

# Sistemas Operativos

(ano letivo 2019-20)

## Laboratório 3

Esta atividade está escrita na forma de tutorial e consiste na interpretação, compilação e execução de pequenos programas que efetuam chamadas a funções do sistema operativo UNIX/Linux para programação multitarefa. A informação referida neste enunciado foi necessariamente simplificada de modo a não sobrecarregar as ideias e conceitos principais que se pretendem transmitir.

Para realizar esta atividade é necessário ter instalada uma distribuição GNU/Linux no seu computador e de preferência ter lido as secções 1.5, 1.6, 2.1, 2.2 e 2.3 do livro [MOS3e] ou os capítulos (secções recomendadas) 3, 4, 5 e 6 do livro [SO2e], nomeadamente os conteúdos relacionados com os conceitos de tarefa (thread), condição de disputa (race condition), região crítica e exclusão mútua. Como leitura complementar aos conteúdos de programação aqui introduzidos, poderá ler o capítulo 4 do livro [ALD], com possível exceção das secções 4.2, 4.3, 4.4.3-7 e 4.5 (por estar desatualizada). Os programas mencionados nesta atividade são fornecidos no ficheiro `lab3-progs.zip`.

Como introdução às tarefas são abordadas as chamadas POSIX Threads (ou pthreads), que constitui um conjunto de definições padrão para programação com tarefas elaborado pelo organismo IEEE. O termo POSIX significa Portable Operating System Interface e o X vem de UNIX. Cada sistema operativo da família UNIX normalmente oferece uma interface de programação de tarefas compatível (em maior ou menor grau) com esta norma, incluindo o Linux.

Os programas e conceitos aqui apresentados podem em princípio ser compilados e executados em qualquer máquina UNIX, embora as instruções de compilação se refiram a uma máquina Linux. É também de referir que o Linux implementa as tarefas no espaço do núcleo, pelo que existe preempção, ou seja, uma tarefa que esteja a ser executada pode a qualquer instante ser suspensa pelo núcleo e escalonada uma outra para execução.

## I - Criação de Tarefas

Existem 6 funções básicas principais fortemente inter-relacionadas relativamente à criação de tarefas:

```
pthread_create()  
pthread_exit()  
pthread_join()  
pthread_detach()  
pthread_attr_init()  
pthread_attr_setdetachstate()
```

Um processo tem sempre pelo menos um percurso de execução ou tarefa, denominada tarefa principal, que inicia a sua execução na função `main()`, aceita vários argumentos (variáveis `argc` e `argv`) e termina naturalmente no fim desta ou por invocação da função `exit()`, retornando em ambos os casos um inteiro com um código de retorno.

De modo algo similar, a tarefa principal pode por sua vez através da função `pthread_create()` criar novas tarefas que têm início em funções a designar, aceitam um apontador como único argumento e terminam naturalmente no fim da respetiva função ou por invocação da função `pthread_exit()`, retornando em ambos os casos um apontador que, tipicamente, aponta para um objeto com o resultado do processamento efetuado ou para uma variável que contém algum código de retorno. As novas tarefas podem também por sua vez criar novas tarefas e assim sucessivamente se desejado.

Nota sobre a notação: A tarefa correspondente à função `main()` é sempre referida por "tarefa principal". Todas as outras tarefas são designadas simplesmente por "tarefa", ou, se se pretender enfatizar o facto de que foi criada por uma outra tarefa do processo, por "sub-tarefa".

Num processo multitarefa (multithreaded) podem assim resumir-se as seguintes diferenças entre tarefas,

- Tarefa principal: existe sempre, tem início na função `main()`, aceita vários argumentos e retorna um inteiro.
- Outras tarefas: são criadas explicitamente por outra tarefa, têm início em função a indicar, aceitam como único argumento um apontador e retornam um apontador.

A função `pthread_create()` é responsável pela criação (alocação das respetivas estruturas de dados no núcleo) e início de execução de uma nova tarefa. O seu protótipo é dado por,

```
#include <pthread.h>  
int pthread_create(pthread_t *thread, pthread_attr_t *attr,  
    void *(*start_routine)(void*), void *arg);
```

O primeiro argumento é um apontador para uma variável que irá servir de identificador da tarefa criada, a ser utilizada em posteriores chamadas a funções que envolvam a tarefa. O segundo argumento é um apontador para uma estrutura com atributos da tarefa a criar. No caso de se pretender os atributos por defeito, este argumento toma simplesmente o valor `NULL`.

O terceiro argumento é o nome da função onde é iniciada a execução da nova tarefa e o quarto é o único argumento dessa função. Isto apenas significa que algures no programa existe a função,

```
void *start_routine(void *arg)
```

de nome `start_routine`, com um argumento do tipo apontador para `void` e que retorna um apontador para `void`. Esta função constitui o ponto de entrada da tarefa, tal como a função `main()` para o programa/tarefa principal. O tipo `void*` é aqui utilizado como apontador genérico, devendo a função `start_routine()` internamente fazer as conversões de tipo (casting) apropriadas.

Um exemplo de um troço de programa que cria e inicia a execução de uma nova tarefa, com os atributos por defeito, é dado por,

```
void *tarefa(void *p); /* prototipo funcao tarefa */
pthread_t trfid;      /* variavel para ID da tarefa */
int trfarg;           /* variavel com argumento da tarefa */
int r;                /* codigo de retorno sucesso/erro */
...
/* criar e iniciar execução de tarefa */
r= pthread_create(&trfid, NULL, tarefa, (void*) &trfarg);
if( r )
    /* erro na criação da tarefa! */
...
```

onde a variável `trfarg` poderia ser de qualquer outro tipo, devendo ser sempre utilizada a conversão de tipo (`void*`) na chamada a `pthread_create()`.

No caso de sucesso na execução de `pthread_create()`, esta função retorna 0, caso contrário retorna um código de erro e nenhuma tarefa foi criada/iniciada. Como é usual, para mais informações consultar as "man pages" executando o respetivo comando `man pthread_create`.

Outro aspecto importante relacionado com a criação de uma tarefa é a relação, em termos de sincronização da tarefa "pai" com a tarefa "filho" quando uma delas termina, podendo ocorrer três cenários:

**Cenário 1** - A tarefa pai cria a tarefa filho mas não tem interesse saber quando a tarefa filho termina nem aceder ao apontador retornado por esta. Diz-se neste caso que a tarefa foi criada "detached" ou desacopolada. Neste caso, quando a tarefa termina, os recursos alocados devido à sua existência são imediatamente libertados.

Para se definir os atributos com que uma tarefa é criada, utiliza-se primeiro a função `pthread_attr_init()` para inicializar uma variável contendo o valor dos atributos por defeito.

Posteriormente, se desejado, invoca-se uma outra função específica para alterar o estado de um determinado atributo, como por exemplo, `pthread_attr_setdetachstate()` para alterar o atributo "detachstate" ou estado de desacoplamento. O protótipo destas duas funções mencionadas é dado por,

```
#include <pthread.h>
int pthread_attr_init(pthread_attr_t *attr);

int pthread_attr_setdetachstate(pthread_attr_t *attr,
    int detachstate);
```

O primeiro argumento é um apontador para uma variável do tipo `pthread_attr_t` que funciona como um contentor de atributos, a ser utilizado em posteriores chamadas a funções específicas para os ler/modificar e na criação de tarefas. O segundo argumento na segunda função modifica o valor do atributo "detachstate" para um de dois estados possíveis através de constantes pré-definidas: `PTHREAD_CREATE_DETACHED` e `PTHREAD_CREATE_JOINABLE` (este último estado a explicar mais à frente).

Continuando o exemplo anterior de um troço de programa que cria e inicia a execução de uma nova tarefa, mas agora com o atributo "detachstate" igual a `PTHREAD_CREATE_DETACHED`, vem,

```
void *tarefa(void *p); /* prototipo funcao tarefa */
pthread_attr_t trfatr; /* variavel para atributos da tarefa */
pthread_t trfid;      /* variavel para ID da tarefa */
int trfarg;           /* variavel argumento da tarefa */
int r;                /* codigo de retorno sucesso/erro */
...
/* inicializar variavel de atributos com valores de defeito */
pthread_attr_init(&trfatr);
/* modificar estado de desacoplamento para "detached" */
pthread_attr_setdetachstate(&trfatr, PTHREAD_CREATE_DETACHED);
/* criar e iniciar execução de tarefa */
r= pthread_create(&trfid, &trfatr, tarefa, (void*) &trfarg);
if( r )
    /* erro na criação da tarefa! */
...
```

Como base desta atividade introdutória à programação multi-tarefa vai ser utilizado um problema muito simples: somar os primeiros N números naturais 1,2,3,...,N. Apesar de simples, este problema é adequado ao fim que se propõe uma vez que o trabalho de somar os N números é facilmente dividido por vários "trabalhadores" ou tarefas.

Analise o programa exemplificativo `pth01.c`. Este programa não tem nada de extraordinário. Os primeiros N números naturais são colocados no vetor global `v[]` e a respetiva soma na variável global `S`, utilizando uma função auxiliar `soma()` que efetua a soma de `n` elementos de um vetor. Apesar de simples, este programa constitui um ponto de partida para outros programas mais complicados e à luz da programação multi-tarefa tem uma nova propriedade: trata-se de um programa mono-tarefa, ou seja, um processo constituído por uma única tarefa. Esta tem início na função `main()` e termina no fim desta, retornando um inteiro.

Compile o programa `pth01.c` com o comando,  
 >> `gcc -Wall -o pth01 pth01.c`  
 e execute-o dando o comando,

```
>> ./pth01
```

**L3.1** Observe o número impresso no ecrã. Este é o resultado da soma para o valor de N indicado no programa. O valor de N será sempre o mesmo para todos os exemplos ao longo desta atividade, e consequentemente, independentemente da forma como a soma total é efetuada, o resultado impresso deverá também ser sempre o mesmo. Para facilidade de consulta, anote-o num papel.

Analise agora o programa exemplificativo pth02.c. Este programa cria uma tarefa com o estado de desacoplamento "detached", com início na função `tarefa()`, que é agora a responsável por efetuar a soma total dos N números naturais contidos no vetor global `v[]`, colocar o resultado na variável global `S` e imprimir o resultado. A tarefa principal contém uma chamada à função `sleep()` introduzindo um tempo de espera de 4 segundos para garantir que a tarefa filho tem tempo para terminar antes da tarefa pai.

Nota: esta função `sleep()` não tem nada a ver com a função de mesmo nome referida na secção 2.3.4 do manual recomendado. Aqui `sleep(s)` faz com que a tarefa que a invoca passe ao estado de bloqueada durante `s` segundos, período a partir do qual a tarefa passa novamente ao estado de executável (ready).

Todos os recursos associados à tarefa criada são automaticamente libertados pelo núcleo quando esta termina, dado que foi criada com o atributo "detached". As duas tarefas deste programa são executadas em paralelo ou em pseudo-paralelismo, conforme o computador em que o programa for executado tiver um ou mais processadores respetivamente.

Compile o programa pth02.c (e os seguintes desta atividade) com o comando

```
>>gcc -Wall -D_REENTRANT -o pth02 pth02.c -lpthread
```

onde o último termo `-lpthread` indica que o programa deve ser ligado com a biblioteca que contém as funções relacionadas com as tarefas. A opção `-D_REENTRANT` indica que algumas funções de biblioteca do C devem ser substituídas por outras com funcionalidade idêntica mas que suportem a execução em simultâneo por várias tarefas.

**L3.2** Execute o programa e confira o valor da soma. Como se pode observar, o valor da soma é igual ao obtido com o programa pth01. Isto permite concluir que as tarefas, ou melhor, as funções que as implementam, têm acesso aos recursos globais do programa tais como variáveis (`v[]` e `S`) e funções (`soma()`).

**L3.3** Analise agora a sequência/temporização das mensagens impressas pelas tarefas. Estão de acordo com o que seria de esperar pela análise do código do programa? Justifique.

**L3.4** Ambas as tarefas reportam o valor do respectivo PID. Na sua opinião, este valor deve ser igual ou diferente para as duas tarefas ? Justifique.

**Cenário 2** - A tarefa pai termina antes da tarefa filho. Neste cenário há que distinguir dois casos.

O primeiro caso é quando a tarefa pai é a tarefa principal. Neste caso o término da tarefa corresponde ao fim (retorno) da função `main()` ou da invocação da função `exit()`. Esta situação por definição corresponde ao término do programa/processo, ou seja, ao término imediato de todas as suas tarefas!

**L3.5** No programa `pth02.c` na função `main()`, altere o argumento da chamada à função `sleep()` de 4 segundos para 0, ou seja, introduzindo um tempo de espera nulo. Assim a tarefa principal vai terminar rapidamente, antes da sub-tarefa. Compile e execute o programa alterado. Observe as mensagens impressas no ecrã. Estão de acordo com o esperado ? Justifique.

O segundo caso é quando a tarefa pai não é a tarefa principal. Neste caso o término da tarefa pai não tem qualquer efeito nas suas sub-tarefas ou em quaisquer outras tarefas do processo, cujas execuções continuam como se nada tivesse acontecido.

Analise o programa exemplificativo `pth03.c`. Neste programa a tarefa principal cria uma sub-tarefa que por sua vez cria uma sub-sub-tarefa, ambas com o estado de desacoplamento "detached", com início nas funções `tarefa1()` e `tarefa2()` respetivamente. A ideia é observar que apesar da `tarefa1()` terminar primeiro que a sua tarefa filho `tarefa2()`, esta última continua a sua execução, calculando e imprimindo o resultado da soma.

**L3.6** Compile e execute o programa `pth03.c`. Observe as mensagens impressas no ecrã. Estão de acordo com o esperado ? Justifique.

**Cenário 3** - A tarefa pai cria a tarefa filho mas pretende saber quando a tarefa filho termina e aceder à informação referenciada pelo apontador retornado por esta. Esta é a situação típica em que uma tarefa é criada com o propósito de efetuar alguma ação/processamento da qual se pretende tomar conhecimento do sucesso/resultado. Obviamente esta tomada de conhecimento só pode ser efetuada após a tarefa terminar, desconhecendo-se à partida o instante temporal exato em que isso vai acontecer. Assim é necessário um mecanismo que permita à tarefa pai esperar pelo término da tarefa filho, sem desperdiçar tempo de CPU (ou seja, no estado bloqueado). Esse mecanismo é providenciado pela função `pthread_join()`, cujo protótipo é dado por,

```
#include <pthread.h>
int pthread_join(pthread_t thread, void **value_ptr);
```

O primeiro argumento é o identificador da sub-tarefa da qual se pretende esperar pelo término. Se a sub-tarefa ainda não terminou, a tarefa que invocou a função é bloqueada até que termine, se a sub-tarefa já terminou, a função retorna imediatamente.

O segundo argumento é um apontador (de facto, um apontador para um apontador). No endereço apontado por esta variável é colocado o valor do apontador retornado pela sub-tarefa quando do seu término. Se não for desejado obter o apontador de retorno da sub-tarefa, deve utilizar-se `NULL` para segundo argumento. No caso de sucesso na execução da função, esta retorna 0, caso contrário retorna um código de erro.

Esta função só pode ser invocada para esperar por uma sub-tarefa criada com o atributo estado de desacoplamento "joinable" ou juntável, através da especificação de `PTHREAD_CREATE_JOINABLE` na invocação da função `pthread_attr_setdetachstate()`.

Analise o programa exemplificativo `pth04.c`. Neste programa é criada uma sub-tarefa com o estado de desacoplamento "joinable" e o trabalho de somar o vetor de números naturais é dividido igualmente pelas duas tarefas. Basicamente, este programa segue a seguinte estrutura temporal:

1. A tarefa principal cria e inicia uma sub-tarefa que soma metade do vetor;
2. A tarefa principal, em paralelo ou em pseudo-paralelismo, soma a outra metade do vetor;
3. Após a tarefa principal concluir a sua parte da soma, espera pelo término da sub-tarefa;
4. A tarefa principal acede ao valor da soma parcial efetuada pela sub-tarefa através do apontador retornado pela sub-tarefa;
5. A tarefa principal soma os resultados parciais e imprime o valor da soma total.
6. A função `main()` retorna e o processo termina.

Um pormenor importante é que o apontador retornado pela sub-tarefa aponta para uma variável global, mais concretamente uma variável de carácter permanente, cuja existência subsiste mesmo após o término da função correspondente à sub-tarefa. Nunca devem ser utilizados endereços de variáveis locais à função da sub-tarefa, pois a memória associada a estas variáveis é considerada livre pelo sistema operativo após o término da função, podendo entretanto ser utilizada para outros fins antes que a tarefa pai consulte o valor de retorno da tarefa filho. Por essa razão foi utilizada a variável global `S1`.

**L3.7** Compile e execute o programa `pth04.c`. Confirme o valor da soma total. Verifique através da sequência de mensagens impressas pelas tarefas se a sequência temporal de execução do programa está de acordo com o esperado.

O programa `pth04`, se executado numa máquina com pelo menos dois processadores, deverá conseguir efetuar a soma total em cerca de  $1/2$  do tempo relativamente ao programa `pth01` (uma só tarefa, que efetua toda a soma). Generalizando, numa máquina com  $n$  processadores deverá conseguir-se efetuar a soma total em cerca de  $1/n$  do tempo. Embora uma máquina com  $n$  processadores (ainda) não esteja ao alcance de qualquer um, é interessante ver um programa com  $n$  tarefas executadas em pseudo-paralelismo.

Análise o programa exemplificativo `pth05.c`. Neste programa foram introduzidas duas novidades relativamente ao programa `pth04`:

- O programa aceita agora um número  $n$  como argumento na linha de comandos que indica quantas tarefas deverão ser utilizadas para efetuar a soma dos primeiros  $N$  números naturais. Para não complicar demasiado o programa, o valor de  $N$  foi escolhido de modo a ser exatamente divisível por um conjunto o maior possível de valores de  $n$ . Assim, o trabalho de efetuar a soma é dividido igualmente por cada tarefa, ou seja, cada tarefa soma um sub-vetor com  $N/n$  elementos. Os valores de  $n$  admitidos pelo programa são de 1 a 15, 20 e 30.

- A estrutura que serve de argumento às tarefas criadas contém agora um campo adicional (variável `s`) destinado à colocação do resultado da soma parcial efetuada pela tarefa. Assim a estrutura contém todas as variáveis de entrada e de saída da função que implementa a tarefa. Este método é mais simples e elegante do que utilizar o apontador de retorno da tarefa. Assim a função `tarefa()` retorna `NULL` e na invocação da função `pthread_join()` é também usado `NULL` no segundo argumento.

Para se poder identificar facilmente cada tarefa no programa, foi introduzido outro campo adicional (variável `i`) que serve como índice da tarefa. As tarefas são numeradas de 0 a  $n-1$ , tendo-se convencionado que a tarefa principal é a tarefa número  $n-1$ .

Compile o programa `pth05.c`. Execute o programa para 15 tarefas com o comando,  
`>>./pth05 15`

**L3.8** Observe a sequência de execução (início/fim) das tarefas. Repita o mesmo comando várias vezes e observe que a sequência de execução das tarefas pode variar cada vez que executa o programa (se não ocorrer esse fenómeno, aumente n).

É esta variabilidade (ou falta de repetibilidade) que por vezes torna tão difícil descobrir erros (bugs) em programas multi-tarefa (entenda-se paralelismo), uma vez que é altamente improvável ocorrerem duas execuções exatamente iguais em termos temporais do mesmo programa.

**L3.9** Justifique porquê (dica: está relacionado com o escalonamento).

**L3.10** Execute também o programa para vários valores de n e confirme que o valor da soma se mantém correto.

## II - Exclusão Mútua

Analise o programa exemplificativo pth06.c. Neste programa é explorada outra estratégia para calcular a soma total. Em vez de cada sub-tarefa guardar o resultado da sua soma parcial numa variável para depois a tarefa principal somar todos os resultados parciais, agora cada sub-tarefa soma diretamente o seu resultado parcial à variável global S. Assim, quando a última sub-tarefa termina, a variável S já contém o valor da soma total.

Esta estratégia, embora simples, levanta a questão de várias tarefas poderem aceder simultaneamente a uma variável comum, sem que se perca a integridade ou validade do seu conteúdo. A "atualização" do valor de uma variável genérica X, ou seja, uma modificação que depende do seu atual valor, envolve, regra geral, três passos:

- 1) Ler o valor atual de X
- 2) Com base no valor lido, calcular o novo valor de X
- 3) Escrever o novo valor de X

Um problema pode ocorrer se entre o passo 1 e 2 ou entre o passo 2 e 3 a tarefa correntemente em execução é suspensa e escalonada uma outra (ou outras) que vão alterar com sucesso o valor de X. Quando a tarefa anterior retoma a execução no passo 2 ou 3, o valor de X lido anteriormente já não é válido e a variável X é atualizada com um valor errado, provocando o funcionamento errado do programa.

No programa pth06.c, a variável global S é atualizada em cada tarefa pela linha de código,

```
S+= soma ( . . . ) ;
```

Esta linha de código após compilação do programa, geram a nível do código máquina algo parecido com o seguinte Pseudo Código Máquina,

```
P= soma ( . . . )  
1) R= S  
2) R= R+P  
3) S= R
```

onde R representa um registo do CPU e P,S o conteúdo de posições de memória que implementam variáveis. Como se pode ver, esta situação é análoga à descrita anteriormente com S no lugar de X. Devido à rapidez de execução do CPU, a execução dos passos 1 a 3 é extremamente rápida, fazendo com que a probabilidade de ocorrência do problema descrito seja extremamente baixa, mas possível, podendo levar a corrupção de bases de dados, etc.

**L3.11** Compile o programa `pth06.c` e execute-o várias vezes com 15 ou 30 tarefas. Verifique se em alguma das vezes obtém um resultado para a soma diferente. Dada a baixa probabilidade de ocorrência do erro, se obteve um valor diferente para a soma, parabéns, foi premiado!

Analise agora o programa exemplificativo `pth07.c`. De modo a aumentar a probabilidade de ocorrência de erro, foi introduzida uma pequena modificação relativamente ao programa anterior. Na tarefa, a linha de código,

```
S+= soma ( a->x, a->n );
```

foi substituída pelo código logicamente equivalente,

```
L= S;  
S= L+soma ( a->x, a->n );
```

onde se recorreu a uma variável auxiliar L. Assim o período de tempo que decorre desde que a variável S é lida até que é escrita com o novo valor é aumentado (por um tempo igual ao necessário para efetuar a `soma()`), aumentando assim a probabilidade que outra tarefa seja escalonada a meio da operação e altere a variável S, gerando um erro no cálculo da soma total.

**L3.12** Compile o programa `pth07.c`. Execute-o várias vezes e vá aumentando progressivamente o número de tarefas 2,4,8,... até que consiga observar um valor errado no resultado final. Imagine agora que se tratava de um saldo bancário ou de o número de registos numa base de dados. Resultado desastroso!!!

A solução para o problema anteriormente descrito, quando uma ou mais tarefas podem simultaneamente alterar/ler o valor de uma variável comum (ou estruturas de dados), é forçar a exclusão mútua (apenas uma tarefa de cada vez) no acesso à variável, quer para escrita, quer para leitura.

Para assegurar a exclusão mútua é utilizado um objeto de sincronização denominado mutex (de Mutual Exclusion), implementado na sua forma mais simples por 4 funções básicas, cujos protótipos se mostram a seguir,

```
#include <pthread.h>  
int pthread_mutex_init(pthread_mutex_t *mutex,  
    pthread_mutexattr_t * attr);  
int pthread_mutex_lock(pthread_mutex_t *mutex);  
int pthread_mutex_unlock(pthread_mutex_t *mutex);  
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

O primeiro argumento é um apontador para uma variável que irá servir de identificador do mutex criado, a ser utilizada em posteriores chamadas a funções que envolvam o mutex.

Na função `pthread_mutex_init()` o segundo argumento é um apontador para uma variável com atributos do mutex a criar. No caso de se pretender os atributos por defeito, este argumento toma simplesmente o valor `NULL`. Nesta atividade serão utilizados os atributos por defeito, que correspondem a um mutex básico utilizado na grande maioria dos casos.

A estratégia de utilização destas funções é bastante simples e exemplificada pelo seguinte troço de código de programa,

```
pthread_mutex_t mtx;
/* inicializar com atributos por defeito */
pthread_mutex_init(&mtx, NULL);
...
/* obter acesso ou esperar pela sua vez */
pthread_mutex_lock(&mtx);
... /* região crítica, aceder aqui a estruturas de dados
    comuns */
... /* este código deve ser de execução o mais breve
    possivel */
pthread_mutex_unlock(&mtx);
...
pthread_mutex_destroy(&mtx);
```

Primeiro é criada uma variável do tipo `pthread_mutex_t` que deve ser global a todas as tarefas que a irão utilizar. Seguidamente é inicializada com os atributos desejados através da função `pthread_mutex_init()`. Antes de aceder à estrutura de dados comum, uma tarefa deve invocar a função `pthread_mutex_lock()` e quando termina de aceder deve invocar a função `pthread_mutex_unlock()`. Se este procedimento for respeitado por todas as tarefas, o sistema operativo garante que em cada instante, no máximo uma tarefa está a aceder à estrutura de dados. Se uma tarefa invoca a função `pthread_mutex_lock()` e outra tarefa já estiver a aceder à estrutura de dados, a tarefa é bloqueada até que a outra efectue uma chamada à função `pthread_mutex_unlock()`. Finalmente, quando o mutex já não é mais necessário, deve ser invocada a função `pthread_mutex_destroy()` para que sejam libertados os recursos associados ao mutex (mais concretamente, recursos associados a uma variável tipo `pthread_mutex_t` que tenha sido inicializada).

Analise o programa exemplificativo `pth08.c`. Para efeitos de comparação, este programa é em tudo idêntico ao anterior com a diferença que o acesso à variável `S` é feito entre um par de chamadas a `pthread_mutex_lock()` e `pthread_mutex_unlock()`

```
pthread_mutex_lock(&mtx); /* impor exclusao mutua no acesso
    variavel S */
/* codigo logicamente equivalente a S+= soma(a->x,a->n); */
L= S;
S= L+soma( a->x, a->n); /* aumenta a probabilidade de
    ocorrencia de erro */
pthread_mutex_unlock(&mtx); /* libertar o "acesso" a
    variavel S */
```

**L3.13** Compile o programa `pth08.c` e execute-o várias vezes com vários valores de número de tarefas. Verifique que agora o valor da soma calculado se mantém sempre correto.

Por fim apresenta-se uma versão final correta, `pth09.c`, em que a programação é feita de modo ao troço de código correspondente à região crítica entre o par de chamadas a `pthread_mutex_lock()` e `pthread_mutex_unlock()` seja o mais breve possível de modo ao programa no geral ser mais eficiente, dado que outras tarefas que pretendam aceder à variável `S` têm de esperar menos tempo. O troço de código que acede à variável `S` é agora dado por,

```
P= soma( a->x, a->n); /* pre-calcular soma parcial */
pthread_mutex_lock(&mtx); /* impor exclusao mutua no acesso
    variavel S */
S+= P; /* acesso deve ser o mais breve possivel */
pthread_mutex_unlock(&mtx); /* libertar o "acesso" a
    variavel S */
```

onde a soma parcial correspondente ao sub-vetor é feita fora da região crítica, de modo a que apenas é realizada uma soma simples no interior da região crítica, minimizando-se assim o tempo de espera das outras tarefas.

FIM