Comunicação entre Processos

IPC- Interprocess communication

- **designação de um conjunto de mecanismos através do qual dois ou mais processos comunicam entre si**

A comunicação entre processos é suportada por todos os sistemas UNIX. Contudo, diferentes sistemas UNIX implementam diferentes métodos de *IPC*.

O UNIX suporta:

- **Para processos correndo na mesma máquina**
  - » ***Pipes* e *FIFOs***
  - » **Mensagens**
  - » **Semáforos**
  - » **Memória partilhada**
- **Para processos correndo em máquinas diferentes**
  - » ***Sockets***
  - » ***TLI - Transport Layer Interface***
  - » ***XTI - X/Open Transport Interface*** **(baseado em *TLI*)**

Os métodos de *IPC* definidos no standard Posix.1b são mensagens, semáforos e memória partilhada, mas a sua sintaxe é diferente da do System V.

Métodos de *IPC* do UNIX System V

Pipes* & *FIFOs

- **Permitem que processos a correr na mesma máquina troquem dados entre si, através de um *pipeline*.**

Mensagens

- **Permitem que processos a correr na mesma máquina troquem dados entre si, através de uma fila de mensagens.**

Memória partilhada

- **Permite que vários processos a correr na mesma máquina partilhem uma região de memória comum.**

Semáforos & *Mutexes*

- **Fornecem mecanismos para que os processos/*threads* sincronizem as suas acções.**

A seguir …

- Filas de mensagens
- Semáforos
- Memória partilhada

} ~~System V - IPC~~ • Posix

Métodos POSIX para *IPC*

Os métodos *IPC* estão definidos na norma POSIX.1003.1b .

Os métodos POSIX para *IPC* são os mesmos do *System V*:
- **filas de mensagens**
- **semáforos**
- **memória partilhada**

A sintaxe das primitivas POSIX que suportam estes métodos é totalmente diferente das do *System V* .

Para compilar programas que usem estas primitivas é necessário *link*ar com a biblioteca *realtime* (`librt.a`)

```
gcc prog1.c -o prog1 -lrt -Wall
```

Filas de mensagens

Primitivas:

```
#include <mqueue.h>

mqd_t mq_open(char* name, int flags, mode_t mode, struct mq_attr* attrp);
int   mq_send(mqd_t mqid, const char* msg, size_t len, unsigned priority);
int   mq_receive(mqd_t mqid, char* buf, size_t len, unsigned* prio);
int   mq_close(mqd_t mqid);
int   mq_notify(mqd_t mqid, const struct mq_sigevent* sigvp);
int   mq_getattr(mqd_t mqid, struct mq_attr* attrp);
int   mq_setattr(mqd_t mqid, struct mq_attr* attrp, struct mq_attr* oattrp);
```

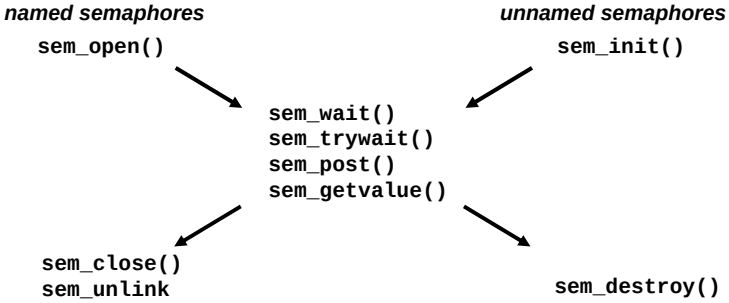
LINUX:

- **Estas primitivas só estão disponíveis a partir da versão 2.6.6-rc1 do *kernel* do Linux, precisando ainda, para funcionamento correcto, de uma *user-space library***
- **O comando `uname -r` permite saber qual a versão do *kernel***

Semáforos

A norma POSIX suporta 2 tipos de semáforos:

- **semáforos com nome (*named semaphores*)**
  - **podem ser partilhados por vários processos**
- **semáforos sem nome (*unnamed semaphores*)**
  - **podem ser partilhados por vários processos que têm acesso a memória comum; para isso, o objecto semáforo tem de ser criado em memória partilhada**



Semáforos

Primitivas:

```
#include <semaphore.h>
sem_t* sem_open(char* name, int flags, mode_t mode, unsigned value);
int    sem_unlink(char* name);
int    sem_init(sem_t* sem, int pshared, unsigned value);
int    sem_close(sem_t* sem);
int    sem_destroy(sem_t* sem);
int    sem_getvalue(sem_t* sem, int* sval);
int    sem_wait(sem_t* sem);
int    sem_trywait(sem_t* sem);
int    sem_post(sem_t* sem);
```

NOTA:
name deve ser da forma /some_name
com "/" no início

Semáforos

PRODUTOR-
-CONSUMIDOR
com *buffer* de tamanho 1

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#define SHARED 0  /* sem. is shared between threads */

void *Producer(void *);  /* the two threads */
void *Consumer(void *);

sem_t empty, full;       /* the global semaphores */
int data;                /* shared buffer         */
int numIters;

int main(int argc, char *argv[]) {
  pthread_t pid, cid;
  numIters = atoi(argv[1]);
  sem_init(&empty, SHARED, 1);  /* sem empty = 1 */
  sem_init(&full, SHARED, 0);   /* sem full = 0  */

  printf("Main started.\n");
  pthread_create(&pid, NULL, producer, NULL);
  pthread_create(&cid, NULL, consumer, NULL);
  pthread_join(pid, NULL);
  pthread_join(cid, NULL);
  printf("Main done.\n");
  return 0;
}
```

```
/* Put items (1, ..., numIters) into the data buffer and sum them */
void *producer(void *arg) {
  int total=0, produced;
  printf("Producer running\n");
  for (produced = 1; produced <= numIters; produced++) {
    sem_wait(&empty);
    data = produced;
    total = total+data;
    sem_post(&full);
  }
 printf("Producer: total produced is %d\n",total);
 return NULL;
}

/* Get values from the data buffer and sum them */
void *consumer(void *arg) {
  int total = 0, consumed;
  printf("Consumer running\n");
  for (consumed = 1; consumed <= numIters; consumed++) {
    sem_wait(&full);
    total = total+data;
    sem_post(&empty);
  }
  printf("Consumer: total consumed is %d\n",total);
  return NULL;
}
```

```
> gcc pc.c -o pc -lpthread -lrt -Wall
> ./pc 20
Main started.
Producer running
Consumer running
Producer: total produced is 210
Consumer: total consumed is 210
Main done.
```

Memória partilhada

Primitivas:

```
#include <sys/types.h>
#include <sys/mman.h>
int shm_open(const char *name, int oflag, mode_t mode);
int shm_unlink(const char *name);
```

NOTA:

- **shm_open() não permite especificar o tamanho da região de memória partilhada**
- **o tamanho tem de ser especificado numa chamada `ftruncate()` subsequente**
- **para juntar e retirar a região de mem. partilhada do espaço de endereçamento de um processo é necessário usar as chamadas `mmap()`e `munmap()`**

Memória partilhada

```
#include <unistd.h>
#include <sys/types.h>
int    ftruncate(int fd, off_t length);


#include <sys/mman.h>
void*  mmap(void  *start,  size_t length, int prot , int flags,
            int fd, off_t offset);
int    munmap(void *start, size_t length);
```

Exemplo

Utilizando mem. partilhada e semáforo(s) para comunicação entre dois processos

ESCRITOR

```
// POSIX shared memory & semaphore - usage example
// writer.c
// Program that writes a digit sequence in shared memory
// and waits for a reader (reader.c) to read it
// The reader must write an '*'
// at the beginning of the shared memory region
// for signaling the writer that the shared memory can be removed
// (another semaphore could have been used instead)
// JAS
// gcc writer.c -lrt -Wall -o writer

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h> // For O_* constants
#include <semaphore.h>
#include <sys/mman.h>
#include <sys/mman.h>
#include <sys/types.h>

#define SHM_SIZE 10

//names should begin with '/'
char SEM_NAME[] = "/sem1";
char SHM_NAME[] = "/shm1";

int main()
{
  int shmfd;
  char *shm, *s;
  sem_t *sem;
  int i, n;
  long int sum = 0;
```

```
 //create the shared memory region
  shmfd = shm_open(SHM_NAME,O_CREAT|O_RDWR,0600);
  if(shmfd<0)
  { ... }
  if (ftruncate(shmfd,SHM_SIZE) < 0)
  { ... }
  //attach this region to virtual memory
  shm = (char *) mmap(0,SHM_SIZE,PROT_READ|PROT_WRITE,MAP_SHARED,shmfd,0);
  if(shm == MAP_FAILED)
  { ... }

  //create & initialize semaphore
  sem = sem_open(SEM_NAME,O_CREAT,0600,0);
  if(sem == SEM_FAILED)
  { ... }
```

CONT →

```
  //write into shared memory region
  s = shm;
  for(i=0; i<SHM_SIZE-1; i++)
  {
    n = i % 10; sum = sum + n;
    *s++ = (char) ('0' + n);
  }
  *s = (char) 0;

  printf("sum = %ld\n", sum);

  sem_post(sem);
```

```
  //this loop could be replaced by semaphore use
  //TO DO by students

  printf("Busy waiting for 'reader' to read shared
memory ...\n");
  while(*shm != '*')
  {
    sleep(1);
  }

  //close and remove shared memory region and semaphore
  sem_close(sem);
  sem_unlink(SEM_NAME);

  if (munmap(shm,SHM_SIZE) < 0)
  { ... }
  if (shm_unlink(SHM_NAME) < 0)
  { ... }

  exit(0);
}
```


Exemplo

```
// POSIX shared memory & semaphore - usage example
// reader.c
// Program that reads a digit sequence
// written in shared memory by a writer (writer.c)
// After reading, this program writes an '*'
// at the beginning of the shared memory region
// signaling to the writer that  the region can be removed
// (another semaphore could have been used instead)
// JAS
// gcc reader.c -lrt -Wall -o reader

#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h> // For O_* constants
#include <semaphore.h>
#include <sys/mman.h>
#include <sys/mman.h>
#include <sys/types.h>

#define SHM_SIZE 10

//names should begin with '/'
char SEM_NAME[] = "/sem1";
char SHM_NAME[] = "/shm1";

int main()
{
  int shmfd;
  char *shm, *s, ch;
  sem_t *sem;
  long int sum = 0;
```

Utilizando mem. partilhada e semáforo(s) para comunicação entre dois processos

LEITOR

```
//open the shared memory region
  shmfd = shm_open(SHM_NAME,O_RDWR,0600);
  if(shmfd<0)
  { ... }

  //attach this region to virtual memory
  shm = (char *) mmap(0,SHM_SIZE,PROT_READ|PROT_WRITE,MAP_SHARED,shmfd,0);
  if(shm == MAP_FAILED)
  { ... }

  //open existing semaphore
  sem = sem_open(SEM_NAME,0,0600,0);
  if(sem == SEM_FAILED)
  { ... }

  //wait for writer to stop writing
  sem_wait(sem);
```

CONT →

```
  //read the message
  s = shm;
  for (s=shm; *s!=0; s++)
  {
    ch = *s;
    putchar(ch);
    sum = sum + (ch - '0');
  }
  printf("sum = %ld\n", sum);

  //once done, signal exiting of reader
  //could be replaced by semaphore use (TO DO by students)

  *shm = '*';

  //close semaphore and unmap shared memory region
  sem_close(sem);

  if (munmap(shm,SHM_SIZE) < 0)
  { ... }

  exit(0);
}
```

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 *mutx,
                         const pthread_mutexattr_t *attr);
```

`mutx`
- 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 omissão)

Mutexes - Lock e *Unlock*

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

`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*

Mutexes

```
#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.crít. */
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

FEUP


```
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

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*.

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);
...
```

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 a *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 outras *threads* que necessitem do *mutex*.

Quando a *thread* A readquirir o *mutex* isso não significa que a condição (x==y, neste caso) ainda seja verdadeira. Daí a necessidade do ciclo `while`, na *thread* A.

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)

Condition variables - wait e signal

```
int pthread_cond_wait (pthread_cond_t *cvar, pthread_mutex_t *mutx);
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`

Condition variables

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

```
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);
   }
}
```

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

Condition variables

Problema do PRODUTOR--CONSUMIDOR

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

#define BUFSIZE 8
#define NUMITEMS 100

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

pthread_mutex_t
buffer_lock = PTHREAD_MUTEX_INITIALIZER;
int sum = 0;
pthread_cond_t  slots_cond = PTHREAD_COND_INITIALIZER;
pthread_cond_t  items_cond = PTHREAD_COND_INITIALIZER;
pthread_mutex_t slots_lock = PTHREAD_MUTEX_INITIALIZER;
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 <=NUMITEMS; 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


```
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 <= NUMITEMS; 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 <= NUMITEMS; 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
```

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

- 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`)