协程实验 REPORT

计15 宋驰 2021010797

Task1: 协程库的编写

- Task1所添加的代码：
  - context.S

```
    # 保存 callee-saved 寄存器到 %rdi 指向的上下文
    # 保存的上下文中 rip 指向 ret 指令的地址（.coroutine_ret）
    leaq .coroutine_ret(%rip), %rax
    movq %rax, 120(%rdi)
    movq %rsp, 64(%rdi)
    movq %rbx, 72(%rdi)
    movq %rbp, 80(%rdi)
    movq %r12, 88(%rdi)
    movq %r13, 96(%rdi)
    movq %r14, 104(%rdi)
    movq %r15, 112(%rdi)

    # 从 %rsi 指向的上下文恢复 callee-saved 寄存器
    # 最后 jmpq 到上下文保存的 rip
    movq 64(%rsi), %rsp
    movq 72(%rsi), %rbx
    movq 80(%rsi), %rbp
    movq 88(%rsi), %r12
    movq 96(%rsi), %r13
    movq 104(%rsi), %r14
    movq 112(%rsi), %r15
    jmpq *120(%rsi)
```

    在coroutine_switch中，保存调度器的寄存器状态至协程的栈存储寄存器中，并将协程的寄存器状态保存到调度器中，然后跳转到rip中保存的下一条指令的地址。

  - resume()函数

```
    coroutine_switch(caller_registers, callee_registers);
    // 调用 coroutine_switch
    // 在汇编中保存 callee-saved 寄存器，设置协程函数栈帧，然后将 rip 恢复到协程
yield 之后所需要执行的指令地址
```

    将调度器的寄存器值暂存入caller，并将callee暂存的寄存器值恢复到调度器中。

  - yield()函数

```
    coroutine_switch(context->callee_registers, context-
>caller_registers);
    // 调用 coroutine_switch 切换到 coroutine_pool 上下文
```

    把协程的寄存器值暂存入callee，并将caller暂存的寄存器值恢复到调度器中。

  - serial_execute_all()函数

```
    bool all_finished = false; // 在函数内用于判断是否所有协程都已经finish
    while (!all_finished) {
        all_finished = true;
        for (int i = 0; i < coroutines.size(); i++) { // 轮询协程池中的所有
协程
            auto context = coroutines[i];
            if(!context->finished) { // 存在一个协程没有finish,则标记协程并未
全部跑完，并执行该协程
                all_finished = false;
                context_id = i;
                context->resume();
            }
        }
    } // 若所有协程都已finish，则跳出循环
```

采用while循环的方式，在循环内部轮询所有协程，检查是否已经finish，若没有finish，则立即执行，并标记所有协程并未全部finish，从而while循环继续，直至所有协程都已经finish，while循环才结束。

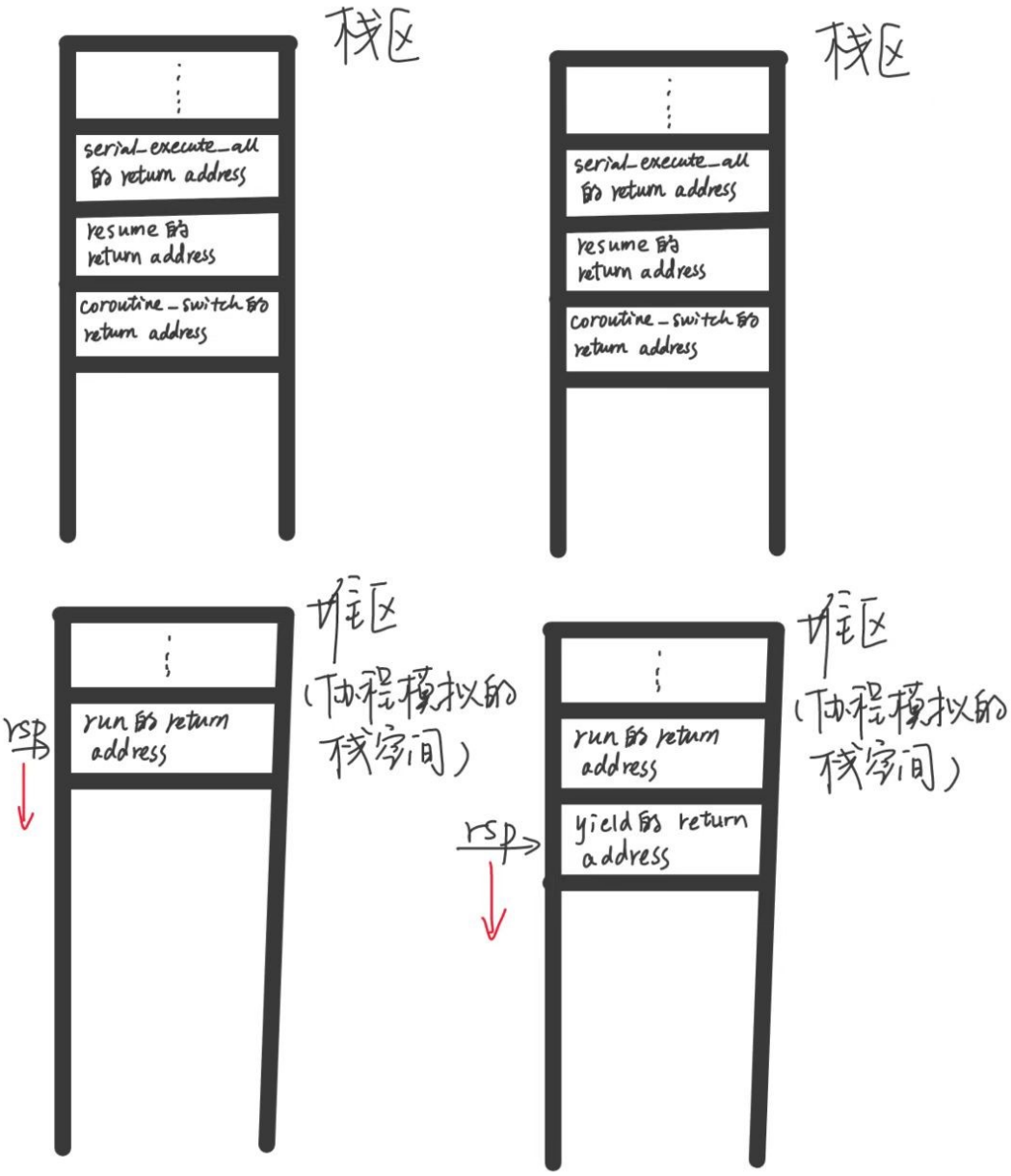
- 代码分析

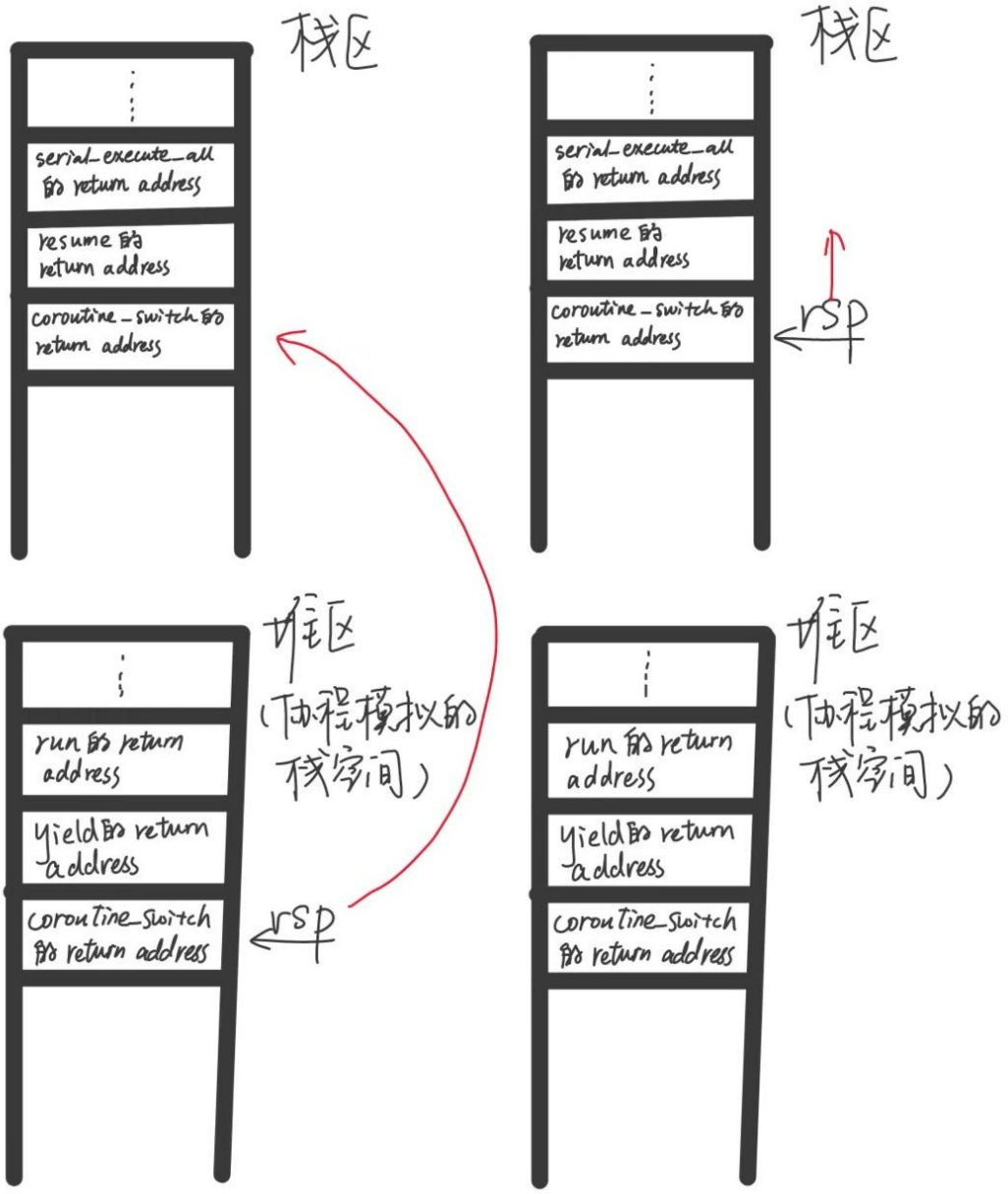
```
    // 对齐到 16 字节边界
    uint64_t rsp = (uint64_t)&stack[stack_size - 1];
    rsp = rsp - (rsp & 0xF);

    void coroutine_main(struct basic_context * context);

    callee_registers[(int)Registers::RSP] = rsp;
    // 协程入口是 coroutine_entry
    callee_registers[(int)Registers::RIP] = (uint64_t)coroutine_entry;
    // 设置 r12 寄存器为 coroutine_main 的地址
    callee_registers[(int)Registers::R12] = (uint64_t)coroutine_main;
    // 设置 r13 寄存器，用于 coroutine_main 的参数
    callee_registers[(int)Registers::R13] = (uint64_t)this;
```

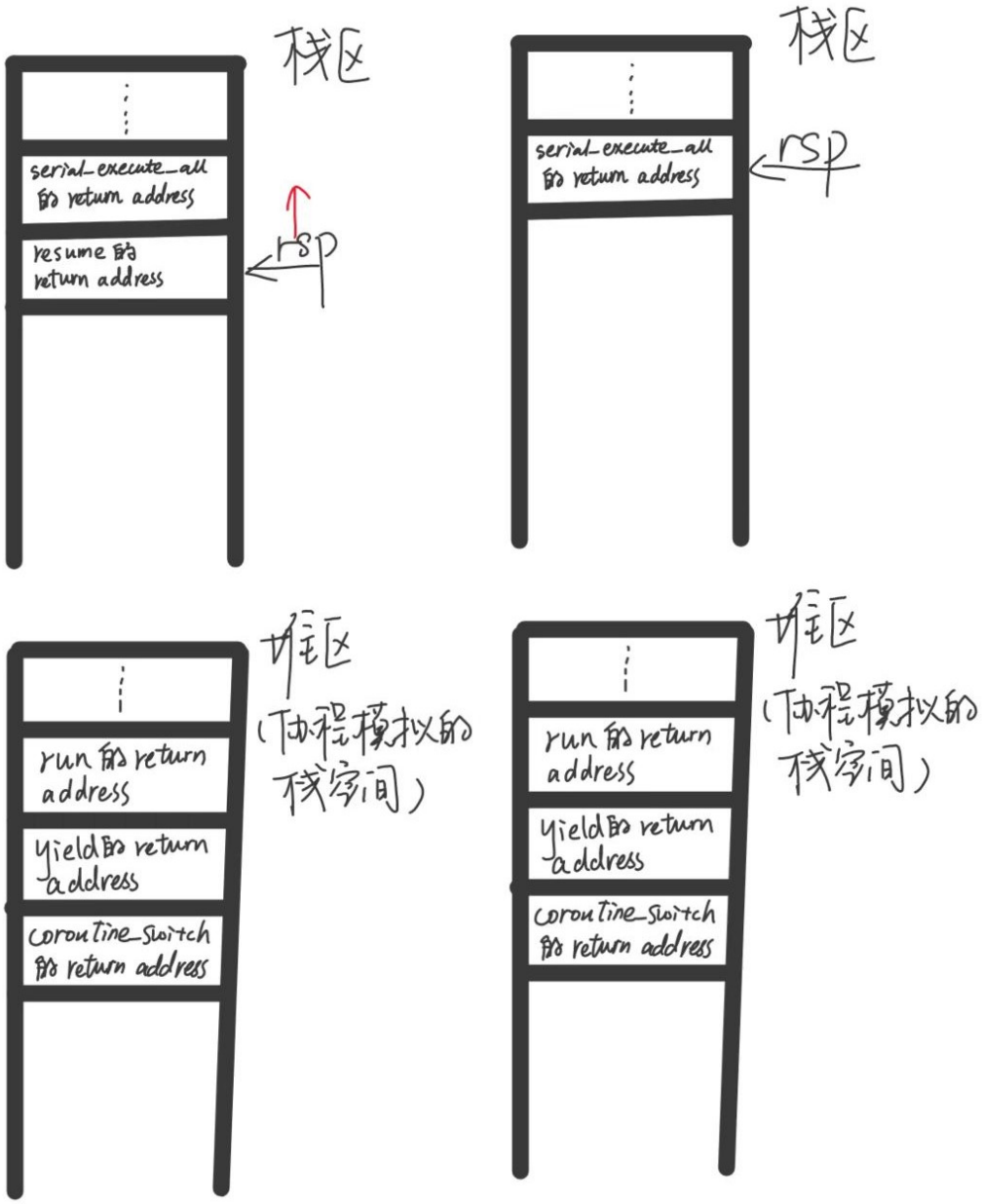
  - 首先，让rsp指向协程的栈空间stack（的末位），并进行16字节对齐，模拟栈从高地址向低地址扩展的性质，与实际的函数调用相符合。
  - 下一步，将callee_registers的rsp设置为协程对象的rsp；将callee_registers的rip设置为coroutine_entry，将callee_registers的r12设置为coroutine_main，将callee_registers的rsp设置为协程对象本身。
  - 协程初始化的时候，在自己的callee_registers中存储了rsp，coroutine_entry、coroutine_main函数的地址。开始执行的时候，resume()函数中调用coroutine_switch()函数，保存caller的寄存器状态，并将之前初始化的callee的寄存器状态同步到真正的栈当中。随后由`jmpq *120(%rsi)`（rip存的coroutine_entry函数入口）跳转至coroutine_entry函数，将寄存器r13中存的指向协程本身的指针作为函数参数，调用coroutine_main函数。

```
void coroutine_main(struct basic_context *context) {
  context->run();
  context->finished = true;
  coroutine_switch(context->callee_registers, context->caller_registers);

  // unreachable
  assert(false);
}
```

    - 在进入coroutine_main后，协程调用run函数，协程finish后将该协程的finished标记调为true，并切出回到调度器。
- 协程切换时的栈的变化过程：
    - 下图中展示了由协程切换到调度器时的栈的变化过程。一开始调度器的栈中依次压入了serial_execute_all、resume、coroutine_switch三个函数的返回地址，在coroutine_switch的作用下，rsp指向协程的run函数的返回地址，随后调用yield函数来切出，因此协程的模拟栈中也压入了yield、coroutine_switch函数的返回地址。从而由协程切回了调度器，并将调度器栈中压入的返回地址依次退栈。
    - 图中的红色线代表了rsp下一步的移动方向。由于从调度器切换到协程的过程只需将图中所示的六个状态反向进行即可，因此并没有重复作画。




栈区
serial_execute_all 的 return address
resume 的 return address
coroutine_switch 的 return address
栈区
serial_execute_all 的 return address
resume 的 return address
coroutine_switch 的 return address
rsp
栈区
(协程模拟的栈空间)
run 的 return address
yield 的 return address
coroutine_switch 的 return address
rsp
栈区
(协程模拟的栈空间)
run 的 return address
yield 的 return address
coroutine_switch 的 return address



Task2: 实现 sleep 函数

- Task2中所添加的代码：
  - serial_execute_all()函数

```
    bool all_finished = false;
    while (!all_finished) {
      all_finished = true;
        for (int i = 0; i < coroutines.size(); i++) {
        auto context = coroutines[i];
        if(!context->finished) {
          all_finished = false;
          if (!context->ready && context->ready_func()) { // 判断协程是否
超时并唤醒
            context->ready = true;
          }
          if (context->ready) { // 协程有ready标志的话，则执行
            context_id = i;
            context->resume();
          }
```

```
        }
      }
    }
```

在Task1的代码基础上，添加ready标志。

若协程的ready_func函数返回值表明协程已经超时，则唤醒协程，将协程的ready标志改为true。

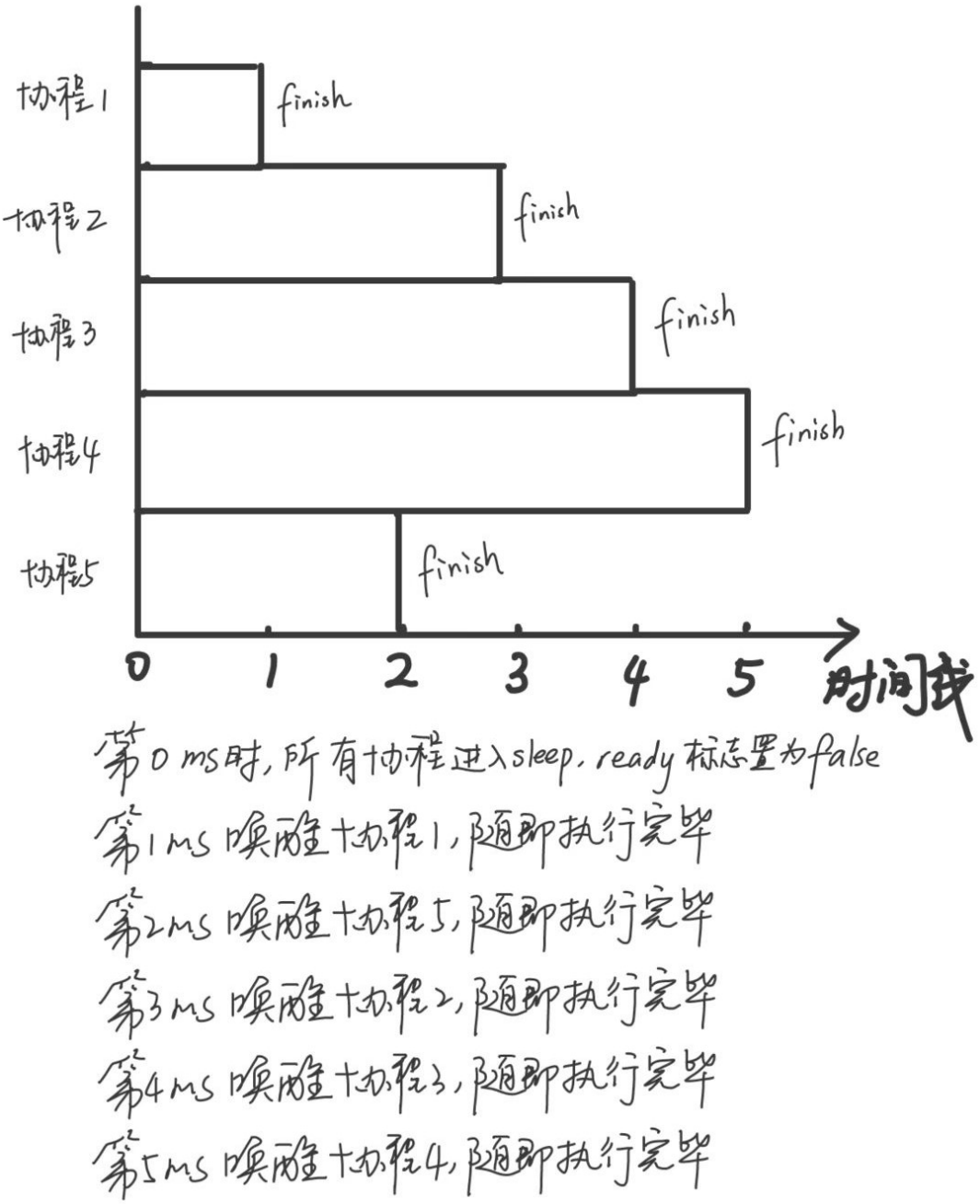
若协程的ready标志是true，则代表时间已到，可以执行。

- sleep()函数

```
    // 从 g_pool 中获取当前协程状态
    auto context = g_pool->coroutines[g_pool->context_id];

    // 获取当前时间，更新 ready_func
    // ready_func：检查当前时间，如果已经超时，则返回 true
    context->ready = false;
    auto cur = get_time();
    context->ready_func = [cur, ms] () {
      return std::chrono::duration_cast<std::chrono::milliseconds>
(get_time() - cur)
            .count() >= ms;
    };
    // 调用 coroutine_switch 切换到 coroutine_pool 上下文
    coroutine_switch(context->callee_registers, context-
>caller_registers);
```

在获取协程状态之后，获取当前时间，若已经超时，则在ready_func函数中返回true，否则返回false。

- sleep_sort不同协程的运行
  - 下图中展示了sleep_sort中各个协程的运行情况，以输入为1,3,4,5,2为例。分别对应协程1,2,3,4,5。



- 更高效的方法

  维护一个队列，将每个协程按照睡眠时间的长短，由低到高依次入队，这样可以保证最先睡眠结束可以被唤醒（ready标志位true）的协程处于队列头部，这样每次就不需要轮询所有的协程，而只需要检查处于队列头部的协程是否ready，从而可以缩短时间。

Task3: 利用协程优化二分查找

- Task3中所添加的代码：

```
    // 使用 __builtin_prefetch 预取容易产生缓存缺失的内存
    __builtin_prefetch(&table[probe], 0, 1);
    // 并调用 yield
    yield();
```

  预取内存，然后调用yield执行其他协程。

- 性能的提升效果

```
Size: 4294967296
Loops: 1000000
Batch size: 16
Initialization done
naive: 1620.31 ns per search, 50.63 ns per access
coroutine batched: 1128.03 ns per search, 35.25 ns per access
```

平均来看，性能提升在20%-30%之间，但也存在使用coroutine后效果并不明显的情况，有可能是由于协程之间来回切换也会消耗时间。

在完成协程实验的过程中的交流情况

思路和代码实现参考了习题课；与王豪达同学交流了优化sleep_sort的思路；与于鉴轩同学交流了协程和调度器切换的原理。

总结和感想

对于协程实验的最大感受就是从一开始的完全不理解到后来逐渐理解的这样一个过程，在不断和同学交流以及参考助教的习题课后，自己也逐渐理解了协程的过程和背后的原理，对于课上提到的GET、SET也有了更深的理解，对于汇编代码也有了更深的理解，尤其是对栈这个结构的理解更加深入。总的来说，是对于自己能力的一次提升，希望未来自己在汇编层面和系统层面的理解能够更加深入。