Instituto Tecnológico de Aeronáutica

CES-33: Sistemas Operacionais

Lab 3

Professor:
Cecília A C César

Grupo:
Juan F D Galvão
Victor R Sales

13 de Abril de 2019

Conteúdo

1 Implementação

1.1 Problema proposto

Para o experimento com threads, propôs-se a resolução de um problema simples, em que se inverte a ordem dos elementos em um array de n elementos, distribuindo entre threads executando, em loops, a tarefa de trocar, dois a dois, os elementos do array.

Para isso, as threads utilizariam um elemento iterador i em comum, guardado em uma variável global, determinando assim qual o elemento do array a ser utilizado pela thread na sua iteração atual. Cada thread, então, trocaria o i-ésimo elemento do array pelo $(n-i)$-ésimo elemento. Desse modo, só é necessário que a thread itere enquanto o iterador i recebido for menor que ou igual a $\lfloor \frac{n}{2} \rfloor$.

Perceba que há uma condição de corrida no uso do iterador: as threads concorrentes devem ler o valor do iterador atual e, em seguida, incrementá-lo para que outra thread obtenha acesso ao próximo elemento do array. Desse modo, é preciso utilizar um mutex para operar sobre o iterador.

Ademais, o problema proposto é completamente CPU-bound e portanto não é possível ter certeza de que os resultados obtidos ao paralelizar a execução entre as threads será mais eficiente que o seu equivalente sequencial, pois há fatores como o overhead de trocas entre threads e bloqueio das mesmas ao tentar acessar o iterador. Desse modo, é preciso analisar os resultados dos testes apresentados na seção 2.

Por fim, foi testada outra versão da solução por threads, substituindo as operações em semáforos por operações sobre um iterador atômico, conforme visto na seção 2.3.

1.2 Implementação em código

Código que resolve o problema utilizando threads e lock:

```
1 #define MAX_THREADS 100
2 #define MAX_ARRAY 1000000
3 #define true  1
4 #define false 0
5 #define WINAPI
6 #define debugtxt(FORMAT) printf(" TID %d: " #FORMAT "\n",(int) pthread_self())
7 #define debug(FORMAT, ARGS...) \
8     printf("TID %d: " #FORMAT "\n",(int) pthread_self(),ARGS)
```

```
9
10  #include <time.h>
11  #include <stdio.h>
12  #include <pthread.h>
13  #include <semaphore.h>
14
15  // Funcoes e variaveis do problema
16  int n, p;
17  int iter;
18  int array[MAX_ARRAY];
19
20  // Funcoes e variaveis das threads
21  pthread_cond_t handleThread[MAX_THREADS];
22  pthread_t threadId[MAX_THREADS];
23  sem_t iterMutex;
24  sem_t factorsMutex;
25
26  // Truque para sabermos qual o semaforo foi chamado e poder imprimi-lo
27  #define up(SEM) _up(SEM,#SEM)
28  #define down(SEM) _down(SEM,#SEM)
29
30  void _up(sem_t *sem, const char * name) {
31          debug("Up %s ...",name);
32          sem_post(sem);
33          debug("Up %s complete!",name);
34  }
35  void _down(sem_t *sem, const char * name) {
36          debug("Down %s ...",name);
37          sem_wait(sem);
38          debug("Down %s complete!",name);
39  }
40
41  void* WINAPI threadFunc(void * lpparam){
42          int i, aux;
43          while (true) {
44                  // Condição de corrida das threads no iterador do array
45                  down(&iterMutex);
46                  i = iter++;
47                  // printf("i = %d\n", i);
48                  up(&iterMutex);
```

```
49
50                  // Precisamos percorrer o array apenas até sua metade
51                  if (i > n/2) break;
52
53                  aux = array[i];
54                  array[i] = array[n-i-1];
55                  array[n-i-1] = aux;
56          }
57  }
58
59  int main() {
60          iter = 0;
61          int sum = 0;
62
63          sem_init(&iterMutex, 0, 1);
64          sem_init(&factorsMutex, 0, 1);
65
66          FILE* entry = fopen("entry.txt", "r");
67
68          printf("Digite N e o número de threads:\n");
69          scanf("%d %d", &n, &p);
70
71          printf("Array inicial:\n");
72          for (int i=0; i<n; i++) {
73                  fscanf(entry, "%d ", &array[i]);
74                  printf("%d\n", array[i]);
75          }
76          printf("\n");
77
78          fclose(entry);
79
80          clock_t start = clock();
81          for (int i=0; i<p; i++){
82                  pthread_create (&threadId[i],
83                                  NULL,
84                                  threadFunc,
85                                  NULL);
86          }
87
88          for(int i=0; i<p; i++) {
```

```
89                  pthread_join (threadId[i], NULL);
90          }
91          clock_t end = clock();
92
93          printf("\nArray trocado:\n");
94          for (int i=0; i<n; i++)
95                  printf("%d\n", array[i]);
96          printf("\n");
97
98          printf("Tempo: %g ms\n", 1000*(double)(end-start)/CLOCKS_PER_SEC);
99          return 0;
100 }
```

Para fazer o código sem locks, foram removidas as chamadas aos semáforos. A única alteração feita para o código usando atômicos foi a inclusão da `stdatomic.h` e a alteração do `int iter` para `atomic_int iter`.

O código sequencial feito foi o seguinte:

```
1  #include <time.h>
2  #include <stdio.h>
3
4  int main() {
5    const int MAXN = 1000000;
6    const int MAXT = 100;
7    int array[MAXN];
8    int n;
9    int p;
10
11   FILE* entry = fopen("entry.txt", "r");
12
13   printf("Digite N e o número de threads:\n");
14   scanf("%d %d", &n, &p);
15
16   printf("Array inicial:\n");
17   for (int i=0; i<n; i++) {
18     fscanf(entry, "%d ", &array[i]);
19     printf("%d\n", array[i]);
20   }
21   printf("\n");
22
```

```
  fclose(entry);

  clock_t start = clock();
  for (int i=0;i<=n/2;++i) {
    array[i] ^= array[n-i-1] ^= array[i] ^= array[n-i-1];
  }
  clock_t end = clock();

  printf("Array final:\n");
  for (int i=0; i<n; i++) {
    printf("%d\n", array[i]);
  }
  printf("\n");

  printf("Tempo: %g ms\n", 1000*(double)(end-start)/CLOCKS_PER_SEC);

  return 0;
}
```

Para criar o arquivo de entrada, o seguinte script em python foi utlizado:

```
fout = open('entry.txt', 'w')
for i in range(1000000):
    fout.write(f'{i}\n')
```

2 Testes e Análise

Para testar os códigos criados, deixamos como entrada um array cujos valores iam de 0 a 999999, com N sendo 1000000 e P sendo 4. Colocar um valor alto para N foi necessário para verificar as disputas no programa.

2.1 Disputa no código com mutexes

Para analisar se houve disputas na execução do programa, temos que verificar se em algum momento a thread foi posta a dormir sem acordar logo em seguida. Podemos verificar isso nas seguintes linhas do arquivo `threads.out`:

```
1000271 TID -399644928: ”Down &iterMutex ...”
1000272 TID -382859520: "Down &iterMutex complete!"
1000273 TID -382859520: "Up &iterMutex ..."
1000274 TID -382859520: "Up &iterMutex complete!"
1000275 TID -382859520: "Down &iterMutex ..."
1000276 TID -382859520: "Down &iterMutex complete!"
1000277 TID -382859520: "Up &iterMutex ..."
1000278 TID -391252224: "Down &iterMutex complete!"
1000279 TID -391252224: "Up &iterMutex ..."
1000280 TID -382859520: "Up &iterMutex complete!"
1000281 TID -382859520: "Down &iterMutex ..."
1000282 TID -382859520: "Down &iterMutex complete!"
1000283 TID -382859520: "Up &iterMutex ..."
1000284 TID -391252224: "Up &iterMutex complete!"
1000285 TID -408037632: "Down &iterMutex complete!"
1000286 TID -391252224: "Down &iterMutex ..."
1000287 TID -408037632: "Up &iterMutex ..."
1000288 TID -382859520: "Up &iterMutex complete!"
1000289 TID -382859520: "Down &iterMutex ..."
1000290 TID -382859520: "Down &iterMutex complete!"
1000291 TID -382859520: "Up &iterMutex ..."
1000292 TID -399644928: ”Down &iterMutex complete!”
1000293 TID -382859520: "Up &iterMutex complete!"
1000294 TID -399644928: ”Up &iterMutex ...”
1000295 TID -382859520: "Down &iterMutex ..."
1000296 TID -408037632: "Up &iterMutex complete!"
1000297 TID -399644928: ”Up &iterMutex complete!”
```

O tempo para execução do código, após excluir as linhas de debug, é de aproximadamente 396 ms (arquivo `threads_out_no_debug.out`).

2.2 Disputa no código sem mutexes

Para analisarmos as disputas aqui basta ver os números que saíram fora de ordem. Podemos encontrálos no arquivo `threads_wrong.out`:

```
1999970 35
1999971 34
1999972 33
1999973 32
1999974 31
1999975 30
1999976 29
1999977 999971
1999978 27
1999979 26
1999980 25
1999981 24
1999982 23
1999983 22
1999984 21
1999985 20
1999986 19
1999987 18
1999988 17
1999989 16
1999990 15
1999991 14
1999992 13
1999993 12
1999994 11
1999995 10
1999996 9
1999997 8
1999998 7
1999999 6
2000000 5
2000001 4
```

```
2000002 3
2000003 2
2000004 1
2000005 0
```

Esse é apenas um exemplo de onde houve disputa e a thread errada ganhou. Outros exemplos podem ser achados no arquivo.

O tempo para execução do código sem locks é cerca de 139 ms.

2.3 Código utilizando atômicos

Observe que a condição de corrida no problema proposto se dá simplesmente pelo incremento do iterador. Desse modo, o uso de semáforos em C para tal tarefa resulta em um overhead desnecessário, pois precisamos apenas atomizar as operações sobre uma variável. Para essa finalidade, poderíamos declarar o iterador como uma variável do tipo `atomic_int`, presente na biblioteca `stdatomic.h`.

Realizando tal substituição no código, eliminando assim o uso de semáforos, o tempo de execução é de cerca de 63 ms, vide `threads_atomic.out`

2.4 Código sequencial

O código sequencial gera a saída correta, como pode ser checado no arquivo de saída correspondente `sequential.out`, em um tempo de apenas 3 ms.

3 Limitações

Como limitações, podemos incluir que não faz muito sentido utilizar um número de threads maior que o disponível pelo hardware, principalmente porque este programa é 100% CPU-bound.

4 Conclusão

Embora sejam uma ótima ideia, threads adicionam um overhead significativo no programa. Como podemos ver, apenas a inclusão das threads aumentou o tempo de execução de 3 ms para 139 ms – o que envolve apenas mudança de contexto e contagem de relógio individual – e, quando adicionamos os mecanismos de lock, o tempo aumenta mais ainda, de 139 ms para 396 ms.

Claramente, para o nosso caso, não compensou a utilização de threads em comparação com o programa sequencial. Um programa que seja mais IO-bound ou que utilize outros recursos, como uma GPU, pode se aproveitar melhor deste recurso.