Processos ao nível do SO

José C. Cunha, DI-FCT/UNL

Concorrência em sistemas de operação

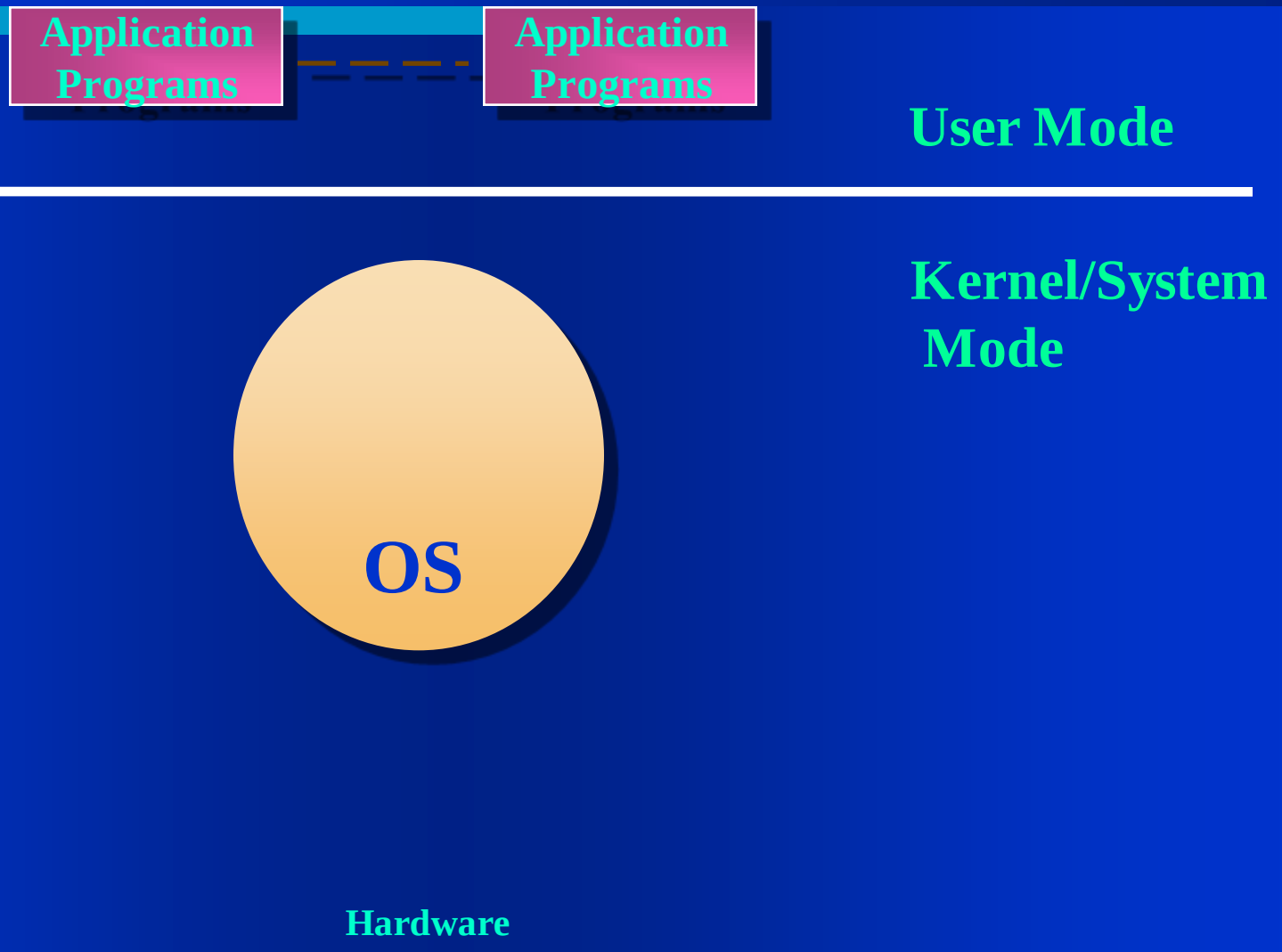
- ***Periféricos operando simultaneamente***
- ***Multiprogramação***
- ***Eventos em instantes imprevisíveis***

SO monolíticos

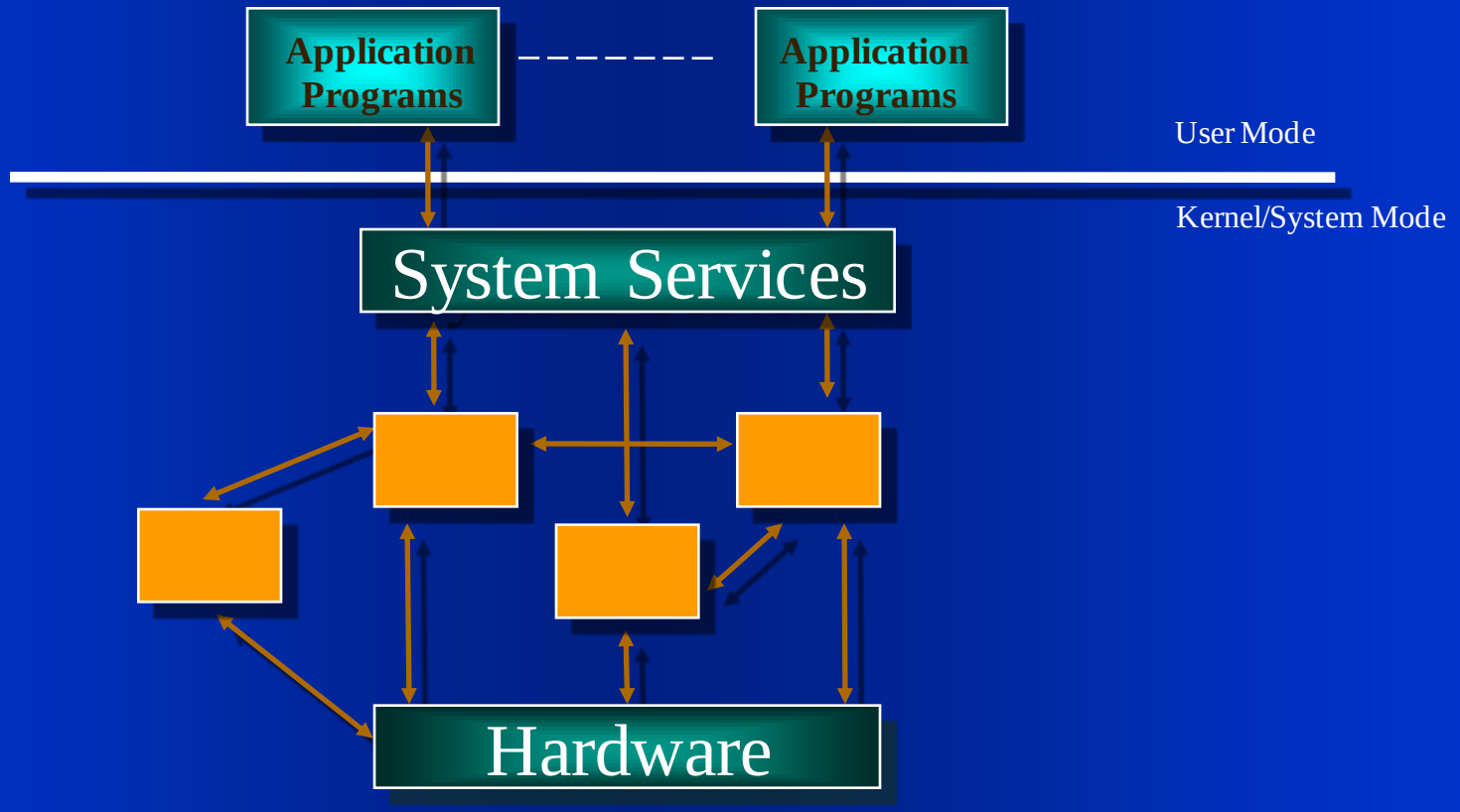
- **Programas e periféricos pedem serviços ao SO**
- **SO ficaria mais simples se todas as acções do SO fossem executadas com as interrupções desligadas**

mas…

SO: abordagem tradicional



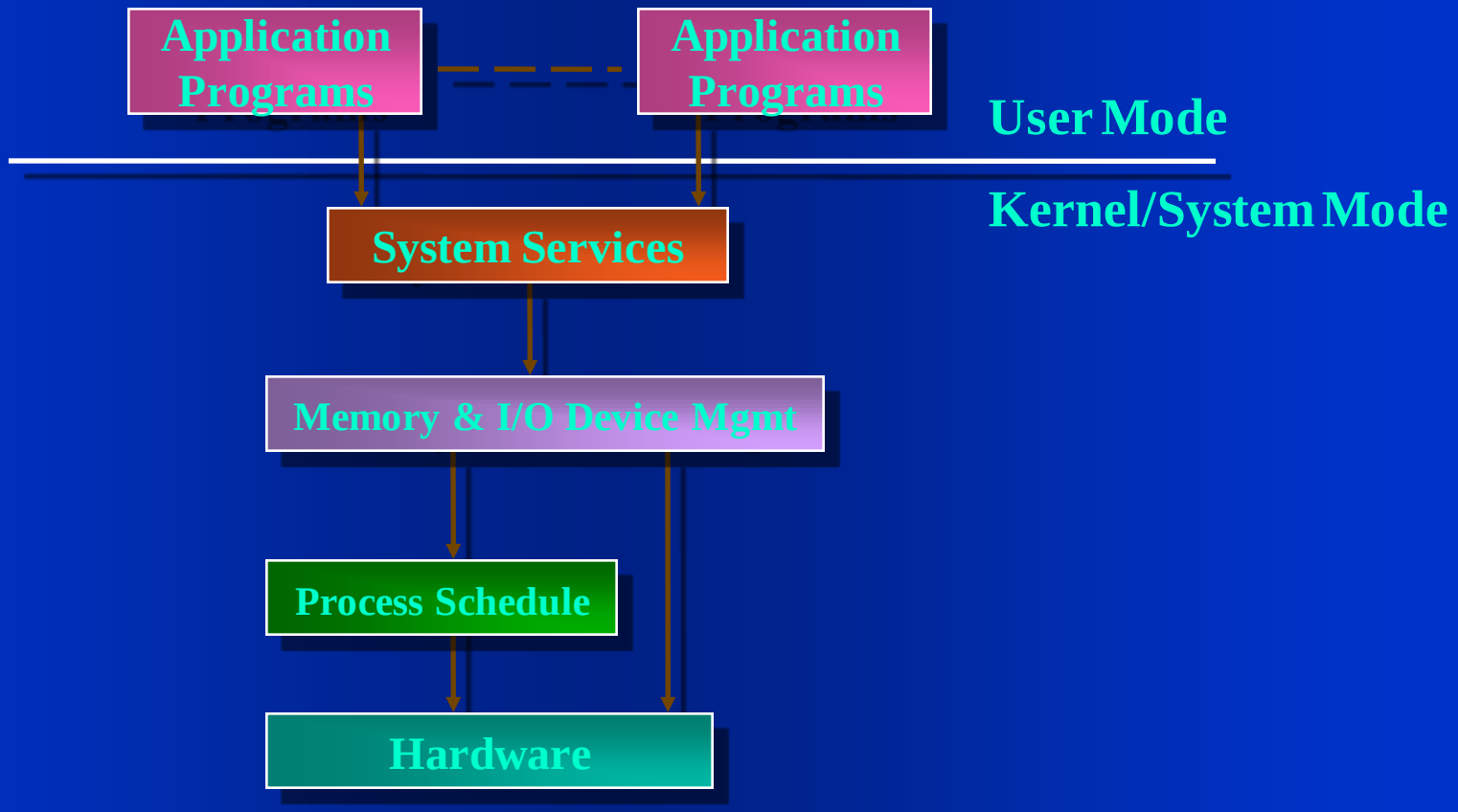
SO monolítico



SO monolíticos

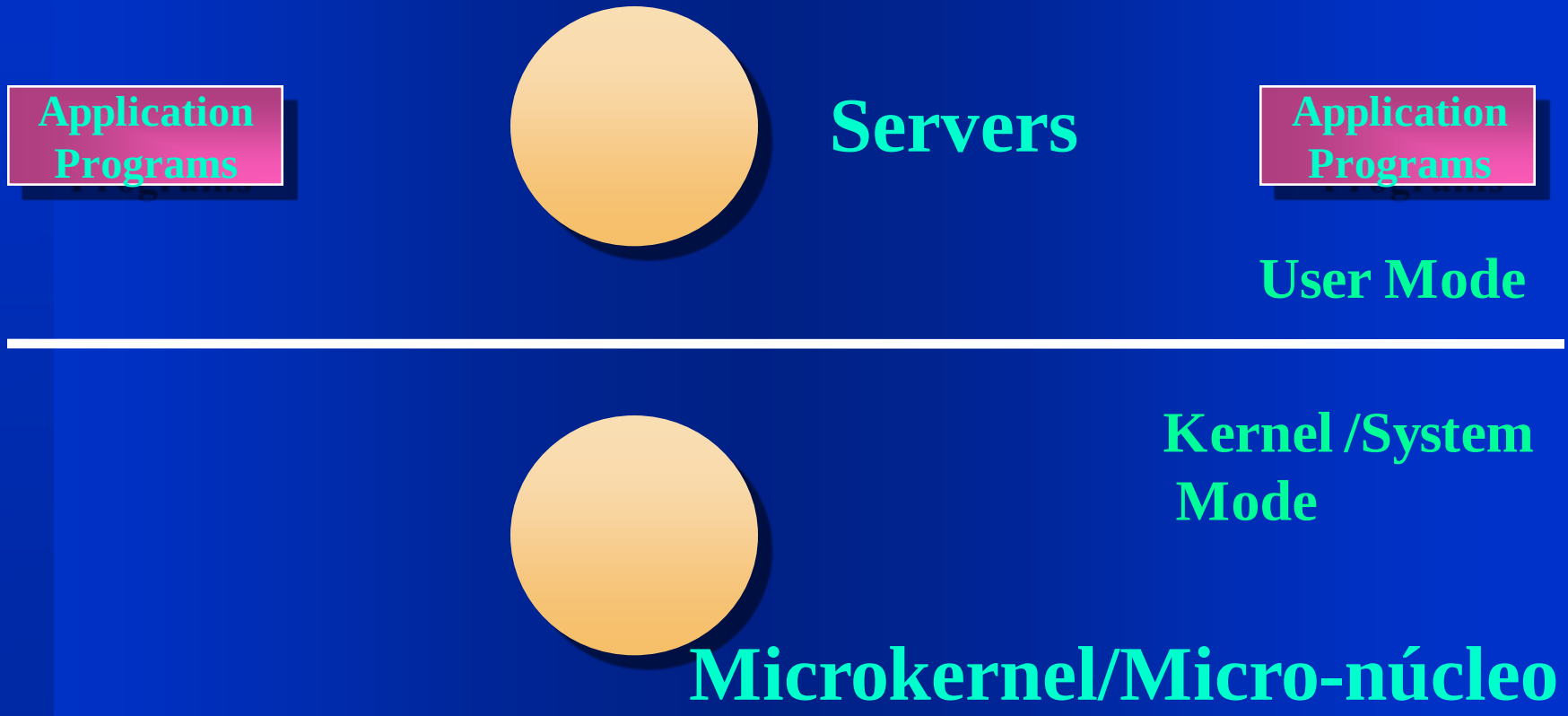
- Programas e periféricos pedem serviços ao SO
- Todas as acções do SO com interrupções desligadas
- SO fica mais simples, mas…
- Processos e periféricos ficam a aguardar muito tempo
- SO grandes ficam muito complexos
- Difíceis de ampliar com novos serviços

SO organizado em camadas



Cada camada acede a uma interface de mais baixo nível

Tendência para SO mais flexíveis



SO baseados em processos

- **Nem todas as acções do SO exigem as interrupções desligadas**

SO baseados em processos

- **Nem todas as acções do SO exigem as interrupções desligadas**
- **Maior descentralização das acções do SO**

SO baseados em processos

- **Nem todas as acções do SO exigem as interrupções desligadas**
- **Maior descentralização das acções do SO**
- **Processos:**

-- entidades assíncronas interrompíveis

-- partilham o CPU e passam por estados: *Pronto, Executando, Bloqueado*

SO baseados em processos

- **Nem todas as acções do SO exigem as interrupções desligadas**
- **Maior descentralização das acções do SO**
- **Processos:**

-- entidades assíncronas interrompíveis

-- partilham o CPU e passam por estados *Pronto, Executando, Bloqueado*

- **Processo para acções de 'device driver'**

SO baseados em processos

- Nem todas as acções do SO exigem as interrupções desligadas
- Maior descentralização das acções do SO
- Processos:

-- entidades assíncronas interrompíveis

-- partilham o CPU e passam por estados executando, bloqueado, pronto

- Processo para acções de ‘device driver’
- Processo para execução de cada programa

Noção de processo associada a cada *device driver*

para cada periférico

Responsável pela sua operação física

Pela detecção do fim de operações e erros

Pelo tratamento de interrupções

Pelo serviço dos pedidos dos programas

Vantagens:

-- maior clareza nos módulos do SO

-- estruturas de dados descentralizadas e sob controlo de processos dedicados

-- maior eficiência no atendimento dos pedidos dos programas e dos periféricos na gestão da concorrência

O Núcleo do SO

- **Distribui o tempo de CPU pelos processos**

O Núcleo do SO

- **Distribui o tempo de CPU pelos processos**
- **Atende interrupções e assinala-as aos respectivos processos responsáveis, sob a forma de sinais**

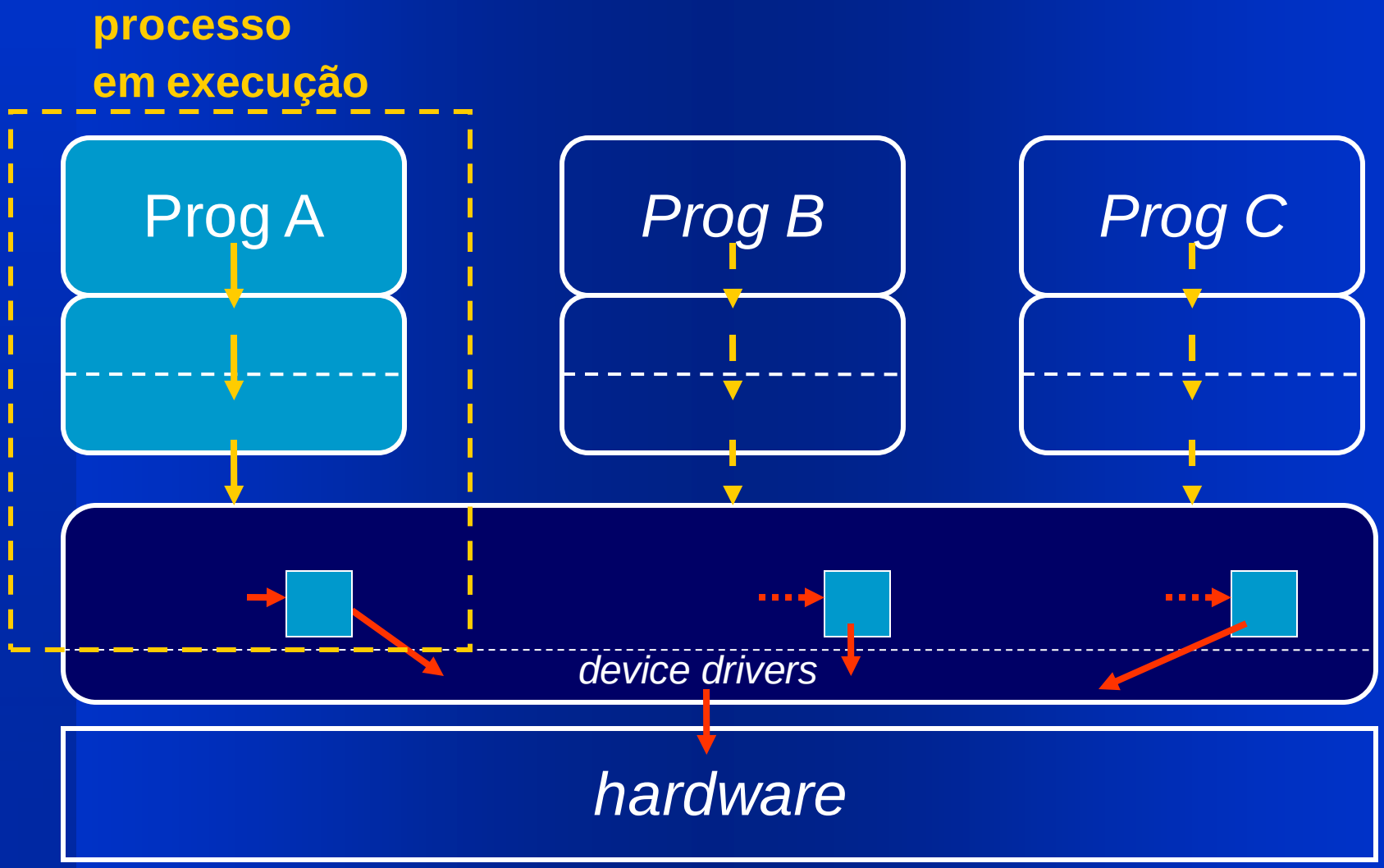
O Núcleo do SO

- Distribui o tempo de CPU pelos processos
- Atende interrupções e assinala-as aos respectivos processos responsáveis, sob a forma de sinais
- Oferece funções para:
  - Processos fazerem pedidos ao SO

O Núcleo do SO

- Distribui o tempo de CPU pelos processos
- Atende interrupções e assinala-as aos respectivos processos responsáveis, sob a forma de sinais
- Oferece funções para:
  - Processos fazerem pedidos ao SO
  - Processos comunicarem entre si e sincronizarem as suas acções concorrentes

Cada processo, seus canais de I/O



Processo ≠ Programa

- **Programa é estático**
  - **código e dados**

Processo ≠ Programa

- **Programa é estático**
  - **código e dados**
- **Processo: instância de execução de um programa**
  - **código e dados**
  - **estado: pilha, registadores (PC, SP, …)**
  - **canais de I/O**
  - **atributos e outros recursos…**

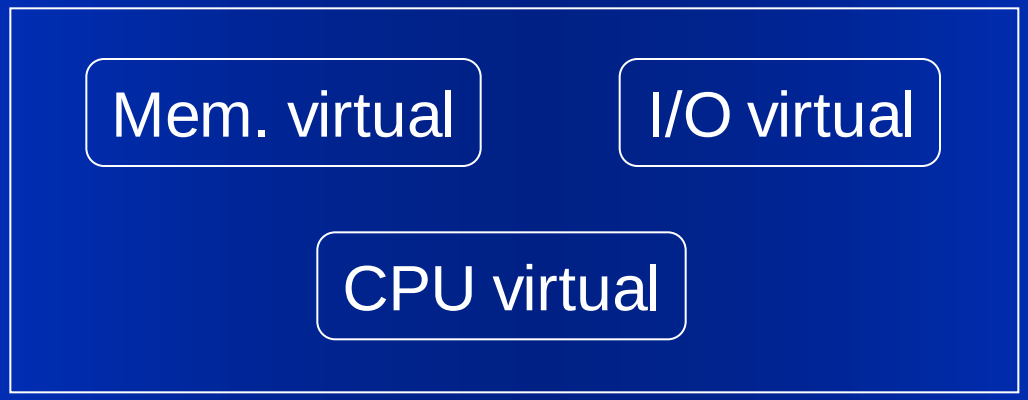
Processo ≠ Programa

- Programa é estático
  - código e dados
- Processo é uma instância de execução
  - código e dados (estes vão variando)
  - estado: pilha, registadores (PC, SP, …)
  - canais de I/O
  - atributos e outros recursos…
- O **mesmo programa** pode ser executado em **diferentes processos**:
  - p.ex. diferentes utilizadores

Controlo de Processos

- **Ambiente de execução dos programas:**
  - **instruções da máquina *hardware***
  - **chamadas a funções de biblioteca**
  - **chamadas ao sistema de operação**
- **Suporte duma máquina virtual dedicada a cada processo:**
  - **Processador Virtual**
  - **Memória Virtual**
  - **Canais Virtuais de Comunicação**

Máquina virtual de um processo



- **O SO gere e projecta na máquina real**
  - **Mem. Virtual → mapa de memória**
  - **CPU virtual → CPU real / *time-sharing***
  - **I/O virtual → canais de I/O → periféricos**

Um Processo no SO

Ambiente de execução de um programa:

Um Processo no SO

Ambiente de execução de um programa:

-- Programa executável (instruções máquina)

Um Processo no SO

Ambiente de execução de um programa:

-- Programa executável (instruções máquina)

-- Zonas de dados e pilha

Um Processo no SO

Ambiente de execução de um programa:

-- Programa executável (instruções máquina)

-- Zonas de dados e pilha

-- Zona para salvaguarda de estado do CPU (PC, SP, outros registadores)

Um Processo no SO

Ambiente de execução de um programa:

-- Programa executável

-- Zonas de dados e pilha

-- Zona para salvaguarda de estado do CPU (PC, SP, outros registadores)

-- Informação sobre canais abertos para ficheiros e outra informação de controlo

Um Processo no SO

Ambiente de execução de um programa:

-- Programa executável

-- Zonas de dados e pilha

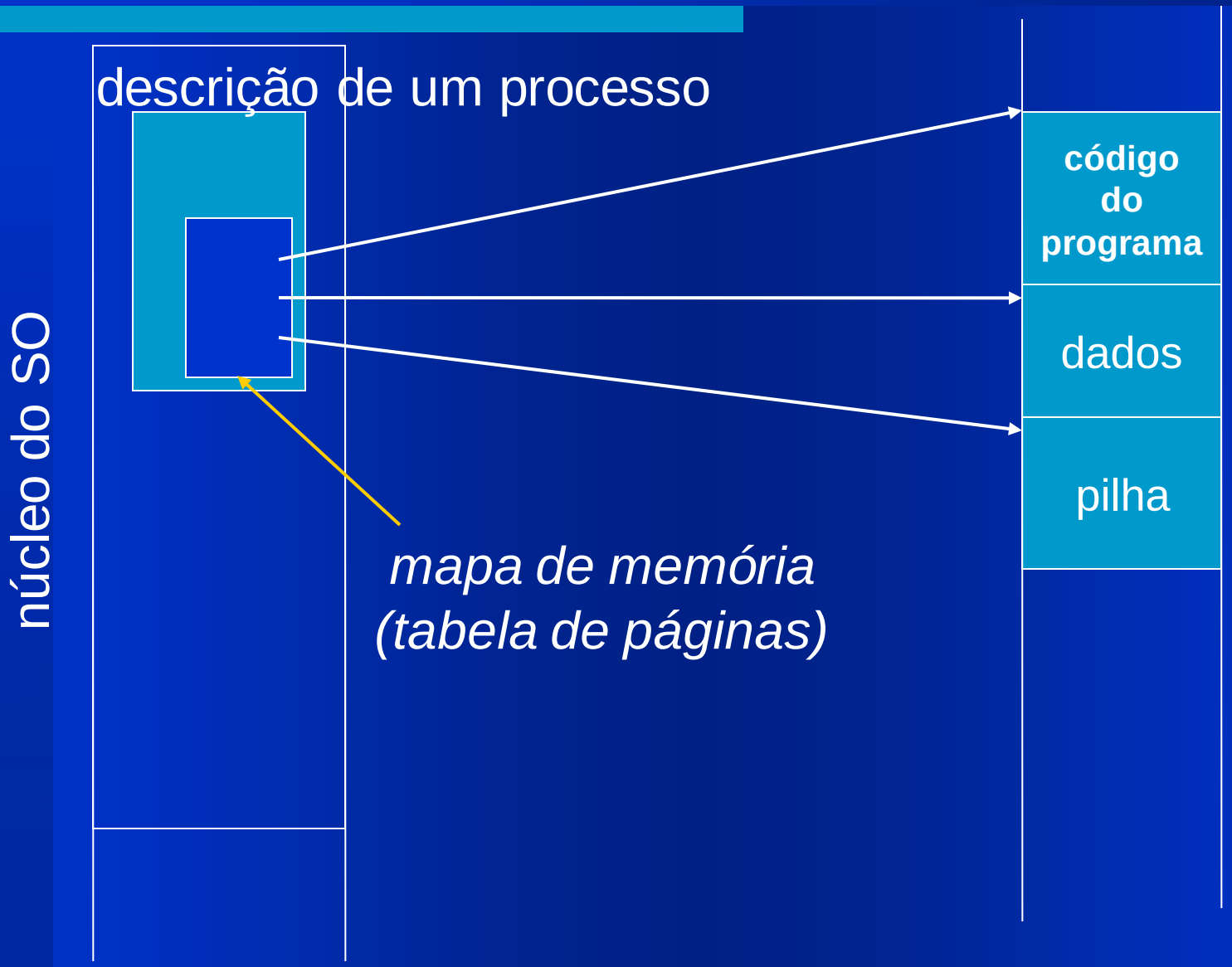
-- Zona para salvaguarda de estado do CPU (PC, SP, outros registadores)

-- Informação sobre canais abertos para ficheiros e outra informação de controlo

Guardada em Descritores internos do SO

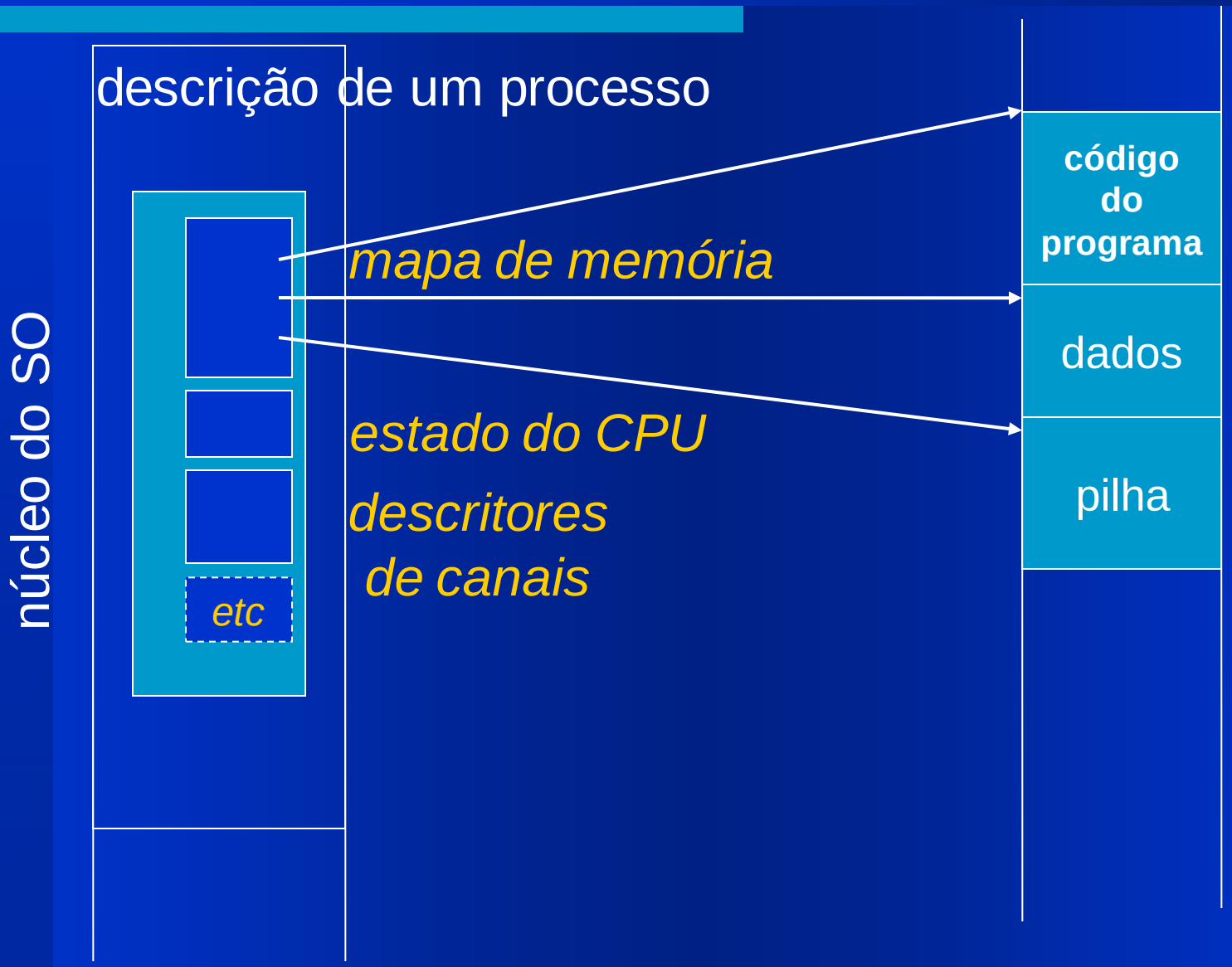
(Process Id: PID -- aponta para descritor)

Processo e sua memória



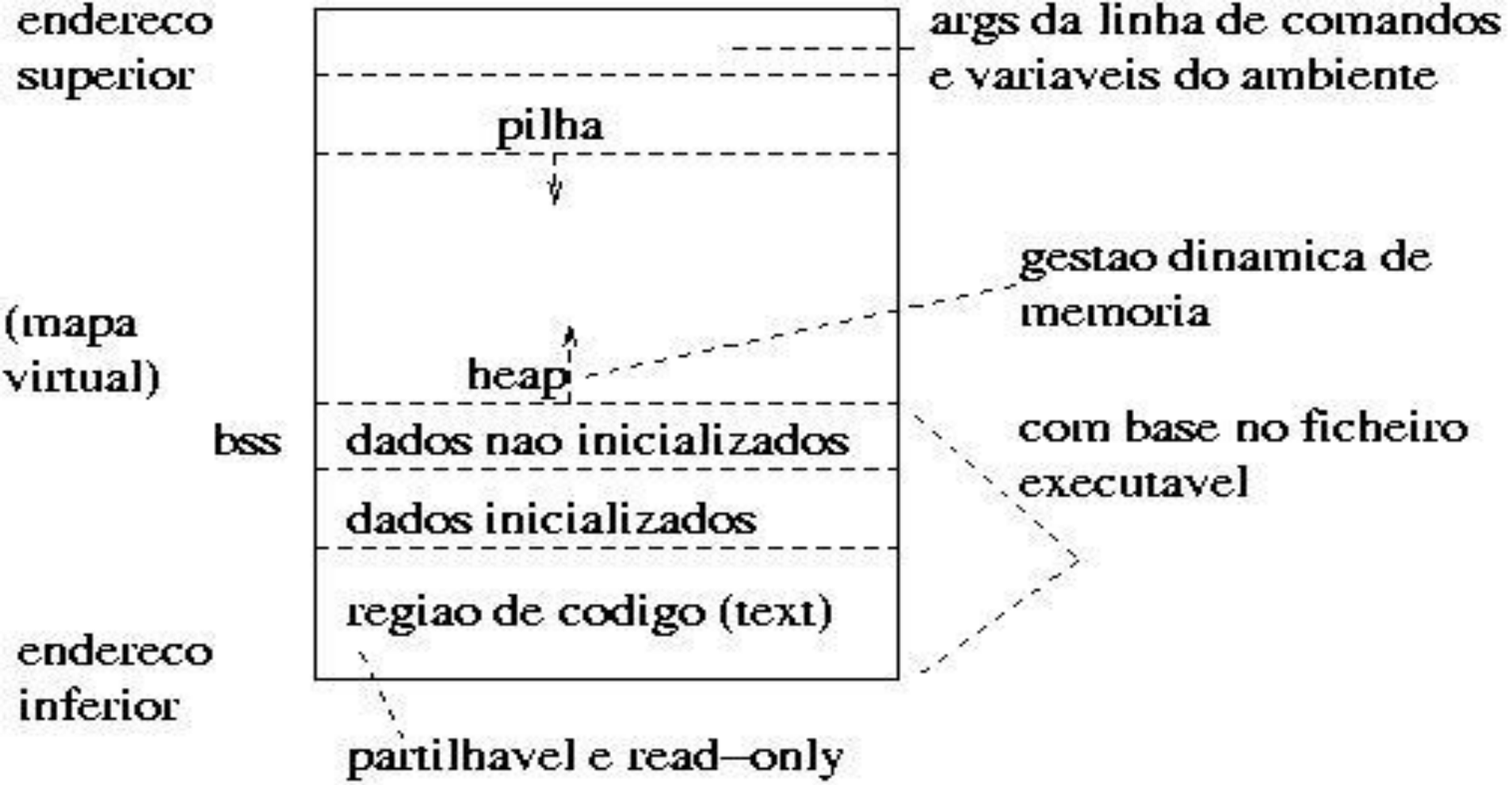
Estados lógicos de um processo

Processo em memória



Ambiente de um processo Unix

- **Mapa de memória do processo**
- **Inicialização -> *main()***
- **Execução e terminação do processo**



$size nome–ficheiro–executavel

text data bss decimal hexa

.. ..

em bytes

Início de execução de um processo

- **Activar um programa C:**
  - **uma função de inicialização (no "*start address*" especificado pelo compilador) é activada:**
    - **p.e. `_start` no *C Run Time* (`crt`)**

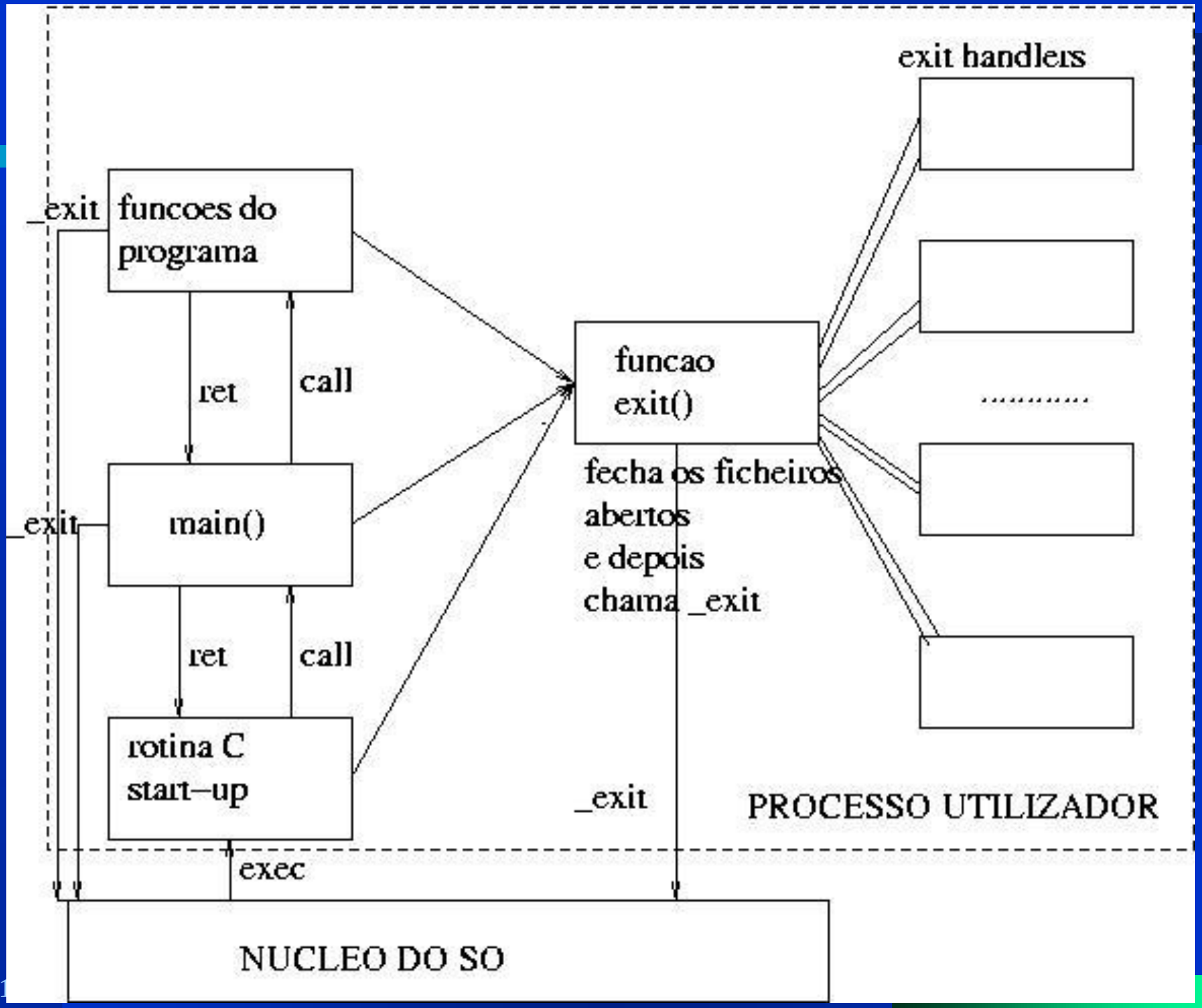
Início de execução de um processo

- **Activar um programa C:**
  - **uma função de inicialização (no "*start address*" especificado pelo compilador) é activada:**
    - **p.e. `_start` no *C Run Time* (`crt`)**
  - **obtém argumentos e vars. ambiente para o mapa do processo, iniciando a pilha**

Início de execução de um processo

- **Activar um programa C:**
  - **uma função de inicialização (no "*start address*" especificado pelo compilador) é activada:**
    - **p.e. `_start` no *C Run Time* (`crt`)**
  - **obtém argumentos e vars. ambiente para o mapa do processo, iniciando a pilha**
  - **e lança a função `main()`:**

```
main(argc, argv, environ);
```


exit handlers
_exit
funcoes do programa
ret
call
funcao exit()
.....
fecha os ficheiros abertos e depois chama _exit
_exit
main()
ret
call
rotina C start-up
_exit
PROCESSO UTILIZADOR
exec
NUCLEO DO SO

Terminação de um processo

- **Normal:**

-- retorno de main()

Terminação de um processo

- **Normal:**

-- retorno de main()

-- chamada exit(): acções de 'fecho', …

Terminação de um processo

- **Normal:**

-- retorno de main()

-- chamada exit(): acções de 'fecho', …

-- chamada _exit(): retorno imediato ao SO

Terminação de um processo

- **Normal:**

-- retorno de main()

-- chamada exit(): acções de 'fecho', …

-- chamada _exit(): retorno imediato ao SO

-- podem devolver o 'exit status' do processo

- **Anormal:**

Terminação de um processo

- **Normal:**

-- retorno de main()

-- chamada exit(): acções de 'fecho', …

-- chamada _exit(): retorno imediato ao SO

-- podem devolver o 'exit status' do processo

- **Anormal:**

-- chamada de abort()

Terminação de um processo

- **Normal:**

-- retorno de main()

-- chamada exit(): acções de 'fecho', …

-- chamada _exit(): retorno imediato ao SO

-- podem devolver o 'exit status' do processo

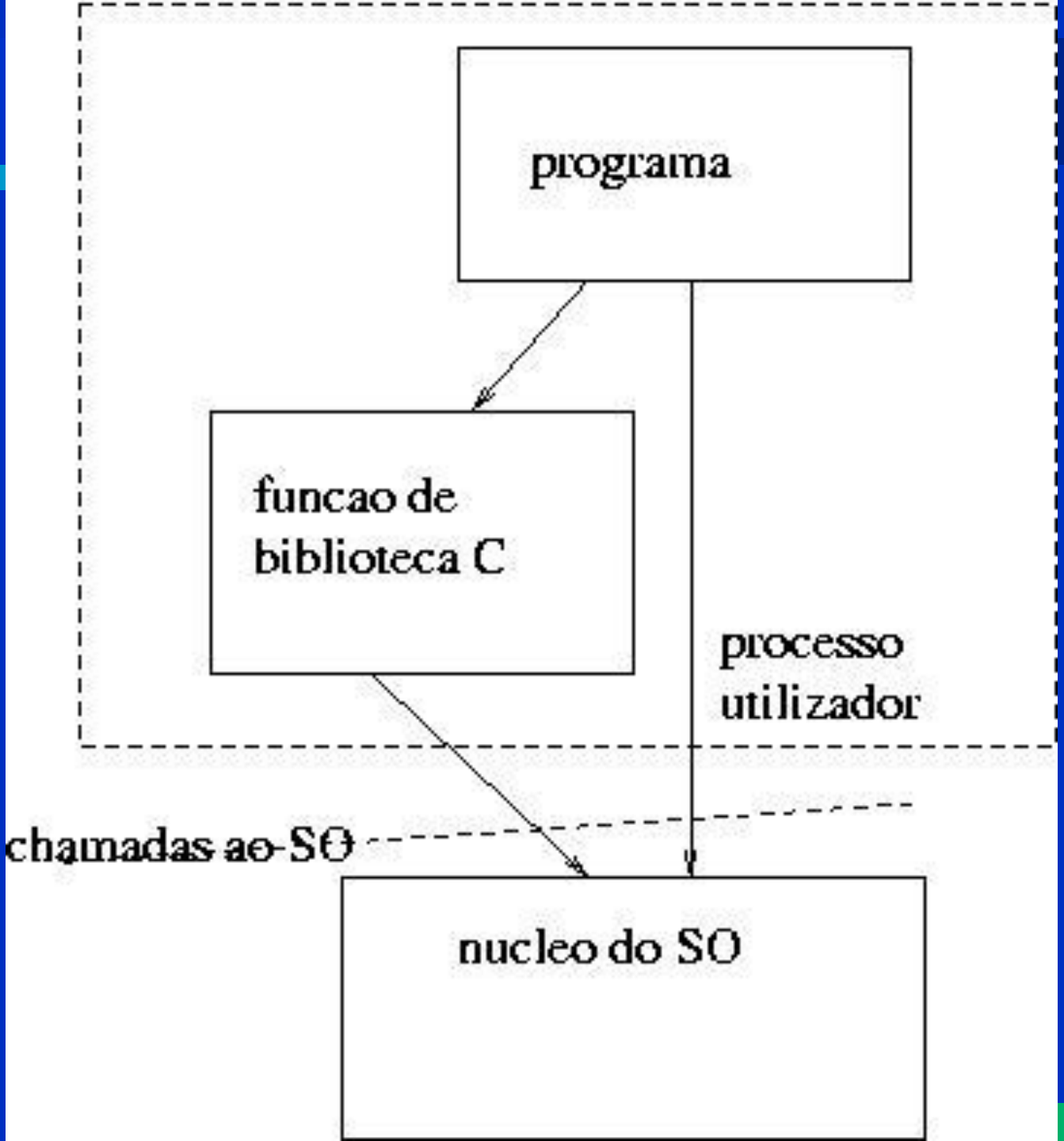
- **Anormal:**

-- chamada de abort()

-- ocorrência de sinal (hardware ou software)

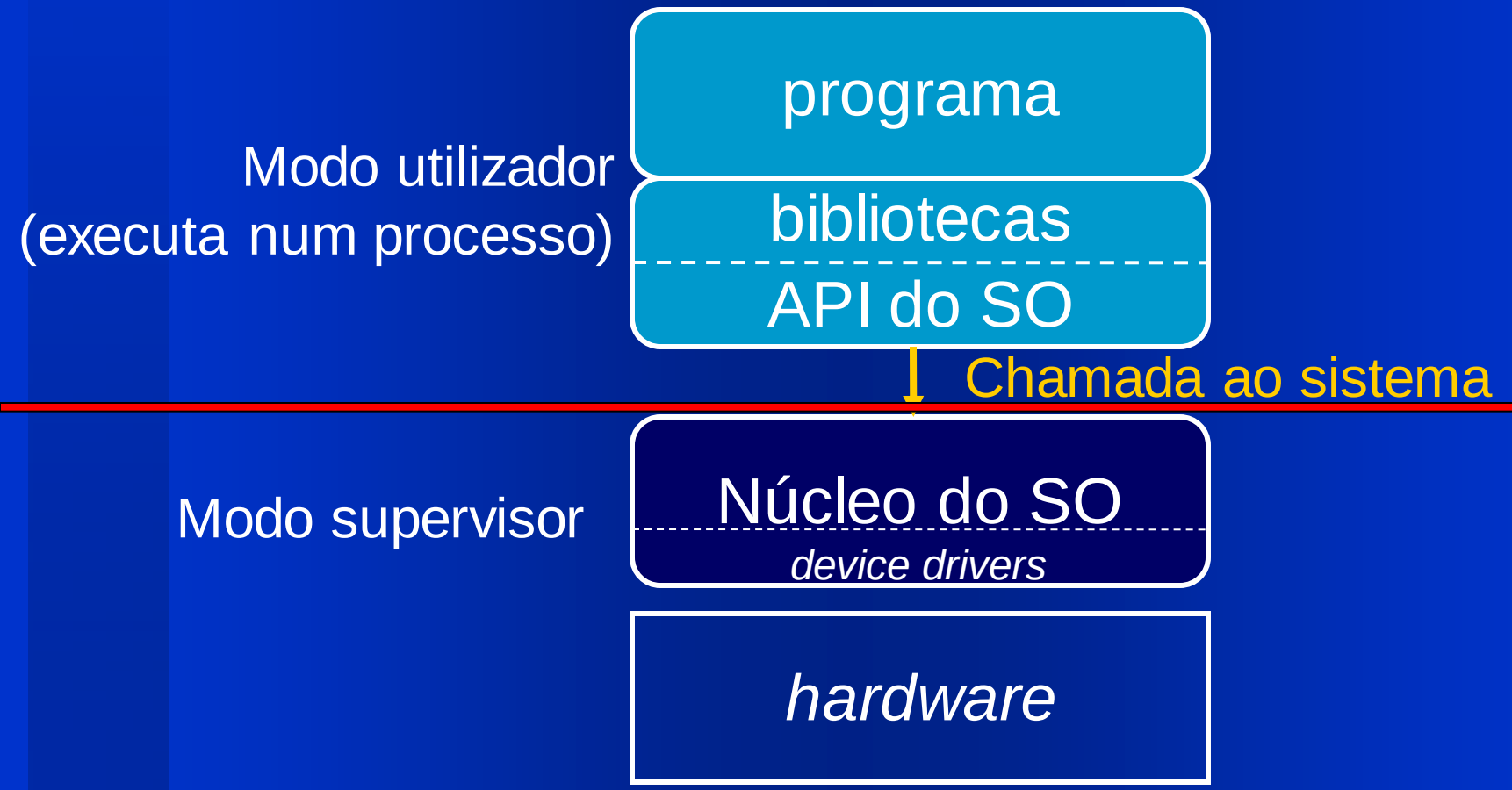
Chamadas a biblioteca e chamadas ao sistema

- **As chamadas ao sistema têm funções correspondentes na biblioteca standard C**
- **Muitas funções de biblioteca não chamam o sistema: operam em MODO UTILIZADOR!**
- **As chamadas ao sistema operam em MODO SUPERVISOR!**
- **Exemplo:**


programa
funcao de
biblioteca C
processo
utilizador
chamadas ao SO
nucleo do SO

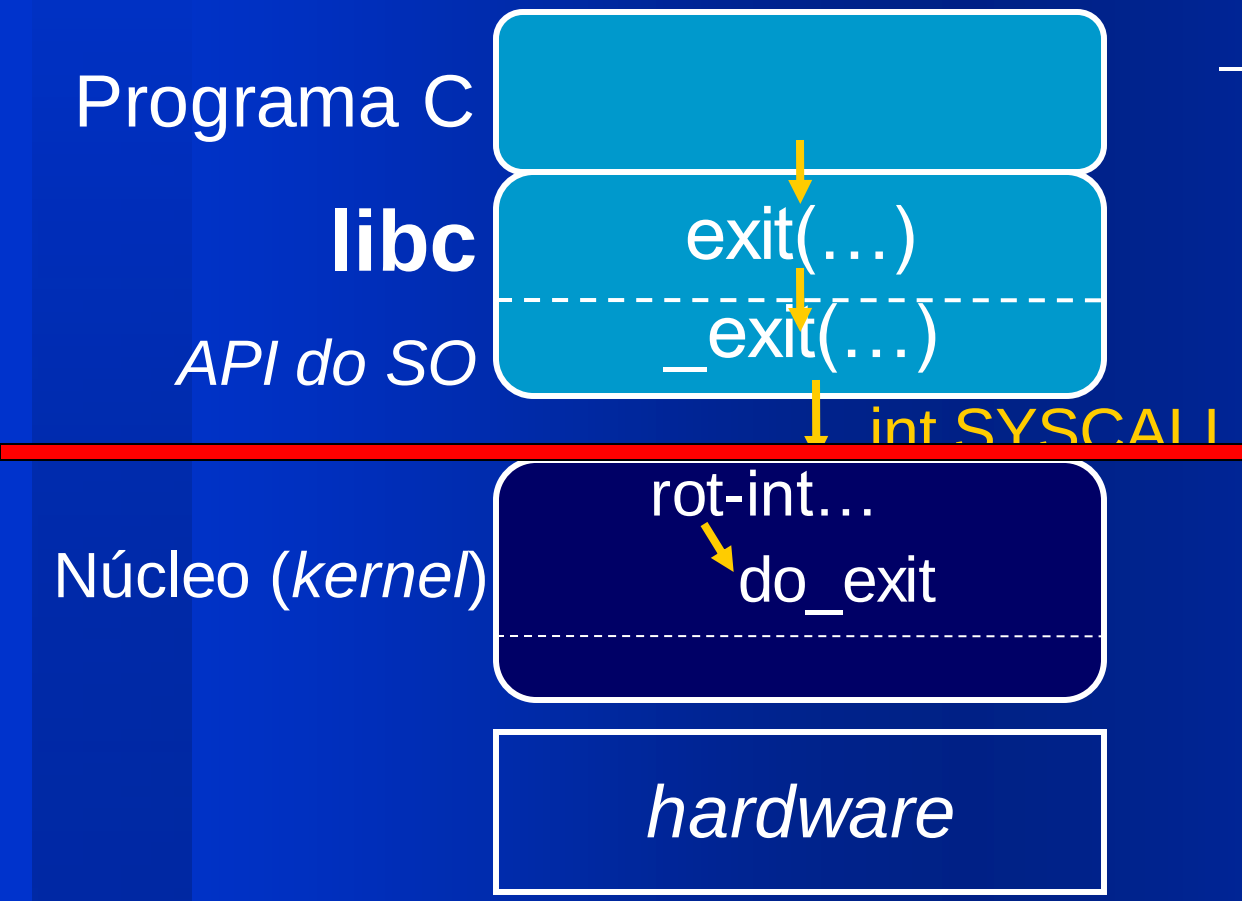
Níveis de software

- **Visão de uma aplicação**



Chamada ao sistema `_exit`

```
void _exit(int status);
```



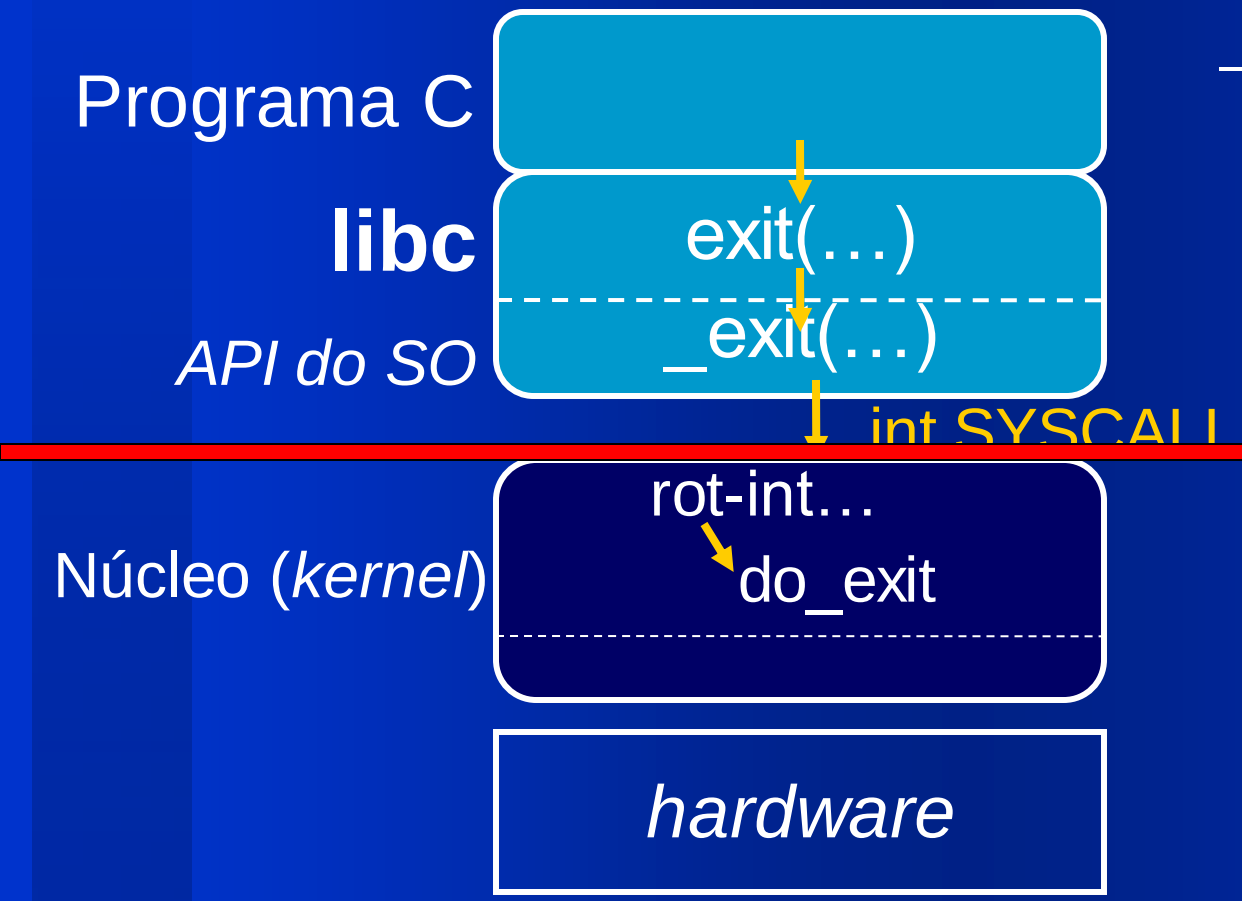
Código possível

```
_exit: …
  mov eax,EXIT_SCALL
  …
```

No caso do linux/x86:
EXIT_SCALL=1
SYSCALL=0x80

Chamada ao sistema `_exit`

```
void _exit(int status);
```



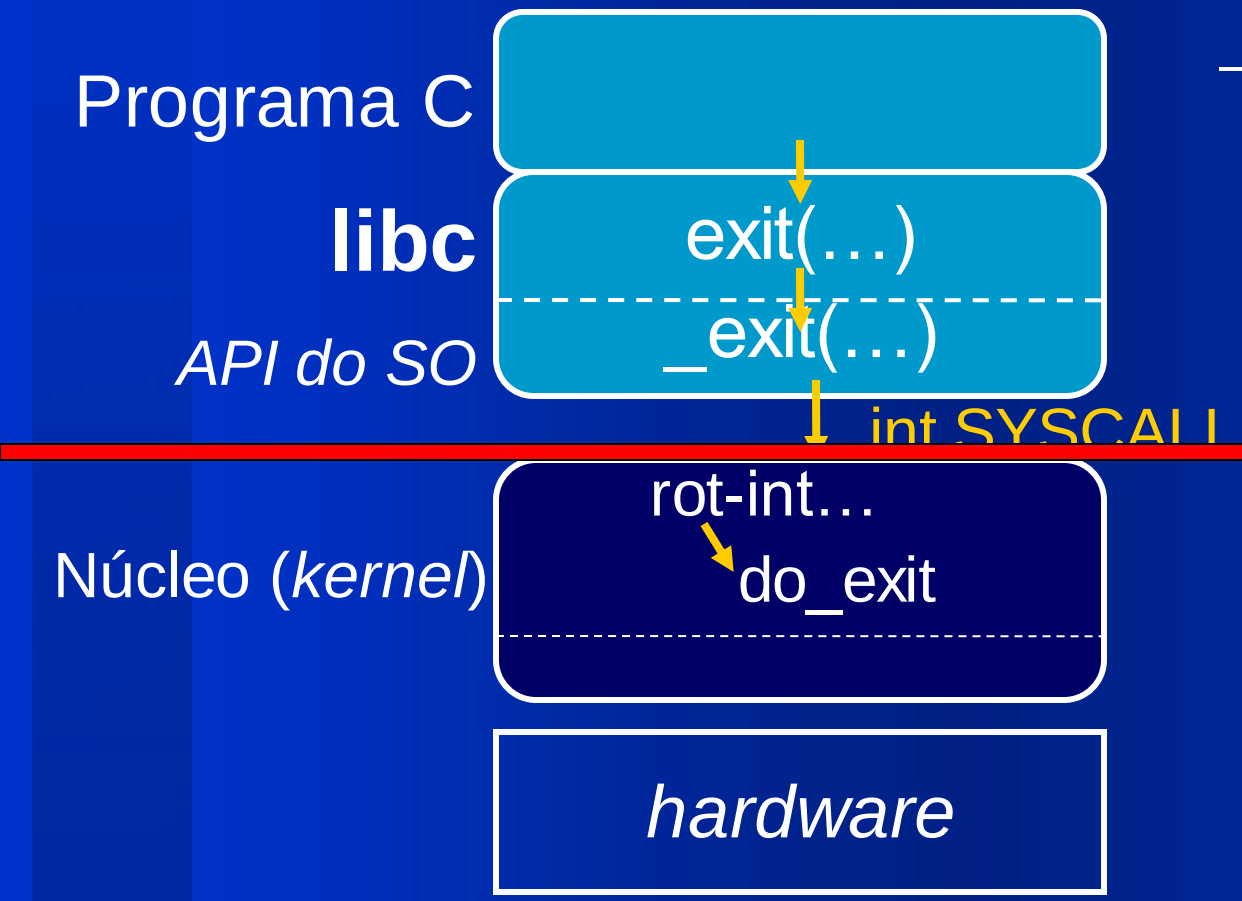
Código possível

```
_exit: …
  mov eax,EXIT_SCALL
   mov ebx, [ebp+4]

  …
```

No caso do linux/x86:
EXIT_SCALL=1
SYSCALL=0x80

Chamada ao sistema `_exit`

```
void _exit(int status);
```



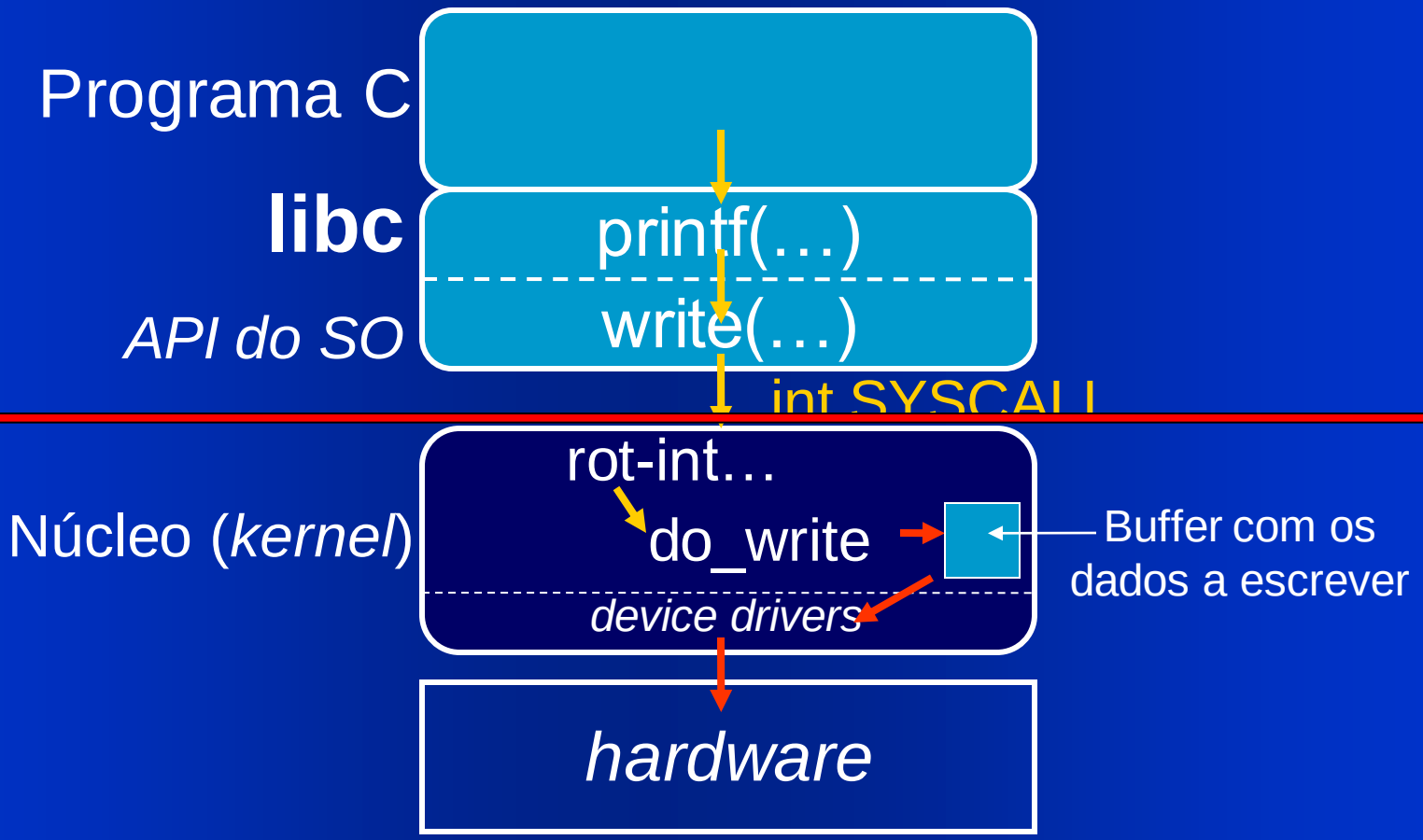
Código possível

```
_exit: …
  mov eax,EXIT_SCALL
   mov ebx, [ebp+4]
   int SYSCALL
   …
```

No caso do linux/x86:
EXIT_SCALL=1
SYSCALL=0x80

Chamadas ao sistema (I/O)

- **Exemplo de saída de dados**



Principais características do UNIX

- **Suporta múltiplos processos**

Principais características do UNIX

- **Suporta múltiplos processos**
- **Suporta múltiplos utilizadores**

Principais características do UNIX

- **Suporta múltiplos processos**
- **Suporta múltiplos utilizadores**
- **Escalonamento com *time-slice***

Principais características do UNIX

- **Suporta múltiplos processos**
- **Suporta múltiplos utilizadores**
- **Escalonamento com *time-slice***
- **Organiza o espaço nos discos num sistema de ficheiros hierárquico**

Principais características do UNIX

- **Suporta múltiplos processos**
- **Suporta múltiplos utilizadores**
- **Escalonamento com *time-slice***
- **Organiza o espaço nos discos num sistema de ficheiros hierárquico**
- **Possui mecanismos de permissões associados a processos e a ficheiros**

Classes de chamadas ao SO

- **Acesso a ficheiros e directorias**

Classes de chamadas ao SO

- **Acesso a ficheiros e directorias**
- **Pedir e libertar memória**

Classes de chamadas ao SO

- **Acesso a ficheiros e directorias**
- **Pedir e libertar memória**
- **Pedir a execução de novos programas**

Classes de chamadas ao SO

- **Acesso a ficheiros e directorias**
- **Pedir e libertar memória**
- **Pedir a execução de novos programas**
- **Controlar a execução de programas**

Classes de chamadas ao SO

- Acesso a ficheiros e directorias
- Pedir e libertar memória
- Pedir a execução de novos programas
- Controlar a execução de programas
- Comunicação e sincronização de processos concorrentes

Classes de chamadas ao SO

- Acesso a ficheiros e directorias
- Pedir e libertar memória
- Pedir a execução de novos programas
- Controlar a execução de programas
- Comunicação e sincronização de processos concorrentes
- Obter informação sobre o estado do sistema

O arranque do SO

- **Um pequeno programa para *bootstrap***
  - **Responsável pelo carregamento para memória e execução do núcleo do SO**
  - **Depende de cada computador**

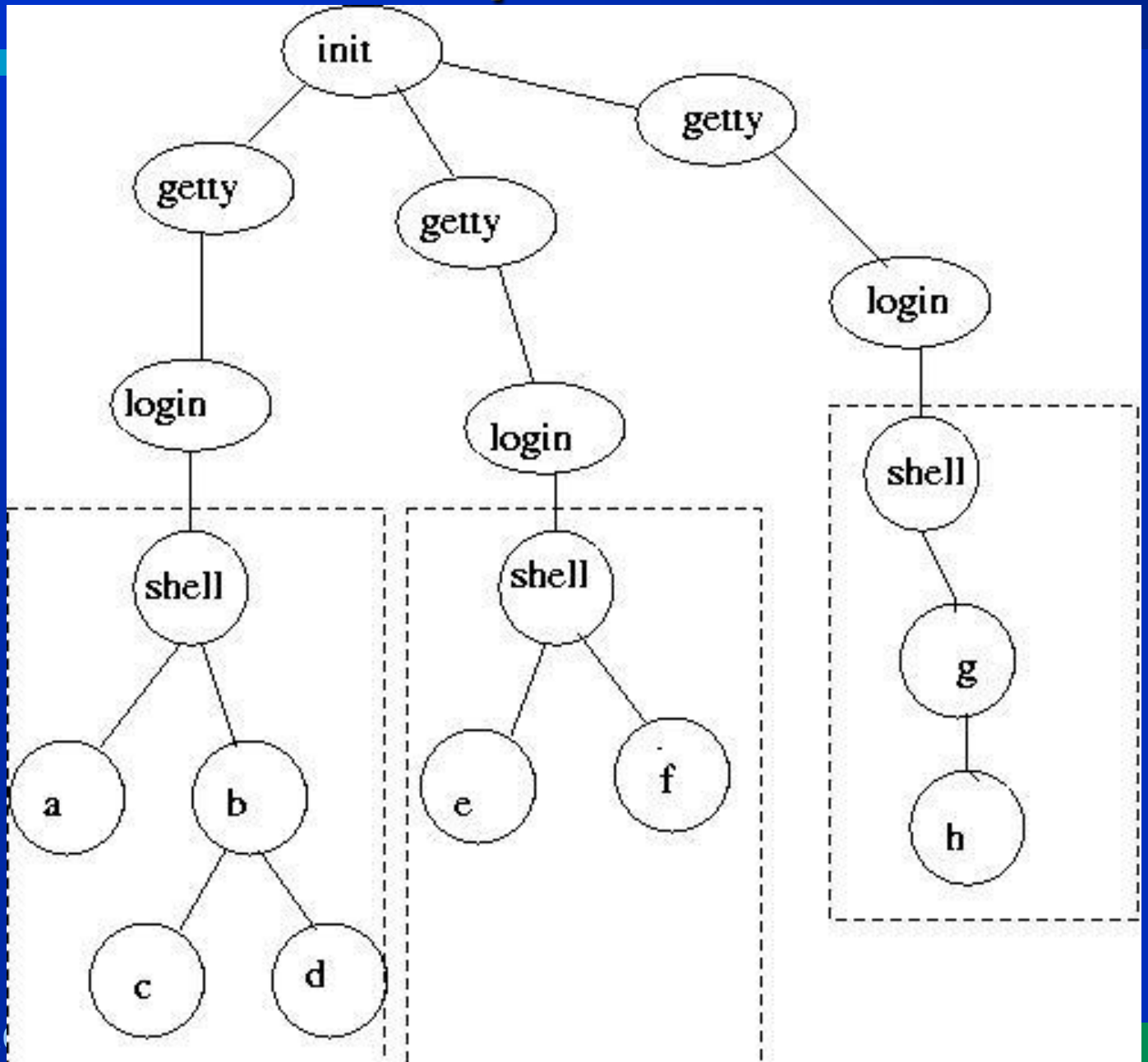
O arranque do SO

- **Um pequeno programa para *bootstrap***
  - **Responsável pelo carregamento para memória e execução do núcleo do SO**
  - **Depende de cada computador**
- **Depois das inicializações o SO cria o primeiro processo (*init*)**

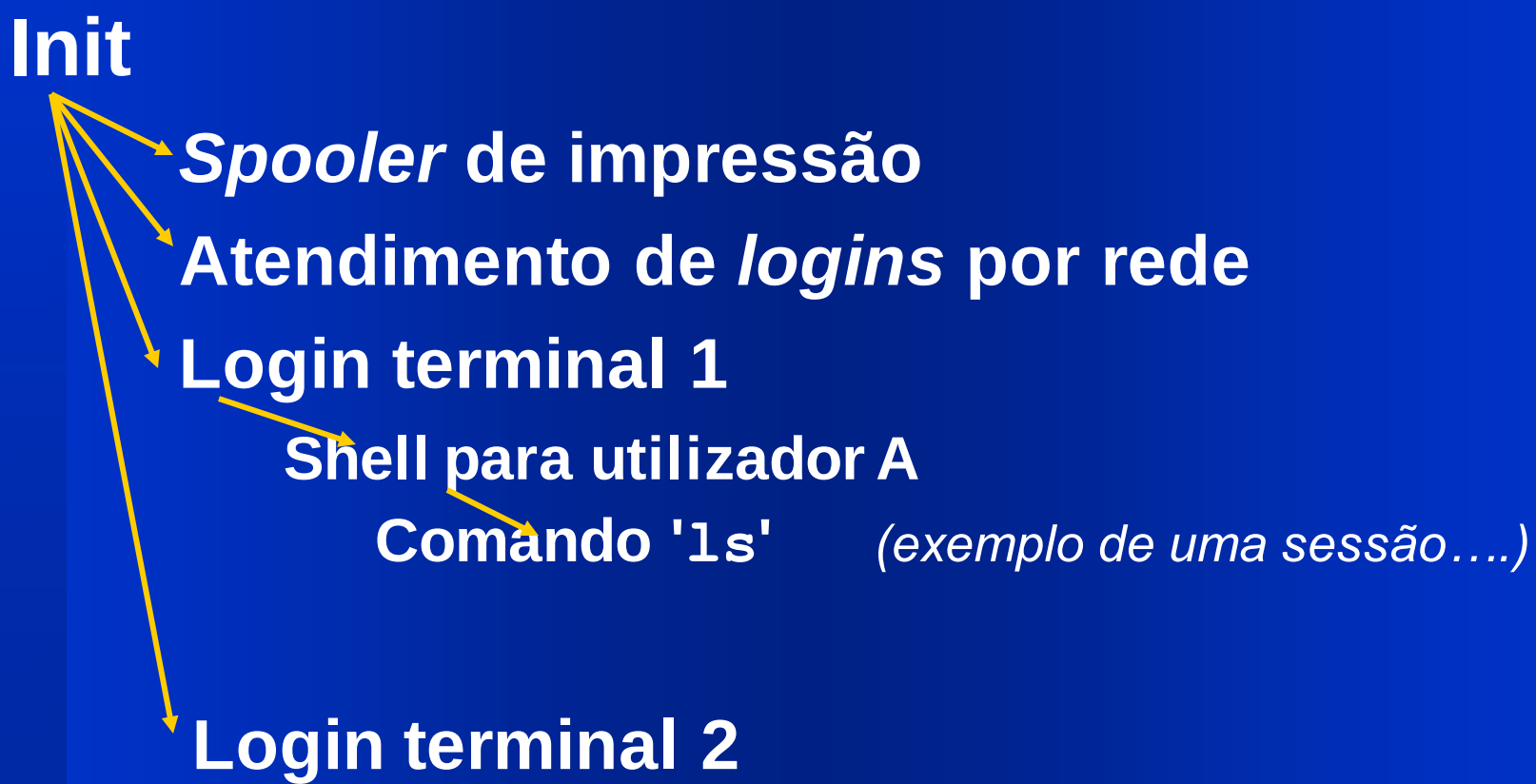
O arranque do SO

- **Um pequeno programa para *bootstrap***
  - **Responsável pelo carregamento para memória e execução do núcleo do SO**
  - **Depende de cada computador**
- **Depois das inicializações o SO cria o primeiro processo (*init*)**
- **O processo *init* dá início a uma cascata de criação de processos**

Árvore de processos no Unix



Árvore de processos (exemplo)



Utilizadores e permissões

- **Utilizadores: identificação, direitos ou permissões**
  - **O que pode ler, escrever, executar, que tempo de CPU, espaço em memória, espaço em disco, etc…**
- **Cada utilizador tem de identificar-se perante o SO: *login***
- **Cada processo recebe os direitos do utilizador que o criou**

Atributos de um processo

- **Identificador (*process identifier*)**

Atributos de um processo

- **Identificador (*process identifier*)**
- **Imagem do programa em memória**
  - **Código, dados e pilha (tabela de páginas)**

Atributos de um processo

- **Identificador (*process identifier*)**
- **Imagem do programa em memória**
  - **Código, dados e pilha (tabela de páginas)**
- **Estado do CPU (quando não executa)**
  - **Conteúdo dos registos (PC, SP, …)**

Atributos de um processo

- **Identificador (*process identifier*)**
- **Imagem do programa em memória**
  - **Código, dados e pilha (tabela de páginas)**
- **Estado do CPU (quando não executa)**
  - **Conteúdo dos registos (PC, SP, …)**
- **Estado do I/O**
  - **Canais de I/O (ficheiros abertos e *buffers* com os dados em trânsito)**

Atributos de um processo

- **Identificador (*process identifier*)**
- **Imagem do programa em memória**
  - **Código, dados e pilha (tabela de páginas)**
- **Estado do CPU (quando não executa)**
  - **Conteúdo dos registos (PC, SP, …)**
- **Estado do I/O**
  - **Canais de I/O (ficheiros abertos e *buffers* com os dados em trânsito)**
- ***Identificador do utilizador e seus direitos (ou permissões)***

Mais atributos

- **Na realidade o processo tem um dono:**
  - **identificador do utilizador (uid) e grupo (gid)**
  - **as permissões estão associadas aos recursos**

Mais atributos

- **Na realidade o processo tem um dono:**
  - **identificador do utilizador (uid) e grupo (gid)**
  - **as permissões estão associadas aos recursos**
- **Directoria corrente**

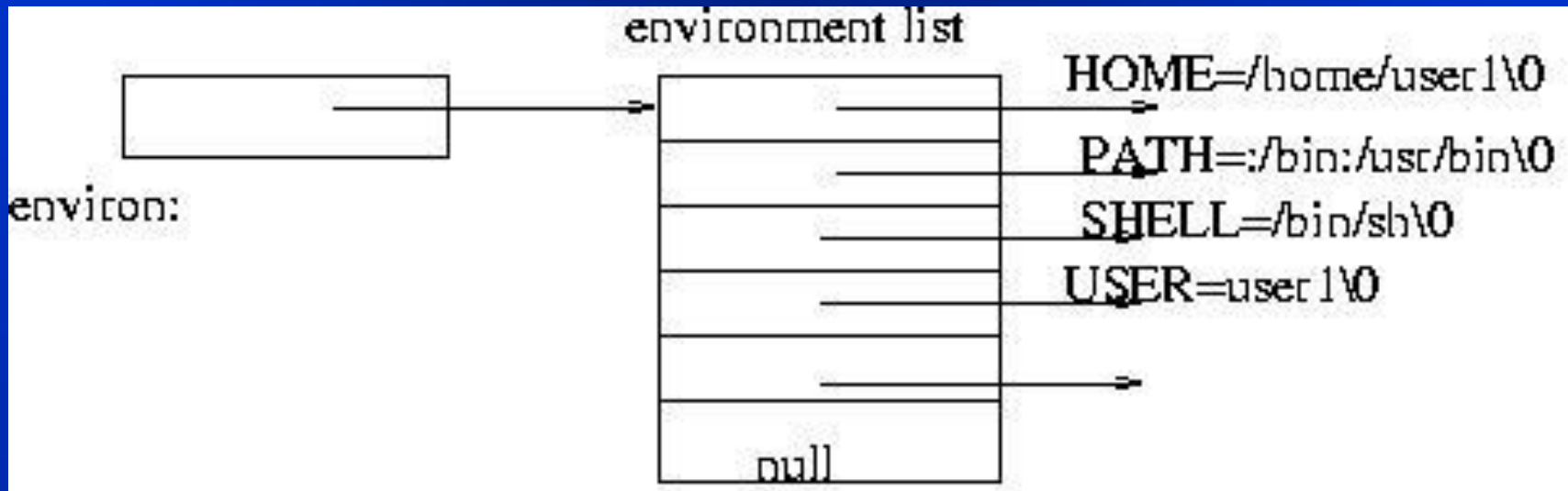
Mais atributos

- **Na realidade o processo tem um dono:**
  - **identificador do utilizador (uid) e grupo (gid)**
  - **as permissões estão associadas aos recursos**
- **Directoria corrente**
- **"Atributos" extra: variáveis de ambiente**
  - **São um conjunto de *strings* `nome=valor`**

Variáveis do ambiente

- **Definem um ambiente "livre" (pode conter qualquer coisa na forma `nome=valor`)**
  - **exemplos:**
    - **HOME – *home directory* (directoria ´´base´´ do utilizador)**
    - **PATH – lista de directorias (caminhos) onde procurar ficheiros executáveis**
    - **LANG – língua preferida pelo utilizador**
- **São definidas:**
  - **automaticamente no login (HOME, USER, …)**
  - **ou definidas num ficheiro de inicializações**
  - **ou explicitamente pelo utilizador!**

main(argc, argv, *environ*);



obter valores das variaveis: char *getenv(char *name);
devolve apontador para valor associado a name

mudar valores ou definir novas variaveis:
putenv/setenv/unsetenv

Exemplo de uso de processos

A nível do Shell no Unix:

-- ler e interpretar comandos do terminal

Exemplo de uso de processos

A nível do Shell no Unix:

-- ler e interpretar comandos do terminal

-- criar um processo por cada comando

Exemplo de uso de processos

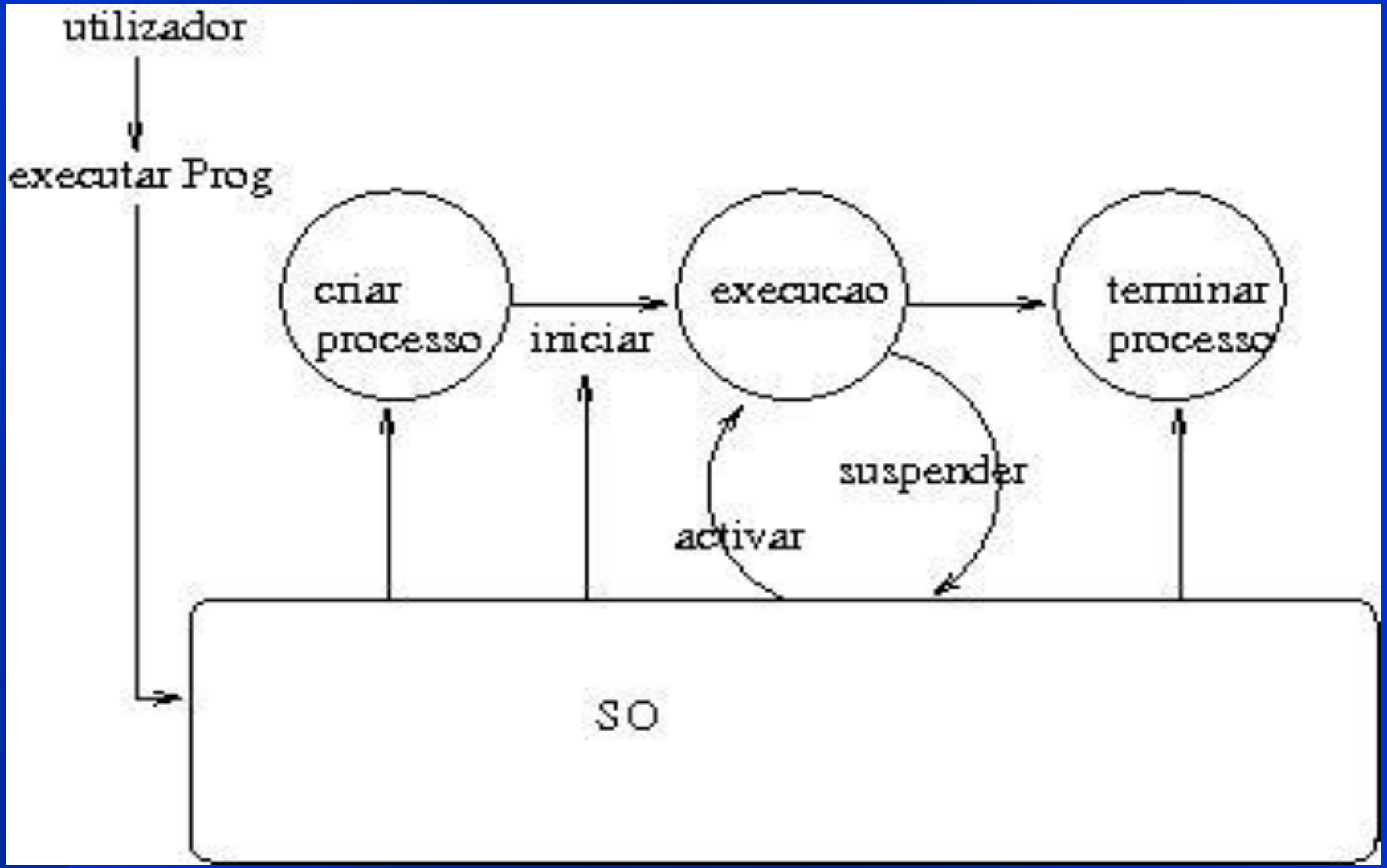
A nível do Shell no Unix:

-- ler e interpretar comandos do terminal

-- criar um processo por cada comando

-- destruir o processo no fim da execução de cada comando

Ciclo de vida de um processo



Executar um programa → Processo

- **Pedir a criação de um processo:**
  - **o SO verifica todas as permissões e…**

Executar um programa → Processo

- **Pedir a criação de um processo:**
  - **o SO verifica todas as permissões e…**
  - **cria as estruturas para gerir o processo**

Executar um programa → Processo

- **Pedir a criação de um processo:**
  - **o SO verifica todas as permissões e…**
  - **cria as estruturas para gerir o processo**
  - **inicia todos os seus atributos**
  - **inicia o mapa de memória com o programa (p. ex. a partir de um ficheiro executável)**

Executar um programa → Processo

- Pedir a criação de um processo:
  - o SO verifica todas as permissões e…
  - cria as estruturas para gerir o processo
  - inicia todos os seus atributos
  - inicia o mapa de memória com o programa (p. ex. a partir de um ficheiro executável)
  - inicia o CPU, transferindo o controlo para o processo criado (PC, SP,…)

Executar um programa → Processo

- **Pedir a criação de um processo:**
  - **o SO verifica todas as permissões e…**
  - **cria as estruturas para gerir o processo**
  - **inicia todos os seus atributos**
  - **inicia o mapa de memória com o programa (p. ex. a partir de um ficheiro executável)**
  - **inicia o CPU, transferindo o controlo para o processo criado (PC, SP,…)**
- **Queremos algo como:**

CriarProcesso(ficheiro,canais-iniciais, atributos, ambiente,…etc) ??

Criar processos

- Exemplo MS-Windows (Win32):

```
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 )
```

Criar processos

- **Exemplo MS-Windows (Win32):**

```
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 )
```

- **Abordagem UNIX:**

```
pid_t fork(void)
int execve(char *path, char *argv[],char *envp[])
```

Chamadas Unix para Processos

- ***Fork*: cria um duplicado exacto do processo original incluindo informação sobre canais abertos**

Chamadas Unix para Processos

- ***Fork*: cria um duplicado exacto do processo original incluindo informação sobre canais abertos**
- ***Exec*: nova imagem executável para o processo, a partir de um ficheiro executável -- um novo mapa de memória**

Chamadas Unix para Processos

- ***Fork*: cria um duplicado exacto do processo original incluindo informação sobre canais abertos**
- ***Exec*: nova imagem executável para o processo, a partir de um ficheiro executável -- um novo mapa de memória**
- ***Wait*: permite a um processo aguardar pela terminação de um filho**

Chamadas Unix para Processos

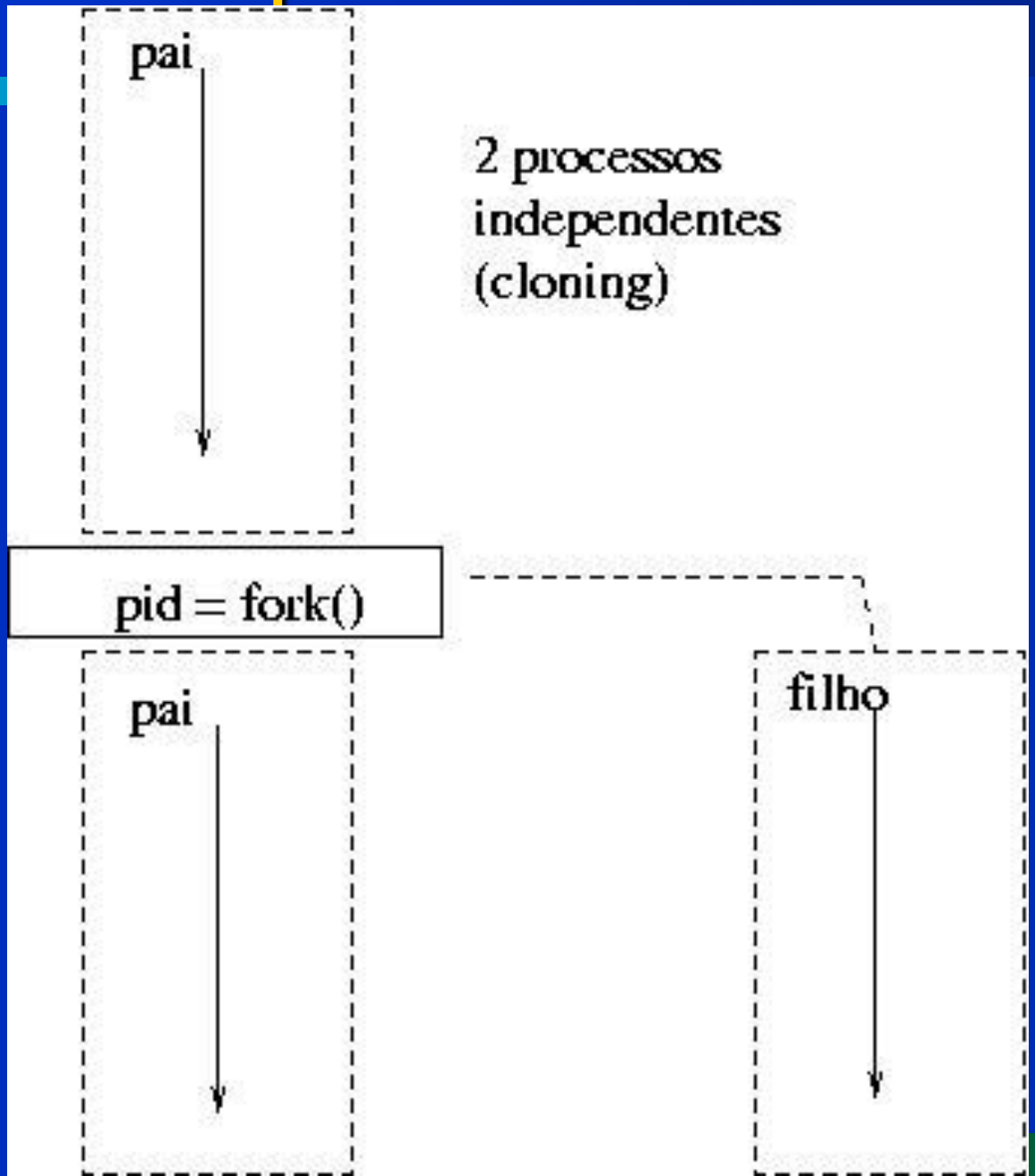
- ***Fork*: cria um duplicado exacto do processo original incluindo informação sobre canais abertos**
- ***Exec*: nova imagem executável para o processo, a partir de um ficheiro executável -- um novo mapa de memória**
- ***Wait*: permite a um processo aguardar pela terminação de um filho**
- ***Exit /_exit*: termina o processo corrente**

Chamadas Unix para Processos

- ***Fork*: cria um duplicado exacto do processo original incluindo informação sobre canais abertos**
- ***Exec*: nova imagem executável para o processo, a partir de um ficheiro executável -- um novo mapa de memória**
- ***Wait*: permite a um processo aguardar pela terminação de um filho**
- ***Exit /_exit*: termina o processo corrente**
- ***Getpid*: indica o identificador do processo**

FORK: criar um processo no Unix

Criação de processos – ´ ´ cloning ´ ´



Como é que o ´´clone´´ sabe que é um ´´clone´´?

- Como é que o ´´clone´´ sabe que é um ´´clone´´?
- Como é que o ´´pai´´ sabe que é o processo original e não a cópia?

Como é que o ´´clone´´ sabe que é um ´´clone´´?

Como é que o ´´pai´´ sabe que é o processo original e não a cópia?

Será necessário distinguir entre eles?

- em geral o pai é um supervisor ou gestor que coordena a criação de múltiplos processos

Um novo processo: criar um ´´clone´´

```
pid_t fork(void)
```

Um novo processo: criar um ´´clone´´

```
pid_t fork(void)
```

- **cria um novo processo (filho) cópia exacta do original (pai)**

  incluindo programa, atributos e informação sobre canais abertos (o *pid* é diferente, claro!)

Um novo processo: criar um ´´clone´´

```
pid_t fork(void)
```

- **cria um novo processo (filho) cópia exacta do original (pai)**

  incluindo programa, atributos e informação sobre canais abertos (o *pid* é diferente, claro!)
- **devolve o *pid* do filho ao pai (no retorno de fork)**

Um novo processo: criar um ´´clone´´

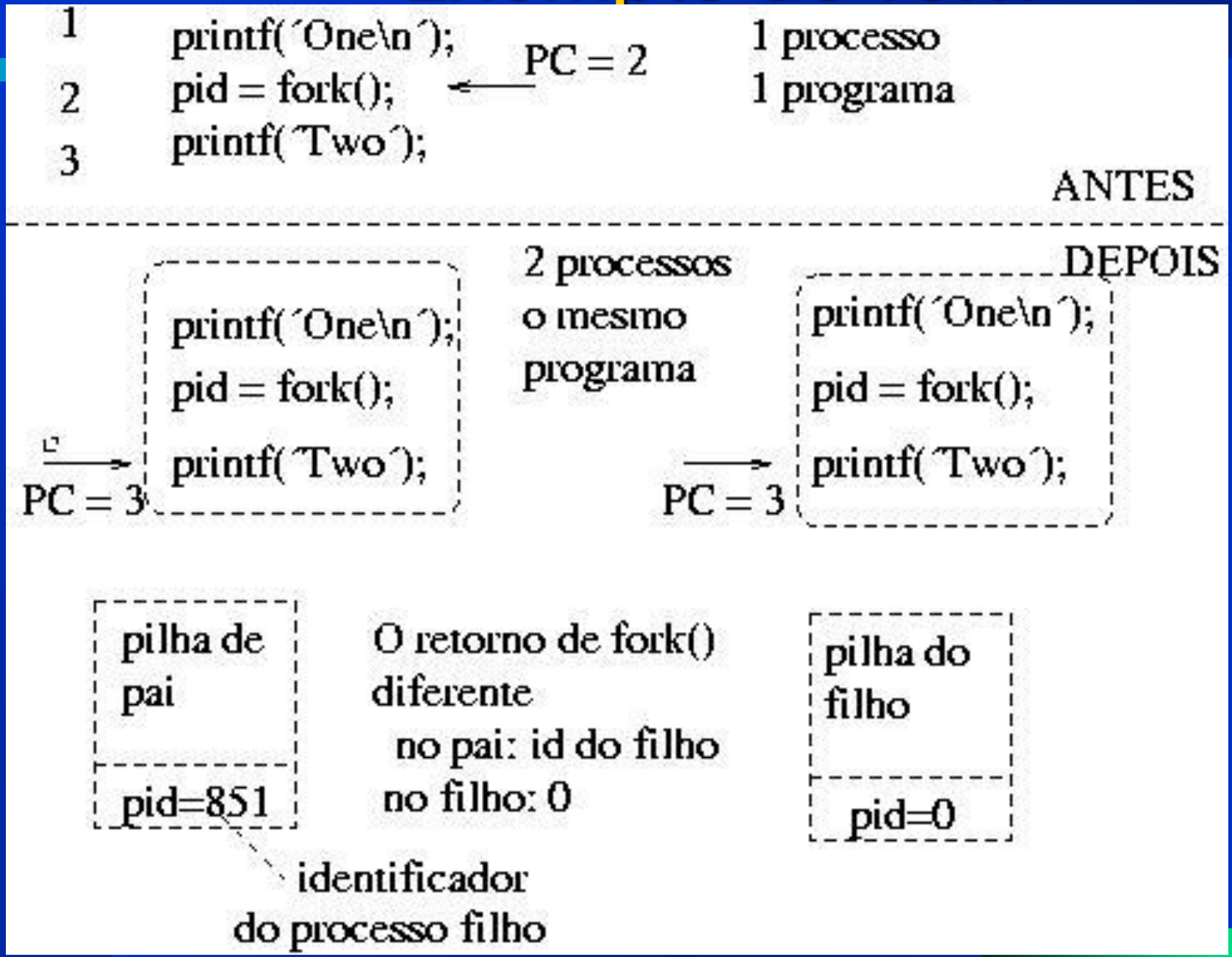
`pid_t fork(void)`

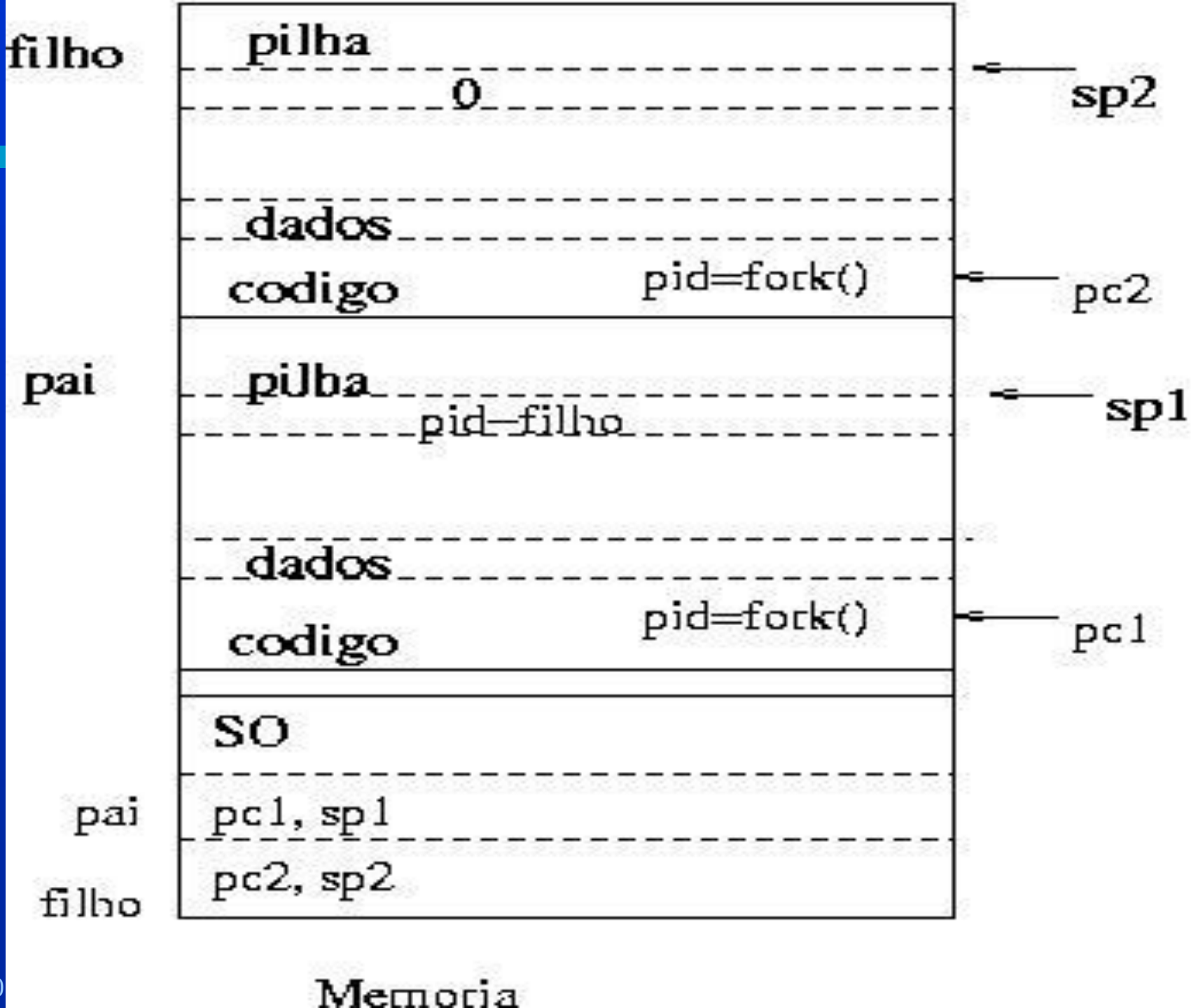
- cria um novo processo (filho) cópia exacta do original (pai)

  incluindo programa, atributos e informação sobre canais abertos (o *pid* é diferente, claro!)
- devolve o *pid* do filho ao pai (no retorno de fork)
- o filho, sendo uma cópia do pai, começa a executar na instrução apontada pelo mesmo valor do PC do pai: instrução logo a seguir ao `fork`!

  o SO devolve um resultado no retorno de `fork`, no novo processo filho (devolve zero)

Exemplo de fork




filho
pilha
0
sp2
dados
codigo
pid=fork()
pc2
pai
pilha
pid=filho
sp1
dados
codigo
pid=fork()
pc1
SO
pai
pc1, sp1
filho
pc2, sp2
Memoria

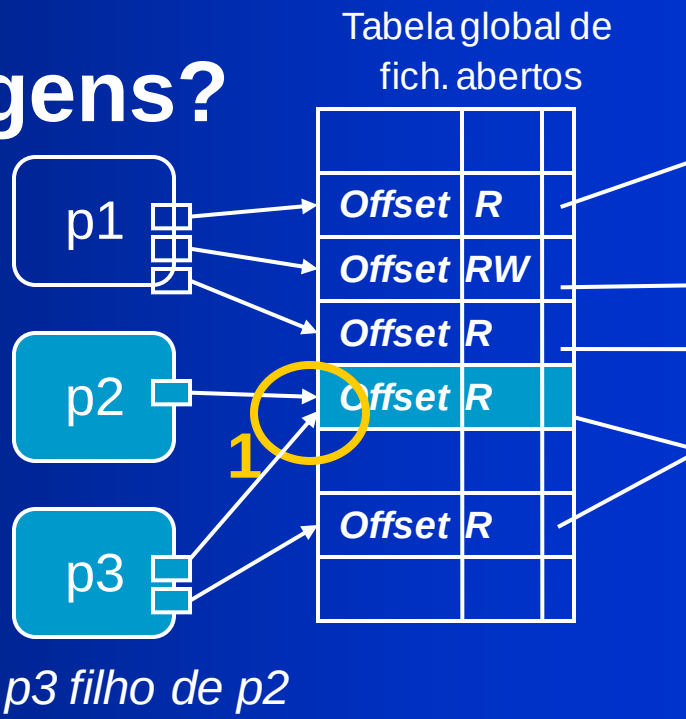
```
pid = fork ();    /* pid: resultado do fork*/
if (pid == 0)     /*pid NÃO é o identificador do filho!*/
        -- código executado pelo filho

else if (pid > 0) /*pid = identifica o filho*/
        -- código executado pelo pai

else  /* pid < 0 */
        -- erro
```

Exemplo:

```
pid = fork();
switch (pid)
{
   case 0:
        printf("Filho: %d\n",getpid());
        exit(0);
   case -1: perror("fork");
            break;
   default:printf("Criei %d\n", pid);
}
```

Exemplo (cont.)

- **Onde aparecem as mensagens?**
  - **Filho "herda" os canais de i/o!**
    - **descritores copiados, logo apontam os mesmos canais**



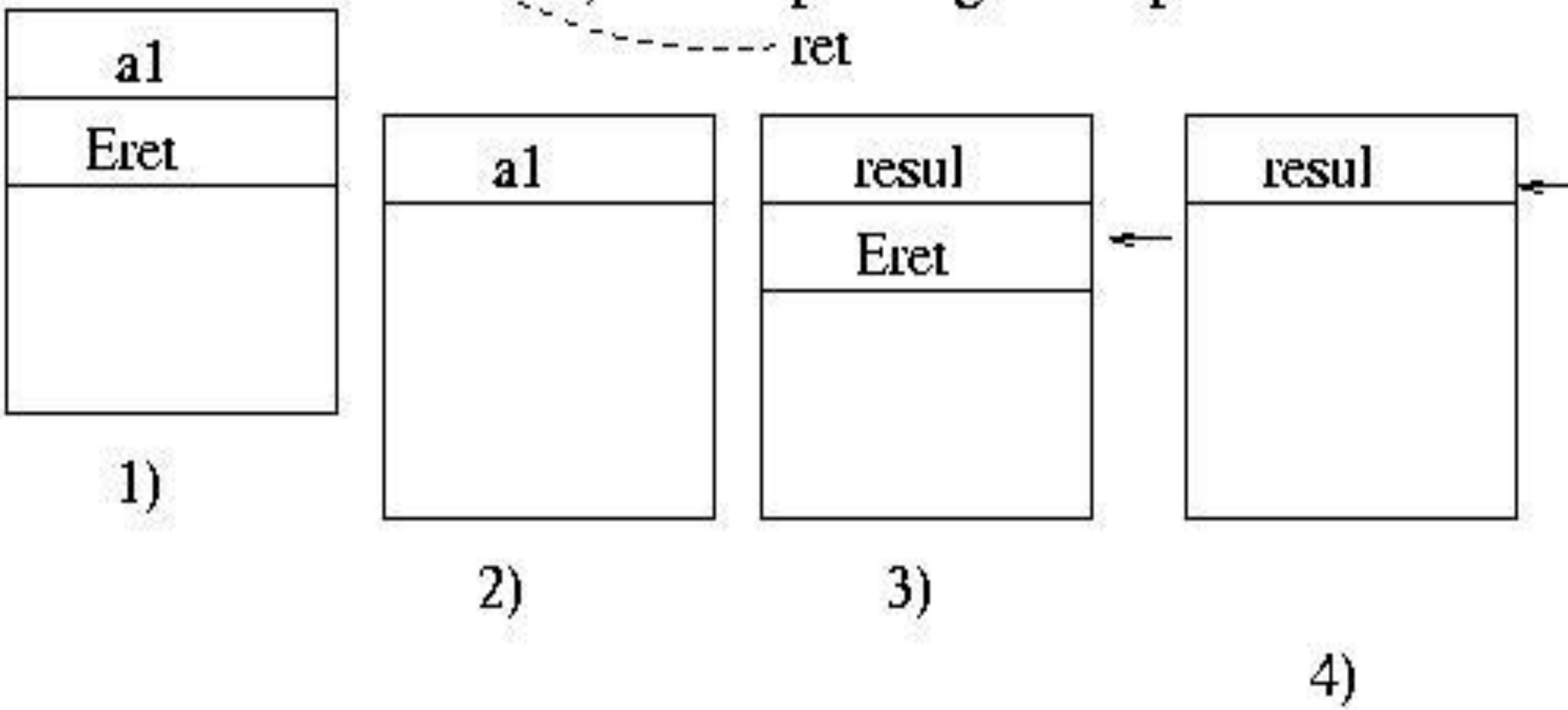
p3 filho de p2

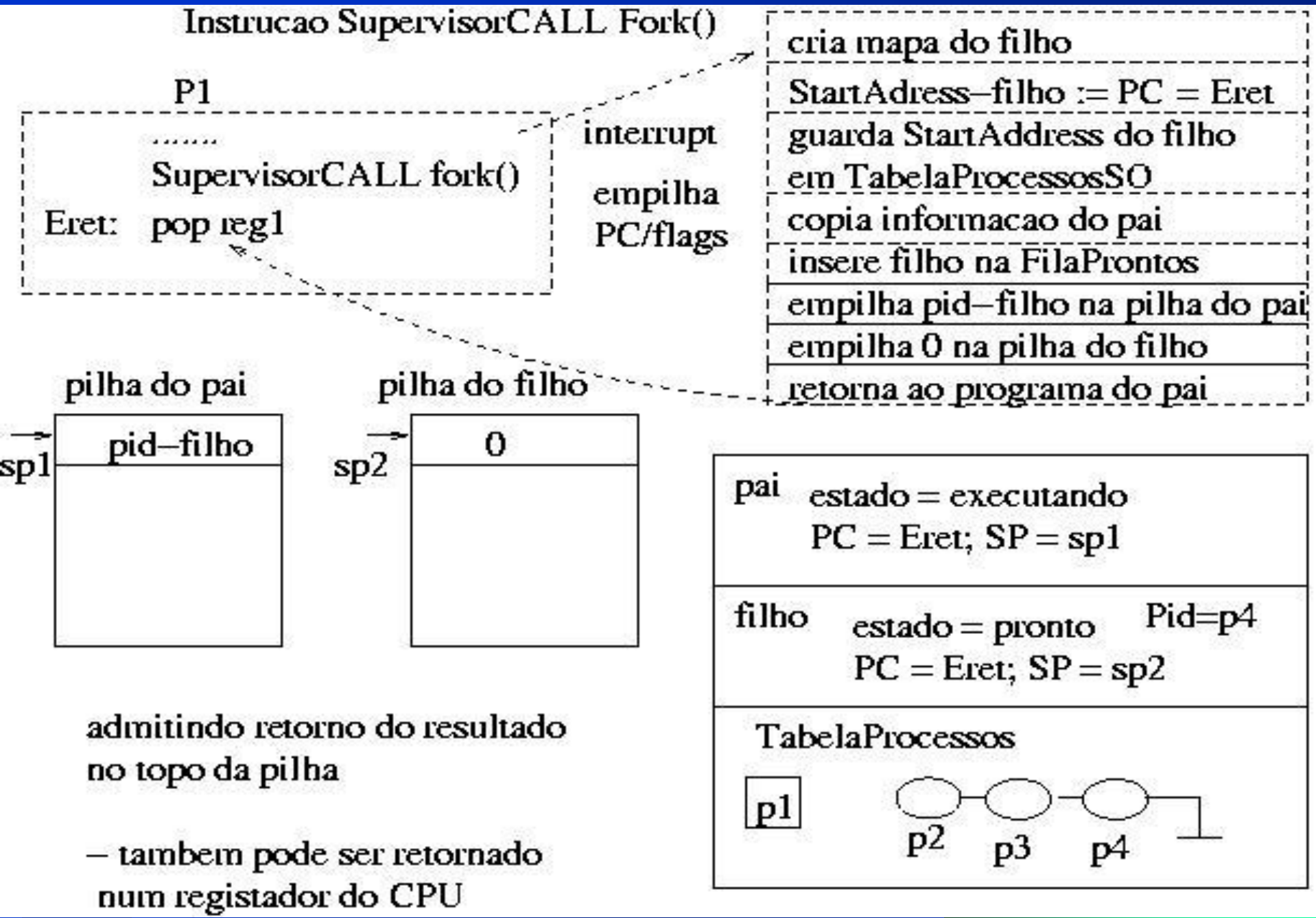
- **Por que ordem? Quem executa primeiro?**
  - **São dois processos concorrentes, o SO decide como os escalonar**
    - **podem executar em simultâneo, se existirem vários CPU!**

Fork: com mais pormenor

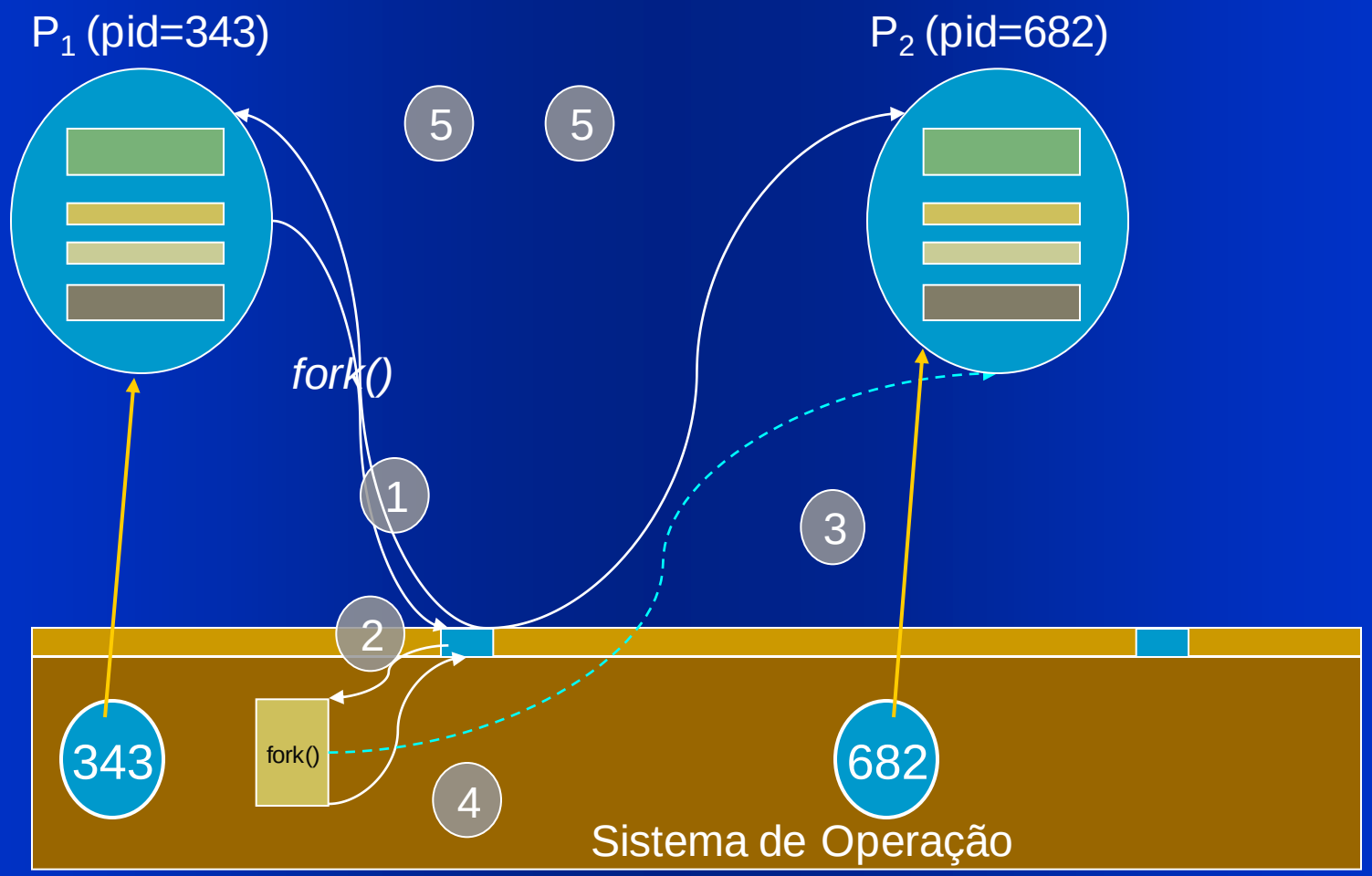
Revisao da instrucao CALL Subrotina

```
        push a1                1)→ End:  pop reg1  -- guarda Eret
        call End              2)        pop reg2  -- acede arg
4)→ Eret: pop reg1                       ......
                                         push resul
                               3)        push reg1 — repoe Eret
                                         ret
```




Instrucao SupervisorCALL Fork()
P1
.....
SupervisorCALL fork()
Eret: pop reg1
interrupt
empilha
PC/flags
cria mapa do filho
StartAdress–filho := PC = Eret
guarda StartAddress do filho
em TabelaProcessosSO
copia informacao do pai
insere filho na FilaProntos
empilha pid–filho na pilha do pai
empilha 0 na pilha do filho
retorna ao programa do pai
pilha do pai
pilha do filho
sp1
pid–filho
sp2
0
pai
estado = executando
PC = Eret; SP = sp1
filho
estado = pronto
Pid=p4
PC = Eret; SP = sp2
TabelaProcessos
p1
p2
p3
p4
admitindo retorno do resultado
no topo da pilha
– tambem pode ser retornado
num registador do CPU

sequência de passos no fork()



Utilização do fork

- **Criar várias instâncias do mesmo programa**
  - **p. ex: lançar o *login* em vários terminais**

Utilização do fork

- **Criar várias instâncias do mesmo programa**
  - **p. ex: lançar o *login* em vários terminais**
- **Tirar partido da multiprogramação**
  - **p. ex: um programa inclui diversas acções que podem ser executadas em concorrência**

Utilização do fork

- **Criar várias instâncias do mesmo programa**
  - **p. ex: lançar o *login* em vários terminais**
- **Tirar partido da multiprogramação**
  - **p. ex: um programa inclui diversas acções que podem ser executadas em concorrência**
- **Combinar fork + exec para criar processos com novos programas**
  - **(antes do exec, o novo processo pode alterar o ambiente para o novo programa)**

EXEC

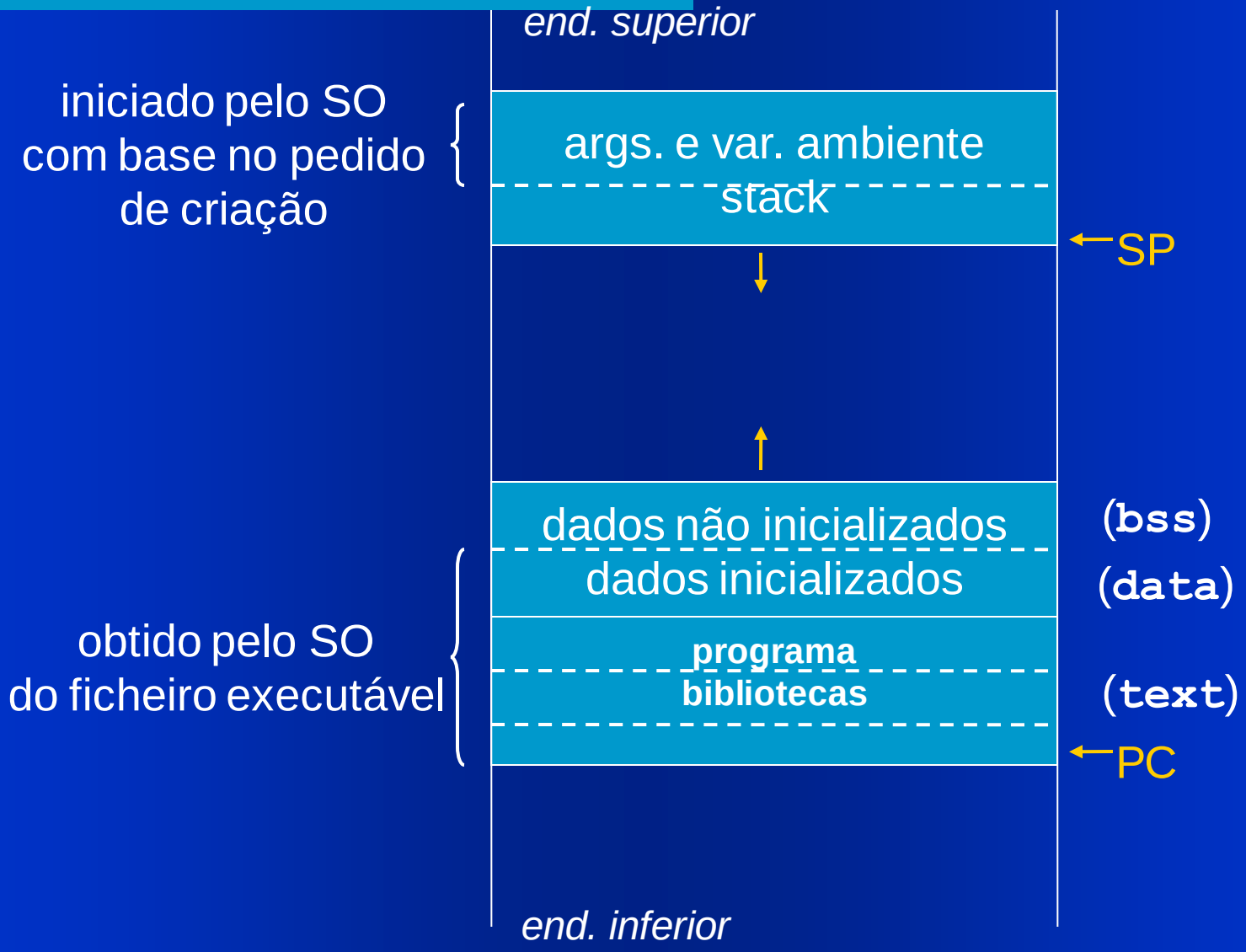
Executar um programa: exec

Num processo existente:

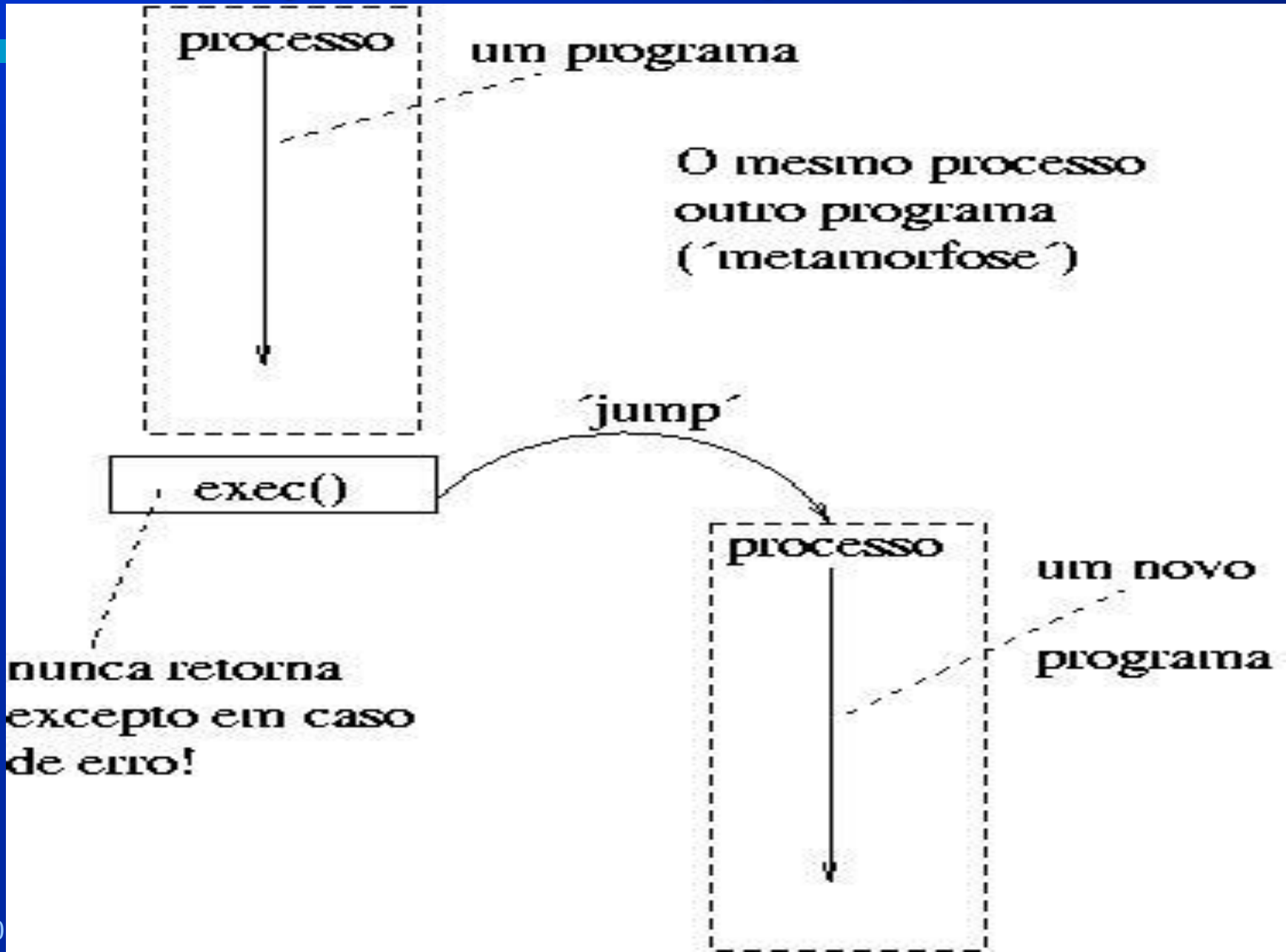
- **Cria um novo mapa de memória para o processo (substitui o antigo mapa)**
  - **na prática, passa a executar um novo programa**
- **Os restantes atributos do processo mantêm-se:**
  - **atributos e canais de I/O (há excepções)**
- **Exemplo:**

```
execve("/bin/ls", argv, environ);
```

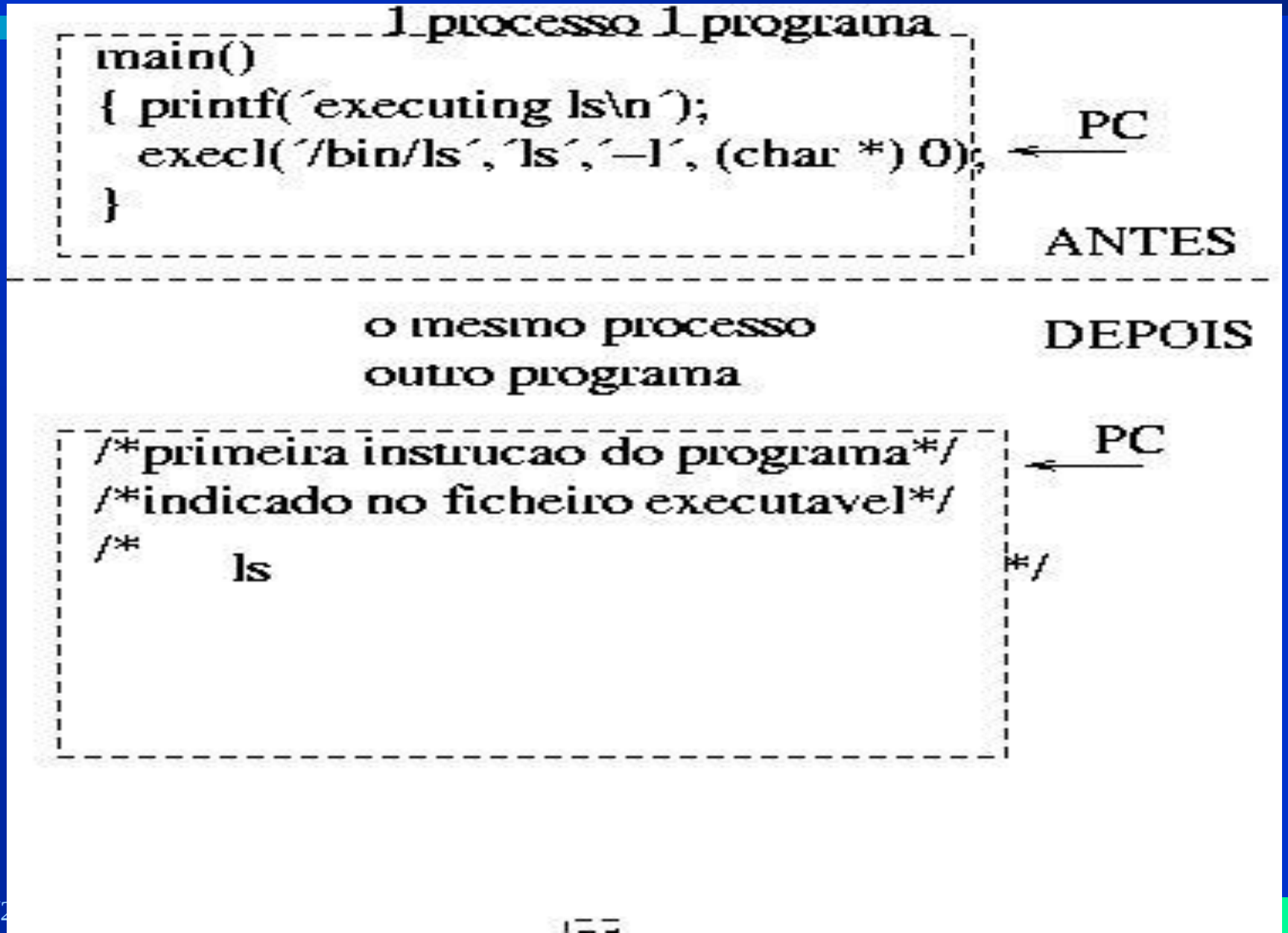
Mapa de memória virtual do processo



Execução de novo programa



Exemplo de exec



Chamadas de fork e exec

int fork(void); -- uma só função

exec: – uma família de funções que recorrem a uma única chamada ao SO execve()

- Só diferem no modo como especificam os argumentos e as variáveis de ambiente:
- execl: lista de argumentos
- execv: vector de apontadores p/ argumentos
- execp: variável PATH
- exece: vector para variáveis de ambiente

execve e Companhia

- **Funções de biblioteca:**

```
execl(char *path, char *arg0, char *arg1,…)
execle(char *path, char *arg0, char *arg1,…,NULL,
                                        char *envp[])

execv(char *path, char *argv[])

execlp(char *file, char *arg0, char *arg1,…)

execvp(char *file, char *argv[])
```

  - as duas últimas usam as directorias indicadas em `PATH` (do ambiente corrente do processo) para procurar `file`

execl: args como uma lista terminada por NULL

e um pathname para o executável

```
int execl (char  *path,
           char  *arg1, ..., char *argn,
           NULL);
```

execlp: só precisa do filename para o executável

```
int execlp (char  *file,
            char  *arg1, ..., char *argn,
            NULL);
```

execv: um array de argumentos
e um pathname para o executável

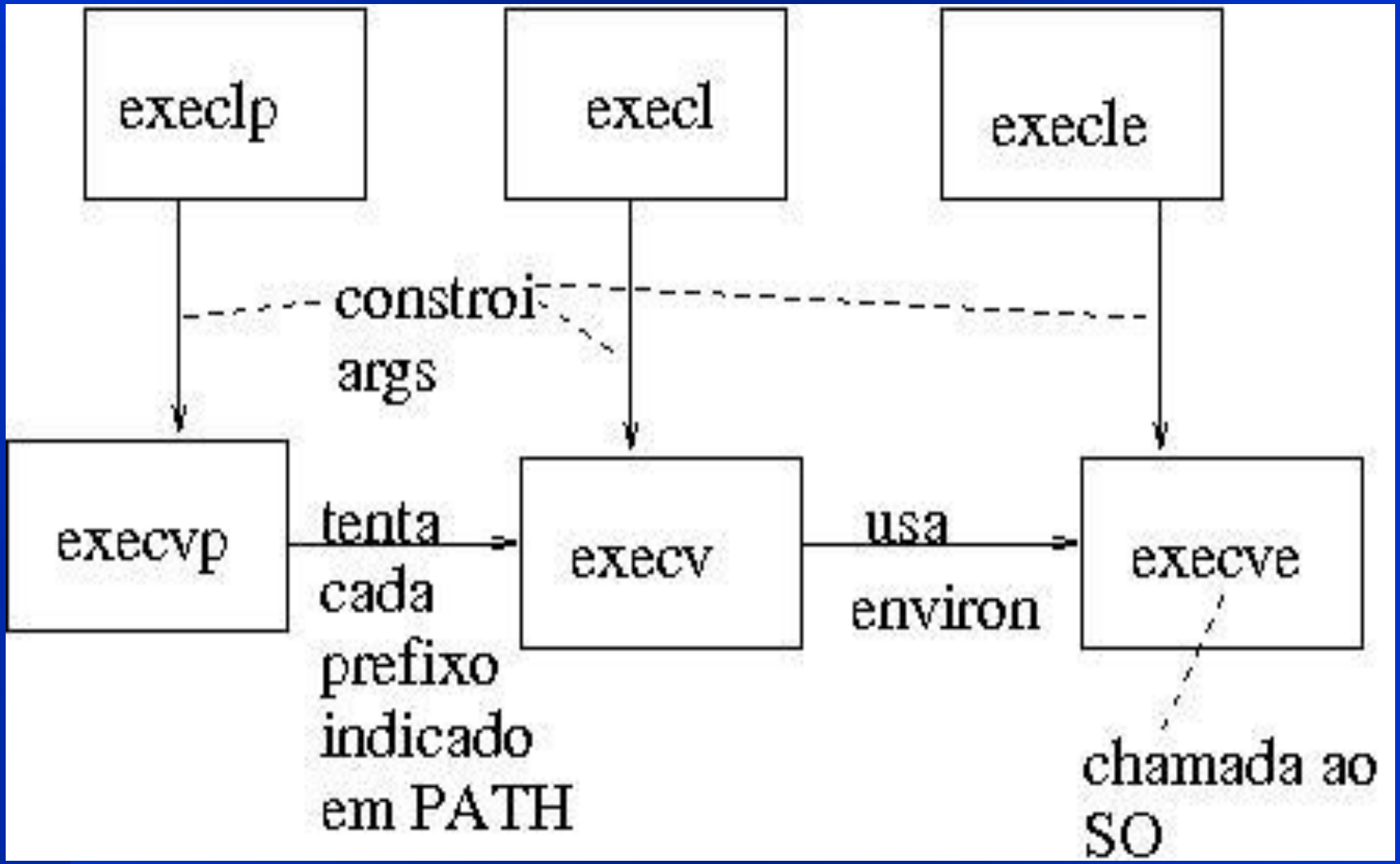
```
int execv(char  *path, char *argv[ ]);
```

execvp: só um filename para o executável

```
int execvp(char  *file, char *argv[ ]);
```

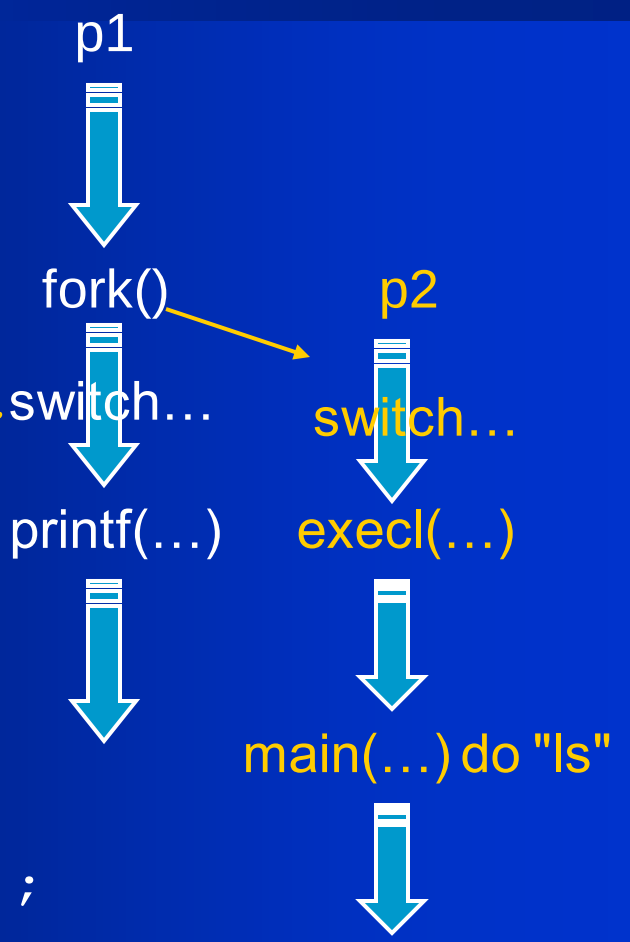
```
main()
{  printf(´´executing ls\n´´);
   execl(´´/bin/ls´´, ´´ls´´, ´´-l´´, NULL);

  /*se retorna aqui: houve um erro*/
   perror(´´execl failed to run ls´´):
   exit(1);
}
```

```
main()
 {   char *av[ ] = {´´ls´´, ´´-l´´, NULL);

ou av[0] = ´´ls´´
     av[1] = ´´-l´´
     av[2] = NULL

  /* os argumentos num vector de strings */

    execv(´´/bin/ls´´,  av);

   /*se retorna aqui: houve um erro*/
    perror(´´execv failed to run ls´´):
    exit(1);
 }
```

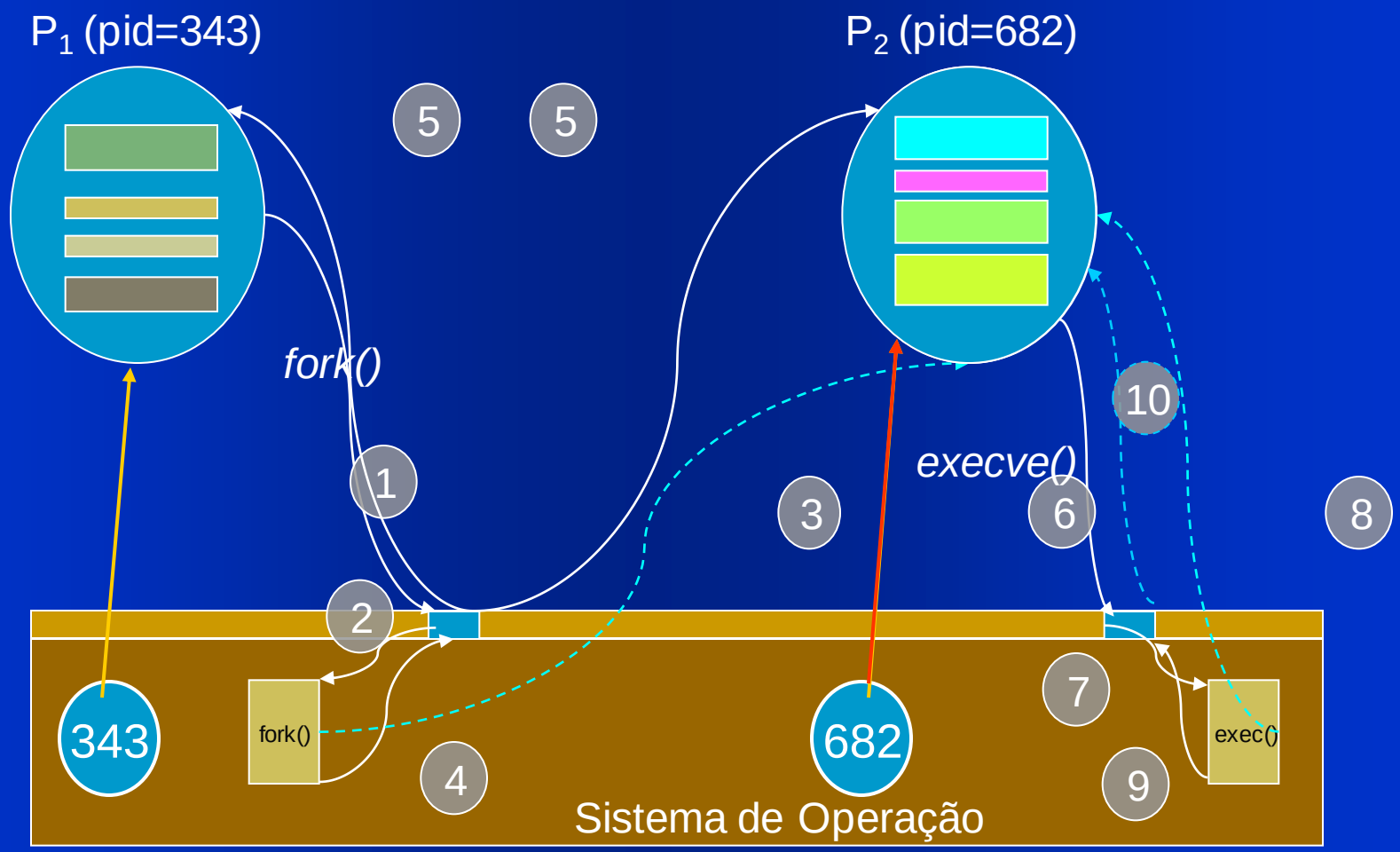

execlp
execl
execle
constroi
args
execvp
tenta
cada
prefixo
indicado
em PATH
execv
usa
environ
execve
chamada ao
SO

fork + exec*

```
pid = fork();
switch (pid)
{
  case 0:
       execl("/bin/ls","ls","/bin",NULL);
       perror("ls");
       exit(1);
  case -1: perror("fork");
           break;
  default:
    printf("Criei o processo %d\n", pid);
}
```



sequência fork() + execve()



FORK	EXEC
mapa do pai duplicado para o filho copia privada	mapa do invocador esmagado e criado um novo mapa para um novo programa executavel
retorna um valor de pid no pai diferente do valor retornado no filho	nunca retorna excepto em caso de erro
o filho herda uma copia dos canais	mantem os canais abertos (por omissao)
pid do pai difere do pid do filho 2 processos com igual programa	pid mantem–se pois o processo nao mudou o programa e´ que mudou

fork vs. execve

`fork`	`execve`
mapa mem. pai copiado para filho	mapa de mem. substituído pelo novo programa
retorna pid do filho ao pai (filho recebe zero)	nunca retorna (só em caso de erro)
pid filho diferente do do pai (processos diferentes!)	o pid mantém-se (mesmo processo!)
filho herda uma copia dos canais de i/O	mantém os mesmos canais (mesmo processo!)

Wait e waipid

int wait(int *status);

Aguarda a terminação de um dos filhos

int waitpid(int pid, int *status, int options);

Se pid == -1 → waitpid é igual a wait

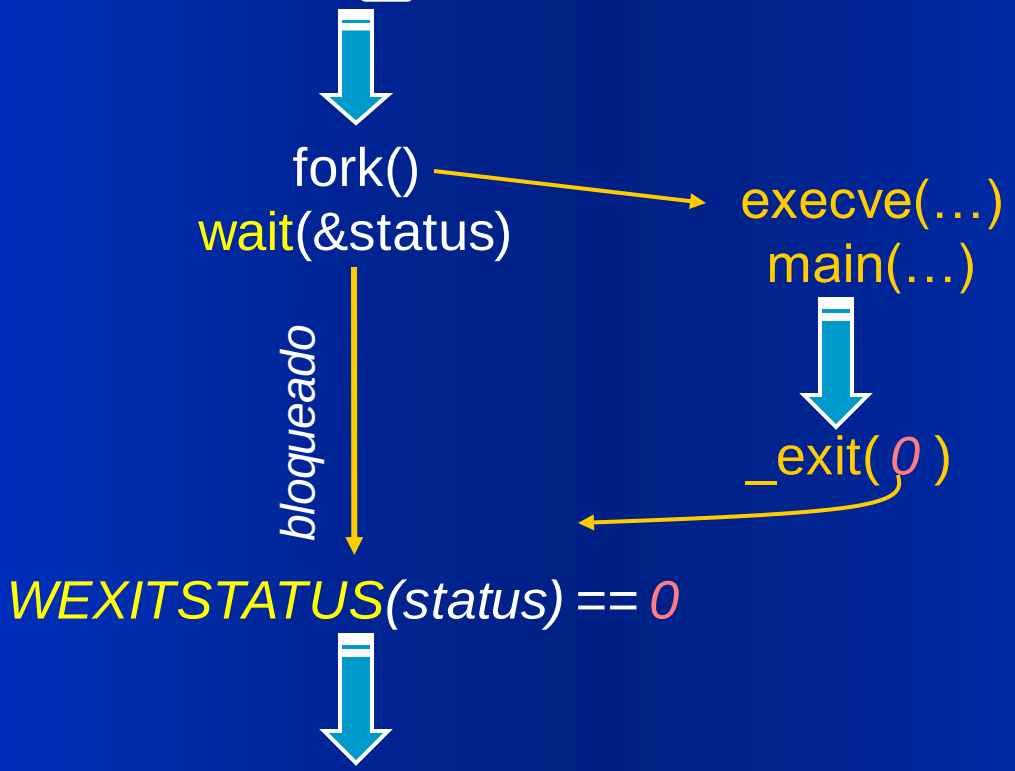
Se pid > 0 → aguarda o filho indicado por pid

Options: WNOHANG: (*wait with no hang*)

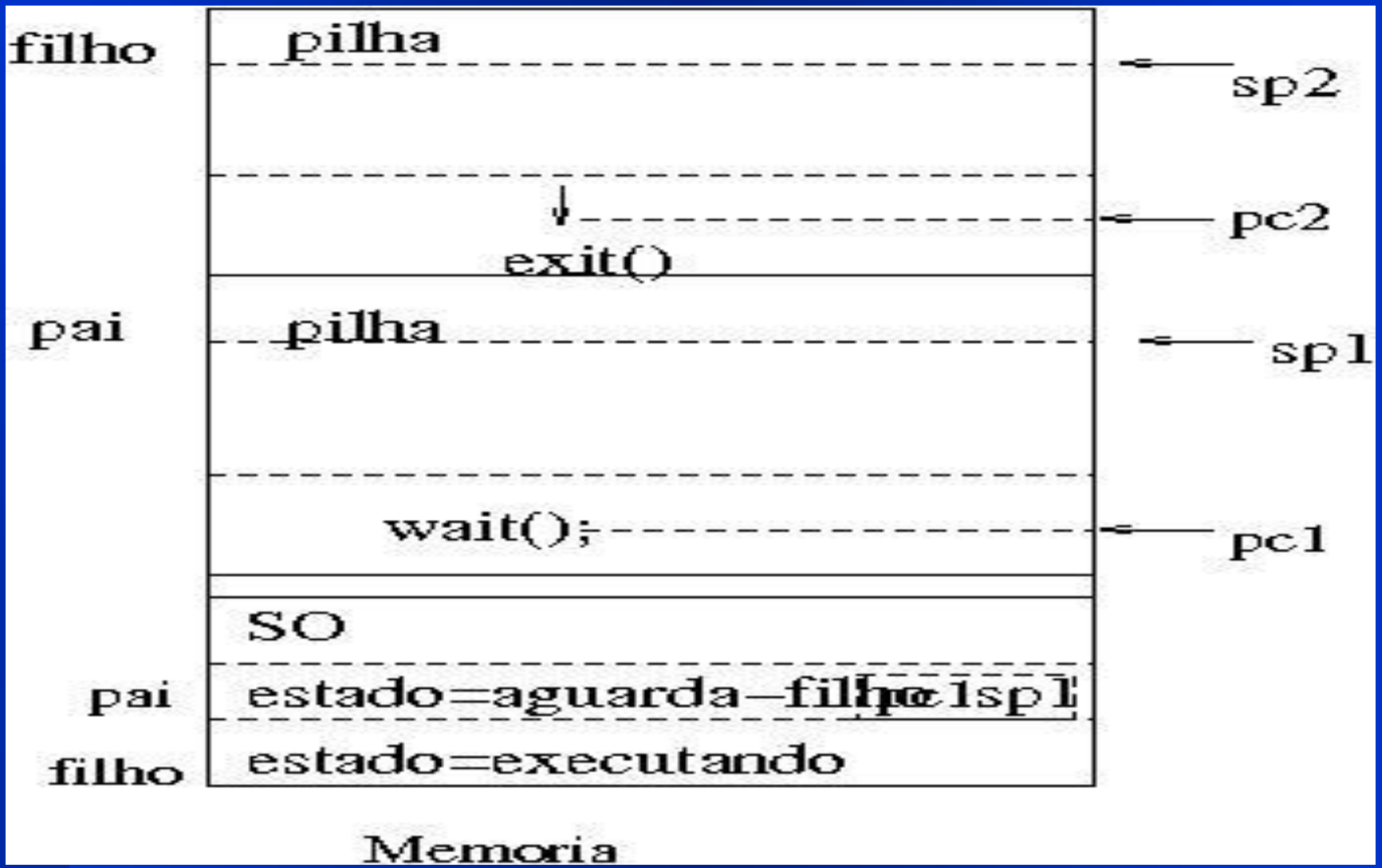
devolve 0 e retorna de imediato, se filho ainda não acabou

Esperar pela terminação

- `int wait(int *status)`
  - aguarda pela terminação de um dos filhos
  - `status`: código de terminação do processo: se terminou com `_exit` pode-se obter o *exit status*



Pai aguarda terminação do filho



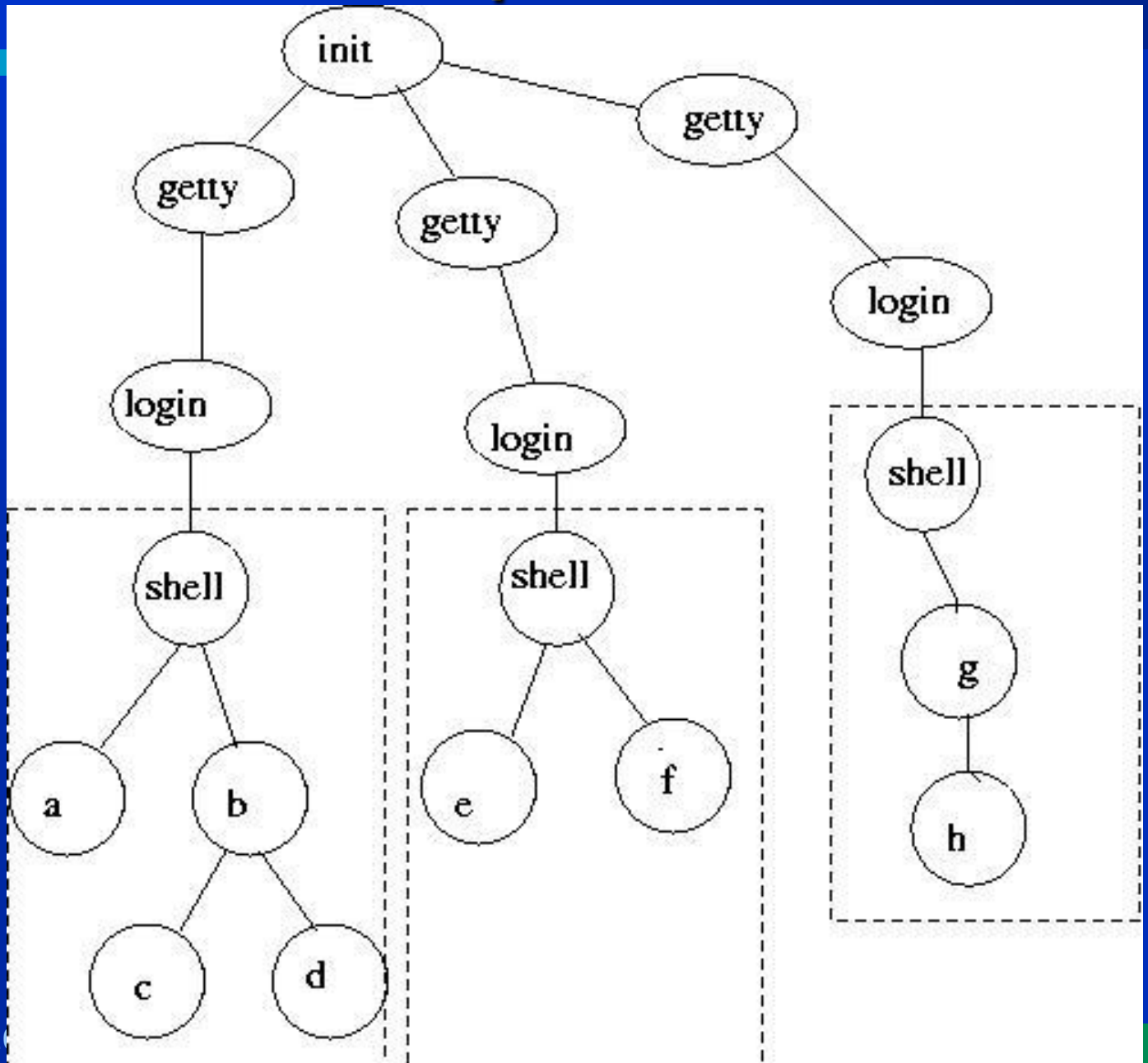
wait: cenários possíveis

- **O pai e o filho: executados concorrentemente em multiprogramação.**

Cenários possíveis:

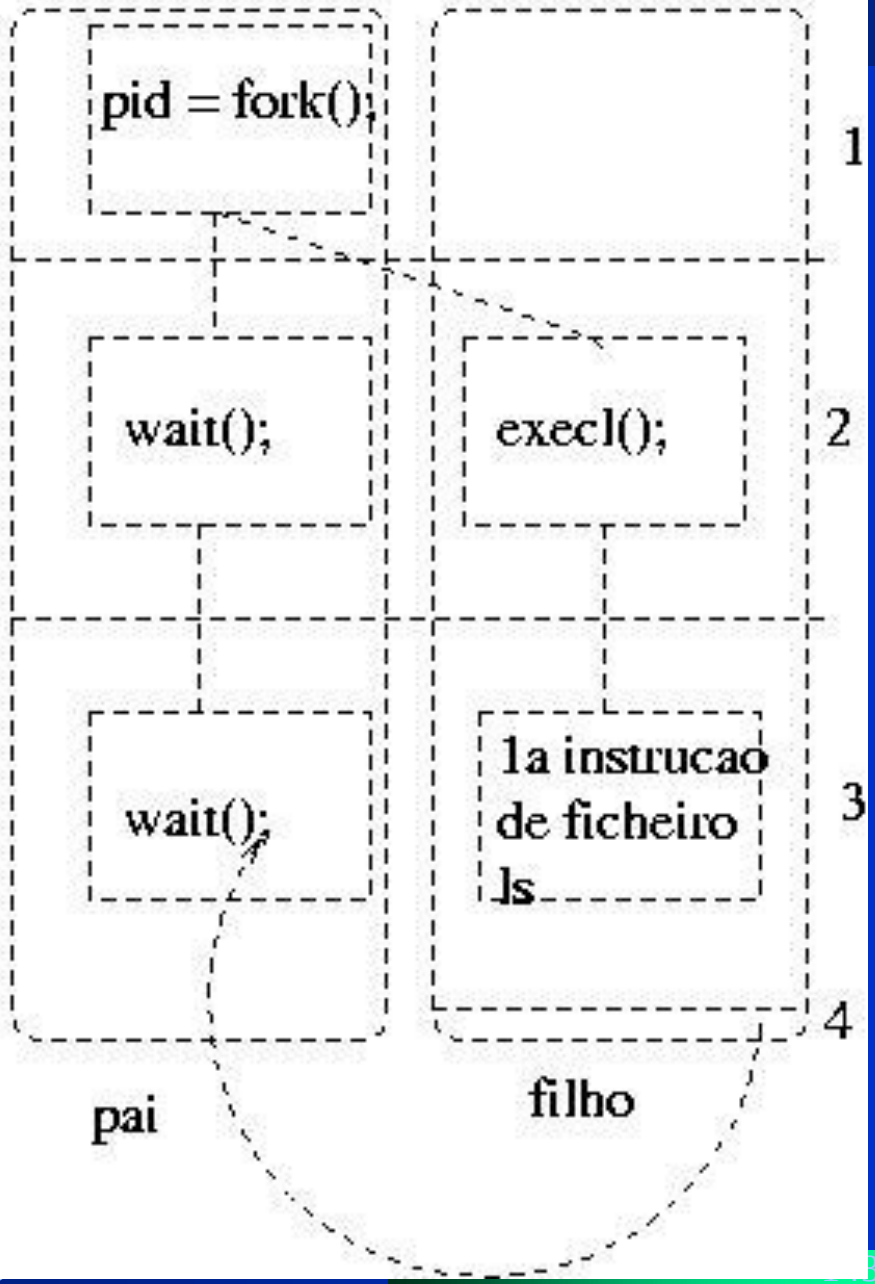
1. **Pai chega primeiro à chamada de `wait`**
   - **fica bloqueado até o filho chamar `_exit`**
2. **Filho chama `_exit` antes do pai chamar `wait`**
   - **o filho fica *ZOMBIE (defunct)***
   - **quando pai chamar `wait`, não bloqueia, e o resíduo do filho é eliminado.**
3. **Pai não chama `wait` e termina**
   - **o filho fica ÓRFÃO: é adoptado pelo `init`**

Árvore de processos no Unix



Exemplo de fork e exec

```
main()
{ int pid;

  pid = fork();

  if (pid>0)  { /* pai*/
       wait( (int *) 0 );
       printf(´ls completed\n´);
       exit(0);
  }

 if (pid == 0) { /*filho*/
     execl(´/bin/ls´,´ls´,´–l´, (char *)0);
     .........
  }
  ..... /*se pid <0*/

}
    o mesmo programa
```



Exemplo com execl

```
pid = fork();
switch (pid)
{
   case 0:
        chdir("/bin");
        execl("/bin/ls","ls",NULL);
        printf(stderr,"erro no execl\n");
   case -1: perror("fork");
            break;
   default:
        printf("Criei o %d\n", pid);
        wait( &st );
        if ( WIFEXIT(st) )
           printf("exit status: %d\n", WEXITSTATUS(st) );
}
```

Exemplo com execvp

```
pid = fork();
switch (pid)
{
  case 0:
      { char *av[3];

        av[0]="ls"; av[1]="/bin"; av[2]=NULL;
        execvp( "ls", av );
        perror( "exec ls" );
        exit(1);
      }
  case -1: perror("fork");
           break;
  default:
      wait( &st );
}
```

Shell: recebe comandos do teclado

Exemplo de *Shell* simplificado

```
while (true) {
    read_command(command, params);
    if (fork () > 0)    -- cria 1 processo
        wait (&status);    --   o pai espera
    else          --  o filho executa o comando
        execv(command,params);
  }
```

cp file1 file2 **na linha de comandos**

main(argc, argv) **no programa**