UNIVERSIDADE
LUSÓFONA
DO PORTO

Threads

SISTEMAS OPERATIVOS

José Ferreira

Miguel Carvalho

Universidade Lusófona do Porto

Engenharia Informática – 2ºano

Porto, 9 de dezembro de 2022

Índice

1. Introdução

Este documento tem como propósito fazer um estudo sobre Threads e todos assuntos relacionados.

Iremos definir o que são Threads e para que são utilizados. Iremos também dizer que tipos existem, como se diferem dos processos e como são implementados.

Por fim iremos mostrar a implementação pratica de Threads num programa.

2. Processos

Para explicar o que são threads temos primeiro de explicar um dos conceitos centrais de qualquer sistema operativo, os processos.

O processo é uma abstração de uma instância de um programa a ser executado pelo processador.

Num sistema de multiprogramação, o processador troca de processo rapidamente, executando cada um durante dezenas ou centenas de milissegundos. Enquanto tecnicamente, em cada instante, o processador apenas está a executar um processo, no decorrer de um segundo pode trabalhar em vários, dando a ilusão de paralelismo de tarefas. Às vezes chamado de pseudoparalelismo em contraste ao paralelismo real que existe em sistemas com vários processadores.

Todos os computadores modernos fazem várias coisas ao mesmo tempo, apesar de os utilizadores não estarem sempre a par deste facto. Toda esta atividade tem de ser gerida e um sistema de multiprogramação que suporta múltiplos processos e threads é imprescindível.

Cada processo tem o seu próprio *program counter*, registos e variáveis.

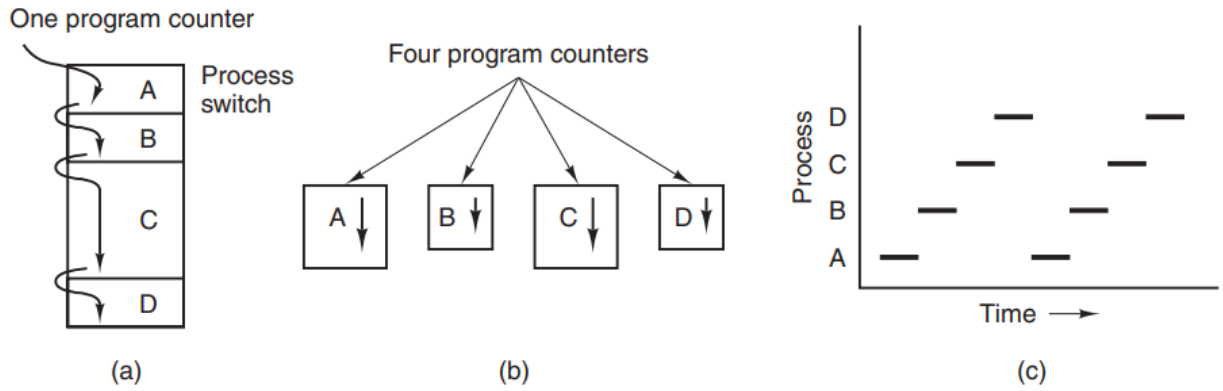

Figura 1- (a) Multiprogramação de 4 programas. (b) Modelo conceptual de 4 processos independentes. (c) Tempo de execução dos processos no processador.

3. Threads

3.1. O que são?

Threads são semelhantes a um processo leve que está a correr um programa, o que permite que o processador possa trocar de tarefas numa escala de tempo na ordem dos nanossegundos.

Por exemplo, se um programa precisa de ler dados da memoria (o que demora muito e deixa o processador parado), um processador *multithreaded* (com mais do que um thread) pode simplesmente trocar para outro thread (ou seja, trocar de tarefa) e continuar a trabalhar.

Multithreading não oferece paralelismo real. Apenas uma tarefa está a ser executada pelo processador, mas trocar de thread é mais rápido que trocar de processo.

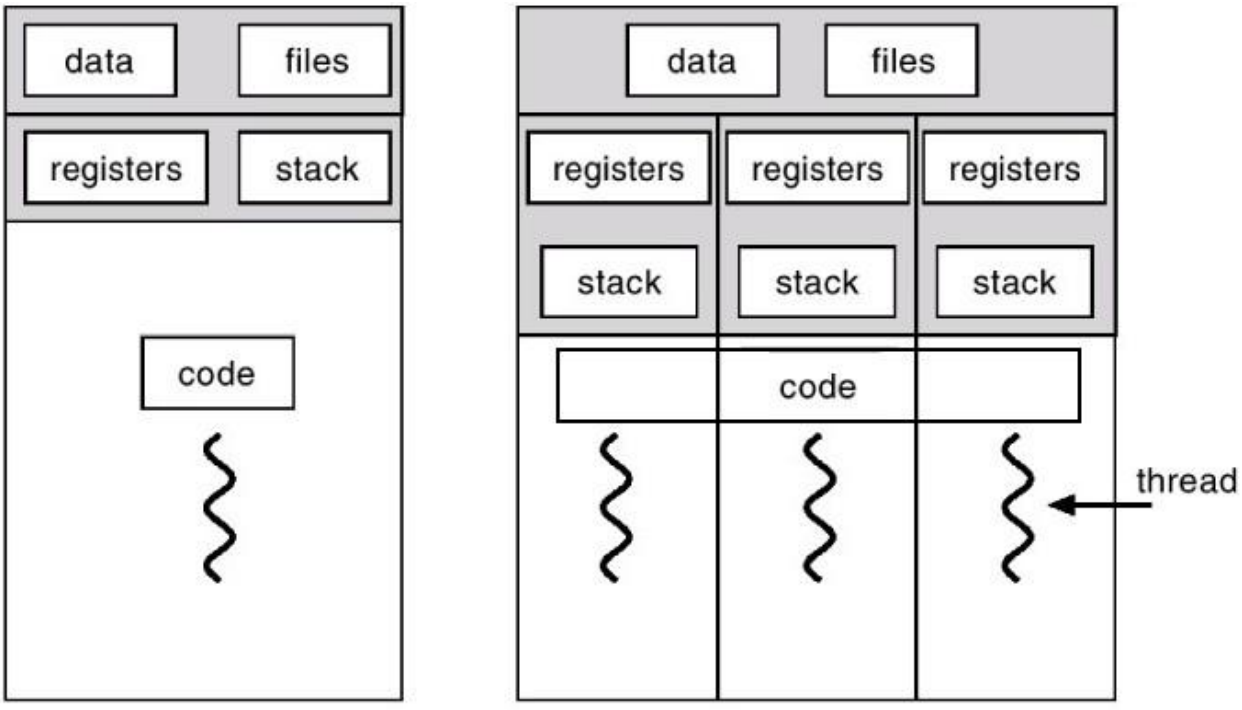

Figura 2-Processos vs threads

Enquanto os processos não partilham nada entre si, os threads partilham o mesmo contexto.

3.2. Porquê utilizar threads?

As principais razões são:

1 – O modelo de programação fica mais simples:

A razão principal para a existência de threads é que em algumas aplicações há múltiplas atividades a acontecer ao mesmo tempo. Decompor uma aplicação deste tipo em múltiplos threads sequenciais que correm em pseudoparalelo torna o modelo de programação mais simples. Em vez de ter com conta *interrupts*, timers e trocas de contexto podemos pensar em processos paralelos que tem a capacidade de partilhar espaço de endereçamento e dados

2 – Velocidade de criação e destruição:

Como os threads são mais leves que processos, também são mais rapidamente criados e destruídos. Em alguns sistemas criar um thread pode ser entre 10 a 100 vezes mais rápido que criar ou destruir um processo. Quando o número de threads necessários muda rápida e dinamicamente esta característica é muito útil.

3 – Performance:

Threads não oferecem mais performance no que toca a execução no processador, mas quando há várias trocas de dados por I/O, os threads permitem uma sobreposição de tarefas de maneira rápida e eficiente, dando a aparência de melhor performance.

4 – Chamadas de bloqueio:

Threads tornam possível reter a ideia de processos sequenciais que fazem chamadas de bloqueio (por exemplo, transmissão de dados de I/O) e manter o paralelismo. Chamadas de bloqueio tornam a programação mais simples e o paralelismo aumenta a performance.

3.3. Exemplo: Processador de texto

Para demonstrar a utilidade dos threads vamos usar o exemplo de um programa de processamento de texto.

Em qualquer momento, o programa precisa de manter as páginas formatas, estar disponível para receber dados vindos do teclado, detetar erros de ortografia, dar sugestões de correção, gravar o ficheiro no disco, entre várias outras tarefas.

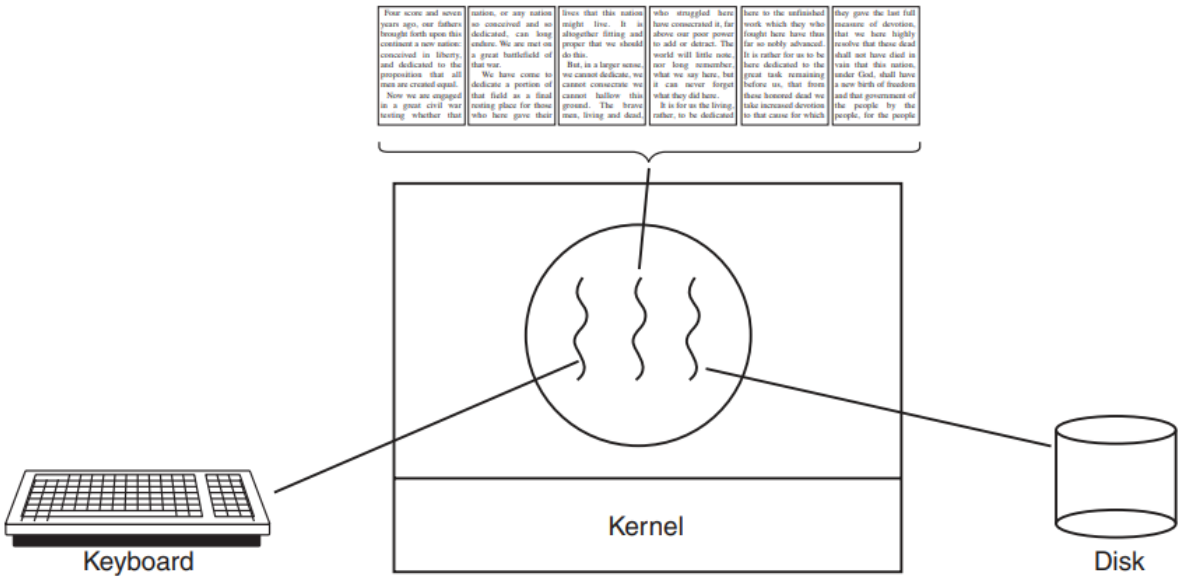

Figura 3- Um programa com 3 threads

Supondo que o programa foi escrito de modo a usar 3 threads. O primeiro thread apenas interage com o utilizador, recebendo os *imputs* vindos do teclado. O segundo mantem formatado as páginas do documento quando há alterações. O terceiro guarda o conteúdo da RAM periodicamente para não se correr o risco de perda de dados.

O uso de threads evita que, por exemplo, sempre que comece um backup de dados para o disco, o utilizador seja interrompido na escrita do documento ou que a formatação da página bloqueie.

Os três threads tem de operar no mesmo documento. Usando threads em vez de processos, eles partilham uma memoria comum e logo tem todos acesso ao documento que está a ser editado. Com processos isto seria impossível.

4. Modelos de Threads

4.1. Modelo clássico

A fundação do modelo de processo é baseada no conceito de agrupação de recursos e execução. Um processo pode ser visto como uma técnica para agregar materiais que estão relacionados. Um espaço de endereçamento de um processo contem vários recursos de um programa, ficheiros abertos, processos-filho, sinais entre outras coisas.

Um thread tem de fazer parte de um processo, mas são conceitos distintos e são tratados de maneira diferente. Threads são as entidades que são executadas no processador e os processos são utilizados para agrupar os recursos.

O paradigma dos processos é melhorado pelos threads uma vez que estes permitem que varias execuções aconteçam no ambiente do mesmo processo.

A operação em paralelo de vários threads num só processo é comparável com a operação de vários processos num computador. Enquanto os threads partilham espaço de endereçamento e outros recursos, os processos partilham memoria física, discos, impressoras, etc. A semelhança entre eles leva a que as vezes os threads sejam chamados de “processos leves”.

Alguns processadores suportam *multithreading* ao nível do hardware permitindo a troca de thread em nanossegundos.

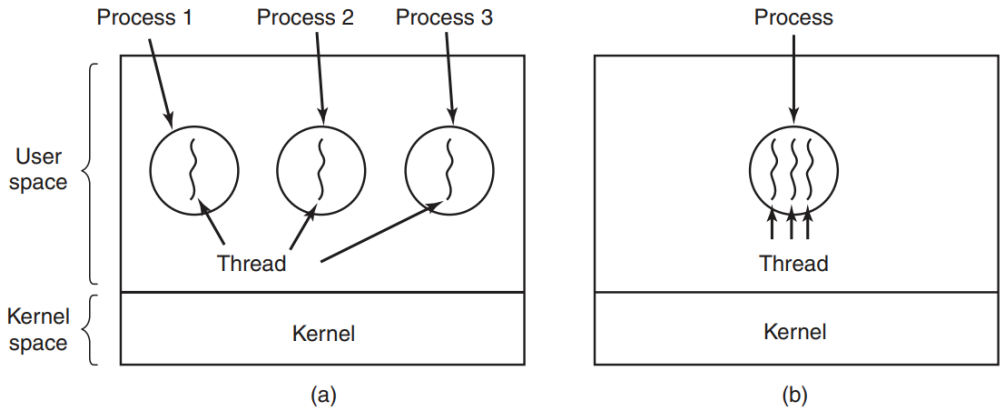

Figura 4- (a) 3 processos tradicionais com um thread cada (b) 1 processo com 3 threads

Diferentes threads num processo não são tão independentes como vários processos. Como cada thread pode aceder a todos os endereços de memoria dentro do espaço de endereçamento de um processo, um thread pode ler, escrever ou até mesmo apagar a pilha de outro thread. No entanto não existe proteção entre threads porque não é necessário e é impossível. Uma vez que ao contrário de processos diferentes, que podem ser de utilizadores diferentes, um só processo apenas pertence a um utilizador e, portanto, presume se que os vários threads desse processo não tenham objetivos diferentes e hostis entre eles.

Problemas ou complicações introduzidas por este modelo:

- Se um processo pode ter vários processos-filho estes também podem criar threads? E o que acontece quando são bloqueados? Quando recebem *imputs*?
- O que acontece se um thread resolver fechar um ficheiro que outro estava a usar?
- Se um thread precisar de mais memoria e enquanto estiver a alocar mais houver uma troca de thread e este repetir o pedido de mais memoria?

Estes problemas podem ser resolvidos, mas requerem esforço e bom design!

4.2. Threads POSIX

POSIX (*Portable Operating System Interface*) é uma família de standards especificados pela IEEE (instituto de engenheiros eletrotécnicos e eletrónicos) para manter compatibilidade entre sistemas operativos. Nela são especificados um pacote de threads chamado *Pthreads* que define mais de 60 chamadas de funções e é suportado por maior parte dos sistemas UNIX.

Todos os *Pthreads* têm certas propriedades. Cada um tem um identificador, um conjunto de registos, (incluindo o *program counter*) e um conjunto de atributos que estão guardados numa estrutura. Estes atributos incluem o tamanho da pilha, parâmetros de agendamento e outros itens necessários.

Aqui iremos apenas falar de algumas das muitas chamadas de funções que existem nos threads POSIX.

Para criar um thread novo é usada a chamada *pthread_create* e o identificador é retornado como o valor da função.

Quando um thread completou o trabalho que lhe tinha sido atribuído, pode ser terminado chamando *pthread_exit*. Esta chamada para o thread e liberta a sua pilha.

Muitas vezes um thread tem de esperar que outro acabe o seu trabalho e seja terminado antes de poder continuar. Para este efeito o thread deve chamar *pthread_join* sendo o identificador do thread que tem de esperar dado como parâmetro.

5. Local de implementação

Há dois locais principais onde implementar os threads: no espaço do utilizador e no *kernel*. A escolha é um bocado controversa e também é possível uma implementação híbrida. Iremos neste capítulo descrever estes métodos e as suas vantagens e desvantagens.

5.1. Implementação no espaço do utilizador

O primeiro método coloca o pacote de threads todo no espaço do utilizador. O *kernel* não sabe nada sobre eles. Do ponto de vista do *kernel*, apenas está a gerir processos de um só thread. A primeira grande vantagem é que este pacote de threads pode ser implementado num sistema operativo que não suporta threads. Nesta abordagem os threads são implementados através de uma biblioteca.

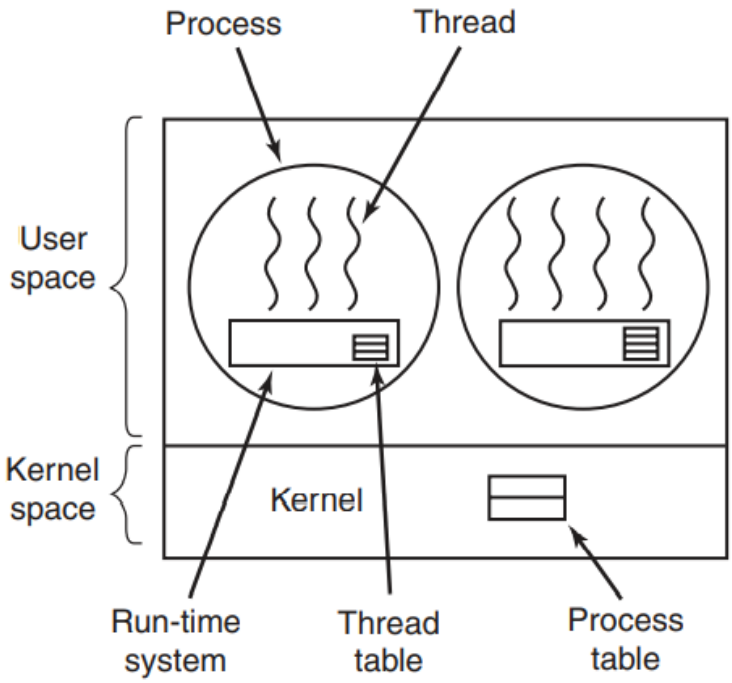

Figura 5-Pacote de threads no espaço do utilizador

Nesta implementação os threads correm em cima de um sistema *run-time*, que é uma coleção de procedimentos que gerem os threads. Alguns destes procedimentos foram descritos no capítulo anterior.

Quando os threads são geridos no espaço do utilizador, cada processo precisa da sua própria tabela que acompanha os threads desse processo. Esta tabela é semelhante a tabela de processos do *kernel*. Quando um thread é colocado no estado *ready* ou no estado bloqueado, a informação necessária para reativar o thread está guardado na tabela.

Todos estes procedimentos acontecem sem se ter de fazer uma chamada do *kernel*, o que torna este método de implementação muito eficiente. Outras vantagens são que permite um algoritmo de agendamento customizado e é facilmente escalável.

No entanto os problemas possíveis neste método são todos complicados de resolver:

- Threads que façam chamadas de bloqueio do sistema causam que os outros threads sejam afetados. Isto obriga a que a biblioteca de chamadas do sistema tenha de ser adaptada.
- Se um thread causar erro de paginação, o *kernel*, que não esta a par dos threads vai bloquear o processo todo até que o I/O tenha terminado.
- Não há *interrupts* de relógio num só processo, o que torna impossível fazer agendar a vez dos processos, os threads terão de libertar o processador de livre vontade.

5.2. Implementação no *Kernel*

No segundo método, o *kernel* está a par e gere os threads. Assim a tabela que segue todos os threads pertence ao *kernel* e para criar ou destruir um thread tem de se fazer uma chamada ao *kernel* que por si trata de destruir ou criar fazendo uma atualização da tabela.

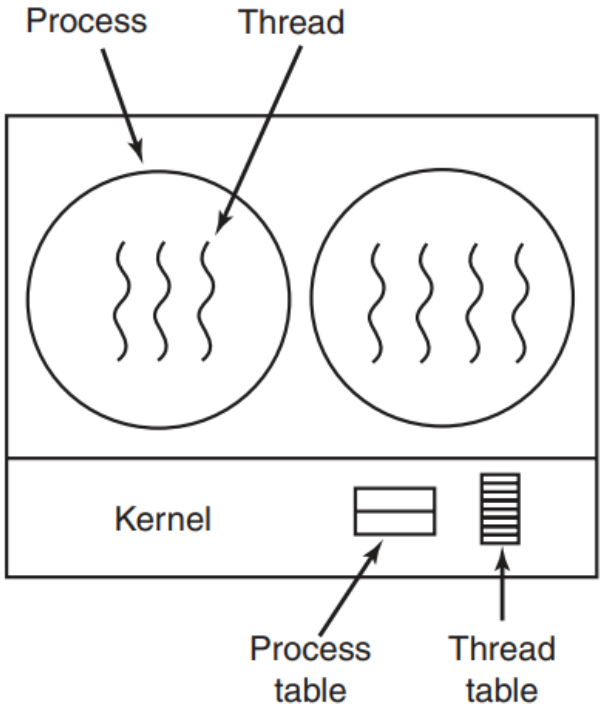

Figura 6-Implementação de threads no kernel

A grande desvantagem deste modelo é que estas chamadas ao *kernel* são muito menos eficientes, há um maior incentivo a diminuir o *overhead*. Para isso os threads em vez de serem destruídos são marcados como *not runnable* e as suas estruturas são mantidas intactas. Assim quando for necessário criar um thread novo apenas se reutiliza o antigo thread.

As desvantagens do modelo de implementação no espaço do utilizador não se aplicam aqui, não é preciso modificar as bibliotecas de chamadas do sistema nem são os processos completamente bloqueados quando há erros de paginação.

5.3. Implementação híbrida

Várias maneiras foram estudadas de tentar combinar as vantagens dos dois modelos apresentados anteriormente. Uma das maneiras é usar threads no *kernel* e depois multiplexá-los em threads do utilizador.

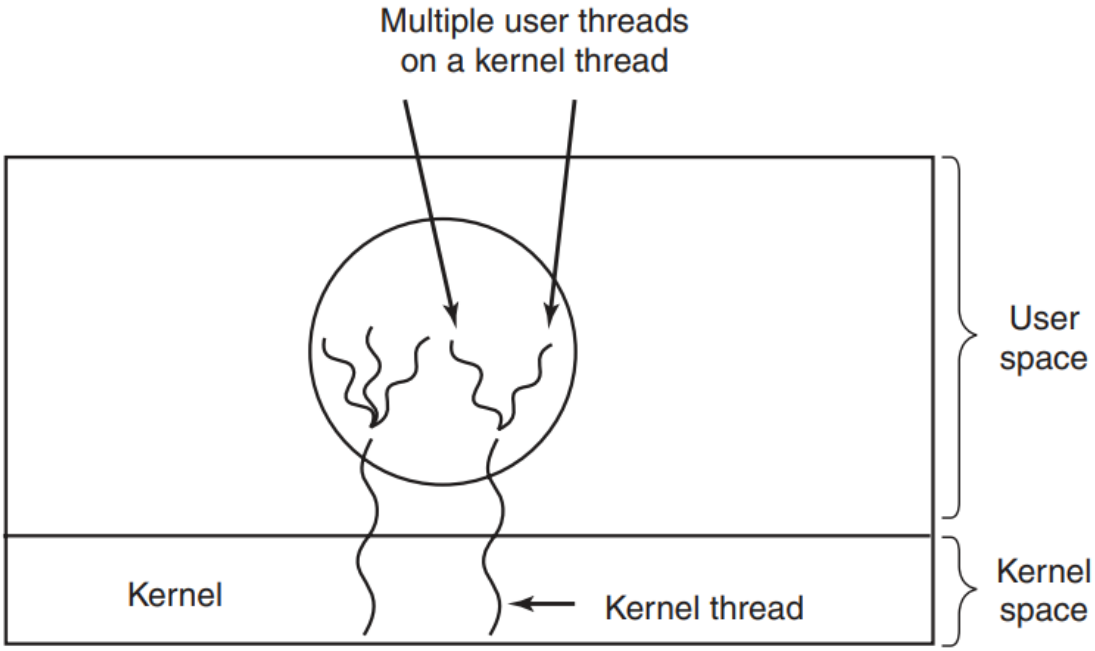

Figura 7- Multiplexação de threads do kernel em threads do utilizador

Com esta abordagem o *kernel* está apenas a par dos threads do seu espaço e trata do agendamento deles. Alguns desses threads podem ter vários threads do espaço do utilizador multiplexados em cima deles. Estes threads são criados, destruídos e agendados tal como no modelo anterior num processo que corre num sistema operativo que pode não ser compatível com *multithreading*. Neste modelo cada thread do *kernel* tem um conjunto de threads do utilizador.

6. Implementação pratica

Para demonstrar o uso de Threads criou se um programa de multiplicação de matrizes. Utilizando a biblioteca *pthread.h* foi possível dividir a tarefa de fazer as multiplicações de uma matriz quadrada “A” pelas colunas de uma matriz “B” com o mesmo tamanho.

A multiplicação de matrizes é uma tarefa complexa que aumentando o tamanho das matrizes rapidamente é expandido os cálculos necessários, o que a torna uma tarefa ideal para decompor em vários threads.

A linguagem de programação utilizada foi *C*, as matrizes são geradas aleatoriamente com números de 0 a 9 e o número de linhas é escolhido pelo utilizador. O número de threads pode ser modificado no código.

Divisão de tarefas pelos threads

O número de linhas a ser calculadas por cada thread é obtido pela divisão do número de linhas das matrizes pelo número de threads.

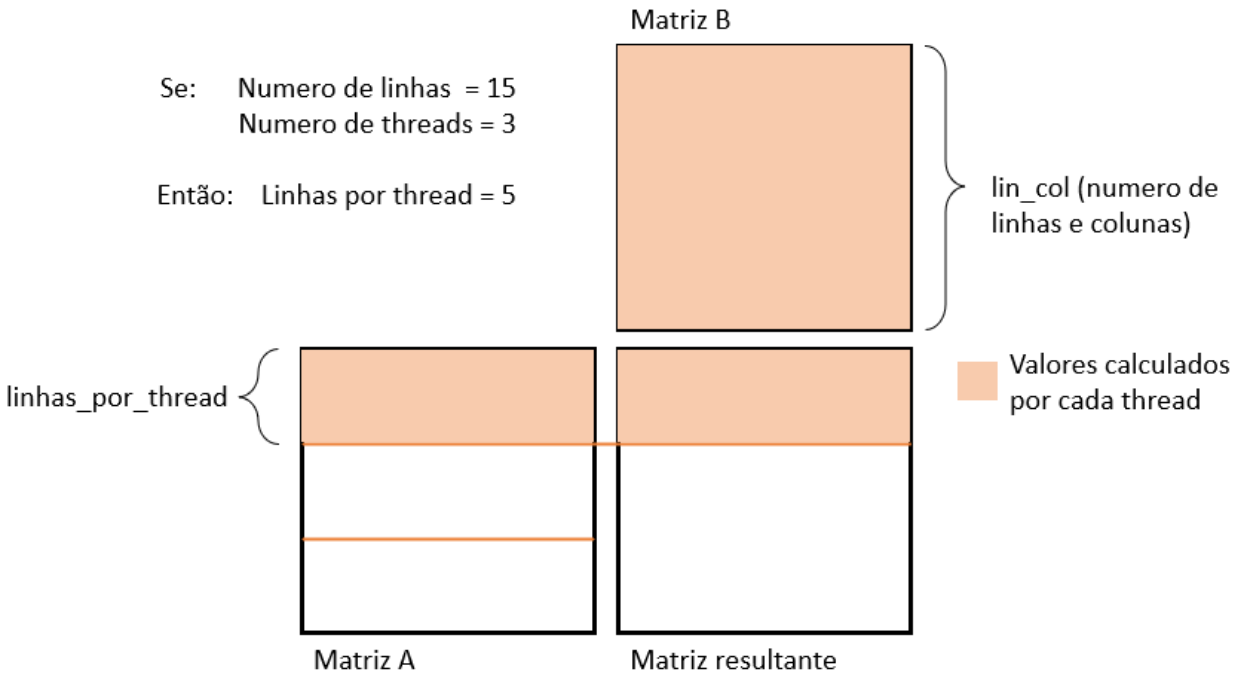

Figura 8 - Multiplicação de matrizes

Maior parte das variáveis são declaradas como variáveis globais, aproveitando a partilha de contexto entre todos os threads.

Durante o percurso da construção do código também foi testada a hipótese de criar um thread para cada linha da matriz, mas concluiu-se que seria um processo altamente ineficiente pois o tempo de criação de cada thread acaba por ser muito significativo em relação ao tempo de execução da tarefa que cada um iria fazer.

Criação dos threads:

O endereço dos threads são alocados na memoria na forma um vetor do tipo estrutura pthread_t e os threads são criados com a chamada de função pthread_create().

```
// criacao do array de threads
pthread_t *threads; // declaração de variavel do tipo pthread_t*
threads = (pthread_t *)malloc(maxt*sizeof(pthread_t)); // array do tamanho maxt

int inicio[maxt]; //criação de vetor que contem os numeros da linha onde começar a multiplicação

for (i = 0; i < maxt; i++) //um ciclo para cada thread
{
    inicio[i] = i * linhas_por_thread; // exemplo: se i = 0 multiplicação começa na linha zero

    // cria os threads e envia ponteiro do inicio
    status = pthread_create(&threads[i], NULL, mult, (void*)(inicio+i));
    if (status != 0)
    {
        printf("Oops. pthread_create returned error code %d\n", status); // em caso de criação correr mal
        exit(-1);
    }
}
```

Figura 9 - Criação dos threads

Multiplicação das matrizes:

Os threads executam a função *mult* e recebem como argumento um ponteiro com o número da linha que é a primeira a ser calculada.

```
void *mult(void *arg)
{
    int *inicio_ptr = (int *)arg; // transformar arg de void* de volta para int*
    int i = 0, j = 0, k = 0, operacao = 0;
    int final;

    final = linhas_por_thread + *inicio_ptr; //multiplicação deve terminar na linha correspondente
                                             //a (linhas_por_thread + inicio)

    for ( i = *inicio_ptr; i < final; i++) //começa na linha indicada por inicio
    {
        for ( j = 0; j < lin_col; j++)
        {
            for ( k = 0; k < lin_col; k++)
            {
                operacao += matA[i][k] * matB[k][j]; //multiplica os valores e soma á variavel operação
            }

            matres[i][j] = operacao; //insere o valor obtido na matriz resultado
            operacao = 0; //retorna a variavel a zero para começar de novo
        }
    }
}
```

Figura 10 - Função Mult

***Join* das threads:**

Por fim na função main utiliza-se a chamada de função pthread_join() para que o programa espere que todos os threads terminem a sua tarefa.

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

Figura 11 - pthread_join

Resultados obtidos:

Com a execução do programa podemos ver o efeito das threads no funcionamento do programa. Podemos concluir que a eficiência das threads depende do número de *cores* do CPU versus número de threads e verificar que tarefas são mais apropriadas para a utilização de threads e as suas vantagens e desvantagens.

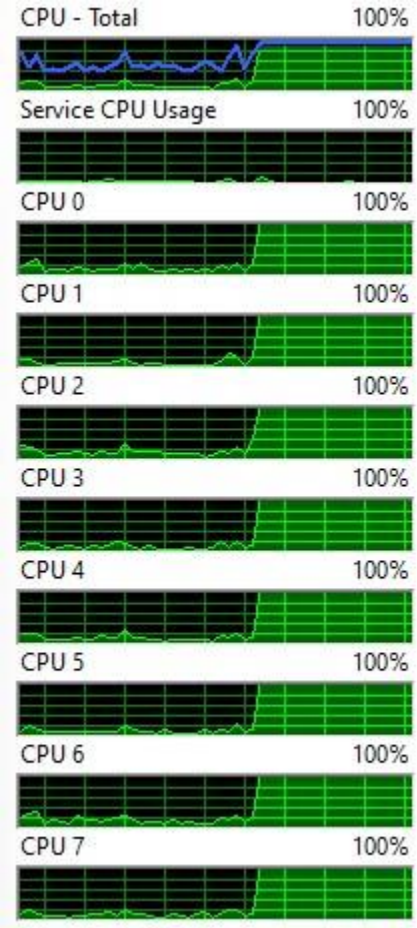

Figura 12 - Utilização do CPU com 10 threads

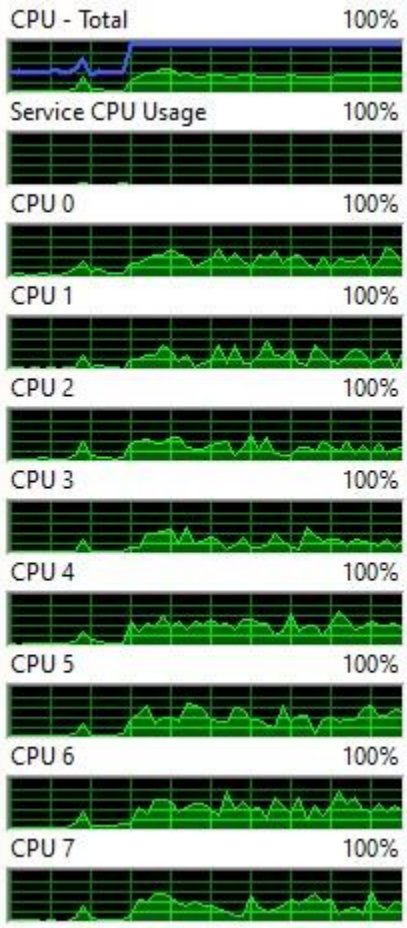

Figura 13 - Utilização do CPU com 2 threads

A figura 12 mostra o programa a funcionar no seu potencial máximo, utilizando todos os *cores* (8) do CPU quando no programa se criou 10 threads, um numero superior ao numero de cores. Se selecionar menos threads, neste caso 2, que o número de *cores* vemos no caso da figura 13 que o CPU não está a ser utilizado a 100%.

7. Conclusão Final

A uso de threads é algo indispensável num sistema operativo moderno, no entanto a sua implementação pode ser executada de várias maneiras cada uma com os seus desafios e vantagens. A criação deste relatório permitiu um pequeno deslumbre deste componente valioso para qualquer sistema.

8. Referencias bibliográficas

1. Modern Operating Systems – 4th edition – Andrew S. Tanenbaum, Herbert Bos