

- RISC-V体系结构规定了三种特权模式：用户模式（User mode, U-mode）、监管者模式（Supervisor mode, S-mode）、机器模式（Machine mode, M-mode）。

- 本阶段目标：在上一阶段的基础上，实现进程的多种状态，以及多进程的调度切换。
 - (1) 完善进程体定义，实现相关系统调用
 - (2) 实现进程的调度
 - (3) 实现sleep和wakeup

实验六：进程管理

任务一：fork + exit + wait

- 首先删去上一次的临时系统调用：sys_copyin()、sys_copyout()、sys_copyinstr()。
- 添加新的系统调用 sys_print()、sys_fork()、sys_exit()、sys_wait()、sys_sleep()。

实验六：进程管理

/ 进程数组

```
static proc_t procs[NPROC];
```

// 第一个进程的指针

```
static proc_t* proczero;
```

// 全局的pid和保护它的锁

```
static int global_pid = 1;
```

```
static spinlock_t lk_pid;
```

```
...
```

由于要引入多个进程, 进程的定义在之前的基础上做了扩充

```
typedef struct proc {  
    spinlock_t lk;           // 自旋锁  
    /* 下面的五个字段需要持有锁才能修改 */  
    int pid;                 // 标识符  
    enum proc_state state;   // 进程状态  
    struct proc* parent;     // 父进程  
    int exit_state;          // 进程退出时的状态(父进程可能关心)  
    void* sleep_space;       // 睡眠是在等待什么  
  
    pgtbl_t pgtbl;           // 用户态页表  
    uint64 heap_top;         // 用户堆顶(以字节为单位)  
    uint64 ystack_pages;     // 用户栈占用的页面数量  
    mmap_region_t* mmap;     // 用户可映射区域的起始节点  
    trapframe_t* tf;         // 用户态内核态切换时的运行环境暂存空间  
  
    uint64 kstack;           // 内核栈的虚拟地址  
    context_t ctx;           // 内核态进程上下文  
} proc_t;
```

实验六 进程管理模块

首先实现以下三个函数：

- `proc_init()` 初始化一些全局变量, 设置`kstack`字段并调用 `proc_free()`
- `proc_alloc()` 从进程数组里申请一个进程, 并申请资源和初始化一些字段, 分配和初始化新的进程结构体, 包括分配PID、创建页表、设置内核栈等关键操作。
- `proc_free()` 向进程数组里归还一个进程, 并释放资源和清空一些字段
- 这三个函数是进程数组这一全局资源的控制函数, 因此将他们视为一组操作

实验六：进程管理

实现进程创建函数proc_fork()

1. 调用proc_alloc分配一个新的进程槽位
 2. 复制父进程的用户内存空间
 3. 复制父进程的陷阱帧(trapframe)
 4. 设置子进程的返回值为0
 5. 复制打开的文件描述符和当前工作目录
 6. 设置子进程状态为RUNNABLE
- 可以发现各个进程会组成一个由 **proczero** 作为根节点的进程树。
 - 这两个进程在用户态怎么区分呢？解决方案是让他们返回值不同
 - 对于父进程，它收到的返回值是子进程的pid，通过函数返回做到
 - 对于子进程，它收到的是0，通过修改 `p->tf->a0` 做到。

实验六：进程管理

完成 `proc_fork()` 后，进入下一组函数：

`proc_wait()` 等待子进程退出：负责扫描子进程并等待其终止。

`proc_free()` 回收子进程：释放ZOMBIE进程的所有资源，包括页表、陷阱帧等。

`proc_exit()` 进程退出：退出当前进程，并将其父进程唤醒，进入ZOMBIE 状态。

这三个函数构成一个常见的组合：

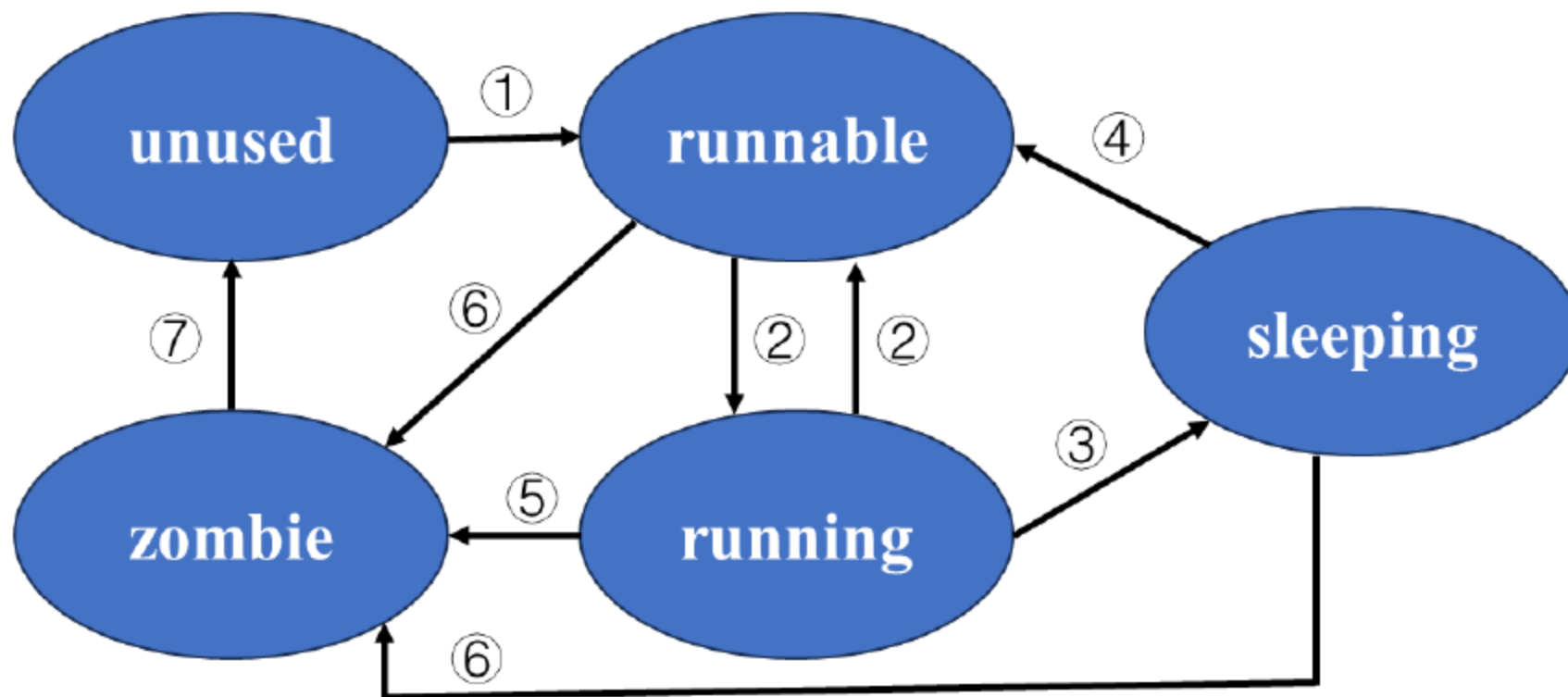
```
int pid = fork(); // 分支
if(pid == 0) { // 子进程
    do something ...
    exit(0);
} else { // 父进程
    int exit_state;
    wait(&exit_state);
    do something ....
}
```

- 这两个函数的实现过程会遇到两个问题：
 - 1. 由于进程的树形结构，如果出现父进程退出但子进程没退出的情况该怎么办呢？

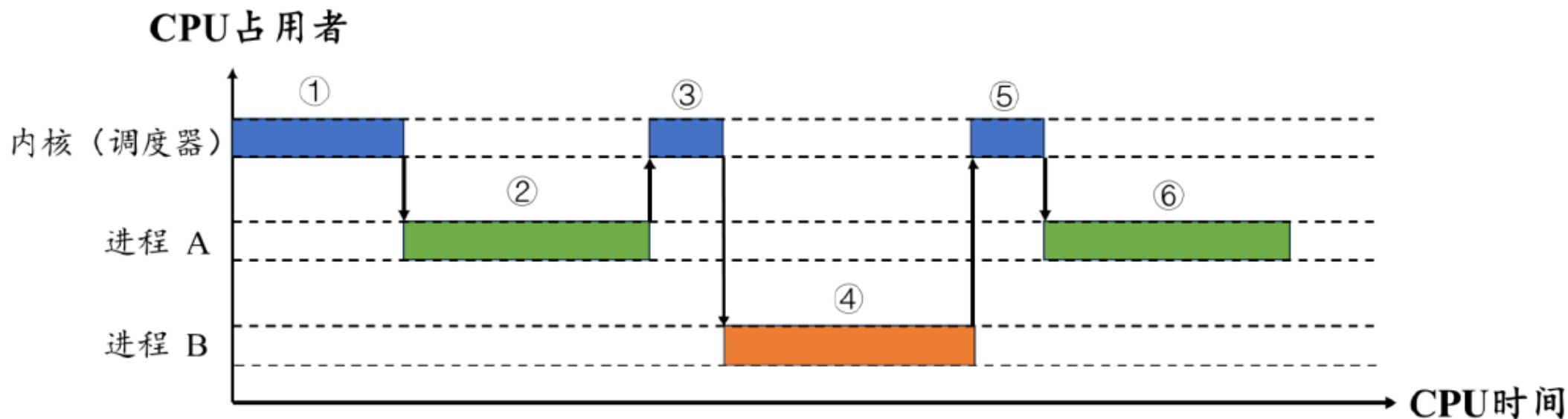
在 `proc_exit()` 函数里调用 `proc_reparent()` 将当前进程的孩子托付给 **proczero**，因为它是整棵进程树的根，是不会退出的。

- 2. 由于 `proc_wait()` 函数是一个循环，所以没等到子进程退出就会一直占用CPU，所以添加一个新的函数 `proc_yield()`，进程放弃CPU使用权进入调度

活跃度: **running** > **runnable** > **sleeping** > **zombie** > **unused**



- ① 空闲进程结构体被取出和初始化
- ② 进程调度：从可运行进程中选出一个换掉正在执行的进程
- ③ 进程等待某种资源或某个事件，进入睡眠状态
- ④ 进程得到所等待的东西后被唤醒
- ⑤ 进程执行完毕，主动杀死自己
- ⑥ 处于可执行状态或睡眠状态的进程被正在执行的进程杀死
- ⑦ 进程的父进程处理后事，释放濒死进程的资源



- ① 内核完成机器启动，打开调度器
- ② 进程A被调度，上CPU执行
- ③ 进程A的执行流被暂停，调度器选择一个可执行进程
- ④ 进程B被调度，上CPU执行
- ⑤ 进程B的执行流被暂停，调度器选择一个可执行进程
- ⑥ 进程A被调度，上CPU执行

任务二：进程调度

- 采用的进程调度算法是时间片轮转（Round Robin, RR）。算法思路如下：在初始化时为每个进程配置一个时间片；当某个进程处于running状态且发生时钟中断时，这个进程的时间片减1；当时间片减到0时，触发调度，将CPU使用权切换到调度器，同时重置该进程的时间片。

调度函数 `proc_sched()` 和 `proc_scheduler()`

这两个函数是调度的两个阶段：

- 第一阶段： `proc_sched()` 检查了一些前提条件（锁的状态、中断状态等），然后保存当前中断状态，进行上下文切换到调度器上下文，当再次被调度回来时恢复中断状态。
当前CPU的用户进程 -> CPU自己的进程(之前称之为高级进程)
 - 第二阶段： `proc_scheduler()` 需要挑选新的RUNNABLE用户进程, 这里采用各进程按顺序轮流执行的调度算法。CPU自己的进程 -> 被选中的新用户进程
- 当占用CPU的进程是CPU自己的进程时, **`mycpu()->proc = NULL`**

- 两个阶段的核心操作都是 `swtch()` 做上下文切换
- `swtch()`通过`ret`返回进程`kernel` 栈，如果该进程第一次调度，即返回到`fork_return()`,如果不是第一次调度，则返回到内核调用`proc_sched()`的地方。
- 当调度器启动后, CPU自己的进程就被困在调度器里了(死循环), 它的上下文就是调度器的运行环境。
- 在 `main()` 的最后, 各个CPU的进程都会进入调度器, 并忠实地留在这里。
- 于是进程切换的过程:`proc-1 -> CPU进程(调度器)-> proc-2`

可以总结出CPU自己的进程做的事：初始化OS各个模块, 然后作为用户进程调度器, 之后各个CPU的调度器会从进程数组里选择RUNNABLE的用户进程执行

实验六：进程管理

- 时间片到了，可以利用时钟中断实现这件事：在发生时钟中断后调用 `proc_yield()`：
 - `proc_yield()` 使本进程进入就绪态，并进而调用 `proc_sched()` 让出CPU。
 - 定时器中断触发进程切换是在 `trap_user_handler()` 和 `trap_kernel_handler()` 中处理。（原因？）
 - `timer_update()` 调用了 `wakeup()`，唤醒那些上了闹钟的进程。

实验六：进程管理

- sleep和wakeup
- 自旋锁在获取不到资源时，CPU会空转直到获取资源，所以就应该有另一种方法，在获取不到时，让别的进程执行，直到某个条件达成时，再返回去执行这个任务。
- XV6使用一种叫睡眠和唤醒的机制，这允许一个进程睡眠并等待一个事件，当事件发生时另一个进程来唤醒它。睡眠和唤醒通常称为序列协调或条件同步机制。
 - `proc_sleep()`：将调用的进程睡眠，释放CPU给其他进程工作。标记进程为SLEEPING，调用`proc_sched`来释放CPU。

实验六：进程管理

- 唤醒函数有两种
 - `proc_wakeup_one()` 只被 `proc_exit()` 调用, 它唤醒指定的单个进程。
 - `proc_wakeup()` 未来还会在其他地方调用, 它唤醒所有睡在 `sleep_space` 的进程。
- 注意锁的使用: 理解xv6为什么sleep和wakeup的锁规则保证了睡眠进程不会丢失唤醒?

- 修改 `proc_make_first()`
 - 由于可以直接调用 `proc_alloc()`, 一些操作可以删去。
 - 由于后面要引入调度器, `proc_make_first()` 无需调用 `swtch()`, 直接返回即可。

实验六：进程管理

- 父进程在等待子进程退出时的行为是：遍历所有进程，发现子进程没退出则 `proc_yield()`，`yield` 之后父进程还是 `RUNNABLE` 的，有可能被调度。
 - 将 `proc_wait()` 中的 `proc_yield()` 换成 `proc_sleep()`
 - 在 `proc_exit()` 中使用 `proc_wakeup_one()` 主动唤醒父进程

