彎曲評論

科技 · 人物 · 潮流

《孙钟秀 . 操作系程教程》注释(稿)

(第二章--处理器管理)

校注: 陈怀临

弯曲评论, www.valleytalk.org

2014年9月7日

第二章

处理器管理

处理器管理是操作系统的重要组成部分，负责管理、调度和分配计算机系统的重要资源——处理器，并控制程序的执行。由于处理器管理是操作系统最核心的部分，无论是应用程序，还是系统程序，最终都要在处理器上执行以实现其功能，因此，处理器管理的优劣将直接影响系统的性能。程序以进程的形式来占用处理器和系统资源，处理器管理中最重要的是处理器调度，即进程调度，也就是控制、协调进程对处理器的竞争。

进程可被调度在一个处理器上交替地执行，或在多个处理器上并行执行。不同类型的操作系统可能采取不同的调度策略，交替执行和并行执行都是并发的类型。为了提高并发粒度和降低并发开销，现代操作系统引进了线程的概念，此时进程仍然是资源分配和管理的单位，线程则成为处理器调度的基本单位。本章在简要介绍处理器的硬件运行环境之后，首先将着重介绍计算机系统的中断机制，然后详细论述进程和线程的基本概念及其实现，最后全面讨论各个层次的处理器调度方法。

2.1 中央处理器 CPU

2.1.1 处理器

Fetch, Decode, Execute

1. 单处理器系统和多处理器系统

处理器的任务是按照程序计数器的指向从主存读取指令，对指令进行译码，取出操作数，然后执行指令。如果计算机系统只包含一个运算处理器，称为单处理器系统；如果计算机系统包含多个运算处理器，则称为多处理器系统。 Pipeline结构。

早期的计算机是基于单处理器的顺序处理机器，程序员编写串行代码，让指令在处理器上串行执行。为了提高计算机的处理速度，采用流水线技术，把指令分解成简单的、可独立执行的操作步骤，并将多条指令或多个操作步骤按照流水线方式部分重叠地执行，以此加快指令的执行速度。后来，流水线技术发展到更高级阶段，形成发射体系结构，其类型有超标量结构、超流水线结

SuperScalar。 SuperPipeline。

超标量（SuperScalar）的意思是：有多个相同的功能部件， 例如，多个整数加法器

VLW。现在通常不太考虑这些了。 Multi－Issue

构和超长指令字结构，设计的基本思路是：在一个机器周期内可以发射多条指令，同时取指令、译码并转储到保持缓冲区中，多个执行部件同时执行，只要存在空闲的执行部件，就会从非空保持缓冲区中读取指令并执行之，以提高指令执行的并行性。

随着硬件技术的不断进步，并行处理技术得到了迅猛的发展，多个功能部件和多个处理器可以同时工作以提高计算机系统的性能和可靠性，发展较为成熟的是多指令流多数据流结构的并行计算机，它分成两类：共享存储（紧密耦合）多处理器系统和分布存储（松散耦合）多处理器系统，前者共享所有处理器且平等地访问同一个物理主存；后者每个处理器均拥有自己的主存，处理器之间通过网络相连，在需要时通过网络交换数据。 MIMD。但工业界不太提这个。

根据处理器分配策略，共享存储多处理器系统分为主从式系统（Main/Slave Multi-Processor，MSP）和对称式系统（Symmetric Multi-Processor，SMP）。主从式系统的基本思路是：在特别的处理器上运行操作系统内核，在其他处理器上运行应用程序和系统程序，内核负责调度和分配处理器，并向其他程序提供各种服务。这种方式实现起来简单易行，但如果主处理器发生崩溃将导致系统崩溃，且极有可能形成性能瓶颈。

在对称式多处理器系统中，操作系统内核可以运行在任意处理机上，一个进程的多个线程可在不同的处理器上同时运行；服务器进程可使用多线程同时处理来自多个用户的请求；操作系统内核也可被设计成多进程或多线程以实现并行执行，系统中的任何处理器都可以访问任何存储单元及设备。SMP 机具有对称性、单一地址空间、低通信延迟和一致性高速缓存等特点，且可靠性及性价比高，可扩充性好。共享存储器只需保存一个操作系统和数据库副本，这样既有利于动态负载平衡，又有利于保证数据的完整性和一致性。很多计算机硬件设备制造商（如 HP、IBM、Sun、SGI）都设计和生产 SMP 机，这种机型大量用于联机分析处理、数据库和数据仓库等应用。

在分布存储多处理器系统中，每个处理机都有独立的主存和通道，各个处理单元之间通过预设的线路或网络进行通信，构成互联的计算机系统。集群（cluster）系统是分布存储多处理器系统的例子，是迄今为止所开发出来的另一种最为成功的并行机，它也是一种分布式系统。集群操作系统是分布式操作系统之一，运行时构成统一的计算资源；在集群系统中，每台计算机都是一个完整结点，脱离集群后可以独立地工作，从而具备极好的容错性和可伸缩性。当今，大多数通用操作系统既支持单处理器系统，又支持 SMP 系统和集群系统。

x86 GPR比较少。PowerPC， ARM， MIPS比较多。例如， 32个。

2. 寄存器

计算机系统的处理器包括一组寄存器，用于存放数据、变量和运算的中间结果。寄存器的数量根据处理器型号的不同而异，它们构成计算机一级存储，虽然其容量比主存小得多，但访问速度却很快，这组寄存器所存储的信息与程序的执行密切相关，构成处理器现场。不同类型的处理器由不同种类和数目的寄存器组成，在此以 Intel x86 为例，介绍各种寄存器。

（1）通用寄存器 GPR（General Purpose Register）

通用寄存器共 4 个，分别是 EAX、EBX、ECX 和 EDX，通常保存 32 位数据。为了进行 16 位操作并与 16 位机保持兼容性，通用寄存器的低位被当成 16 位寄存器，即 AX、BX、CX 和 DX；为了支持 8 位操作，寄存器的低 16 位可进一步划分为 8 位一组的高位字节和低位字节这两部分，

UMA（Unified Memory Architecture）

NUMA（No Unified Memory Architecture）

有些CPU不是拿特殊控制寄存器来做这些目的。而是通过GPR。但通过一个约定，例如，某个register是堆栈pointer。

作为 8 个 8 位寄存器，分别命名为 AH、BH、CH、DH 和 AL、BL、CL、DL。因此，4 个通用寄存器既支持 1 位、8 位、16 位和 32 位数据运算，又支持 16 位和 32 位存储器寻址。

(2) 指针及变址寄存器

指针及变址寄存器共 4 个，分别是 ESP、EBP、ESI 和 EDI。使用其低 16 位时，得到 SP、BP、SI 和 DI 寄存器，与 16 位机保持兼容性。ESP 是堆栈指针寄存器，保存栈顶元素的地址，由于栈是向下增长的，压入数据时首先要将栈指针 ESP 减 4，当然在弹出数据时 ESP 要加 4。为函数调用而分配的那部分栈空间称为栈帧，ESP 是很重要的寄存器，指向主存中当前堆栈的顶端，栈内保存着已经调用但尚未退出的每个函数的栈帧，其中包含返回地址、传递参数、局部变量等。EBP 是栈帧指针寄存器，指向栈中当前帧的位置。ESI 是源变址寄存器，EDI 是目的变址寄存器，也可当成通用寄存器使用，具有自动增量和减量的功能。

(3) 段选择符寄存器

段选择符寄存器是 6 个 16 位的段寄存器，它们分别是 CS、DS、SS、ES、FS、GS。在实模式下，段寄存器存放段基址；在保护模式下，段寄存器所存放的是某个段的选择符。因为 16 位的寄存器无法存放 32 位段基址，只好将其存放在描述符表中，由描述符给出 32 位段基址。因此，在 Intel x86 中，段寄存器称为选择符寄存器，主存中的指令和数据的地址可以通过段寄存器和指针寄存器共同给出。

(4) 指令指针寄存器和标志寄存器

指令指针寄存器 EIP 中存放下一条待执行指令的偏移量(相对于目前正在运行的代码段寄存器 CS 而言)偏移量加上当前代码段的基址形成下一条指令的地址。EIP 中的低 16 位可以被单独访问，叫做指令指针 IP 寄存器，用于 16 位寻址。

标志寄存器 EFLAGS 存放相关的处理器控制标志。

Protected

Paging

(5) 控制寄存器

共有 4 个 32 位控制寄存器，分别是 CR0、CR1、CR2 和 CR3，主要用于操作系统的分页机制。CR0 中包含标志位，第 0 位是保护允许位，用于启动保护模式；CR0 的第 31 位是分页允许位，表示硬件的分页部件是否被允许工作，其他位的功能还包括监控协处理器、任务转换等。CR1 暂未使用。CR2 是页故障线性地址寄存器，保存最近一次出现缺页时的 32 位线性地址。CR3 是页目录基址寄存器，保存页目录的物理地址，页目录总是置于以 4 KB 为单位的存储器边界上，因此，其地址的低 12 位总是 0。

Page Fault

CR3是内存中的页表基地址

(6) 外部设备使用的寄存器

① 数据寄存器或缓冲区

存放当前要在设备和主机之间传送的 I/O 数据，程序可以访问数据寄存器或缓冲区。

② 状态寄存器

保存外部设备或接口的状态信息，以便 CPU 在必要时测试状态，了解设备的工作情况，例如，命令是否完成、设备是否出错等。

③ 控制寄存器

segment register-->segment descriptor-->寻址。

现在都不用段寻址空间。都是flat memory。平坦结构。CS，DS，SS ES，FS，。。都是相同的value。

Memory Mapped I／O。基本上都是这种了。

CPU 传送给外部设备或接口的控制命令通过控制寄存器发送，例如，CPU 要启动磁盘工作，必须发送启动命令等。

每个设备控制器都有上述 3 种类型的寄存器用于和 CPU 之间的通信，每个接口所配备的寄存器数量是根据设备的具体需要而定的，例如，键盘只有一个 8 位数据寄存器，并把状态寄存器和控制寄存器合二为一；复杂的磁盘则需要多个数据寄存器、状态寄存器和控制寄存器。

CPU 和控制寄存器、数据寄存器及缓冲区之间的通信方式有以下三种。

① 为每个控制寄存器分配一个 I/O 端口号（8 位或 16 位），通过使用核心态 I/O 指令（IN/OUT），CPU 可以读写端口。在这种方式下，主存空间和 I/O 端口空间是各自独立的，早期的大多数计算机采用此方式工作。

② 把所有控制寄存器映射到主存空间，为每个寄存器分配唯一的主存地址，且与用户的可用主存地址不重叠。为控制寄存器所分配的地址位于地址空间的顶端，可以像主存单元一样被读写，不再需要专门的 I/O 指令。此方法由 PDP11 首先引入。

③ 混合方式。

既在主存空间开辟数据缓冲区，而控制寄存器又有其单独的 I/O 端口。Intel x86 采用这种结构：I/O 地址为 16 位，每个地址对应于一个字节宽的 I/O 端口，共表示 64 KB 的 I/O 地址空间；640 KB～1 MB 的主存地址保留，用做设备数据缓冲区。

上述方式的基本工作过程是：

① CPU 读取数据时，把所需数据地址（主存地址或 I/O 端口号）放在地址总线上；

② 在控制总线上插入读信号，同时另一条信号线表明数据来自 I/O 空间还是主存空间；

③ 由相应的对象（设备或主存）对请求做出响应。

3. 特权指令与非特权指令

计算机的基本功能是执行程序，最终被执行的是存储在主存中的机器指令代码，处理器根据程序计数器的指向从主存中取指令到指令寄存器，然后译码并执行，程序计数器将自动地增长或变为转移地址以指明下一条待执行指令的地址。

每台计算机的机器指令集合称为指令系统，反映计算机的功能和处理能力。一般来说，处理器的机器指令集是专用的，指令往往分为 5 类。 ISA

(1) 数据处理类

执行算术和逻辑运算。

(2) 转移类

如无条件转移、条件转移、计数转移等用于改变指令的执行序列。

(3) 数据传送类

在处理器的寄存器之间、寄存器和主存单元之间、主存单元之间交换数据。

(4) 移位与字符串类

移位分为算术、逻辑和循环移位；字符串处理包括字符串的传送、比较、查询和转换。

(5) I/O 类

现代CPU的指令都是RR模型。都是通过把内存地址的内容要装载到寄存器后才开始操作。

[注释]: 学习OS的时候，会接触CPU，例如x86. 其实x86不太适合做CPU教材例子。IA32为了支持许多历史设计含有了许多老东西，例如段地址等。现代CPU，包括Intel，都已经是采用Flat Memory结构。学生们没有必要去知道和理解之前的技术。应该直接一步到位。

任何一个CPU都有Status（状态）控制器

例如，某些CPU的控制寄存器也是独一份。如果应用程序都R／W，系统的状态就在颠簸。

启动和控制设备，使得主存和设备之间交换数据。

存在一个位来控制CPU状态

在置操作系统之后，操作系统内核程序拥有较高级别的特权，可以使用全部的机器指令。但是，应用程序的权限较低，只能使用指令系统的子集。这是因为应用程序在执行有关资源管理的机器指令时易于导致系统混乱，造成系统或用户信息被破坏，例如，置程序状态字指令将导致处理器占有程序的变更；如果错误地启动设备 I/O 指令，会有多个程序竞争使用设备而导致 I/O 混乱，这些指令只有操作系统才有权使用。因此，在多道程序设计环境中，从资源管理和控制程序执行的角度出发，必须把指令系统中的指令分成两类：特权指令（privileged instruction）和非特权指令。所谓特权指令是指仅供内核程序使用的指令，如启动设备、设置时钟、控制中断屏蔽位、清空主存、建立存储键、加载 PSW 等敏感性操作。内核能够执行全部指令，应用程序只能使用非特权指令。如果应用程序执行特权指令，会导致非法执行而产生保护中断，继而转向操作系统的“用户非法执行特权指令”的异常处理程序进行处理。

4. 处理器状态

把并发使用共享资源并为串行化。从而保障系统状态一致性。

当CPU上电启动的时候，都不是在用户态。

（1）核心态和用户态

处理器如何得知当前是操作系统还是应用程序在其上运行呢？这将取决于处理器状态标志，在执行不同的程序时，根据执行程序对资源和机器指令的使用权限将处理器设置成不同的状态，处理器状态又称处理器模式，可划分为核心态（kernel mode，又称管态（supervisor mode））和用户态（user mode，又称目态）。

当处理器处于核心态时，CPU 运行可信软件，硬件允许执行全部机器指令，可以访问所有主存单元和系统资源，并具有改变处理器状态的能力；当处理器处于用户态时，CPU 运行非可信软件，程序无法执行特权指令，且访问仅限于当前 CPU 上进程的地址空间，这样就能防止内核受到应用程序的侵害。处理器模式位扩展了操作系统的保护权限，意味着在核心态执行的程序比用户态的拥有更多权限去访问主存和执行特权指令，所以模式位被用来区分可信软件和非可信软件。

系统调用本身是通过一个特殊指令，使得CPU产生异常处理。

Intel x86 处理器的状态有 4 种，分别支持 4 个特权级别，0 级权限最高，3 级权限最低。典型的应用是特权级别依次设定为：0 级为内核级，处理 I/O 操作、存储管理并执行其他关键操作；1 级为系统调用级，可以执行系统调用，获得特定的和受保护的程序的服务；2 级为共享库级，可被多个运行进程共享，允许调用库函数，读取但不修改相关的数据；3 级为应用程序级，所受到的保护最少。当然，各个操作系统在实现过程中可以根据具体策略有选择地使用硬件所提供的保护级别，如运行在 Intel x86 上的 Windows 操作系统只使用 0 级和 3 级特权。

（2）处理器模式转换

大多数其他CPU就两个Mode，或者加一个Reall Mode

有两类情况会导致处理器从用户态向核心态转换。一是程序请求操作系统服务，执行系统调用；二是在程序运行时，产生中断或异常事件，运行程序被中断，转向中断处理程序或异常处理程序工作。这两类情况都通过中断机制发生，可以说中断和异常是用户态到核心态转换的仅有途径。当系统中产生中断或异常（如设备中断或非法操作码异常），处理器将做出响应并交换程序状态字，此时会导致处理器从用户态转向核心态，中断事件或异常处理程序的程序状态字中的处理器模式位一定是“核心态”。从用户模式转向内核模式，不存在类似指令或其他方法，否则任

不要太纠结这个所谓的“可信”的概念。在核心态，CPU可以运行许多高级指令，和改变控制寄存器

不同的CPU具体方法不一样。但都差不多。
中断／异常返回指令会把之前都CPU状态
寄存器的值恢复。

何进程都可以轻易转入特权状态，从而使保护机制失效。那么，如何实现核心态到用户态的转换呢？计算机通常提供一条称作加载程序状态字的特权指令（IBM 370 为 LOAD PSW 指令，Intel x86 为 IRET 指令），用来实现从系统（核心态）返回用户态，将控制权转交给应用进程。

2.1.2 程序状态字

计算机如何知道当前处理器处于何种状态？此时能否执行特权指令？操作系统通常引入程序状态字（Program Status Word，PSW）来区分不同的处理器状态。如图 2.1 所示是 IBM 370 系统计算机程序状态字的基本格式。

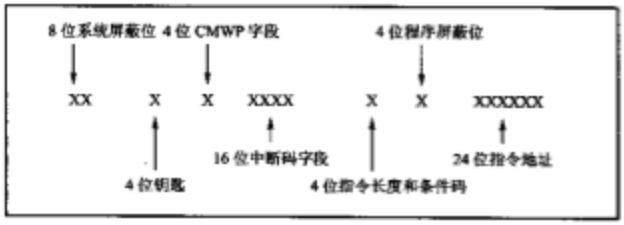


图 2.1 IBM 370 系统程序状态字基本格式

（1）系统屏蔽位（0～7 位）：表示允许或禁止某个中断事件发生，相应位为 0 或 1 时分别表示屏蔽或响应中断。0～7 位依次是 0～6 号通道和外中断屏蔽位。

（2）钥匙（8～11 位）：当未设置存储器保护时，这 4 位是 0；当设置存储器保护时，PSW 中的这 4 位钥匙与欲访问的主存区域的存储锁必须匹配，否则将不能访问。

（3）CMWP（12～15 位）：依次是 PSW 基本/扩充控制方式位、开/关中断位、运行/等待位、用户态/核心态位。

（4）中断码字段（16～31 位）：中断码字段与中断事件相对应，记录当前的中断源。

（5）指令长度（32～33 位）：01/10/11 分别表示半字长指令、整字长指令和一字半长指令。

（6）条件码（34～35 位）：表示指令执行的结果状态。

（7）程序屏蔽位（36～39 位）：表示允许（为 1）或禁止（为 0）程序性中断，自左向右各位所对应的程序性事件是：定点溢出、十进制数溢出、阶下溢，第 39 位备用。

（8）程序计数器（40～63 位）：指明下一条执行指令的绝对地址。

不难看出，PSW 用来指示处理器状态，控制指令的执行顺序，并且保留和指示与运行程序有关的各种信息，主要作用是实现程序状态的保护和恢复。每个正在执行的程序都有一个与其当前状态相关的 PSW，而每个处理器都设置一个硬件 PSW 寄存器，一个程序占用处理器执行时，其 PSW 将占用硬件 PSW 寄存器。

不同处理器的控制寄存器的组织方式不同，多数计算机的处理器现场可能找不到一个具体的 PSW 寄存器，但总会有一组控制寄存器和状态寄存器实际上起到这一作用。为了方便操作

不同的CPU具体的实现不一样。但基本原理和目标是一样的。

系统原理的讨论，本书使用 PSW 和 PSW 寄存器这两个术语。

在 Intel x86 中，PSW 由标志寄存器 EFLAGS 和指令指针寄存器 EIP 组成，均为 32 位。EFLAGS 的低 16 位称为 FLAGS，可以当做一个单元来处理。标志划分为三组：状态标志、控制标志、系统标志。

(1) 状态标志

使得一条指令的执行结果影响其后指令的执行。算术运算指令使用 OF(溢出标志)、SF(符号标志)、ZF(结果为 0 标志)、AF(辅助进位标志)、CF(进位标志)、PF(奇偶校验标志)、AC(地址对齐检查)；串扫描、串比较、循环指令使用 ZF 通知其操作结束。

(2) 控制标志

控制标志包括 DF、VM、TF 和 IF 位。DF(方向标志)控制串指令操作，DF 为 1 时，使得串指令自动减量，即从高地址向低地址处理串操作；DF 为 0 时，串指令自动增量。VM(虚拟 8086 方式标志)为 1 时，从保护模式进入虚拟 8086 模式。TF(步进标志)为 1 时，使处理器执行单步操作。IF(中断允许标志)为 1 时，允许响应中断，否则关中断。

(3) 系统标志

系统标志与进程管理有关，共 3 个：IOPL(I/O 特权级别标志)、NT(嵌套任务标志)和 RF(恢复标志)，用于保护模式。指令指针寄存器 EIP 的低 16 位称为 IP(保护模式使用 32 位)，存放顺序执行的下一条指令相对于当前代码段起始地址的一个偏移量，IP 可以当做一个单元使用。

2.2 中断技术

Interrupt（中断）和Exception（异常）有时不太分。不需要过分纠结。

2.2.1 中断概念

中断机制是现代计算机系统的重要组成部分之一，每当应用程序执行系统调用要求获得操作系统服务、I/O 通道及设备报告传输情况或者产生形形色色的内部和外部事件时，都要通过中断机制产生中断信号并启动内核工作，可以说操作系统是由“中断驱动”的。最初，中断技术仅作为设备向 CPU 报告 I/O 操作情况的一种手段，以免 CPU 因不断轮询设备而耗费时间，中断的出现帮助解决主机和设备的并行性问题。现在中断技术的应用范围越来越广，在计算机运行过程中，许多事件有可能随机发生，如硬件故障、网络通信、人机交互和程序出错等，必须对这些事件及时加以处理，所以，为了请求系统服务，实现并行工作，处理突发事件，满足实时要求，需要打断处理器的正常工作流，为此提出中断的概念。中断(interrupt)是指在程序执行过程中，遇到急需处理的事件时，暂时中止现行程序在 CPU 上的运行，转而执行相应的事件处理程序，待处理完成后再返回断点或调度其他程序执行。例如，从磁带上读入一组信息，当发现读入操作有错时，将产生读数据错中断，操作系统暂停当前的工作，并组织磁带退回重读这一组信息，这样就有可能克服错误。

在不同的计算机系统中，通常有不同的中断源和中断装置，它们有一个共性，即在中断事件

中断是一个重要的概念。涉及：
现场保护，执行中断；恢复现场；

每个CPU对不同的异常（中断）的处理都有具体的规定。
编程时需要仔细阅读CPU手册。

发生后，中断装置能改变处理器内操作的执行顺序。可见中断是现代操作系统实现并行性的基础之一。引入中断机制，操作系统在让应用程序放弃控制权或从应用程序获得控制权时将具有更大的灵活性。

2.2.2 中断源分类

1. 按中断事件的性质和激活方式划分

按中断事件的性质和激活方式可将中断分成两类：强迫性中断和自愿性中断。强迫性中断事件肯定不是正在运行的程序所期待的，是由随机事件或外部请求信息引起的。强迫性中断事件有以下几种。

(1) 机器故障中断 Machine check

机器执行指令过程中硬件可能出现种种事件，例如，电源故障、通路校验错误、主存出错等。

(2) 程序性中断

程序执行过程中可能发生种种例外情况，例如，非法操作码、定点溢出、除数为0、地址越界等。

(3) 外部中断

由计算机系统外部发送中断信号，反映外界对本机的种种要求，例如，时钟中断、控制台中断、它机中断等。

(4) 输入输出中断

来自通道、控制器、设备的中断能够反映I/O操作情况，例如，设备出错、传输结束、启动失败等。

自愿性中断事件是正在运行的程序所期待的，是由于执行"访管指令"而引起的，它表示运行程序对操作系统有某种需求，一旦机器执行访管指令，就会使CPU状态从用户态转向核心态，停止现行程序的执行而转入内核的相应系统调用例程进行处理。例如，要求操作系统协助启动外部设备工作。

这两类中断事件的响应过程略有不同，如图2.2所示。

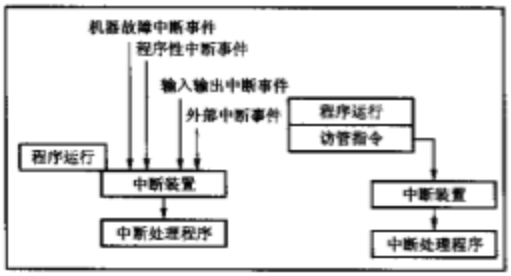


图2.2 按中断事件的性质和激活方式分类

2. 按中断事件的来源和实现手段划分

理解中断的一个重要知识点：
中断切换后，之前的指令都已经完成操作；之后的都没有操作。

OS里，大多数情况下，会支持嵌套中断。在中断处理程序里，会从新开中断。

按中断事件的来源和实现手段可将中断划分为硬中断和软中断两类，如图 2.3 所示。

(1) 硬中断：外中断和内中断

硬中断可划分为外中断和内中断两种。

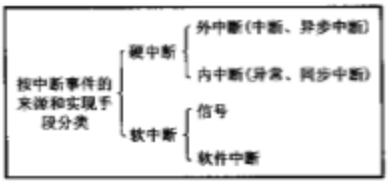


图 2.3 按中断事件的来源和实现手段分类

① 外中断，又称中断或异步中断，是指来自处理器之外的中断信号，包括时钟中断、键盘中断、它机中断和设备中断等。外中断又分为可屏蔽中断和不可屏蔽中断，各个中断具有不同的中断优先级，表示事件的紧急程度，在处理高一级中断时，往往会部分或全部屏蔽低级中断。

② 内中断，又称异常(exception)或同步中断，是指来自处理器内部的中断信号，通常是由于在程序执行过程中，发现与当前指令关联的、不正常的或错误的事件。内中断可被细分为三种：访管中断，由执行系统调用而引起；硬件故障中断，如电源失效、协处理器错误、奇偶校验错误、总线超时等；程序性异常，如非法操作、地址越界、页面故障、调试指令、除数为 0 和浮点溢出等。所有这些事件均由异常处理程序来处理，响应处理器中状态的变化且通常依赖于执行程序的当前现场。内中断不能被屏蔽，一旦出现应立即予以响应并进行处理。

中断和异常之间的区别如下。

① 中断是由与当前程序无关的中断信号触发的，系统不能确定中断事件的发生时间，所以，中断与 CPU 是异步的，CPU 对中断的响应完全是被动的。中断的发生与 CPU 模式无关，既可发生在用户态，又可发生在核心态，通常在两条机器指令之间才能响应中断。一般来说，中断处理程序所提供的服务不是当前进程所需要的，如时钟中断、硬盘中断等，中断处理程序在系统的中断上下文中执行。

current context

异常里可以再发生高级的异常例如，Page Fault里可以发生Machine Check

② 异常是由 CPU 控制单元产生的，源于现行程序执行指令过程中检测到例外。异常与 CPU 是同步的，允许指令在执行期间响应异常，而且允许多次响应异常，大部分异常发生在用户态。异常处理程序所提供的服务通常是当前进程所需要的，如处理程序出错或页面故障，异常处理程序在当前进程的上下文中执行。以 Linux 为例，异常按照错误报告方式分为 4 种：故障(fault)、陷阱(trap)、终止(abort)和编程异常(programmed exception)。此外，中断的嵌套发生是允许的，但异常多为一重，异常处理过程中可能会产生中断，但反之则不会发生。下面是 Intel x86 体系结构中发布的且在 Linux 中使用的部分异常：除法溢出错(0 号，故障)、调试异常(1 号，故障或陷阱)、断点中断(3 号，陷阱)、算术溢出(4 号，陷阱)、边界异常(5 号，故障)、无效操作(6 号，故障)、设备不可用(7 号，故障)、双异常(8 号，故障)、协处理器段溢出(9 号，故障)、任务状态段异常(10 号，故障)、段不存在(11 号，故障)、堆栈溢出(12 号，故障)、一般性保护(13 号，故障)、页故障(14 号，故障)、浮点数错(16 号，故障)和边界异常(17 号，故障)等。

编程异常用于实现系统调用，进程自愿进入核心态以请求系统服务，所以属于主动行为；故障是指程序运行中系统捕获的潜在可恢复的错误，经处理后可返回当前指令再次执行，页面故障是最典型的例子；终止是指致命的不可恢复的错误，如主存芯片发生奇偶校验错误，通常不会返

OS／CPU都是通过一个中断／异常向量表（Vector）来处理的。通常在内存低端的某个部分。

中断和异常的本质就是处理一个突发事件。按照预先定义好的步骤来处理
操作系统启动最开始的工作最重要的就是填写好中断向量表。

回原来的程序而转向内核 abort 例程，有时甚至需要重新启动计算机系统；陷阱是在执行特定的调试指令时触发的，被调试的进程遇到所设置的断点时会暂停等待。

发生故障时所保存的返回指令地址指向触发异常的那条指令，故障处理完毕后会重新执行这条指令。编程异常和陷阱是由于执行访管指令而引起的同步操作，异常将返回至触发异常的下一条指令。

IBM 中大型机上的中断源使用上述第一种分类方法，Intel x86 机上的中断源则采用上述第二种分类方法。

signal, soft irq

(2) 软中断：信号和软件中断

外中断与内中断（中断和异常）要通过硬件设施来产生中断请求，它们都是硬中断。与其相对应，不必由硬件产生中断源而引发的中断称为软中断。软中断利用硬中断的概念，采用软件方法对中断机制进行模拟，实现宏观上的异步执行。

软中断可分为两种。“信号”是一种软中断机制，信号的发送者相当于中断源，而信号的接收者必然是一个进程（相当于 CPU）。“软件中断”是另一种软中断机制，其第一个典型应用例子是 Linux 中的 bottom half，它的升级就是 Linux 中复杂、庞大的软中断子系统 softirq 机制；第二个典型应用例子是 Windows 中由内核发出的 Dispatch/DPC 和 APC 等中断，用于启动线程调度，延迟过程调用和异步过程调用的执行。

(3) 硬中断与软中断的类比

上述几种中断的通常用法如下。

① “中断”（硬中断）用于外部设备对 CPU 的中断（中断正在运行的任何程序），转向中断处理程序执行。

② “异常”（硬中断）因指令执行不正常而中断 CPU（中断正在执行这条指令的程序），转向异常处理程序执行。

③ “软件中断”（软中断）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。

④ “信号”（软中断）用于内核或进程对某个进程的中断，向进程通知某个特定事件发生或迫使进程执行信号处理程序。

中断机制与信号机制可以进行类比：

① 两者在概念上是一致的，进程接收到信号与 CPU 接收到中断是相似的。进程可以向其他进程发送信号，而 CPU 也可以向其他 CPU 发出中断请求。

② 两者都是“异步”的，CPU 和进程在工作过程中，并不知道中断或信号何时产生，更不需要停顿下来等待中断或信号的发生。硬中断在一条指令执行结束时安排查询，信号则在异常处理末尾或时钟中断处理结束之前进行查询。

③ 两者在实现上均采用“向量表”，中断机制设置“中断向量表”，以向量号为索引查找中断向量，然后转入中断处理程序或异常处理程序。信号机制采用“信号向量表”，通过信号编号查表，从而找到并转入相应的信号处理程序。

感知和处理的时机是理解signal最重要的部分。
在系统调用返回，或者中断处理结束的时候，操作系统坚持一个进程是否有signal

[注释:] 中断 / 异常处理是掌握一个CPU重要的部分。不同的CPU实现都有细微的区别，但本质都类似。 保护现场，处理中断 / 异常，根据情况决定是恢复现场执行，还是reboot CPU。

信号（signal）可用来两个进程通信。发送方“kill”系统调用给另外一个进程发送一个信号。 可捕获的signal的处理程序是用户进程可以自己提供。

④ 两者均有“屏蔽”设施。中断机制可直接对某些中断源加以屏蔽；信号机制中也存在相似的手段，可以设置信号字段位屏蔽，使进程对于发来的信号不予理睬。

中断机制与信号机制之间的区别是：前者由硬件和软件相结合来实现，而后者则完全由软件实现；中断向量表和中断处理程序（全部由系统提供）均位于系统空间，而信号向量表虽然处于系统空间，但信号处理程序往往由应用程序提供，并在用户空间执行。

也可以把硬中断和软中断 BH 进行类比。

① 数组 bh _ base[ ]相当于硬件中断机制中的数组 irq _ desc[ ]，不过 irq _ desc[ ]中的每个元素代表一个中断服务程序队列，而 bh _ base[ ]的每个元素只代表一个 BH 函数；

② bh _ active 在概念上相当于硬件的“中断请求寄存器”，而 bh _ mask 则相当于硬件中的“中断屏蔽寄存器”；

③ 需要执行一个 BH 函数时，就通过 mark _ bh()将 bh _ active 中的某一位设置成 1，相当于中断源发出（软件）中断请求，而所设置的具体标志位则类似于“中断向量”；

④ 如果相当于“中断屏蔽寄存器”的 bh _ mask 中的相应位也是 1，即系统允许执行这个 BH 函数，那么就会在每次执行完 do _ IRQ()中的中断服务程序后，及每次系统调用结束后，在函数 do _ bottom _ half()中执行相应的 BH 函数，而 do _ bottom _ half()则类似于 do _ IRQ()。

除了操作系统中的一部分低层硬件异常由内核异常处理程序来处理之外，大部分异常均转化为信号（软中断）再处理。由于异常与当前运行进程紧密相关，每当执行指令产生异常事件时，可通过信号处理程序向当前运行进程发送一个信号。

关于硬中断或软中断处理的延迟问题，一般来说，CPU 在接收和响应硬中断之后会立即调用中断处理程序或异常处理程序；对于所接收的信号或软件中断，由于此时进程未必占有处理器运行或内核正在执行敏感性操作，通常会有一定的时间延迟，内核或相关进程在适当的时刻才能加以处理。信号和软件中断虽然都由软件产生，并由软件处理，但是它们的中断来源、适用场合、实现手段并不相同。

2.2.3 中断和异常的响应及服务

无论是外部产生的中断，还是内部出现的异常，或程序执行系统调用自愿访管；无论中断源是被动的，还是主动的，CPU 的响应过程基本上是一致的，这就是：在执行完当前指令后，或在执行当前指令的中途，根据中断源所提供的“中断向量”，在主存中找到相应服务程序的入口地址并调用此服务程序。中断向量由硬件或操作系统内核预先分配和设置，系统调用所对应的向量则在访管指令中给出，各种异常的向量在 CPU 的硬件结构中预先规定，这样，不同情况就因中断向量的不同而区分开来。因此，中断和异常可以作为统一模式加以实现，国内许多教材一律将其归入“中断机制”就是这个道理。

那么，处理器如何响应中断和异常，操作系统如何转到中断处理程序和异常处理程序执行呢？先看中断，它主要是由外部设备、时钟部件或其他机器发出的，发现中断源并产生中断的硬件称为中断装置，这些硬件包括中断逻辑线路和中断寄存器，当前指令执行结束后，CPU 会检查

BottomHalf在Linux里更多是一个对硬件中断或者异常的一个优化措施。从而使得硬件中断可以迅速完成，重新开中断。BH的部分都是硬件中断开开启的，要确保不会有重入的不一致性的问题。

有的CPU是跳转到一个相同的入口，然后查表，确定中断源；有的CPU是直接一步到位直接跳转到中断源对应的向量入口。

中断寄存器是否有中断事件发生，若无中断信号或中断信号被屏蔽，继续执行程序的后续指令，否则将暂停执行当前程序，转向内核的中断处理程序。再看异常，它是在执行指令时，由于指令本身的原因发生的，指令的实现和执行逻辑一旦发现异常情况，便转向内核的异常处理程序，这个由硬件对中断和异常事件做出反应的过程称为中断响应。

迄今为止，所有计算机系统都采用硬件和软件，即硬件中断装置和软件中断处理程序/异常处理程序相结合的方法实现中断/异常处理。一般来说，中断/异常的响应需要顺序做4件事。

(1) 发现中断源

在中断未被屏蔽的前提下，硬件发现中断/异常事件，并由CPU响应中断/异常请求。当发现多个中断源时，将根据预定的中断优先级先后响应中断请求。

(2) 保护现场

暂停当前程序的运行，硬件将中断点的现场信息(PSW)保存至核心栈，使得中断处理程序/异常处理程序在运行时，不会破坏被中断程序中的有用信息，以便在中断处理结束后返回原程序继续运行。

(3) 转向中断/异常事件的处理程序

这时处理器状态已从用户态切换至核心态，中断处理程序/异常处理程序开始工作。

(4) 恢复现场

当中断处理结束后，恢复PSW，重新返回中断点以便执行后续指令。当异常处理结束后，返回点会因异常类型而异。大部分应用程序指令执行出错时，异常处理会结束进程，不可能回到原程序；如果是执行访管指令，则异常处理完成后返回这条访管指令的下一条指令；对于页面故障，异常处理结束后会返回发生异常的那条指令重新执行。

如图2.4所示是IBM中大型机的中断响应过程。如果把被中断程序的PSW称为旧PSW，而把中断处理程序的PSW称为新PSW，如何实现新旧PSW之间的交换呢？系统通常为每种中断都开辟主存的固定单元以存放新的和旧的PSW，主存中开辟专用的双字单元(用十六进制数标出)，用于存放各类中断的旧的和新的PSW(分别为旧的和新的外中断、访管中断、程序中断、机器故障中断和I/O中断)，CPU中还有硬件PSW寄存器保存运行程序的现行PSW。当响应中断时，由硬件执行①把中断码装配至现行PSW中，然后执行②把现行PSW保存到中断类型对应的旧PSW单元，再执行③把相应的新PSW加载至现行PSW，这样就会引出相应的中断处理程序。中断事件处理结束后，如果执行④便可从断点返回，继续执行被中断的程序。

Linux中断机制在保护模式下的实现采用中断描述符表(Interrupt Descriptor Table，IDT)，此表包含256个中断描述符，与中断或异常一一对应。描述符的作用是把程序控制权转给中断/异常服务程序，每个描述符均占用8 B，通过它就能找到服务程序的起始地址、属性以及程序特权级别等信息。IDT的位置由硬件中断描述符表寄存器IDTR指定，它是一个48位的寄存器，高32位是IDT的基址，低16位限定IDT的长度。

每个中断/异常都有一个向量号，其取值范围在0～255之间，是中断在IDT中的索引。每个中断/异常均有其相应的处理程序，在使用中断之前，必须在IDT中注册以保证发生中断时能

CPU内部的控制寄存器。

在内存中事先放好的IDT。

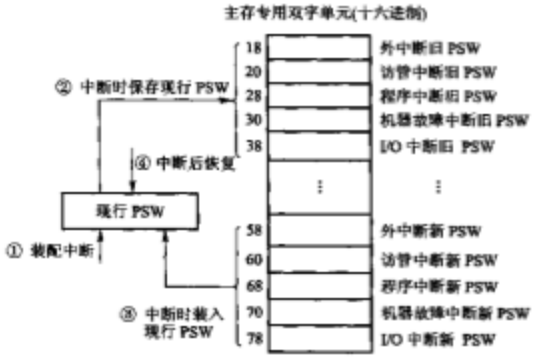


图 2.4 IBM 中大型机的中断响应过程

找到相应的处理程序。在系统初始化时创建 IDT,向量号的使用情况为:0－31 号对应异常或硬件非屏蔽中断,32～47 号分配给硬件可屏蔽中断,48～255 号分配给软件中断,其中,128 号(0x80)中断向量用来实现系统调用。

2.2.4 中断事件处理

Linux为例子

1. 中断和异常的一般处理过程

中断事件的处理比异常事件的处理复杂得多,这是因为:对于当前进程而言,中断是异步事件,中断处理程序工作在核心态的中断上下文中,不允许被阻塞;中断处理程序应能够为共享同一根中断请求线的多台设备服务;中断处理程序的执行时间要尽可能的短,以缩短关中断时间,为此,中断处理被分成上半部分和下半部分,这也增加了中断处理的复杂性。内核处理所有中断的过程是相同的,其执行流程大致如下。

(1) 当设备发出中断请求时,中断信号由设备发送到中断控制器,中断控制器将 IRQ 号转换成中断向量号并传送给 CPU;CPU 响应中断之后,自动保护现场,把 PSW(EFLAGS、CS 和 EIP)、中断向量号、用户栈段寄存器 SS 和栈指针 ESP 压入核心栈,并根据中断向量号查找 IDT,生成指向中断描述符表的指针,找到中断服务程序 IRQn _ interrupt 的地址,开始执行中断处理程序;

(2) 进入中断公共代码段 common _interrupt 执行 SAVE _ ALL,将硬件未保护的所有寄存器内容保护到核心栈;

(3) 调用 do _ IRQ( )函数,对中断控制器进行确认,设置中断源状态等;

(4) 根据 IRQ 为发出中断请求的设备提供服务,调用服务程序执行相关的中断处理任务,标记中断下半部分;如果有多个设备共享此中断 IRQ,需要执行此中断线上所有设备的中断服

Intel x86 CPU特有的。其他CPU有区别。但大致相同。

ret_from_intr和ret_from_exp都定义在Linux的entry.S里。离开的时候，都是检查signal和是否需要调度。

务程序；

(5) 检查 softirq _ active 和 softirq _ mask 是否有标记的下半部分，若有则调用 do _ softirq() 执行之；

(6) 跳转至 ret _ from _ intr 退出，恢复发生中断之前的现场。

异常处理可分为三步：一是当前进程执行指令产生异常；二是进入异常处理程序并执行之；三是从异常处理程序返回。产生异常之后，将触发0－31 号中断向量所对应的异常，并自动转向异常处理程序的公共入口以执行下列操作。

(1) 将硬件错误代码和异常向量号存入当前进程的 PCB(Process Control Block，进程控制块)中；

(2) 判别异常是产生于核心态还是用户态，对于前者，将直接转向内核预定义服务程序进行处理，未被处理的核心态异常是操作系统的致命错误；对于后者，终止当前进程的运行，调用 force _ sig()函数向当前进程发信号；

(3) 从 ret _ from _ exception 返回用户空间时，将检查进程是否有信号等待处理，若有则根据信号类型调用相应的函数进行处理。

当然，也有些异常处理是很特殊的，典型的例子是页面故障异常，它是实现分页式虚存管理的硬件支撑，由系统来处理这种异常，异常处理程序最终转向 do _ page _ fault()执行页面调度。

中断和异常的处理过程大致相同，但在产生异常时，硬件并不清除中断标志位，此时还允许外部硬件中断；而在产生中断时，硬件将立即清除中断标志位，以禁止其他硬件中断。

2. 硬件故障中断

一般来说，这种中断事件是由硬件故障导致的，如电源故障、主存故障、设备故障，排除这种故障需要人工干预，如复位、设置或替换。中断处理程序所能做的事情一般是保护现场，停止设备工作，停止处理器运行，将故障信息向操作员报告，并对故障所造成的破坏进行估计和恢复。

3. 程序性中断

应用程序的错误有以下几类：一是语法错误，这可由编译程序发现并报错；二是逻辑错误，可由测试程序发现并报错；三是程序运行过程中所产生的异常，例如，定点溢出、阶码下溢、除数为0 等。不同的用户往往有不同的处理要求，借助于信号机制，操作系统可将所捕获的这类中断事件原封不动地转交给应用程序自行处理。

4. I/O 中断

I/O 中断的处理原则如下。

(1) I/O 操作正常结束

把等待传输的进程均设置为就绪态，查看是否有其他进程等待设备或通道，若有则释放之。

(2) I/O 操作发生故障

先向设备发命令索取状态字，然后分析产生故障的确切原因，再采用复执方式或请求人工

通常都是禁止中断发生。所以，中断处理程序需要迅速处理一些保护，然后重开中断。Linux的BottomHalf机制就是为了
使得中断处理不要时间太长而设计的。

系统调用就是显示（Explicitely）的调用特权指令，从而CPU进入Exception模式。系统调用传进来的参数通常是通过GPR寄存器，按照一个约定。

干预。

(3) I/O 操作发生异常

分析情况，采取相应的措施，向操作员报告，如通知操作员换卷、装纸等。

(4) 设备报到或设备结束

表示有设备接入可供使用或设备断开暂停使用，操作系统应修改系统数据中相应设备的状态。

5. 访管中断

这类中断是由程序执行访管指令而引起的，表示当前运行程序对操作系统功能的调用，可看做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，前者表示此指令是访管指令，后者则表示具体的访管要求。IBM 出品的机器在执行访管指令时，将访管参数作为中断字并入 PSW，同时送入主存中的指定单元，操作系统的访管中断处理程序分析访管参数、进行合法性检查后，按照访管参数的具体要求进行相应的处理。在 Intel x86 机器上，当 CPU 接收一个中断请求时，由设备接口发出唯一的"中断向量"，Linux 将其作为系统调用号装入 EAX 寄存器，其他参数装入 EBX、ECX、EDX、ESI 和 EDI 寄存器中。访管参数超过 6 个的情况并不多见，寄存器存放参数在用户空间的地址，以此为指针来传递参数，然后执行访管指令 int 0x80 陷入操作系统。返回值则放入 EAX 寄存器中。

不同的访管参数对应不同的服务要求，就像机器指令的不同操作码对应不同的动作要求一样，系统调用机制在本质上通过特殊的硬指令和中断机制来实现。不同机器其系统调用命令的格式和功能号的含义不尽相同，但是任何机器的系统调用都有如下共性处理流程。

(1) 程序执行访管指令，并通过适当的方式指明系统调用号；

(2) 通过中断机制进入访管中断处理程序，现场保护到核心栈，按功能号实现跳转；

(3) 通过系统调用入口表找到相应功能服务例程的入口地址；

(4) 执行相应的服务例程，正常情况下在结束后返回系统调用的下一条指令继续执行。

Timer 6. 时钟中断 需要做一个系统调用的查表。。。

时钟是操作系统进行调度工作的重要工具，如维护系统的绝对日期和时间、让分时进程按时间片轮转、让实时进程定时发送或接收控制信号、系统定时唤醒或阻塞进程、对用户进程记账、测量系统性能等，利用定时器能够确保操作系统在必要时获得控制权，陷入死循环的进程最终会因时间片耗尽而被迫让出处理器。

Counter 时钟可分为绝对时钟和间隔时钟。通常使用一个硬件时钟，它按照固定周期发出中断请求，Linux 早期版本每 10 ms(频率 100 Hz)时钟中断 1 次，从 Linux 2.5 版开始，时钟中断频率提高至 1 000 Hz，即每 1 ms 产生 1 次时钟中断。系统设置一个绝对时钟寄存器，定时地把此寄存器的内容加 1。如果这个寄存器的初始内容为 0，那么，只要操作员告诉系统开机时的年、月、日、时、分、秒，就可以据此推算当前的年、月、日、时、分、秒。

间隔时钟在每个时间切换点将间隔时钟寄存器的内容减 1，通过程序设置此寄存器的初值，当间隔时钟寄存器的内容为 0 时，就产生间隔时钟中断，等同于闹钟的作用，意味着预定时间已

int 0x80是Intel CPU的sys call指令。有的cpu的指令是sc，有的就是syscall。形式不同，目的都是一个：系统调用。从而一个用户态的进程能陷入核心。

通过Counter，可以控制中断的精度。特别是工业控制系统。

到。操作系统经常利用间隔时钟执行调度控制。

时钟硬件的作用是按已知时间间隔产生中断，与时间有关的其他各项任务必须由软件完成，要建立操作系统和应用程序所使用的高层时钟和定时器服务。

(1) 绝对时钟服务

绝对时钟服务提供以下函数：update _ clock()更新当前时间，每个时钟滴答时刻都会调用此函数，标识从某个已知起始时刻起的时钟滴答的数目；get _ time()返回当前时钟值，既可以是整数值，也可以经过换算返回"年、月、日、时、分、秒"；set _ clock()把当前时间设置为新的时钟值。

(2) 间隔定时器

进程可以被延迟或阻塞，直至被间隔定时器的中断信号唤醒。间隔定时器提供以下函数。

① delay(tdel)把调用进程阻塞由参数 tdel 所指定的时间长度，进程保持阻塞直至本地时间到达发生进程阻塞时的当前时间 + tdel 的时刻。

② set _ timer(tdel)硬件间隔定时器被设置为初始递减值 tdel，当此值变为 0 时，将产生间隔时钟中断，调用 timeout()函数进行处理。

利用间隔定时器，函数 delay(tdel)可实现如下：

```
delay(tdel) {
    set _ timer(tdel); //硬件间隔定时器设置为 tdel
    P(delsem);         //进程执行 P 操作而发生阻塞，等待中断
}                      //delsem 为二元信号量，其初值为 0
timeout() {
    V(delsem);         //中断产生，执行 V 操作以释放进程
};
```

POSIX的Timer机制，例如。可以产生许多定时器的组合。

(3) 逻辑定时器

硬件间隔定时器可供所有进程共享，当存在多个进程在一个时间段内都需要延迟执行时，操作系统将按照进程阻塞时间的升序排成延时队列，队首指针可放在实时时钟的描述数据结构中，相当于为每个进程创建逻辑定时器。为此系统需要提供以下函数：

① tn = create _ ltime()：创建逻辑定时器，在 tn 中存放返回标识符。

② destroy _ ltime(tn)：撤销 tn 所标识的逻辑定时器。

③ set _ ltime(tn,tv)：将 tv 值装入逻辑定时器 tn 中，当其值为 0 时，产生时间到中断。

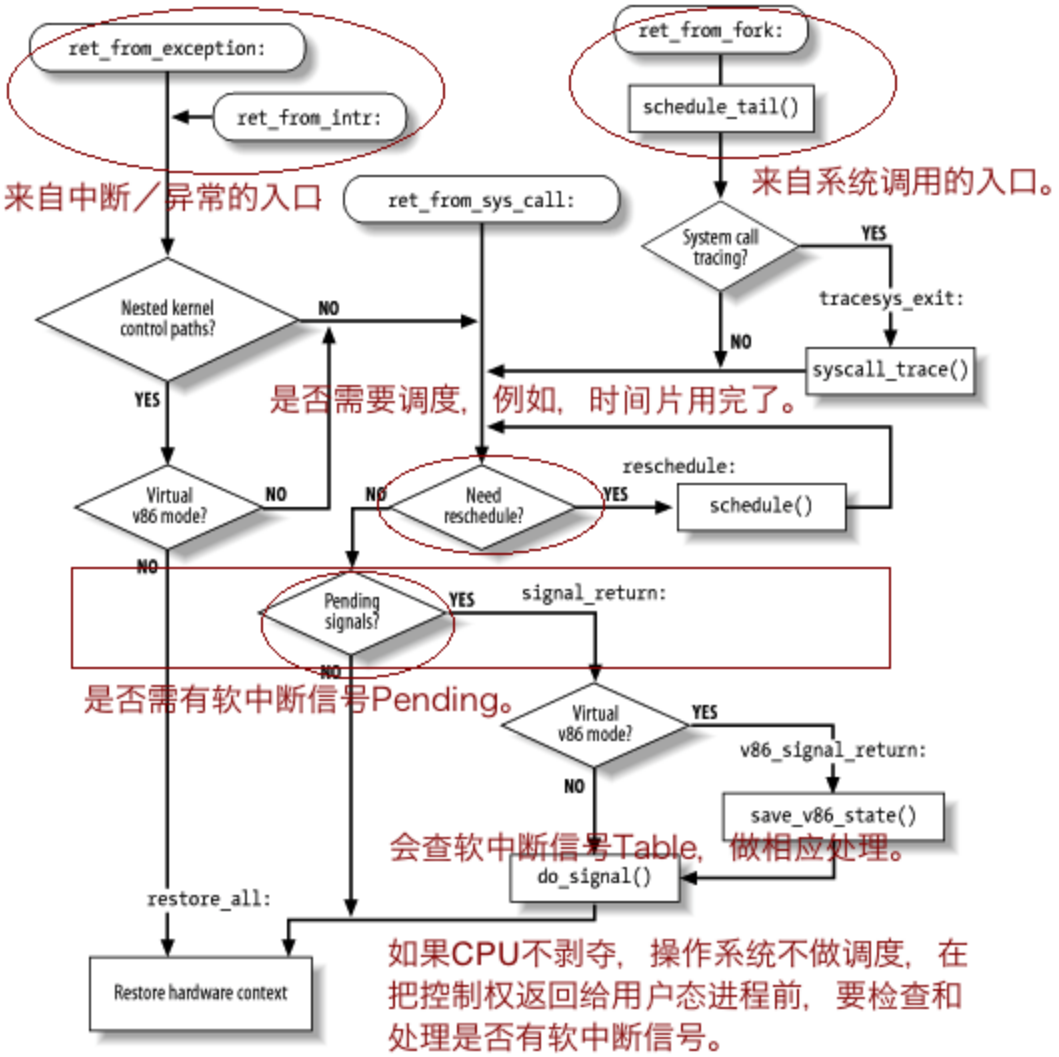
使用硬件间隔定时器实现进程的逻辑定时器的常用方法有以下两种。

① 使用带有绝对唤醒定时器的优先级队列

此方法使用绝对时钟和间隔定时器，被阻塞进程的唤醒时间保存在定时器的优先级延时队列中，每个队列元素都是三元组(p,tn,wakeup)，其中，p 是进程标识，tn 是进程的逻辑定时器标识，wakeup 是未来时间值(即进程阻塞结束的时刻)。优先级延时队列对元素进行排序，wakeup 值最小的进程其优先级最高。图 2.5 上半部分是定时器优先级延时队列的例子，为了表达清晰

有的CPU的Counter是填一个数，每个时钟clock减一下，到0，产生中断。有的CPU的Counter是从0开始，每次加，到了threshold value产生中断

[注释:] 信号在Unix中比较绕，是异步的通知，进程之间通过互相发kill（ ）。也可以是kernel发给一个进程。有的signal不能被捕获，进程必须消亡；有的可以自己接管，接着运行。看是否有signal的时机需要很清楚：异步方式：在系统调用，中断返回的时候。



[注释:] 如图所示， 操作系统从中断，异常，或者系统调用进来，要结束的时候，都存在一个检查点（checkpoint），看看是否需要调度，是否需要处理软中断（signal）。

因为中断处理程序会来检查和调整这个队列。

起见，元素中省略了逻辑时钟标识，进程 P1 希望在标准时刻 115 被唤醒，进程 P2、P3、P4 以此类推。图 2.5 中本地时钟的当前时间为 103，间隔时钟的当前值为 12，下一个中断将在 12 个时间单位后产生，即在时刻 115 产生。

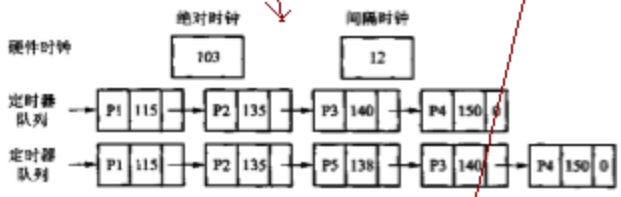


图 2.5 绝对唤醒定时器队列

需要阻塞 tdel 延时值的进程进入定时器优先级延时队列的算法如下。

步骤 1：使用 get _ time( )函数，利用计算机本地时钟来获得当前时间；

步骤 2：wakeup = 当前时间 + tdel，用以确定新元素移入队列的时刻；

步骤 3：如果新元素的唤醒值小于队列第一个元素的 wakeup 值，则新元素移入队列头部，同时使用 set _ timer(tdel)将硬件间隔定时器设置为新的延时间隔 tdel；

步骤 4：否则，按照新元素的唤醒值将其移入队列中的适当位置。

上述算法必须在临界区执行，设置间隔时钟的过程中禁止产生定时器中断。考察图 2.5 下半部分的例子，假设新进程 P5 发出函数调用 set _ ltimer(tn,35)，由于它所希望的唤醒时刻为 103 + 35 = 138，则在新元素移入 P3 元素前面之前，间隔时钟的值保持不变。但是如果调用函数 set _ ltimer(tn,5)，所创建的新元素将成为此队列的当前头部，且间隔时钟被设置为 5。

至此，delay(tdel)函数可利用逻辑定时器来实现，这与硬件间隔时钟的效果相同，它使用函数 set _ ltimer(tn,tdel)为逻辑定时器 tn 装载 tdel 值，再将调用进程阻塞在 P(delsem)操作上。当硬件间隔时钟到达值 0 时，时钟中断处理程序为定时器优先级延时队列的第一个元素服务，移出此元素，并用 V(delsem)唤醒相应的进程，而且也为硬件间隔定时器设置新值，此值由队列中下一个元素的 wakeup 值减去当前时间(由 get _ time( )函数获取)而得到。

② 使用带有时间差值的优先级队列

如果不使用绝对时钟，仅利用间隔时钟也可以实现定时器优先级延时队列，其基本思路是：记录相邻元素的唤醒时间之差值，而不记录唤醒的绝对时间。图 2.6 上半部分给出实现这种定时器延时队列的例子，进程 P1 在队首，12 个时间单位后，间隔时钟值减为 0，进程 P1 被唤醒。进程 P2 会在 P1 之后 20 个时间单位被唤醒，依此类推。这样，时钟中断处理程序的任务很简单，把定时器队列的队首移走，硬件间隔时钟被设置为下一个元素中所存储的延时值。

set _ ltimer(tn,tdel)会变得复杂，其任务是查找左、右两个元素，在它们之间插入新元素，这个工作是通过累加元素中的差值来完成的。从间隔时钟的当前值开始，直至到达 tdel 值；然后，把右元素中当前存储的差值分为两部分：一部分被存储在新元素中，并且代表左元素与新元素之

millisecond (ms): 0.001 second
microsecond (us): 0.000 001 second
nanosecond (ns): 0.000 000 001 second
picoseconds (ps): 0.000 000 000 001 second

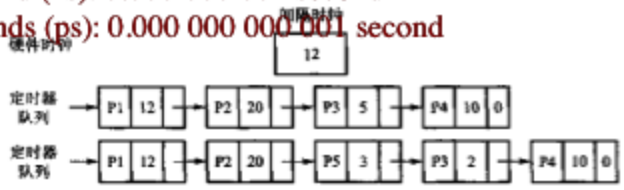


图 2.6 带有时间差值的定时器队列

间的差值，另一部分被存储在右元素中，代表新元素与右元素之间的差值。例如，如果此时进程 P5 执行函数 set _ ltimer(tn,35)，P5 的元素被插入进程 P2 和 P3 的元素之间，因为 P5 的唤醒时间位于这两个进程的累加元素中的差值 32 和 37 之间。进程 P3 中最初的差值(5)被分成两部分 3 和 2，其中，3 被存储在 P5 对应的新元素中，2 被存储在右邻进程 P3 的元素中。

CPU的频率与时钟中断的频率是不一样的。CPU可以是2G；时钟中断通常是100M。
100M意味着（1/100M）＝0.01微秒（us）＝10纳秒（ns）跳一次。

2.2.5 中断优先级和多重中断

1. 中断优先级

中断是随机发生的，在计算机执行的每一瞬间，都可能有几个中断事件同时出现。例如，发生机器故障中断的同时可能产生设备中断请求，这时，中断装置如何响应这些同时发生的中断呢？一般来说，以不发生中断丢失为前提，把紧迫程度相当的中断源归为同一级别，紧迫程度差别大的中断源归于不同级别，级别高的中断有优先获得响应的权利。中断装置所预设的响应顺序称为中断优先级。优先级是按照中断请求的轻重缓急程度，若得不到及时响应将造成计算机出错的严重程度来界定的。其重要性在于，如果系统正在执行某优先级的中断服务程序，那么只有更高优先级的中断请求才能中断此服务程序。例如，机器故障中断表明已经产生硬件故障，对计算机和当前任务造成的影响最大，因而其优先级最高；至于设备中断事件，其优先级可降低，这样做只会推迟一次 I/O 启动或推迟处理一个 I/O 中断事件，对系统正确工作的影响不大；同样是设备，可设定高速设备的优先级高，慢速设备的优先级低，以提高高速设备的利用率。

当某一时刻有多个中断源或中断级提出中断请求时，中断系统如何按照预先规定的优先顺序予以响应呢？可以使用硬件和软件的方法，前者根据排定的优先级顺序做一个硬件链式排队器，当产生高一级的中断事件时，应该屏蔽比它优先级低的所有中断源；后者编写一个查询程序，依据优先级顺序从高到低进行查询，一旦发现有中断请求，便转入相应的中断事件处理程序入口。

在 IBM 系列机中，硬件排定的中断优先级响应顺序从高到低是：机器校验中断、自愿性中断、程序性中断、外部中断、I/O 中断；在 Intel x86 体系结构中，最多允许有 256 个中断或异常信号，这些信号分为 5 类，硬件排定的中断和异常优先级响应顺序从高到低是：复位、异常、软件中断、非屏蔽中断和可屏蔽中断。

时钟中断源，Counter（间隔时间中断）源通常是不一样的。
有的是通过CPU的主频分频过来；有的是通过接Bus（总线）的clock。
和总线的Clock通常是不一样的。

被屏蔽的中断会被pending。一旦开了相应级别的中断，会别立刻做中断检查点（例如指令的最后一个执行周期）检查，并执行。

需要注意的是，中断优先级只是表示中断装置响应中断的次序，在系统设计时已经确定并且不能被改变；操作系统在处理所响应的中断时也存在先后次序的问题，中断处理顺序和响应顺序未必一致，也就是说，先响应的中断可滞后处理。

2．中断屏蔽

计算机均配置可编程中断控制器，CPU 可以通过指令设置可编程中断控制器的屏蔽码。中断屏蔽是指产生并提出中断请求后，CPU 允许响应或禁止响应的状态位，大部分中断源均有对应的中断屏蔽位，当被中断装置接收并产生一个中断请求后，如果相应的中断屏蔽位复位，暂时不能被 CPU 响应，需要等待直至中断屏蔽位置位，被屏蔽的中断才能被 CPU 响应并获得处理。某些中断是不能被禁止的，例如，电源断电中断、自愿访管中断就不能被禁止。

中断屏蔽位的作用是暂时禁止对某些中断做出响应，例如，处理某优先级中断时，防止同级或低级中断的干扰，或在执行不可分割操作及临界区时，需要防止被中断事件打断。此外，中断屏蔽位还能协调中断响应与中断处理之间的关系，保证高级中断处理能够打断低级中断处理，反之不然。

在当前 PSW 中通常已设置某些中断屏蔽位，当中断屏蔽位置位(1)时，允许响应相关的中断；当中断屏蔽位复位(0)时，禁止响应相关的中断。有了中断屏蔽功能，能够增加中断排队的灵活性，采用编程方法在某段时间内屏蔽一些中断请求，可以改变中断的响应顺序。

中断屏蔽方式在计算机设计时即决定，有的采用整级屏蔽方式，有的采用单个屏蔽方式。在 IBM 370 系统中，使用 PSW 的 0—7 位作为系统屏蔽位整级屏蔽通道及外中断，但其 36—39 程序屏蔽位可单个屏蔽程序性中断；在运行 Windows 的 Intel x86 系统中，先把所有单个中断事件排定一个中断请求优先级，称为处理器 IRQL，当内核线程运行时，可以提高或降低处理器 IRQL，如果中断源的 IRQL 等于或低于当前中断优先级，则此中断被屏蔽，直至内核降低 IRQL。

3．多重中断事件的处理

在计算机系统的运行过程中，可能同时出现多个中断，或者前一个中断尚未处理完，接着又发生新的中断，于是 CPU 暂停正在运行的中断处理程序，转而执行新的中断处理程序，这就叫做多重中断或中断嵌套。一般来说，优先级别高的中断具有打断优先级别低的中断的中断处理程序的权利，反之却不允许优先级别低的中断干扰优先级别高的中断的中断处理程序的运行。系统如何处理多重中断呢？这是操作系统的中断处理程序所必须解决的问题。

对于多重中断，可能是同一优先级的不同中断，也可能是不同优先级的中断。对于前者，通常由同一个中断处理程序按自左至右的顺序逐个处理并清除之；对于后者，可区分不同的情况做如下处理。

(1) 串行处理

在运行一个中断处理程序时，禁止再次发生中断，这可以通过屏蔽中断来实现。例如，在执行 I/O 中断处理程序时，可以屏蔽外中断或其他 I/O 中断，甚至所有中断。这种方法简单易行，所有中断都严格地按顺序串行处理，但并未考虑相对优先级和时间限制方面的要求。

本质上是操作系统做中断部分是否支持重入（reentry）的问题。每次嵌套要考虑好堆栈，控制寄存器和一些独享的
数据结构的保护。否则回不去了。

处理好堆栈，控制寄存器和一些non volatile的寄存器问题。就会安全。

(2) 嵌套处理

对于有些必须处理且优先级更高的中断事件，采用屏蔽方法似有不妥，因此，往往允许在运行某些中断处理程序时，仍然能够响应中断，这时，系统应负责保护被中断的中断处理程序的现场，然后转向处理新中断事件的中断处理程序，以便在处理结束时可以返回原来的中断处理程序继续运行。操作系统必须预先规定，赋予每类中断以一定的优先级，允许高优先级中断打断低优先级中断的中断处理程序；还要规定嵌套的最大重数，嵌套的重数视系统规模而定，一般以不超过 3 重为宜，过多的"嵌套"会增加不必要的系统开销。

(3) 即时处理

在运行中断处理程序时，如果出现任何程序性中断事件，在一般情况下，表明此时中断处理程序有例外，应对其立即响应并进行处理。

2.2.6 Linux 中断处理

1. Linux 内核处理流程

如图 2.7 所示是 Linux 内核处理流程，在用户态执行的进程，因执行系统调用，或者因产生中断或异常，促使 CPU 转换为核心态，内核开始工作。假定进程在用户态执行应用程序，并且进行系统调用，那么，进程会按照调用号自陷至内核代码中的服务函数入口点，此函数由内核任务执行，执行结束后，返回至内核 ret_from_sys_call 代码来完成一组标准任务，分派累积的系统工作，例如，下半部分处理、信号处理以及调度其他任务。系统调用结束时，进程处于 TASK_RUNNING 态(就绪态)，一旦 CPU 可用时进程就投入运行。如果内核无法立即完成进程所请求的工作，此进程将处于 TASK_INTERRUPTIBLE 或 TASK_UNINTERRUPTIBLE 状态，如果进程被阻塞，或时间片耗尽，那么，调度程序就会选择新进程。

当中断事件发生时，正在运行的进程执行完当前指令之后，CPU 响应中断，转入相应的中断服务例程(Interrupt Service Routine，ISR)处理，区分两类中断事件：快中断和慢中断，两者的主要区别如下。

(1) 处理慢中断之前需保存所有寄存器的内容，而处理快中断仅保存那些被常规 C 函数修改的寄存器。

(2) 在处理慢中断时，通常不屏蔽其他中断信号，而处理快中断时会屏蔽其他中断。

(3) 慢中断处理完毕后，通常不返回被中断的进程，而是转向调度程序重新进行调度，调度结果未必是被中断的进程继续运行(属于抢占式调度)。而快中断处理完毕后，通常会恢复现场，返回被中断的进程继续执行(属于非抢占式调度)。

慢中断要做很多工作，如果全部在屏蔽中断状态下来完成，则不能及时响应中断处理期间到达的其他中断信号，于是引入中断下半部分的概念。

2. 下半部分处理概述

中断处理程序的特点是：它以异步方式运行，有可能打断关键代码的执行，甚至打断其他中断处理程序的执行；它在屏蔽中断状态下运行，最坏的情形会禁止所有中断；它要对硬件进行操作，对

BottomHalf。是Linux引入的一种机制。
对中断处理程序做性能优化的一种方法。

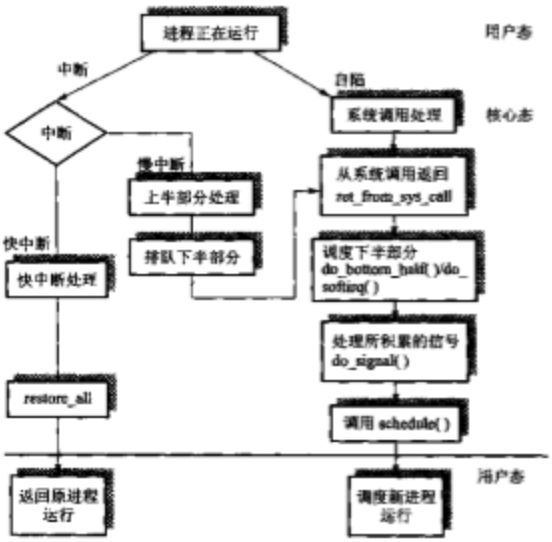


图 2.7 Linux 内核处理流程

于时限有很高的要求;它在中断上下文中运行,故不能被阻塞。凡此种种都要求中断处理程序的执行越快越好,因为缩短屏蔽中断的时间对于系统的响应能力和性能都至关重要。于是,Linux 中断服务程序被分成上半部分(top half)和下半部分(bottom half)两个处理函数。每当上半部分函数接收到中断,便立即开始工作,在关中断状态下只执行限时严格且与硬件相关的任务,如获取来自设备的数据,将其保存到预定缓冲区,然后"标记中断",通知下半部分做剩余的工作,其他的工作由可中断代码来处理,称为中断下半部分处理,可见这是一种任务延迟处理机制。

上半部分是中断向量表中登记的中断服务程序的入口部分,在执行过程中它要决定其相关的下半部分函数是否执行。当处理快中断时,如果有其他中断到达,无论是快中断还是慢中断,它们都必须等待,否则,为了更快地处理所到来的其他中断,内核要尽可能地将耗时的处理工作放到下半部分执行,由于它是可中断的,如果在运行期间其他设备产生新的中断,则下半部分暂时被中断,等到那个设备的上半部分运行完毕,再回来运行它。

将上半部分与下半部分加以区分的另一个原因是下半部分函数含有某些中断处理并不一定非要执行的操作,对于这些操作,内核可以在一系列中断之后对其集中处理一次,在这种情况下,每次都单独执行下半部分显然是一种浪费。

与上半部分只能通过中断处理程序实现不同,下半部分可采用多种机制来实现,不同的机制由不同的子系统和接口组成,它们分别是:bottom half、task queue、tasklet、work queue 和 softirq。

可中断：意味着，如果这个时候再有中断发生，不会发生全局变量（数据结构）被冲乱的场景。

都已经被Linux Kernel废弃的机制。没有必要去深入了解。可以直接看V
Working queue机制。

3. 下半部分

Linux 最早提供的任务延迟执行机制称为 BH，用于实现下半部分(bottom half)，提供静态创建的下半部分处理的数据结构，建立一个函数指针数组，采用数组索引的方式访问，最多有 32 个不同的下半部分处理函数。bh _ base 是下半部分处理函数的入口指针，bh _ mask 和 bh _ active 共同控制下半部分处理函数能否运行，分别指明是否有下半部分处理函数被安装和上半部分处理函数是否标出哪个 BH 要执行。若 bh _ mask 的位 n 被置位，则 bh _ base 的第 n 个元素包含一个下半部分处理函数的入口地址；若 bh _ active 的位 n 被置位，则第 n 个下半部分处理函数能够被调度程序调度执行。只要对 bh _ mask 和 bh _ active 进行"按位与"运算就能表明应该运行哪个下半部分处理函数。索引在入口表中被静态地设置，计时器下半部分处理函数的入口位于索引 0，具有最高的优先级；控制台下半部分处理函数的入口位于索引 1，等等。

BH 机制存在两方面的局限性：

(1) 下半部分处理函数的数量限制为 32 个，且每个 BH 上只能挂接一个函数，随着系统设备的增多，BH 的应用范围越来越广，这个数目还不够用。

(2) 每个 BH 在全局范围内同步，即使属于不同的处理器，也不允许任何两个 BH 同时执行，这种机制使用方便但不够灵活，安全简单但尚存在性能瓶颈。所以，在开发 V2.5 内核版本时，BH 接口最终被抛弃。

4. 任务队列

进一步的改进方法是引入任务队列(task queue)机制，实现对各种任务的延迟执行。为此，内核定义一组队列，每个队列包含一个由等待调用的函数所组成的链表，不同队列中的函数在某个时刻会被触发执行。显然，任务队列不一定非要与中断处理有关，但是可以用它来替代 BH，当驱动程序或内核相关部分要将任务排队进行延迟处理时，可将任务添加到相应的任务队列中，然后，采用适当的方式通知内核执行任务队列函数。

典型的下半部分处理均有相关联的任务队列，任务队列在 Linux 中的应用范围相当广泛，还在其他场合被使用。任务队列与下半部分相比较，任务队列可以动态地定义和管理，而下半部分却由内核静态定义，且其处理函数不能超过 32 种。由于任务队列的灵活性较差，无法代替整个 BH 接口，也不能胜任微网络等对性能要求较高的子系统，此接口已经从 V2.5 版本中除去。

任务队列由 tq _ struct 单向链表结构组成，每个 tq _ struct 数据结构是任务队列的结点，包含处理函数 routine()的地址、避免重复入队标志 sync、函数参数的指针 data 和下一个任务结点的指针 next，当任务队列中的结点被处理时，将调用相应的处理函数并传递参数指针。Linux 核心中的任何部分，如设备驱动程序，都可以建立并使用任务队列机制。内核已经预定义以下 4 个任务队列。

(1) 定时器队列(TQ _ TIMER)

每次时钟滴答时刻到来时，都要检查定时器队列是否为空，如果不空则把定时器的下半部分标记为活动状态，以便激活定时器任务队列机制。

(2) 即时队列(TQ _ IMMEDIATE)

下半部分即时处理被一些设备驱动程序用来对任务进行排队，此队列的优先级较低，与定时

[注释:] 中断处理程序要求的是快速。因此Linux里有一套完整的Bottom Half的机制来做优化，把不存在重入危险的，不是那么急需立刻需要处理的，可以通过Bottom Half的一些手段，来“延迟”执行。从而使得整体系统并发度提高。早期BH，TaskQueue机制都已经不用了。

BH, softer, Tasklet都还是在Interrupt context下工作，所以，任何代码不能有sleep，等待sem，或者yield cpu的行为。

器队列的下半部分处理函数相比，这个队列中的任务处理要稍微迟缓一些。

(3) 进程调度队列(TQ_SCHEDULE)

当内核运行 schedule()函数时，此任务队列会被处理，调度程序在被调度进程的上下文中运行，所以此队列中的任务很少会受到限制。

(4) 磁盘队列(TQ_DISK)

内核的主存管理使用磁盘 I/O 请求队列。

run_task_queue()函数用于启动在某队列中排队的任务，通过将其挂接在 bottom half 向量表中来自动启动任务的执行。

下面通过定时器中断(0 号中断 IRQ0)的例子来说明中断服务程序的上半部分与下半部分之间的联系。定时器中断服务程序的名称是 timer_interrupt，其上半部分处理函数是 do_timer(·)，下半部分处理函数是 timer_bh()。当产生定时器中断时，timer_interrupt()调用上半部分处理函数 do_timer()，并做如下工作：更新全局变量 jiffies(加 1)；校正未被下半部分处理的时钟滴答数；执行 mark_bh(timer_bh)标记下半部分为活动状态；如果定时器队列中有任务在等待，执行 mark_bh(tq_timer)标记下半部分。此外，如果进程运行在核心态，则需递增所失去的时钟滴答。整个定时器中断的处理十分简单，这是因为很多工作已被延迟到下半部分进行处理。timer_bh()需要调用 update_times()计算与时间有关的统计数据；系统平均负载、当前时间全局变量、更新进程的 CPU 时间并修改 PCB 的有关成员。此外，还要执行定时器操作，调用新老定时器 run_old_timer()和 run_times_list()，检查并执行定时服务。

5. 小任务 Tasklet

在 Linux 2.3 版之后，有两种机制用来实现任务的延迟执行：tasklet 和 softirq。tasklet(小任务)也是一种下半部分机制，能够更好地支持对称式多处理器，它基于软中断来实现，但比软中断的接口要简单，锁保护的要求低；softirq 保留给对执行频率及时间要求特高的下半部分使用(如网络和 SCSI)。在大多数场合下可以使用 tasklet。

使用 tasklet 的步骤如下：

(1) 声明 tasklet。既可以静态地创建并直接引用，也可以动态地创建并通过指针间接引用。

(2) 编写 tasklet 处理程序。注意在运行时可以响应中断但不能发生阻塞，若使用共享数据结构需提前获取锁。

(3) 调度 tasklet。通过调用 tasklet 调度函数，并向它传递相应的结构体参数指针，启动 tasklet 处理程序。

BH不支持SMP。是被废弃的原因。softirq, tasklet支持SMP。

因为 BH 是全局串行处理，不适应对称式多处理器系统环境，引入 tasklet 之后，不同的 tasklet 可同时运行于不同的 CPU 之上。当然，由系统保证相同的 tasklet 不会同时在不同的 CPU 上运行。在这种情形下，tasklet 就无须是可重入的。在新版 Linux 中，tasklet 是系统推荐的异步任务延迟执行机制。

6. 工作队列 work queue

Linux 2.5 内核引入另一种任务延迟执行机制——工作队列(work queue)，与其他机制的工

work queue。是kernel thread。运行在Process Context下了。可以使用kernel里许多机制，等待资源。有最大的并发度。

需要快的用softirq。例如，网络。一般的用tasklet。复杂和需要线程之间交互的用work queue

作原理都不同，它把一个任务延迟，并将其交给内核线程去完成，且此任务总是在进程上下文中执行。这样，通过工作队列执行的代码能够占尽进程上下文的优势，最重要的是，工作队列允许重新调度和阻塞。如果延迟执行的任务需要阻塞、需要获取信号量或需要获得大量主存空间，那么，可以选择工作队列，否则就使用 tasklet 或 softirq。

工作队列子系统是一个用于创建内核线程的接口，由它所创建的线程负责执行由内核的其他部分排到队列中的任务，这些内核线程称为工作者线程。工作队列可以让驱动程序创建一个专门的工作者线程来处理需要延迟的任务，但是工作队列子系统已经提供一个默认的工作者线程来处理延迟任务，这样一来，工作队列最基本的表现形式就是把需要推迟执行的任务交给特定的通用线程这样一种接口。默认的工作者线程是 events/*n*，*n* 是处理器编号，每个处理器对应一个线程。许多内核驱动程序都将其下半部分交给默认的工作者线程去完成，除非一个驱动程序或子系统必须建立属于自己的内核线程，否则最好使用默认线程。

7. 软中断

迄今，Linux 沿用最早的 BH 思想，但在此机制上实现了庞大和复杂的软中断子系统——softirq，它既是一种软中断机制，又是一个框架，包括 tasklet 及为网络操作专门设计的软中断。tasklet 允许动态注册，但 softirq 在编译时静态定义。每个软中断的结构表示如下：

```
struct softirq_action
    void (*action)(struct softirq_action *);//待执行的函数
    void *data;                             //传递给函数的参数
};
```

最多可以注册 32 个软中断，目前 Linux 版本预定义 6 个元素，枚举表示如下：

```
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
};
```

softirq 保留给对时间要求严格的下半部分使用，其中，网络和 SCSI 属于这种情况，它们直接使用软中断。软中断的使用步骤如下：

(1) 声明一个 softirq。在编译期间静态声明并分配索引。

(2) 注册 softirq 处理函数。在软中断子系统进行初始化时，调用 open_softirq()注册中断处理函数，编程时注意不能发生阻塞，但允许响应中断。

(3) 触发软中断 softirq。raise_softirq()函数可置标志位来激活 softirq，在中断上半部分调用。

softirq：支持SMP的BH。同一个softirq可以在不同的CPU下并发执行；但同一个softirq不能在同一个CPU下同时执行。

通过spin lock来解决多个CPU上同时执行一个soft irq处理程序。

(4) 处理 softirq。当调用 do _ softirq()函数时，将查找软中断有标志位的 softirq，调用相应的软中断处理函数执行下半部分。

do _ softirq()共有 4 个执行时机，分别是：从系统调用中返回(ret _ from _ sys _ call)时、从异常中返回(ret _ from _ exception)时、在调度程序中(schedule)以及处理完硬件中断之后(do _ irq)。在 V2.6 内核中，do _ softirq()的执行时机有些变化，分别是：在处理完硬件中断之后；在 ksoftirqd 内核线程中；在显式检查和执行待处理的软中断的代码中，如网络子系统中。处理软中断时将遍历所有 softirq _ vec[ ]元素，依次启动其中的 action()函数，在中断服务程序(上半部分)中触发 softirq 是常见的形式，它执行硬件设备相关操作，再触发 softirq，内核在执行完中断服务程序之后，调用 do _ softirq()函数，于是 softirq 开始执行下半部分任务。需要注意的是，不允许 softirq 服务函数递归执行，但允许同一个 softirq 服务函数的两个实例同时在两个 CPU 上并行执行，当然，这时 softirq 必须是可重入的。

每个处理器都有一组称为 ksoftirqd/n 的辅助处理软中断的内核线程，例如，在进行大流量网络通信期间，软中断被触发的频率相当高，内核便在特殊时刻唤醒 ksoftirqd/n，它们就会调用 do _ softirq()进行处理。在一个双处理器机器上就有两个线程 ksoftirqd/0 和 ksoftirqd/1，它们在最低优先级(nice=19)上运行，这样就避免与重要任务抢夺资源，也能够保证所累积的软中断终究会获得处理。

2.2.7 Windows 2003 中断处理

1. 陷阱调度

在 Windows 2003 中，中断是异步事件，随时都有可能发生。与处理器正在执行的指令无关，中断主要由设备、时钟或定时器产生，可以启用或禁用。异常是同步事件，是执行某条特定指令的结果，如主存访问错误、调试指令及被 0 除，内核将系统调用也视为异常。硬件和软件都可以产生中断和异常，如总线出错异常由硬件造成，而被 0 除异常则由软件引起。同样的，设备可以产生硬件中断，而内核可以发出启动线程调度之类的软件中断。

陷阱调度设施(trap dispatching)是用来处理意外事件的硬件机制，当中断或异常发生时，硬件或软件可以检测并捕获它们，暂停当前线程的执行，让处理器从用户态切换到核心态，并将控制权交给内核陷阱调度程序。陷阱调度程序将检测中断和异常的类型并进行调度，将控制权交给相应的处理代码。如图 2.8 所示是 Windows 陷阱调度程序框架。

当陷阱调度程序被触发时，在记录机器状态期间将禁用中断，并创建陷阱帧(trap frame)来保存被中断线程的运行现场，以便适时恢复线程的执行。陷阱调度程序本身可以处理一些事件，如虚地址异常，但大多数情况下它只判断和确定所发生的情况，并把控制权转交给内核的其他部分或执行体。例如，如果是设备中断事件，将把控制权转交给设备驱动程序的中断服务例程 ISR；如果是调用系统服务事件，将把控制权转交给执行体中的系统服务代码；其他异常事件则由内核自身的异常调度器发送并由相应的处理程序进行处理。

softirq对per－cpu的数据结构不需要锁。目的就是追求高效，快！

进程及其实现

2.3.1 进程的定义和属性

"进程"是操作系统中最基本、最重要的概念，是在多道程序系统出现后，为了刻画系统内部的动态状况、描述运行程序的活动规律而引进的新概念，所有多道程序设计操作系统都建立在进

进程（Process）是最小的管理单位。是一个“家庭”

例如，时钟中断可以在任何两个指令间发生！

程的基础上。从理论角度看，进程的概念是对当前运行程序的活动规律的抽象；从实现角度看，进程是一种数据结构，用来准确地刻画系统动态变化的内在规律，有效地管理和调度在计算机系统主存运行的程序。基于进程的概念及其衍生思想，应用程序和操作系统都可以分解为逻辑上并发和物理上并行的程序。首先讨论操作系统中引入进程的目的。

一是刻画系统的动态性，发挥系统的并发性。在多道程序环境下，处理器在各程序之间来回切换，程序是并发执行的，程序的任意两条指令之间都可能发生事件而引发程序切换，因而程序的执行可能不是连续的而是走走停停的。此外，程序的并发执行又会引发资源共享和竞争问题，造成并发执行的各个程序之间可能存在制约关系，执行的程序不再处于封闭式环境中，出现许多新特征，而"程序"自身只是计算任务的指令和数据的描述，这种静态的概念无法刻画程序的并发性，系统要寻找能够描述程序动态执行过程的概念，这就是进程。进程是并发程序设计的一种有力工具，操作系统中的进程概念能较好地刻画系统内部的"动态性"，发挥系统的"并发性"，从而提高资源利用率。

二是解决共享性，正确地描述程序的执行状态。首先，引进"可再入"程序的概念，所谓"可再入"程序是指能够被多个程序同时调用的程序；另一种称为"可再用"的程序在被调用过程中可以自身修改，在调用它的程序退出之前是不允许其他程序来调用的。"可再入"程序是纯代码，在执行过程中不被修改；调用它的各应用程序提供工作区。因此，"可再入"程序可同时被几个应用程序调用。

在多道程序设计系统中，编译程序常常是"可再入"程序，可同时编译若干源程序。假定编译程序 P 正在编译源程序$_{甲}$，P 从起始点 A 开始工作，当执行到 B 点时需要将信息记录到磁盘上，那么 P 应在 B 点等待磁盘上的数据传输完成。此时处理器空闲，为了提高系统效率，利用编译程序的"可再入"特性，让 P 再为源程序$_{乙}$进行编译，仍然从起始点 A 开始工作，现在应如何描述编译程序 P 的状态呢？称它在 B 点等待磁盘数据传输完成，还是称它处于从 A 点开始执行的运行状态？P 只有一个，但其加工对象有源程序$_{甲}$和源程序$_{乙}$两个，所以，以程序作为占用处理器的单位显然不适宜。为此，可以把 P 与服务对象联系起来，P 为源程序$_{甲}$服务就构成进程$_{甲}$，P 为源程序$_{乙}$服务则构成进程$_{乙}$，两个进程虽然共享同一个编译程序，但是这两个进程可同时执行且彼此按各自的速度独立推进。现在可以说进程$_{甲}$（在 B 点）处于"等待磁盘数据传输完成状态"，而进程$_{乙}$从 A 点开始执行并处于"运行状态"。可见，程序与程序的执行（计算）不再一一对应，延用程序的概念不能描述这种共享性，必须引入新概念，这就是进程。

程的基础上。从理论角度看，进程的概念是对当前运行程序的活动规律的抽象；从实现角度看，进程是一种数据结构，用来准确地刻画系统动态变化的内在规律，有效地管理和调度在计算机系统主存运行的程序。基于进程的概念及其衍生思想，应用程序和操作系统都可以分解为逻辑上并发和物理上并行的程序。首先讨论操作系统中引入进程的目的。

一是刻画系统的动态性，发挥系统的并发性。在多道程序环境下，处理器在各程序之间来回切换，程序是并发执行的，程序的任意两条指令之间都可能发生事件而引发程序切换，因而程序的执行可能不是连续的而是走走停停的。此外，程序的并发执行又会引发资源共享和竞争问题，造成并发执行的各个程序之间可能存在制约关系，执行的程序不再处于封闭式环境中，出现许多新特征，而"程序"自身只是计算任务的指令和数据的描述，这种静态的概念无法刻画程序的并发性，系统要寻找能够描述程序动态执行过程的概念，这就是进程。进程是并发程序设计的一种有力工具，操作系统中的进程概念能较好地刻画系统内部的"动态性"，发挥系统的"并发性"，从而提高资源利用率。

二是解决共享性，正确地描述程序的执行状态。首先，引进"可再入"程序的概念，所谓"可再入"程序是指能够被多个程序同时调用的程序；另一种称为"可再用"的程序在被调用过程中可以自身修改，在调用它的程序退出之前是不允许其他程序来调用的。"可再入"程序是纯代码，在执行过程中不被修改；调用它的各应用程序提供工作区。因此，"可再入"程序可同时被几个应用程序调用。

程序被加载，执行——就产生了一个进程。

在多道程序设计系统中，编译程序常常是"可再入"程序，可同时编译若干源程序。假定编译程序 P 正在编译源程序$_甲$，P 从起始点 A 开始工作，当执行到 B 点时需要将信息记录到磁盘上，那么 P 应在 B 点等待磁盘上的数据传输完成。此时处理器空闲，为了提高系统效率，利用编译程序的"可再入"特性，让 P 再为源程序$_乙$进行编译，仍然从起始点 A 开始工作，现在应如何描述编译程序 P 的状态呢？称它在 B 点等待磁盘数据传输完成，还是称它处于从 A 点开始执行的运行状态？P 只有一个，但其加工对象有源程序$_甲$和源程序$_乙$两个，所以，以程序作为占用处理器的单位显然不适宜。为此，可以把 P 与服务对象联系起来，P 为源程序$_甲$服务就构成进程$_甲$，P 为源程序$_乙$服务则构成进程$_乙$，两个进程虽然共享同一个编译程序，但是这两个进程可同时执行且彼此按各自的速度独立推进。现在可以说进程$_甲$（在 B 点）处于"等待磁盘数据传输完成状态"，而进程$_乙$从 A 点开始执行并处于"运行状态"。可见，程序与程序的执行（计算）不再一一对应，延用程序的概念不能描述这种共享性，必须引入新概念，这就是进程。

把上述两点结合起来，进程是既能描述程序的并发执行状态、又能共享资源的一个基本单位，当然操作系统也要为引入进程而付出（进程占用的）空间和（调度进程的）时间代价。

进程的概念最早是 1960 年在美国麻省理工学院 MULTICS 和 IBM 公司 CTSS/360 系统中提出和实现的，直到目前为止，进程的定义和名称尚未统一，美国麻省理工学院称进程（process）、IBM 公司称任务（task）、Univac 公司称活动（action）。下面给出进程的定义：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。进程具有以下一些属性。

一个进程可以有多个线程。类似一个家庭可以有多个人。

程序可执行代码在磁盘上是一个elf格式。
加载时在内存中要展开，例如，堆栈，
堆，代码，数据。才可以执行。

(1) 结构性

进程包含数据集合和运行于其上的程序，它至少由程序块、数据块和进程控制块等要素组成。

(2) 共享性

同一程序同时运行于不同的数据集合上时，将构成不同的进程，即多个进程可以执行相同的程序，所以，进程和程序不是一一对应的。共享性还表现在进程之间可以共享某些公用变量，通过引用公用变量就能够交换信号，从而，进程的运行环境不再是封闭的。

(3) 动态性

进程是程序在数据集合上的一次执行过程，是动态的概念，同时，进程有生命周期，由创建而产生、由调度而执行、由事件而等待、由撤销而消亡；而程序是一组有序指令所组成的序列，是静态的概念，所以，程序作为一种系统资源是永久存在的。

(4) 独立性

进程是系统中资源分配、保护和调度的基本单位，说明它具有独立性，凡是未建立进程的程序，都不能作为独立单位参与调度和运行。此外，每个进程都可以有各自独立的、不可预知的速度在处理器上推进，即按照异步方式执行，这也表现出进程的独立性。

(5) 制约性

并发进程之间存在着制约关系，造成进程执行速度的不可预测性，必须对进程的并发执行次序、相对执行速度加以协调。

(6) 并发性

进程的执行可以在时间上有所重叠，在单处理器系统中可并发执行，在多处理器系统中可并行执行。对于单处理器系统而言，m 个进程 P_1、P_2、…、P_m 轮流占用处理器并发地执行。例如，进程 P_1 执行 N_1 条指令后让出处理器给进程 P_2，P_2 执行 N_2 条指令后让出处理器给进程 P_3，…，P_m 执行 N_m 条指令后让出处理器给进程 P_1，…。因此，进程的执行是可以被打断的，或者说，进程在执行完一条指令且下一条指令未执行前，可能需要被迫让出处理器，由其他若干进程执行若干条指令之后才能再次获得处理器继续执行。

2.3.2 进程的状态和转换

1. 三态模型

进程从因创建而产生直至撤销而消亡的整个生命周期中，有时占用处理器执行，有时虽然可以运行但分不到处理器，有时虽然处理器空闲但因等待某个事件发生而无法执行，这一切都说明进程和程序不同，进程是活动的且有状态变化，状态及状态之间的转换体现进程的动态性。为了便于系统管理，一般来说，按照进程在执行过程中的不同情况至少要定义三种进程状态。

(1) 运行态(running)：进程占用处理器运行的状态。

(2) 就绪态(ready)：进程具备运行条件，等待系统分配处理器以便其运行的状态。

(3) 等待态(wait)：又称阻塞态(blocked)或睡眠态(sleep)，是指进程不具备运行条件，正在等待某个事件完成的状态。

进程状态的转换可以是被迫，主动放弃，
时间片用完。时钟中断一直在跳。没有一
个进程能够逃脱调度。

[注释:] 在学习2.2中断技术方面，可以不要太陷入Linux独有的中断技术细节中。大概了解一些概念就可以了。把握两个重要的概念：中断上下文；进程上下文。中断处理程序需要快，然后迅速开中断，防止系统堵塞。这是Linux各种各样的Bottom Half机制的由来。

要补充阅读ELF结构。理解Data，BSS等段的含义，和进程在内存中的布局。

等待的事件独立于挂起条件，事件结束并不能导致进程具备可执行条件；此进程进入挂起状态是由于操作系统、父进程或进程自身阻止其运行；进程挂起状态的结束命令只能通过操作系统或父进程发出。

2.3.3 进程的描述和组成

1. 进程映像

进程的活动包括占用处理器执行程序以及对相关的数据进行操作，因而，程序和数据是进程必需的组成部分，两者刻画进程的静态特征。还需要数据结构来刻画进程的动态特征，描述进程状态、占用资源状况、调度信息等，通常使用一种称为进程控制块的数据结构。由于进程的状态在不断发生变化，某时刻进程的内容及其状态集合称为进程映像(process image)，包括以下一些要素。

(1) 进程控制块

每个进程捆绑一个控制块，用来存储进程的标志信息、现场信息和控制信息。进程创建时建立进程控制块，进程撤销时回收进程控制块，它与进程一一对应。

(2) 进程程序块

Code。代码段。

进程程序块是被执行的程序，规定进程的一次运行所应完成的功能。

(3) 进程核心栈

每个进程捆绑一个核心栈，进程在核心态工作时使用，用来保存中断/异常现场，保存函数调用的参数和返回地址，等等。

Stack从上往下。Data，例如堆（Heap）是从下往上长。

(4) 进程数据块

进程数据块是进程的私有地址空间，存放各种私有数据，用户栈也要在数据块中开辟，用于在函数调用时存放栈帧、局部变量等参数。

可见，每个进程由4个要素组成：控制块、程序块、核心栈和数据块。如果只具备前面3个要素，而共享用户地址空间，就称"用户线程"；如果完全没有用户空间，就称"内核线程"。

为了运行应用程序，必须要创建进程，进程运行过程中产生中断或执行系统调用时又要运行操作系统内核程序，这时系统将保存应用程序的所有现场信息及其用户栈，使之不被内核程序破坏；而内核程序在运行时，系统使用为进程所分配的核心栈，以便内核程序在函数调用或中断时存放现场信息。

实质上，进程在系统中存在及活动除了本身的实体外，还需要环境的支撑，如硬件寄存器、程序状态字寄存器、支持动态地址转换的页表和相关的核心数据结构。在操作系统中，进程物理实体和支持进程运行的环境合称进程上下文(process context)，进程在其当前上下文中运行，当系统调度新进程占有处理器时，新老进程随之发生上下文切换。进程上下文由以下三部分组成。

(1) 用户级上下文

用户级上下文(user level context)由正文(程序)、数据、共享存储区、用户栈所组成，它们占用进程的虚拟地址空间。对换至磁盘的分段或页面仍然是用户级上下文的组成部分。

(2) 寄存器上下文

Data。BSS属于Data。BSS就是程序中没有初始化的全局变量。

Kernel

Stack

Heap

BSS

全局变量

代码

进程结构

在理解上下文（context）的时候，要注意一点，不是每次都要存所有的寄存器的value。寄存器的使用通常上有一个约定。volatile的寄存器不需要保存。主要保存的寄存器是控制寄存器，状态寄存器等。

寄存器上下文(register context)由程序状态字寄存器、指令计数器、栈指针(机器状态决定它指向用户栈或核心栈)、控制寄存器、通用寄存器等组成。

(3) 系统级上下文

系统级上下文(system level context)由进程控制块、主存管理信息(页表或段表)、核心栈等组成。

在 UNIX 中，系统级上下文分为静态部分和动态部分，前者由 proc 结构、user 区和主存管理信息表等组成；后者由核心栈及其"上下文层"所组成。当发生中断、系统调用或进程上下文切换时，内核将对系统级上下文的动态部分(包含前一层寄存器上下文的内容)进行压入和弹出操作。机器硬件所支持的中断级数目将决定"上下文层"的数目，假设支持 5 级中断，加上系统调用和用户级，一个进程至多有 7 个"上下文层"。例如，应用进程发出一个系统调用，在执行系统调用服务程序时机器收到磁盘中断，执行磁盘中断服务程序时机器又收到时钟中断并执行时钟中断服务程序。每个中断事件产生时，内核将创建一个新的"上下文层"，并向核心栈压入当前的"上下文层"，于是核心栈中将依次压入应用程序寄存器上下文、系统调用服务程序寄存器上下文、磁盘中断服务程序寄存器上下文，而当前时钟中断服务程序正在运行。

2. 进程控制块

每个进程有且仅有一个进程控制块(Process Control Block，PCB)，或称进程描述符(process descriptor)，它是进程存在的唯一标识，是操作系统用来记录和刻画进程状态及有关信息的数据结构，是进程动态特征的一种汇集，也是操作系统掌握进程的唯一资料结构和管理进程的主要依据。进程控制块包括进程执行时的情况以及进程让出处理器之后所处的状态、断点等信息，一般来说，应包含以下三类信息。

(1) 标识信息

标识信息用于唯一地标识一个进程，分为用户使用的外部标识符和系统使用的内部标识号。系统中的所有进程都被赋予唯一的、内部使用的数值型进程号(通常是 0～32 768 的正整数)，操作系统内核函数可通过进程号来引用 PCB。常用的标识信息包括进程标识 ID、进程组标识 ID、用户进程名、用户组名等。

(2) 现场信息

现场信息用于保留进程在运行时存放在处理器现场中的各种信息。进程在让出处理器时，必须将此时的现场信息保存到它的 PCB 中，而当此进程恢复运行时也应恢复处理器现场。现场信息包括：通用寄存器内容、控制寄存器内容、栈指针等。

(3) 控制信息

控制信息用于管理和调度进程，包括：进程调度的相关信息，如进程状态、等待事件和等待原因、进程优先级、队列指针等；进程组成信息，如正文段指针、数据段指针、进程之间的族系信息，如指向父/子/兄弟进程的指针；进程间通信信息，如消息队列指针、所使用的信号量和锁；进程段/页表指针、进程映像在辅助存储器中的地址；CPU 的占用和使用信息，如时间片剩余量、已占用 CPU 时间、进程已执行时间总和、定时器信息、记账信息；进程特权信息，如主存访问权限和处理器特权；资源清单，如进程所需的全部资源、已经分得的资源，比如主存、设备、打开文件表等。

内核中，queue的查询需要特别快。对进程控制块的多维的index，是为了高效的队列操作。

索引方式是线性方式的一种改进，利用索引表记录不同状态进程的 PCB 地址，系统建立若干索引表，如就绪索引表、等待索引表、空闲索引表等。状态相同进程的 PCB 组织在同一张索引表中，每个索引表的表目中存放 PCB 在 PCB 表中的编号或地址，各索引表在主存中的起始地址放在内核的专用指针单元中。如图 2.15 所示是索引方式的原理。

图 2.14 链接方式

图 2.15 索引方式

下面介绍 Linux 进程链表。

Linux 中有双向循环链表、进程可运行队列链表、散列表和等待队列链表等多种组织进程控制块 task _ struct 结构的方式，它们都属于上述三种方式的一种。

双向循环链表反映进程的创建顺序及进程之间的亲属关系，每个 task _ struct 中的 prev _ task 和 next _ task 域用来实现这个结构，链表的表头是 init _ task 的进程控制块，任务数组 * task 的第一个元素指向它。

当内核执行进程调度，需要查找和选择就绪进程到处理器中运行时，引入可运行状态(TASK _ RUNNING)进程的双向循环链表，解决了浏览链表效率低的问题，它通过 task _ struct 结构中的 prev _ run 和 next _ run 成员链接，而就绪进程的总数存放在 nr _ running 中。

pidhash 散列表是通过 32 位 pid 来快速查找所对应的进程控制块 task _ struct 结构而引入的，采用索引方式，同一表项所有进程 task _ struct 的 pidhash _ next 和 pidhash _ prev 组成双向链表。

把 TASK _ INTERRUPTIBLE 和 TASK _ UNINTERRUPTIBLE 状态进程排入不同特定事件的等待队列的进程链表中，以备在事件发生时，由内核唤醒进程。等待队列与进程调度相结合，用于实现内核的异步事件通知机制。

2.3.4 进程切换与模式切换

1. 进程上下文切换

切换必须调整CPU的一些控制寄存器。这些必须在CPU的核心态模式下。

中断和异常是激活操作系统的仅有方法，它暂停当前运行进程的执行，把处理器切换至核心态，内核获得处理器的控制权之后，如果需要就可以实现进程切换。所以，进程切换必定在核心态而非用户态发生。内核在处理中断事件或系统调用的过程中可能会导致被阻塞的高优先级进程变为就绪态，或在处理时钟中断事件期间发现运行进程的时间片耗尽，或当前运行进程执行阻塞型 I/O 指令等，均有可能引发内核实施进程上下文切换。由于进程在让出处理器时，寄存器上下文将被保存到系统级上下文的相应的现场信息位置，这时内核就把这些信息压入核心栈的一个上下文层，当内核处理完中断返回时，或进程完成其系统调用返回用户态，内核进行上下文切换并从核心栈中弹出一个上下文层。因此，上下文切换总会引起系统级上下文层的进栈和退栈。内核在下列情况会发生上下文切换。

可能有更高优先的进程READY了。

(1) 当进程因各种原因进入等待态时；

(2) 当进程完成其系统调用返回用户态，但尚无资格获得 CPU 时；

(3) 当内核完成中断处理，进程返回用户态但尚无资格获得 CPU 时；

(4) 当进程执行的时间片到时或进程结束时。

在执行进程上下文切换时，保存老进程的上下文且装入被保护的新进程的上下文，以便新进程运行。进程切换的实现步骤如下。

(1) 保存被中断进程的处理器现场信息；

(2) 修改被中断进程 PCB 的有关信息，如进程状态等；

(3) 把被中断进程的 PCB 加入相关队列；

(4) 选择占用处理器运行的另一个进程；

(5) 修改被选中进程 PCB 的有关信息，如改为就绪态；

(6) 设置被选中进程的地址空间，恢复存储管理信息；

(7) 根据被选中进程的上下文信息来恢复处理器现场。

2. 进程上下文切换的时机

进程在运行过程中执行系统调用、产生中断或异常时，操作系统从当前运行进程那里获得控制权，此后，进程切换可以在任何时刻发生。系统在发现当前进程时间片耗尽，或执行一个阻塞型系统调用时，都会导致当前进程让出处理器，请求重新调度，内核转向低级调度执行，当低级调度程序选中新的就绪进程后，就会进行进程上下文切换。按道理说，请求重新调度事件之后，执行低级调度和进程上下文切换的工作应该连续进行，但实际上往往出于某种原因不能一气呵成。例如，在运行内核中断处理程序期间，发生一个优先级更高的 I/O 中断，当高优先级中断的处理结束之后，因为等待此 I/O 中断的进程转换为就绪态，故此时会请求低级调度。如果立即执行低级调度并做进程上下文切换，原先被高优先级中断所打断的那个低级中断处理程序尚未完成，不但会影响中断的响应时间，原先保存在 I/O 硬件中的现场信息也可能丢失；再如，若内核正在系统的临界区中执行，如果此时产生时钟中断，将导致请求低级调度的事件发生，若立即进行进程上下文切换则会带来临界区所占资源不能尽快释放的后果。此外，在各种原语操作、现场保护和恢复等过程中，即使发生请求低级调度的事件，也不能立刻执行低级调度和进行进程上下

形形色色的概念，机制，目标是一个：要保证数据结构的一致性，不能因为切换，中断，造成混乱。

通常是通过一个特殊的CPU指令来完成模式切换。例如，对状态处理器的模式bit的操作。

文切换。

如果在上述过程中出现引起调度的条件，并不能立即进行调度和切换，系统将采用置请求调度标志，延迟至敏感性操作完成之后才进行低级调度和进程上下文切换。为此，Linux 在进程 task _ struct 中设计重调度标志 need _ resched，当需要进程重调度时先行置位，在调度时机来临时，判别标志位是否为 1，以决定是否进行调度；UNIX System V 中引入重调度标志 runrun，其作用与 need _ resched 相同；Windows 操作系统为了实现这类延迟调度，专门设计延迟过程调用 Dispatch/DPC 软件中断，将 IRQL 设置为 2，以推迟线程调度的执行。

3. 处理器模式切换

与进程上下文切换有关的是 CPU 模式切换，从用户态到核心态或者从核心态到用户态的转换是 CPU 模式切换，此时仍然在同一个进程中运行。当发生中断或系统调用时，暂停正在运行的进程，把处理器状态从用户态切换到核心态，执行操作系统服务程序，这就是一次模式切换，此时进程仍在自己的上下文中执行，仅模式发生变化，内核在被中断进程的上下文中进行处理。模式切换的步骤如下：

Linux里有Interrupt Context的概念。

(1) 保存被中断进程的处理器现场信息；

(2) 处理器从用户态切换到核心态，以便执行系统服务程序或中断处理程序；

(3) 如果处理中断，可根据所规定的中断级别设置中断屏蔽位；

(4) 根据系统调用号或中断号，从系统调用表或中断入口地址表中找到系统服务程序或中断处理程序的地址。

BH，task let，softirq都允许在中断上下文里

模式切换不同于进程切换，它不一定会引起进程状态的转换，在大多数情况下，也不一定引起进程切换，在完成系统调用服务或中断处理之后，可通过逆向模式切换来恢复被中断进程的运行。CPU 上所执行进程在任何时刻必定处于三个活动范围之内：

(1) 用户空间中，处于进程上下文，用户进程在运行，使用用户栈。

(2) 内核空间中，处于进程上下文，内核代表某进程在运行，使用核心栈。

(3) 内核空间中，处于中断上下文，与进程无关，中断服务程序正在处理特定的中断，Intel x86 未提供中断栈，借用核心栈。

第一种情况通常对应于应用进程正常运行的状态，当请求执行系统调用时，发生第二种情况，内核接受控制，代表此进程在运行。也可以说，应用程序通过系统调用在内核空间运行，而内核运行于进程的上下文中，这种复杂的关系，即应用程序通过系统调用陷入内核，是应用程序在请求操作系统服务，完成其工作任务。在应用进程正常运行的过程中，若发生中断事件，系统将予以响应并转入相应的中断服务程序进行处理，第三种情况发生，许多操作系统的中断服务程序不在进程上下文中执行，而是在一个与所有进程毫无关联的执行环境——专门的中断上下文中执行，其目的是保证中断事件在第一时间获得响应和处理，然后尽快退出。既然不存在进程背景，中断上下文就不可能被阻塞，不能从中断上下文中调用某些引起阻塞的函数。另外，中断处理程序没有自己的栈，工作时共享被中断进程的核心栈，如果没有正在运行的进程，便会使用 idle 进程的核心栈，所以中断处理程序非常节省存储空间。需要注意的是，用户空间和内核空间

CPU模式不同。但进程都在“运行”状态下。

Kernel Thread，例如。

的数据不能互访，在应用程序执行系统调用的过程中，如果有需要，可通过内核函数 copy _ from _ user()和 copy _ to _ user()将数据在用户空间和内核空间中进行复制。

4. UNIX/Linux 中的进程切换与模式切换

在 UNIX/Linux 中，进程通常运行在用户态和核心态。如果当前运行的是应用程序或内核之外的系统程序(如 shell)，对应的进程就在用户态运行；如果应用程序执行系调用，或发生中断/异常事件，就要运行系统程序，且处理器处于核心态。Linux 把内核空间中运行的程序称为任务，而把用户空间中运行的程序称为进程，且经常交替地使用这两个术语。那么，在系统中存在两种进程(任务)：系统进程(任务)和用户进程(任务)，实质上是指一个进程(任务)的两个侧面，系统任务是在核心态执行操作系统代码的进程，用户进程是在用户态执行应用程序的进程，应用程序通过系统调用陷入内核时，系统任务就代表用户进程开始执行。但是，这两个进程所执行的程序不同，映射到不同的物理地址空间，使用不同的堆栈。

在如图 2.16 所示的进程生命周期中，列出进程切换和模式切换的示意。事实上，进程状态在动态地转换，图 2.16 中有箭头所指的状态转换是合法的，通常进程每次完成且只完成一个合法的状态转换。在任何时刻，处理器仅执行一个进程，所以，至多只有一个进程处于状态(1)或状态(2)，这两个状态相当于处理器处于用户模式和内核模式。当发生系统调用或中断时，进程通过一次模式切换从用户态运行转换为核心态运行，在核心态运行的进程是不能被抢占的。当进程在内核模式运行时，可以继续响应中断，处理中断或系统调用之后，通过一个模式切换返回用户态继续运行。也可以根据具体的情况使进程等待某个事件发生(状态(4))，只有此时才允许内核发生进程切换，使原先运行的进程让出处理器。当被等待事件完成后，进程会从等待状态被唤醒，进入就绪状态(状态(3))，若被调度程序选中，此进程会重新占用处理器运行。在多道程序设计系统中，有许多进程并发执行，如果两个以上的进程同时执行系统调用，并要求在内核模式执行，就有可能破坏核心数据结构中的信息。通过禁止任意的上下文切换和控制对于中断的响应，就能够保证数据的一致性和完整性。

运行状态；CPU模式不同。

图 2.16 进程上下文切换和模式切换

现代操作系统，例如，Linux可以支持在内核中的某些地方，剥夺式的调度了。总之，是在保证内核数据结构一致性的基础上。

[注释:] 传统Unix的内核是非剥夺式的调度（non preemptive），目的是简单从而保证内核数据结构一致性（内核是各个进程共享的地址空间）。现代OS，如2.6之后的Linux，可以支持内核preemptive schedule了。目标是支持高并发系统。

4. 进程挂起和激活

当出现引起挂起的事件时，系统或进程会利用挂起原语把指定进程或处于等待态的进程挂起，其执行过程大致是：检查要被挂起进程的状态，若处于活动就绪态，就修改为挂起就绪态；若处于等待态，就修改为挂起等待态，被挂起进程 PCB 的非常驻部分要交换到磁盘对换区。

当系统资源尤其是主存资源充裕或请求激活指定进程时，系统或相关进程会调用激活原语把指定进程激活，此原语所做的工作是：把被挂起进程 PCB 的非常驻部分调入主存，然后修改其状态，挂起等待态改为等待态，挂起就绪态改为就绪态，并将进程移入相应的队列中。需要注意的是，挂起原语既可由进程自己也可由其他进程调用，但是激活原语只能由其他进程调用。

在 Windows 中，处理器的调度对象是线程，用户通过系统调用 SuspendThread 和 ResumeThread 来挂起和激活线程。一个线程可被多次挂起和多次激活，在线程控制块中有一个挂起计数器，挂起操作将使此计数器加 1，激活操作将使此计数器减 1。当挂起计数从 0 变为 1 时，线程进入等待态；当挂起计数从 1 变为 0 时，线程被唤醒并恢复运行。

2.4 线程及其实现

进程与线程

2.4.1 引入多线程的动机

如果说操作系统中引入进程的目的是为了使多个程序并发执行，以便改善资源利用率和提高系统效率，那么，在进程之后再引入线程的概念，则是为了减少程序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。解决问题的基本思路是：把进程的两项功能"独立分配资源"和"被调度分派执行"分离开来，前一项任务仍然由进程完成，作为系统资源分配和保护的独立单位，无须频繁地切换；后一项任务交给称作线程的实体来完成，线程作为系统调度和分派的基本单位，会被频繁地调度和切换。在这种思想的指导下，产生了多线程的概念，即多线程（结构）进程。

传统的操作系统一般只支持单线程（结构）进程，如 MS-DOS 支持单用户进程，进程是单线程的；传统 UNIX 支持多用户进程，进程也是单线程的。目前，很多操作系统都支持多线程（结构）进程，如 Solaris、Mach、SVR4、OS/390、OS/2、Windows NT、CHORUS 等；Java 的运行引擎则是单进程多线程的例子。许多计算机公司都推出自己的线程接口规范，如 Solaris 线程接口规范、OS/2 线程接口规范、Windows NT 线程接口规范等，IEEE 也推出 UNIX 类操作系统的多线程程序设计标准 POSIX 1003.4a。

2.4.2 多线程环境中的进程与线程

1. 多线程

多线程结构进程如图 2.17 所示。在此给出多线程环境中进程的定义：进程是操作系统中进

为什么有了进程的机制，还需要线程呢？为了支持高并发，都拆分为进程不是很好嘛？原因是：属于同一个进程的线程是属于一个家的，是共享一个管理，共享内存的，通常是属于一个战壕的。类似一个家庭，每天大家各自出去上班。但都在为这个家打拼。

通常是用pthread的APIs来编程。

程也转换为等待态；对于另外一些线程实现机制，如果存在另一个处于就绪态的线程，则调度此线程运行，否则进程才会转换为等待态。显然前一种做法欠妥，丧失多线程机制的优越性，降低系统的灵活性。对于多线程进程的进程状态，由于进程不是调度单位，不必将其划分成过细的状态，如 Windows 操作系统中仅把进程分成可运行态和不可运行态，挂起状态属于不可运行态。

3. 线程管理和线程库

多线程技术利用线程库提供一整套有关线程的过程调用或系统调用来支持多线程运行，有的操作系统直接支持多线程，有的语言则提供线程库。因而，线程库可分为用户空间线程库和内核空间线程库，线程库实际上是多线程应用程序的开发和运行环境。一般来说，线程库至少提供以下功能的调用：孵化、封锁、活化、结束、通信、同步、互斥、切换（保护和恢复线程上下文），还要提供线程调度的代码，同时提供相关的 API 以支持应用程序的创建、调度、撤销并管理线程的运行。

4. 多线程程序设计的优点和线程的组织

多线程概念的引入使得单个进程可以利用 CPU 和设备之间的并行性，及多处理器之间的并行性。多线程程序设计的优点是提高系统性能，具体表现在：快速线程切换；节省主存空间；减少管理开销；通信易于实现；并发程度提高。由于队列管理和 CPU 调度是以线程为单位的，因此，涉及执行的大多数状态信息被维护在线程数据结构中。然而，存在一些影响进程中所有线程的活动，如挂起、终止等操作必须在进程级进行管理。

一个进程可以包含若干线程，这些线程可以有多种组织方式：第一种是调度员－工作者模式，进程中的一个线程担任调度员，接收和处理工作请求，其他线程是工作者线程，由调度员线程分配任务并唤醒工作者线程；第二种是组模式，进程中的各个线程都可以取得并处理此请求，不存在调度员线程。有时每个线程被设计成专门执行特定的任务，同时，建立相应的任务队列；第三种是流水线模式，线程排成某个次序，第一个线程所产生的数据传送给下一个线程进行处理，依次类推，数据按照排定的次序由线程依次传递以完成被请求的任务。

多线程技术在现代计算机软件中得到广泛的应用，多线程技术的主要应用包括：前台和后台工作、客户－服务器应用模式、任务异步处理、用户界面设计等。

在此列举多线程在字处理程序中发挥作用的例子。如果字处理程序是单线程的，那么输入、编辑和存盘三项任务串行完成。例如，在进行磁盘备份时，来自键盘和鼠标的命令和信息不会被处理，直至备份结束，这样用户会感觉系统性能很差。假设字处理程序被编写成含有 3 个线程，第一个线程在前台与用户交互，监控键盘和鼠标，一旦发现有变化便通知第二个线程；第二个线程在后台工作，得到通知时重新进行编辑处理；第三个线程在后台周期性地将文件写盘，以免因发生故障而丢失已处理的内容。由于 3 个线程处于同一个地址空间中，且共享同一个文件，因而能协调、高效地工作。

2.4.3 线程的实现

多线程的实现分为三类：用户级线程（User Level Thread，ULT），如 POSIX1003.4a 的

进程切换的开销其实不大。多线程最大的好处是：在一个共享内存的进程里，可以与并发的执行多个逻辑。共同完成一个进程的任务。。。

1:1. 任何一个线程同时在内核调度逻辑中对应一个task。存在一个调度的object。因此一个进程的多个线程的调度可以是SYSTEM_SCOPE的，而非用户级线程的APPLICATION或者说PROCESS SCOPE的。

Pthreads、Java 线程库；内核级线程(Kernel Level Thread，KLT)，如 Windows 2003、OS/2 和 Mach C-thread；某些系统(如 Solaris UI-threads)提供混合方式，同时支持 ULT 和 KLT 两种线程。

1. 内核级线程

内核级线程是指线程的管理工作由内核完成，由内核所提供的线程 API 来使用线程，当任务提交给操作系统执行时，内核为其创建进程和一个基线程，线程在执行过程中可通过内核的创建线程原语来创建其他线程，应用程序的所有线程均在一个进程中获得支持。内核需要为进程及进程中的单个线程维护现场信息，所以，应在内核空间中建立和维护进程控制块 PCB 及线程控制块 TCB，内核的调度在线程的基础上进行。

内核级线程的主要优点是：首先，在多处理器上，内核能够同时调度同一进程中的多个线程并行执行；其次，若进程中的一个线程被阻塞，内核能够调度同一进程的其他线程占有处理器运行，也可以运行其他进程中的线程；最后，由于内核级线程只有很小的数据结构和堆栈，其切换速度比较快，内核自身也可用多线程技术实现，从而提高系统的执行效率。内核级线程的主要缺点是：线程在用户态运行，而线程的调度和管理在内核实现，在同一进程中，控制权从一个线程传送到另一个线程时需要用户态－核心态－用户态的模式切换，系统开销较大。

2. 用户级线程

用户级线程是指线程的管理由应用程序完成，在用户空间中实现，内核无须感知线程的存在。用户级多线程由用户空间中的线程库来完成，应用程序通过线程库进行设计，再与线程库连接、运行以实现多线程。线程库是由用户级线程管理的例行程序包，在这种情况下，线程库是线程运行的支撑环境。

用户级线程有许多优点：线程切换无须使用内核特权方式，所有线程管理的数据结构均在进程的用户空间中，管理线程切换的线程库也在用户空间中运行，这样可以节省模式切换的开销和内核的宝贵资源；允许进程按照应用的特定需要选择调度算法，且线程库的线程调度算法与操作系统的低级调度算法无关，能够运行在任何操作系统上，内核无须做任何改变。用户级线程的明显缺点是：由于大多数系统调用是阻塞型的，因此，一个用户级线程的阻塞将引起整个进程的阻塞；用户级线程不能利用多重处理的优点，进程由内核分配到 CPU 上，仅有一个用户级线程可以执行。因此，不可能得益于多线程的并发执行。

3. 混合式线程

某些操作系统既支持用户级线程，又支持内核级线程，Solaris 便是一个例子。线程的实现分为两个层次：用户层和核心层。用户层线程在用户线程库中实现；核心层线程在操作系统内核中实现，处于两个层次的线程分别称为用户级线程和内核级线程。在混合式线程系统中，内核必须支持内核级多线程的建立、调度和管理，同时也允许应用程序建立、调度和管理用户级线程。

在混合式线程中，一个应用程序中的多个用户级线程能分配和对应于一个或多个内核级线程，内核级线程可同时在多处理器上并行执行，且在阻塞一个用户级线程时，内核可以调度另一个线程执行，使得宏观上和微观上都具有很好的并行性。例如，窗口系统是典型的逻辑并行性程

N：1 用户级别线程。多个线程，但在内核中只有一个调度的object，例如一个task。线程的并发局限在这个task被分配的time slice范围内。

这句话这么理解：没有0-3G的用户态虚拟空间。这Linux中，不管用户态，还是核心态，MMU虚拟内存管理都是打开的。地址都需要MMU转换。

度较高的应用，用一组用户级线程来表达多个窗口，用一个内核级线程来支持这一组用户级线程，这样，屏幕上同时出现多个窗口(对应于多个用户级线程)，窗口之间的切换很频繁，但某一时刻只有一个窗口(对应于内核级线程)处于活跃状态，系统开销小，窗口系统执行效率高。又如，大规模并行计算是对物理并行性要求较高的应用，可以把数组按行或列划分给不同的用户级线程处理，同时，让每个用户级线程与一个内核级线程绑定，每个内核级线程占用一个 CPU 并行执行，减少了线程切换的次数，通过并行计算提高系统效率。

2.5 Linux 进程与线程

Linux 是多用户多任务操作系统，支持多道程序设计、分时处理和软实时处理，从 V2.2.x 内核开始，支持对称式多处理器和多线程。Linux 认为线程就是共享地址空间及其他资源的进程，因而并未单独为线程定义数据结构，有一套在用户模式下运行的线程库 pthread，但每个线程都拥有属于自己的唯一 task _ struct。Linux 还支持内核线程，这类线程周期性地被内核唤醒和调用，没有虚拟地址空间，运行在核心态，工作在内核空间中，可以被调度和抢占，主要用于实现系统后台操作，如页面对换、刷新磁盘缓存、网络连接等系统工作。

在 Linux 系统启动时，最先产生 idle 进程(其 pid 为 0)，此进程会创建一个内核线程来执行初始化工作，结果使处理器的运行模式由核心态切换至用户态，内核线程演变成用户进程 init (其 pid 为 1)，一切用户进程都是它的后代进程。在创建每个进程时，要为其建立新的进程描述符结构体：

Linux在数据结构上，没有单独的线程数据结构。都是通过task_struct来刻画一个可调度的object。

```
struct task _ struct {
    volatile long state;                          //进程状态
    unsigned long flags;                          //进程标志
    pid _ t pid;                                  //进程 ID
    struct thread _ info  * thread _ info;        //指向进程的 thread _ info 指针
    int load _ weight;                            //平衡负载所用的权重
    int prio, static _ prio, normal _ prio;       //动态优先级、静态优先级和正常优先级
    struct list _ head run _ list;                //连接 prio _ array 中的元素
    struct prio _ array  * array;                 //当前 CPU 的就绪队列
    unsigned long policy, sleep _ avg, rt _ priority;  //调度策略、平均等待时间、实时进程优
                                                       先级
    unsigned int time _ slice, first _ time _ slice;   //时间片余额；是否是第一次分得时间
                                                       片的标志
    struct list _ head tasks;                     //任务队列
    struct mm _ struct  * mm,  * active _ mm;     //进程虚存信息；内核线程借用的地址空间
    struct task _ struct  * parent;               //父进程
```

什么是进程？ 其实就是操作系统程序里的一个数据结构。一个task数组里的一个数组元素！

fork, pthread_create最后都调用clone（）函数。传进去的参数不同而已。子进程，线程，在内核中，都是产生一个task_struct元素。

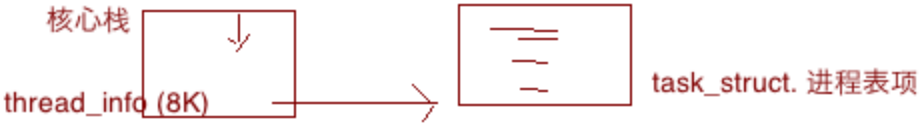

```
struct list _ head children, sibling;          //子进程链表、兄弟进程链表
struct pid _ link pids[PIDTYPE _ MAX];         //pid 的哈希表链
struct list _ head thread _ group;             //线程队列链
struct timer _ list real _ timer;              //时间信息
unsigned long it _ real _ value,it _ prof _ value,it _ virt _ value ,utime,stime;
uid _ t uid,euid,suid,fsuid;                   //各种用户的标识符
gid _ t gid,egid,sgid,fsgid;                   //各种组的标识符
int link _ count, total _ link _ count;        //各种文件链接数
struct fs _ struct * fs;                       //文件系统信息
struct files _ struct * files;                 //打开文件表指针
struct signal _ struct * signal;               //信号处理信息
struct sighand _ struct * sighand;
sigset _ t blocked, real _ blocked;
struct sigpending pending;
};
```

"CLONE_THREAD"参数可以告诉clone调用，这个新产生的object是一个thread逻辑，从而，会在thread_group里把这个新产生的object加入，作为一个线程来对待。但这种对待是一种逻辑理解而已。

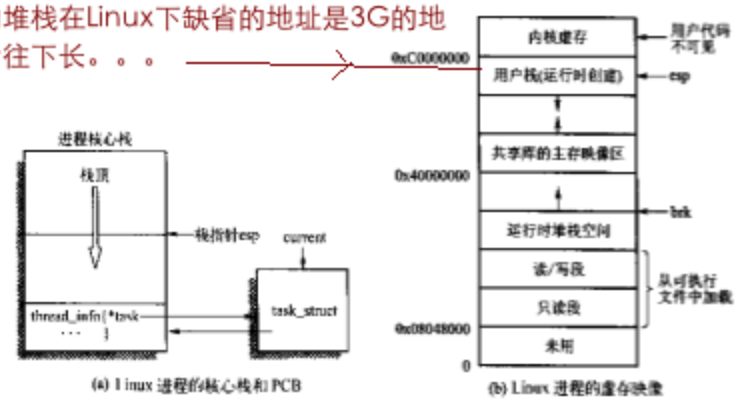
进程描述符中包含：进程标识、状态、标志、链接、调度、文件、虚存、信号等信息。在 V2.2 之前的版本中，进程重调度标志 need _ resched 是全局变量；在 V2.2 到 V2.4 版中它被移入进程 task _ struct 中；而在 V2.6 版中，它被移至 thread _ info 结构体中，用特别的标志变量中的一位 TIF _ NEED _ RESCHED 来表示。

Linux 内核中保留一个大小为 NR _ TASKS 的全局数组 * task[NR _ TASKS]，默认长度为 512 bit，数组元素是指向 task _ struct 的指针，在创建新进程时，从系统空间中分配一个 task _ struct 结构，并将其首地址填入 * task 数组。进程核心栈和 task _ struct 结构间存在紧密的联系，V2.4 中，内核为进程建立 task _ struct 时，一次分配两个连续页框(8 KB)，其底部约 1 KB 空间用做 task _ struct 结构，上端约 7 KB 为进程栈心栈。在 V2.6 中，task _ struct 结构变为由 slab 分配器动态分配，如图 2.18(a)所示，进程的运行环境信息 thread _ info 结构在栈心栈尾端分配，此结构中的 task 成员是指向此进程的 task _ struct 的指针。current 指针通过堆栈指针 esp 来计算，去掉 esp 末尾 13 位得到 thread _ info 的地址，再通过 thread _ info - >task _ struct 得到 task _ struct 的地址。图 2.18(b)为 Linux 进程的虚存映像。

Linux 进程的运行环境信息 thread _ info 是 V2.6 新创建的一个结构，除进程描述符 task _ struct 指针外，还包含当前 CPU 号、底层标志、运行域、线程同步标志、内核抢占计数器等。task _ struct 约占 1.7 KB，基本内容与先前版本相同，为了支持新的 $O(1)$ 调度算法，增加用于调度的新成员有：动态优先级 prio、静态优先级 static _ prio、正常优先级 normal _ prio、优先级数组 prio _ array、进程时间片剩余值 time _ slice、进程平均等待时间 sleep _ avg、负载平衡权重 load _ weight 等，延用的有实时优先级 rt _ priority、调度策略 policy 等。



用户态的堆栈在Linux下缺省的地址是3G的地方。然后往下长。。。



(a) Linux 进程的核心栈和 PCB　　(b) Linux 进程的虚存映像

图 2.18　Linux 进程

处于相同状态的进程通过前后向队列指针连接成双向循环链表，链表的第一个元素是 init_task 描述符，又称 0 号进程，它是所有进程的祖先。内核还会建立运行队列链表及多个等待队列链表，宏 current 是当前正在运行的进程的指针，在对称式多处理器系统中则指向 CPU 组中正在被调度的 CPU 上的当前进程。

Linux 进程的状态 state 共有以下 6 种。

(1) TASK_RUNNING

可运行状态：进程正在运行(已获得 CPU)或准备运行(等待获得 CPU)，由 run_list()把所有运行态进程链接起来。V2.6 内核改为每个 CPU 单独维护一个进程运行队列。

(2) TASK_INTERRUPTIBLE

可中断等待状态：进程排入等待队列中，一旦资源可用时进程就被唤醒，也可由其他进程通过信号或中断来唤醒。

(3) TASK_UNINTERRUPTIBLE

不可中断等待状态：进程排入等待队列中，一旦资源可用时进程就被唤醒，但是不可由信号或中断来唤醒。此状态通常意味着进程在等待某些硬件条件，如进程打开设备文件时会处于此状态。

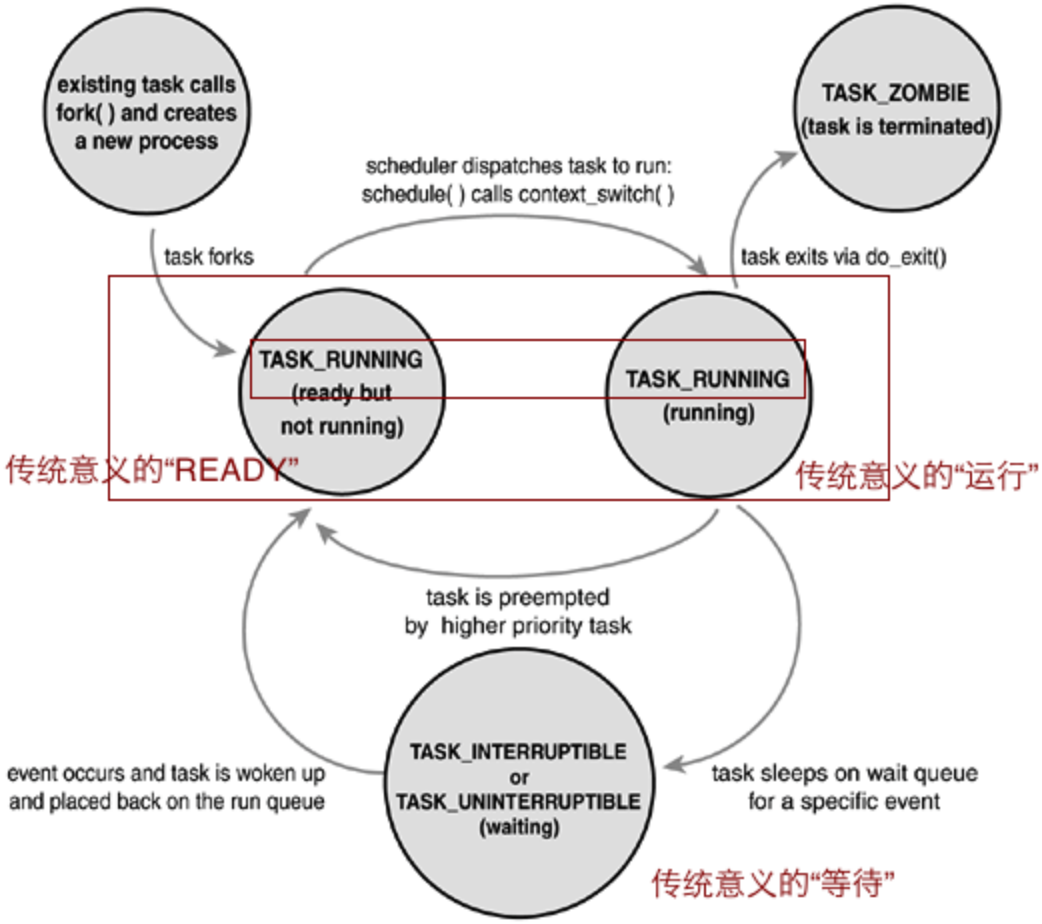
(4) TASK_ZOMBIE

僵死状态：进程运行完成，已释放所占用的大部分资源，尚未释放 task_struct 结构体。

(5) TASK_STOPPED

暂停状态：此状态用于调试，因收到信号而暂停执行，也可能受到其他进程的跟踪和调用而暂时让出处理器。

例如，一个可能时间很长的读磁盘的操作。在sleep的时候，如果是可中断的。另外一个进程可以发signal。这样这个进程就能够从这个读数据的系统调用里退出来了。。。


existing task calls fork( ) and creates a new process
TASK_ZOMBIE (task is terminated)
scheduler dispatches task to run: schedule( ) calls context_switch( )
task forks
task exits via do_exit()
TASK_RUNNING (ready but not running)
TASK_RUNNING (running)
传统意义的“READY”
传统意义的“运行”
task is preempted by higher priority task
event occurs and task is woken up and placed back on the run queue
TASK_INTERRUPTIBLE or TASK_UNINTERRUPTIBLE (waiting)
task sleeps on wait queue for a specific event
传统意义的“等待”

每个线程是自己对立的堆栈。这个概念很重要。类似，家庭里，每个人有自己的钱包。

行除处理器以外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和数据），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、外部设备和其他进程（进程间通信）。

各个线程共享全局变量，或者说DATA、BSS。

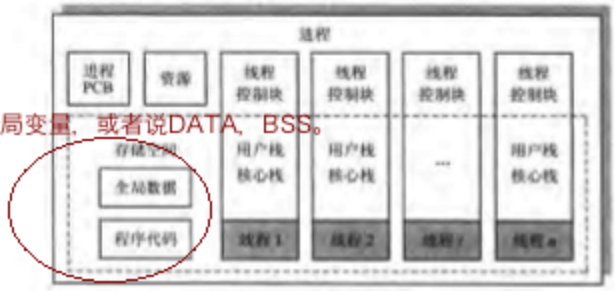


图2.17 多线程结构进程

函数调用进进出出都需要stack。必须每个thread有一个自己的stack。概念要非常清楚。

线程是进程中能够并发执行的实体，是进程的组成部分，也是处理器调度和分派的基本单位。允许进程包含多个可并发执行的线程，这些线程共享进程所获得的主存空间和资源，可以为完成某一项任务而协同工作。线程的组成部分有：

(1) 线程的唯一标识符及线程状态信息；

(2) 未运行时所保存的线程上下文；可以把线程看成进程中一个独立的程序计数器；

(3) 核心栈，在核心态工作时保存参数，在函数调用时的返回地址，等等；

(4) 用于存放线程局部变量和用户栈的私有存储区。

进程可以分为两个部分：资源集合和线程集合。进程要支撑线程的运行，为线程提供地址空间和各种资源，进程封装管理信息，包括对指令代码、全局数据、打开的文件和信号量等共享部分的管理；线程封装执行信息，包括对 CPU、寄存器、执行栈（用户栈与核心栈）和局部变量、过程调用参数、返回值等私有部分的管理。由于线程具有传统进程的许多特征，所以，也把线程称为轻量进程（Light Weight Process，LWP）。

2. 线程状态

与进程类似，线程也有生命周期，因而也存在各种状态。线程的状态有运行、就绪和等待，线程的状态转换也与进程类似。由于线程不是资源的拥有单位，挂起状态对于线程是没有意义的。如果进程在挂起后被对换出主存，它的所有线程因共享地址空间，也必须全部对换出去。可见由挂起操作所引起的状态是进程级状态，不是线程级状态。类似地，进程的终止将导致进程中所有线程的终止。

进程中可能有多个线程，当处于运行态的线程在执行过程中要求系统服务，如执行 I/O 请求而转换为等待态时，那么，多线程进程中是否要阻塞整个进程，对于某些线程实现机制，所在进

[注释:] 为什么需要线程（thread）来支持并发？都用进程不就可以了嘛？因为属于同一个进程的多个线程是共享全局变量的，共享内存的，可以合作来完成一个任务。而如果拆分为多个进程，通常就需要IPC通信才能完成这样的任务。这就是线程的好处。

用户级线程是一个模拟。在内核中不存在一个object对应。N：1的模型。所有的线程共享这个进程的时间片。

丰富的联机控制命令供交互型用户使用，具体的键盘命令主要有：

(1) 作业控制类：编辑、编译、链接、运行、排错等；

(2) 资源申请类：申请/归还主存资源、申请/归还独占型设备等；

(3) 文件操作类；

(4) 目录操作类；

(5) 设备控制类。

在用户级线程模型中，一个线程堵塞，例如，I/O，会导致这个进程的其他线程都无法获得CPU。

现代操作系统很少使用真正的批处理策略，为了减轻联机用户重复输入命令的负担，允许将一组系统命令组织成一个批处理文件，然后对这个文件中的命令进行批处理，使联机用户可以享受到自动批处理操作方式的好处。

2.8 处理器调度算法

2.8.1 低级调度的功能和类型

低级调度的最初对象是进程，随着现代操作系统引入多线程机制，进程演变成资源分配和保护的单位，从而进程只作为中级调度的对象，内核级线程替代进程成为低级调度对象。适用于进程调度的算法一般都适用于内核级线程调度，因此在下面的讨论中，并不严格地区分进程和内核级线程(有时用"进程"，有时用"进程/线程")。需要指出的是，用户级线程的调度是应用程序自己的事，虽然程序设计语言的函数库会提供默认的调度算法，但事实上它们与操作系统无关。在纯用户级多线程系统中，低级调度的对象依然是进程；而在内核级多线程或混合式策略中，低级调度的对象是内核级线程。

1. 低级调度的主要功能

操作系统的调度程序(scheduler)担负两项任务：调度和分派。前者实现调度策略(scheduling policy)，确定就绪态进程/线程竞争使用处理器的次序的裁决原则，即进程/线程何时应放弃CPU和选择哪个进程/线程来执行；后者实现调度机制(scheduling mechanism)，确定如何时分复用CPU，处理上下文交换细节，完成进程/线程同CPU的绑定及放弃的实际工作。

当进程/线程正常运行时，为什么会出现低级调度的需求呢？正在执行的进程/线程运行结束或由于某事件而无法继续运行，也可能出现更高优先级的进程/线程，需要剥夺当前使用者所占用的处理器，这些原因都可能引起低级调度需求。具体地说，下述事件发生后需要执行低级调度：创建新进程/线程，进程/线程终止，进程/线程阻塞自己，进程/线程运行过程中发生中断或异常，进程/线程执行系统调用，进程/线程所请求的I/O操作完成，进程/线程耗尽时间片，进程/线程改变优先级，等等。

从概念上来看，调度机制由3个逻辑功能程序模块组成：

(1) 队列管理程序 Queue Management

当一个进程/线程转换为就绪态时，其PCB/TCB会被更新以反映这种变化，队列管理程序

不太讲究 高，中，低级调度这个说法。可以更多的通过"调度策略"来理解更高级的调度Policy。

通常是一段汇编程序。20，30行。不多。主要是一些相关寄存器，堆栈的准备工作。

(queuer)将 PCB/TCB 指针放入等待 CPU 资源的进程/线程列表中，每当把进程/线程移入就绪队列时，可计算为此进程/线程分配 CPU 的优先级备用。

(2) 上下文切换程序

当调度程序把 CPU 从正在运行的进程/线程那里切换至另一个进程/线程时，上下文切换程序(context switcher)将当前运行进程/线程的上下文信息保存至其 PCB/TCB 中，恢复选中进程/线程的上下文信息，从而使其占用处理器运行。

(3) 分派程序

当一个进程/线程让出 CPU 资源后，分派程序(dispatcher)被激活。当然，为了运行分派程序，需要将其上下文装入 CPU，分派程序从就绪队列中选择进程/线程，而后完成从其自身到所选择的进程/线程间的又一次上下文切换，把 CPU 让给被选中的进程/线程。

在一些计算机系统中，使用两组或多组硬件上下文寄存器来缩短上下文切换时间，CPU 在核心态使用一组寄存器，CPU 在用户态使用另一组寄存器，那么，在上下文切换的过程中，当操作系统与应用程序代码来回切换时，只需花费改变指向当前寄存器组的指针的时间。

2. 低级调度的基本类型

低级调度的基本类型是指调度策略选择下一个运行进程的时间点和仲裁方式，在其他时间不能改变已获得处理器的进程的分派情况。据此，低级调度有两类基本的调度方式。

第一类称剥夺式(preemptive)：又称抢先式。当进程/线程正在处理器上运行时，系统可根据所规定的原则剥夺分配给此进程/线程的处理器，并将其移入就绪队列，选择其他进程/线程运行。有两种常用的处理器剥夺原则，一是高优先级进程/线程可剥夺低优先级进程/线程；二是当运行进程/线程的时间片用完后被剥夺，在动态改变进程/线程优先级的系统中，经常会出现这种情况。

第二类称非剥夺式(nonpreemptive)：又称非抢先式。一旦某个进程/线程开始运行后便不再让出处理器，除非此进程/线程运行结束，或主动放弃处理器，或因发生某个事件而不能继续执行。

Yield CPU

剥夺式策略的开销比非剥夺式策略的开销要大，主要是调度程序自身的开销，及主存和磁盘对换程序、数据的开销，但是由于可以避免进程/线程长时间地独占处理器，对于实时系统和分时系统有利，能为用户提供高性能的服务。很多操作系统使用两种调度策略的组合，内核关键程序是非剥夺式的，用户进程是剥夺式的。

2.8.2 作业调度和低级调度算法

在操作系统中，存在多种调度算法，有的算法仅适用于作业调度，有的算法仅适用于进程/线程调度，但大多数调度算法对两者都适用。

1. 先来先服务算法

先来先服务算法(First Come First Served，FCFS)按照作业进入系统后备作业队列的先后次序来挑选作业，先进入系统的作业将优先被挑选进入主存，创建用户进程，分配所需资源，然后，

不少早期的RTOS是非剥夺式的。例如，早期的vxWorks。

在调度方面，有两个重要的考虑：死锁（DeadLock）和饿死（Starving）。都是调度最不希望看见的。

平均作业周转时间 $T=(20+(25-10)+(40-5)+(50-15))/4=26.25$ ms

平均带权作业周转时间 $W=(20/20+15/5+35/15+35/10)/4\approx2.46$ ms

可见 HRRF 算法的性能界于 SJF 算法和 FCFS 算法之间。HRRF 算法同样可用于进程调度，此时，响应比可定义为：

http://en.wikipedia.org/wiki/Priority_inversion

1＋进程等待处理器时间/进程估计计算时间

5. 优先级调度算法

优先级调度算法根据确定的优先级来选取进程/线程，总是选择就绪队列中的优先级最高者投入运行。在进程/线程的运行过程中，如果就绪队列中出现优先级更高的进程/线程，系统可预先规定策略：非剥夺式或剥夺式。前者让当前进程/线程继续运行，直到它结束或出现等待事件而主动让出处理器，再调度另一个优先级高的进程/线程运行；后者则立即重新调度，剥夺当前运行进程/线程所占有的处理器，分配给更高优先级的进程/线程使用。

规定用户进程/线程优先级的方法多种多样，一种方法是用户提出优先级，称作外部指定法，有的用户为了尽快获得计算结果，就设法提高其进程/线程的优先级，系统可规定优先级越高所需付出的费用就越多，以施加限制；另一种方法是由系统综合考虑有关因素来确定进程/线程的优先级，称作内部指定法。例如，根据进程/线程类型、空间需求、运行时间、打开文件数、I/O 操作多少、资源申请情况等来确定，确定优先级时各因素所占的比例应根据系统设计目标来分析这些因素在系统中的权重而定。

优先级通常用 0～4 095 的整数表示，这个整数叫做优先数，优先级与优先数之间的关系因系统而异，如在 UNIX/Linux 系统中，优先数越小，优先级越高，而有些系统做出相反的规定。

进程/线程优先级的确定可采用静态和动态两种方式。静态优先级在进程/线程创建时即确定，且生命周期中不再改变，可按照外部指定和内部指定方法计算静态优先级。静态优先级算法的实现简单，但会产生饥饿现象，使某些低优先级进程/线程无限期地被推迟执行。动态优先级使各进程/线程的优先级随时间而改变，基本原则是：正在运行的进程/线程随着占有 CPU 时间的增加，逐渐降低其优先级；就绪队列中等待 CPU 的进程/线程随着等待时间的增加，逐渐提高其优先级，等待时间足够长的进程/线程会因其优先级不断提高而被调度运行，克服了静态优先级的饥饿问题。

美国NASA登陆火星的系统曾经出现优先级翻转的事故。导致火星登陆器反复重启。

6. 轮转调度算法

轮转法调度（Round-Robin，RR）也称时间片调度，具体做法是：调度程序每次把 CPU 分配给就绪队列首进程/线程使用规定的时间间隔，称为时间片，通常为 10 ms～200 ms，就绪队列中的每个进程/线程轮流地运行一个时间片，当时间片耗尽时，就强迫当前运行进程/线程让出处理器，转而排列到就绪队列尾部，等候下一轮调度，所以，一个耗时型进程/线程需要经过多次轮转才能完成。实现这种调度需要使用间隔时钟，进程/线程开始运行时，就将时间片的值置入间隔时钟内，发生间隔时钟中断时，表明连续运行且用光时间片，此时，时钟中断处理程序通知处理器调度进行进程/线程切换。RR 调度策略可以防止那些很少使用设备的进程/线程长时间地占用处理器，导致要使用设备的那些进程/线程没有机会去启动设备。

例如，饿死：低优先级的任务一直无法得到CPU，一直有高优先级别的任务持续不断的占有CPU。这叫做饿死（Starving）

通过时钟中断，例如。

对一些突发事件，需要迅速的调度某一个任务，通常采用的调度方法是：Sporadic Scheduling

轮转法调度是一种剥夺式调度，系统耗费在进程/线程切换上的开销比较大，这个开销与时间片的大小有关。如果时间片取值太小，将导致大多数进程/线程都不可能在一个时间片内运行完毕，就会频繁切换，开销显著增大，所以，从系统效率的角度来看，时间片取得大一点好；另一方面，时间片的长度较大，那么，随着就绪队列中进程/线程数目的增加，轮转一次所耗费的总时间加长，即对每个进程/线程的响应速度均放慢，甚至时间片大到让进程/线程足以完成其所有任务，RR 调度算法便退化成 FCFS 算法。为了满足用户对响应时间的要求，要么限制就绪队列中的进程/线程数量，要么采用变化的时间片长度，根据当前负载状况，及时调整时间片大小。所以，确定时间片长度要从进程数目、切换开销、系统效率和响应时间等多方面加以考虑。

Windows 2003 时间片由注册表的一个记录项控制，短时间片持续 2 个时钟滴答，用于客户端；长时间片持续 12 个时钟滴答，能够减少线程调度上下文切换的次数，用于服务器端，使得响应性能和吞吐能力更好。可以设置所有线程为定长时间片，也可以把后台和前台线程的时间片设置为不同的值，后者的时间片可取 2～12 个时钟滴答(以 2 为增量)。

7. 多级反馈队列调度算法

多级反馈队列调度(Multi－Level Feedback Queue，MLFQ)又称反馈循环队列，其主要思想是：由系统建立多个就绪队列，每个队列对应于一个优先级，第一个队列的优先级最高，第二个队列的优先级次之，其后队列的优先级逐个降低。较高优先级队列的进程/线程分配给较短的时间片，较低优先级队列的进程/线程分配给较长的时间片，最后一个队列进程/线程按 FCFS 算法进行调度。处理器调度每次先从第一个队列中选取执行者，同一队列中的进程/线程按 FCFS 原则排队，只有在未选到时，才从较低一级的就绪队列中选取，仅当前面所有队列为空时，才会运行最后一个就绪队列中的进程/线程。

进程/线程的优先级可事先规定。例如，使用设备频繁者优先级高，计算型进程/线程的优先级低；终端用户定为高优先级，非终端用户定为低优先级。优先级也可事先不规定，一个新进程/线程被创建后，首先移入第一个就绪队列末尾等待调度，如果能够在给定的时间片内完成，便可从系统中撤离；凡是耗尽时间片仍未运行结束的，就移入下一个就绪队列的末尾，直至进入最低优先级就绪队列。

MLFQ 调度算法具有较好的性能，能满足各类应用的需要。对于分时交互型短作业，系统通常可在第一队列(最高优先级队列)规定的时间片内完成工作，使终端型用户感到满意；对于短的批处理作业，通常只需在第一队列和第二队列中各执行一个时间片就能完成工作，周转时间仍然很短；对于长的批处理作业，它将依次在第一队列、第二队列、第三队列等各个队列中获得时间片运行。例如，在 XDS940 操作系统中，把进程划分为 4 个优先级类，分别是终端、I/O 操作、短时间片和长时间片；当一个等待终端输入的进程被唤醒时，它排入最高优先级类(终端类)；当一个等待磁盘数据传输的进程就绪时，它将排入第二类优先级队列；当进程在时间片用完时仍处于就绪态时，它将排入第三类优先级队列；如果一个进程多次用完时间片而从未因终端或其他 I/O 操作的原因发生阻塞，它将被排入第四类优先级队列，即最低优先级类。

MLFQ 调度算法会导致“饥饿”问题。假如有一个长作业进入系统，它最终必将移入优先级

一个应用系统的设计应该是，event driven。没有事情的时候，任何线程不能去空转。应该block等待在一个事件上。

或者是避免高优先级别的任务LOOP的在运行。应该block在事件上。

最低的就绪队列中，如果其后有源源不断的短作业进入系统，且形成稳定的作业流，则长作业一直等待，陷入"饥饿"状态。解决此问题的一种有效方法是对于低优先级队列中等待时间足够长的进程提升其优先级，从而让它获得运行机会。如图 2.27 所示是三级反馈队列调度算法示例。

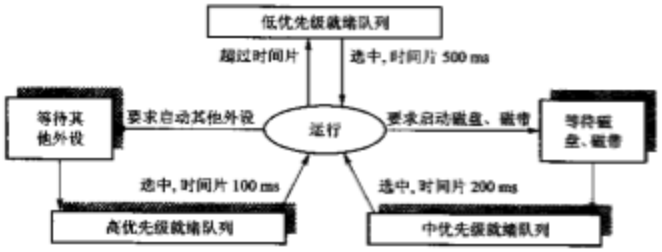


图 2.27 三级反馈队列调度算法示例

8. 彩票调度算法

一种可给出预见的结果、实现起来也较为简单的调度算法称为彩票调度算法（lottery scheduling），其基本思想是：为进程/线程发放针对各种资源（如 CPU 时间）的彩票，当调度程序需要做出决策时，随机选择一张彩票，彩票的持有者将获得相应的系统资源。对于 CPU 调度，系统可能每秒钟抽取彩票 50 次，中奖者每次可获得 20 ms 的运行时间。

在这种情况下，所有进程/线程都是平等的，它们享有相同的运行机会。如果某些进程/线程需要更多的机会，就可给予其额外的彩票，以增加中奖机会。如果发出 100 张彩票，某进程拥有 20 张彩票，它就有 20% 的中奖概率，也将获得大约 20% 的 CPU 时间。

彩票调度与优先数调度不同，后者很难说明优先数为 40 终将意味着什么，而前者则很清楚，进程拥有多少彩票份额，就能获得多少资源。彩票调度算法具有有趣的特性：其反应非常迅速，例如，新进程被创建并得到一些彩票，那么在下次抽奖时，这个进程的中奖机会就立即与其所持有的彩票成正比。此外，协作进程之间可交换彩票，客户进程向服务器进程发送一条消息并阻塞，客户进程可以把所持有的彩票全部交给服务器进程，以增加后者下一次被选中运行的机会；服务器进程在完成响应服务之后，又将彩票返还给客户进程使其能够再次运行。实际上，在没有客户时，服务器进程/线程根本不需要彩票。

彩票调度算法还可用来解决其他算法难以解决的问题。例如，一个视频服务器有若干将视频信息传送给相应客户的进程，假设它们分别需要 10 帧/秒、20 帧/秒和 25 帧/秒的传输速度，则分别给这些进程分配 10、20 和 25 张彩票，它们将自动按照正确的比例获得 CPU 资源。

2.8.3 实时调度算法

1. 实时操作系统的特性

实时系统是那些时间因素起关键作用的系统，如监控系统、自动驾驶系统、安全控制系统等，

错过时间，有重大影响的。

等于1，就是CPU忙的恰恰好。没有一点空闲。

在这些系统中，迟到的响应即使正确，也同没有响应一样糟糕。实时系统中存在若干实时进程或任务，用来响应外部事件，由于存在时间上的紧迫性，对任务调度提出特殊的要求。实时任务分为硬实时(hard real time)任务和软实时(soft real time)任务，前者意味着必须满足任务的开始截止期限或完成截止期限；后者意味着任务与预期截止时间关联，但偶尔超出截止时间的限制还是可以容忍的。实时进程的行为预先可知，处理周期都很短，当检测到外部事件时，调度程序会按照满足其截止期限的方式调度这些任务。按照任务的执行是否呈周期性来划分，实时任务可分为周期性任务和非周期性任务，前者指以固定的时间间隔出现的事件；后者指事件的出现无法预计，但事先规定开始截止期限或完成截止期限。

一个系统可能有多个周期性事件的事件流，都要求系统做出及时的响应，系统能否及时处理所有事件取决于每个任务的处理时间。例如，有 m 个周期性任务，任务 i 的出现周期为 P_i，处理所需要的 CPU 时间为 C_i，则给出条件：

$$\boxed{\sum_{i=1}^{m} \frac{C_i}{P_i} \leqslant 1}$$

满足这一条件的实时系统称为任务可调度的。举例来说，一个实时系统要处理 3 个事件，其出现周期分别为 100 ms、200 ms 和 500 ms，如果事件的处理时间分别为 50 ms、30 ms 和 100 ms，则这个系统是任务可调度的，因为

$$0.5+0.15+0.2\leqslant 1$$

如果加入周期为 1 s 的第 4 个事件，只要其处理时间不超过 150 ms，此实时系统仍将是任务可调度的。当然，隐含条件是进程切换的时间足够短，可以忽略不计。

2. 实时调度算法

实时调度算法分为动态实时调度和静态实时调度两类，前者在运行时做出调度决定；后者在提供截止期限等信息的前提下，在系统开始运行之前完成调度决策。下面介绍几种经典的实时调度算法。

Soft RT不一定慢。是指可以错过Deadline。没关心。是一种调度方式。

(1) 单比率调度算法

在实时系统中，事件是周期性发生的，单比率调度是视周期长度而定的抢占式策略：周期越短，优先级越高。在具体实现时，可事先为每个实时进程/线程分配与事件发生频率成正比的优先数，运行频率越高的进程/线程其优先数越高。例如，周期为 20 ms 的进程/线程的优先数为 50，周期为 100 ms 的进程/线程的优先数为 10，运行时调度程序总是调度优先数最高的就绪进程/线程，并采取抢占式分配策略。

(2) 限期调度算法

当一个事件发生时，对应的实时进程/线程就被加入就绪队列，此队列按照截止期限排序，对于周期性事件，截止期限即为事件下一次发生的时间。限期调度算法首先运行队首的截止期限最近的那个进程/线程；新进程/线程就绪后，系统检测其截止期限是否比当前运行者早，以决定是否剥夺当前运行进程/线程的资源。

(3) 最少裕度法

Hard RT：不一定是快。是指系统调度不能（会）错过Deadline时间要求。会确保。这就是硬实时系统。

首先计算各个进程/线程的富裕时间，即裕度(laxity)，然后选择裕度最少者执行。计算公式为：

$$裕度 = 截止时间 - (就绪时间 + 计算时间)$$

裕度越小说明进程/线程越紧迫，就绪后令其尽快运行。

2.8.4 多处理机调度算法

单处理机调度和多处理机调度存在一定的区别，现代操作系统往往采用进程调度与线程调度相结合的方式来完成多处理机调度。

1. 多处理机调度的设计要点

多处理机调度的设计要点有3个：为进程分配处理机、在单个处理机上是否使用多道程序设计技术和实际指派进程的方法。所需考虑的这些问题与可用处理机数目及并行处理的应用程序之间的同步频率密切相关。

(1) 为进程分配处理机

假定多处理器系统中的处理机都是同构的，可将所有处理机放入处理机池，采用静态或动态方式分配。静态分配方式把进程分配到一个处理机上，每个处理机对应于一个就绪队列，其优点是调度开销小，缺点是易造成处理机忙闲不均；动态分配方式让所有处理机共用一个就绪队列，当某处理机空闲时，就选择一个进程占有此处理机运行，这样，进程可以在不同时间运行于不同的处理机上。

无论采取哪种分配策略，系统必须提供相应的分配和调度机制。在主从式结构中，只使用一个调度程序，它运行在主控机上，考虑所有处理机和所有符合条件的进程。这种方式简单易行，调度效率高，但系统的坚定性与主控机上所运行的操作系统之间的关系过大，极易成为系统性能瓶颈。在对等式结构中，不同的处理机有不同特性，各处理机自行实施调度，且可以使用不同的调度程序，从就绪队列中选出进程运行，但必须保证两个处理机不能选中同一个进程，操作系统的调度过程比较复杂。

(2) 在单个处理机上是否使用多道程序设计技术

当有多个处理机可用时，使单个处理机处于繁忙状态不是那么重要，系统追求的是为应用程序提供最好的平均性能。如果一个应用程序由多线程组成，应该让所有线程同时在多处理机上运行，而不是在单个处理机上多道运行，这时性能最佳。

(3) 实际指派进程的方法

大量实验数据证明，随着处理机数目的增多，采用复杂的低级调度算法来调度会降低有效性，这时往往为进程设置一个就绪队列或按照优先数排列的多个就绪队列，采用最简单的FCFS调度算法或优先数算法。这样做不但实现起来简单，而且系统开销很低。

2. 多处理机调度算法

下面讨论几种经典的多处理机进程/线程调度算法。

(1) 负载共享调度算法

现代multi CPU系统许多采用CPU Affinity的调度方式。就是把task固定分配指某个或者某些CPU上。

这种模型通常是追求最大程度的并行。都是while（T）的任务执行体。一个循环。不sleep。通信系统许多是这样。

负载共享调度算法(load sharing)的基本思想是:进程并不被指派到特定的处理机上,系统维护全局性进程就绪队列,当处理机空闲时,就选择进程的一个线程去运行。

此算法的优点是:把负载均分到所有的可用处理机上,保证没有处理机是空闲的;无须集中调度,调度程序可以运行在不同的处理机上,可采用优先级等方法组织全局队列。此算法的缺点是:就绪队列必须被互斥访问,当系统拥有很多处理机时,将成为性能瓶颈;被剥夺线程很难在原处理机上运行,恢复高速缓存信息会带来系统性能的下降;所有线程都被放在一个公共的池中,一个进程的所有线程未必能同时获得处理机,而如果其线程间的同步频繁,那么进程的切换次数增多将导致系统性能下降。

处理器资源池

具体的负载共享调度算法有:先来先服务、最少线程数优先和剥夺式最少线程数优先。著名的 Mach 操作系统包括一个全局共享的就绪线程队列,而且每个处理机还对应于一个局部的就绪线程队列,其中含有临时绑定到此处理机上的就绪线程,处理机调度时首先在局部就绪线程队列中选择绑定线程,如果没有,才到全局就绪线程队列中选择未绑定线程。

(2) 群调度算法

群调度算法(gang scheduling)的基本思想是:基于一对一的原则,一群相关线程被同时调度到一组处理机上运行。紧密相关线程的并行执行能够减少同步阻塞,从而减少进程切换,降低调度代价,系统性能得到提高。

群调度引发对处理机分配的要求。如果有 N 个处理机和 M 个应用进程,每个应用进程至多有 N 个线程,那么,每个应用进程将分得 N 个处理机的可用时间的 $1/M$,此分配策略的效率可能并不高。考虑一个例子,有 4 个处理机和两个应用进程,应用进程 1 有 4 个线程,应用进程 2 有 1 个线程,使用均分时间分配的方法,每个应用进程可获得 50% 的 CPU 时间,由于应用进程 2 只有 1 个线程运行,将有 3 个处理机空闲,所浪费的 CPU 资源为 37.5%。可以采用线程数加权调度法,具体来说,为应用进程 1 分配 80% 的 CPU 时间,为应用进程 2 分配 20% 的 CPU 时间,则处理机时间的浪费可降至 15%。

(3) 专用处理机调度算法

Affinity的模型

专用处理机调度算法(dedicated processor assignment)的基本思想是:将同属一个进程的一组线程同时分派到一组处理机上运行,每个线程均获得一个处理机,专用于处理这个线程,直到进程运行结束。这是群调度的一种极端形式。

采用专用处理机调度算法,处理机将不适用于多道程序设计,当一个线程因等待事件而阻塞后,其所专用的处理机处于空闲状态。此算法所追求的是通过高度并行来达到最快的执行速度,它在应用进程的运行过程中避免低级调度和切换。对于拥有几十或数百个处理机的高度并行系统而言,单个处理机只占系统总代价的一小部分,处理机的使用效率不再是衡量有效性或系统性能的重要因素。

(4) 动态调度算法

在实际应用中,一些应用程序通过语言或工具能够动态地改变进程中的线程数目,这就要求操作系统能够调整负载以提高处理机利用率。动态调度算法(dynamic scheduling)的基本思想

让相关的tasks同时在处理器池里获得CPU。从而可以最大的支持应用程序。支持一个“群”task。

[注释:]1. 调度算法学习中要注意“死锁”（deadlock）与”饿死“（Starving）的区别。2. Hard Realtime不一定调度快是指调度系统保证不错过Deadline；Soft RT不一定慢是指调度系统保证错过Deadline。 3. 有兴趣的同学看看NASA的Mars Lander的优先级翻转的事故。

是：由操作系统和应用进程共同做出调度决策，操作系统负责在应用进程之间分配处理机；应用进程在所分配的处理机上执行可运行线程的子集，哪些线程应该执行？哪些线程应该挂起？完全是应用进程的任务，可借助于运行时库函数完成。

在动态调度算法中，当一个进程请求一个或多个处理机时，操作系统的调度程序按照下面的原则进行处理。

① 如果有空闲处理机，则满足请求；

② 否则，对于新创建的进程，从当前分得多个处理机的任一进程中收回一个处理机，把它分配给新进程；

③ 如果这个请求的任何分配都不能被满足，则此进程保持未完成状态，直到有处理机可用或此进程取消请求；

④ 当释放一个或多个处理机时，对尚未处理过的请求处理机的进程队列进行扫描，且为队列中尚未分配处理机的每个进程分配一个处理机，然后，再次扫描进程队列，按照先来先服务原则分配其余处理机。

2.9 Linux 调度算法

2.4与2.6差别太大。主要是因为SMP和多核的冲击，使得之前的调度算法不scaling。

2.9.1 Linux 传统调度算法

Linux 2.4 内核调度函数 schedule()决定采用何种调度策略选择就绪进程运行，确保进程所获得的 CPU 访问时间和其指定的优先级及进程类型相匹配，消除进程对 CPU 资源的饥饿状态。调度机制由内核实现，包括调度时机的选择、调度参数的计算等工作。

1. 进程调度策略

在进程 task _ struct 结构中，存放着与进程调度有关的重要成员供进程调度使用。

(1) policy：标识进程的调度策略，有以下三种类型。

① SCHED _ OTHER 普通类任务：普通进程采用基于优先级的时间片轮转法调度，只要有实时进程就绪，这类进程便不能运行。

② SCHED _ FIFO 先进先出实时类任务：进程一直运行，除非出现等待事件或有另一个具有更高 rt _ priority 的实时进程出现，才让出处理器。

③ SCHED _ RR 轮转法实时类任务：除了时间片是定量之外，与 SCHED _ FIFO 类似，在时间片耗尽之后，使用相同的 rt _ priority 排到原队列的队尾。

(2) priority 进程静态优先级：内核 V2.4 及以上版本已取消此变量。

(3) nice 进程可控优先级因子：其值是 −20～19 的整数，可用于改变进程的静态优先级。其默认值为 0，增加 nice 值会降低进程的优先级。

(4) rt _ priority 实时进程静态优先级：仅被实时进程所使用，是 0～99 的一个整数，用于区分实时进程的等级，权值较高的进程总是优先于权值较低的进程。rt _ priority + 1000 给出实时

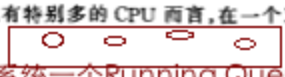
建议新学Linux的同学不要再接触2.4调度算法。
直接从2.6开始。2.6的Kernel开始是kernel剥夺式的调度了。

2.4之前（包括2.4），只有一个调度队列。所有的READY任务在一个Queu里。这就是常说的需要O（n）的复杂度来选择下一个运行任务。

switch _ to( )函数可进行进程切换，改由 next 进程占用处理器，切换代码与处理器密切相关，用汇编代码来实现，按照 task _ struct 结构中的相关信息进行进程上下文切换；如果当前进程或新进程使用虚拟主存，那么相关页表必须同步更新。

4. SMP 进程调度

Linux 2.0 版开始支持 SMP，多处理器系统中的每个 CPU 上都可以运行一个进程，因此，在实施进程调度时，可供选择的 CPU 不止一个。在每个进程的 task _ struct 结构中都含有当前所用的 CPU 编号及刚用过的 CPU 编号，虽然系统并不限制进程每次必须在同一个 CPU 上运行，但可以利用 processor _ mask 掩码限制进程只能在某些 CPU 上运行。如果掩码的第 n 位设置为 1，则此进程可在第 n 个 CPU 上运行，即进程可与某个 CPU 绑定，称为亲缘关系。在实施进程调度时，总是优先考虑进程在其刚用过的 CPU 上运行，因为此 CPU 的高速缓存中最有可能包含此进程的一些特定数据，若改变 CPU 则会增加对存储器的访问时间，在一定程度上影响系统效率。然而，亲缘关系的使用必须特别仔细，在某个进程限制于特定 CPU 而无其他进程可在此 CPU 上执行时，CPU 亲缘关系很有意义。但是这仅针对系统有特别多的 CPU 而言，在一个双 CPU 计算机系统中使用亲缘关系肯定太过奢侈。



CPUs

系统一个Running Queue

Running Queue

2.9.2 Linux 2.6 调度算法

从 Linux 2.5 内核版本开始使用新的称为 $O(1)$ 的调度程序，其特点是能够保证无论系统负载（进程数目或处理器数目）怎样增加，选择合适的进程且为其分配处理器的时间是恒定的。此外，还具有新的特性：支持 SMP，每个处理器都拥有可运行队列；强化 SMP 的亲和力，尽量将相关任务分配到一个处理器上连续运行；确保响应时间，及时地调度交互式进程；保证公平性，无进程处于饥饿状态。进程描述符 task _ struct 中含有与调度有关的成员是：policy、static _ prio、prio、rt _ priority、sleep _ avg、time _ slice 和 load _ weight 等。

发挥Cache缓存，提高性能

1. 调度算法的数据结构

调度程序中最主要的数据结构是可运行队列（runqueue），它是给定处理器上的就绪进程链表，每个处理器都有可运行队列，每个就绪进程都归属于一个可运行队列。此外，可运行队列还包含每个处理器的调度信息，所以也是处理器的重要数据结构。

```
struct runqueue {
    spinlock _ t lock;                    //可运行队列自旋锁
    unsigned long nr _ running;           //总的就绪进程数
    unsigned long nr _ switches;          //上下文切换数
    unsigned long expired _ timestamp;    //队列最后被对换时间
    unsigned long nr _ uninterruptible;   //不可中断阻塞态的任务数
    struct task _ struct * curr;          //处理器当前运行进程
    struct task _ struct * idle;          //处理器的空闲进程
    struct mm _ struct * prev _ mm;       //最后运行任务的 mm _ struct
```

每一个CPU对应一个这个runqueue object

CPU

Running Queue

某一个优先级的task存在READY可被调度，相应的bitmap的bit会被set_bit()置位。


```
    struct prio_array *active;                  //指向活动优先级数组的指针
    struct prio_array *expired;                 //指向过期优先级数组的指针
    struct prio_array array[2];                 //活跃、过期两类优先级数组
    int prev_cpu_load[NR_CPUS];                 //处理器的负载
    struct task_struct *migration_thread;       //处理器上的移出线程
    struct list_head *migration_queue;          //处理器上的移出队列
    atomic_t *nr_iowait;                        //等待 I/O 操作的进程数
};
```

在系统中，内核为每个可运行队列（处理器）维持两个调度使用的数据结构，称为优先组。其中，一个是活跃的，另一个是过期的。

每一个优先级对应一个queue。系统可以有140个queue。

```
struct prio_array
{
    unsigned 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
};
```

140

优先级数组使每一种优先级都包含一个相应的队列，这些队列包含对应优先级上的就程链表；优先级数组还拥有一个优先级位图，当需要查找当前系统内拥有最高优先级的就绪时，有助于提高查找效率。

O（1）

MAX_PRIO 定义优先级的个数，其默认值是 140；BITMAP_SIZE 是优先级位图数组的大长 32 位，若每一位代表一个优先级，140 个优先级需要 5 个长整型数来表示，所以位图含有 5 组项，共 160 位（20 位未用），位图指示哪些队列是空的；nr_active 指示此优先级数组内就绪进个数。在初始化时，位图被设置为 0，且所有可运行队列都为空，当一个进程就绪时，将它放入的优先级队列，位图中的相应位置 1，这样，查找系统中的最高优先级就变成查找位图中被设置一个位。因为优先级的个数是定值，所以查找时间恒定，并不受系统所含进程数的影响，Linux 所支持的每一种体系结构都提供相应的快速查找算法 sched_find_first_bit()。

2. 进程优先级和时间片

通过特殊汇编指令，可以一个指令判断一个32位整数的哪个bit是否是1。

Linux 2.6 内核实现基于优先级的进程调度算法，优先级高的就绪进程先运行，优先低的就绪进程后运行，优先级相同的进程以轮转方式运行。由于实时进程的优先级总高于非实时所以仅当没有实时任务时，才轮到非实时任务运行。

在优先级队列中，实时进程具有静态优先级，优先级范围为 0～99，不做动态优先级的改SCHED_FIFO 实时任务没有时间片，以 FIFO 方式运行；如果一个 SCHED_FIFO 实时任务阻塞，当它被唤醒时，将返回到相同优先级活动可运行队列中；SCHED_RR 实时进程分得时片，时间片用尽后，返回原先的优先级队列，从不移至过期队列。

每个非实时任务都有静态优先级

指定后就不能改变。调度程序所用到的动态优先级存放在 prio 域中，它通过一个关于静态优先级和进程交互性的函数关系计算而来。

effective_prio()函数可以返回一个进程的动态优先级，它以 nice 值为基数，再加上 -5～+5 之间的进程交互性奖励或罚分。例如，对于交互性很强的进程，即使其 nice 值为 10，它的动态优先级最终也有可能达到 5；相反地，对于温和的处理器吞噬者，虽然 nice 值原本是 10，它最后的动态优先级却可能是 12；交互性不强不弱的进程不会得到优先级奖励，同样也不会被罚分，其动态优先级与 nice 值相当。

如果进程大部分时间都在睡眠，它属于 I/O 消耗型；如果进程的执行时间比睡眠时间长，它属于处理器消耗型。Linux 记录进程用于睡眠和执行的时间，此值存放在 task_struct 的 sleep_avg 域中，范围从 0 到 MAX_SLEEP_AVG，默认值为 10 ms。当一个进程从睡眠状态恢复到运行态时，sleep_avg 会根据睡眠时间的长短而增加，直至达到 MAX_SLEEP_AVG 为止；相反地，进程每运行一个时钟滴答，sleep_avg 就相应地递减，直至 0 为止。

上述计算不仅基于睡眠时间，而且运行时间也要被计算进去，所以，尽管某进程的睡眠时间很长，但是如果它把自己的时间片用得一干二净，就不会得到优先级奖励。这种推断机制不仅会奖励交互性强的进程，还会惩罚处理器资源耗费量大的进程，如果某个进程发生变化，开始大量地占用处理器时间，它很快就会失去曾经得到的优先级提升。

由于动态优先级已经以 nice 值和交互性为基础来取值，重新计算时间片就相对较简单，在进程创建时，子进程和父进程平均分配父进程的剩余时间片，这样的分配方法很公平，可以防止用户通过不断创建新进程来不停地攫取时间片。然而，当一个进程的时间片用完后，就要根据进程的动态优先级重新计算时间片。task_timeslice()函数为给定任务返回一个新的时间片，时间片的计算只需将优先级按比例进行缩放，使其符合时间片的数值范围要求，进程的优先级越高，它每次执行所得到的时间片就越长。优先级最高的进程所能获得的最大时间片长度(MAX_TIMESLICE)是 800 ms；优先级最低的进程所获得的最短时间片(MIN_TIMESLICE)是 5 ms；默认优先级(nice=0，没有交互性奖励或罚分)的进程所得时间片长度为 100 ms。

调度程序还提供另外一种机制以支持交互式进程。如果进程的交互性非常强，当其时间片用完之后，会再次被放置到活跃数组而不是过期数组中。重新计算时间片是通过活跃数组与过期数组之间的切换来进行的，进程在用尽时间片后，通常会被移至过期数组，当活跃数组中没有剩余进程时，这两个数组就会被交换。活跃数组变成过期数组，过期数组替代活跃数组，对换操作提供时间复杂度为 $O(1)$ 的时间片重新计算。在发生对换之前，交互性很强的某进程有可能处于过期数组中，当它需要交互时却无法执行，因为必须等待数组发生交换，而将交互式进程重新插入活跃数组可以避免这类问题。

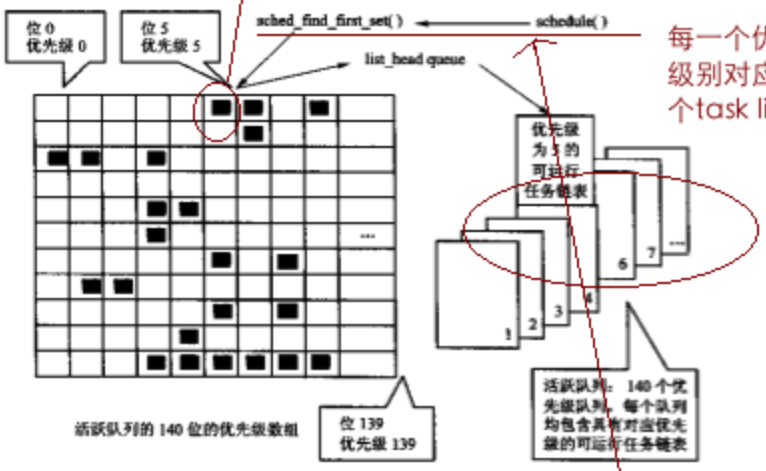
3. $O(1)$ 调度算法

调度算法由 schedule()函数实现，既简单又有效。对于给定的处理器，调度程序选择优先级活跃数组中第一个被设置的位，对应于优先级最高的就绪进程队列，然后，选择此优先级链表中的表头进程，它就是要被调度执行的进程。如图 2.28 所示是 Linux 2.6 处理器的调度数据结构

sched_find_first_bit()

在优先级位图里，有可运行任务的最高优先级的bit会被SET。

示意图，其中仅画出活跃队列的优先级数组及相应的优先级队列；过期队列的优先级数组及相应的优先级队列完全类似。新调度程序减少对循环的依赖，活跃数组内的可执行队列上的进程都有剩余时间片；过期数组内的可执行队列上的进程都耗尽时间片。当某进程的时间片用完时，重新计算它的时间片，并移至过期数组。因为数组是通过指针进行访问的，活跃和过期数组之间来回切换所用的时间就是交换指针所需要的时间。



每一个优先级别对应一个task linklist。

图 2.28 Linux 2.6 处理器调度数据结构图

schedule()函数进入调度，寻找和调度最高优先级进程的执行过程则由以下代码段实现。

两个数组。如果active数组是zero，说明运行任务时间片都用完了。直接换成expired的数组。

```
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)) {       //运行队列的 active 变成空
    rq->active = rq->expired;            //活跃、过期数组切换
    rq->expired = array;
    array = rq->active;
```

sched_find_first_set()会在第一个100个bit里找。100:是Linux Kernel 实时进程的最高优先级别。 0-99是RT Task。100-140是非实时。

相应的优先级的任务队列的队头。

这个时候不能开中断，因为找调整重要数据结构。

```
    ...
  } 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
```

4. 负载平衡

系统中的每个处理器都有自己的运行队列，只对属于自己的进程进行调度。如果运行队列出现负载不均的情况，则由负载平衡程序 load_balance 来解决这个问题。有两种调用方法，在执行 schedule()函数时，只要当前可运行队列为空就会被调用，找到一些就绪进程并插入这个队列中；负载平衡程序还会被定时器调用，每隔适当的时间调用一次，这时需要解决所有运行队列间的失衡问题。

load_balance()函数工作时需要锁住当前处理器的运行队列并屏蔽中断，避免出现竞争，负载平衡任务完成后，解除对当前运行队列的锁定并开放中断。具体的操作步骤如下。

(1) 找到最繁忙的运行队列，此队列中的就绪进程数目最多，如果没有其他运行队列中的就绪进程数目比当前运行队列中的就绪进程数目多25%或更多，就结束平衡负载处理。

(2) 找到最繁忙的运行队列，从中选择一个优先级数组以便抽取就绪进程，最好是过期数组，因为那里的就绪进程已经有相对较长的时间未曾运行，很可能不在处理器的高速缓存中。如果过期数组为空，就只能选择活跃数组。

(3) 找到含有进程且优先级最高的链表，把优先级高的就绪进程平均分散开来。

(4) 分析所找到的所有优先级相同的就绪进程，选择一个未运行、不会因为处理器相关性而不可移动且不在高速缓存中的进程。如果有满足这些条件的进程，便将其从最繁忙的队列中抽取至当前可运行队列。

在不同的CPU之间均衡任务。避免某个处理器超载。

(5) 只要可运行队列之间的负载仍然不均衡，就重复上述步骤，继续从繁忙队列中抽取进程到当前可运行队列，最终消除不平衡局面。

换言之，只要是可能产生不一致的，例如，多个CPU同时存取，中断存取，就应该上lock。

5. 用户抢占和内核抢占

内核即将返回用户空间时，如果 TIF_NEED_RESCHED 标志被设置，将导致 schedule()被调用，此时发生用户抢占。这时内核知道自己是安全的，既可继续执行当前进程，也可选择新进程运行。用户抢占通常发生在：从系统调用返回用户空间时；从中断处理程序返回用户空间时。

Linux 2.6 完整地支持内核抢占，只要重新调度是安全的，内核就可以在任何时间抢占正在运行的任务。那么，重新调度何时是安全的呢？只要没有持有锁，内核就可以进行抢占，锁是非抢占区域的标志。由于内核支持 SMP，所以，如果没有持有锁，正在运行的代码是可重入的，也就是可抢占的。

为了支持内核抢占而做的一处修改是为进程引入 preempt_count 计数器，其初值为 0，每当使用锁时此计数器的值加 1，释放锁时其值减 1。当计数器的值为 0 时，内核就可以执行抢占。

这个时候，内核的那些独有的数据结构都调整完毕。不会存在不一致性问题。

例如，clock或者counter中断。内核里许多场合，上开中断的。所以，这个时候内核任务可能会被剥夺。

从中断返回内核空间的时候，内核会检查 TIF_NEED_RESCHED 和 preempt_count 的值，如果 TIF_NEED_RESCHED 已被设置且 preempt_count 的值为 0，说明有更为重要的任务需要执行并且可安全地抢占，此时就会调用调度程序。如果 preempt_count 的值不为 0，说明当前任务持有锁，抢占是不安全的，这时就会如常直接从中断返回当前运行进程。如果当前进程所持有的所有锁已经被释放，那么 preempt_count 的值会重新为 0，此时，释放锁的代码会检查 TIF_NEED_RESCHED 是否被设置，如果已被设置，就会调用调度程序。内核抢占发生时刻如下：

当前进程：在核心态下，例如，系统调用呢。

(1) 当从中断处理程序返回内核空间的时候；

(2) 当内核代码再一次具有可抢占性的时候；

(3) 如果内核中的任务显式调用 schedule()；如果内核中的任务阻塞，也会导致调用 schedule()。

有些内核代码需要允许或禁止内核抢占，这可以通过 preempt_disable() 和 preempt_enable() 实现。由于内核是可抢占的，内核中的进程在任何时刻都有可能停止以便另一个具有更高优先级的进程运行，这意味着一个任务与被抢占的任务可能会在同一个临界区内运行。为了避免发生这类情况，内核抢占代码使用自旋锁作为非抢占区域的标记，如果一个自旋锁被持有，内核便不能进行抢占。因为内核抢占和 SMP 面对相同的并发性问题，而且内核已经是 SMP 安全的，因此，这种简单的变化使得内核也是抢占安全的。

spinlock

2.10 Windows 2003 调度算法

说白了，在内核里，如果是spin lock在hold。一定不能剥夺。因为可能发生不一致现象了。

1. 线程调度及其特征

Windows 支持内核级线程，是一个基于优先级抢占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，线程可以在任何可用的处理器上运行，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 调度函数选择其所偏好的处理器。

Windows 在内核中实现线程调度代码，这些代码分布于内核中与调度相关的事件的出现位置，并不存在单独的线程调度模块。内核中完成线程调度功能的函数统称为内核调度器，线程调度出现在 DPC/dispatch 中断优先级，触发事件如下。

(1) 一个线程进入就绪态，如新线程刚被创建或线程刚刚结束等待态。

(2) 一个线程由于时间配额用完而从运行态转入退出态或等待态。

(3) 一个线程由于调用系统服务而改变优先级或被系统本身改变其优先级。

(4) 一个正在运行的线程改变其亲合处理器集合。

在这些事件出现时，系统选择下一个将要运行的线程。当选中一个新线程进入运行态时，将执行线程上下文切换，保存当前线程的运行环境，加载另一个线程的运行环境，并开始执行新线程。

处理器调度是严格地针对线程进行的，并不考虑被调度线程属于哪个进程。例如，进程 A 有 10 个可运行的线程，进程 B 有两个可运行的线程，若这些线程的优先级相同，则每个线程将得到

Kernel Preemptive的Checkpoint！

本章小结

中断和异常是激活操作系统的两种手段，中断装置识别并响应中断事件，通过交换 PSW 让

进程是管理单位；线程是调度单位。

中断和异常：可恢复的，不可恢复的。
可恢复的：处理完毕后，继续执行。
不可恢复的：系统需要reboot。

中断处理程序占有处理器，在处理完此中断事件后，通常会改变一些进程的状态，引起相应进程队列的调整，然后，返回被中断程序或转向低级调度执行，由于中断能够改变处理器内操作的执行顺序，它成为操作系统实现并发性的硬件基础之一；异常事件由硬件指令逻辑发现，其响应、处理和通回的过程与中断类似。中断按照事件来源和实现手段可分为硬中断和软中断，硬中断又分为外中断（中断、异步中断）和内中断（异常、同步中断）。异常可分为错误、陷阱、终止和编程异常。软中断又分为信号和软件中断。

进程是操作系统中最重要和最基本的概念之一，引入进程是系统资源的有限性和系统内操作的并发性所决定的。进程具有生命周期，由创建而产生，由调度而执行，由撤销而消亡，操作系统的基本功能是进程的创建、管理和撤销。为了实现对进程的管理，操作系统维护每个进程的资料结构——进程控制块，这是进程的唯一标志，创建进程必须为其创建进程控制块，撤销进程时系统便回收进程控制块。

进程是并发程序的执行，是动态的概念，但是为了在处理器上执行仍然需要静态描述。从结构上来看，进程由控制块、程序块、数据块和核心栈组成，进程的静态描述及其处理器执行环境合称为进程上下文，包括用户级上下文、系统级上下文、寄存器上下文。一个进程在运行过程中或执行系统调用，或产生中断事件，处理器都会进行模式切换，操作系统接收控制权，内核完成必需的操作之后，或恢复被中断进程或切换至新进程。当系统调度新进程占有处理器时，新老进程随之发生上下文切换，操作系统在完成必要的操作之后，可以恢复被中断的进程或切换至其他进程。

如果说操作系统中引入进程的目的是使多个程序并发执行，以便改善资源利用率和提高系统效率，那么，在操作系统中再引入线程，则是为了减少程序并发执行时所付出的时空代价，使得并发粒度更细、并发性更好。在多线程环境中，进程封装管理信息，线程封装执行信息；线程的实现有用户级线程、内核级线程和混合式三种方法。

处理器调度分为三级：高级调度、中级调度和低级调度。高级调度用于决定哪些满足资源需求的后备作业被选中进入主存去多道运行，FCFS、SJF、SRTF、HRRF、优先级调度算法等是常用的作业调度算法；低级调度用于决定选择哪个进程或线程占有处理器运行，FCFS、RR、优先数、MLFQ 等是常用的进程/线程调度算法；实时系统调度算法有：单比率调度、限期调度和最少裕度调度；多处理器调度算法有：负载共享调度、群调度、处理器专派调度和动态调度。衡量调度算法优劣的因素包括响应时间、周转时间、资源利用率、作业吞吐率和公平性等。

调度： 调度策略和调度机制要分离。是操作系统设计的重要原则。

[注释:] Linux 2.6的调度是一个分水岭：支持Kernel内部可剥夺。支持SMP下O（1）算法。忘却具体技术细节，把握：1. 只要数据结构不存在非一致性，就可以开中断和支持被剥夺调度。否则通过锁来保护。2. 多个Queue一定好于一个Queue。每个处理器有一个任务队列。