HW8.5. Fun with Parallelism

Feel free to check out the guide that we have prepared to help you in this problem.

In a multithreaded system, any order of operations could possibly happen, from each thread happenening serially, to all of them running at the exact same time. For example, if you have four threads, all 4 threads will execute (and inevitably finish) that same code flow. However, the threads do not necessarily start at the same time; the threads do not necessarily execute at the same rate (the code that each thread executes is not perfectly interleaved); the threads do not necessarily finish running the whole program at the same time.

For these sets of problems, let's explore what happens when you try to execute the same code on multiple threads.

Q1: We run the following code on four threads:

```
#pragma omp parallel
{
// Hint: loop variable i is declared inside the #pragma directive
// therefore, it is a private variable per thread.
    for (int i = 0; i < 3; i++) {
        printf("%d",i);
    }
}
```

Q1: Select all of the following outputs that are possible:
Hint: Think of the expected output of a single thread first, then imagine what happens if more threads start executing their own loops independently with different levels of interleaving.

- [x] (a) 012012012012 ✓
- [ ] (b) 010011201212
- [x] (c) 000011112222 ✓
- [ ] (d) 100001222112
- [x] (e) 010012201212 ✓
- [ ] (f) 000111102222

Select all possible options that apply. ?

✓ 100%

Q2: Say we have the following RISC-V code to be run by multiple threads:

```
li t0 10
sw t0 0(s0)
lw t0 0(s0)
add t0 t0 t0
sw t0 0(s0)
```

Hint: Note that the memory location being accessed by each thread is the same (i.e. it is a shared variable). Think about the access pattern that would result to different threads reading a different data from the shared memory.

Q2.1: What is the **smallest** possible value stored at `0(s0)` after two threads finish executing the code?

20 ? ✓ 100%

Q2.2: What is the **largest** possible value stored at `0(s0)` after two threads finish executing the code?

40 ? ✓ 100%

Q2.3: Now, let's extend it further. What is the **largest** possible value stored at `0(s0)` after four threads finish executing the code?
Hint: Answer Q2.2 correctly first to understand the execution pattern needed to get the worst case scenario.

Homework 8

Assessment overview

Total points:	100/100
Score:	100%

Question

Value:	40
History:	40
Awarded points:	40/40

Report an error in this question

Previous question

Next question

Attached files

No attached files

Attach a file

Attach text

160

✓ 100%

Q3: We run the following code on two threads:

```
// Hint: The following variables are declared outside the #pragma directive
// therefore, these variables are shared for all threads.
int y = 0;
int x = 3;
#pragma omp parallel
{
    while(x > 0) {
        y = y + 1;
        x = x - 1;
    }
}
```

Note that the expression `y = y + 1` is equivalent to three instructions: load value of `y`, add 1, store result to `y`. Another thread can execute in between those instructions. Moreover, `y` is a shared variable, which means that any changes that one thread does to `y` will be seen by the other thread. The same can be said for variable `x`. Exploit this fact when answering the questions below. It might be helpful if you can answer Q2 first before answering this one since you can imagine breaking down the C code into the assembly for a more fine-grained analysis.

Q3.1: What is the **smallest** value `y` can contain after the code runs?
Hint: One thread can complete the entire loop while the other thread has just started.

1

✓ 100%

Q3.2: What is the **largest** value `y` can contain after the code runs?
Hint: Since `x` is relatively small (loop count is small), try writing out the different `y` values for each thread as they execute. If you want to maximize the value of `y`, how can two threads 'force' more iterations to happen?

If `x` was initialized to 2 instead of 3, the largest possible value for `y` would be 5. Think about how to get this.

9

✓ 100%

Q3.3: Now, let's extend it further. If `x` was initialized to **10** instead of 3 (still a shared variable), what is the **largest** value `y` can contain after the code runs?
Hint: Answer Q3.2 correctly first to understand the execution pattern needed to get the worst case scenario.

If you answered Q3.2 by brute force (by writing all cases as each thread executes), you can find a formula on the worst case value of `y` given the initial value of `x` to help you answer this question.

65

✓ 100%

Try a new variant

Correct answer

Feel free to check out the guide that we have prepared to help you in this problem.

In a multithreaded system, any order of operations could possibly happen, from each thread happenening serially, to all of them running at the exact same time. For example, if you have four threads, all 4 threads will execute (and inevitably finish) that same code flow. However, the threads do not necessarily start at the same time; the threads do not necessarily execute at the same rate (the code that each thread executes is not perfectly interleaved); the threads do not necessarily finish running the whole program at the same time.

For these sets of problems, let's explore what happens when you try to execute the same code on multiple threads.

Q1: We run the following code on four threads:

```
#pragma omp parallel
{
// Hint: loop variable i is declared inside the #pragma directive
// therefore, it is a private variable per thread.
    for (int i = 0; i < 3; i++) {
        printf("%d",i);
    }
}
```

Q1: Select all of the following outputs that are possible:
Hint: Think of the expected output of a single thread first, then imagine what happens if more threads start executing their own loops independently with different levels of interleaving.

(a) 012012012012
(c) 000011112222
(e) 010012201212

000011112222: This is possible when all of the four threads are perfectly interleaved (i.e. thread 1 runs first printing 0, then thread 2 printing 0, ..., then thread 1 executes the next iteration printing 1, and so on.)

100001222112: This is not possible since we cannot print a 1 without printing a 0 first.

000111102222: This is not possible since we cannot print four 1s before we finish print all four 0s.

010011201212: This is not possible since there are five 1s in there (there are only four threads).

012012012012: This is possible when each thread starts and finishes execution one after the other.

010012201212: This is possible since the order of 0,1,2 are preserved for some manner of execution interleaving between the four threads.

Q2: Say we have the following RISC-V code to be run by multiple threads:

```
li t0 10
sw t0 0(s0)
lw t0 0(s0)
add t0 t0 t0
sw t0 0(s0)
```

Hint: Note that the memory location being accessed by each thread is the same (i.e. it is a shared variable). Think about the access pattern that would result to different threads reading a different data from the shared memory.

Q2.1: What is the **smallest** possible value stored at `0(s0)` after two threads finish executing the code?
`20`

We can imagine that all load words from all threads execute one after the other. This causes all the threads to load 10 and store 20.

Q2.2: What is the **largest** possible value stored at `0(s0)` after two threads finish executing the code?
`40`

We can imagine that each thread runs until the first store word instruction first (thus storing 10), then thread 1 runs to completion storing 20 at the memory location. When thread 2 resumes execution, it will load 20 from the memory location, which will result to thread 2 storing 40 at the end of the code.

Q2.3: Now, let's extend it further. What is the **largest** possible value stored at `0(s0)` after four threads finish executing the code?
Hint: Answer Q2.2 correctly first to understand the execution pattern needed to get the worst case scenario.

160

We can just extend the pattern we identified from Q2.2. We know that if all threads all execute until the first store word instruction, all of them will store 10. Then, thread 1 will run to completion, storing 20 at the memory location. Thread 2 will resume execution, loading 20, then storing 40 at the end of the code. Thread 3 will resume execution, loading 40, then storing 80. Finally, thread 4 resumes execution, loading 80, storing 160.

Q3: We run the following code on two threads:

```
// Hint: The following variables are declared outside the #pragma directive
// therefore, these variables are shared for all threads.
int y = 0;
int x = 3;
#pragma omp parallel
{
    while(x > 0) {
        y = y + 1;
        x = x - 1;
    }
}
```

Note that the expression `y = y + 1` is equivalent to three instructions: load value of `y`, add 1, store result to `y`. Another thread can execute in between those instructions. Moreover, `y` is a shared variable, which means that any changes that one thread does to `y` will be seen by the other thread. The same can be said for variable `x`. Exploit this fact when answering the questions below. It might be helpful if you can answer Q2 first before answering this one since you can imagine breaking down the C code into the assembly for a more fine-grained analysis.

Q3.1: What is the **smallest** value `y` can contain after the code runs?
Hint: One thread can complete the entire loop while the other thread has just started.

1

We can imagine all threads enter the loop and reads the value of `y = 0`. Then, thread 1 runs to completion, which will result to `x = 0` by the end of the loop. Thread 2 resumes execution, performs `y = y + 1` using the initial value of `y` it read before, which was 0. Thus, `y = y + 1 = 1`. `x` will be updated as well, but since `x` is already 0 as set by thread 1, `x = x - 1 = -1`. Thus, the loop for thread 2 ends, giving `y = 1`.

Q3.2: What is the **largest** value `y` can contain after the code runs?
Hint: Since `x` is relatively small (loop count is small), try writing out the different `y` values for each thread as they execute. If you want to maximize the value of `y`, how can two threads 'force' more iterations to happen?

If `x` was initialized to 2 instead of 3, the largest possible value for `y` would be 5. Think about how to get this.

9

We can imagine all threads enter the loop, each executes `y = y + 1`, and reads the value of `x = 3`. At this point, thread 1 and thread 2 would have contributed a +1 to `y`. Then, thread 1 runs *almost* to completion, until the final `x = x - 1`, which would set `x = 0`. At this point, thread 1 would have contributed a +3 to `y` in total (it looped three times). But then, thread 2 resumes execution and performs `x = x - 1` using the value of x that it read earlier, which was 3. Thus, `x = x - 1 = 2`. Thread 2 has 'reset' the value of `x`. Thus, thread 1 can loop again with `x = 2`. The entire process repeats where thread 1 runs *almost* to completion again, which would have contributed another +2 to `y`. The pattern continues where thread 2 will 'reset' the value of `x` again to 1 this time, and so on. In the end, it is as if thread 2 is executing normally, which would contribute +3 to `y`, but thread 1 first loops 3 times, then 2 times, then 1 time, because thread 2 keeps on 'resetting' `x`. Thread 1 will contribute +3+2+1 to `y`. Thus, the maximum value `y` can reach is 3 (from thread 1) + 3+2+1 (from thread 2) = 9.

Q3.3: Now, let's extend it further. If `x` was initialized to **10** instead of 3 (still a shared variable), what is the **largest** value `y` can contain after the code runs?

Hint: Answer Q3.2 correctly first to understand the execution pattern needed to get the worst case scenario.

If you answered Q3.2 by brute force (by writing all cases as each thread executes), you can find a formula on the worst case value of `y` given the initial value of `x` to help you answer this question.

65

Following the same pattern we observed from Q3.2. We can imagine thread 2 executing normally, which would contribute +10 to `y`, but thread 1 first loops 10 times, then 9 times, then 8 times, and so on, because thread 2 keeps on 'resetting' `x`. Thread 1 will contribute +10+9+8+...+1 to `y`. Thus, the maximum value `y` can reach is 10 (from thread 1) + 10+9+8+7+6+5+4+3+2+1 (from thread 2) = 65.

Submitted answer 2 correct: 100%

Submitted at 2022-11-09 04:39:26 (PST)

hide

Feel free to check out the guide that we have prepared to help you in this problem.

In a multithreaded system, any order of operations could possibly happen, from each thread happenening serially, to all of them running at the exact same time. For example, if you have four threads, all 4 threads will execute (and inevitably finish) that same code flow. However, the threads do not necessarily start at the same time; the threads do not necessarily execute at the same rate (the code that each thread executes is not perfectly interleaved); the threads do not necessarily finish running the whole program at the same time.

For these sets of problems, let's explore what happens when you try to execute the same code on multiple threads.

Q1: We run the following code on four threads:

```
#pragma omp parallel
{
// Hint: loop variable i is declared inside the #pragma directive
// therefore, it is a private variable per thread.
    for (int i = 0; i < 3; i++) {
        printf("%d",i);
    }
}
```

Q1: Select all of the following outputs that are possible:
Hint: Think of the expected output of a single thread first, then imagine what happens if more threads start executing their own loops independently with different levels of interleaving.

(a) 012012012012✓
(c) 000011112222✓
(e) 010012201212✓
✓100%

Q2: Say we have the following RISC-V code to be run by multiple threads:

```
li t0 10
sw t0 0(s0)
lw t0 0(s0)
add t0 t0 t0
sw t0 0(s0)
```

Hint: Note that the memory location being accessed by each thread is the same (i.e. it is a shared variable). Think about the access pattern that would result to different threads reading a different data from the shared memory.

Q2.1: What is the **smallest** possible value stored at `0(s0)` after two threads finish executing the code?
20 ✓100%

Q2.2: What is the **largest** possible value stored at `0(s0)` after two threads finish executing the code?

40 ✓ 100%

Q2.3: Now, let's extend it further. What is the **largest** possible value stored at `0(s0)` after four threads finish executing the code?
Hint: Answer Q2.2 correctly first to understand the execution pattern needed to get the worst case scenario.

160 ✓ 100%

Q3: We run the following code on two threads:

```
// Hint: The following variables are declared outside the #pragma directive
// therefore, these variables are shared for all threads.
int y = 0;
int x = 3;
#pragma omp parallel
{
    while(x > 0) {
        y = y + 1;
        x = x - 1;
    }
}
```

Note that the expression `y = y + 1` is equivalent to three instructions: load value of y, add 1, store result to y. Another thread can execute in between those instructions. Moreover, y is a shared variable, which means that any changes that one thread does to y will be seen by the other thread. The same can be said for variable x. Exploit this fact when answering the questions below. It might be helpful if you can answer Q2 first before answering this one since you can imagine breaking down the C code into the assembly for a more fine-grained analysis.

Q3.1: What is the **smallest** value y can contain after the code runs?
Hint: One thread can complete the entire loop while the other thread has just started.

1 ✓ 100%

Q3.2: What is the **largest** value y can contain after the code runs?
Hint: Since x is relatively small (loop count is small), try writing out the different y values for each thread as they execute. If you want to maximize the value of y, how can two threads 'force' more iterations to happen?

If x was initialized to 2 instead of 3, the largest possible value for y would be 5. Think about how to get this.

9 ✓ 100%

Q3.3: Now, let's extend it further. If x was initialized to **10** instead of 3 (still a shared variable), what is the **largest** value y can contain after the code runs?
Hint: Answer Q3.2 correctly first to understand the execution pattern needed to get the worst case scenario.

If you answered Q3.2 by brute force (by writing all cases as each thread executes), you can find a formula on the worst case value of y given the initial value of x to help you answer this question.

65 ✓ 100%

Submitted answer 1 partially correct: 85%
Submitted at 2022-11-09 04:39:00 (PST)

show