Felipe Cancella - 2210487
Diego Miranda - 2210996

EX1

Objetivo;

No EX1 o objetivo era criar um código para iterar sobre um vetor de inteiros de duas formas diferentes e compará-las.

A primeira maneira consistia em 10 threads iterando juntos sobre seções do vetor principal de 10000 posições. Dessa forma o vetor de 10000 seria dividido em uma parte de 1000 para cada um dos processos que rodariam ao mesmo tempo (PARALELAMENTE).
Após isso deveríamos ter a soma de todas essas seções.

(Um processo idêntico também deve ser feito para 100 trabalhadores)

Já a segunda maneira, porém, consistia em iterar sobre esse vetor de forma SEQUENCIAL, ou seja, apenas um processo seria responsável por fazer as operações em cada item do vetor.

Ao final das duas maneiras devemos comparar os resultados e o tempo de execução de cada uma.

Obs.: Em ambas as maneiras também fizemos o mesmo código para iterar sobre um vetor maior.:

ex1pg.c - (paralelo grande - 10 threads, vetor 100k)
ex1sg.c - (sequencial grande - vetor 100k)
ex1100g.c - (paralelo grande - 100 threads, vetor 100k)

Estrutura do programa;

ex1p.c: (paralelo - 10 threads, vetor 10k)

O Código inicia criando um vetor para armazenar os inteiros (10000 posições), e definindo o número de threads a serem usadas (10).
Logo depois as 10 threads são iniciadas, e com isso, para cada um dos processos, chamamos a função “soma” usando o código:

```
pthread_create(&threads[i], NULL, soma, (void *)i);
```

A função soma…EXPLICAR FUNCAO SOMA
Depois disso, o código espera cada thread finalizar com o código abaixo:

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

E por fim, mostra o tempo de processamento total (Inicia antes de criar as 10 threads, e termina após todas finalizarem), e também mostra o resultado da soma de todos os itens do vetor iterado.

ex1pg.c: (paralelo grande - 10 threads, vetor 100k)

O código funciona igual ao ex1p.c, porém faz a iteração paralela sobre um vetor de tamanho 100.000 e não 10.000

ex1s.c: (sequencial 10k)

O código inicia criando o vet (10.000 inteiros), que é o vetor que vai ser iterado/manipulado.
Depois disso ele itera sobre cada item desse vetor fazendo as operações (*2 e + i).
E por fim mostra o tempo de processamento e a soma final de cada item do vetor.

ex1sg.c: (sequencial grande - vetor 100k)

O código funciona igual ao ex1s.c, porém faz a iteração sequencial sobre um vetor de tamanho 100.000 e não 10.000

ex1100.c: (paralelo - 100 threads, vetor 10k)

Igual ao ex1p.c, só que com 100 threads (vetor de tamanho 10.000)

ex1100g.c: (paralelo grande - 100 threads, vetor 100k)

Igual ao ex1pg.c, só que com 100 threads (vetor de tamanho 100.000)

Solução;

Para realizar o objetivo, iniciamos o vetor com todas as casas = 5. Depois iniciamos cada thread chamando a função soma para cada uma.

```
pthread_create(&threads[i], NULL, soma, (void *)i);
```

Essa função é responsável por fazer o cálculo de cada item do vetor.:

```
vet[cont] = vet[cont]*2+cont;
```

Com isso, cada thread itera sobre sua parcela do vetor (0-999, 1000-1999, 2000-2999, etc). Para que todos os processos terminem de rodar antes de continuar o código, usamos um for que espera a chegada de todos eles.:

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

Por fim, calculamos o tempo levado desde a criação das 10 threads, até o fim desses 10 processos. Além disso calculamos também a soma de todos os itens do vetor.
E dessa forma obtemos a seguinte saída:

Saída ex1p.c:

Creating thread 0
Creating thread 1
Creating thread 2
Creating thread 3
Creating thread 4
Creating thread 5
Creating thread 6
Creating thread 7
Creating thread 8
Creating thread 9

Soma: 50095000

Tempo de exec: 2527 usec ou 0.002527 sec

O ex1100.c realiza exatamente os mesmo passos de ex1p.c (PARALELO) porém com um número maior de threads, e assim temos a seguinte saída:

Saída ex1100.c:
Creating thread 0
Creating thread 1
…
Creating thread 98
Creating thread 99

Soma: 50095000

Tempo de exec: 23184 usec ou 0.023184 sec

Já o ex1s.c (SEQUENCIAL) itera o vetor de forma sequencial, ou seja, apenas um processo é responsável por todo o vetor. Assim, ele inicia o vetor de forma simples, itera sobre ele, e printa o tempo final.

Saída ex1s.c:

Soma: 50095000

Tempo de exec: 3 usec ou 0.000003 sec

Para os casos em que aumentamos os vetores de tamanho para 100.000, o resultado da soma é sempre 705982704, e os tempos de execução ligeiramente maiores.

Observações e conclusões;

Apesar dos códigos terem sido montados corretos aparentemente, os resultados finais nos mostraram que o código sequencial é mais veloz que o paralelo. Nós dois ficamos surpresos com o resultado e chegamos a imaginar que tenha dado algum erro em alguma etapa, visto que faria mais sentido o processo rodar mais rapidamente quando dividimos o trabalho para mais processos. Pensando mais um pouco porém, por estarmos realizando um código de tao baixo custo (*2 + 1), também vemos sentido no código sequencial ser mais veloz.
Enfim, como a diferença do resultado foi discrepante, começamos a desconfiar um pouco do código, mas se considerarmos que ele está correto, a explicação mais correta realmente é a de que o sequencial roda mais rápido por ser um processo de baixo custo.

EX2

Objetivo;

O objetivo do com o enunciado é demonstrar na prática como funciona quando dois processos concorrem sobre a manipulação de espaços de memória, em que ao programas diferentes lerem o mesmo valor e ambos aplicarem operações sobre, o resultado foge do esperado.

Estrutura do programa;

Em primeiro lugar, criamos o vetor de 10000 elementos, e depois preenchemos todas as casas com o valor 5. Separamos em 2 threads, que tem como objetivo fazer operações sobre os elementos do vetor. Verificamos ao final se algum valor é diferente dos outros do vetor.

Solução;

A solução que encontramos se assemelha bastante ao exercício 1, em alterando a quantidade de threads a serem criadas para dois, e todos os seus processos relativos. Além disso, também modificamos como será manipulado o vetor, removendo a setorização para cada thread, não adicionando a posição atual, mas multiplicando por 2 e somando 2 também.

Saída do programa:

Creating thread 0
Creating thread 1

Houve concorrencia!

Tempo de exec: 1017 usec ou 0.001017 sec

Observações e conclusões;

Percebemos como na prática a concorrência entre processos é capaz de afetar um sistema, uma vez que ao ler um valor ao mesmo tempo, os diferentes processos alteram o mesmo valor original, que pode não ser o desejado. Sofremos bastante para compreender onde iríamos encontrar as evidências de concorrência, além de ao rodar a mesma aplicação em diferentes ambientes como Replit e Ubuntu com WSL, não fomos capazes de ver resultados, apenas nos computadores de sala com sistema Linux, e precisando aumentar o tamanho do vetor em alguns casos.