

Sistemas Operativos	Threads
<h1>Threads</h1>	
Jorge Silva	MIEIC / FEUP

1

Sistemas Operativos	Threads
<h2>Objectivos</h2>	
<p>No final desta aula, os estudantes deverão ser capazes de:</p> <ul style="list-style-type: none">• escrever programas que usem um ou múltiplos <i>threads</i> (<i>multithreaded</i>)• transferir dados (entrada/saída) entre <i>threads</i>• identificar os problemas que se colocam na manipulação de dados comuns e na passagem de dados entre <i>threads</i> e resolver alguns desses problemas• usar um mecanismo básico de sincronização entre <i>threads</i> (<code>pthread_join()</code>)	
Jorge Silva	MIEIC / FEUP

2

Threads

- A **Pthreads API** está definida na norma ANSI / IEEE POSIX 1003.1c
 - A Native POSIX Thread Library (NPTL), introduzida com a versão 2.6 do *kernel* do Linux kernel, é totalmente compatível com este standard POSIX.
- As funções desta **API** (cerca de 100) podem ser divididas em 3 grupos:
- **GESTÃO DE THREADS**
 - Permitem **criar** e **terminar threads**, **esperar** pela sua terminação, etc. Incluem funções para ler / alterar os **atributos** dos **threads** (de escalonamento e outros, ex: se é possível esperar que um **thread** termine).
- **MUTEXES** (ver cap. sobre sincronização)
 - Permitem proteger uma **secção crítica**. Incluem funções para criar, destruir, trancar (**lock**) e destrancar (**unlock**) **mutexes** e alterar os seus atributos.
- **CONDITION VARIABLES (VARIÁVEIS DE CONDIÇÃO)** (ver cap. sobre sincronização)
 - Permitem bloquear um **thread** até que se verifique uma certa condição e entrar protegido numa **secção crítica**. São usadas em conjunto com um **mutex** associado.

As principais funções destes 2 últimos grupos serão analisadas no capítulo sobre sincronização.

Jorge Silva

MIEIC / FEUP

3

Compilação e execução

- Todos os programas que usem chamadas Posix relacionadas com **threads**, devem incluir a seguinte linha de controlo:

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

- Para compilar, por exemplo o programa **prog1.c**, dar o comando:

```
gcc prog1.c -o prog1 -pthread -Wall OU
gcc prog1.c -o prog1 -D_REENTRANT -lpthread -Wall
```

- **-D_REENTRANT** - para incluir a versão reentrante das bibliotecas de sistema
(em alguns compiladores pode não ser necessário)
- **-lpthread** - para "**link**" com a biblioteca Posix de **threads** (**libpthread**)
(**-pthread** ou **-pthreads** em alguns compiladores)

- Valor de retorno das chamadas relacionadas com **threads**:

```
Retorno:
0 se OK
ou um valor positivo (Exxx, definido em errno.h) se erro
```

NOTA: há algumas chamadas que não retornam a quem as invoca (ex: **pthread_exit()**)

Jorge Silva

MIEIC / FEUP

4

Criação de *threads*

```
int pthread_create ( pthread_t *tid, const pthread_attr_t *attr,
                    void * (*func)(void *), void *arg );
```

função de início do *thread*

```
void *func (void *arg) {
    /* CÓDIGO DO THREAD */ ... }
```

tid

- apontador para a identificação do *thread*, retornada pela chamada
- a *tid* é usada noutras chamadas da API de *threads*

attr

- usado para especificar os atributos do *thread* a criar, ex: política de escalonamento, tamanho da *stack*, ...; ver chamadas `pthread_attr_xxx`
- NULL = usar atributos por omissão; é a situação mais frequente

func

- função que o *thread* executará quando for criado
- esta função só admite 1 argumento, que lhe é passado através do parâmetro *arg*

arg

- apontador para o(s) argumento(s) do *thread*; pode ser NULL
- **NOTA:** para passar vários argumentos é necessário compactá-los numa estrutura de dados

Terminação de *threads*

Formas de um *thread* terminar:

- O *thread* retorna normalmente (na função inicial é executada a instrução `return` ou atinge-se a `}` final)
- O *thread* invoca `pthread_exit()`
- O *thread* é "cancelado" por outro *thread*, através de `pthread_cancel()`
- O processo a que o *thread* pertence termina
- O processo a que o *thread* pertence substitui o seu código devido a uma chamada `exec()`

Notas:

- Se `main()` terminar porque executou `exit()`, `_exit()`, `return` ou atingiu a última instrução os *threads* por si criados também terminarão automaticamente.
- No entanto, se `main()` terminar com a chamada `pthread_exit()` os outros *threads* continuarão em execução; as variáveis globais não serão destruídas e os ficheiros abertos não serão fechados.
- Um *thread* pode esperar que outros *threads* terminem usando a chamada `pthread_join()`.

Sistemas Operativos	Threads
<h2 style="margin: 0;">Terminação de <i>threads</i></h2>	
<div style="border: 1px solid black; padding: 5px; margin-bottom: 10px;"> <pre>void pthread_exit (void *status);</pre> <p>Não retorna a quem fez a chamada.</p> </div> <p>status</p> <ul style="list-style-type: none"> valor de retorno, especificando o estado de terminação do <i>thread</i> NULL, quando não se pretende retornar nada <p>NOTAS:</p> <ul style="list-style-type: none"> se a função inicial do <i>thread</i> terminar com <u>return ptr</u>, o valor de <u>status</u> será o apontado por <u>ptr</u> (ver exemplo adiante) <p>⇒ o apontador <u>status</u> <u>não deve apontar para um objecto que seja local ao <i>thread</i></u> pois esse objecto deixará de existir quando o <i>thread</i> terminar</p> <ul style="list-style-type: none"> pthread_exit() não fecha os ficheiros que estiverem abertos 	
Jorge Silva	MIEIC / FEUP

7

Sistemas Operativos	Threads
<h2 style="margin: 0;">Esperando pela terminação de <i>threads</i></h2>	
<div style="border: 1px solid black; padding: 5px; margin-bottom: 10px;"> <pre>int pthread_join (pthread_t tid, void **status);</pre> </div> <ul style="list-style-type: none"> O <i>thread</i> que invocar esta função bloqueia até que o <i>thread</i> especificado por <code>tid</code> termine <p>tid</p> <ul style="list-style-type: none"> <i>thread</i> pelo qual se quer esperar (= valor obtido ao invocar <code>pthread_create()</code>) <p>status</p> <ul style="list-style-type: none"> apontador para apontador para o valor de retorno do <i>thread</i> <p>⇒ Os <i>threads</i> podem ser <i>joinable</i> (por omissão) ou <i>detached</i>. É impossível esperar por um <u><i>detached thread</i></u>. Quando um <u><i>joinable thread</i></u> termina, a sua <i>ID</i> e <i>status</i> são mantidos pelo S.O. até que outro <i>thread</i> invoque <code>pthread_join()</code>.</p> <p>⇒ NOTA: Não há forma de esperar por qualquer um dos <i>threads</i> como acontecia no caso dos processos com as chamadas <code>wait()</code> e <code>waitpid(-1, ...)</code></p>	
Jorge Silva	MIEIC / FEUP

8

Outras chamadas

Um *detached thread* (*thread* separado) é um *thread* pelo qual não é possível esperar.

Quando termina, todos os recursos que lhe estão associados são libertados.

Usando `pthread_detach()` é possível transformar um *joinable thread* em *detached*.

```
int pthread_detach (pthread_t tid);
```

Esta função é frequentemente invocada pelo *thread* que quer passar de *joinable* a *detached*,

o que pode ser conseguido executando `pthread_detach(pthread_self())`

```
pthread_t pthread_self (void);
```

Retorna: *thread ID* do *thread* que fez a chamada

Para criar um *thread* no estado *detached* ao invocar `pthread_create()` é necessário preencher devidamente o atributo `attr` desta chamada.

Exemplo - criação e terminação

NOTA:

nos exemplos que se seguem não são feitos testes de erro nas chamadas para melhorar a legibilidade dos programas

NÃO FAZER ISTO NOS TRABALHOS PRÁTICOS

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

int global;

void *thr_func(void *arg);

int main(void)
{
    pthread_t tid;
    printf("Hello from main thread\n");
    pthread_create(&tid, NULL, thr_func, NULL);
    pthread_exit(NULL);
}

void *thr_func(void *arg)
{
    sleep(3);
    printf("Hello from auxiliar thread\n");
    return NULL;
}
```

NOTA:
desta forma, o *thread* auxiliar
pode continuar a executar
mesmo depois de `main()` terminar

Exemplo - criação de múltiplos *threads*

```

...
void * thrfunc(void * arg)
{
    int i;

    fprintf(stderr, "Starting thread %s\n", (char *) arg);
    for (i = 0; i < 10000; i++) write(1, (char *) arg, 1);
    return NULL;
}

int main()
{
    ...
    int retcode;
    pthread_t ta, tb;
    void * retval;

    retcode = pthread_create(&ta, NULL, thrfunc, "A");
    retcode = pthread_create(&tb, NULL, thrfunc, "B");
    ...
    retcode = pthread_join(ta, &retval);
    retcode = pthread_join(tb, &retval);
    ...
    return 0;
}

```

SAÍDA:

```

Starting thread A
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AStarting thread B
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
...
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
.
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
...
...
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB

```

Jorge Silva

MIEIC / FEUP

11

Exemplo - passagem de valores usando variáveis globais

```

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

int global;

void *thr_func(void *arg)
{
    global++;
    printf("Aux thread: %d\n", global);
    return NULL;
}

int main(void)
{
    pthread_t tid;
    global = 10;
    printf("Main thread: %d\n", global);
    pthread_create(&tid, NULL, thr_func, NULL);
    printf("Main thread: %d\n", global);
    return 0;
}

```

Qual é o problema ?

Jorge Silva

MIEIC / FEUP

12

Exemplo - passagem de valores usando variáveis globais

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

int global;

void *thr_func(void *arg);

int main(void)
{
    pthread_t tid;
    global = 10;
    printf("Main thread: %d\n", global);
    pthread_create(&tid, NULL, thr_func, NULL);
    pthread_join(tid, NULL);
    printf("Main thread: %d\n", global);
    return 0;
}

void *thr_func(void *arg)
{
    global++;
    printf("Aux thread: %d\n", global);
    return NULL;
}
```

o *thread* principal
esperou
que o *thread* auxiliar
terminasse

o programa
pode terminar
sem problema

Exemplo - passagem de valores usando variáveis globais

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

int global;

void *thr_func(void *arg)
{
    printf("Aux thread: %d\n", global);
    return NULL;
}

int main(void)
{
    pthread_t tid;
    global = 20;
    printf("Main thread: %d\n", global);
    pthread_create(&tid, NULL, thr_func, NULL);
    pthread_exit(NULL);
}
```

NOTAR

Sistemas Operativos	Threads
Exemplo - passagem / retorno de valores em argumentos	
<pre>#include <stdio.h> #include <stdlib.h> #include <pthread.h> void *thr_func(void *arg); int main(void) { pthread_t tid; int k = 10; void *r; pthread_create(&tid, NULL, thr_func, &k); pthread_join(tid, &r); printf("Main thread: %d\n", *(int *)r); free(r); return 0; }</pre>	<pre>int pthread_join (pthread_t tid, void **status); void *thr_func(void *arg) { void *ret; int value; value = *(int *) arg; printf("Aux thread: %d\n",value); value++; ret = malloc(sizeof(int)); *(int *)ret = value; return ret; }</pre>
<div style="border: 1px solid black; padding: 5px; display: inline-block; margin: 10px;">NOTAR</div>	
Jorge Silva	MIEIC / FEUP

15

Sistemas Operativos	Threads
Exemplo - passagem de argumentos	
<pre>#include <pthread.h> #include <stdio.h> #define NUM_THREADS 5 void *PrintHello(void *threadnum) { printf("Thread %d: Hello World!\n", *(int *)threadnum); pthread_exit(NULL); } int main() { pthread_t tid[NUM_THREADS]; int t; for(t=0; t<NUM_THREADS; t++){ printf("Creating thread %d\n", t); pthread_create(&tid[t], NULL, PrintHello, &t); } ... }</pre>	
<div style="border: 1px solid black; padding: 10px; display: inline-block;"> <p>PASSAGEM INCORRECTA DE ARGUMENTOS:</p> <p>Porquê ?</p> <p>Qual a solução ?</p> </div>	
Jorge Silva	MIEIC / FEUP

16

Sistemas Operativos	Threads
Exemplo - passagem de argumentos	
<pre>#include <pthread.h> #include <stdio.h> #define NUM_THREADS 5 void *PrintHello(void *threadnum) { printf("Thread %d: Hello World!\n", *(int *)threadnum); pthread_exit(NULL); } int main() { pthread_t tid[NUM_THREADS]; int t; for(t=0; t<NUM_THREADS; t++){ printf("Creating thread %d\n", t); pthread_create(&tid[t], NULL, PrintHello, &t); } ... }</pre>	<pre>... int thrarg[NUM_THREADS]; for(t=0; t < NUM_THREADS; t++) { thrarg[t] = t; printf("Creating thread %d\n", t); pthread_create(&threads[t], NULL, PrintHello, &thrarg[t]); } ...</pre>
<div style="border: 1px solid black; padding: 5px; width: fit-content; margin: 10px auto;"> PASSAGEM INCORRECTA DE ARGUMENTOS: o ciclo que cria os <i>threads</i> modifica o conteúdo do endereço passado como argumento possivelmente antes de o <i>thread</i> criado conseguir aceder-lhe </div> <div style="border: 1px solid black; padding: 5px; width: fit-content; margin: 10px auto;"> PASSAGEM CORRECTA DE ARGUMENTOS ? ... depende de quando terminar este código </div>	
Jorge Silva	MIEIC / FEUP

17

Sistemas Operativos	Threads
Exemplo - passagem de argumentos	
<pre>... int main(void) { int thrarg[NUM_THREADS]; for(t=0; t < NUM_THREADS; t++) { thrarg[t] = t; printf("Creating thread %d\n", t); pthread_create(&threads[t], NULL, PrintHello, &thrarg[t]); } ... }</pre>	<pre>... int main(void) { int *thrarg[NUM_THREADS]; ... for(t=0; t < NUM_THREADS; t++) { thrarg[t] = (int *) malloc(sizeof(int)); *thrarg[t] = t; printf("Creating thread %d\n", t); pthread_create(&threads[t], NULL, PrintHello, thrarg[t]); } ... }</pre>
<div style="border: 1px solid black; padding: 5px; width: fit-content; margin: 10px auto;"> PASSAGEM CORRECTA DE ARGUMENTOS </div> <div style="border: 1px solid black; padding: 5px; width: fit-content; margin: 10px auto;"> SOLUÇÃO ALTERNATIVA Quando se justifica que o espaço p/os argum.s seja reservado no "heap"? </div>	
Jorge Silva	MIEIC / FEUP

18

Sistemas Operativos
Threads

Exemplo - criação e terminação

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

int global=0;

void *thr_func(void *arg)
{
    while (global++ < 20) {
        printf("t%d - %d\n",*(int *)arg,global); sleep(1);
    }
    return NULL;
}

int main(void)
{
    pthread_t tid1, tid2;
    int t1=1, t2=2; //thread number

    printf("Hello from main thread\n");
    pthread_create(&tid1, NULL, thr_func, (void *)&t1);
    pthread_create(&tid2, NULL, thr_func, (void *)&t2);
    ... // Wait for both threads
}
```

SAÍDAS:

```
Hello from main thread
t1 - 1
t2 - 2
t2 - 3
t1 - 4
t1 - 5
t2 - 6
t2 - 7
t1 - 8
t1 - 9
t2 - 10
t2 - 11
t1 - 12
t1 - 13
t2 - 14
t2 - 15
t1 - 16
t1 - 17
t2 - 18
t2 - 19
t1 - 20
```

Haverá aqui algum "perigo" na utilização da variável global?

Jorge Silva
MIEIC / FEUP

19

Sistemas Operativos
Threads

Exemplo – resultados inesperados ...?

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

int global=0;

void *thr_func(void *arg)
{
    while (global++ < 20) {
        printf("t%d - %d\n",*(int *)arg,global); sleep(1);
    }
    return NULL;
}

int main(void)
{
    pthread_t tid;
    int t1=1, t2=2; //thread number

    printf("Hello from main thread\n");
    pthread_create(&tid, NULL, thr_func, (void *)&t1);
    pthread_create(&tid, NULL, thr_func, (void *)&t2);
    ... // wait for both threads
}
```

SAÍDAS:

num comput. dif. do ant.
com sleep() sem sleep()

```
Hello ...
t2 - 2
t1 - 1
t2 - 3
t1 - 4
t2 - 5
t1 - 6
t2 - 7
t1 - 8
t2 - 9
t1 - 10
t2 - 11
t1 - 12
t2 - 13
t1 - 14
t2 - 15
t1 - 16
t2 - 17
t1 - 18
t2 - 19
t1 - 20
```

```
Hello ...
t2 - 1
t2 - 3
t2 - 4
t2 - 5
t2 - 6
t2 - 7
t2 - 8
t2 - 9
t2 - 10
t2 - 11
t1 - 2
t1 - 13
t1 - 14
t1 - 15
t1 - 16
t1 - 17
t1 - 18
t1 - 19
t1 - 20
t2 - 12
```

X

Jorge Silva
MIEIC / FEUP

20

Exemplo - passagem de argumentos múltiplos

```

...
struct thread_data {
    int  thread_num;
    int  value;
    char message[50]; };

struct thread_data thr_data_array[NUM_THREADS];

void *PrintHello(void *thread_arg)
{
    struct thread_data *my_data;
    ...
    my_data = (struct thread_data *) thread_arg;
    tasknum = my_data->thread_num;
    value = my_data->value;
    hello_msg = my_data->message;
    ...
}

int main()
{
    ...
    thread_data_array[t].thread_num = t;
    thread_data_array[t].value = ...;
    strcpy(thread_data_array[t].message, ...);
    pthread_create(&threads[t], NULL, PrintHello, (void *) &thr_data_array[t]);
    ...
}

```

Notas finais

- ➡ • As funções invocadas num *thread* têm de ser ***thread-safe***.
- As funções ***thread-unsafe*** podem ser classificadas em 4 classes:
 - » Classe 1 - não protegem variáveis partilhadas
 - » Classe 2 - baseiam-se na persistência de estado entre invocações
 - » Classe 3 - retornam um apontador para uma variável estática
 - » Classe 4 - invocam funções *thread-unsafe*
- Uma **função** diz-se **reentrante** se puder ser parcialmente executada por uma tarefa, “reentrada” por outra tarefa e depois continuada pela tarefa original. Isto requer que a informação de estado seja guardada na *stack*, não em variáveis globais ou *static*.
As funções reentrantes são um sub-conjunto das funções *thread-safe*
- A maior parte das chamadas de sistema em Unix/Linux são ***thread-safe*** com poucas excepções
 - » EX: `asctime`, `ctime`, `gethostbyaddr`, `gethostbyname`, `inet_ntoa`, `localtime`, `rand`
 Destas, todas pertencem à Classe 3 (acima) com excepção de `rand` que pertence a Classe 2. Para estas funções existe normalmente uma função reentrante com o mesmo nome acrescido de `_r` (ex: `ctime_r`).

Threads & Signals

- Dealing with signals can be complicated even with a process-based paradigm. Introducing threads into the picture makes things even more complicated.
- Each thread has its own signal mask (see `pthread_sigmask()`), but the signal disposition is shared by all threads in the process.
 - This means that individual threads can block signals, but when a thread modifies the action associated with a given signal, **all threads share the action**.
 - Thus, if one thread chooses to ignore a given signal, another thread can undo that choice by restoring the default disposition or installing a signal handler for the signal.
- **Signals are delivered to a single thread in the process.**
 - If the signal is related to a hardware fault or expiring timer, the signal is sent to the thread whose action caused the event.
 - **Other signals**, on the other hand, are **delivered** to an arbitrary thread.
- To send a signal to a thread, we call `pthread_kill(tid, signo)`.

Jorge Silva

MIEIC / FEUP

23

C++11 Threads

- **C++ includes built-in support for**
 - **threads**
 - **mutual exclusion**
 - **condition variables and**
 - **futures**

TO COMPILE:

`g++ prog.cpp -pthread -std=c++11 -Wall -o prog`

```
#include <iostream>
#include <thread>
using namespace std;

void thrFunc()
{
    cout << "In aux thread" << endl;
}

int main()
{
    thread t(thrFunc);
    cout << "In main thread ..." << endl;
    t.join();
    cout << "...back to main thread" << endl;
    return 0;
}
```

```
#include <iostream>
#include <thread>

using namespace std;

void func(int x)
{
    cout << "Inside thread: received
           parameter = " << x << endl;
}

int main()
{
    int i = 10;
    cout << "Launching thread ... parameter
=" << i << endl;
    thread t(func, i);
    t.join();
    cout << "Thread ended" << endl;
    return 0;
}
```

Jorge Silva

MIEIC / FEUP

24

C++11 Threads - parameters

```
#include <iostream>
#include <thread>
#include <string>
using namespace std;

// The thread function can have multiple parameters
// ... but all them are passed "by value"
// string parameter may be "const string &s"

void func(int i, double d, string s)
{
    cout << i << " ", " << d << " ", " << s << endl;
}

int main()
{
    thread t(func, 10, 1.75, "hello");
    t.join();

    return 0;
}
```

```
#include <iostream>
#include <thread>
#include <string>
using namespace std;

// To pass a parameter by reference
// it must be wrapped in a std::ref object
// (see the call below)

void func(int &i, double &d, string &s)
{
    cout << i << " ", " << d << " ", " << s << endl;
    i++;
    d--;
    s = s + " world";
}

int main()
{
    int a = 10; double b = 1.75; string c = "hello";
    thread t(func, ref(a), ref(b), ref(c));
    t.join();
    cout << a << " ", " << b << " ", " << c << endl;
    return 0;
}
```