Fundamentos de Sistemas de Operação

Programas, Processos e o SO:

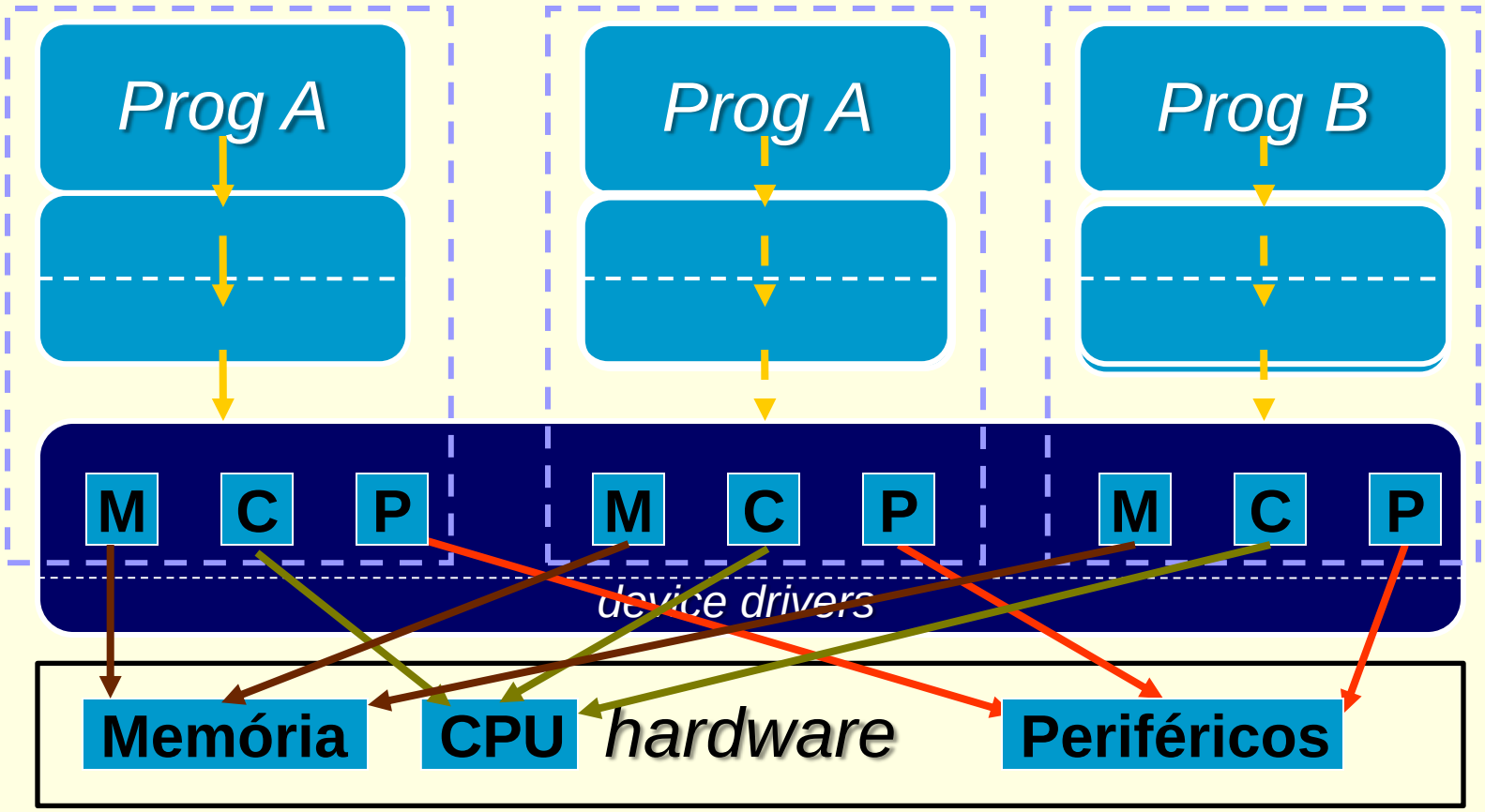
Continuação

Resumo da aula anterior…

- *O processo, conceito oferecido pelo SO,*
  - *"É um contentor" que tem um CPU, uma memória e periféricos idênticos aos da "máquina real".*
  - *Cada programa corre nesta máquina virtual como se mais nada existisse…*
  - *O SO encarrega-se de garantir que as zonas de memória, os "CPUs" e "periféricos virtuais" "adjudicados" a cada um dos processos e que funcionam correctamente sem se "sobreporem" ou "interferirem"*

O processo como uma Máquina Virtual
que executa um programa

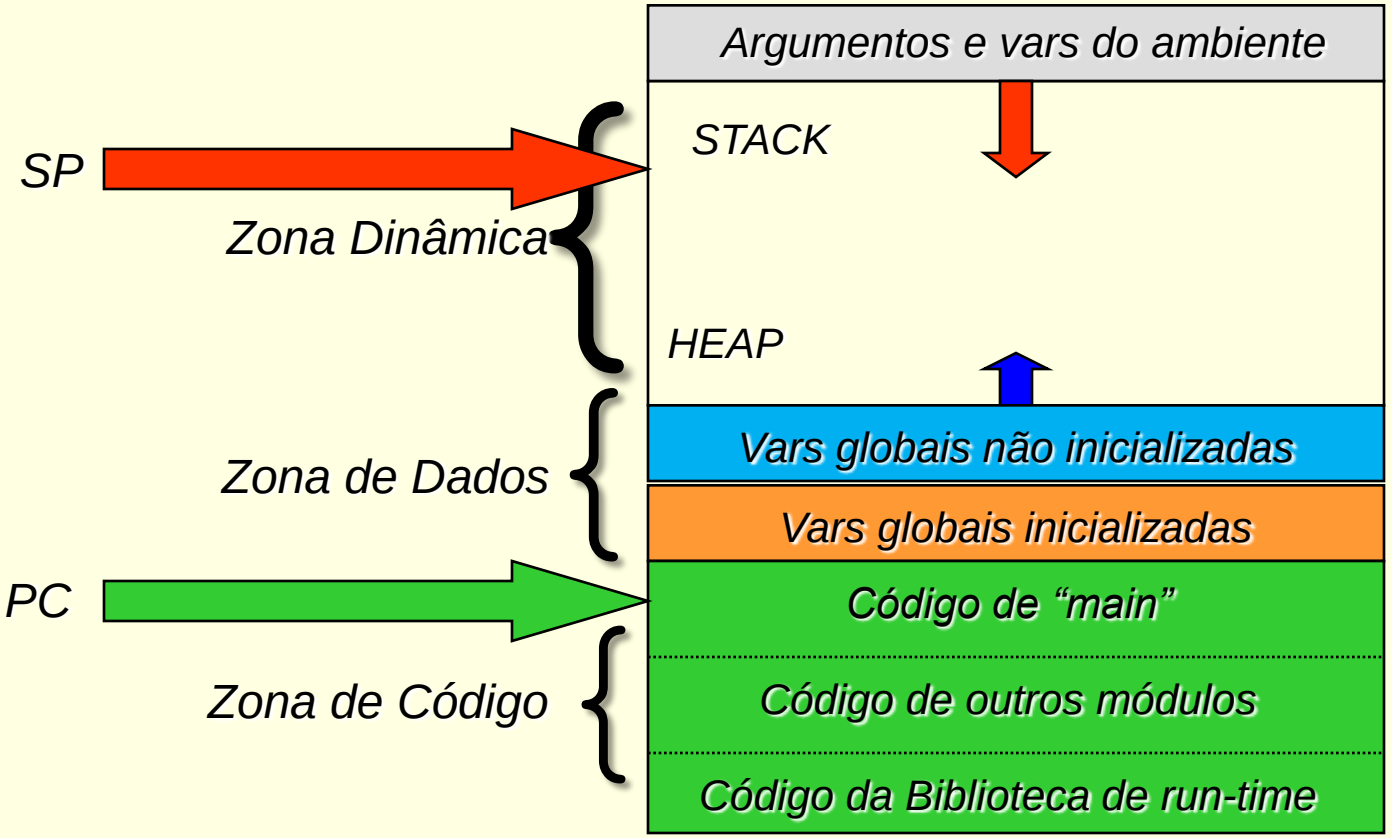
Unix Windows NT Netware MacOS DOS/VS Vax/VMS
Linux Solaris HP/UX AIX Mach
Chorus



Espaço de endereçamento de um processo

- O espaço de endereçamento de um processo,
  - É o conjunto de posições de memória que podem ser referenciadas pelo programa.
  - Pode ser subdividido em várias zonas, e segundo vários critérios: Dados vs. Código; Dimensão Fixa vs. Variável
  - **Dados:** Constantes, Variáveis: Inicializadas ou não-inicializadas
  - **Código:** Constante (não se modifica durante a execução).

Mapa do Espaço de Endereçamento (Linux)



Para pensar…

- Recorde o programa do Trabalho prático TP0,
  - Tente identificar as zonas no mapa de endereçamento em que vão “existir” as várias entidades presentes no código fonte…

```
int x = 5;
int main( int argc, char *argv[] )
{
        int y;  int *z;

        if( argc != 2 )
        {       printf( "%s num\n", argv[0] );
                return 1;
        }
        y = atoi( argv[1] );
        z = &x;
        printf( "%d\n", *z * y );
        return 0;
}
```

`fork()`: *criar um novo processo* (1)

- Observe o programa
  - Durante 30s vai ter a oportunidade de observar que existem dois processos em execução, sendo que o PPID do filho é igual ao PID do pai…

```
int main( int argc, char *argv[] )
{
        if( fork() ) {
          printf( "Sou o pai\n" ); sleep(30);
        } else {
          printf( "Sou o filho\n" ); sleep(30);
        }

        return 0;
}
```

`fork()`: *criar um novo processo* (2)

- Como funciona?
  - Quando o (único) processo em execução executa a instrução `fork()`, o SO (Unix e descendentes…)
    - Suspende o pai; cria um novo processo (o filho), por "clonagem" do pai: clona o espeço de endereçamento, os atributos, o estado do CPU, etc. Depois,
    - Como resultado do `fork()` retorna ao pai o PID do filho, e retorna ao filho o valor zero. Reactiva (termina a suspensão) dos dois processos
    - Isto faz com que o pai continue a execução no `else`, enquanto o filho continua na zona "then"

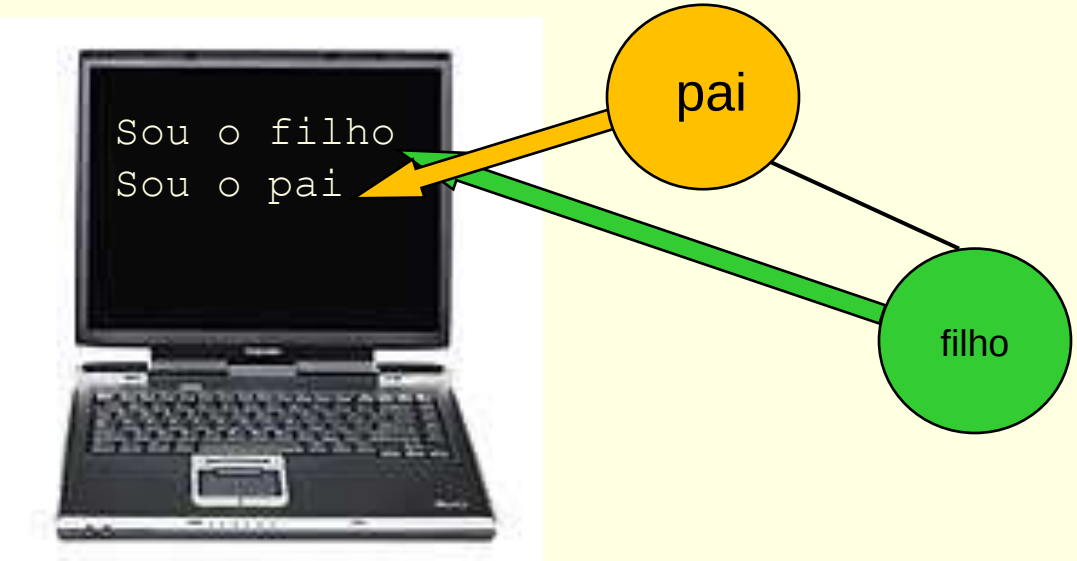
1a instr. executada pelo processo filho

1a instr. executada pelo processo pai

```
if( fork() ) {
    printf( "Sou o pai\n" ); sleep(60);
} else {
    printf( "Sou o filho\n" ); sleep(60);
}
```

`fork()`: *criar um novo processo* (3)

- As mensagens (do pai e filho) aparecem no mesmo local...



  - Porque os canais de comunicação do processo pai com o "exterior" foram clonados para o filho… mais detalhes sobre "canais" mais tarde ☺

`exec()`: *executar um programa* (1)

- O `fork()` cria um processo, mas mantém o mesmo programa…
  - O `exec()` permite a um processo "substituir" o programa que está a correr por um outro…

```
int main( int argc, char *argv[] )
{
  if( fork() ) {
    printf( "Sou o pai\n" ); sleep(30);
} else {
    execl( "/bin/ls", "ls", "-l", NULL); }

  return 0;
}
```

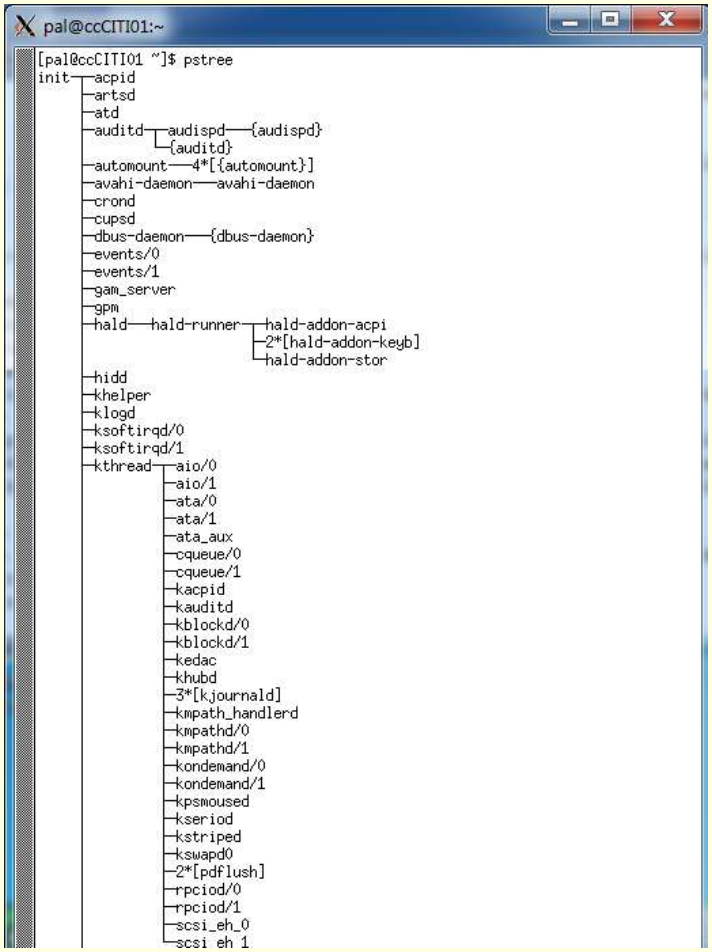
  - Neste exemplo, o filho é "clone" durante uns brevissimos milésimos de segundo, para logo a seguir ser o programa que lista os ficheiros existentes numa directoria (pasta).

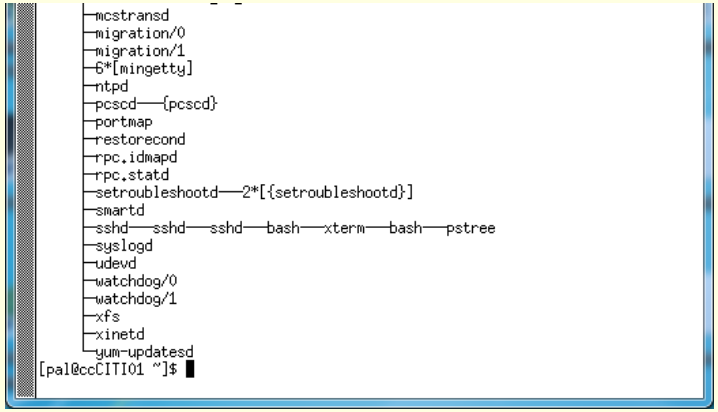
`exec()`: *executar um programa* (2)

- O `exec()`
  - Não é uma função, mas uma família de funções que fazem todas o mesmo, mas diferem nos argumentos - nuns casos é mais apropriado/fácil usar uma, noutros outra…
    - `execl()`, `execle()`, `execlp()`
    - `execv()`
    - `execvp()`: a verdadeira chamada de sistema; as outras são de biblioteca
- A única mudança é a do programa executável (também designado imagem) e, por consequência, do espaço de endereçamento
  - Mantem-se o PID, PPID, utilizador, etc... Mantêm-se os canais que o pai abriu (a menos que o filho os feche).

A árvore de processos no Linux

Unix Windows NT Netware MacOS DOS/VS Vax/VMS
Linux Solaris HP/UX AIX Mach
Chorus





`wait()`: *esperar que um filho termine* (1)

- Uma situação comum é lançar um processo para desempenhar uma tarefa, e quando este acaba lançar outro…
  - O `wait()` permite ao pai esperar (bloqueado) até que o processo filho termine; aí, o pai é desbloqueado…

```
...
if( fork() ) {
  wait(NULL);
  printf( "Sou o pai\n" );
} else {
  execl( "/bin/ls", "ls", "-l", NULL);
}
```

  - Neste exemplo, a mensagem "Sou o pai" nunca se misturará com a lista de ficheiros…

`wait()`: *esperar que um filho termine* (2)

- Quando 2 processos (pai, filho) são executados concorrentemente
  - Se o pai chega ao `wait()` antes do filho terminar: o pai espera bloqueado até que o filho termine; aí, o pai é desbloqueado…
  - Se o pai chega ao `wait()` depois do filho já ter terminado
    - O filho termina, mas deixa atrás de si um processo zombie (`defunct`); os recursos usados pelo filho ainda não foram totalmente devolvidos ao SO…
    - Quando o pai chega ao `wait()`, a libertação de recursos do filho é concluída, e o pai continua a sua execução…
  - Se o pai não faz a chamada `wait()` e termina antes do filho…
    - O processo `init` passa a ser o pai…
    - Quando o filho termina, deixa atrás de si um processo zombie (`defunct`), mas ao fim de um certo tempo o `init` trata de libertar os recursos…

Criar um novo processo: API Windows

- A chamada de sistema no Windows

```
BOOL CreateProcess( LPCSTR AppName,
                    LPTSTR CommLine,
                    LPSECURITY_ATTRIBUTES ProcAttr,
                    LPSECURITY_ATTRIBUTES ThAttr,
                    BOOL InheritHandles,
                    DWORD CreatFlags,
                    LPVOID Environment,
                    LPCSTR CurrDirectory,
                    LPSTARTUPINFO StartupInf,
                    LPPROCESS_INFORMATION ProcInfo )
```

Note que, entre outros parâmetros, é especificado o executável (`AppName`), o ambiente (`Environment`) e a directoria corrente…

Algumas chamadas *interessantes para processos*

- `getpid()`: Devolve o PID do processo que a executa
- `getppid()`: Devolve o PPID – PID do pai do processo que a executa
- `kill(pid,…)`: "Mata" o processo cujo PID é indicado [o processo pode não morrer… ☺]
- `_exit(st)`: Termina a execução do processo que a executa, e passa o valor `st` ao processo pai

Para pensar…

- Quantos processos são criados?

```
...
  for (i= 0; i < 3; i++)
    fork();
...
```

- Supondo que se queriam criar apenas 3, corrija o programa ☺

Programação concorrente

Unix Windows NT Netware MacOS DOS/VS Vax/VMS Linux Solaris HP/UX AIX Mach Chorus

- Quando uma aplicação é realizada por vários processos… Um novo leque de problemas se coloca ao programador:
  - Como é que os processos podem comunicar? Trocar dados entre si? Afinal, todos os esforços que fizemos até ao momento foram no sentido de **isolar** os processos uns dos outros…
  - Como é que os processos se podem sincronizar? Isto é, por exemplo, como é que um processo pode suspender a sua execução até que outro chegue a um dado ponto da sua execução?
    - O wait() pode ser usado para isso, mas só num caso: o filho terminar…
- Estas (e outras) questões serão abordadas mais tarde… ☺