18-344 Recitation 10

11/21/2025

Outline

1. Logistics
2. Lab 3 & 4
3. Concurrency, Parallelism, and Synchronization

Logistics

- Homework 9 released, due Nov 24 (Mon) (***in 3 days***)
- Homework 10 to be released on Nov 24, due Dec 3 (Wed) (in 1 week + 5 days)

- Lab 3
  - Code freeze: November 21 (Friday) (***today***)
    - Submit (push a commit to your repo) on GitHub Classroom
  - Report: November 24 (Monday) (***in 3 days***)
    - Submit on Gradescope
- Lab 4
  - Code freeze: December 11 (in 2 weeks + 6 days)
    - Submit (push a commit to your repo) on GitHub Classroom
  - Report: December 14 (in 3 weeks + 2 days)
    - Submit on Gradescope

- Exam 2 on December 3 (Wednesday) (in 1 week + 5 days)
  - Same location and time as regular lecture
  - ***If you miss no more than 2 lectures, for any reason, <u>after</u> the midterm exam, you may choose to use the homework average from the 2st half in place of the 2st exam.***

Lab 3 & 4
Final Tips & Reminders

Lab 3

- TLB configuration is up to you.
  - size, associativity, replacement policy?
  - You'll need to justify your design choice and implementability.
- Evaluate how the hierarchical page table you implement reduces the amount of memory required for page table.
- `vm_trace`
  - Normal to take a relatively long time to produce a full trace for a microbenchmark.
    e.g. Takes about 10 minutes for `void main() {return;}` and produces a 800KB full trace.
  - We recommend ***against*** running `vm_trace` with SPEC benchmarks.
    Write your own microbenchmarks with ***diverse memory access patterns*** instead.
  - Trace contains memory accesses to unmapped memory regions?
- Anyone implementing a hashing page table?

Lab 4

- Binning by source or destination? Why?
- What ***cache performance metrics*** are useful for evaluation?
  - Evaluate using your cache simulator from lab 2.
    - What configuration would you choose and why?
  - Talk to us if your cache simulator may have correctness issues.
- What ***runtime metrics*** to measure and how to measure them?
  - Time (aka wall-clock time) (aka elapsed real time) of:
    The whole program? Part of the program? Which part of the program?
  - If using `time` – use ***real*** time. Better options exist but not required (e.g. `std::chrono`).
  - Consider and do your best to eliminate the influence of external factors.
    e.g. Don't benchmark on a crowded machine where people are running a million processes.
  - Measuring performance in real-world terms will always give varying results.
    Measure multiple times and record variance.

Lab 4 cont'd

- We highly recommend using large, real-world graphs.
  - Makes the effectiveness of propagation blocking more significant and observable.
  - For SNAP, `web-Google`, and various social media graphs are good starting points.
  - You'll need to write scripts to convert real-world graphs to `EL` format.
    Make sure to handle unidirectional edges.
  - Real-world graphs are large.
    Please don't turn them in, and make sure to not exceed your ECE AFS disk quota.

- `graph.h` defines `MAX_VTX`
  - Need to change it for graphs of different sizes.
  - This is a C macro, so there are multiple ways to change it without needing to modify the code.
  - This affects binning, so ***do not*** set it to an arbitrarily large number (e.g. `INT_MAX`).

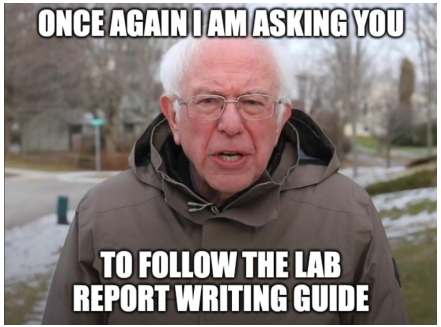
On reporting results and writing the report

- Be comprehensive.

  Read the handout (all of it, not just the final sections) carefully to find what's required in your report.

- Be quantitative.
  - e.g. For graph `G`, using binning by `{src,dst}`, simulating with a cache with configuration `C`:
    - Quantitative: Increasing the number of bins from `N` to `M` results in a `X%` increase in cache hit rate.
    - Qualitative: Increasing the number of bins results in an increase in cache hit rate.
  - Quantitative when reporting most evaluation results, qualitative when reporting general trends.

- Lab report writing resources:
  - Lab report writing guide
  - Report writing tips from recitations 1&5
  - Our feedback from recitation 9
  - Your graded lab reports from labs 1&2



Concurrency, Parallelism, and Synchronization

Concurrency & Parallelism

- Concurrency
  - Executing multiple tasks at once by allowing them to make progress in ***overlapping time periods***.
  - Tasks start, run, and complete in an interleaved way, but not necessarily at the exact same instant.
  - e.g. A single-core, single-thread system running multiple software threads.
    Only one software thread runs at a time, but the OS switches between them.

- Parallelism
  - Performing multiple operations at the ***same instant***.
  - Many types of parallelism available (non-exhaustive):
    - Data-level parallelism
      e.g. Using SIMD instructions to add two vectors together.
    - Task-level parallelism
      e.g. A single-core, multi-thread system running multiple software threads.
      All hardware threads run at the same time.
      In this case, the software threads are also executed concurrently.

*showing abridged code to save space

An awkwardly parallel program

```
1  void *thread(void):
2    for (int i = 0; i < K; i++):
3      size_t ind = rand() % N
4      arr[ind]++
```

Assuming a global, heap-allocated array of integers, **`arr`**, is shared across all threads.

Each thread randomly increment an element in **`arr`** and repeat until K times.

At any instant, multiple threads could be getting the same **`ind`**. Since **`++`** is a Read-Modify-Write sequence and ***not atomic***, there is a ***data race***.

*showing abridged code to save space

Adding synchronization, **spinlock** edition

```
1  void *thread(void):
2    for (int i = 0; i < K; i++):
3      size_t ind = rand() % N
4      spin_lock(&locks[ind])
5      arr[ind]++
6      spin_unlock(&locks[ind])
```

Assuming a global, heap-allocated array of `spinlock_t`s, **`locks`**, is shared across all threads and initialized appropriately.

When a thread attempts to acquire a `spinlock` and finds it already locked by another thread, it continuously checks the lock variable in a tight loop until the lock becomes available.

A spinlock can be implemented like:

```
while(test_and_set(&lock)) {
  // Busy-wait
}
```

*showing abridged code to save space

Adding synchronization, **__sync_fetch_and_add** edition

```
1  void *thread(void):
2    for (int i = 0; i < K; i++):
3      size_t ind = rand() % N
4      __sync_fetch_and_add(
5        &arr[ind], // ptr
6        1          // value
7      )
```

This is [a real gcc builtin function for x86](#).

type **__sync_fetch_and_add**
*(type *ptr, type value)*

As the name suggests, **__sync_fetch_and_add** fetches data from a given address `*ptr` then adds a given value to it ***in-place and atomically***.

*showing abridged code to save space

Adding synchronization, **__sync_bool_compare_and_swap** edition

```
1  void *thread(void):
2    for (int i = 0; i < K; i++):
3      size_t ind = rand() % N
4      bool success = false;
5      do:
6        success =
7          __sync_bool_compare_and_swap(
8            &arr[ind],     // *ptr
9            arr[ind],      // oldvar
10           arr[ind] + 1   // newval
11         )
12     while (!success)
```

This is a real gcc builtin function for x86.

```
bool __sync_bool_compare_and_swap
    (type *ptr, type oldval,
     type newval)
```

As the name suggests, **__sync_fetch_and_add** performs an ***atomic compare-and-exchange*** operation:

It compares the value at `*ptr` with `oldval`, and if they are equal, it atomically stores `newval` into `*ptr` and returns `true`; otherwise it leaves `*ptr` unchanged and returns `false`.

*showing abridged code to save space

Adding synchronization, **Transactional Memory**, 1st edition

```
1  void *thread(void):
2    for (int i = 0; i < K; i++)
3      size_t ind = rand() % N
4      bool success = false
5      do:
6        if (tm_begin() == TM_STARTED):
7          arr[ind]++
8          tm_end()
9          success = true
10     while (!success)
```

The do–while loop retries the same update until `tm_begin()` succeeds.

On success, the thread increments `arr[ind]` and commits.

On any abort or failure to start, the transaction is retried immediately.

If a transaction repeatedly aborts on the chosen index, the thread ***keeps retrying the same conflicting operation forever***.

*showing abridged code to save space

Adding synchronization, **Transactional Memory**, 2nd edition

```
1  void *thread(void):
2    for (int i = 0; i < K; i++)
3      size_t ind = rand() % N
4      for (int j = 0; j < LIMIT; j++):
5        if (tm_begin() == TM_STARTED):
6          arr[ind]++
7          tm_end()
8          goto 12
9      spin_lock(&locks[ind])
10     arr[ind]++
11     spin_unlock(&locks[ind])
12
```

Solution: add back-off to `tm_begin()`.

If a transaction repeatedly aborts or fails to start on the chosen index, the thread tries up to **`LIMIT`** times before giving up and falling back to use other synchronization methods.

On a `tm_begin()` success, control needs to skip the fall-back using goto.

Transactions can ***inevitably fail*** if another thread is in fall-back and about the increment the array element.

Abort becomes more expensive for longer transactions.

*showing abridged code to save space

Adding synchronization, **Transactional Memory**, final edition

```
1  void *thread(void):
2    for (int i = 0; i < K; i++)
3      size_t ind = rand() % N
4      for (int j = 0; j < LIMIT; j++):
5        if (tm_begin() == TM_STARTED):
6          if (locks[ind] == LOCKED):
7            tm_abort()
8          else:
9            arr[ind]++
10           tm_end()
11           goto 15
12     spin_lock(&locks[ind])
13     arr[ind]++
14     spin_unlock(&locks[ind])
15
```

Optimization: after beginning a transaction, ***proactively check for the lock status*** of all memory addresses participating in the transaction, and ***`tm_abort()` early*** if any memory address' lock has already been acquired.

Transactional memory would have been great if it weren't for the many **security vulnerabilities**