CS416 - User Level Thread Library

Paul John (pj242), Marko Elez (me470)

March 2021

1 Introduction

This document explains our implementation of a purely user-level thread library that closely resembles the POSIX thread (pthread) library interface.

2 API

2.1 Methods: Thread

This section documents the implementation of every method in the thread library along with that of the scheduler.

- `rpthread_create`

  Every time this method is called, a new thread control block is initialized and is added to the top most run queue in the scheduler.

  The very first time this method is called, the thread library also creates thread control blocks for the scheduler and main context, starts a periodic timer interrupt, and adds the thread control block for the main context to the run queue.

- `rpthread_yield`

  Every time this method is called, the thread currently running will save its context and swap into the scheduler context.

- `rpthread_exit`

  Every time this method is called, the thread currently running will deallocate its stack and change its status to 'Finished'. Additionally, before switching into the scheduler context, any thread waiting for this thread's completion is added to the run queue.

- `rpthread_join`

  Every time this method is called, the status of the thread we are joining is checked. If the status is 'Finished', this method will hand back the return value of the thread we are joining immediately to the current thread.

  Otherwise, the method will change the current thread's status to 'Blocked' and set a parameter in the control block of the thread we are joining to indicate that the current thread is waiting for it to complete. Then, the method will switch into the scheduler context.

- `rpthread_mutex_init`

  Every time this method is called, nothing occurs since the parameters in the mutex struct are initialized by default.

- `rpthread_mutex_lock`

  Every time this method is called, the current thread will try to set the atomic flag (indicating if this particular mutex is in use) to true. If the current thread is successful in setting the flag, the method will return immediately after indicating that the current thread owns the instance of the mutex.

  Otherwise, the thread currently running will get added to the blocked queue and its status will get changed to 'Blocked'.

- `rpthread_mutex_unlock`

  Every time the method is called, the atomic flag will get cleared and all the threads in the blocked queue will get added to the run queue. After clearing the blocked queue, the method will immediately return.

- `rpthread_mutex_destroy`

  Every time the method is called, the queue storing the blocked threads will be deallocated.

2.2 Methods: Scheduler

- `scheduler_round_robin`

  Every time the method is called, it will check if the thread currently executing is in a blocked/finished state. If the thread is in a blocked/finished state, the thread's control block will be removed from the run queue. Otherwise, the current thread (at the front of the run queue) will get moved to the back of the run queue.

  Finally, the next thread at the front of the run queue will subsequently get scheduled.

- `scheduler_multi_level_feedback_queue`

  Every time the method is called, it will check if the thread currently executing is in a blocked/finished state. If the thread is in a blocked/finished state, the thread's control block will be removed from the run queue. Otherwise, if the current thread yielded to the scheduler or is in the last queue, it will get moved (from the front of the run queue) to the back of the run queue. Lastly, any thread that did not fall into any those categories, will get demoted to the back of a lower run queue.

  Finally, the thread sitting at the front of the top most (non-empty) run queue will subsequently get scheduled.

2.3 Structures

This section details the different structures used in the thread library.

Struct 1: Thread Control Block

```
typedef struct threadControlBlock {
    uint id;                // Unique ID
    int status;             // State
    ucontext_t context;     // Context (stack, ...)
    void* retVal;           // Return Value
    uint joiningThread;     // ID of thread wanting to join
} tcb;
```

Struct 2: Mutex

```
typedef struct rpthread_mutex_t {
    std::atomic_flag flag; // Indicates whether this mutex is locked/freed
    Queue<uint> queue;     // Contains list of threads blocked on mutex
    uint owner = 0;        // Indicates which thread has locked mutex
} rpthread_mutex_t;
```

3 Performance

Thread Count	Linux pThread Performance (µs)			My pThread Performance (RR/MLFQ µs)		
	`vector_multiply`	`parallel_cal`	`external_cal`	`vector_multiply`	`parallel_cal`	`external_cal`
2	204	1484	4083	194/195	2857/2855	7334/7334
5	242	965	2970	195/196	2853/2857	7459/7342
10	332	763	2765	197/198	2858/2853	7347/7358
50	376	715	2724	304/311	2853/2857	7354/7351
100	394	716	2745	261/267	2854/2866	7364/7376

Table 1: pThread Library Performance Comparison

Analysis: The Linux pThread library vastly outperforms our pThread library implementation when run against the given workloads: `parallel_cal` and `external_cal`. Interestingly enough, the Linux pThread library seems to perform better as the thread count increases. On the other hand, our pThread library produces a relatively consistent running time even when the thread count increases. In regards to `vector_multiply` workload, our pThread library actually outperforms the Linux pThread library by a slight margin. Lastly, varying the type of scheduler we use Round Robin/Multi-Level Feedback Queue for the thread library does not seem to have an effect on the performance.