第二部分. 进程管理

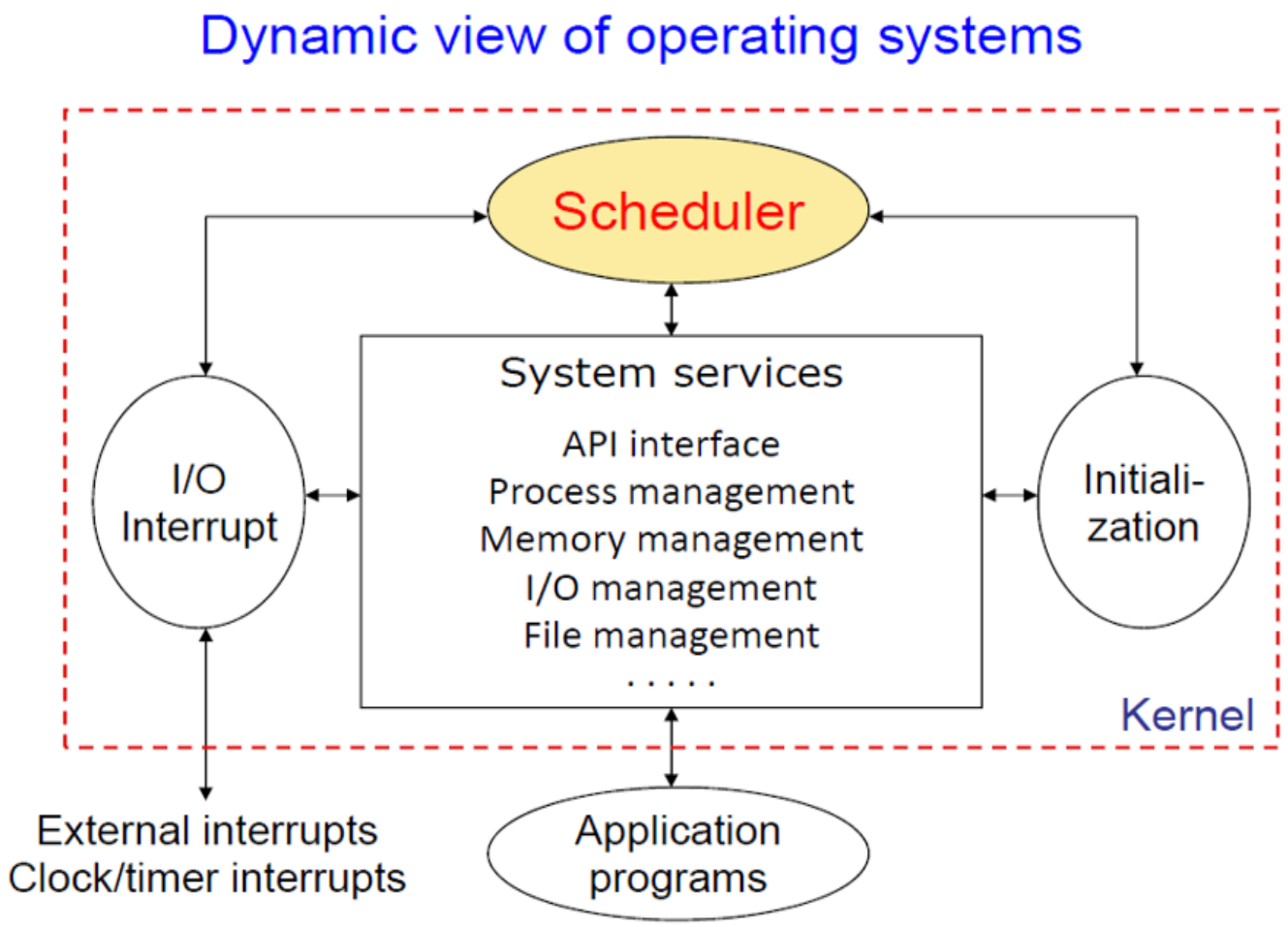
第五章:CPU 调度

1. 基本概念
2. 调度准则
3. 调度算法
4. 线程调度
5. 多处理器调度
6. 实时调度

Note : These lecture materials are based on the lecture notes prepared by the authors of the book titled *Operating System Concepts.*

学习内容

- 介绍CPU调度，是多道程序操作系统的基础
- 介绍各种CPU调度算法
- 讨论为特定系统选择CPU调度算法的评估标准
- 介绍实时系统及调度算法


Dynamic view of operating systems
Scheduler
System services
API interface
Process management
Memory management
I/O management
File management
. . . . .
I/O
Interrupt
Initiali-
zation
Kernel
External interrupts
Clock/timer interrupts
Application
programs

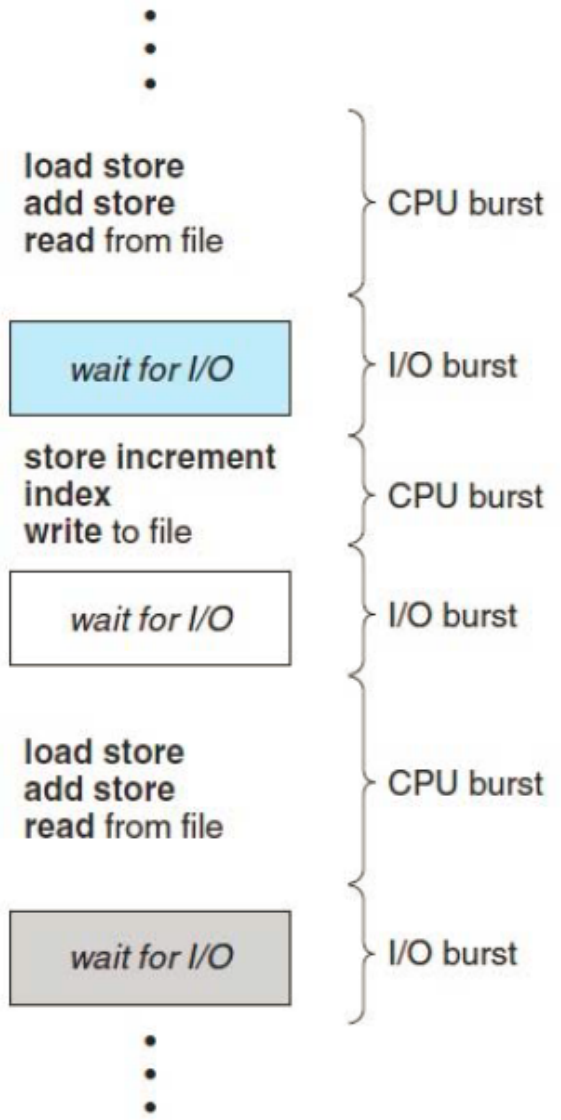
第一节 基本概念介绍

1. 基本概念

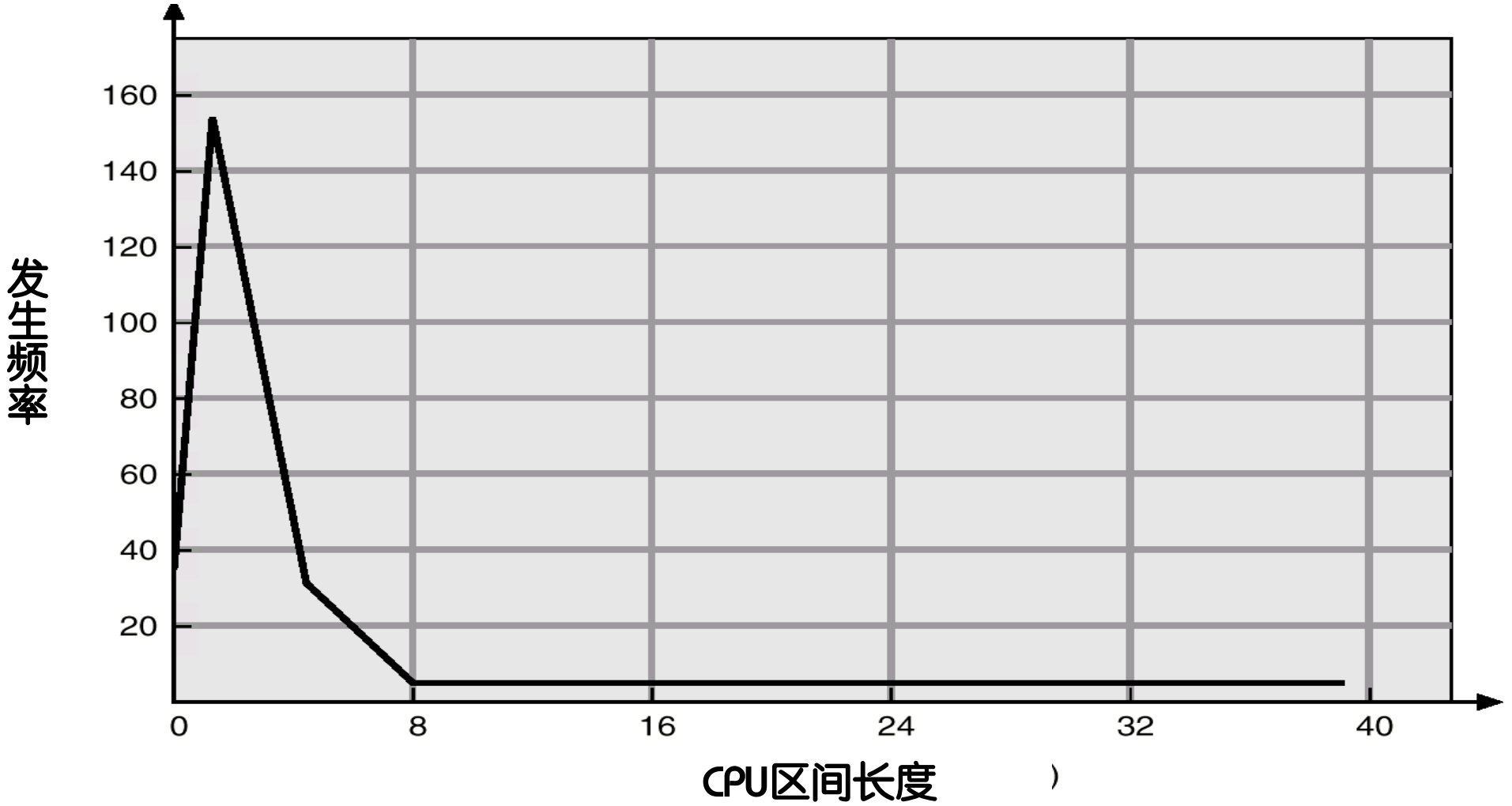
- 多道程序的目的是使CPU的使用率最大化
- 一个进程的执行由CPU执行(CPU区间)和I/O等待(I/O区间)组成。进程在执行的过程中，不断在这两个状态之间的进行切换
- 经过对大量进程进行分析，得出的结论显示

进程一般由大量的短CPU区间（<8ms）和少量的长CPU区间组成

1. I/O为主的程序里短CPU区间较多，
2. CPU为主的程序里长CPU区间较少



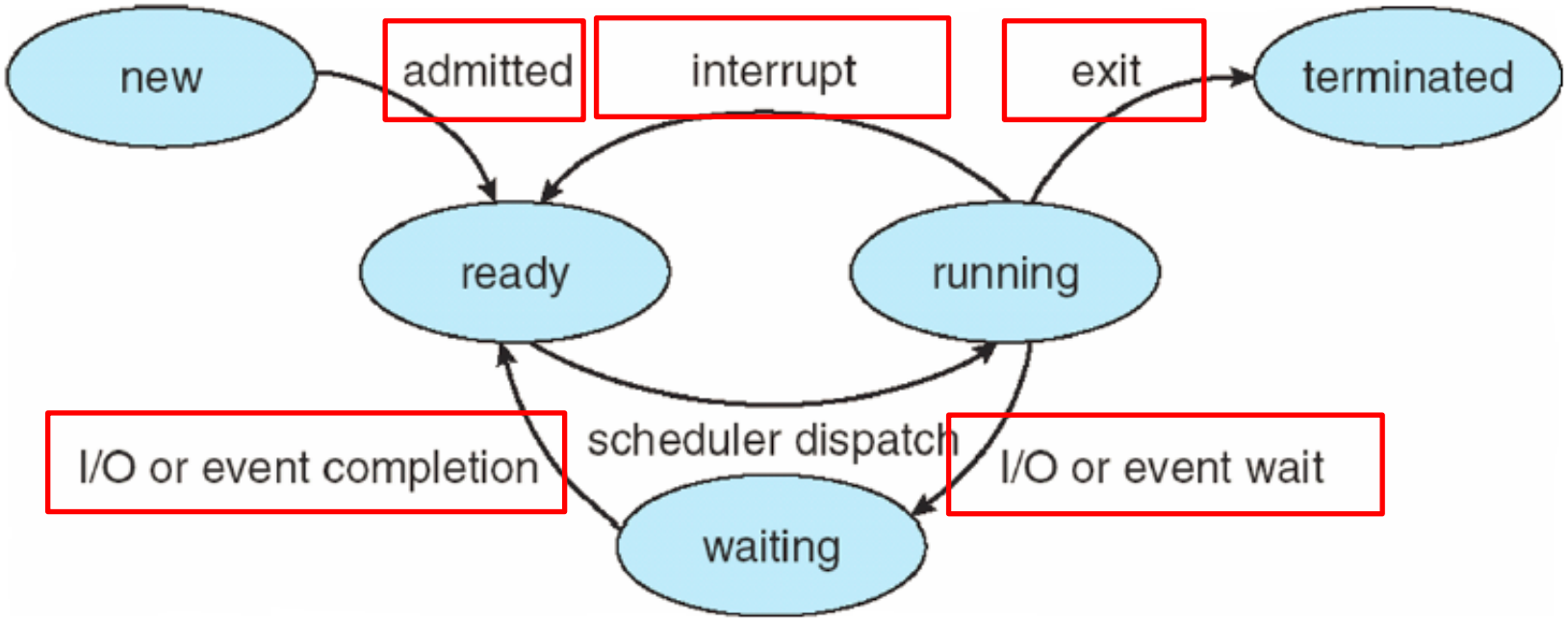
CPU区间时间曲线图



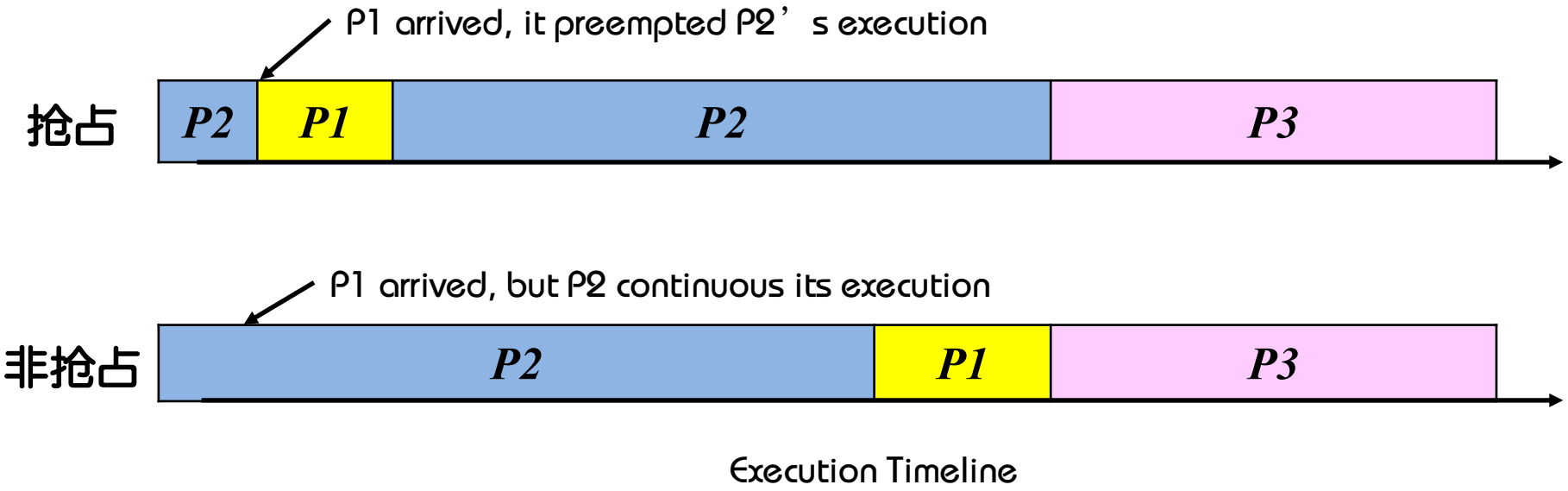
CPU 调度程序

- 每当CPU空闲时，调度程序(短期调度程序)从就绪队列中选择一个进程，并为之分配CPU
- 可以采用各种不同的排序方式对就绪队列中的进程进行排序(可以理解为不同的调度算法)
- CPU调度的决策可能在如下环节发生，当一个进程
  1. 从运行状态转换成等待状态(如发生I/O)
  2. 从运行状态转换成就绪状态(如发生中断)
  3. 从等待状态转换成就绪状态(I/O完成)
  4. 进程终止
  5. 进程进入就绪队列

短期调度程序:short-term scheduler
调度点:scheduling point


new
admitted
interrupt
exit
terminated
ready
running
I/O or event completion
scheduler dispatch
I/O or event wait
waiting

- 非抢占调度（nonpreemptive） 1,4
  - 一旦把CPU分配给一个进程，直到该进程结束之前，不能把CPU分配给其他进程
- 抢占调度（preemptive） 2,3
  - 进程在执行过程中，可以被其他进程抢占CPU使用权



分派程序(dispatch) vs. 调度器

- 分派程序是一个模块，用来将CPU的控制交给由短期调度程序选择的进程。其功能包括:
  1. 切换上下文（context switch）
  2. 切换到用户模式
  3. 跳转到用户程序的合适位置，以重新启动程序

  合适的位置:The address space which is specified by the program counter (PC)
- 分派延迟（Dispatch latency）

分派程序停止一个进程而启动另一个进程所要花的时间称为分派延迟

- Homework #3
- 思考题: 分派延迟和上下文切换时间有区别吗?有的话，有什么区别?

第二节 CPU调度准则和调度算法

2. 调度准则

不同的调度算法具有不同的属性，分析CPU调度算法需要考虑以下准则

1. CPU使用率（CPU Utilization）:需要使CPU尽可能忙
2. 吞吐量（Throughput）:指的是在一个时间单元内所完成的进程数量
3. 周转时间（Turnaround Time）:从进程提交，到进程完成的时间段称为周转时间。周转时间为所有时间段之和，包括等待进入内存，在就绪队列中等待，在CPU上执行和I/O执行
4. 等待时间（Waiting Time）:在就绪队列中等待的时间
5. 响应时间（Response Time）：从提交请求到产生第一相应的时间。注意，它是开始响应所需要的时间，而不是输出响应所需要的时间（交互系统）

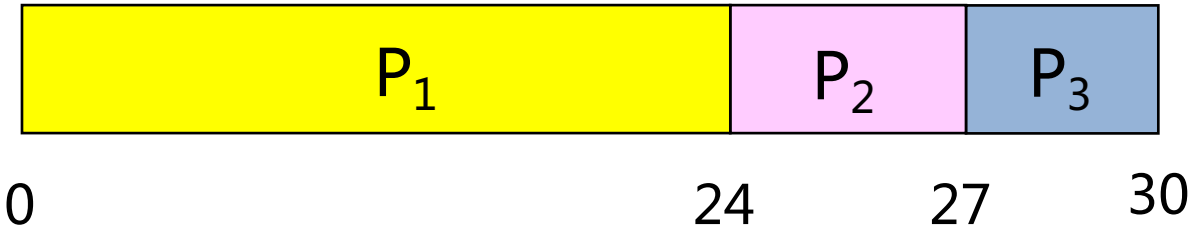
3. 调度算法

1. 先到先服务调度(First Come, First Service Scheduling)
2. 最短作业优先调度(Shortest Job First Scheduling)
3. 优先级调度(Priority Scheduling)
4. 轮转法调度(Round Robin Scheduling)
5. 多级队列调度(Multilevel Queue Scheduling)
6. 多级反馈队列调度(Multilevel Feedback Queue Scheduling)

- 先请求CPU 的进程先分配CPU，非抢占式调度。考虑如下进程

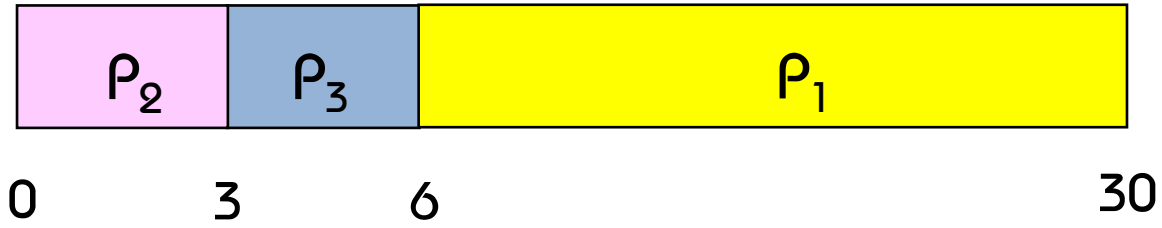
进程	区间时间
P_1	24
P_2	3
P_3	3

- 如进程运行顺序（arrival time, release time）为P_1，P_2，P_3，甘特图（Gantt Chart）为如下:



- 个进程的等待时间为 $P_1 = 0$， $P_2 = 24$， $P_3 = 27$
- 平均等待时间: (0 + 24 + 27)/3 = 17

- 如运行顺序为 P_2 , P_3 , P_1 ， 则甘特图为如下:



- 等待时间分别为 $P_1 = 6, P_2 = 0, P_3 = 3$
- 平均等待时间为: (6 + 0 + 3)/3 = 3
- 平均等待时间要比前一种情况要好，原因是无护航效果

- 护航效果(convoy effect):所有其他小进程都等待一个大进程释放CPU，即短进程跟在长进程后面。

1. 大进程:CPU区间长，I/O区间短
2. 小进程:I/O区间短，CPU区间短

- 大进程运行CPU区间时，其他短进程很快完成I/O操作并移到就绪队列等待分配CPU;(I/O设备空闲)
- 大进程释放CPU后，移到I/O队列，其他短进程很快完成CPU操作并移到I/O队列;(CPU空闲)

选择CPU区间长度最短的进程，这里的CPU区间是进程的下一个CPU区间（也就是剩余CPU区间），而不是进程整个区间长度。可分为非抢占调度和抢占调度。

定理:对于给定的一组进程，在平均等待时间上，最短作业优先调度算法是最优算法

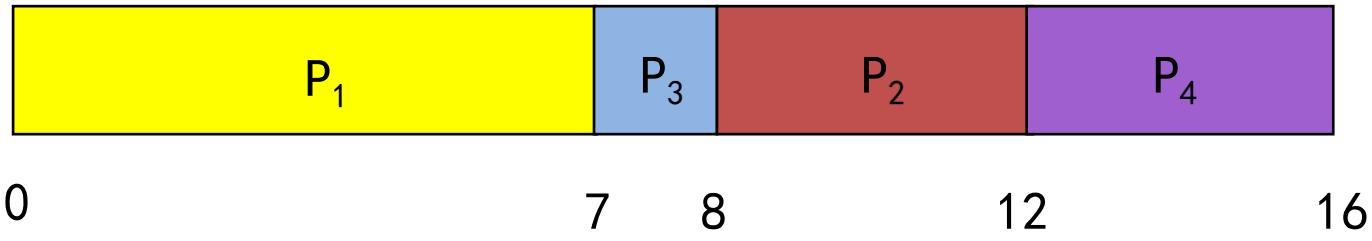
但，问题是预先知道下一个CPU区间(remaining execution time)的长度是有难度的。

非抢占式最短作业优先调度

- 给定的进程如下

进程	到达时间	区间时间
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- 非抢占式 最短作业优先调度



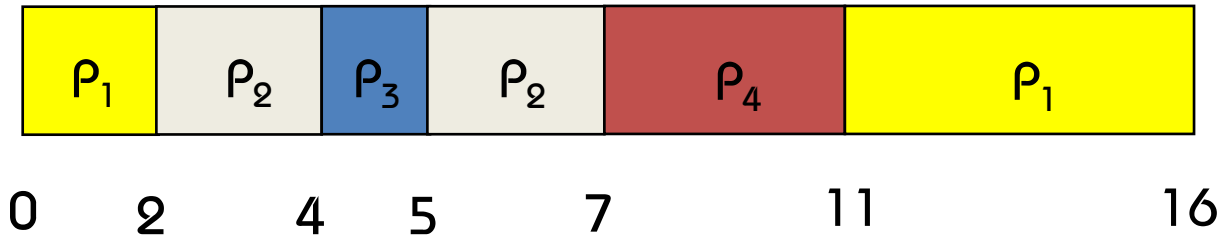
- 平均等待时间为 (0 + 6 + 3 + 7)/4 = 4

抢占式最短作业优先调度

- 给定的进程如下

进程	到达时间	区间时间
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- 抢占式 最短作业优先调度



- 平均等待时间= (9 + 1 + 0 +2)/4 = 3

- 每个进程都有它的优先级，通常用数字来表示进程的优先级，数字值越小它的优先级越高。分为非抢占式优先级调度，抢占式优先级调度

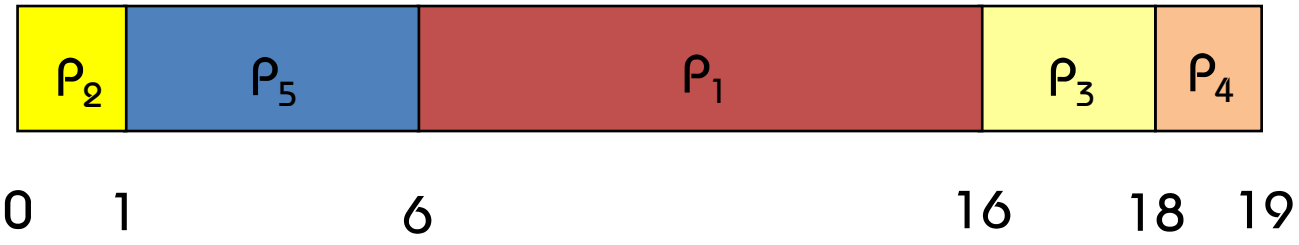
- 最短作业优先调度算法是优先级调度算法，一个进程的优先级与它的下一个CPU区间的长度成反比

- 举例 in the next slide

优先级调度算法例子

进程	区间时间	优先级
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

优先级调度甘特图



平均等待时间为（0+1+6+16+18+19）/5=8.2

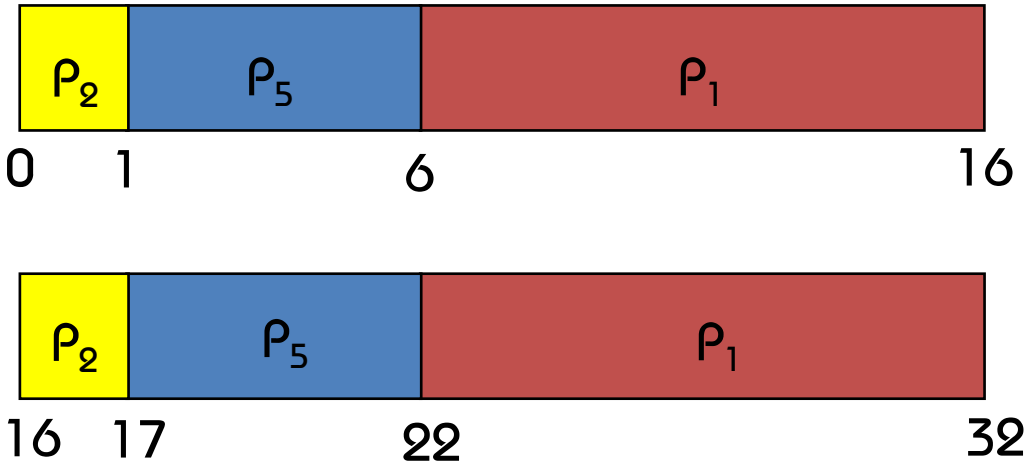
可能会发生的问题是

饥饿(starvation)或无限阻塞(infinite blocking)，

即低优先级进程无法运行的问题

举例 in the next slide

假设P1， P2， P5，每16个时间周期到达(release)一次，P3和P4就永远没有机会运行

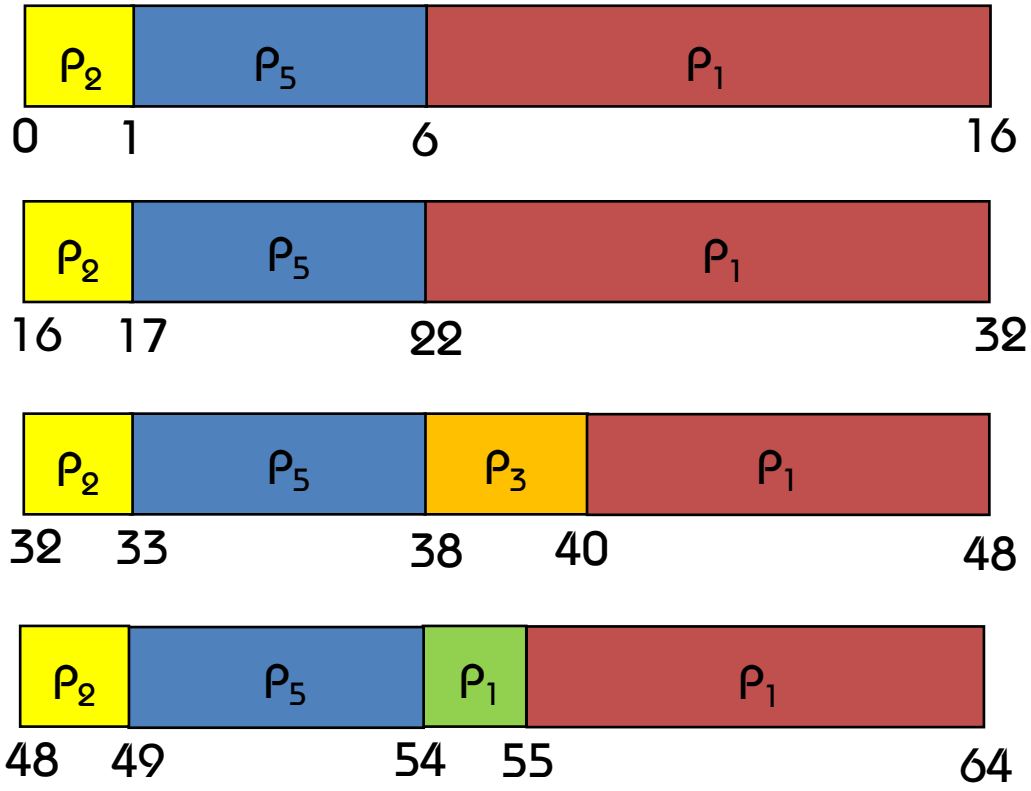


P3 and P4 in starvation, Infinite blocking

。。。

解决方法

– 老化(aging):即随着等待时间的长度，增加进程的优先级



P3 increases priority from 4 to 3
P4 increases priority from 5 to 4

P3 increases priority from 3 to 2
P4 increases priority from 4 to 3

P4 increases priority from 3 to 2

优先级调度算法例子

- Aging 注意的是，低优先级进程再次release时，它的优先级不是增加的优先级，而是原来的优先级。

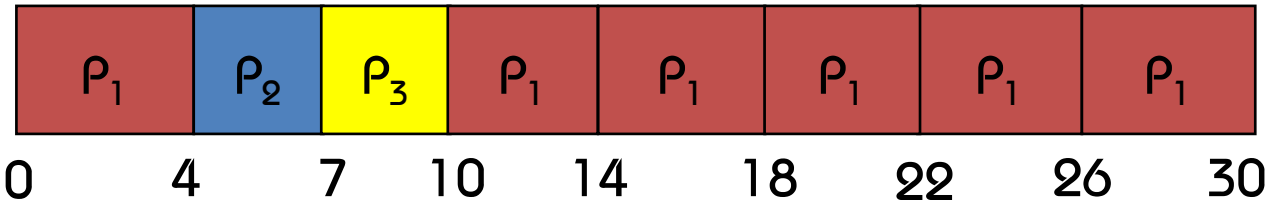
- 如上例子中P3和P4 再次release 时，它们的优先级还是4和5。

- 是FCFS调度算法的变种，给每个进程分配一个时间片(time slice, time quantum)。时间片一般是10-100毫秒
- 系统每隔一个时间片发出一个时钟中断(clock interrupt)，并调度另一个进程执行.
  1. 每个进程只能运行给定的时间片并释放CPU，并被放入到就绪队列 -（所以，RR调度是抢占调度）
  2. 如果进程在给定的时间片内提前结束，发生中断并调度另一个进程

时间片为4的轮转调度的例子

进程	区间时间
P_1	24
P_2	3
P_3	3

甘特图为:

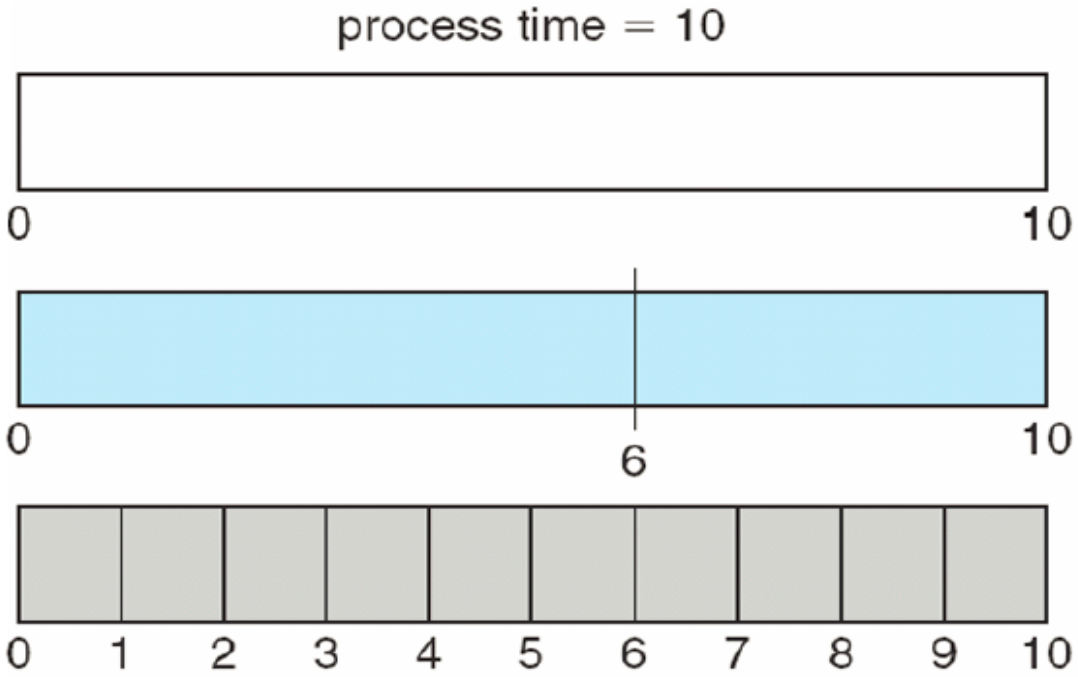


- 平均等待时间为 17/3 = 5.66
- 周转时间比最短作业优先调度长，但响应时间短
- 周转时间依赖于时间片的大小

那么，如何确定时间片的长度呢?

1. 如果时间片太大会导致轮转法调度与FCFS（FIFO）相同
2. 如果时间片太小会导致上下文切换负载太重

时间片和上下文切换时间

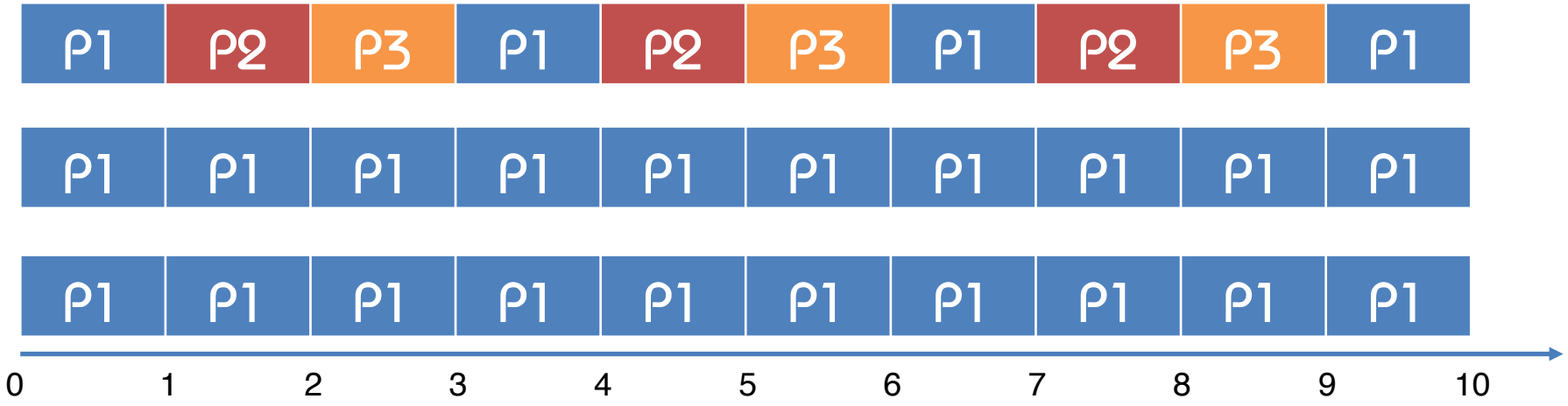


quantum	context switches
12	0
6	1
1	9

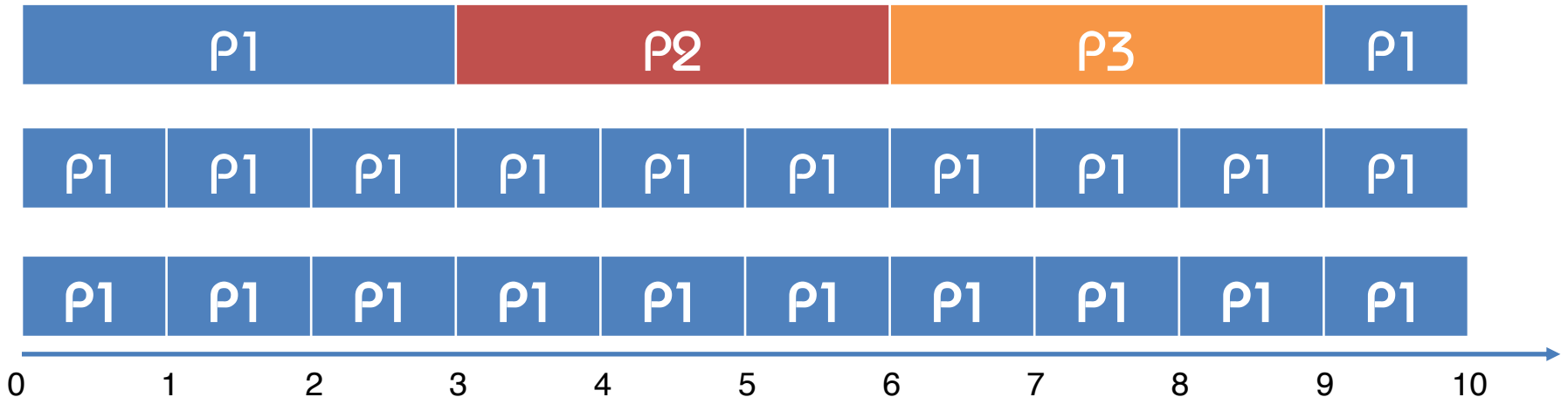
如果上下文切换时间约为时间片10%，那么约10%的CPU时间会浪费在上下文切换上

时间片和平均周转时间

- 通常，如果绝大多数进程能在一个时间片内完成，那么平均周转时间会改善。
- P1 区间时间 24; P2区间时间 3;P3区间时间 3为例

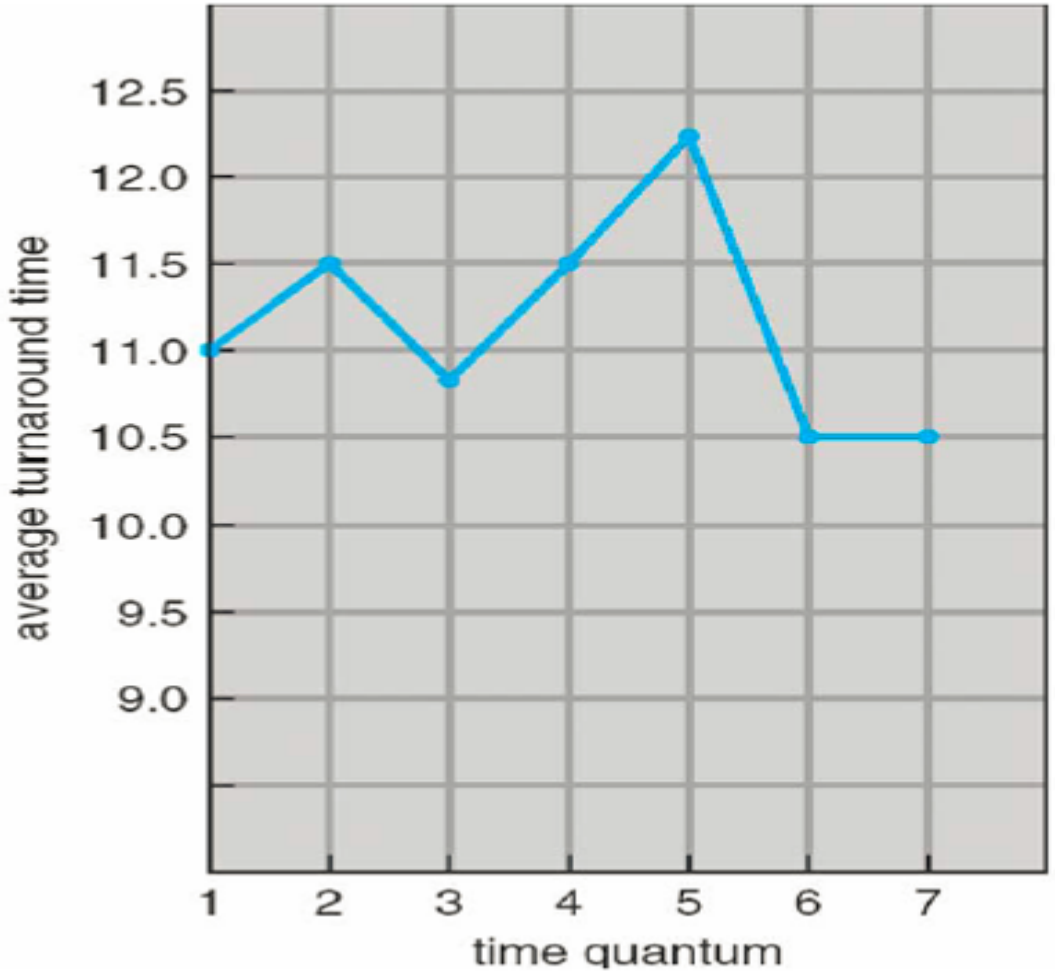


时间片为1 时，平均周转时间为 (30 + 8 + 9) / 3 = 15.7



时间片为3 时，平均周转时间为 30 + 6 + 9 / 3 = 15

根据时间片的周转时间变化



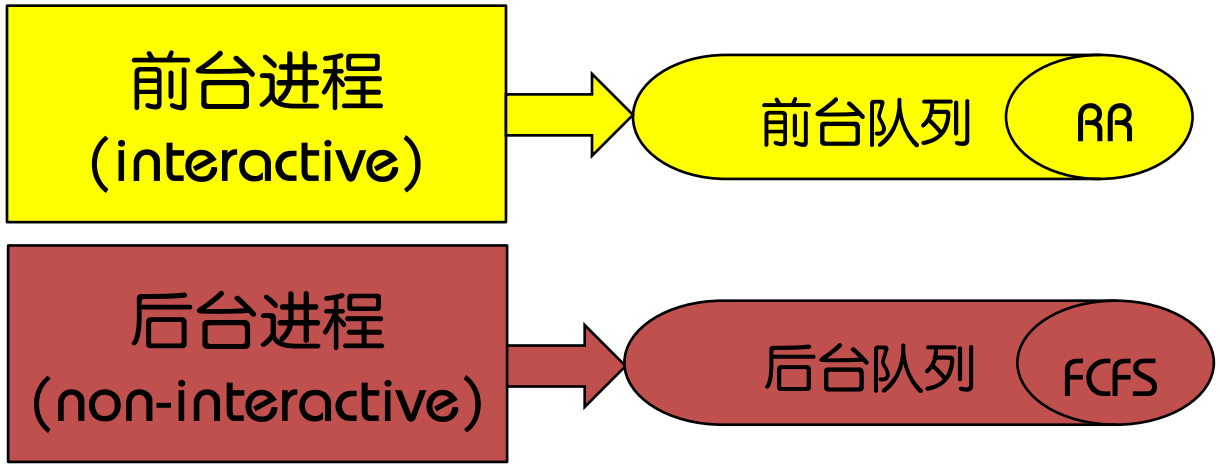
process	time
P_1	6
P_2	3
P_3	1
P_4	7

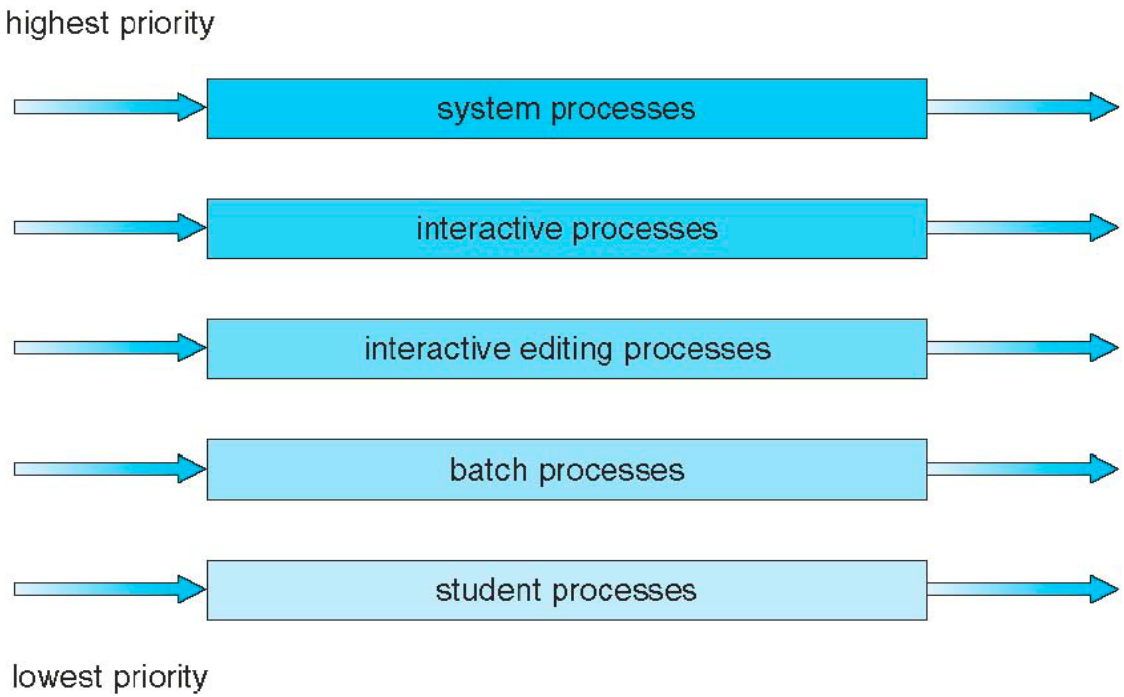
- 周转时间并未随着时间片大小的增加而改善
- 根据经验，时间片应大于80%的CPU区间

将就绪队列分成多个独立的队列，每个队列有自己的调度算法，并提供队列之间调度机制

1. 队列内的采用的调度机制
2. 队列之间可采用的调度机制
   - I. 固定优先级抢占调度
     - 如前台队列比后台队列具有高优先级，但有可能产生饥饿问题
   - II. 队列之间划分时间片
     - 每个队列都有给定的CPU时间， 这个时间可用于调度队列内的进程
     - 例如赋予前台队列80%CPU时间，赋予后台队列20%CPU时间

多级队列调度例子-固定优先级

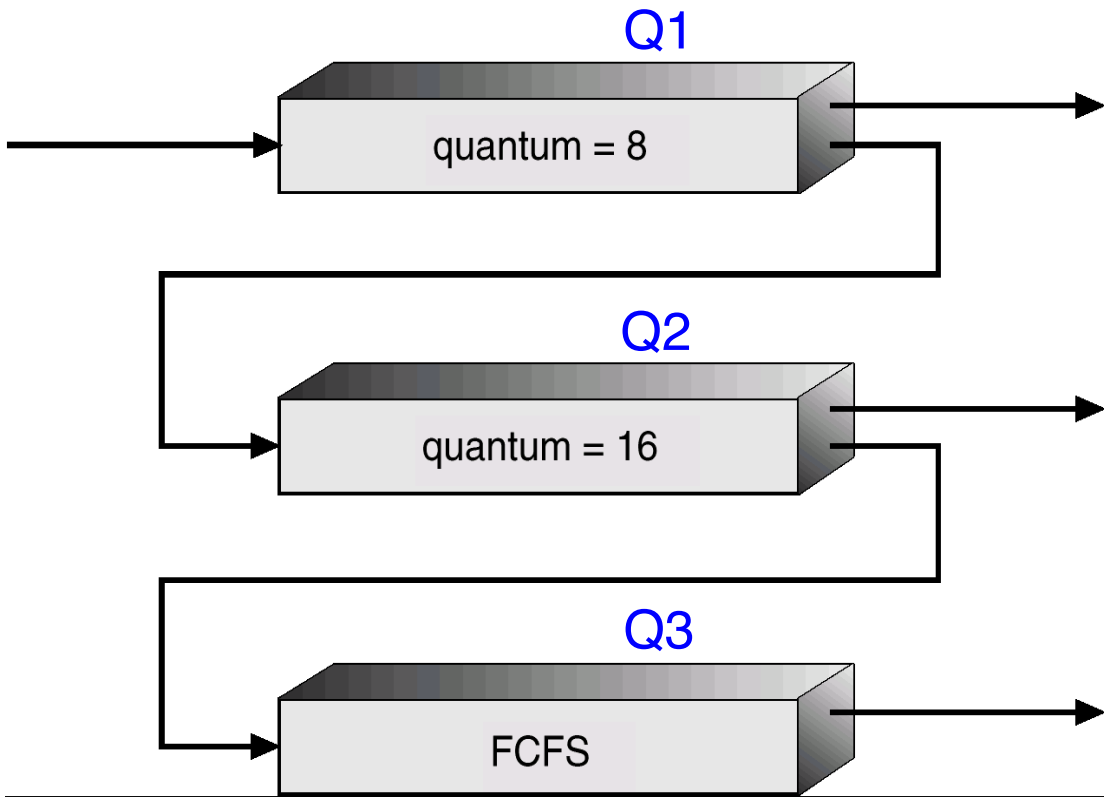




调度开销低，
缺乏灵活性

- Multi-Level Queue Feedback Scheduling
- 多级队列，每个队列都有自己的调度机制;
- 进程在多个队列之间有移动 - migration is allowed
  - 老化(aging)可以采用这种方式实现

It can be regarded as a kind of Feedback!



新进程放入Q1队列赋予8时间片。
如在8时间片内未完成执行，就放入Q2队列并赋予16时间片。
如在给定的时间片内未完成执行，就放入Q3队列

Q1 level > Q2 level > Q3 level

调度程序需要确定下列参数

1. 队列数量
2. 每个队列的调度算法
3. 确定何时升级到更高优先级队列的方法
4. 确定何时降级到更低优先级队列的方法
5. 确定进程在需要服务时应进入哪个队列的方法

第三节 多处理器调度

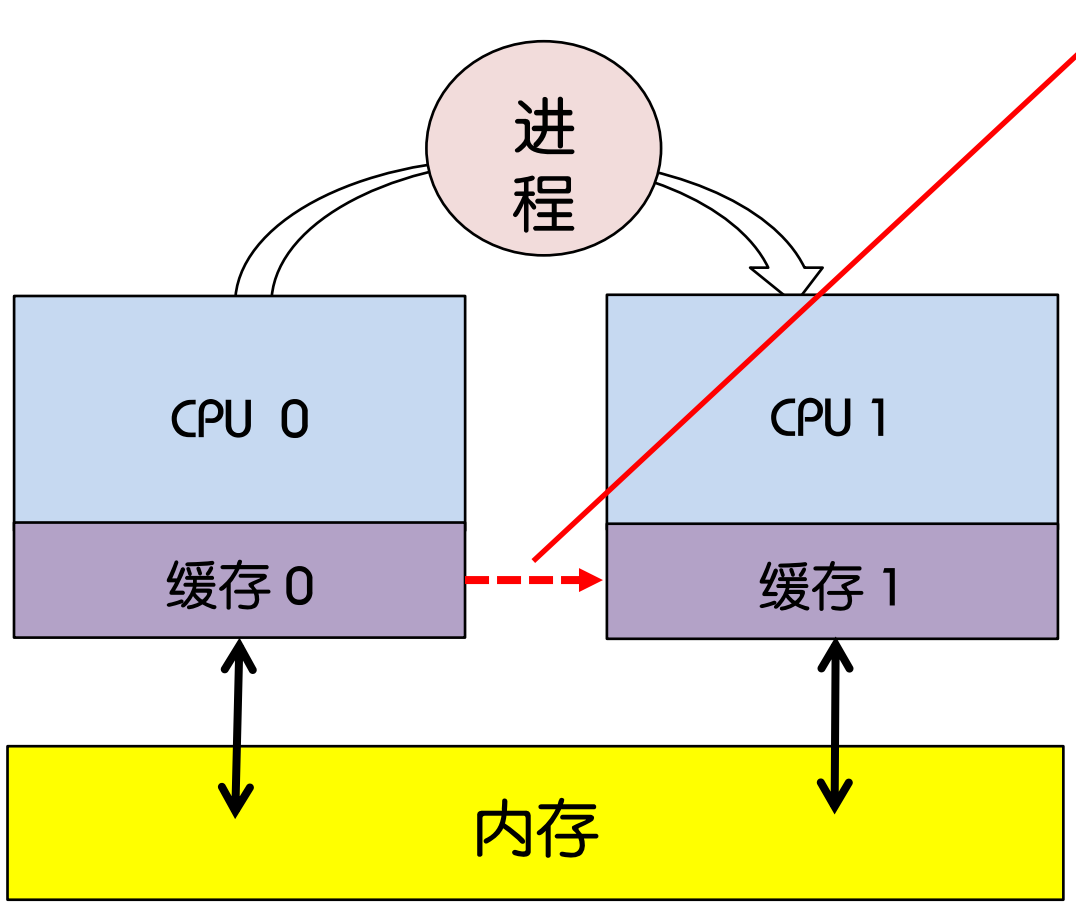
4. 多处理器调度

1. Homogeneous （同构处理器）:处理器功能相同
2. Heterogeneous（异构处理器）:处理器功能不同

- 对称多处理(Symmetric）:又称SMP，每个处理器有自己的调度算法，不允许处理器之间的移动

- 非对称多处理 (Asymmetric) :只有一个处理器进行调度任务，允许处理器之间的移动

处理器亲和性（CPU Affinity）



需要实时的同步
缓存数据

尽量把进程放入到同一个处理器上运行

1. 软亲和性:允许处理器之间的移动
2. 硬亲和性:不允许处理器之间的移动

负载平衡（load balancing）

- 在SMP系统中，保持所有处理器的工作负载平衡
- 负载平衡是设法将工作负载平均地分配到SMP系统中的所有处理器上
- 负载平衡通常有两种方式，并相互不能排斥

1. Push（推） migration – periodic task checks load on each processor, if found unbalancing, pushes task from overloaded CPU to idle or not busy CPUs
2. Pull（拉） migration – idle processors pulls waiting task from busy processor

第四节 线程调度

区分用户线程和内核线程， 多线程系统的调度对象是线程

1. Local Scheduling (Many-to-One and Many-to-Many Models): 线程库调度用户级线程到一个有效的 LWP【也就是用户线程映射内核线程】
   - 被称为进程竞争范围（Process-Contention Scope :PCS）
   - 根据优先级完成，一般是由程序员给定
   - 竞争发生在相同进程的线程之间
2. Global Scheduling (One-to-One Model): 系统将内核线程调度到有效的物理处理器上，被称为系统竞争范围（System Contention Scope :SCS）
   - 竞争发生在系统的所有线程之间（Window XP)

- 在线程创建过程中允许指定PCS 或 SCS

  1. PTHREAD_SCOPE_PROCESS 调度线程采用 PCS
  2. PTHREAD_SCOPE_SYSTEM 调度线程采用SCS

- 设置线程优先级级别

3. int pthread_attr_setschedparam(pthread_attr_t * attr,
   const struct sched_param * param)
4. int pthread_attr_getschedparam(const pthread_attr_t * attr,
   struct sched_param * param);

```
#include <pthread.h>
#include <stdio.h>
#define NUM THREADS 5
int main(int argc, char *argv[]) {
   int i, scope;
   pthread_t tid[NUM THREADS];
   pthread_attr_t attr;
   /* get the default attributes */
   pthread_attr_init(&attr);
   /* first inquire on the current scope */
   if (pthread_attr_getscope(&attr, &scope) != 0)
      fprintf(stderr, "Unable to get scheduling scope\n");
   else {
      if (scope == PTHREAD_SCOPE_PROCESS)
         printf("PTHREAD_SCOPE_PROCESS");
      else if (scope == PTHREAD_SCOPE_SYSTEM)
         printf("PTHREAD_SCOPE_SYSTEM");
      else
            fprintf(stderr, "Illegal scope value.\n");
   }
                                        /* to be continued */
```

```
/* set the scheduling algorithm to PCS or SCS */
pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
   /* create the threads */
      for (i = 0; i < NUM_THREADS; i++)
         pthread_create(&tid[i],&attr,runner,NULL);
   /* now join on each thread */
      for (i = 0; i < NUM_THREADS; i++)
         pthread_join(tid[i], NULL);
}
/* Each thread will begin control in this function */
void *runner(void *param)
   {
   /* do some work ... */
   pthread_exit(0);
}
```

```
static int get_thread_policy(pthread_attr_t attr)
{
    int policy;
    pthread_attr_getschedpolicy(&attr, &policy);
    switch(policy)
    {
        case SCHED_FIFO:
            printf("policy = SCHED_FIFO\n");
            break;
        case SCHED_RR:
            printf("policy = SCHED_RR\n");
            break;
        case SCHED_OTHER:
            printf("policy = SCHED_OTHER\n");
            break;
        default:
            printf("policy = UNKOWN\n");
            break;
    }
    return policy;
}
```

```
#include <unistd.h>
#include <pthread.h>
#include <sched.h>
```

```c
static void show_thread_priority(pthread_attr_t attr, int policy)
{
    int priority = sched_get_priority_max(policy);
    printf("max_priority = %d\n", priority);
    priority = sched_get_priority_min(policy);
    printf("min_priority = %d\n", priority);
}

static int get_thread_priority(pthread_attr_t attr)
{
    struct sched_param param;
    pthread_attr_getschedparam(&attr, &param);
    printf("priority = %d\n", param.sched_priority);
    return param.sched_priority;
}

static void set_thread_policy(pthread_attr_t attr, int policy )
{
    pthread_attr_setschedpolicy(&attr, policy);
    get_thread_policy(attr);
}
```

```c
int main(void)
{
    pthread_attr_t attr;
    struct sched_param sched;
    int rs = pthread_attr_init(&attr);
    int policy = get_thread_policy(attr);
    printf("- show current configuration of priority\n");
    show_thread_priority(attr, policy);
    printf("- show SCHED_FIFO of priority\n");
    show_thread_priority(attr, SCHED_FIFO);
    printf("- show SCHED_RR of priority\n");
    show_thread_priority(attr, SCHED_RR);
    printf("- show priority of current thread\n");
    int priority = get_thread_priority(attr);
    printf("SET THREAD POLICY\n");
    printf("set SCHED_FIFO policy\n");
    set_thread_policy(attr, SCHED_FIFO);
    printf("set SCHED_RR policy\n");
    set_thread_policy(attr, SCHED_RR);
    printf("restore current policy\n");
    set_thread_policy(attr, policy);
    pthread_attr_destroy(&attr);
    return 0;
}
```

第五节 实时调度

6. 实时调度（Real-Time Scheduling）

实时系统中的进程具有截止时间(deadline)要求，进程必须在给定的截止时间内完成执行，有截止时间的进程称为实时任务(real-time tasks)

1. 硬实时系统（Hard Real-Time Systems）
   - 具有硬截止时间
   - 必须在截止时间内结束任务
   - 例如，军用设备、医疗设备 等
2. 软实时系统（Soft Real-Time Systems）
   - 任务具有软截止时间
   - 可以适当的超过截止时间结束任务
   - 例如，音频、视频系统 等

Task Model

1. Periodic Tasks: the task is released periodically
2. Aperiodic Tasks:is not released periodically, no deadline, so the response time is performance index
3. Sporadic Tasks:is not released periodically, but is released a given time interval, have deadline

Real-Time Scheduling Algorithm

1. Rate Monotonic
2. Earliest Deadline First
3. Earliest Deadline Zero Laxity

- Earliest Deadline First
  - 急的任务先处理的算法
  - Deadline最近的就是急的任务
- Earliest Deadline Zero Laxity (EDZL)
  - EDF + Zero Laxity
  - Laxity = absolute deadline – current time - remaining execution time（不能再延迟执行的任务）