

## Sincronização de *threads*

A sincronização de *threads* pode ser feita recorrendo a:

- **semáforos** (ver cap. anterior)
- **mutexes**
  - podem ser vistos como semáforos inicializados em 1 servindo fundamentalmente p/ garantir a exclusão mútua de secções críticas
- **condition variables** (variáveis de condição)
  - permitem que um *thread* aceda a uma secção crítica apenas quando se verificar uma determinada condição sem necessidade de ficar a ocupar o processador para testar essa condição; enquanto ela não se verificar o *thread* fica bloqueado

Estes 2 últimos mecanismos de sincronização foram introduzidos pela norma POSIX que definiu a *API* de utilização de *threads*.



## *Mutexes*

Sequência típica de utilização de um *mutex*:

- Criar e inicializar a variável do *mutex*.
- Vários *threads* tentam trancar (*lock*) o *mutex*.
- Só um deles consegue. Esse passa a ser o dono do *mutex*.
- O dono do *mutex* executa as instruções da secção crítica.
- O dono do *mutex* destranca (*unlock*) o *mutex*.
- Outro *thread* adquire o *mutex* e repete o processo.
- ...
- Finalmente, o *mutex* é destruído



## Mutexes - Inicialização

Um *mutex* é uma variável de tipo `pthread_mutex_t`.

Antes de poder ser usado, um *mutex* tem de ser inicializado.

Há 2 formas alternativas de fazer a inicialização.

Inicialização estática, quando a variável é declarada:

```
pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;
```

Inicialização dinâmica, invocando `pthread_mutex_init()`:

```
int pthread_mutex_init ( pthread_mutex_t *mutex,  
                        const pthread_mutexattr_t *attr);
```

`mutex`

- apontador p/ a variável que representa o *mutex*

`attr`

- permite especificar os atributos do *mutex* a criar; ver `pthread_mutexattr_init()`
- se igual a `NULL` é equivalente a inicialização estática (por defeito)



FEUP

MIEIC  
Faculdade de Engenharia da Universidade do Porto

## Mutexes - Lock e Unlock

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

`pthread_mutex_lock`

- tenta adquirir o *mutex*;  
se ele já estiver *locked*, bloqueia o *thread* que executou a chamada até que o *mutex* esteja *unlocked*

`pthread_mutex_trylock`

- se o *mutex* ainda não estiver *locked*, faz o *lock*
- se o *mutex* estiver *locked*, não bloqueia o *thread* e retorna `EBUSY`

`pthread_mutex_unlock`

- faz o *unlock* do *mutex*
- retorna erro se o *mutex* já estiver *unlocked*  
ou estiver na posse de outro *thread*  
(NOTA: *lock* e *unlock* de um dado *mutex* têm de ser feitos pelo mesmo *thread*)

`pthread_mutex_destroy`

- destrói o *mutex*



FEUP

MIEIC  
Faculdade de Engenharia da Universidade do Porto

M  
u  
t  
e  
x  
e  
s

FEUP

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

#define MAXPOS 10000000 /* nr. max de posições */
#define MAXTHRS 100 /* nr. max de threads */
#define min(a, b) (a)<(b)?(a):(b)
int npos;
pthread_mutex_t mut=PTHREAD_MUTEX_INITIALIZER; /* mutex para a sec.crit. */
int buf[MAXPOS], pos=0, val=0; /* variáveis partilhadas */

void *fill(void *);
void *verify(void *);

int main(int argc, char *argv[]) {
    int k, nthr, count[MAXTHRS]; /* array para contagens */
    pthread_t tidf[MAXTHRS], tidv; /* tid's dos threads */
    if (argc != 3) {
        printf("Usage: fillver <nr_pos> <nr_thrs>\n");
        return 1;
    }
    npos = min(atoi(argv[1]), MAXPOS); /* nr. efectivo de posições */
    nthr = min(atoi(argv[2]), MAXTHRS); /* nr. efectivo de threads */
    for (k=0; k<nthr; k++) {
        count[k] = 0; /* criação dos threads fill() */
        pthread_create(&tidf[k], NULL, fill, &count[k]);
    }
    for (k=0; k<nthr; k++) {
        pthread_join(tidf[k], NULL); /* espera pelos threads fill() */
        printf("count[%d] = %d\n", k, count[k]);
    }
    pthread_create(&tidv, NULL, verify, NULL);
    pthread_join(tidv, NULL); /* thread-verificador */
    return 0;
}
```

continua

```
void *fill(void *nr)
{
    while (1) {
        pthread_mutex_lock(&mut);
        if (pos >= npos) {
            pthread_mutex_unlock(&mut);
            return NULL;
        }
        buf[pos] = val;
        pos++; val++;
        pthread_mutex_unlock(&mut);
        *(int *)nr += 1;
    }
}

void *verify(void *arg)
{
    int k;
    for (k=0; k<npos; k++)
        if (buf[k] != k) /* detecta valores errados */
            printf("buf[%d] = %d\n", k, buf[k]);
    return NULL;
}
```

```
> fillver 10000000 5
count[0] = 2802585
count[1] = 2469019
count[2] = 1361699
count[3] = 1765168
count[4] = 1601529
```



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

## Condition variables

- **Mutexes**
  - permitem a sincronização no acesso aos dados
  - são usados para trancar (*lock*) o acesso
- **Condition variables**
  - permitem a sincronização com base no valor dos dados
  - são usadas para esperar

Sem *condition variables*,  
um programa que quisesse esperar que uma certa condição se verificasse teria de estar continuamente a testar (possivelmente numa secção crítica)  
o valor da condição (*polling*), consumindo, assim, tempo de processador.

As *condition variables*  
permitem fazer este teste sem *busy-waiting*.

Uma *condition variable* é sempre usada conjuntamente com um *mutex*.



FEUP

MIEIC  
Faculdade de Engenharia da Universidade do Porto

Espera pela condição ( $x == y$ ) em *busy-waiting*:

```
...  
while (1) {  
    pthread_mutex_lock(&mut);  
    if (x == y)  
        break;  
    pthread_mutex_unlock(&mut);  
}  
  
/* SECÇÃO CRÍTICA */  
pthread_mutex_unlock(&mut);  
...
```



FEUP

MIEIC  
Faculdade de Engenharia da Universidade do Porto

**Espera pela condição ( $x == y$ ) usando variáveis de condição:***Thread A*

```
...
pthread_mutex_lock(&mut);
while (x != y)
    pthread_cond_wait(&var,&mut);

/* SECÇÃO CRÍTICA */
pthread_mutex_unlock(&mut);
...
```

*Thread B*

```
...
pthread_mutex_lock(&mut);

/* MODIFICA O VALOR DE x E/OU y */
pthread_cond_signal(&var);
pthread_mutex_unlock(&mut);
...
```

Se ( $x != y$ ) `pthread_cond_wait` bloqueia o *thread A* e simultaneamente (de forma indivisível) liberta o *mutex* `mut`.

Quando o *thread B* sinalizar a variável de condição `var`, o *thread A* é desbloqueado; `pthread_cond_wait()` só retorna depois de *A* ter readquirido o *mutex* `mut`, tendo, para isso, de "competir" com outros *threads* que necessitem do *mutex*.

Quando o *thread B* sinalizar a variável de condição isso não significa que a condição ( $x=y$ , neste caso) seja verdadeira. Daí a necessidade do ciclo `while`, no *thread A*.



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

***Condition variables* - Inicialização**

Uma *condition variable* é uma variável de tipo `pthread_cond_t`.

Antes de poder ser usada uma *condition variable* tem de ser inicializada e tem de ser criado um *mutex* associado.

Inicialização estática, quando a variável é declarada:

```
pthread_cond_t mycondvar = PTHREAD_COND_INITIALIZER;
```

Inicialização dinâmica, invocando `pthread_cond_init()`:

```
int pthread_cond_init ( pthread_cond_t *cvar,
                      const pthread_condattr_t *attr);
```

`cvar`

- apontador p/ a *condition variable*

`attr`

- permite especificar os atributos do *condition variable* a criar; ver `pthread_condattr_xxx()`
- se igual a `NULL` é equivalente a inicialização estática (por defeito)



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

## Condition variables - wait e signal

```
int pthread_cond_wait (pthread_cond_t *cvar, pthread_mutex_t *mutex);
int pthread_cond_signal (pthread_cond_t *cvar);
int pthread_cond_broadcast (pthread_cond_t *cvar);
int pthread_cond_destroy (pthread_cond_t *cvar);
```

### pthread\_cond\_wait

- bloqueia o *thread* que fez a chamada até que a condição especificada seja "assinalada"
- deve ser chamada após `pthread_mutex_lock()`
- durante o bloqueio o *mutex* é libertado

### pthread\_cond\_signal

- usada para "assinalar" ou "acordar" outro *thread* (MANUAL: "desbloqueia pelo menos 1 *thread*") que está à espera da condição

### pthread\_cond\_broadcast

- desbloqueia todos os *threads* que nesse momento estiverem bloqueados na variável *cvar* (os *threads* desbloqueados "lutarão" pela aquisição do *mutex* de acordo com a política de escalonamento, como se cada um tivesse executado `pthread_mutex_lock()`; prosseguirá o que obtiver o *mutex*)

### pthread\_cond\_destroy

- destrói a *condition variable*

**NOTA:** `pthread_cond_signal` e `pthread_cond_broadcast` não têm qualquer efeito se não houver processos bloqueados em *cvar*



FEUP

MIEIC  
Faculdade de Engenharia da Universidade do Porto

## Condition variables

```
int x = 0, y = 10;
pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cvar = PTHREAD_COND_INITIALIZER;
```

```
void *incr(void *a)
{
    while (1) {
        pthread_mutex_lock(&mut);
        x = x + 1;
        if (x == y)
            pthread_cond_signal(&cvar);
        pthread_mutex_unlock(&mut);
    }
}
```

```
void *test(void *a)
{
    while (1) {
        pthread_mutex_lock(&mut);
        while (x != y)
            pthread_cond_wait(&cvar, &mut);
        printf("x = y = %d\n", x);
        x = 0;
        y = y + 10;
        pthread_mutex_unlock(&mut);
    }
}
```



FEUP

MIEIC  
Faculdade de Engenharia da Universidade do Porto

## Condition variables

### Problema do PRODUTOR-CONSUMIDOR

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

```
#define BUFSIZE 8
#define SUMSIZE 100
```

```
static int buffer[BUFSIZE];
static int bufin = 0;
static int bufout = 0;
static int items = 0;
static int slots = 0;
```

```
static pthread_mutex_t
buffer_lock = PTHREAD_MUTEX_INITIALIZER;
static int sum = 0;
static pthread_cond_t slots_cond = PTHREAD_COND_INITIALIZER;
static pthread_cond_t items_cond = PTHREAD_COND_INITIALIZER;
static pthread_mutex_t slots_lock = PTHREAD_MUTEX_INITIALIZER;
static pthread_mutex_t items_lock = PTHREAD_MUTEX_INITIALIZER;
```

```
int main(void){
    pthread_t prodtid, constid;
    int i, total;
    slots = BUFSIZE;
    total = 0;
    for (i = 1; i <= SUMSIZE; i++)
        total += i;
    printf("The checksum is %d\n", total);
    if (pthread_create(&constid, NULL, consumer, NULL)){
        perror("Could not create consumer");
        exit(EXIT_FAILURE);
    }
    if (pthread_create(&prodtid, NULL, producer, NULL)){
        perror("Could not create producer");
        exit(EXIT_FAILURE);
    }
    pthread_join(prodtid, NULL);
    pthread_join(constid, NULL);
    printf("The threads produced the sum %d\n", sum);
    exit(EXIT_SUCCESS); //EXIT_SUCCESS e EXIT_FAILURE <- stdlib.h
}
```

continua

FEUP

MIEIC  
Faculdade de Engenharia da Universidade do Porto

```
void put_item(int item){
    pthread_mutex_lock(&buffer_lock);
    buffer[bufin] = item;
    bufin = (bufin + 1) % BUFSIZE;
    pthread_mutex_unlock(&buffer_lock);
    return;
}
```

```
void get_item(int *itemp){
    pthread_mutex_lock(&buffer_lock);
    *itemp = buffer[bufout];
    bufout = (bufout + 1) % BUFSIZE;
    pthread_mutex_unlock(&buffer_lock);
    return;
}
```

```
void *producer(void * arg1){
    int i;
    for (i = 1; i <= SUMSIZE; i++) {
        /* acquire right to a slot */
        pthread_mutex_lock(&slots_lock);
        while (!(slots > 0))
            pthread_cond_wait(&slots_cond, &slots_lock);
        slots--;
        pthread_mutex_unlock(&slots_lock);
        put_item(i);
        /* release right to an item */
        pthread_mutex_lock(&items_lock);
        items++;
        pthread_cond_signal(&items_cond);
        pthread_mutex_unlock(&items_lock);
    }
    pthread_exit(NULL);
}
```

```
void *consumer(void *arg2){
    int myitem;
    int i;
    for (i = 1; i <= SUMSIZE; i++) {
        pthread_mutex_lock(&items_lock);
        while(!(items > 0))
            pthread_cond_wait(&items_cond, &items_lock);
        items--;
        pthread_mutex_unlock(&items_lock);
        get_item(&myitem);
        sum += myitem;
        pthread_mutex_lock(&slots_lock);
        slots++;
        pthread_cond_signal(&slots_cond);
        pthread_mutex_unlock(&slots_lock);
    }
    pthread_exit(NULL);
}
```

The checksum is 5050  
The threads produced the sum 5050

FEUP

Faculdade de Engenharia da Universidade do Porto

## Notas finais

- Os *mutexes* e as *condition variables* poderão ser partilhados entre processos se forem criados em memória partilhada e inicializados com um atributo em que se inclua a propriedade `PTHREAD_PROCESS_SHARED`

( Esta funcionalidade não é suportada pelo Linux )

- Em Linux os *threads* são implementados através da chamada `clone()` a qual cria um processo-filho do processo que a invocou partilhando parte do contexto do processo-pai.
- As funções invocadas num *thread* têm de ser *thread-safe*
  - » as funções *thread-unsafe* são, tipicamente, funções não reentrantes que guardam resultados em variáveis partilhadas
  - » em Unix, algumas chamadas de sistema que são *thread-unsafe* têm versões *thread-safe* (têm o mesmo nome acrescido de `_r`)

