浅谈进程中断处理

王思琪 金燊

在 P7 中，我们初步接触了“中断”这一概念。由定时器 timer0 产生中断；处理器各级采取相对应的动作处理中断。本文尝试联系软件与硬件的配合，从处理器与操作系统双维度来简要分析中断。围绕题目的 3 个关键词“进程”，“中断”，“处理”，本文将从以下 3 个方面详细展开。

- 进程
  - 进程要素
  - *进程控制块
  - 进程状态
- 中断
  - 中断意义和产生背景
  - 中断分类
  - 中断和指令周期
- 中断处理
  - 回顾 P7 软硬件配合
  - 操作系统的进程调度

1．进程

1.1 进程要素

（1）定义的理解

“进程”这一概念的提出最早是在 20 世纪 60 年代。在关于进程的诸多定义中，我们不妨从最通俗易懂的开始了解。

- 一个正在执行的程序
- 计算机中正在运行的程序的一个实例
- 可以分配给处理器并由处理器执行的一个实体
- 由单一的顺序的执行线索、一个当前状态和一组相关的系统资源所描述的活动单元

在这层层递进的定义中，不难看出，进程由“程序”这一泛化的、、熟悉的对象，不断具体化，直至表现出“进程被当做数据结构来实现”这一特性。下面，我们对此数据结构作进一步详细的说明。

进程可以看做由以下 3 部分组成：

- 一段可执行的程序 —> 单一顺序的执行线索
- 程序的执行上下文，又称为进程状态 —> 当前状态
- 程序所需要的相关数据（变量、工作空间、缓冲区等）—> 相关的系统资源

进一步高度概括的话，进程是由一组元素组成的实体。进程的两个基本元素是程序代码和与代码相关联的数据集。这一元素概念的提出为我们 1.1.2 进程控制块的讨论有所准备。

（2）概念提出的内涵——解决问题的基础

在了解到进程的具体结构后，我们不禁要问：进程这一概念的提出意义何在？为何要有如此的设计？

问题的背景从计算机系统的发展开始谈起。计算机系统的发展有 3 条主线：多道程序批处理操作、分时和实时系统。我们在这里着重讨论与中断，多道程序相关的话题。多道程序的设计就是为了有效地同时利用处理器和 I/O 等设备的资源，以达到最大的效率。系统程序员在开发早期的多道程序和多用户交互系统时使用的主要工具是中断。处理器保存某些上下文，转而执行中断处理器，处理中断，然后恢复用户关于被中断作业或其他作业的处理。关于处理器这一部分的操作，我们在 P7 中都有所接触，硬件方面的各处理步骤都很熟悉，这里便不再赘述；然而，为了协调这些不同活动而进行的系统软件设计是相当困难的。问题的困难主要体现在复杂性和解决方法的脆弱性上。复杂性在于任何时刻都有许多进程在运行中，且每个进程都有要按顺序执行的诸多步骤，要想枚举出所有可能的中断情况显然是不可行的。也正是由于这一点，程序员只得根据自己的理解，设计出较特殊的方法，而这种方法在面对诸多意想不到的小错误的情况下又是无比脆弱的。这些错误的诊断往往及其困难且不易分析出错误产生的精确条件。一般而言，主要有一下 4 个错误原因：

- 不正确的同步：信号机制
- 失败的互斥：不能保证一次只允许一个例程对一部分数据执行事务处理
- 不正确的程序操作：重写相同的内存区域发生不可预测的相互干扰
- 死锁：多个程序相互挂起等待

解决这些问题需要一种系统方法监控处理器中不同程序的执行，进程的概念便为这些问题的解决提供了基础。

（3）进程管理的初步感知

如图 1，每个进程被分配给一块存储器区域，并由操作系统建立和维护进程表中记录。进程表包含记录每个进程的表项。在图中，由进程索引知进程 B 正在执行，以前执行的进程 A 被临时中断。中断同时，寄存器的全部内容被记录在它的执行上下文中。进程的切换过程包括保存 B 的上下文和恢复 A 的上下文。

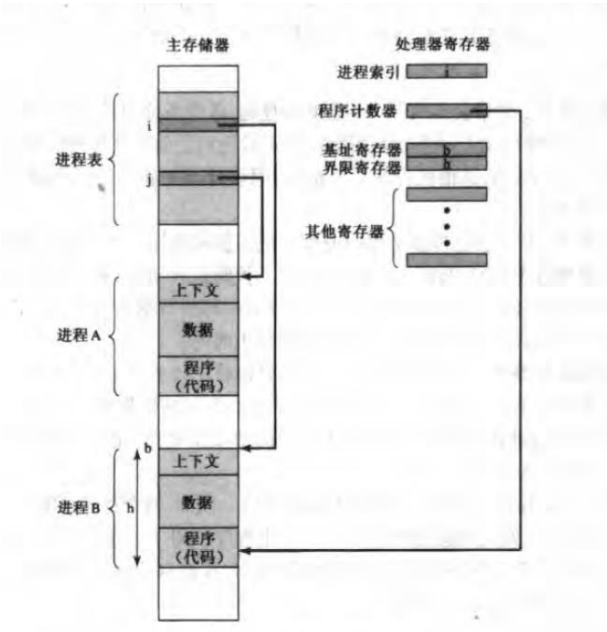

图 1 典型的进程实现方法

*1.2 进程控制块

任意给定的时间，进程都可以唯一被表征成一下元素:

- 标识符：进程的“ID”
- 状态
- 优先级
- 程序计数器
- 内存指针：与程序带代码和相关数据的指针
- 上下文数据
- I/O 状态信息
- 审计信息

上述这些信息被存放在一个 “进程控制块”数据结构中；该控制块由操作系统控制和管理，包含了充足的信息面对中断的处理，使得恢复中断的进程时，进程的运行好像未被中断过。

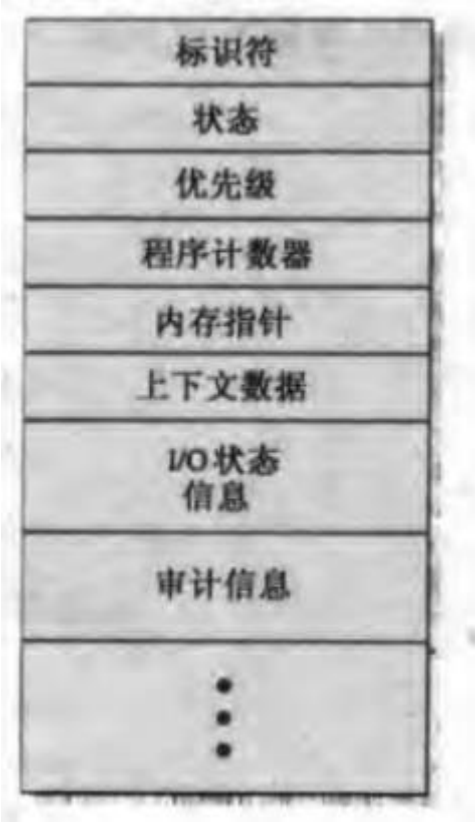

图 2 简化的进程控制块

1.3 进程状态

（1）两状态进程模型

在任何一个时刻，一个进程或者正在执行，或者没有执行，由此可以构造出最简单的两状态进程模型：运行态和非运行态。

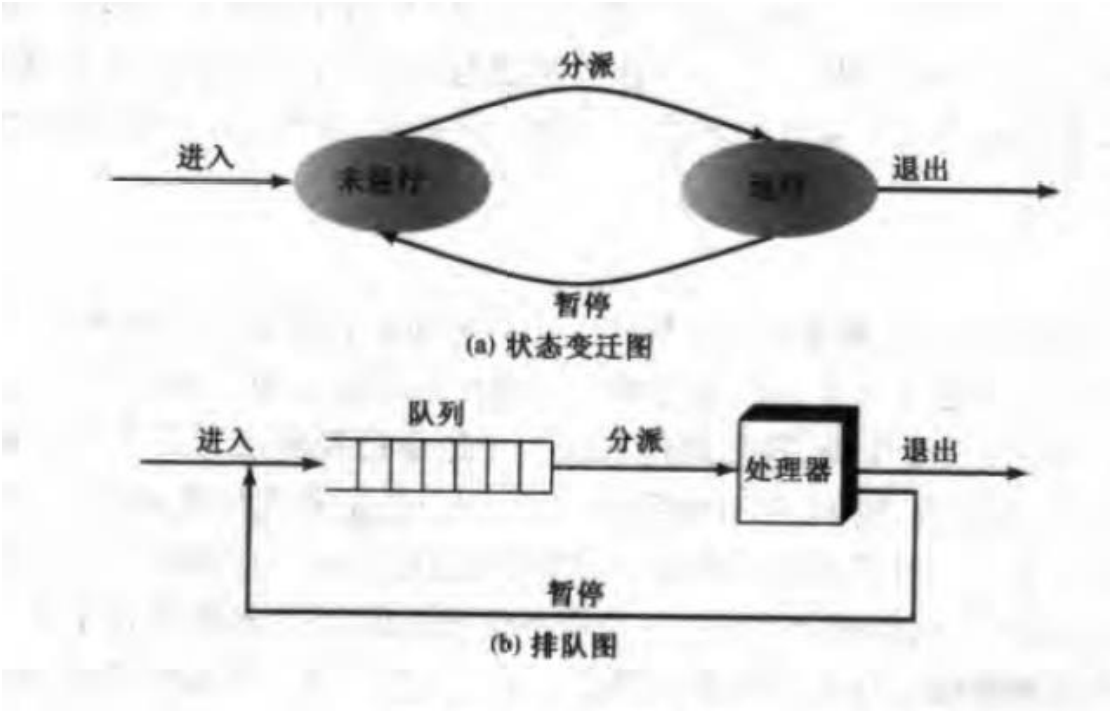

图 3 两状态进程模型

（2）五状态进程模型

两状态进程模型在考虑某些情况时未免太过简略。例如存在一些处于非运行态但已经准备就绪的等待执行的进程，也同样有有处于非运行态但是在等待 I/O 操作结束的进程，仅仅是“非运行态”的描述不足以表现出上述状态的差异。因此提出了下述更为具体的五进程态模型：

- 运行态： 该进程正在执行。在单处理器计算机中，一次最多只有一个进程处于该状态；
- 就绪态： 进程做好了准备，只要有机会就开始执行；
- 阻塞态： 进程在某些事件发生前不能执行，如 I/O 操作完成；
- 新建态： 刚刚新建的进程，操作系统还没有把它加入到可执行进程组中。通常是进程控制块已经创建但还没有加载到主存中的新进程；
- 退出态： 操作系统从可执行进程组中释放的进程，或者是因为它自身停止了，或者是因为某种原因被取消；

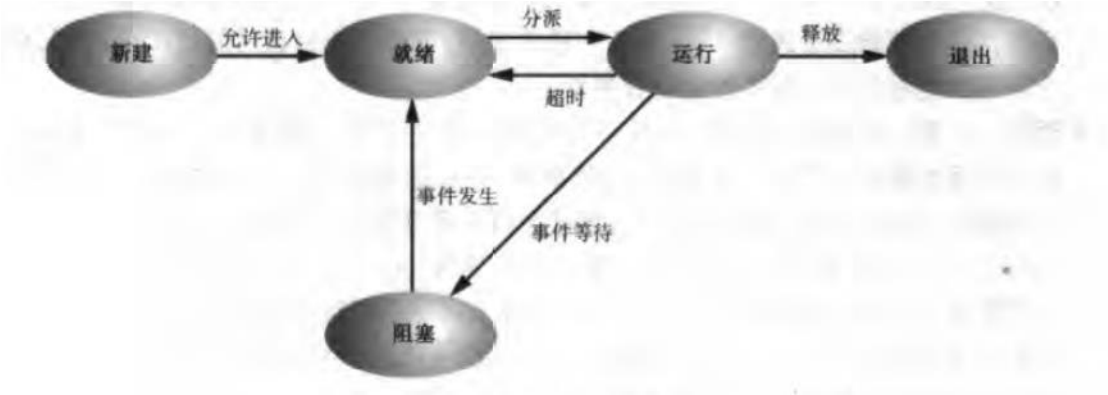

图 4 五状态进程模型

（3）LINUX 内核情景分析

与五状态类似 ，LINUX 系统对“非运行态”进行了更为细致的划分：

TASK_INTERRUPTIBLE:进程处于睡眠状态并且可以因”信号“的到来而被唤醒；

TASK_UNINTERRUPTIBLE:进程处于”深度睡眠“而不受”信号“（signal,也称”软中断“）的打扰；

TASK_RUNNING:可调度，就绪状态；

TASK_ZOMBIE:进程已经”去世“（exit）；

TASK_STOPPED:挂起；

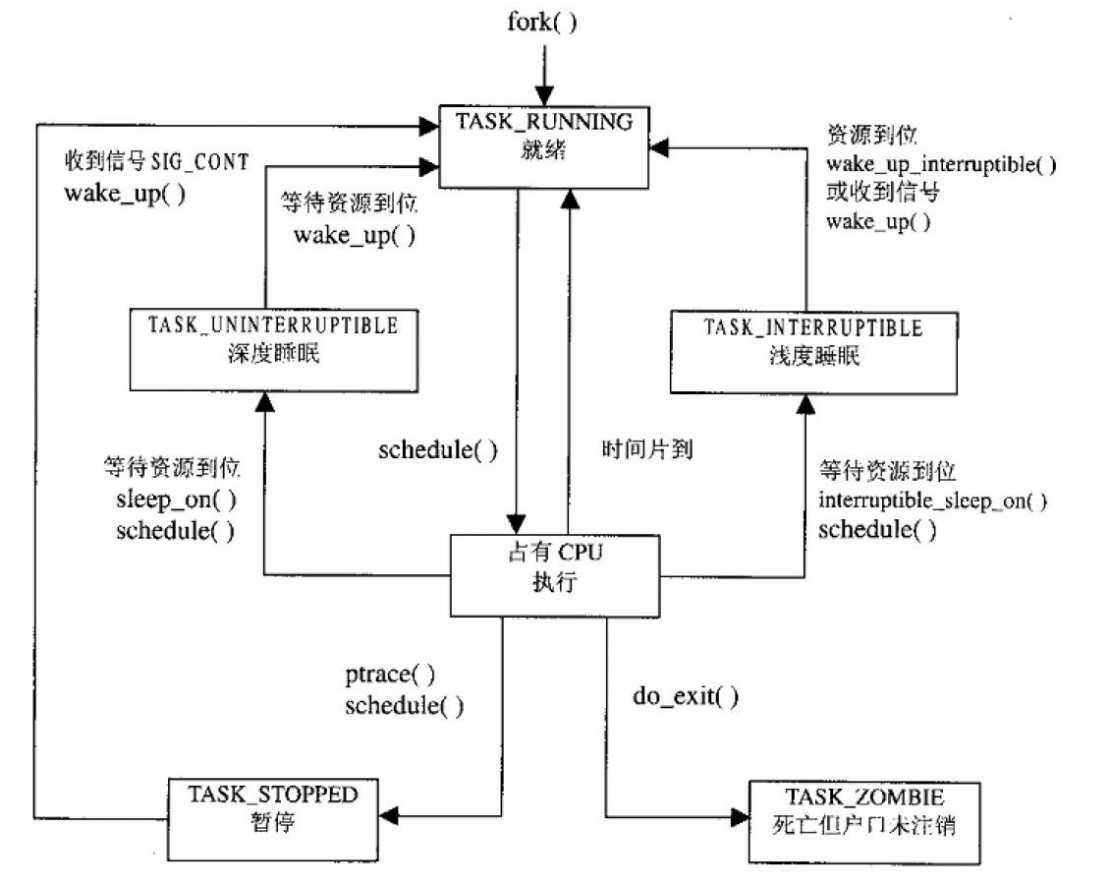

图 5 LINUX 状态转换图

篇幅所限，图中所涉及到的函数这里就不做介绍。

2. 中断

2.1中断的意义和产生背景

- 协调外设与 CPU 处理速度不一致的问题
- 实现多个应用程序之间，和应用程序与操作系统程序之间的切换，使得 CPU 的资源得到有效的利用

2.2 中断分类

- 定时中断：

由定时器差生定时的中断，用于实现多个应用程序之间，和应用程序与操作系统程序之间的切换，使得 CPU 的资源的到有效的利用

- 非定时中断

有外部设备产生，P7 中由协处理器 CP0 处理，用于协调外设与 CPU 处理速度不一 致的问题

2.3 中断和指令周期

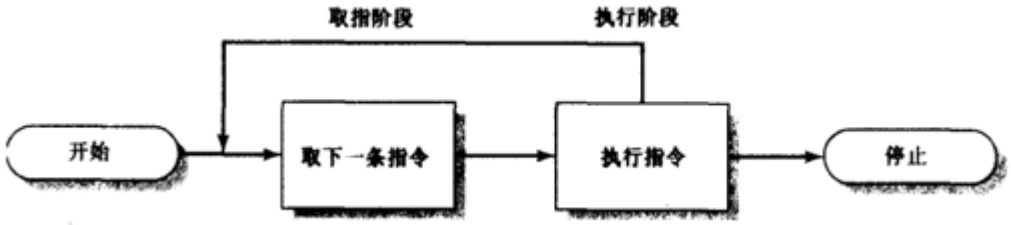

图 6 基本指令周期

图 6 展示了传统的指令执行周期，图 1.7 则在指令的执行周期中加入了中断部分，图 8 很好的展示了中断对于协调 CPU 与外设速度不比配问题的解决：在等待外设信息的时候，CPU 可以执行其他的程序，等待外设数据准备好后以中断的方式请求 CPU 对其数据的处理

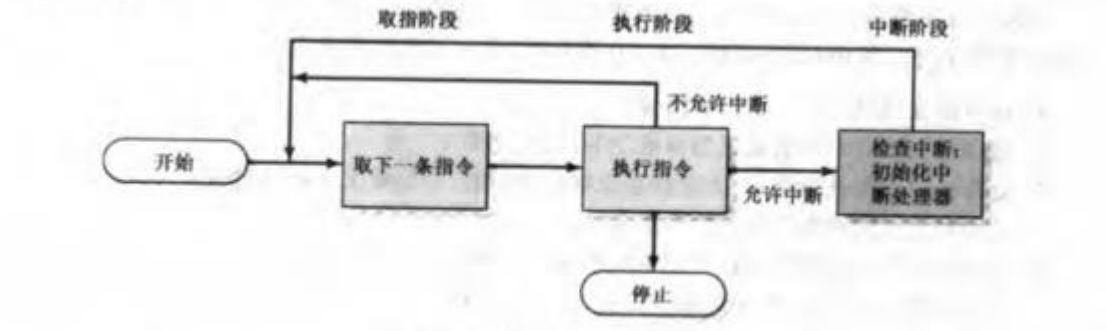

图 7 中断和指令周期

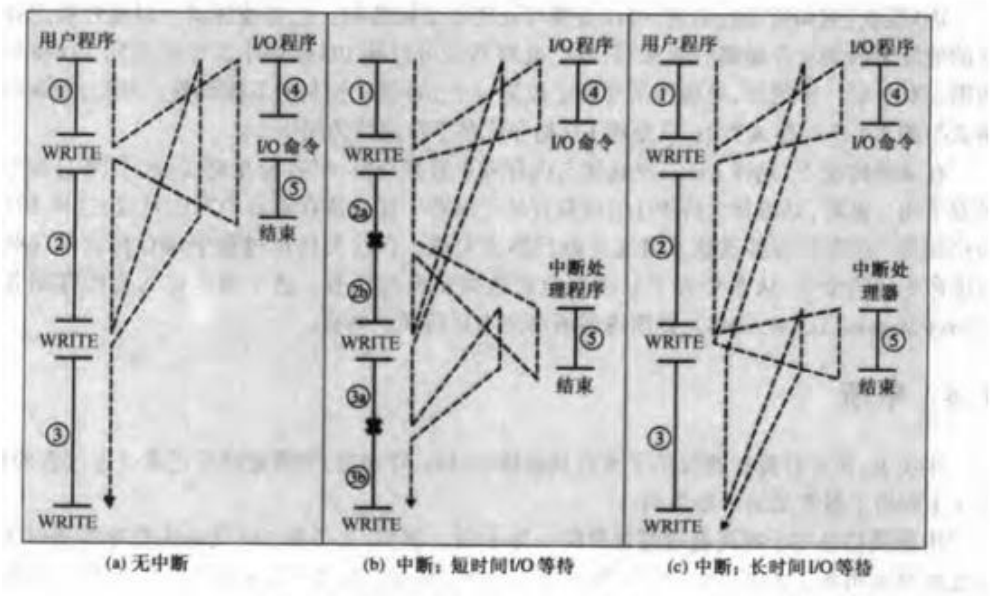

图 8 有中断和无中断时程序的控制流

3. 中断处理

3.1 回顾 P7 软硬件配合

（1）硬件方面的处理

P7 在 P6 的基础上，要求实现一个微系统。这个微系统由 CPU, bridge, timer 构成（具体架构见下图）。CPU 相较 P6，需要加入一个协处理器 CP0，多支持 ERET，MTC0, MFC0 三条指令，并且在 DM 中分配一部分空间给外设（timer）。

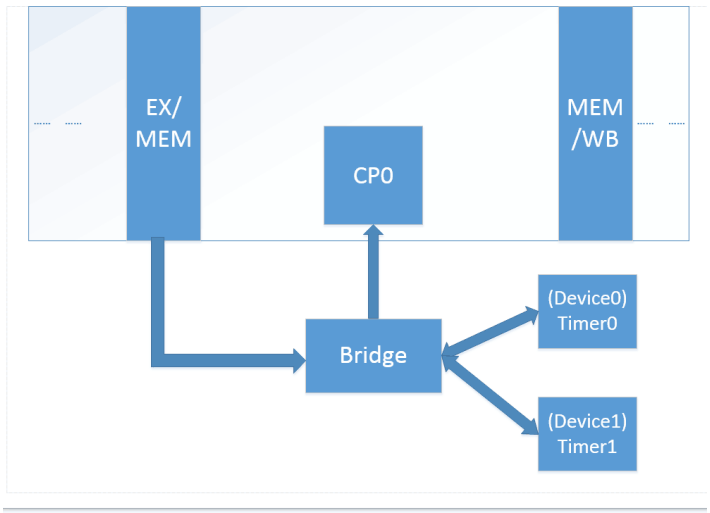

图 9 硬件关系图

- 计数器 timmer

功能：模式 1：周期性计数； 模式 0：产生周期性中断

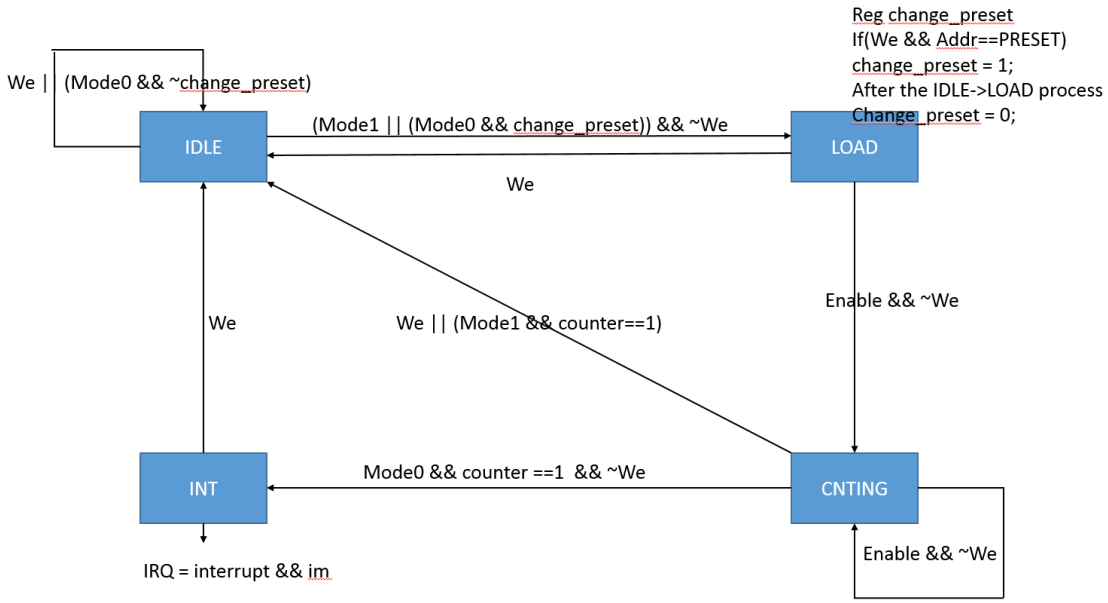

图 10 timer 状态图

- Bridge

功能实现 TIMMER 与 CPU 的通信：

1、CPU 对 TIMMER 的控制通过 Bridge 传入 TIMMER

2、不同外设的中断信号通过 Bridge 整合传入 CPU

- CPU

1、将对 TIMMER 的控制信号（通过写入不同计数器的内容实现）传入 bridge，经过判断将相应的数据和控制信号（写使能）传入 TIMMER

2、设置协处理器 CP0 来响应外部设备的中断信息，并且控制程序的执行顺序（在中断程序和主程序之间的跳转，以及 PC 值得保存）

2.1、 中断发生时，通过多选器将适当的 PC 值写入到 EPC 寄存器当中

2.2、 P7 不支持中断嵌套，在执行中断子程序时候，屏蔽其他外部中断

2.3、 中断子程序执行结束后，通过 ERET 指令将 PC 值改为主程序当中的值（from EPC），并在 MEM 段出现主程序中的指令时，再次修改 CP0，允许下一次中断的发生

3、支持 MTC0,MFC0 来实现对 CP0 的控制和监控

3.1 实现对 CP0 的监控

通过读取 cause 寄存器的信息，系统得知产生中断的原因，获知当前的状态;

通过读取 SR 寄存器，系统得以了解协处理器的控制信息

3.2 实现对 CP0 的控制

MTP0 可以写入 CP0，修改 CP0 的 SR，CAUSE 寄存器，控制 CP0 对外设中断信号的响应

4、支持 ERET 指令，实现子程序之后的返回

（2） 软件方面的处理

- 整体分析

框架结构：保存现场、中断处理、恢复现场、中断返回

1、保存现场

将所有寄存器都保存在堆栈中

2、中断处理

读取特殊寄存器了解哪个硬件中断发生

执行对应的处理策略（例如读写设备寄存器、存储器等）

3、恢复现场

从堆栈中恢复所有寄存器

4、中断返回

执行eret指令

其中步骤1,3,4 在中断处理中具有通用性，而步骤2依据中断类型的不同采取不同的处理策略。通过具体的exception handler的代码再来体会一下中断服务程序的框架；此代码的应用环境是只有timer0 设备会产生中断信号；

- exception handler 的分析（代码见附录）

1、_int_entry：

```
.ktext 0x00004180    #中断服务程序的PC入口
#  在中断使能的值恢复之前，0x2E00 ~ 0x02FFC(512-Bytes)的空间都是用来存放数据的；只有$k0,$k1是可以借做他用的。
# 存储栈顶的位置save sp to system stack top
ori $k0, $0, 0x1000
sw $sp, -4($k0)      ##保存$sp,
# save SR to stack
mfc0 $k1, $12        ##取SR存入DM
sw $k1, -8($k0)
# sp point to stack bottom with  absolutely safety(half of system stack)
addiu $k0, $k0, -256
move $sp, $k0        ##$SP指向栈底，准备存入寄存器中的值
# save context
j _save_context
nop
```

2、j _save_context   ##从栈底开始存寄存器的数值
_int_entry_save_context ##保存后返回

3、jal timer0_handler##执行中断子程序

```
 # first we load the global variable cnt:
# ++cnt, then save to global variable cnt   ##cnt是一个全局变量，cnt++
# address of cnt is 0                       ##起到当时间片的记录作用
lw $t0, 0($0)            # get cnt
```

```
    addi $t0, $t0, 1         # add cnt
    sw $t0, 0($0)            # update cnt
    nop
    nop
    ori $v0, $0, 0x200
    ori $t1, $0, 0x7f00          # $t1 is base of Timer 0
    lw $t0, 0($t1)               # $t0 is the CTRL Reg of Timer 0
                            ##读取timmer0的控制信息
    sw $0, 0($t1)            # disable Timer 0
                               ##enable==0准备load
    sw $v0, 4($t1)               # refill the count number
                               ##load preset
    sw $t0, 0($t1)               # Timer 0 restart count
                               ##重新开始计数
    jr $ra
    nop
```

4、_restore_context ##恢复现场

```
_int_entry_restore_context
     li $k0, 0x1000
    lw $sp, -4($k0)     #恢复寄存器值后，恢复$sp
   # 保存现场——协处理器中关于中断的信息restore CP0.SR and enable
interrupt
    lw $k1, -8($k0)     ##恢复$k1  $k1中存的是CP0.SR寄存器的值
    ori $k1, $k1, 0x1     ##恢复，允许中断
    mtc0 $k1, $12
    mfc0    $k0, $14    # get original PC where exception happened
     ####此处涉及系统的调度
     ####更新当前处于运行态的进程的进程控制块；
     ####把进程控制块转移到相应的队列；选择一个进程执行；
     ####更新选择的进程的进程控制块；
     ####更新内存管理的数据结构；
     mtc0    $k0, $14             # write to EPC
    # return from interrupt
    Eret    ##返回原函数
    nop    # also a delay slot
```

3.2 操作系统的进程调度

3.2.1 进程的调度与切换

（1）进程的调度

在多进程的操作系统中，进程的调度是全局性的关键问题。好的进程调度机制要兼顾到三种不同应用的需要：

- 交互式应用：着重与系统的响应速度，当多个用户公用一个操作系统时，要保证延迟小于用户可察觉的范围下界；

- 批处理应用：批处理往往是作为”后台作业“运行的，虽然对响应速度没有要求，但还是要考虑”平均速度“。
- 实时应用。时间性最强的应用，要同时兼顾到进程执行的”平均速度“和”即时速度“。注重对程序执行的”可预测性“。

针对于上述的目标，进程调度机制需考虑如下的问题：

- 调度的时机：在什么情况下进行调度；
- 调度的政策：根据什么准则选择下一进入运行的进程；
- 调度的方式：“可剥夺”还是“不可剥夺”？

注：

不可剥夺（非抢占式优先权算法）

在这种方式下，系统一旦把处理机分配给就绪队列中优先权最高的进程后，该进程便一直执行下去，直至完成。或因发生某事件使该进程放弃处理机时，系统方可再将处理机重新分配给另一优先权最高的进程，这种调度算法主要用于批处理系统中；也可用于某些对实时性要求不严的实时系统中。

可剥夺（抢占式优先权调度算法）

系统同样把处理机分配给优先权最高的进程使之执行，但在其执行期间,只要又出现了另一个其优先权更高的进程，进程调度程序就立即停止当前进程(原优先权最高的进程)的执行，重新将处理机分配给新到的优先权最高的进程。这种抢占式的优先权调度算法,能更好地满足紧迫作业的要求。

关于进程调度的进一步讨论将在 3.2.2 单处理器调度展开。

（2）进程的切换

我们需要考虑的问题有：

- 什么时间触发进程的切换

结合 P7 的任务要求，这里我们只对时钟中断进行分析。

在介绍时钟中断之前，先了解一下 “时间片”的概念：每个进程被分配一个时间段，称作它的时间片，即该进程允许运行的时间。

时钟中断是指操作系统确定当前正在运行的进程的执行时间是否已经超过了最大允许时间段，如果已经超过了，进程必须被切换到就绪态，然后掉入另一个进程；

- 模式切换与进程切换的区别

发生模式切换可以不改变正处于运行态的进程状态，保存和恢复上下文只需要很少的开销；但是如果是进程切换的话，操作系统必须使其环境发生实质性的变化。

- 操作系统在进程切换时都做了什么
  - 保存处理器上下文，包括程序计数器和其他寄存器；
  - 更新当前处于运行态的进程的进程控制块；
  - 把进程控制块转移到相应的队列；
  - 选择另一个进程执行；
  - 更新选择的进程的进程控制块；
  - 更新内存管理的数据结构；
  - 恢复处理器在被选择的进程最近一次切换出运行时态的上下文；

3.2.2 单处理器调度

3.2.2.1 调度类型（整体框架见图片）

项目	说明
长程调度	决定加入到待执行的进程池中
中程调度	决定加入到部分或全部在主存中的进程集合中
短程调度	决定哪一个可用进程将被处理器执行
I/O 调度	决定哪一个进程挂起的 I/O 请求将被可用的 I/O 设备处理

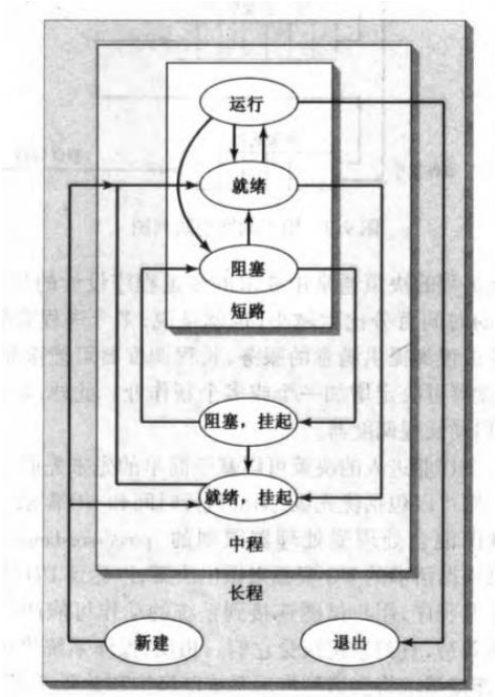

图 9.2　调度的层次

3.2.2.2 短程调度

由于本文关注的重点在于时间片定时中断的机制，与过程中软硬件的配合问题，因此，只对短程调度展开讨论

当可能导致当前进程阻塞或可能抢占剥夺当前正在运行的事件发生时，调用短程调度器，这类事件包括：

- 时钟中断
- I/O 中断
- 操作系统调用
- 信号

（1）准则

主要在两个维度上来考虑：面向用户和面向系统；面向用户：对于用户在进程上的感知而言，调度的关键在于响应时间；面向系统：重点是处理器的使用和效率；下表所列的几种影响调度方式考虑的因素是相互依赖的，难以同时达到最优。例如，提供较好的响应时间可能意味着在不同进程中进行频繁地切换，这样一来处理器的效率难免有所损失。因此调度策略需在各种竞争因素中进行折中，通过不同的权重与配比，实现最优调度。

面向用户，与性能相关	
周转时间	指一个进程从提交到完成之间的时间间隔，包括实际执行时间加上等待资源（包括处理器资源）的时间。对批处理作业而言，这是一种很适宜的度量
响应时间	对一个交互进程，这是指从提交一个请求到开始接收响应之间的时间间隔。通常进程在处理该请求的同时，就开始给用户产生一些输出。因此从用户的角度来看，相对于周转时间，这是一种更好的度量。该调度原则应该试图达到较低的响应时间，并且在响应时间可接受的范围内，使得可以交互的用户的数目达到最大
最后期限	当可以指定进程完成的最后期限时，调度原则将服从于其他目标，使得满足最后期限的作业数目的百分比达到最大
面向用户，其他	
可预测性	无论系统的负载如何，一个给定的工作运行的总时间量和总代价是相同的。用户不希望响应时间或周转时间的变化太大。这可能需要对系统工作负载大范围抖动发出信号或者需要系统处理不稳定性
面向系统，与性能相关	
吞吐量	调度策略应该试图使得每个时间单位完成的进程数目达到最大。这是关于可以执行多少工作的一种度量。它完全取决于一个进程的平均执行长度，也受调度策略的影响，调度策略会影响使用率
处理器使用率	这是处理器忙的时间百分比。对昂贵的共享系统而言，这是一个重要的准则。在单用户系统和一些其他的系统如实时系统中，该准则与其他相比显得不太重要
面向系统，其他	
公平	在没有来自用户的指导或其他系统提供的指导时，进程应该被平等地对待，没有一个进程会被饿死
强制优先级	当进程被指定了优先级后，调度策略会先选择高优先级的进程
平衡资源	调度策略将保持系统中所有资源忙，负担较重且资源使用较少的进程应该受到照顾。该准则也可用于中程调度和长程调度

表 1 调度准则

（2） 轮转调度策略

短程调度的策略有很多，这里我们只讨论与时钟中断，时间片相关的轮转法。轮转法对于进程公平对待，有最小的开销，并能对短进程提供较好的响应时间。对于轮转法的调度设计核心在于时间片的长度。过短的时间片着实可以保证很好的响应时间，在面向用户维度上有很好的效果；但是频繁地切换却会让处理器的效率大大降低，所以要尽量避免使用过短的时间片；资料显示，时间片最好略大于一次典型交互所用时间，若小于此时间，大多数进程并不能满足一个在一个时间片里完成的要求了。

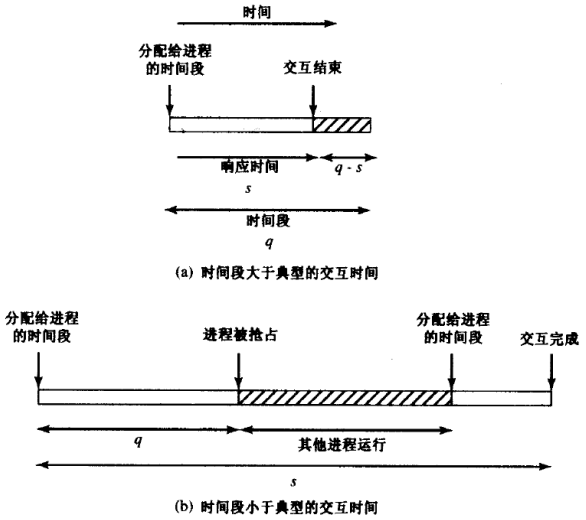

图 11 时间片大小的影响

（3） 与时间片相关的调度分析

在 P7 的定时器设计中，模式 0 与时间片调度机制有极为紧密的联系，下面简要概述一下时间片调度机制：时间片轮转调度是一种最古老，最简单，最公平且使用最广的算法。如果在时间片结束时进程还在运行，则 CPU 将被剥夺并分配给另一个进程。如果进程在时间片结束前阻塞或结束，则 CPU 当即进行切换。当进程用完它的时间片后，它被移到队列的末尾。

如果调度的性质是有条件的可剥夺，那么，在什么情况下剥夺就成了重要的问题。例如，可以每个时间片来一次时钟中断，而调度可以在时间片中断时进行。按进程的优先级的高低进行调度，每个时间片一次，除此之外就只能在进程自愿时才能进行调度。这样，只要时间片划分得当，交互式的应用便可满足。但是这样的系统又不适合实时的应用。优先级高的进程等待运行，无奈正在运行的较低优先级的进程“慢条斯理，耗费大好时光”，别的进程只能苦等它把时间片用完而错失良机。从另一个角度来看，这与 CPU 的速度也息息相关，如果可以把时间片分得足够小，实时应用也是有满足要求的可能性的。

（3） UNIX 进程调度

基于时间片的多级反馈队列算法设置多个就绪队列，分别赋予不同的优先级，如逐级降低，队列 1 的优先级最高。每个队列执行时间片的长度也不同，规定优先级越低则时间片越长，如逐级加倍。新进程进入内存后，先投入队列 1 的末尾，按 FCFS 算法调度；若按队列 1 一个时间片未能执行完，则降低投入到队列 2 的末尾，同样按 FCFS 算法调度；如此下去，降低到最后的队列，则按“时间片轮转”算法调度直到完成。仅当较高优先级的队列为空，才调度较低优先级的队列中的进程执行。如果进程执行时有新进程进入较高优先级的队列，则抢先执行新进程，并把被抢先的进程投入原队列的末尾。

注：FCFS 算法：“先来先服务”：当每个进程就绪后，它加入就绪队列。当当前正在运行的进程停止执行时，选择在就绪队列中存在时间最长的进程运行。

展望与思考

P7 的实验涉及了一些系统层面的背景，在写报告、阅读文献的过程中，对计算机在系统层面的设计有了更多的了解。

计算机体系结构面对的两大问题，其一是在系统层次高效的利用资源（包括计算资源，存储资源等等）、其二是通过并行执行提高计算机性能。而中断是一种很有效的手段，可以很好的分配计算机的资源，选择合理方式的让各个进程“并行”执行，提高了计算机的性能。

进一步的展望未来，对于更大型的、更复杂的、分布式的体系，合理的分配资源是非常重要的。比如将来要发展智能城市，要发展智能工厂，要实现物联网。大规模的信息采集，处理，对于计算资源的高效分配，计算资源的可扩展性要求越来越高。无疑，中断提供了一种很好的解决问题的思路，不过还是有很远的路要走，还有很多事情值得去研究。按照笔者的理解，中断本质上想要处理的是资源的供给与需求之间的矛盾，关于中断的产生机制，分配机制，可靠性，安全性等等问题都值得深入的研究，并且对于不同的系统而言，侧重点是不同的，这就带来了宽广的研究空间。比如应用排队论的一些知识可以使得资源的分配更加合理；比如利用信息编码的知识，可以增加系统间通信，和计算资源分配的可靠性和安全性；再比如设计更精细的多级的处理中断的单元，为中断设计软件层面的普适性的接口，增加系统的可扩展性和可移植性。另外，还有很多问题有待解决和研究，比如信息一致性问题、智能存储问题等等。

总之，未来充满机遇。

致谢

非常感谢刘乾助教能给我们这次宝贵的学习机会，并在完成的过程中不断指导我们学习的重点与方向 ，推荐了很多优秀的资源以供参考。刘乾助教对我们的鼓励与帮助是这篇报告得以完成的重要因素。一个学期的计组实验令人难忘——多少次挑灯夜战，面对波形图指指点点；多少次梦回计组，在睡梦中还说着 PC，DM 的梦话；计组实验对于我们是一种工程化的训练，培养我们系统的，有体系的去解决问题；这次的研究性报告就不太相同，更多是一种探索，学习，领悟，交流，总结的过程。面对完全未知的知识，各种铺面而来的资料，确实一开始不知从何下手。在经历了与同伴的交流、讨论过程后，我们的思路便逐渐清晰，方向也愈加明确；同伴的合作一方面让我们的思路得到延伸扩展，灵感之间碰撞出火花；另一方面也积极督促彼此更好更快地完成任务，不拖延，不懈怠，因此我们感谢彼此为这份报告共同付出的努力！

附录一：exception handler 源码

```
.ktext 0x00004180   #中断服务程序的PC入口
_int_entry:
    #  在中断使能的值恢复之前，0x2E00 ~ 0x02FFC(512-Bytes)的空间都是用来存放数据的；只有$k0,$k1是可以借做他用的。
    # 存储栈顶的位置save sp to system stack top
    ori $k0, $0, 0x1000
    sw $sp, -4($k0)
    # save SR to stack
    mfc0 $k1, $12
    sw $k1, -8($k0)

    # sp point to stack bottom with  absolutely safety(half of system stack)
    addiu $k0, $k0, -256
    move $sp, $k0

    # save context
    j _save_context
    nop

_int_entry_save_context:
    # jump the timer 0 handler
    jal timer0_handler
    nop

    # restore context
    j _restore_context
    nop

_int_entry_restore_context:
    # restore sp
    li $k0, 0x1000
    lw $sp, -4($k0)
# 保存现场——协处理器中关于中断的信息restore CP0.SR and enable interrupt
    lw $k1, -8($k0)
    ori $k1, $k1, 0x1
    mtc0 $k1, $12

    mfc0    $k0, $14            # get original PC where exception happened
```

```
    mtc0    $k0, $14            # write to EPC
    # return from interrupt
    eret
    nop                         # also a delay slot


timer0_handler:
    # first we load the global variable cnt:
    # ++cnt, then save to global variable cnt
    # address of cnt is 0
    lw $t0, 0($0)           # get cnt
    addi $t0, $t0, 1        # add cnt
    sw $t0, 0($0)           # update cnt
    nop
    nop
    ori $v0, $0, 0x200
    ori $t1, $0, 0x7f00         # $t1 is base of Timer 0
    lw $t0, 0($t1)              # $t0 is the CTRL Reg of Timer 0
    sw $0, 0($t1)           # disable Timer 0
    sw $v0, 4($t1)              # refill the count number
    sw $t0, 0($t1)              # Timer 0 restart count
    jr $ra
    nop

_save_context:
    sw $1,4($sp)
     sw  $2, 8($sp)
     sw  $3, 12($sp)
     sw  $4, 16($sp)
     sw  $5, 20($sp)
     sw  $6, 24($sp)
     sw  $7, 28($sp)
     sw  $8, 32($sp)
     sw  $9, 36($sp)
     sw  $10, 40($sp)
     sw  $11, 44($sp)
     sw  $12, 48($sp)
     sw  $13, 52($sp)
     sw  $14, 56($sp)
     sw  $15, 60($sp)
     sw  $16, 64($sp)
     sw  $17, 68($sp)
     sw  $18, 72($sp)
     sw  $19, 76($sp)
```

```
    sw  $20, 80($sp)
    sw  $21, 84($sp)
    sw  $22, 88($sp)
    sw  $23, 92($sp)
    sw  $24, 96($sp)
    sw  $25, 100($sp)
    sw  $26, 104($sp)
    sw  $27, 108($sp)
    sw  $28, 112($sp)
    sw  $29, 116($sp)
    sw  $30, 120($sp)
    sw  $31, 124($sp)
    j _int_entry_save_context
    nop

_restore_context:
    lw $1,4($sp)
    lw  $2, 8($sp)
    lw  $3, 12($sp)
    lw  $4, 16($sp)
    lw  $5, 20($sp)
    lw  $6, 24($sp)
    lw  $7, 28($sp)
    lw  $8, 32($sp)
    lw  $9, 36($sp)
    lw  $10, 40($sp)
    lw  $11, 44($sp)
    lw  $12, 48($sp)
    lw  $13, 52($sp)
    lw  $14, 56($sp)
    lw  $15, 60($sp)
    lw  $16, 64($sp)
    lw  $17, 68($sp)
    lw  $18, 72($sp)
    lw  $19, 76($sp)
    lw  $20, 80($sp)
    lw  $21, 84($sp)
    lw  $22, 88($sp)
    lw  $23, 92($sp)
    lw  $24, 96($sp)
    lw  $25, 100($sp)
    lw  $26, 104($sp)
    lw  $27, 108($sp)
    lw  $28, 112($sp)
```

```
lw  $29, 116($sp)
lw  $30, 120($sp)
lw  $31, 124($sp)
j _int_entry_restore_context
nop
```

参考文献

[1][美]William, Stallings, 著；陈渝, 译；向勇, 审校. 操作系统——精髓与设计原理（第五版）[M]. 北京:电子工业出版社, 2006.
[2]毛德操, 胡希明. LINUX 内核源代码情景分析[M]. 杭州:浙江大学出版社, 2001.
[3]沈雪峰;多 CPU 系统的中断机制
http://wenku.baidu.com/link?url=N09JrVjWGPzCFliLdOpPd-aRSmWL9xIbQnoRxOBoVsELHLwtVElYnamUylx4oWuudvzPwaxsmjSmOkd72bUfFrQ87ZNYp3NqDe_kHZYsJku&pn=1