# 第一次讨论报告

1. 什么是用户态和内核态？两者有何区别？什么是中断和系统调用？两者有何区别？计算机在运行时，是如何确定当前处于用户态还是内核态的？

用户态和内核态是CPU的两种工作状态，其中用户态是用户程序运行时的状态，内核态是操作系统管理程序运行时的状态。两者的区别在于用户态权限级别较低，只能执行一部分级别较低的指令，而内核态权限最高，可执行所有的指令。

系统调用（system call）是应用程序主动向操作系统发出的服务请求。中断（hardware interrupt)是来自硬件设备的处理请求。中断由外部设备引起，采取异步的响应方式，处理方式对于用户应用程序是透明的。系统调用由应用程序引起，响应方式可以是同步或异步的，处理系统调用时可能会需要用户程序进行等待。

计算机运行时，根据程序状态字中的内容确定当前程序处于用户态或者内核态。

1. 计算机开始运行阶段就有中断吗？

是，因为在xv6的bootloader中在开始实模式之前要关闭中断，这表明计算机开始运行时是存在中断的，这些中断可能来自于Bios提供的一部分中断服务程序。

xv6的中断管理是如何初始化的？

1.初始化中断描述符表（IDT）。在trap.c中，tvinit对idt进行初始化，其中vectors中存放着每个中断处理程序的入口地址。

2. 加载中断描述符表，由trap.c中的idtinit函数完成

xv6 是如何实现内核态到用户态的转变的？

当前指令执行的特权级存在于%cs寄存器中的CPL域中。一个程序要进行系统调用时，会根据系统调用编号获取描述符信息，将%cs的域CPL与描述符中记录的特权级DPL比对，当CPL<=DPL时（目标指令权限较高），则保存当前寄存器状态，并根据任务段描述符设置%cs和%eip，从而完成特权级的转变。

XV6 中的硬件中断是如何开关的？

使用cli关中断，使用sti开中断。

实际的计算机里，中断有哪几种？

中断，异常和系统调用。

1. 什么是中断描述符，中断描述符表？在 XV6 里是用什么数据结构表示的？

中断描述符表是一个系统表，其与每一个中断或异常向量相关，每一个向量在表中有相应的中断或一场处理程序的入口地址。

在xv6中，中断描述符表以数组形式存储，中断描述符表的表项即为中断描述符。一般来讲，中断描述符表有256个表项，每个为8字节。

1. 请以某一个中断（如除零，页错误等）为例，详细描述 XV6 一次中断的处理过程。包括：涉及哪些文件的代码？如何跳转？内核态，用户态如何变化？涉及哪些数据结构等等。

以除以0中断为例。

在xv6启动时创建idt，trap.c中的tvinit()函数将其初始化后通过idtinit()加载到内存中。当除0导致的中断发生后，中断控制器通过idt找到该中断的中断描述符(异常号为0)。根据mmu.h中idt表项gatedesc的结构，可以得到中断的类型及中断处理程序的入口地址。在获得入口地址后，cpu跳到alltraps（位于trapasm.S中）处执行。中断被捕捉进入中断服务程序后，首先会保存寄存器状态，并产生一个中断帧(trapframe)并压入内核栈中。然后会调用trap函数(位于trap.c中)。 trap函数首先判断是否为系统调用，本例中不是系统调用，再判断是否为硬件主动发生的中断或不可预料的硬件错误，本例也不是硬件错误，所以trap认为它是一个发生中断前的一段代码中的错误行为导致的中断。这段中断的代码来自于用户程序，xv6将打印其错误信息并设置cp->killed使之稍后被清除。trap函数正常执行结束后回到trapasm.s中，接下来掉用trapret恢复寄存器并返回到中断发生前的状态。

1. 请以系统调用 setrlimit（该系统调用的作用是设置资源使用限制）为例，叙述如何在 XV6 中实现一个系统调用。（提示：需要添加系统调用号，系统调用函数，用户接口等等）

在系统调用后，首先以中断的形式调用trap，trap执行中遇到系统调用将会调用syscall(位于syscall.c)。xv6通过系统调用号与syscall.h中的系统调用号对应，从而通过中断转入相应的处理程序。根据以上流程，当系统调用setrlimit时，首先在proc.c或sysfile.c中添加函数sys\_setrlimit(void)。然后在syscall.c中声明external int sys\_setrlimit(void)，在syscall.h中定义setrlimit的系统调用号，最后在usy.S中添加SYSCALL(setrlimit)。

1. 其他你认为有趣有价值的问题。
2. Elf文件结构

Elf是一种目标文件的格式，即可执行可连接格式的简称。下面是一个典型的Elf可重定位目标文件的格式。

|  |
| --- |
| ELF头 |
| .text |
| .rodata |
| .data |
| .bss |
| .symtab |
| .rel.text |
| .rel.data |
| .debug |
| .line |
| .strtab |
| 节头部表 |

Elf头以一个16字节序列开始，描述了生成该文件的系统的字的大小和字节顺序。Elf头剩余部分包含了帮助链接器语法分析和解释目标文件的信息。

.text：已编译的机器代码

.rodata：只读数据

.data：已初始化的全局和静态变量

.bss：未初始化的全局和静态变量和被初始化为0的全局和静态变量

.symtab：符号表，存放的是在程序中定义和引用的函数和全局变量的信息

.rel.text：一个.text节中位置的列表，在链接时，需要修改这些位置。

.rel.data：被引用的所有全局变量的重定位信息。

.debug：一个调试符号表。

.line：原始行号和.text节中机器指令之间的映射

.strtab：一个字符串表，包括.symtab和.debug节中的符号表，以及节头部表中节的名字。

1. 进程结束后的内存释放问题

绝大多数现代操作系统在进程终止后会自动释放动态分配的内存。包括MacOS X, Linux, 绝大多数现代版本的Windows。大多数现代操作系统都使用内存管理器，所有用户态进程只能看到所谓的虚拟内存，这与程序可以检查的实际系统内存无关。这意味着程序不能直接读取另一个进程的内存或内核内存。而且，当进程终止时，内存管理器将完全“释放”已经分配给进程的所有内存，以便程序内的内存泄漏不会“影响”系统的其他部分（除了可能强制大量的磁盘交换以及可能存在的一些“内存不足”行为）。这并不意味着以任何方式轻松地对待内存泄漏，这只意味着没有一个单一的程序可以随意地破坏现代多任务操作系统上的其他进程（当然，仍存在滥用管理权限的问题）。

而一些比较古老的操作系统则不会自动释放内存，比如MacOS 9，Win 3x和Dos等。