7.2

(1)

内部异常(缺页、越权、越级、整除0、溢出等) 发生在

外部中断(Ctrl-C、打印缺纸、DMA结束等) 硬件层

进程的上下文切换(发生在操作系统层)

一个进程直接发送信号给另一个进程(发生在应用软件层)

(2)

程序指按某种方式组合形成的代码和数据集合, 代码即是机器指令序列, 因而程序是一种静态概念.

进程指程序的一次运行过程. 更确切说, 进程是具有独立功能的一个程序关于某个数据集合的一次运行活动, 因而进程具有动态含义.

同一个程序处理不同的数据就是不同的进程, 它们之间最大的区别是程序是静态的, 而进程是动态的.

(3)

进程的引入为应用程序提供了以下两方面的假象:

一个独立的逻辑控制流:

每个进程拥有一个独立的逻辑控制流, 使得程序员以为自己的程序在执行过程中独占处理器.

一个私有的虚拟地址空间:

每个进程拥有一个私有的虚拟地址空间, 使得程序员以为自己的程序在执行过程中独占存储器.

进程的引入带来的好处:

简化了程序员的编程以及语言处理系统的处理, 即简化了编程、 编译、 链接、 共享和加载等整个过程.

(4)

进程的上下文切换机制和异常/中断处理机制.

(5)

主要完成以下三项工作:

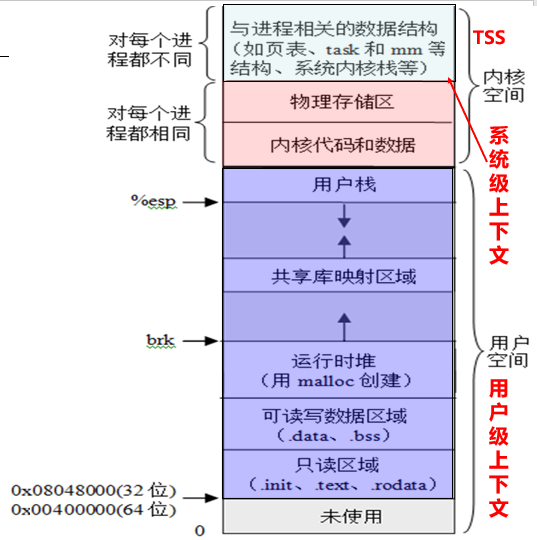
将当前进程的寄存器上下文保存到当前进程的系统级上下文的现场信息中.

将新进程系统级上下文中的现场信息作为新的寄存器上下文恢复到处理器的各个寄存器中.

将控制转移到新进程执行.

(6)

如下图所示, 整个虚拟地址空间分为两大部分: 内核虚拟存储空间(内核空间)和进程虚拟存储空间(用户空间).



对于IA-32, 其内核虚拟存储空间在0xc0000000以上的高端地址上, 用户栈区从内核起始位置0xc0000000开始向低地址增长, 堆栈区中的共享库映射区域从0x40000000开始向高地址增长, 只读区域从0x08048000开始向高地址增长, 只读区域后面跟着可读写数据区域, 其起始地址通常按4KB字节对齐.

(7)

a. cpu捕获到一个异常或中断事件.

b. 保护当前进程的断点和程序状态.

c. 识别异常和中断事件的具体内容, 判断是否需要关中断, 并转到相应的处理程序执行.

d. 根据异常或中断处理程序的结果, 选择终止当前进程或者回到被打断的进程的相应位置继续执行.

(8)

调试程序时的单步跟踪是通过陷阱机制实现的.

(9)

cpu保存的最基本信息包括断点(中断处理后返回的地址)和断点处的机器状态(各种标志信息).

对于IA-32, cpu会保存用户栈的SS和ESP, 以及EFLAGS、CS和EIP(断点).

(10)

主要是通用寄存器中的内容.

(11)

相同之处:

都会从一个程序段跳转到另一个程序段执行, 而且都会返回到被打断的程序段继续执行.

不同之处:

采用的指令不同: IA-32下, 过程调用使用call指令, 过程返回使用ret指令. 系统调用采用陷阱类指令(如int和sysenter指令), 系统返回采用异常/中断返回指令(如iret和sysexit指令).

执行的具体过程不同: 过程调用是在同一个进程内的代码之间进行跳转, 不会改变运行级别; 系统调用则是从用户态下的用户进程代码跳转到内核态的操作系统内核代码去执行, 相当于进行一个特殊的异常处理, 返回时也需要从内核态返回到用户态.

开销不同: 显然, 系统调用指令的开销比过程调用指令的开销大得多.

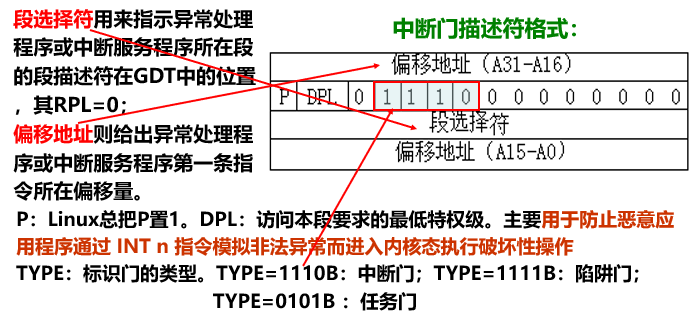
(12)

中断向量表中存放了256个中断向量; 中断描述符表中存放了256个门描述符.

在IA-32实地址模式下, 异常处理程序或中断服务程序的入口地址称为中断向量, 每一个中断向量占4个字节, 其中高16位是段地址, 低16位是偏移地址.

在IA-32保护模式下, 使用门描述符来找到异常处理程序或中断服务程序的入口地址. 每一个门描述符占8个字节, 分为中断门, 陷阱门, 或任务门.

下图显示了中断门描述符的格式(其余的门描述符类似):



中断门描述符和陷阱门描述符都会给出一个16位的段选择符和32位的偏移地址. 利用这两个信息即可找到异常处理程序或中断服务程序的入口地址.

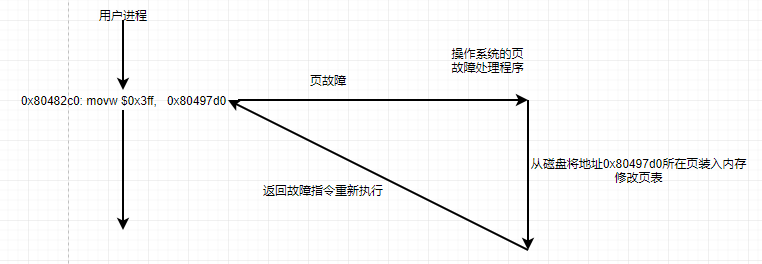
7.4

(1)

linux在初始化时, 通常将所有段的段基址设为0, 所以虚拟地址中的有效地址就是线性地址. 第1行指令的虚拟地址为0x80482c0, 线性地址也为0x80482c0, 因为页大小为4KB, 所以该指令不在一个页的起始位置, 同时这7条指令的总长度加上第1条指令的初始位置不足以产生跨页, 这就说明在执行第1条指令之前的指令时, 上述7条指令被同时装入主存. 综上所述, 上述7条指令的执行过程中都不会在取指令时发生缺页.

(2)

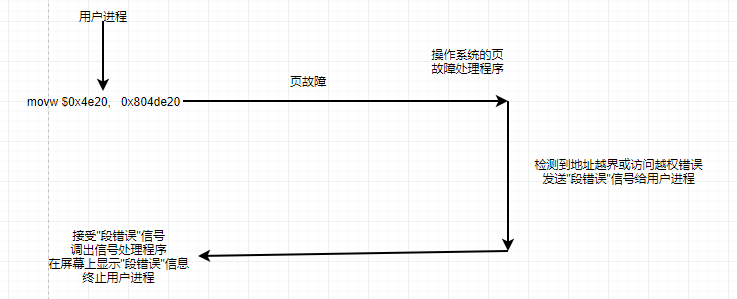
执行第1条指令时, 会发生页故障, 是可以恢复的. 因为对于地址0x80497d0的访问是对其所在页(起始地址为0x8049000)的第一次访问, 因此该页面不在主存中, 执行该指令时将会检测到缺页异常, 即发生了页故障. 此时, cpu从用户态跳转到内核态中执行内核代码, 此情况对应的处理程序是”页故障处理程序”. 在页故障处理程序中首先检查地址越界和访问越权, 这里显然不存在上述情况, 于是该程序将地址0x80497d0所在页面从磁盘调入内存, 处理完毕后再返回这条指令重新执行. 故障的处理过程示意图如下:



执行第2条指令时, 会发生页故障, 是可以恢复的. 因为对于地址0x804a324的访问是对于该地址所在页的第一次访问, 所以会发生缺页故障, 具体处理流程类似于第1条指令.

执行第6条指令时, 不会发生页故障, 只是会破坏原来存放在地址0x804a324中的变量k的值.

执行第7条指令时, 会发生页故障, 并且很有可能是不能恢复的页故障. 因为该指令访问的地址0x804de20偏离数组首地址0x8049000有4个多页面, 很可能已经超过了可读写数据区的范围. 若未超过可读写数据区范围, 则后续执行过程类似于第1条指令的处理过程. 若超过了可读写数据区的范围, 在执行页故障处理程序时, 会检测到地址越界或访问越权错误, 从而页故障处理程序会发送一个”段错误”信号给用户进程, 用户进程接收到该信号后将调出一个信号处理程序执行, 该信号处理程序根据信号类型, 在屏幕上显示”段错误”信息, 并终止用户进程. 该情况的故障处理过程示意图如下:



(3)

会发生除0故障, 不能恢复. 这是因为k是未初始化的全局变量, 在.bss节中, 而.bss节中的变量初始值都会自动设为0. 因此, 在执行第5条指令时, 会发生”除0”故障, 该故障是不能恢复的.

7.5

(1)

执行该段代码时, 系统处于用户态, 因为该段代码完成的是用户程序功能, 其作用是为系统调用准备相应的入口参数. 执行完第5条指令后的下一个时钟周期, 系统处于内核态.

(2)

第5条指令属于陷阱指令. 执行该指令时, 通过系统门描述符来激活异常处理程序. 对应的中断类型号是0x80, 十进制为128. 对应门描述符中的字段P为1, DPL为3, TYPE为1111B.

因为执行系统调用后应执行内核代码, 所以门描述符中的段选择符应该是内核代码段对应的段描述符, 其详细信息见下图中红色部分:



(3)

a. cpu捕获到int $0x80为一条陷阱指令, 从IDTR里获取IDT的首地址, 根据中断类型号0x80, 从IDT中取出第0x80个表项, 其中P为1, DPL为3, TYPE为1111B, 段选择符为0x60.

b. 从GDTR里获取GDT的首地址, 根据段选择符0x60, 从GDT中取出相应的段描述符, 得到(2)中内核代码段对应的信息.

c. 判断当前特权等级CPL与段描述符中的DPL大小, 若CPL小于DPL, 则产生13 号异常(#GP). 同时判断是否发生了特权级变化, 即CPL是否与DPL不同. 这里CPL = 3, 而DPL = 0, 不会产生13号异常, 同时由于CPL和DPL不相等, 所以需要从用户态切换到内核态继续执行.

d. 读TR寄存器, 以访问TSS, 从TSS中将内核栈的段寄存器内容和栈指针装入SS和ESP.

e. 依次将执行完指令int $0x80时的SS、ESP、EFLAGS、CS、EIP的内容(即断点和程序状态)保存到内核栈中, 即当前SS∶ESP所指之处.

f. 将IDT中段选择符(0x60)装入CS, 偏移地址装入EIP. 这里, CS:EIP即是系统调用处理程序system\_call(所有系统调用的入口程序)第一条指令的逻辑地址.

g. 从下一时钟周期开始, 开始执行内核代码, 执行完最后一条指令iret后, 回到第五条指令的下一条指令继续执行.

8.3

(1)

在标准输出设备上打印字符串”Hello,world.“

(2)

执行到第16行和第20行的指令时.

(3)

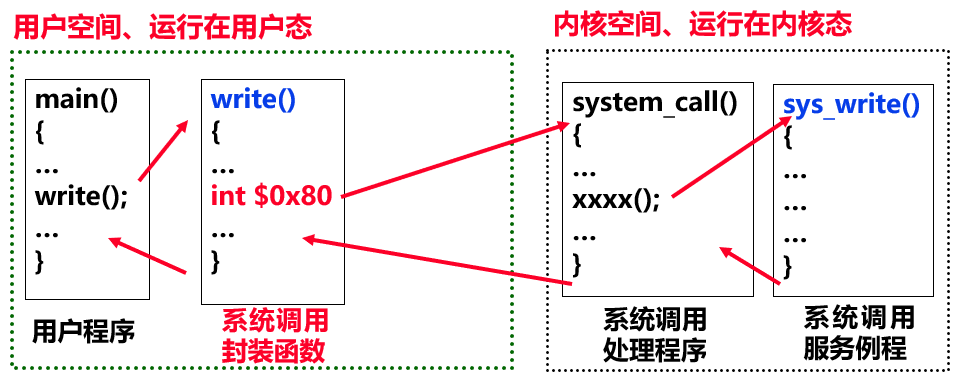
第16行调用了write, 第20行调用了exit.

8.4

(1)

|  |
| --- |
| old %ebp |
| …… |
| 0xe |
| 字符串首地址 |
| 1 |

(2)



(3)

便捷性与灵活性比第三题的实现方式好, 只需改变传给write函数的参数就可以改变程序的功能, 而第三题的实现方式需要重新编写指令. 但程序执行性能不如第三题的实现方式, 因为第三题使用汇编代码, 省去了高级语言中函数调用带来的开销.

8.5

(1)

因为hello.c中使用了C语言的标准输入输出函数printf(), 所以要加上”#include <stdio.h>”. 因为在stdio.h头文件中有printf()函数的原型声明, 经过预处理后, 编译器可以得到printf()函数的原型声明和相关信息, 在链接阶段, 就可以从C语言的标准函数库里得到printf()函数的详细内容, 从而完成链接, 所以main函数在引用它时没有发生错误.

(2)

需要经过预处理, 编译, 汇编, 链接才能在机器上执行hello程序.

预处理: 在高级语言源程序中插入所有用#include命令指定的文件和用#define声明指定的宏.

编译: 将预处理后的源程序文件编译生成相应的汇编语言程序.

汇编: 由汇编程序将汇编语言源程序文件转换为可重定位的机器语言目标代码文件.

链接: 由链接器将多个可重定位的机器语言目标文件以及库例程(如printf()库函数)链接起来, 生成最终的可执行目标文件.

(3)

因为printf()函数默认的输出设备就是标准输出设备stdout.

(4)

该字符串的机器码为:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 48H | 65H | 6CH | 6CH | 6FH | 2CH | 77H | 6FH | 72H | 6CH | 64H | 2EH | 0AH | 00H |
| H | e | l | l | o | , | w | o | r | l | d | . | \n | \0 |

放在hello.o的rodata节中, 在可执行目标文件hello的只读代码段中.

(5)

printf.o模块在静态库libc.a中, 链接后, printf.o的.text节被映射到虚拟地址空间的只读代码段中. 若采用动态链接, 函数printf()的代码在虚拟地址空间的共享库映射区中.

(6)

调用者栈帧情况见8.4(1)

push %ebx //被调用者保存寄存器%ebx入栈

mov 0x10(%esp), %edx //字符串长度14送%edx

mov 0xc(%esp), %ecx //字符串首地址送%ecx

mov 0x8(%esp), %ebx //文件描述符1送%ebx

mov $0x4, %eax //系统调用号4送%eax

int $0x80 //进行系统调用

pop %ebx //恢复%ebx

cmp $0xfffff001, %eax //根据系统调用返回值判断是否出错

jae 8051910 <\_\_syscall\_error> //满足出错条件, 进行出错处理

ret //返回到调用者的过程继续执行

(7)

对于便捷性与灵活性而言, 第5题优于第4题, 第4题优于第3题. 第3题采用汇编语言, 只要参数不同, 就要重新编写指令; 第4题采用write函数进行系统调用, 只能在支持该函数的系统上运行; 而本题使用C标准库函数, 只要求系统具有C语言的设计环境即可运行, 可移植性更好.

对于执行效率而言, 第3题优于第4题, 第4题优于第5题. 因为本题的系统调用过程比第4题更为复杂, 而第3题使用汇编语言, 相对于第4题省去了大量函数调用的开销, 所以第3题的实现方式效率最高.

8.6

(1)

设备驱动程序层.

(2)

与设备无关的操作系统I/O软件层.

(3)

用户I/O软件层.

(4)

设备驱动程序层和I/O中断处理程序层.

(5)

设备驱动程序层和I/O中断处理程序层.

8.8

该台打印机最快情况下每分钟打印6x50x80 = 24000个字符, 即每秒打印400个字符. 每个字符需要1000个时钟周期, 即该打印机每秒需要400x1000 = 400000个时钟周期. 该计算机的主频是500MHz, 对应每秒500x100000 = 50000000个时钟周期, 远大于打印机所需的时钟周期, 所以可以用中断控制I/O的方式来进行字符打印输出.