Ficha 4 – Posix Threads - Sincronização

2025

{patricio.domingues, carlos.grilo, vitor.carreira, gustavo.reis, rui.ferreira}@ipleiria.pt

1 Introdução

As *threads* de um processo partilham os dados entre si. Como a sua execução é concorrente, é necessário **sincronizá-las** sempre que partilham dados comuns para escrita e leitura (secções críticas). Caso contrário, corre-se o risco de os dados partilhados ficarem corrompidos. Existem duas formas de sincronização de *threads*, os *mutexes* (*mutual exclusion*) e as *variáveis de condição*. Ambas são descritas em pormenor nas secções seguintes.

1.1 Instalação das *man pages* das funções pthreads

A disponibilização das páginas do manual de ajuda eletrónica (*man*) requer a instalação dos pacotes `manpages-posix` e `manpages-posix-dev`. Isso pode ser conseguido através do seguinte comando:

```
sudo apt-get install manpages-posix
sudo apt-get install manpages-posix-dev
```

2 Mutexes

Os *mutexes* não são mais do que semáforos de exclusão mútua (semáforos binários) para que duas ou mais *threads* possam aceder a uma secção crítica de forma mutuamente exclusiva. A utilização de *mutexes* é feita da seguinte forma:

- Inicialização do *mutex*, através da função `pthread_mutex_init` (ou recorrendo à macro `PTHREAD_MUTEX_INITIALIZER`), antes das threads serem criadas;
- Criação das *threads* (partilhando o *mutex*);
- Utilizar a função `pthread_mutex_lock` para entrar numa secção crítica (equivalente à operação *down* de um semáforo);
- Utilizar a função `pthread_mutex_unlock` para sair da secção crítica (equivalente à operação *up* de um semáforo);

- Antes da aplicação terminar, destruir o *mutex* utilizando a função `pthread_mutex_destroy`.

> ***NOTA***
>
> *Um mutex **nunca** deve ser passado **por valor** a uma função porque o resultado dessa cópia será indefinido.*

2.1 Iniciar um *mutex*

```
#include <pthread.h>

int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *mutexattr)
```

A função `pthread_mutex_init` permite inicializar o *mutex* passado por parâmetro. Recebe como parâmetros:

- `mutex` – ponteiro para o *mutex* a inicializar;
- `mutexattr` – estrutura com os atributos pretendidos para o *mutex* (tipo de *mutex*). Caso se pretenda utilizar os valores pré-definidos, passa-se como argumento o valor `NULL`. Caso pretenda alterar o comportamento por omissão, consulte a página do manual:

```
man pthread_mutexattr_init
```

2.1.1 Valores de retorno

- Sucesso – devolve o valor zero.
- Insucesso – devolve um valor positivo contendo o código de erro que seria atribuído à variável `errno`.

2.1.2 Exemplo

```
/* Declaração do mutex */
pthread_mutex_t mutex;

/* Inicia o mutex */
if ((errno = pthread_mutex_init(&mutex, NULL)) != 0) {
    ERROR(C_ERRO_MUTEX,"pthread_mutex_init() failed");
}
```

2.2 Destruir um *mutex*

```
#include <pthread.h>

int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

A função `pthread_mutex_destroy` permite destruir o *mutex* passado por parâmetro:

- `mutex` – ponteiro para o *mutex* a destruir.

2.2.1 Valores de retorno

- Sucesso – devolve o valor zero.
- Insucesso – devolve um valor positivo contendo o código de erro que seria atribuído à variável `errno`.

2.2.2 Exemplo

```
/* Declaração do mutex */
pthread_mutex_t mutex;

/* Inicia o mutex */
if ((errno = pthread_mutex_init(&mutex, NULL)) != 0) {
    ERROR(C_ERRO_MUTEX,"pthread_mutex_init() failed");
}
...

if ((errno = pthread_mutex_destroy(&mutex)) != 0) {
    ERROR(C_ERRO_MUTEX,"pthread_mutex_destroy() failed");
}
```

2.2.3 Lab 1

1. Utilize o código do Lab1 e inclua, nos locais que achar mais adequados, os blocos de código referentes à inicialização e à destruição de um *mutex*.
2. Compile o projeto, execute-o e verifique se não existem quaisquer erros ou *warnings*.

2.3 Acesso a uma secção crítica

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

A função `pthread_mutex_lock` adquire o *mutex* (equivalente à operação *down* de um semáforo). A operação bloqueia a *thread* que chama a função se o *mutex* está ocupado por outra *thread*.

A função `pthread_mutex_unlock` liberta o *mutex* (equivalente à operação *up* de um semáforo). Se existirem outras *threads* bloqueadas à espera da secção crítica, uma delas é desbloqueada e entra na secção crítica (as restantes permanecem bloqueadas).

A função `pthread_mutex_trylock` tenta adquirir o *mutex*. Caso o *mutex* esteja na posse de outra *thread*, a operação não bloqueia e devolve o valor **`EBUSY`**. Caso contrário, o *mutex* fica na posse da *thread* que chamou a função até que a mesma o liberte através da função `pthread_mutex_unlock`.

As funções recebem um único parâmetro:

- *mutex* – **endereço** do *mutex* a adquirir/libertar.

2.3.1 Valores de retorno

- Sucesso – devolve o valor zero.
- Insucesso – devolve um valor positivo contendo o código de erro que seria atribuído à variável `errno`.

2.3.2 Exemplo

```
/* Entra na secção crítica */
if ((errno = pthread_mutex_lock(&mutex)) != 0) {
    ERROR(C_ERRO_MUTEX,"pthread_mutex_lock() failed");
}

/* Secção crítica */

/* Sai da secção crítica */
if ((errno = pthread_mutex_unlock(&mutex)) != 0) {
    ERROR(C_ERRO_MUTEX,"pthread_mutex_unlock() failed");
}
```

2.3.3 Lab 2

1. Compile o projeto do Lab 2. Execute o Lab2 até verificar que a variável `contador` pode ter um dos seguintes valores: 2, 4 ou 6;
2. Explique como é que o valor 4 pode ser o valor final de `contador`;
3. Pretende-se que no final a variável `contador` tenha apenas o **valor 6**. Altere o código de modo a garantir apenas este resultado.

3 Variáveis de condição

Os *mutexes* implementam a sincronização controlando o acesso das *threads* às secções críticas, ou seja, garantem que só uma *thread* acede à secção crítica de cada vez. No entanto, na grande maioria dos problemas de exclusão mútua, o acesso a uma secção crítica está condicionado a um conjunto de circunstâncias. Por exemplo, no caso do problema do produtor/consumidor, a *thread* produtora apenas poderá produzir se o *buffer* partilhado tiver espaço disponível. De forma similar, a *thread* consumidora apenas poderá consumir se existirem dados no *buffer*. Recorrendo a *mutexes*, seriam necessários três *mutexes* para resolver o problema: um *mutex* para garantir exclusão mútua no acesso à secção crítica (*buffer* partilhado) e dois para cada uma das situações de acesso acima mencionadas. Utilizando variáveis de condição, apenas é necessário um *mutex* e uma variável de condição.

As variáveis de condição permitem a sincronização de *threads* no acesso a uma secção crítica baseada numa determinada condição. Ou seja, uma *thread* ficará bloqueada numa variável de condição se, apesar de ter conseguido bloquear o *mutex* da secção crítica, a condição para executar o código da secção crítica não se verificar. Retomando o exemplo do produtor e do consumidor, uma *thread* desempenha o papel de produtor (colocando os dados num *buffer* enquanto existir espaço disponível) e uma *thread* retira os dados do *buffer* (quando este tiver elementos para consumir). Neste cenário, a sincronização das *threads* é efetuada de acordo com uma variável de condição - o estado do *buffer* (espaço disponível e elementos a consumir). O produtor fica bloqueado à espera que o *buffer* tenha espaço disponível. E o consumidor fica à espera que o *buffer* tenha elementos para consumir.

A utilização de variáveis de condição procede da seguinte forma:

- Inicialização da condição, através da função `pthread_cond_init`, antes das *threads* serem criadas;
- Criação das *threads* (partilhando a variável de condição e o *mutex*);
- Utilizar a função `pthread_cond_wait` para esperar numa secção crítica que a condição se verifique;
- Utilizar a função `pthread_cond_signal` para notificar **apenas uma** das threads que se encontram à espera da condição. Também se pode utilizar a função `pthread_cond_broadcast` caso se pretenda notificar **todas**

as threads em espera;
- Antes de a aplicação terminar, deve-se destruir a condição utilizando a função `pthread_cond_destroy`.

> ***NOTA***
>
> *As variáveis de condição apenas podem ser utilizadas **dentro de uma secção crítica em conjunto com um mutex**. Uma variável de condição **nunca** deve ser passada por valor porque o resultado dessa cópia será indefinido. Associada a uma variável de condição existem uma ou mais variáveis de controlo que indicam o estado da condição. Por exemplo, no caso do produtor/consumidor, existem duas variáveis de controlo: "buffer tem espaço disponível" e "buffer tem elementos".*

3.1 Inicializar uma variável de condição

```
#include <pthread.h>

int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *cond_attr);
```

A função `pthread_cond_init` permite inicializar a condição passada por parâmetro. Recebe os seguintes parâmetros:

- `cond` - ponteiro para a condição a inicializar;
- `cond_attr` - estrutura com os atributos pretendidos para a condição (tipo de *mutex*). Atualmente este parâmetro é ignorado pela norma POSIX. Deve passar-se como argumento, o valor `NULL` (man `pthread_condattr_init`).

3.1.1 Valores de retorno

- Sucesso – devolve o valor zero.
- Insucesso – devolve um valor positivo contendo o código de erro que seria atribuído à variável `errno`.

3.2 Destruir uma variável de condição

```
#include <pthread.h>

int pthread_cond_destroy(pthread_cond_t *cond);
```

A função `pthread_cond_destroy` destrói a condição passada por parâmetro. Recebe um único parâmetro:

- `cond` – ponteiro para a condição a destruir.

3.2.1 Valores de retorno

- Sucesso – devolve o valor zero.
- Insucesso – devolve um valor positivo contendo o código de erro que seria atribuído à variável `errno`.

3.2.2 Lab 3

1. Utilize o código do Lab 3 e inclua, nos locais que achar mais adequados, os blocos de código referentes à inicialização e à destruição de uma variável de condição.

2. Compile o projeto, execute-o e verifique se não existem quaisquer erros ou *warnings*.

3.3 Esperar pela condição indefinidamente

```
#include <pthread.h>

int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mut);
```

A função `pthread_cond_wait` bloqueia a *thread* atual enquanto não se verificar a condição pela qual está à espera. Esta função só pode ser chamada caso a *thread* tenha adquirido o *mutex* `mut`. Antes de bloquear a *thread*, a função liberta o *mutex* passado por parâmetro. Parâmetros recebidos:

- `cond` – ponteiro para a variável de condição;
- `mut` – ponteiro para o mutex utilizado na secção crítica.

3.3.1 Valores de retorno

- Sucesso – devolve o valor zero.
- Insucesso – devolve um valor positivo contendo o código de erro que seria atribuído à variável `errno`.

3.3.2 Lab 4

1. Analise o código do Lab 4 sem o executar e indique qual o *output* do programa.

2. Compile e execute o projeto do Lab 4 e verifique se respondeu de forma correta na alínea anterior.

3.4 Espera, reverificação e ciclo

Como foi referido anteriormente, a função `pthread_cond_wait`, liberta o *mutex* antes de bloquear a *thread*, readquirindo-o assim que a *thread* for desbloqueada. O pseudo-código na Listagem 1 ilustra o comportamento **interno** desta função:

Listing 1: Pseudo-código para função pthread_cond_wait

```
1 pthread_cond_wait(cond, mut);
2 begin
3     pthread_mutex_unlock(mut);
4     block_on_cond(cond);
5     pthread_mutex_lock(mut);
6 end
```

Listing 2: Ciclo e função pthread_cond_wait

```c
/* Exemplo do código do produtor.
* Nota: O código para tratamento de erros foi omitido.
*/
/* Entra na secção crítica */
pthread_mutex_lock(&mutex);

/* Espera que o buffer tenha espaço disponível */
while(buffer_cheio) {
    pthread_cond_wait(&cond, &mutex);
}
...
/* Coloca dados no buffer */
...

/* Sai da secção crítica */
pthread_mutex_unlock(&mutex);
```

Adicionalmente, como exemplificado na Listagem 2, a função `pthread_cond_wait` é sempre chamada num ciclo while que verifica novamente a condição pelas seguintes razões:

1. A figura Figura 1 mostra o caso em que um *buffer* está cheio, uma *thread* produtora P1 está a dormir, enquanto outra thread produtora P2 está acordada a processar um item. Quando uma *thread* consumidora C1 chama `pthread_cond_signal` e em seguida `pthread_mutex_unlock(mut)`, tal irá retirar a *thread* produtora P1 da fila de espera, no entanto esse processo leva algum tempo. Entretanto pode ocorrer que a *thread* produtora P2 tenha nesse momento terminado o seu trabalho, e obtenha o *lock*, colocando o item no buffer, fazendo com que este volte a estar cheio, antes que a *thread* P1 sequer tenha acordado. No momento em que finalmente a *thread* P1 começa a correr, o *buffer* já está de novo cheio, e esta é obrigada a dormir novamente.

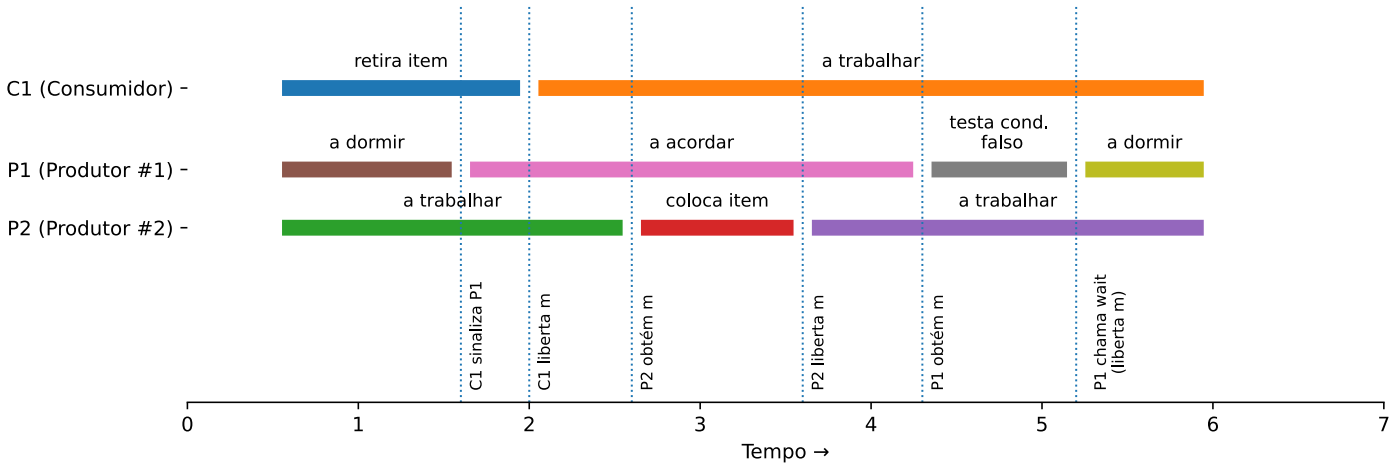


Figure 1: Sequência temporal com variável de condição: a thread P2 adquire o mutex antes da thread P1 acordar.

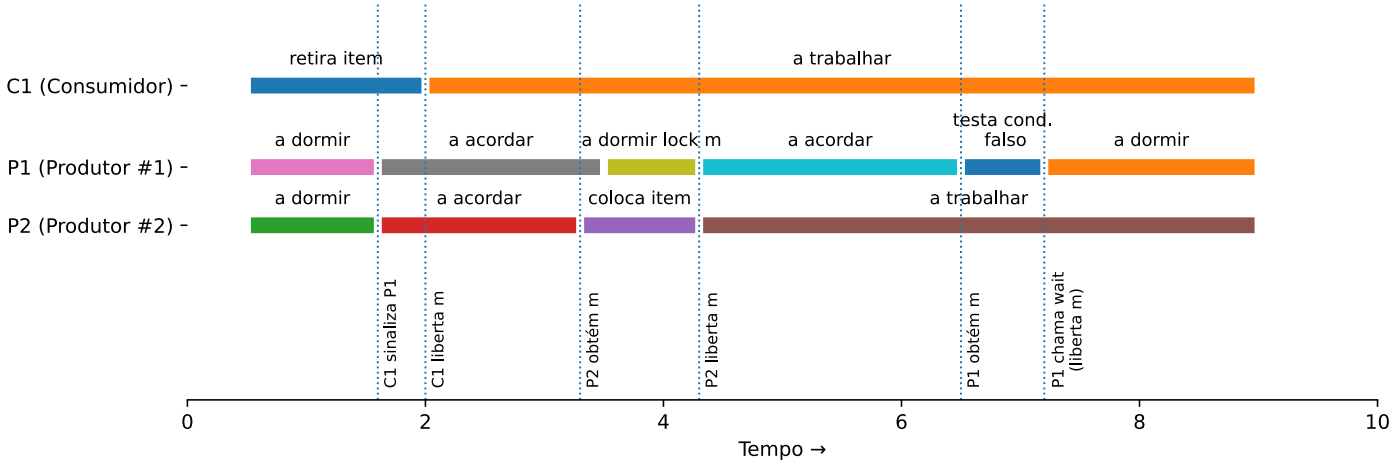


Figure 2: Sequência temporal com variável de condição: as threads P1 e P2 são acordadas via broadcast entrando em contenção por um mutex, com a thread P2 a conseguir a posse do mutex.

2. A Figura 2 mostra o caso em que a thread consumidora usa `pthread_cond_broadcast`. Suponhamos que temos duas *threads* produtoras `P1` e `P2`. Em determinado momento, `P1` e `P2` podem ficar ambas bloqueadas em `block_on_cond(cond)`. Se houver uma *thread* consumidora que consuma um caracter do *buffer*, e esta sinalizar **ambas** as *threads* produtoras usando `pthread_cond_broadcast`, só uma das *threads* `P1` ou `P2` irá adquirir o *mutex* em `pthread_mutex_lock(mut)`, ficando a outra bloqueada nesse mesmo ponto. Vamos assumir que a *thread* `P2` adquire o *mutex* e que a *thread* `P1` fica bloqueada. A *thread* `P2` irá libertar o *mutex* depois de adicionar um novo caracter ao *buffer* (enchendo-o). Isso significa que quando a *thread* `P1` adquirir o *mutex*, o *buffer* já está cheio novamente. Por isso, `P1` tem que voltar a verificar a condição assim que sai da função `pthread_cond_wait`, daí a necessidade de termos a função dentro de um ciclo `while` e não num `if`.

3. Finalmente, a função `pthread_cond_wait` pode retornar sem que a thread tenha sido sinalizado e portanto tenha mudado a condição. Este fenómeno chama-se *spurious wakeup* e trata-se de uma otimização do sistema operativo. Uma vez que pelas razões anteriores a condição tem de ser reverificada, para otimizar a implementação da função o sistema operativo não garante que quando esta retorne tal tenha sido devido a sinalização.

3.5 Espera pela condição por um período limitado

```
#include <pthread.h>

int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mut, const struct timespec
        abstime);
```

A função `pthread_cond_timedwait` bloqueia a *thread* atual por um período de tempo limitado até que a condição pela qual está à espera se verifique. Esta função só pode ser chamada caso a *thread* tenha adquirido o *mutex* `mut`. Antes de bloquear a *thread*, a função liberta o *mutex* passado por parâmetro. Parâmetros recebidos:

- `cond` – ponteiro para a variável de condição;
- `mut` – ponteiro para o *mutex*;
- `abstime` – tempo absoluto de espera (número de segundos desde 1 janeiro de 1970).

3.5.1 Valores de retorno

- Sucesso – devolve o valor zero.
- Insucesso – devolve um valor positivo contendo o código de erro que seria atribuído à variável `errno`.

3.5.2 Lab 5

1. Analise o código do Lab 5 sem o executar e indique qual o *output* do programa.

2. Compile e execute o projeto do Lab 5 e verifique se respondeu de forma correta na alínea anterior.

3.6 Sinalização de uma condição

De seguida mostra-se como se pode sinalizar uma única *thread* ou todas as *threads* simultaneamente.

3.6.1 Sinalização de uma única *thread*

```
#include <pthread.h>

int pthread_cond_signal(pthread_cond_t *cond);
```

A função `pthread_cond_signal` acorda uma *thread* bloqueada na variável de condição `cond`. Recorde-se que cada *thread* readquire o *mutex* antes que possa retornar, de modo que as várias *threads* possam sair da função `pthread_cond_wait` de forma sequencial (uma de cada vez). Esta função recebe um único parâmetro:

- `cond` – ponteiro para a variável de condição a sinalizar.

3.6.2 Lab 6

1. Analise o código do Lab 6 e faça um esquema onde seja representada a alocação de memória do processo principal (*main thread*) e respetiva utilização por parte das 4 *threads*.

   Nota 1: neste exemplo é utilizada a mesma função para todas as *threads*;

   Nota 2: é utilizado um identificador para distinguir o código que é específico a cada *thread*.

2. Ainda sem executar o programa, qual é o *output*?

3. Compile e execute o programa (mais do que uma vez) e observe os resultados.

3.6.3 Sinalização de todas as *threads*

```
#include <pthread.h>

int pthread_cond_broadcast(pthread_cond_t *cond);
```

A função `pthread_cond_broadcast` acorda todas as *threads* bloqueadas na variável de condição `cond`. Recorde-se que cada *thread* readquire o *mutex* antes que possa retornar. Assim, apenas uma única *thread* irá retomar a sua execução de cada vez.

- `cond` – ponteiro para a variável de condição a sinalizar;

3.6.4 Valores de retorno para ambas as funções de sinalização

- Sucesso – devolve o valor zero.
- Insucesso – devolve um valor positivo contendo o código de erro que seria atribuído à variável `errno`.

3.6.5 Lab 7

1. Modifique o código do Lab 7 para que, desta vez, todas as *threads* sejam sinalizadas.

2. Qual a diferença para os resultados obtidos no Lab 6?

3.7 Exemplo: produtor/consumidor

Listing 3: Produtor Consumidor

```c
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>
#include "debug.h"

#define C_ERRO_PTHREAD_CREATE 1
#define C_ERRO_PTHREAD_JOIN 2
#define C_ERRO_MUTEX_INIT 3
#define C_ERRO_MUTEX_DESTROY 4
#define C_ERRO_CONDITION_INIT 5
#define C_ERRO_CONDITION_DESTROY 6

#define MAX 5     /* Capacidade do buffer */
#define LIMITE 20 /* Total de elementos a produzir */

typedef struct {
    int buffer[MAX];
    int index_leitura;
    int index_escrita;
    int total;
    pthread_mutex_t mutex;
    pthread_cond_t cond;
} thread_params_t;

/* protótipos */
void *produtor(void *arg);
void *consumidor(void *arg);

/* main */
int main(int argc, char *argv[]) {
    pthread_t t1, t2;
    thread_params_t thread_params;

    (void)argc;
    (void)argv;

    /* Inicia o mutex */
    if ((errno = pthread_mutex_init(&thread_params.mutex, NULL)) != 0)
        ERROR(C_ERRO_MUTEX_INIT, "pthread_mutex_init() failed!");

    /* Inicia variavel de condicao */
    if ((errno = pthread_cond_init(&thread_params.cond, NULL)) != 0)
        ERROR(C_ERRO_CONDITION_INIT, "pthread_cond_init() failed!");

    /* Inicia os restantes parametros a passar 'as threads */
    memset(thread_params.buffer, 0, sizeof(thread_params.buffer));
    thread_params.total = 0;
    thread_params.index_escrita = 0;
    thread_params.index_leitura = 0;

    /* Cria thread para executar o consumidor */
    if ((errno = pthread_create(&t1, NULL, consumidor, &thread_params)) != 0)
        ERROR(C_ERRO_PTHREAD_CREATE, "pthread_create() failed!");

    /* Cria thread para executar o produtor */
    if ((errno = pthread_create(&t2, NULL, produtor, &thread_params)) != 0)
```

```
60          ERROR(C_ERRO_PTHREAD_CREATE, "pthread_create() failed!");
61
62      /* Espera que todas as threads terminem */
63      if ((errno = pthread_join(t1, NULL)) != 0)
64          ERROR(C_ERRO_PTHREAD_JOIN, "pthread_join() failed!");
65
66      if ((errno = pthread_join(t2, NULL)) != 0)
67          ERROR(C_ERRO_PTHREAD_JOIN, "pthread_join() failed!");
68
69      /* Destroi o mutex */
70      if ((errno = pthread_mutex_destroy(&thread_params.mutex)) != 0)
71          ERROR(C_ERRO_MUTEX_DESTROY, "pthread_mutex_destroy() failed!");
72
73      /* Destroi a condicao */
74      if ((errno = pthread_cond_destroy(&thread_params.cond)) != 0)
75          ERROR(C_ERRO_CONDITION_DESTROY, "pthread_cond_destroy() failed!");
76
77      return 0;
78  }
79
80  /* produtor */
81  void *produtor(void *arg) {
82      thread_params_t *params = (thread_params_t *)arg;
83      int i;
84
85      for (i = 0; i < LIMITE; i++) {
86
87          /* Adormece entre 0 a 4 segundos */
88          int num = random() % 100 + 1;
89          /* simula tempo de processamento para criar número */
90          sleep(random() % 5);
91
92          if ((errno = pthread_mutex_lock(&(params->mutex))) != 0) {
93              WARNING("pthread_mutex_lock() failed\n");
94              return NULL;
95          }
96
97          /* Espera que o buffer tenha espaco disponivel */
98          while (params->total == MAX)
99              if ((errno = pthread_cond_wait(&(params->cond),
100                                             &(params->mutex))) != 0) {
101                 WARNING("pthread_cond_wait() failed");
102                 return NULL;
103             }
104
105         /* Coloca um valor no buffer */
106         params->buffer[params->index_escrita] = num;
107         printf(">> %d\n", params->buffer[params->index_escrita]);
108         params->index_escrita = (params->index_escrita + 1) % MAX;
109         params->total++;
110
111         /* Notifica consumidores 'a espera */
112         if (params->total == 1)
113             if ((errno = pthread_cond_signal(&(params->cond))) != 0) {
114                 WARNING("pthread_cond_signal() failed");
115                 return NULL;
116             }
117
118         /* Sai da seccao critica */
119         if ((errno = pthread_mutex_unlock(&(params->mutex))) != 0) {
```

```
120             WARNING("pthread_mutex_unlock() failed");
121             return NULL;
122         }
123 
124 
125     }
126     return NULL;
127 }
128 
129 /* consumidor */
130 void *consumidor(void *arg) {
131     thread_params_t *params = (thread_params_t *)arg;
132     int i;
133 
134     for (i = 0; i < LIMITE; i++) {
135         if ((errno = pthread_mutex_lock(&(params->mutex))) != 0) {
136             WARNING("pthread_mutex_lock() failed\n");
137             return NULL;
138         }
139 
140         /* Espera que o buffer tenha dados */
141         while (params->total == 0)
142             if ((errno = pthread_cond_wait(&(params->cond),
143                                            &(params->mutex))) != 0) {
144                 WARNING("pthread_cond_wait() failed");
145                 return NULL;
146             }
147 
148         /* Retira um valor no buffer */
149         int num = params->buffer[params->index_leitura];
150         printf("       %d >>\n", num);
151         params->index_leitura = (params->index_leitura + 1) % MAX;
152         params->total--;
153 
154         /* Notifica produtores 'a espera */
155         if (params->total == MAX - 1)
156             if ((errno = pthread_cond_signal(&(params->cond))) != 0) {
157                 WARNING("pthread_cond_signal() failed");
158                 return NULL;
159             }
160 
161         /* Sai da seccao critica */
162         if ((errno = pthread_mutex_unlock(&(params->mutex))) != 0) {
163             WARNING("pthread_mutex_unlock() failed");
164             return NULL;
165         }
166 
167         /* Adormece entre 0 a 4 segundos
168            simula processamento a ser efetuado usando num */
169         sleep(random() % 5);
170     }
171     return NULL;
172 }
```

3.7.1 Lab 8

1. Analise o código do Lab 8, e indique qual o *output* deste programa.

2. Compile e execute o projeto do Lab 8 de modo verificar se percebeu o conceito de produtor/consumidor.

4 Funções *thread-safe* e reentrantes

As funções *thread-safe* são funções que garantem que apenas uma *thread* de cada vez executa a secção crítica da função. Desta forma, o seu resultado não depende da imprevisível ordem de execução das várias *threads*. Uma função reentrante é uma função que não guarda dados estáticos para serem utilizados em chamadas sucessivas. Todos os dados necessários à execução da função são passados por parâmetro. A função `strtok` é um exemplo de uma função **não reentrante**. O conceito de reentrância também se pode aplicar aos *mutexes*. Um *mutex* reentrante permite que a mesma *thread* o adquira múltiplas vezes. Caso o *mutex* não seja reentrante, pode levar a uma situação de *auto-deadlock* de uma *thread* (ex. funções recursivas com secções críticas). Os *mutexes* da norma `POSIX 1003.1c` **não são reentrantes**.

Quando as normas `ANSI C` e `POSIX 1003.1c` (1990) foram criadas, não tiveram em conta as *threads* (dado que estas ainda não se encontravam implementadas no Unix). A maioria das funções definidas nessas normas foram modificadas para se tornarem *thread-safe* sem alterar a sua interface externa (para não comprometer a compatibilidade com código existente). No entanto, para tornar algumas funções *thread-safe*, foi necessário torná-las reentrantes e, logo, alterar a sua interface externa:

- Funções que devolvem ponteiros para *buffers* estáticos internos, como por exemplo `ctime`;
- Funções que requerem um contexto estático entre uma série de várias chamadas, como por exemplo `strtok`;

Na sequência da criação da API *pthreads*, foram criadas nestes casos variantes das funções existentes, que são reentrantes, designadas pelo sufixo `_r` no nome das funções. Assim, as versões reentrantes dos exemplos apresentados são `ctime_r` e `strtok_r`. Quando pesquisamos informação acerca de uma função nas páginas do manual e existe uma versão reentrante, a mesma página faz referência a ambas as versões.

5 *Threads* e sinais

Os sinais (SIGxxx) são tratados ao nível dos processos, isto quer dizer que o sinal pode ser entregue a qualquer uma das *threads* de um processo. Por exemplo, se o sinal SIGCHLD for gerado pelo término de um processo filho, este pode ser entregue a uma *thread* diferente daquela que criou o processo. A exceção a esta regra são os sinais que estão relacionados com erros de hardware (SIGFPE, SIGSEGV, SIGTRAP, SIGPIPE), que são entregues à *thread* que originou o sinal.

Os sinais afetam os processos mesmo quando estes têm várias *threads*. Por exemplo, se enviarmos o sinal SIGKILL a um processo ele vai terminar independentemente de ter *threads* ou não. Isto também se aplica ao comportamento por omissão dos sinais que não são tratados, ou seja, o processo termina, bem como todas as *threads* a si associadas. É possível uma *thread* enviar sinais a outra *thread*, desde que se encontre dentro do mesmo processo, através da função `pthread_kill`. Se pretender aprofundar os conhecimentos nesta matéria deve consultar [1][2][3][4].

6 Exercícios

> ***NOTA***
>
> *Na resolução de cada exercício deve utilizar o template de exercícios que inclui uma `makefile` bem como todas as dependências necessárias à compilação.*

6.1 Para a aula

1. Altere a resolução do exercício 2 da Ficha 3 de modo a que o resultado da soma seja o correto.

2. Elabore o programa `ping-pong` que crie duas *threads*. Uma das *threads* escreve para o ecrã a palavra **“ping…”** enquanto a outra escreve a palavra **“pong!”**. A escrita deve ser alternada e devidamente sincronizada recorrendo a um *mutex* e uma variável de condição. O programa deve terminar ao fim de 10 sequências **“ping…pong!”**.

6.2 Extra-aula

3. Elabore o programa `letter-histogram` que calcula a frequência de cada letra do alfabeto (total de 26 letras) encontradas num ficheiro de texto. O programa deve paralelizar o processo recorrendo a *threads*. Cada *thread* deve processar o ficheiro em blocos de *N* bytes até que não existam mais blocos a processar. Por exemplo, com duas *threads*, um bloco de tamanho 100 e um ficheiro com 450 bytes, a distribuição seria a seguinte:

Thread	Bytes a ler no ficheiro
`Thread #1`	0-99,200-299,400-450
`Thread #2`	100-199, 300-399

O programa deve receber como parâmetro de entrada o número de *threads* a serem utilizadas, o ficheiro a processar, bem como o tamanho de cada bloco. Na página Moodle da UC encontra um ficheiro a utilizar nos testes. Exemplo:

```
bash-4.3$ ./letter-histogram -t 4 -f ../les-miserables.txt -b 2048
a:207145
b:37461
c:67300
...
y:39223
z:1906
```

> ***NOTA***
>
> *A existência de race conditions vai depender da forma como o exercício é resolvido. Caso existam, deve minimizar os pontos de contenção.*

4. Elabore o programa `find-pdf` que, recorrendo à biblioteca *pthreads*, deve encontrar todos os ficheiros do tipo PDF a partir de uma dada pasta, incluindo subpastas. A implementação deve seguir o paradigma produtor-consumidor, existindo uma *thread* produtora e um número N de *threads* consumidoras. A *thread* produtora é responsável por distribuir pelas *threads* consumidoras todos os ficheiros a analisar. Cada *thread* consumidora deve abrir o ficheiro e verificar se os primeiros 4 bytes correspondem à sequência `0x25 0x50 0x44 0x46`. A pasta onde a pesquisa é iniciada bem como o número N de *threads* consumidoras devem ser especificadas através da linha de comandos. Exemplo:

```
bash-4.3$ ./find-pdf -f /home/pa -t 2
/home/pa/Ficha01---Controlo_de_Processos/EI_2016-17_Ficha1_PUB_v1.1b.pdf
/home/pa/Ficha02---signals/EI_2016-17_Ficha2_PUB_v1.1.pdf
/home/pa/gengetopt/Revista_PROGRAMAR_43_gengetopt.pdf
Elapsed time: 0.889 ms
```

5. Um conjunto de pessoas são responsáveis pela preparação de tostas. Infelizmente, cada pessoa só sabe executar uma tarefa sendo necessário coordenar os esforços de cada uma. Na preparação de uma tosta são utilizados os seguintes ingredientes:

   - 2 fatias de pão;
   - 1 fatia de fiambre;
   - 2 fatias de queijo.

   Instruções a utilizar na preparação:

   - Cortar 2 fatias de pão;
   - Colocar a 1ª fatia de queijo;
   - Colocar a fatia de fiambre;
   - Colocar a 2ª fatia de queijo;
   - Colocar a tosta na tostadeira, esperar entre 1 a 3 segundos e embalar.

   Restrições:

   - Só existe uma tostadeira;
   - Só existe uma mesa para a preparação das tostas, a mesa fica livre quando a tosta se encontra na tostadeira ou foi embalada;
   - Só é possível embalar uma tosta após esta ter saído da tostadeira;
   - É necessário respeitar a ordem de preparação (i.e., o pão tem de estar cortado e o fiambre fica entre as duas fatias de queijo).

   Elabore o simulador `toast-academy` recorrendo a 4 threads (uma por pessoa):

   - Thread #1 – responsável por cortar o pão;
   - Thread #2 – responsável por colocar o queijo;
   - Thread #3 – responsável por colocar o fiambre;
   - Thread #4 – responsável por finalizar a tosta (colocar na tostadeira e embalar).

   O simulador não deve bloquear e deve permitir o embalamento de uma tosta no máximo a cada 3 segundos. Exemplo de output:

```
bash-4.3$ ./toast-academy
Bread sliced!
Putting cheese slice #1
Putting ham slice
Putting cheese slice #2
Toasting....
Bread sliced!
Putting cheese slice #1
Putting ham slice
Putting cheese slice #2
2016-10-09 20:13:42: Toast is ready!!
Toasting....
Bread sliced!
Putting cheese slice #1
Putting ham slice
Putting cheese slice #2
2016-10-09 20:13:44: Toast is ready!!
Toasting....
Bread sliced!
```

6. O jogo da vida é um autómato celular desenvolvido pelo matemático britânico John Horton Conway em 1970 [5]. O jogo é uma simulação que ocorre numa grelha de duas dimensões composta por células. A simulação decorre aplicando em cada geração as seguintes regras:

   - Qualquer célula viva com menos de dois vizinhos vivos morre de solidão;
   - Qualquer célula viva com mais de três vizinhos vivos morre de sobrepopulação;
   - Qualquer célula morta com exatamente três vizinhos vivos transforma-se numa célula viva;
   - Qualquer célula viva com dois ou três vizinhos vivos continua no mesmo estado para a próxima geração.

   Elabore o programa **`game-of-life`** que, utilizando a biblioteca *pthreads*, deve simular o jogo da vida acima descrito. O programa recebe um ficheiro contendo a grelha inicial (tamanho MxM), o número de *threads* a utilizar (N) e o número de gerações a gerar (G). Cada *thread* deve ser responsável pelo processamento de um bloco de tamanho M/N linhas. Isto é, se o tamanho da grelha for 5x5 e o número de *threads* for 2, a 1ª thread processa as linhas 0, 1 e 2, e a 2ª *thread* processa as linhas 3 e 4.

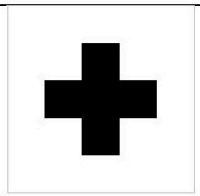
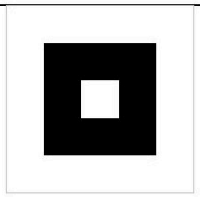
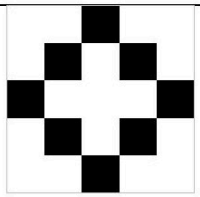
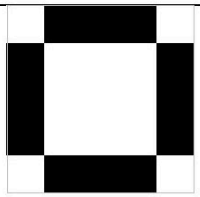
   Para cada geração (incluindo a geração 0), o programa deve criar um ficheiro com o formato YYYYMMDD-HHMMSS-I com a configuração da grelha. As letras correspondem, respetivamente, a ano (4 dígitos), mês, dia, hora, minutos e número da geração. A data para a criação do ficheiro deve ser obtida quando o programa é iniciado (todas as gerações irão ter a mesma data).

   O formato do ficheiro inicial, bem como o das gerações é o seguinte:

   - 1ª e única linha – contém a configuração da grelha. Cada célula é representada pelo valor 1 (célula viva) ou pelo valor 0 (célula morta). A primeira célula corresponde à posição (0, 0). Não existe qualquer separador. O tamanho da grelha é dado pela raiz quadrada do número de células (assume-se que a grelha é sempre quadrada).

   Exemplo para uma grelha de 5x5 que corresponde à configuração de uma cruz.
   `0000000100011100010000000`

   A imagem seguinte mostra o estado da grelha nas próximas 4 gerações:

Geração 0	Geração 1	Geração 2	Geração 3
			

Imagens das gerações

O programa termina quando o número de gerações geradas for atingido. Para efeitos de teste, pode utilizar os ficheiros disponibilizados no Moodle. Encontrará também código fonte para gerar uma imagem animada a partir dos ficheiros criados pelo programa. Em alternativa em https://bitstorm.org/gameoflife/ poderá comparar o estado expectável na grelha em cada geração.

NOTA

Uma thread só pode passar para a próxima geração se todas as threads tiverem terminado a geração atual. Se assim entender, pode utilizar as barreiras de sincronização disponibilizadas pela biblioteca pthreads.

7 Bibliografia

[1] D. Butenhof, “Programming with POSIX Threads”, Addison-Wesley.

[2] https://computing.llnl.gov/tutorials/pthreads/

[3] http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html

[4] K. Robbins and S. Robbins, “Unix Systems Programming”, Prentice-Hall (capítulos 12 e 13).

[5] https://pt.wikipedia.org/wiki/Jogo_da_vida