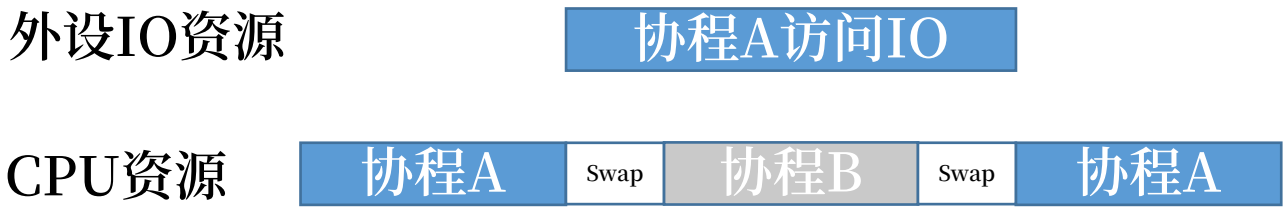
# 实验2：coroutinelab

2024年11月

# 从线程到协程

- 线程的切换是慢的
  - 内核态，OS 调度，>1000cycle
- 线程内有频繁的阻塞
  - cache miss, 网络 I/O, 磁盘 I/O
- 细粒度的切换模式：协程
  - 用户态切换，～100cyc
  - 执行其它就绪的逻辑，遮盖短时的阻塞

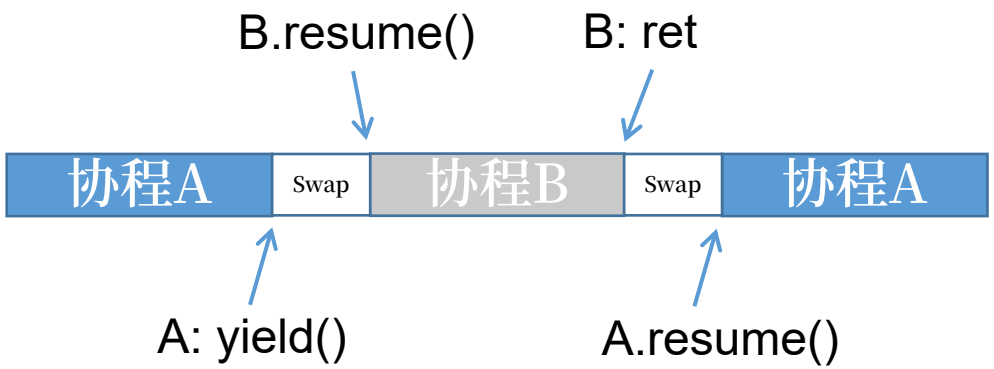


# 用户态切换

- 一个函数的执行 上下文/现场/context：
  1. 寄存器
  2. 栈帧

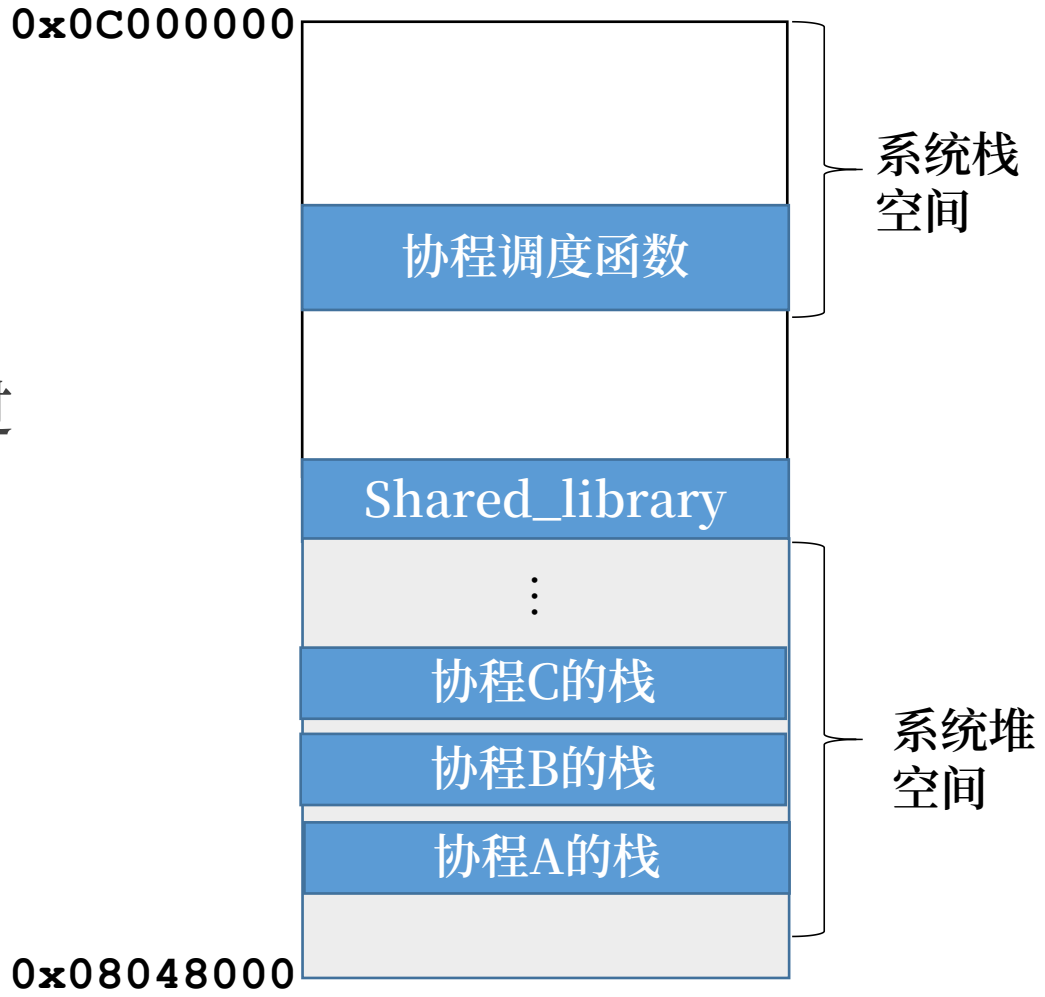
- 通过保存/恢复 context，完成切换
  - coroutine_switch

    - 调度器主协程
    - yield 切出
    - resume 切入



# 实验内容

- Task 1：实现一个有栈协程库
  - 使用汇编&C++代码完善基础协程库
- Task 2：完善功能
  - 完成 sleep 函数和 ready 参数的支持，通过 sleep_sort 测试
- Task 3：使用协程优化二分查找



# 实验流程

1. 下载项目，阅读代码逻辑，阅读 README.md 要求，完成实验

2. 编写实验报告
   - Markdown/PDF，不超过7页

3. 将整个项目与实验报告打包成zip格式上传网络学堂

```
coroutine/
├── bin                    # 编译结果
├── inc                    # 协程库
│   ├── common.h           # 功能函数
│   ├── context.h          # 协程 context 定义
│   └── coroutine_pool.h # 调度器
├── lib/context.S          # 协程切换的汇编实现
├── Makefile
├── README.md
└── src                    # 三个 Task 的主逻辑
    ├── binary_search.cpp
    ├── sample.cpp
    └── sleep_sort.cpp
```

# Task 1：完善协程库（65分）

1. 调度（15分）
   - coroutine_pool::serial_execute_all
   - 选择一个协程开始执行 （完善并调用该协程的 resume 函数）
2. 切换（20分）
   - lib/context.S 中完成 coroutine_switch，实现上下文切换
3. 主动切出（10分）
   - common.h 完成 yield 函数，实现当前协程切出
4. 实验报告额外要求（20分）：
   - 绘制协程切换时，栈的变化过程
   - 分析源代码，详细请见 代码注释 及 README.md

**测试：./bin/sample**
**期望结果**

```
in show(): 0
in show(): 0
in show(): 1
in show(): 1
in show(): 2
in show(): 2
in show(): 3
in show(): 3
in show(): 4
in show(): 4
in main(): 0
in main(): 0
in main(): 1
in main(): 1
in main(): 2
in main(): 2
in main(): 3
in main(): 3
in main(): 4
in main(): 4
```

# Task 2：sleep_sort（20分）

- 由于所有协程属于同一线程，所以不能调用系统sleep函数，会阻塞整个线程
- 所以协程需要有自己的sleep函数，具体做法是：
  - 协程需要设置一个ready标记，如果ready才能重新调度
  - 如果非ready，需要查询是否已经睡眠足够时间

1. 完善 sleep 函数（10分）
2. 修改coroutine_pool::serial_execute_all，支持 ready 逻辑（10分）

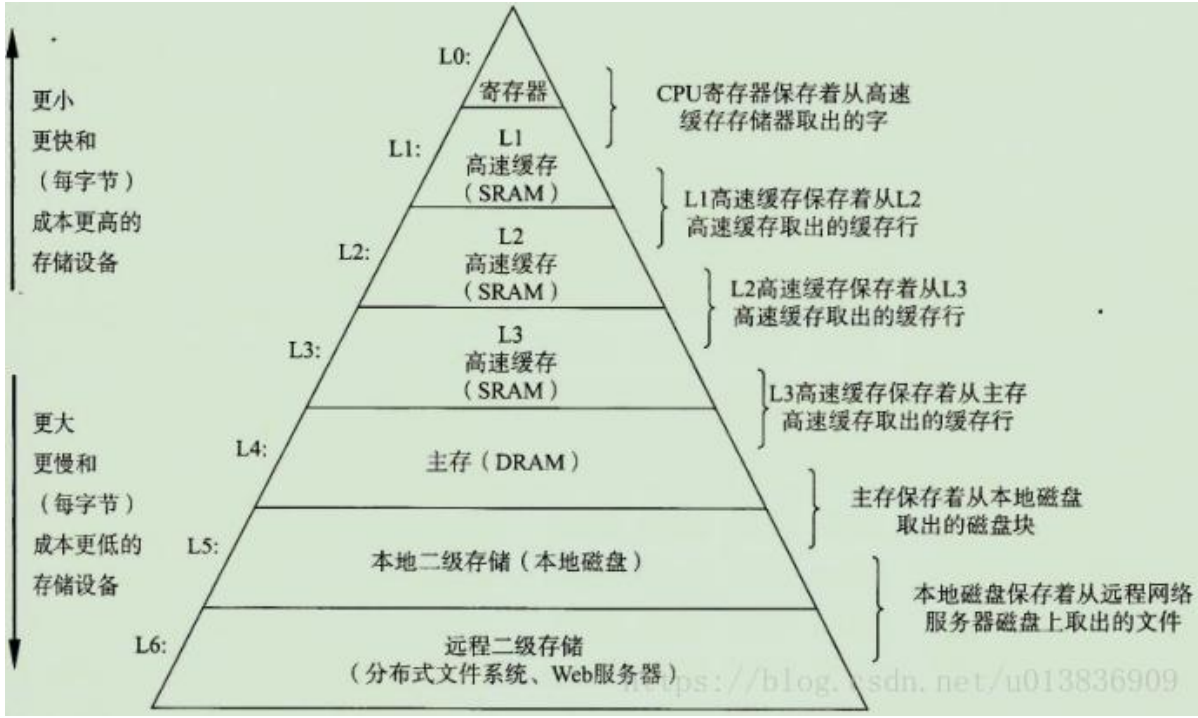
**测试：./bin/sleep_sort**
**期望结果**

```
$ ./bin/sleep_sort
5
1 3 4 5 2
1
2
3
4
5
```

# Task 3：优化二分算法（15分）

- 二分搜索核心函数需要频繁的不规则访存，会导致cache miss，影响程序执行效率
- 通过 **预取后切出** 优化搜索函数
  - 预取函数：__builtin_prefetch(addr)

1. 在src/binary_search.cpp 中使用协程优化（5分）
2. 在实验报告中性能分析（10分）

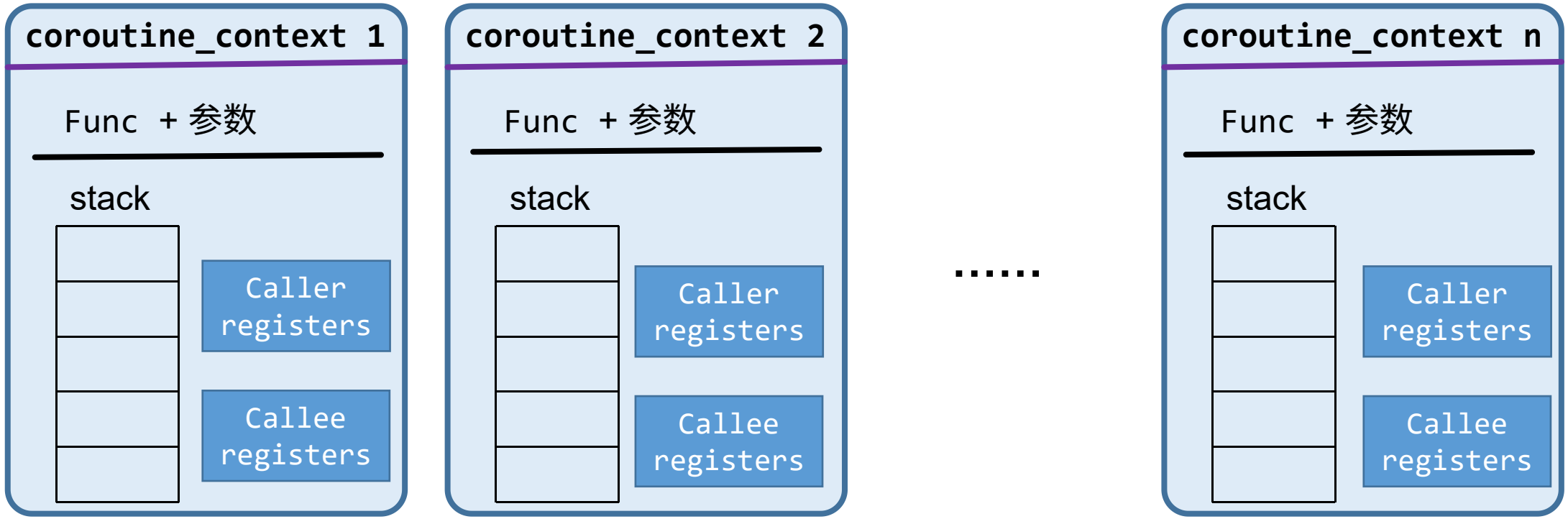



**测试：`./bin/binary_search`**

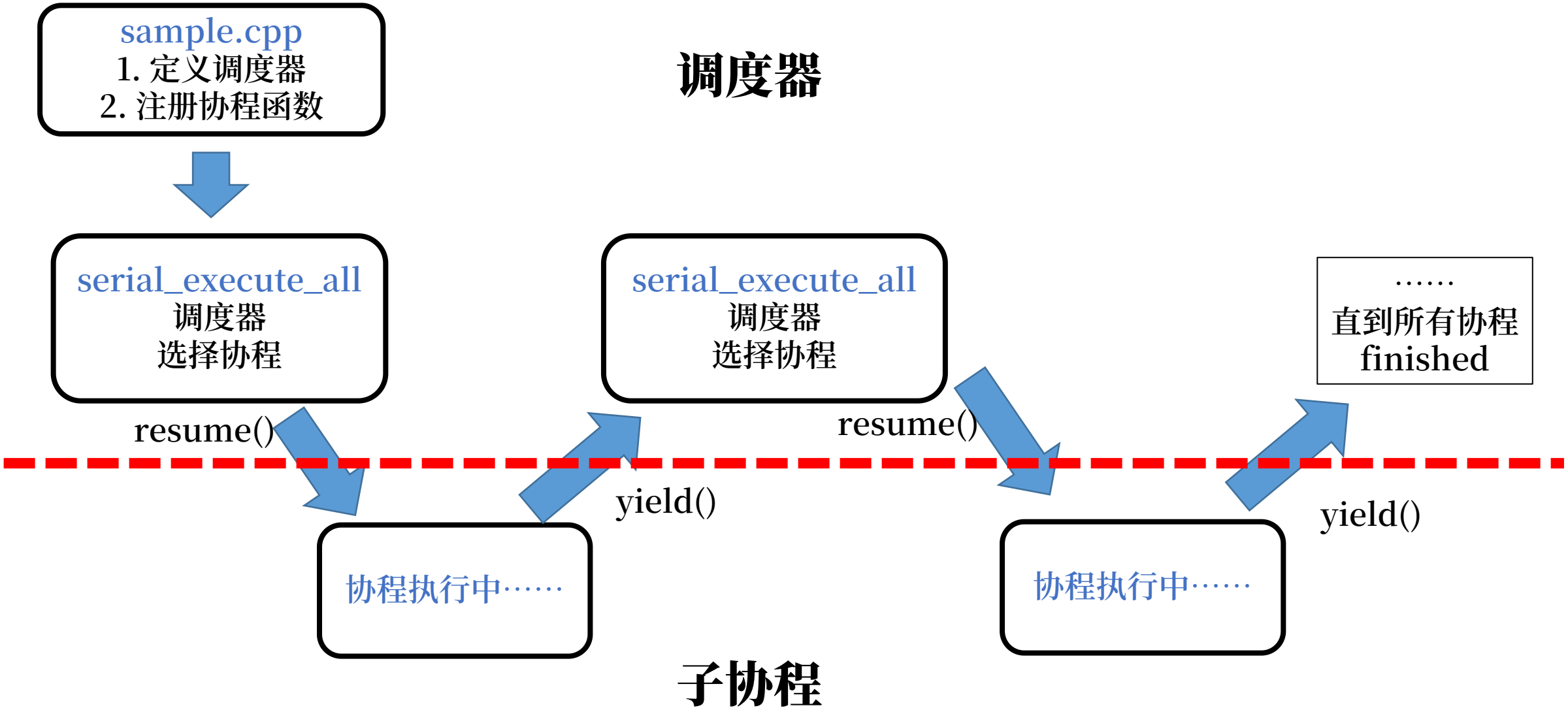
# 实验框架导读

# 协程框架导读

- inc/coroutine_pool.h: coroutine_pool # 协程池、调度器
  - vector<basic_context*> coroutines;
  - context_id # 正在执行的协程ID



# Task 1 执行流程


sample.cpp
1. 定义调度器
2. 注册协程函数
调度器
serial_execute_all
调度器
选择协程
resume()
协程执行中……
yield()
serial_execute_all
调度器
选择协程
resume()
协程执行中……
yield()
……
直到所有协程
finished
子协程

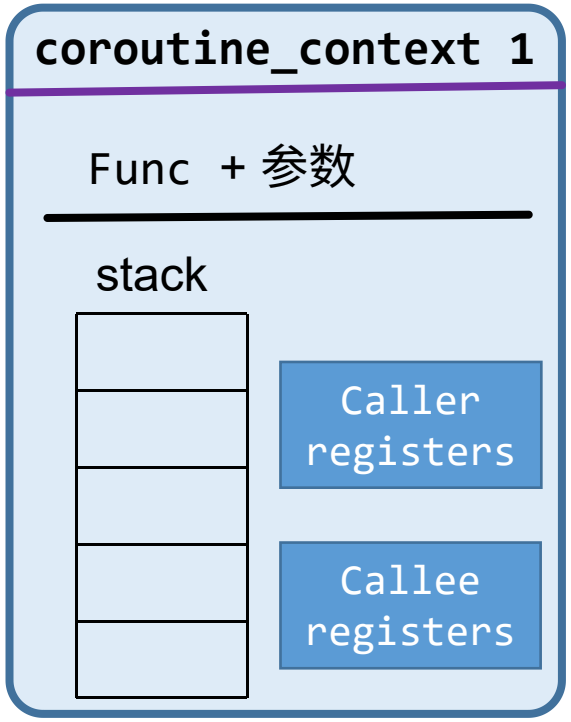
# 调度器

```
# 对于 Task 1
void serial_execute_all() {
    while(存在协程未结束):
        for(轮询):
            if(!finished):
                resume
                更改正在执行ID
}
```

coroutine_pool.h inc/coroutine_pool.h/coroutine_pool/serial_execute_all()

```
14  struct coroutine_pool {
49      /**
50       * @brief 以协程执行的方式串行并同时执行所有协程函数
51       * TODO: Task 1, Task 2
52       * 在 Task 1 中，我们不需要考虑协程的 ready
53       * 属性，即可以采用轮询的方式挑选一个未完成执行的协程函数进行继续执行的操作。
54       * 在 Task 2 中，我们需要考虑 sleep 带来的 ready
55       * 属性，需要对协程函数进行过滤，选择 ready 的协程函数进行执行。
56       *
57       * 当所有协程函数都执行完毕后，退出该函数。
58       */
59      void serial_execute_all() {
60          is_parallel = false;
61          g_pool = this;
62
63          for (auto context : coroutines) {
64              delete context;
65          }
66          coroutines.clear();
67      }
68  };
69
```

# 协程状态




C context.h inc/context.h/...

```
33  struct basic_context {
34      uint64_t *stack;
35      uint64_t stack_size;
36      uint64_t caller_registers[(int)Registers::RegisterCount];
37      uint64_t callee_registers[(int)Registers::RegisterCount];
38      bool finished;
39      bool ready;
40      std::function<bool()> ready_func;
41
42      basic_context(uint64_t stack_size)
43          : finished(false), ready(true), stack_size(stack_size) {
44          stack = new uint64_t[stack_size];
```

- *stack 是一个指向一段堆区的指针（上图为方便表示放在context里）
- caller_registers是调用者（调度器）的寄存器值暂存区
- callee_registers是被调者（当前协程）的寄存器值暂存区

# 协程切换的封装

- `void coroutine_switch(uint64_t *save, uint64_t *restore);`
- 切入（`resume`）：
  - 现在运行的是调度器，当选择一个协程执行之后：
  - 把调度器目前的寄存器值暂存入 `caller_registers` 数组
  - 把 `callee_registers` 暂存的寄存器值恢复
  - `coroutine_switch(caller_registers, callee_registers);`

- 切出（`yield`）：
  - 现在执行的是某协程，切回到调度器后：
  - 把现在（协程）的寄存器值暂存入 `callee_registers`
  - 把 `caller_registers` 暂存的寄存器值恢复
  - `coroutine_switch(callee_registers, caller_registers);`

# 协程切换本身

```
coroutine_switch(
    callee_registers, # %rdi
    caller_registers  # %rsi
);
```

- 需要保存/恢复所有 callee-saved 寄存器
- 例如处理 `%rbx`，索引是 `9`
  - `...`
  - `movq %rbx, 72(%rdi) # 保存`
  - `...`
  - `movq 72(%rsi), %rbx # 恢复`
  - `...`
  - `jmpq *120(%rsi)`

ASM context.S lib/context.S

```
7   coroutine_switch:
8       # TODO: Task 1
9       # 保存 callee-saved 寄存器到 %rdi 指向的上下文
10      # 保存的上下文中 rip 指向 ret 指令的地址（.coroutine_ret）
11
12      # 从 %rsi 指向的上下文恢复 callee-saved 寄存器
13      # 最后 jmpq 到上下文保存的 rip
14
15  .coroutine_ret:
16      ret
```

C context.h inc/context.h/...

```
8   enum class Registers : int {
9     RAX = 0,
10    RDI,
11    RSI,
12    RDX,
13    R8,
14    R9,
15    R10,
16    R11,
17    RSP,
18    RBX,
19    RBP
```

# 协程的注册

- new coroutine调用了协程的构造函数并在堆区为协程申请了空间

- 协程的构造函数
  - 给每个协程 new 一段栈空间
  - 初始化部分 callee_registers
    - %rsp 指向"栈帧"末尾
    - %rip 设置为（coroutine_entry）函数地址
    - %r12 设置为（coroutine_main）
    - %r13 设置为（本协程指针）

context.h inc/context.h/ coroutine_main(basic_context *)

```
33  struct basic_context {
42      basic_context(uint64_t stack_size)
43          : finished(false), ready(true), stack_size(stack_size) {
44          stack = new uint64_t[stack_size];
45
46          // TODO: Task 1
47          // 在实验报告中分析以下代码
48          // 对齐到 16 字节边界
49          uint64_t rsp = (uint64_t)&stack[stack_size - 1];
50          rsp = rsp - (rsp & 0xF);
51
52          void coroutine_main(struct basic_context * context);
53
54          callee_registers[(int)Registers::RSP] = rsp;
55          // 协程入口是 coroutine_entry
56          callee_registers[(int)Registers::RIP] = (uint64_t)coroutine_entry;
57          // 设置 r12 寄存器为 coroutine_main 的地址
58          callee_registers[(int)Registers::R12] = (uint64_t)coroutine_main;
59          // 设置 r13 寄存器，用于 coroutine_main 的参数
60          callee_registers[(int)Registers::R13] = (uint64_t)this;
61      }
```

# 协程的执行

- `%rsp` 指向“栈帧”末尾
- `%rip` 设置为（`coroutine_entry`）函数地址
- `%r12` 设置为（`coroutine_main`）
- `%r13` 设置为（本协程指针）

ASM context.S lib/context.S

```
1   .global coroutine_entry
2   coroutine_entry:
3       movq %r13, %rdi
4       callq *%r12
```

- 首次调用 **`resume()`**，恢复到保存的 **`%rip`**，即 **`coroutine_entry`**
- **`coroutine_entry`** 使用 本协程指针 为参数调用 **`coroutine_main`**

C context.h inc/context.h/ coroutine_main(basic_context *)

```
71  void coroutine_main(struct basic_context *context) {
72    context->run();
73    context->finished = true;
74    coroutine_switch(context->callee_registers, context->caller_registers);
75
76    // unreachable
77    assert(false);
78  }
```

# Q&A