代码实现

因为报告的要求，这里长话短说。

线程初始化

在运行线程之前，首先分配内存，然后初始化主线程（main_thread）和当前进程（current_thread），不能让他为 NULL，然后设置 rsp：

```
long long sp = ((long long)&uthread->stack + STACK_SIZE) & (~(long long)15);
sp -= 8;
uthread->context.rsp = sp;
```

设置 rip：

```
uthread->context.rip = (long long)_uthread_entry;
```

这是两个比较重要的寄存器，其他寄存器不再赘述。
然后设置线程的状态

```
uthread->state = THREAD_INIT;
```

调度器

初始化一个队列，然后用队列调度。

```
// 从队列中取出一个线程控制块
struct uthread* next_thread = active_queue.threads[active_queue.front];

// 更新队列前端索引
active_queue.front = (active_queue.front + 1) % active_queue.rear;
active_queue.count--;
```

下面是对各种状态的处理，注意只有线程是 STOP 的才可以被恢复上下文：

```
if (next_thread->state == THREAD_INIT) {
    // 将状态设置为 THREAD_STOP 并开始执行
    next_thread->state = THREAD_STOP;
}
if (next_thread->state == THREAD_STOP) {
    // 将状态设置为 THREAD_STOP 并开始执行
    // 恢复下一个线程的上下文
    uthread_resume(next_thread);
}
```

```
if (next_thread->state == THREAD_SUSPENDED) {
    next_thread->state = THREAD_STOP;
}
```

恢复上下文

先看一下 resume 函数。

“

```
tcb->state = THREAD_RUNNING;
// 更新当前线程为要恢复的线程
struct uthread* previous_thread = current_thread;
current_thread = tcb;

// 切换到目标线程的上下文
thread_switch(&previous_thread->context, &current_thread->context);
    注意它和 uthread_entry 其实是一个闭环，因为这是 entry：
thread_func(arg);
thread_switch(&current_thread->context, &main_thread->context);
thread_destroy(tcb);
```

调度遇到 STOP 的线程时，先运行到 resume 函数，然后传入一个 tcb，然后首先要用 switch 换到这个 tcb，这个 tcb 指向的是 uthread_entry，然后在 uthread_entry 里运行函数，运行完了再 switch，又恢复到 resume 函数，resume 再 return 回去进入调度，这样就形成了一个闭环。

Yield 函数

首先找到当前进程在队列里是多少号，当前线程然后找到下一个线程的 tcb 传到 resume 函数里面去，切换回来的时候再改变调用 Yield 函数的线程的状态。下面是 Yield：

“

```
int nextcb = (find_tcb_position(current_thread) + 1) % active_queue.rear;

// 将当前线程状态设置为挂起
current_thread->state = THREAD_SUSPENDED;

// 从队列中取出一个线程控制块
struct uthread* next_thread = active_queue.threads[nextcb];
next_thread->state = THREAD_STOP;
uthread_resume(next_thread);

// 上下文切换回来后，恢复之前线程的状态
next_thread->state = THREAD_RUNNING;

// 返回 0 表示成功
return 0;
```

困难及解决

作业要求理解

第一次写这个作业，光是读懂要求就花了一个小时，做写代码花十几个小时，然后写完提交 git，没啥经验，又卡了三个小时……第一次做这种作业，下次 lab 应该没有这种问题了。

Thread_Switch 函数的理解，调用和返回

Thread_Switch 函数是用汇编写的，非常的难懂，我花了很长的时间理解它，即使是有 GPT 的帮助下。然后为什么在 demo.c 里把 main_thread 传回去就可以恢复运行 switch 的代码段的上下文我也不太理解，只能在自己的代码里照用。然后这个函数好像必须要配套用，换过去了之后要换回来，之前也不太清楚这个规则，又花了很久的时间才用明白这个函数。

如何获取当前线程的下一个线程

这个主要是 Yield 里用的，因为 Yield 要运行下一个 tcb 而且不能通过调度器，调度器只负责一个线程完了再调度，对于这种中途进来要调用下一个线程的毫无办法。

获取下一个线程不太简单，我之前直接用调度器的队列前端 front 找的，但是这样就会有一个问题，当前线程传球给下一个线程，下一个线程再 Yield 的话由于 front 是固定的，他就会卡在这个线程里死循环，所以要直接拿 tcb 一个一个匹配，找到它在队列的位置再把下一个取出来，不知道有没有更高明的办法。

字段错误

字段错误报错方式多种多样，不会用 Gdb 调试就会非常的绝望，不知道 Gdb 调试的有难了。

Challenge

thread_swtich 里只保存了整数寄存器的上下文。如何拓展到浮点数?

x86-64 架构中的浮点寄存器主要是 SSE（xmm0 到 xmm15）和 AVX（ymm0 到 ymm15）寄存器。在切换线程前，需要将浮点寄存器的值保存到栈或线程控制块中（TCB），当切换到新线程时，需要将之前保存的浮点寄存器值恢复。

代码见 SwitchPlus。

上面我们只实现了一个 1 kthread :n uthread 的模型，如何拓展成 m : n 的模型呢

我们实现的是经典的一个内核线程对多个线程，要拓展到 2 个内核线程的话，要：

- 在 makefile 文件里添加对 pthread 库的选项

> “
>
> ```
> LDFLAGS := -lpthread  # 添加链接 pthread 库的选项
> ```

- 新创建一个全局调度函数，创建两个内核线程和两个队列

“

```
struct thread_queue active_queue_1 = { .front = 0, .rear = 0, .count = 0 };
struct thread_queue active_queue_2 = { .front = 0, .rear = 0, .count = 0 };

static struct uthread* current_thread_1 = NULL;
static struct uthread* current_thread_2 = NULL;

pthread_t kernel_thread1, kernel_thread2;
void global_scheduler() {
    pthread_create(&kernel_thread1, NULL, schedule, (void*)&queue1);
    pthread_create(&kernel_thread2, NULL, schedule, (void*)&queue2);
    pthread_join(kernel_thread1, NULL);
    pthread_join(kernel_thread2, NULL);
}
```

->pthread_create 启动线程时，两个内核线程是并发执行的，而不是按顺序执行。

- uthread_create 分配线程的时候自动交替分配到两个不同内核线程

“

```
if (toggle % 2 == 0) {
    queue1.threads[queue1.rear] = uthread;
    queue1.rear = (queue1.rear + 1) % MAX_THREADS;
    queue1.count++;
} else {
    queue2.threads[queue2.rear] = uthread;
    queue2.rear = (queue2.rear + 1) % MAX_THREADS;
    queue2.count++;
}
toggle++;
```

- 由于现在有两个队列，所以 find_tcb_position 函数需要修改

“

```
int find_tcb_position(struct thread_queue* queue, struct uthread* tcb) {
    for (int i = queue->front; i != queue->rear; i = (i + 1) % MAX_THREADS) {
        if (queue->threads[i] == tcb) {
            return i; // 找到 tcb，返回其位置
        }
    }
    printf("error not found\n");
    return -1; // 没找到
}
```

->这里需要额外传入一个队列号参数，不然不可能找得到的。

上述的实现是一个非抢占的调度器，如何实现抢占的调度呢？

要实现抢占式调度器，关键在于引入定时器中断或某种机制，使得调度程序可以在某个线程执行过程中强制暂停它，并切换到另一个线程，不过最常用的还是定时器。只要能打断，就可以重新分配，实现抢占式调度。

“

```
// 定时器中断处理函数，每次触发中断时调用调度器
void timer_interrupt_handler() {
    // 保存当前正在运行的线程上下文
    uthread_suspend(current_thread);

    // 切换到调度器
    schedule();
}

void schedule() {
    /*
    实现抢占式调度器。通过定时器中断和上下文切换，实现线程的抢占。
    */
    while (1) {
        // 检查队列是否为空
        if (active_queue.count == 0) {
            return; // 没有可调度的线程，返回
        }

        // 从队列中取出一个线程控制块
        struct uthread* next_thread = active_queue.threads[active_queue.front];

        // 更新队列前端索引
        active_queue.front = (active_queue.front + 1) % QUEUE_SIZE;
        active_queue.count--;

        if (next_thread->state == THREAD_INIT) {
            // 如果线程处于初始状态，设置为运行状态
            next_thread->state = THREAD_STOP;
            // 开始执行新线程，使用上下文切换
            uthread_resume(next_thread);
        }
        else if (next_thread->state == THREAD_STOP) {
            // 如果线程之前被暂停，恢复它的执行
            next_thread->state = THREAD_STOP;
            uthread_resume(next_thread);
        }
        else if (next_thread->state == THREAD_SUSPENDED) {
            // 线程被抢占或挂起时，恢复它的执行
            next_thread->state = THREAD_STOP;
            uthread_resume(next_thread);
        }
        // 当线程执行完毕后，重新将其加入队列尾部
        active_queue.threads[active_queue.rear] = next_thread;
        active_queue.rear = (active_queue.rear + 1) % QUEUE_SIZE;
        active_queue.count++;
    }
}
```

->这里是 STOP 而不是 RUNNING 的原因是，resume 只恢复已经 STOP 的函数。

- 在实现抢占的基础上，如何去实现同步原语（例如，实现一个管道 channel）

多线程或多进程并发访问共享资源时，有可能会发生诡异的错误，需要同步原语保证安全。所以按理来说，这里应该是多个内核线程调度多个用户态线程，并且这些用户态线程都访问共享资源的时候才能用得到。首先需要针对某些会访问共享资源的线程，给他们添加互斥锁这种确保只有一个线程能够访问共享资源的东西。

但是有一个问题是，我们不知道哪些线程需要锁，有可能这个线程只是 read-only，或者连读都不读，而提前知道线程需不需要锁是一件不太可能的事情，所以如果说在创建线程的时候，不提供这个线程需不需要锁的信息的话，还不如直接一个内核进程管理，这样还不会有同步的烦恼。

所以首先，用户程序需要提供这个线程是否需要锁的信息，然后把线程分为两种类型，使用两种调度函数，一种调度函数直接调度，一种调度函数需要设置互斥锁（或其他），在一个线程使用该调度函数时其他需要锁的线程无法调度，这样就能实现同步原语了。

但是上面抢占式的函数，我是基于单内核线程调度改的，所以这个问题就不用代码实现了。

感悟

在暑假的时候，我已经自学了操作系统的课程，但是听课写作业发现讲的太不一样了，考研比较倾向于抽象出一些概念并学习，而老师教的更加本源一点，难度也更大。这次 lab 花费了我相当多的时间，但是也收获了很多。