p2exp1. The (weights) of questions are relative and may not necessarily add up to 100.

Convert this docx to PDF before submitting it to Collab
Complete in-semester survey: https://forms.gle/KzZAHpkkPDcjrbab8

Races and Synchronization

Code:

https://github.com/fxlin/p2-concurrency/tree/master/exp1

Short Q&A (5pts each)

- How many threads can run simultaneously (i.e. thread-level parallelism offered by CPU hardware) on the course server? If you are using a different machine, state it.
    - 40
- What does pthread_join() do in the given code?
    - The `pthread_join()` function checks if the threads can be suspended for another thread to run
- Why concurrent, unsynchronized updates to `the_counter` leads to program errors?
    - It leads to a race condition because multiple worker threads are updating the counter without excluding each other.
- The given counter.c invokes `atexit()`. What does the function do?
    - The function calls the `cleanUpLocks()` function when the program terminates.

1. Q&A. Zoom in the scene of race condition (5pts each)

Here is the assembly of function `add(long long *pointer, long long value)`, as dumped from objdump. Note that without assuming x86 knowledge from you, I showed the ARMv8 version below (compiled with -O2).

```
0000000000001600 <add>:
    long long sum = *pointer + value;
       f9400002    ldr x2, [x0]
       8b010041    add x1, x2, x1
    *pointer = sum;
       f9000001    str x1, [x0]
```

```
646 }
647         d65f03c0    ret
```

Read the assembly and answer:

i) How many bits in a `long long` type of integer?

   a. 64 bits

ii) Point out which instructions (by their line numbers above) constitute the window for a race condition.

   a. Instructions 642 and 645 since they both access memory

iii) Will the race condition still exist, if we run the program with multiple threads but on a single-core machine?

   a. It will not exist because each thread will not be modifying the same resource at the exact same time

2. Use spinlock & CAS (30pts)

Add the following mechanisms to the source code:

- one that protects the addition by a spin-lock, enabled by a **--sync=s** option. You will have to implement your own spin-lock operation.
- one that performs the add using compare-and-swap (CAS) primitives to ensure atomic updates to the shared counter, enabled by a **--sync=c** option. Note the name: compare-and-swap is the same as compare-and-exchange.

Note: the provided code can already parse these new options.

Sample output

Before (by the given code), count values corrupted, i.e. !=0.

```
$./counter --iterations=10000 --threads=10 --sync=s
test=add-s threadNum=10 iterations=10000 numOperation=200000
runTime(ns)=5640178 avgTime(ns)=28 count=-10113
$./counter --iterations=10000 --threads=10 --sync=c
test=add-c threadNum=10 iterations=10000 numOperation=200000
runTime(ns)=4469589 avgTime(ns)=22 count=-7513
```

After (expected from your code). With spinlocks and CAS, the final count value is correct, i.e. ==0.

```
$./counter --iterations=10000 --threads=10 --sync=s
test=add-s threadNum=10 iterations=10000 numOperation=200000
runTime(ns)=27917650 avgTime(ns)=139 count=0
```

```
$./counter --iterations=10000 --threads=10 --sync=c
test=add-c threadNum=10 iterations=10000 numOperation=200000
runTime(ns)=20609670 avgTime(ns)=103 count=0
```

Implementation hints: both spinlock and CAS shall be implemented using the GCC's atomic built-ins. Since the built-ins are architecture-independent, you do not have to write any assembly.

- The documentation can be found here. Some related discussion.
- Useful functions include `__atomic_compare_exchange_n()` and `__atomic_store_n()`
- These functions require memory order, for which you may specific `__ATOMIC_SEQ_CST`. (Q: could other memory order work?)
- Note: older GCC offers `__sync_XXX` built-ins, which are still supported today for backward compatibility. Avoid them. They are deprecated by the _atomic builtins.

Search for "todo" in the given source code for extra hints.

Deliverable:

[Upload a diff file named as ComputingID.diff]

3. Q&A. Measure slowdown due to synchronization (30 pts)

Compare the times taken for parallel updating the shared counter:

- Without any synchronization
- With mutex (--sync=m)
- With spinlock (--sync=s)
- With CAS (--sync=c)

Report the performance with the following arguments. Write a small paragraph to explain your observation. How many repeated runs did you execute? How do you ensure your executions were unaffected by other students who may run the experiments at the same time? Would different thread count and iteration count affect your observation?

```
./counter-nolock --iterations=100000 --threads=10
```

```
./counter --iterations=100000 --threads=10 --sync=m
./counter --iterations=100000 --threads=10 --sync=s
./counter --iterations=100000 --threads=10 --sync=c
```

p2exp1. The (weights) of questions are relative and may not necessarily add up to 100.

For each command, I ran the experiment around 5 – 10 times back-to-back. It seems as if running the experiment with no lock ran the fastest with an average time of around 13ns. However, the count did not work appropriately. The next fastest was using CAS with an average time per operation of about 100 ns but performed correctly, next was using mutex with an average time per operation around 150 ns but performed correctly, and the slowest was with spinlock with an average time per operation around 700 ns but performed correctly. I ensured my executions were unaffected by other students by running it multiple times and taking the average.   With a low number of iterations, it seems like each approach yields similar times. When using fewer threads, the spinlock and mutex struggled with time per operation averaging around 140 ns and 390 ns respectively.