

Ficha 4 – Posix Threads – Sincronização

Tópicos abordados:

- *Mutexes*
- Variáveis de condição

Duração prevista: 2 aulas

©2019: {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 *pthread*s

A disponibilização das páginas do manual de ajuda eletrónica (*man*) requer a instalação dos pacotes **manpages-posix** e **manpages-posix-dev**. Tal 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:

- Iniciação 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 iniciar o *mutex* passado por parâmetro. Recebe como parâmetros:

- `mutex` – ponteiro para o *mutex* a iniciar;
- `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 de 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");
}
```

Exemplo 1 – Iniciação de um *mutex*

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

```
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");
}
```

Exemplo 2 – Destruição de um *mutex*

Lab 1

Utilize o código do *Lab 1* e inclua, nos locais que achar mais adequados, os blocos de código referentes à inicialização e à destruição de um *mutex*.

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). Acorda todas as *threads* que se encontrem bloqueadas no *mutex*.

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");
}
```

Exemplo 3 – Acesso a uma secção crítica

Lab 2

- a) 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;
- b) Explique quando como é que o valor 4 pode ser o valor final de “contador”;
- c) Pretende-se que a variável “contador” tenha apenas o seguinte 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 único *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:

- Iniciação 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. Iniciar 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 iniciar a condição passada por parâmetro.

Recebe os seguintes parâmetros:

- `cond` – ponteiro para a condição a iniciar;
- `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*.

Lab 3

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.

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. Exemplo

```
/* 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);
```

Exemplo 4 – Espera pela condição

Lab 4

- a) Analisando apenas o código do *Lab 4*, indique qual o *output* do programa.
- b) Compile e execute o projeto do *Lab 4* e verifique se respondeu de forma correta na alínea anterior.

3.3.3. Especificidades

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 seguinte ilustra o comportamento interno desta função:

```
pthread_cond_wait (cond, mut)
begin
    pthread_mutex_unlock(mut);
    block_on_cond(cond);
    pthread_mutex_lock(mut);
end
```

Exemplo 5 – Pseudo-código da função `pthread_cond_wait`

Repare-se que a função liberta o *mutex* antes de bloquear (*block_on_cond*), readquirindo o *mutex* antes de terminar. A reaquisição do *mutex* pode bloquear (outra *thread* readquiriu-o), pelo que se torna necessário verificar a condição pela qual a *thread* foi assinalada após a execução de `pthread_cond_wait` (razão pela qual se verifica a

condição num ciclo *while*). As variáveis de condição servem apenas para sinalizar *threads* e não para fazer exclusão mútua. Por isso, é necessário especificar o *mutex* associado à variável de condição.

3.4. 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).

3.4.1. Valores de retorno

Sucesso – devolve o valor zero.

Insucesso – devolve um valor positivo contendo o código de erro que seria atribuído à variável *errno*.

Lab 5

- Analizando apenas o código do *Lab 5*, indique qual o *output* do programa.
- Compile e execute o projeto do *Lab 5* e verifique se respondeu de forma correta na alínea anterior.

3.5. Sinalização de uma condição

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

3.5.1. 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.

Lab 6

a) 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*.

b) Ainda sem executar o programa: qual é o *output*?

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

3.5.2. 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;

3.5.3. 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*.

Lab 7

- a) Modifique o código do Lab 7 para que, desta vez, todas as *threads* sejam sinalizadas.
- b) Qual a diferença para os resultados obtidos no *Lab 6*?

3.6. Exemplo: produtor/consumidor

```
#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_COND_INIT          5
#define C_ERRO_COND_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;
} PARAM_T;

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

int main(int argc, char *argv[]) {
    pthread_t t1, t2;
    PARAM_T param;

    (void)argc; (void)argv;

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

    /* Inicia variavel de condicao */
    if ( (errno = pthread_cond_init(&param.cond, NULL)) != 0 ) {
        ERROR(C_ERRO_COND_INIT, "pthread_cond_init() failed!");
    }

    /* Inicia os restantes parametros a passar 'as threads */
    memset(param.buffer, 0, sizeof(param.buffer));
```

```

param.total = 0;
param.index_escrita = 0;
param.index_leitura = 0;

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

/* Cria thread para executar o produtor */
if ((errno=pthread_create(&t2,NULL,produtor,&param)) != 0) {
    ERROR(C_ERRO_PTHREAD_CREATE, "pthread_create() failed!");
}

/* Espera que todas as threads terminem */
if ( (errno = pthread_join(t1, NULL)) != 0) {
    ERROR(C_ERRO_PTHREAD_JOIN, "pthread_join() failed!");
}

if ( (errno = pthread_join(t2, NULL)) != 0) {
    ERROR(C_ERRO_PTHREAD_JOIN, "pthread_join() failed!");
}

/* Destroi o mutex */
if ( (errno = pthread_mutex_destroy(&param.mutex)) != 0) {
    ERROR(C_ERRO_MUTEX_DESTROY, "pthread_mutex_destroy failed!");
}

/* Destroi a condicao */
if ( (errno = pthread_cond_destroy(&param.cond)) != 0) {
    ERROR(C_ERRO_COND_DESTROY, "pthread_cond_destroy failed!");
}
return 0;
}

void *produtor(void *arg) {
    PARAM_T *p = (PARAM_T *) arg;
    int i;

    for (i = 0; i < LIMITE; i++) {
        if ( (errno = pthread_mutex_lock(&(p->mutex))) != 0) {
            WARNING("pthread_mutex_lock() failed");
            return NULL;
        }

        /* Espera que o buffer tenha espaco disponivel */
        while (p->total == MAX)
            if ((errno=pthread_cond_wait(&(p->cond), &(p->mutex))) != 0) {
                WARNING("pthread_cond_wait() failed");
                return NULL;
            }

        /* Coloca um valor no buffer */
        p->buffer[p->index_escrita] = random() % 100 + 1;
        printf(">> %d\n", p->buffer[p->index_escrita]);
        p->index_escrita = (p->index_escrita + 1) % MAX;
    }
}

```

```

    p->total++;

    /* Notifica consumidores 'a espera */
    if (p->total == 1)
        if ((errno = pthread_cond_signal(&(p->cond))) != 0) {
            WARNING("pthread_cond_signal() failed");
            return NULL;
        }

    /* Sai da seccao critica */
    if ((errno=pthread_mutex_unlock(&(p->mutex))) != 0) {
        WARNING("pthread_mutex_unlock() failed");
        return NULL;
    }

    /* Adormece entre 0 a 4 segundos */
    sleep(random() % 5);
}
return NULL;
}

void *consumidor(void *arg) {
    PARAM_T *p = (PARAM_T *) arg;
    int i;

    for (i = 0; i < LIMITE; i++) {
        if ((errno = pthread_mutex_lock(&(p->mutex))) != 0) {
            WARNING("pthread_mutex_lock() failed");
            return NULL;
        }

        /* Espera que o buffer tenha dados */
        while (p->total == 0)
            if ((errno=pthread_cond_wait(&(p->cond), &(p->mutex)) != 0) {
                WARNING("pthread_cond_wait() failed");
                return NULL;
            }

        /* Retira um valor no buffer */
        printf("<< %d\n", p->buffer[p->index_leitura]);
        p->index_leitura = (p->index_leitura + 1) % MAX;
        p->total--;

        /* Notifica produtores 'a espera */
        if (p->total == MAX-1)
            if ((errno = pthread_cond_signal(&(p->cond))) != 0) {
                WARNING("pthread_cond_signal() failed");
                return NULL;
            }

        /* Sai da seccao critica */
        if ((errno=pthread_mutex_unlock(&(p->mutex))) != 0) {
            WARNING("pthread_mutex_unlock() failed");
            return NULL;
        }
    }
}

```

```
/* Adormece entre 0 a 4 segundos */  
sleep(random() % 5);  
}  
return NULL;  
}
```

Listagem 1 – Produtor/Consumidor utilizando variáveis de condição

Lab 8

- a) Analise o código do *Lab 8*, e indique qual o *output* deste programa.
- b) 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 as funções reentrantes, foi necessário alterar a interface externa das funções:

- 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*;

Nestes casos, a API *pthread*s definiu 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* associadas ao processo).

No entanto, 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 deverá utilizar o *template* de exercícios que inclui uma *makefile* bem como todas as dependências necessárias à compilação.

6.1. Para a aula

1. Elabore o programa **race-condition** que deverá recorrer a 200 *threads* para incrementar 1000 vezes um contador partilhado (tipo *int*). No final o programa deverá escrever para o ecrã o valor do contador.

Se o contador começar em zero, o valor final é sempre 200000? Execute várias vezes o programa para comprovar os resultados. Altere o programa por forma a garantir que o resultado final é sempre 200000.

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 deverá ser alternada e devidamente sincronizada recorrendo a 1 *mutex* e uma variável de condição. O programa deverá 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 deverá paralelizar o processo recorrendo a *threads*. Cada *thread* deverá processar o ficheiro em blocos de *N* bytes até que não existam mais blocos a processar. Por exemplo, com 2 *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 encontrará 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, deverá minimizar os pontos de contenção.

4. Elabore o programa **find-pdf** que, recorrendo à biblioteca *pthread*, 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 3s 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 deverá 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 1970¹. 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 superpopulação;
- 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 *pthread*s deverá 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* deverá 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 deverá 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:



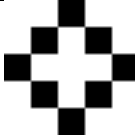
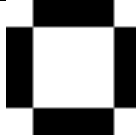
- 1ª e única linha – contém a configuração da grelha. Cada célula é representada pelo valor 1 (célula viva) ou pelo valor 0 (célula morta). A primeira célula corresponde à posição (0,0). Não existe qualquer separador. O tamanho da grelha é dado pela raiz quadrada do número de células (assume-se que a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

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

¹ https://pt.wikipedia.org/wiki/Jogo_da_vida

0000000100011100010000000

As imagens seguintes mostram o estado da grelha nas próximas 4 gerações:

Geração 0	Geração 1	Geração 2	Geração 3
			

O programa termina quando o número de gerações geradas for atingido. Para efeitos teste poderá 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 poderá passar para a próxima geração se todas as *threads* tiverem terminado a geração atual. Se assim entender poderá utilizar as barreiras de sincronização disponibilizadas pela biblioteca *pthread*.

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).