ECE 459 W24 Final Exam Solutions

(1)

(a) You can argue this one either way, either saying it's distributing the load a bit such that we don't get rate limited across accounts. But you can also argue that it's not a great plan because the other party will probably catch on and ban us.

(b) We should create the next students' repos and come back to the branch protection rules afterwards. Or alternatively, we should wait for a notification or poll to see when it's ready so we can move on ASAP. Either one of those answers is sufficient.

(c) Any reasonable scenario works here, like database (deciding to commit/rollback a transaction), or an embedded system (choose the "safe" option by default).

(d) Ideas:
1. Archive those accounts (move them to a different table or DB);
2. Delete those accounts (remove them entirely)
3. Add an index on a relevant field to make it faster for the DB server to skip them.

(e) This is task parallelism; you can think of it as each thread spawning n other threads until no further threads are needed. The benefit of the system comes from the fact that after the first round, we have n people calling others in parallel, allowing the messages to be passed concurrently.

(f) It could be trading more accuracy for time if your system can do the 16-bit operations faster than the 32-bit ones; if you just end up with type promotion or there's no way to do two 16-bit operations in the space of one 32-bit operation then it doesn't give any benefit.

(g) Data Stream: `00000050 00bc6386 e9451dd1 b3e5e7b7 00153658 00009ab2`

(h) If we delete some unneeded abstraction (e.g., we have a superclass that would make sense to just turn into two concrete classes) then it makes the compiler's job much easier.

(i) Possible answers include: Not enough time to do all of them in total, or some optimizations conflict with each other, or that after doing one of these optimizations, another provides no/minimal additional benefit. Any two would get full marks.

(j) As long as the answer addresses the question about the student's use of LLM tools here and is 2-3 sentences gets full marks.

(2.1)

Arrivals are 135 000 per hour or 37.5/s. 15 ms to complete a request is a service rate of 0.015 which makes $\rho = 0.28125$.

Property	**Value**	**Property**	**Value**
Kendall Notation Description	M/M/2	Utilization (ρ)	0.28125
Intermediate Value (K)	0.9083	Completion time average (T_q)	0.0162
Average queue length (W)	0.04821	Maximum arrivals/hour the system can handle (full utilization)	480 000
Maximum average service time the system can tolerate	0.0533 ms	–	–

To find the max arrivals per hour, leave s unchanged but set utilization to 1 and solve for λ. To find the max average service time leave λ unchanged but set utilization to 1 and solve for s.

(2.2)

There's two good ways to argue this, and you need only one to get full marks on this question.

Option 1: It matters because the willingness to wait is proportional to the size of the task being done. It's frustrating to wait a long time for a short job. In particular, the user might think it is reasonable if it takes 10 times longer to process file A than it takes for B if A is 10 times bigger than B. From the user's point of view, they don't really care how much time was waiting versus processing, they just care about (total) response time.

Option 2: Mean response time can be skewed by one or two extremely large jobs since they have a very large processing time. On the other hand, all jobs count equally in the slowdown calculation because the job size appears in the calculation of slowdown as the service time component.

(2.3) Answers all certainly vary here, but the discussion should touch on queue length and guest experience at least once each. Some of my immediate thoughts are that if it becomes just money to go to the fast lane, it's possible that everyone will just pay it and the fast lane won't be any better, negating the benefit. Another one is that it would negatively affect guest experience for those who aren't so wealthy as to be able to spend that money – the people for whom this is a once-in-a-lifetime experience, but at the same time positively affect the guest experience of those with the extra money to spend. But you could say it would actually be good for everyone if it distributes demand better than the existing method, or that it's more transparent and therefore people would be better able to make good decisions about where and how to spend time.

(3.1)

Part i We're looking for something that shows an expensive operation with a low percentage of CPU time charged by perf followed by a large cost charged to an innocent instruction. So something like:

```
ld r1,0x12341234     0.1%
add r2,r3            1.0%
sub r3,r4            1.0%
NOP                 27.0%
```

The NOP is obviously not the cause here, it's the load instruction that really is the expensive one.

Part ii You were asked to speculate so it's okay if the answer aren't exactly what's here, but something like:

- It might make the code execute slower as a result of needing to keep track of things more precisely
- It might not (and actually doesn't) support all types of instructions/scenarios so it might not work at all for the part of the program you want to profile
- It might present misleading results sometimes that would be worse than the mode with skid

Any two such points would do.

(3.2)

Register Allocation A register is a resource in the CPU and conflicts over those resources would force threads to wait until the register becomes available. Optimizing register uses means minimizing conflict and therefore waiting.

(For this you could also say that if registers aren't available we have to write to / read from memory and that's more expensive in time than register access. Same idea, different explanation)

Exception Handling Code Separation Exception handling code is supposed to run rarely, so moving it away from the main code that runs frequently should decrease the chance that a function call or return results in a cache miss on the instructions page of the destination.

Loop Vectorization Doing more than one iteration of the loop in one pass or chunk is a significant increase in performance since it may reduce both the execution time of the loop contents, but also decreases the number of times the loop has to be repeated (reducing branch evaluation and misprediction costs).

(4.1)

```
// fn concat(x: &mut String, y: &mut String) -> &String {
fn concat<'a>(x: &'a mut String, y: &mut String) -> &'a String {
    println!("str to be concatenated {}", y);
    x.push_str(y.as_str());
    x
}

fn main() {
    let mut string1 = String::from("abcd");
    let result;
    {
        let mut string2 = String::from("xyz");
        result = concat(&mut string1, &mut string2);
    }
    println!("The string after concat is {}", result);
}
```

The only rule is that the lifetime `'a` should be added exactly as in the solution. It should not be added to `y`, otherwise it will cause the issue that `string2` does not live long enough.

Part ii. The question is based on a question "store a value and a reference to that value inside the same struct" on Stack Overflow.

TL;DR The error will happen at line `let family = Family { parent, child };`, because when you create `family`, the memory location of the original `parent` changes, causing `child.parent` to point to an invalid address (potentially).

The easiest fix is to use a reference to `parent` when creating the `family`, as shown here, with `Debug` as the derived trait for all structs.

```
#[derive(Debug)]
// ...
fn main () {
    // ...
    // let family = Family { parent, child };
    let family = Family { parent: &parent, child };
}
```

Another fix is to remove the reference entirely, as shown below. Note that all lifetimes should be also removed since they are no longer used. This is actually more preferred since your structure nesting will mimic the lifetimes of your code.

```
#[allow(dead_code)]
#[derive(Debug, Clone)]
struct Parent { name: String }

#[allow(dead_code)]
#[derive(Debug)]
struct Child { parent: Parent, name: String }

#[allow(dead_code)]
#[derive(Debug)]
struct Family { parent: Parent, child: Child }
```

```
fn main() {
    let parent = Parent {
        name: String::from("parent"),
    };
    let child = Child {
        parent: parent.clone(),
        name: String::from("child"),
    };

    let family = Family { parent, child };
    println!("{:?}", family);
}
```

(4.2)

Part i. The idea is simply spawning multiple threads. Note that since the question specifies the number of cores to be 8, so we spawn 8 threads, assuming there is no hyper-threading.

```
fn main() {
    let queue = setup_queue();
    let mylock = MySpinLock::new();
    let counter = MyCounter::new(0);

    thread::scope(|s| {
        for _ in 0..8 {
            s.spawn(|| loop {
                match queue.pop() {
                    None => return,
                    Some(task) => {
                        if task.execute() {
                            mylock.lock();

                            // Non-atomically increment the counter, while holding the lock.
                            let old = counter.load(Relaxed);
                            let new = old + 1;
                            counter.store(new, Relaxed);

                            mylock.unlock();
                        }
                    }
                }
            });
        }
    });
}
```

Part ii. In both `lock()` and `unlock()`, using `Relaxed` may cause problems. For example, in architectures such as x86 which use the strong memory consistency model—*i.e.*, no re-ordering in execution—the implementation is fine. However, for architectures which use the weak memory consistency model, `Relaxed` indicates that line can be ordered arbitrarily. Among all possible execution orders, some may produce incorrect result. For instance, if the line `mylock.lock()` is executed after `counter.store()`, it produces incorrect counts.

I verified the behavior using the code I put into the `code` folder. With `Relaxed`, the outputs are

```
single thread
counted: 30000000
multi thread
counted: 29999997
```

The solution is to make sure both `lock()` and `unlock()` are not incorrectly ordered during execution. Changing both `Relaxed` to `SeqCst` is one approach; a better approach is to use `Acquire` for `lock()` and `Release` for `unlock()`.

(5)

```
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)
{
    // Each thread computes one element of C
    // by accumulating results into Cvalue
    float Cvalue = 0;
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    for (int e = 0; e < A.width; ++e)
        // compute Cvalue using A.elements and B.elements
        Cvalue += A.elements[row * A.width + e]
                * B.elements[e * B.width + col];
    // set Cvalue to matrix C.elements
    C.elements[row * C.width + col] = Cvalue;
}
```

1. Each thread reads an entire row from `A` and an entire column from `B`, resulting in a total of `2048` reads (`1024` from `A` and `1024` from `B`).
2. Each thread writes a single value to `C`, resulting in `1` write per thread.

Part ii. This question will be trivial is one understands the engineering design of the blocked matrix multiplication. However, one can still learn it since enough context was given.

The high-level idea is that, instead of letting every thread to read all elements from the global memory, each thread just reads a subset of them, while the full set of elements are covered by all threads within the same thread block.

Each thread still calculates a single `Cvalue`, leading to `1` write to the global memory. However, despite needing `2048` elements for calculations and performing `1024` multiplications, the process is divided into smaller, identical steps:

In each step, a thread reads one element into `As` and one into `Bs`. Collectively, threads in a block gather all necessary elements for `As` and `Bs`.

After all elements in both `As` and `Bs` to be ready, every thread (say `thread[i][j]`) then calculates an accumulated sum (tracked by `Cvalue`) of `16` elements within the row of `A` (e.g., `As[i][0..15]`) and `16` elements within the column of `B` (e.g., `Bs[0..15][j]`).

Once every thread has successfully calculated the accumulated sum, the submatrix `As` is shifted by `16` elements to the right (e.g., `As[row][16..31]`) and the submatrix `Bs` is shifted `16` elements to the bottom for `B` (e.g., `Bs[15..31][col]`). Given a thread only reads one element into `As` and one element into `Bs` in each step out of `(1024/16)` steps, therefore, the total reads a thread performs is `1*(1024/16) + 1*(1024/16) = 128`.

From above, the first `__syncthreads` is to ensure all elements in both `As` and `Bs` to get ready, and the second `__syncthreads` is to ensure all threads has finished calculating the temporary sum so that `As` and `Bs` can be updated.

Therefore, the answers are:

1. The number of multiplications each thread needs to perform is still `1024`.
2. The first `__syncthreads` is to ensure all elements in both `As` and `Bs` to get ready
3. The second `__syncthreads` is to ensure all threads finish calculating the temporary sum
4. Each thread needs to perform `128` reads from the the global memory
5. Each thread needs to perform `1` write to the global memory?

Comparing to `2048` reads in Q1, we reduced the number of reads from the global memory by `16`.

(6)

```
Current Node Shutdown [3 marks] {
  for i in items {
    if i is in state M {
      write i to the database
    }
  }
  for node n in known_nodes {
    send leaving message to n
  }
}


Node Leaves ( node n ) [1 mark] {
  Remove n from known_nodes
}


Get Item ( item i ) [5 marks] {
  if i is in local cache {
    return i
  } else {
    for node n in known_nodes {
      if n has item i
        add to cache ( i )
        set i state to S
        return i
      }
    }
  }
  retrieve i from database
  add to cache ( i )
  set i state to E
  return i
}
```

```
Other Node Searching ( item i ) [4 marks] {
  if i is in local cache {
    if i is in state M {
      write i to the database
      set i state to S
      return i
    } else if i is in state E {
      set i state to S
      return i
    } else if i is in state S {
      return i
        }
  }
  return null //Or other indicator of not-found
}

Update Item ( item i ) [5 marks] {
  if i is in local cache {
    if i is in state S {
      for node n in known_nodes {
        send invalidate i message to n
      }
      set i state to M
    } else if i is in state E {
      set i state to M
      return
    } else if i is in state M {
      return // Nothing to do
    }
  }
  add i to the cache in state M
}

Invalidate ( item i ) [2 marks] {
  if i is in local cache {
    set i state to I
  }
}
```