OS 2025 MP1
Thread operation

TA:
Hsuan-Chun Lin(林宣均),
Chen-Wei Tang(唐承瑋)

Overview

- Implement a user-level thread package using setjmp and longjmp. The threads explicitly yield CPU time, and scheduling must be handled manually.

Score

- Part 1 (60%)
  - 40% public test case + 20% private test case.
- Part 2 (40%)
  - 20% public test case + 20% private test case.

Part 1 (60%):

- thread_create()
- thread_add_runqueue()
- thread_yield()
- schedule()
- dispatch()
- thread_exit()
- thread_start_threading()

Thread_add_runqueue()

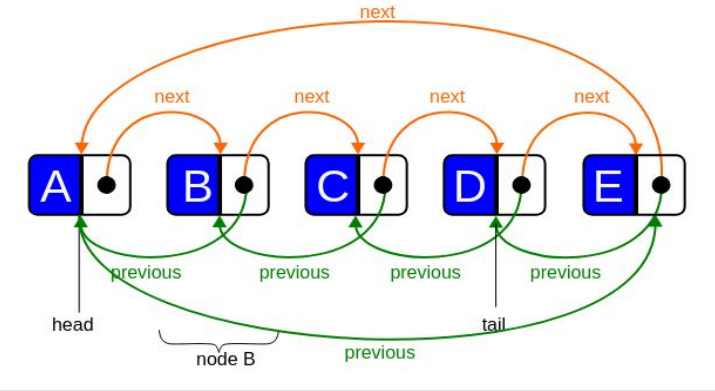
```c
struct thread {
    void (*fp)(void *arg);
    void *arg;
    void *stack;
    void *stack_p;

    jmp_buf env;
    int buf_set;
    int ID;
    struct thread *previous;
    struct thread *next;

    // part 2
    void (*sig_handler[2])(int);
    int signo;
    jmp_buf handler_env;
    int handler_buf_set;

    int suspend;
};
```

```c
void thread_add_runqueue(struct thread *t) {
    if (current_thread == NULL) {
        current_thread          = t;
        current_thread->next    = current_thread;
        current_thread->previous = current_thread;
    } else {
        //TO DO
    }
}
```



thread_yield()

```
void thread_yield(void) {
    // TO DO
}
```

- Voluntarily yields the CPU to allow another thread to run.

- After saving the context, you should call
  1. schedule( ) to decide which thread to run next.
  2. dispatch( ) to execute the next thread.

- If the the thread is resumed later, thread_yield( ) should return to the calling place in the function.

schedule()

- This function will decide which thread to run next.
- Since you will just run the next thread in the circular linked list of threads. You can simply change current thread to the next field of current thread like FCFS scheduling.

```
void schedule(void){
    current_thread = current_thread->next;
    // part 2: TO DO
}
```

dispatch()

```
void dispatch(void) {
    // TO DO
}
```

- The function execute the thread which is decided by schedule().
- If the thread has never run before, some initialization is needed.
- If the thread was executed before, just restore the context with longjmp().
- In case the thread function just return, the thread needs to be removed from the runqueue and the next one has to be dispatched.

thread_exit()

- This function has to
  1. Remove the calling thread from runqueue.
  2. Free its stack entire thread structure.
  3. Update current thread.
  4. Call dispatch().

```
void thread_exit(void) {
    if (current_thread->next != current_thread) {
        // TO DO
    } else {
        free(current_thread->stack);
        free(current_thread);
        longjmp(env_st, 1);
    }
}
```

thread_start _threading()

- This function will be called by the main function after some thread is added to the runqueue.
- It should return only if all threads have exited.
- Thread_exit() -> thread_start_threading() -> main function.

```
void thread_start_threading(void) {
    //TO DO
}
```

Test Case:

```
$ mp1-part1-0
mp1-part1-0
thread 1: 100
thread 2: 0
thread 3: 10000
thread 1: 101
thread 2: 1
thread 3: 10001
thread 1: 102
thread 2: 2
thread 3: 10002
thread 1: 103
thread 2: 3
thread 3: 10003
thread 1: 104
thread 2: 4
thread 3: 10004
thread 1: 105
thread 2: 5
thread 1: 106
thread 2: 6
thread 1: 107
thread 2: 7
thread 1: 108
thread 2: 8
thread 1: 109
thread 2: 9

exited
```

```
$ mp1-part1-1
mp1-part1-1
thread 1: 100
thread 2: 0
thread 3: 10000
thread 1: 101
thread 2: 1
thread 4: 1000
thread 5: 10
thread 3: 10001
thread 1: 102
thread 2: 2
thread 4: 1001
thread 5: 11
thread 3: 10002
thread 1: 103
thread 2: 3
thread 4: 1002
thread 5: 12
thread 1: 104
thread 2: 4
thread 4: 1003
thread 5: 13
thread 2: 5
thread 4: 1004
thread 5: 14
thread 2: 6
thread 4: 1005
thread 5: 15
thread 2: 7
thread 4: 1006
thread 5: 16
thread 2: 8
thread 4: 1007
thread 4: 1008

exited
```

Part2 (40%):

- thread_suspend()
- thread_resume()
- thread_register_handler()
- thread_kill()
- get_current_thread()

thread_suspend()

- This function suspends a thread itself, saving its state and pausing execution until the previous thread resumed. It's useful for waiting on resources or external conditions without termination.

```
void thread_suspend(struct thread *t) {
    // TO DO
}
```

thread_resume()

- This function resumes a suspended thread (*t), allowing it to be scheduled again when its waiting condition is met.

```
void thread_resume(struct thread *t){
    // TO DO
}
```

yield vs suspend

- Both will stay in the runqueue, but the suspended thread will be passed by the scheduler due to its thread suspend state.
- Suspended thread will not be executed until it’s resumed. However, yield thread will be executed next round.

thread_register_handler()

- This function assigns a signal handler function to the current thread for a given signal number. If a handler already exists for the signal, it is replaced.

```c
struct thread {
    void (*fp)(void *arg);
    void *arg;
    void *stack;
    void *stack_p;

    jmp_buf env;
    int buf_set;
    int ID;
    struct thread *previous;
    struct thread *next;

    // part 2
    void (*sig_handler[2])(int);
    int signo;
    jmp_buf handler_env;
    int handler_buf_set;

    int suspend;
};
```

```c
//PART 2
void thread_register_handler(int signo, void (*handler)(int)){
    current_thread->sig_handler[signo] = handler;
}
```

thread_kill()

- This function sends signal signo to thread t.
- If t has a handler, it executes upon resumption; otherwise, t exits. The thread resumes execution after handling the signal, and no context switch occurs upon sending.

```
void thread_kill(struct thread *t, int signo){
    // TO DO
}
```

TA information

- TA:
  - Hsuan-Chun Lin(林宣均), Chen-Wei Tang(唐承瑋).
- TA hour:
  - Mon. & Wed. 13:00-14:00.
  - CSIE Building Room R439.

Test Case:

```
$ mp1-part2-0
mp1-part2-0
thread 1: 100
handler 3: 20
thread 1: 101
handler 3: 22
thread 1: 102
handler 3: 24
thread 3: 10000
thread 1: 103
thread 1: suspending
thread 3: 10001
thread 3: 10002
thread 3: 10003
thread 3: 10004
thread 3: 10005
thread 1: resuming
thread 1: 104
thread 1: 105

exited
```

```
$ mp1-part2-1
mp1-part2-1
thread 1: 100
thread 2: 0
thread 3: 10000
thread 1: 101
handler 2: 1
thread 2: 1
thread 3: 10001
thread 1: 102
thread 2: 2
thread 3: 10002
thread 1: 103
thread 2: 3
thread 3: 10003
thread 1: 104
thread 2: 4
thread 2: suspending
handler 3: 21
thread 1: 105
handler 3: 23
thread 1: 106
handler 3: 25
thread 1: 107
handler 3: 27
thread 1: 108
handler 3: 29
thread 3: 10004
thread 1: 109
thread 2: resuming
thread 2: 5
thread 2: 6
thread 2: 7
thread 2: 8
thread 2: 9

exited
```

Q&A