Threads

threads em Linux

Taisy Weber

Ambiente UNIX

- ✓ 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

Mitchell, cap 4

Matthew & Stones, cap 11

Operating System Concepts - Silberschatz & Galvin, 1998

Threads em Linux

- ✓ conceito de threads
- ✓ Posix threads
- ✓ create, exit and join
- ✓ simultaneidade e concorrência
- ✓ seção crítica
- ✓ semáforos, mutex

no contexto de threads

Threads

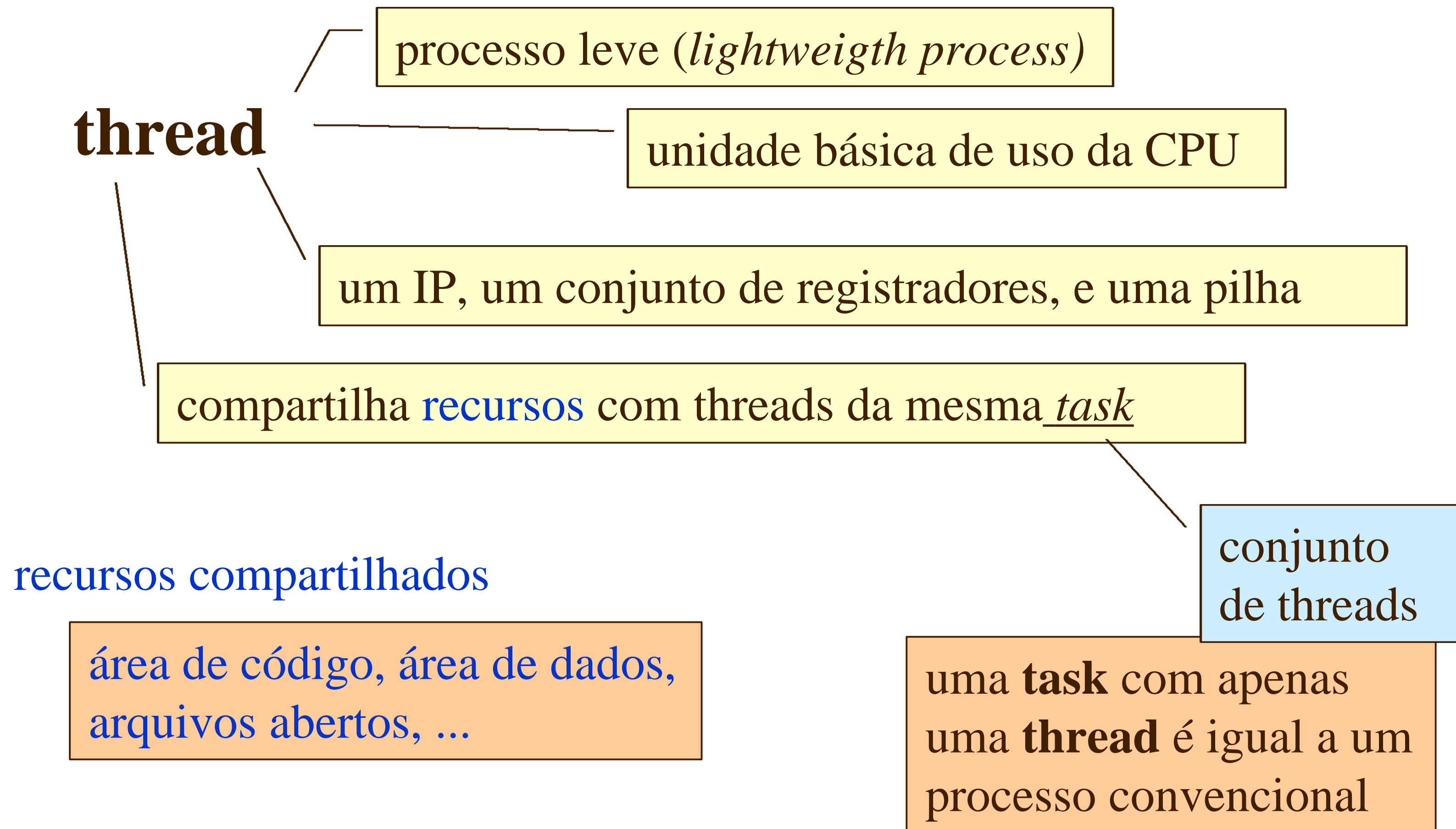
- ✓ processo
  - definido pelo recursos usados e espaço de memória onde está executando

pode ser interessante partilhar recursos concorrentemente

- ✓ thread

semelhante a um processo filho com novo IP mas executando no mesmo espaço de endereçamento do pai

Thread



Thread

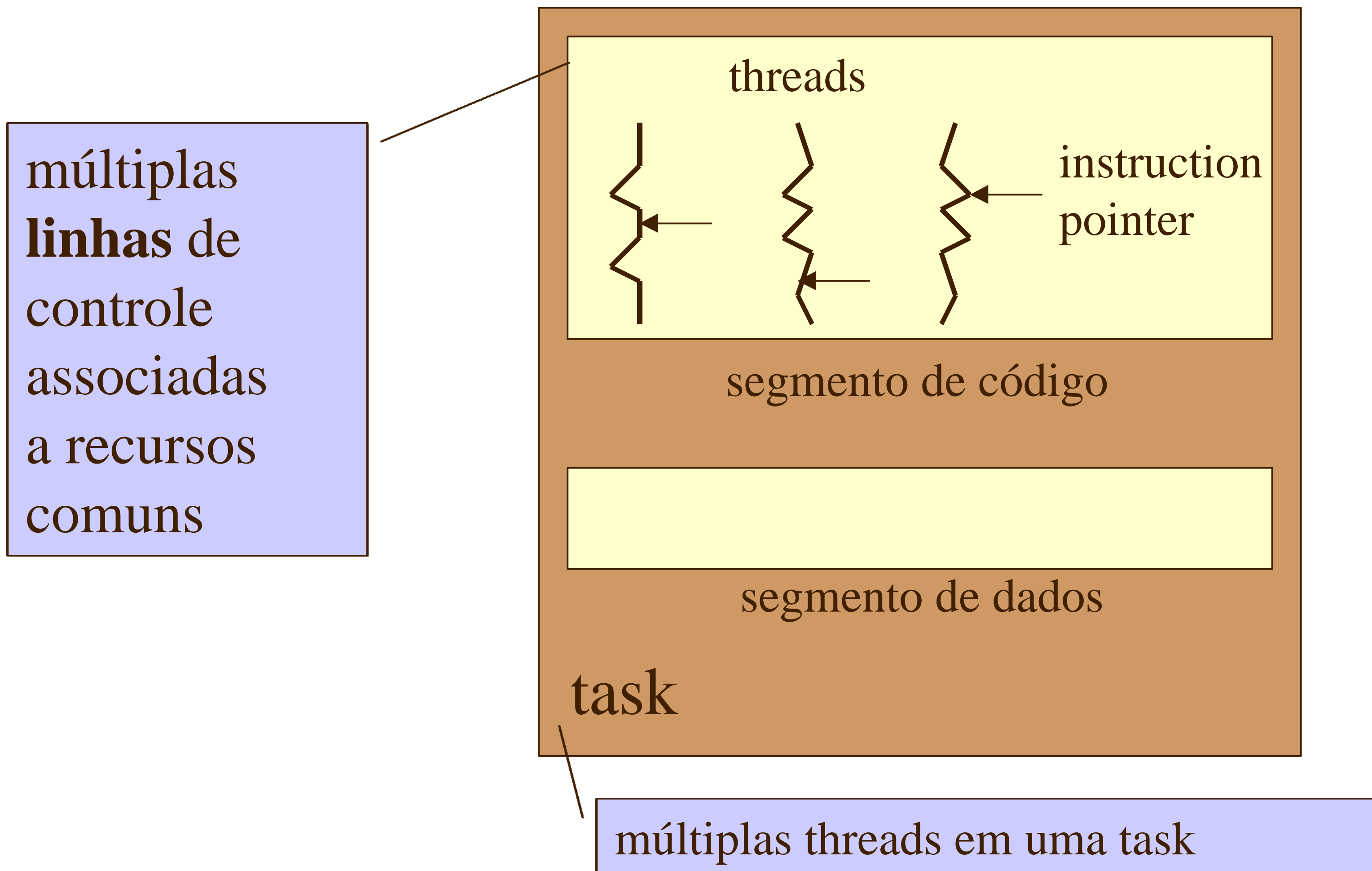
- principais vantagens frente a processos:
  - maior facilidade na criação
  - maior velocidade de troca de contexto entre threads da mesma task

ainda é necessário troca de registradores

não é necessário troca de páginas de memória

CUIDADO: multithread pode acarretar problemas de concorrência

Task & threads



Threads semelhantes a processos

- operam de forma semelhante a processos
  - estados:

    pronto, bloqueado, executando, terminado

  - apenas uma thread de cada vez em execução na CPU
  - uma thread executa seqüencialmente
  - uma thread pode criar threads filhas

Threads diferentes de processos

- ✓ mas são diferentes de processos
  - ✓ quando uma thread bloqueia, outra thread da mesma task pode ser executada

    se thread for suportada pelo SO

    um processo fica inteiramente bloqueado
  - ✓ threads não são independentes umas das outras
    - uma thread pode invadir o espaço de outra thread invalidando-a
  - ✓ threads não são protegidas uma das outras

Implementação de threads

- ✓ ou suportadas pelo kernel

  através de system calls como os processos

- ✓ ou suportadas por chamadas de rotinas em uma biblioteca no nível do usuário

  threads de usuário são rápidas na troca de contexto entre si

  uma chamada ao SO faz todo o processo esperar (o SO ignora threads do usuário e só reconhece processos)

- ✓ ou ambas

  Linux suporta threads de usuário através de package

  system call **clone** do Linux

Processos e threads no Linux

- Linux trata tudo como task
  - threads são casos particulares de processos
  - internamente **processos** e **threads** são tratados da mesma forma

a diferença aparece apenas na hora da criação (por syscall):
fork cria novo processo
clone cria thread

thread: processo com identidade própria, com próprio contexto de escalonamento, mas que compartilha a estrutura de dados com seu pai

ambas *system calls* chamam a rotina **do-fork** do kernel

Suporte a threads

- padrão POSIX
  - _POSIX_VERSION
    - valor 199506L ou maior

procurar nos arquivos header

- teste de suporte
  - no livro texto há um exemplo de programa para verificação do suporte a threads
  - fonte: `thread1.c`
  - aconselhável testar suporte a threads no seu sistema antes de iniciar com os exercícios

Matthew & Stones, cap 11

Posix threads

- biblioteca para threads Posix
  - lpthread

linkar usando `-lpthread`

- designação
  - nomes de chamadas de função iniciam com `pthread`

incluir arquivo `pthread.h`

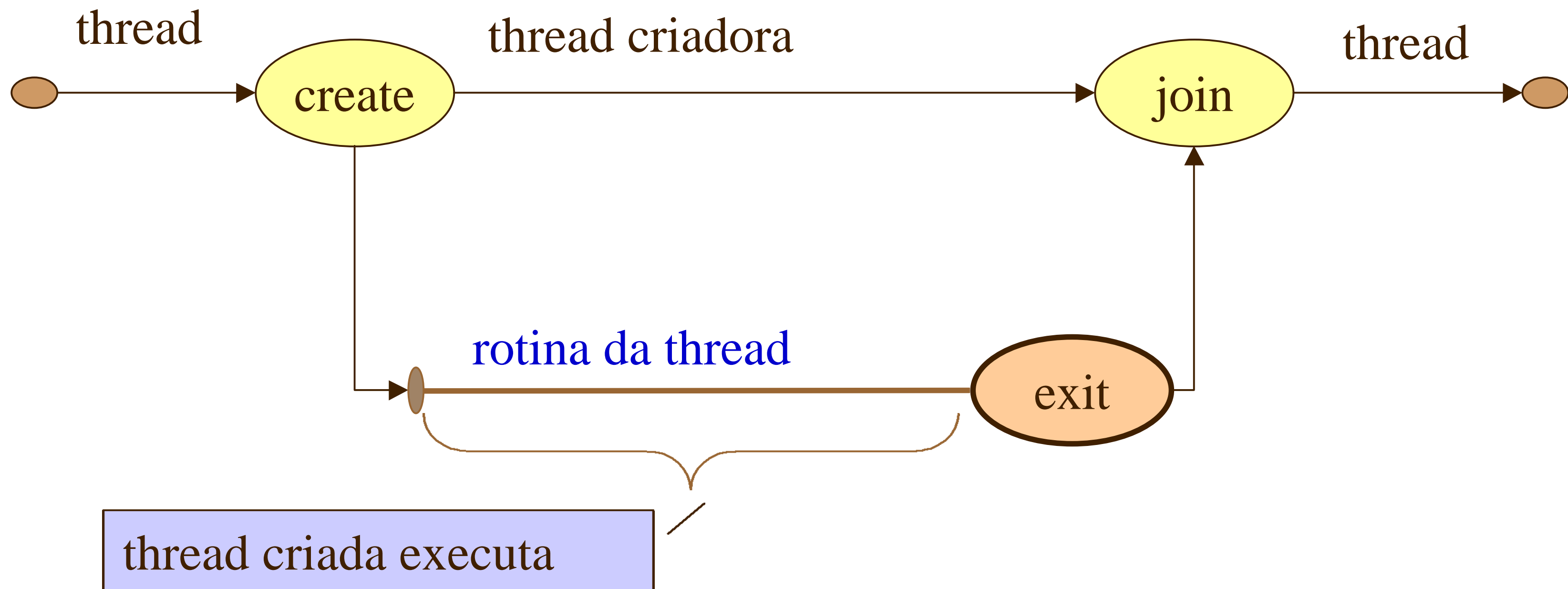
  - nome do header file: `pthread.h`

Código “reentrante”

- ✓ pode ser chamado mais de uma vez e ainda funcionar corretamente
  - – vale para invocações aninhadas
  - – vale para threads
  - condição: usar apenas variáveis locais
    - – assim a execução de cada chamada vai possuir sua cópia particular de variáveis
- ✓ macro `_REENTRANT`
  - usar antes de qualquer `#include` em programas multithread

informa ao compilador que é necessário gerar código reentrante

create, exit and join



uma thread é criada pela função `pthread_create`, que indica qual a rotina deve ser executada pela nova thread

create

```
#include <pthread.h>

int pthread_create(pthread_t *thread, pthread_attr_t *attr,
                   void *(*start_routine) (void*), void *arg);
```

`pthread_t *thread` — identificador da *thread*

`pthread_attr_t *attr` — conjunto de atributos da *thread*

`void *(*start_routine) (void*)` — função que a *thread* executa após sua criação

passa o endereço de uma função tomando um ponteiro para `void` como parâmetro e retornando um ponteiro para `void`

`void *arg` — argumentos passados a `start_routine`

exit & join

```
#include <pthread.h>

void pthread_exit(void *retval);
```

semelhante a exit para processos

retorna um ponteiro para um objeto (jamais retornar *pointer* para uma variável local a *thread*, pois essas variáveis desaparecem após **exit**)

```
#include <pthread.h>

int pthread_join(pthread_t th, void **pthread_return);
```

semelhante a wait para coleta de processos filhos

`pthread_t th` identificador da *thread* que se deseja esperar

`void **pthread_return` ponteiro para um ponteiro com o valor de retorno da *thread*

Primeiro programa com threads

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>

void *thread_function(void *arg);

char message[] = "Hello World";
```

cria uma *thread* extra, mostra compartilhamento de variáveis e captura valor de retorno da *thread* extra

`message` é variável global

fonte: `thread2.c`

```c
int main() {
    int res;
    pthread_t a_thread;
    void *thread_result;
    res = pthread_create(&a_thread, NULL,
        thread_function, (void *)message);
    if (res != 0) {
        perror("Thread creation failed");
        exit(EXIT_FAILURE);
    }
    printf("Waiting for thread to
            finish...\n");
    res = pthread_join(a_thread,
                       &thread_result);
    if (res != 0) {
        perror("Thread join failed");
        exit(EXIT_FAILURE);
    }
    printf("Thread joined, it returned %s\n",
           (char *)thread_result);
    printf("Message is now %s\n", message);
    exit(EXIT_SUCCESS);
}
```

Continua: prim. prog.

```
void *thread_function(void *arg) {
    printf("thread_function is running. Argument was %s\n", (char *)arg);
    sleep(3);
    strcpy(message, "Bye!");
    pthread_exit("Thank you for the CPU time");
}
```

start_routine: `thread_function`

`strcpy` - string copy

```
$ cc -D_REENTRANT thread2.c -o thread2 -lpthread
```

executar e determinar a saída

Simultaneidade e concorrência

em computadores monoprocessados, *threads* não executam simultaneamente mas concorrentemente

✓ tudo, exceto variáveis locais, é partilhado entre as *threads* de uma tarefa

- potencial para gerar **conflitos** entre *threads*

vários processos acessam concorrentemente uma variável comum e o resultado depende da ordem dos acessos

✓ evitando **conflitos**

- garantir que apenas um processo (ou *thread*) de cada vez manipule uma variável compartilhada
  - usar semáforos e mutexes para sincronizar *threads*

Seção crítica

- o que é?
  - **trecho de código** no qual um processo pode estar alterando variáveis comuns, escrevendo uma tabela, escrevendo um arquivo, ...

acesso a recursos compartilhados

- para evitar conflitos
  - quando um processo executa em sua seção crítica, nenhum outro processo pode executar na sua seção crítica

exclusão mútua no tempo

cada região crítica pode ser diferente

Solução para seção crítica

entry section

- ✓ cada processo pede permissão para entrar em sua seção crítica

exit section

- ✓ a seção crítica é seguida por uma seção de saída

entry section
critical section
exit section
remainder section

*se nenhum outro processo pode executar na região crítica, **variáveis compartilhadas** são preservadas*

Requisitos para SC

- ✓ exclusão mútua

  quando um processo executa em sua **SC**, nenhum dos outros processos pode executar na sua **SC**

- ✓ progresso

  se não existem processos na SC e se existem procs. que querem entrar na SC, **então** apenas esses procs. podem participar da decisão de quem será o próximo **e** a decisão não pode ser postergada infinitamente

- ✓ espera limitada

  existe um número limitado de vezes que um outro processo pode entrar sua **SC** depois que um processo que pediu permissão a receba

Semáforo

fora a inicialização

- **S** é uma variável **inteira** que é acessada apenas por duas operações atômicas: P e V

wait e signal são seções críticas

semáforo

P

wait(S): **while** S $\leq$ 0 **do** *no-op;*
S:= S - 1;

V

signal(S): S:= S + 1;

se um processo altera o valor do semáforo, nenhum outro processo pode simultaneamente modificar o mesmo semáforo

teste do semáforo e sua alteração é uma operação atômica

Seção crítica com mutex

*mut*ual *ex*clusion

✓ seção crítica com n processos (ou *threads*)

implementação de ***mutex*** como semáforo

n processos compartilham o semáforo ***mutex*** com valor inicial **1**

repeat

wait(mutex);

critical section

signal(mutex);

remainder section

until ...

lock

while mutex $\leq$ 0 **do** *no-op;*
mutex := mutex - 1;

unlock

mutex := mutex +1;

Semáforo com *busy waiting*

- problemas:
  - soluções com **espera em laço** de execução
  - semáforo implementado com busy waiting é chamado *spinlock*

while *condition* **do** *no-op*

queda de desempenho: processos ocupam CPU sem executar trabalho útil

útil em multiprocessadores,
evita chaveamento de contexto

Semáforo sem *busy waiting*

- wait

se o semáforo não é positivo, o processo bloqueia

processo vai para **fila de espera** associada ao semáforo, estado do processo = *wait*

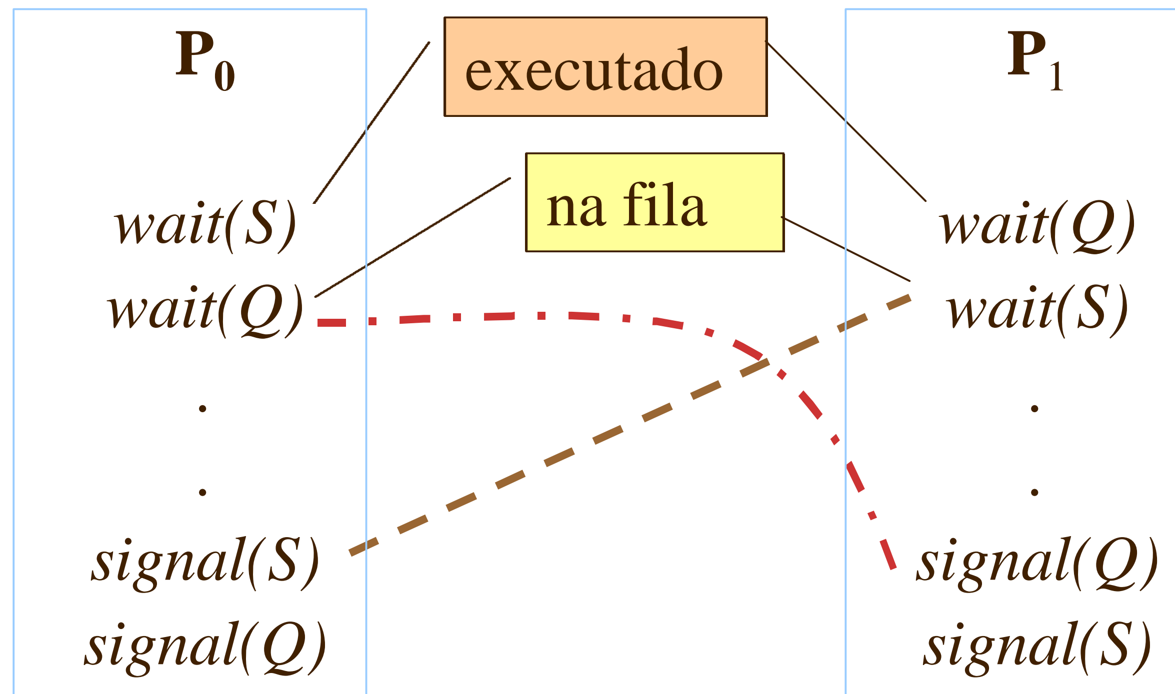
- signal

coloca o outro processo em *ready*

um processo executando ***signal*** retira um outro processo da **fila de espera** associada ao semáforo

como mutex é um caso especial de semáforo, a mesma implementação vale para lock e unlock

Deadlock com semáforos



2 ou mais processos **esperam indefinidamente** por um evento causado apenas por outro processo também bloqueado

Sincronização de threads com mutexes

```
#include <pthread.h>

int pthread_mutex_init(pthread_mutex_t *mutex,
                       const pthread_mutexattr_t *mutexattr);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

atributos para mutex (`mutexattr`) podem ser alterados de forma a evitar deadlocks

aqui serão usados atributos padrões (passando `null` no argumento)

Exemplo usando mutex: parte 1

fonte `thread5.c`

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>

void *thread_function(void *arg);
pthread_mutex_t work_mutex;     /* protege work_area e time_to_exit */

#define WORK_SIZE 1024
char work_area[WORK_SIZE];
int time_to_exit = 0;
```

declara mutex e as variáveis comuns
mutex protege essas variáveis

determinar a linha de comando para compilação
executar e determinar a saída

lab

Exemplo usando mutex: parte 2

main ...

```
int main() {
    int res;
    pthread_t a_thread;
    void *thread_result;

    res = pthread_mutex_init(&work_mutex, NULL);
    if (res != 0) {
        perror("Mutex initialization failed");
        exit(EXIT_FAILURE);
    }

    res = pthread_create(&a_thread, NULL, thread_function, NULL);
    if (res != 0) {
        perror("Thread creation failed");
        exit(EXIT_FAILURE);
    }
```

inicializa mutex `work_mutex`

cria thread `a_thread`

`errno` não é usado por essas funções pthread, `res` deve ser testado

Exemplo usando mutex: parte 3

...main ...

```
    pthread_mutex_lock(&work_mutex);
    printf("Input some text. Enter 'end' to finish\n");
```

`'end'` faz a outra *thread* ligar `time_to_exit`

```
while(!time_to_exit) {
        fgets(work_area, WORK_SIZE, stdin);
        pthread_mutex_unlock(&work_mutex);
        while(1) {
            pthread_mutex_lock(&work_mutex);
            if (work_area[0] != '\0') {
                pthread_mutex_unlock(&work_mutex);
                sleep(1);
            }
            else {
                break;
            }
        }
    }
    pthread_mutex_unlock(&work_mutex);
```

verifica se a outra *thread* consumiu o *string* lido testando variável compartilhada dentro de uma região crítica

essa *thread* lê o string para a outra *thread* processar

Exemplo usando mutex: parte 4

```
... main
```

```
    printf("\nWaiting for thread to finish...\n");
    res = pthread_join(a_thread, &thread_result);
    if (res != 0) {
        perror("Thread join failed");
        exit(EXIT_FAILURE);
    }
    printf("Thread joined\n");
    pthread_mutex_destroy(&work_mutex);
    exit(EXIT_SUCCESS);
}
```

espera pela outra *thread* terminar

destroi mutex

Exemplo usando mutex: parte 5

rotina da thread

```
void *thread_function(void *arg) {
    sleep(1);

    pthread_mutex_lock(&work_mutex);
    while(strncmp("end", work_area, 3) != 0) {
        printf("You input %d characters\n", strlen(work_area) -1);
        work_area[0] = '\0';
        pthread_mutex_unlock(&work_mutex);
        sleep(1);
        pthread_mutex_lock(&work_mutex);
        while (work_area[0] == '\0' ) {
            pthread_mutex_unlock(&work_mutex);
            sleep(1);
            pthread_mutex_lock(&work_mutex);
        }
    }
    time_to_exit = 1;
    work_area[0] = '\0';
    pthread_mutex_unlock(&work_mutex);
    pthread_exit(0);
}
```

primeira entrada na região crítica

`strncmp` compara strings
`strlen` comprimento do string

última saída da região crítica

coloca null no *string* para indicar que processou *string*

Fim

- ✓ existem mais detalhes sobre threads …
- ✓ fogem ao escopo da nossa disciplina
  - olhar livros textos
  - olhar man sobre threads