CS518: OS Design
Assignment 1: Thread Scheduling

Members

Roshan Patel	rrp78
Daniel Hidalgo	dh624
Saad Ismail	sni7

Run Instructions

Run the following set of commands:

```
> make
> ./my_pthread_t [NUM_THREADS]
```

Files Included

makefile => makefile to compile the assignment
my_pthread_t.c => contains our main and definitions for functions of our structs
my_pthread_t.h => header file for scheduler, `my_pthread_t`, `thread_unit` `thread_unit_list`
thread_unit_lib.c=> library of `thread_unit` helper functions
thread_unit_lib.h => header file for `thread_unit` and `thread_unit_list` helper functions

Overview

Our implementation of a multi-level thread scheduler manages user generated `my_pthread_t`'s and `my_pthread_mutex`'s. We initially designed a robust set of data structures and methods to handle various possible scheduler designs. Eventually, we converged on a design solution and trimmed down our data structures accordingly. The resulting implementation is fairly optimal and robust to large input.

We elected to protect our underlying thread metadata by storing it in a structure inaccessible to the user: a `struct thread_unit`. The `thread_units` contain a pointer to a `my_pthread_t` along with useful metadata that is used by our scheduler. Scheduling methods utilizes queues of `thread_units`. These queues manifest as linked lists of type `struct thread_unit_list`. Helper functions to enqueue, dequeue, peek, get size of, and print are in `thread_unit_lib.c`. The scheduler manages thread_units by storing them in various queues depending on if they are running or waiting to be scheduled to run.

Notable Data Structures

```
typedef struct thread_unit_ {

    my_pthread_t*               thread;
    ucontext_t*                 ucontext;
    state                       state;
    int                         time_slice;
    int                         run_count;
    int                         priority;
    long int                    joinedID;
    char                        stack[PAGE_SIZE];

    struct thread_unit_*        next;
    struct thread_unit_list_*   waiting_on_me;
    struct thread_unit_*        wait_next;
    struct thread_unit_*        mutex_next;

}thread_unit;
```

THREAD_UNIT

This is the struct that contains a pointer to the user generated my_pthread_t. It additionally stores a pointer to a ucontext and the stack of that context. This is used for manual context switching by our scheduler. The struct also contains various pointers to other structs to manage queues. Lastly, other useful information regarding the thread is stored in this struct. They are useful criteria for thread selection when scheduling.

SCHEDULER

The scheduler struct contains pointers to all the queues of importance. The priority_array is an array of queues. There is a queue per priority level. The running queue and currently_running thread_unit are also stored as as pointers here.

```
typedef struct scheduler_t {

    int                         initialized;
    long int                    threadID_count;
    struct thread_unit_list_*   priority_array[PRIORITY_LEVELS];
    struct mutex_node_*         mutex_list;
    struct thread_unit_*        currently_running;
    struct thread_unit_list_*   running;

}scheduler_t;
```

Scheduler Design

The scheduler struct stores all of the user generated pthreads in the union of two structures: the `priority_array` and the `running` queue. The `priority_array` is of the size of the number of priority levels. Each index of this array stores a list of thread_units that belong in that respective priority (index 3 = priority level 3 where 0 is most desirable priority). The running queue is a list of thread_units that are currently set to execute in sequence. Threads are allowed to run for their alloted `time_slice` or until they exit, join, or yield themselves. At this point, the next thread in the `running` queue will run.

When all threads in the running queue have run for their time_slice, the `maintenance_cycle` is invoked. The maintenance cycle will put all the thread_units in the running queue back into their respective spots in the priroity_array. Priority adjustments are made. Then, a new running queue is selected. The process then repeats. We have iteratively designed and tested different methods for priority adjusting and selecting new thread to enqueue into running.

1. Dequeue some from priority_array into running queue
2. Execute all threads in running queue
3. Dequeue all of running queue and put back into priority array
4. Priority Adjustments
5. Repeat

Design Implementation

Parameters

The following are the key parameters to our scheduler:

```
#define PRIORITY_LEVELS     15                  // Number of priority levels
#define TIME_QUANTUM        50000               // 50 ms = 50000 us
#define RUNNING_TIME        50                  // Experimental value
```

PRIORITY_LEVELS is the number of priorities the scheduler will sort the threads into. 0 is the highest and most desirable priority.
TIME_QUANTUM is the default time slice a thread will be given. Depending on our heuristics, some threads are scheduled to run for a multiple of this quantum.
RUNNING_TIME is the number of quanta that will run before a call to the maintenance_cycle(). Note that some threads will run for multiple time quanta, so it is not necessarily true that `RUNNING_TIME` = 50 means 50 threads will run before a call to maintenance_cycle().

View the performance.pdf file to see the effects of varying these parameters.

Priority Inversion

The special case of priority inversion is where a resource that a highly prioritized thread depends on is being held by a thread in a low priority level. This is problematic for many performance criteria.

Our scheduler helps prevent this case from occurring by preventing threads of importance from degrading below a certain priority level. A thread of importance includes a thread that has others joined to it or a thread that has other threads waiting for it to release a mutex. By preventing these threads from falling too low in the priority levels, they will have the opportunity to be scheduled to run more often.

The check to find these thread of importance is relatively simple with our design. Thread_units contain pointers to both threads that have joined it or threads that are waiting for it to release a lock. By simply checking if these pointers are null or not, we can determine if the threads are of importance to prevent priority inversion.

Heuristics

- Threads that have just run will be demoted (1) priority level.
- Threads that have not yet run will be boosted (1) priority level.
- Threads that hold a lock or have threads joined to them will not descend below (`PRIORITY_LEVEL/3`) to avoid priority inversion.
- Threads will be enqueued into the running queue as follows:

  for each priority level, i:
  Select (`PRIORITY_LEVELS-i`) threads from priority level i.
  Selected threads will run for `( (i+1) * TIME_QUANTUM )`
  Stop when `RUNNING_TIME` is met.

Functions

```
int my_pthread_create( my_pthread_t * thread, my_pthread_attr_t * attr,
void *(*function)(void*), void * arg)
```

- If a scheduler has not been initialized, scheduler_init() is called
- Set multiple different variables of thread and also encapsulate it inside a thread_unit struct
- This thread_unit struct is stored inside the scheduler's multi-level priority queue

```
void my_pthread_yield()
```

- If currently running thread is not terminated or waiting, it sets the state to ready
- If end of running queue, runs maintenence cycle
- If more threads to run, picks the next one in running queue, initializes it and runs

```
void my_pthread_exit(void *value_ptr)
```

- Exits the current running thread
- Appropriately sets state to TERMINATED and threadID to -1 (for termination purposes)
- Calls `my_pthread_yield()`

```
int my_pthread_join(my_pthread_t thread, void **value_ptr)
```

- Changes state of thread to WAITING
- Stores it in waiting_on_me

```
int my_pthread_mutex_init(my_pthread_mutex_t *mutex, const
my_pthread_mutexattr_t *mutexattr)
```

- This function essentially sets the isInitialized paraments in the mutex structure to 1
- This parameter is checked in the other mutex related functions before applying any mutex operations
- No functionally is changed by the mutex attr

```
int my_pthread_mutex_lock(my_pthread_mutex_t *mutex)
```

- This function allows the calling thread to obtain the lock of a mutex
- If the lock is currently unavailable, the caller is put in a waiting queue that implements a FIFO structure and changing its state to WAITING
- Once the lock is released by the owner, the lock is then passed to the next thread in the queue and change the state of the waiting thread to READY

```
int my_pthread_mutex_unlock(my_pthread_mutex_t *mutex)
```

- This function will release the lock of the mutex if the caller is the owner of the mutex otherwise the call fails
- On a successful unlock, the caller will pass the lock to the next thread in the waiting queue(if any) otherwise will leave it unlocked.

```
int my_pthread_mutex_destroy(my_pthread_mutex_t *mutex)
```

- This function will destroy the mutex pointed to by *mutex.
- This call will fail if the current mutex is in use and the caller is not the owner or if there is a waiting queue for the mutex. We chose to not destroy a mutex if there was a waiting queue because this will cause the waiting queue to fail once they return in the mutex lock function.

Maintenance Cycle:

- We have a running queue and a multi-level priority queue
- Adjust priorities of all threads in multi-level priority queue (increases priority)
- For each thread in running queue, lower priority and put in appropriate priority level
- Now our priority queue has all the threads back in it, we pick the set of threads to put in running queue until next maintenance cycle