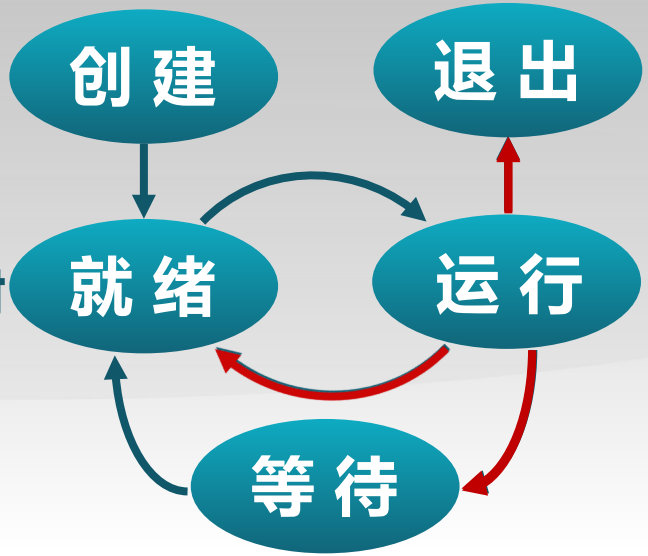
CPU资源的时分复用

- 进程切换：CPU资源的当前占用者切换
  - 保存当前进程在PCB中的执行上下文(CPU状态)
  - 恢复下一个进程的执行上下文
- 处理机调度
  - 从就绪队列中挑选下一个占用CPU运行的进程
  - 从多个可用CPU中挑选就绪进程可使用的CPU资源
- 调度程序：挑选就绪进程的内核函数
  - 调度策略
    - 依据什么原则挑选进程/线程?
  - 调度时机
    - 什么时候进行调度?

调度时机

- 在进程管理的哪些时机中的什么时候进行调度？
  - 进程从运行状态切换到等待状态
  - 进程被终结了
- 非抢占系统
  - 当前进程主动放弃CPU时
- 可抢占系统
  - 中断请求被服务例程响应完成时
  - 当前进程被抢占
    - 进程时间片用完
    - 进程从等待切换到就绪



操作系统

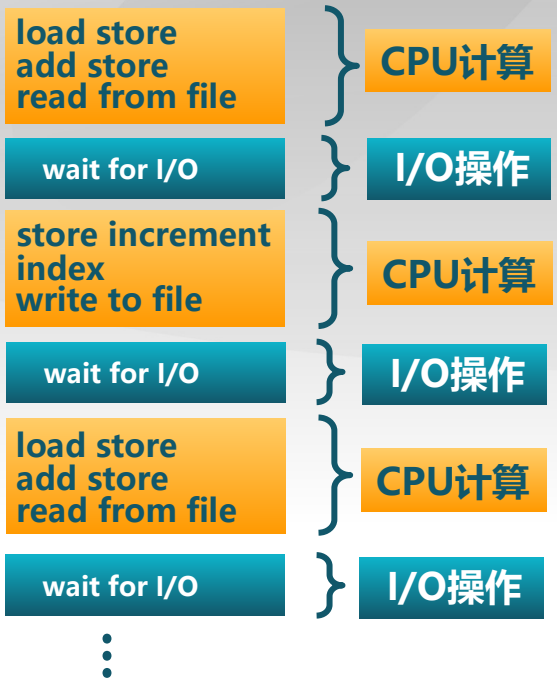
Operating Systems

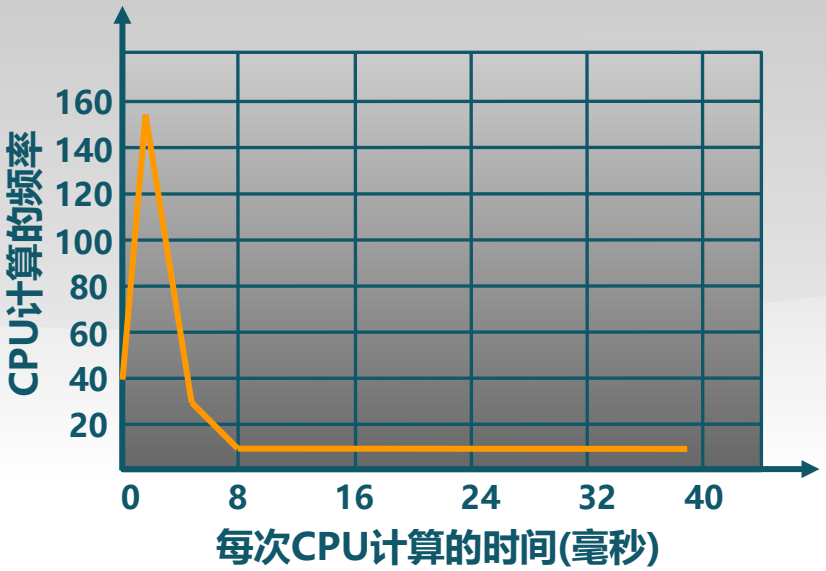
调度策略

- 调度策略
  - 确定如何从就绪队列中选择下一个执行进程
- 调度策略要解决的问题
  - 挑选就绪队列中的哪一个进程?
  - 通过什么样的准则来选择?
- 调度算法
  - 在调度程序中实现的调度策略
- 比较调度算法的准则
  - 哪一个策略/算法较好?

处理机资源的使用模式

- 进程在CPU计算和I/O操作间交替
  - 每次调度决定在下一个CPU计算时将哪个工作交给CPU
  - 在时间片机制下，进程可能在结束当前CPU计算前被迫放弃CPU





比较调度算法的准则

- CPU使用率
  - CPU处于忙状态的时间百分比
- 吞吐量
  - 单位时间内完成的进程数量
- 周转时间
  - 进程从初始化到结束(包括等待)的总时间
- 等待时间
  - 进程在就绪队列中的总时间
- 响应时间
  - 从提交请求到产生响应所花费的总时间

吞吐量与延迟

- 调度算法的要求
  - 希望“更快”的服务
- 什么是更快?
  - 传输文件时的高带宽，调度算法的高吞吐量
  - 玩游戏时的低延迟，调度算法的低响应延迟
  - 这两个因素是独立的
- 与水管的类比
  - 低延迟：喝水的时候想要一打开水龙头水就流出来
  - 高带宽：给游泳池充水时希望从水龙头里同时流出大量的水，并且不介意是否存在延迟

处理机调度策略的响应时间目标

- **减少响应时间**
  - 及时处理用户的输入请求，尽快将输出反馈给用户
- **减少平均响应时间的波动**
  - 在交互系统中，可预测性比高差异低平均更重要
- 低延迟调度改善了用户的交互体验
  - 如果移动鼠标时，屏幕中的光标没动，用户可能会重启电脑
- 响应时间是操作系统的计算延迟

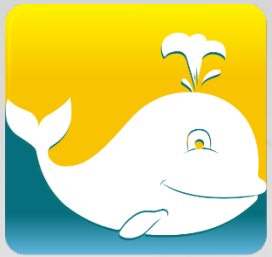
处理机调度策略的吞吐量目标

- **增加吞吐量**
  - 减少开销（操作系统开销，上下文切换）
  - 系统资源的高效利用（CPU，I/O设备）
- **减少等待时间**
  - 减少每个进程的等待时间
- 操作系统需要保证吞吐量不受用户交互的影响
  - 操作系统必须不时进行调度，即使存在许多交互任务
- 吞吐量是操作系统的计算带宽

处理机调度的公平性目标

- 公平的定义
  - 保证每个进程占用相同的CPU时间
  -
  - 这公平么?
  - 一个用户比其他用户运行更多的进程时，怎么办?

  保证每个进程的等待时间相同
- 公平通常会增加平均响应时间



操作系统
Operating Systems

调度算法

- 先来先服务算法
- 短进程优先算法
- 最高响应比优先算法
- 时间片轮转算法
- 多级反馈队列算法
- 公平共享调度算法

调度算法

- 先来先服务算法
  - FCFS: First Come, First Served
- 短进程优先算法
- 最高响应比优先算法
- 时间片轮转算法
- 多级反馈队列算法
- 公平共享调度算法

调度算法

- 先来先服务算法
- 短进程优先算法
  - SPN: Shortest Process Next
  - SJF: Shortest Job First (短作业优先算法)
  - SRT: Shortest Remaining Time (短剩余时间优先算法)
- 最高响应比优先算法
- 时间片轮转算法
- 多级反馈队列算法
- 公平共享调度算法

调度算法

- 先来先服务算法
- 短进程优先算法
- 最高响应比优先算法
  - HRRN: Highest Response Ratio Next
- 时间片轮转算法
- 多级反馈队列算法
- 公平共享调度算法

调度算法

- 先来先服务算法
- 短进程优先算法
- 最高响应比优先算法
- 时间片轮转算法
  - RR: Round Robin
- 多级反馈队列算法
- 公平共享调度算法

调度算法

- 先来先服务算法
- 短进程优先算法
- 最高响应比优先算法
- 时间片轮转算法
- 多级反馈队列算法
  - MFQ: Multilevel Feedback Queues
- 公平共享调度算法

调度算法

- 先来先服务算法
- 短进程优先算法
- 最高响应比优先算法
- 时间片轮转算法
- 多级反馈队列算法
- 公平共享调度算法
  - FSS: Fair Share Scheduling

先来先服务算法(First Come First Served, FCFS)

- 依据进程进入就绪状态的先后顺序排列
  - 进程进入等待或结束状态时，就绪队列中的下一个进程占用CPU
- FCFS算法的周转时间
  - 示例：3个进程，计算时间分别为12,3,3

任务到达顺序：P_1, P_2, P_3

周转时间=(12+15+18)/3=15

任务到达顺序：P_2, P_3, P_1

周转时间=(3+6+18)/3=9

先来先服务算法的特征

- 优点
  - 简单
- 缺点
  - 平均等待时间波动较大
    - 短进程可能排在长进程后面
  - I/O资源和CPU资源的利用率较低
    - CPU密集型进程会导致I/O设备闲置时，I/O密集型进程也等待

短进程优先算法(SPN)

- 选择就绪队列中执行时间最短进程占用CPU进入运行状态
  - 就绪队列按预期的执行时间来排序

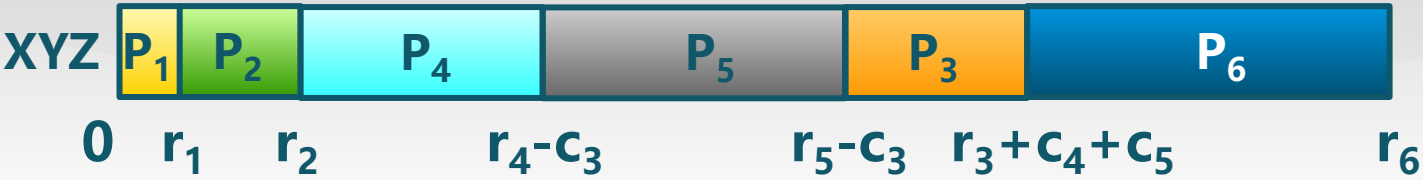
- 短剩余时间优先算法(SRT)
  - SPN算法的可抢占改进

短进程优先算法具有最优平均周转时间

- SPN算法中一组进程的平均周转时间

周转时间=$(r_1+r_2+r_3+r_4+r_5+r_6)/6$

修改进程执行顺序可能减少平均等待时间吗?



周转时间=$(r_1+r_2+r_4-c_3+r_5-c_3+r_4+c_4+c_5+r_6)/6$

=$(r_1+r_2+r_3+r_4+r_5+r_6+(c_4+c_5-2c_3))/6$

短进程优先算法的特征：缺点

- 可能导致饥饿
  - 连续的短进程流会使长进程无法获得CPU资源
- 需要预知未来
  - 如何预估下一个CPU计算的持续时间?
  - 简单的解决办法：询问用户
    - 用户欺骗就杀死相应进程
    - 用户不知道怎么办?

短进程优先算法的执行时间预估

- 用历史的执行时间来预估未来的执行时间

```
process P
begin
   loop
     <read input from user>
     <process input>
   end loop
end P
```

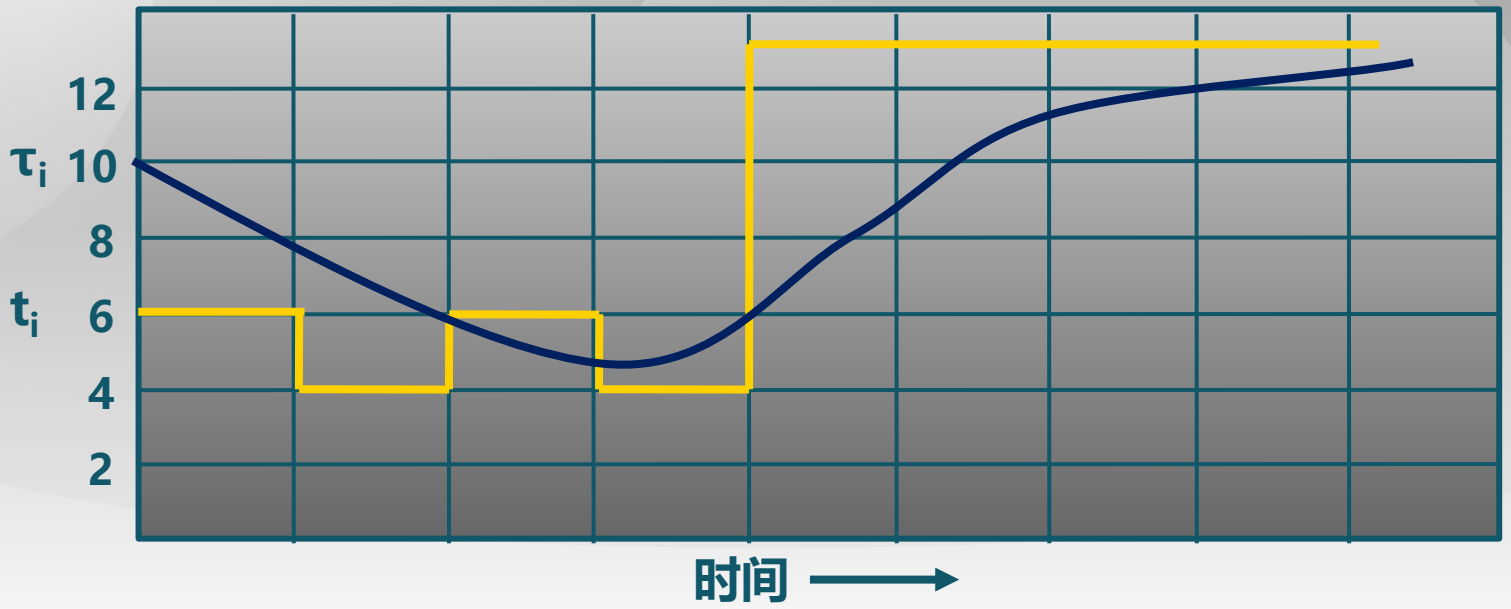
$\tau_{n+1} = \alpha t_n + (1-\alpha)\,\tau_n$，其中 $0 \le \alpha \le 1$

t_n——第n次的CPU计算时间

τ_{n+1}——第n+1次的CPU计算时间预估

$\tau_{n+1} = \alpha t_n + (1-\alpha)\,\alpha t_{n-1} + (1-\alpha)(1-\alpha)\,\alpha t_{n-2} + \ldots$

预估执行时间



实际CPU执行时间 (t_i)		6	4	6	4	13	13	13	...
预估CPU执行时间(τ_i)	10	8	6	6	5	9	11	12	...

最高响应比优先算法(HRRN)

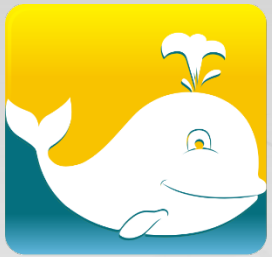
- 选择就绪队列中响应比R值最高的进程

  R = （w+s)/s

  w: 等待时间(waiting time)

  s: 执行时间(service time)

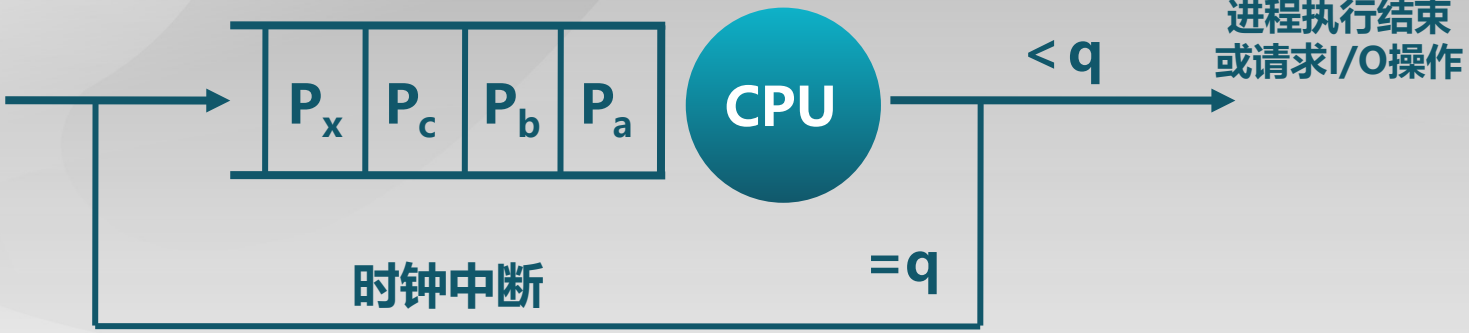
  - 在短进程优先算法的基础上改进
  - 不可抢占
  - 关注进程的等待时间
  - 防止无限期推迟



操作系统
Operating Systems

时间片轮转算法(RR, Round-Robin)

- 时间片
  - 分配处理机资源的基本时间单元



- 算法思路
  - 时间片结束时，按FCFS算法切换到下一个就绪进程
  - 每隔(n – 1)个时间片进程执行一个时间片q

时间片为20的RR算法示例

- 示例: 4个进程的执行时间如下

P1	53
P2	8
P3	68
P4	24

甘特图如下:

等待时间 $P_1=(68-20)+(112-88)=72$

$P_2=(20-0)=20$

$P_3=(28-0)+(88-48)+(125-108)=85$

$P_4=(48-0)+(108-68)=88$

平均等待时间 = $(72+20+85+88)/4=66.25$

时间片轮转算法中的时间片长度

- RR算法开销
  - 额外的上下文切换
- 时间片太大
  - 等待时间过长
  - 极限情况退化成FCFS
- 时间片太小
  - 反应迅速，但产生大量上下文切换
  - 大量上下文切换开销影响到系统吞吐量
- 时间片长度选择目标
  - 选择一个合适的时间片长度
  - 经验规则：维持上下文切换开销处于1%以内

比较FCFS和RR

- 示例: 4个进程的执行时间如下

P1	53
P2	8
P3	68
P4	24

假设上下文切换时间为零

FCFS和RR各自的平均等待时间是多少?

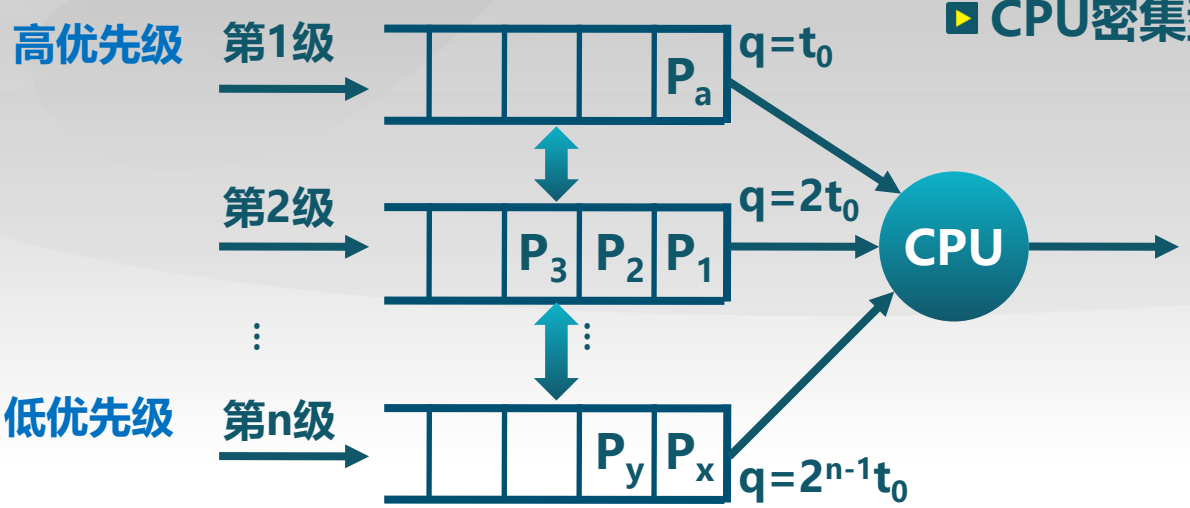
时间片	P_1	P_2	P_3	P_4	平均等待时间
RR(q=1)	84	22	85	57	62
RR(q=5)	82	20	85	58	61.25
RR(q=8)	80	8	85	56	57.25
RR(q=10)	82	10	85	68	61.25
RR(q=20)	72	20	85	88	66.25
BestFCFS	32	0	85	8	31.25
WorstFCFS	68	145	0	121	83.5

多级队列调度算法(MQ)

- 就绪队列被划分成多个独立的子队列
- 每个队列有其自己的调度策略
- 例如前台–RR、后台–FCFS
  - 固定优先级
    - 先处理前台，然后处理后台
    - 可能导致饥饿
  - 时间片轮转
    - 每个队列都得到一个确定的能够调度其进程的CPU总时间
    - 如：80%CPU时间用于前台，20%CPU时间用于后台

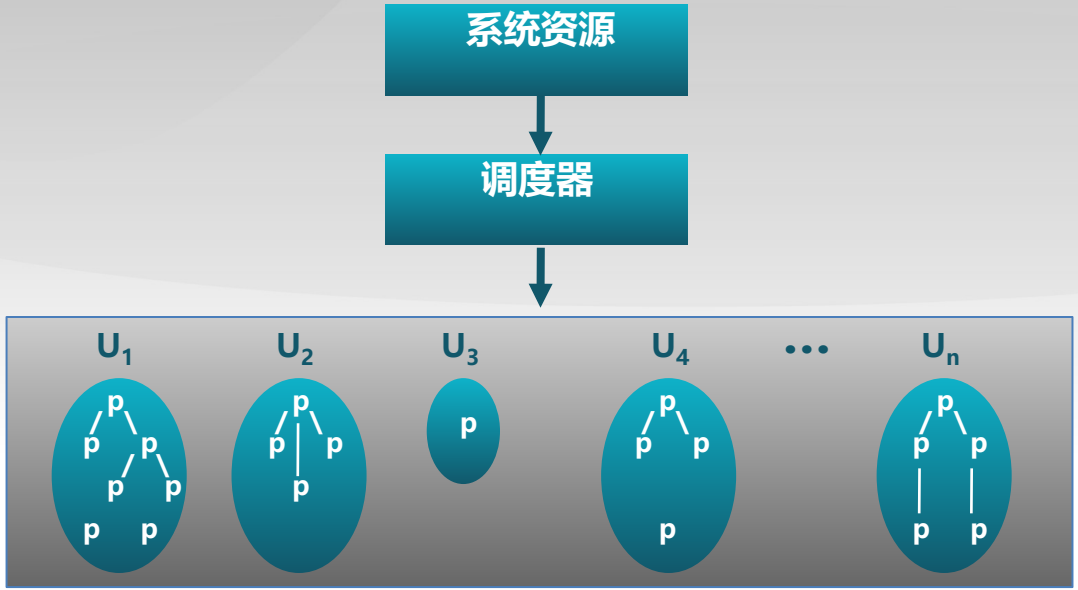
多级反馈队列算法(MLFQ)

- 进程可在不同队列间移动的多级队列算法
  - 时间片大小随优先级级别增加而增加
  - 如进程在当前的时间片没有完成，则降到下一个优先级
  - I/O密集型进程停留在高优先级
- MLFQ算法的特征
  - CPU密集型进程的优先级下降很快



公平共享调度(FSS, Fair Share Scheduling)

- FSS控制用户对系统资源的访问
  - 一些用户组比其他用户组更重要
  - 保证不重要的组无法垄断资源
  - 未使用的资源按比例分配
  - 没有达到资源使用率目标的组获得更高的优先级



传统调度算法总结

- 先来先服务算法
- 短进程优先算法
- 最高响应比优先算法
- 时间片轮转算法
- 多级反馈队列
- 公平共享调度

传统调度算法总结

- 先来先服务算法
  - 不公平，平均等待时间较差
- 短进程优先算法
- 最高响应比优先算法
- 时间片轮转算法
- 多级反馈队列
- 公平共享调度

传统调度算法总结

- 先来先服务算法
- 短进程优先算法
  - 不公平，平均周转时间最小
  - 需要精确预测计算时间
  - 可能导致饥饿
- 最高响应比优先算法
- 时间片轮转算法
- 多级反馈队列
- 公平共享调度

传统调度算法总结

- 先来先服务算法
- 短进程优先算法
- 最高响应比优先算法
  - 基于SPN调度
  - 不可抢占
- 时间片轮转算法
- 多级反馈队列
- 公平共享调度

传统调度算法总结

- 先来先服务算法
- 短进程优先算法
- 最高响应比优先算法
- 时间片轮转算法
  - 公平，但是平均等待时间较差
- 多级反馈队列
- 公平共享调度

传统调度算法总结

- 先来先服务算法
- 短进程优先算法
- 最高响应比优先算法
- 时间片轮转算法
- 多级反馈队列
  - 多种算法的集成
- 公平共享调度

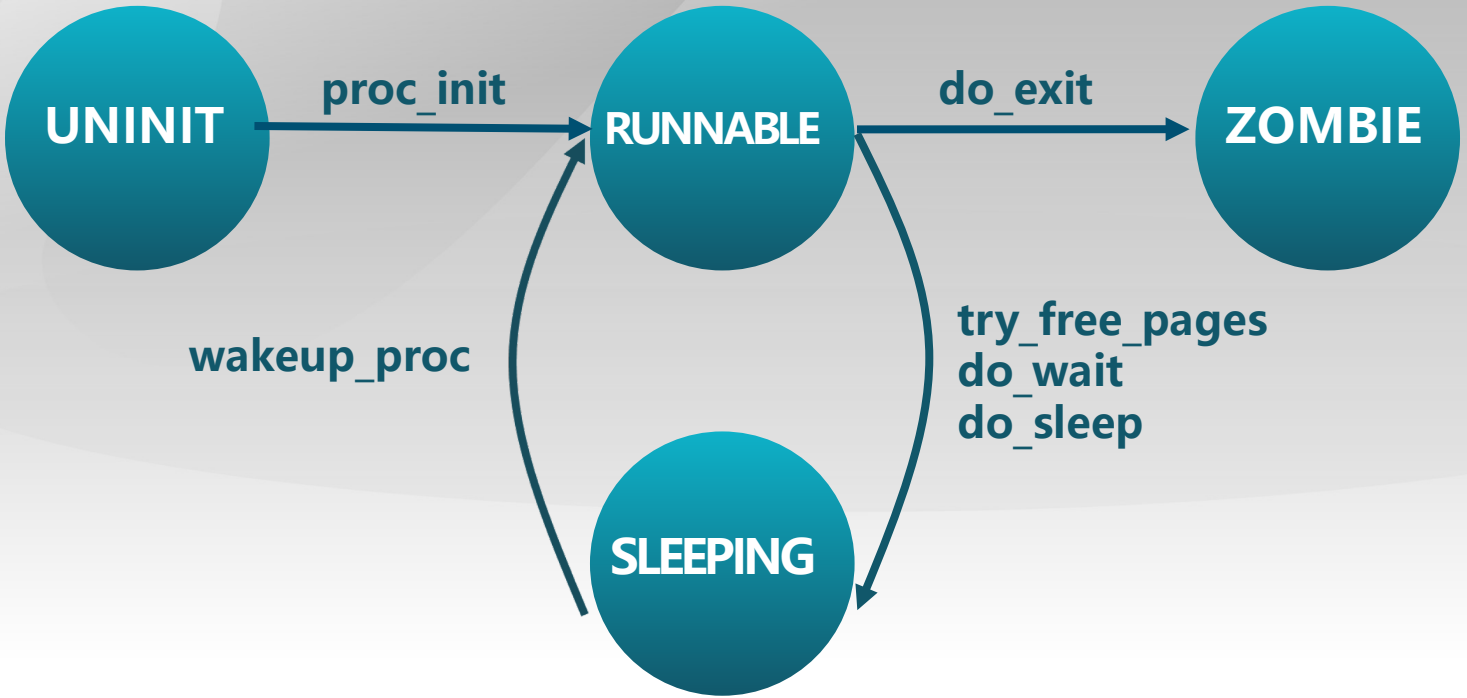
传统调度算法总结

- 先来先服务算法
- 短进程优先算法
- 最高响应比优先算法
- 时间片轮转算法
- 多级反馈队列
- 公平共享调度
  - 公平是第一要素

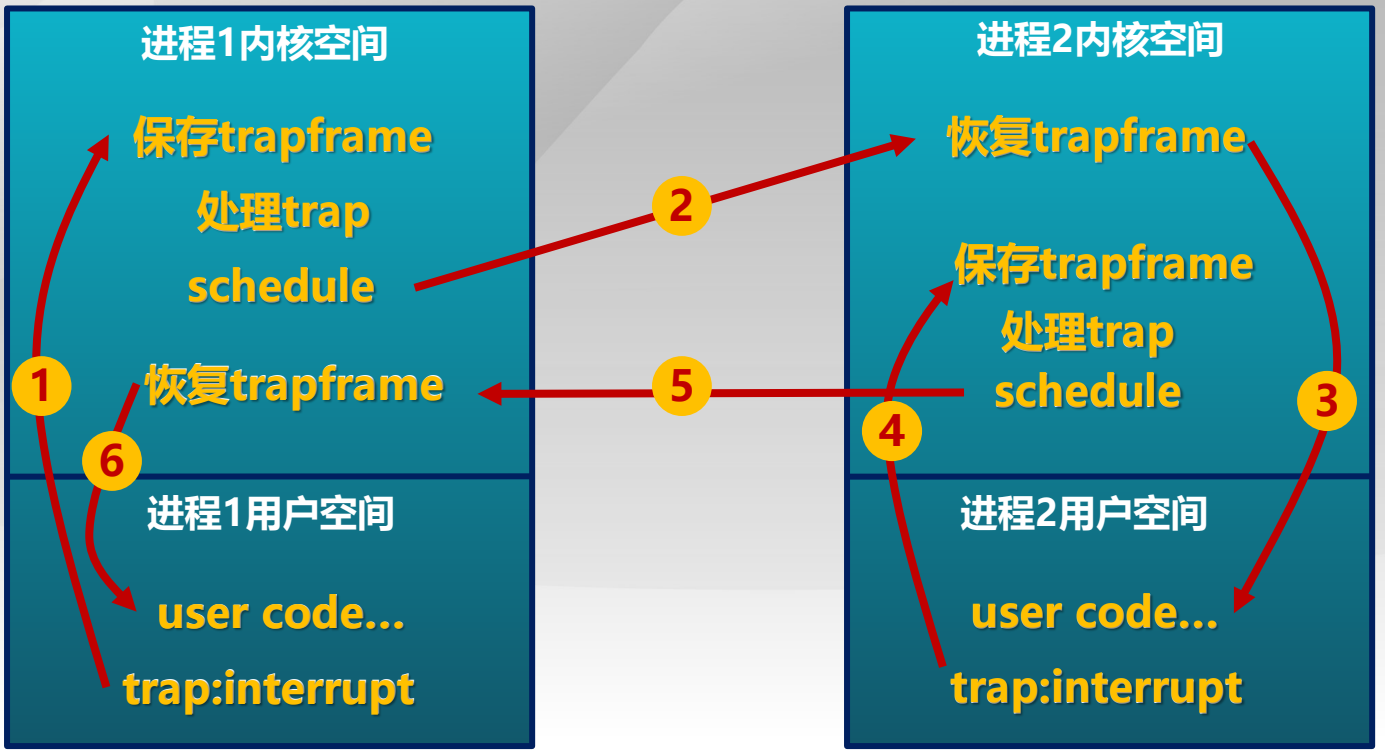
ucore的调度队列run_queue

```
struct run_queue {
    list_entry_t run_list;
    unsigned int proc_num;
    int max_time_slice;
    list_entry_t rq_link;
};
```

ucore的线程状态


UNINIT
proc_init
RUNNABLE
do_exit
ZOMBIE
wakeup_proc
try_free_pages
do_wait
do_sleep
SLEEPING

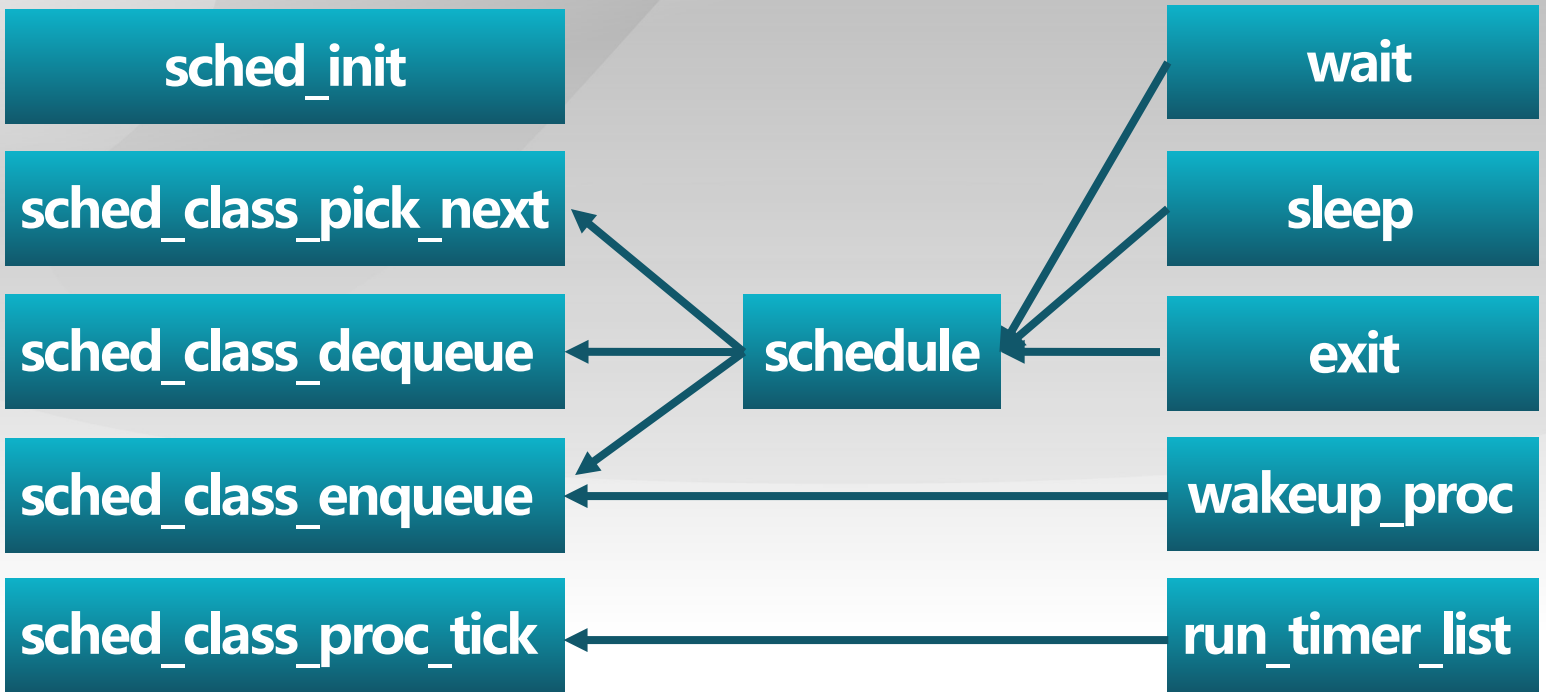
ucore的调度时机和进程切换

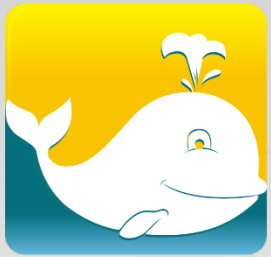


ucore的调度算法接口sched_class

```
struct sched_class {
    const char *name;
    void (*init)(struct run_queue *rq);
    void (*enqueue)(struct run_queue *rq, struct proc_struct *proc);
    void (*dequeue)(struct run_queue *rq, struct proc_struct *proc);
    struct proc_struct *(*pick_next)(struct run_queue *rq);
    void (*proc_tick)(struct run_queue *rq, struct proc_struct *proc);
};
```

ucore调度框架


sched_init
sched_class_pick_next
sched_class_dequeue
sched_class_enqueue
sched_class_proc_tick
schedule
wait
sleep
exit
wakeup_proc
run_timer_list



操作系统

Operating Systems

实时操作系统

- 实时操作系统的定义
  - 正确性依赖于其时间和功能两方面的操作系统
- 实时操作系统的性能指标
  - 时间约束的及时性（deadlines)
  - 速度和平均性能相对不重要
- 实时操作系统的特性
  - 时间约束的可预测性

实时操作系统分类

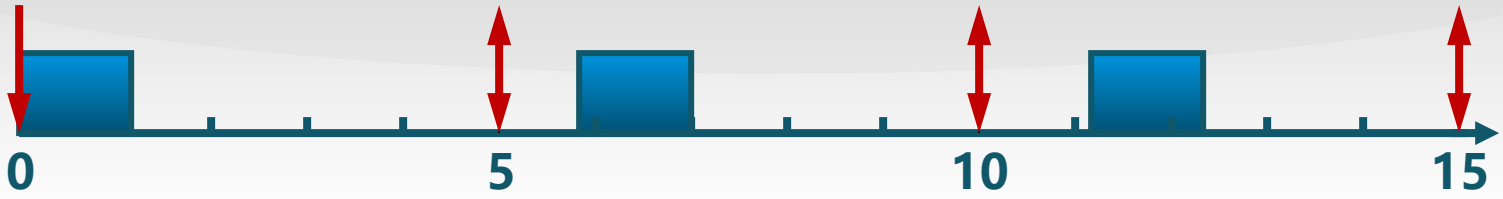
- 强实时操作系统
  - 要求在指定的时间内必须完成重要的任务
- 弱实时操作系统
  - 重要进程有高优先级，要求尽量但非必须完成

实时任务

- 任务（工作单元）
  - 一次计算，一次文件读取，一次信息传递等等
- 任务属性
  - 完成任务所需要的资源
  - 定时参数

周期实时任务

- 周期实时任务：一系列相似的任务
  - 任务有规律地重复
  - 周期p = 任务请求时间间隔 (0 <p)
  - 执行时间e = 最大执行时间(0 < e < p)
  - 使用率U = e/p

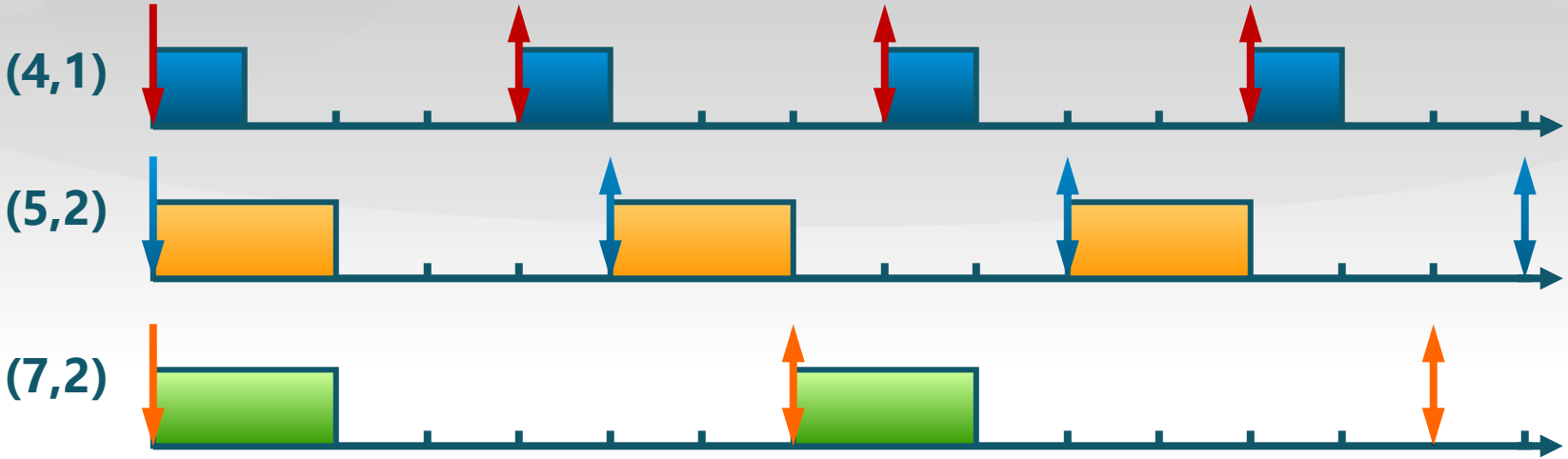


软时限和硬时限

- 硬时限（Hard deadline)
  - 错过任务时限会导致灾难性或非常严重的后果
  - 必须验证，在最坏情况下能够满足时限
- 软时限(Soft deadline)
  - 通常能满足任务时限
    - 如有时不能满足，则降低要求
  - 尽力保证满足任务时限

可调度性

- 可调度表示一个实时操作系统能够满足任务时限要求
  - 需要确定实时任务的执行顺序
  - 静态优先级调度
  - 动态优先级调度

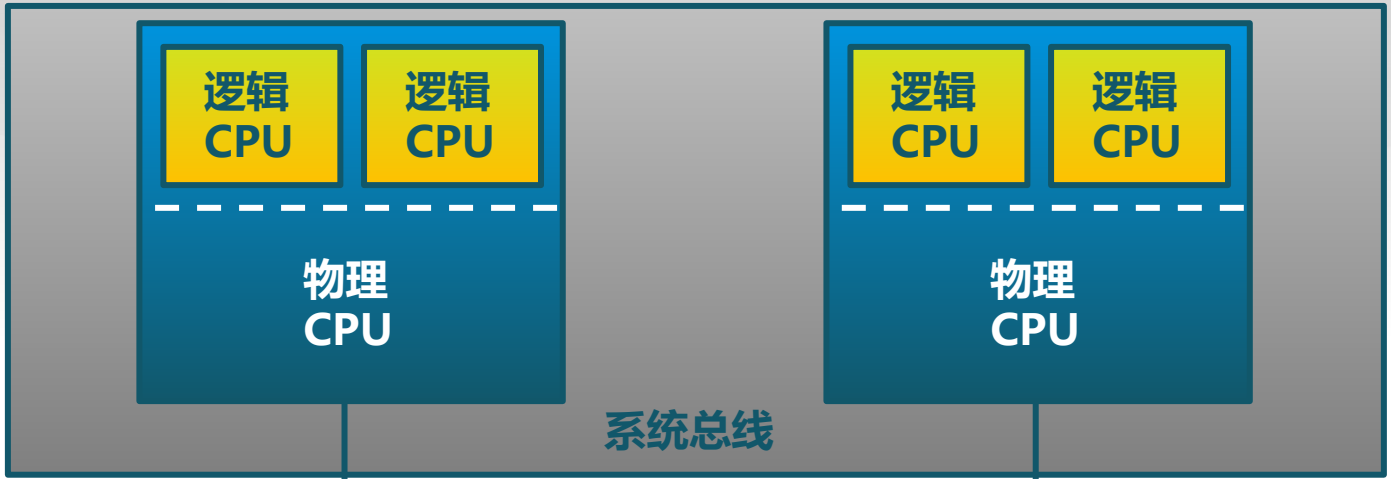


实时调度

- 速率单调调度算法(RM, Rate Monotonic)
  - 通过周期安排优先级
  - 周期越短优先级越高
  - 执行周期最短的任务
- 最早截止时间优先算法 (EDF, Earliest Deadline First)
  - 截止时间越早优先级越高
  - 执行截止时间最早的任务

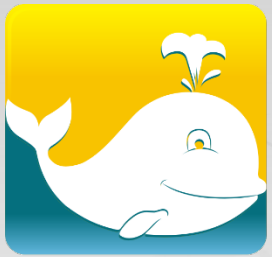
多处理器调度

- 多处理机调度的特征
  - 多个处理机组成一个多处理机系统
  - 处理机间可负载共享
- 对称多处理器(SMP, Symmetric multiprocessing)调度
  - 每个处理器运行自己的调度程序
  - 调度程序对共享资源的访问需要进行同步



对称多处理器的进程分配

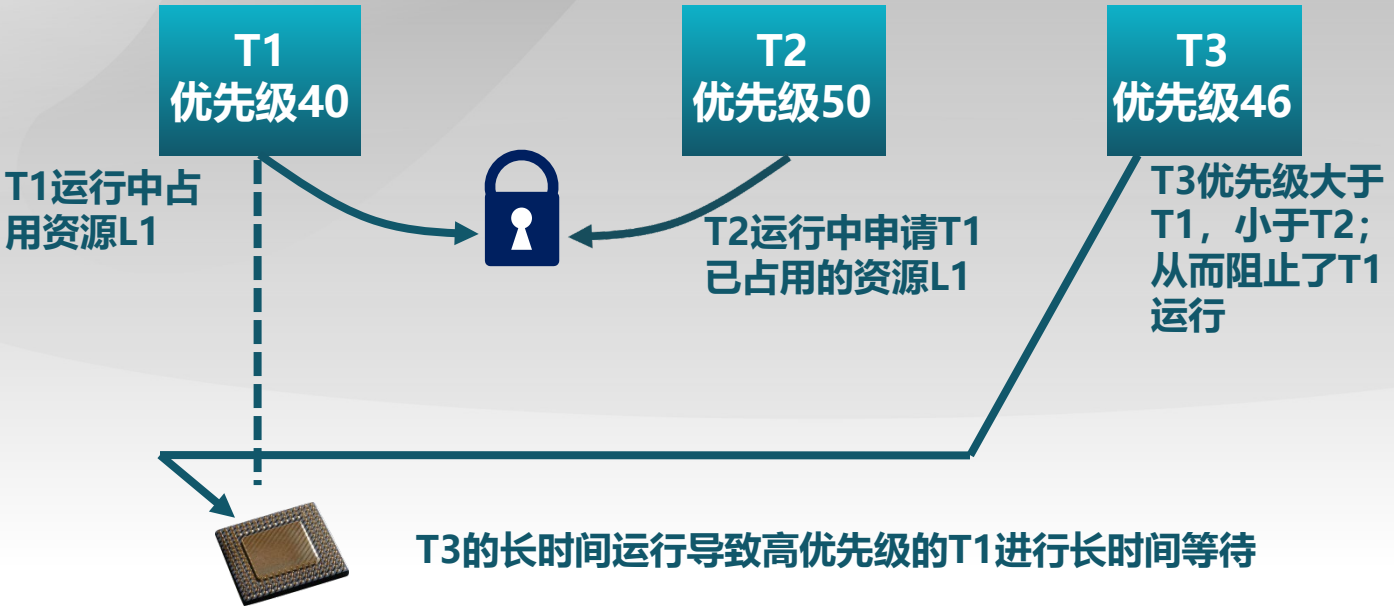
- 静态进程分配
  - 进程从开始到结束都被分配到一个固定的处理机上执行
  - 每个处理机有自己的就绪队列
  - 调度开销小
  - 各处理机可能忙闲不均
- 动态进程分配
  - 进程在执行中可分配到任意空闲处理机执行
  - 所有处理机共享一个公共的就绪队列
  - 调度开销大
  - 各处理机的负载是均衡的



操作系统
Operating Systems

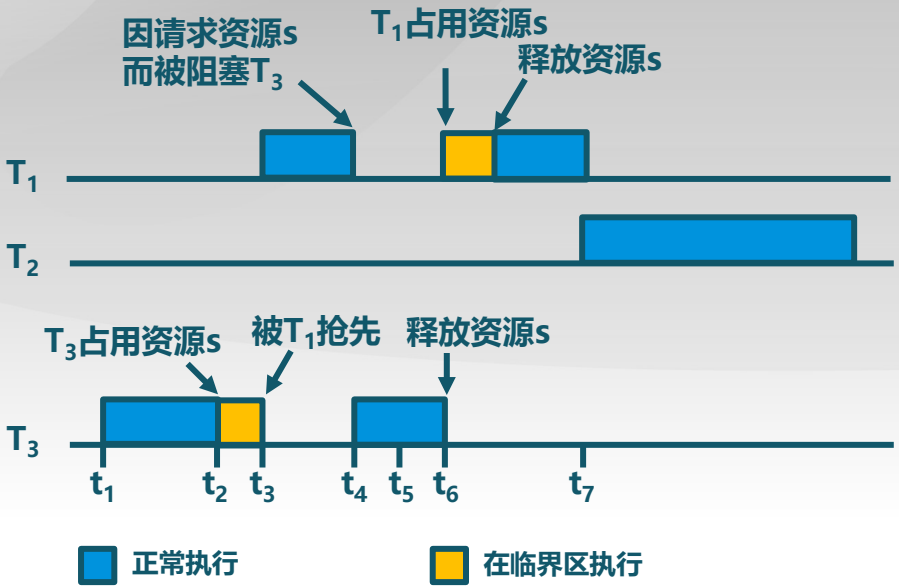
优先级反置(Priority Inversion)

- 操作系统中出现高优先级进程长时间等待低优先级进程所占用资源的现象
- 基于优先级的可抢占调度算法存在优先级反置



优先级继承（Priority Inheritance）

- 占用资源的低优先级进程继承申请资源的高优先级进程的优先级
  - 只在占有资源的低优先级进程被阻塞时,才提高占有资源进程的优先级



优先级天花板协议（priority ceiling protocol）

- 占用资源进程的优先级和所有可能申请该资源的进程的最高优先级相同
  - 不管是否发生等待,都提升占用资源进程的优先级
  - 优先级高于系统中所有被锁定的资源的优先级上限，任务执行临界区时就不会被阻塞

第十一讲：处理机调度

第 7 节：rCore 调度框架

向勇、陈渝

清华大学计算机系

xyong,yuchen@tsinghua.edu.cn

2020 年 5 月 5 日

提纲

rCore 调度框架

rcore-thread/src/scheduler/mod.rs

```rust
/// The scheduler for a ThreadPool
pub trait Scheduler: 'static {
    /// Push a thread to the back of ready queue.
    fn push(&self, tid: Tid);
    /// Select a thread to run, pop it from the queue.
    fn pop(&self, cpu_id: usize) -> Option<Tid>;
    /// Got a tick from CPU.
    /// Return true if need reschedule.
    fn tick(&self, current_tid: Tid) -> bool;
    /// Set priority of a thread.
    fn set_priority(&self, tid: Tid, priority: u8);
    /// remove a thread in ready queue.
    fn remove(&self, tid: Tid);
}
```

与调度相关的线程控制函数

rcore-thread/src/thread_pool.rs

```rust
pub fn add(&self, mut context: Box<dyn Context>) -> Tid
pub(crate) fn tick(&self, cpu_id: usize, tid: Option<Tid>) -> bool
pub fn set_priority(&self, tid: Tid, priority: u8)
pub(crate) fn run(&self, cpu_id: usize) -> Option<(Tid, Box<dyn Context>)>
pub(crate) fn stop(&self, tid: Tid, context: Box<dyn Context>)
fn set_status(&self, tid: Tid, status: Status)
pub fn wakeup(&self, tid: Tid)
```

调度数据结构：struct ThreadPool

rcore-thread/src/thread_pool.rs

```
pub struct ThreadPool {
    threads: Vec<Mutex<Option<Thread>>>,
    scheduler: Box<dyn Scheduler>,
    timer: Mutex<Timer<Event>>,
}
```

调度算法和参数设置

rCore/kernel/src/process/mod.rs

```
pub fn init() {
    // NOTE: max_time_slice <= 5 to ensure 'priority' test pass
    let scheduler = scheduler::RRScheduler::new(5);
    let manager = Arc::new(ThreadPool::new(scheduler, MAX_PROCESS_NUM));
    unsafe {
        for cpu_id in 0..MAX_CPU_NUM {
            PROCESSORS[cpu_id].init(cpu_id, Thread::new_init(), manager.clone());
        }
    }
    crate::shell::add_user_shell();
    info!("process: init end");
}
```

时间片轮转（Round Robin）调度算法：数据结构

rcore-thread/src/scheduler/rr.rs

```rust
pub struct RRScheduler {
    inner: Mutex<RRSchedulerInner>,
}
struct RRSchedulerInner {
    max_time_slice: usize,
    infos: Vec<RRProcInfo>,
}
#[derive(Debug, Default, Copy, Clone)]
struct RRProcInfo {
    present: bool,
    rest_slice: usize,
    prev: Tid,
    next: Tid,
}
```

RR 调度算法实现

```rust
impl RRSchedulerInner {
    fn push(&mut self, tid: Tid) {…
    }

    fn pop(&mut self) -> Option<Tid> {…
    }

    fn tick(&mut self, current: Tid) -> bool {…
    }

    fn remove(&mut self, tid: Tid) {…
    }
}

impl RRSchedulerInner {
    fn _list_add_before(&mut self, i: Tid, at: Tid) {…
    }
    fn _list_add_after(&mut self, i: Tid, at: Tid) {…
    }
    fn _list_remove(&mut self, i: Tid) {…
    }
}
```

时间片用完时的线程调度和切换过程

```
/Users/xyong/github/rCore/kernel/src/arch/riscv/interrupt.rs
  pub extern "C" fn rust_trap(tf: &mut TrapFrame)
/Users/xyong/github/rCore/kernel/src/trap.rs
  pub fn timer()
/Users/xyong/github/rcore-thread/src/processor.rs
  pub fn tick(&self)
/Users/xyong/github/rcore-thread/src/processor.rs
  pub(crate) fn yield_now(&self)
```