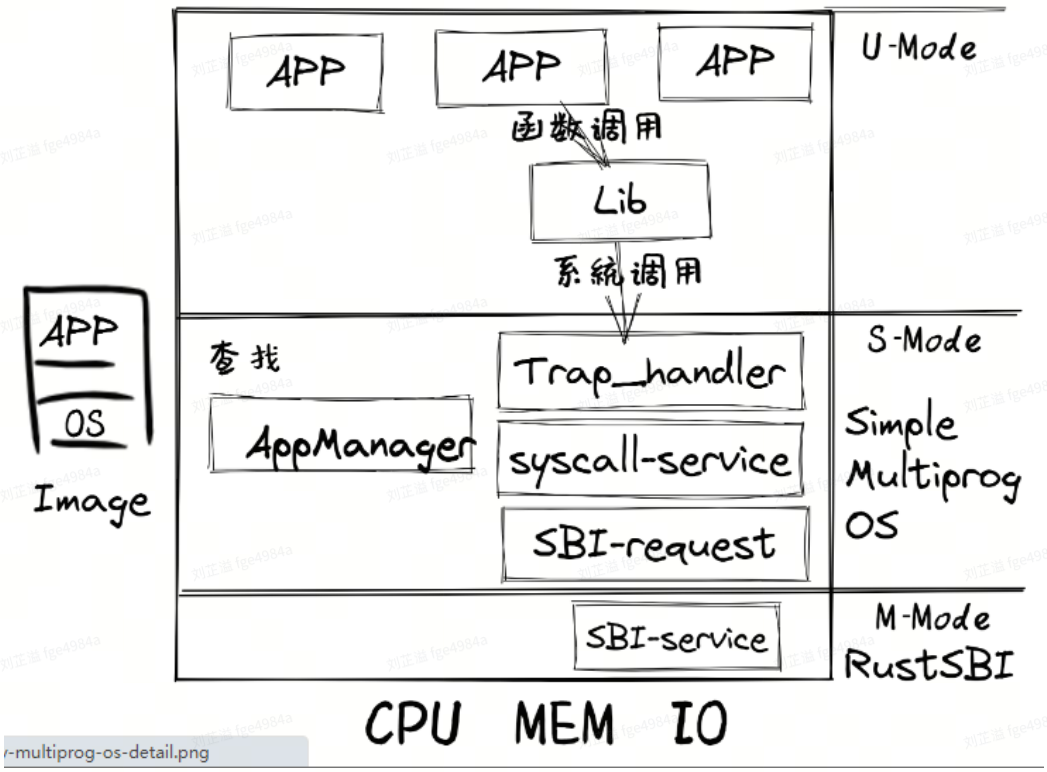
TimeSharingOS

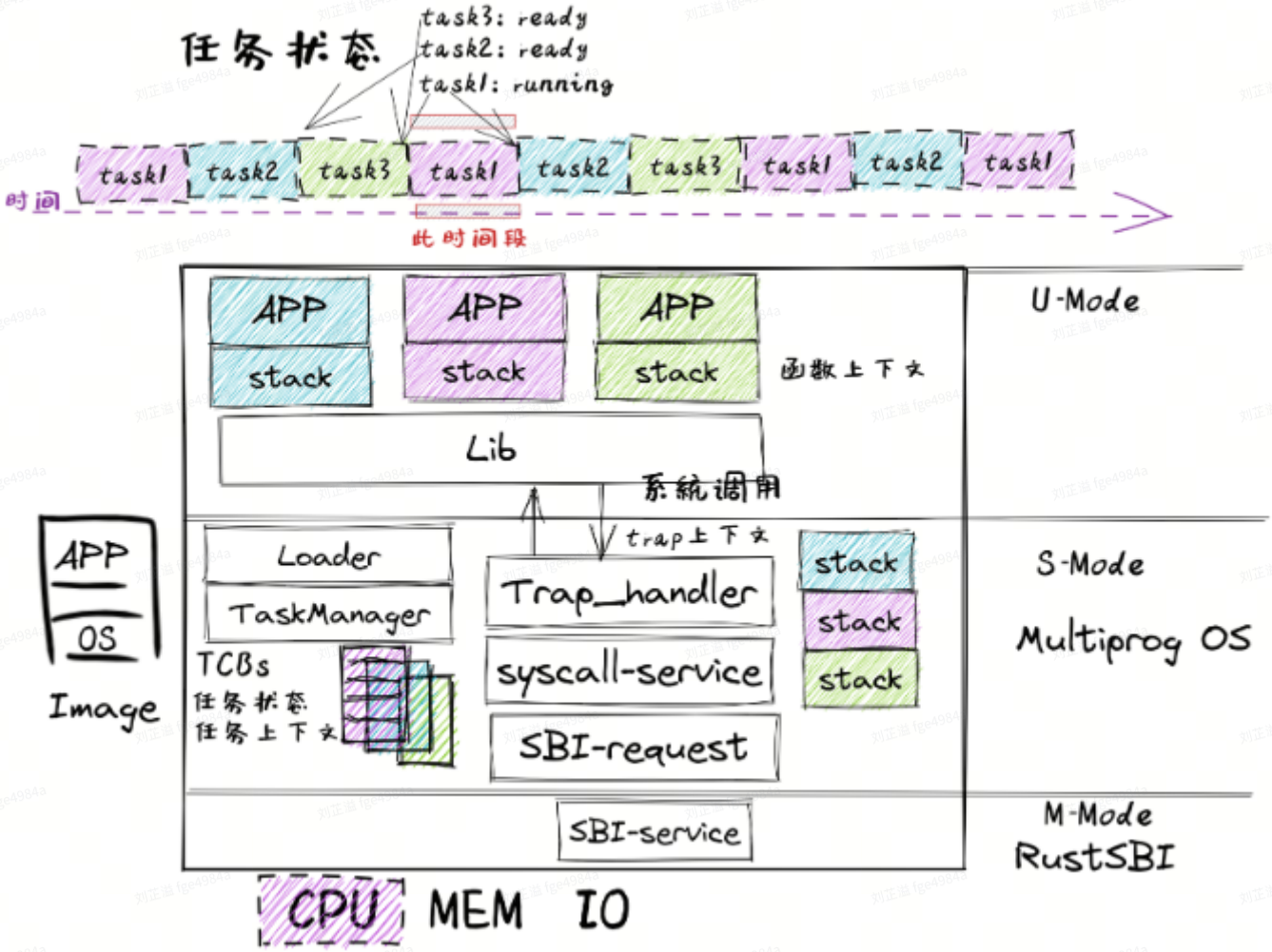
TimesharingOS

MultiprogOS



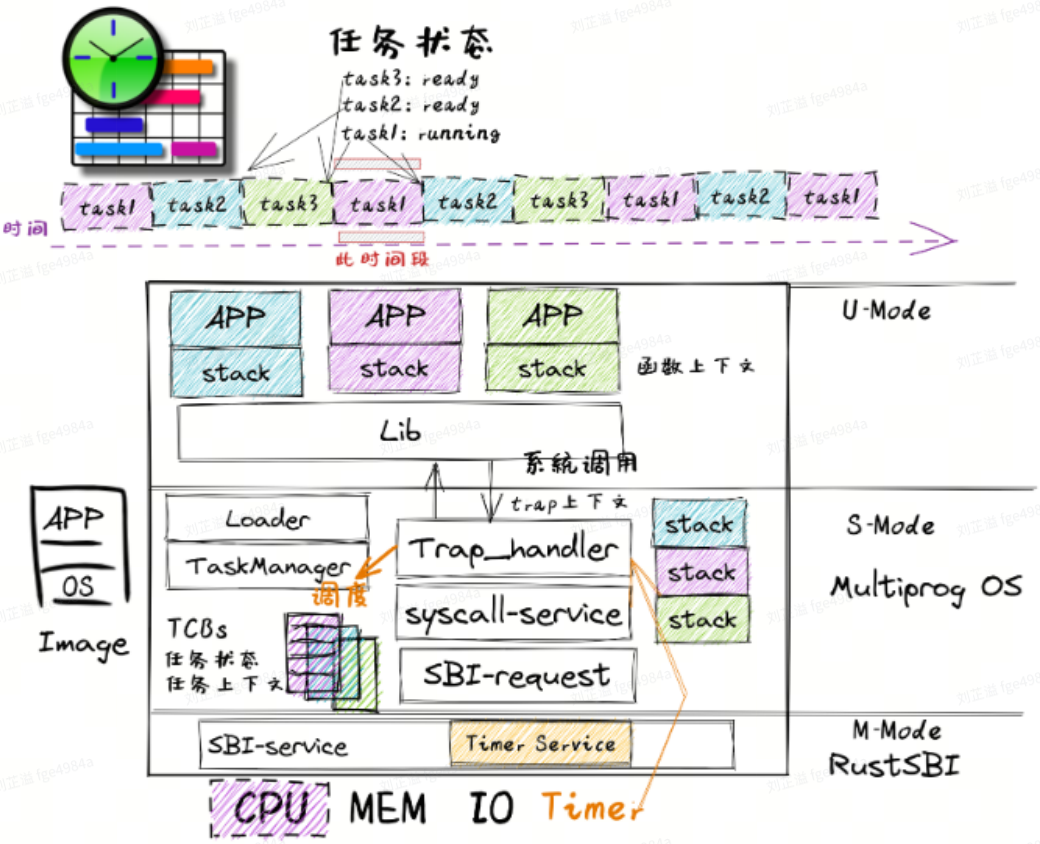
- Qemu把包含多个app的列表和MultiprogOS的image镜像加载到内存中，RustSBI（bootloader）完成基本的硬件初始化后，跳转到MultiprogOS起始位置
- MultiprogOS首先进行正常运行前的初始化工作，即建立栈空间和清零bss段，然后通过改进的AppManager 内核模块从app列表中把**所有app都加载到内存**中，并按指定顺序让app在用户态一个接一个地执行。
- app在执行过程中，会通过系统调用的方式得到MultiprogOS提供的OS服务，如输出字符串等。

CoopOS



- CoopOS进一步改进了 AppManager 内核模块，把它拆分为负责**加载应用的 Loader 内核模块**和管理应用运行过程的 TaskManager 内核模块。
- TaskManager 通过 task 任务控制块来管理应用程序的执行过程，**支持应用程序主动放弃 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。
- 应用程序在运行时有自己所在的内存空间和栈，确保被切换时相关信息不会被其他应用破坏。如果当前应用程序正在运行，则该应用对应的任务处于运行（Running）状态；如果该应用主动放弃处理器，则该应用对应的任务处于就绪（Ready）状态。
- 操作系统进行任务切换时，需要把要暂停任务的上下文（即任务用到的通用寄存器）保存起来，把要继续执行的任务的上下文恢复为暂停前的内容，这样就能让不同的应用协同使用处理器了。

TimersharingOS



- TimesharingOS最大的变化是改进了 Trap_handler 内核模块，**支持时钟中断**，从而可以抢占应用的执行。
- 并通过进一步改进 TaskManager 内核模块，提供**任务调度功能**，这样可以在收到时钟中断后统计任务的使用时间片，如果任务的时间片用完后，则切换任务。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。

多道程序放置与加载

放置

- 每个app的起始地址为 `hex(base_address+step*app_id)` ，实现起始地址的不同

```python
# user/build.py

import os

base_address = 0x80400000
step = 0x20000
linker = 'src/linker.ld'

app_id = 0
apps = os.listdir('src/bin')
apps.sort()
for app in apps:
    app = app[:app.find('.')]
```

```
        lines = []
        lines_before = []
        with open(linker, 'r') as f:
            for line in f.readlines():
                lines_before.append(line)
                line = line.replace(hex(base_address),
    hex(base_address+step*app_id))
                lines.append(line)
        with open(linker, 'w+') as f:
            f.writelines(lines)
        os.system('cargo build --bin %s --release' % app)
        print('[build.py] application %s start with address %s' %(app,
    hex(base_address+step*app_id)))
        with open(linker, 'w+') as f:
            f.writelines(lines_before)
        app_id = app_id + 1
```

加载

- 第i个应用加载到不同的物理地址上 `APP_BASE_ADDRESS + i * APP_SIZE_LIMIT`

```
// os/src/loader.rs

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

pub fn load_apps() {
    extern "C" { fn _num_app(); }
    let num_app_ptr = _num_app as usize as *const usize;
    let num_app = get_num_app();
    let app_start = unsafe {
        core::slice::from_raw_parts(num_app_ptr.add(1), num_app + 1)
    };
    // load apps
    for i in 0..num_app {
        let base_i = get_base_i(i);
        // clear region
        (base_i..base_i + APP_SIZE_LIMIT).for_each(|addr| unsafe {
            (addr as *mut u8).write_volatile(0)
        });
        // load app from data section to memory
        let src = unsafe {
            core::slice::from_raw_parts(
                app_start[i] as *const u8,
                app_start[i + 1] - app_start[i]
```

```
            )
        };
        let dst = unsafe {
            core::slice::from_raw_parts_mut(base_i as *mut u8, src.len())
        };
        dst.copy_from_slice(src);
    }
    unsafe {
        asm!("fence.i");
    }
}
```

任务切换

- 在内核中这种机制是在 `__switch` 函数中实现的。 任务切换支持的场景是：一个应用在运行途中便会主动或被动交出 CPU 的使用权，此时它只能暂停执行，等到内核重新给它分配处理器资源之后才能恢复并继续执行。
- 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。当一个应用 Trap 到 S 模式的操作系统内核中进行进一步处理（即进入了操作系统的 Trap 控制流）的时候，其 Trap 控制流可以调用一个特殊的 `__switch` 函数。
- 这个函数表面上就是一个普通的函数调用：在 `__switch` 返回之后，将继续从调用该函数的位置继续向下执行。但是其间却隐藏着复杂的控制流切换过程。具体来说，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*。
  - 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。
  - 不过，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**。

设计与实现

- 对于当前正在执行的任务的 Trap 控制流，我们用一个名为 `current_task_cx_ptr` 的变量来保存放置当前任务上下文的地址；而用 `next_task_cx_ptr` 的变量来保存放置下一个要执行任务的上下文的地址。利用 C 语言的引用来描述的话就是：

```
TaskContext *current_task_cx_ptr = &tasks[current].task_cx;
TaskContext *next_task_cx_ptr    = &tasks[next].task_cx;
```

- 任务切换包含4个阶段

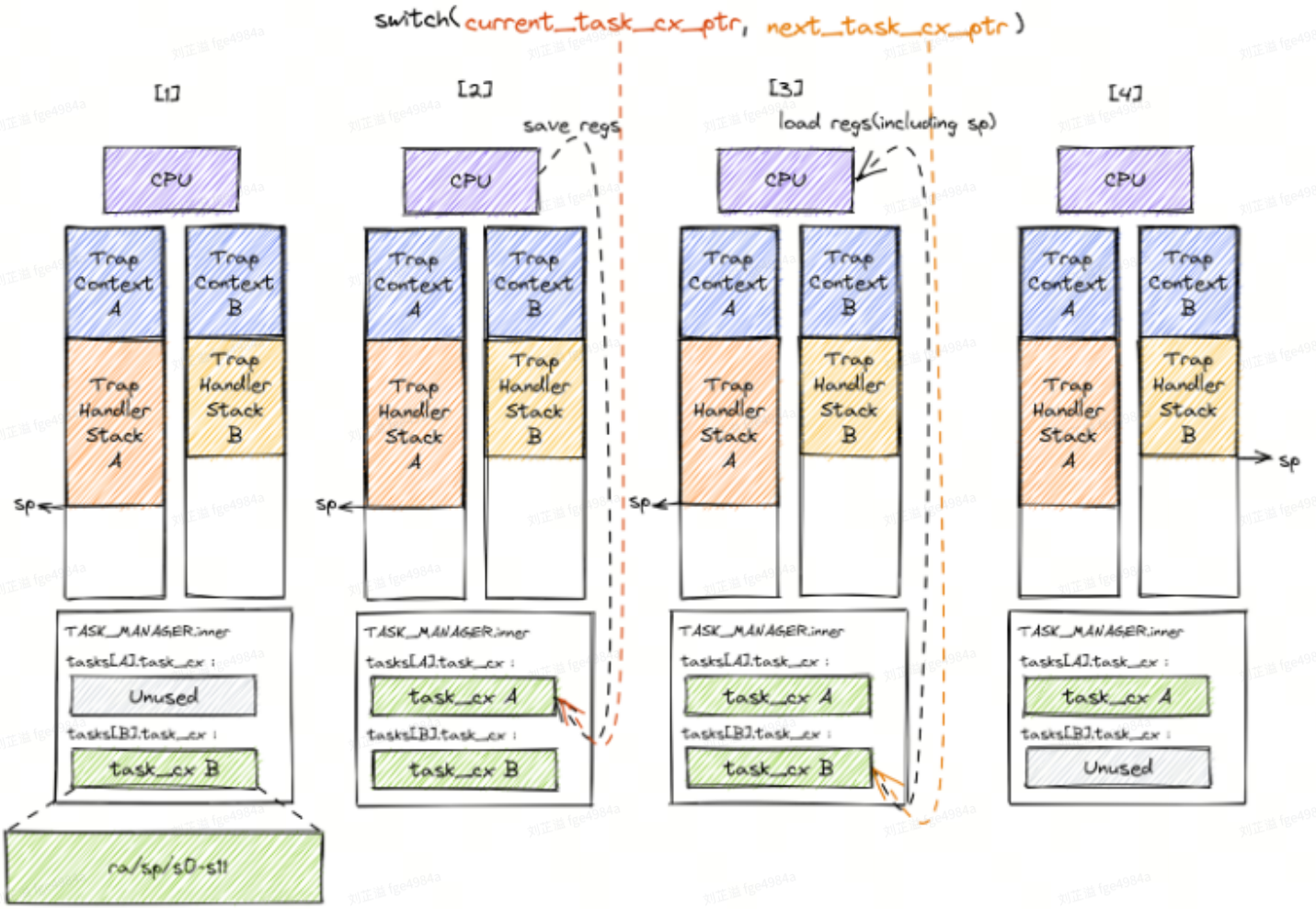
a. 在 Trap 控制流 A 调用 `__switch` 之前，A 的内核栈上只有 Trap 上下文和 Trap 处理函数的调用栈信息，而 B 是之前被切换出去的；

b. A 在 A 任务上下文空间在里面保存 CPU 当前的寄存器快照；

c. 读取 `next_task_cx_ptr` 指向的 B 任务上下文，根据 B 任务上下文保存的内容来恢复 `ra` 寄存器、`s0~s11` 寄存器以及 `sp` 寄存器。只有这一步做完后，`__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* 。

d. 上一步寄存器恢复完成后，可以看到通过恢复 `sp` 寄存器换到了任务 B 的内核栈上，进而实现了控制流的切换。这就是为什么 `__switch` 能做到一个函数跨两条控制流执行。此后，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。

- 从结果来看，我们看到 A 控制流 和 B 控制流的状态发生了互换， A 在保存任务上下文之后进入暂停状态，而 B 则恢复了上下文并在 CPU 上继续执行。



__switch实现

```
1  // os/src/task/switch.rs
2
3  global_asm!(include_str!("switch.S"));
4
```

```rust
use super::TaskContext;

extern "C" {
    pub fn __switch(
        current_task_cx_ptr: *mut TaskContext,
        next_task_cx_ptr: *const TaskContext
    );
}
```

```
# os/src/task/switch.S

.altmacro
.macro SAVE_SN n
    sd s\n, (\n+2)*8(a0)
.endm
.macro LOAD_SN n
    ld s\n, (\n+2)*8(a1)
.endm
    .section .text
    .globl __switch
__switch:
    # 阶段 [1]
    # __switch(
    #     current_task_cx_ptr: *mut TaskContext,
    #     next_task_cx_ptr: *const TaskContext
    # )
    # 阶段 [2]
    # save kernel stack of current task
    sd sp, 8(a0)
    # save ra & s0~s11 of current execution
    sd ra, 0(a0)
    .set n, 0
    .rept 12
        SAVE_SN %n
        .set n, n + 1
    .endr
    # 阶段 [3]
    # restore ra & s0~s11 of next execution
    ld ra, 0(a1)
    .set n, 0
    .rept 12
        LOAD_SN %n
        .set n, n + 1
    .endr
    # restore kernel stack of next task
```

```
    ld sp, 8(a1)
    # 阶段 [4]
    ret
```

TaskContext

```rust
// os/src/task/context.rs

pub struct TaskContext {
    ra: usize,
    sp: usize,
    s: [usize; 12],
}
```

任务管理器

任务控制块(TCB)

```rust
// os/src/task/task.rs

#[derive(Copy, Clone)]
pub struct TaskControlBlock {
    pub task_status: TaskStatus,
    pub task_cx: TaskContext,
}
```

- `TaskContext::goto_restore` 将 `ra` 设置为 `__restore` ；因为切换任务时我们设定内核栈栈中包含 `__alltraps` 和 `trap_handler` 的栈帧
- 当执行第一个程序时，我们需要向内核栈中压入一个 `TrapContext`

```rust
// os/src/loader.rs

pub fn init_app_cx(app_id: usize) -> usize {
    KERNEL_STACK[app_id].push_context(
        TrapContext::app_init_context(get_base_i(app_id),
USER_STACK[app_id].get_sp()),
    )
}
// os/src/task/mod.rs

```

```rust
pub struct TaskManager {
    num_app: usize,
    inner: UPSafeCell<TaskManagerInner>,
}

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

lazy_static! {
    pub static ref TASK_MANAGER: TaskManager = {
        let num_app = get_num_app();
        let mut tasks = [
            TaskControlBlock {
                task_cx: TaskContext::zero_init(),
                task_status: TaskStatus::UnInit
            };
            MAX_APP_NUM
        ];
        for i in 0..num_app {
            tasks[i].task_cx = TaskContext::goto_restore(init_app_cx(i));
            tasks[i].task_status = TaskStatus::Ready;
        }
        TaskManager {
            num_app,
            inner: unsafe { UPSafeCell::new(TaskManagerInner {
                tasks,
                current_task: 0,
            })},
        }
    };
}
```

sys_yield和sys_exit系统调用

```rust
// os/src/syscall/process.rs

use crate::task::suspend_current_and_run_next;

pub fn sys_yield() -> isize {
    suspend_current_and_run_next();
    0
}
// os/src/syscall/process.rs
```

```rust

use crate::task::exit_current_and_run_next;

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!");
}
```

- `suspend_current_and_run_next` 和 `exit_current_and_run_next` 均是切换当前Task的运行状态，切换到下一个应用

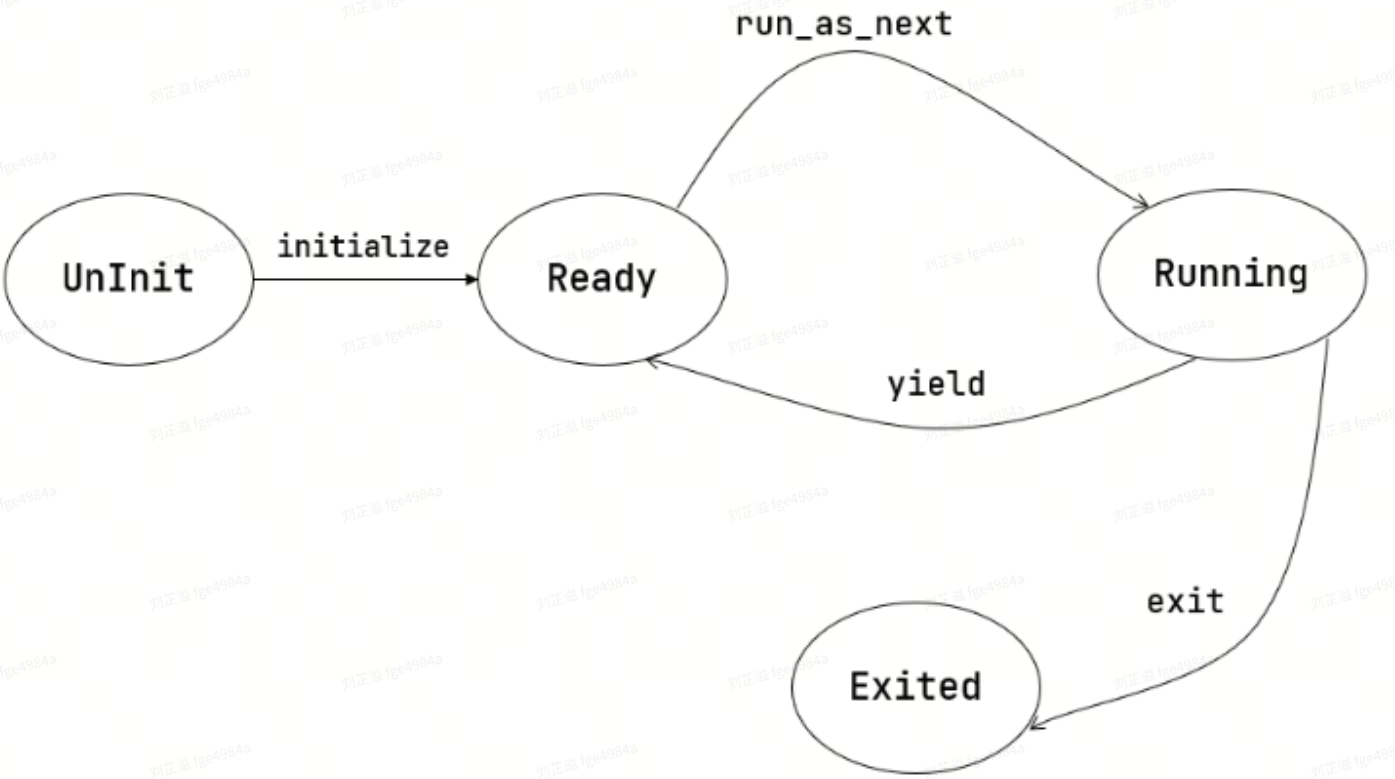
```rust
// os/src/task/mod.rs

fn mark_current_suspended() {
    TASK_MANAGER.mark_current_suspended();
}

fn mark_current_exited() {
    TASK_MANAGER.mark_current_exited();
}

impl TaskManager {
    fn mark_current_suspended(&self) {
        let mut inner = self.inner.borrow_mut();
        let current = inner.current_task;
        inner.tasks[current].task_status = TaskStatus::Ready;
    }

    fn mark_current_exited(&self) {
        let mut inner = self.inner.borrow_mut();
        let current = inner.current_task;
        inner.tasks[current].task_status = TaskStatus::Exited;
    }
}
```



run_next_task

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

fn run_next_task() {
    TASK_MANAGER.run_next_task();
}

impl TaskManager {
    fn run_next_task(&self) {
        if let Some(next) = self.find_next_task() {
            let mut inner = self.inner.exclusive_access();
            let current = inner.current_task;
            inner.tasks[next].task_status = TaskStatus::Running;
            inner.current_task = next;
            let current_task_cx_ptr = &mut inner.tasks[current].task_cx as
*mut TaskContext;
            let next_task_cx_ptr = &inner.tasks[next].task_cx as *const
TaskContext;
            drop(inner);
            // before this, we should drop local variables that must be
dropped manually
            unsafe {
                __switch(
                    current_task_cx_ptr,
                    next_task_cx_ptr,
```

```
                );
            }
            // go back to user mode
        } else {
            panic!("All applications completed!");
        }
    }
}
```

时间片轮转调度(Round-Robin)

时钟中断与计时器

- 在 RISC-V 64 架构上，该计数器保存在一个 64 位的 CSR `mtime` 中，我们无需担心它的溢出问题，在**内核运行全程可以认为它是一直递增的**。
- 另外一个 64 位的 CSR `mtimecmp` 的作用是：一旦计数器 `mtime` 的值超过了 `mtimecmp`，就会触发一次时钟中断。这使得我们可以方便的通过设置 `mtimecmp` 的值来决定下一次时钟中断何时触发。
- 可惜的是，它们都是 M 特权级的 CSR，而我们的内核处在 S 特权级，是不被允许直接访问它们的。好在运行在 M 特权级的 SEE （这里是RustSBI）已经预留了相应的接口，我们可以调用它们来间接实现计时器的控制
- 常数 `CLOCK_FREQ` 是一个预先获取到的各平台不同的时钟频率，单位为赫兹，也就是一秒钟之内计数器的增量。

```
// os/src/timer.rs

use riscv::register::time;

pub fn get_time() -> usize {
    time::read()
}
// os/src/sbi.rs

const SBI_SET_TIMER: usize = 0;

pub fn set_timer(timer: usize) {
    sbi_call(SBI_SET_TIMER, timer, 0, 0);
}
```

- `set_next_trigger` 设置下一个时钟中断
- `get_time_us` 以微秒为单位返回当前计数器的值

```rust
// os/src/timer.rs

use crate::config::CLOCK_FREQ;
const TICKS_PER_SEC: usize = 100;

pub fn set_next_trigger() {
    set_timer(get_time() + CLOCK_FREQ / TICKS_PER_SEC);
}

// os/src/timer.rs

const MICRO_PER_SEC: usize = 1_000_000;

pub fn get_time_us() -> usize {
    time::read() / (CLOCK_FREQ / MICRO_PER_SEC)
}
```

抢占式调度

```rust
// os/src/trap/mod.rs

match scause.cause() {
    Trap::Interrupt(Interrupt::SupervisorTimer) => {
        set_next_trigger();
        suspend_current_and_run_next();
    }
}
```

- 我们只需在 `trap_handler` 函数下新增一个条件分支跳转，当发现触发了一个 S 特权级时钟中断的时候，首先重新设置一个 10ms 的计时器，然后调用上一小节提到的 `suspend_current_and_run_next` 函数暂停当前应用并切换到下一个。
- 初始化设置

```rust
// os/src/main.rs

#[no_mangle]
pub fn rust_main() -> ! {
    clear_bss();
    println!("[kernel] Hello, world!");
    trap::init();
    loader::load_apps();
```

```
    trap::enable_timer_interrupt();
    timer::set_next_trigger();

    task::run_first_task();
    panic!("Unreachable in rust_main!");
}

// os/src/trap/mod.rs

use riscv::register::sie;

pub fn enable_timer_interrupt() {
    unsafe { sie::set_stimer(); }
}
```

sleep

- 目前在等待某些事件的时候仍然需要 `yield` ，其中一个原因是为了节约 CPU 计算资源，另一个原因是当事件依赖于其他的应用的时候，由于只有一个 CPU，当前应用的等待可能永远不会结束。**这种情况下需要先将它切换出去，使得其他的应用到达它所期待的状态并满足事件的生成条件，再切换回来**。
- 这里我们先通过 yield 来优化 **轮询** (Busy Loop) 过程带来的 CPU 资源浪费。在 `03sleep` 这个应用中：

```
// user/src/bin/03sleep.rs

#[no_mangle]
fn main() -> i32 {
    let current_timer = get_time();
    let wait_for = current_timer + 3000;
    while get_time() < wait_for {
        yield_();
    }
    println!("Test sleep OK!");
    0
}
```