LAB 4: Threads

OBJETIVO:COMO CRIAR E CONTROLAR THREADS?

Que interfaces devem estar presentes no SO para a criação e controlo de **threads**? Como devem essas interfaces ser desenhadas para permitir o seu uso de forma fácil e se tornarem úteis?

CRUX: HOW TO CREATE AND CONTROL THREADS

What interfaces should the OS present for thread creation and control? How should these interfaces be designed to enable ease of use as well as utility?

4.1. Criação de threads

Para criar um thread utiliza-se a biblioteca POSIX. Esta tem a assinatura da chamada pthread_create com os parâmetros indicados na figura 4.1

```
#include <pthread.h>
int
pthread_create(      pthread_t *       thread,
               const pthread_attr_t *  attr,
                     void *            (*start_routine)(void*),
                     void *            arg);
```

Fig 4.1: *pthread_create()*

A declaração da função utiliza o que se designa em C por apontadores para funções - function pointers**.**

O segundo argumento é usado para especificar atributos, que o thread tenha. Algum exemplos incluem a dimensão da pilha, ou informação sobre as prioridades de escalonamento de um thread. Um atributo é inicializado através uma chamada ao `pthread_attr_init()`. Na maior parte dos casos usam-se os atributos por defeito e, nesse caso, o valor de `attr` é `NULL`.

O terceiro argumento é o mais complexo. Ele pede a função que é, inicialmente, executada pelo thread. Ela declara uma função com o nome start_routine com argumento apontador para void. Se quisessemos executar um thread com um argumento inteiro, então utilizaríamos a declaração na figura 4.2.

```
int pthread_create(..., // first two args are the same
                   void *    (*start_routine)(int),
                   int       arg);
```

Fig. 4.2*: Criação de pthread com um argumento inteiro.*

Se a rotina retornar um inteiro, mantendo um apontador como argumento obtemos a declarção na figura 4.3.

```
int pthread_create(..., // first two args are the same
                   int       (*start_routine)(void *),
                   void *     arg);
```

Fig. 4.3*: Criação de pthread retornando um inteiro.*

O quarto argumento, `arg`, é exatamenteo o argumento que é passado na função.

No exemplo da figura 4.4 cira-se um thread com dois argumentos definidos num único tipo definido no programa (myarg_t). O thread, assim que criado, pode fazer o cast do argumento para o tipo que deseja e depois "desempacotar" os argumentos como deseja.

```
1   #include <pthread.h>
2
3   typedef struct __myarg_t {
4       int a;
5       int b;
6   } myarg_t;
7
8   void *mythread(void *arg) {
9       myarg_t *m = (myarg_t *) arg;
10      printf("%d %d\n", m->a, m->b);
11      return NULL;
12  }
13
14  int
15  main(int argc, char *argv[]) {
16      pthread_t p;
17      int rc;
18
19      myarg_t args;
20      args.a = 10;
21      args.b = 20;
22      rc = pthread_create(&p, NULL, mythread, &args);
23      ...
24  }
```

Fig. 4.4*: Criar un thread utilizando uma estrutura myarg_t que mantém dois inteiros.*

4.2 Terminar um thread

Sempre que precisamos de esperar que um thread esteja completo para continuar a execução, precisamos de um comando específico para esperar por o final dessa execução. Esse comando é `pthread_join()`:

```
int pthread_join(pthread_t thread, void **value_ptr);
```

Esta função tem dois argumentos. O primeiro é do tipo pthread_t e é utilizado para especificar que thread se fica à espera. A variável é inicializada com a função de criação do thread. O segundo é um apontador para o valor de retorno. Um vez que a função pode não retornar nada, é usado o void.
No exemplo seguinte (figura 4.5), um thread é criado com dois argumentos do tipo myret_t. Quando o thread termina a seua execução, o thread principal que esperava a terminação dos outros threads usando a função pthread_join(), retorna e é possível aceder aos valores retornados do thread, neste caso, os valores guardados em myret_t.

Algumas notas adicionais:
- Não é necessário criar um thread com argumentos. O valor dos argumentos pode ser NULL;
- Não é necessário aceder a um valor de retorno da execução do thread. Podemos simplesmente associar NULL a esse parâmetro.
- Podemos passar um tipo simples como argumento, não sendo necessário passar um tipo composto.

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

typedef struct __myarg_t {
    int a;
    int b;
} myarg_t;

typedef struct __myret_t {
    int x;
    int y;
} myret_t;

void *mythread(void *arg) {
    myarg_t *m = (myarg_t *) arg;
    printf("%d %d\n", m->a, m->b);
    myret_t *r = Malloc(sizeof(myret_t));
    r->x = 1;
    r->y = 2;
    return (void *) r;
}

int
main(int argc, char *argv[]) {
    pthread_t p;
    myret_t *m;

    myarg_t args = {10, 20};
    Pthread_create(&p, NULL, mythread, &args);
    Pthread_join(p, (void **) &m);
    printf("returned %d %d\n", m->x, m->y);
    free(m);
    return 0;
}
```

***Fig. 4.5**: Utilizar o pthread_join() esperando pela terminação do thread mythread. Repare que o main() corresponde ao pthread principal (main thread).*

4.3 Locks

Para além da criação de threads, os **locks** permitem a criação de secções críticas através da exclusão mútua. O par de funções que pode ser utilizado para é o indicado na figura 4.6.

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

***Fig. 4.6:** Assinaturas de ferrolho mutex utilizando as rotinas pthread_mutex_lock e pthread_mutex_unlock.*

***Fig. 4.7:** Código para gerir uma secção crítica*

O objetivo do código é o seguinte: Se nenhum outro **thread** tem o ferrolho (*holds the lock*), quando o comando `pthread_mutex_lock()` é executado o **thread** adquire o **lock** e entra na secção crítica. Se outro **thread** pretende adquirir o lock, fica à espera.

No entanto este código não funciona bem:

1- Falta a inicialização Todos os ferrolhos precisam de uma inicialização de forma a que se garanta que os valores corretos são inseridos.

2- Falta verificar que uma determinada chamada falha. Para isso utiliza-se o “assert” para garantir que não é atribuída a secção critica se houver uma falha na operação de fecho do ferrolho.

4.3.1 Inicialização

No POSIX a inicialização é efetuada de duas formas. A primeira:

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```

A outra alternativa é inicializar o thread de forma dinâmica, isto é, quando o programa está a ser executado, chamando o comando `pthread_mutex_init`:

```
int rc = pthread_mutex_init(&lock,NULL);
assert(rc == 0);
```

O primeiro argumento é o endereço do ferrolho enquanto que o segundo corresponde a um conjunto opcional de atributos.

Quando termina a execução, deve-se executar o comando que elimina o ferrolho:

```
pthread_mutex_destroy();
```

4.3.2 *Wrapper*

Uma forma de garantir que existe alguma proteção ao erro associado à operação de *lock*, como é apresentado na figura 4.7.

```
// Use this to keep your code clean but check for failures
// Only use if exiting program is OK upon failure
void Pthread_mutex_lock(pthread_mutex_t *mutex) {
  int rc = pthread_mutex_lock(mutex);
  assert(rc == 0);
}
```

Fig. 4.7: Exemplo de um wrapper

4.4 Variáveis de condição

A outra componente útil na biblioteca dos **threads** são as **variáveis de condição**. Elas são úteis quando é necessário estabelecer uma comunicação/sinalização entre **thread**, garantindo, por exemplo, que um **thread** executa determinado comando antes de continuar a sua execução.

Os dois comandos que se utilizam na biblioteca POSIX são os da figura 4.8.

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
```

***Fig. 4.8**: Comandos que comandam a variável condição cond*

Para usar a variável condição, é necessário adicionar um ferrolho que está associado à condição. Este ferrolho garante que a sinalização é feito dentro de uma secção crítica.

A rotina `pthread_cond_wait()` coloca o **thread** a dormir e, assim, espera que algum outro **thread** o sinalize. Por exemplo, quando alguma coisa no programa mudou que interessa ao **thread** a dormir.

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

Pthread_mutex_lock(&lock);
while (ready == 0)
    Pthread_cond_wait(&cond, &lock);
Pthread_mutex_unlock(&lock);
```

***Fig. 4.9a**: Utilização da variável condição*

A figura **4.9a.** mostra um exemplo da sua utilização. No código, depois da inicialização do ferrolho *lock* e da variável condição *cond*, o **thread** testa se a variável *ready* tem um valor diferente de 0. Se não, o **thread** chama a rotina que o vai permitir dormir até que algum outro **thread** o acorde.
O código para acordar o **thread** encontra-se na figura 4.9b.

```
Pthread_mutex_lock(&lock);
ready = 1;
Pthread_cond_signal(&cond);
Pthread_mutex_unlock(&lock);
```

***Fig. 4.9b**: Acordar um thread que espera pelo valor ready diferente de 0.*

Algumas coisas podem acontecer com o código apresentado. Primeiro, quando se executa a sinalização (e também quando se modifica a variável global ready), garantimos que o ferrolho está fechado, garantindo que não introduzimos uma condição de corrida (race condition) no código. Em segundo lugar, pode observar-se que o comando *wait* usa o lock como segundo parâmetro enquanto que o comando *signal* apenas usar a condição. A razão para esta diferença é a de que na função *wait* é libertado o ferrolho antes de colocar o thread a dormir. Se esta operação não fosse feita, nenhum thread poderia acordar o thread a dormir.
Depois, quando é acordado, antes de passar à execução, ou seja ainda no contexto de pthread_cond_wait(), re-adquire o ferrolho assegurando que durante a execução do código dentro da secção crítica é apenas feito por um thread.

4.5 Compilar e correr

Para correr os exemplos apresentados nesta laboratório, é necessário incluir o ficheiro header pthread.h no código. Também é necessário adicionar a flag -pthread.
Por exemplo, para compilar o programa com mais do que um thread, devemos executar o comando seguinte:

```
> gcc –o main main.c –Wall –pthread
```

4.6 Questões

4.6.1 Faça a compilação do programa *main-race.c*. Examine o código e procura perceber as indicações que o código traz, por exemplo, “unprotected access”.
a) Execute o comando: `valgrind --tool=helgrind ./main-race`. Que informação o programa fornece?
b) Explique o que acontece quando comenta apenas uma das linhas indicadas no código como “unprotected access”?
c) Adicione um ferrolho (**lock**) em redor de uma das atualizações da variável partilhada. Corra o *helgrind*. Verifique se continua a dar erro.
d) Adicione o ferrolho aos dois momentos de atualização dessa variável. O que informa o *helgrind* neste caso?

4.6.2 Considere o programa *main-deadlock.c*. Examine o código.
a) Este código tem um problema conhecido como o **deadlock**. Consegues perceber que tipo de problema se trata?
b) Corra o programa. Deteta alguma falha? Altere agora o programa introduzindo a seguinte instrução entre as linhas indicadas:

```
Pthread_mutex_lock(&m1);
 for (int i=0;i<10000;i++);
Pthread_mutex_lock(&m2);
```

Faça a compilação e execute o programa várias vezes. Consegue detetar alguma falha? Explique o que poderá estar a acontecer.
c) Corra o helgrind sobre este código. O que informação apresenta? O que significa "lock order … violated"?
d) Corra agora o programa em background. Certifique-se que ele entra em deadlock.

(i) Utilize o ps -T para localizar os threads em execução. Vai verificar que existem 2 thread filho e 1 thread pai (main).

(ii) Execute o comando kill -9 PID em que PID é o identificador do thread, para matar os processos pendurados.

4.6.5. Corra agora o helgrind no programa *main-deadlock-global.c*. Examine o código.
a) Será que ele tem o mesmo problema registado no programa anterior? Será que o helgrind deveria reportar o mesmo erro?
b) O que é que este resultado te diz acerca de ferramentas como o helgrind?

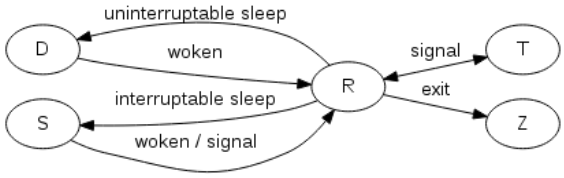
4.6.6 Corra agora o main-signal.c. Este código utiliza a variável *done* para sinalizar que um filho terminou e que o **thread** pai pode continuar.
a) Por que razão este código é ineficiente? O que faz o pai gastar tanto tempo à espera que o filho termine uma tarefa, que pode ser longa?
b) Corra agora o helgrind. Que reporta o programa? Será que o código está correto?
c) Introduza o código entre as linhas indicadas (altere o número de zeros para conseguir ver a execução dos threads utilizando o comando ps apresentado em seguida):

```
void* worker(void* arg) {
    printf("this should print first\n");
    for (int i=0;i<1000000;i++);
    done = 1;
    return NULL;
}
```

Enquanto o programa corre, execute o comando:

```
ps -T -o pid,spid,tty,state,command,time
```

Identifique o estado dos thread utilizando a lista de códigos



Códigos de estado do processo: R running or runnable (on run queue); D uninterruptible sleep (usually IO); S interruptible sleep (waiting for an event to complete); Z defunct/zombie, terminated but not reaped by its parent; T stopped, either by a job control signal or because it is being traced

4.6.7. Considere agora um programa *main-signal-cv.c* ligeiramente modificado. Nesta versão utiliza-se uma variável de condição para realizar a sinalização, associando-lhe um ferrolho.

a) Por que razão este código é preferível ao código anterior? Será que é pela correção ou pelo desempenho? Ou por ambos?

b) Corra o *helgrind*. Será que este reporta algum erro?

c) Coloque o código seguinte no programa (exatamente como fez para 4.6.6):

```
void* worker(void* arg) {
    printf("this should print first\n");
    for (int i=0;i<1000000000;i++);
    signal_done(&s);
    return NULL;
}
```

Enquanto o programa corre, execute o comando:

```
ps -T -o pid,spid,tty,state,command,time
```

Identifique o estado dos **thread** utilizando a lista de códigos.

Que diferença essencial distingue entre os dois casos (4.6.6 e 4.6.7)?

4.6.10.Exemplo de utilização dos mutex e variável condição

```
#include "mythreads.h"
#include <unistd.h>
#include <stdio.h>
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int  counter = 0;
void* contador()
{
   Pthread_mutex_lock( &mutex );
   counter++;
   sleep(1);
   printf("Valor do contador: %d\n",counter);
   Pthread_mutex_unlock( &mutex );

   return NULL;
}

void main()
{
   int rc1, rc2;
   pthread_t thread1, thread2;
   /* Cria threads independentes, cada um executando a função contador */
   Pthread_create( &thread1, NULL, contador, NULL);
   Pthread_create( &thread2, NULL, contador, NULL);
   Pthread_join( thread1, NULL);
   Pthread_join( thread2, NULL);
   exit(0);
}
```

(Exemplo retirado de http://www.cs.cmu.edu/afs/cs/academic/class/15492f07/www/pthreads.html)

a) Faça a compilação e linkagem deste programa.

b) Comente os comandos **pthread_mutex_lock** e **pthread_mutext_unlock**. Execute o programa novamente. Explique as diferenças que observa.

c) Pretende agora criar um pequeno programa que gere uma conta bancária. São criadas **thread** que são os clientes. Há quem deposite nessa conta e quem retire dinheiro nessa conta. Mostre que o mutex garante que o resultado de operações é consistente com o valor que aparece no depósito. Há ainda a ação de ver o valor que está depositado. Essa operação deve estar dentro do mutex?

d) Nessa mesma conta bancária, pretendemos criar uma **variável condição** que ativa um thread sempre que a conta chega a zero, dando essa informação no ecrã.