Programação em Threads

Linux Threads – PThreads

Compilação

$ gcc –lpthread –D_REENTRANT –Wall program.c –o program

-lpthread: inclui biblioteca de threads

-D_REENTRANT: torna as bibliotecas *thread safe*

Operações básicas: Gestão de Threads

```
#include <pthread.h>
pthread_t id;

int pthread_create(pthread_t* id, pthread_attr_t *attr,
                   void* (*start_routine)(void* ), void* args);

void pthread_exit(void* return_value);

int pthread_join(pthread_t id, void** returned_value);

pthread_t pthread_self(void);
```

Gestão de threads

- Criação de threads (ex01.c)
  - Não deixar a thread principal morrer
  - A rotina retorna void* e leva como parâmetro void*
- Passagem de parâmetros (ex02.c)
  - Passar um ponteiro para o valor a enviar à thread
  - Utilizar sempre pthread_exit() ou equivalente
  - pthread_join() leva como parâmetro o id da thread, não um ponteiro
- Identificadores de threads (ex03.c)
  - Utilizar sempre ponteiros para valores diferentes

Sincronização simples
Exclusão Mútua

```
#include <semaphore.h>

pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

pthread_mutex_lock(&lock);
// …
// zona crítica
// …
pthread_mutex_unlock(&lock);

pthread_mutex_destroy(&lock);
```

Sincronização simples
Semáforos

```
sem_t empty;

int sem_init(sem_t* id, int pshared, unsigned int value);
                                  (pshared always 0 in Linux)

int sem_wait(sem_t* id);

int sem_post(sem_t* id);

int sem_destroy(sem_t* id);
```

Produtor/Consumidor

```
#define BUF_SIZE     10

typedef struct
{
  int slots[BUF_SIZE];
  int read_pos;                    // 0
  int write_pos;                   // 0

  pthread_mutex_t lock;            // unlocked
  sem_t empty;                     // BUF_SIZE
  sem_t full;                      //  0
} synch_buffer;
```

put_value()

```
void put_value(synch_buffer* buf, int value)
{
  sem_wait(&buf->empty);

  pthread_mutex_lock(&buf->lock);

  buf->slots[buf->write_pos] = value;
  buf->write_pos = (buf->write_pos + 1)%BUF_SIZE;
  pthread_mutex_unlock(&buf->lock);

  sem_post(&buf->full);
}
```

get_value()

```
int get_value(synch_buffer* buf)
{
  int to_return;

  sem_wait(&buf->full);

  pthread_mutex_lock(&buf->lock);
  to_return = buf->slots[buf->read_pos];
  buf->read_pos = (buf->read_pos + 1)%BUF_SIZE;
  pthread_mutex_unlock(&buf->lock);

  sem_post(&buf->empty);

  return to_return;
}
```

init_buffer()

```
void init_buffer(synch_buffer* buf)
{
  pthread_mutex_init(&buf->lock, 0);
  sem_init(&buf->empty, 0, BUF_SIZE);
  sem_init(&buf->full, 0, 0);

  buf->read_pos  = 0;
  buf->write_pos = 0;
}
```

Let’s see it work

Sincronização "avançada" Variáveis de condição

- Generalização do conceito de semáforo

  (a condição associada a um semáforo é um teste >=0)

- Necessitam sempre de um `mutex` e de uma variável do tipo `pthread_cond_t`

Sincronização "avançada" Variáveis de Condição

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

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

int pthread_cond_signal(pthread_cond_t* cond);
int pthread_cond_broadcast(pthread_cond_t* cond);

int pthread_cond_wait(pthread_cond_t* cond,
                      pthread_mutex_t* lock);

int pthread_cond_destroy(pthread_cond_t* cond);
```

Thread que espera uma condição

```
pthread_cond_t buf_alterado = PTHREAD_COND_INITIALIZER;
pthread_mutex_t buf_mutex  = PTHREAD_MUTEX_INITIALIZER;
int total_valores = 0;

(...)
// Espera que o buffer encha
pthread_mutex_lock(&buf_mutex);
while (total_valores < MAX)
  pthread_cond_wait(&buf_alterado, &buf_mutex);

// Zona crítica: BUFFER CHEIO!

pthread_mutex_unlock(&buf_mutex)
(...)
```

Thread que altera uma condição

```
(...)

pthread_mutex_lock(&buf_mutex);

// Zona crítica: A alterar o buffer
(...)
++total_valores;
pthread_cond_signal(&buf_alterado);

pthread_mutex_unlock(&buf_mutex)
(...)
```

Regras importantes

- Colocar sempre `while()` em torno da condição a testar
- Fazer sempre a notificação da variável e então libertar o `mutex`. Nunca ao contrário!

Exemplo: Barreira

Objectivo: escrever uma função que:

a) Quando chamada bloqueia até que todas as threads a chamem.
b) Quando a última thread a chama deixa todas as threads prosseguirem com a sua execução

Simplificação:

Não necessita de funcionar correctamente quando chamada múltiplas vezes

```
#define N 100
pthread_cond_t   GO = PTHREAD_COND_INITIALIZER;
pthread_mutex_t LOCK  = PTHREAD_MUTEX_INITIALIZER;
int global_data=0; // accessed by the threads before calling barrier()
pthread_mutex_lock(&LOCK);
...
pthread_mutex_unlock(&LOCK);
...
pthread_cond_wait(&GO,&LOCK);
...
pthread_cond_signal(&GO);
...
pthread_cond_broadcast(&GO);
...  // CREATE N THREADS
int thr[N];
int id[N];
for(i=0;i<N;i++){
    id[i] = i;
    pthread_create(&thr[i], NULL, my_thread, &id[i]);
    }
```

```
void *my_thread(void *id_ptr){
int id= *((int *)id_ptr);
for(i=0; i<10; i++){
            pthread_mutex_lock(&LOCK);
            global_data ++;
            pthread_mutex_unlock(&LOCK);
            barrier();
            }
printf("done !");
}
```

Barrier

```
pthread_cond_t can_go = PTHREAD_COND_INITIALIZER;
pthread_mutex_t lock  = PTHREAD_MUTEX_INITIALIZER;

int entered = 0;

void barrier()
{
  pthread_mutex_lock(&lock);

  ++entered;
  if (entered != N){
    pthread_cond_wait(&can_go, &lock);
  }else{
    pthread_cond_broadcast(&can_go);
     entered=0;
     }
  pthread_mutex_unlock(&lock);
}
```

Para casa...

- Implementar um produtor/consumidor utilizando variáveis de condição.