**Student Information:**

Jeff Chen (chenjeff4840)

Jackson Burkey (jacksonb01)

**Description of Base Functionality:**

* struct thread\_pool\* thread\_pool\_new(int nthreads)
  + Creates a dynamically allocated threadpool. Initializes it then returns it.
* void thread\_pool\_shutdown\_and\_destroy(struct thread\_pool\* pool)
  + Sets shutdown flag on. Broadcast to the threads using the same pathway used for signaling a new task (threads will read shutdown flag first before any other actions). Afterwards, destroy global queue. Then wait for all worker threads to shutdown. Then clean worker thread dynamically allocated data and remaining fields of the threadpool.
* void future\_free(struct future\* fut)
  + Destroy and clean up fields from fut.
* void\* future\_get(struct future\* fut)
  + Check if is worker thread. If is worker thread, check if fut is queued. If queued, remove it from the queue it is in, update status, work on the queue, update its status again, and return result.
    - If worker thread and not queued or is not worker thread, if fut is not complete, wait until it is completed then return result.
* struct future\* thread\_pool\_submit(struct thread\_pool\* pool, fork\_join\_task\_t task, void\* data)
  + Initializes an empty future. Set status to queued. If task is internal (called by worker thread), submit to the front of worker’s queue. If external, submit to back of global queue. Update current queue field for the future. Notify threads that a task is available then return future.

**Description of Private Functions:**

* static void worker\_init(struct worker\* w)
  + Initialize a worker struct. No threads are created,
* static void thread\_pool\_init(struct thread\_pool\* pool, int nthreads)
  + Initialize threadpool with nthreads. All fields such as the pool’s worker and worker threads are initialized here.
* static void future\_init(struct future\* fut, fork\_join\_task\_t, void\* args, struct thread\_pool\* pool)
  + Construct a future using the provided arguments. Arguments not provided such as current queue, state, and result are set to default values.
* static struct future\* get\_future\_front(struct list\* queue)
  + Returns list element in front of queue as a future. Remove the list element from the queue.
* static struct future\* get\_future\_back(struct list\* queue)
  + Same as previous function but back of queue
* static void queue\_clean\_all(struct list\* queue)
  + Removes and frees all list elements within queue.
* static int search\_all\_queues(struct thread\_pool\* pool)
  + Searches all queues within pool for an available task. Checks worker’s own queue first, then global queue, then from queues of other workers. If a task is found, the queue id it is found in is returned. If not found, an impossible queue id is returned.
* static void run\_task(struct thread\_pool\* pool, int q)
  + Looks at queue id q within q and grabs a task. If q matches worker thread’s id, it will grab task from the front of the worker’s queue. If in any other queue, it will grab task from the back of the queues. The task is retrieved as a future. The future status is updated, ran, then updated again with run results,
* static void\* worker\_thread\_job(void\* args)
  + Worker receives data from args which is converted into poolID struct containing thread pool and worker\_id assigned to the worker. After initializing worker’s \_Thread\_local id, it will run in a loop until shut down flag is triggered. Immediately upon entering the loop, the worker thread waits for thread\_pool\_submit to be triggered. Once triggered, the worker will check if shutdown flag has been triggered. If triggered, it will exit the loop. If not, it will enter another loop where it will continuously search for a task within any of the queues then run them. When no more tasks exists, it will exit both loops. After exiting, the worker thread cleans up its worker\_id struct and its data then exits.
* static bool remove\_from\_queue(struct list\* queue, struct future\* target)
  + Searches and removes target from queue starting from the front of the queue to the back of the queue. Only 1 list element matching target can be removed. Returns true if removed, false if not removed.

**Structs:**

/\*\*

\* Used as worker threads.

\*/

struct worker

{

struct list local\_queue; // Queue held locally by worker thread

pthread\_mutex\_t local\_queue\_lock; // Lock for local queue

pthread\_t thread; // Worker's thread (may change later on)

};

/\*

\* Opaque forward declarations. The actual definitions of these

\* types will be local to your threadpool.c implementation.

\*

\* Has all kinds of locks, however, is necessary since there is a condition

\* and having a universal lock for threadpool and waiting on queued\_cond is bad news

\*/

struct thread\_pool

{

uintptr\_t id; // ID of worker

struct worker \*workers; // Worker threads working for threadpool, unknown number of workers

// Worker lock might not be needed as only accessed for queue and queue has own lock

struct list global\_queue; // Global queue accessed by all workers

pthread\_mutex\_t global\_queue\_lock; // Lock for global queue

pthread\_cond\_t queued\_cond; // Conditional to wake workers up when new task is queued

pthread\_mutex\_t queued\_lock; // Lock that goes together with queued conditional

int worker\_count; // Number of workers

pthread\_mutex\_t worker\_lock; // Lock for worker arr, needed since threads are started while some threads are active

bool shutdown; // Shutdown flag, true to shut down, false to not

sem\_t t\_continue; // To allow broadcast to go through

};

/\*\*

\* Worker ID, used as \_Thread\_local

\*/

struct workerID

{

uintptr\_t id; // ID of worker

};

struct future

{

struct list\_elem elem; // List tag element

int curr\_queue; // Current queue future is in, -1 for global queue, <-1 for not set

fork\_join\_task\_t task; // Task future will execute

void \*args; // arguments for task

void \*result; // Results from running task

int state; // Current state of task, 0 = queued, 1 = in progress, 2 = finished, -1 for unset

struct thread\_pool \*pool; // Threadpool instance that contains/contained future

};

/\*

\* Submit a fork join task to the thread pool and return a

\* future. The returned future can be used in future\_get()

\* to obtain the result.

\* 'pool' - the pool to which to submit

\* 'task' - the task to be submitted.

\* 'data' - data to be passed to the task's function

\*

\* Returns a future representing this computation.

\*/

struct future \*thread\_pool\_submit(

struct thread\_pool \*pool,

fork\_join\_task\_t task,

void \*data);

**Globals:**

static \_Thread\_local struct workerID \*worker\_id; // External threads == NULL, Worker threads

// ID corresponds to worker placement in struct

// thread\_pool's worker\* workers.