9

Programação com Posix Threads

Revisão: Data: 21-04-2018

Programação multi-threaded com Pthreads

9.1. O que é uma *thread*?

Um processo representa a execução pelo sistema operativo dum programa. Uma *thread* representa uma linha de execução das instruções deste programa. Um processo poderá conter várias threads, pelo menos uma. O comando Unix `ps` permite ver os processos que estão a correr e dando a opção correcta as threads que estão a executar. O comando **top** também é útil para obter uma lista ordenado dos processos. Pode também utilizar as ferramentas gráficas (process explorer, system monitor etc.) e ligar a opção para ver as threads.

9.2. O que são Pthreads?

Historicamente cada sistema operativo/hardware implementava a sua própria versão de threads. Claro que cada implementação varia entre si portanto era difícil para programadores de desenvolver aplicações portáveis que utilizassem threads. Para superar estas dificuldades a padrão POSIX1003.1-2001 foi desenvolvida. Esta define um *application programming interface* (API) para a escrita de aplicações *multithreaded.* Implementações de threads que aderem a este padrão são conhecidas como POSIX threads ou simplesmente *pthreads*.

- Pthreads são definidas como um conjunto de tipos de dados em C e um conjunto de rotinas.
- O "API" Pthreads contêm mais de 60 sub-rotinas.
- Todos os identificadores na biblioteca começam com **pthread_**
- O ficheiro pthread.h tem que ser incluído em cada ficheiro de código fonte.
- A linkagem com qualquer biblioteca dinamica/estatica necessária é feita conforme o sistema.

9.3. Pthread Criação e Terminação

Quando um programa começa a executar terá um processo com uma thread a executar. Mais threads são depois criadas com a função pthread_create() e destruídas com a função pthread_exit().

Exemplo 9.1 Pthread Criação e Terminação

```
#include <stdio.h>       /* standard I/O routines                 */
#include <pthread.h>     /* pthread functions and data structures */

/* function to be executed by the new thread */
void *OLA(void *argumentos)    {
   printf("\nOla\n");
   pthread_exit(NULL);
}

int main ( )
{
   pthread_t thread;
   int flag, i;

   printf("A criar uma nova thread\n");
   flag = pthread_create(&thread, NULL, OLA, NULL);
   if (flag!=0) printf("Erro na criação duma nova thread\n");

   OLA(NULL);
   pthread_exit(NULL);
   return 0;   /* O programa não vai chegar aqui.         */
}
```

Notas Explicativas:
A chamada à função pthread_create() tem quatro argumentos. O primeiro é usado para guardar informação sobre a thread criada. O segundo especifica algumas propriedades da thread a criar, utilizaremos o valor NULL para significar os valores por defeito. O Terceiro é a função que a nova thread vai executar e o ultimo é usado para representar argumentos a esta função.

A chamada à função pthread_exit() provoca a terminação da thread e a libertação dos recursos que esta está a consumir. Aqui não há realmente necessidade para usar esta função porque quando a função da thread termine a thread seria destruída. A função é apenas útil se for necessário destruir a thread no meio da sua execução.

A função a ser executada por uma nova thread tem sempre o seguinte protótipo:

```
void * funcao ( void * argumentos ) ;
```

Exercício 9.1

Escreve o programa do exemplo 9.1. Compile o programa (cc –Wall -o criar criar.c -lpthread) e depois executá-lo. Quantos threads são criados? Quantas mensagens aparecem no ecrã ?

9.4. Esperando pela Terminação duma Thread

As threads podem executar duma forma desunidas (*detached*) da thread que as criou ou unidas. Desta maneira usando a rotina pthread_join() uma thread pode esperar pela terminação duma thread específica.

Exemplo 9.2 Esperando uma thread

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

#define NUM_THREADS     5

void *funcao(void *argumentos)
{
   printf("\nOLA\n");
   return (NULL);
}

int main ()
{
  pthread_t threads[NUM_THREADS];
  int i;

  for(i=0;i<NUM_THREADS;i++)
       pthread_create(&threads[i], NULL, funcao, NULL);

  printf("Thread principal a esperar a terminação das threads criadas \n");

  for(i=0;i<NUM_THREADS;i++)
      pthread_join(threads[i],NULL);  /* Esperara a junção das threads */

return 0;   /* O programa chegará aqui. */
}
```

Exercício 9.2

Escreve, Compile e Execute o programa do exemplo 9.2.

Exercício 9.3

- Neste exercício vai escrever um programa que possa ser parado para depois inspecionar as threads criadas usando as ferramentas do sistemas (ps, activity monitor etc.).
- Usando como base o exemplo 9.2 escreve uma programa multithreaded onde as threads criadas pela thread principal entrarão num ciclo enquanto o valor duma variável global, **x**, seja igual ao valor 1.
- O programa deverá declarar e inicializar a variável global **x** com valor 1.
- Na thread principal depois de criar as threads pede o novo valor de **x** usando por exemplo scanf.
  Ver o código em baixo!

```
void * funcao(void *args)
{
  while (1==x)
     ;/*fazer nada(spin)*/
  printf("\nOla\n");
  return (NULL);
}
```

```
int x=1;
main() {
  for...pthread_create( funcao );

  printf("Introduza novo Valor de x.");
  scanf("%d",&x);

  for...pthread_join();
```

(i) Compile e Execute o programa.
(ii) Execute o programa mas na altura de efetuar a introdução dum novo valor de **x** parar o programa com *ctrl-z* e depois inspecionar o processo e as suas threads usando o comando ***ps*** e/ou o comando ***top*** e as opções apropriadas.
(iii) Por o processo novamente a correr (usando os comandos de job control : jobs, fg, o comando kill etc.). Numa outra janela de terminal inspecione as threads do processo e os seus estados usando os comandos **ps/top/activity monitor etc**. p.ex "ps -alT | grep myprogram"

Nota: Podem ser úteis as opções A,l,m,M,T do commando ps

9.5. Passagem de Argumentos para threads

- A rotina *pthread_create()* permite ao programador passar **apenas um** argumento a rotina da thread nova.
- Um argumento é passado por referencia (apontador) e é feito um cast para (void *).
- Para 'passar' vários argumentos temos que empacotar os valores numa estrutura (um bloco do memoria contigua) e depois passar o endereço desta estrutura.

Exemplos 9.3 –(I) Passagem dum Inteiro (II) uma String (III) duma estrutura

I. Passagem dum inteiro

int x=5;

```
pthread_create(&threads[i], NULL, funcao, (void*)&x);

void * funcao ( void * argumentos ){
    int valor =  * (int *) argumentos;
    printf("recebi um inteiro: %d \n", valor );
}
```

II. Passagem duma String

char mesg[ ]="Ola";

```
pthread_create(&threads[i], NULL, funcao, (void*)mesg);

void * funcao ( void * argumentos ){
    char *message = (char *) argumentos ;
    printf(" %s ", message );
}
```

III. Passagem de múltiplos parâmetros usando uma estrutura

tyypedef struct { int a; float b; } ST;
ST v;

v.a = 5; v.b = 2.5; //atribuir valores

```
pthread_create(&threads[i], NULL, funcao, (void*)&v);  //passer para a thread

void * funcao ( void * argumentos ){
    ST * in =  (ST *) argumentos ;
    printf("recebi dois valores: %d %f ", in->a, in->b );
}
```

9.6. Condições de Corrida na Passagem de Argumentos para threads

Existe uma dificuldade adicional em relação a passagem de dados para as novas threads. Dado a sua inicialização não-determinística e dificuldade de controlar o escalonamento e despacho das threads será assim difícil assegurar a passagem de dados com segurança, sem condições de corrida.
Muitas vezes precisamos de identificar as threads para as depois controlar, e portanto será passado uma variável inteiro para as identificar. O exemplo seguinte ilustre a dificuldade e perigo deste processo.

Exemplo 9.4 O ficheiro test1.c

```
#define NUM_THREADS 5

void *funcao(void *args)
{
   //sleep( rand()%3);
   int id = *(int *)args;
   printf("Thread %d\n",id);
   return (NULL);
}

int main ()
{
  pthread_t threads[NUM_THREADS];
  int i;
  for(i=0;i< NUM_THREADS;i++)
       pthread_create(&threads[i], NULL, funcao, &i);

  for(i=0;i< NUM_THREADS;i++)
      pthread_join(threads[i],NULL);
  return 0;
}
```

Gostávamos de obter um "id" diferente pra cada thread, quer dizer os valores "Thread 0,Thread 1, Thread 2, Thread 3, Thread 4" impressos no ecrã num ordem qualquer. No entanto observe agora em baixo alguns outputs típicos de execução do programa:

alunos: $ cc -o test1 test1.c -lpthread

alunos: $ ./test1	alunos: $ ./test1	alunos: $ ./test1
Thread 2	Thread 0	Thread 4
Thread 2	Thread 1	Thread 4
Thread 2	Thread 2	Thread 3
Thread 5	Thread 4	Thread 3
Thread 3	Thread 4	Thread 3

Exercício 9.4 (i) Escreve, Compile e Execute os programas "test1" com mais threads e com e sem o sleep !

9.7. A Passagem Correta dos argumentos para identificar uma thread.

A maneira correta é passar um endereço para uma variável que identifique unicamente apenas uma thread e é usada exclusivamente por apenas uma thread .. vejam e estudam o próximo exemplo.

Exemplo 9.5 O ficheiro test2.c

```
void *funcao(void *args)
{
   int id = *(int *)args;
   printf("Thread %d\n", id);
   return (NULL);
}
int main ()
{
  pthread_t threads[NUM_THREADS];
  int i, ids[NUM_THREADS];

  for (i = 0; i < NUM_THREADS; i++) ids[i]=i;

  for(i=0;i < NUM_THREADS;i++)
       pthread_create(&threads[i], NULL, funcao, &ids[i]);

   for(i=0;i < NUM_THREADS;i++)
       pthread_join(threads[i],NULL);
   return 0 ;
 }
```

Uma alternativa é usar **malloc** (cuidado aqui com a memória alocada e "memory leeks").

```
int *id;
for(i=0;i < NUM_THREADS;i++)  {
    id = (int *)malloc(sizeof(int));
    *id = i;
    pthread_create(&threads[i], NULL, funcao, id);
}
//free(id) ??  -- O free deverá ser feito dentro da nova thread depois de
//               ter usado o valor i.e dentro da funcao
```

Ouputs Corretos de execução do programa são agora mostrados

alunos: $ ./test2	alunos: $ ./test2
Thread 1	Thread 0
Thread 0	Thread 1
Thread 2	Thread 2
Thread 3	Thread 4
Thread 4	Thread 3

Desta maneira podemos controlar o fluxo de cada thread dentro da função da thread usando instruções de condição (if,switch etc).

Exercício 9.4 (ii) :Escreve, Compile e Execute os programas "test2." Do exemplo 9.5 em cima.

Exercício 9.5

Escreve um Programa multithreaded para calcular o valo da funcao $y=\sin^3(x)+\sqrt{\cos(x)}$.
Deverá criar duas threads novas, uma para calcular $f_1=\sin^3(x)$ e uma segunda para calcular $f_2=\sqrt{\cos(x)}$.
A thread principal deverá depois calcular o valor final (f_1+f_2) depois de terminação e junção das duas threads.

Exercício 9.6

Escreve, Compile e Execute um Programa multithreaded para calcular o produto de duas matrizes quadrados de dimensão dois. Deverá criar quatro threads novas, cada uma usada para calcular um dos quarto elementos de matriz resultante. Poderá usar as estruturas de dados em baixo !

int m1[2][2]={1,2,3,4}, m2[2][2]={3,4,7,8}, mproduto[2][2].