

Sistemas Operativos Threads

# **Objectivos**

No final desta aula, os estudantes deverão ser capazes de:

- escrever programas que usem um ou múltiplos threads ( multithreaded )
- passar dados (entrada/saída) entre threads
- identificar os problemas que se colocam na manipulação de dados comuns e na passagem de dados entre threads e resolver esses problemas
- usar um mecanismo básico de sincronização entre threads (pthread\_join())



FEUP

MIEIC

Threads

### **Threads**

- A Pthreads API está definida na norma ANSI / IEEE POSIX 1003.1 (1995)
- As funções desta API (mais de 60) 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).
- MIITEXES
  - 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.

**FEUP** 

Faculdade de Engenharia da Universidade do Porto

Sistemas Operativos

Threads

**MIEIC** 

## 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 progl.c, dar o comando:

gcc prog1.c -o prog1 -D\_REENTRANT -lpthread -Wall

- -D\_REENTRANT para incluir a versão reentrante das bibliotecas de sistema (http://pauillac.inria.fr/~xleroy/linuxthreads/faq.html; em alguns compiladores pode não ser necess.)
- -lpthread para "linkar" 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())

Faculdade de Engenharia da Universidade do Porto

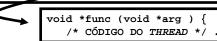
Page

**Threads** 

## Criação de threads

int pthread\_create ( pthread\_t \*tid, const pthread\_attr\_t \*attr, void \* (\*func)(void \*), void \*arg );

função de início do thread

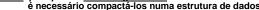


- apontador para a identificação do thread, retornada pela chamada
- a tid é usada noutras chamadas da API de threads

- usado para especificar os atributos do thread a criar, ex: politica de escalonamento, tamanho da stack, ...; ver chamadas pthread\_attr\_xxx
- NULL = usar atributos por defeito; é a situação mais frequente

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

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





**FEUP** 

**MIEIC** 

Faculdade de Engenharia da Universidade do Porto

### Sistemas Operativos

Threads

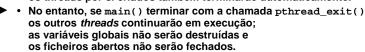
# Terminação de threads

### 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 execxx()

#### Notas:

 Se main() terminar porque executou exit(), \_exit(), return ou atingiu a última instrução os threads por si criados também terminarão automaticamente.



Um thread pode esperar que outros threads terminem usando a chamada  ${\tt pthread\_join()}$ .



**Threads** 

# Terminação de threads

void pthread\_exit (void \*status);

Não retorna a quem fez a chamada.

#### status

- valor de retorno, especificando o estado de terminação do thread
- · NULL, quando não se pretende retornar nada

#### NOTAS:

- se a função inicial do thread terminar com return ptr o valor de <u>status</u> será o apontado por <u>ptr</u> (ver exemplo adiante)
- o apontador status não deve apontar para um objecto que seja local ao thread pois esse objecto deixará de existir quando o thread terminar
- pthread\_exit() não fecha os ficheiros que estiverem abertos



**FEUP** 

Faculdade de Engenharia da Universidade do Porto

Sistemas Operativos

**Threads** 

## Esperando pela terminação threads

int pthread\_join (pthread\_t tid, void \*\*status);

• O thread que invocar esta função bloqueia até que o thread especificado por tid termine

thread pelo qual se quer esperar
(= valor obtido ao invocar pthread\_create())

status

- · apontador para apontador para o valor de retorno do thread
- Os threads podem ser joinable (por defeito) ou detached.

É impossível esperar por um detached thread.

Quando um joinable thread termina a sua ID e status são mantidos pelo S.O. até que outro thread invoque pthread\_join().

NOTA: Não há forma de esperar por qualquer um dos threads como acontecia no caso dos processos com as chamadas wait() e waitpid(-1,...)



Threads

**MIEIC** 

### **Outras chamadas**

Um <u>detached thread</u> (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
```

FEUP

**FEUP** 

Para criar um thread no estado detached ao invocar pthread\_create() é necessário preencher devidamente o atributo attr desta chamada.

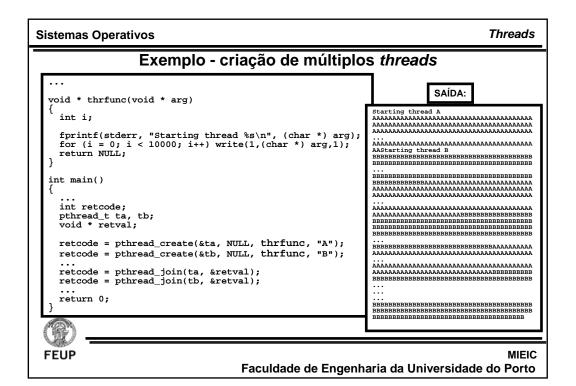
devidamente o atributo acci desta chamada.

Faculdade de Engenharia da Universidade do Porto

Faculdade de Engenharia da Universidade do Porto

**Threads** Sistemas Operativos 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** #include <stdio.h>
#include <pthread.h>
#include <unistd.h> TRABALHOS PRÁTICOS 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); NOTA: desta forma, o thread auxiliar void \*thr\_func(void \*arg) pode continuar a executar mesmo depois de main() terminar return NULL;

Page

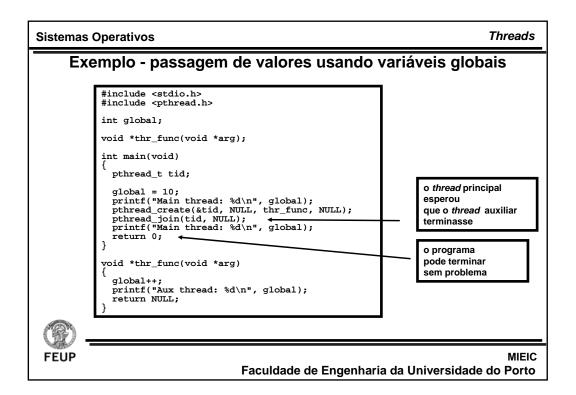


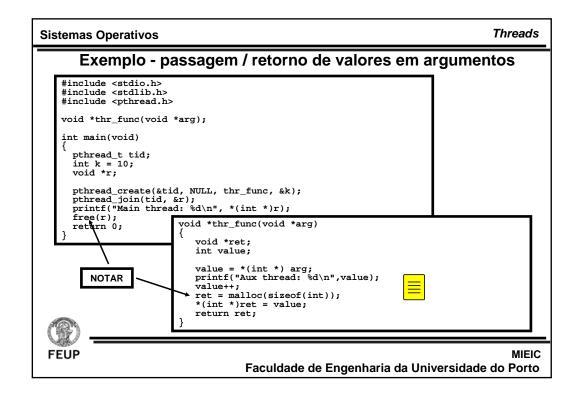
Exemplo - passagem de valores usando variáveis globais

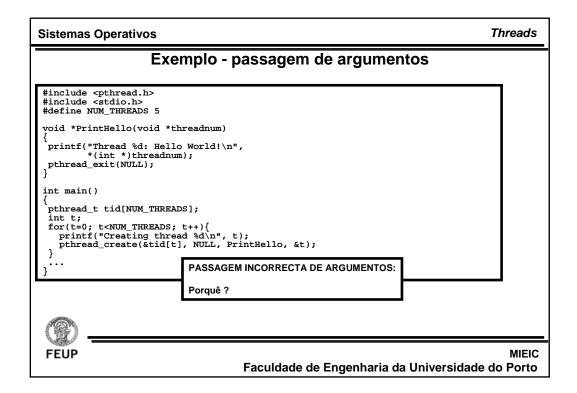
#include <stdio.h>
#include <pthread.h>
int global;
void \*thr\_func(void \*arg);
int main(void)
{
pthread\_t tid;
global = 20;
printf("Main thread: %d\n", global);
pthread\_exit(NULL);
}

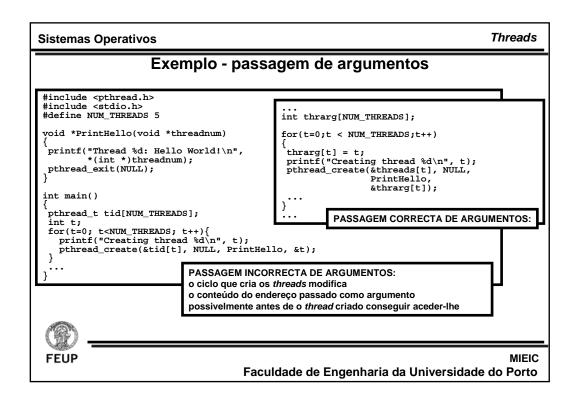
void \*thr\_func(void \*arg)
{
printf("Aux thread: %d\n", global);
return NULL;
}

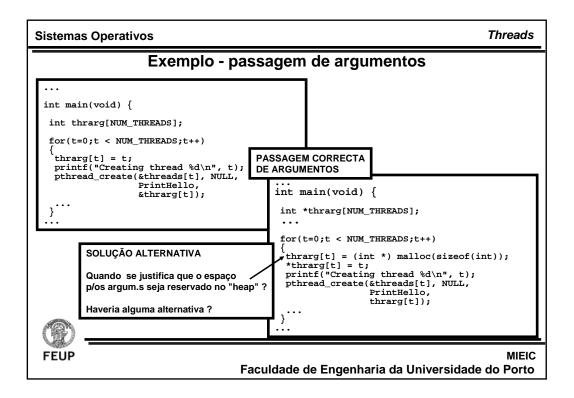
MIEIC
Faculdade de Engenharia da Universidade do Porto

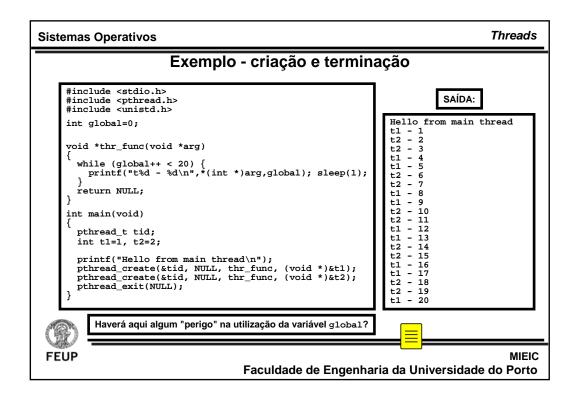












**Threads** 

### Exemplo - passagem de argumentos múltiplos

```
struct thread_data {
   int thread_num;
   int value;
   char *message; };

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 = x;
   thread_data_array[t].wssage = messages[t];
   pthread_create(&threads[t], NULL, PrintHello, (void *) &thr_data_array[t]);
   ...
}
```

**FEUP** 

**MIEIC** 

Faculdade de Engenharia da Universidade do Porto

#### **Sistemas Operativos**

Threads

### **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 <u>função</u> diz-se <u>reentrante</u> se e só se <u>não aceder a variáveis partilhadas</u> quando invocada por várias *threads*; as funções reentrantes são um sub-conjunto das funções *thread-safe*
- A maior parte das chamadas de sistema em Unix 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).



**FEUP** 

MIEIC