Computadores de Programação (DCC/UFRJ)

Aula 17: Programas em execução

Prof. Paulo Aguiar

Processos

- Exceções são o mecanismo básico que permite a construção do conceito de **processo** ("programa em execução")
- O conceito de processo permite a "ilusão" de que cada programa dispõe de todos os recursos da máquina (processador, memória, I/O)

Contexto de execução dos processos

Cada processo possui um **contexto de execução**:

- código e dados do programa na memória
- conteúdo dos registradores de uso geral
- ponteiro de programa
- pilha
- variáveis de ambiente
- descritores de arquivos abertos

Abstrações de processos

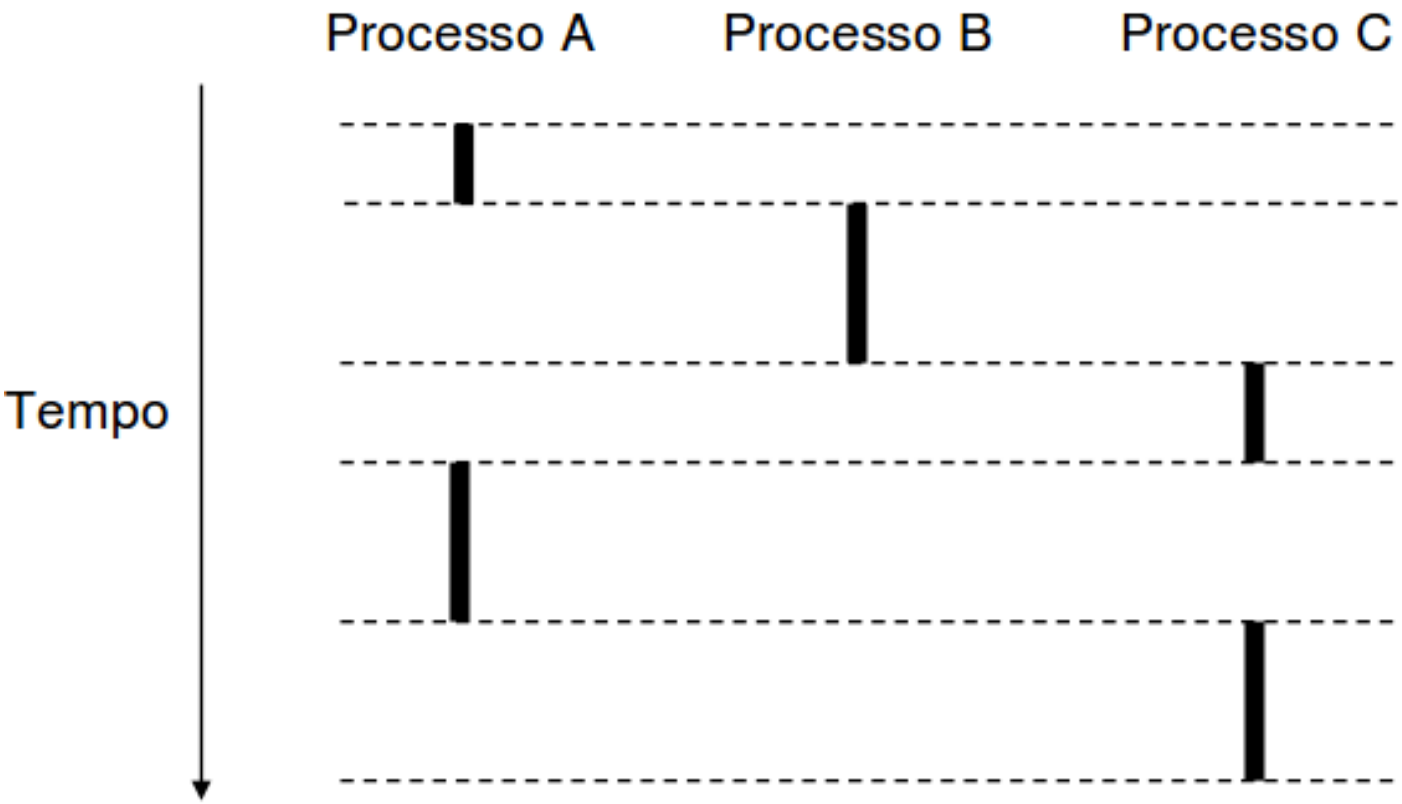
Abstrações que um processo provê para a aplicação:

1. **fluxo de controle lógico independente** (provê a ilusão de que o processo possui uso exclusivo do processador)
2. **espaço de endereçamento privado** (provê a ilusão de que o processo possui uso exclusivo da memória)

Fluxo de controle lógico

- O uso de um **depurador** passo-a-passo durante a execução de um programa permite observar uma **série de valores para o ponteiro de programa** que corresponde exclusivamente às **instruções do programa**
- Essa sequência de valores do PC (ponteiro de programa) corresponde ao **fluxo de controle lógico**

Fluxo de controle lógico



Fluxos de controle concorrentes

- Um fluxo de controle lógico cuja execução sobrepõe no tempo outro fluxo de execução é chamado **fluxo concorrente**
- Nesse caso diz-se que os dois fluxos **executam concorrentemente** (*ver exemplos no slide anterior:* $A \mid\mid B$, $A \mid\mid C$, B e C não são concorrentes)

Definição

Os processos X e Y são concorrentes entre si se e somente se

- X começa enquanto Y ainda está em execução,
- ou Y começa enquanto X ainda está em execução.

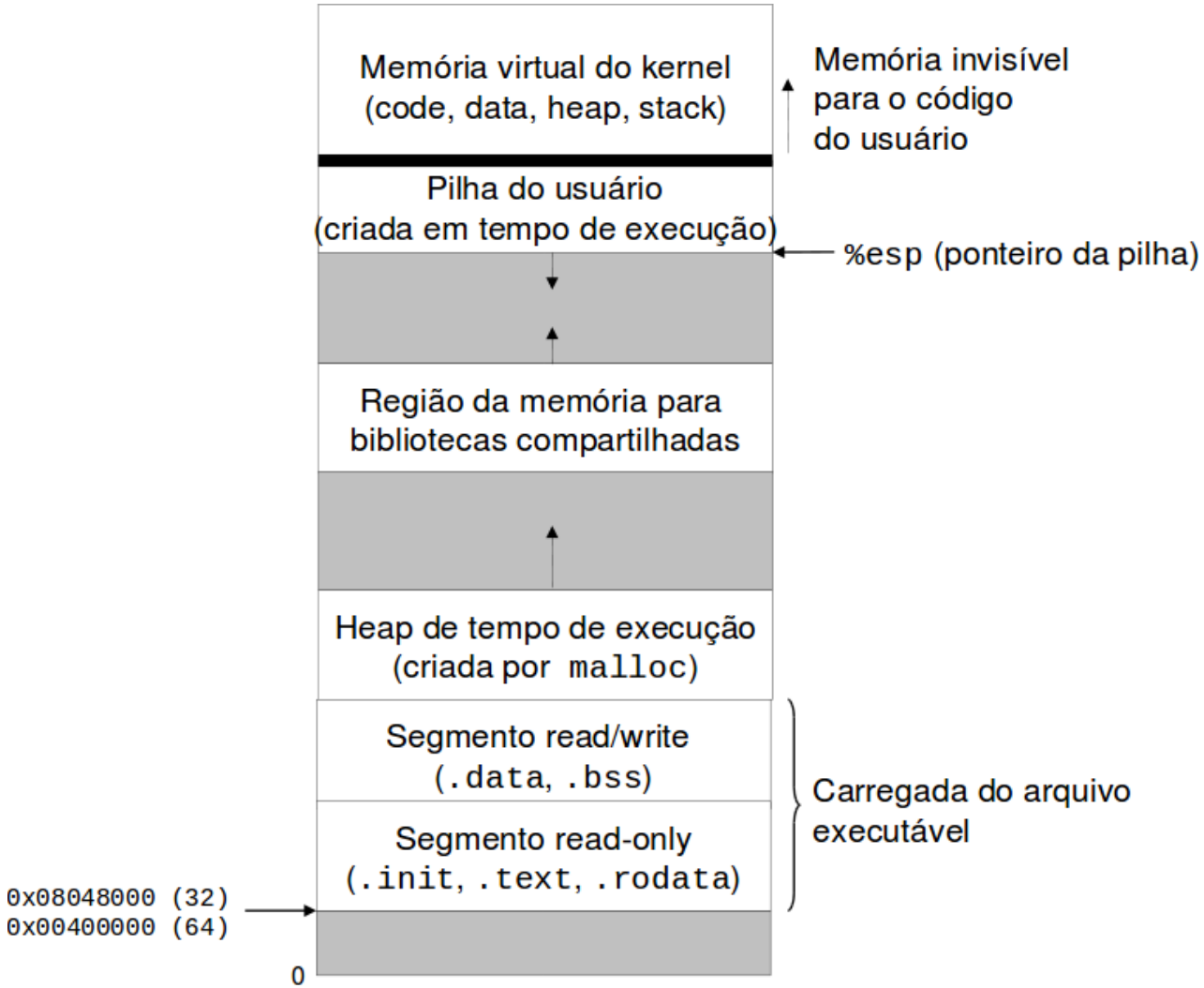
Fluxos de controle concorrentes

- A noção de **fluxos concorrentes** independe do número de núcleos do processador: se dois fluxos de execução se sobrepõem no tempo eles são concorrentes, independemente se rodam no mesmo núcleo ou não
- Fluxos concorrentes executando em núcleos ou computadores diferentes são chamados **fluxos paralelos**

Espaço de endereçamento privado

- Em uma máquina com palavra de endereço de N bits, o **espaço de endereçamento** de um processo é de 2^N endereços (bytes 0 a $2^N - 1$)
- Cada processo possui um **espaço de endereçamento privado** (todos os bytes de memória associados a um processo particular não podem ser acessados por outros processos)
- Embora o conteúdo da memória associado a um espaço de endereçamento privado seja diferente em geral, cada espaço possui a mesma organização geral

Organização da memória dos processos (x86/Linux)



Modo usuário e modo kernel

Um **bit de modo** (em um registrador especial) caracteriza o privilégio que o processo em execução dispõe

Bit de modo **ON** indica **modo kernel** (ou "supervisor")

Pode executar qq instrução e acessar qq parte da memória

Bit de modo **OFF** indica **modo usuário**

- Restrição das instruções executáveis: instruções privilegiadas como modificar o bit de modo, iniciar I/O, ou alterar processos não são permitidas
- Restrição da memória que pode acessar: não pode acessar diretamente código ou área de dados do kernel, devendo fazer uso de system calls

Execução de instruções privilegiadas

- Para executar instruções privilegiadas ou acessar código de dados do kernel, as aplicações devem fazer **chamadas de sistema** ao sistema operacional
- Os programas começam a executar no **modo usuário** e só mudam para o modo kernel via uma **exceção** (interrupção, falha, trap)
- Quando o controle passa para o **tratador da exceção**, o processador muda de modo usuário para **modo kernel**

Acessando estruturas de dados do kernel no Linux

- O Linux permite que processos em modo usuário consultem o conteúdo de estruturas de dados do kernel através de arquivos texto
- Isso é feito através do conteúdo dos arquivos nos diretórios **/proc** e **/sys**

Trocas de contexto

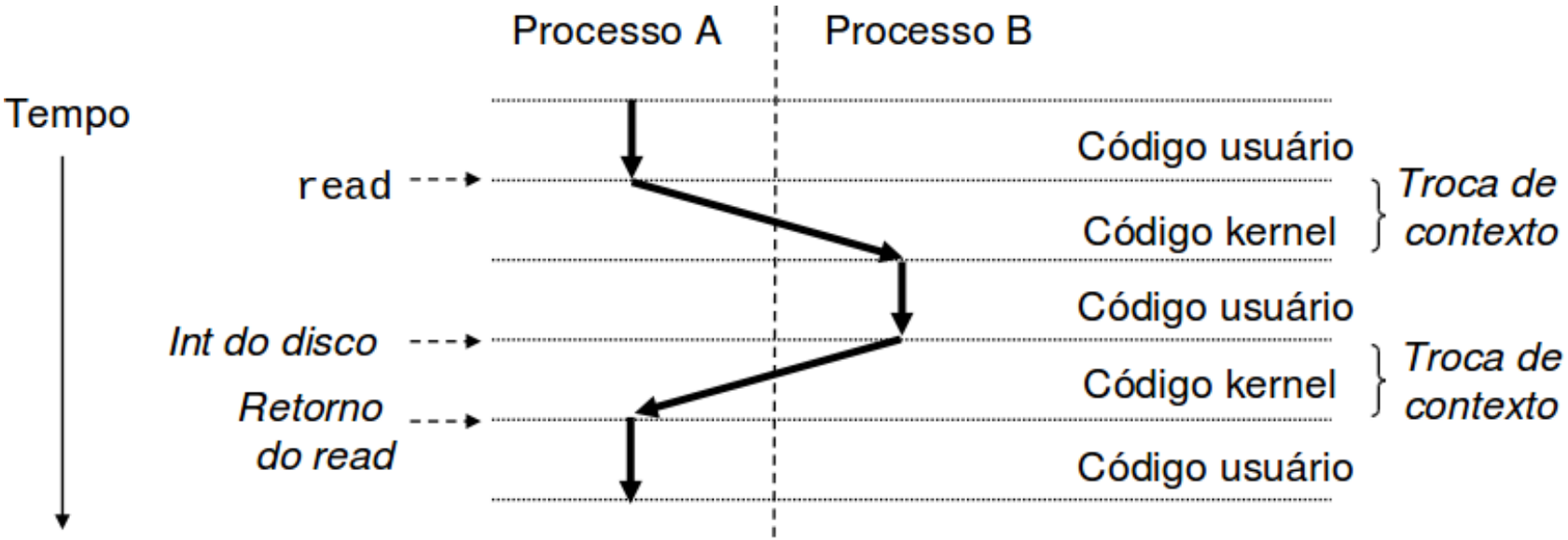
- O SO provê **multitarefa** (vários programas em execução ao mesmo tempo) usando uma forma de fluxo de controle excepcional chamada **troca de contexto**
- O mecanismo de troca de contexto é construído a partir dos mecanismos de exceção de nível mais baixo (*vistos na aula anterior*)

- O kernel mantém/guarda o **contexto** de cada processo (conjunto de valores de variáveis) necessário para reiniciar a execução do processo
- Quando um processo é selecionado para o processador, é feita uma "troca de contexto"

Passos para a troca de contexto

1. Salva o contexto do processo corrente
2. Restaura o contexto do processo escolhido
3. Passa o controle para o processo escolhido

Passos para a troca de contexto



Enquanto o processo A aguarda a leitura de disco, o processo B é colocado em execução

Criando e terminando processos no Linux

FORK

- Um processo cria um processo filho chamando a função **fork()**
- O processo fiho herda uma cópia idêntica (mas separada) do espaço virtual do pai, incluindo segmentos read-only (código), read/write (dados, bss), heap, pilha do usuário, além dos descritores dos arquivos abertos pelo pai (o que permite leitura e escrita pelo filho nestes arquivos)
- Pai e filho têm PIDs (número de identificação de processo ou *process ID*) diferentes

Criando e terminando processos no Linux

FORK

A função **fork** é chamada uma vez e retorna duas vezes:

- Para o processo pai que a chamou, `fork` retorna o **PID do processo filho**
- Para o processo filho criado, `fork` retorna **0**

Os processos pai e filho são processos separados que executam concorrentemente e sem garantia de quem irá reiniciar primeiro após o fork

Entendendo o fork

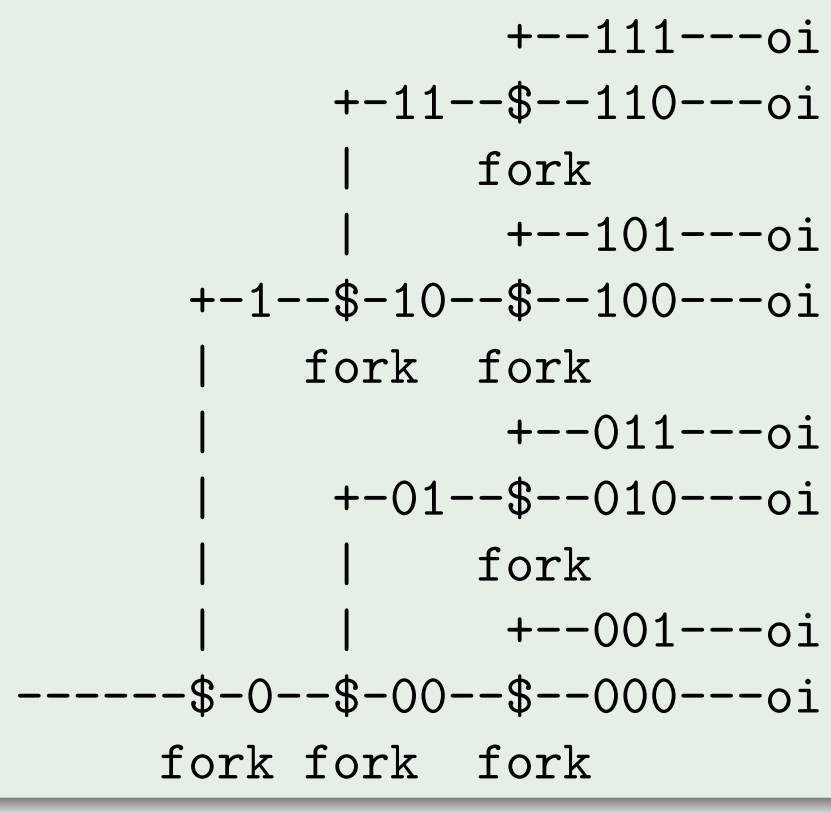
Quantas linhas serão impressas?

```
int main() {
    fork();
    fork();
    fork();
    printf("Oi\n");
    exit(0);
}
```

Entendendo o fork

Quantas linhas serão impressas?

```
int main() {
    fork();
    fork();
    fork();
    printf("Oi\n");
    exit(0);
}
```



Resposta: $2^3 = 8$

Criando processos com `fork`

Ver exemplos de código: fork.c; mostrar o comando `pstree` *para ver a árvore de processos*

Executar:

```
shell> fork
shell> ps u
shell> pstree <pid do pai>
```

Terminando processos

- Quando um processo termina, o kernel não o remove do sistema e ele é mantido no estado terminado até ser resgatado pelo processo pai
- O kernel passa ao pai o estado de saída do processo filho e então o descarta definitivamente
- Um processo que ainda não foi resgatado é denominado um zumbi
- Se o processo pai termina sem resgatar o filho zumbi, o kernel repassa a tarefa ao processo `init`, criado durante a incialização do sistema

Processos zumbis consomem memória e devem ser eliminados!

Esperando pelo término de processos filhos

`waitpid(pid_t pid, int *statusp, int options)`

Permite a um pai esperar pelo término ou parada de seus filhos

- Retorna PID do filho se OK, 0 (se não há filhos para esperar, pois já terminaram) ou -1 em erro
- Por default (`options=0`), suspende a execução do processo invocador até que um processo filho do conjunto em espera (*wait set*) terminar
- Se um processo filho já terminou antes da chamada, então `waipid` retorna imediatamente com o PID do filho terminado e o processo filho é descartado

Esperando pelo término de processos filhos

Função `waitpid(pid_t pid, int *statusp, int options)`

Conjunto em espera determinado pelo argumento `pid`

- Se `pid > 0`, espera pelo filho com o PID = `pid`
- Se `pid = -1`, então o conjunto consiste de todos os filhos

Ponteiro não nulo no argumento `int *statusp` permite o retorno de informação de estado do processo que causou o retorno

Alterando o default através das constantes `WNOHANG` e `WUNTRACED` em `options`

- `WNOHANG`: retorno imediato (com 0) se nenhum dos filhos não terminou ainda
- `WUNTRACED`: retorna PID de filho, quando um filho pára (*stop*) ou termina (default é só terminado)
- `WNOHANG | WUNTRACED`: retorna imediatamente com 0, se nenhum dos filhos parou ou terminou, ou o PID do filho que parou ou terminou.

Checando o estado do término de processos filhos

`wait.h` inclui macros para interpretar o argumento `status`

WIFEXITED(status): retorna TRUE se o filho terminou OK (com `exit` ou `return`)

WEXITEDSTATUS(status): retorna o estado de término de um filho que terminou normalmente. Só válido se WIFEXITED(status) retornou TRUE.

WIFSIGNALED(status): retorna TRUE se filho terminou porque não pegou um sinal

WTERMSIG(status): retorna o número do sinal que causou o término do processo filho. Só válido se WIFSIGNALED(status) retornou TRUE.

WIFSTOPPED(status): retorna TRUE se o filho que causou o retorno está parado

WSTOPSIG(status): retorna o número do sinal que forçou o filho a parar. Só válido se WIFSTOPPED(status) retornou TRUE.

Condições de Erro

Incluir errno.h

- Se o processo não possui filhos, `waitpid` retorna -1 e seta `errno = ECHILD`
- Se a função `waitpid` foi interrompida por um sinal, então retorna -1 e seta `errno = EINTR`

Esperando pelo término de processos filhos

A função `wait` é uma versão mais simples de `waitpid`

`statusp` é um ponteiro para inteiro

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

retorna PID do filho se OK ou -1 em caso de erro

A função `wait(&status)`

Equivalente a `waitpid(-1, &status, 0)`

Fork executando errado?

Programa

```
int main(){
    printf("Acima do fork PID %d ",getpid());
    fork();
    printf("\nEmbaixo do fork PID %d\n",getpid());
    exit(0);
}
```

Saída não esperada do programa

```
Acima do fork PID 4529 (Pai)
Embaixo do fork PID 4529 (Pai)
Acima do fork PID 4529 (?)
Embaixo do fork PID 4530 (Filho)
```

Rodar errofork e errofork1

Fork executando errado? Explicação

Sem o \n na lista do printf, o buffer de saída herdado pelo filho não é esvaziado e o processo filho ao imprimir sua saída, acaba imprimindo o lixo do pai novamente

Solução

Antes do for, dar um fflush(STDOUT), colocando \n a cada printf

```
int main(){
    printf("Acima do fork PID %d\n ",getpid());
    fork();
    printf("Embaixo do fork PID %d\n",getpid());
    exit(0);
}
```

Saída agora é a esperada e correta

```
Acima do fork PID 4529 (Pai)
Embaixo do fork PID 4529 (Pai)
Embaixo do fork PID 4530 (Filho)
```

Checando o estado do término de processos filhos

Problema

```
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
int main() {
    int status;
    pid_t pid;
    printf("Oi\n");
    pid = fork();
    printf("%d\n", !pid);
    if (pid != 0) {
       if (waitpid(-1, &status, 0) >0) {
           if (WIFEXITED(status) != 0)
              printf("%d\n", WEXITSTATUS(status));
       }
    }
    printf("Tchau\n");
    exit(2);
}
```

a) Quantas linhas de saída são impressas?

b) Qual uma possível ordem das linhas de saída?

Checando o estado do término de processos filhos

Problema

```
int main() {
    int status;
    pid_t pid;
    printf("Oi\n");
    pid = fork();
    printf("%d\n", !pid);
    if (pid != 0) {
       if (waitpid(-1, &status, 0) >0) {
           if (WIFEXITED(status) != 0)
              printf("%d\n", WEXITSTATUS(status));}
    }
    printf("Tchau\n");
    exit(2); }
```

a) Quantas linhas de saída são impressas? R: 6 sempre
b) Qual uma possível ordem das linhas de saída?

```
      --> 0 --> 2 --> Tchau   Processo Pai
    /
Oi-
    \
      --> 1 --> Tchau         Processo Filho
```

Processo pai espera pelo termino do processo filho

Suspendendo processos

`sleep` suspende o processo por um período definido de tempo

```
#include <unistd.h>
unsigned int sleep(unsigned int secs);
```

- Retorna 0 se o tempo de suspensão já passou; ou
- Retorna os segundos que restariam de suspensão (quando processo retorna prematuramente devido a interrupção por `signal`

`pause` adormece processo até um `signal` ser recebido

```
int pause(void);
```

Sempre retorna -1

Exercício

Escreva uma função envelope para `sleep`, chamada `adormece`:

```
unsigned int adormece(unsigned int secs);
```

A função se comporta como sleep, retornando o número de segundos faltantes, exceto que imprime o número de segundos que de fato o processo ficou adormecido

Solução

```
unsigned int adormece(unsigned int segs) {
    unsigned int rc = sleep(segs);
    printf("Adormeceu por %u de %u segs.\n",segs-rc,segs);
    retorna rc;
}
```

Referências bibliográficas

- *Computer Systems–A Programmer's Perspective* (**Cap. 8, até 8.4.4**)