Processos

processos UNIX e Linux

threads em Linux

Taisy Weber

Ambiente UNIX

Operating System Concepts - Silberschatz & Galvin, 1998

Matthew & Stones, cap 10

Mitchell, cap 3

- ✓ Processos:
  - ✓ revisão de conceitos básicos
    - processos no SO UNIX
  - ✓ programação
    - criação (exec, fork, etc), sincronização (wait), eliminação, processos zombie
    - signals
  - ✓ conceito de threads, threads em Linux

Processo

- ✓ sistema de batch — job
- ✓ sistema time-shared — task
- ✓ PCs — programa

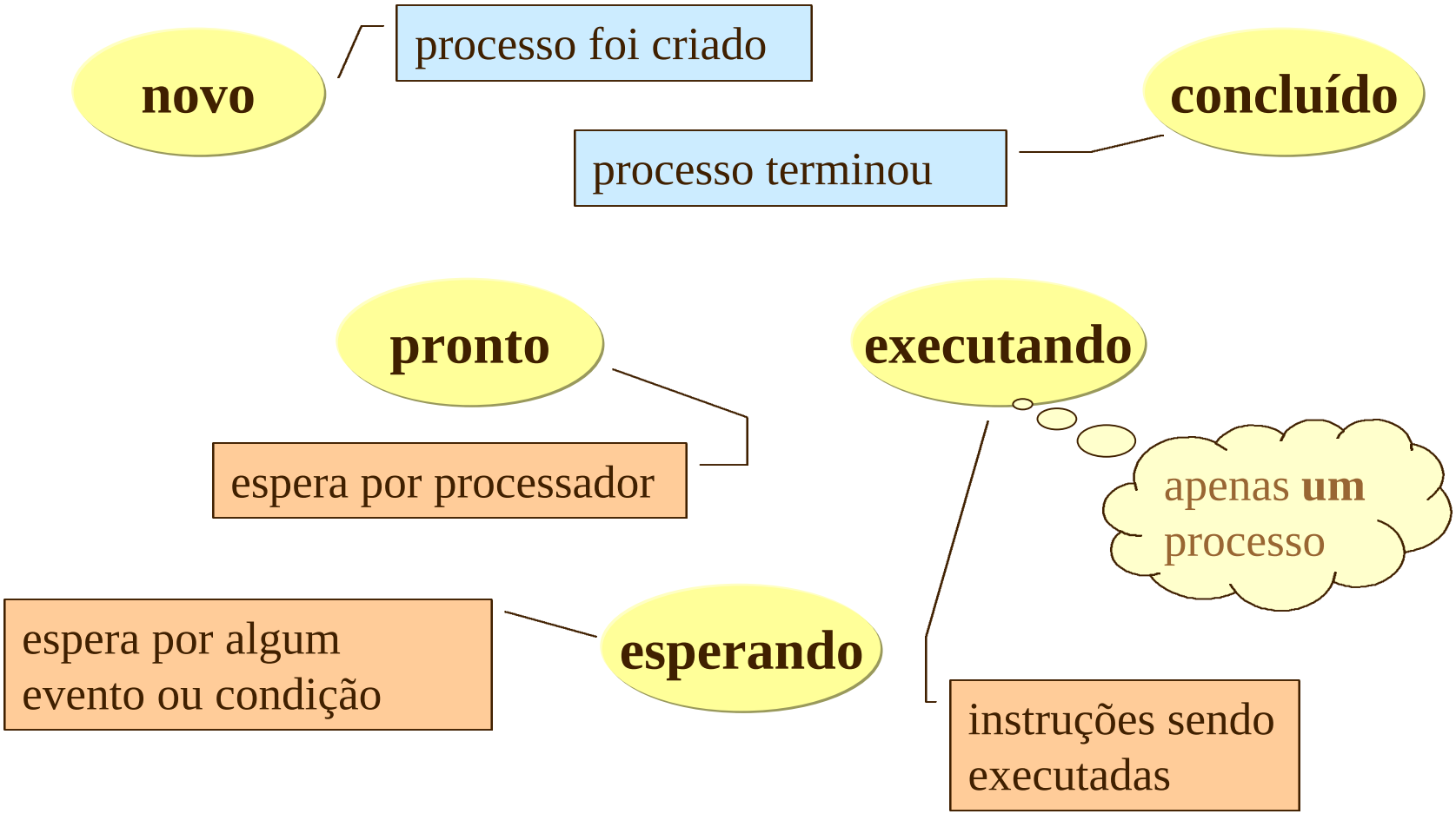
programa em execução

código, IP, pilha, área de dados

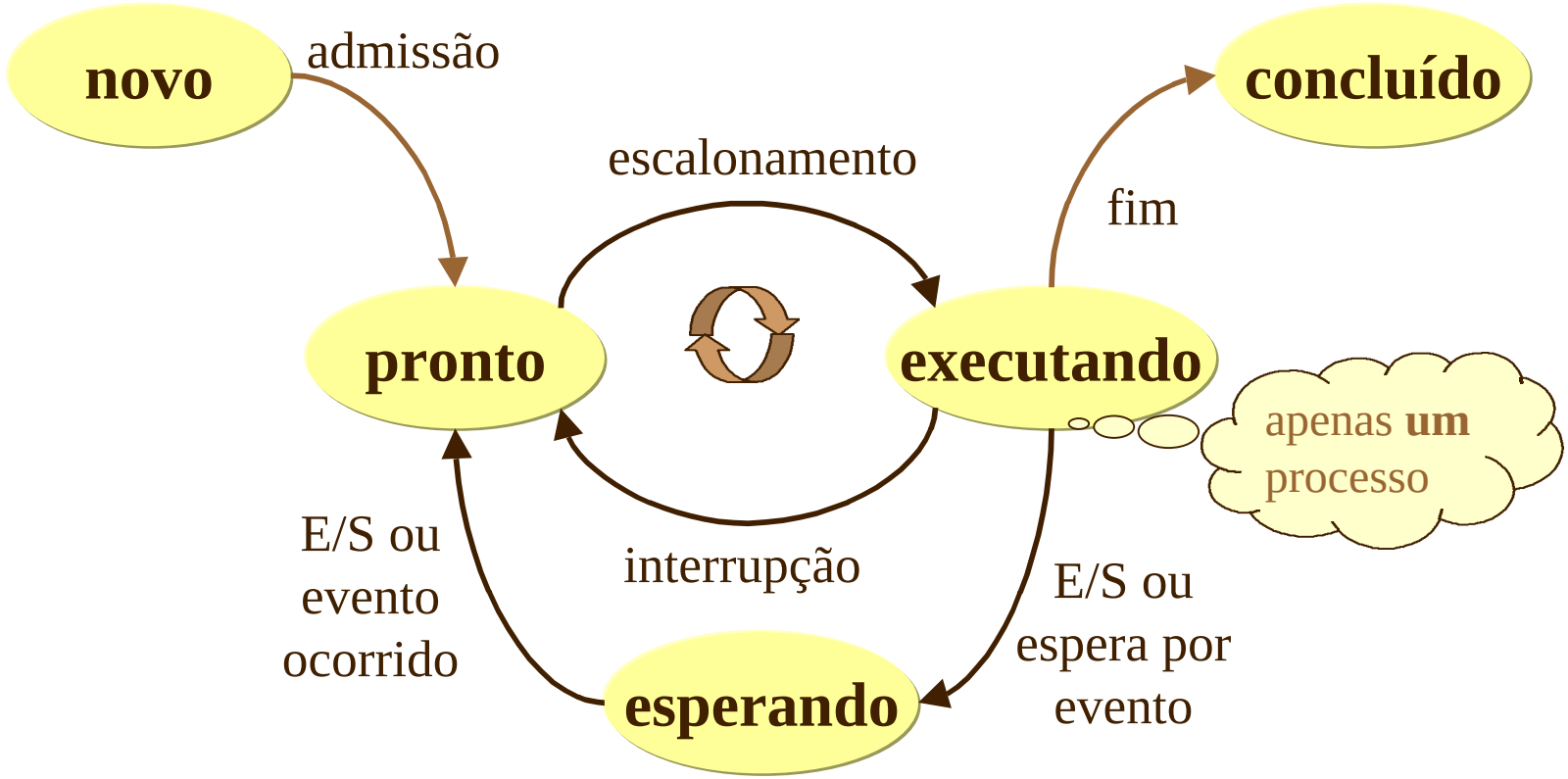
vários processos podem corresponder a um mesmo programa

processos são ativos, programas são passivos

Estados de um processo

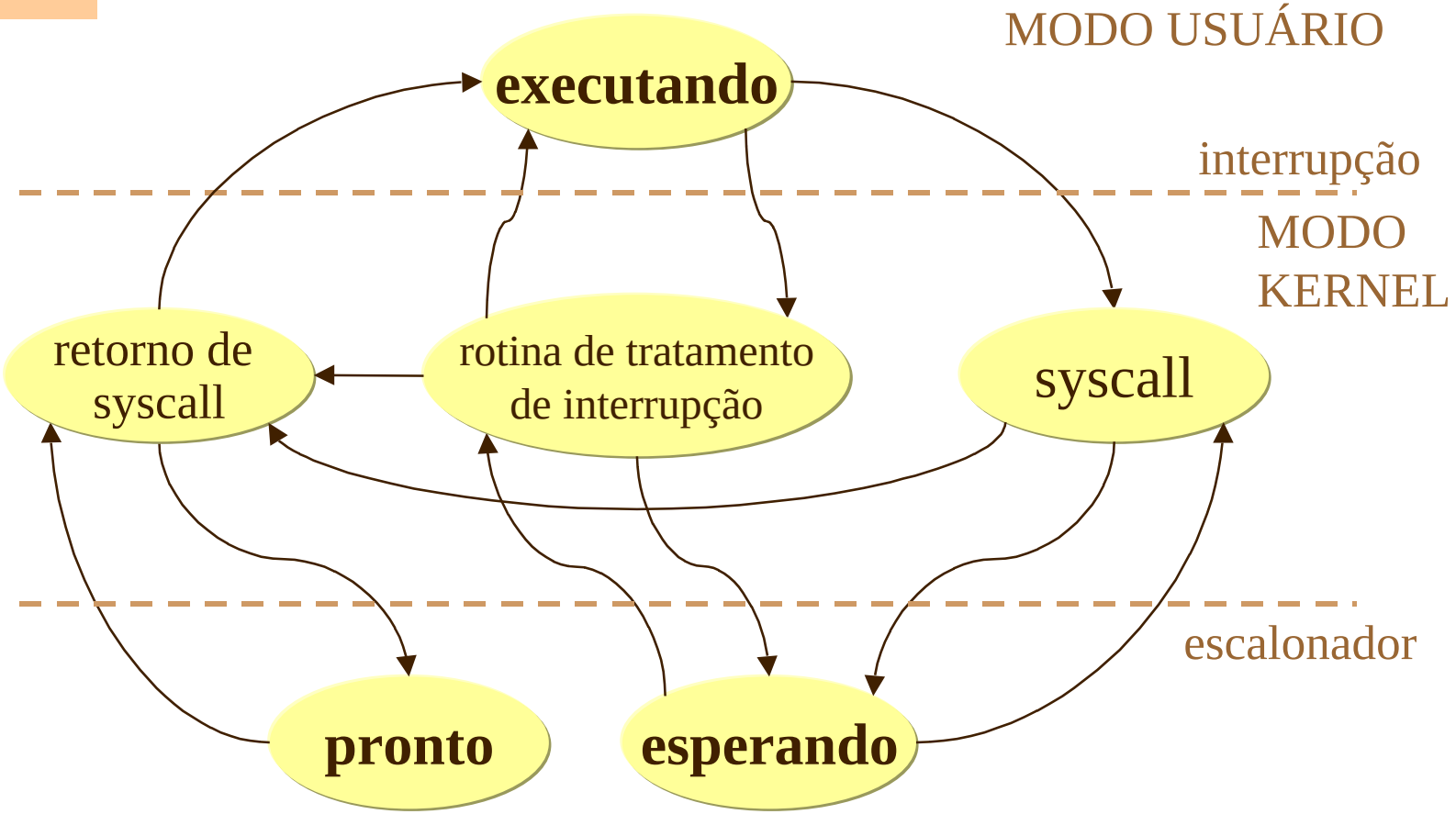


Estados de um processo



Estados de um processo Linux

Beck



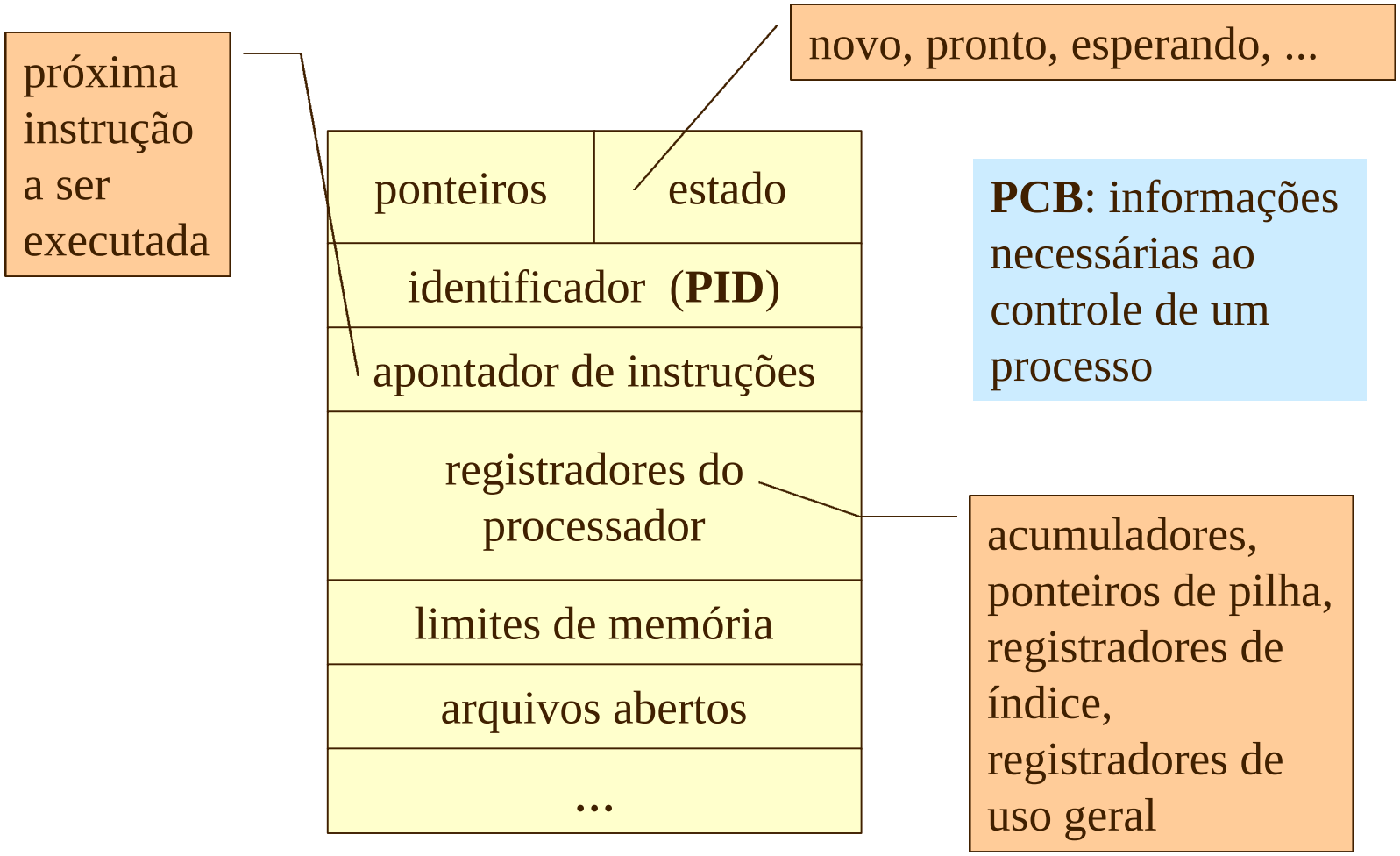
a implementação é diferente

Processo executando no Linux

- 2 fases de um processo
  - processo em modo usuário (*user mode*)
    - a maior parte do trabalho é executado nesse modo
  - processo em modo kernel
    - quando uma system call está em execução
    - no modo kernel é usada outra pilha (kernel stack) para o processo

quando não está executando syscalls (por exemplo no atendimento de interrupção) o kernel usa outra pilha, exclusiva do kernel

Bloco de controle de um processo



Bloco de controle do processo

representação interna

contém toda a informação que o SO precisa para o controle do processo

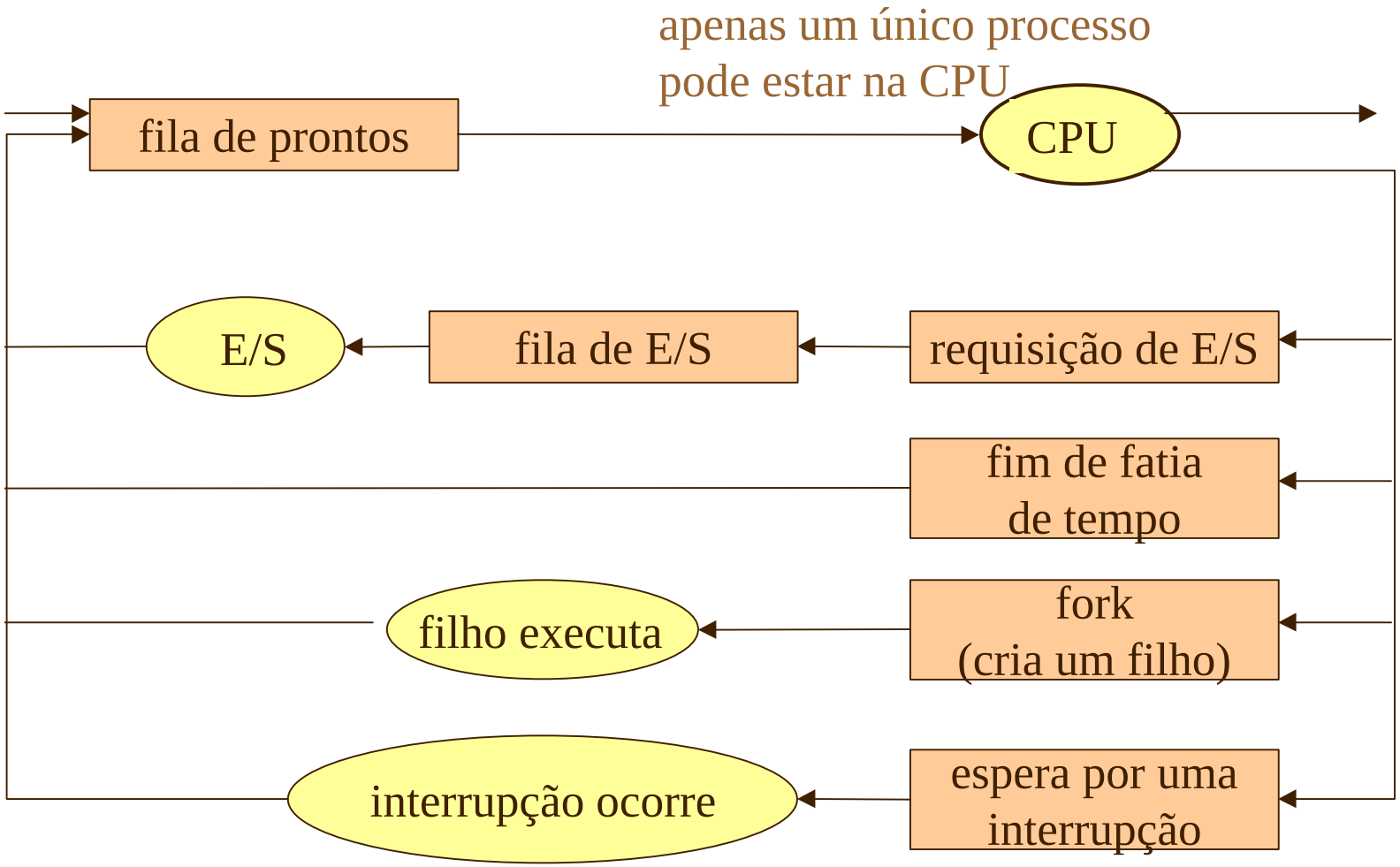
- ✓ estrutura de dados da tarefa

task structure

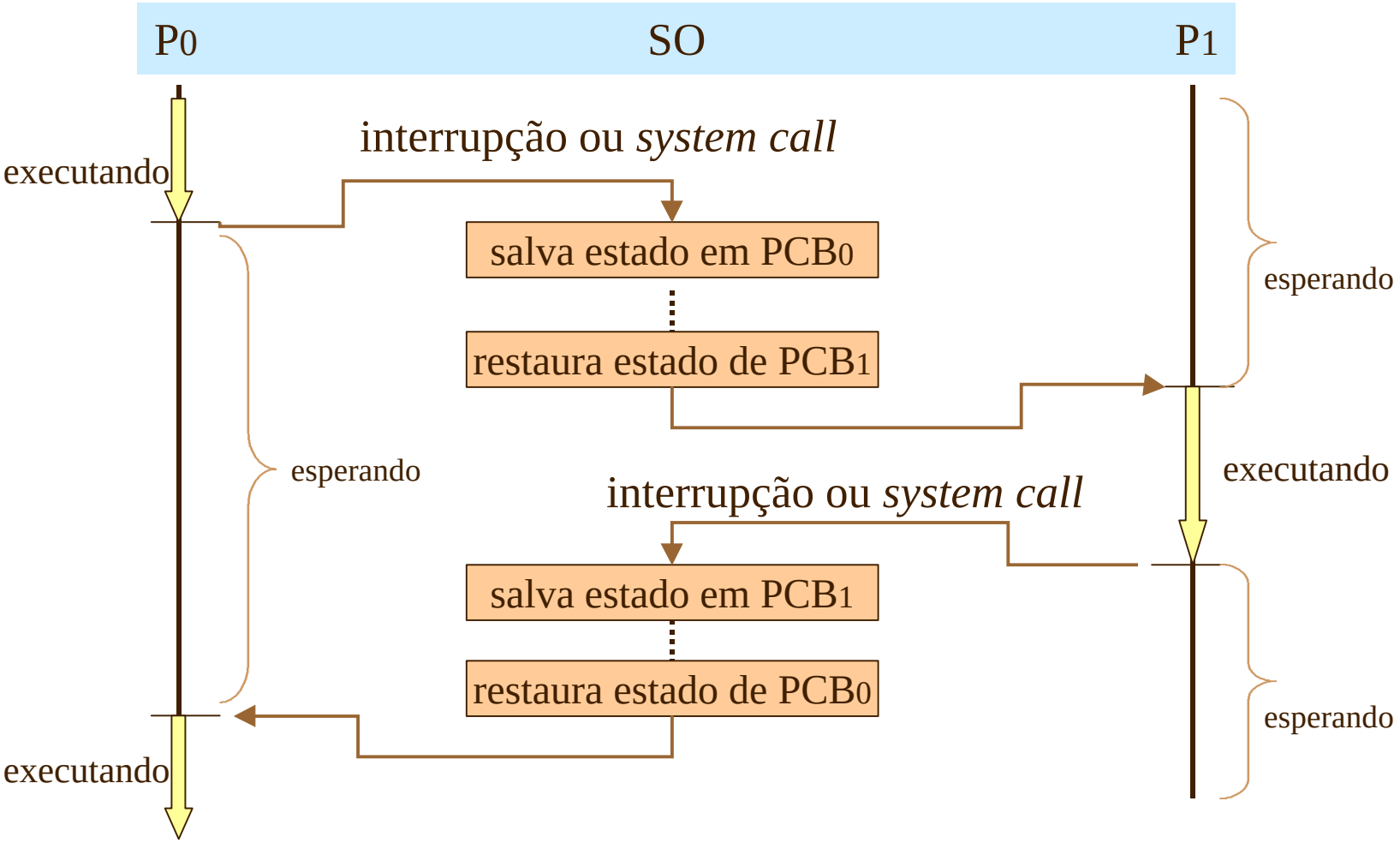
  - identificador do processo
  - estado
  - registradores
  - informação para escalonamento
  - ponteiros para outros processos
  - ponteiros para a *page table* do processo
  - *current directory*
  - arquivos abertos

Filas

um processo percorre várias filas durante sua vida útil



Chaveamento entre processos



Escalonamento

- não preemptivo
  - o processo fica na CPU até liberá-la **voluntariamente**
- preemptivo
  - a CPU pode ser retirada de um processo

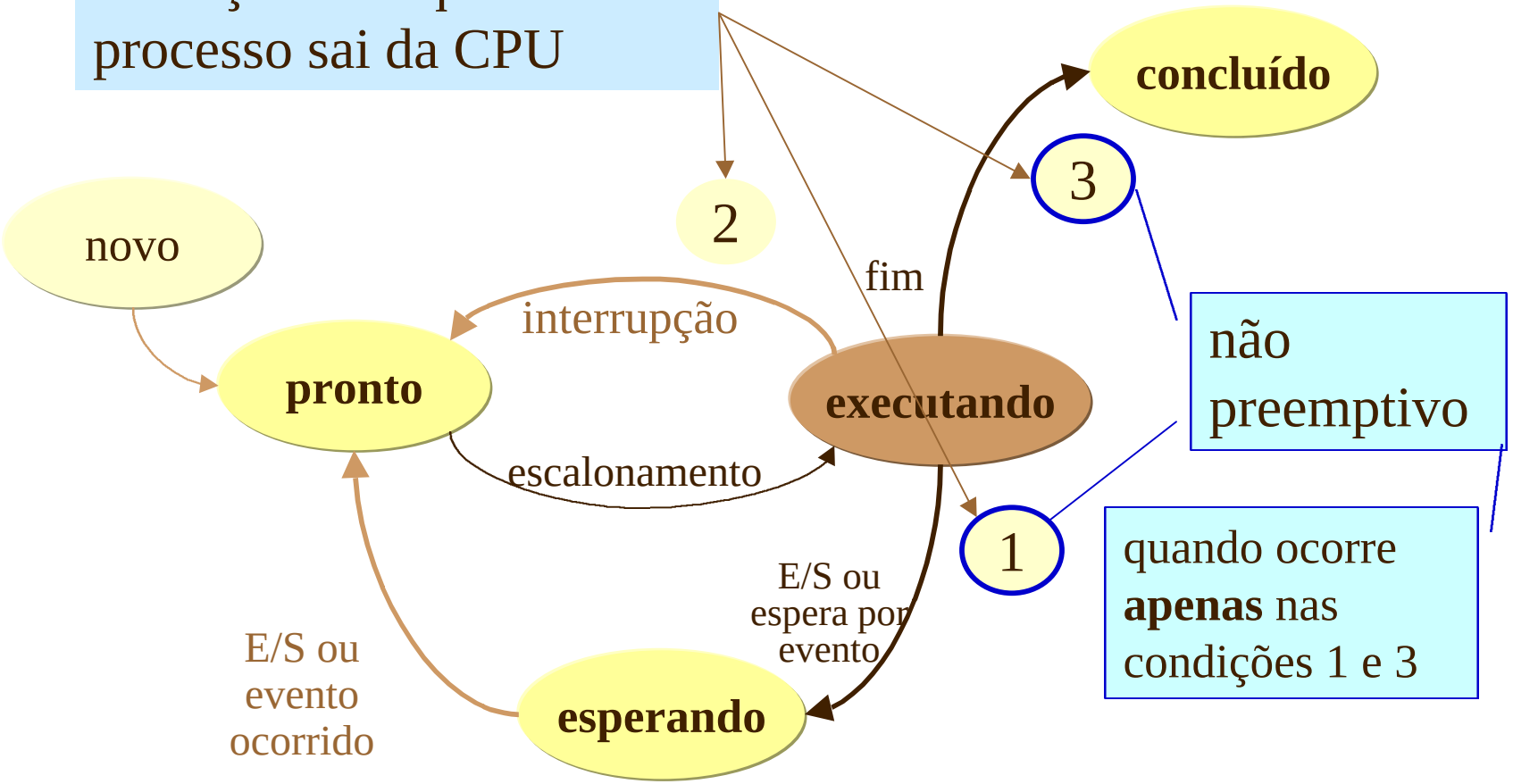
lembrar que um único processo pode estar ocupando a CPU

por interrupção de um contador (**timer**)

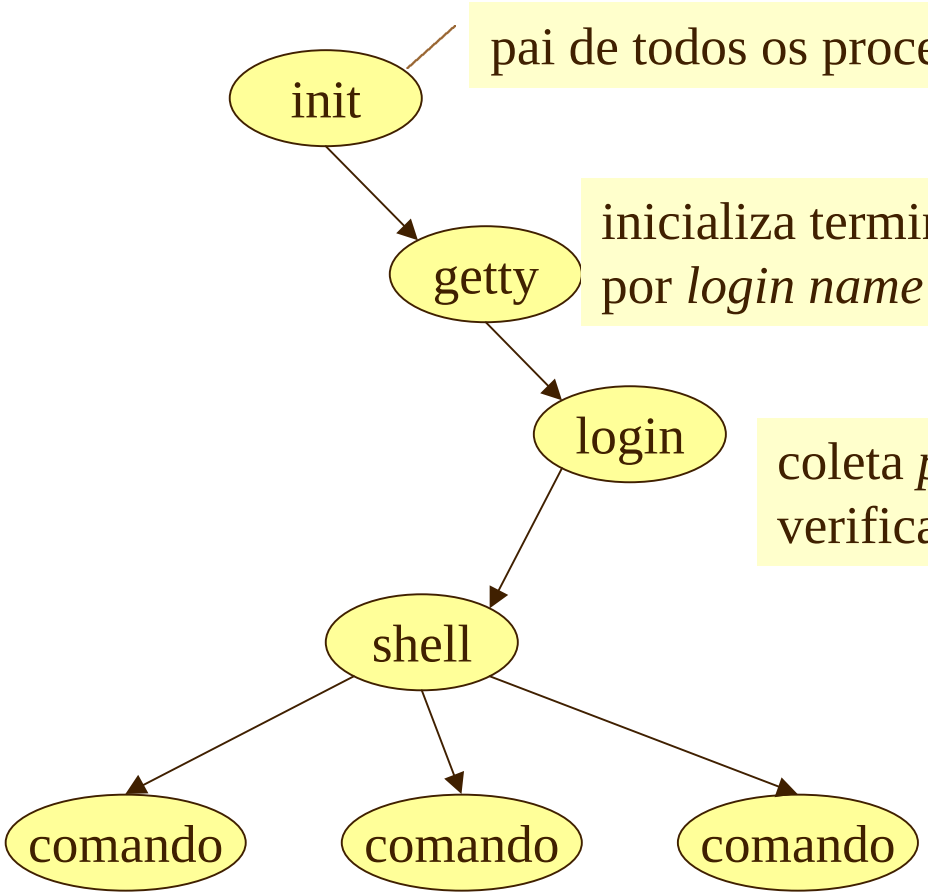
problemas de coordenação entre processos no uso de memória compartilhada

Escalonamento não preemptivo

condições em que um processo sai da CPU



Processos iniciais UNIX



pai de todos os processos

PID = 1

inicializa terminal e espera por *login name* do usuário

um para cada *user*

coleta *password*, encripta e verifica

inicia uma *shell* para o usuário

a habilidade de um processo iniciar outro processo é fundamental em UNIX

PID

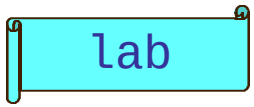
- ✓ **p**rocess **id**entifier
  - inteiro positivo entre 2 e 32768
  - PID=1 reservado para o processo `init`
  - cada novo processo recebe o próximo inteiro disponível na seqüência

comando **ps**

mostra os processos carregados do usuário, os processos de outro usuário e/ou todos os processos do sistema

```
$ ps -af
```

a - all
f - full information



Comando ps

- ✓ tabela de processos
  - descreve todos os processos carregados
    - processos são localizados pelo seu PID
    - tabela é limitada

PID é o índice da tabela

atualmente pela quantidade de memória disponível

- ✓ `ps`
  - mostra os processos carregados na memória
    - status S ou R

ready ou waiting

R - *ready* (fila de prontos)

campo STAT na listagem de `ps -ax`

Função `system`

- ✓ permite iniciar um processo dentro de um outro processo
  - pouco usado
  - `system` executa o comando passado como *string* e espera que o comando complete sua execução

invoca uma shell para executar o comando

```
#include <stdlib.h>
int system(const char *string)
```

equivale a:
`$ sh -c string`

retorna: 127 se uma shell não pode ser iniciada
-1 se outro erro
ou código de saída (*exit code*) do comando

Função `system`: exemplo

```
#include <stdlib.h>
int system(const char *string)
```

```
#include <stdlib.h>
#include <stdio.h>

int main()
{
    printf("Running ps with system\n");
    system("ps -ax");
    printf("Done.\n");
    exit(0);
}
```

fonte: `system.c`

verificar o que ocorre executando o programa `$ ./system`

lab

Função `system`: segundo exemplo

fonte: `system2.c`

```
#include <stdlib.h>
#include <stdio.h>

int main()
{
    printf("Running ps with system\n");
    system("ps -ax &");
    printf("Done.\n");
    exit(0);
}
```

& coloca o comando `ps` em background

```
$ ./system2
```

o programa termina antes do comando `ps`

`lab`

Substituindo a imagem do processo

- exec
  - família de funções
    - **execl**
    - **execlp**
    - **execle**
    - **execv**
    - **execvp**
    - **execve**

substitui o processo atual por um novo código cujo arquivo ou *path* é passado como parâmetro

terminando com p: usam variável de ambiente PATH

dois grupos:

- as primeiras 3 execs apresentam um número variável de argumentos terminando com um null pointer;
- as 3 últimas possuem como segundo argumento um *array* de *strings*

execs

p: execs com p usam variável de ambiente PATH para achar o arquivo executável do novo programa

```
#include <unistd.h>
char **environ;
```

variável global para passar um valor ao ambiente do novo programa

UNIX environment
Matthew & Stones, cap 4

```
int execl (const char *path,
           const char *arg0,. . .,(char *)0);
int execlp(const char *file,
           const char *arg0,. . .,(char *)0);
int execle(const char *path,
           const char *arg0,. . .,(char *)0,
           const char *envp[]);

int execv (const char *path, const char *argv[]);
int execvp(const char *file,
           const char *argv[]);
int execve(const char *path,
           const char *argv[, const char *envp[]);
```

array de *strings* para ser usado como novo ambiente

Exemplo: execlp

fonte: pexec.c

```
#include <unistd.h>
#include <stdio.h>

int main()
{
    printf("Running ps with execlp\n");
    execlp("ps", "ps", "-ax", 0);
    printf("Done.\n");
    exit(0);
}
```

Done. nunca será impresso

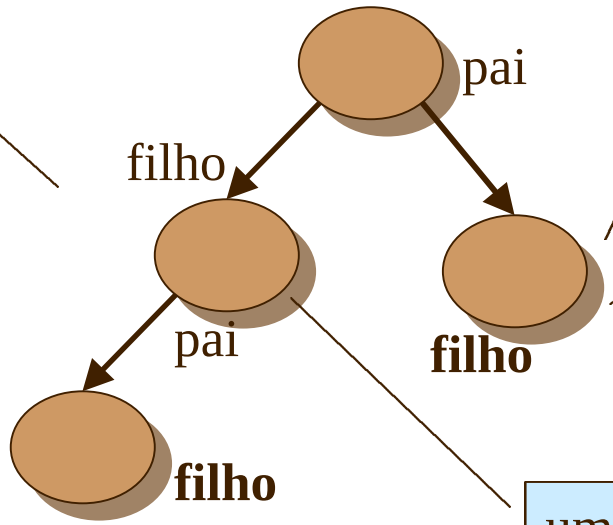
```
$ ./pexec
```

lab

Criação de processos: árvore

um processo pode criar outros processos

pode receber dados inicializados do pai



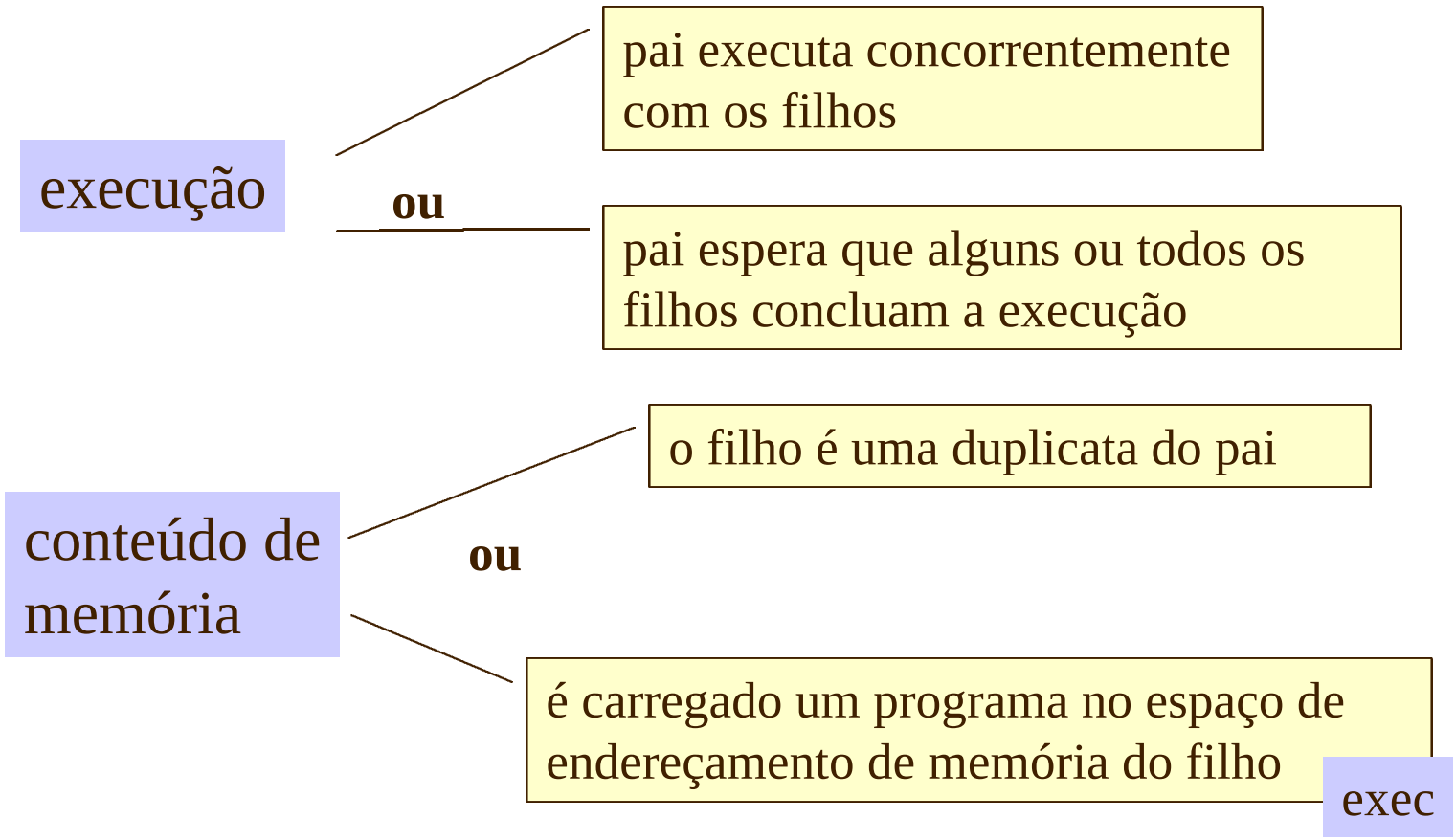
pode obter recursos diretamente do SO

ou

pode compartilhar apenas parte dos recursos do pai

um filho pode ser pai de outros processos

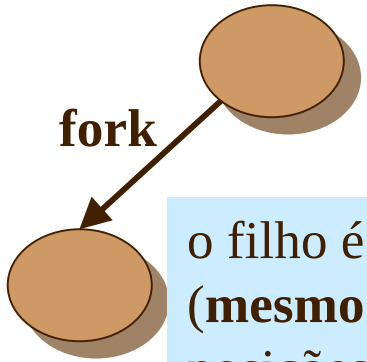
Processos pais e filhos



Criação: syscall `fork`

pai executa **fork**

permite ao pai se comunicar facilmente com o filho



o filho é uma **cópia** do pai (**mesmo conteúdo** mas em posições diferentes de memória)

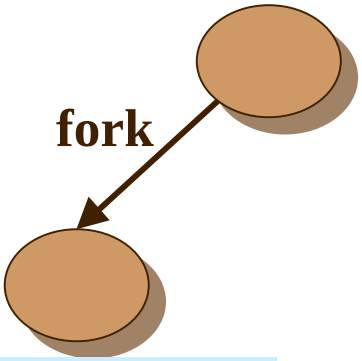
os dois processos continuam a execução após o **fork** (na instrução seguinte ao **fork** em cada processo)

todos os arquivos abertos antes do **fork** continuam abertos

o filho recebe 0 e o pai recebe o **process id** do filho

... syscall `fork`

pai executa **fork**



o filho é uma **cópia** do pai

o **fork** dá ao filho um novo identificador de processo PID (que é informado ao pai)

memória principal é alocada para **dados** e **pilha**

uma nova *page table* é construída

geralmente não é necessário copiar o código (**text**) pois os processos compartilham o mesmo código

são preservados arquivos abertos, identificadores de user e group, manipuladores de signals

Criação: função `fork`

- ✓ função fork corresponde a chamada de sistema para criação de novos processos

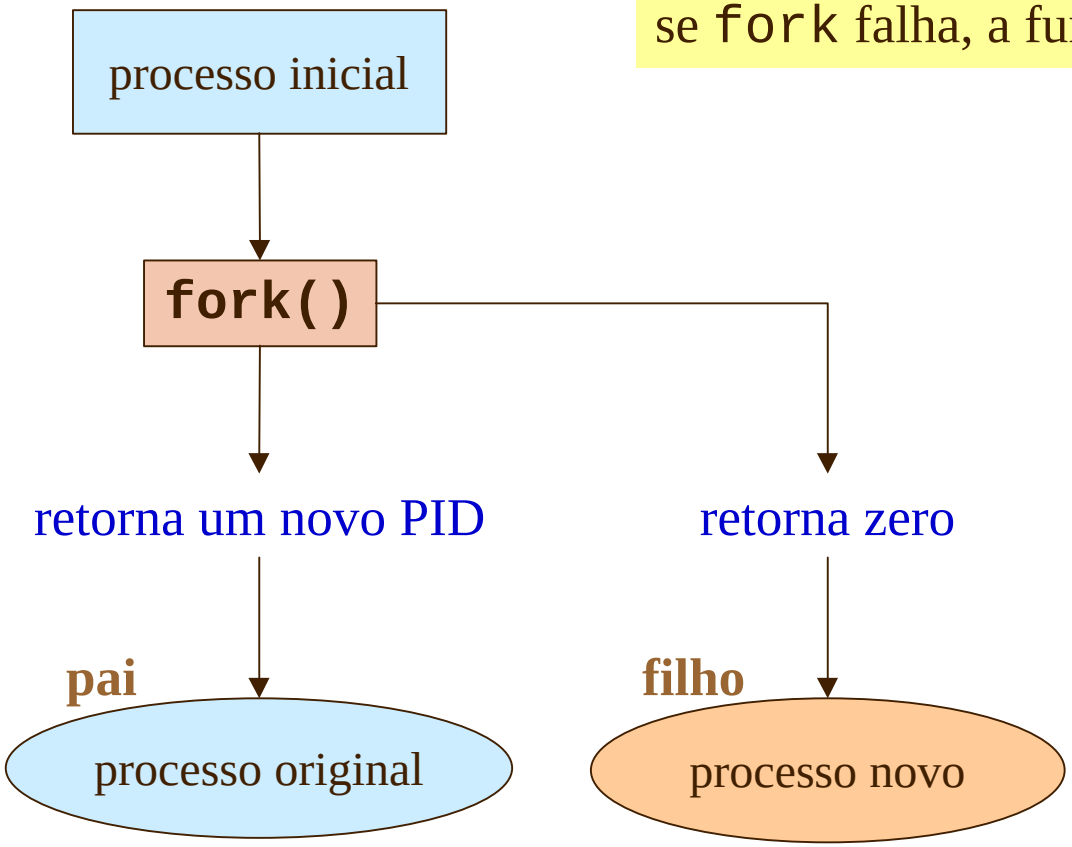
```
#include <sys/types.h>
#include <unistd.h>

pid_t fork(void);
```

exemplo

```
pid_t newpid;
newpid = fork();
switch(newpid) {
    case -1:  /* fork failed */
        break;
    case 0:   /* child */
        break;
    default:  /* parent  */
        break;
    }
```

... função `fork`

se `fork` falha, a função retorna -1



Exemplo: `fork`

fonte: `fork.c`

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main()
{
    pid_t pid;
    char *message;
    int n;

    printf("fork program starting\n");
    pid = fork();
```

lab

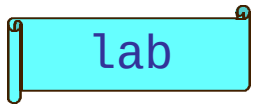
Exemplo: `fork`

```
switch(pid) {
    case -1:
        perror("fork failed");
        exit(1);
    case 0:
        message = "This is the child";
        n = 5; break;
    default:
        message = "This is the parent";
        n = 3; break;
    }
```

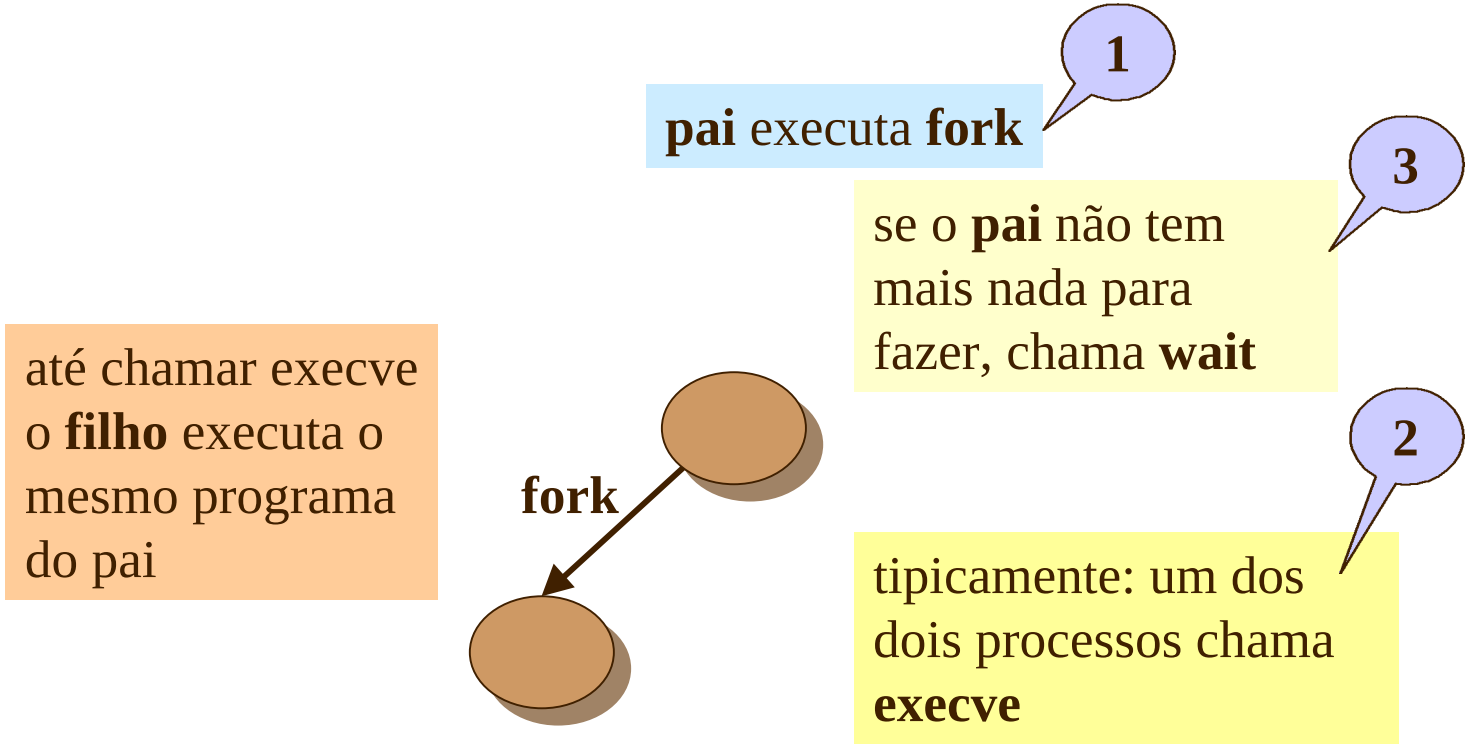
```
for(; n > 0; n--) {
        puts(message); sleep(1);
    }
    exit(0);
}
```

```
$ ./fork
```

Determine qual a saída.



fork & exec

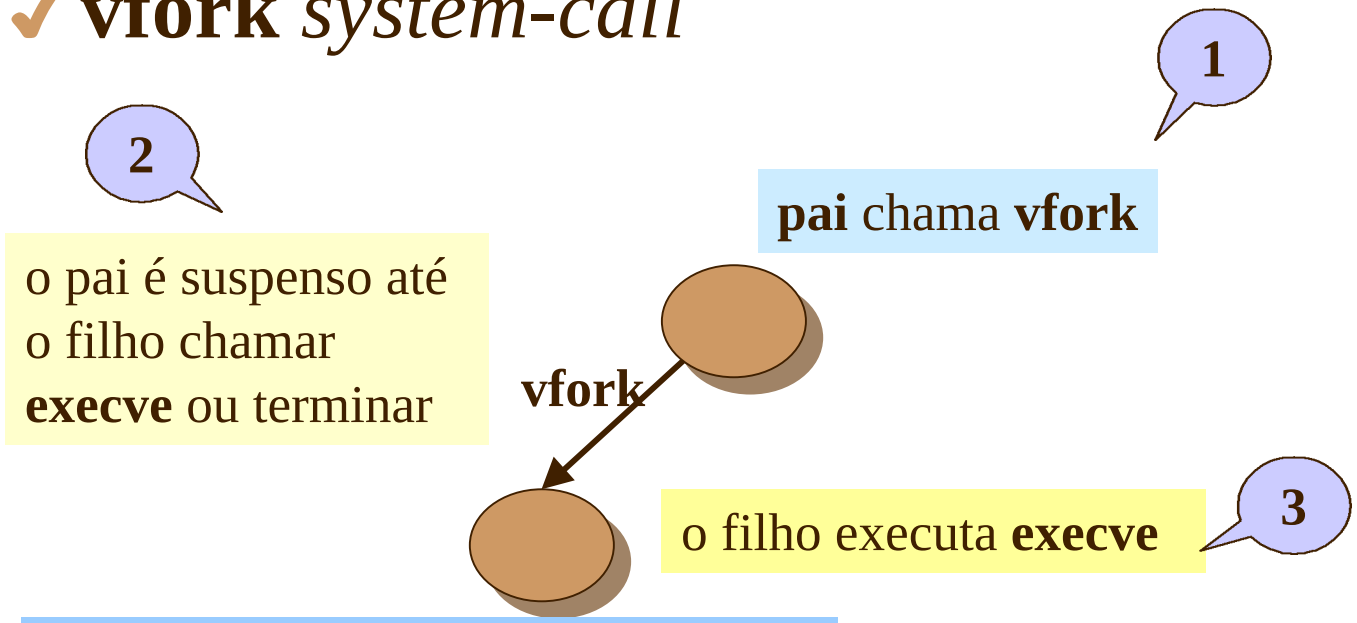


execve: carrega um arquivo binário na memória do processo (destruindo a imagem de memória do processo que chamou execve)

execve é um dos mais usados, mas qualquer exec serve

vfork

- **vfork** *system-call*



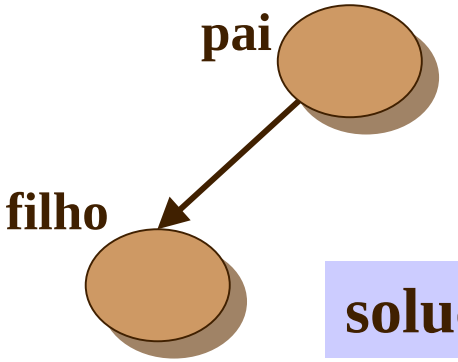
vfork: não copia dados e pilha para o novo processo, o filho permanece **compartilhando** as páginas do pai

estratégia perigosa - o filho pode alterar os dados do pai antes de chamar execve

fork e vfork no UNIX

vfork é **perigosa**: **compartilhamento** da mesma área de dados e pilha

fork é **ineficiente**: envolve **duplicação** de área de dados e pilha



solução: copy-on-write

Linux

sem duplicação inicial

memória do pai é marcada **read-only**; quando o primeiro processo tentar escrever, a página de memória em questão é duplicada

Esperando por um filho: `wait`

processo filho criado por fork roda independente do pai e evtl termina

- ✓ e o pai?
  - às vezes o pai quer saber o que o filho anda fazendo

por exemplo, se o filho já terminou suas tarefas

- ✓ wait syscall
  - o pai fica esperando até o filho terminar (ou até um de seus filhos terminar)

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *stat_loc);
```

permite determinar o status de terminação do filho

info de status definido em `sys/wait.h`

Término de processo ...

- ✓ um processo termina quando:
  - executa a última instrução
  - e pede para o SO para sair do sistema

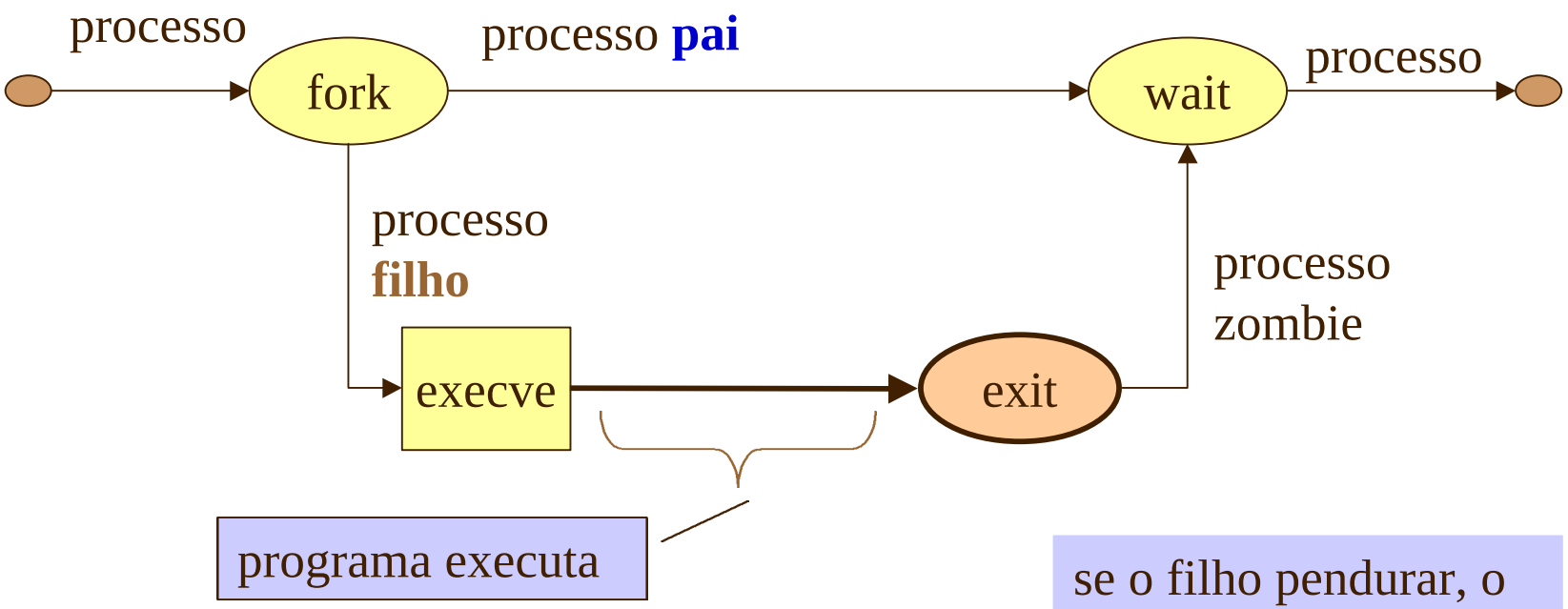
exit system call

- ✓ outras circunstâncias:
  - ✓ um processo aborta outro processo
    - geralmente apenas os pais podem abortar os filhos

kill system call

um processo não deve poder arbitrariamente matar outros processos que estão em execução

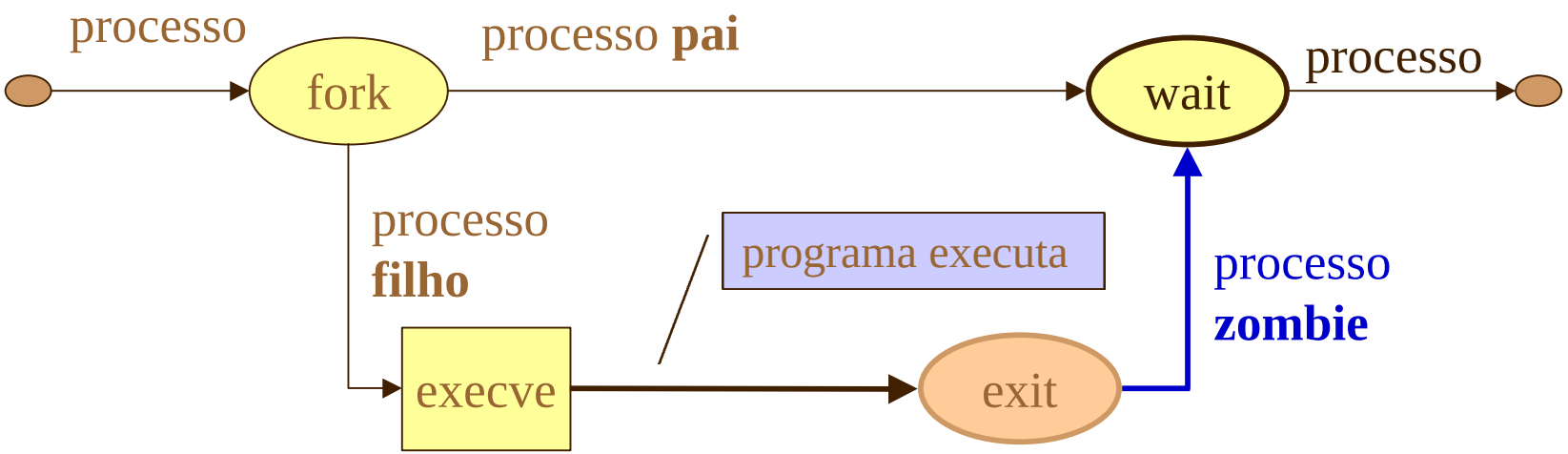
Término de processo filho



se o filho pendurar, o sistema emula um exit

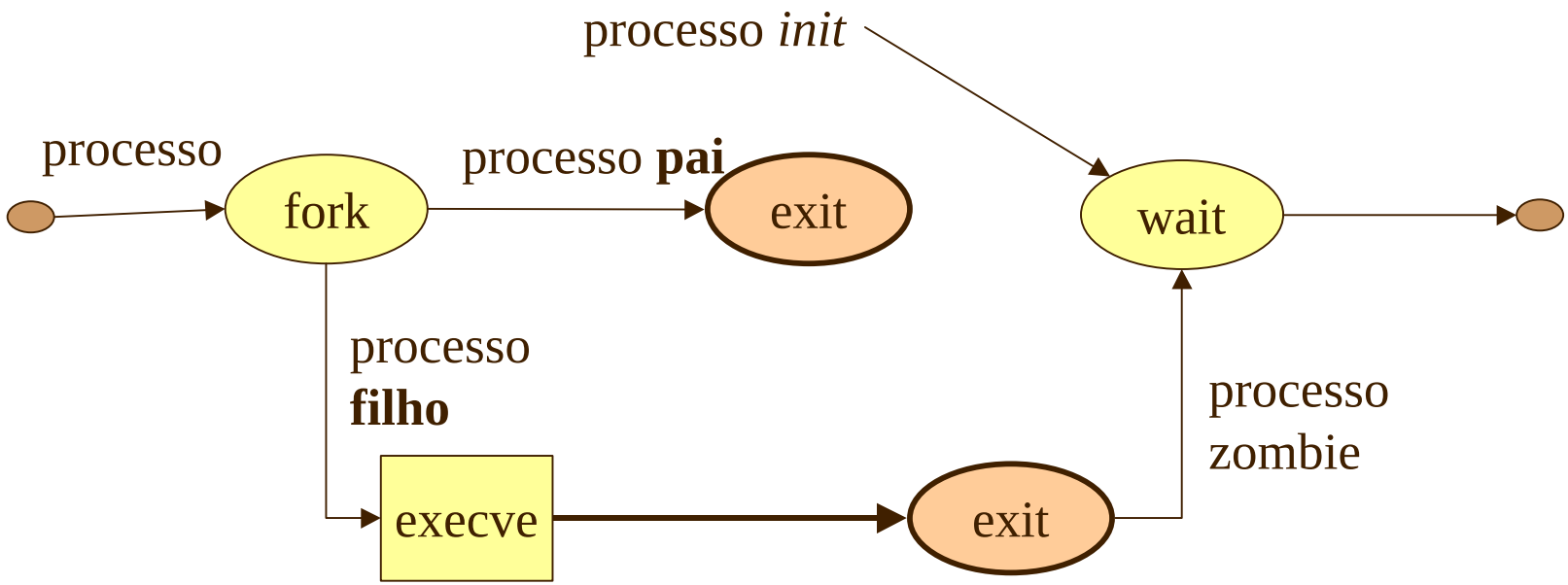
o pai (que chamou **wait**) recebe o **process id** do filho (que chamou **exit**)

Zombie



A associação de um filho com seu processo pai permanece, mesmo após o seu término, até que o pai termine ou chame `wait`. O filho não está mais ativo, processo **zombie**. Seu código de saída (exit code) está armazenado e pode ser usado pelo pai num `wait` posterior.

Término de processo pai



quando o pai termina antes do filho, o *zombie* é adotado pelo processo *init*

init é o pai de todos os processos de usuário

Terminação forçada

- ✓ razões para aborto de um filho
  - o filho excedeu a permissão de uso de determinado recursos
  - a tarefa assinalada ao filho não é mais necessária
  - o pai está saindo do sistema

o **SO** geralmente **não** permite que filhos permaneçam executando após a terminação do processo pai

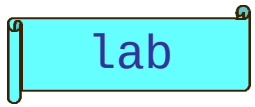
Exemplo: wait

semelhante ao programa fork

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>

int main()
{
    pid_t pid;
    char *message;
    int n;
    int exit_code;

    printf("fork pgr starting\n");
    pid = fork();
```

executar esse programa

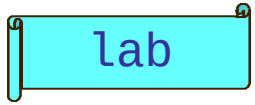
```
switch(pid)
    {
    case -1:
        exit(1);
    case 0:
        message = "This is the child";
        n = 5;
        exit_code = 37;
        break;
    default:
        message = "This is the parent";
        n = 3;
        exit_code = 0;
        break;
    }

    for(; n > 0; n--) {
        puts(message);
        sleep(1);
    }
```



Continua exemplo wait

essa parte do programa espera para o processo filho terminar

```
/*  This section of the program waits for the child process to finish.  */

    if(pid) {
        int stat_val;
        pid_t child_pid;

        child_pid = wait(&stat_val);

        printf("Child has finished: PID = %d\n", child_pid);
        if(WIFEXITED(stat_val))
            printf("Child exited with code %d\n", WEXITSTATUS(stat_val));
        else
            printf("Child terminated abnormally\n");
    }
    exit (exit_code);
}
```



Redirecionamento

- ✓ recursos comuns entre pais e filhos
  - descritores de arquivos são preservados

todos os arquivos abertos antes do **fork** continuam abertos após o **exec**

exemplo de filtro pg 361

Matthew & Stones, cap 10

Signal

header `signal.h`

- ✓ evento gerado:

instrução ilegal, violação de segmento de memória, erro de FP

  - pelo sistema em resposta a alguma condição de erro
  - por um outro processo `kill`
    - e passado explicitamente para o processo

pode ser considerado uma mensagem especial

- ✓ processo reage a recepção do sinal **executando alguma ação** ou **ignorando-o**
    - o processamento normal é interrompido (em qualquer lugar, a qualquer momento)
    - semelhante a interrupção de software

Comentários

- ✓ sinais podem ser perdidos
  - se um segundo sinal do mesmo tipo ocorrer antes que o processo capturar o primeiro, o primeiro será sobreescrito
- ✓ não existe prioridade entre os sinais
  - se dois sinais forem enviados a um processo ao mesmo tempo, é indeterminada a ordem de recebimento dos sinais
- ✓ sinais podem conduzir a condições ambíguas (*race conditions)*
  - 4.3 BSD introduziu padrão mais confiável para manipulação de sinais

função **sigaction** é mais robusta que a função **signal**

Exemplos de signals

nome de *signal* sempre começa com SIG

- SIGILL
  - produzido por instrução ilegal
- SIGINT

mata o processo (a menos que ele trate o sinal)

  - gerado pelo terminal teclas *Ctrl-C*
  - enviado ao processo em *foreground*
- SIGHUP

comando **kill** pode enviar um *signal* qualquer (por exemplo SIGHUP) a um dado processo conhecendo seu **pid**

  - `kill -HUP 512`

Exemplos de sinais

- ✓ SIGABORT
- ✓ SIGALRM
- ✓ SIGFPE
- ✓ SIGHUP
- ✓ SIGILL
- ✓ SIGINT
- ✓ SIGKILL
- ✓ SIGPIPE
- ✓ SIGQUIT
- ✓ SIGSEGV
- ✓ SIGTERM
- ✓ SIGUSR1
- ✓ SIGUSR2

definidos pelo usuário

não pode ser capturado ou ignorado

recebendo qualquer desses sinais o processo termina imediatamente, a menos que tenha providenciado a captura do sinal (exceto SIGKILL)

Outros exemplos de sinais

- ✓ SIGCHLD — ignorado por default
- ✓ SIGCONT — o processo parado volta a executar
- ✓ SIGSTOP — não pode ser capturado ou ignorado
- ✓ SIGTSTP
- ✓ SIGTIN
- ✓ SIGTOUT

recebendo qualquer desses sinais o processo para (stops)

raramente usados em programas
muito usados em shell

signal library function

```
#include <signal.h>
void (*signal(int sig, void (*func(int)))(int);
```

- ✓ parâmetros de `signal`
  - **`sig`**: sinal a ser capturado ou ignorado
  - **`func`**: função a ser chamada quando o sinal ocorre
    - um único argumento do tipo `int`
    - função do tipo `void`

signal handler

número do sinal recebido

- ✓ retorno de `signal`
  - função usada anteriormente para tratar o sinal
  - SIG_IGN ou SIG_DFL

restaurar comportamento default

ignorar o sinal

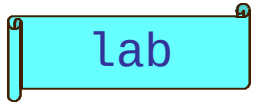
Exemplo: manipulando sinais

fonte: `ctrlc.c`

```
/*  The function reacts to the
signal passed in the parameter sig.
This function is called when a
signal occurs.
    We print a message, then reset
the signal handling for SIGINT
(generated by pressing Ctrl-C) back
to the default behavior.
    Let's call this function ouch.
*/

#include <signal.h>
#include <stdio.h>
#include <unistd.h>

void ouch(int sig)
{
    printf("OUCH! - I got signal
%d\n", sig);
    (void) signal(SIGINT, SIG_DFL);
}
```

```
/*  The main function has to
intercept the SIGINT signal.
    For the rest of the time, it
just sits in an infinite loop,
printing a message once a second.
*/

int main()
{
    (void) signal(SIGINT, ouch);

    while(1) {
        printf("Hello World!\n");
        sleep(1);
    }
}
```

Executar e digitar `Ctrl-C` duas vezes. O que ocorre?



Enviando sinais

✓ kill

usando **kill** um processo pode enviar um *signal* qualquer a um outro processo conhecendo seu **pid**

```
#include <sys/types.h>
#include < signal.h>
int kill (pid_t pid, int sig);
```

sinal a ser enviado

- superusuário pode enviar sinais a qualquer processo
- processos comuns só podem enviar sinais a processos com o mesmo user ID (mesmo *owner*)

`kill` falha **se** o sinal for inválido, **se** não possuir as permissões adequadas, **se** o processo destino não existir

Exemplo: alarme (função e main)

alarm.c

```
/*  In alarm.c, the first
function, ding, simulates
an alarm clock.  */

#include <signal.h>
#include <stdio.h>
#include <unistd.h>

static int alarm_fired = 0;

void ding(int sig)
{
    alarm_fired = 1;
}
```

```
/*  In main, we tell the child process to
wait for five seconds before sending a
SIGALRM signal to its parent.  */

int main()
{
    int pid;

    printf("alarm application starting\n");

    if((pid = fork()) == 0) {
        sleep(5);
        kill(getppid(), SIGALRM);
        exit(0);
    }
```

ding é chamado quando ocorre o sinal SIGALRM

Exemplo: alarme (processo pai)

```
/*  The parent process arranges to catch
SIGALRM with a call to signal and then waits
for the inevitable.  */

    printf("waiting for alarm to go off\n");
    (void) signal(SIGALRM, ding);

    pause();
    if (alarm_fired)
        printf("Ding!\n");

    printf("done\n");
    exit(0);
}
```

pause: suspende a execução até que um sinal ocorra (se o sinal já tiver ocorrido, o processo vai esperar par sempre)

existe uma função `alarm` que faz praticamente a mesma coisa do exemplo

Executar. O que ocorre?

`lab`

Problemas usando signals

- ✓ syscalls
  - algumas syscall podem falhar quando um sinal ocorre durante a sua execução
- ✓ race conditions
  - sinal ocorre antes do esperado (o programa ainda não havia chegado lá e fica esperando o sinal eternamente)
- ✓ temporização

usar signals com muito cuidado

sigaction: um interface mais robusta

- **sigaction** mais robusta que função **signal**

```
#include <signal.h>

int sigaction(int sig, const struct sigaction *act,
              struct sigaction *oact);
```

- 3 parâmetros
  - **`sig`**: sinal a ser capturado ou ignorado
  - dois ponteiros para estruturas sigaction
    - primeiro ponteiro: ação desejada para `sig`
    - segundo ponteiro: ação anterior

sigaction: estrutura

- ✓ estruturas usadas no segundo e terceiro parâmetro

`struct sigaction` possui no mínimo os seguintes elementos:

```
void (*) (int) sa_handler  /* função, SIG_IGN ou SIG_DFL
sigset_t sa_mask           /* sinais a bloquear em sa_handler
int as_flags               /* modificadores da ação
```

`sa-handler` (signal handler) função que manipula o sinal

`sa-mask` permite eliminar condições de race, sinais bloqueados não são enviados ao processo

signal handler

- ✓ manipular de sinais
  - ✓ deve executar o mínimo de tarefas
    - evitar operações de E/S ou chamadas de funções do sistema
- ✓ cuidado:
  - o processo pode estar numa situação muito frágil quando interrompido pelo sinal
  - o manipulador pode ser interrompido pela ocorrência de outro sinal
    - problemas de sincronização e seção crítica
    - tentar usar apenas operações atômicas quando operar com varáveis globais

      assinar valor a inteiro é atômico em Linux

Exemplo: usando sigaction

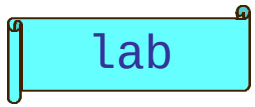
fonte: `ctrlc.c`

```
/*  The function reacts to a
signal sig. This function is
called when a signal occurs.*/

#include <signal.h>
#include <stdio.h>
#include <unistd.h>

void ouch(int sig)
{
    printf("OUCH! - I got signal
%d\n", sig);
}
```

uma mensagem será sempre gerada pressionando `ctrl-\`

para terminar o programa usar `ctrl-\` gerando SIGQUIT

```
/*  The main function intercepts
the SIGINT signal. For the rest
of the time, it just prints a
message once a second.  */

int main()
{
    struct sigaction act;

    act.sa_handler = ouch;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;

    sigaction(SIGINT, &act, 0);

    while(1) {
        printf("Hello World!\n");
        sleep(1);
    }
}
```



Fim

- ✓ de processos ...