ECF:Exceptions & Processes

王昕兆

2019 年 11 月 14 日

Outline 1

异常的定义

- 控制流：程序计数器值的一个序列
  - “普通”的控制流：指令之间要么相邻要么产生由由程序变量表示的内部程序状态引起的突变

异常的定义

- 控制流：程序计数器值的一个序列
  - “普通”的控制流：指令之间要么相邻要么产生由由程序变量表示的内部程序状态引起的突变
  - 异常控制流：由系统状态引起的突变
    例如：包到达网络适配器后，必须放在内存中；程序向磁盘请求数据，然后休眠直到被通知说数据已经就绪；子进程结束时，创造这些子进程的父进程必须得到通知
- 异常：控制流中的突变：用来响应处理器中的某些变化

在异常之后

根据异常的类型，异常处理结束后，会发生三种可能的情况：

1. 执行事件发生时正在执行的指令
2. 执行如果没有发生异常将会执行的下一条指令（不一定是相邻的指令）
3. 终止被中断的程序

异常处理

异常处理的流程

- 处理器检测到有事件（处理器状态发生重要变化，例如，发生虚拟内存缺页，算数溢出，或者一个系统定时器产生的信号）发生，并且确定了对应的异常号（和每种异常唯一对应的非负整数）

异常处理

异常处理的流程

- 处理器检测到有事件（处理器状态发生重要变化，例如，发生虚拟内存缺页，算数溢出，或者一个系统定时器产生的信号）发生，并且确定了对应的异常号（和每种异常唯一对应的非负整数）
- 从异常表基址寄存器中取出异常表的起始地址（一张跳转表）并用类似间接跳转的方式转移控制给对应的异常处理程序

异常处理

异常处理的流程

- 处理器检测到有事件（处理器状态发生重要变化，例如，发生虚拟内存缺页，算数溢出，或者一个系统定时器产生的信号）发生，并且确定了对应的异常号（和每种异常唯一对应的非负整数）
- 从异常表基址寄存器中取出异常表的起始地址（一张跳转表）并用类似间接跳转的方式转移控制给对应的异常处理程序
- 在完成处理后，异常处理程序将控制返回给被中断的程序或终止

异常处理

异常处理和过程调用

- 异常处理程序都运行在内核模式下（拥有更高权限的模式，可以访问所有系统资源，换句话说突破了虚拟地址空间的限制）

异常处理

异常处理和过程调用

- 异常处理程序都运行在内核模式下（拥有更高权限的模式，可以访问所有系统资源，换句话说突破了虚拟地址空间的限制）
- 在过程调用中，为了在结束后恢复，需要把返回地址压入栈中，它始终是 call 的下一条指令，而在异常处理中，可能是当前指令，也可能是事件不发生应该执行的下一条指令

异常处理

异常处理和过程调用

- 异常处理程序都运行在内核模式下（拥有更高权限的模式，可以访问所有系统资源，换句话说突破了虚拟地址空间的限制）
- 在过程调用中，为了在结束后恢复，需要把返回地址压入栈中，它始终是 call 的下一条指令，而在异常处理中，可能是当前指令，也可能是事件不发生应该执行的下一条指令
- 要压栈的除了返回地址，还有诸如当前条件码等其他处理器状态

异常处理

异常处理和过程调用

- 异常处理程序都运行在内核模式下（拥有更高权限的模式，可以访问所有系统资源，换句话说突破了虚拟地址空间的限制）
- 在过程调用中，为了在结束后恢复，需要把返回地址压入栈中，它始终是 call 的下一条指令，而在异常处理中，可能是当前指令，也可能是事件不发生应该执行的下一条指令
- 要压栈的除了返回地址，还有诸如当前条件码等其他处理器状态
- 每个进程都有自己的私有地址空间，对 x86-64，大于 $2^{48}-1$ 的高地址区域只有在内核模式下才可以访问，内核虚拟内存中也有自己的代码、数据、堆、栈，当控制从用户程序转移到内核，所有的数据都会被压到内核栈中（而非用户栈中）

异常的类别

类别	原因	异步/同步	返回行为
中断	来自 I/O 设备的信号	异步	总是返回到下一条指令
陷阱	有意的异常	同步	总是返回到下一条指令
故障	潜在可恢复的错误	同步	可能返回到当前指令
终止	不可恢复的错误	同步	不会返回

异步/同步是异常的两种触发方式：处理器外部的 I/O 设备中的事件和执行当前指令的直接产物

异常的类别

1. 中断：I/O 设备主要是：网络适配器、硬盘控制器和定时器芯片
2. 陷阱：陷阱最重要的的用途是实现系统调用，普通的调用只能从用户模式到用户模式，访问权限也只是虚拟内存中非内核部分，而陷阱、最常见的是 syscall 指令，可以调用适当的内核程序（异常处理程序是在内核模式下运行的）
3. 故障：经典的故障是缺页故障，它会影响到当前指令的执行，但是是可能被修复的，因此异常处理程序要么终止原程序，要么处理异常后重新执行引起故障的指令
4. 终止：通常是一些不可恢复的致命错误，一般是硬件错误

Outline 2

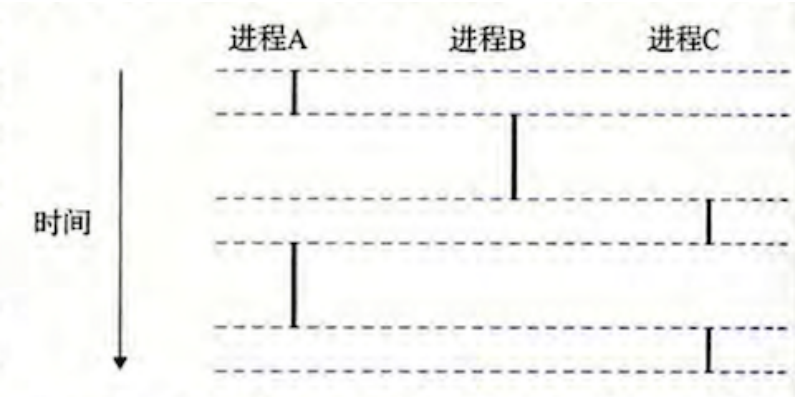
进程的定义

进程是一个执行中程序的实例，系统中的每个程序都运行在某个进程的上下文中（包含存放在内存中的代码和数据，通用目的寄存器的内容，程序计数器等等）
进程给应用程序提供了一个抽象化的“计算机”，而把应用程序在这个抽象的计算机上的行为变成在真实硬件上的行为需要操作系统的帮助。

1. 抽象计算机的处理器：一个独立的（独立是指和其他进程之间互不干扰）逻辑控制流
2. 抽象计算机的内存系统：一个私有（私有是指该程序认为只有自己能够访问内存，不用担心被其他进程修改）的地址空间

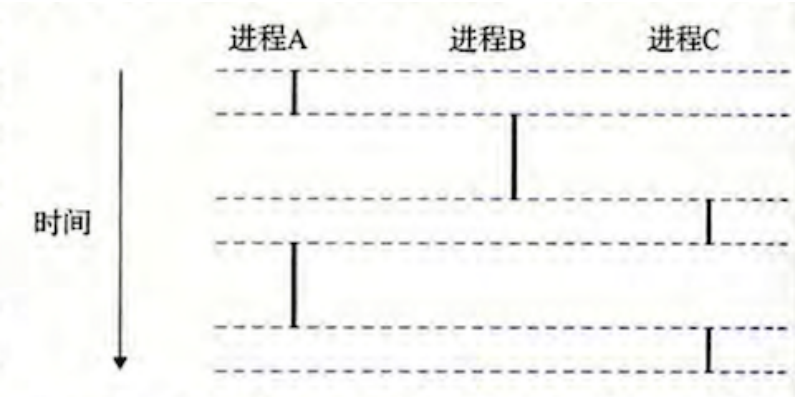
并发流

- 但实际上，处理器是在同时执行多个进程的，真实的处理器中的 PC 中储存的指令地址组成了物理控制流，而对于一个进程的抽象出的逻辑控制流则是这个物理控制流的子列，进程轮流使用处理器，每个进程执行它的流的一部分，然后被抢占（暂时挂起）
  这里要注意，我们之前讨论的异常处理的过程都是对于一个进程（也就是一个逻辑控制流）而言的，而现在我们尝试站在物理控制流的角度思考，所以可能会产生一些看似不满足之前异常处理要求的行为，但是实际上，我们只需要保证在某个逻辑控制流的视角，我们的行为是符合要求的即可



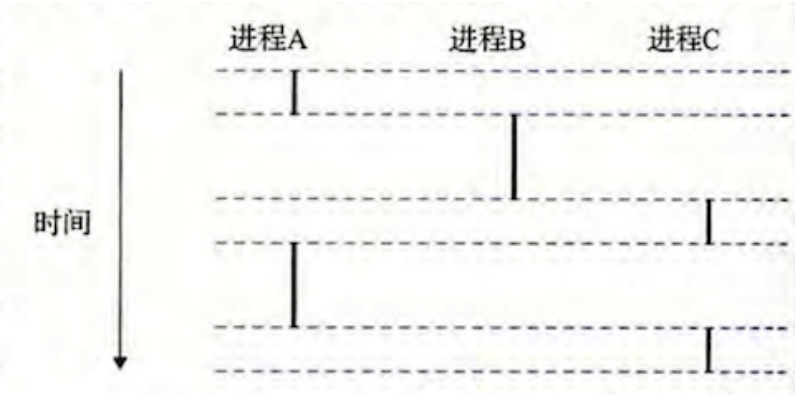
并发流

- 但实际上，处理器是在同时执行多个进程的，真实的处理器中的 PC 中储存的指令地址组成了物理控制流，而对于一个进程的抽象出的逻辑控制流则是这个物理控制流的子列，进程轮流使用处理器，每个进程执行它的流的一部分，然后被抢占（暂时挂起）
  这里要注意，我们之前讨论的异常处理的过程都是对于一个进程（也就是一个逻辑控制流）而言的，而现在我们尝试站在物理控制流的角度思考，所以可能会产生一些看似不满足之前异常处理要求的行为，但是实际上，我们只需要保证在某个逻辑控制流的视角，我们的行为是符合要求的即可
- 并发流：一个流的执行时间和另一个流有重叠



并发流

- 但实际上，处理器是在同时执行多个进程的，真实的处理器中的 PC 中储存的指令地址组成了物理控制流，而对于一个进程的抽象出的逻辑控制流则是这个物理控制流的子列，进程轮流使用处理器，每个进程执行它的流的一部分，然后被抢占（暂时挂起）
  这里要注意，我们之前讨论的异常处理的过程都是对于一个进程（也就是一个逻辑控制流）而言的，而现在我们尝试站在物理控制流的角度思考，所以可能会产生一些看似不满足之前异常处理要求的行为，但是实际上，我们只需要保证在某个逻辑控制流的视角，我们的行为是符合要求的即可
- 并发流：一个流的执行时间和另一个流有重叠
- 并发现象和处理器核数没有关系，它是现代操作系统都有的性质



多任务的实现：上下文切换

为了实现多任务，操作系统需要在进程执行的某个阶段决定抢占当前进程，并重新开始一个之前被抢占的进程，这个决策叫做调度。而为了具体实现在两个进程之间的控制转移，操作系统内核使用了一种称为上下文切换的较高层形式的异常控制流，它分为三步

1. 保存当前进程的上下文
2. 恢复某个先前被抢占的进程被保存的上下文
3. 将控制转移给这个新恢复的进程

进程控制：创建子进程

下面的内容是 c 语言对进程控制的具体实现

- 每一个进程都有唯一的正整数不能是零ID，用 pid_t getpid(void) 获取当前进程的 ID

进程控制：创建子进程

下面的内容是 c 语言对进程控制的具体实现

- 每一个进程都有唯一的正整数不能是零ID，用 pid_t getpid(void) 获取当前进程的 ID
- 进程的三种状态
  1. 运行：正在处理器上运行，或者被挂起，但是将会被内核调度
  2. 停止：进程被挂起，并且不会被调度直到收到某些信号
  3. 终止：进程永远停止

进程控制：创建子进程

下面的内容是 c 语言对进程控制的具体实现

- 每一个进程都有唯一的正整数不能是零ID，用 pid_t getpid(void) 获取当前进程的 ID
- 进程的三种状态
  1. 运行：正在处理器上运行，或者被挂起，但是将会被内核调度
  2. 停止：进程被挂起，并且不会被调度直到收到某些信号
  3. 终止：进程永远停止
- 创建子进程：pid_t fork(void) 创建一个子进程，他和父进程有完全相同但是相互独立的虚拟地址空间，并返回子进程的 ID（和父进程不同）
  1. 完全相同：子进程从 fork 处开始运行（但是子进程的 fork 不会创建新进程，而是直接返回 0，而这也是子进程和父进程之后运行情况不同的唯一原因）
     父进程之前在栈和堆中的数据都继承到子进程，父进程打开的文件也继承到子进程中
  2. 相互独立：子进程一旦创建，父子进程之间的操作就没有关联，互不影响对方，并且二者执行的顺序也是完全任意的（不考虑 wait 时）
     我们不应该对执行顺序有任何假设（像习惯的递归调用一样）

进程控制：回收子进程

进程被终止的时候并没有被直接从系统中清除，相反，进程保持在已终止的状态等待父进程的回收，当父进程回收已终止的子进程时，内核将子进程的退出状态传递给父进程，然后抛弃已终止的进程。而在正常情况下，父子进程独立运行，所以 c 语言通过 pid_t waitpid(pid_t pid, int *statusp,int options)

1. pid: 指定等待集合： pid>0 一个子进程； pid = -1 所有子进程
2. statusp: 记录了子进程的返回状态信息
3. option: 对函数的行为略加修改，默认是当子进程集合中的一个进程终止时，返回
4. 返回值：被中断或子进程集合为空返回-1，否则返回引起函数返回的子进程 ID

进程控制：进程休眠

1. unsigned int sleep(unsigned int secs): 让当前进程被挂起指定时间
   返回值：时间到了：0；时间没到：还需要休眠的时间
2. int pause(void): 让当前进程挂起直到该进程收到一个信号
   返回值：-1

进程控制：加载并运行程序

int execve(const char *filename, const char *argv[], const char *envp[]) 函数会在当前进程的上下文中加载并运行一个新的程序，该新程序的文件名为 filename，参数和环境变量是 argv 和 envp

execve 和 fork 的联系

1. fork 创造的子进程和 execve 运行的程序上下文和原来都一样
2. execve 不会返回，而 fork 会返回 2 次
3. fork 创建了一个新的进程，和父进程有不同的 ID，而 execve 只是将当前进程执行的程序换了，没有增加进程，进程的 ID 也没有变