XV6 系统调用

by 1801210840 姜慧强

XV6 在启动的时候依次调用 bootasm.S -> bootmain.c -> entry.S -> main.c

当 XV6 启动时，调用 bootasm.S，系统需要初始化 CPU 运行状态。即将 x86 的 16 位实模式切换到 32 位保护模式。

然后设置初始的 GDT(Global Descriptor Table)全局描述表，将虚拟地址按值映射到物理地址

最后，调用 bootmain.c 中的 bootmain()函数。其主要任务是将内核中的 ELF 文件从硬盘加载进内存，并将控制权转交给内核程序。

- 在 bootmain()中先把暂存空间赋值给 elf 寄存器
- 然后读取第一页 Page(4096B)
- 检查是否 ELF 可执行
- 然后载入所有程序块(根据 ELF 文件头里记录的文件大小载入完整的 ELF 文件)
- 根据 ELF 调用 entry（根据 ELF 文件中的记录入口点）

entry.S 的主要任务是设置页表，初始化分页硬件，然后跳转到 main.c 的 main()函数处，开始整个操作系统的运行。

而 main()函数则依次初始化物理页表，内核态页表，初始化 segment，打印 CPU 信息，初始化中断控制模块，初始化另外一个中断控制模块，初始化 I/O 设备及其中断设备,初始化串行端口，初始化进程控制表，初始化陷入向量表，初始化缓存块，初始化文件控制表，初始化 I 节点区，初始化磁盘，开始用户进程，初始化用户进程的物理页表等等。至此，整个 XV6 系统启动完毕。

```
int
main(void)
{
  kinit1(end, P2V(4*1024*1024)); // phys page allocator
  kvmalloc();      // kernel page table
  mpinit();        // collect info about this machine
  lapicinit();
  seginit();       // set up segments
  cprintf("\ncpu%d: starting xv6\n\n", cpu->id);
  picinit();       // interrupt controller
  ioapicinit();    // another interrupt controller
  consoleinit();   // I/O devices & their interrupts
  uartinit();      // serial port
  pinit();         // process table
  tvinit();        // trap vectors
  binit();         // buffer cache
  fileinit();      // file table
  iinit();         // inode cache
  ideinit();       // disk
  if(!ismp)
    timerinit();   // uniprocessor timer
  startothers();   // start other processors
  kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after
startothers()
  userinit();      // first user process
  // Finish setting up this processor in mpmain.
  mpmain();
}
```

总的来说 BootLoader 起到的是硬件检查功能，通过在 main.c 之前的加载，确保所有硬件是可用的。

内核态、用户态

内核态是操作系统为了保护自身程序在运行时的转态，设定的具有较高权限的程序运行状态。

当用户程序需要调用具有较高优先级的程序时会去调用相应的内核态指令，通过访管指令来实现系统调用。

用户态一般指的是更接近用户端程序的一些进程，不需要更多的系统权限来完成其所需要实现的功能。

中断，系统调用

中断与系统调用是操作系统中实现异常控制流的方式

中断是指体系结构响应内部或者外部事件的机制

系统受到某种信号，打断了目前所执行的应用程序的执行流，进入相应的中断处理程序中，在程序中完成对此信号的事件处理，并返回原来的程序执行流

- 外部中断：时钟、DMA 控制器、鼠标键盘、电源等硬件引发的中断
- 内部中断：由于中断指令/指令出错等原因引发的中断

中断需要软硬件的紧密协同，代码复杂。

而系统调用时指一种应用程序请求操作系统的某种服务的机制

系统调用时通过中断机制来实现，用户程序使用访管指令 INT 指令主动陷入中断，在特定的中断处理程序内实现系统调用的功能。

在 XV6 中，中断与系统调用的代码执行流是一样的，只是参数不一样。

XV6 中使用 Trap 来代指所有由访管指令 INT 引发的中断与系统调用。

在 XV6 中允许用户程序请求的系统调用有 21 种。

- 进程管理：fork, wxit, wait, kill, exec, getid
- 输入输出: read, write, pipe, dup
- 文件管理: fstat, open, close, chmod, link, unlink, chdir, mkdir

中断分为软件中断和硬件中断，当计算机启动的时候在 BIOS 阶段即有硬件中断，如果检查得到硬件不满足要求则会发生中断。而当 main()执行到 picinit()则软件中断也安排到位了。

XV6 中中断初始化发生在 main()之中，在这之前的 bootasm.S 利用 cli 进行关中断操作。

- main()中的 picinit(), oapicinit()初始化可编程中断控制器。
- consoleinit()设置 I/O 设备及其中断
- uartinit()设置串行端口中断
- tvinit()调用 trap.c 初始化中断描述附表。
- 关联的 vectors.S 中的中断 IDT 表项，
- idinit()在调度之前设置 32 号时钟中断
- 最后通过 scheduler()调用 sti()开中断
- 从而完成中断管理的初始化

XV6 通过 trap frame 来控制特权级别, DPL_USER。当进程调用 trapret，此时构造的 trap frame 中保存的寄存器转移到 CPU 中，设置了%cs 的地位，使得进程的用户代码运行在 CPL=3 的情况下，完成内核态到用户态的转义。

XV6 中的硬件中断是通过 picirq.c ioapic.c timer.c 实现的。 ioapicenable 控制 IOAPIC 中断，cpu 通过空着 eflags 寄存器的 IF 位来控制是否要收到中断。

中断管理

中断管理是通过中断描述符表来完成的

中断描述符表是 X86 体系结构中保护模式下用来存放中断服务程序信息的数据结构，其中的条目被称为中断描述符。在 XV6 数据结构中，涉及的数据结构如下

```
// Gate descriptors for interrupts and traps
struct gatedesc {
  uint off_15_0 : 16;   // low 16 bits of offset in segment
  uint cs : 16;         // code segment selector
  uint args : 5;        // # args, 0 for interrupt/trap gates
  uint rsv1 : 3;        // reserved(should be zero I guess)
  uint type : 4;        // type(STS_{TG,IG32,TG32})
  uint s : 1;           // must be 0 (system)
  uint dpl : 2;         // descriptor(meaning new) privilege level
  uint p : 1;           // Present
  uint off_31_16 : 16;  // high bits of offset in segment
};

// Interrupt descriptor table (shared by all CPUs).
struct gatedesc idt[256];
extern uint vectors[];  // in vectors.S: array of 256 entry pointers
```

第 i 条中断描述符，寄存器 cs 中存储的是内核代码段的段编号 SEG_KCODE,offset 部分处的是中断向量 vector[i]的地址。在 XV6 系统中，所有的 vector[i]均指向 trapasm.S 的 alltraps 函数。

中断举例-除零错误

- 应用程序执行 DIV 0
- CPU 硬件发现除零错误，执行保存 CONTETX，执行压栈，跳转到 IDT[0]处程序。
- 然后执行 vector.S 的 vector0 代码，pushl $0;push $0; jmp alltraps；
- 执行 trapasm.S 把寄存器压栈，设置内核数据段，call trap
- 执行 trap.c 先处理系统调用，再检查是不是外部硬件中断，如果都不是，如果在内核态，则系统停机； 如果在用户态，则把用户进程的 killed 设为 1.
- 适当的时候调用 Exit()，在 Exit()中进行一些进程管理的操作，设置当前进程状态为销毁，然后调用调度函数。
- 调度算法选择了另一个进程执行

实现 SETRLIMIT()

setrlimit 是 Linux 中的一个系统调用，用于设置进程资源使用限制。

I. 在 syscall.h 中添加新的系统调用定义 sys_setrlimit()

```
#define SYS_setrlimit 22
```

II. 在 syscall.c 的指针数组内添加新的系统调用函数指针 sys_setrlimit()

```
[SYS_setrlimit] sys_setrlimit,
```

III. 在 sysproc.c 中声明并实现这个函数

```
int sys_setrlimit(int resource) {
    currproc->resource = resource; // Example of set resource
}
```

IV. user.h 中声明 setrlimit 函数的用户调用接口 setrlimit()

```
int setrlimit(int resource, const struct rlimit *rlim);
SYSCALL(setrlimit);
```

V. usys.S 中实现 setrlimit()