What structures you used to implement the TCB, mutex, runqueues, any other queues or structures, etc.

For TCB, we just used a struct. It wasn't anything too crazy. The TCB contained essential information for a thread such as its stack, state, context, and other metadata needed to perform specific operations. For the runqueue we used a linked list, when we probably should have used a queue. A linked list works fine though, because we can just use a simple function to find the end. When we need to change the head of the list, we just set the current head equal to current head->next, and then move the current head to the end of the list. Another key point of the runqueue is we always make the head the current thread that is running. This allows for us to always know where the currently running thread is and smooth/consistent linked list manipulation. about Same functionality with blockedqueue and mutex system.

A key point is that we allot two separate linked lists for threads waiting on mutex locks and threads calling join (blocked mutex queue and blocked join queue). This allowed for us to clearly divide the purpose and current operation of each thread while also physically dividing the code for easier readability. That is, we decided to implement all basic thread management operations first before implementing mutexes. When implementing the mutexes, using a separate linked list proved advantageous because all we needed to do was add an extra check in the scheduler. For mutexes, we also used a struct in the mutex_types.h file, which allowed for us to store essential data such as the current mutex state, the TCB of the current owner of the lock, and the actual mutex that the structure belongs to.

What logic you used for implementing the thread and mutex API functions:

For threading logic, what we did was create a runqueue and blockingqueue. Runqueue is a linked list of threads in an order that they need to run in. Blockedqueue is a linked list of threads that are currently blocked because they are waiting for a different thread to finish. Another linked list exists for the mutex system, the 3rd linkedlist is for threads that are blocked for mutex related reasons. Atomic operations are used in the mutex api system, specifically test and set. These functions are used to set a variable to 0 or 1 based on if the mutexed value is in use or not. The following explains logic for implementing the thread and mutex API functions:

Threading

- Worker_create
  - A new context, TCB with that context, and node with that TCB as the data were allocated and designated with initial values.
  - The context was assigned the passed in function and a new stack, and the node was added to the runqueue in the READY state.
  - For the first time a thread is created, the scheduler context was created which runs the sched_rr() function. This context is stored globally for easy access.

- For the first time a thread is initialized the timer is also set up as well, with the timer only being a 1 shot timer requiring manual reset. We decided to resort to manually resetting the timer to ensure that the timer would never go off during an unwanted operation (during sched_rr() for example).

- Worker_yield
  - The timer is manually set and the context is switched to the scheduler context so the current running head is switched out of execution.
- Worker_exit
  - We set the currently running thread's (head of runqueue) state to DONE.
  - If the valueptr parameter is provided we go through the blocked linked list for join, look for the thread that is currently waiting for the existing thread, and store the valueptr in that thread's TCB. This will be useful for worker_join.
  - We finally manually reset the timer and switch to the scheduler context so the thread can be switched out, freed and deleted, and the next thread can run.
- Worker_join
  - We first look through the runqueue for the thread that the calling thread is waiting for, and store that thread's TCB in the calling thread's TCB (if this thread isn't found we simply return, because what you tryna wait for?)
  - After this, a new TCB and node are created for the calling thread that is populated with the calling context and other default values with one of them being the state = BLOCKED_JOIN (we decided to create this node in worker_join to avoid any unnecessary linked list manipulation)
  - After this, we swap to the scheduler context using swapcontext() so that the calling context's current state is saved in its TCB.
  - This is key since once the calling thread executes again, it will execute at the statement after the swapcontext, which is when we free that node and assign the valueptr if necessary.

Mutex

- Worker_mutex_init
  - This initializes default values for the mutex structure such as initial state to 0, owner to null, and the mutex in question to the mutex parameter
- Worker_mutex_lock
  - This function utilizes the test-and-set atomic hardware instruction to check if the lock has been taken.
  - We apply test-and-set and check the return value. If the return value is 0, we know that the lock has not been taken yet so the currently running thread can access the lock. As a result, the mutex's owner in its structure is set to the currently running thread and the scheduler context is invoked.
  - If the return value is 1, that means that the calling thread cannot access the lock (since the original value is returned in test-and-set, and the thread wants to see is 0). As a result, the calling thread is added to the blocked mutex queue, its state is changed to BLOCKED_MUTEX, and the scheduler is summoned.
- Worker_mutex_unlock

- ➢ The owner of the mutex is reset to NULL, the state is reset to 0, and swapcontext is used to switch to the scheduler context. We want to save the context of the calling thread so it can resume at the statement right at the end of the function.
- ➢ All of the threads that were waiting on that lock are also moved back to the runqueue so they all have a chance at grabbing the lock.

❖ Worker_mutex_destroy

- ➢ All of the threads that were waiting on the lock in question are moved from the blocked mutex queue to the runqueue, since there is nothing to wait on anymore.

What logic you used for implementing the scheduler(s). (How it keeps track of which threads are in what states, how it handles transitions between states, how it chooses which thread to pick next, etc.)

We use an enum to figure out what state a thread is in. We used a signal handler to trigger when a timer interval goes off, a signal goes off. A function is run one time to set up the scheduler context and timer. It uses round robin functionality to pick what thread to run next, and skips over blocked threads (because they are in the blockedqueue not the runqueue). We used a timer that requires manual reset to trigger the scheduler. That is, every time quantum, the schedule() signal handler is triggered, and the scheduler context is summoned (using setcontext() so the function starts from the beginning).

Once the scheduler is triggered, the first thing that is done is a check in all of the blocked queues. This check relates to all of the nodes that are waiting for a thread to finish (i.e. calling join). The mutex and join queues are iterated through, and if that thread that the calling thread is found, then it remains in the blocked queue. However, if the thread is not found, that means that that thread has finished executing, so the calling thread can be moved to the runqueue.

After this check, a check is done if the current thread's state is DONE, and if it is, that node is reaped essentially. After all of this, round robin is performed. That is, the currently executing thread is put in the READY state, that thread is moved to the back of the runqueue, and the new head is then put into the RUNNING state, and it is that thread's time to shine (for now…).

Collaboration and References: State clearly all people and external resources (including on the internet) that you consulted. What was the nature of your collaboration or usage of these resources?

Sanjay Kethineni (sk2425), Adhit Thakur (at1186) - Partners on this project
Stackoverflow - various resources and references for syntax
Valgrind - for finding memory related issues
GDB - for debugging