CS 111
Operating Systems Principles
Section 1E Week 5

Tengyu Liu

Project 4

- If you don’t have a beaglebone kit, BUY ONE NOW!

Previously in project 2A…

- Time per operation increases with the number of threads
  - Shouldn't it be decreasing?

Previously in project 2A…

- Time per operation increases with the number of threads
  - Shouldn’t it be decreasing?
- The problem
  - A lot of waiting during each operation
  - While waiting, other threads are working in parallel
  - More threads = more waiting

Previously in project 2A…

- Time per operation increases with the number of threads
  - Shouldn't it be decreasing?
- The problem
  - A lot of waiting during each operation
  - While waiting, other threads are working in parallel
  - More threads = more waiting
- The bottleneck
  - Parallel threads cannot run in parallel if they acquire the same lock

Previously in project 2A…

- Multithreading does slow down things…
    - All threads can run in parallel until they acquire a lock
    - At that point they have to wait in line
    - This is called **contention**

Previously in project 2A…

- Multithreading does slow down things…
  - All threads can run in parallel until they acquire a lock
  - At that point they have to wait in line
  - This is called **contention**
- Contention is worse for list operations. Why?

Previously in project 2A…

- Multithreading does slow down things…
  - All threads can run in parallel until they acquire a lock
  - At that point they have to wait in line
  - This is called **contention**
- Contention is worse for list operations. Why?
  - Inserting in a list requires more work than addition
  - The time spent in the critical section is higher for list operations
  - Therefore all other threads need to wait longer

Previously in project 2A…

- Multithreading does slow down things…
  - All threads can run in parallel until they acquire a lock
  - At that point they have to wait in line
  - This is called **contention**
- Contention is worse for list operations. Why?
  - Inserting in a list requires more work than addition
  - The time spent in the critical section is higher for list operations
  - Therefore all other threads need to wait longer
- How can we decrease contention for lists?

Previously in project 2A…

- Multithreading does slow down things…
  - All threads can run in parallel until they acquire a lock
  - At that point they have to wait in line
  - This is called **contention**
- Contention is worse for list operations. Why?
  - Inserting in a list requires more work than addition
  - The time spent in the critical section is higher for list operations
  - Therefore all other threads need to wait longer
- How can we decrease contention for lists?
  - Splitting lists into sub-lists and lock them independently

Project 2B

Part A

- Identify the contention problem
  - Starting from project 2A, measure the throughput (# of total ops/sec)
  - Pinpoint the most time-consuming part of the program

Part B

- Resolve the lock contention problem

Measuring throughput

- In a single-threaded program, time per op is a reasonable performance measure
- For multi-threaded implementations, we are also concerned about **how well we are taking advantage of parallelism**
- Measuring the aggregated throughput
  - # of total ops / total time taken
  - Plot with 1000 iterations and (1,2,4,8,12,16,24) threads for both mutex and spin-lock
  - You should see the throughput drop as #threads increase

Pinpointing bottleneck

- You used 2 types of locks to protect critical sections
  - Spin-locks
  - Mutexes
- Both prevented race conditions
  - Do they add any synchronization cost?
  - Is one better than another?

Synchronization cost

Spin-Lock

Mutex

Synchronization cost

Spin-Lock

- CPU cycles are used as thread tries to acquire the lock

Mutex

- Context switches when the thread can’t acquire the lock

Synchronization cost

Spin-Lock

- CPU cycles are used as thread tries to acquire the lock

Mutex

- Context switches when the thread can’t acquire the lock

How can we measure that overhead?

Measuring the cost for spin-locks

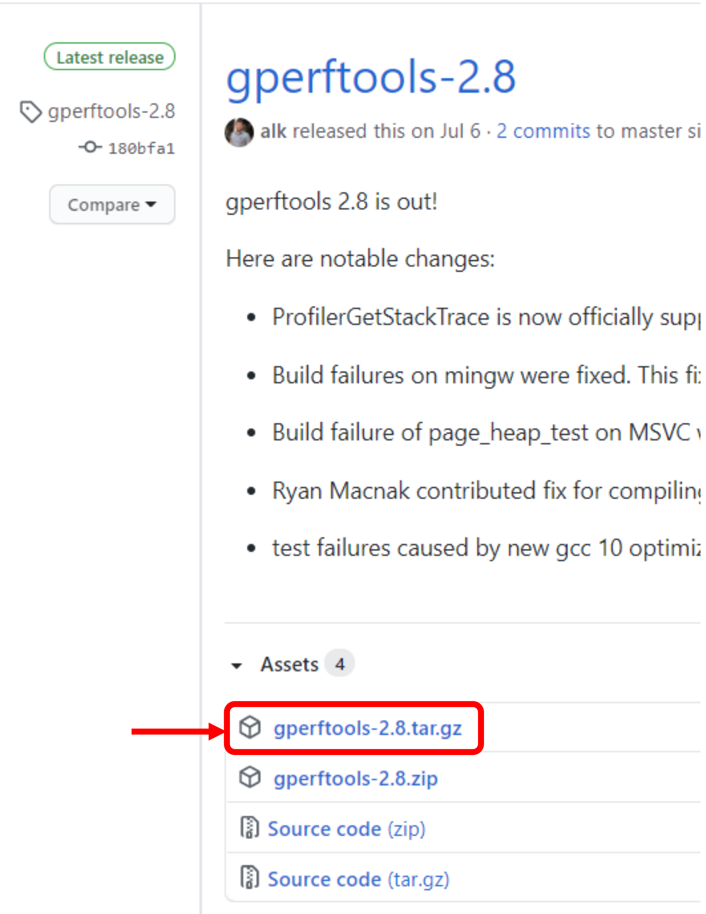
- CPU Profiling tool
  - Takes a snapshot of the CPU every millisecond
    - Interval is a parameter
  - At each snapshot, it records what function is running
  - After the process has finished, we can look at the log
  - If 100 snapshots were taken in function X, we can estimate that function X ran for 100ms

Measuring the overhead for spin-locks

- CPU Profiling tool
  - Takes a snapshot of the CPU every millisecond
    - Interval is a parameter
  - At each snapshot, it records what function is running
  - After the process has finished, we can look at the log
  - If 100 snapshots were taken in function X, we can estimate that function X ran for 100ms
- What do we measure for spin-locks?

CPU profiling tool - gperftools

- Installation
  - https://github.com/gperftools/gperftools/releases
  - Unpack and cd to directory
  - ./configure --prefix=<path>
  - make
  - make install
  - (make clean)



CPU profiling tool - gperftools

- Usage
  - Run your executable with the CPUPROFILE environment variable set
    - LD_PRELOAD=<path/to/libprofiler> CPUPROFILE=<path/to/profile_output> <path/to/binary> [binary args]
  - Run pprof to analyze the CPU usage
    - pprof <path/to/binary> <path/to/profile_output>
    - Profiling report will be printed to stdout

CPU profiling tool - gperftools

- Usage
  - Run your executable with the LD_PRELOAD and CPUPROFILE environment variable set
    - LD_PRELOAD=<path/to/libprofiler> CPUPROFILE=<path/to/profile_output> <path/to/binary> [binary args]
  - Run pprof to analyze the CPU usage
    - pprof <path/to/binary> <path/to/profile_output>
    - Profiling report will be printed to stdout

- Example
  - Run your executable with the LD_PRELOAD and CPUPROFILE environment variable set
    - LD_PRELOAD=/usr/lib/libprofiler.so CPUPROFILE=raw.perf lab2_list --threads=12 --iterations=1000 --sync=s
  - Run pprof to analyze the CPU usage
    - pprof lab2_list raw.perf

CPU profiling tool - gperftools

- Output
  - Obtained from
    - `pprof --text <path_to_binary> <path_to_profile_output>`
  - Text output will produce lines that look like
  - `14    2.1%    17.2%      58    8.7%    std::_Rb_tree::find`

CPU profiling tool - gperftools

- Output
  - Obtained from
    - `pprof --text <path_to_binary> <path_to_profile_output>`
  - Text output will produce lines that look like
  - `14     2.1%     17.2%     58     8.7%     std::_Rb_tree::find`
  - # samples (ms) spent in this function
  - % of samples in this function
  - Cumulative % of samples spent so far
    - % on this line + % on all previous lines. Last line shows 100%.
  - # samples in this function + callees
  - % of samples spent in this function + its callees
  - Function name

CPU profiling tool - gperftools

- Sampling result
  - Obtained from
    - pprof --list=<func_name> <path_to_binary> <path_to_profile_output>

Sample count

```
 1622   1622    17:   for (i = 0; i < MAX_SUM/thread_num; i++)
    .      .    18:   {
13794  13794    19:           while (__sync_lock_test_and_set(&lock, 1));
  769    769    20:           sum++;
    2      2    21:           __sync_lock_release(&lock);
    .      .    22:   }
```

Measuring the overhead for mutexes

- Can we use gperftools to measure?

Measuring the overhead for mutexes

- We can’t use gperftools! Why?
  - When a mutex can’t be acquired, the thread goes to sleep.
  - The time spent sleeping can’t be measured by counting CPU cycles.
- How do we measure the overhead for mutexes?

Measuring the overhead for mutexes

- We can't use gperftools! Why?
  - When a mutex can't be acquired, the thread goes to sleep.
  - The time spent sleeping can't be measured by counting CPU cycles.
- How do we measure the overhead for mutexes?
  - Directly measure the time taken to acquire a mutex

Timing mutex waits

- Computing wait-for-lock time
  - Measure time before and after getting the lock
  - Add up all wait time for all threads
  - Divide by number of operations
  - Output statistics for the run

Discovering and addressing the problem

- The timings allow you to prove the original intuition: degradation is a result of increased contention
- To decrease contention, we can split the list
  - Add a --lists option to specify the number of sub-lists
- Each thread does:
  - Insert *#iterations* nodes to the multi-list
    - Which sub-list to insert in is determined by a hash function
  - Look-up and delete inserted nodes

Inserting into different sub-lists

- No requirement of total ordering
  - Each sub-list must be ordered
  - But there is no order requirement between sub-lists
- Selecting sub-list
  - Use a hash function to map value -> sub-list
  - Doesn’t matter which hash function you use
  - Could be as simple as a modulo operation

Why is this more efficient?

Why is this more efficient?

- Splitting lists increases efficiency by decreasing contention
  - Not all threads try to acquire the same lock
- What else?

Why is this more efficient?

- Splitting lists increases efficiency by decreasing contention
  - Not all threads try to acquire the same lock
- The sub-lists are shorter!
  - Insertion operations ($O(N)$) will be faster
  - The time spent in the critical section will decrease
  - All threads, on average, wait a shorter time to acquire a lock
  - Fewer cycles wasted to acquire a lock

In this project you will…

- Modify lab2a to support
  - insertion and deletion to multiple sub-lists
- Time synchronization cost
  - Using profiling tool and regular timing function
- Evaluate profiling report and answer questions

Questions?