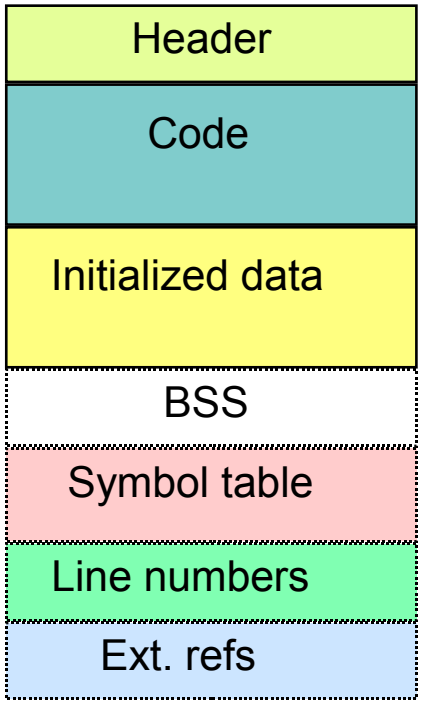
# Processos no UNIX

*Wagner M Nunan Zola*
***wagner@inf.ufpr.br***
***Departamento de Informática – UFPR***

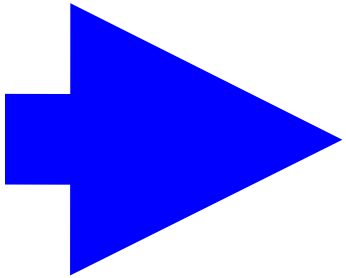
# Introdução

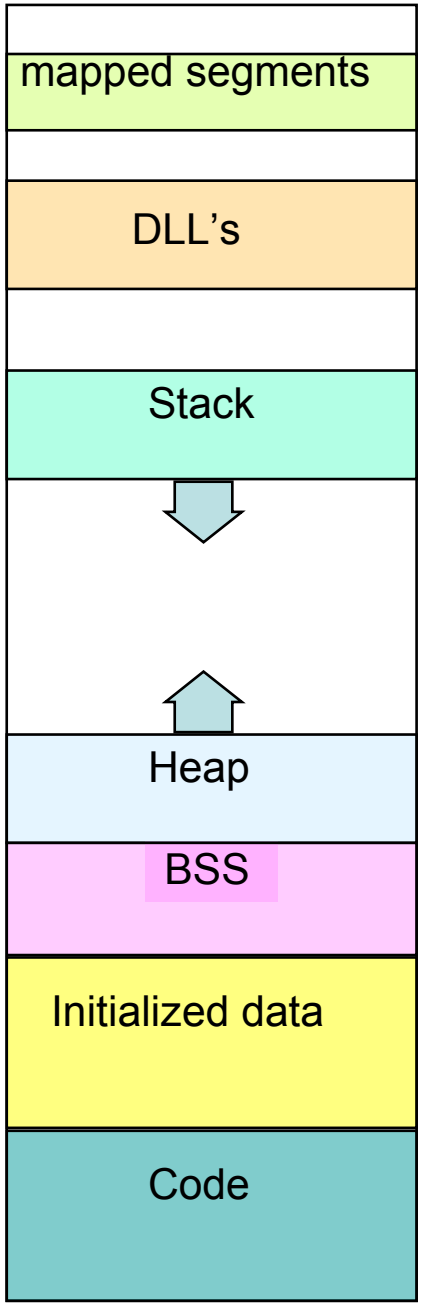
- Programas X Processo
  - Programa é passivo
    - Código + Dados
  - Processo é um programa em execução
    - Pilha, regs, program counter, espaços de memória, heap
- Exemplo:
  - Dois usuários executando Firefox:
    - é o mesmo programa
    - são processos diferentes

# Programa e seu “espaço de endereçamento”



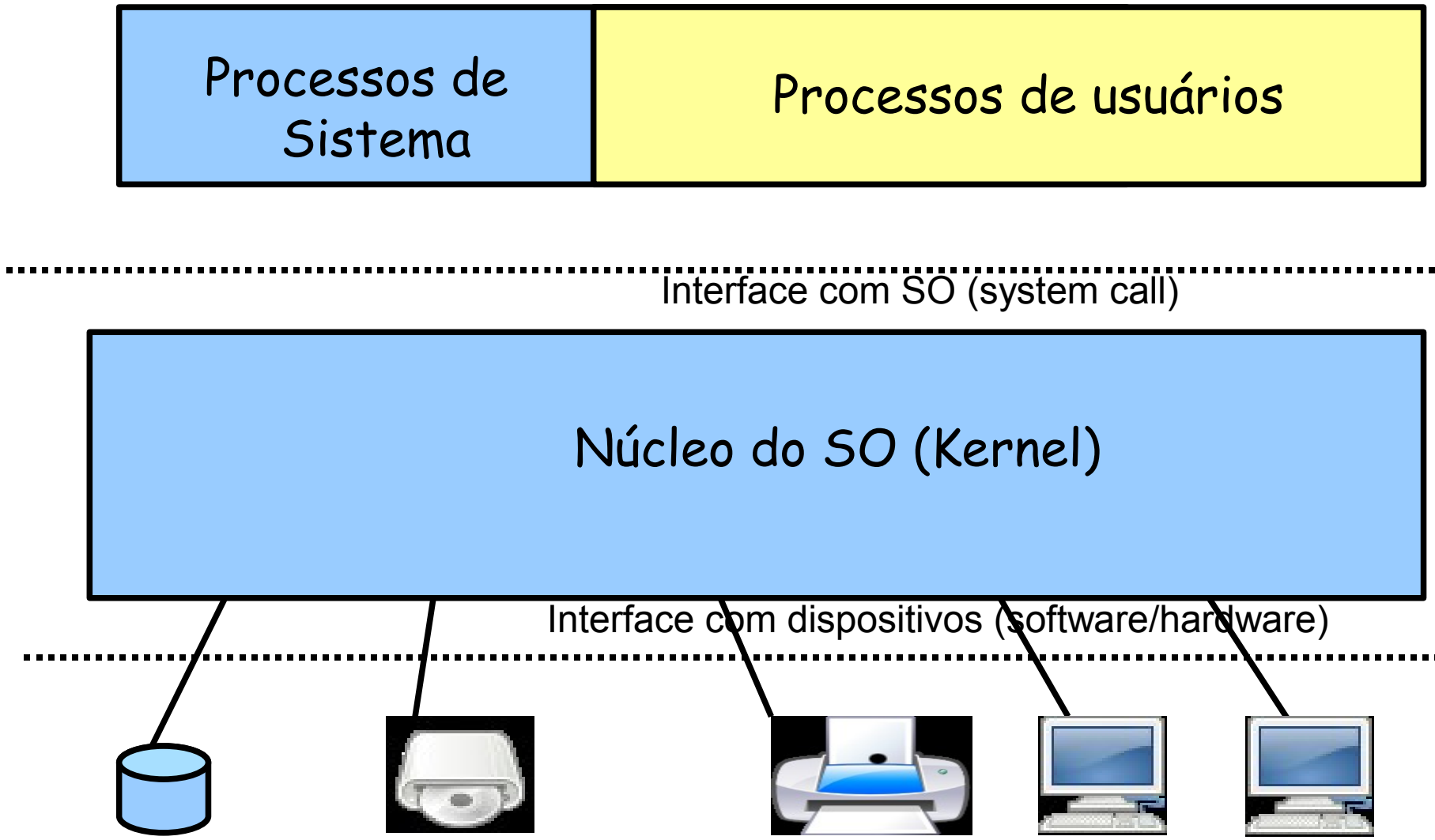

**Binário “Executavel” (programa)**



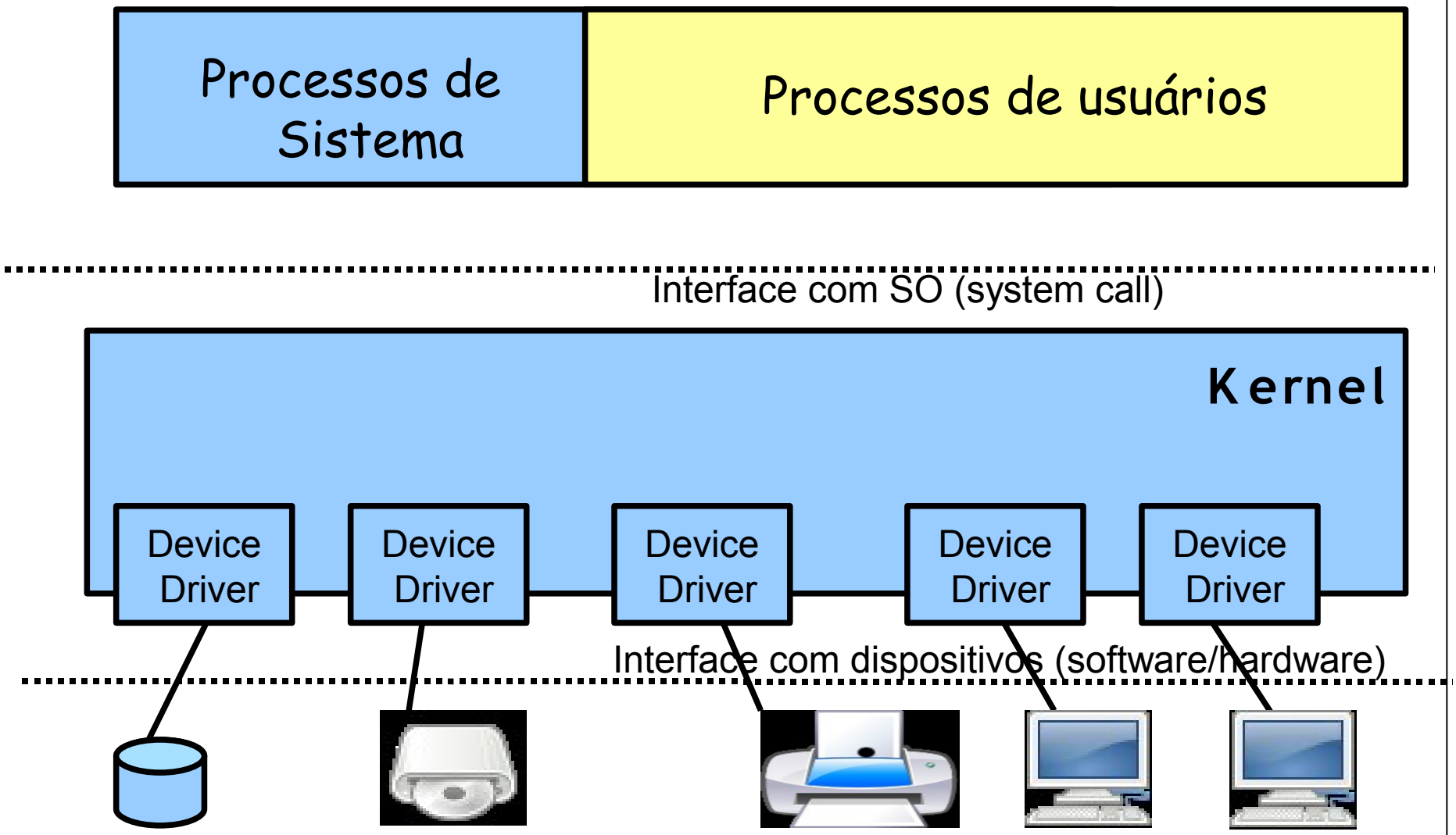



**Process address space (espaço em memória)**

# Estrutura do Sistema Operacional (SO)

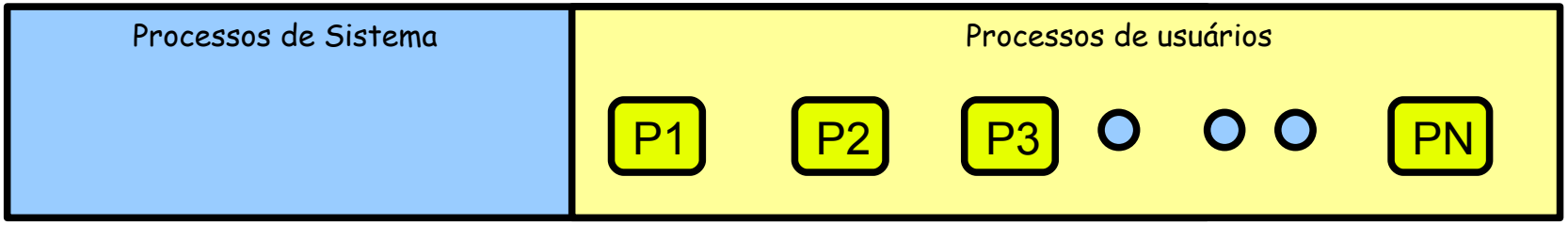


# Estrutura do Sistema Operacional (SO)

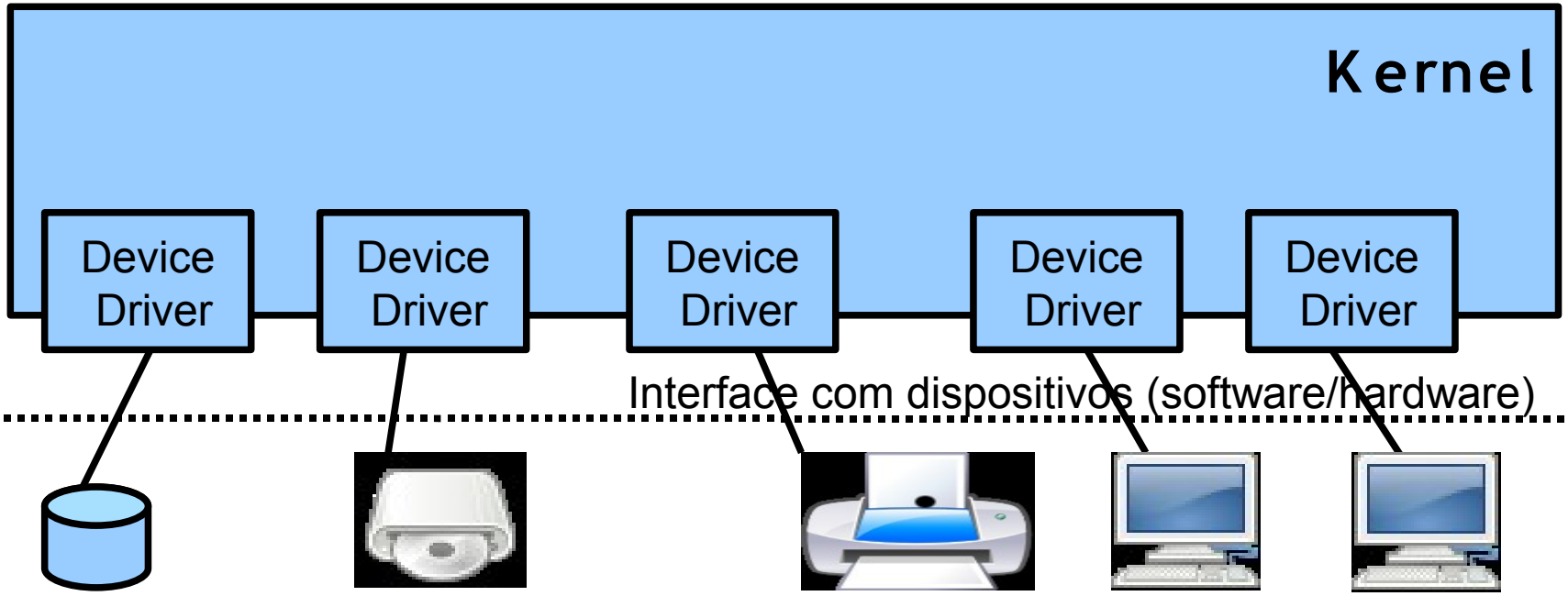


# Estrutura do Sistema Operacional (SO) ==>

**Processos de usuários**

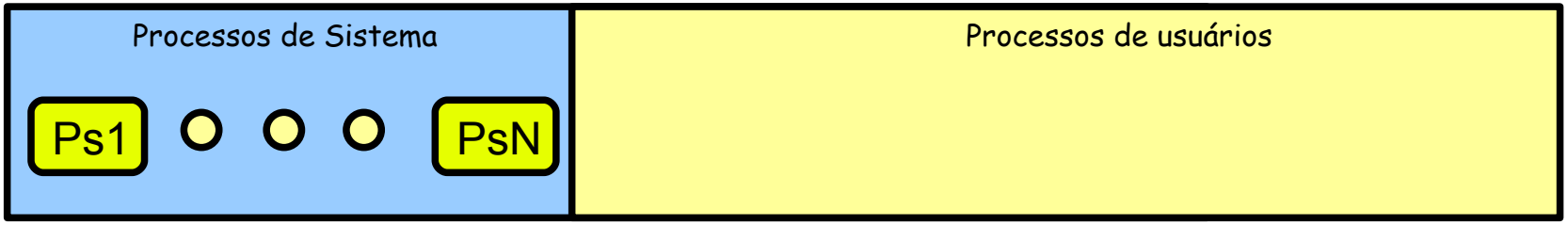



Interface com SO (system call)

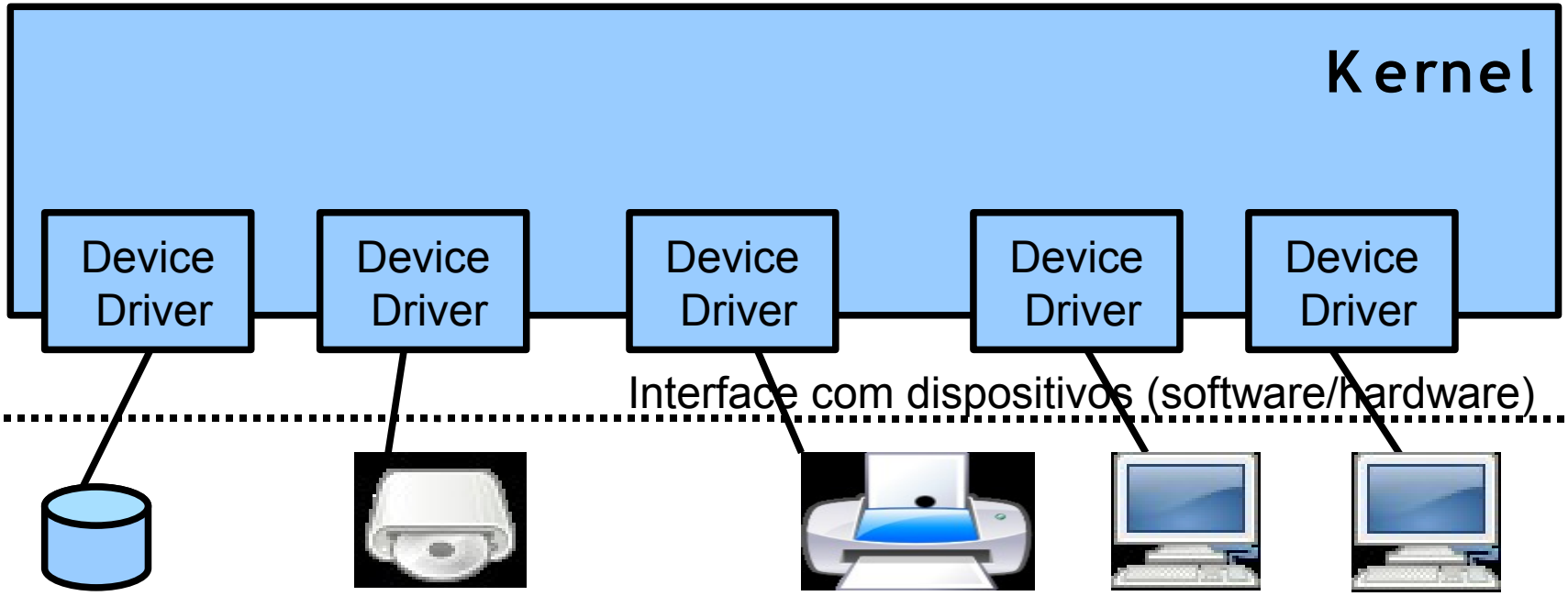


# Estrutura do Sistema Operacional (SO) ==>

## Processos de sistema



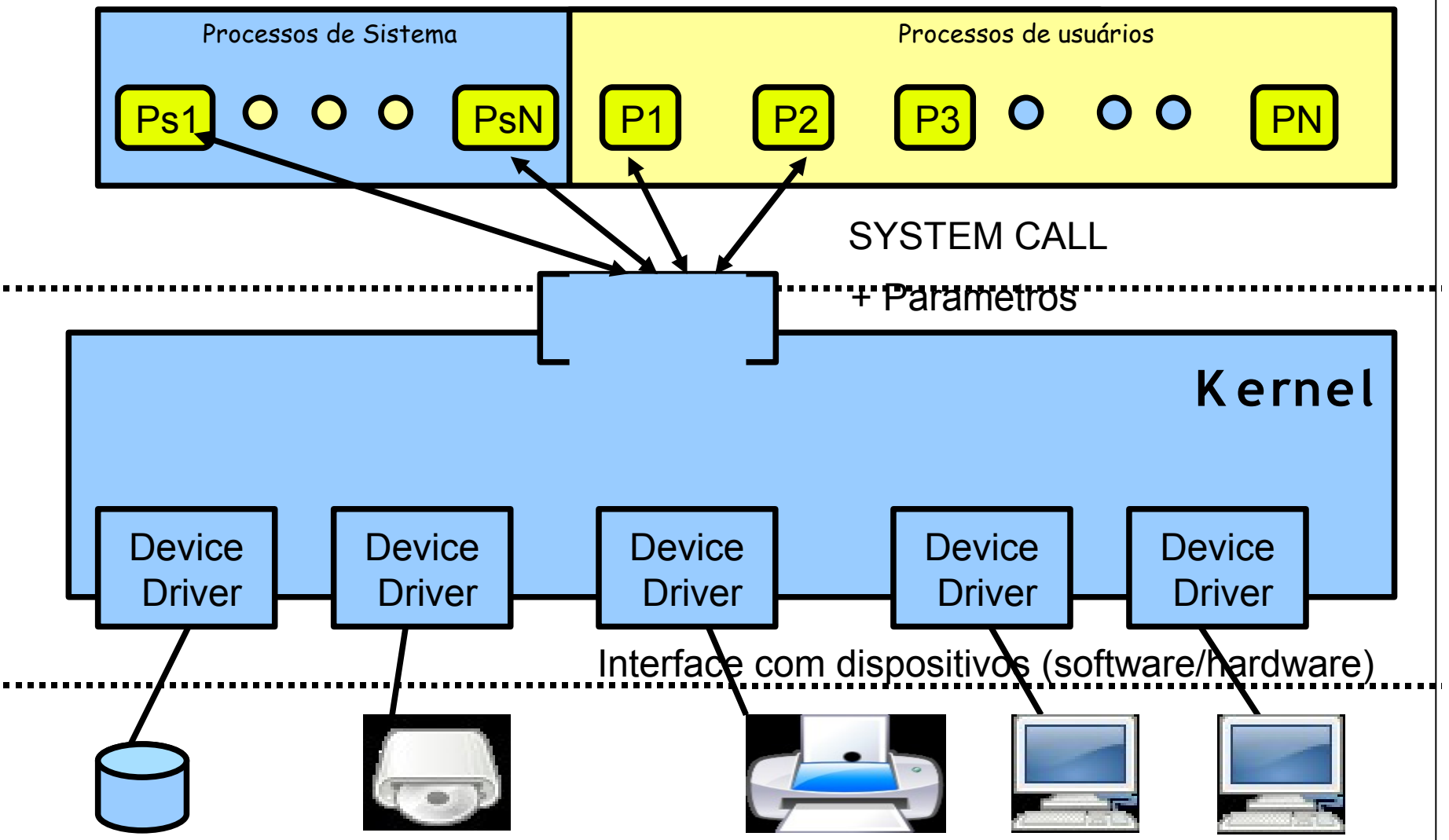

Interface com SO (system call)



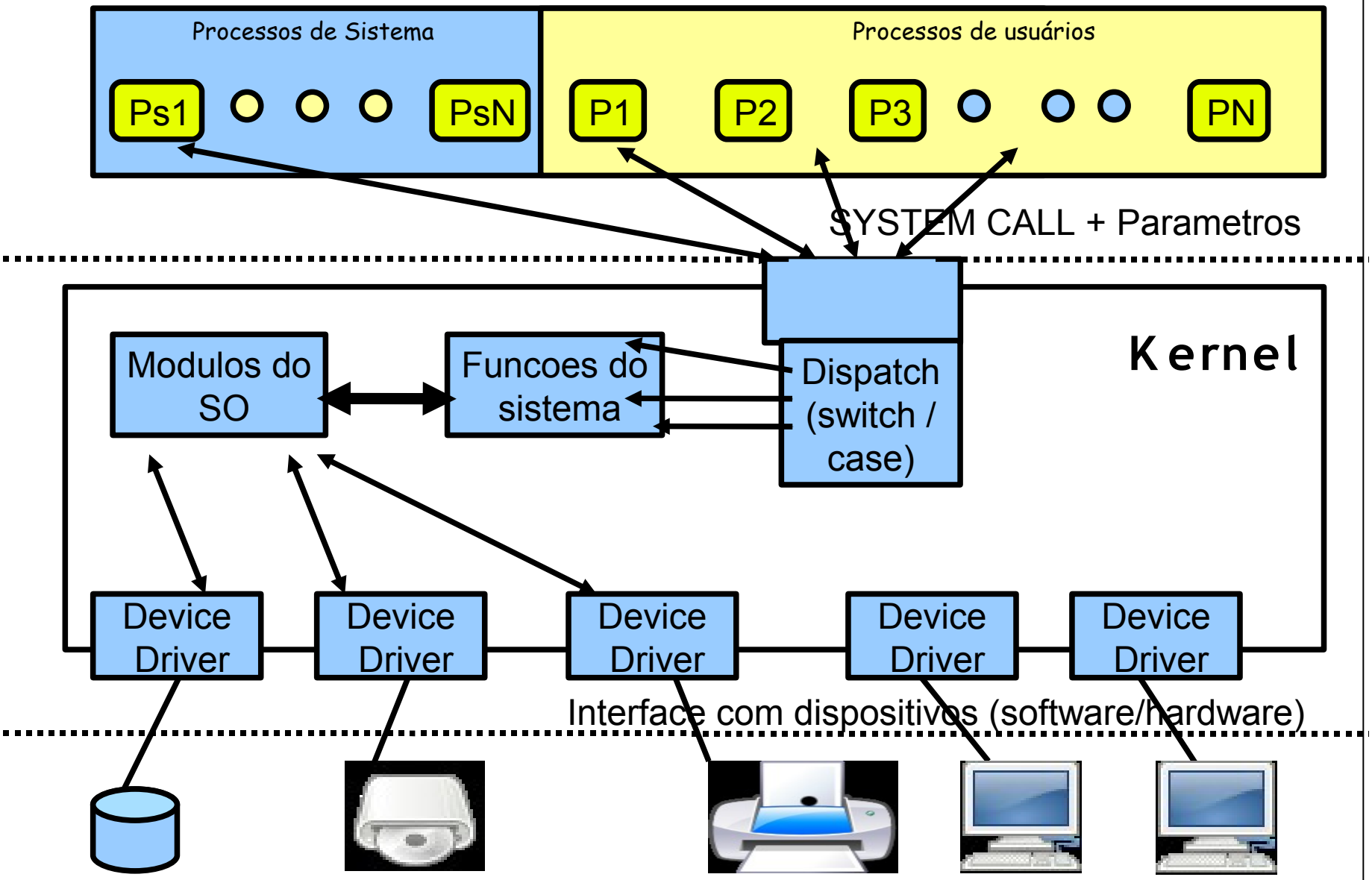

Interface com dispositivos (software/hardware)

Estrutura do Sistema Operacional

# Interface com SO



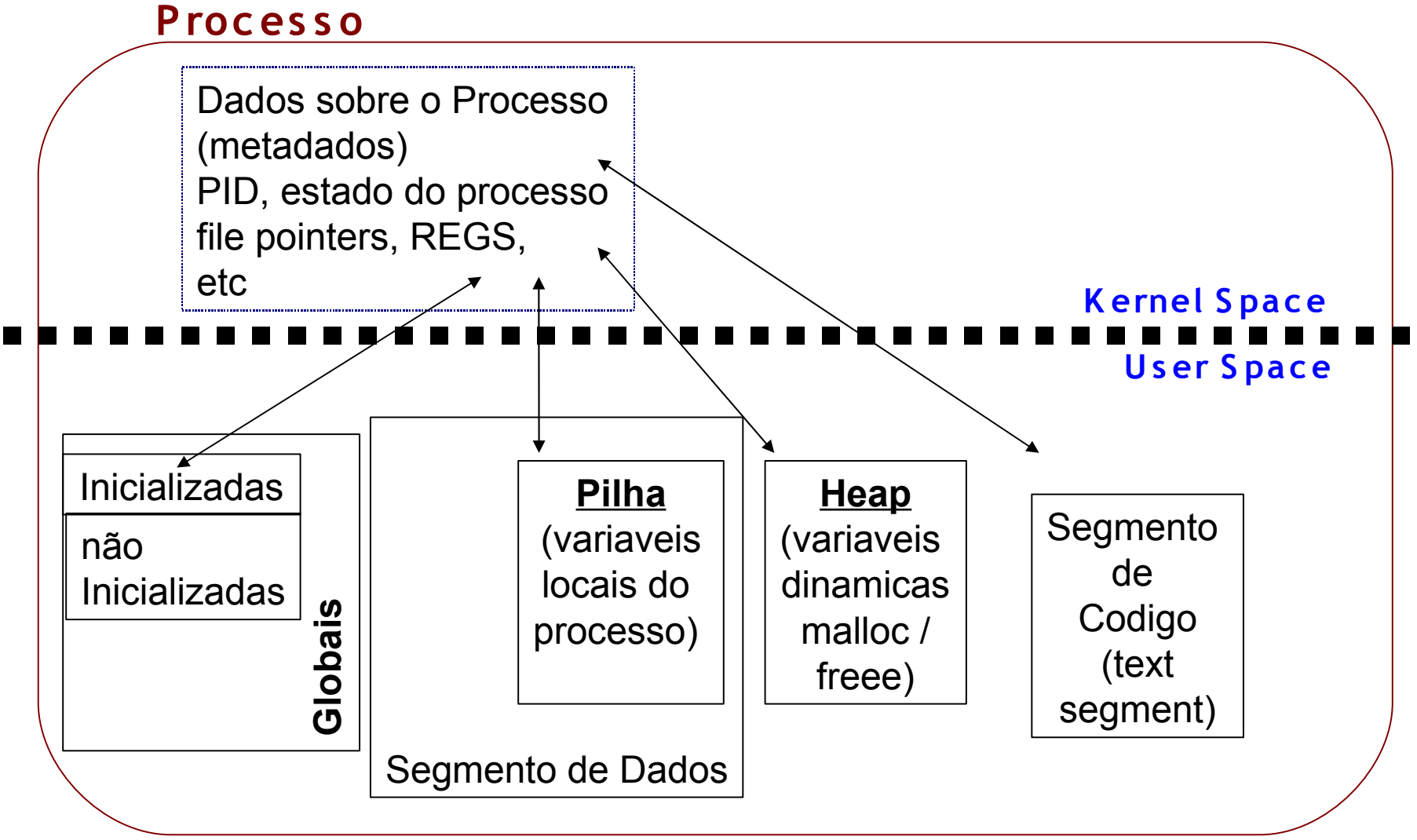
# System Call Dispatch



# Componentes do SO

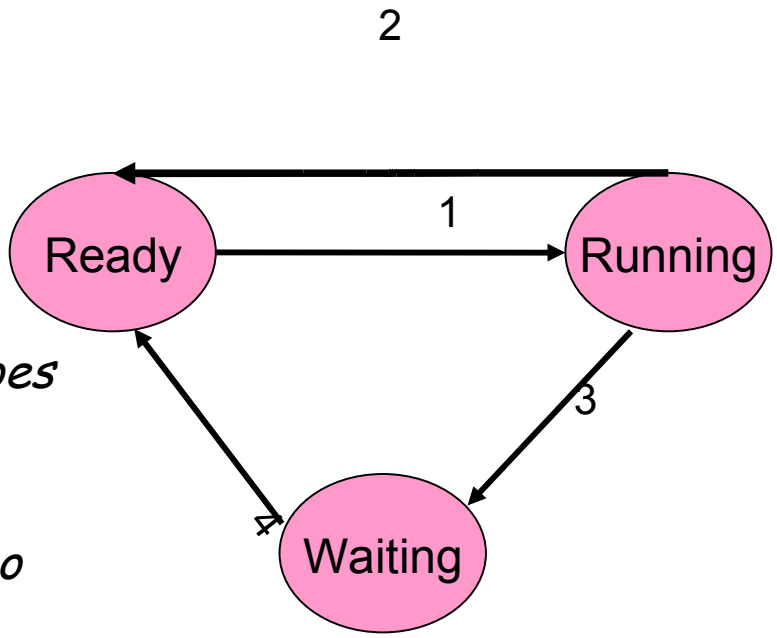
- Kernel
  - Functors de sistema
  - Modulos do SO
    - Subsistemas=modularizacao, grupos de funcoes relacionadas
      - ex= subsistema virtual de arquivos (VFS), network file system (NFS)
  - Bibliotecas (no kernel)
- Processos de Sistema
  - Shells
  - Daemons (ex= line printer daemon, NFS daemon)
- Bibliotecas de nivel de usuario

# Processos
# Modelo em memoria



# Estados de Processos

- *"Estado" da execucao de um processo:*
  - Indica o que ele esta fazendo
  - Basicamente necessitamos no minimo de 3 estados:
    - ***Ready**: pronto para ser colocado na CPU*
    - ***Running**: executando instrucoes na CPU*
    - ***Waiting**: (ou "blocked") esperando por algum evento no sistema (ex. completar I/O)*
- *Processos passam por diferentes estados*
  - » *ver diagrama de estados de processos*

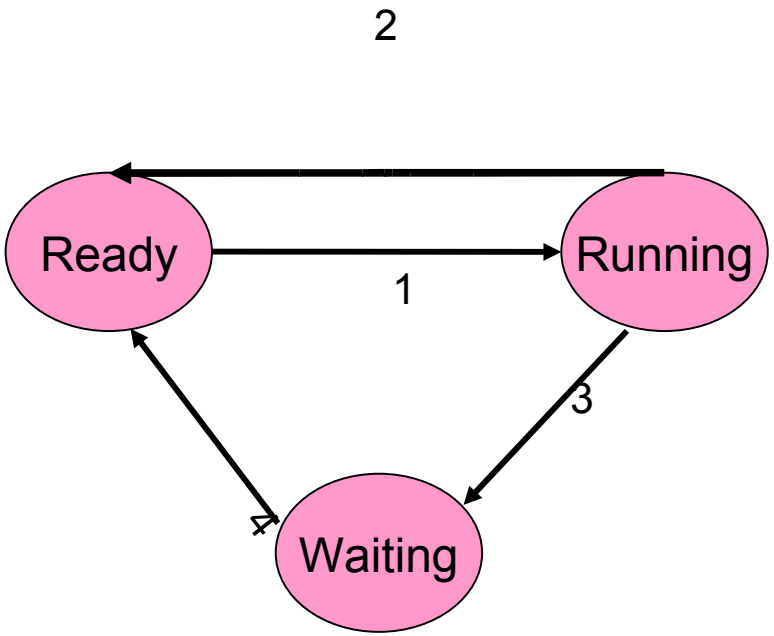


# Estados de Processos:
## *Entendendo as transicoes (usando 3 estados)*

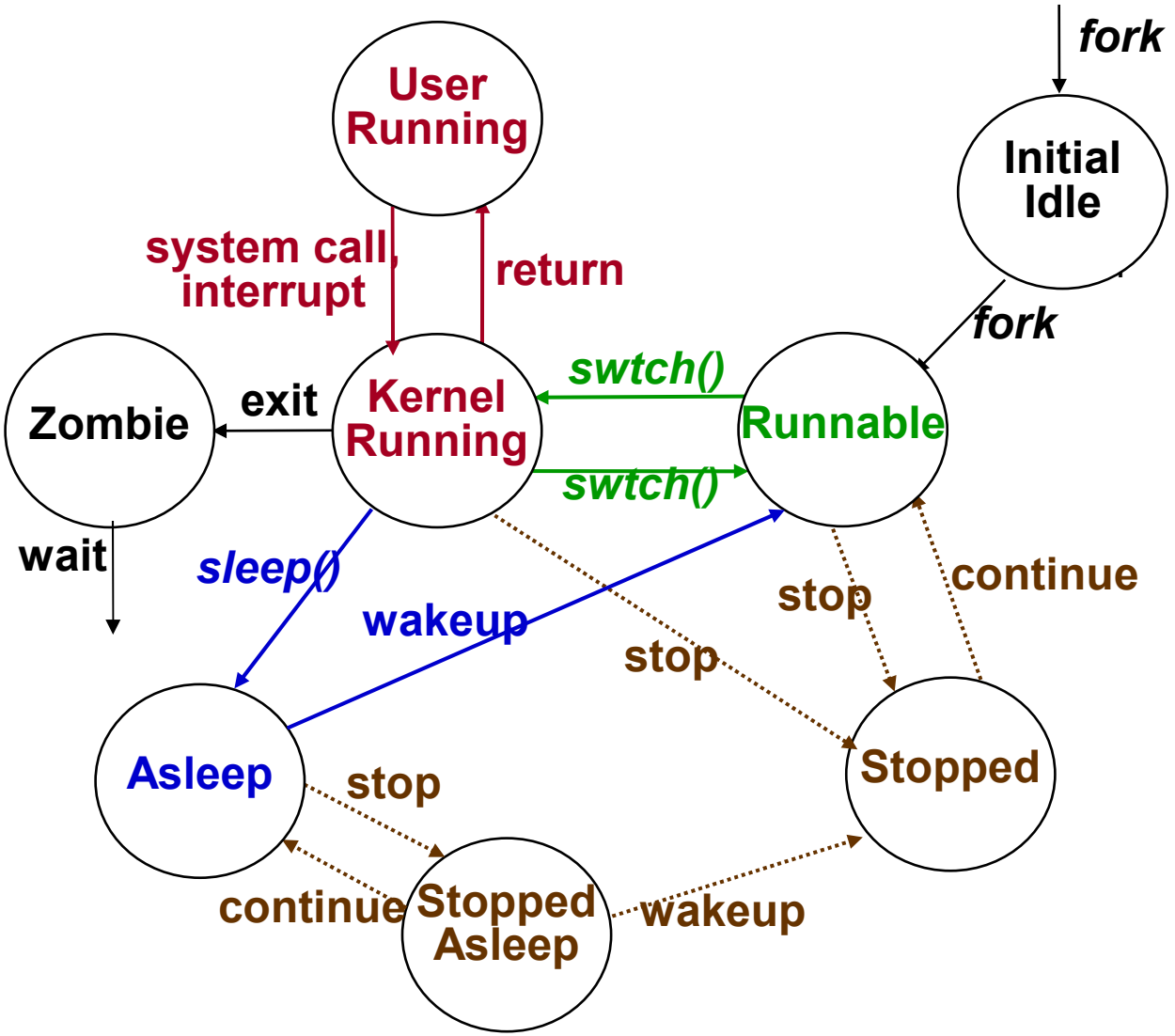
*1) processo bloqueado para fazer I/O (por exemplo)*

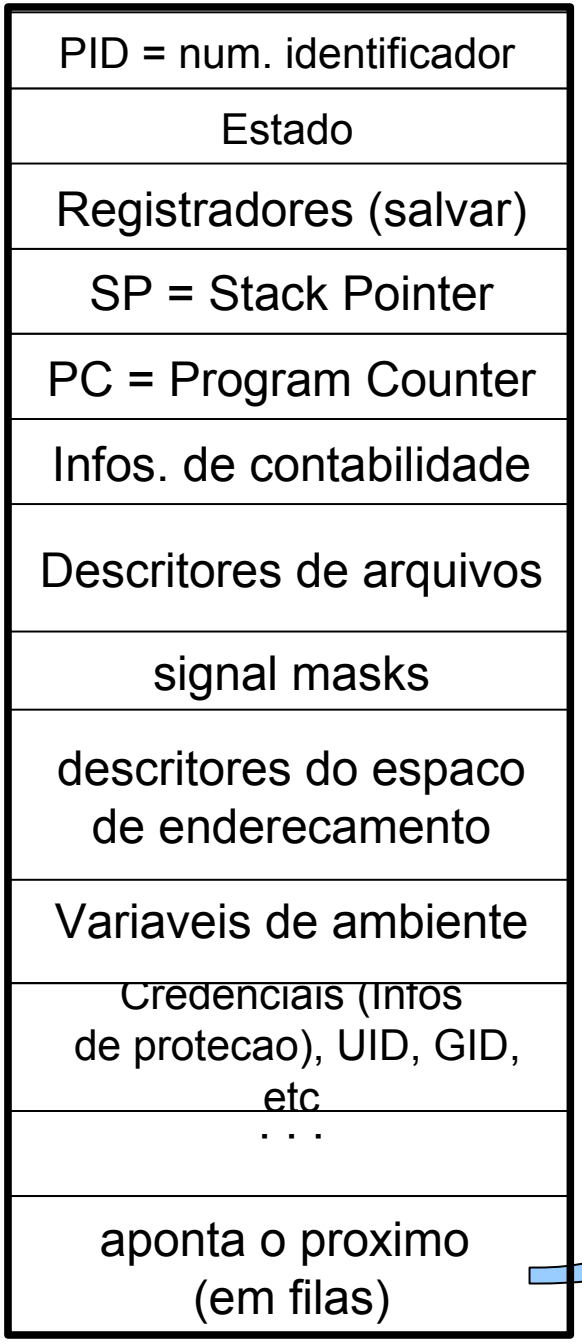
- *"escalonador" de processos seleciona **outro** processo para rodar (preempt).*
- *"escalonador" de processos seleciona **este** processo para rodar (preempt).*

*4) ocorre evento esperado (e.g. I/O completa)*



# Estados de Processos:
## *Exemplo real de diagrama de estados de processos*






## Estruturas de Dados que representam processos,

# O Descritor de Processos

- *tambem denominado* ***"Process Control Block" (PCB)***
- *estrutura de dados mantida pelo SO para cada processo*

# Trocas de contexto = **trocas de processos**

- Um processo em execucao usa (potencialmente) todos os registradores da CPU
  - ex. PC, SP, PSW, Regs. de proposito geral
- (a) Para "retirar" o processo corrente da CPU...
  - Salvar todos os registradores (execution context) no PCB
  - para retornar o mesmo de onde parou basta carregar novamente os registradores
- (b) para executar outro P2...
  - carregar os registradores com o contexto de P2

⇒ **Troca de contexto ou "Context Switch"**

- Significa, **trocar um processo da CPU por outro**
- envolve os dois passos (a e b) acima
- dependente de arquitetura (tipos e quantidades de registradores)

# Detalhe das trocas de contexto

- dificil de ser implementada
  - SO deve salvar estado sem trocar de estado
  - tem de ser feito sem perder valores em registradores (praticamente nao usa registradores no codigo da troca)
    - CISC: instrucao complexa salva todos os registradores
    - RISC: reserva registradores para uso exclusivo no kernel
- Custos: troca de contexto e' "overhead"
    - custo direto:
      - custo de copiar os valores dos registradores para a memoria e vice-versa
      - custo da "selecao" do proximo processo
    - custo indireto:
      - custo de invalidacao do(s) cache(s), TLB, etc
      - esperar tambem que instrucoes no pipeline terminem

# Credenciais de um Processo

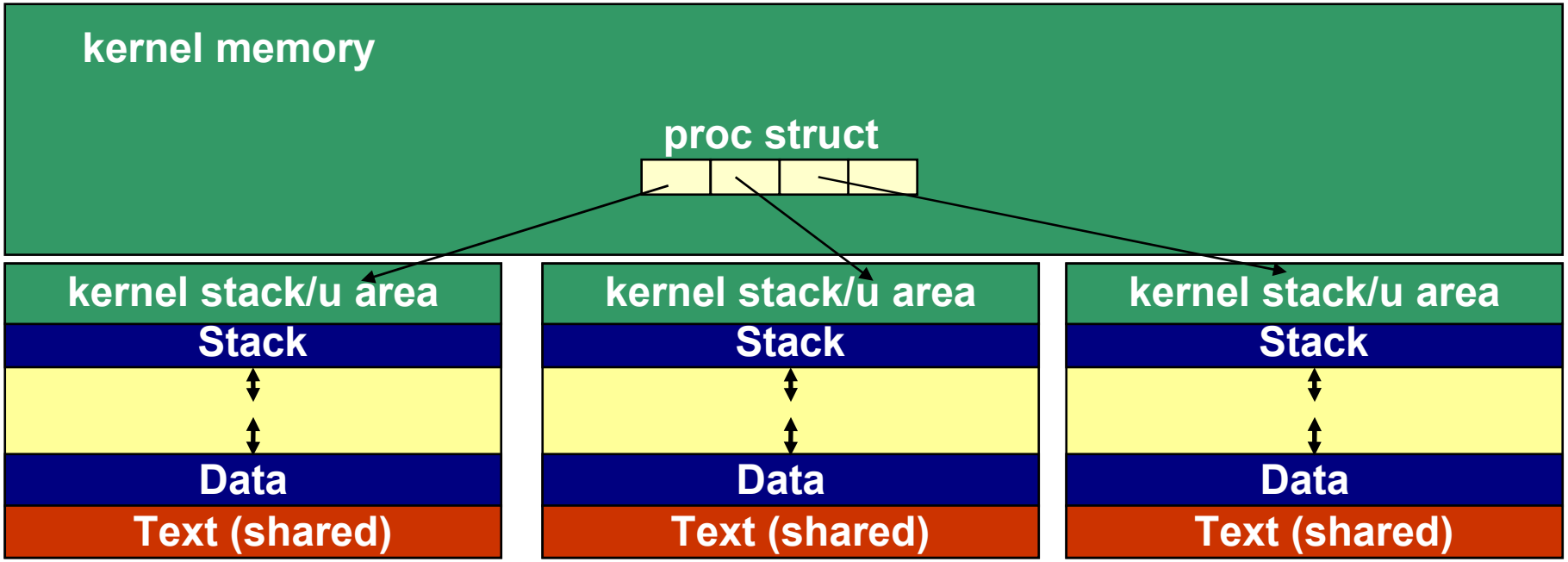
- processos tem UID/GID **real e efetiva**.
- Efetiva: usada para abrir arquivos.
- Real: usada para enviar sinais.
- programas com atributo **suid** (bit): mudam UID efetivo.
- programas com atributo **sgid** (bit): mudam GID efetivo.
- setuid() ou setgid(): permitem voltar ao ID real.
  - SYSV mantém copias de UID e GID que são restaurados por setuid.
  - BSD suporta vários grupos por utilizador.

# Descritores e seus processos no Linux (`task struct`)

Em /usr/src/linux/include/kernel/sched.h

```
struct task_struct {
        /*
         * offsets of these are hardcoded
         * elsewhere - touch with care
         */
    /* -1 unrunnable, 0 runnable, >0 stopped*/
        volatile long state;
    /* per process flags, defined below */
        unsigned long flags;
        int sigpending;
    /* thread address space:
            0-0xBFFFFFFF for user-thead
            0-0xFFFFFFFF for kernel-thread
    */
        mm_segment_t addr_limit;
        struct exec_domain *exec_domain;
        volatile long need_resched;
        unsigned long ptrace;
    /* Lock depth */
        int lock_depth;
/*
 * offset 32 begins here on 32-bit platforms.
 * We keep all fields in a single cacheline
 * that are needed for
 * the goodness() loop in schedule().
 */
        long counter;
        long nice;
        unsigned long policy;
        struct mm_struct *mm;
        int has_cpu, processor;
        unsigned long cpus_allowed;
        /*
         * (only the 'next' pointer fits
         *  into the cacheline, but
         * that's just fine.)
         */
        struct list_head run_list;
        unsigned long sleep_time;
        struct task_struct *next_task,
                           *prev_task;
        struct mm_struct *active_mm;
/* task state */
        struct linux_binfmt *binfmt;
        int exit_code, exit_signal;
  /*  The signal sent when the parent dies  */
        int pdeath_signal;
        /* ??? */
        unsigned long personality;
        int dumpable:1;
        int did_exec:1;
        pid_t pid;
        pid_t pgrp;
        pid_t tty_old_pgrp;
        pid_t session;
        pid_t tgid;
  /* boolean value for session group leader */
        int leader;
  /*
   * pointers to (original) parent process,
   * youngest child, younger sibling,
   * older sibling, respectively.
   * (p->father can be replaced with
   * p->p_pptr->pid)
   */
        struct task_struct *p_opptr, *p_pptr,
                           *p_cptr, *p_ysptr,
                           *p_osptr;
        struct list_head thread_group;
        /* PID hash table linkage. */
        struct task_struct *pidhash_next;
        struct task_struct **pidhash_pprev;
        /* for wait4() */
        wait_queue_head_t wait_chldexit;
            /* for vfork() */
        struct semaphore *vfork_sem;
        unsigned long rt_priority;
        unsigned long it_real_value,
                it_prof_value, it_virt_value;
        unsigned long it_real_incr,
                it_prof_incr, it_virt_incr;

   struct timer_list real_timer;

;* ... continua ... *;
```

# Descritores e seus processos no UNIX




- OBS. as estruturas acima sao de Unix
- OBS2. em Linux /proc eh **outra coisa**...
  - faz o mapeamento de variaveis do SO (algumas) que podem ser consultadas via chamadas de I/O (read e write, dependendo do caso).

# Descritores e seus processos no UNIX (cont)

- U area.
  - ✓ Parte do espaço de usuario (user space) acima da pilha.
  - ✓ contem infos necessarias quando processo esta rodando.
  - ✓ pode ser "swapped"
- Proc
  - ✓ contem infos necessaria sobre o processo mesmo quando nao esta rodando
  - ✓ Nao pode ser transferido para swap
  - ✓ tradicionalmente uma tabela de tamanho fixo

Como criar processos no Unix ...
proxima aula!

# Inicio da criação de processos Unix

- Depois do "boot" SO inicia o primeiro processo
  - E.g. sched for Solaris
- O primeiro processo cria os outros:
  - o processo criador de outro processo é chamado "pai"
  - o novo processo (criado) é denominado "filho"
    - ==> conceitualmente poderiamos obter um diagrama formando uma "árvore" de processos
- por exemplo, no UNIX o primeiro processo é chamado ***init***
  - *com PID = 1*
  - ele cria todos os gettys (processos de login) e daemons
  - o init não deve morrer nunca
    - » controla a configuracao do sistema (ex: núm. de processos, prioridades, etc)

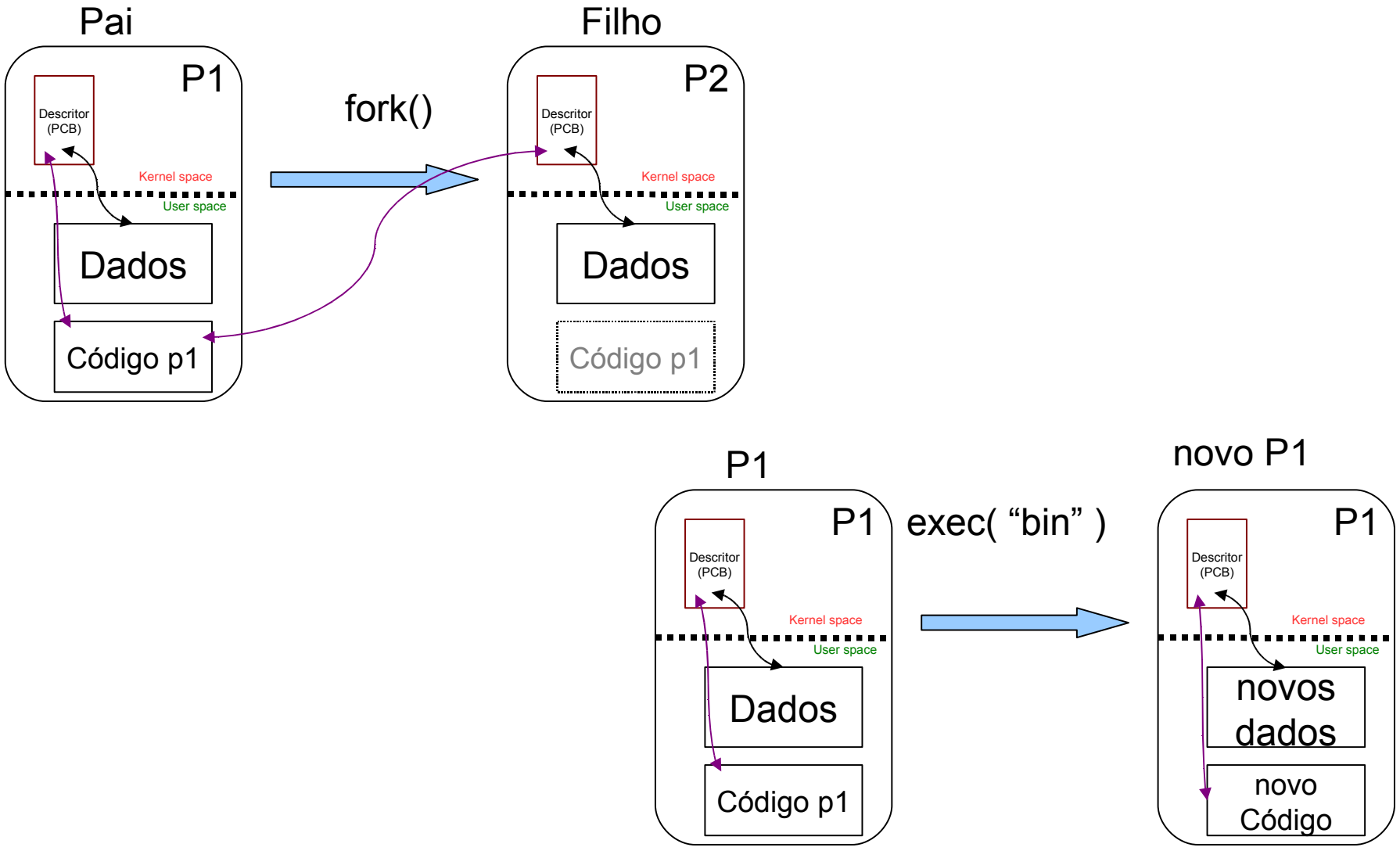
# Criação de processos (genericamente)

- Processo pai cria "filhos" (ou clones), que por sua vez criam outros formando uma árvore de processos
- opções possíveis para o **compartilhamento de recursos**:
  - Pai e filhos compartilham todos os recursos
  - Filhos compartilham subconjunto de recursos do pai
  - Pai e filhos não compartilham recursos
- Execuções possíveis: (opções)
  - Pai e filhos executam concorrentemente
  - Pai espera até filhos terminarem

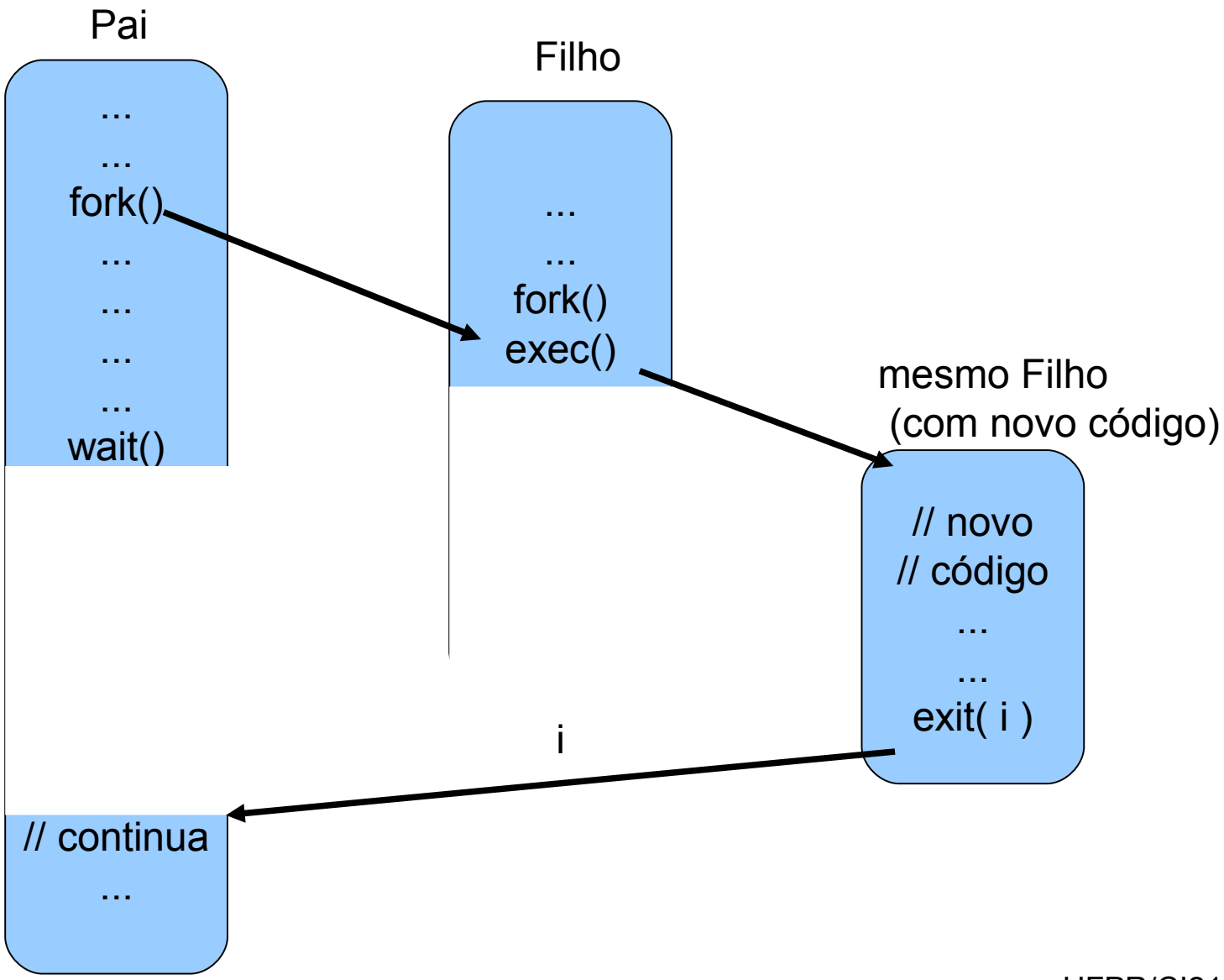
# Criação de Processos (Cont.)

- Possibilidades para o espaço de endereçamento
  - Filho é duplicata do pai (espaço separado)
  - Filho tem programa carregado no espaço do pai (substituição)
- Exemplos do UNIX
  - Chamada a **fork** cria novo processo
  - Chamada **exec** (ou execve) usada depois do fork (no código do "clone/filho") para substituir programa no espaço do "clone"

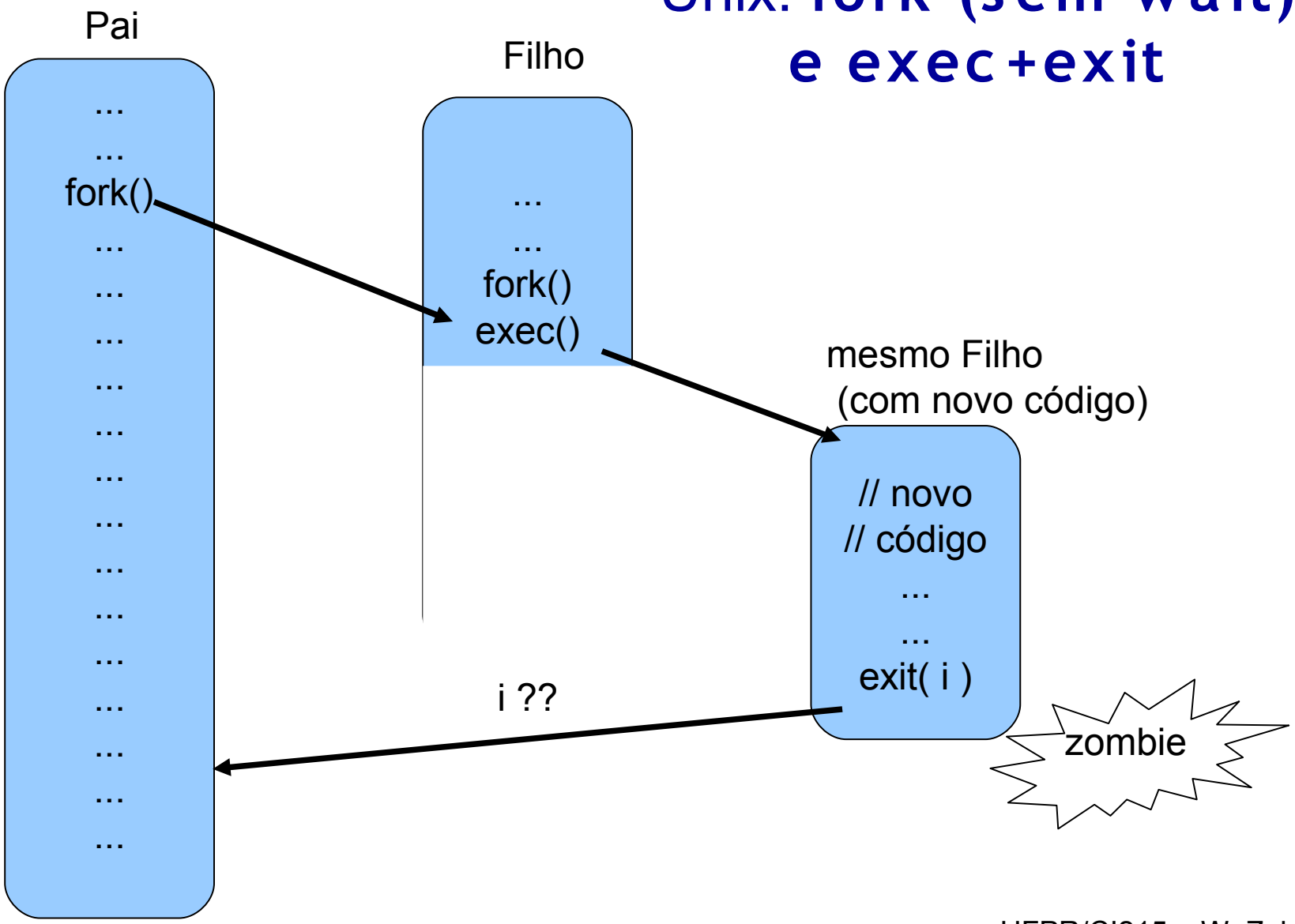
# Criação de Processos
# Unix: **fork e exec**



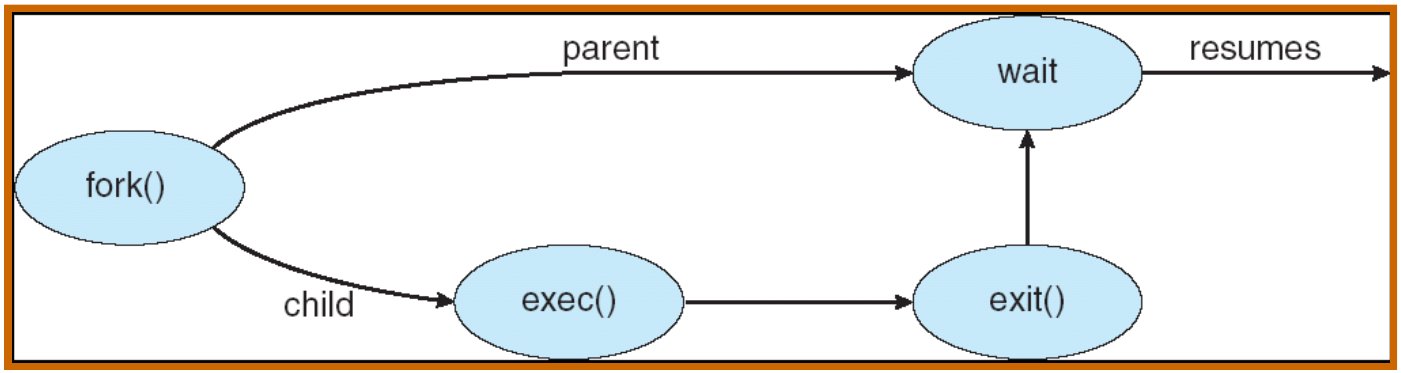
# Criação de Processos
# Unix: fork+wait e exec+exit



# Criação de Processos Unix: **fork (sem wait) e exec+exit**



# Criação de Processos
# Unix: **fork+wait e exec+exit (outra visão)**




- conceitualmente:
  - fork() é uma função que: chamada uma vez, retorna duas vezes
    - uma vez no processo pai
    - outra vez no processo filho
- fork retorna um número inteiro:
  - » **0 (zero), no filho** ---> inteiro retornado na chamada fork()
  - » **-1 , no pai** ---> em caso de erro, ex. não foi possível criar o filho
  - » **valor positivo, no pai** ---> é o PID do filho

# observações sobre fork() e exec()

- arquivos abertos:
  - são compartihados entre pai e filho, i.e. após o fork() os arquivos abertos são os mesmos
- no **exec**:
  - PID não muda

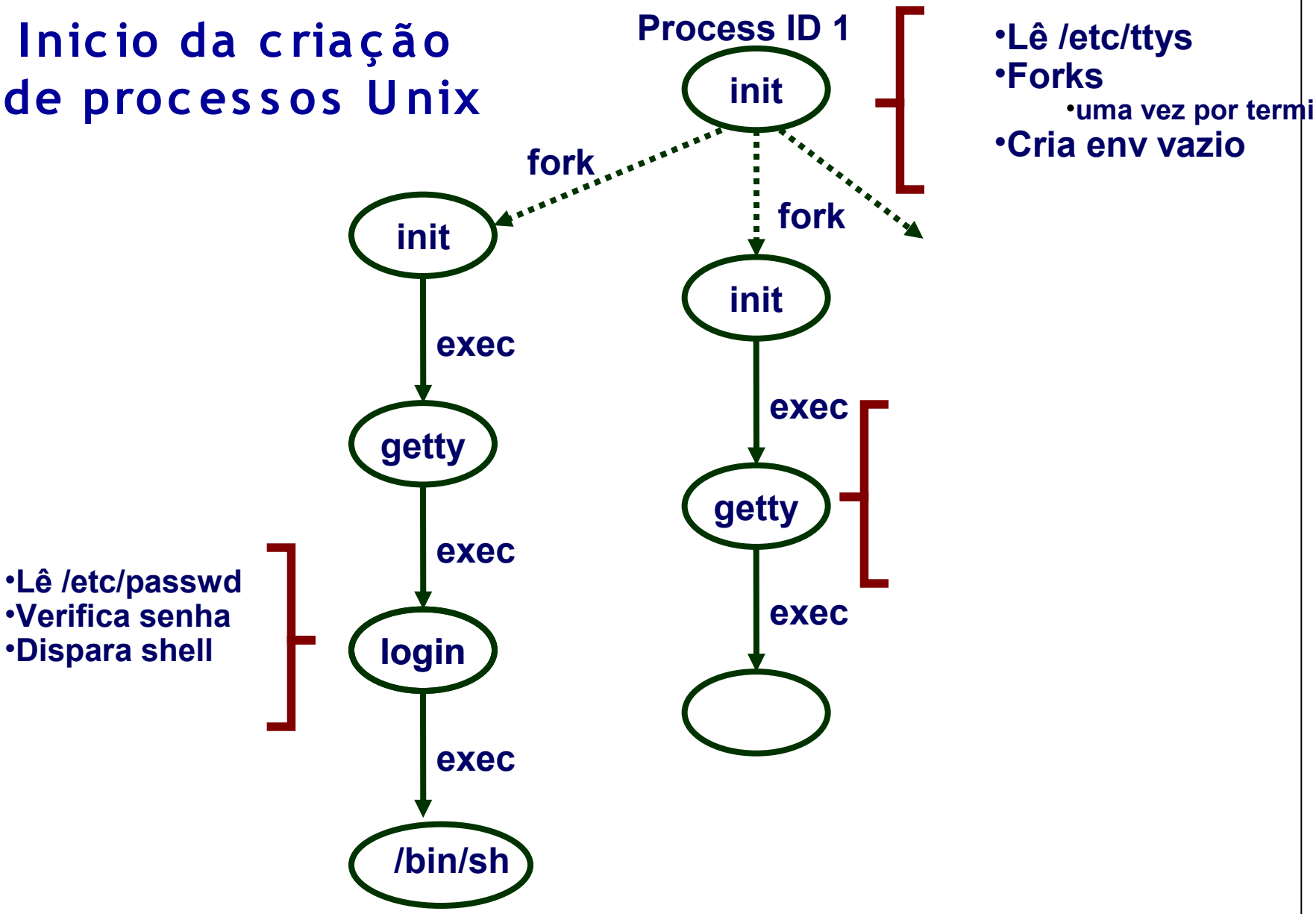
# código exemplo com fork() e exec()

```
int main()
{   pid_t  r;

   /* fork another process */
   r = fork();
   if (r < 0) { /* error occurred */

   fprintf(stderr, "Fork Failed");
       exit(-1);

     } else if (r == 0) { /* child process */
              execlp("/bin/ls", "ls", NULL);

   } else { /* parent process */

         /* parent will wait for the child to
   complete */
         wait (NULL);
         printf ("Child Complete");
         exit(0);
   }
}
```

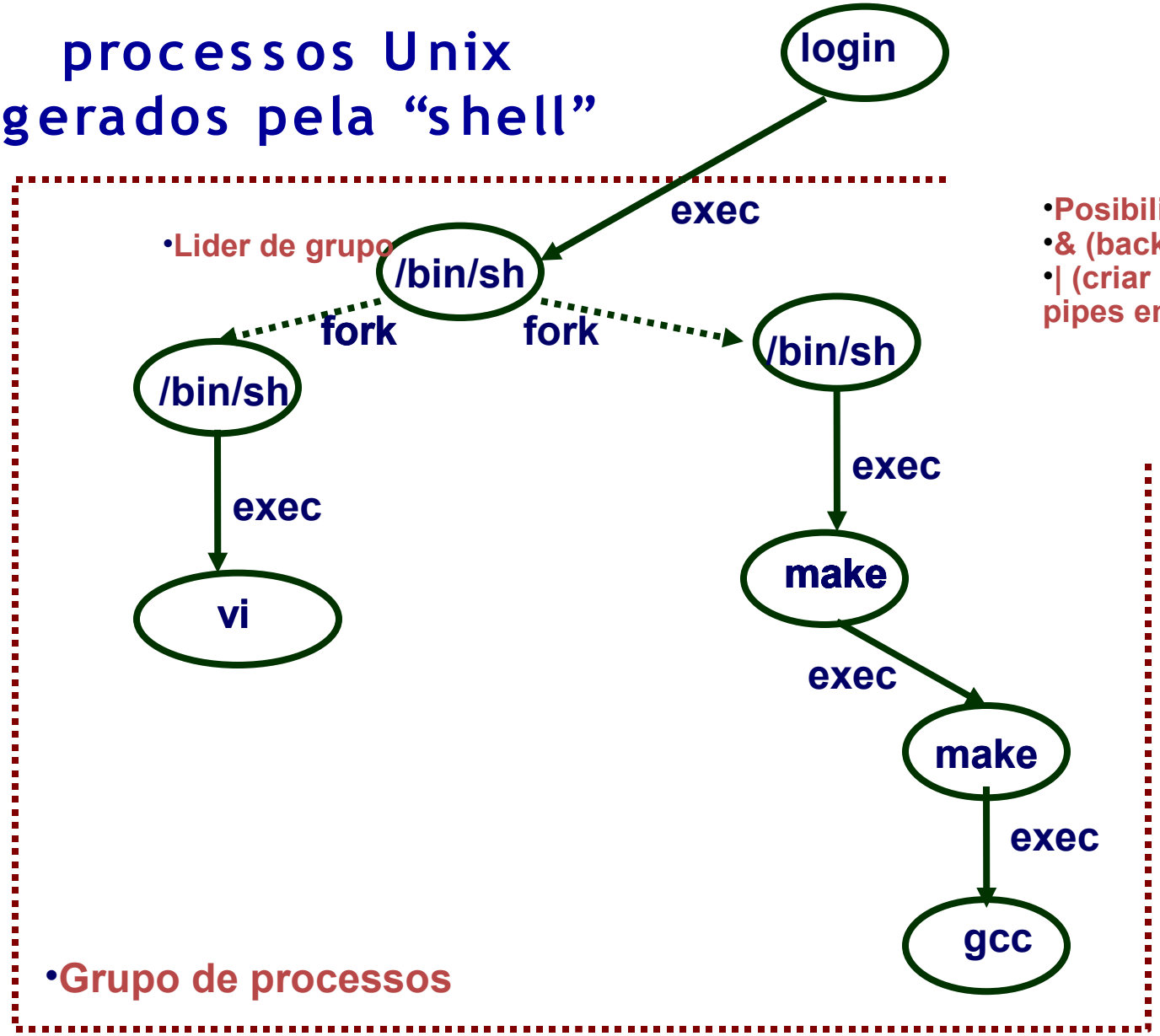
## código exemplo 2 com fork() e wait(...)

```
int main()
{ int r, pid, s;

   r = fork();                  // fork() another process
   if (r < 0) {                       // error occurred
       fprintf(stderr, "Fork Failed\n");
       exit(-1);
   }
   else if (r == 0) {          // child process
           printf("sou o filho com PID: %d, meu pai tem PID: %d\n",
                        getpid(), getppid() );
           // inserir qq código aqui ... inclusive exec
   }
   else {       // processo pai
              printf("sou o pai com PID: %d ", getpid(),
                           "criei um filho com PID: %d\n", r );

                 // inserir qq codigo aqui ...
                 // agora vou esperar meu filho terminar
               pid = wait( &s );
               printf( "meu filho com pid %d terminou com status
   %d\n", pid, s );

              exit(0);
   }
}
```

# Inicio da criação de processos Unix



# processos Unix gerados pela "shell"




•Posibilidades na shell:
•& (background=sem wait)
•| (criar processos com pipes entre eles

# função **wait**

- Processo espera o término dos filhos (fila de evento)
  - Pai é acordado (sai da fila e vai para ready) com término de qualquer filho
  - Pode voltar a dormir se quer esperar todos (controle no número de filhos deve ser feito no código do pai)

```
int wait( int *status )
```

- **Retorna:**
  - PID do filho que terminou
  - Código de status (int) retornado pela função **exit( int status )** executada no filho
- **Pergunta 1)** O que acontece quando pai termina antes dos filhos ?
  - R:
    - filhos trocam de pai para PPID=1 (**init** é pai de todos)
    - Em caso de **exit()** nesse processo → status entregue para processo **init**

# (mais uma pergunta...)

- Processo espera o término dos filhos (fila de evento)
  - Pai é acordado (sai da fila e vai para ready) com término de qualquer filho
  - Pode voltar a dormir se quer esperar todos (controle no número de filhos deve ser feito no código do pai)

```
int wait( int *status )
```

- **Retorna:**
  - PID do filho que terminou
  - Código de status (int) retornado pela função **exit( int status )** executada no filho

- **Pergunta 2)** O que acontece quando não está em **wait** ainda (ou se nunca chama **wait**) ?
  - R:
    - Filho não pode entregar status ao pai
    - Filho vira zombie
    - A maior parte dos recursos do zombie são liberados (ex: espaços de memória)

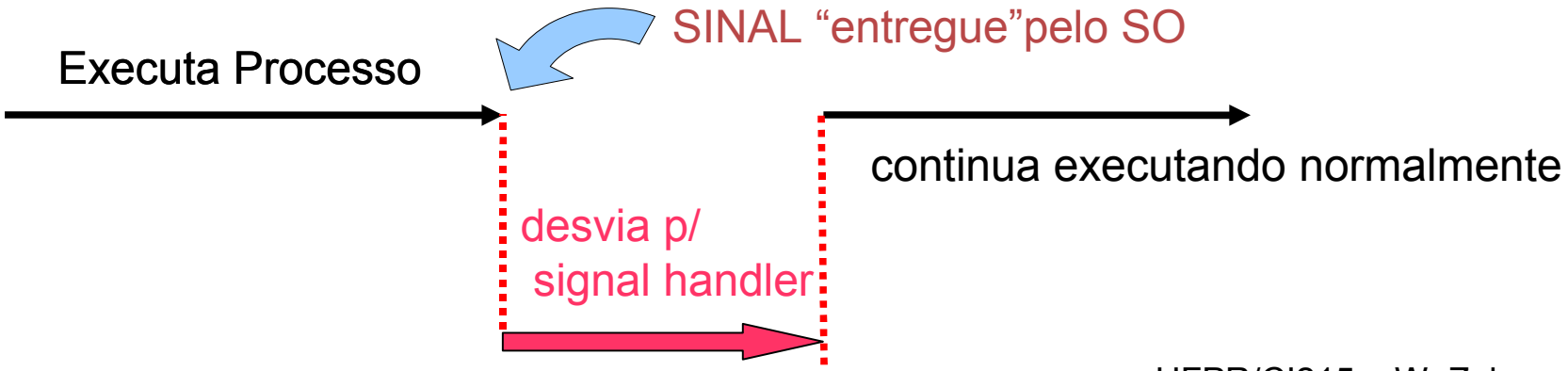
# Sinais (signals) no Unix...
# próxima aula!

# Sinais no Unix (Signals)

- **Sinais são notificações assíncronas entre processos**

- "Interrupções por software"
- cada SIGNAL corresponde a uma constante inteira
- alguns sinais são pré-definidos (função ou uso pré-determinados)
- existem sinais cuja funcionalidade pode ser definida por programas de usuários
- sinais podem vir do mesmo processo ou ser "enviados" de um processo para outro
- é possivel definir procedimentos (funções) em programas que serão executados quando o processo for notificado por algum sinal

-> Signal Handler (tratador de interrupcoes)



# para "instalar" um signal handler em um programa

- usar system call: signal()

```
#include <signal.h>
void trata_sinal( int sig )
{
    signal( SIGINT, trata_sinal ); // reinstala handler
    ...
    printf( "recebi o sinal SIGINT\n" );
    ...
}
main()
{
    ...
    // instala handler para SIGINT pela primeira vez
    signal( SIGINT, trata_sinal );
    ...
}
```

- para enviar SIGNAL para um processo, system call: kill()

```
int kill( int pid, int sig );
```

# sinais enviados por ações de teclado (sistema de I/O)

- **Ctrl-C**
- **apertando essa tecla causa envio de (SIGINT) para o processo corrente**

- **Ctrl-Z**
- **apertando essa tecla causa envio de (SIGTSTP) para o processo corrente**
-
- **Ctrl-**
- **apertando essa tecla causa envio de (SIGABRT) para o processo corrente**

**constantes são definidas em <signal.h> e incluidas em programas**

**comando kill:**

**chama a funcao kill para enviar sinais**

**exemplos: kill -9 3031**

**ou kill -SIGKILL 3031**

# bloqueios (filtros) de sinais

- pode-se controlar que sinais um processo irá receber
- habilitar / desabilitar recepcao de sinais
- alguns sinais não podem ser bloqueados
- usar system call: signal()

```
// BSD signal API (disponiveis em Linux, etc)
 sigvec, sigblock, sigsetmask, siggetmask, sigmask

// Linux
 sigaction(2), sigpending(2), sigprocmask(2),  sigqueue(2),
 sigsuspend(2),  bsd_signal(3),   sigsetops(3),  sigvec(3),
 sysv_signal(3),
```

# Standard Signals (de: man 7 signal)

Linux supports the standard signals listed below ...(OBS: term action is terminate)

| Signal | Value | Action | Comment |
|---|---|---|---|
| SIGHUP | 1 | Term | Hangup detected on controlling terminal or death of controlling process |
| SIGINT | 2 | Term | Interrupt from keyboard |
| SIGQUIT | 3 | Core | Quit from keyboard |
| SIGILL | 4 | Core | Illegal Instruction |
| SIGABRT | 6 | Core | Abort signal from abort(3) |
| SIGFPE | 8 | Core | Floating point exception |
| SIGKILL | 9 | Term | Kill signal |
| SIGSEGV | 11 | Core | Invalid memory reference |
| SIGPIPE | 13 | Term | Broken pipe: write to pipe with no readers |
| SIGALRM | 14 | Term | Timer signal from alarm(2) |
| SIGTERM | 15 | Term | Termination signal |
| SIGUSR1 | 30,10,16 | Term | User-defined signal 1 |
| SIGUSR2 | 31,12,17 | Term | User-defined signal 2 |
| SIGCHLD | 20,17,18 | Ign | Child stopped or terminated |
| SIGCONT | 19,18,25 | Cont | Continue if stopped |
| SIGSTOP | 17,19,23 | Stop | Stop process |
| SIGTSTP | 18,20,24 | Stop | Stop typed at tty |
| SIGTTIN | 21,21,26 | Stop | tty input for background process |
| SIGTTOU | 22,22,27 | Stop | tty output for background process |

The signals SIGKILL and SIGSTOP cannot be caught, blocked, or ignored. .... more ...

# comentários sobre o trabalho 1