VIRGINIA TECH

CS 3214: Project 2

Fork-Join Threadpool

Tuesday October 15th, 2024; 7:00pm EDT

Max Fisher<maxhfisher@vt.edu>
Forrest Meng<forrestm@vt.edu>

Topics

- Getting Started and Basics
- Threadpool Design
- Codebase Intro
- Logistics
  - Grading
  - Test Driver
  - Scoreboard
- Debugging
- Performance
- Advice
- Questions

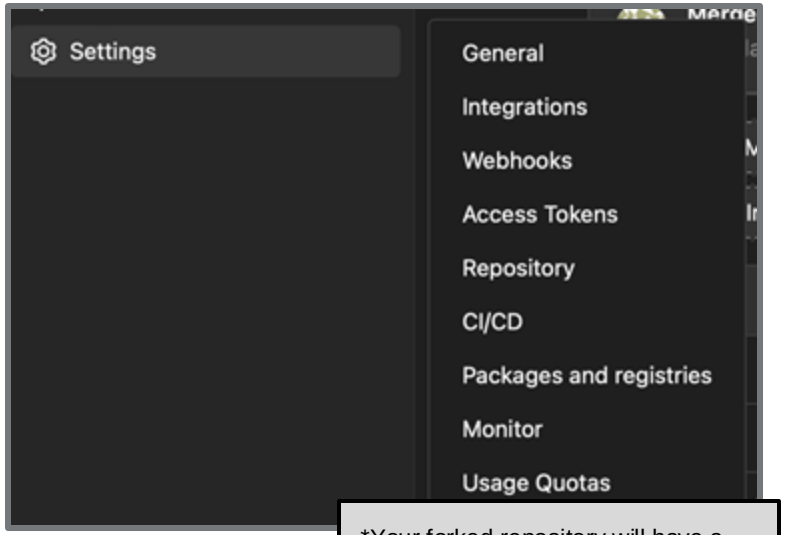
Getting Started and Basics

First Step!

1. One member will fork the base repository:
   https://git.cs.vt.edu/cs3214-staff/threadlab
1. Invite partner to collaborate
   - Go to Settings > Members to add them
   - Check partner role permissions too
1. Both members will clone the forked repository on their machines:

```
$ git clone <your git repo url>.git  →  threadlab
```

1. ***IMPORTANT:*** Set forked repository to **private**
   - Go to Settings > General > Visibility, project features, permissions
   - Potential Honor Code Violation if not set to private

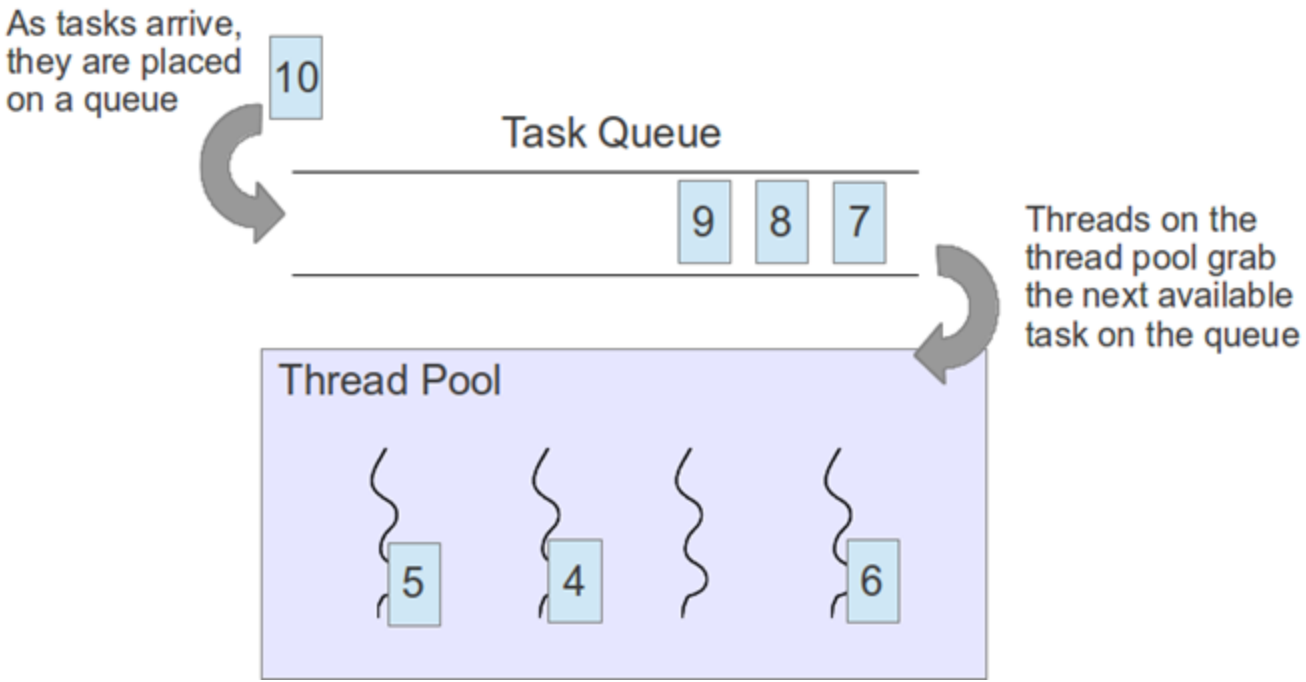


*Your forked repository will have a navigation menu on the left side. Click under Settings to add members and set repo to private

Some Basics

- Thread
  - A single sequential flow within a program
  - A single process can have multiple threads
- Threadpool - collection of idle threads that to do work for an external program
  - Doesn’t do anything unless work is given to it
  - Provides an API for external programs or clients to use without adding locking semantics
  - Less-headache way to add concurrency to programs

Basic Illustration



Source: https://www.classes.cs.uchicago.edu/archive/2016/spring/12300-1/pa3.html

Where you come in...

- You will create your own Threadpool API that external programs will call
- What do we write?
  1. `threadpool.c`
  2. Implementations for functions and structs from `threadpool.h`
  3. Static helper functions

Functions you will implement

```
struct thread_pool * thread_pool_new(int nthreads);
```

```
void thread_pool_shutdown_and_destroy(struct thread_pool *);
```

```
struct future * thread_pool_submit(struct thread_pool *pool, fork_join_task_t task, void *
data);
```

```
void * future_get(struct future *);
```

```
void future_free(struct future *);
```

- Read over threadpool.h for full documentation: you must implement these functions!
- Not included are static function(s) you'll add to threadpool.c

Threadpool Design

Threadpool Design

- Methodologies
  - Split up tasks among n workers
  - Work Sharing / Work Stealing
  - Internal and External Submissions
  - Work Helping
- **No global variables!** (exception of thread-local variables - we will talk about these later)

Work Sharing

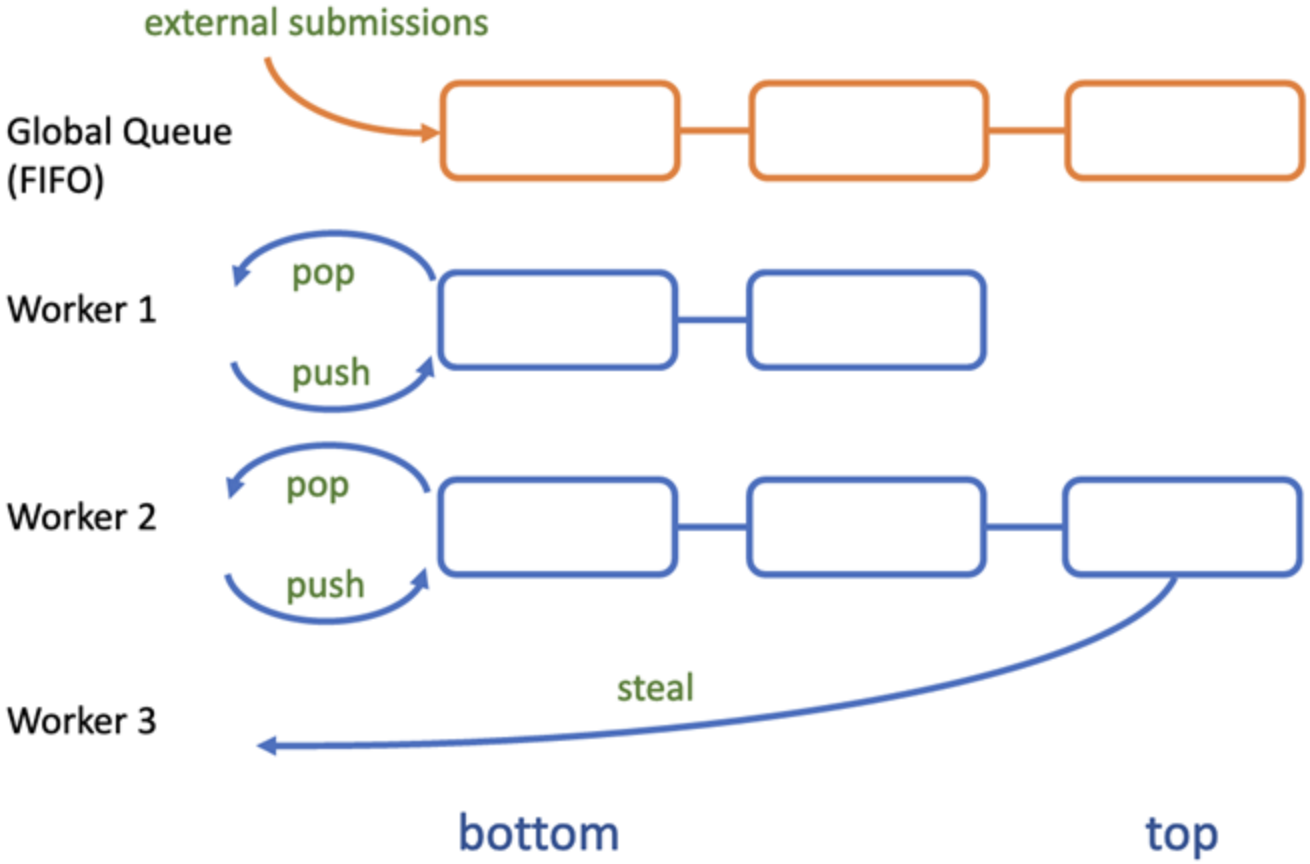
- Single, central queue from which all threads remove tasks
- Drawback: queue can become a point of contention especially with handling small tasks

Work Stealing

- Global list of tasks
- Local lists of tasks for each worker
- Worker main loop:
  - Do I have tasks? Pop from front (LIFO)
  - Are there global tasks? Pop from back (FIFO)
  - Does anyone else have tasks? Pop from back (FIFO)

Work Stealing (cont.)

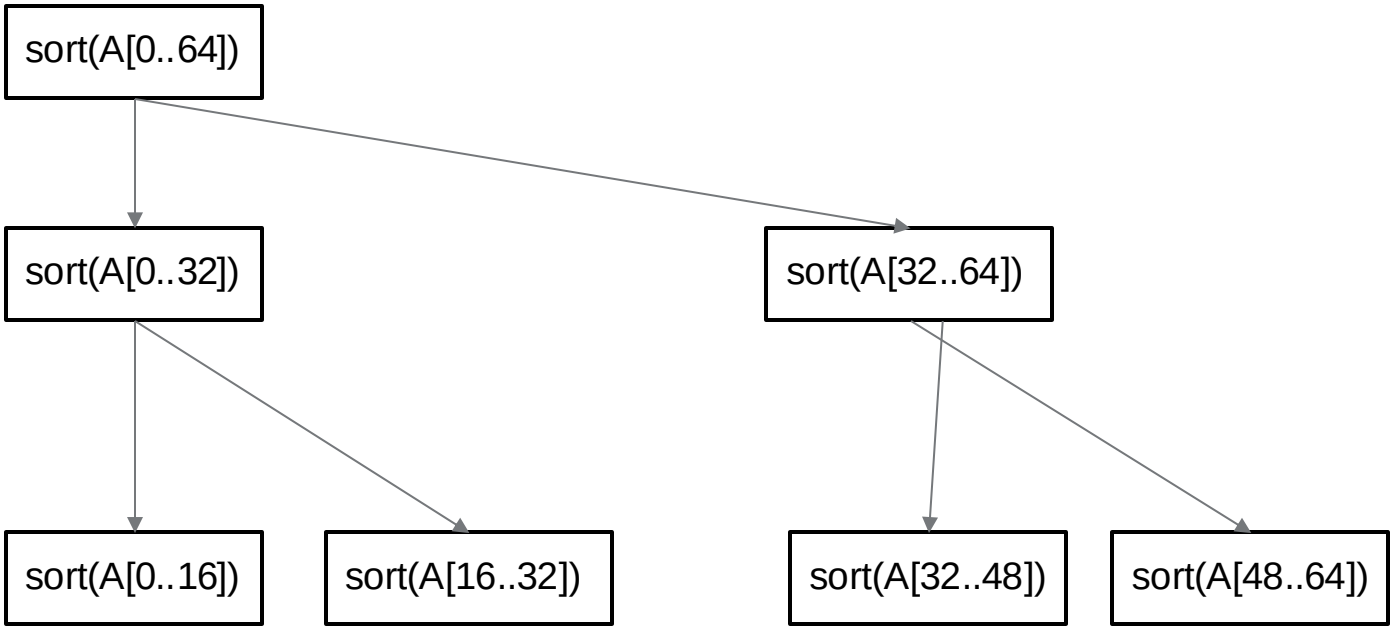
- Stealing spreads work evenly to idle threads
- Each queue/deque needs to be protected
- Workers still wait for other threads to steal and finish futures they depend on (we’ll get back to this)


external submissions
Global Queue
(FIFO)
Worker 1
pop
push
Worker 2
pop
push
steal
Worker 3
bottom
top

Internal vs External Task Submissions

- External Submission - client submits a new task to threadpool
  - Task gets added to the global queue
- Internal Submission - thread submits a subtask
  - “Subtask” gets added to worker’s local deque
    - Worker executes it later
    - Or a co-worker steals the task to execute itself
- For submissions to the threadpool, you’ll need to distinguish these cases
  - But how?

Mergesort


sort(A[0..64])
sort(A[0..32])
sort(A[32..64])
sort(A[0..16])
sort(A[16..32])
sort(A[32..48])
sort(A[48..64])

Example

* Check out mergesort.c to see full functions

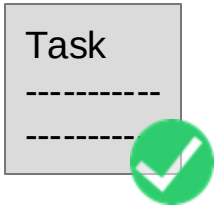
```
mergesort_parallel(int *array, int N) {
    int * tmp = malloc(sizeof(int) * (N));
    struct msort_task root = {
        .left = 0, .right = N-1, .array = array, .tmp = tmp
    };

    struct thread_pool * threadpool = thread_pool_new(nthreads);
    //EXTERNAL submission from client
    struct future * top = thread_pool_submit(threadpool,         //internal function
                                             (fork_join_task_t) mergesort_internal_parallel,
                                             &root);
    //demands answer once it's ready
    future_get(top);
    future_free(top);
    thread_pool_shutdown_and_destroy(threadpool);
    free (tmp);
}
```

Example (part 2)

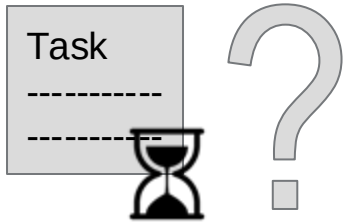
```
static void
mergesort_internal_parallel(struct thread_pool * threadpool, struct msort_task * s)
{             //If array small, no more submitting just internal sort (BASE CASE)
              if (right - left <= min_task_size) { mergesort_internal(array, tmp+left, left, right); }

                        ... not all code shown ....

        //INTERNAL Submission from the worker thread
        struct future * lhalf = thread_pool_submit(threadpool, (fork_join_task_t) mergesort_internal_parallel,
                                  &mleft);

              //Worker thread works on other half
        mergesort_internal_parallel(threadpool, &mright);
        future_get(lhalf);
        future_free(lhalf);
        merge(array, tmp, left, left, m, right);
    }
}
```

Work Helping

- In `future_get`, you can only return the result once the future is done executing
- The task might not be completed when `future_get` is called (or even running)
- Consider cases for getting the result from future_get:
  - If future already executed -> Hurray!



  - But what happens if the future isn't ready?
    - What should the thread do while waiting?



Work Helping (cont.)

- Threads should make best use of their time by…
  - Minimizing sleeping
  - Maximizing time spent executing tasks
- If no threads are executing a task you depend on, do it yourself
  - Workers can then recursively handle their own dequeue (FIFO)
- If a thread is executing the task you depend on? May be beneficial to execute other tasks instead of waiting

Design Advice

- You are asked to implement the work stealing with work helping thread pool design
  - Better load balancing
  - Lower synchronization requirements
- However, you can implement work sharing for only 80% credit *(not recommended)*

> In this assignment, you are asked to implement a work stealing thread pool. Since work stealing is purely a performance optimization, you may for reduced credit (corresponding to a B letter grade) implement a work sharing approach.

Thread Local Variables

- Want to be able to access your workers deque (and probably locks) during **`thread_pool_submit()`**
- How can we distinguish external/internal submissions?

Thread Local Variables

- Naive approach would be to loop through workers and check **`pthread_self()`**,...
- Instead, use some variable which would be different for each thread
  - AKA thread-local variables/storage

```
/*thread-local worker struct.*/
static _Thread_local struct worker * current_worker;
```

Implementation Tips

struct thread_pool

- Should contain any state you need for a threadpool
- Ideas:
  - Locks (`pthread_mutex_t`)
    - To protect the global queue
  - Queues/Deques (provided `list` struct from previous project)
  - Semaphores (`sem_t`)
  - Conditional Variables (`pthread_cond_t`)
  - Shutdown flag
  - List of workers associated with this thread_pool
  - Etc.

struct worker

- Should contain a worker struct as well
- Ideas:
  - Maintain which pool this worker is for
  - Queue of internal submissions
  - Lock for local queue
  - etc…

Futures

- How do we represent a task we need to do?
  - `future`
  - Threadpool: an instance of a task that you must execute
  - Client: a promise we will give them a reply when they ask for it

```
struct future
{
        fork_join_task_t task; // typedef of a function pointer type that you will execute
        void* args;                         // the data from thread_pool_submit
        void* result;                       // will store task result once it completes
execution
                …
        // may also need synchronization primitives (mutexes, semaphores, etc)
};
```

Futures (cont.)

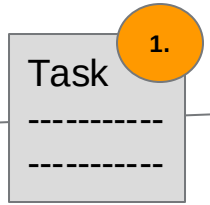
- You will invoke “task” as a method, it represents the method passed through by **`thread_pool_submit`**, the return value gets stored into the result

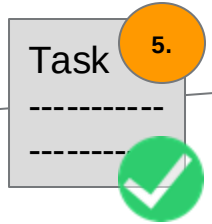
```
fut->result = fut->task(pool, fut->data);
```

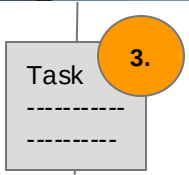
Future Illustration

KEY:

1. Client submits task to threadpool's global queue
1. Client immediately receives a future
1. A worker thread snags the submitted task to their local deque for work
1. Client demands to get the completed task affiliated with future
1. The completed task is returned, client frees the future







Functions you will implement

```
struct thread_pool * thread_pool_new(int nthreads);
```

```
void thread_pool_shutdown_and_destroy(struct thread_pool *);
```

```
struct future * thread_pool_submit(struct thread_pool *pool, fork_join_task_t task, void *
data);
```

```
void * future_get(struct future *);
```

```
void future_free(struct future *);
```

- Read over threadpool.h for full documentation: you must implement these functions!
- You can also add static function(s) to threadpool.c

thread_pool_new

- Create thread pool
- Initialize worker threads
- Call pthread_create: starts a new thread in the calling process. The new thread starts execution by invoking start_routine(); arg is passed as the argument of start_routine()

```
#include <pthread.h>

int pthread_create(pthread_t *restrict thread,
                   const pthread_attr_t *restrict attr,
                   void *(*start_routine)(void *),
                   void *restrict arg);
```

VIRGINIA TECH

Logistics

Grading

- Submit code that compiles
- Test using the driver before submitting
- When grading, tests will be run 3-5 times, if you crash a single time it’s considered failing
- Benchmarked times will be the average of the 3-5 runs, assuming you pass all of them

Grading (cont.)

- Breakdown
  - Git Usage
  - Functionality Tests (Basic/Advanced ~ 25% each)
  - Performance ~ 40%
- GTAs will determine the exact breakdown of points
- Performance will breakdown into rough categories based on real time scores
- You must pass the basic tests before getting anything for performance

Performance

- Relative to peers and sample implementations
- Points only for the tests on the scoreboard
  - N Queens, Mergesort, Quicksort (8, 16, and 32 threads), possibly Fibonacci
- A rough cutoff for real time benchmarks will be posted later on by Dr. Back (last semester's scoreboard)

Performance

Based on the scoreboard and my own implementation(s), we'll be using the following cutoffs (in seconds):

Test	Good	Mediocre	Lacking	Serial (estimated)
MSL/8	<12	12-30	>30	48
MSL/16	<8	8-30	>30	48
MSL/32	<6.5	6.5-30	>30	48
QS/8	<9	9-30	>30	39
QS/16	<6.5	6.5-30	>30	39
QS/32	<5.5	5.5-30	>30	39
NQ/32	<7	7-60	>60	127
Fib/32	<4	4-20	>20	21
Simp/32	<2	2-20	>20	27
Kenken/32	<2	2-20	>20	25

Note that "Good" performance for all but Fib/32 and perhaps NQ/32 is achievable even with a single lock implementation.

ONLY for reference - numbers will likely change

Performance

Test name:	1	2	4	8	16	32
BASIC1: Basic functionality testing (1)						
basic test 1	[X]	[X]	[X]			
BASIC2: Basic functionality testing (2)						
basic test 2	[X]	[X]	[X]			
BASIC3: Basic functionality testing (3)						
basic test 3	[X]	[X]	[X]			
BASIC4: Basic functionality testing (4)						
basic test 4		[X]	[X]			
BASIC5: Basic functionality testing (5)						
basic test 5		[X]	[X]			
BASIC6: Basic functionality testing (6)						
basic test 6	[X]					
MERGESORT: parallel mergesort						
mergesort small	[X]	[X]	[X]	[X]	[X]	
mergesort medium	[X]	[X]	[X]	[X]	[X]	
mergesort large				[8.162s]	[5.764s]	[4.609s]
QUICKSORT: parallel quicksort						
quicksort small	[X]	[X]	[X]	[X]	[X]	
quicksort medium	[X]	[X]	[X]	[X]	[X]	
quicksort large				[8.884s]	[5.059s]	[4.291s]
PSUM: parallel sum using divide-and-conquer						
psum_test small	[X]	[X]	[X]	[X]	[X]	
psum_test medium	[X]	[X]	[X]	[X]	[X]	
psum_test large				[X]	[X]	[X]
NQUEENS: parallel n-queens solver						
nqueens 11	[X]	[X]	[X]	[X]	[X]	
nqueens 12	[X]	[X]	[X]	[X]	[X]	
nqueens 13				[X]	[X]	[X]
nqueens 14					[9.114s]	[6.659s]
FIBONACCI: parallel fibonacci toy test						
fibonacci 32	[X]	[X]	[X]	[X]	[X]	
fibonacci 41					[X]	[X]

Visual Studio Code Terminal Issues

- **Use a separate terminal (like git bash) to run the tests**
- VS Code spins up some extra processes on rlogin to manage files, they interfere with the somewhat strict thread limits we enforce on the tests to guarantee your thread pool isn't creating additional workers to juice performance numbers

Test Driver

```
$ ~cs3214/bin/fjdriver.py [options]
```

- Can take a long time to run all tests
- Reports if you passed each test, and times for the benchmarked ones

```
10 klalitha@pawpaw in ~/CS3214/threadlab/tests>fjdriver.py -h

Usage: /home/courses/cs3214/bin/fjdriver.py [options]
    -v              Verbose
    -V              Print Version and exit
    -a              Run benchmark anyway even if machine is not idle
    -r              Only run required tests.
    -h              Show help
    -p              <file> - Location of threadpool implementation, default ./threadpool.c
    -l              List available tests
    -t              Filter test by name, given as a comma separated list.
                    e.g.: -t basic1,psum
```

Test Driver

- Make sure to run tests multiple times, race conditions can cause you to crash only 20% of the time
- Will run multiple times to ensure consistency when grading (and get a good average for times)
- All of the tests are C programs, compiled against your threadpool

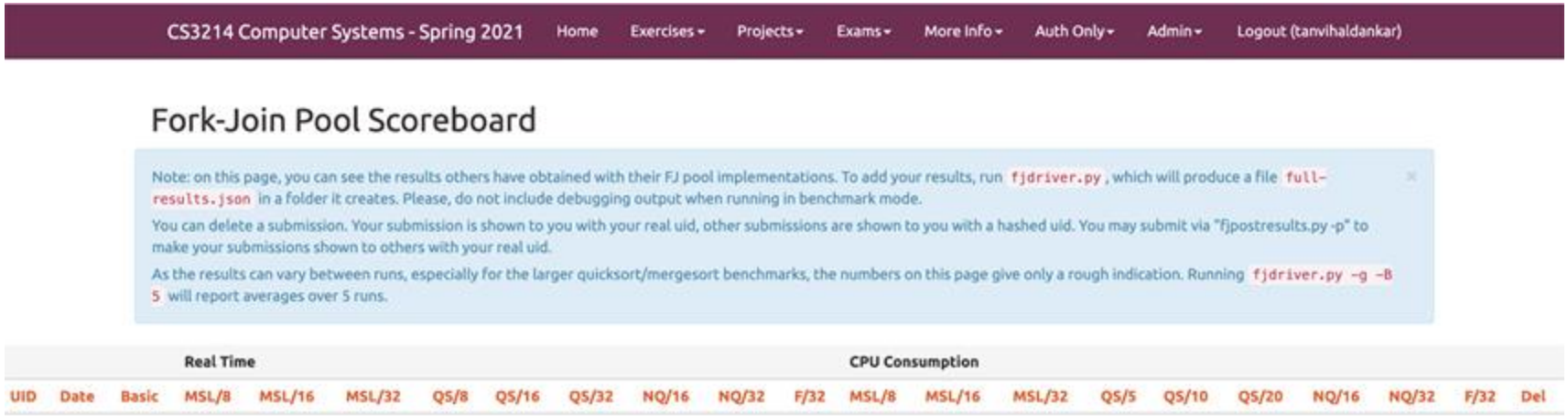
Test Driver

```
$ ~cs3214/bin/fjdriver.py -g -B 5
```

- Runs the tests 5 times and averages the results
- Helpful to simulate grading environment

Scoreboard

- https://courses.cs.vt.edu/cs3214/fall2024/projects/project2scoreboard
- You can post your results to the scoreboard by using the fjpostresults.py script



Debugging Tools

Debugging

- Debugging multi-threaded programs can be difficult
  - **Don’t just use printf()**
- This project will challenge you in your debugging skills (GDB, Helgrind..)
- Helgrind**
  - Valgrind tool
  - Enable using **`--tool=helgrind`** in Valgrind command line
  - Your best friend for tracing deadlocks and synchronization errors
  - https://www.valgrind.org/docs/manual/hg-manual.html/

GDB Demo

- `info thread` - see how many threads there are
- `thread <thread_num>` - switch current thread
- `thread apply all bt` - see what each thread is doing
- Checking who owns a lock

Improving Performance

- Make sure you aren't on a busy rlogin node!
  - `ssh <username>@portal.cs.vt.edu`
- Minimize sleeping, maximize execution of tasks
- "... We recommend that you intentionally break the rule of signaling with the lock held"
- Advanced Optimizations - CPU Pinning, Fixing False Sharing, Lockless Queues
- CPU profiling – htop, perf
- Ask on Discourse! There's a lot of other optimizations to try

General Advice

- Start Early (...now)
- How many lines of code?
  - ~250-350 lines (not a good benchmark for difficulty)
- Most of time is spent debugging
  - GDB, Helgrind, and Valgrind are your friends
  - Debugging multi-threaded programs is difficult and time consuming
- Try different strategies
  - Most of the learning is trying out different approaches - telling you exactly what would give the best results would reduce the educational experience

VIRGINIA TECH

Any Questions?

VIRGINIA TECH

Good Luck!