Threads

In this assignment, you are to implement a non-preemptive threading library `NPTlib` for application threads. The `OS` view of an application is a sequential program with one private stack and registers. `NPTlib` allows an application thread to create non-preemptive threads without changing the `OS` view of the application.

1 NPTlib interfaces

`NPTlib` provides four interfaces:

- `void thread_create(func_t func, void *param);`
- `void thread_yield();`
- `void thread_exit();`
- `void wait_for_all();`

2 thread_create

An application thread can create more threads using `thread_create` library. `thread_create` takes the target function (that is going to execute in the new thread) of type `func_t` and a pointer of type `void*` as arguments. The pointer is passed to the target function when it is scheduled. `func_t` is the type of a function that accepts a pointer of type `void*` and returns `void`. A thread that is created using `thread_create` can also create more threads by calling `thread_create`. The target function of `thread_create` never returns and always calls `thread_exit` to terminate itself. `struct thread` represents a thread in the scheduler list and also used to save the stack pointer during the context switch. `thread_create` allocates a `struct thread` structure for the target thread and adds to the end of the `ready_list` of the `scheduler`. `ready_list` is a list maintained by the `scheduler` that contains all the threads that need CPU. `thread_create` also allocates stack for the target thread and saves the stack pointer in the newly created `struct thread`. `thread_create` sets up the stack in such a way that after returning from `context_switch`, the new thread jumps to the start of the target routine, where the first argument on the stack is the

pointer passed to `create_thread`. You can use `malloc` to allocate stack for the new thread. All the threads created using `thread_create` have a fixed size stack of 4096 bytes.

3 thread_yield

A thread can voluntarily yield the CPU using `thread_yield`. `thread_yield` puts the current thread to the end of the `ready_list` and schedules the thread that is next in the FIFO order.

4 thread_exit

The `thread_exit` routine terminates the current thread (by making sure that it won't be added to the ready_list) and schedules a new thread.

5 wait_for_all

The `wait_for_all` routine yields until there are no other threads to `schedule`.

6 schedule1

The `schedule1` routine adds the current thread (`cur_thread`) to `ready_list` and calls `schedule`. You have to be careful when `cur_thread` is NULL.

7 schedule

`schedule` implements FIFO scheduling. `schedule` pops a thread that is next in the FIFO order from the `ready_list` and calls the `context_switch` routine.

8 context_switch

The implementation of `context_switch` is provided in `context.s` file. You are not supposed to change this implementation. It takes pointers to `struct thread` corresponding to previous and next threads. The switching logic has already been discussed in the class.

9 push_back

`push_back` routine pushes the input thread to the end of the `ready_list`. This can be used to implement the scheduler logic.

10 pop_front

`pop_front` routine pops the first element from the `ready_list` and returns it to the caller. This can be used to implement the scheduler logic.

11 Implementation

You have to implement everything in `thread.c`. You are not allowed to change the `struct thread`. For this assignment, you might not need to add any extra routines apart from the ones that are provided in the skeleton, but please feel free to add new routines if you need them. You have to make sure that all updates to `ready_list` happen in `push_back` and `pop_front` APIs.

12 Environment

For this assignment, you need to clone the assignment repo from `https://github.com/Systems-IIITD/NPTlib`.
Install `gcc-multilib` using, `sudo apt-get install gcc-multilib`.
`NPTlib` contains a test case `app.c`, the library (`thread.c`), and context switch logic (`context.s`). You are to implement all APIs in `thread.c` as discussed before. You are not supposed to change the `struct thread`. `struct thread` also serves as a node in the `ready_list`. ``make'' command builds the test case and the `NPTlib` library. To run the test case, run ``make run''. The output of ``make run'' should be:

```
./app
starting main thread: num_threads: 3
thread: 0 running iteration:0
thread: 1 running iteration:1
thread: 2 running iteration:2
thread: 0 running iteration:1
thread: 1 running iteration:2
thread: 2 running iteration:3
thread: 0 running iteration:2
thread: 1 running iteration:3
thread: 0 running iteration:3
main thread exiting: counter:267
```

You are not supposed to change the test case.

12.1 Design documentation

You also have to submit design documentation along with your implementation; otherwise, the assignment will not be graded. Answer the following questions in your design documentation.

- Paste your code corresponding to `push_back`.
- Paste your code corresponding to `pop_front`.
- Paste your code corresponding to `create_thread`. If you are calling functions that are defined by you in `create_thread`, paste the code of them as well.
- Dump the output of the ‘‘`make test`’’.
- Suggest a strategy to free `struct thread` and the stack corresponding to the thread that has exited (using thread_exit API). You don't need to implement this logic.

12.2 How to submit.

To be done individually. Submit a zip folder that contains two files: "thread.c" and design documentation (in pdf format). Please make sure that your implementation is not printing any debug messages before submitting the final code. The submission link is on backpack.