User-level Thread Library and Scheduler

Akshith Dandemraju (Acd218), Abhinav Acharya (Aa2372)

March 3, 2024

1 Structures Implemented

In our Thread Control Block (TCB) structure, we had:

- Thread ID
- Thread context
- Thread status
- Thread stack
- Thread return value

We also implemented a `Node` structure, which was used in the scheduler and mutex blocked queue. Each `Node` has:

- TCB
- Next pointer
- Queue type

We utilized two enum structures:

- `QueueType`: `QUEUE_TYPE_READY`, `QUEUE_TYPE_MUTEX_BLOCK`
- `ThreadState`: `READY`, `RUNNING`, `BLOCKED`, `TERMINATED`

The scheduler structure has:

- Ready queue head
- Ready queue tail
- Context

Additionally, we implemented an array list used to map thread IDs to `Nodes`. It consists of:

- Array of `Nodes`
- Usage status
- Size

Our mutex structure includes:

- Volatile integer `lock`
- Owner thread
- Waitqueue head
- Waitqueue tail

2 Logic for API Functions

2.1 Thread API Functions

2.1.1 `worker_create()`

The `worker_create()` function is responsible for creating a new worker thread. It initializes the thread control block (TCB) with the required parameters and allocates resources for the thread. Once created, the thread is added to the scheduler's ready queue for execution.

2.1.2 `worker_yield()`

The `worker_yield()` function allows a thread to voluntarily relinquish the CPU and return to the scheduler. It changes the state of the current thread to `READY` and selects the next thread from the ready queue for execution.

2.1.3 `worker_exit()`

The `worker_exit()` function terminates the execution of the current thread. It updates the thread's state to `TERMINATED`.

2.1.4 `worker_join()`

The `worker_join()` function allows a calling thread to wait for the termination of a specific worker thread. It employs a while loop implementation to continuously check the state of the target thread every time its scheduled. Once the target thread terminates, the joining thread resumes execution.

2.2 Mutex API Functions

2.2.1 `mutex_init()`

The `mutex_init()` function initializes a mutex object. It allocates memory for the mutex structure and initializes its internal state, such as the lock flag and wait queue.

2.2.2 `mutex_lock()`

The `mutex_lock()` function is used to acquire a lock on the mutex. If the mutex is already locked by another thread, the calling thread is blocked and added to the mutex's wait queue. Once the mutex becomes available, the calling thread acquires the lock and proceeds with its execution.

2.2.3 `mutex_unlock()`

The `mutex_unlock()` function releases the lock on the mutex. It updates the mutex's lock flag and notifies any waiting threads in the mutex's wait queue. If there are no waiting threads, the mutex remains unlocked.

2.2.4 `mutex_destroy()`

The `mutex_destroy()` function deallocates the resources associated with the mutex object. It frees the memory allocated for the mutex structure and performs any cleanup operations necessary.

3 Logic for Scheduler Implementation

The scheduler keeps track of thread states, handles transitions between states, and selects the next thread using the following approach:

- **Scheduler Context**: Every time a node is scheduled, the scheduler context is reset back to the top of the schedule function. This ensures that the scheduler is ready to select the next thread for execution.

- **Tracking Thread States**: The scheduler maintains a queue of threads and keeps track of their states. Each thread can be in one of the following states: READY, RUNNING, BLOCKED, or TERMINATED.
- **Handling State Transitions**: Transitions between thread states are managed through context swaps. When a thread is scheduled to run, its state transitions from READY to RUNNING. When a thread is blocked, its state transitions from RUNNING to BLOCKED. Upon termination, the state transitions to TERMINATED.
- **Selecting Next Thread**: The scheduler employs a round-robin algorithm to select the next thread for execution. This ensures fairness in thread scheduling, as each thread gets an equal share of CPU time. Whenever the timer goes off, the context switches from the currently running thread to the scheduler, which then selects the next thread from the ready queue.

4 Collaboration and References

None