bloqueados.
    - Quando um evento de E/S se completa o OS deve ter a certeza de qual deve ser o processo que deve ser acordado e tornado pronto para ser executado novamente.
  - O SO guarda ainda um conjunto de dados necessário para a identificação de cada um dos processos **- Process Control Block (PCB)**
    - Estrutura em C que guarda a informação acerca dos processos.

PCO

- Descritor do processo (task_struct)



PCO

```
// the registers xv6 will save and restore
// to stop and subsequently restart a process
struct context {
  int eip;
  int esp;
  int ebx;
  int ecx;
  int edx;
  int esi;
  int edi;
  int ebp;
};

// the different states a process can be in
enum proc_state { UNUSED, EMBRYO, SLEEPING,
                  RUNNABLE, RUNNING, ZOMBIE };

// the information xv6 tracks about each process
// including its register context and state
struct proc {
  char *mem;                  // Start of process memory
  uint sz;                    // Size of process memory
  char *kstack;               // Bottom of kernel stack
                              // for this process
  enum proc_state state;      // Process state
  int pid;                    // Process ID
  struct proc *parent;        // Parent process
  void *chan;                 // If non-zero, sleeping on chan
  int killed;                 // If non-zero, have been killed
  struct file *ofile[NOFILE]; // Open files
  struct inode *cwd;          // Current directory
  struct context context;     // Switch here to run process
  struct trapframe *tf;       // Trap frame for the
                              // current interrupt
};
```

Questões

- Qual é a diferença entre um programa e um processo? Qual deles é uma entidade passiva e porquê?
- Quais são os estados de um processo? O que significam?
- Explique qual é a diferença entre “code”, “static data”, “heap” e “stack”.
- O que significa um processo em execução?
- Um processo que utiliza E/S utiliza tempo de CPU?
- Qual é a informação que um sistema operativo guarda de um processo?