INF1316 - Laboratório 05 - Tarefas (*threads*) em sistemas Unix

Nome:

- Guilherme Riechert Senko
- Pedro de Almeida Barizon

Matrícula:

- 2011478
- 2211350

Professor: Luiz Fernando Seibel

Turma: 3WA

Data: 01/10/2024

Objetivo: implementar e compreender tarefas (*threads*) de processos em sistemas Unix.

Observações preliminares

- Para mais detalhes, todos os códigos fonte comentados encontram-se em anexo. Por isso, para evitar redundâncias, não serão aqui exibidos.
- Todos os arquivos fonte foram compilados com auxílio do *GNU C Compiler* (GCC) fazendo `gcc -Wall -o programa fonte1.c fonte2.c … fonten.c` e executados com `./programa arg1 arg2 … argn`. Caso haja particularidades, estas serão abordadas ao longo do relatório.

Exercício 1) Criar um vetor a de 10.000 posições inicializado com valor 5. Criar 10 processos trabalhadores que utilizam áreas diferentes do vetor para multiplicar a sua parcela do vetor por 2 e somar as posições do vetor retornando o resultado para um processo coordenador que irá apresentar a soma de todas as parcelas recebidas dos trabalhadores. Obs: O 1º trabalhador irá atuar nas primeiras 1.000 posições, o 2º trabalhador nas 1.000 posições seguintes e assim sucessivamente. Repita o código do Ex1 usando agora com 100 trabalhadores e com um vetor de 100.000 posições.
Indique o que ocorreu

1.1 Estrutura

Criou-se apenas um módulo, `ex1.c`, que continha as funções:

- `main`: responsável pela execução do programa.
- `trabalhador`: rotina a ser executada pelas tarefas do processo. Multiplica uma porção das entradas do vetor global (`vetor`) por dois e armazena seu somatório no vetor global de somas parciais (`soma_parcial`).

1.2 Solução

Inicialmente, criam-se dois vetores globais de inteiros: `vetor`, com 10.000 entradas; e `soma_parcial`, que guardará em cada posição ***i*** a soma respectiva à ***i***-ésima tarefa criada. A seguir, já dentro da rotina `main`, inicializam-se os elementos de `vetor` com 5, e define-se o vetor `thread` de tipo `pthread_t`, para armazenar os identificadores das *threads*.

Logo após, geram-se 10 tarefas com `pthread_create`, que começam imediata e concorrentemente a executar a rotina `trabalhadora`. Espera-se cada tarefa terminar com `pthread_join`, passando-se seu identificador como argumento. Cada uma finaliza por meio da chamada de `pthread_exit`. Finalmente, agregam-se as somas parciais sob a forma de uma soma total, cujo valor é exibido na console.

Em seguida, repetimos o mesmo programa, mas desta vez com um vetor de 100.000 posições e 100 trabalhadoras. As saídas encontram-se abaixo:

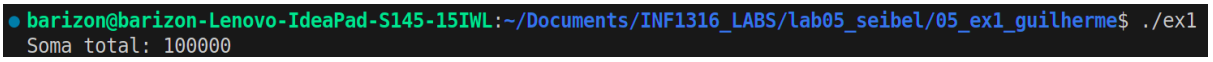

Figura 1.1 - Saída de `ex1` com 10 tarefas e 10.000 posições

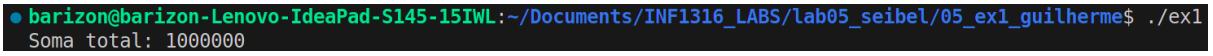

Figura 1.2 - Saída de `ex1` com 100 tarefas e 100.000 posições

1.3 Observações e conclusões

1.3.1 Cenário com 10 threads

O uso de 10 *threads* para manipular o vetor de 10.000 posições distribuiu a carga de forma eficiente. Como cada *thread* manipula exatamente 1.000 posições, o processamento foi dividido de forma equitativa, resultando em uma execução rápida e com baixo *overhead* (excesso de alocação de recursos) de gerenciamento de *threads*. O resultado obtido foi conforme o esperado: 100.000.

1.3.2 Cenário com 100 threads

O uso de 100 *threads* com o vetor de 100.000 posições também resultou em uma divisão equitativa de 1.000 posições por *thread*. No entanto, o *overhead* de criação e sincronização de um número maior de *threads* é maior do que no primeiro caso. Dependendo das especificações do *hardware*, pode haver um impacto no tempo de execução devido à sobrecarga de *threads* e à necessidade de sincronização entre elas. Em um Lenovo IdeaPad S145-15IWL, com processador Intel Core i7, não houve um aumento nítido do tempo de execução — tal acréscimo não foi medido de forma quantitativa; apenas qualitativa. Seja como for, o resultado obtido também foi conforme o esperado: 1.000.000.

1.3.3 Conclusão geral

Ambos os cenários resultaram nos valores esperados para a soma final do vetor. A implementação com 10 *threads* foi eficiente, com menor *overhead* e rápida execução. No cenário com 100 threads, o aumento do número de trabalhadores introduziu um maior *overhead* de gerenciamento, mas a tarefa foi igualmente dividida, e o resultado final foi alcançado conforme esperado.

Exercício 2) Considere o vetor de 10.000 posições inicializado com o valor 5. Crie 2 trabalhadores, ambos multiplicam por 2 e somam 2 em todas as posições do vetor. Verifique se todas as somas têm valores iguais e explique o que ocorreu. Repita o código do Ex2 usando agora 10 ou 100 trabalhadores e um vetor de 100.000 posições.
Indique o que ocorreu.

2.1 Estrutura

Criou-se apenas um módulo, `ex2.c`, que continha as funções:

- `main`: responsável pela execução do programa.
- `atualizaVetor`: rotina a ser executada pelas tarefas do processo. Multiplica cada entrada de `vetor` por dois e, logo após, a incrementa em dois (antes de atualizar a próxima).

2.2 Solução

Inicialmente, cria-se um vetor global com 10.000 entradas `unsigned long long`, denominado `vetor`. A seguir, já dentro da rotina `main`, define-se o vetor `thread` de tipo `pthread_t`, para armazenar os identificadores das *threads*. Por fim, inicializam-se os elementos de `vetor` com 5.

Logo após, geram-se duas tarefas com `pthread_create`, que começam imediata e concorrentemente a executar a rotina `atualizaVetor`. Espera-se cada tarefa terminar com `pthread_join`, passando-se seu identificador como argumento. Cada uma finaliza por meio da chamada de `pthread_exit`.

Finalmente, à semelhança do procedimento feito no LAB01, toma-se o primeiro elemento de `vetor` como referência, e percorre-se o restante em busca de divergentes. As primeiras dez posições em que ocorrer divergência juntamente com seus respectivos valores são exibidas. Ao final, mostra-se o total de divergentes. A saída encontra-se abaixo:

```
barizon@barizon-Lenovo-IdeaPad-S145-15IWL:~/Documents/INF1316_LABS/lab05_seibel/05_ex2$ ./ex2
>>> REFERENCIA: 26

*** Todos os valores sao iguais a referencia
```

Figura 2.1 - Saída de `ex2`

```
barizon@barizon-Lenovo-IdeaPad-S145-15IWL:~/Documents/INF1316_LABS/lab05_seibel/05_ex2$ ./ex2
>>> REFERENCIA: 18446744073709551614
10210 - Difere da referencia o valor 18446744073709551582
10858 - Difere da referencia o valor 18446744073709551582
10869 - Difere da referencia o valor 18446744073709551582
14811 - Difere da referencia o valor 18446744073709551582
18443 - Difere da referencia o valor 18446744073692774398
18445 - Difere da referencia o valor 8388606
18458 - Difere da referencia o valor 8388606
18459 - Difere da referencia o valor 18446744073692774398
18959 - Difere da referencia o valor 18446744073692774398
18970 - Difere da referencia o valor 18446744073692774398

!!! Nao serao exibidos mais valores para evitar saturacao do log

*** Total de diferentes: 365
```

Figura 2.2 - Saída de `ex2` modificado com 100 *threads* e 100.000 posições em `vetor`

2.3 Observações e conclusões

À semelhança do LAB01, para o pequeno número de duas tarefas, a condição de corrida não descoordenou a atualização do vetor, de modo que não houve divergentes. Em um segundo momento, porém, aumentando-se para 100 tarefas e para um vetor de 100.000 posições, começaram a surgir valores diferentes entre si. Isso se explica de modo completamente análogo ao caso que envolvia processos: sendo o sistema operacional preemptivo, pode interromper uma tarefa antes de que seja concluída. Logo, graças à condição de corrida em que se encontravam as tarefas, o escalonador deve ter interrompido uma tarefa depois de ter multiplicado por dois uma posição do vetor, mas antes de que a pudesse ter incrementado em dois. Quando a próxima trabalhadora atuou na posição, multiplicando-a por dois, gerou-se, então, uma divergência em relação aos elementos que foram antes incrementados. Isto é, de modo geral,

$$(((2a) + 2) \cdot 2) + 2 \neq (((2a) \cdot 2) + 2) + 2$$

Por fim, com base na comparação das duas situações, concluímos que a preempção tem maior probabilidade de produzir descoordenações à medida que se aumentem o número de tarefas ou seu tempo de duração (mais posições a serem atualizadas).

3 Conclusões finais

Os programas foram compilados sem erro usando-se o GCC, o ambiente de desenvolvimento do Visual Studio Code e o sistema operacional Ubuntu. Ademais, conforme se observa na saída do programa, todos os testes obtiveram êxito. Por fim, devemos reconhecer a importância deste laboratório, uma vez que nos permitiu vislumbrar a eficiência das tarefas em sistemas Unix.