计算机系统概论协程实验报告

陈墨涵 计14 2021010769

Task1

此部分任务要求利用汇编代码和C++代码我们完成协程库的编写，并且通过基础测试。

为了模拟一个协程库，首先用枚举类型定义了协程运行时所需要使用的十六个寄存器：

```
enum class Registers : int {
  RAX = 0,
  RDI,
  RSI,
  RDX,
  R8,
  R9,
  R10,
  R11,
  RSP,
  RBX,
  RBP,
  R12,
  R13,
  R14,
  R15,
  RIP,//当前指令的下一条指令对应的行的地址
  RegisterCount
};
```

其中 `rip` 寄存器较为特殊，用于保存当前指令的下一条指令对应的地址，每个形如 `(int)Registers::RAX` 的量都表示该寄存器的位置,特别地， `RegisterCount` 表示寄存器的个数。接下来，需要用汇编语言编写如下两个函数 `coroutine_entry()` 和 `coroutine_switch()` ，分别实现进入协程和切换协程的功能。

```
coroutine_entry:
  movq %r13, %rdi
  callq *%r12
```

在如上的 `coroutine_entry()` 函数中，将 `rdi` 寄存器中的值保存进入 `r13` 寄存器，并访问 `r12` 寄存器中存储的地址。我们需要做的是编写 `coroutine_switch()` 函数，编写如下：

```
coroutine_switch:
  movq %rax, (%rdi)
  movq %rdi, 8(%rdi)
  movq %rsi, 16(%rdi)
  movq %rdx, 24(%rdi)
  movq %r8, 32(%rdi)
  movq %r9, 40(%rdi)
  movq %r10, 48(%rdi)
  movq %r11, 56(%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)
  leaq .coroutine_ret(%rip), %r8
  movq %r8, 120(%rdi)

  movq (%rsi), %rax
  movq 24(%rsi), %rdx
  movq 32(%rsi), %r8
  movq 40(%rsi), %r9
  movq 48(%rsi), %r10
  movq 56(%rsi), %r11
  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)
```

在这一实现中，前一段是按顺序将寄存器中的值以8个字节的跨度依次存入 `rdi` 寄存器中。前十五个寄存器中的值直接采用 `movq` 指令即可，对于最后一个 `rip` 寄存器，由于其存储着下一条指令对应的地址，因此需要获取 `coroutine_ret` 返回值中 `rip` 寄存器的地址，借由某个中间寄存器（例如 `r8` ），先进行 `leaq .coroutine_ret(%rip), %r8` 操作，再进行 `movq %r8, 120(%rdi)` 操作，从而达到存储效果。
后一段操作中，是将 `rsi` 寄存器上下文所存储的值按顺序恢复到各原始寄存器中，整体语法大同小异，需要注意的是 `rdi` 与 `rsi` 寄存器本身存放的值不能通过 `movq` 操作修改，因此编写时 `8(%rsi)` 与 `16(%rsi)` 位置对应的值不必移动。这一部分最后利用 `jmpq *120(%rsi)` 返回即可。至此， `coroutine_switch()` 函数编写完毕，可供后续调用。

对协程的模拟，采用结构体来定义：定义了 `uint64_t *stack` 作为协程可以使用的栈空间；数组 `caller_registers` 和 `callee-registers` 分别表示调用协程前的寄存器状态和协程正在运行时的寄存器状态； `bool` 型变量 `finished` 和 `ready` 表示协程是否结束与协程是否可以运行；函数 `ready_func()` 的返回值为 `bool` 类型，用于进一步判断是否恢复协程运行。在成员声明完成后，又对一些变量的初始值进行定义。

```
uint64_t rsp = (uint64_t)&stack[stack_size - 1];
rsp = rsp - (rsp & 0xF);
```

此处获取协程中 `rsp` 寄存器中的存储值，并且将其后四位置零，以确保 `rsp` 寄存器中的值对齐16字节，使其存储的指令地址位于其长度的整数倍。

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

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

此处定义了 `coroutine_main` 函数，表示控制协程的运行。在函数中，首先调用 `run()` 成员函数，在协程运行结束后，设置 `finished` 变量为 `true` ，并且调用 `coroutine_switch` 函数，将当前寄存器状态切换为调用前的寄存器状态。若正常定义，则不会运行至 `assert(false)` 一行。

```
virtual void resume() {
  coroutine_switch(caller_registers, callee_registers);
}
```

以上是对 `context::resume()` 函数的实现，该函数的功能是将某个协程恢复运行，显然，只需要调用一次 `coroutine_switch` 函数，将调用前的寄存器状态切换至协程正在运行时的寄存器状态即可。

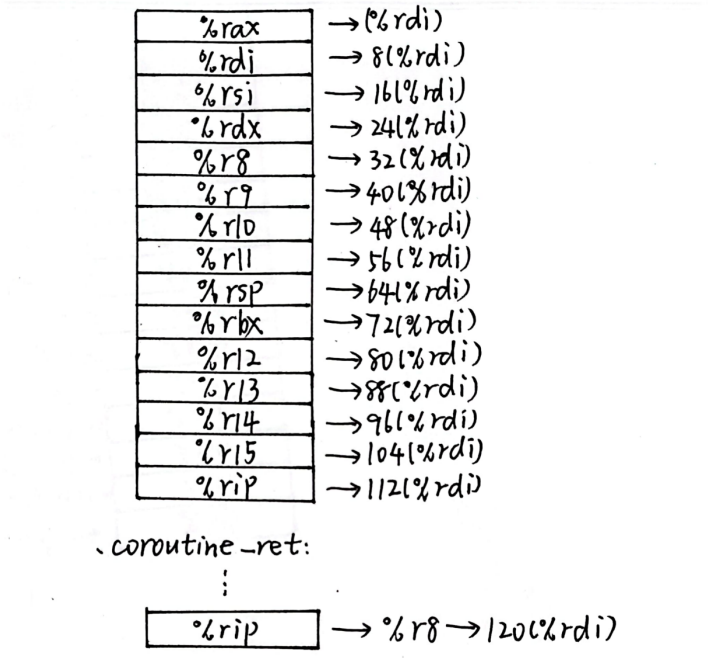
```
void yield() {
  if (!g_pool->is_parallel) {
    auto context = g_pool->coroutines[g_pool->context_id];
    coroutine_switch(context->callee_registers, context->caller_registers);
  }
}
```

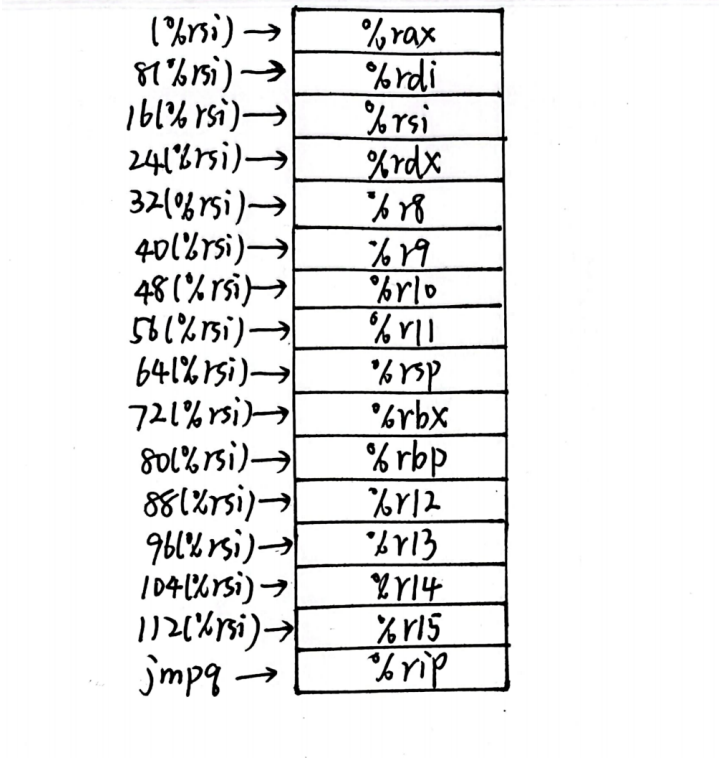
以上是对 common::yield() 函数的实现，该函数的功能是暂停某个协程。由于此函数为全局定义，则先从 g_pool->is_parallel 获取协程池状态，若协程之间不是并行的，则从 context_id 获取 g_pool->coroutines 中对应的协程 context ， 之后调用 coroutine_switch 函数，将当前正在运行的寄存器状态切换至运行前的状态，则实现了暂停协程运行的功能。

```
void serial_execute_all() {
  is_parallel = false;
  g_pool = this;
  while(true) {//不断遍历
    int cnt = 0;//未完成协程的数目
    for (int i = 0; i < coroutines.size(); i ++) {
      if (coroutines[i]->finished == false) {
        context_id = i;
        cnt += 1;
        coroutines[i]->resume();//若finished值为false，则恢复运行
      }
    }
    if (cnt == 0) {
      break;//若某一轮遍历cnt=0，则停止轮询，退出循环
    }
  }
  for (auto context : coroutines) {
    delete context;
  }
  coroutines.clear();
}
```

以上是对 coroutine_pool::serial_execute_all() 函数的实现，该函数的功能是根据协程的 finished 和 ready 状态确定是否恢复协程运行。在 Task 1中我们无需考虑 ready ， 则可以采用轮询的方式，遍历 coroutines 中的所有协程。我们定义一个临时变量 cnt 存储未完成的协程个数，若协程 coroutines[i] 的 finished 为 false ， 则令这一轮的 context_id 值为 i ， 并调用 coroutines[i]->resume() 函数恢复运行，同时 cnt+=1 。重复进行遍历，直到某一轮遍历后 cnt=0 ， 停止遍历，此时所有未完成的协程都已恢复运行。

在切换协程的过程中，栈的变化如下：





在一个协程被调用时，通过 `coroutine_entry()` 函数获取传输给协程的指令，接着在 `coroutine_main()` 函数中调用 `run()` 运行协程并进行后续操作，其中 `run()` 函数获取函数以及对应的参数并执行。初始的协程状态为 `finished=false` ， 同时 `ready=true` ， 即可以恢复运行，但未运行完成。

Task 2

在Task 2中，我们需要完成sleep_sort的功能，为此需要实现 `sleep` 函数，并且修改 `coroutine_pool::serial_execute_all()` 函数，考虑协程的 `ready` 和 `ready_func` 属性。

```
void sleep(uint64_t ms) {
  if (g_pool->is_parallel) {
    auto cur = get_time();
    while (std::chrono::duration_cast<std::chrono::milliseconds>(get_time() - cur).count() < ms)
      ;
  } else {
    auto context = g_pool->coroutines[g_pool->context_id];//获取协程
    auto current_time = get_time();//获取开始时间
    context->ready = false;//设置ready=false，暂不可恢复协程
    context->ready_func = [ms, current_time]() {return std::chrono::duration_cast<std::chrono::milliseconds>(get_time() - current_time
    yield();//暂停协程
  }
}
```

以上是 `common::sleep()` 函数的实现，这个函数的功能是暂停某一协程，直至其停止时间大于传入的参数。我们首先通过 `g_pool->context_id` 获取当前协程 `context` ， 接着，由于协程暂停，条件未满足之前我们无法将其恢复，则设置 `context->ready=false` ， 同时在此处定义 `context->ready_func` 。首先调用事先定义好的 `get_time()` 函数获取调用 `sleep` 时的系统时间 `current_time` ， 之后采用 `lambda` 表达式的写法，传入参数 `ms` 和 `current_time` ， 返回之后的系统时间与 `current_time` 的差值是否大于等于 `ms` 。若为 `true` ， 则代表暂停时间足够， `ready_func()` 返回值为真，便于后续恢复协程运行。最后，为了达到暂停协程的目的，还需调用 `yield()` 函数。至此， `sleep()` 函数实现完成。

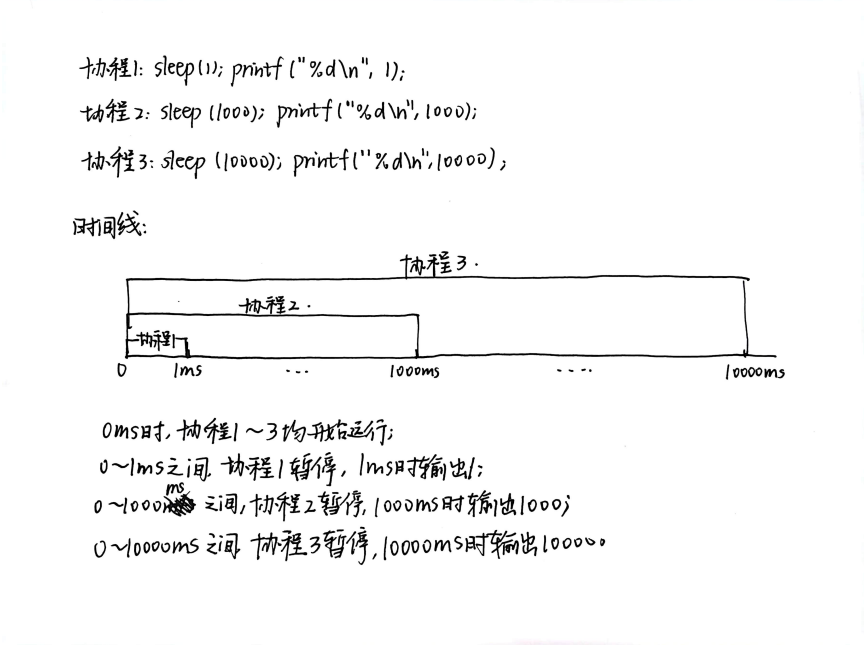
```
void serial_execute_all() {
    is_parallel = false;
    g_pool = this;
    while(true) {
      int cnt = 0;
      for (int i = 0; i < coroutines.size(); i ++) {
        if (coroutines[i]->finished == false) {
          context_id = i;
          cnt += 1;
          if (coroutines[i]->ready == true) {
            coroutines[i]->resume();
          }
          else {
            if(coroutines[i]->ready_func() == true) {
              coroutines[i]->ready = true;
              coroutines[i]->resume();
            }
          }
        }
      }
      if (cnt == 0) {
        break;
      }
    }
    for (auto context : coroutines) {
      delete context;
    }
    coroutines.clear();
  }
```

以上是 coroutine_pool::serial_execute_all() 函数的完整实现。由于需要考虑 ready 和 ready_func()，因此在判断是否恢复协程运行的逻辑处进行了修改。依然是在遍历所有协程的过程中，若 finished=false，且 ready=true，则恢复协程运行；若 ready=false，则检查 ready_func() 的返回值，若返回 true，则修改 ready=true，同时恢复协程运行。其余代码实现不做修改。这样便可以考虑完整的情况，实现完成。

以如下输入为例：

```
3
1000 1 10000
```

在传入上述参数之后，协程的运行情况如下：



本题中，协程库的实现采用了轮询 ready_func() 是否返回 true 的方法。在实现中，我们可以定义一个优先队列按照首次 resume 的顺序存储各个协程，之后每次循环时，优先检查最先恢复过的协程是否恢复，这样可以避免重复搜索，提升运行效率。

Task 3

在Task 3中，我们需要使用协程来优化二分查找，其最终原理就是利用 `__builtin_prefetch` 预读取数据，之后立刻切换协程。实现中只需在 `lookup_coroutine` 函数中 `size_t probe = low + half;` 这一行之后加上如下两行：

```
__builtin_prefetch(table + probe);
yield();
```

其中 `__builtin_prefetch(table + probe);` 表示预先读取下一步查找所需要用到的内存数据， `yield();` 表示暂停并切换到其他协程。这样就可以确保装载器读取内存和CPU运行同时进行，效率最大化，而不会出现单线程时大部分时候装载器和CPU只有一方工作的情况。这样便实现了优化。

在WSL中运行 `./bin/binary_search` 之后，普通运行模式与协程运行模式的数据如下：

```
Size: 4294967296
Loops: 1000000
Batch size: 16
Initialization done
naive: 1345.09 ns per search, 42.03 ns per access
coroutine batched: 1087.51 ns per search, 33.98 ns per access
```

可见使用协程优化后的二分查找效率提升了约23%，每次access的所用时间也有了显著提升。

优化前后的CPU和装载器工作状况如下图所示：

优化前：

	1	2	3	···
CPU		✓		···
装载器	✓		✓	···

优化后：

	1	2	3	···
CPU		✓	✓	···
装载器	✓	✓	✓	···

可以直观看出效率优化之所在。

总结与感想

总结：本次实验使我深入理解了协程的概念与作用，直观感受了协程在调度分配、优化效率上的强大功能。同时，我也对汇编指令有了进一步的理解，更深入了解了计算机的内存架构和运转方式，并且在完成实验的过程中锻炼了自己的代码阅读理解能力以及汇编语言程序设计能力。
感想：在接触自己之前较为陌生的概念时，需要充分利用已有的提示，先确保做到了解原理，再思考具体实现的方式，这样能快速高效地完成任务，收获提升。