# CS 111 week 5
# Project 2b: Lock contention

Discussion 1B

Tianxiang Li

# Lock contention: the problem

```
#define MAX_SUM 100000000
void * thread_worker(void * unused)
{
    int i = 0;
    for (i = 0; i < MAX_SUM/thread_num; i++) //thread_num is the total number of threads
    {
        while (__sync_lock_test_and_set(&lock, 1));
        sum++;
        __sync_lock_release(&lock);
    }
}
```

Will we gain speedup if we increase the number of threads?

*Parallel threads are not able to run in parallel if they concurrently attempt to access a shared lock.*

# Project 2B: Overview

Part A: Identify the lock contention problem

1. Starting from lab2a_list, measure throughput (# of total ops/seconds)
2. Use google profile tool to pinpoint which the line causing the problem (spin-lock)
3. Measure the time spent on acquiring mutex (mutex-lock)

Part B: Resolve the lock contention problem:

Implement the linked list with N sublists.

--enable parallel access to the list

# Submission Tarball

- **sortedList.h** (do not modify)
  - linked list operation interfaces, must implement
- **sortedList.c**
  - implements insert, delete, lookup, and length methods for a sorted doubly linked list
- **lab2_list.c**
  - --threads, --iterations, --yield, --sync, --lists
  - Parallel threads operate on linked list, output performance result
- **profile.out**
  - profiling report: where time was spent in the un-partitioned spin-lock implementation
- **lab2b_list.csv**
- **graphs** (5 png)
- **README** (included files, refs, other info, e.g. research, limitations, features, testing methodology)
- **Scripts** (lab2b.gp)
- **Makefile**
  - **default** ... complile executable ( **-Wall -Wextra** options)
  - **tests** ... run tests cases, generate CSV results
  - **profile** ... run tests with profiling tools (google-pprof)
  - **graphs** ... generate graphs (gnuplot lab2b.gp)
  - **dist** ...  *.c,*.h,*.gp, *.csv,*.out,*.png, README, Makefile
  - **clean** ...  tarball,exec,*.o

# Review Your results from previous Lab

- All the locks experience exponential rise of **cost/operation** vs **#threads**
- Reason
  - *Spin lock is using cycles*
  - *Mutex lock have increased wait time*

# Testing

- In a single-threaded implementation, time per operation is a very reasonable performance metric.
- But for **multi-threaded** implementations we are also concerned about how well we are **taking advantage of the available parallelism**.
- This is more clearly seen in the **aggregate throughput** (total operations per second for all threads combined).
  - **Mutex** synchronized list operations, 1,000 iterations, 1,2,4,8,12,16,24 threads
  - **Spin-lock** synchronized list operations, 1,000 iterations, 1,2,4,8,12,16,24 threads
  - Capture the output, and produce a plot of **the total number of operations per second** for each synchronization method

# Part A: Using the Profile Tool

# Profile Tool: motivation

```
#define MAX_SUM 100000000
void * thread_worker(void * unused)
{
    int i = 0;
    for (i = 0; i < MAX_SUM/thread_num; i++) //thread_num is the total number of threads
    {
        while (__sync_lock_test_and_set(&lock, 1));
        sum++;
        __sync_lock_release(&lock);
    }
}
```

How can we design a tool to help us find the bottleneck of the program?
(which function/LOC takes the most time)

Ideally such a tool needs to be
1. Transparent: Does not require modifications to the program
2. Low overhead
3. Easy to enable/disable

# Profile Tool Key idea: Sampling

Key idea of profile tool: sampling

For every N (say 1,000,000) instructions/cycles, check the program counter, see which function/LOC it falls in. Aggerate the results when the program exits.

Sampling tool satisfies all the three goals we have mentioned:

Transparent,

Low-overhead: Sampling rate is low

Easy to enable/disable: Explained later

# Execution Profiling

**Analyze a program's execution to determine how much time is being spent in which routines**

- The standard Linux *gprof(1)* tool
  - is quite simple to use, but its **call-counting mechanism** is **not-multi-thread safe**, and its execution sampling is not multi-thread aware.
  - not usable for analyzing the performance of multi-threaded applications.
- *valgrind*
  - best known for its memory leak detector, which has an interpreted execution engine that can extract a great deal of information about where CPU time is being spent, even estimating cache misses.
  - It does work for multi-threaded programs, but its **interpreter does not provide much parallelism**. As such it is not useful for examining high contention situations.
- ***gperftools***
  - a wonderful set of performance optimization tools from Google. It includes a profiler that is quite similar to gprof (in that it samples real execution). This is probably the **best tool to use for this problem**.

# Google Profile Tool: Installation

**Using package management tool:**

With root permission:

*apt-get install google-perftools* (Ubuntu)

*yum install gperftools* (Old Version of CentOS, Fedora)

*dnf install gperftools* (New Version of CentOS, Fedora)

**Install from source code:**

*git clone [https://github.com/gperftools/gperftools](https://github.com/gperftools/gperftools)*

*./autogen.sh*

*./configure* # use --prefix option to set installation path if needed

*make*

*make install*

*Note: First check whether lnxsrv09 already has installed gperftools*

# Google Profile Tool: pprof

- pprof **reads** a collection of **profiling samples** and **generates reports** to visualize data
  - run the spin-lock list test (1,000 iterations 12 threads) under the profiler
- Usage
  - **google-pprof** [<u>options</u>] <u><program></u> <u><profile></u>
    - options
      - **--text** Generate text report [default]: Shows you which routine is consuming most of the CPU time
      - **--list=**<routine> Generate source listing of matching routines: Give you a source-level breakdown of how much time is being spent on each instruction
    - program
      - ./lab2_list
    - profile
      - ./raw.gperf

# Google Profile Tool: Usage

1. **Compile** your program with **–g**
2. Find where you have installed **libprofiler.so** and **pprof**, the below example assume ~/lib/libprofiler.so (profiling library) and ~/bin/pprof (user level app)
3. **LD_PRELOAD**=~/lib/libprofiler.so

   **CPUPROFILE**=./raw.gperf ./lab2_list --threads=12 --iterations=1000 --sync=s

   # **LD_PRELOAD**: *link the library*, overload existing function in my_prog

   # **CPUPROFILE**: gperf specific *environment variable*, tells gperf where to store the raw

   # profiling data. (perform sampling on ./lab2_list output to ./raw.gperf)

4. ~/bin/pprof --text ./lab2_list ./raw.gperf

#analyze raw sampling file, show sampling results in functions (i.e. how often does sampling fall into individual functions) step 4,5 generates readable output from raw output

5. ~/bin/pprof --list=thread_worker ./lab2_list ./raw.gperf

   # show sampling results in function thread_worker (i.e. how often does sampling fall into each lines of code in thread_worker)

# Analyzing text output

*~/bin/pprof --text ./myprog ./raw.gperf*

Total: 16187 samples

Text mode has lines of output that look like this:

16187 100.0% 100.0% 16187 100.0%. thread_worker

Here is how to interpret the columns:

- Number of profiling samples this function was running in
- Percentage of profiling samples this function was running in
- Percentage of profiling samples in the functions printed so far
- Number of profiling samples in this function and its callees
  - The number of samples in which the function appeared (either running or waiting for a called function to return)
- Percentage of profiling samples in this function and its callees
- Function name

# Sampling results

*~/gperf/bin/pprof --list=thread_worker ./test ./raw.gperf*

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:  }
```

# Timing Mutex Waits

- **spin-lock**
  - profiling can tell us **where we are spending most of our time**.
- **mutex**
  - a thread that cannot get the lock is **blocked**, **does not consume CPU time**.
  - Profiling only tells us **what code we are executing**. It doesn't tell us anything about the **time we spend** blocked.
  - How could we confirm that, in the mutex case, most threads are spending most of their time waiting for a lock?

# Part A: Measuring the lock contention time

# Code snippet of thread_worker from lab2a

```
void * thread_worker(threadNum) {
        startIndex = threadNum * iteration;
        for (i = startIndex; i < startIndex + iteration; i++)
        {
                pthread_mutex_lock(&list_lock);
                SortedList_insert(listhead, pool[i]);
                pthread_mutex_unlock(&list_lock);
        }
        ….
}
```

# Measure lock contention time

```
void * thread_worker(threadNum) {
        startIndex = threadNum * iteration;
        for (i = startIndex; i < startIndex + iteration; i++)
        {
                clock_gettime(CLOCK_MONOTONIC, &start_time);
                pthread_mutex_lock(&list_lock);
                clock_gettime(CLOCK_MONOTONIC, &end_time);

                SortedList_insert(listhead, pool[i]);
                pthread_mutex_unlock(&list_lock);

                //wait_time is an unsigned long array with length being number of worker threads.
                wait_time[threadNum] += calc_diff(&start_time, &end_time);
        }
        ....
}
```

# Addressing the Underlying Problem

- Fundamental problem
  - Throughput degrade is the result of **increased contention**.
- Classic solution
  - **partition the single resource** (in this case a linked list) into multiple independent resources
  - divide the requests among those sub-resources

# Part B: Break linked lists into sublists

# Part B requirement

- Add **--lists=#** option
  - **break the single sorted list** into the specified number of sub-lists
  - each with its own list header and synchronization object
- select which sub-list a particular key should be in
  - based on a simple hash of the key, modulo the number of lists
- obtain the length of the list
  - enumerating all of the sub-lists.
- each thread:
  - starts with a set of pre-allocated and **initialized elements** (--iterations=#)
  - **inserts** them all into the multi-list (which sublist the key should go into determined by a hash of the key)
  - gets the **list length**
  - **looks up** and **deletes** each of the keys it inserted
  - exits to re-join the parent thread

# Part B: overview

```
SortedListElement_t  listheads[], * pool;
pthread_mutex_t locks[];
void * thread_worker(threadNum) {
        startIndex = threadNum * iteration;
        for (i = startIndex; i < startIndex + iteration; i++) Mul_SortedList_insert(pool[i]);
        Mul_SortedList_length();
        for (i = startIndex; i < startIndex + iteration; i++) {
                e = Mul_SortedList_lookup(pool[i]->key);
                SortedList_delete(e);
        }
}
```

# Insert of multiple sublists

Note: below is pseudocode

```
SortedListElement_t  listheads[]; //one listhead for each sublist
pthread_mutex_t locks[]; //one lock for each sublist

void Mul_SortedList_insert(SortedListElement_t *element)
{
        int list_num = hashkey(element->key); //hashkey return between [0 , # of sublists-1]
        pthread_mutex_lock(&locks[list_num]);
        SortedList_Insert(&listheads[list_num], element);
        pthread_mutex_unlock(&locks[list_num]);
}
```

# Lookup of multiple sublists

Note: below is pseudocode

```
SortedListElement_t  listheads[];
pthread_mutex_t locks[];

SortedListElement_t * Mul_SortedList_ lookup(char * key)
{
        SortedListElement_t * ret = NULL;
        int list_num = hashkey(key); //determine which sublist the element of key belongs to
        pthread_mutex_lock(&locks[list_num]);
        ret = SortedList_lookup(&listheads[list_num], key); //get the element of the key
        pthread_mutex_unlock(&locks[list_num]);
        return ret;
}
```

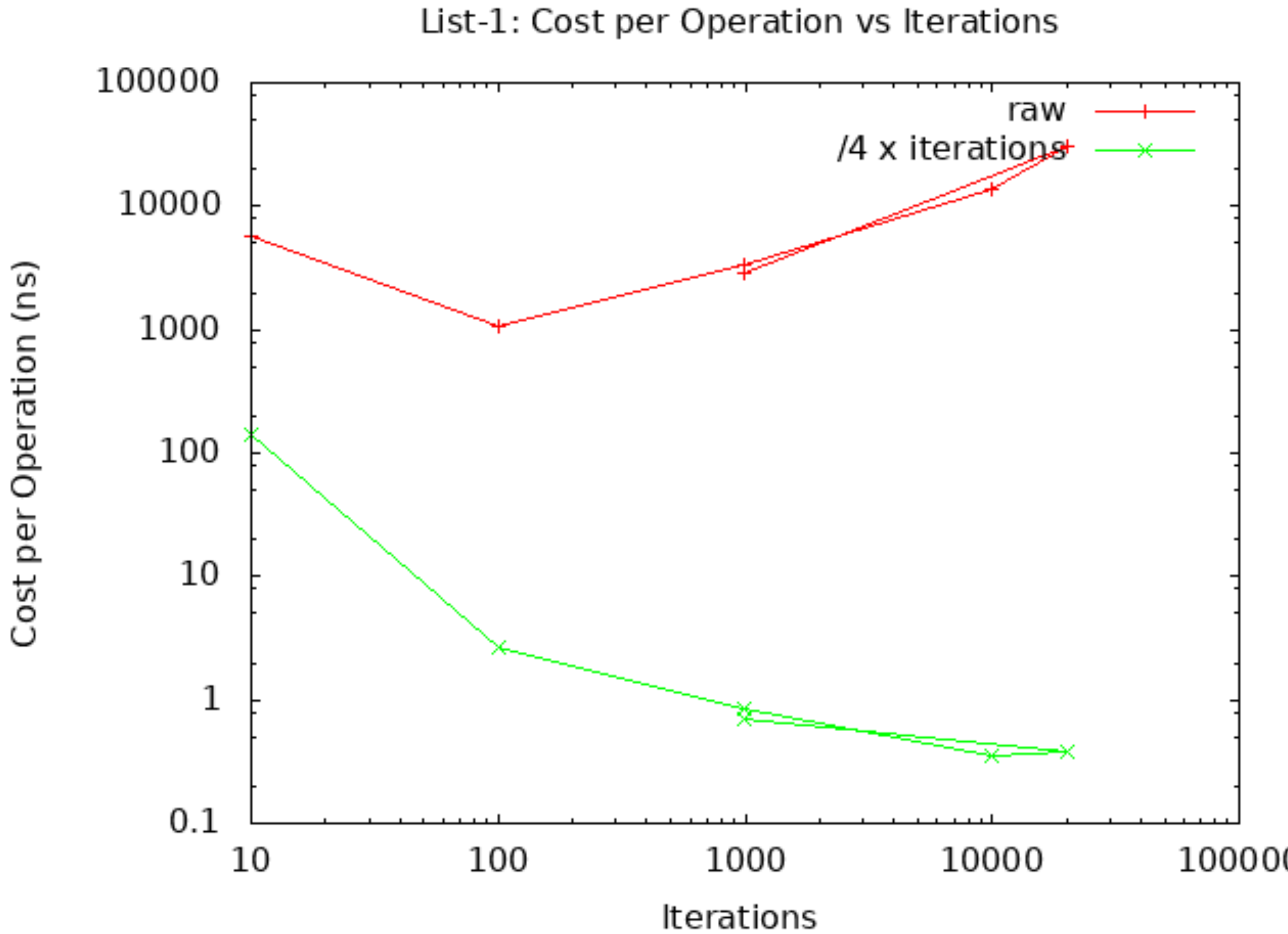
# Length of multiple sublists: naïve implementation

Note: below is pseudocode

```
SortedListElement_t  listheads[];
pthread_mutex_t locks[];

int Mul_SortedList_ length(void)
{
        int i = 0, length = 0;
        for (i = 0; i < num_sub_lists; i++)
        {
                pthread_mutex_lock(&locks[i]);
                length += SortedList_length(listheads[i]);
                pthread_mutex_unlock(&locks[i]);
        }
        return length;
}
```

# Drawing Figures with gnuplot

```
set terminal png
set datafile separator ","

set title "List-1: Cost per Operation vs Iterations"
set xlabel "Iterations"
set logscale x 10
set ylabel "Cost per Operation (ns)"
set logscale y 10
set output 'lab2_list-1.png'

plot \
    "< grep 'list-none-none,1,' lab2_list.csv" \
        using ($3):($7)  title 'raw' with \
        linespoints lc rgb 'red', \
    "< grep 'list-none-none,1,' lab2_list.csv" \
        using ($3):($7)/(4*($3)) \
        title '/4 x iterations' with linespoints lc rgb 'green'
```




```
list-none-none,1,10,1,30,139945,4664
list-none-none,1,100,1,300,165364,551
list-none-none,1,1000,1,3000,3928455,1309
list-none-none,1,10000,1,30000,495197933,16506
list-none-none,1,20000,1,60000,3054392526,50906
list-none-none,1,1000,1,3000,3879764,1293
```