

高级大数据解析

Module 2: 深入理解操作系统的运行机制

(1) 操作系统处理器管理

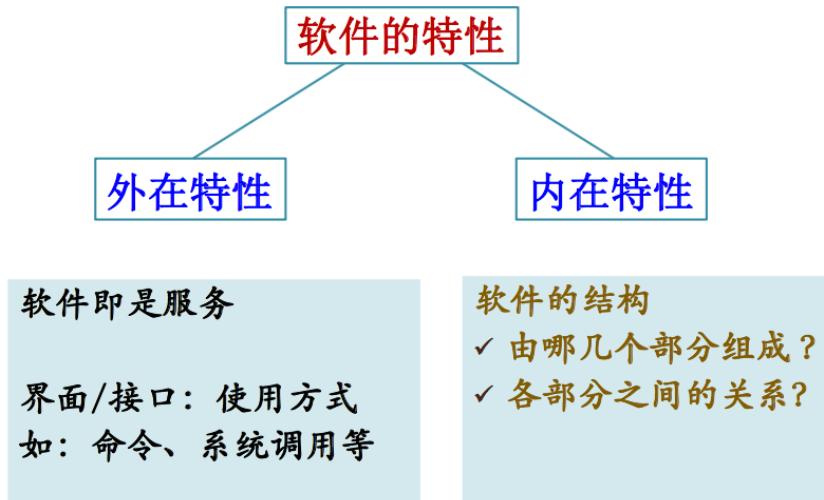
Module 1: Intro. & Linux Shell

Module 2: Insights of OS

Module 3: Quick intro. To Spark

<https://www.ee.columbia.edu/~cylin/course/bigdata/>

I. 作为软件来看的观点



怎样管理资源?

- 跟踪记录资源使用状况
 - 如: 哪些资源空闲, 好坏与否, 被谁使用, 使用多长时间等
- 分配和回收资源 (资源分配策略与算法)
 - 静态分配策略
 - 动态分配策略 ✓
- 提高资源利用率
- 保护
- 协调多个进程对资源请求的冲突

公平竞争
防止非法
使用

2. 资源管理的观点

自底向上 → 操作系统 是 **资源的管理者**

硬件资源:

**CPU, 内存, 设备 (I/O设备、磁盘、时钟、网络
接口等)**

软件资源:

磁盘上的文件、信息

从资源管理的角度—五大基本功能

- 进程和线程管理 (CPU管理、调度)
进程控制、同步互斥、通信、调度
- 存储管理
分配/回收、地址映射、存储保护、内存扩充
- 文件管理
文件目录、磁盘空间、文件系统布局、存取控制
- 设备管理
设备驱动、分配回收、缓冲技术
- 用户接口
系统命令、编程接口

3. 进程的观点

从操作系统运行的角度动态的观察操作系统

按照这一观点：

- 操作系统是由一些可同时、独立运行的进程和一个对这些进程进行协调的核心组成

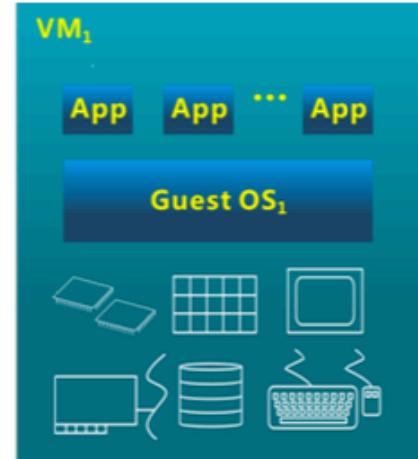
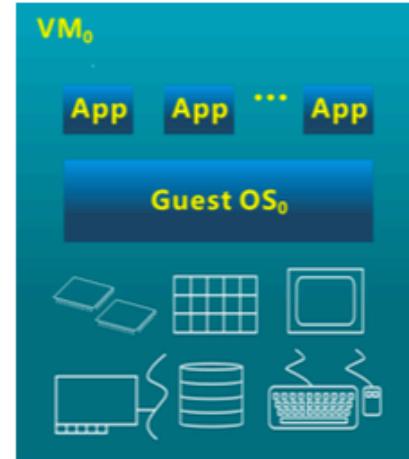
进程：完成某一特定功能的程序
是程序的一次执行过程
动态的、有生命的，存在/消亡

4. 虚机器观点

从操作系统内部结构来看：

- 把操作系统分成若干层 分层结构
- 每一层完成其特定功能从而构成一个虚机器，并对上一层提供支持
- 通过逐层功能扩充，最终完成整个操作系统虚机器
- 而操作系统虚机器向用户提供各种功能，完成用户请求

无虚拟机 / 有虚拟机 各有什么区别？



无虚拟机：单操作系统拥有所有硬件资源

有虚拟机：多操作系统共享硬件资源

关键词解读

有效：系统效率，资源利用率

(CPU利用的充足与否，I/O设备是否忙碌)

合理：

是否公平合理？

如果不公平则会产生？

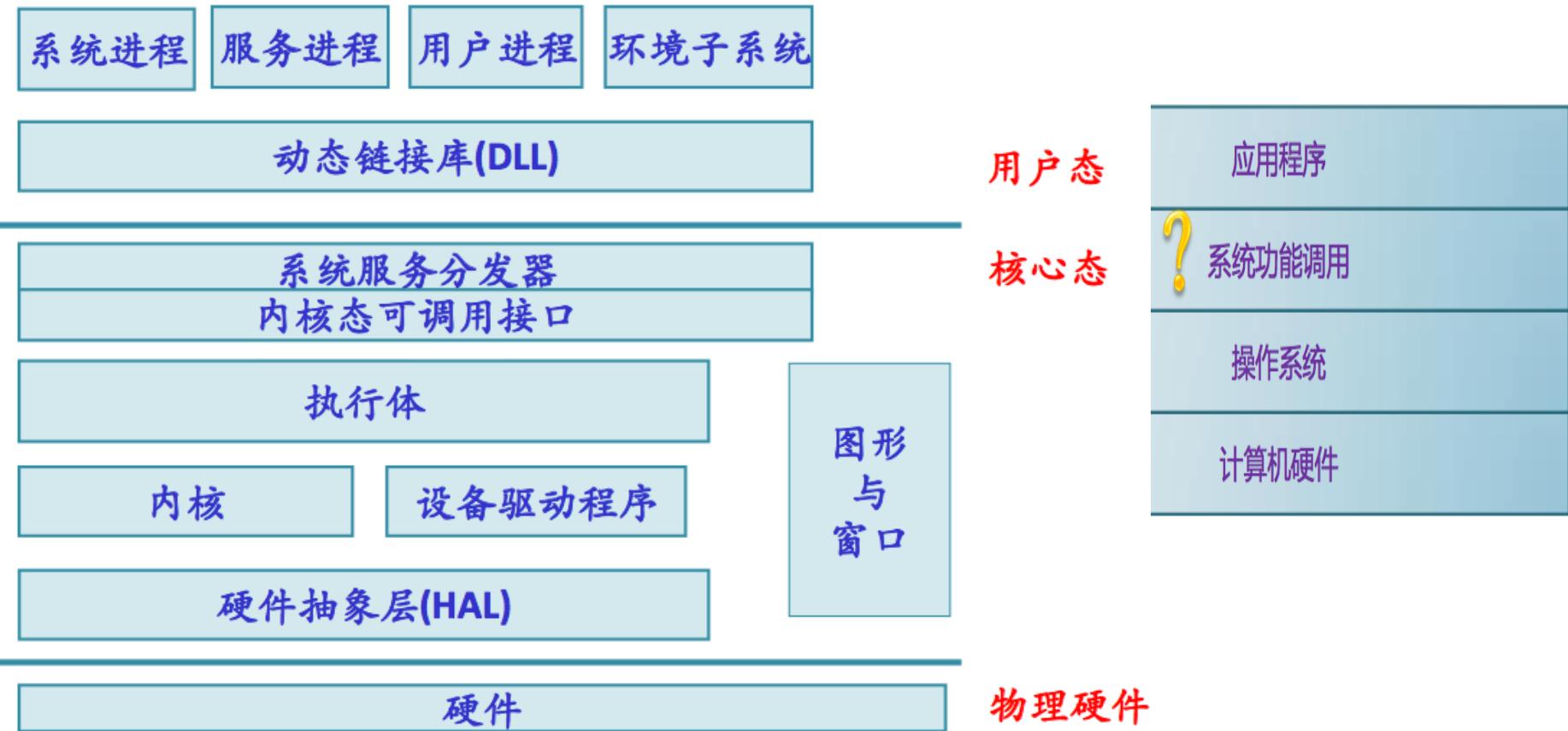


如果不合理可能会产生？

方便使用：

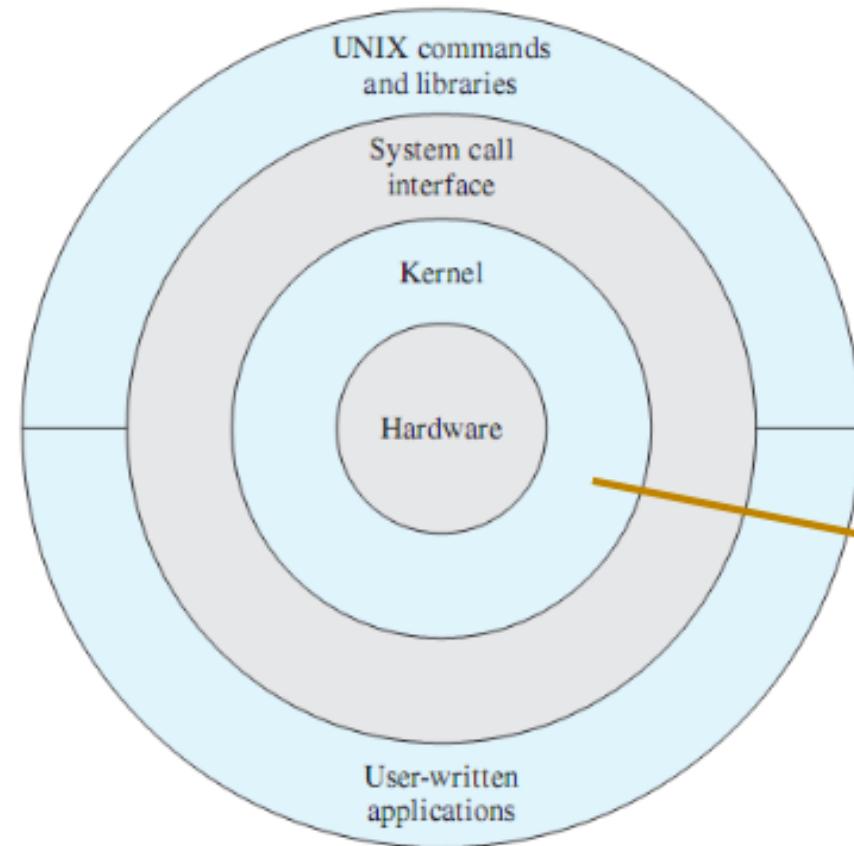
两种角度：用户界面 与 编程接口

Windows操作系统的体系结构

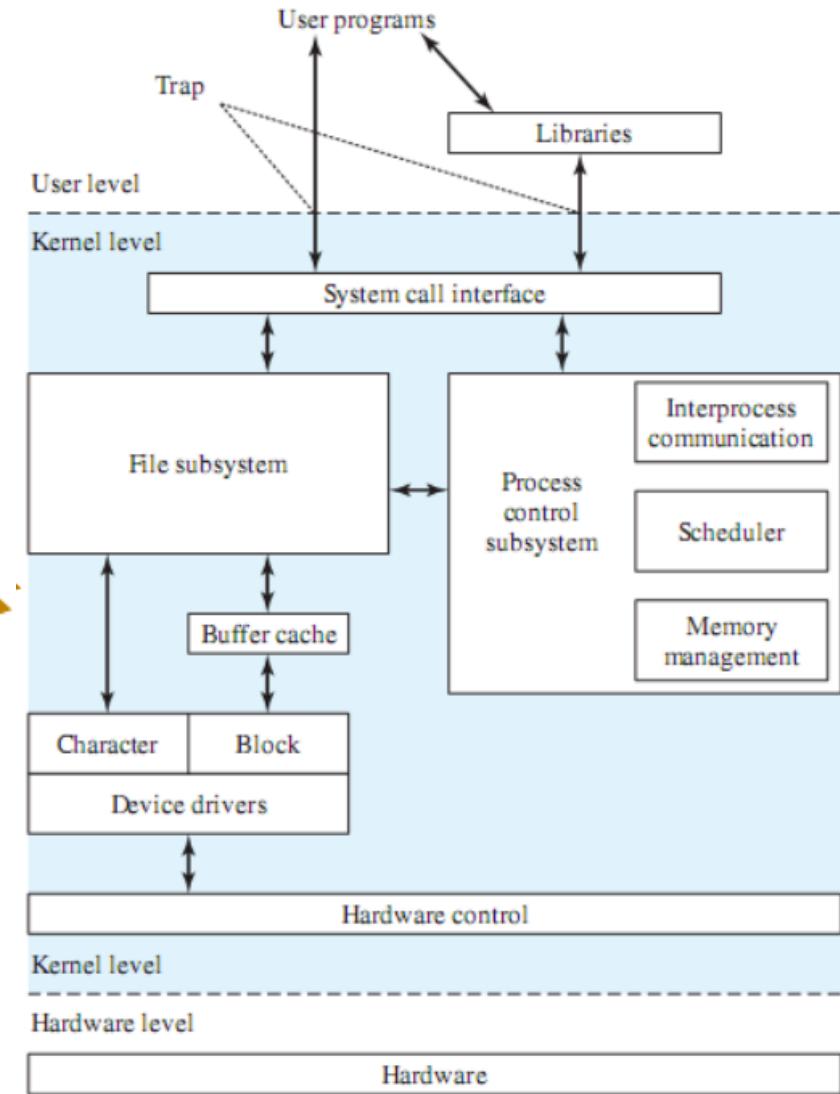


UNIX 操作系统的体系结构

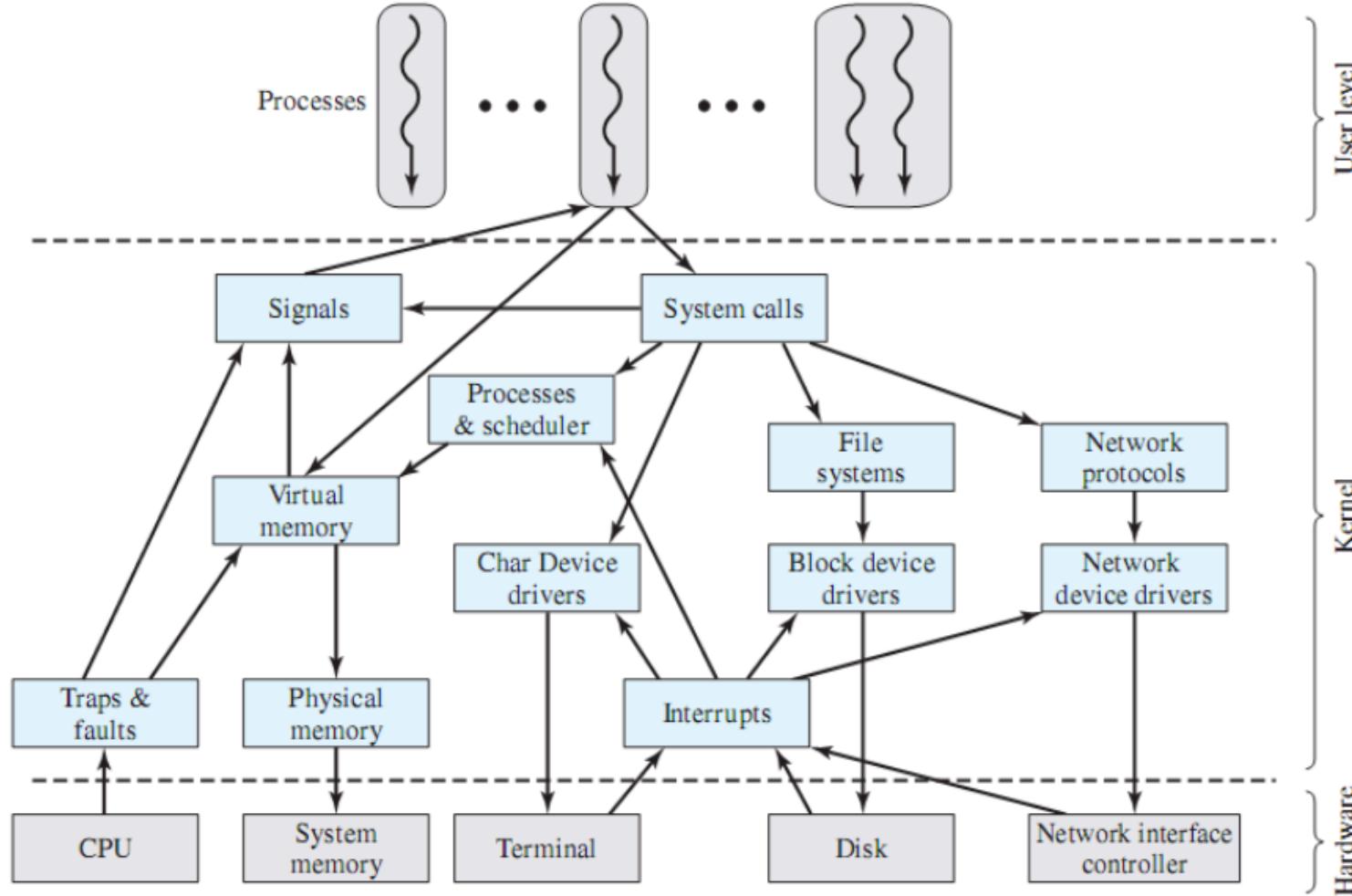
内核结构



层次结构



Linux 架构



Linux操作系统内核



Android 架构

Android应用程序

Email客户端，SMS短消息程序，日历，地图，浏览器，联系人管理等

应用程序框架

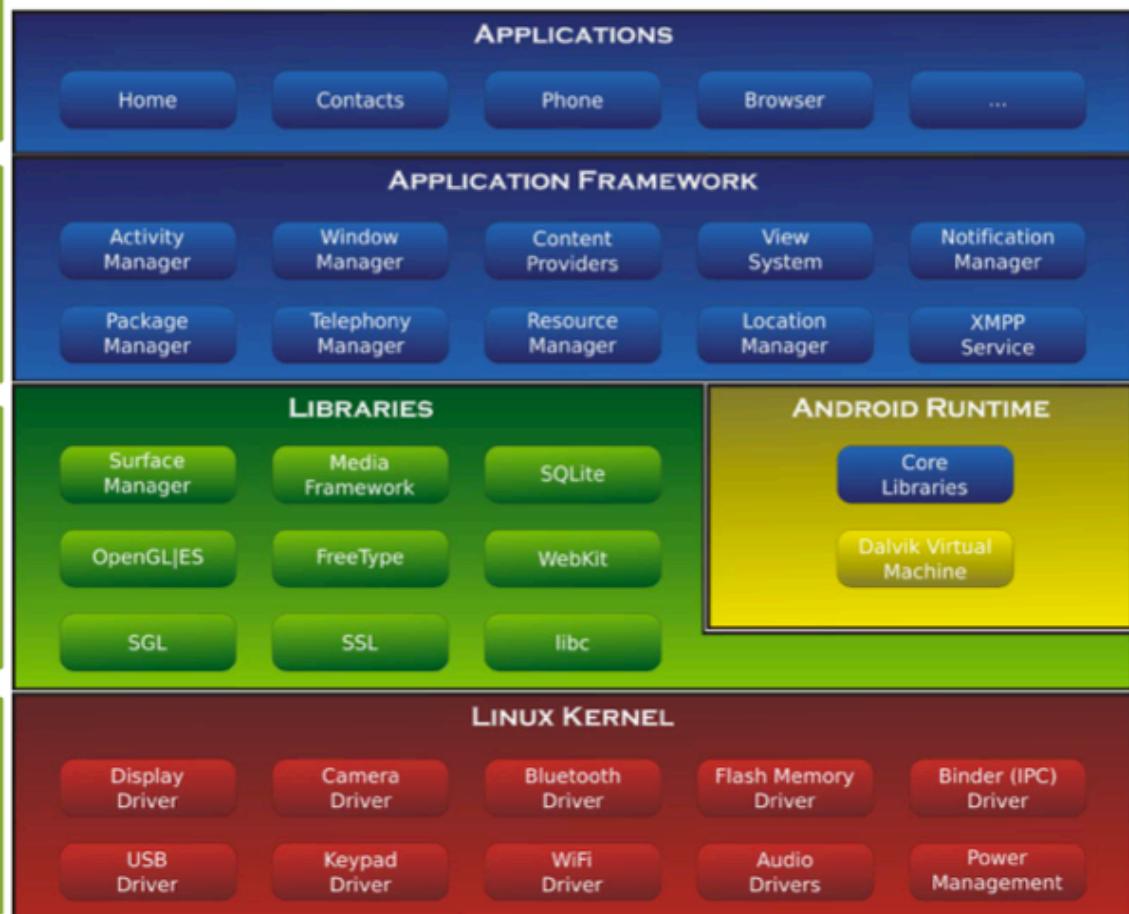
开发者可以完全使用核心应用程序所使用的框架APIs
视图、内容提供者、资源管理器等

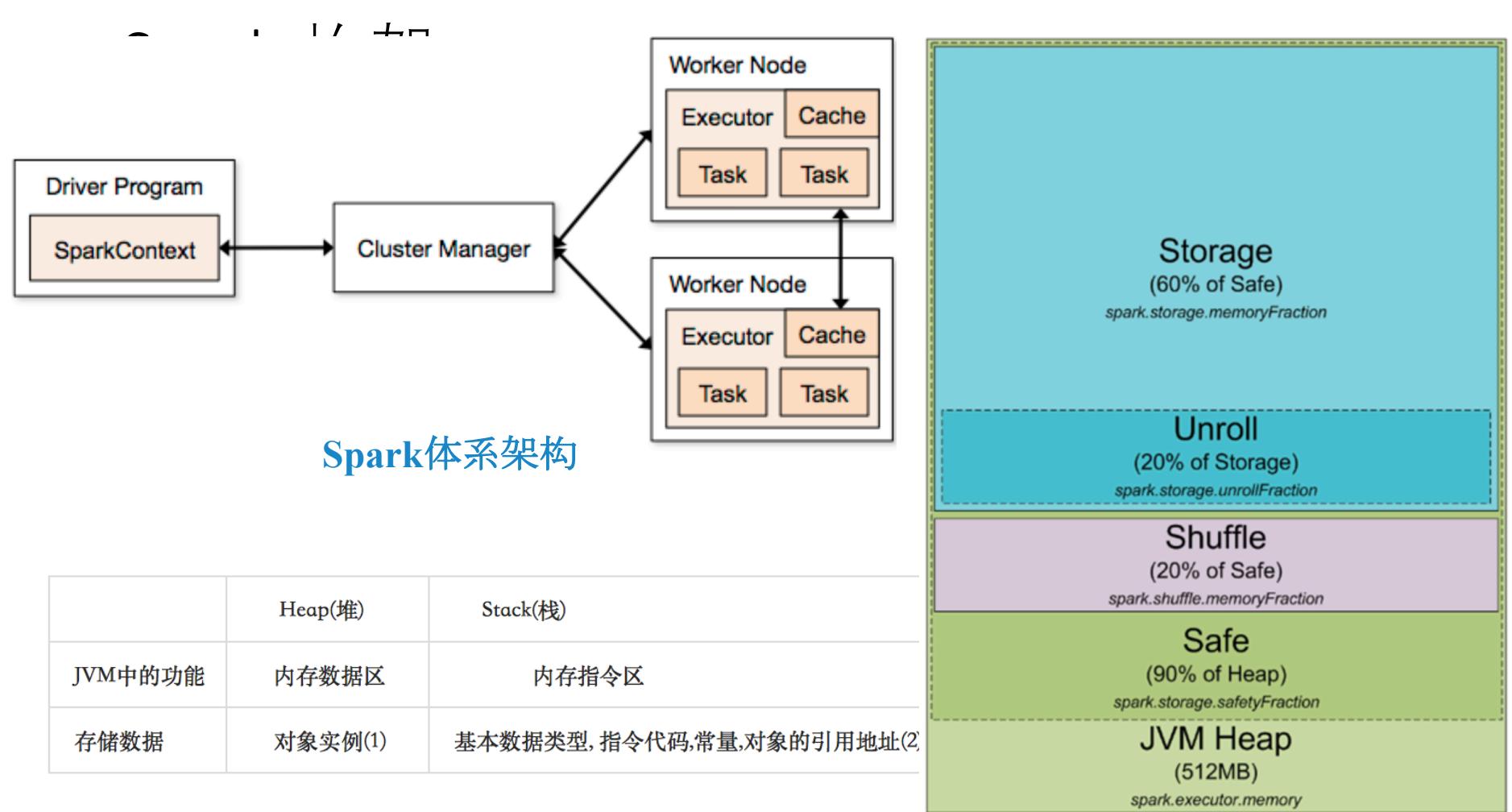
库

Android包含一个C/C++库的集合，供Android系统的各个组件使用。
如：系统C库、3D库、SQLite、媒体库等

Linux内核

提供核心系统服务，例如：安全、内存管理、进程管理、网络堆栈、驱动模型

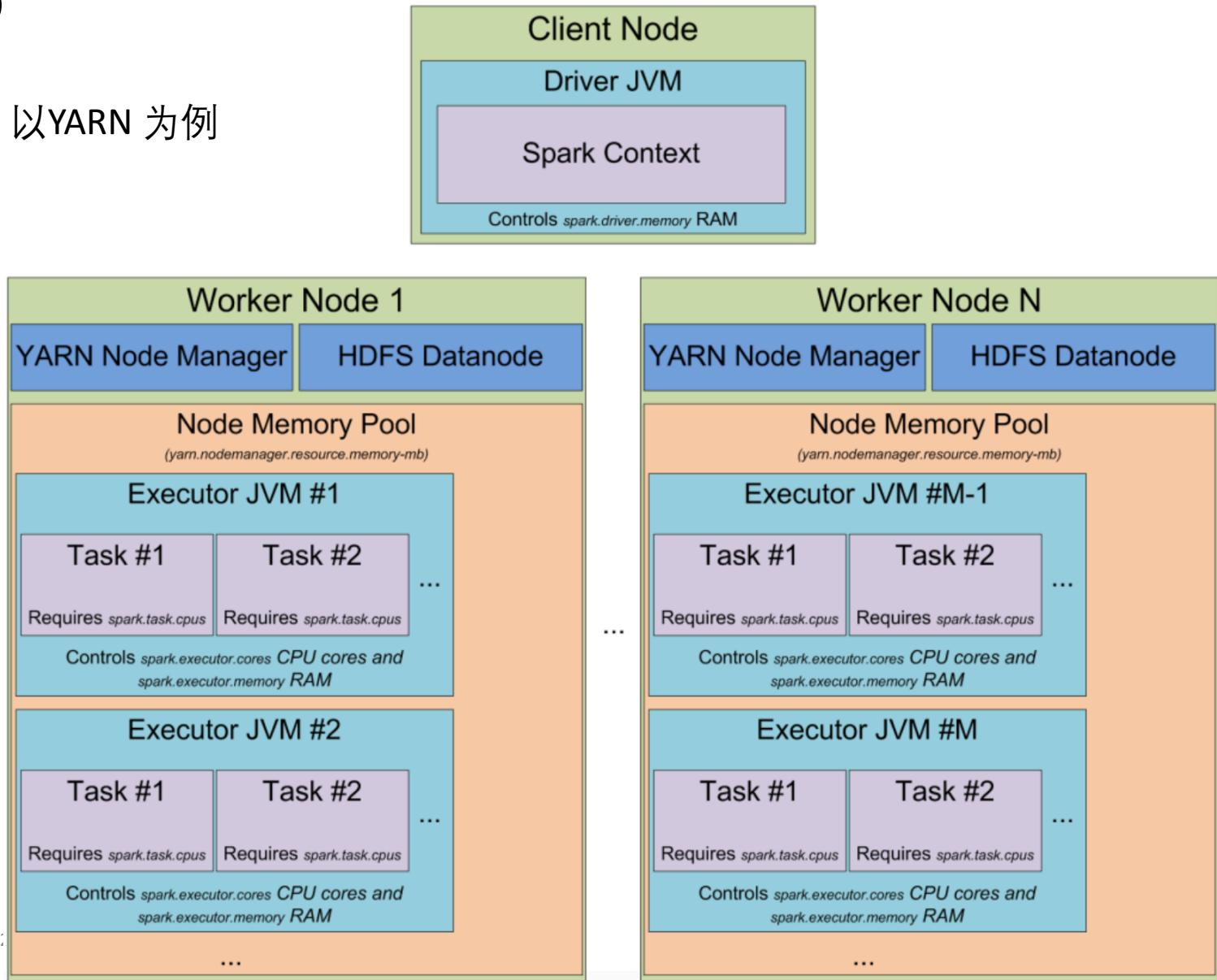




Spark JVM堆内存分配图

Sp

以YARN为例



总结：操作系统的特征

并发：处理多个同时性活动的能力

共享：操作系统与多个用户程序共同使用计算机系统中的资源。

虚拟：一个物理实体映射为若干个对应的逻辑实体--分时或分空间。虚拟技术是操作系统管理系统资源的重要手段， 可提高资源利用率

随机：操作系统必须随时对以不可预测的次序发生的事件 进行响应

操作系统的分类

- 批处理操作系统（多道批处理）
- 分时系统
- 实时操作系统
- 个人计算机操作系统
- 网络操作系统
- 分布式操作系统
- 嵌入式操作系统

通用操作系统

TinyOS, Raspberry Pi

Raspberry Pi(树莓派)

➤ 在树莓派上尝试使用 darkflow

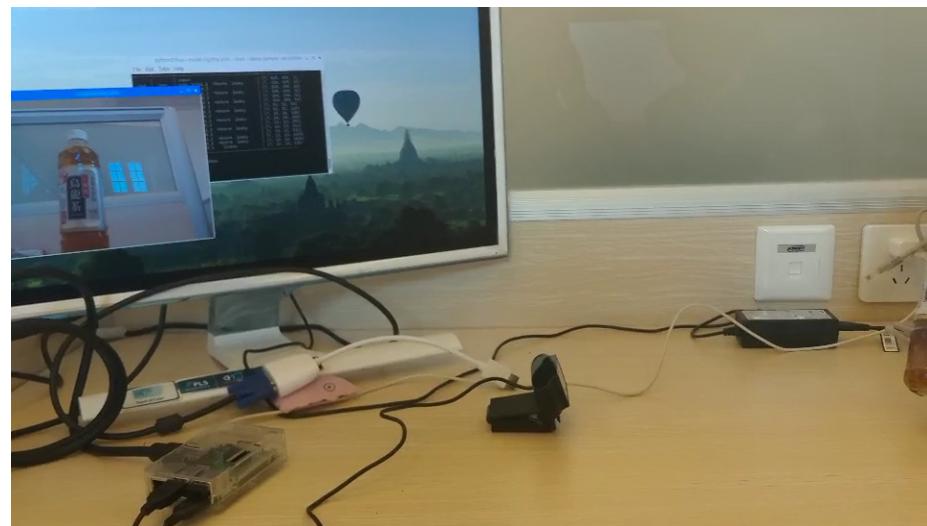
目前在 **树莓派** 上成功编译 tensorflow 和 opencv，并试运行 darkflow 框架

运行时间统计：

tiny-yolo-voc：载入网络 155.76s / 8 张图片 71.3s

tiny-yolo-drink : 24 张图片 : 189.23s / 9 张未识别

实时监控视频：约 0.1 fps



操作系统做了什么? (1/4)

```
#include <stdio.h>
int main(int argc, char *argv[])
{
    puts("hello world");
    return 0;
}
```

操作系统做了什么? (3/4)

- 执行程序的第一条指令，发生[缺页异常](#)
- 操作系统：分配一页物理内存，并将代码从磁盘读入内存，然后继续执行程序
- **helloworld**程序执行**puts**函数（系统调用），在显示器上写一字符串
- 操作系统：找到要将字符串送往的显示设备，通常设备是由一个进程控制的，所以，操作系统将要写的字符串送给该进程

操作系统做了什么? (2/4)

- 用户告诉操作系统执行程序([如何告知?](#))
- 操作系统：找到程序的相关信息，检查其类型是否是可执行文件；并通过程序首部信息，确定代码和数据在可执行文件中的位置并计算出对应的磁盘块地址 ([文件格式?](#))
- 操作系统：创建一个新的进程，并将可执行文件映射到该进程结构，表示由该进程执行程序
- 操作系统：为程序设置CPU上下文环境，并跳到程序开始处 ([假设调度程序选中hello程序](#))

操作系统做了什么? (4/4)

- 操作系统：控制设备的进程告诉设备的窗口系统它要显示字符串，窗口系统确定这是一个合法的操作，然后将字符串转换成像素，将像素写入设备的存储映像区
- 视频硬件将像素转换成显示器可接收的一组控制/数据信号
- 显示器解释信号，激发液晶屏
- **OK! ! !** 我们在屏幕上看到了“**hello world**”

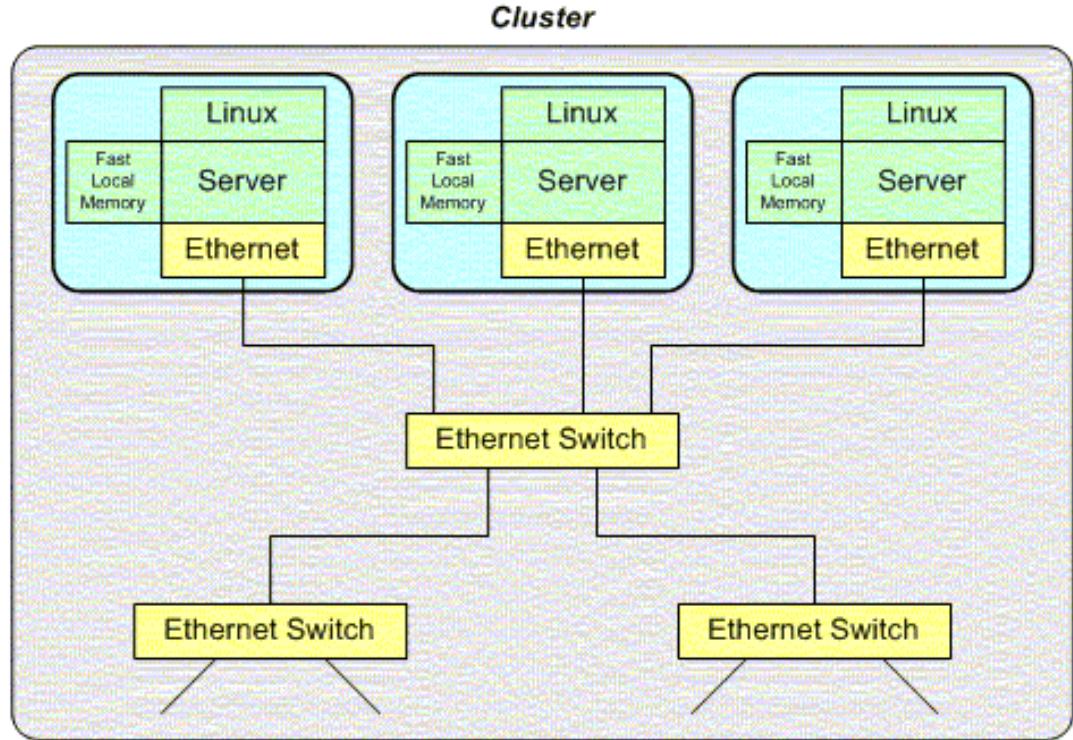
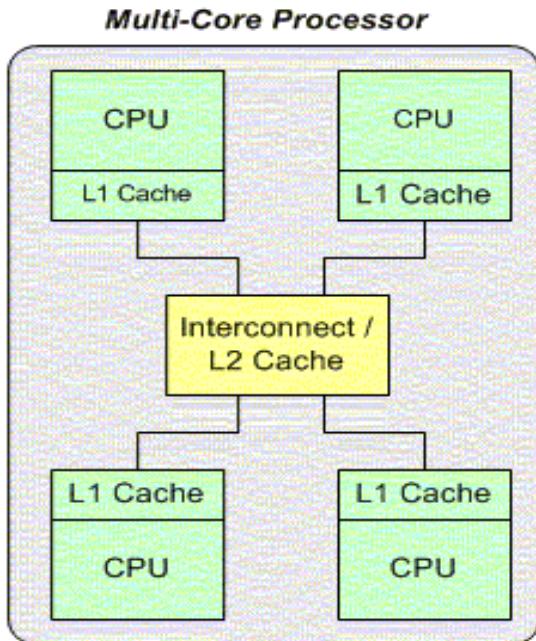
So, what happened when I submit a Spark Job?

Next: 处理器管理

- 处理器管理是操作系统最核心的功能
 - 处理器及中断
 - 进程及其实现
 - 线程及其实现
 - 处理器调度
 - Linux和Windows处理器调度算法

2.1 中央处理器 (CPU)

- 单处理器
 - 指令流水线
- 多处理器
 - 松耦合
 - 紧耦合



单指令流单数据流(SISD, Single instruction, single data)
单指令流多数据流(SIMD)
多指令流单数据流(MISD)
多指令流多数据流(MIMD)

紧耦合多处理器

- 主从式
 - 主：操作系统内核
 - 从：其他程序
- 对称式(SMP)
 - 每个处理器是对等的
 - 内核可以运行在任意处理器上
 - 负载均衡

AMD	AMD X2	SMP, 双 CPU
Intel®	Xeon	SMP, 双 CPU 或四 CPU
Intel	Core2 Duo	SMP, 双 CPU
ARM	MPCore	SMP, 最多四 CPU



寄存器

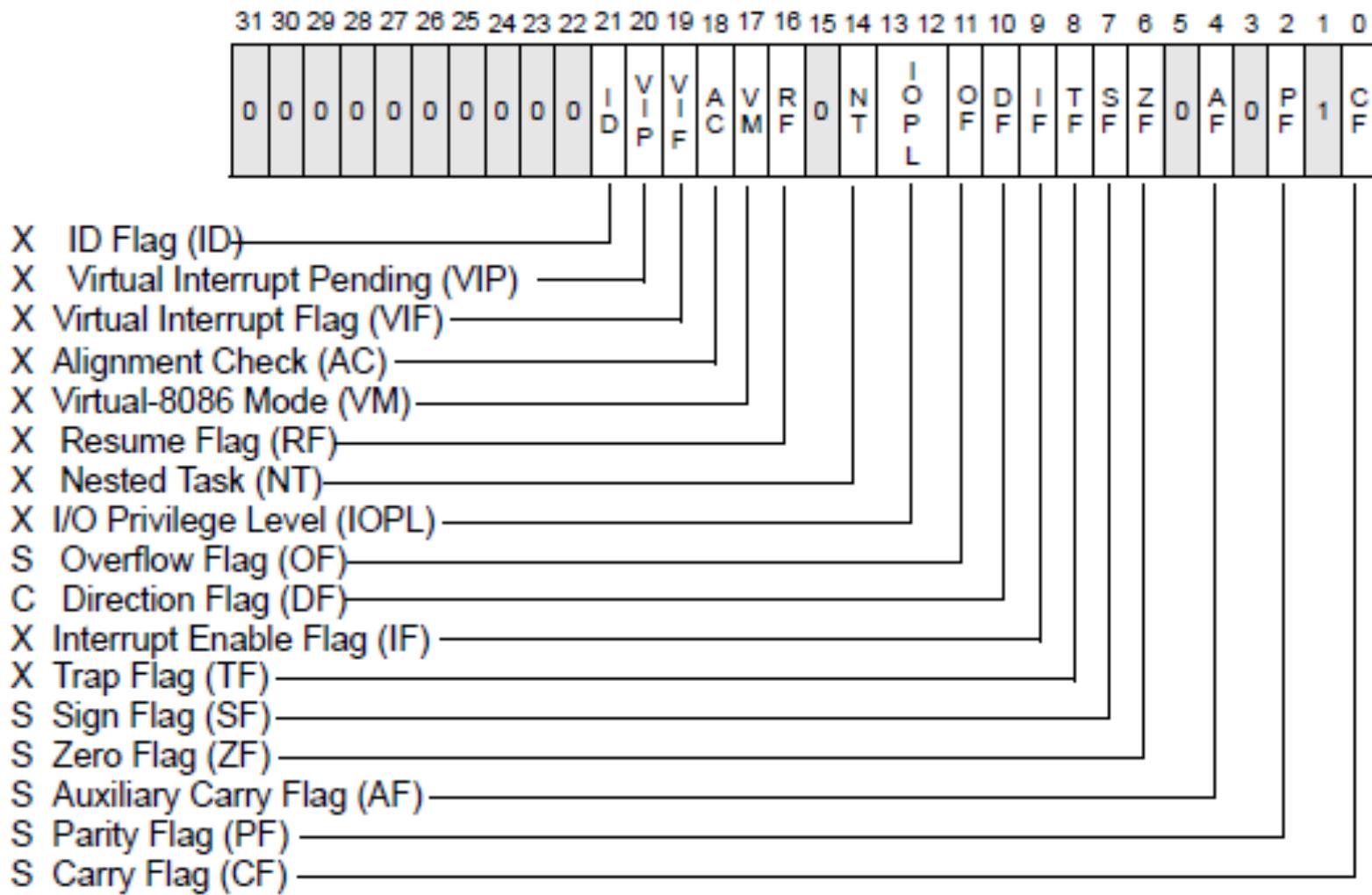
- CPU的重要组成部分
- 一级存储器
- 机型不同数量不同
- 寄存器所存储的信息与程序的执行有很大关系，构成了处理器现场

两类寄存器

- **用户可见寄存器**: 高级语言编译器通过优化算法分配并使用之, 以减少程序访问内存次数.
- **控制和状态寄存器**: 用于控制处理器的操作 通常由操作系统代码使用.

寄存器分类

- 通用寄存器-- EAX, EBX, ECX和EDX
- 指针及变址寄存器--ESP, EBP, ESI及EDI
- 段选择符寄存器--CS、DS、SS、ES 、FS、 GS
- 指令指针寄存器和标志寄存器--EIP、EFLAGS
- 控制寄存器--CR0, CR1, CR2和CR3
- 外部设备使用的寄存器



特权指令与非特权指令

- 指令系统：机器指令集合
 - x86
- 指令分类：
 - 数据处理类指令；
 - 转移类指令；
 - 数据传送类指令；
 - 移位与字符串指令；
 - I/O类指令。

大多数系统把处理器状态简单的划分为
管理状态(又称特权 状态、系统模式、
特态或管态) 和**用户状态**(又称目标状态、
用户模式、常态或目态)

特权指令与非特权指令

- 操作系统将指令划分成：特权指令和非特权指令
- 特权指令
 - 有关资源管理的指令
 - 只能由操作系统执行
- 用户程序执行特权指令：保护中断

下列哪些是特权指令？哪些是非特权指令？

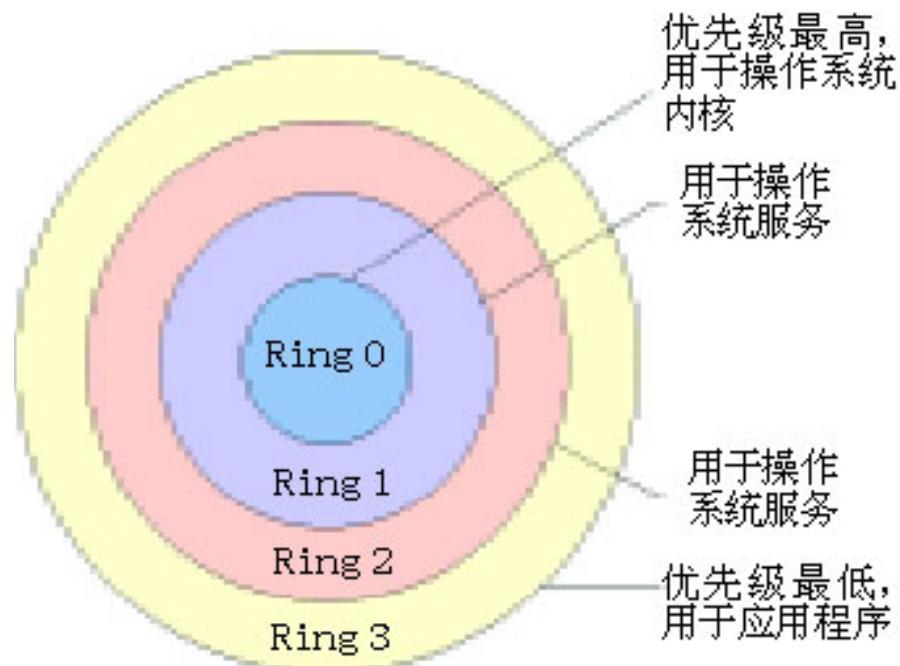
启动I/O设备、设置时钟、控制中断屏蔽位、清主存、建立存储键，
加载PSW、控制转移 内存清零 修改程序状态字、设置时钟、算术运算、
允许/禁止中断、访管指令、取数指令、停机

访管指令

- 访管指令（如UNIX 中用的 trap 指令， MS-DOS 中用的 int 指令， IBM370 中用的 supervisor 指令等）， 访管指令本身不是特权指令， 其基本功能是让程序拥有“自愿进管”的手段， 从而引起访管中断。
- 它表示运行程序对操作系统功能的调用， 所以也称系统调用， 可以看作是机器指令的一种扩充。 访管指令包括操作码和访管参数两部分， 前者表示这条指令是访管指令， 后者表示具体的访管要求。

处理器状态

- 处理器怎么知道当前是操作系统还是一般用户程序在运行呢?
 - 程序状态字寄存器
- 处理器状态
- 核心态(Ring 0)
 - 运行内核代码
 - 可执行特权指令
- 用户态→核心态
 - 系统调用
 - 中断

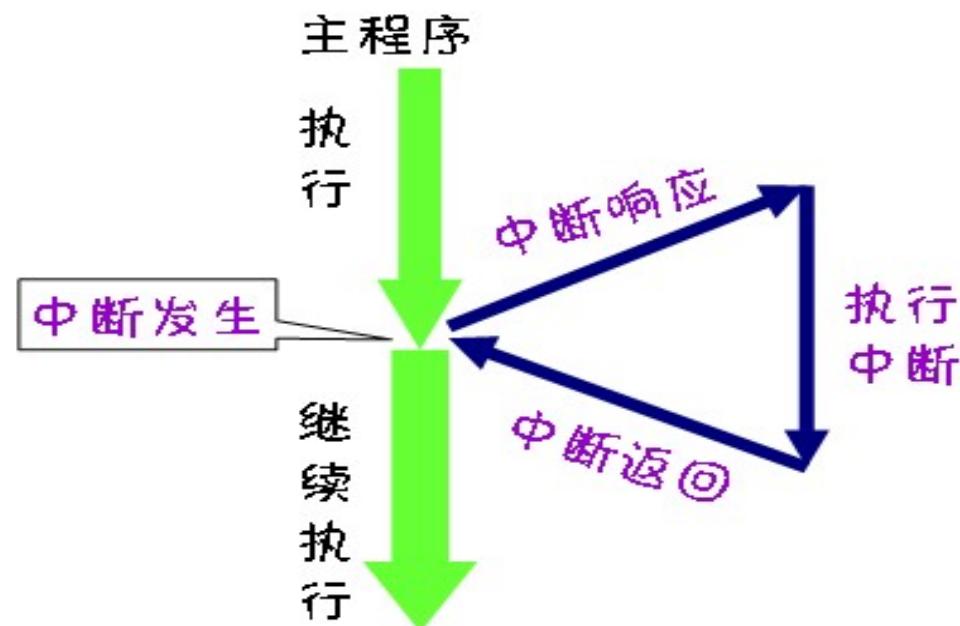


程序状态字寄存器 (PSW)

- 程序基本状态：
 - 程序计数器；
 - 条件码；
 - 处理器状态位。
- 中断码。保存程序执行时当前发生的中断事件。
- 中断屏蔽位。指明程序执行中发生中断事件时，是否响应出现的中断事件。

2.2 处理器的中断技术

- 中断是指程序执行过程中，遇到急需处理的事件时，暂时中止CPU上现行程序的运行，转去执行相应的事件处理程序，待处理完成后再返回原程序被中断处或调度其他程序执行的过程。



为什么要中断

- 请求系统服务 – 系统调用
- 实现并发工作
- 处理突发事件
- 满足实时要求
- . . .

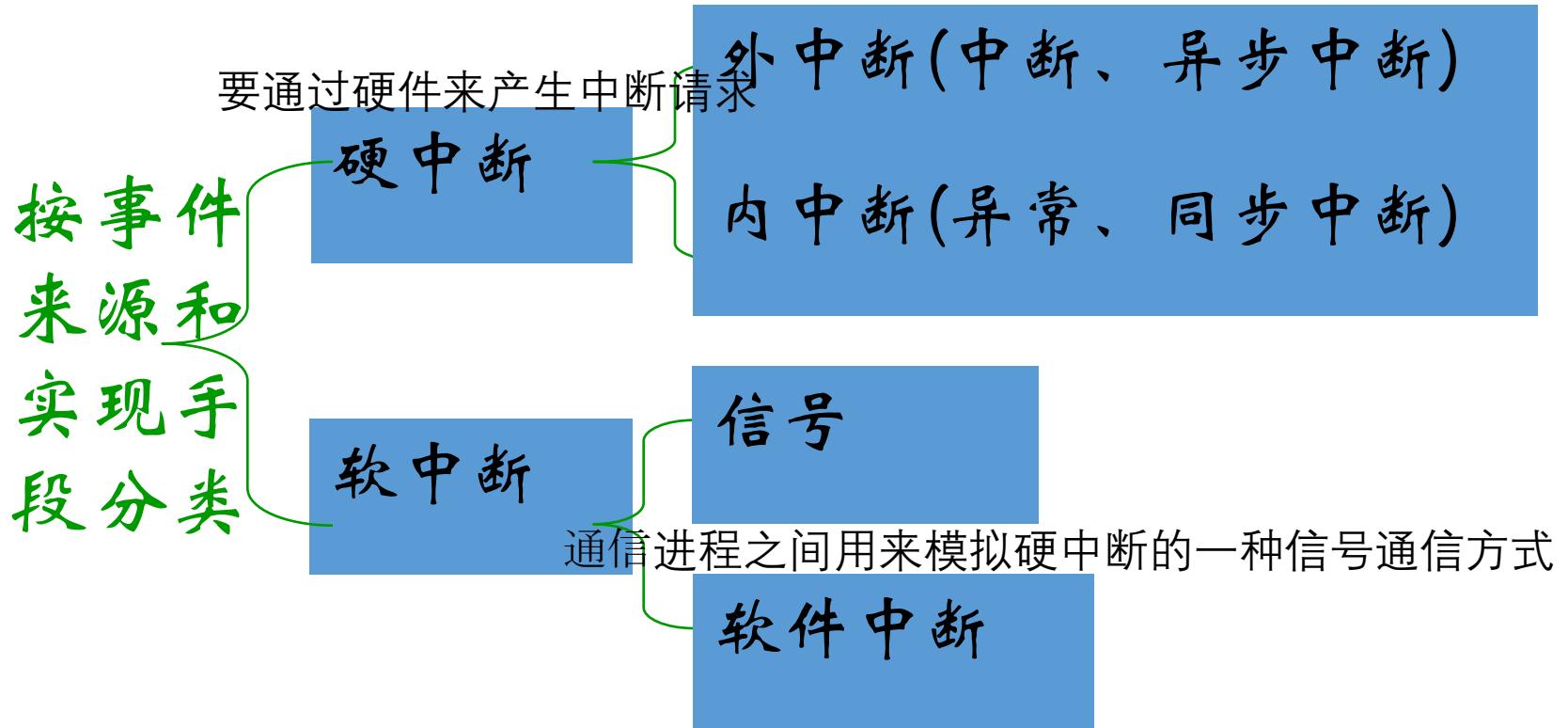
中断对于操作系统的重要性 就像汽车发动机、飞机引擎

- →→ 操作系统是由 “中断驱动” 或 “事件驱动” 的

中断分类 (1)

- 从中断事件性质分：
 - 强迫性中断
 - 机器故障中断事件：电源故障，主存储器出错
 - 程序性中断事件：定点溢出，除数为 0，地址越界等
 - 外部中断事件：时钟的定时中断，控制台发控制信息等
 - 输入输出中断事件：设备出错，传输结束
 - 自愿性中断

中断分类 (2)



中断 vs 异常

- 中断
 - 现行指令无关的中断信号触发的(异步的)
 - 指令之间才允许中断
 - 中断的发生与CPU处在用户模式或内核模式无关
 - 一般来说，中断处理程序提供的服务不是为当前进程所需的
- 异常
 - 由处理器正在执行现行指令而引起
 - 一条指令执行期间允许响应异常
 - 异常处理程序提供的服务是为当前进程所用
 - 异常包括很多方面，有出错(fault)，也有陷入(trap)等

中断与异常的小结

类别	原因	异步/同步	返回行为
中断 Interrupt	来自I/O设备、其他硬件部件	异步	总是返回到下一条指令
陷入Trap	有意识安排的	同步	返回到下一条指令
故障Fault	可恢复的错误	同步	返回到当前指令
终止Abort	不可恢复的错误	同步	不会返回

- 硬件该做什么事? —— 中断/异常响应
捕获中断源发出的中断/异常请求, 以一定方式响应, 将处理器控制权交给特定的处理程序

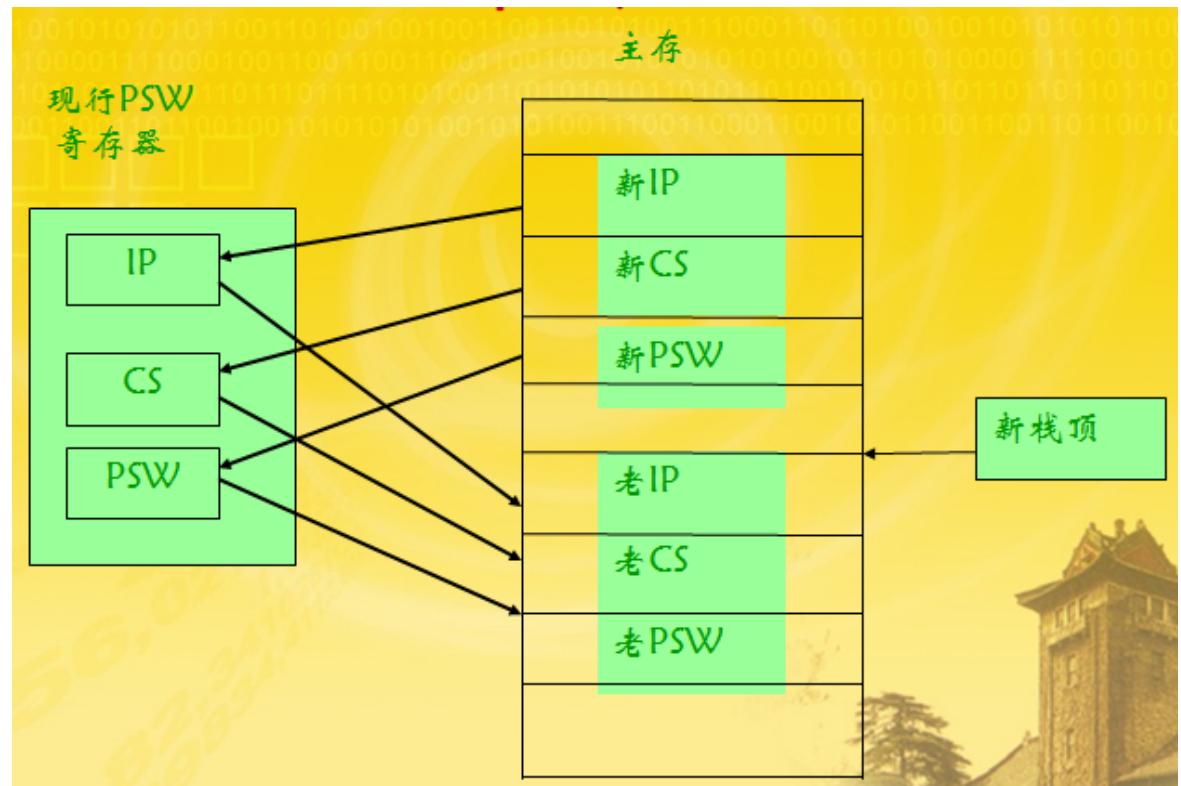
- 软件要做什么事? —— 中断/异常处理程序
识别中断/异常类型并完成相应的处理

中断的通常用途

- 外中断用于外部设备对CPU的中断(中断的是正在运行的任何程序), 转向中断处理程序上半部分执行 ;
- “异常” 因指令执行不正常而中断CPU(中断的是正在执行这条指令的程序), 转向异常处理程序 ;
- “软件中断” 用于硬中断服务程序对内核的中断, 在上半部分中发出软件中断(即标记下半部分), 使得中断下半部分在适当时刻获得处理 ;
- “信号” (软中断)用于内核或进程对某个进程的中断, 通知进程某个特定事件发生或迫使进程执行信号处理程序。

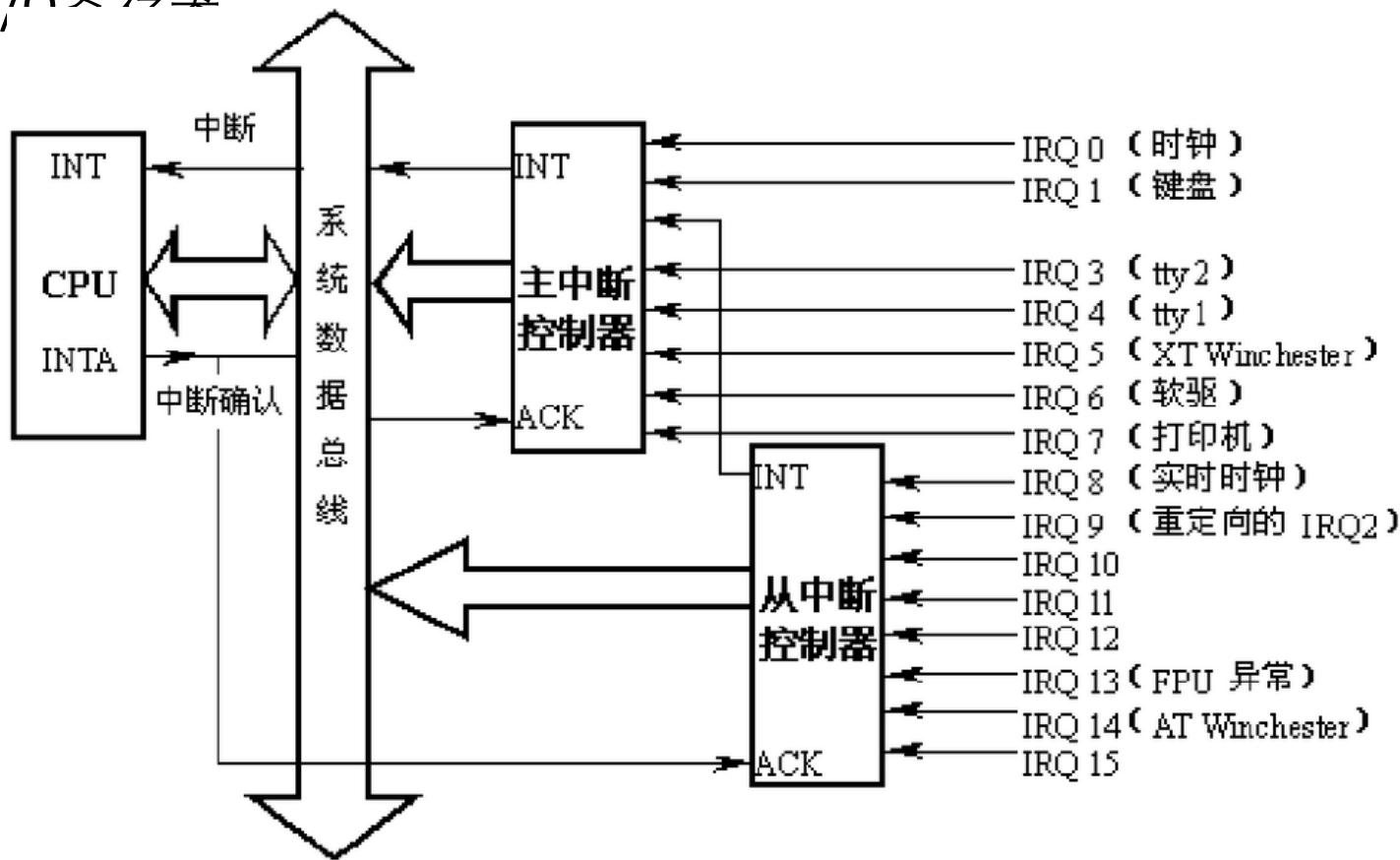
中断/异常响应要做四件事

- 发现中断源
- 保护现场
- 转向处理中断/异常事件的处理程序
- 恢复现场



中断处理流程 (1)

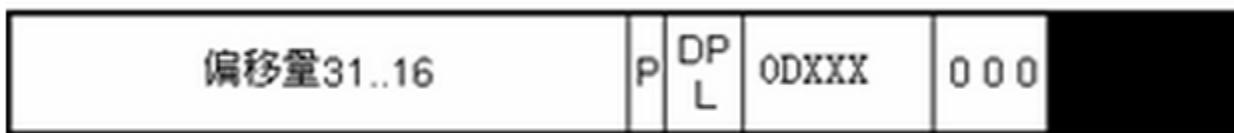
- (1) 设备发出中断请求，中断控制器将中断号转换成中断向量存储I/O空间



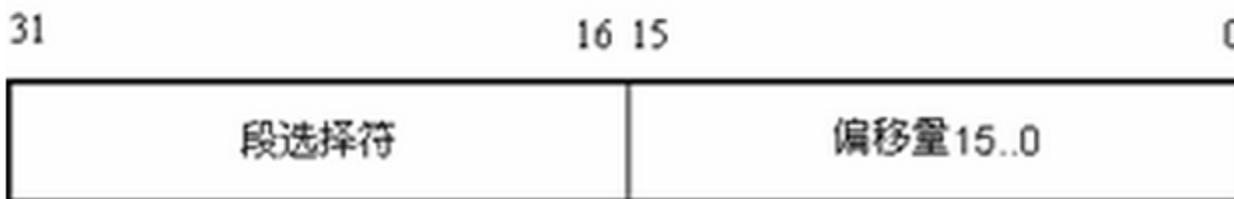
中断处理流程 (2)

- (2)

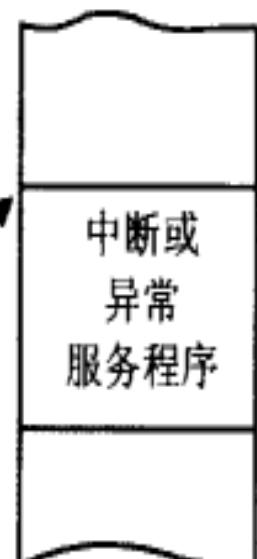
31 16 15 12 8 7 54 0 基址



47



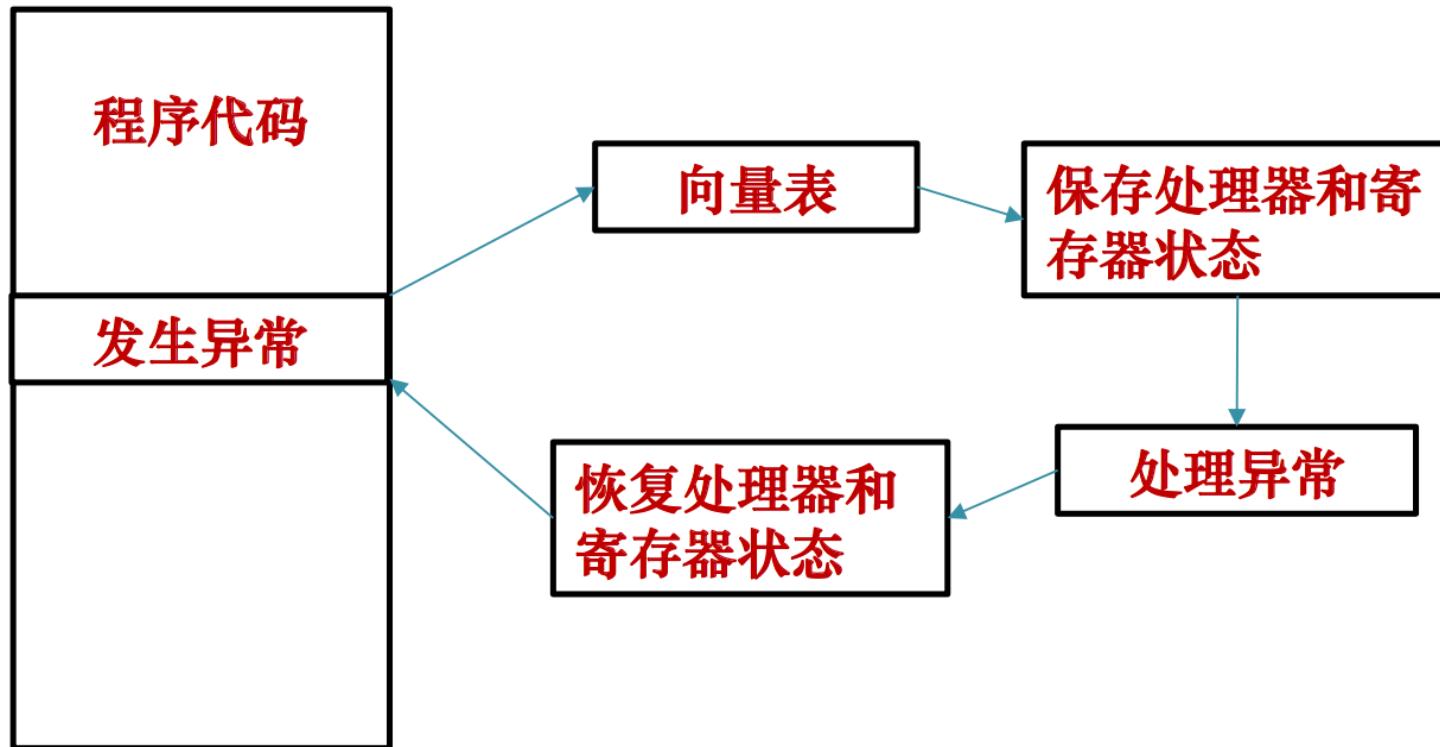
n 号异常	DPL	段描述符的特权级
n 号中断	偏移量	入口函数地址的偏移量
int n 指令	P	段是否在内存中的标志
	段选择符	入口函数所处代码段的选择符
	D	标志位, 1=32位, 0=16位
	XXX	3位门类型码
		基址 线性地址



中断处理流程 (3)

- 保存现场：将CPU寄存器的值压入核心栈
- 对中断控制器进行确认，设置中断源状态等
- 根据IRQ为发出中断请求的设备提供服务(执行上半部分、标记下半部分)
 - 服务程序队列
- 退出，恢复中断前的现场。

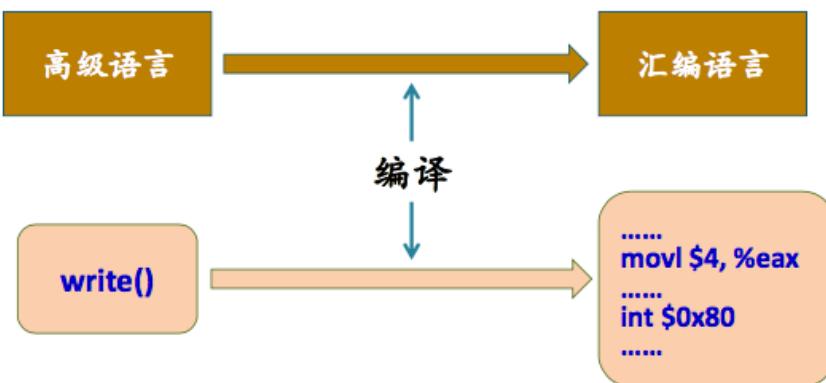
异常处理流程



异常处理流程

- 当异常产生后，自动转向异常处理程序公共入口执行，执行下列操作：
 - (1) 将硬件错误码和异常向量号存入当前进程PCB中；
 - (2) 判别异常产生于核心态还是用户态，对于前者，将简单地转向内核预定义服务程序处理，**没有被处理的核心态异常是操作系统的致命错误**；
 - (3) 对于用户态异常，终止当前进程运行，并给当前进程发信号；
 - (4) 从ret_from_exception处返回用户空间时，检查进程是否有信号等待处理，如果有则根据信号类型调用相应函数进行处理。

系统调用举例(1/3)



系统调用举例(3/3)

```
1. .section .data
2. output:
3.   .ascii "Hello!\n"
4. output_end:
5.   .equ len, output_end - output

6. .section .text
7. .globl _start
8. _start:
9.   movl $4, %eax
10.  movl $1, %ebx
11.  movl $output, %ecx
12.  movl $len, %edx
13.  int $0x80
14. end:
15.   movl $1, %eax
16.   movl $0, %ebx
17.   int $0x80
```

汇编语
言视角

eax存放系统调用号

引发一次系统调用

!这个系统调用的作用?

系统调用举例(2/3)

```
#include <unistd.h>
int main(){
    char string[7] = {'H', 'e', 'l', 'l', 'o', '!', '\n'};
    write(1, string, 7);
    return 0;
}
```

输出结果: Hello!

高级语
言视角

系统调用的执行过程

当CPU执行到特殊的陷入指令时:

- **中断/异常机制**: 硬件保护现场; 通过查中断向量表把控制权转给系统调用总入口程序
- **系统调用总入口程序**: 保存现场; 将参数保存在内核堆栈里; 通过查系统调用表把控制权转给相应的系统调用处理例程或内核函数
- **执行系统调用例程**
- **恢复现场, 返回用户程序**

中断优先级和多重中断

- 中断优先级：
 - 按中断的紧迫程度分类
 - 以不发生中断丢失为前提，优先响应优先级高的中断
 - 高优先级中断可以中断低优先级中断的处理程序，反之不可以
 - 处理高优先级中断时，屏蔽低优先级中断
- Intel x86体系：
 - 256个中断

2.3 进程及其实现

- 进程的定义和性质
- 进程的状态和转换
- 进程的描述和组成
- 进程切换与模式切换
- 进程的控制和管理

进程的定义和性质

- 进程是可并发执行的程序在某个数据集合上的一次计算活动，也是操作系统进行资源分配和保护的基本单位。
- 进程是一个既能用来共享资源，又能描述程序并发执行过程的一个基本单位。

为什么引入进程？

- 原因1-需要刻画系统的动态性
 - “程序”自身只是计算任务的指令和数据的描述，是静态概念，无法刻画并发特性
 - 系统需要寻找一个能描述程序动态执行过程的概念，这就是进程
- 原因2-它能解决系统的“共享性”，正确描述程序的执行状态。
 - “可再入”程序

进程属性

- 结构性

数据块、代码块、控制块

- 共享性

- 动态性

- 独立性

资源分配、保护和调度的基本单位

- 制约性

- 并发性

进程与程序的区别

- 进程是动态的， 程序是静态的
- 进程是暂时的， 程序是永久的
- 进程：代码+数据+状态，
程序：代码+数据， 没有状态
- 进程和程序：多对一

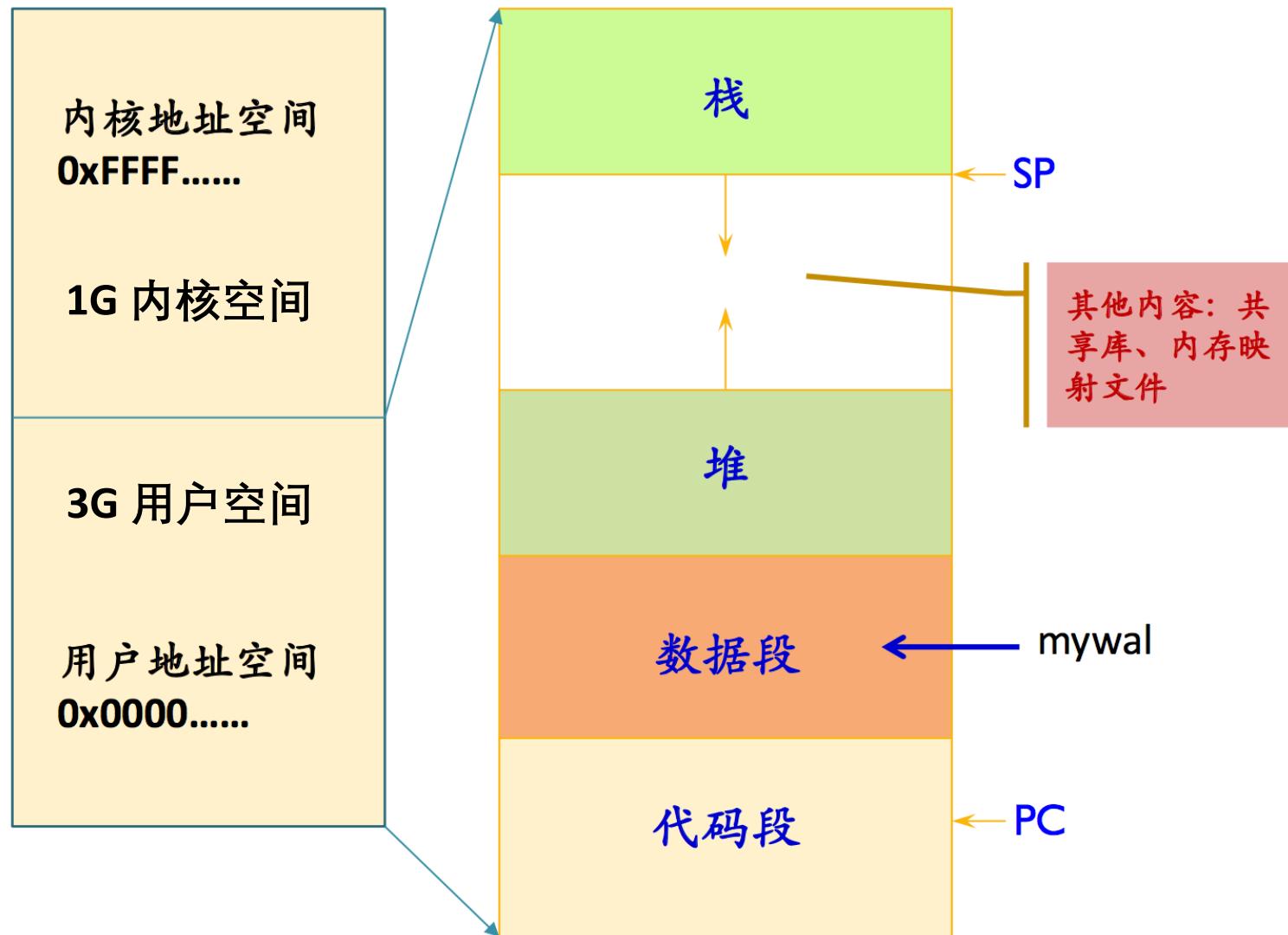
```
#include<stdio.h>
int myval;

int main(int argc, char *argv[]) {
    myval = atoi(argv[1]);
    while (1)
        printf("myval is %d, loc 0x%lx\n", myval, (long) &myval);
}
```

```
myval is 10, loc 0x109ca6020
```

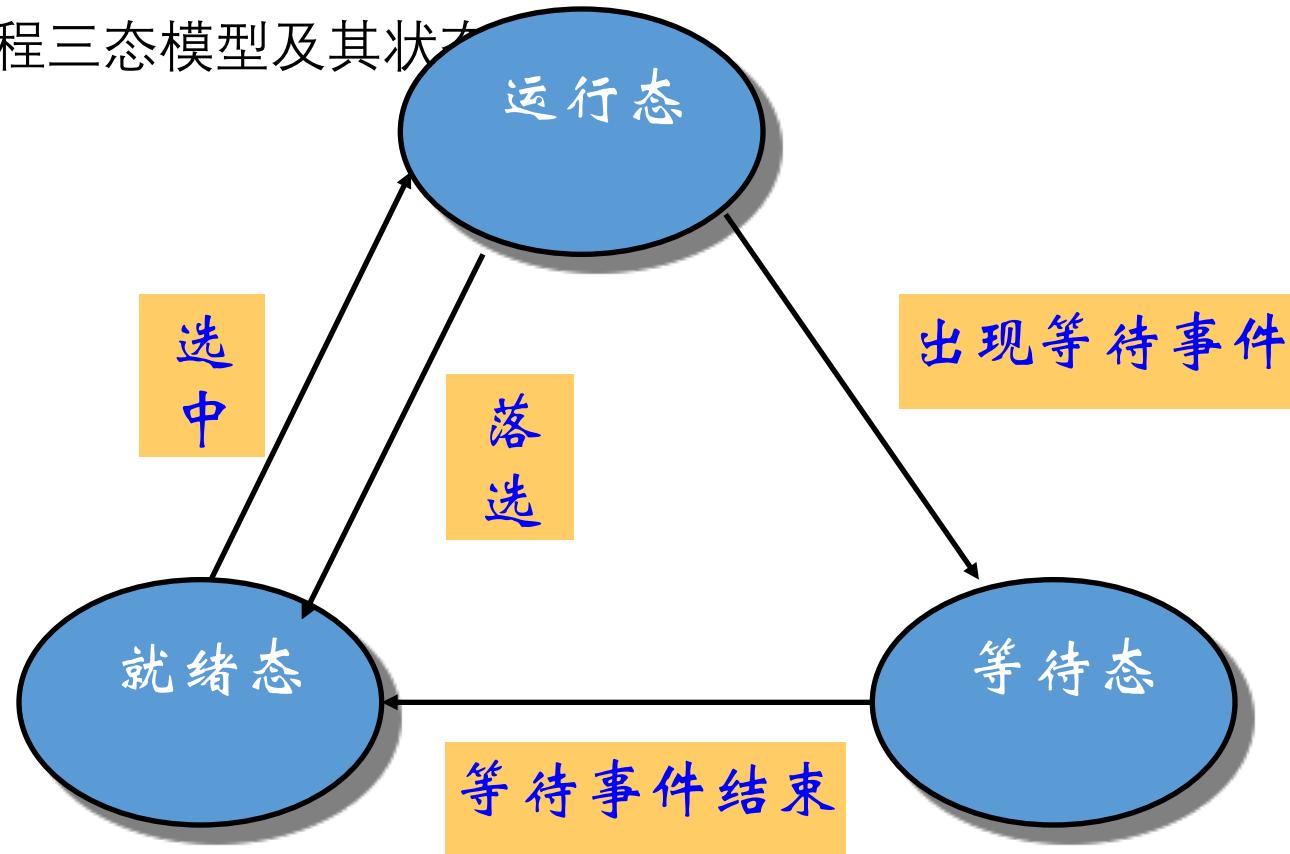
```
myval is 9, loc 0x10e9a4020
```

操作系统给每个进程都分配了一个地址空间

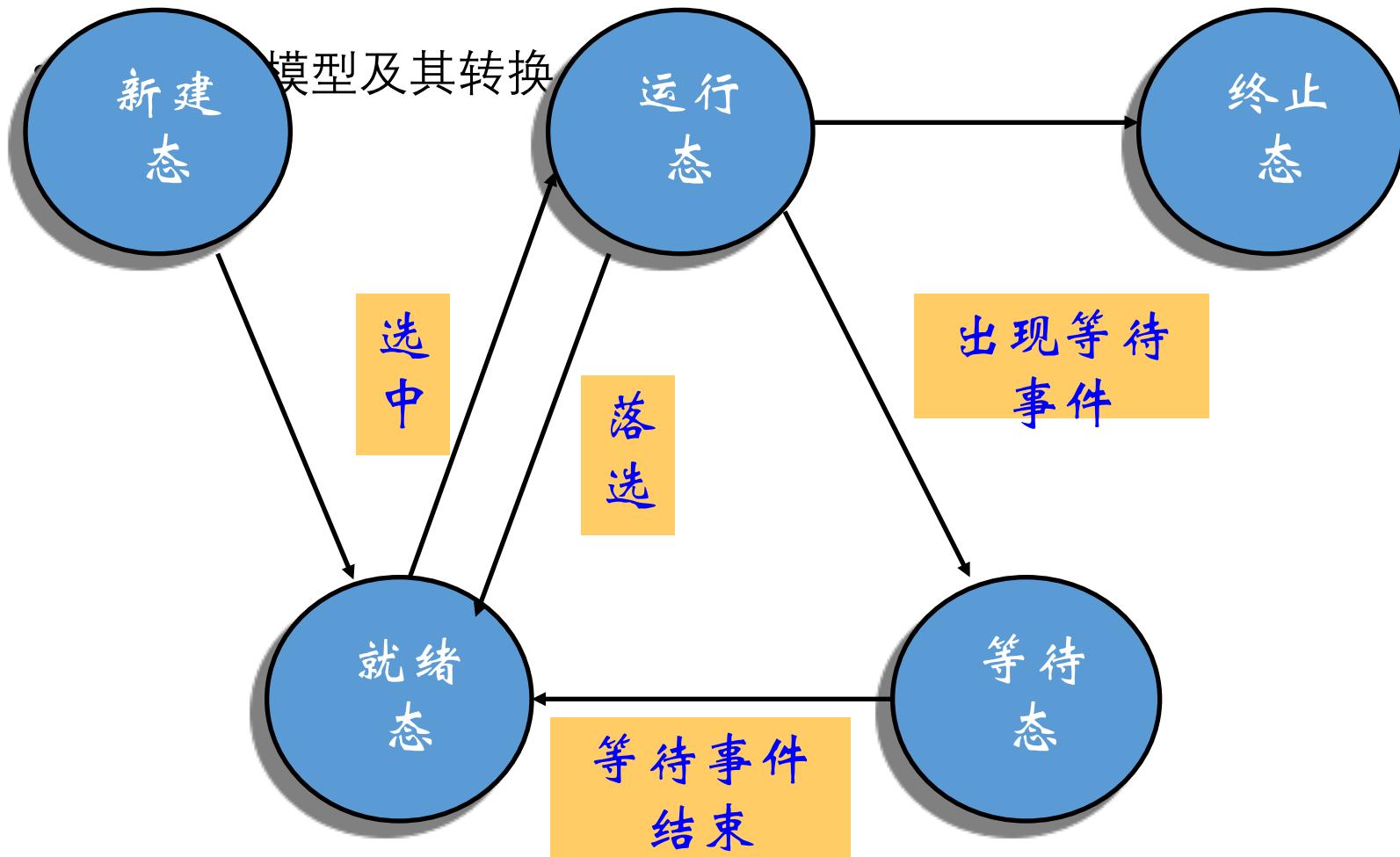


进程的状态和转换 (1)

- 进程三态模型及其状态



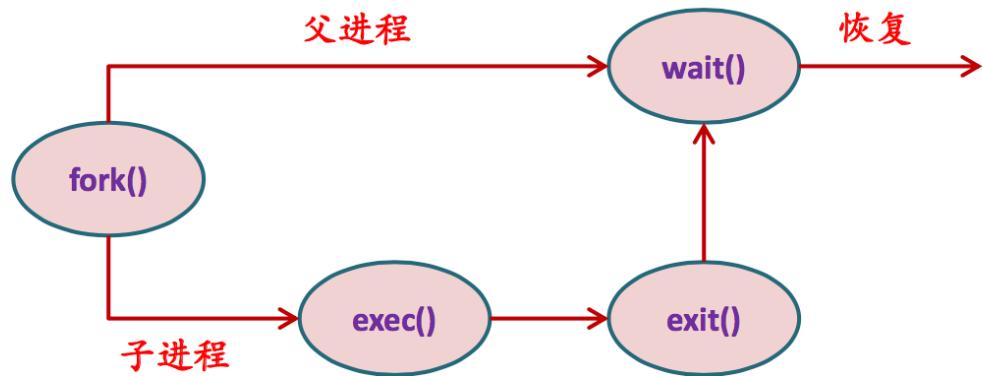
进程的状态和转换 (2)



使用fork()的示例代码

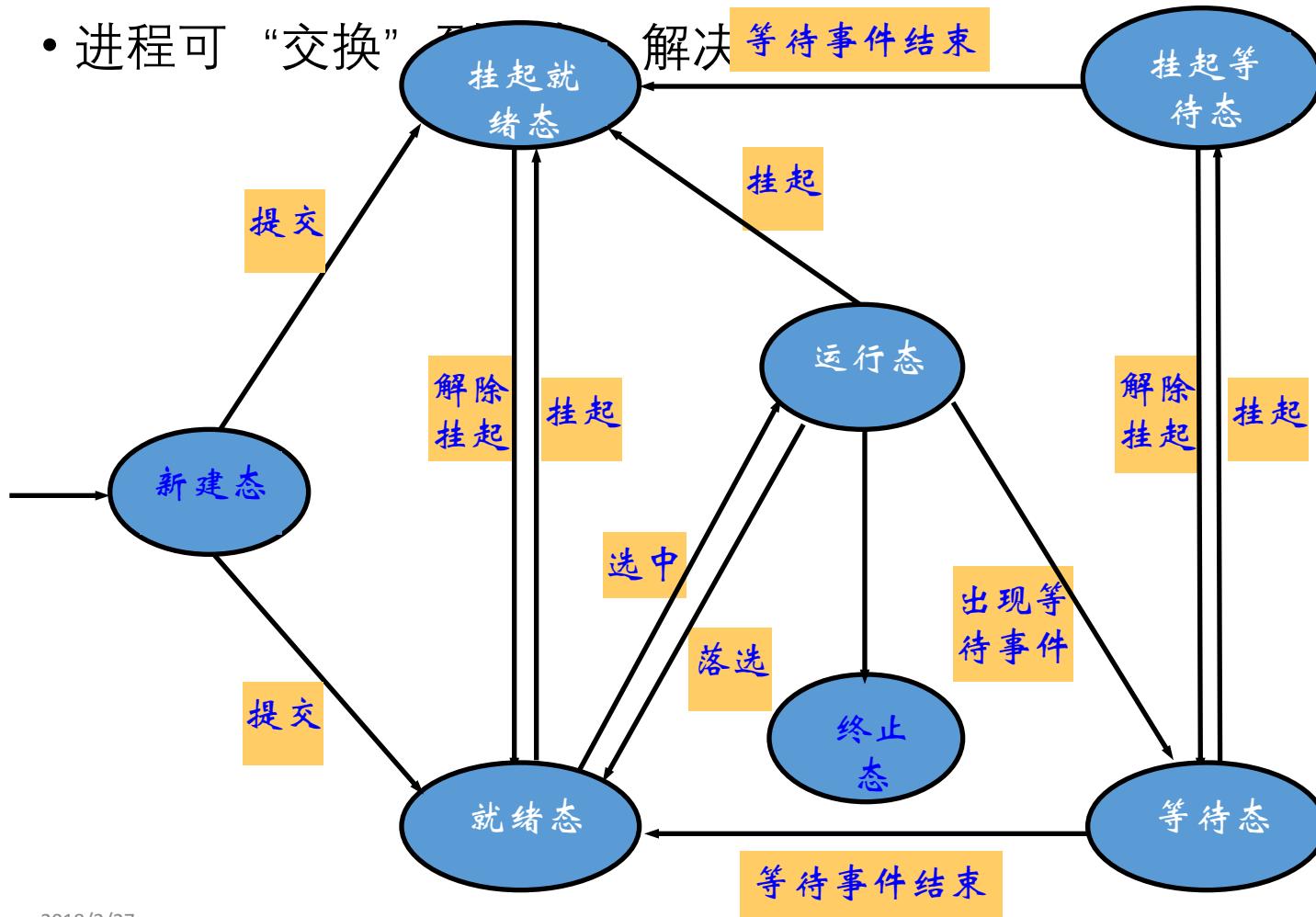
```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
void main(int argc, char *argv[])
{
pid_t pid;

pid = fork(); /* 创建一个子进程 */
if (pid < 0) { /* 出错 */
    fprintf(stderr, "fork failed");
    exit(-1);
} else if (pid == 0) { /* 子进程 */
    execvp("/bin/ls", "ls", NULL); }
else { /* 父进程 */
    wait(NULL); /* 父进程等待子进程结束 */
    printf("child complete");
    exit(0);
}
}
```



进程的挂起

- 进程可“交换”



ps ps -A 显示所有进程信息

ps -u root 显示指定用户信息

ps -ef 显示所有进程信息，连同命令行

ps -ef|grep ssh ps 与grep组合，查找特定进程.

ps aux 列出目前所有的正在内存当中的程序

ps aux |more 可以用|管道和more连接起来分页查看

```

fyw:~ yanwei$ ps aux |head
USER          PID %CPU %MEM      VSZ      RSS   TT  STAT STARTED      TIME COMMAND
root          286  4.2  0.1 2515820  8588   ??  Ss   11:12PM  3:43.45 sysmond
yanwei       10943  2.5  0.8 4602992 130328   ??  Ss   5:06PM  1:31.77 /System/Library/Frameworks/WebKit.framework/Versions/A/
s/com.apple.WebKit.Plugin.64.xpc/Contents/MacOS/com.apple.WebKit.Plugin.64
root          309  1.6  0.0 669612   1800   ??  S   11:12PM  11:27.51 /opt/cisco/anyconnect/bin/acnvmagent -console
yanwei       11921  1.0  0.8 2868800 129904   ??  R    8:16PM  0:31.66 /Applications/Utilities/Terminal.app/Contents/MacOS/Tel
_windowserver 198  0.8  1.3 7259864 212064   ??  Ss   11:12PM  26:26.64 /System/Library/PrivateFrameworks/SkyLight.framework/Re
ndowServer -daemon
yanwei        680  0.4  0.4 2691896  59964   ??  S   11:14PM  5:45.96 /Applications/DrCleaner.app/Contents/MacOS/DrCleaner
yanwei       11988  0.4  1.3 4244532 224268   ??  Ss   8:18PM  0:06.90 /System/Library/Frameworks/WebKit.framework/Versions/A/
s/com.apple.WebKit.WebContent.xpc/Contents/MacOS/com.apple.WebKit.WebContent
yanwei        731  0.3  0.2 2705960  37680   ??  S   11:14PM  0:32.17 /Applications/Utilities/Activity Monitor.app/Contents/\
ity Monitor
yanwei       2071  0.1  2.3 3815608 387548   ??  S   12:20AM  5:40.74 /Applications/WeChat.app/Contents/MacOS/WeChat

```

VSZ 进程所使用的虚存的大小 (Virtual Size)

RSS 进程使用的驻留集大小或者是实际内存的大小, Kbytes字节。

TTY 与进程关联的终端 (tty)

STAT 进程的状态 : 进程状态使用字符表示的 (STAT的状态码)

R 运行 **Runnable (on run queue)** 正在运行或在运行队列中等待。

S 睡眠 **Sleeping** 休眠中, 受阻, 在等待某个条件的形成或接受到信号。

I 空闲 **Idle**

Z 僵死 **Zombie (a defunct process)** 进程已终止, 但进程描述符存在, 直到父进程调用**wait4()**系统调用后释放。

D 不可中断 **Uninterruptible sleep (usually IO)** 收到信号不唤醒和不可运行, 进程必须等待直到有中断发生。

T 终止 **Terminate** 进程收到**SIGSTOP, SIGSTP, SIGTIN, SIGTOU**信号后停止运行运行。

P 等待交换页

W 无驻留页 **has no resident pages** 没有足够的记忆体分页可分配。

挂起进程性质

- 无法立即执行
- 挂起进程可能等待事件，但挂起条件**独立于**所等待事件
- 进程进入挂起状态是由于操作系统、父进程或进程本身阻止它的运行。
- 结束进程挂起状态的命令只能通过操作系统或父进程发出。

进程的描述和组成

- 进程映像：
 - 代码块
 - 数据块
 - 进程核心栈
 - 进程控制块：存储进程的标识信息、现场信息和控制信息等
- 用户线程
- 内核线程

进程上下文

- 进程上下文(context)：
 - 物理实体
 - 支持进程运行的环境：硬件寄存器、PSW、页表等
- 进程切换 → 上下文切换
- 进程上下文的三部分
 - 用户级上下文：代码、数据（用户级堆栈）、共享存储区等
 - 寄存器上下文
 - 系统级上下文：进程控制块、主存管理信息、核心栈等

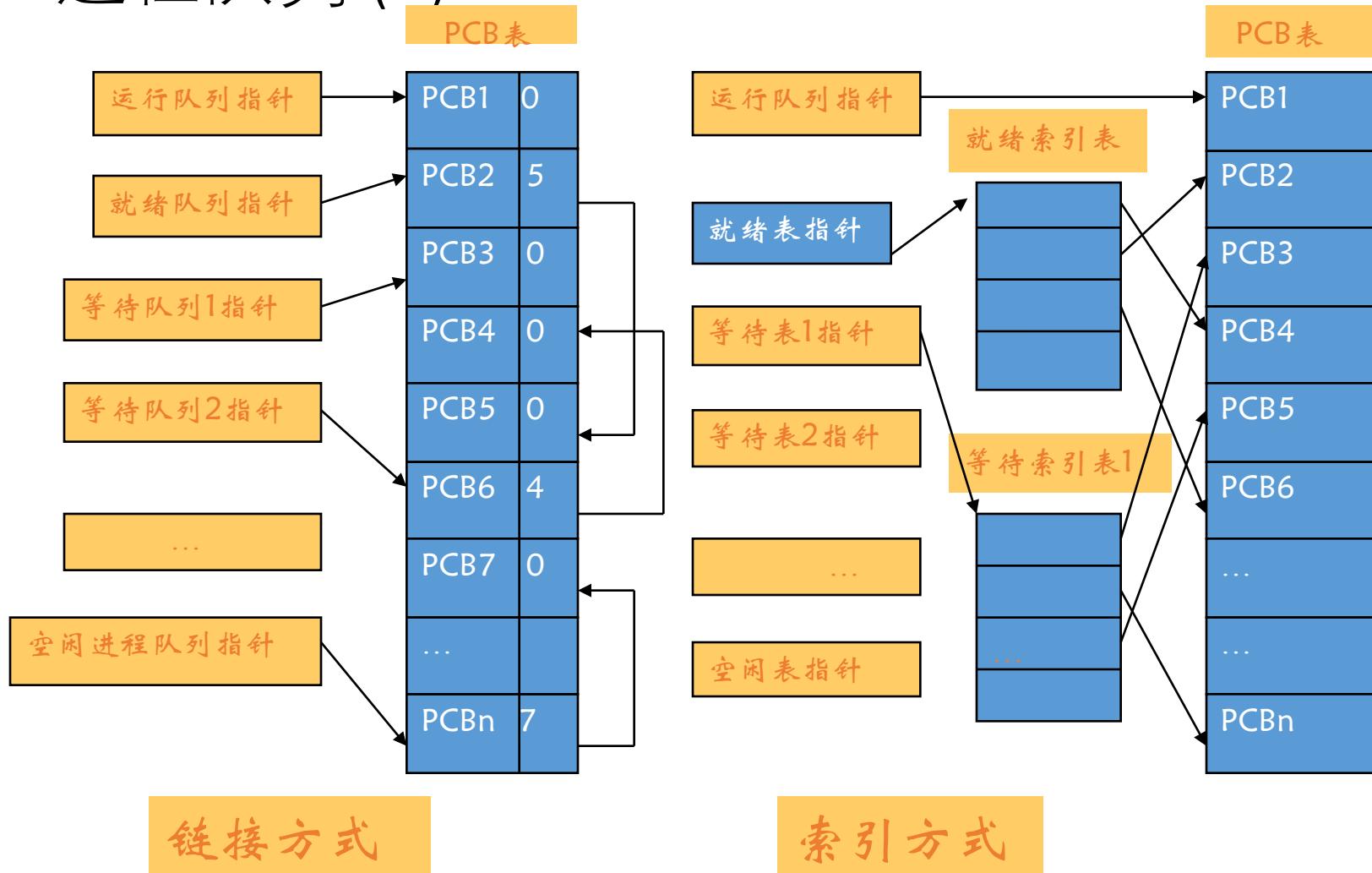
进程控制块 (PCB)

- 维护进程运行状态的数据结构
- 组成：
 - 标识信息：进程ID、进程组ID、用户进程名、用户组名等
 - 现场信息：通用寄存器内容、控制寄存器内容、堆栈指针等
 - 控制信息：进程状态、等待事件、进程优先级等

进程队列 (1)

- 进程队列：将处于同一状态的进程PCB组织成一个队列
- 同一状态进程的PCB既可按先来先到的原则排成队列;也可按优先数或其它原则排成队列。
- 通用队列组织方式：
 - 线性方式
 - 链接方式
 - 索引方式
 - 混合使用

进程队列 (2)



Linux 进程链表

- Linux的PCB : task_struct
 - 双向循环列表
 - 可运行队列链表
 - 散列链表
 - 等待队列链表

linux/task_struct+1)

```

struct task_struct {
    volatile long state; /* -1 unrunnable, 0 runnable, >0 stopped */
    struct thread_info *thread_info;
    atomic_t usage;
    unsigned long flags; /* per process flags, defined below */
    unsigned long ptrace;

    int lock_depth; /* Lock depth */

    int prio, static_prio;
    struct list_head run_list;
    prio_array_t *array;

    unsigned long sleep_avg;
    long interactive_credit;
    unsigned long long timestamp;
    int activated;

    unsigned long policy;
    cpumask_t cpus_allowed;
    unsigned int time_slice, first_time_slice;

    struct list_head tasks;
    /*
     * ptrace_list/ptrace_children forms the list of my children
     * that were stolen by a tracer.
     */
    struct list_head ptrace_children;
    struct list_head ptrace_list;

    struct mm_struct *mm, *active_mm;

/* task state */
    struct linux_binfmt *binfmt;
    int exit_code, exit_signal;
    int pdeath_signal; /* The signal sent when the parent dies */
/* ??? */
    unsigned long personality;
}

```

```

    int did_exec:l;
    pid_t pid;
    pid_t tgid;
    /*
     * pointers to (original) parent process, youngest child, younger
     * sibling,
     * older sibling, respectively. (p->father can be replaced with
     * p->parent->pid)
     */
    struct task_struct *real_parent; /* real parent process (when being
debugged) */
    struct task_struct *parent; /* parent process */
    /*
     * children/sibling forms the list of my children plus the
     * tasks I'm ptracing.
     */
    struct list_head children; /* list of my children */
    struct list_head sibling; /* linkage in my parent's children list */
    struct task_struct *group_leader; /* threadgroup leader */

    /* PID/PID hash table linkage.*/
    struct pid_link pids[PIDTYPE_MAX];

    wait_queue_head_t wait_chldexit; /* for wait4() */
    struct completion *vfork_done; /* for vfork() */
    int __user *set_child_tid; /* CLONE_CHILD_SETTID */
    int __user *clear_child_tid; /* CLONE_CHILD_CLEARTID */

    unsigned long rt_priority;
    unsigned long it_real_value, it_prof_value, it_virt_value;
    unsigned long it_real_incr, it_prof_incr, it_virt_incr;
    struct timer_list real_timer;
    unsigned long utime, stime, cutime, cstime;
    unsigned long nvcsw, nivcsw, cnvcsw, cnivcsw; /* context switch
counts */
    u64 start_time;
}

```

```

/* mm fault and swap info: this can arguably be seen as either mm-
specific or thread-specific */
    unsigned long min_flt, maj_flt, cmin_flt, cmaj_flt;
/* process credentials */
    uid_t uid,euid,suid,fsuid;
    gid_t gid,egid,sgid,fsgid;
    struct group_info *group_info;
    kernel_cap_t cap_effective, cap_inheritable, cap_permitted;
    int keep_capabilities;l;
    struct user_struct *user;
/* limits */
    struct rlimit rlim[RLIM_NLIMITS];
    unsigned short used_math;
    char comm[16];
/* file system info */
    int link_count, total_link_count;
/* ipc stuff */
    struct sysv_sem sysvsem;
/* CPU-specific state of this task */
    struct thread_struct thread;
/* filesystem information */
    struct fs_struct *fs;
/* open file information */
    struct files_struct *files;
/* namespace */
    struct namespace *namespace;
/* signal handlers */
    struct signal_struct *signal;
    struct sighand_struct *sighand;

    sigset_t blocked, real_blocked;
    struct sigpending pending;

    unsigned long sas_ss_sp;
    size_t sas_ss_size;
    int (*notifier)(void *priv);
    void *notifier_data;

    sigset_t *notifier_mask;

    void *security;
    struct audit_context *audit_context;

/* Thread group tracking */
    u32 parent_exec_id;
    u32 self_exec_id;
/* Protection of (de-)allocation: mm, files, fs, tty */
    spinlock_t alloc_lock;
/* Protection of proc_dentry: nesting proc_lock, dcache_lock,
   write_lock_irq(&tasklist_lock); */
    spinlock_t proc_lock;
/* context-switch lock */
    spinlock_t switch_lock;

/* journalling filesystem info */
    void *journal_info;

/* VM state */
    struct reclaim_state *reclaim_state;

    struct dentry *proc_dentry;
    struct backing_dev_info *backing_dev_info;

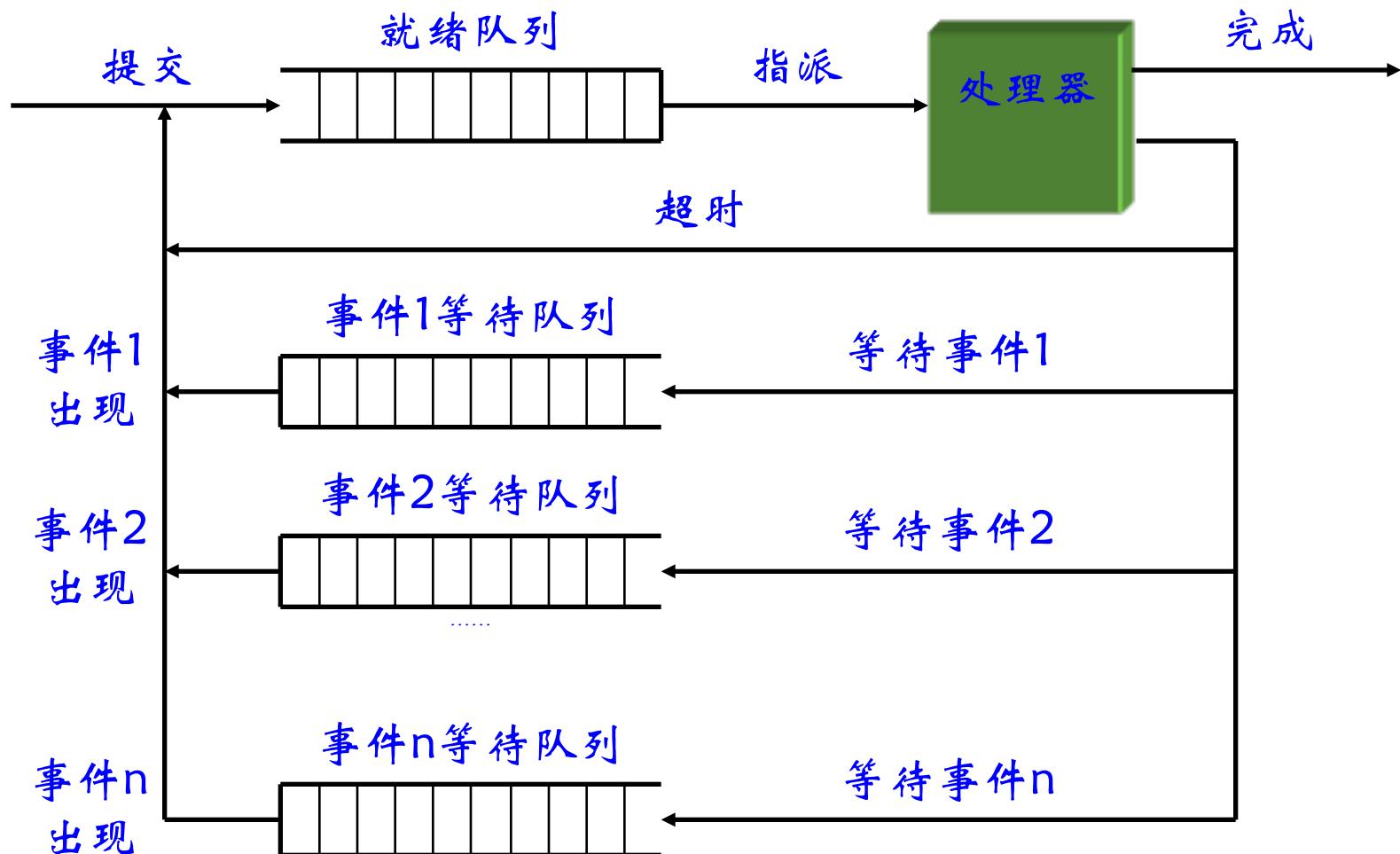
    struct io_context *io_context;

    unsigned long ptrace_message;
    siginfo_t *last_siginfo; /* For ptrace use. */

#endif CONFIG_NUMA
    struct mempolicy *mempolicy;
    short il_next; /* could be shared with used_math */
#endif
};

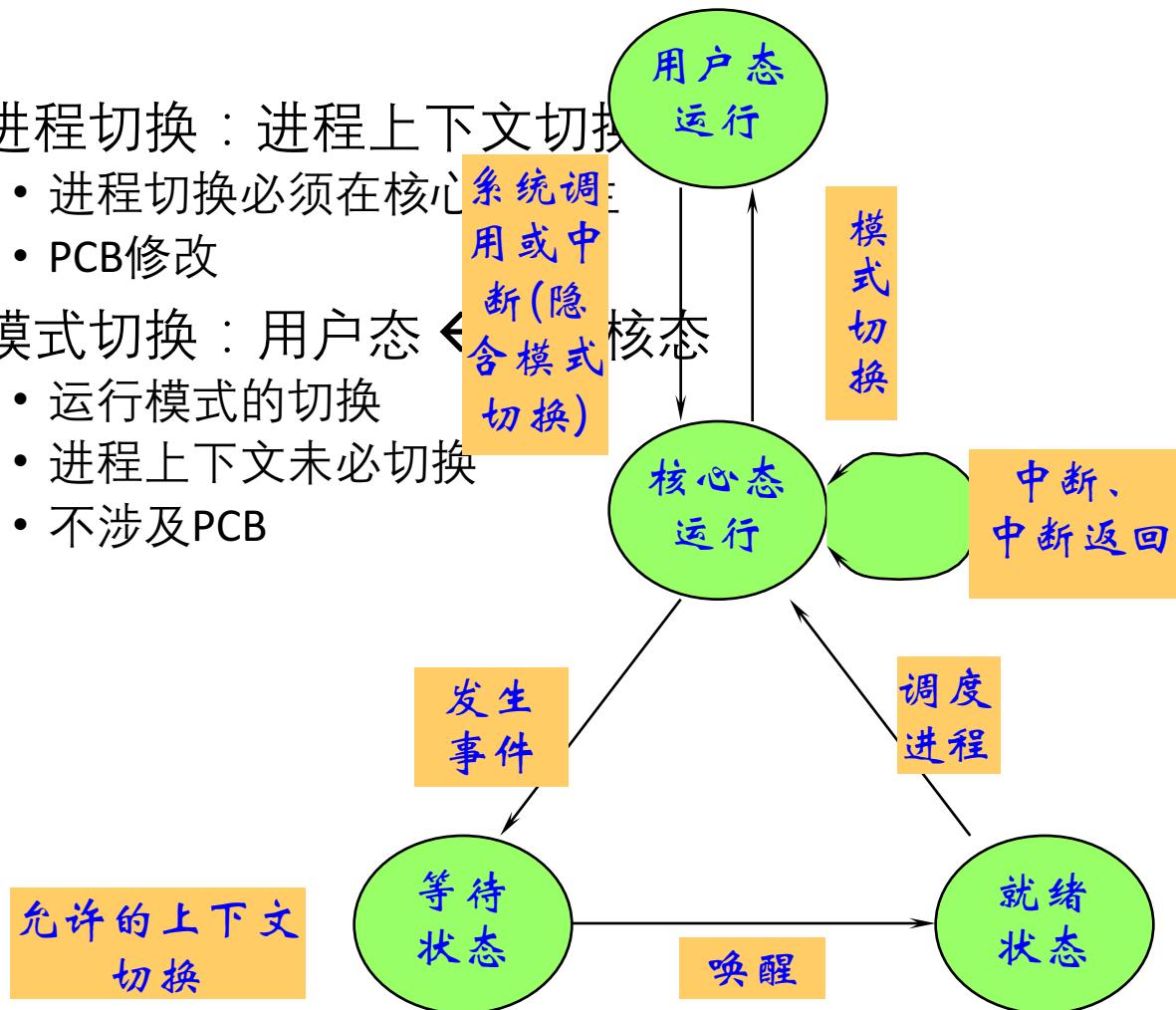

```

队列管理和状态转换



进程切换与CPU模式切换

- 进程切换：进程上下文切换
 - 进程切换必须在核心态
 - PCB修改
- 模式切换：用户态 \leftarrow 核态
 - 运行模式的切换
 - 进程上下文未必切换
 - 不涉及PCB



进程的管理和控制

- 进程管理与控制
 - 创建进程、阻塞进程、唤醒进程、挂起进程、激活进程、终止进程和撤销进程等
- 原语操作
 - 执行过程中不允许被中断

2.4 线程及其实现

2.4.1 引入多线程的动机

2.4.2 多线程环境中的进程和线程

2.4.3 线程的实现

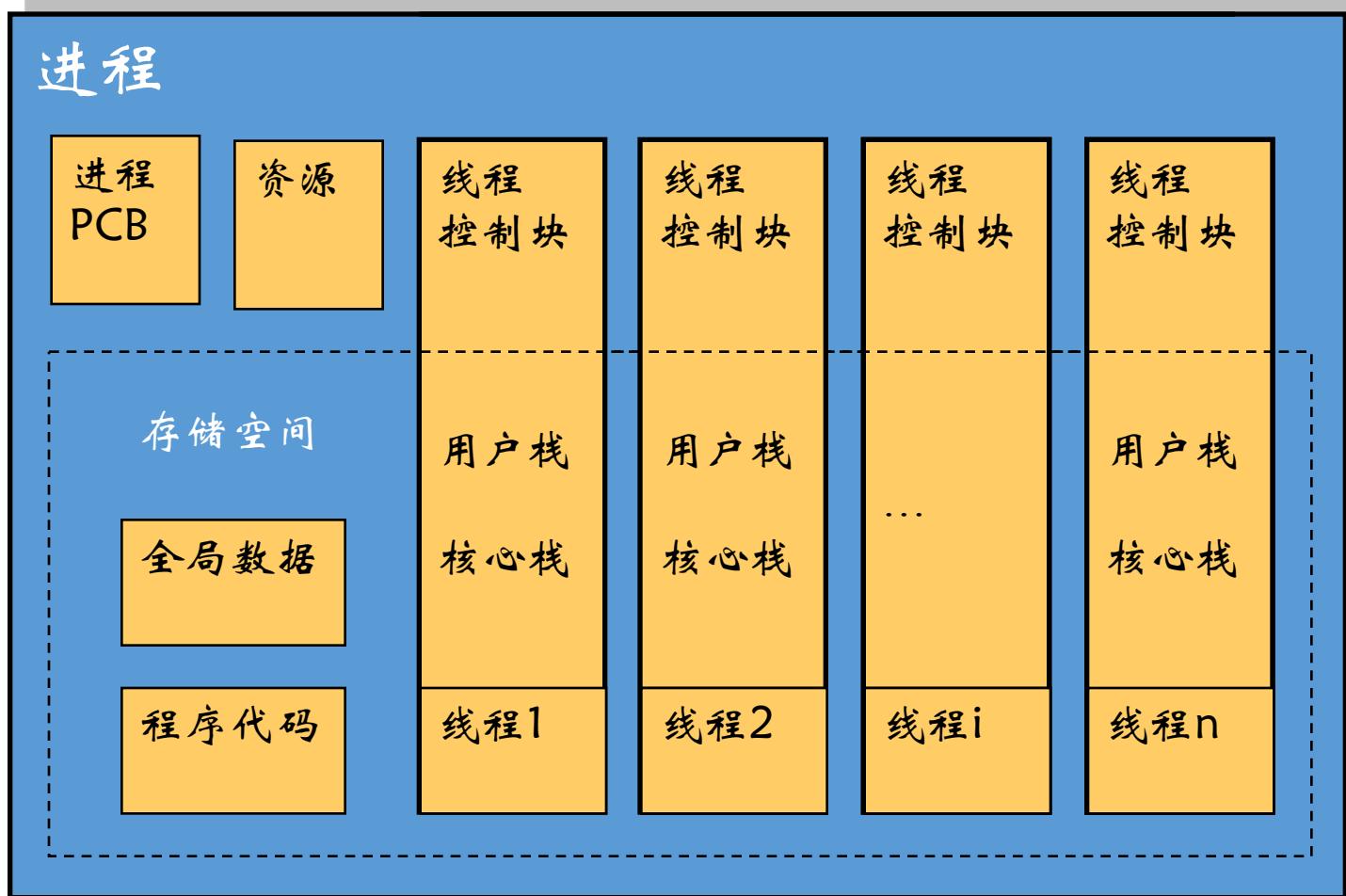
引入线程的动机 (1)

- 单线程进程的问题
 - 进程切换开销大
 - 进程通信代价大
 - 进程间的并发性粒度较粗，并发度不高
 - 不适应并行计算和分布并行计算的要求
 - 不适合客户/服务器计算的要求

引入线程的动机(2)

- 引入进程的动机
 - 使多个程序可以并发执行，以改善资源使用率和提高系统效率
- 引入线程的动机
 - 减少进程并发执行时所付出的时空开销，使得并发粒度更细、并发性更好
- 解决思路
 - 分离进程的两项功能：“独立分配资源”与“被调度分派执行”
 - 进程：资源分配单位，无需频繁切换
 - 线程：调度单位，体量小，频繁切换

多线程环境中的进程与线程



多线程环境中进程的定义

进程是操作系统中除处理器外进行的资源分配和保护的基本单位，它有一个独立的虚拟地址空间，用来容纳进程映像(如与进程关联的程序与数据)，并以进程为单位对各种资源实施保护，如受保护地访问处理器、文件、外部设备及其他进程(进程间通信)。

多线程环境中进程的定义

线程是操作系统进程中能够独立执行的实体（控制流），是处理器调度和分派的基本单位。

线程是进程的组成部分，每个进程中允许包含多个并发执行的实体（控制流），这就是多线程。

线程组成与状态

- 组成
 - 线程惟一标识符及线程状态信息；
 - 未运行时保存的线程上下文
 - 核心栈：核心态下工作时，保存参数，函数调用时的返回地址等；
 - 私有存储区：用于存放线程局部变量及用户栈。
- 状态
 - 运行、就绪和阻塞
 - 线程没有挂起状态

多线程程序设计优点

- 减少（系统）管理开销
- （线程）通信易于实现
 - 无需通过内核
- 并行程度提高
 - 一个进程多个线程可以同时运行
- 节省主存空间
 - 多个线程共享进程空间

线程的实现

- 用户级线程
 - 线程的管理由用户程序完成
 - 优点：切换更快，无需操作系统支持
 - 一个堵塞，整个进程堵塞；无法利用SMP
- 内核级线程
 - 线程的管理工作由内核完成
 - 优点：避免了用户级线程的缺点
 - 缺点：线程切换开销大
- 混合线程

2.7 处理器调度

2.7.1 处理机调度的层次

2.7.2 选择调度算法的原则

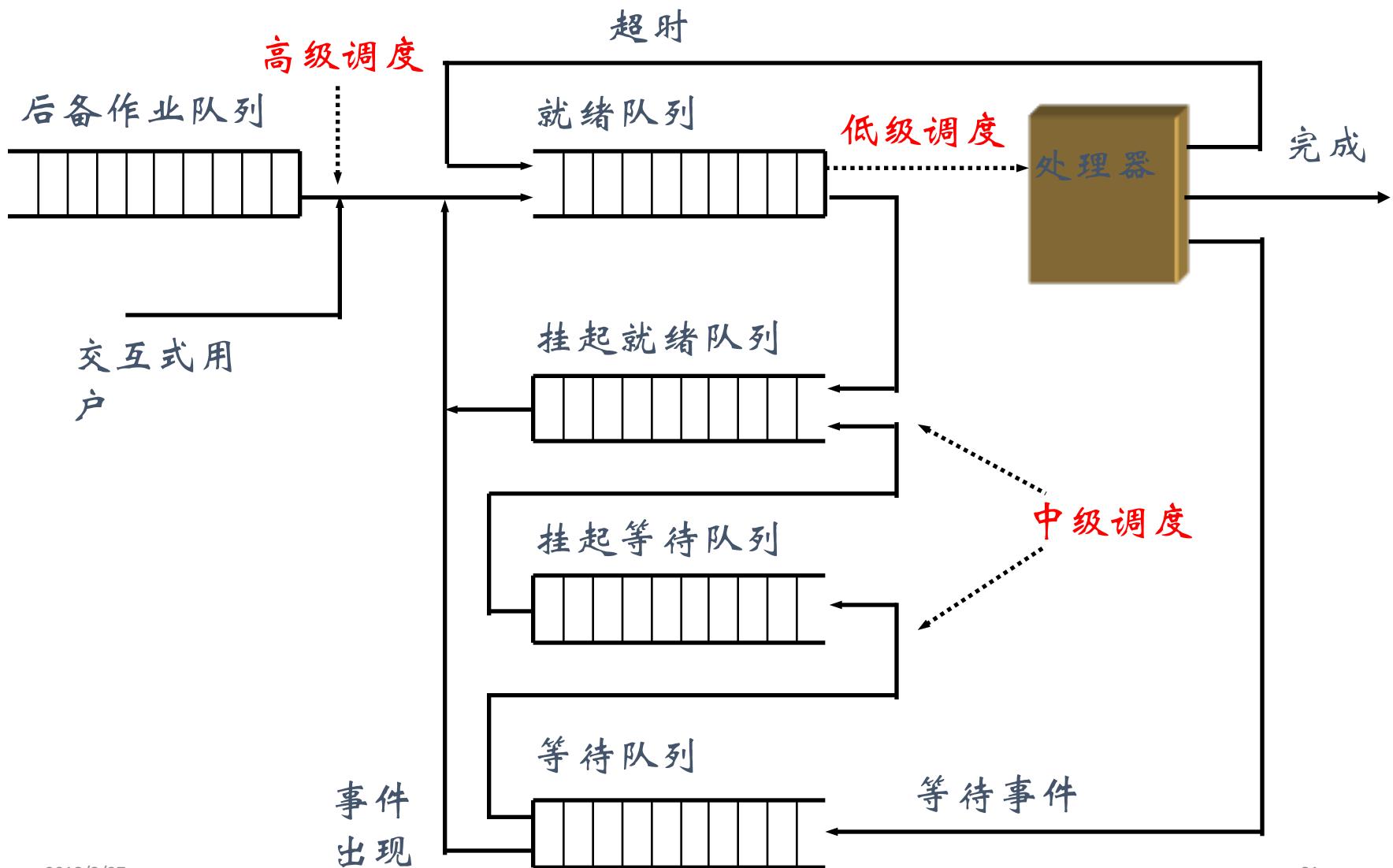
2.7.3 作业和进程的关系

2.7.4 作业的管理与调度

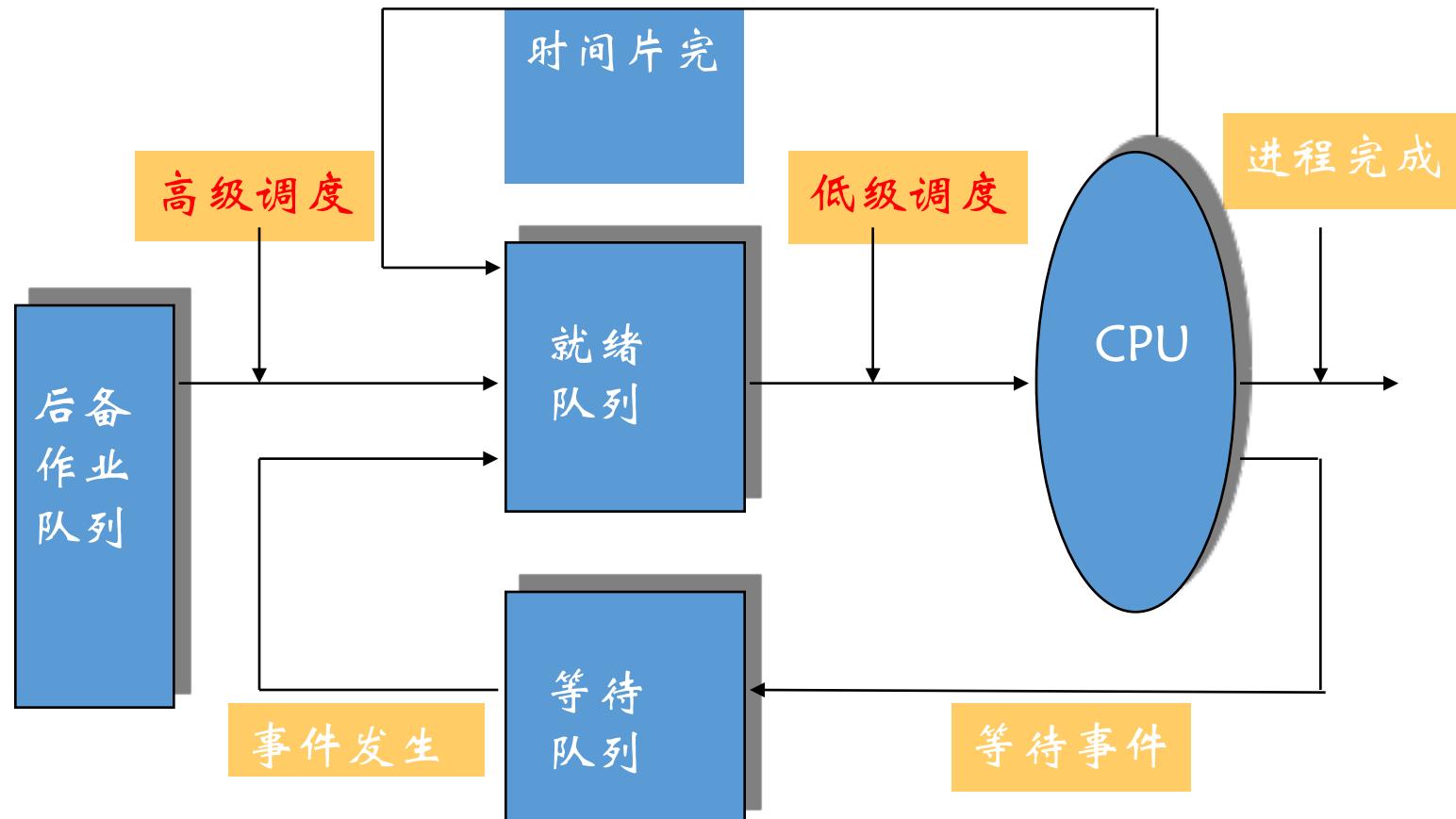
处理其调度的层次

- 高级调度
 - 多道批处理系统，选择作业进入主存
 - 分时系统通常不需要
- 中级调度
 - 外存与内存中的进程对换
 - 提高主存的利用率
- 低级调度
 - 就绪→运行
 - 操作系统必备

处理器的三级调度模型



处理器两级调度模型



选择调度算法的原则 (1)

- 资源(CPU)利用率
 - CPU利用率=CPU有效工作时间/CPU总的运行时间
 - CPU总的运行时间=CPU有效工作时间+CPU空闲等待时间
- 吞吐率
 - 单位时间内CPU处理作业的数目
- 公平性
 - 避免饥饿

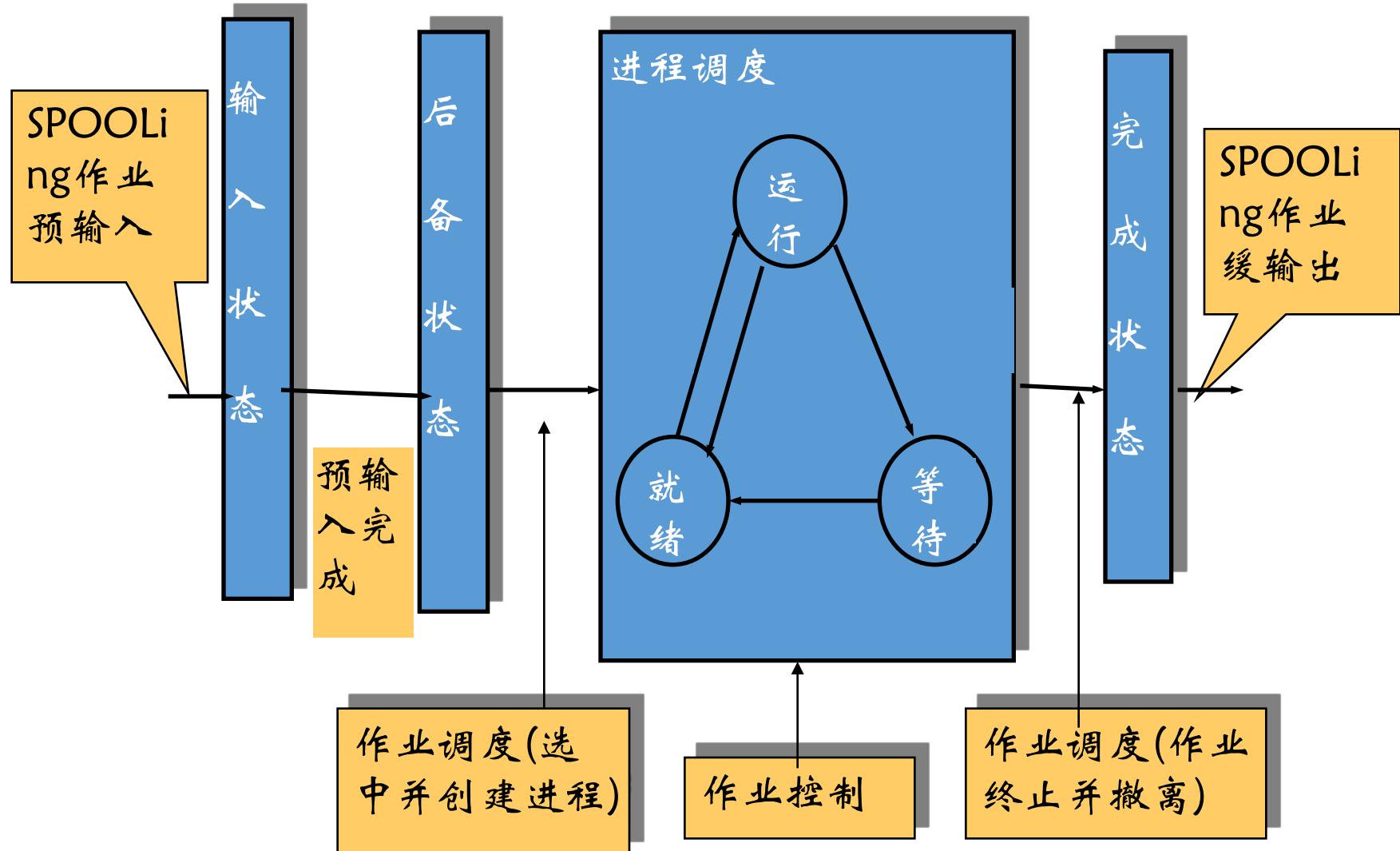
选择调度算法的原则 (2)

- 响应时间
 - 交互式进程从提交一个请求(命令)到接收到响应之间的时间间隔称响应时间
 - 实时系统、分时系统的重要指标
- 周转时间
 - 批处理用户从作业提交给系统开始，到作业完成为止的时间间隔称作业周转时间
 - 批处理系统的重要指标

平均作业周转时间

- 作业周转时间
 - 如果作业i提交给系统的时刻是 t_s , 完成时刻是 t_f , 该作业的周转时间 t_i 为：
$$t_i = t_f - t_s$$
- 平均作业周转时间 : $T = (\sum t_i) / n$
- 如果作业i的周转时间为 t_i , 所需运行时间为 t_k , 则称 $w_i = t_i / t_k$ 为该作业的带权周转时间
- 平均作业带权周转时间 $W = (\sum w_i) / n$

进程调度 vs 作业调度



处理器调度算法

- 2.8.1 低级调度的功能和类型
- 2.8.2 作业调度和低级调度算法
- 2.8.3 实时调度算法
- 2.8.4 多处理机调度算法

低级调度的功能

- 调度程序两项任务：调度、分派
- 调度：实现调度策略
 - 组织和维护就绪进程队列
 - 包括确定调度算法、按调度算法组织和维护就绪进程队列
- 分派：实现调度机制
 - 当处理机空闲时，从就绪队列队首中移一个PCB，并将该进程投入运行
 - 负责上下文切换等

低级调度的类型

- 剥夺式（抢占式）
 - 高优先级进程/线程可剥夺低优先级进程/线
 - 当运行进程/线程时间片用完后被剥夺
 - 用户程序
- 非剥夺式
 - 一直运行，直到完成或自我放弃
 - 内核关键程序

常用的调度算法

- 先来先服务(FCFS-First Come First Serve)
- 最短作业优先(SJF-Shortest Job First)
- 最短剩余时间优先(SRT-Shortest Remaining Time Next)
- 最高响应比优先(HRRN-Highest Response Ratio Next)

作业调度和低级调度算法 (1)

- 先来先服务算法 (FCFS)
 - FCFS调度算法的平均作业周转时间与作业提交的顺序有关
- 例子
 - 作业1：28；作业2：9；作业3：3 (**都处于就绪状态**)
 - 提交顺序：1、2、3 → 平均周转时间：35
 - 提交顺序：2、1、3 → 平均周转时间：28
 - 提交顺序：3、2、1 → 平均周转时间：18

例子：

- 三个进程按顺序就绪：P1、P2、P3
进程P1 执行需要24s， P2和P3各需要3s

◎ 采用FCFS调度算法：



吞吐量： $3 \text{ jobs} / 30\text{s} = 0.1 \text{ jobs/s}$

周转时间TT： P1:24； P2:27； P3:30

平均周转时间： 27s

例子：

- 三个进程按顺序就绪：P1、P2、P3
进程P1 执行需要24s， P2和P3各需要3s

- ◎ 改变调度顺序：P2、P3、P1



吞吐量： $3 \text{ jobs} / 30 \text{ s} = 0.1 \text{ jobs/s}$

周转时间TT：P1:30；P2:3；P3:6；
平均周转时间：13s

作业调度和低级调度算法(2)

- 最短作业优先算法(SJF)
 - 以进入系统的作业所要求的CPU时间为标准，总选取估计计算时间最短的作业投入运行
 - 缺点：
 - 很难估计作业运行时间
 - 容易出现饥饿(长运行时间作业)
 - 优点：
 - SJF的平均作业周转时间比FCFS要小，故它的调度性能比FCFS好。

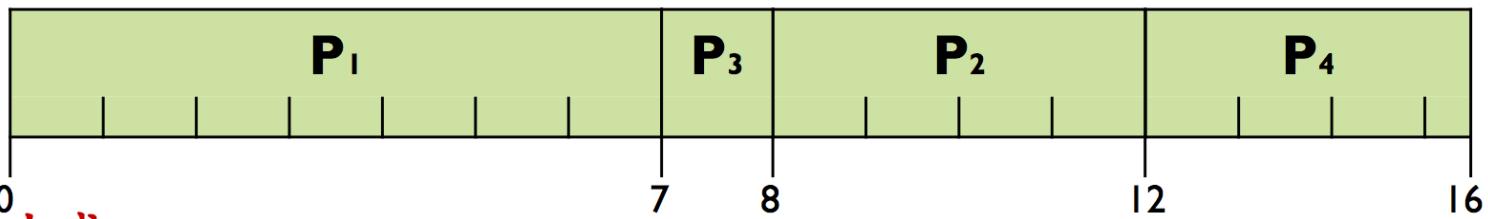
作业调度和低级调度算法 (2)

- SJF例子
 - 四个作业同时到达系统并进入调度： 作业名/所需CPU时间:作业1/9， 作业2/4， 作业3/10， 作业4/8。
 - SJF作业调度顺序为作业2、 4、 1、 3，
 - 平均作业周转时间 $T = 17$ ， 平均带权作业周转时间 $W= 1.98$ 。
 - 如果施行FCFS调度算法， 平均作业周转时间 $T = 19$ ， 平均带权作业周转时间 $W = 2.61$

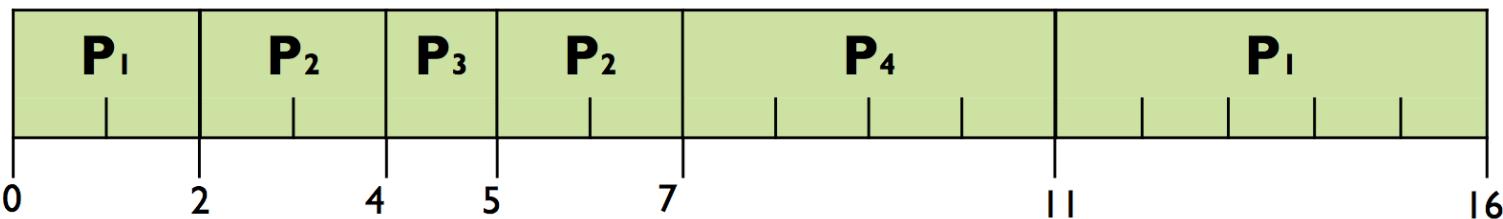
短作业优先SJF调度算法举例

进程	到达时刻	运行时间
P1	0	7
P2	2	4
P3	4	1
P4	5	4

非抢占式



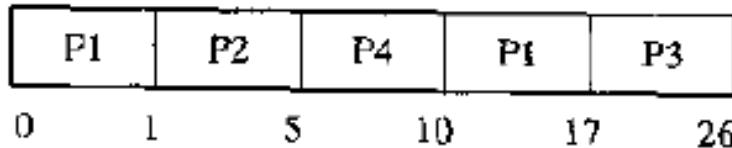
抢占式



作业调度和低级调度算法 (3)

- 最短剩余时间优先算法(SRTF)
 - SRTF把SJF算法改为抢占式的
 - 此算法不但适用于JOB调度，同样也适用于进程调度
- 例子

进程	到达系统时间	所需 CPU 时间/ms
P1	0	8
P2	1	4
P3	2	9
P4	3	5



SRTF平均等待时间 = 6.5 毫秒

SJF调度平均等待时间 = 7.75 毫秒

作业调度和低级调度算法 (4)

• 响应比最高者优先算法(HRRF)

- FCFS: 只考虑等待时间，忽略运行时间
- SJF: 只考虑运行时间，忽略等待时间
- HRRF : 折中方案
 - 响应比 = $1 + \text{已等待时间}/\text{估计运行时间}$
 - 响应比最大的最先运行
- 优点：
 - 短作业容易得到较高响应比，
 - 长作业等待时间足够长后，也将获得足够高的响应比 → 饥饿现象不会发生。

作业调度和低级调度算法(4)

- 是一个综合的算法
- 计算每个进程的响应比R
- 总是选择响应比最高的进程



响应比R = 作业周转时间 / 作业处理时间
= (作业处理时间+作业等待时间) / 作业处理时间
= 1 + (作业等待时间 / 作业处理时间)



作业调度和低级调度算法 (5)

- 优先级调度算法-静态优先数法

- 使用外围设备频繁者优先数大，这样有利于提高效率；
- 重要算题程序的进程优先数大，这样有利于用户；
- 进入计算机时间长的进程优先数大，这样有利于缩短作业完成的时间；
- 交互式用户的进程优先数大，这样有利于终端用户的响应时间等等，

作业调度和低级调度算法 (6)

- 优先级调度算法-动态优先数法

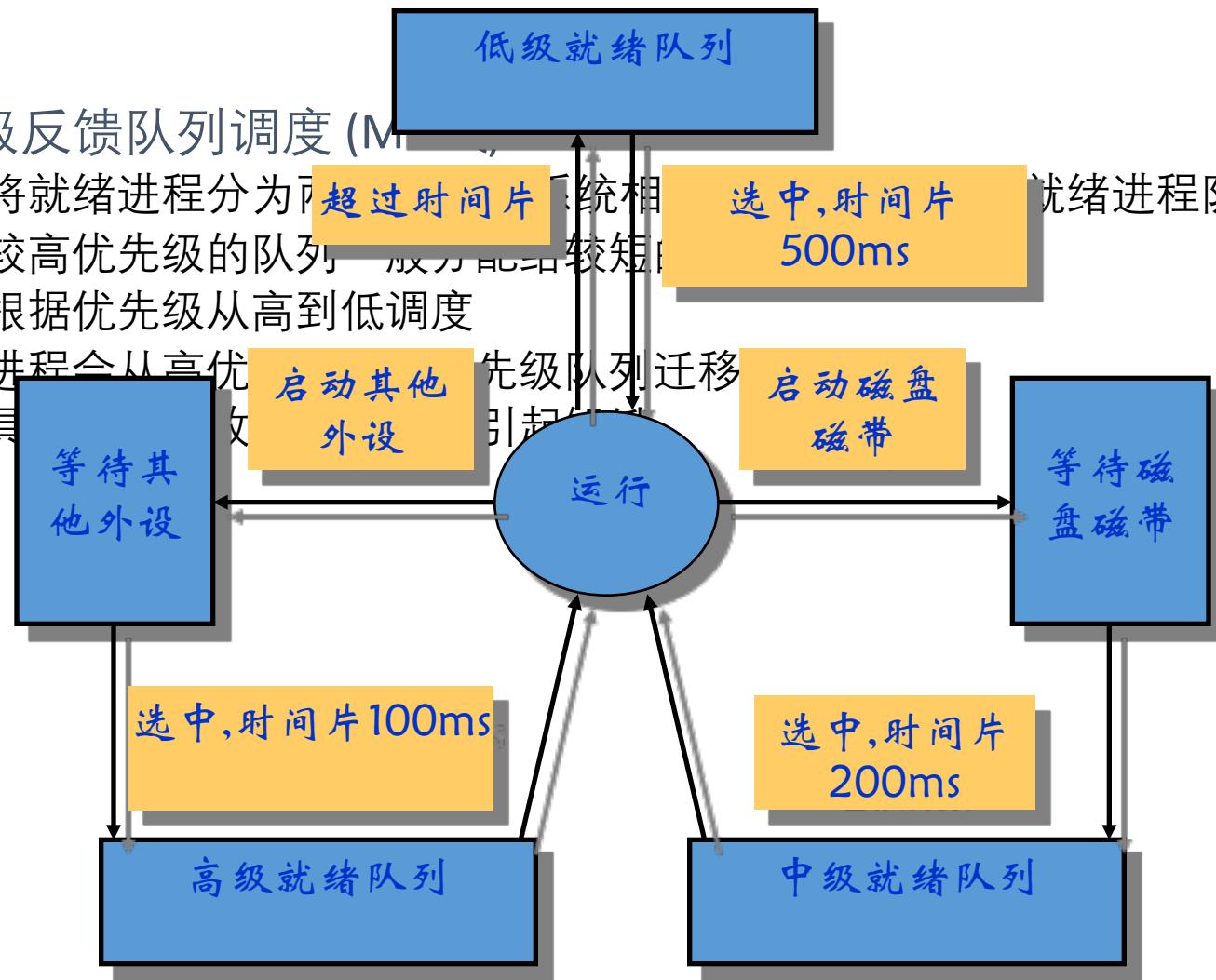
- 根据进程占有CPU时间多少来决定:当进程占有CPU时间愈长,那么, 在它被阻塞之后再次获得调度的优先级就越低, 反之,进程获得调度的可能性越大;
- 根据进程等待CPU时间多少来决定:当进程在就绪队列中等待时间愈长,那么, 在它被阻塞之后再次获得调度的优先级就越高, 反之,进程获得调度的可能性越小

作业调度和低级调度算法 (7)

- 时间片轮转调度算法
 - 调度程序每次把CPU分配给就绪队列首进程使用一个时间片。当这个时间片结束时，强迫一个进程让出处理器，让它排列到就绪队列的尾部，等候下一轮调度
- 轮转策略可防止那些很少使用外围设备的进程过长的占用处理器而使得要使用外围设备的那些进程没有机会去启动外围设备
- 轮转策略与间隔时钟

作业调度和低级调度算法 (8)

- 多级反馈队列调度 (Multi-level Feedback Queue Scheduling)
 - 将就绪进程分为不同优先级的队列
 - 较高优先级的队列一般分配较短的时间片
 - 根据优先级从高到低调度
 - 进程会从高优先级队列启动其他外设
 - 具有反馈机制，引起优先级迁移



作业调度和低级调度算法 (9)

- 彩票调度算法
 - 为进程发放针对各种资源（如CPU时间）的彩票。调度程序随机选择一张彩票，持有该彩票的进程获得系统资源
 - 进程都是平等的，有相同的运行机会。如果某些进程需要更多的机会，可被给予更多彩票，增加其中奖机会

• Spark 调度策略

- Spark调度分三个层次
 - 第一层，Spark应用间：Spark提交作业到YARN上，由YARN来调度各作业间的关系，可以配置YARN的调度策略为FAIR或FIFO。
 - 这一层可以再分两层，第一小层是YARN的队列，第二小层是队列内的调度。通过设置不同队列的minishare、weight等，来实现不同作业调度的优先级，这一点Spark应用跟其他跑在YARN上的应用并无二致，统一由YARN公平调度。比较好的做法是每个用户单独一个队列，这种配置FAIR调度就是针对用户的了，可以防止恶意用户提交大量作业导致拖垮所有人的问题。这个配置在hadoop的yarn-site.xml里。

- ~~Spark调度策略~~^{第二层，Spark应用内}（同一个sparkContext），可以配置一个应用内的多个TaskSetManager间调度为FIFO还是FAIR。

欲知后事如何，且听下回分解。

Acknowledge

These slides are adapted from 南京大学计算机系课程ppt; 以及北京大学陈向群老师的操作系统slides.

Appendix

更多的调度算法

实时调度算法

- 实时系统
 - 硬实施系统：必须满足时间限制
 - 软实施系统：允许偶尔超时
- 实时系统响应事件
 - 周期性事件
 - 非周期性事件
- 可调度条件
 - m 个周期性事件，事件*i*的周期为 P_i ，每个事件需要 C_i 秒的CPU时间来处理

$$C_1/P_1 + C_2/P_2 + \dots + C_m/P_m \leq 1$$

实时调度算法

- 单比率调度算法
 - 为每个进程分配一个与事件发生频率成正比的优先数
 - 调度优先数最高的就绪进程
 - 采取抢占式分配策略
- 限期调度算法
 - 就绪队列按截止期限排序
 - 采取抢占式分配策略
- 最少裕度法
 - 裕度=截止时间-(就绪时间+计算时间)
 - 选择裕度最少的进程执行

多处理器调度算法 (1)

- 负载共享调度算法
 - 维护全局就绪队列
 - 有一个CPU空闲就选择一个进程（线程）去运行
- 优点
 - 调度程序可以运行在任何CPU上
 - 避免空闲CPU
- 缺点
 - 就绪队列必须互斥 → 性能瓶颈
 - 被剥夺线程无法保证在原CPU上运行 → 高速缓存失效
 - 一个进程的线程可能运行在不同CPU上 → 通信、同步开销大

多处理器调度算法 (2)

- 群调度算法 (Gang Scheduling)
 - 基于1对1的原则将一组紧密相关的进程（线程）作为整体进行调度
 - 例如两个需要通信的线程
- 优点：减少进程切换，增加运行效率
- 缺点：
 - 如何确定线程的相关性

多处理器调度算法 (3)

- 专用处理器调度算法
 - 将一个进程的一组线程调度到同一组CPU上运行，直到进程结束
- 优点：
 - 保证进程获得最快的运行速度
- 缺点：
 - 阻塞线程不会让出CPU
 - 这在拥有几十、上百个处理器的高度并行系统上并不是太大问题

多处理器调度算法 (4)

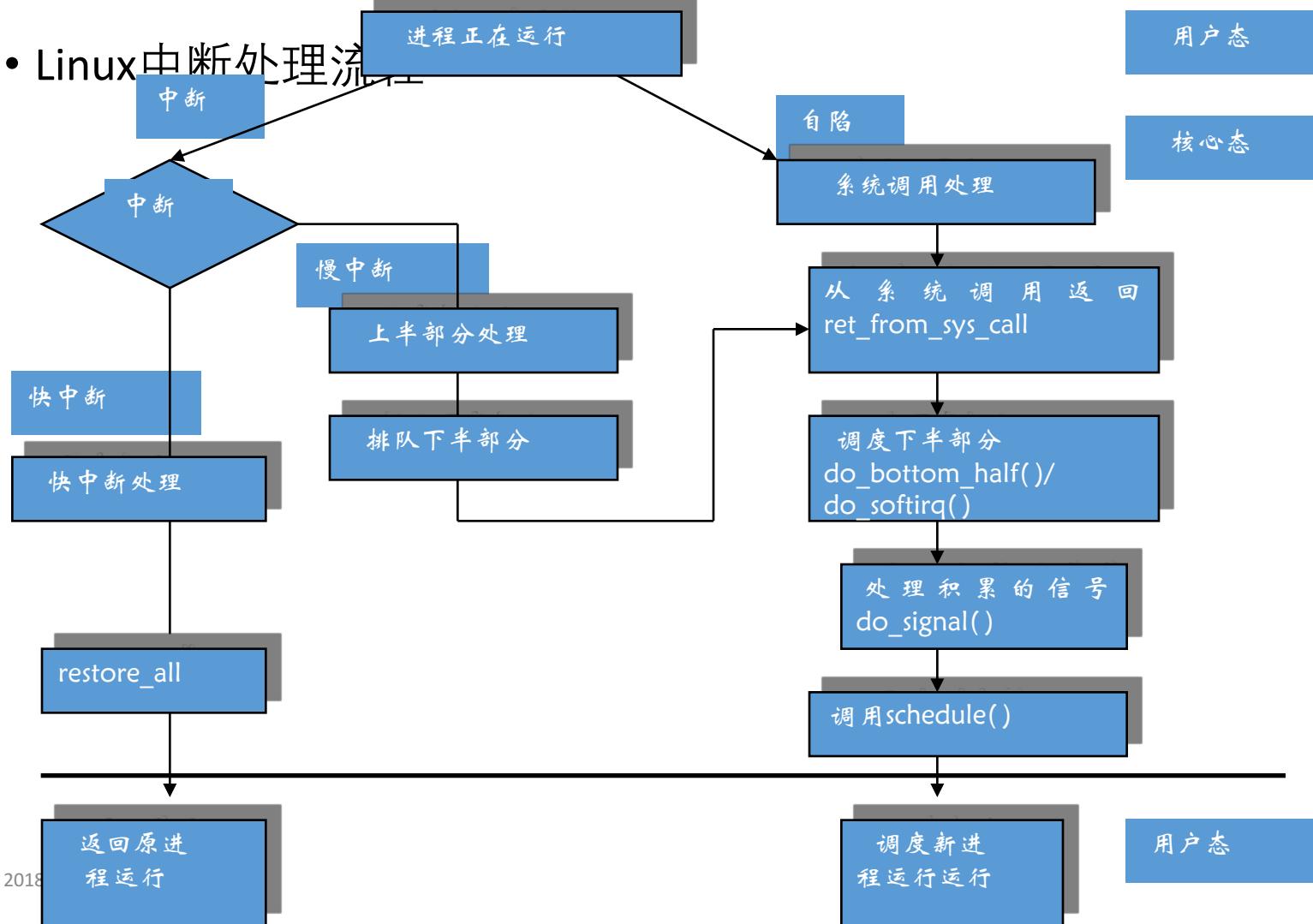
- 动态调度算法
 - 操作系统负责在应用进程中分配CPU
 - 应用进程负责在线程中分配分配到的CPU
- 对专用处理器算法的改进
 - 允许抢占CPU

Appendix

Linux and windows 中断处理具体机制

Linux中断处理

• Linux中断处理流程



快中断 vs 慢中断

慢中断	快中断
保存所有寄存器	只保存常规C函数可以修改的寄存器
开中断	关中断
处理结束，重新调度	返回原进程

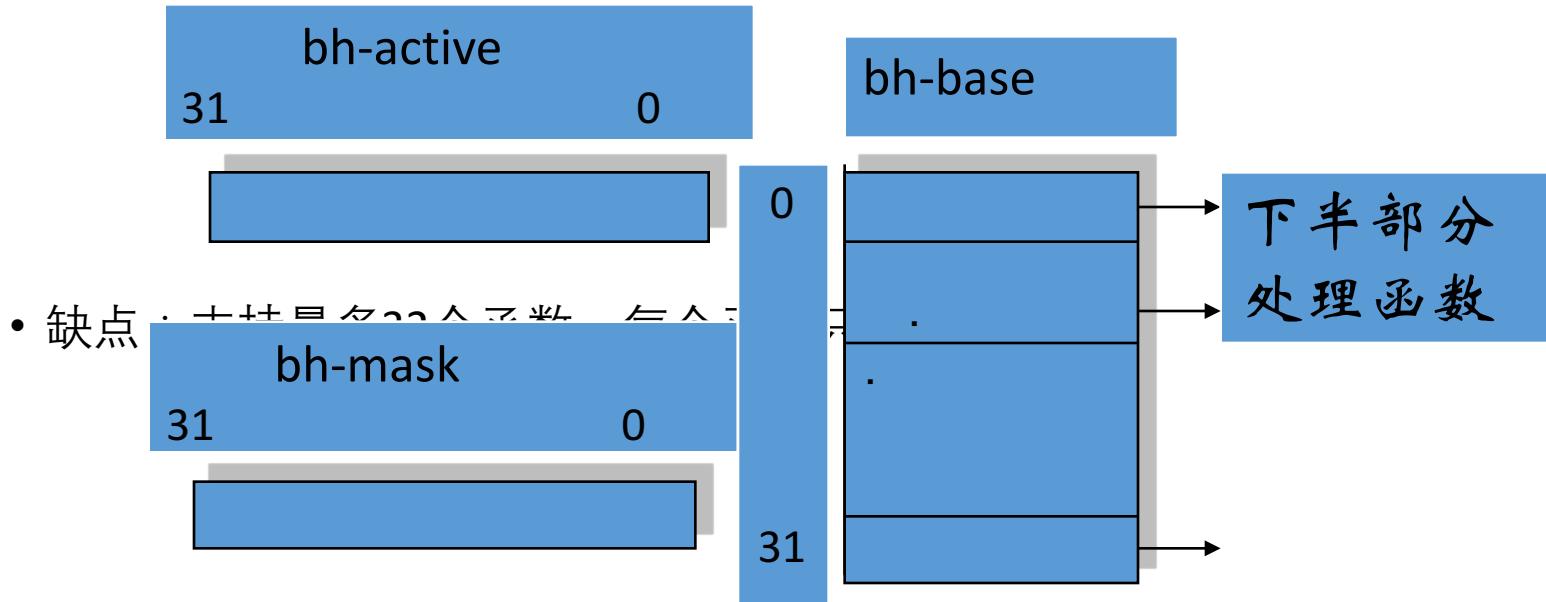
- Linux慢中断
 - 上半部，下半部

下半部处理

- 下半部
 - 开中断
 - 可延迟执行
- 实现方式：
 - bottom half、
 - task queue、
 - tasklet、
 - work queue、
 - Softirq

下半部分(bottom half)

- 实现原理
 - BH数组、函数入口标指针 bh_base、函数安装标志bh_mask、函数处理志bh_active



任务队列

- 实现原理：将下半部分处理函数扔到队列中去，在bh向量表中挂接run_task_queue
- 预定任务队列：
 - 1) 定时器队列(TQ_TIMER)：
 - 2) 即时队列(TQ_IMMEDIATE)：
 - 3) 进程调度队列(TQ_SCHEDULE)：
 - 4) 磁盘队列(TQ_DISK)：
- 定时器Top half 与bottom half协调工作的例子
- 任务队列扩展了bh支持的处理函数的个数，但仍然无法支持并行

小任务(tasklet)

- Linux 2.4开始
- 更好的支持SMP
 - 多个小任务可以被多个cpu同时执行，自己串行
 - 允许中断，但不能阻塞
 - 优先级的概念
- 基于软中断实现
 - 封装软中断，接口更简单
- 在新版Linux中，tasklet是建议的异步任务延迟执行机制

工作队列(work queue)

- Linux 2.5开始
- 把一个任务延迟交由内核线程处理
 - 允许重新调度、阻塞
- 如果下半部任务需要阻塞，需要获取信号量或需要获得大量主存时，那么，可选择工作队列，否则可使用tasklet或softirq。

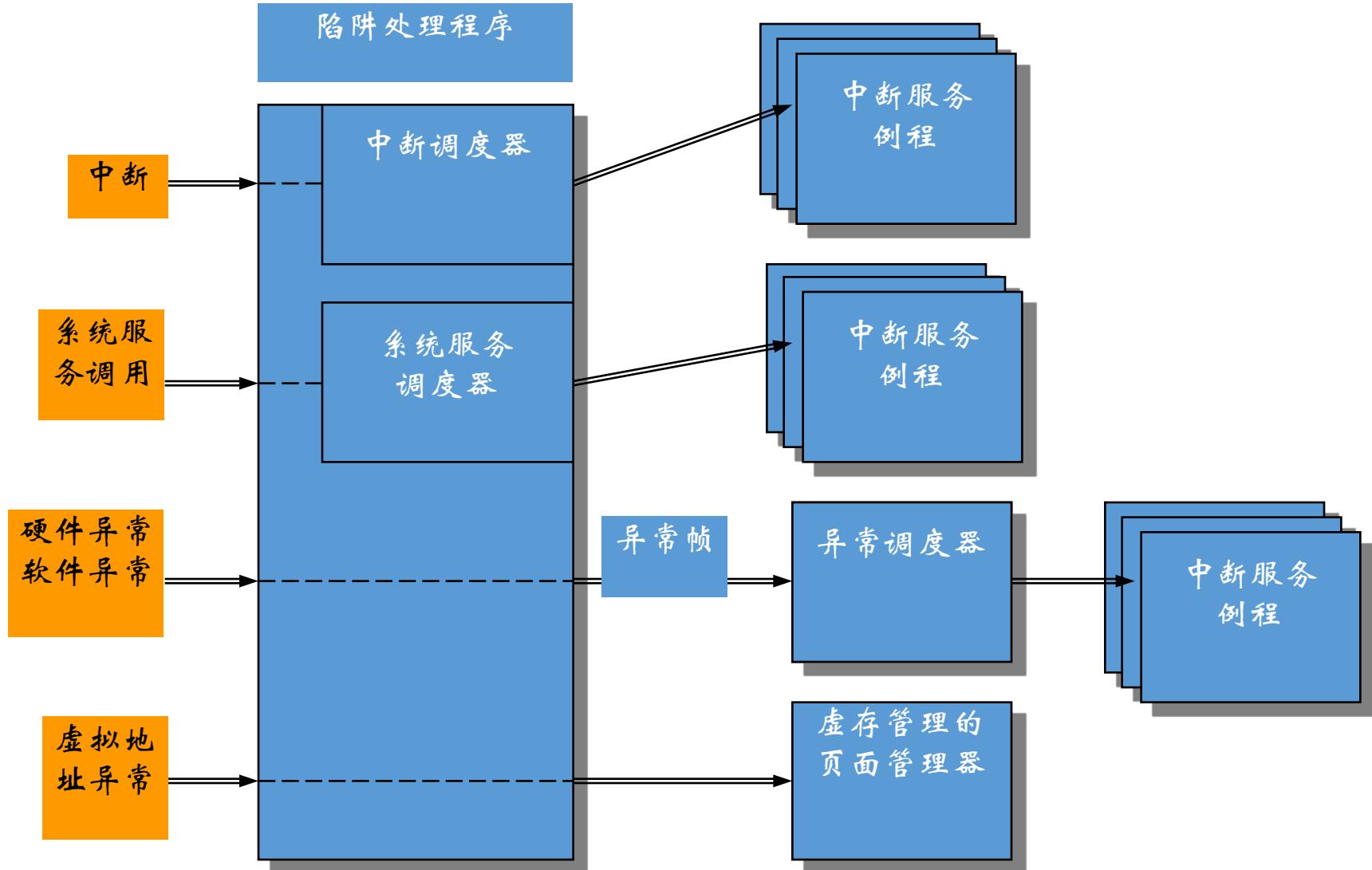
软中断(softirq)

- Tasklet 允许动态注册，软中断编译时静态确定
- 供执行频率和时间要求很高的下半部分使用

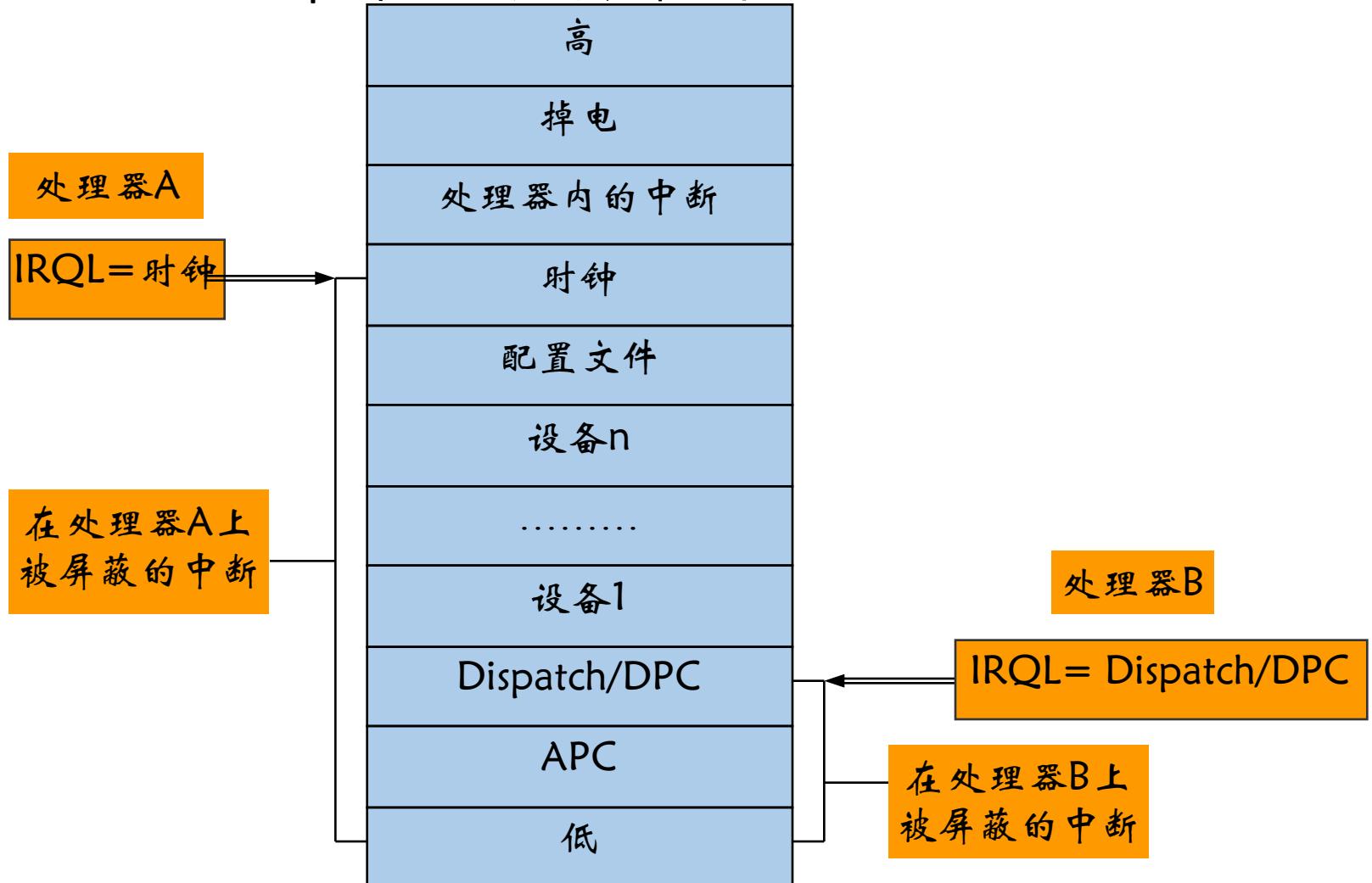
```
enum {  
    HI_SOFTIRQ,          //高优先级tasklet  
    TIMER_SOFTIRQ,        //定时器下半部分  
    NET_TX_SOFTIRQ,       //发送网络数据包  
    NET_RX_SOFTIRQ,       //接收网络数据包  
    SCSI_SOFTIRQ,         //SCSI下半部分  
    TASKLET_SOFTIRQ,      //公共tasklet  
};
```

- 系统调用返回，从异常中返回，在调度程序中以及处理完硬件中断之后

Win2000/xp中断处理机制



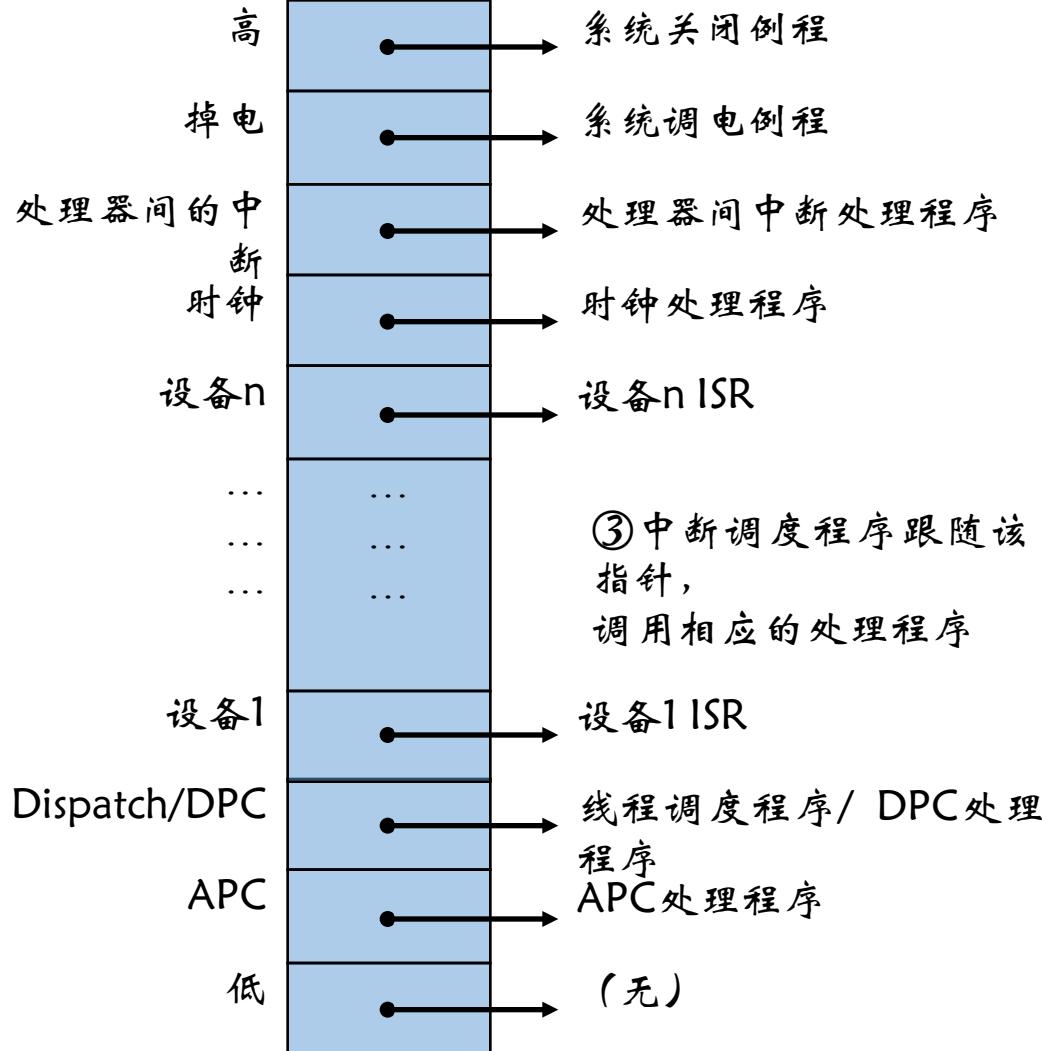
Win2000/xp中断屏蔽机制



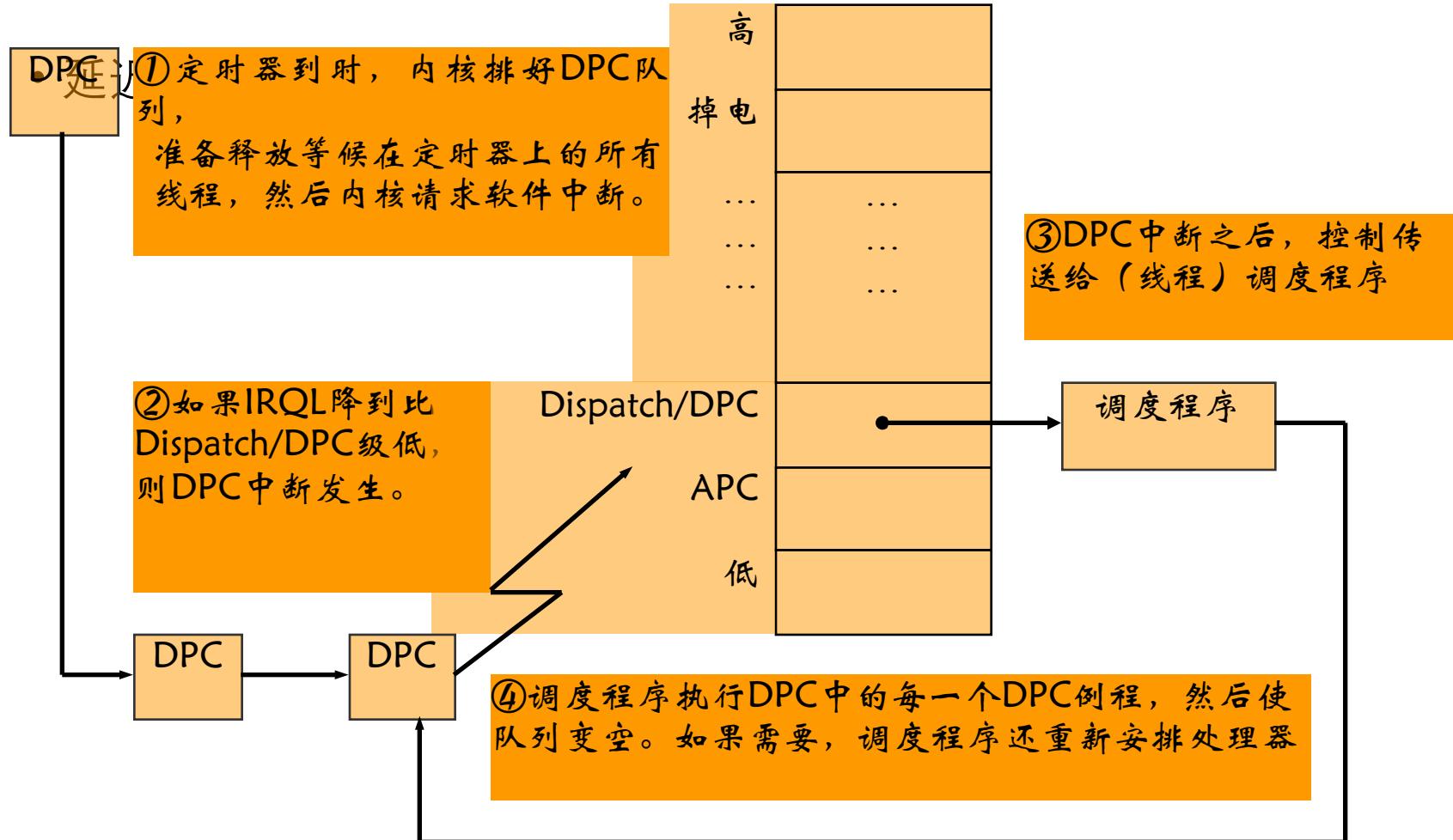
Win2000/xp中断处理流程

①有中断产生

②中断调度程序接收
到中断源的IRQL,
用作查询IDT的索引



Win2000/xp 软件中断 (1)



Win2000/xp 软件中断 (2)

- 异步过程调用 (APC)
 - 中断特定线程运行
 - 常用于异步I/O通知
- DPC vs APC
 - DPC队列是系统范围的， APC队列是线程范围的
 - DPC优于任何一个线程函数， 屏蔽线程调度
 - APC特定于一个线程， 在这个线程执行前执行

Appendix(II)

Linux 和window 调度算法详解

Linux 调度算法

- Linux 2.4 调度算法
- Linux 2.6 调度算法

Linux 2.4 调度算法

- 策略(Policy)
 - SCHED_OTHER普通类任务
 - 用于非实时进程的调度
 - 基于优先级的时间片轮询
 - SCHED_FIFO先进先出实时类任务
 - 用于实时进程
 - 没有时间片的概念
 - SCHED_RR轮转法实时类任务
 - 用于实时抢占进程
 - 时间片定量

一些重要变量

- priority: 静态优先级 (从2.4开始就取消了)
- nice : 进程可控优先级因子 (-20~19), 2.4以后就是静态优先级
- rt_priority : 实时进程静态优先级 (0~99)
 - 实时进程优先级= $rt_priority + 1000$
- counter : 进程目前时间片配额, 也称进程动态优先级
- weight=counter+20-nice

动态优先级的产生与变化

- 静态优先级和调度策略从父进程继承
- 初始动态优先级也从父进程继承
 - $\text{counter}(\text{子进程}) = (\text{counter}(\text{父进程}) + 1) / 2$
- counter值的变化
 - 每一次时钟中断， $\text{counter}--$
 - 减为0：让出处理器
 - 处于等待状态的进程counter逐渐增加
- 当所有可运行进程的counter都为0时， 系统重新计算所有进程的counter：
 - $\text{p-}>\text{counter} = (\text{p-}>\text{counter} \gg 1) + \text{NICE_TO_TICKS}(\text{p-}>\text{nice})$
 - 等待进程的counter大于nice

进程切换时机

- 被动放弃
 - 时间片用完
 - 一个更高优先级的进程被唤醒
 - 置need_resched为1, 合适时机调度
- 主动放弃1
 - 执行系统调用改变状态
 - sleep()、pause()、wait()...
 - 直接调用schedule()
- 主动放弃2
 - 执行等待系统调用, 如read、write等

Schedule函数

- 1、处理软中断
- 2、处理当前进程
 - Policy为SCHED_RR&counter=0,移入可运行状态尾部
 - 可中断等待&消息已到：移入可运行状态尾部
 - 从可运行队列移出
- 3、选择进程运行：
 - 查找weight最大的进程执行，或者重新计算所有进程动态优先级

Linux 调度算法

- Linux 2.4 调度算法
- Linux 2.6 调度算法

Linux 2.4 调度算法

- 策略(Policy)
 - SCHED_OTHER普通类任务
 - 用于非实时进程的调度
 - 基于优先级的时间片轮询
 - SCHED_FIFO先进先出实时类任务
 - 用于实时进程
 - 没有时间片的概念
 - SCHED_RR轮转法实时类任务
 - 用于实时抢占进程
 - 时间片定量

一些重要变量

- priority: 静态优先级 (从2.4开始就取消了)
- nice : 进程可控优先级因子 (-20~19), 2.4以后就是静态优先级
- rt_priority : 实时进程静态优先级 (0~99)
 - 实时进程优先级= rt_priority+1000
- counter : 进程目前时间片配额, 也称进程动态优先级 (SCHED_OTHER)
- weight=counter+20-nice

动态优先级的产生与变化

- 静态优先级和调度策略从父进程继承
- 初始动态优先级也从父进程继承
 - $\text{counter}(\text{子进程}) = (\text{counter}(\text{父进程}) + 1) / 2$
- counter值的变化
 - 每一次时钟中断， counter--
 - 减为0：让出处理器
 - 处于等待状态的进程counter逐渐增加
- 当所有可运行进程的counter都为0时， 系统重新计算所有进程的counter：
 - $\text{p->counter} = (\text{p->counter} \gg 1) + \text{NICE_TO_TICKS}(\text{p->nice})$
 - 等待进程的counter大于nice

进程切换时机

- 被动放弃
 - 时间片用完
 - 一个更高优先级的进程被唤醒
 - 置need_resched为1, 合适时机调度
- 主动放弃1
 - 执行系统调用改变状态
 - sleep()、pause()、wait()...
 - 直接调用schedule()
- 主动放弃2
 - 执行等待系统调用, 如read、write等

Schedule函数

- 1、处理软中断
- 2、处理当前进程
 - Policy为SCHED_RR&counter=0,移入可运行状态尾部
 - 可中断等待&消息已到：移入可运行状态尾部
 - 从可运行队列移出
- 3、选择进程运行：
 - 查找weight最大的进程执行，或者重新计算所有进程动态优先级

SMP进程管理

- Linux 2.0开始支持SMP
- 进程task_struct含有当前以及上次使用CPU的编号
- CPU亲合力(CPU Affinity)
 - 一个或者多个进程可以与一个或者多个CPU绑定
 - `sched_setaffinity(pid_t pid, unsigned int cpusetsize, cpu_set_t *mask)`

Linux 2.6 调度算法

- O(1)调度程序
 - 调度时间恒定—常数时间
 - 与进程数目、CPU数目无关
- 其他特性：
 - 更好的支持SMP：每个CPU都有可运行队列
 - 强化SMP亲和力 & 负载均衡
 - 确保响应时间，及时调度交互式进程
 - 保证公平性

task_struct中调度相关的变量

- Policy
 - SCHED_NORMAL 非实时进程
 - SCHED_FIFO 实时进程，采用先进先出的调度算法；
 - SCHED_RR 实时进程，采用轮转法。
- rt_priority-实时进程的优先级
 - 0~99
 - 不做动态优先级的改变
 - 数值越大优先级越低

task_struct中调度相关的变量

- static_prio-非实时进程静态优先级
 - 由nice转变而来 : $\text{static_prio} = \text{MAX_RT_PRIO} + 20 - \text{nice}$
 - 100-139的优先级范围
- sleep_avg-进程平均等待时间
 - 体现进程的交互程度, 或者说进程需要运行的紧迫程度
 - 该值越大, 进程的优先级就越高
 - 睡眠增加, 运行减小

task_struct中调度相关的变量

- prio-进程动态优先级
 - -5~5的奖惩
 - 主要由sleep_avg决定
 - 创建时子进程继承父进程的动态优先级
- prio_array_t *array-进程优先级数组
 - nr_active:数组中的进程数
 - Bitmap: 优先级位图
 - Queue: 优先级队列
 - 每个可运行队列维持两个优先级数组：活跃数组和过期数组

task_struct中调度相关的变量

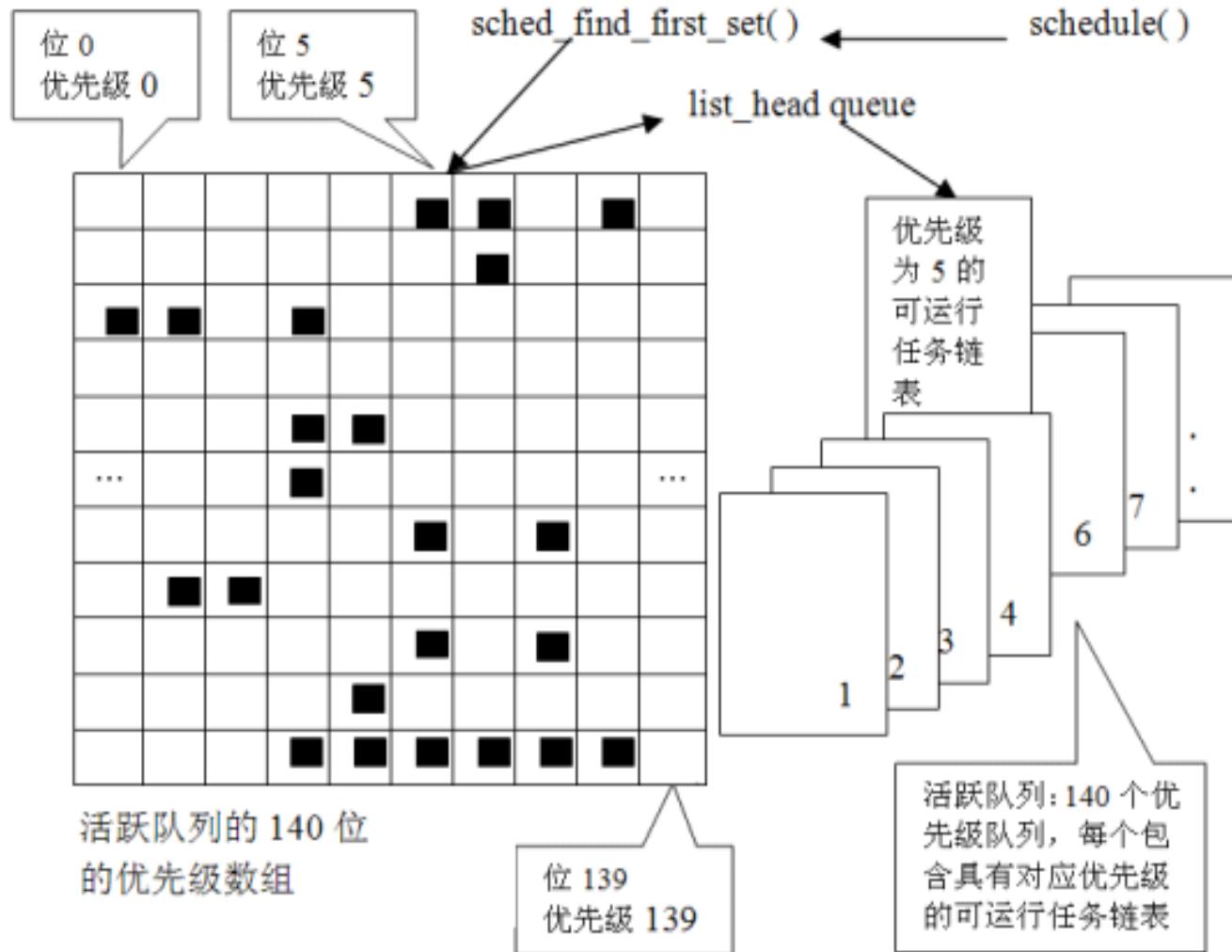
- `load_weight` : 平衡负载用的权重
 - 解决可运行队列出现的负载不均现象
- `CONFIG_PREEMPT`-内核可剥夺编译选项，当该开关开启时，v2.6 内核将会在更多内核安全点上检测`TIF_NEED_RESCHED`位，从而让刚被唤醒的高优先级任务减少延迟而尽快获得处理器运行。

调度算法的数据结构

- runqueue
 - 给定处理器上的就绪进程链表
 - 每个处理器一个
 - 每个就绪进程都归属于一个可运行队列
- 优先级数组

```
struct prio_array {  
    int nr_active; // 数组中的进程数  
    unsigned long bitmap[BITMAP_SIZE];  
        // 优先级位图  
    struct list_head queue[MAX_PRIO];  
        // 优先级队列  
};
```

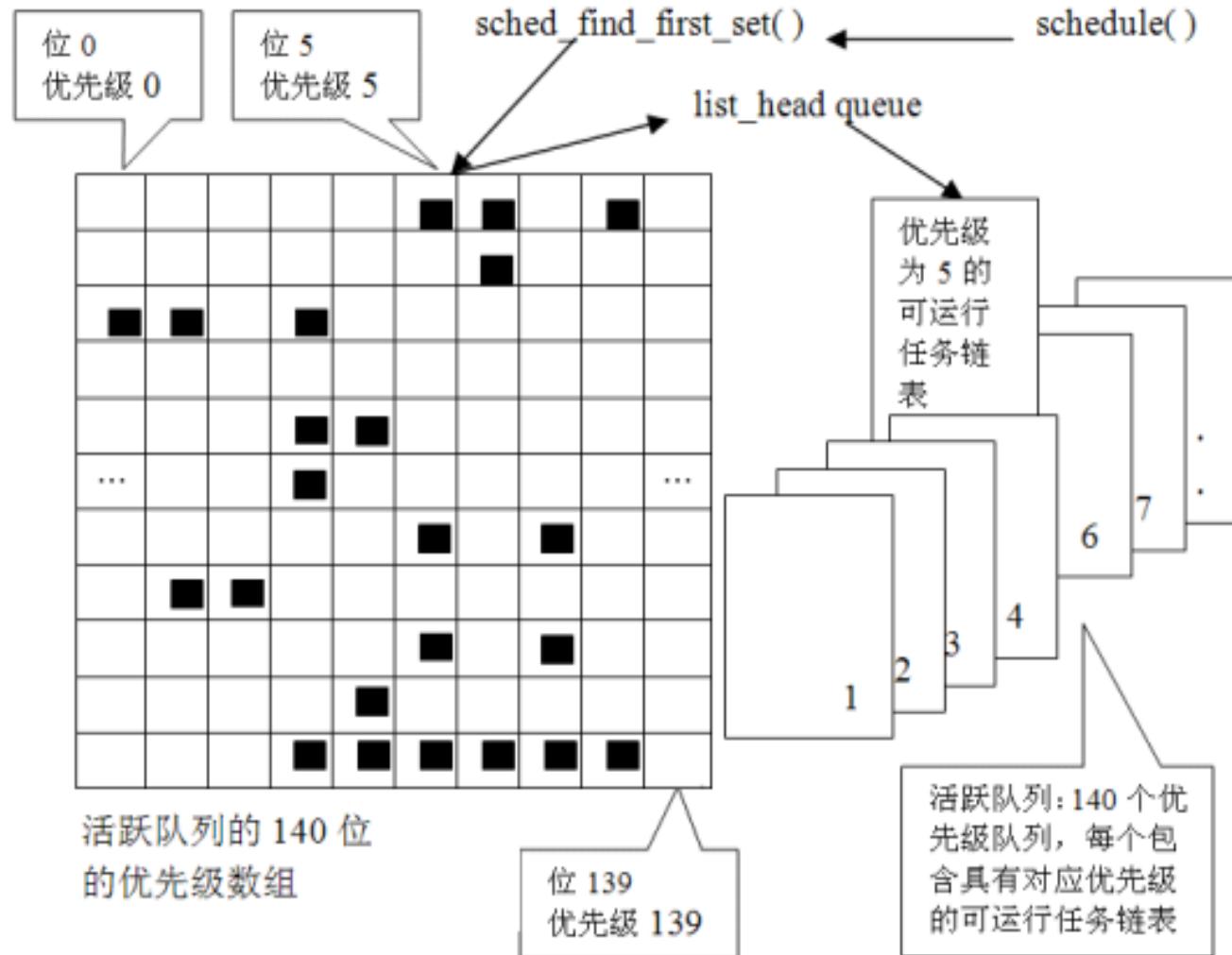
调度算法的数据结构



进程优先级和时间片

- 实时进程具有静态优先级，范围为0到99
- 非实时任务都有静态优先级对应于100到139的优先级范围，默认值为120
- 进程的动态优先级，它以nice值为基数，再加上-5到+5之间的进程交互性奖励或罚分
- SCHED_NORMAL调度时间片用完，进程移入过期队列
- SCHED_RR调度时间片用完后，进程返回原优先级队列，不会移到过期队列

O(1)调度算法(1)



O(1)调度算法(2)

- task_t *prev,*next; runqueue *rq; prio_array_t *array; int idx;
 - prev = current; //保存当前进程
 - rq = this_rq(); //当前运行队列
 - array = rq - > active;
 - if (unlikely(!array - > nr_active)) { //运行队列active变成空
 - rq - > active = rq - > expired; //活跃、过期数组切换
 - rq - > expired = array;
 - array = rq - > active;
 - ...
 - }
 - else idx = sched_find_first_bit(array - > bitmap); //从active队列选择进程
 - queue = array - > queue + idx;
 - next = list_entry(queue - > next,task_t,run_list);
 - ... //任务切换

负载平衡

- Load_balance()函数
- 执行时机：
 - 执行schedule()时当前可运行队列为空
 - 定时器调用
- 基本步骤
 - 找到最繁忙的运行队列
 - 从中选择一个优先级数组（过期数组）
 - 找到非空的其优先级最高的链表
 - 选择一个未运行、可移动、不在高速缓存中的进程，移到当前可运行队列
 - 重复以上步骤直至负载均衡

用户抢占和内核抢占

- Linux2.6完整地支持内核抢占，只要重新调度是安全的，内核就可在任何时间抢占正在运行的任务
- 用户抢占时机：系统调用或者中断处理程序返回用户空间
- 内核抢占时机：
 - 从中断处理程序返回内核空间
 - 内核中的任务显示调用schedule()或者因阻塞调用schedule()
- 抢占条件:当前任务没有持有锁

Windows 2003 进程调度

- 处理器调度的对象是线程，也称线程调度。
- 实现了基于优先级抢先式的多处理器调度系统，系统总是运行优先级最高的就绪线程。
- 线程可在任何可用处理器上运行，但可限制某线程只能在某处理器上运行，亲合处理器集合允许用户线程通过Win32调度函数选择它偏好的处理器

线程调度触发事件

- 线程调度出现在DPC/dispatch中断优先级
- 触发事件：
 - 一个新的线程进入就绪态
 - 当前运行线程时间配额用完
 - 当前运行线程阻塞
 - 当前运行线程通过系统调用或者被系统本身改变优先级
 - 当前运行线程改变其亲和处理器集合

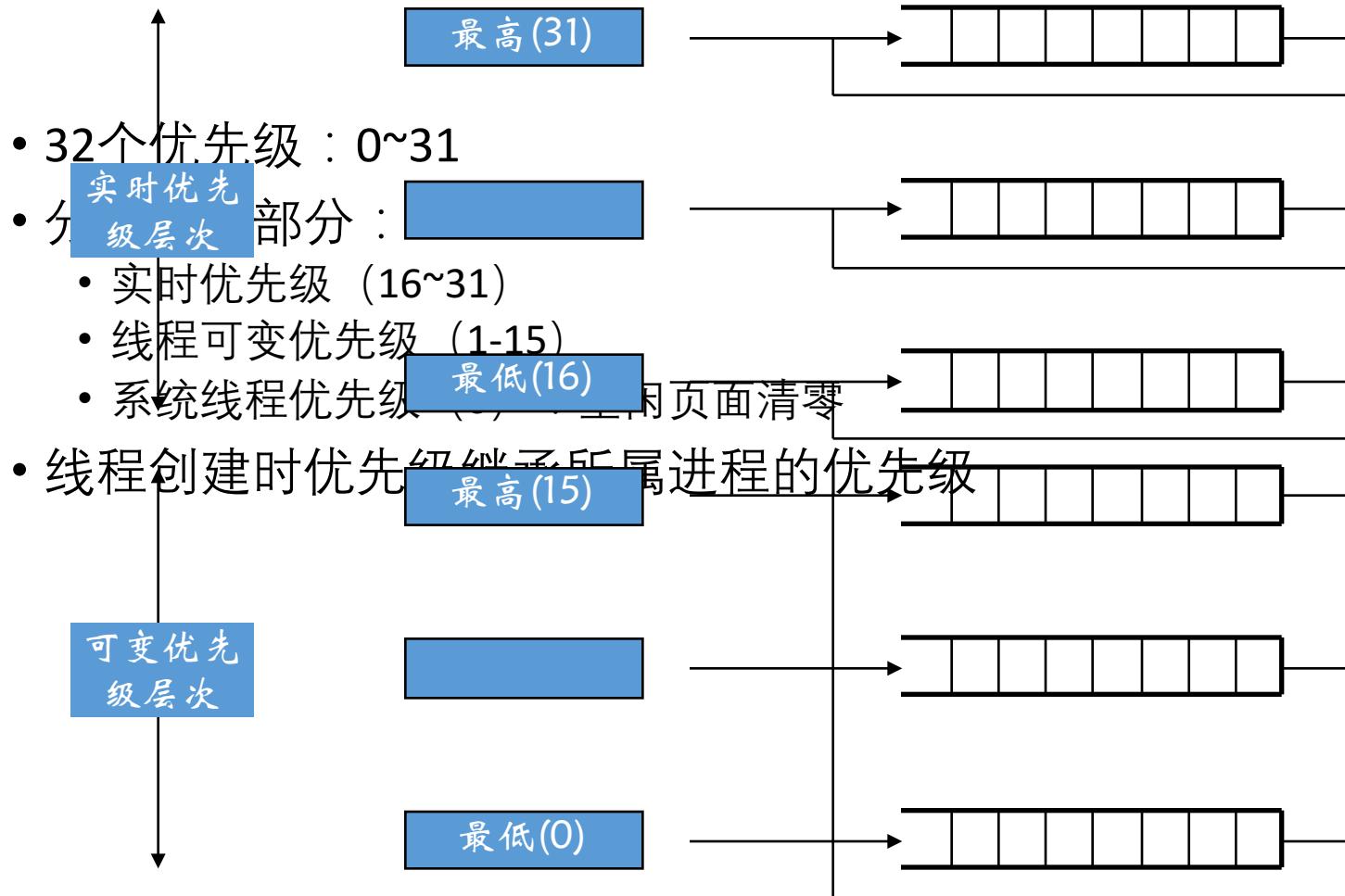
进程优先级

- 进程分为4类
 - 空闲进程 (优先级为4)
 - 普通进程 (优先级为7)
 - 高优先级进程 (优先级为13)
 - 实时进程 (优先级为24)
- 进程初始优先级由父进程指派
- 普通进程
 - 前台运行：优先级为9
 - 后台运行：优先级为7
- 任务管理器进程：优先级为13
- 核心进程：优先级为24

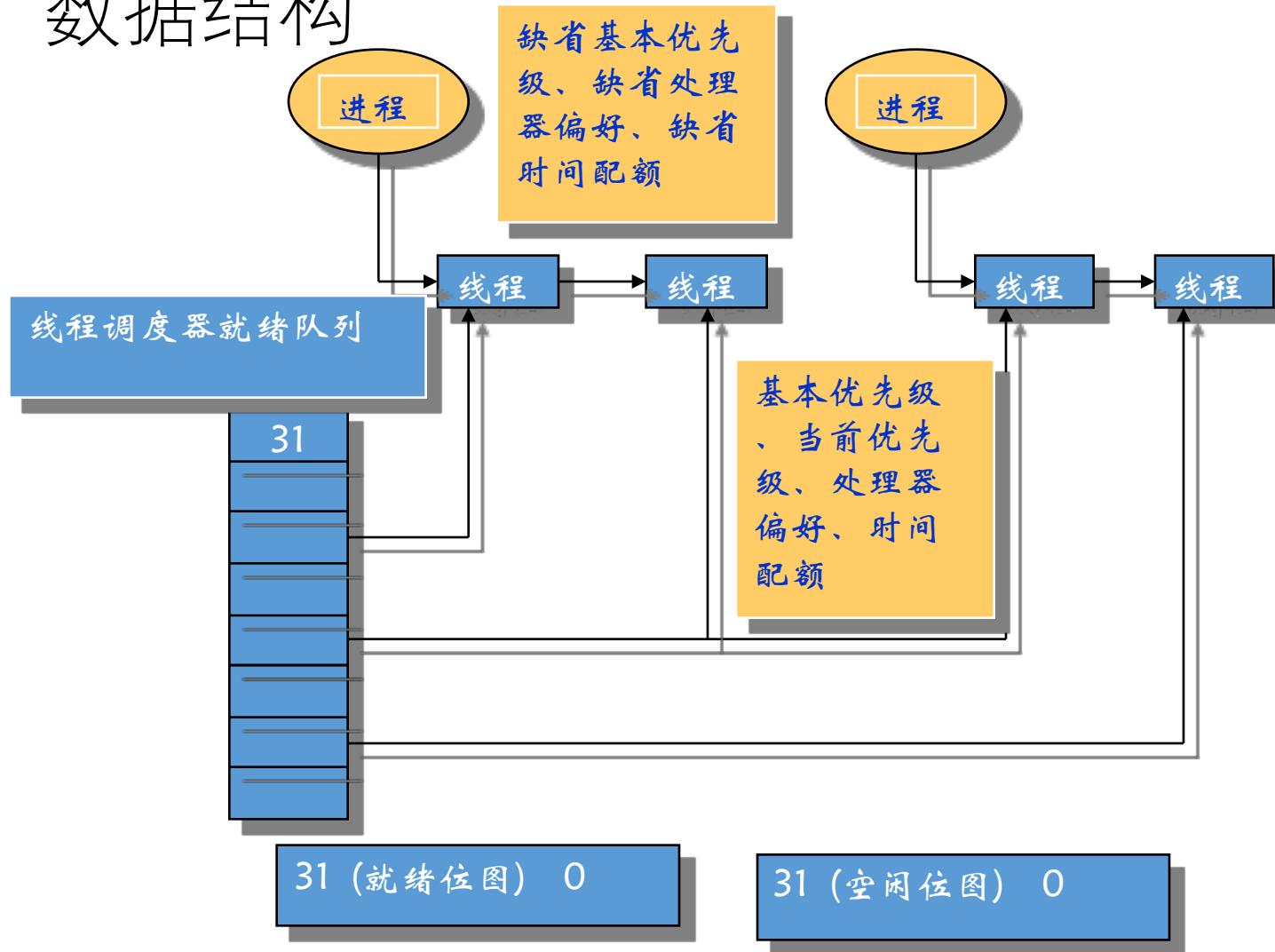
线程时间配额

- 线程被调度运行时，可运行一个被称为时间配额（quantum）的时间。
- 时间配额是允许线程连续运行的最大时间长度，随后系统会中断线程运行，判断是否需要降低该线程的优先级，查找是否有其他高优先级或相同优先级的线程等待运行。
- 由于抢先式调度特征，一个线程的一次调度执行可能并没有用完它的时间配额就被抢先

线程优先级



Windows 2003线程调度 数据结构



线程切换策略

- 主动切换
- 抢占
 - 以优先级为准，用户态可以抢占内核态
 - 被抢占线程放置到相应优先级就绪队列的队首
- 时间配额耗完
- 结束

线程优先级提升

提升线程优先级的情况：

- 1) I/O操作完成。
- 2) 信号量或事件等待结束。
- 3) 前台进程中的线程完成一个等待操作。
- 4) 由于窗口活动而唤醒图形用户接口线程。
- 5) 线程处于就绪状态超过一定时间，但没能进入运行状态（处理器饥饿）

SMP上的线程调度

- 亲和关系
 - 亲和掩码从进程继承
 - 默认状态进程的掩码是系统上所有可用处理器的集合
- 线程的首选处理器和第二处理器
- 就绪线程处理器的选择
 - (1) 首选处理器
 - (2) 亲和处理器与空闲处理器的交集
 - (3) 空闲的最近使用处理器
 - (4) 正在执行调度代码的处理器
 - (5) 依据处理器标志从高到低扫描空闲处理器