# Threads

Um thread é uma sequência de instruções que vão ser executadas num programa. No ambiente UNIX, os threads encontram-se dentro de um processo, utilizando os recursos desse processo. Um processo pode ter vários threads.

Pode-se dizer que um thread é um procedimento que é executado dentro de um processo de uma forma independente. Para melhor perceber a estrutura de um thread é útil entender o relacionamento entre um processo e um thread. Um processo é criado pelo sistema operativo e podem conter informações relacionadas com os recursos do programa e o estado de execução do programa:

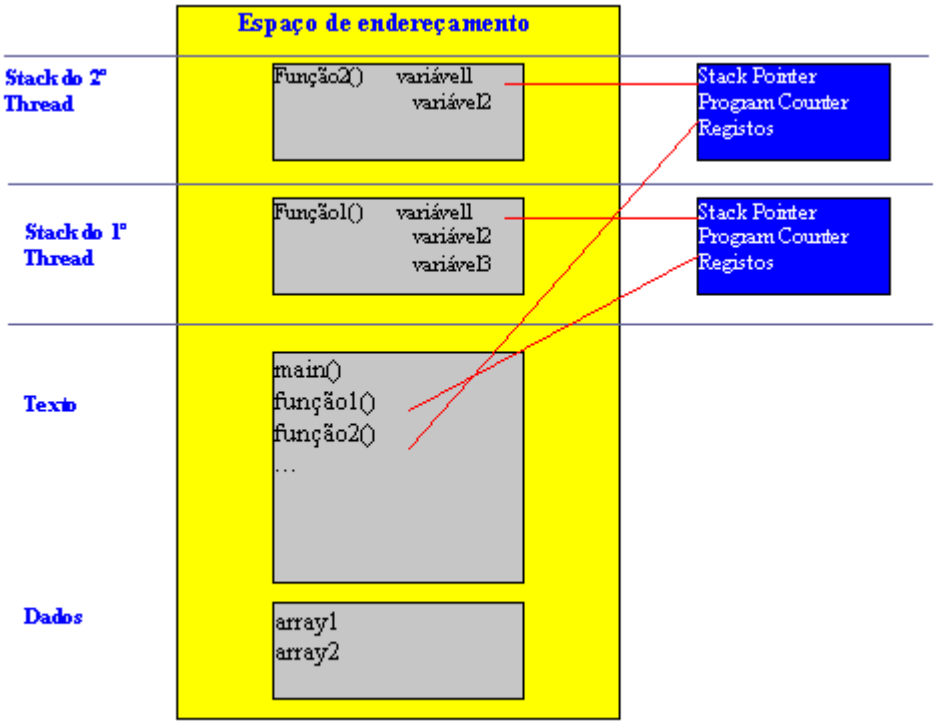
- Process ID, process group ID, group ID
- Ambiente
- Directoria de trabalho
- Instruções do programa
- Registos
- Pilha (Stack)
- Espaço de endereçamento comum, dados e memória
- Descritores de ficheiros
- Comunicação entre processos (pipes, semáforos, memória partilhada, filas de mensagens)

Os threads utilizam os recursos de um processo, sendo também capazes de ser escalonados pelo sistema operativo e serem executados como entidades independentes dentro de um processo.

Um thread pode conter um controlo de fluxo independente e ser escalonável, porque mantêm o seu próprio:

- Pilha
- Propriedades de escalonamento
- Dados específicos do thread

Um processo pode ter vários threads, partilhando cada um os recursos do processo e são executados no mesmo espaço de endereçamento. Com vários threads, temos em qualquer instante vários pontos de execução.




Como os threads partilham os recursos de um processo:

- as alterações feitas por um thread num recurso partilhado (ex: fechar um ficheiro) serão "vistaa" pelos outros threads.

- Dois apontadores com o mesmo valor, apontam para os mesmos dados.
- É possível ler e escrever nas mesmas posições de memória, sendo por isso necessária uma sincronização explícita que tem de ser feita pelo programador.

Todos os threads estão no mesmo nível hierárquico. Um thread não mantém uma lista dos threads criados nem tem conhecimento do thread que o criou.

Vantagens:

- ganhos nas performances dos programas;
- quando comparados com a criação e gestão de um processo, os threads podem ser criados com muito menos sobrecarga para o sistema operativo. A gestão dos threads necessita de menos recursos que a gestão dos processos;
- todos os threads num processo partilham o mesmo espaço de endereçamento. A comunicação entre threads é mais eficiente e na maior parte das situações mais fácil do que a comunicação entre processos;
- As aplicações com threads são mais eficientes e têm mais vantagens práticas:

- Sobrecarregar o CPU com operações de entrada/saída de dados. Por exemplo, um programa tem situações onde necessita de efectuar operações de entrada/saída de dados muito demoradas. Enquanto um thread espera que a operação de entrada/saída de dados termine, o CPU pode ser utilizado para efectuar outras operações através de vários threads;

- Tratamento de eventos assíncronos. Por exemplo, um servidor web pode transmitir dados de pedidos anteriores e ao mesmo tempo fazer a gestão dos novos pedidos.

## Gestão de Threads

Inicialmente, o processo apenas tem um thread (main()). Todos os outros threads são criados explicitamente pelo programador.

#### Criação de threads

**`pthread_create(thread, attr, função, argumento)`**

- Esta função cria um novo thread e torna-o executável. Os threads podem criar outros threads.
- A função retorna a identificação do thread através do parâmetro *thread*. Esta identificação pode ser utilizada para efectuar várias operações nesse thread.
- *attr* é usado para definir as propriedades do thread. Podem-se especificar os atributos ou o NULL para os valores por defeito.
- *função* é o nome da função que vai ser executada pelo thread.
- Pode-se *passar* um argumento para a *função* através de *argumento*. Deve-se fazer sempre o cast para o apontador de um void. Para passar mais do que um argumento, pode-se utilizar uma estrutura que contenha todos os argumentos e depois *passar* um apontador para essa estrutura.

#### Terminar a execução de um thread

Existem várias maneiras de um thread terminar:

- O thread retorna da sua função inicial (a função main para o thread inicial).
- O thread executa a função `pthread_exit`
- O thread foi cancelado por outro thread através da função pthread_cancel
- O thread recebeu um sinal que fez com que terminasse;
- Todo o processo foi terminado devido à utilização da função exec (que substitui a imagem do processo) ou através da função exit (que

termina um processo!)

**`pthread_exit(estado)`**

- Esta função termina explicitamente um thread, Utiliza-se esta função depois de um thread já ter completado o seu trabalho.
- Se o main() termina antes dos threads que foram criados, e termina através da utilização de `pthread_exit()`, os outros threads continuarão a ser executados, caso contrário terminam automaticamente quando main() termina.

**Exemplo 1:** Este exemplo cria três threads, cada um imprimindo a mensagem "Sistemas Operativos II " e o número do thread, terminando de seguida com a função pthread_exit()

```
/* Para compilar fazer: gcc -o fich fich.c -lpthread */



#include <pthread.h>

#include <stdio.h>




void *escreve(void *numero)

{

    int *n =(int *) numero;

    printf("Sistemas Operativos II thread n. = %d\n",*n);

    pthread_exit(NULL);

}



void main()

{

    pthread_t threads[3];

    int s,i;

    for (i=0; i<3; i++)

    {

        printf("A criar o thread n. %d\n",i);

        s=pthread_create(&threads[i], NULL, escreve, (void *) &i);

        if (s)
```

```
        {
            perror("Erro ao criar o thread");
            exit(-1);
        }
    }
    pthread_exit(NULL); /* se a função main não terminar com a função phtread_exit o ultimo
                                    thread não chega a terminar a sua execução */
}
```

**Nota: Neste exemplo pode acontecer que apareça o mesmo número repetido (ex: 233). Isto deve-se ao facto de quando o thread mostra o valor da variável i, este já foi alterado. Para evitar esta situação deve-se utilizar um array de inteiros, sendo cada posição utilizada apenas por um thread.**

**Funções para identificação de threads**

**pthread_self()**

- retorna o identificador desse thread.

**pthread_equal()**

- compara dois identificadores de threads. Se são diferentes retorna 0, senão retorna um valor diferente de 0.

## Sincronização de threads

**pthread_join(thread_id, estado)**

- bloqueia o thread que *chamou* esta função até que o thread com o identificador *thread_id* termine.
- também é possível obter o estado do thread que terminou, se esse terminou com a função pthread_exit(estado).
- Tem um conportamento semelante à função *wait* utilizadanos processos.

Quando se cria um thread, um dos atributos que pode ser alterado é se outros threads podem ou não *chamar* a função pthread_join sobre ele. Para isso são possíveis dois valores:

- Detached – não pode ser feito o join (PTHREAD_CREATE_DETACHED);
- Undetached – pode ser feito o join (PTHREAD_CREATE_JOINABLE)

Funções para manipulação de atributos

**pthread_attr_init(attr)**

**pthread_attr_setdetachstate(attr, detachstate)**

**pthread_attr_destroy(attr)**

**pthread_detach(tread_id, estado)**

Para utilizar  são necessários os seguintes passos:

- Declarar a variável com o tipo de dados pthread_attr_t ;
- Inicializar a variável com a função pthread_attr_init
- Definir o estado do atributo com a função pthread_attr_setdetachstate;
- Libertar os recursos utilizados por essa variável com a função pthread_attr_destroy

**Exemplo 2:** Este exemplo espera que todos os threads terminem através da utilização da função pthread_join.

```
#include <pthread.h>

#include <stdio.h>

void *ocupa_tempo(void *null)

{

      int i,j;

      for (i=0;i<50000;i++)

      j+=i;

      pthread_exit(NULL);

}


void main()

{

    pthread_t threads[5];

    pthread_attr_t attr;

    int s,i,estado;

    pthread_attr_init(&attr);

    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

    for(i=0; i<5; i++)

    {

        printf("A criar o thread %d\n",i);

        s=pthread_create(&threads[i], &attr,ocupa_tempo, NULL );

        if (s)
```

```
        {
            perror("Erro ao criar o thread");
            exit(-1);
        }
    }
    pthread_attr_destroy(&attr);
    for(i=0; i<5; i++)
    {
        s=pthread_join(threads[i], (void **) &estado);
        if (s)
        {
            perror("Erro no join");
            exit(-1);
        }
        printf("O thread %d terminou com o estado %d\n",i,estado);
    }
    pthread_exit(NULL);
}
```

### Variáveis de exclusão mútua

As variáveis de exclusão mútua (mutex) são uma das formas de obter sincronização entre *threads*. Actuam como um semáforo com um único recurso, fechando e protegendo o acesso a um recurso partilhado. Num dado instante apenas um *thread* consegue "fechar" uma variável de exclusão mútua. Estas variáveis podem ser usadas para evitar "race conditions". Segue-se um exemplo de uma "race condition" numa transacção bancária assumindo que o balanço inicial é 10000$00:

| **Instante** | **Thread 1** | **Thread 2** | **Balanço** |
|---|---|---|---|
| 1 | A ← balanço | | 10000$00 |
| 2 | | B ← balanço | 10000$00 |
| 3 | | balanço← B - 5000$00 | 5000$00 |
| 4 | balanço← A + 6000$00 | | 16000$00 |

Neste exemplo deveria ser usada uma variável mutex para "fechar" o "balanço" enquanto é utilizado por cada thread, de modo a evitar situações de incoerência (neste caso o resultado correcto é 11000$00 !).

### Funções para a criação/destruição de variáveis mutex:

**`pthread_mutex_init(mutex,attr)`**

- cria e inicializa uma variável mutex, definindo os seus atributos de acordo com *attr*;
- As variáveis mutex devem ser do tipo **pthread_mutex_t**;
- *attr* é utilizado para definir as propriedades da variável mutex, e deve ser do tipo **pthread_mutexattr_t** se for usada (para utilizar os valores por defeito, pode-se usar NULL em substituição de *attr*)

**`pthread_mutex_destroy(mutex)`**

- destrói a variável mutex.

**`pthread_mutexattr_init(attr)`**

- inicializa attr

**`pthread_mutexattr_destroy(attr)`**

- remove attr

### Funções para a abrir/fechar varáveis mutex:

**`pthread_mutex_lock(mutex)`**

- fecha a variável mutex. Se já estiver "fechada" a função bloqueia até que seja possível fechar.

**`pthread_mutex_trylock(mutex)`**

- tenta fechar a variável mutex. Se já estiver "fechada" esta função não bloqueia e devolve um valor de erro (EBUSY).

**`pthread_mutex_unlock(mutex)`**

- "abre" uma variável mutex. Esta função dará erro se a variável mutex já estiver aberta ou se tentar abrir uma variável mutex que foi "fechada" por outro thread.

**Exemplo 3:** Este exemplo utiliza variáveis de exclusão mútua para garantir a coerência dos dados.

```
#include <pthread.h>

#include <stdio.h>

/* definição das variáveis globais */

int balanco=10000;

pthread_t threads[2];

pthread_mutex_t mutex_balanco;

void *deposita(void *valor)

{
```

```
	int *a= (int *) valor;
	/*fecha a variável mutex_balanco . Se já estiver “fechada” espera até
	que seja possível fechar*/
	pthread_mutex_lock(&mutex_balanco);
	balanco += *a;
	/* “abre” a variável mutex_balanco */
	pthread_mutex_unlock(&mutex_balanco);
	pthread_exit(NULL);

}
void *levanta(void *valor)
{
	int *b= (int *) valor;
	/*fecha a variável mutex_balanco . Se já estiver “fechada” espera até
	que seja possível fechar*/
	pthread_mutex_lock(&mutex_balanco);
	balanco -= *b;
	/* “abre” a variável mutex_balanco */
	pthread_mutex_unlock(&mutex_balanco);
	pthread_exit(NULL);

}

void main()
{
	int i,s,estado,depositar=6000,levantar=5000;
	/* inicializa a variável mutex_balanco utilizando os valores por defeito */
	pthread_mutex_init(&mutex_balanco, NULL);
	pthread_create(&threads[0],NULL,deposita,(void *)&depositar);
	pthread_create(&threads[1],NULL,levanta,(void *)&levantar);
```

```
      for(i=0; i<2; i++)
      {
        s=pthread_join(threads[i], (void **) &estado);
        if (s)
        {
            perror("Erro no join");
            exit(-1);
        }
        printf("O thread %d terminou com o estado %d\n",i,estado);
    }
      printf("O balanço é =  %d\n",balanco);
      pthread_mutex_destroy(&mutex_balanco);
      pthread_exit(NULL);
}
```

### Variáveis de Condição

As variáveis de condição possibilitam outro modo de sincronização de threads. Enquanto as variáveis de exclusão mútua implementam a sincronização através do controlo do acesso aos dados, as variáveis de condição permitem a sincronização de trheads através do valor desses dados. Sem a utilização destas variáveis, os threads têm que estar sempre a verificar se uma dada variável tem um valor específico, ocupando assim tempo de processamento, pois os threads estão constantemente ocupados com esta actividade. Uma variável de condição obtém os mesmos resultados sem a necessidade de estar sempre a verificar o valor.

Nota: Uma variável de condição é sempre usada em conjunto com o fecho de uma variável de exclusão mútua.

### Funções para a criação/destruição de variáveis de condição:

**`pthread_cond_init(variável_de_condição,attr)`**

- cria e inicializa uma variável de condição, definindo os seus atributos de acordo com *attr*;
- As variáveis de condição devem ser do tipo **pthread_cond_t**;
- *attr* é utilizado para definir as propriedades da variável de condição, e deve ser do tipo **pthread_mutexattr_t** se for usada (para utilizar os valores por defeito, pode-se usar NULL em substituição de *attr*)

**`pthread_cond_destroy(variável_de_condição)`**

- destrói a variável de condição.

**`pthread_condattr_init(attr)`**

- inicializa attr

**pthread_condattr_destroy(attr)**

- remove attr

### Funções para a utilização das varáveis de condição:

**pthread_cond_wait(variável_de_condição,mutex)**

- bloqueia o thread até que a variável de condição seja "sinalizada". Esta função deve ser chamada quando a variável de exclusão mútua se encontra "fechada". Esta função "abre" automaticamente a variável mutex de modo a ser utilizada por outros threads, "adormecendo" de seguida o thread.
- Quando chega o "sinal", fecha a variável mutex.

**pthread_cond_signal(variável_de_condição)**

- É utilizada para "sinalizar" (ou acordar) um thread que está à espera numa variável de condição. Deve ser chamada quando a variável de exclusão mútua se encontra "fechada" e deve ser "aberta" a variável de exclusão mútua para permitir que a função pthread_cond_wait do outro thread prossiga.

**pthread_cond_broadcast(variável_de_condição)**

- Deve ser utilizada em substituição da função pthread_cond_signal, quando temos mais que um thread à espera.

**Exemplo 4:** Este exemplo demonstra a utilização de variáveis de condição. São utilizados dois threads para incrementar a variável "contador"

```
#include <pthread.h>

#include <stdio.h>

/* definição das variáveis globais */

int contador=0;

pthread_cond_t condicao_contador;

pthread_t threads[3];

pthread_mutex_t mutex_contador;




void *incrementa(void *valor)

{

      int i,j;
```

```
	for(i=0;i<50;i++)
	{
		pthread_mutex_lock(&mutex_contador);
		contador++;
		printf("O valor do contador e' %d\n",contador);

		if (contador == 10)
		{
			pthread_cond_signal(&condicao_contador);
			printf("O valor do contador e' 10\n");
		}
		pthread_mutex_unlock(&mutex_contador);

	   for(j=0;j<2000;j++);
	}
	 pthread_exit(NULL);

}
void *espera(void *valor)
{
	printf("O thread está à espera que o valor do contador seja 10\n");

	/* Fecha a variável mutex e espera pelo "sinal". Nota: a função pthread_cond_wait "abre"
automaticamente a variável mutex para que possa ser usada por outros threads enquanto espera.
Também é necessário salientar que se o valor de contador já for 10, o ciclo não é feito,
evitando assim uma situação em que o thread ficaria eternamente à espera de um "sinal" */
	pthread_mutex_lock(&mutex_contador);
	while (contador < 10)
	{
		pthread_cond_wait(&condicao_contador,&mutex_contador);
		/* quando esta função receber o "sinal", fecha a variável mutex */
```

```
            printf(“O thread recebeu o ‘sinal’\n”);

      }

      printf(“O contador tem o valor 10\n”);
      pthread_mutex_unlock(&mutex_contador)
      pthread_exit(NULL);

}

void main()
{
      int i,s,estado;
      /* inicializa a variável mutex_balanco utilizando os valores por defeito */
      pthread_mutex_init(&mutex_contador, NULL);
      pthread_cond_init(&condicao_contador,NULL);
      pthread_create(&threads[0],NULL,incrementa,NULL);
      pthread_create(&threads[1],NULL,incrementa,NULL);
      pthread_create(&threads[2],NULL,espera,NULL);

      for(i=0; i<3; i++)
      {
        s=pthread_join(threads[i], (void **) &estado);
        if (s)
        {
            perror("Erro no join");
            exit(-1);
        }
        printf("O thread %d terminou com o estado %d\n",i,estado);
    }
```

```
	pthread_mutex_destroy(&mutex_contador);

	pthread_cond_destroy(&condicao_contador);

	pthread_exit(NULL);

}
```