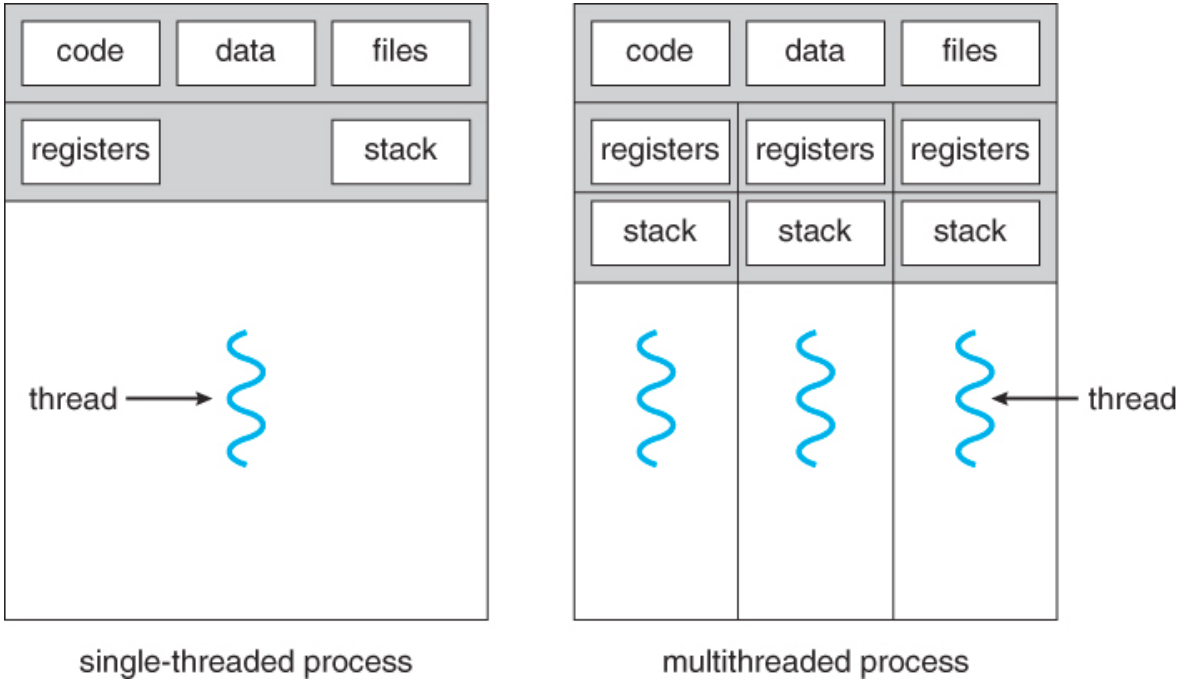
POSIX Threads

O que é um thread?

- Um **thread** (fio-de-execução, processo leve) é um **conjunto independente de instruções** (um fluxo de execução) que pode ser **escalonado** (agendado) pelo sistema operativo
- Na **perspetiva do programador**, um thread é como uma **função que corre de forma independente** do programa principal
- Um programa **multithreaded** é uma aplicação que executa múltiplos threads **simultaneamente** e/ou de forma **independente** dentro de um mesmo processo
- Os threads são escalonados pelo sistema operativo e executam como entidades independentes, **duplicando** apenas **os recursos estritamente necessários** que possibilitam essa execução

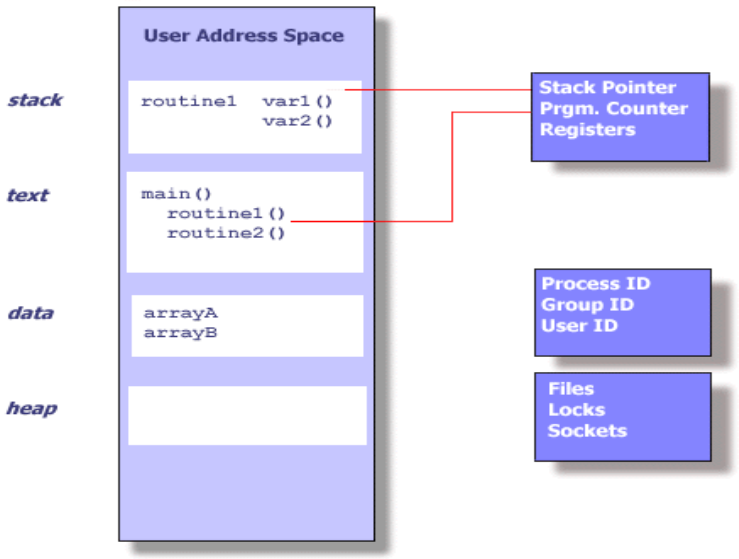
O que é um thread?

- Os threads existem dentro de um processo
- Um processo tem pelo menos o **thread principal** (**main thread**)
- Qualquer thread pode criar outros threads

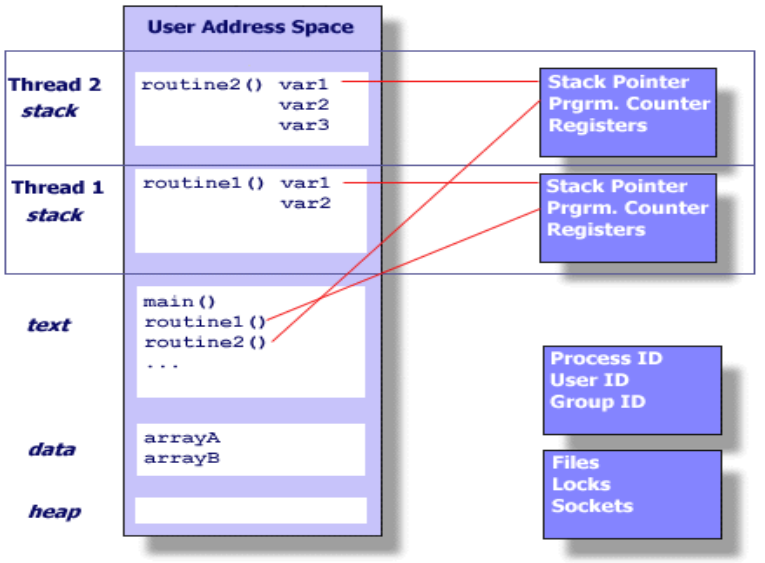


O que é um thread?

- Os threads existem dentro de um processo - exemplo



processo single-threaded



processo dual-threaded

- Recursos globais, usados por todo o processo (instruções, **dados globais**, **heap**, ficheiros abertos, …) vs. recursos específicos a cada thread (program counter, stack pointer, **stack**, ...)

Pthreads em Linux

- A norma **IEEE POSIX 1003.1c** especifica uma API com as funções básicas para criar e controlar threads em sistemas UNIX
- As implementações conformes concretizam aquilo que chamamos de **POSIX threads** ou **pthreads**
- Cabeçalho necessário: **`<pthread.h>`**
- Compilar com **`gcc ... myprog.c -pthread`** (não usar **`-lpthread`**)
- Para mais detalhes: **`man 7 pthreads`**

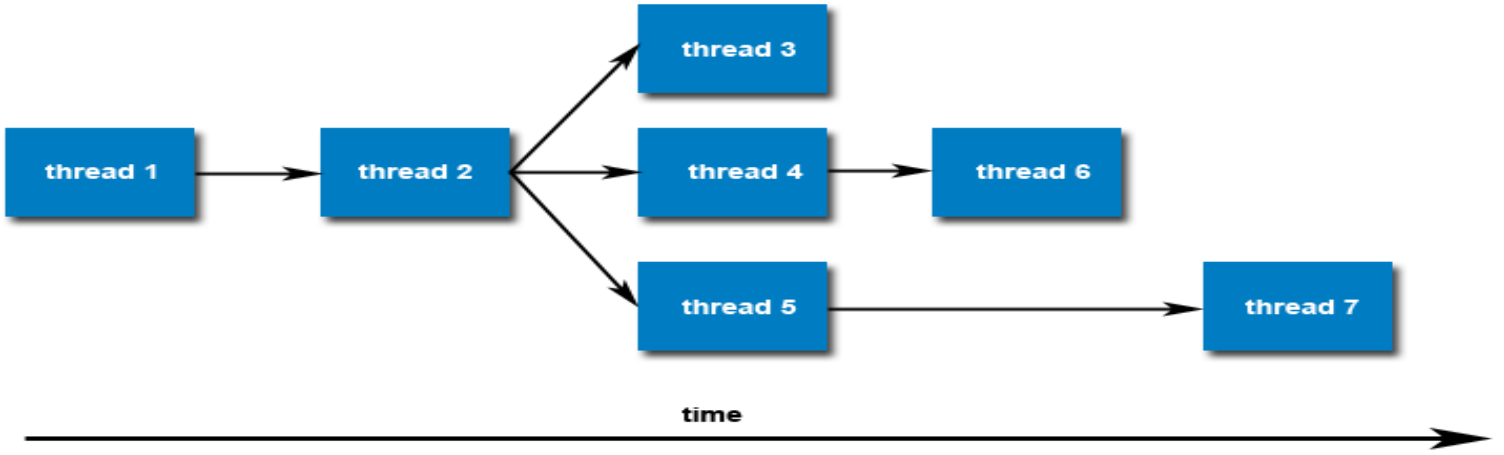
Criação de threads

```
int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void *(*start_routine) (void *),
                   void *arg)
```

- **`thread`** é um identificador opaco único da nova thread devolvido por `pthread_create` (no Linux é do tipo `unsigned long int`)
- **`attr`** é um objeto opaco de atributos usado para definir propriedades da thread; colocar `NULL` para usar os atributos por omissão
- **`start_routine`** é a função onde a thread começa a executar
- **`arg`** é o argumento único que pode ser passado à função `start_routine`; deve ser passado como apontador convertido para void*; usa-se NULL se não houver argumento a passar
- Em caso de sucesso, retorna `0`; em caso de erro, retorna um código de erro
- **Opaco**: conteúdo interno escondido, dependente da implementação, só podendo ser manipulado através das operações fornecidas pela biblioteca

Criação de threads

- Por omissão, a função **`main()` executa num único thread**; todos os outros threads devem ser criados explicitamente pelo programador
- **`pthread_create()`** pode ser chamada **qualquer número de vezes**, em qualquer ponto do programa
- Após uma chamada a pthread_create(), **é indeterminado qual thread será executado a seguir**; todos os threads são iguais (pares/*peers*), podem criar outros threads e **`não existe hierarquia`** ou dependência implícita entre eles



Terminação de threads

- **`pthread_exit()`** termina o thread, opcionalmente retornando um valor; o `main()` deve usá-lo se se pretende deixar que os outros threads continuem a executar
- Se um thread (exceto o main) faz **`return`** da sua `start_routine`, isso equivale a chamar `pthread_exit()` usando o valor retornado como estado do thread. Já um **`return`** na função `main()` termina imediatamente o processo, encerrando todos os threads
- **`pthread_cancel()`** permite que um thread cancele outro (não será abordado em SO)
- **`exit()`** termina o processo e inerentemente todos os seus threads
- Quando um thread termina, os recursos partilhados do processo (mutexes, variáveis de condição, semáforos, descritores de ficheiro) não são libertados automaticamente
- Quando o último thread de um processo termina, o processo termina

Terminação de threads

`void pthread_exit(void *retval)`

- **`retval`** é um **apontador para o estado de retorno do thread**, disponível para qualquer outro thread do mesmo processo que faça **join** ao thread terminado
- Pode retornar **`NULL`**, caso não haja estado útil a transmitir
- **`retval`** deve apontar para **memória alocada dinamicamente** (heap) ou para uma **variável global** (segmento de dados) e **não para uma variável na stack** (variável local que é destruída na terminação do thread)

Exemplo 1

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

void *aThread(void *arg){
    printf("Process %d: new thread: Hello World !\n", getpid()); // A
    pthread_exit(NULL);
}

int main(){
    pthread_t thread_opaque_id;
    int ret;

    printf("Process %d: main thread: creating a new thread\n", getpid()); // B
    ret = pthread_create(&thread_opaque_id, NULL, aThread, NULL);
    if(ret){
        printf("ERROR; return code from pthread_create() is %d\n", ret);
        exit(ret);
    }

    printf("Process %d: main thread: program completed: exiting\n", getpid()); // C
    pthread_exit(NULL);    /* Last thing that main() should do */
}
```

Processo com 2 threads. O thread **`main`** cria um novo thread usando **`pthread_create`**. O novo thread inicia a execução da função **`aThread`**. Ambos threads terminam com **`pthread_exit`**. O novo thread não recebe nenhum parâmetro do **`main`**.
Q1: quais são as ordens de execução possíveis de **A**, **B** e **C**?
Q2: o que pode acontecer se a função **`main`** não chamar **`pthread_exit`**?

Em exemplos futuros vamos omitir a verificação de erros, por uma questão de simplificação

Exemplo 2

Cria 5 threads com **pthread_create**. Cada thread imprime o PID do processo anfitrião e o pseudo-ID recebido como parâmetro, terminando de seguida com **pthread_exit**.
O parâmetro "**(void*)t**" em **pthread_create** é passado (copiado) por valor à stack do novo thread e assim não há risco de "**t**" ser modificado no ciclo do thread principal (**t++** só acontece depois de **t** ser copiado para a stack do novo thread).

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#define NUM_THREADS 5

void *aThread(void *arg){
    long tid=(long)arg; // don't *(long*)arg;
    printf("Process %d: thread %ld: Hello World !\n", getpid(), tid);
    pthread_exit(NULL);
}

int main(){
    pthread_t threads[NUM_THREADS];
    long t;

    for(t=0; t<NUM_THREADS; t++){
        printf("Process %d: thread main: creating thread %ld\n", getpid(), t);
        pthread_create(&threads[t], NULL, aThread, (void *)t); // don't (void *)&t
        // note that sizeof(int) <= sizeof(void*)
    }

    printf("Process %d: thread main: program completed: exiting\n",getpid());
    pthread_exit(NULL);
}
```

Junção de threads

```
int pthread_join(pthread_t thread, void **retval)
```

- Permite a **sincronização** entre threads (outros mecanismos incluem mutexes e variáveis de condição)
- Bloqueia o thread chamador até que o **`thread`** especificado termine; se o **`thread`** alvo já tiver terminado, retorna imediatamente; o **`thread`** alvo tem de ser **joinable** (não pode ser **detached**)
- Se **`retval`** ≠ **`NULL`**, o valor de saída do **`thread`** é copiado para o local por ele apontado; o valor pode ser um inteiro convertido para apontador ou um verdadeiro apontador
- Um **thread terminado só pode ser junto uma única vez**; múltiplos joins sobre o mesmo thread não são permitidos (nem pelo mesmo thread, nem por threads diferentes)

Exemplo 3

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

#define NUM_THREADS 5

void *aThread(void *arg){
    long tid = (long)arg;
    printf("thread %ld: Hello World !\n", tid);
    pthread_exit(NULL);
}
```

Mostra como esperar pela terminação de um thread usando **`pthread_join`**.
Q: A ordem dos joins importa?

Neste exemplo, os threads criados retornam NULL

A frase **(*)** é sempre impressa no fim.
Q: e se comentarmos o último ciclo for?

```c
int main(){
 pthread_t threads[NUM_THREADS];
 long t; int ret;

 for(t=0;t<NUM_THREADS;t++){
     printf("thread main: creating thread %ld\n", t);
     pthread_create(&threads[t], NULL, aThread, (void *)t);
 }

 // wait for the other threads
 for(t=0; t<NUM_THREADS; t++){
     ret = pthread_join(threads[t], NULL);
     if(ret){
        printf("ERROR; return code from pthread_join() is %d\n", ret);
        exit(ret);
     }
     printf("thread main: joined with thread %ld\n", t);
 }

 printf("thread main: program completed: exiting\n"); // (*)
 pthread_exit(NULL);
}
```

Em exemplos futuros vamos omitir a verificação de erros, por uma questão de simplificação

Exemplo 4

Esta abordagem permite a um thread retornar qualquer tipo de dados, uma vez que o que realmente é retornado é um apontador; para confirmar que funciona, use o código **B** em vez do **A**.

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

#define NUM_THREADS 5

void *aThread(void *arg){
 long tid = (long)arg;
 long *square; // A
 //double *square; // B

 square=(long*)malloc(sizeof(long)); // A
 //square=(double*)malloc(sizeof(double)); // B
 *square = tid*tid;
 printf("thread %ld: Hello World !\n", tid);
 pthread_exit((void*)square);
}
```

Neste exemplo, cada thread secundário reserva memória dinâmica (heap) para um valor e retorna o apontador para esse valor. O thread principal usa esse apontador para aceder ao valor de retorno. É responsabilidade do thread principal a libertação da memória dinâmica (porquê?).

```c
int main(){
 pthread_t threads[NUM_THREADS];
 long t; void *status;

 for(t=0;t<NUM_THREADS;t++){
     printf("thread main: creating thread %ld\n", t);
     pthread_create(&threads[t], NULL, aThread, (void *)t);
 }

 // wait for the other threads
 for(t=0; t<NUM_THREADS; t++){
     pthread_join(threads[t], &status);
     printf("thread main: joined with thread %ld: \
             exit status is: %ld\n", t, *(long*)status); // A
     //printf("thread main: joined with thread %ld: \
             //exit status is: %f\n", t, *(double*)status); // B
     free(status);
 }

 printf("thread main: program completed: exiting\n");
 pthread_exit(NULL);
}
```

Exemplo 5

Esta abordagem também é compatível com qualquer tipo de dados; todavia, devemos usar variáveis globais com cautela: para além de propensas a condições de corrida, o uso intensivo deste tipo de variáveis penaliza o desempenho (assim, variáveis locais e alocação de memória dinâmica são abordagens preferíveis).

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define NUM_THREADS 5

long squares[NUM_THREADS]; // A
//double squares[NUM_THREADS]; // B

void *aThread(void *arg){
 long tid = (long)arg;
 squares[tid] = tid*tid
 printf("thread %ld: Hello World !\n", tid);
 pthread_exit(NULL);
}
```

Neste exemplo, os valores de retorno são guardados num array global partilhado por todos os threads; cada thread escreve numa posição distinta, evitando assim condições de corrida.

```
int main(){
 pthread_t threads[NUM_THREADS];
 long t;

 for(t=0;t<NUM_THREADS;t++){
     printf("thread main: creating thread %ld\n", t);
     pthread_create(&threads[t], NULL, aThread, (void *)t);
 }

 for(t=0; t<NUM_THREADS; t++){
     pthread_join(threads[t], NULL);
     printf("thread main: joined with thread %ld: \
             thread square is: \
             %ld\n", t, squares[t]); // A
     //printf("thread main: joined with thread %ld: \
             //thread square is: \
             //%f\n", t, squares[t]);  // B
 }

 printf("thread main: program completed: exiting\n");
 pthread_exit(NULL);
}
```

Exercícios

5.1. Codifique um programa

a) que crie 10 000 processos e

b) outro que crie 10 000 threads,

sendo que processos e threads devem executar a mesma tarefa após serem criados. Compare o tempo de execução de ambos os programas utilizando o comando time e retire conclusões.

Exercícios

5.2. Construa um programa que crie três threads, além do thread principal, e que:

a) receba um inteiro diferente como parâmetro em cada thread criado;

b) receba dois inteiros como parâmetros em cada thread criado.

Exercícios

5.3. (variante do exercício 2.5) Codifique um programa que recorra a dois threads para calcular a soma dos primeiros 100 números inteiros positivos. O thread principal deverá calcular a soma dos primeiros 50 números e o thread secundário deverá calcular a soma dos últimos 50 números. O thread principal deverá apresentar a soma global.

a) Utilize uma variável global para partilhar o valor calculado pelo thread secundário.

b) Faça com que o thread secundário retorne o valor calculado, de forma a ser utilizado pelo thread principal.

Exercícios

5.4. (variante do exercício 2.6) Codifique um programa que utilize 10 threads (para além do main thread) para calcular a soma dos primeiros 100 números positivos, à custa de somas parciais: o primeiro thread calculará a soma de 1 a 10, o segundo calculará a soma de 11 a 20, …, e o décimo thread calculará a soma de 91 a 100. O thread principal deverá calcular a soma final e apresentar o resultado.

Exercícios

5.5. (variante do exercício 2.7) Construa um programa que calcule a soma dos quadrados dos primeiros 10 números positivos utilizando threads. O programa deve criar dez threads secundários, devendo o cálculo ser efetuado em Round-Robin, de forma que apenas um thread esteja ativo de cada vez. O thread principal cria o primeiro thread secundário. Cada thread secundário calcula o quadrado do seu número correspondente (de 1 a 10) e acumula o resultado numa variável global, passando a vez para o próximo thread secundário. Após o décimo thread calcular o quadrado de 10, a execução retorna ao thread principal, que apresenta a soma final.