OS HW#2

109370210 工管四乙 黃鈺凱

Q 4.27

The main function creates the new thread using pthread_create and waits for it to finish using pthread_join. Once the child thread finishes, the main thread outputs the generated sequence.

Note that the fib_sequence and fib_count variables are shared by both threads and therefore must be protected from race conditions. However, this program does not use mutexes or other synchronization primitives to ensure thread safety, so it may produce incorrect results when run with multiple threads. To make this program thread-safe, the fib_sequence and fib_count variables should be protected using mutexes or other synchronization primitives.

To compile the program, command:

```
gcc -pthread -o Fibonacci Fibonacci.c
```

This command will compile the Fibonacci.c source file and link it with the Pthread library (-pthread option) to produce an executable named Fibonacci.c

After compilation, you can run the program with a command-line argument specifying the number of Fibonacci numbers to generate, like this:

```
huangyukai@huangyukaideMacBook-Pro programming % gcc Fibonacci.c -o Fiboacci
huangyukai@huangyukaideMacBook-Pro programming % ./Fiboacci
Usage: ./fibonacci <number of fibonacci numbers>
huangyukai@huangyukaideMacBook-Pro programming % ./Fiboacci 10
Fibonacci sequence: 0 1 1 2 3 5 8 13 21 34
huangyukai@huangyukaideMacBook-Pro programming % ./Fiboacci 20
Fibonacci sequence: 0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181
huangyukai@huangyukaideMacBook-Pro programming %
```

4.24 (Optional)

Version 1

1. Program Notes:

This program will generate a series of random points based on user input. These points will be (x, y) coordinates that fall in the Cartesian coordinates in a square which has a circle inscribed in it. Based on how many points lands in the circle it generates pi.

2. To start the program:

To run the assignment, go to the terminal and find the directory that contains the compiled version of the file. To run the program, type:

./MonteCarlo.o

Then enter the desired number of points when prompted.

```
huangyukai@huangyukaideMacBook-Pro 4.24(Optional) % gcc MonteCarlo.c -o MonteCarlo
huangyukai@huangyukaideMacBook-Pro 4.24(Optional) % ./MonteCarlo

MultiThreaded Monte Carlo
Please enter a positive number for the amount of points you would like to generate: 100

The approximate value of pi for the desired amount of points (100) is: 3.160000

huangyukai@huangyukaideMacBook-Pro 4.24(Optional) % ./MonteCarlo

MultiThreaded Monte Carlo
Please enter a positive number for the amount of points you would like to generate: 10000

The approximate value of pi for the desired amount of points (10000) is: 3.164400

huangyukai@huangyukaideMacBook-Pro 4.24(Optional) % ./MonteCarlo

MultiThreaded Monte Carlo
Please enter a positive number for the amount of points you would like to generate: 1000000

The approximate value of pi for the desired amount of points (1000000) is: 3.142168
```

Version 2

In this implementation, I use four threads to generate a total of 10 million random points, if you want to change the random points just modify the (NUM_POINTS), with each thread generating 2.5 million points. The count variable is protected by a mutex to prevent data races and ensure thread safety.

When a thread finishes, it acquires the lock on the mutex, updates the global count variable with its local count, and then releases the lock. After all threads have finished, the main thread calculates the estimated value of pi using the formula, and outputs the result to the console.

Note that the random number generator is initialized with a seed that is unique to each thread, based on the current time and the thread ID. This ensures that each thread generates a different sequence of random numbers. To create a compile file, command:

gcc monte.c -o monte.o -lm

And then, type:

./monte.o

```
1 warning generated.
huangyukai@huangyukaideMacBook-Pro 4.24(Optional) % ./monte
Estimated value of pi: 3.128387
huangyukai@huangyukaideMacBook-Pro 4.24(Optional) % ./monte
Estimated value of pi: 3.129144
huangyukai@huangyukaideMacBook-Pro 4.24(Optional) % ./monte
Estimated value of pi: 3.126545
huangyukai@huangyukaideMacBook-Pro 4.24(Optional) %
```

6.33

(a) Data involved in the race condition:

The shared resource available_resources is involved in the race condition. Multiple processes can access and modify this resource concurrently, leading to inconsistencies if proper synchronization mechanisms are not in place.

(b) Location of the race condition in the code:

The race condition occurs in the decrease_count() function where available_resources is decremented without any synchronization mechanism. If multiple processes try to decrement available_resources simultaneously, it can lead to incorrect results or data corruption.

(c) Fix the code:

The code was been fixed which show in ./6.33/count.c.

- A mutex lock resource_lock is initialized and used to protect access to available_resources.
- The pthread_mutex_lock() function is used to acquire the lock before accessing available_resources, ensuring that only one thread can modify it at a time.
- The pthread_mutex_unlock() function is used to release the lock after the critical section is executed.

Programming Projects
CH4

The merge sort algorithm will have three parameters: a starting pointer, a length, and a thread-depth. For our purposes the thread depth will be N in a situation where we are using at-most 2N-1 threads. (More on that later, but trust me, it makes it easier to do the math this way).

- If the thread depth has reached zero OR the sequence length is below a minimum threshold *we set), do not setup and run a new thread. Just recurse into our function again.
- Otherwise, split the partition. Setup a structure that holds a partition definition (which for us will be a starting point and a length as well as the thread depth which will be N/2), launch a thread with that parameter block, then do NOT launch another thread. instead use the current thread to recurse into merge_sort_mt() for the "other" half.
- Once the current thread returns from its recursion is must wait on the other thread via a join. once that is done both partitions will be done, and they can be merged using your trivial merge algorithm.

```
gary@gary-myubuntu:~/Desktop/HW2/Ch4_proj_2$ ./MultiThreadSorting.o
 542  483  199  994  981  121  874 1004
 564  843  918  502  445  171  467  307
 139   17  928  894  502  263  166    8
 930  598  535  885  251  263  287  793
 746  486  763  703  607  614  684  147
 433  578  649  878  750   92  161  889
 109   65  760  612  328  926  620  234
 500  132   95  752  395  382  521  117
 868  261  821  451  875  481  598  284
  35  223  139  785  315  300  651  425
 366  387   13  694  289  633  929  790
 765    0  518  136  383   15  254  227
 276   51  679  127  532  253  412  567
 477  551  329  792  851  980  193  193
 343  206  888  632  840  793  398  581
 793  916  718  152  932  972  380  184
1023   35  312  531  288  724   74  765
 251  403  534   78  359  727  272  702
```

```
 486  978  924  746  364  158  571  107
 171  174  725  280  546  955    5  496
 697  977  718  391  641  971  986  949
 494  349  632  585  433  803  137  919

Starting subthread...
Starting subthread...
Starting subthread...
Starting subthread...
Starting subthread...
Starting subthread...
Finished subthread.
Finished subthread.
Starting subthread...
Finished subthread.
Finished subthread.
Finished subthread.
Finished subthread.
Finished subthread.
   0    0    0    0    1    1    1    2
   2    2    3    5    5    5    5    6
   6    6    8    8    8    9    9   10
  10   12   12   12   13   13   13   13
```

```
 949  950  950  952  953  954  954  955
 955  955  956  956  957  957  958  958
 959  961  961  962  962  963  963  964
 964  964  965  966  966  967  968  970
 970  971  971  971  971  971  972  972
 972  973  973  973  973  973  973  973
 974  974  974  974  975  975  976  977
 977  978  978  978  978  978  979  979
 979  980  980  980  981  981  981  981
 982  982  982  983  984  984  985  985
 985  985  986  986  986  986  986  986
 986  986  987  987  988  988  988  988
 989  989  990  991  991  991  992  992
 992  993  993  993  993  994  994  994
 995  997  999 1000 1000 1001 1001 1001
1002 1002 1003 1003 1004 1004 1005 1005
1005 1005 1006 1006 1007 1008 1009 1009
1009 1011 1012 1012 1012 1013 1013 1013
1014 1014 1014 1015 1015 1015 1016 1017
1017 1018 1019 1020 1020 1020 1021 1021
1021 1022 1023 1023 1023 1023 1023 1023
```

The most important part of this is the limiters that keep us from going thread wild, which is easy to accidentally do with recursive threaded algorithms, and the join of the threads before merging their content with the other half of the partition, which we sorted on our thread, and may also have done the same thing.

CH5

The code first creates copies of the original process list for each scheduling algorithm. This is done to ensure that each algorithm operates on the same set of processes, allowing for a fair comparison of their performance.

After the process lists are copied, the program runs each scheduling algorithm in turn, printing the results to the console. The FCFS, SJF, and Priority Scheduling algorithms are run without any additional user input.

For the Round Robin Scheduling algorithm, the program prompts the user to enter a time quantum. The time quantum is a parameter of the Round Robin algorithm that determines how long each process is allowed to run before it is interrupted and moved to the back of the queue. After the user enters the time quantum, the Round Robin algorithm is run and its results are printed to the console.

If we enter the data like this

```
huangyukai@huangyukaideMacBook-Pro CH5 % ./process
Enter the number of processes: 5
Enter arrival time, burst time, and priority for process 1: 0 3 2
Enter arrival time, burst time, and priority for process 2: 2 6 1
Enter arrival time, burst time, and priority for process 3: 4 4 3
Enter arrival time, burst time, and priority for process 4: 6 5 2
Enter arrival time, burst time, and priority for process 5: 8 2 4
```

◆ FCFS Scheduling:

```
FCFS Scheduling:
Process Arrival Time    Burst Time      Waiting Time    Turnaround Time
1       0               3               0               3
2       2               6               3               9
3       4               4               9               13
4       6               5               13              18
5       8               2               18              20
Average Waiting Time: 8.60
Average Turnaround Time: 12.60
```

- SJF Scheduling:

```
SJF Scheduling:
Process Arrival Time    Burst Time      Waiting Time    Turnaround Time
1       0               3               0               3
2       2               6               1               7
5       8               2               1               3
3       4               4               7               11
4       6               5               9               14
Average Waiting Time: 3.60
Average Turnaround Time: 7.60
```

- Priority Scheduling:

```
Priority Scheduling:
Process Arrival Time    Burst Time      Priority        Waiting Time    Turnaround Time
P2      2               6               1               0               6
P1      0               3               2               6               9
P4      6               5               2               3               8
P3      4               4               3               10              14
P5      8               2               4               10              12

Average Waiting Time: 5.80
Average Turnaround Time: 9.80
```

- Round Robin Scheduling (time quantum = 4)

```
Enter time quantum: 4

Round Robin Scheduling:
Process Arrival Time    Burst Time      Waiting Time    Turnaround Time
P1      0               3               0               3
P3      4               4               3               7
P5      8               2               7               9
P2      2               6               11              17
P4      6               5               9               14

Average Waiting Time: 6.00
Average Turnaround Time: 10.00
```

- Priority with Round Robin Scheduling:

```
Priority with Round Robin Scheduling:
Process Arrival Time    Burst Time      Priority        Waiting Time    Turnaround Time
1       0               3               2               0               3
2       2               6               1               1               7
4       6               5               2               3               8
3       4               4               3               10              14
5       8               2               4               10              12
Average Waiting Time: 4.80
Average Turnaround Time: 8.80
```

Summary:

Team members

工管四乙 黃鈺凱 109370210 50%

資工三 林兆玄 110590021 50%