



操作系统课程设计实验报告

姓名： 张童

同组： 张延慈 陆宇霄

学号： 201600262022

班级： 16 级泰山学堂计算机取向

指导老师： 杨兴强

目录

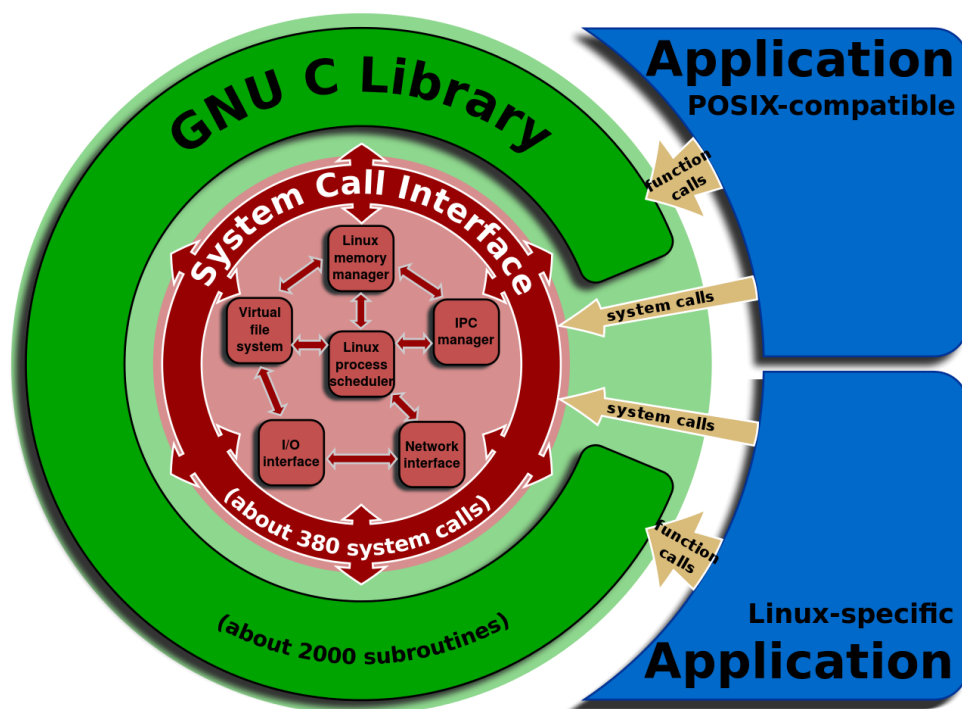
- 综述.....3
- 实验部分.....4
 - 实验目的4
 - 实验环境5
 - 数据输出及Linux运行5
 - 可视化工具的选择5
 - 实验内容6
 - 实验流程6
 - 代码阅读6
 - 数据提取28
 - 可视化及关键帧.....29
- 三、 总结.....32

综述

Linux 操作系统是 UNIX 操作系统的一种克隆版本。它诞生于 1991 年 10 月 5 日（这是第一次正式向外公布的时间）。以后借助互联网，在全世界各地计算机爱好者的共同努力下，发展成为世界上用户最多的一种类 UNIX 操作系统，并且使用人数还在迅猛增加。

Linux 操作系统的诞生(1991 年),发展和成长过程依赖于以下五个重要支柱

1. UNIX 操作系统 (诞生于 1969 年, 版权和专利问题不断, 大公司不愿公开操作系统原理和源码)
2. MINIX 操作系统 (诞生于 1987 年, 意为 Mini UNIX. 教学使用是开源免费的,Linus 从中学习了操作系统的工作原理)
3. GNU 计划 (诞生于 1984 年, 意为 GNU's Not Unix 递归缩写. 宗旨是开发一个类 Unix 的自由软件操作系统)
4. POSIX 标准 (V1 诞生于 1988 年, Portable Operating System Interface for Computing Systems) 描述了操作系统的调用服务接口标准, 便于应用程序在不同操作系统上的移植。这为 linux 系统对应用程序的兼容提供了一套标准. 也是 linux 能流行起来的基础条件之一.
5. Internet 网络 (确保了 linux 系统由众人开发维护, 其发展和推广都离不开 Internet!)



Linux, GNU, POSIX 的关系

正如 Linux 系统的创始人 Linus 所说，要理解一个系统真正的运行机制，一定要阅读其源代码。系统本身是一个整体，具有很多看似不重要的细节，只有在详细阅读过完整的内核代码之后才会对整个系统的运行过程有深刻的理解。我们的操作系统老师也说过，想要写出优秀的，有质量的代码，也要多阅读像 Linus 这样的人写的代码。

而 Linux-0.11 和目前 Linux 内核功能较为相近，又麻雀虽小，五脏俱全。代码总量只有不到两万行代码，十分适合阅读和学习。

实验部分

1. 实验目的

阅读 Linux0.11 源代码，了解各部分的工作原理，清楚进程之间的关系，提取开机到创建进程 4 过程以及进程间通信数据，并将整个过程可视化。

II . 实验环境

古人云：“工欲善其事，必先利其器。”在此实验中，实验的环境以及可视化工具的选择是一件比较重要的事情。

一、数据输出及 Linux 运行

由于 Linux 0.11 年代的久远，在现代机器上肯定是已经不能运行了，所以需要模拟当时的运行环境。一共有以下四种环境：

- 在 Bochs 模拟器上运行 Linux 0.11 进行内核学习，Bochs 模拟了 x86 的硬件环境（CPU 指令）以及外围设备，可以说是比较理想的运行 Linux 0.11 的环境，但是一方面编程体验较差，另一方面操作繁琐，需要反复备份，因此很多时间浪费在与操作系统无关的操作上。
- 为了避免 Bochs 的相关问题，尝试了前人基于 qemu 的实验环境，使得在 Linux 环境下，内核的修改、编译、运行变得容易方便。
- 王天浩学长基于上一环境开发的环境，进一步简化了，数据提取的工作。
- 陈宇翔同学开发的使用 gdb 进行调试的环境

在经过不断尝试中，最后我选择了使用基于王天浩学长的环境来运行 Linux 0.11，而数据的提取一共分为三部分：一部分使用了王天浩环境中的 log 函数，但由于设计开机过程，此时 log 函数无法使用，所以一部分使用了手动阅读代码提取的办法，而后由于在进入用户态之后，王天浩学长的 log 函数也无法使用，另一部分选择使用了 c 语言自带的 printf 函数进行了数据输出。

二、可视化工具的选择

在可视化工具的选择上，一开始我们选择的方案是使用 pyqt，因为一开始的设想是实现一个实时输出、处理数据，并显示出来的系统，但由于找不到接口，以及

工程量过于庞大放弃了。而少了实时性 pyqt 可能就不如 processing 更加适合了。

于是在最后可视化工具的使用上，我们使用了 processing，这一被称为对设计师最友好的编程软件。

III . 实验内容

对 Linux 0.11 进程模块进行可视化。模拟了从开机启动到系统稳定运行起来的整个过程以及进程如何完成创建，销毁，调度，通信这一系列的动作。

主要包括五个子模块：

- 开机启动和进程初始化
- 进程创建
- 进程切换
- 进程结束
- 进程通信

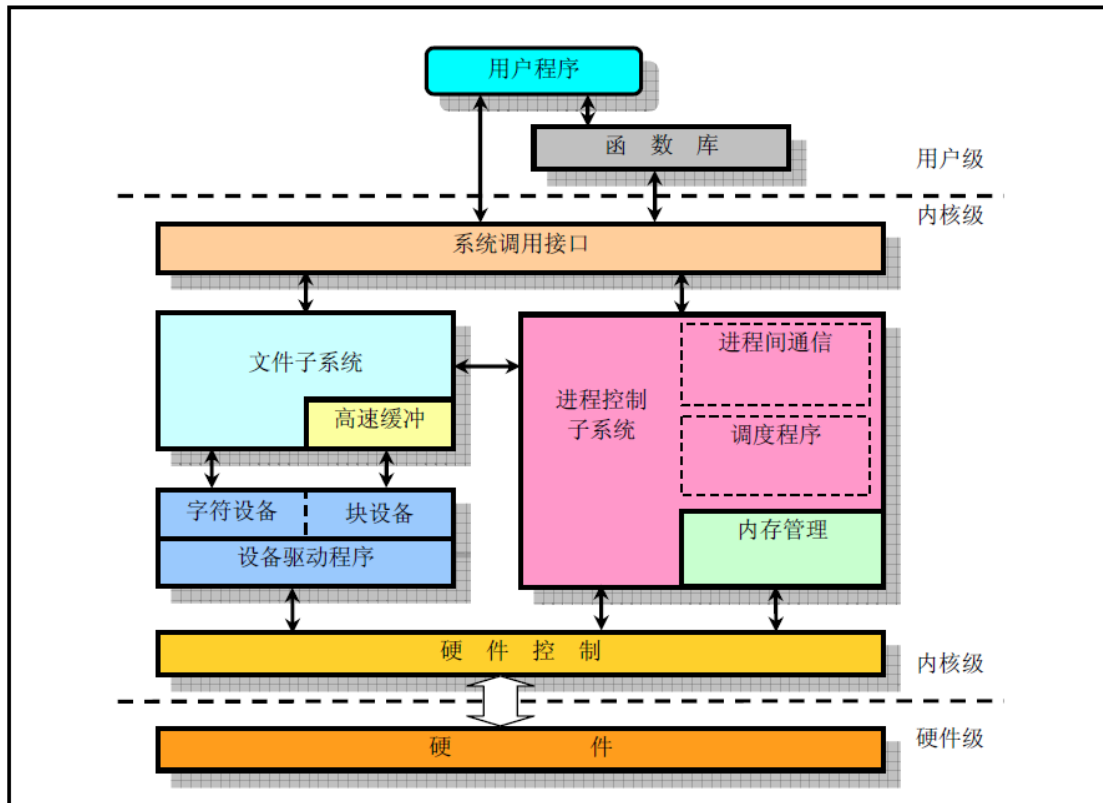
之所以选择进程，是因为在整个系统中，进程模块是一个十分重要的部分，它串联起来了内存管理，文件系统模块。因而或许我们可以通过进程这一角度，去看待整个 Linux 0.11 是如何运行起来的。

我主要的工作是负责开机启动、进程初始化（包括 0、1、2、3、4 进程）和进程通信部分。

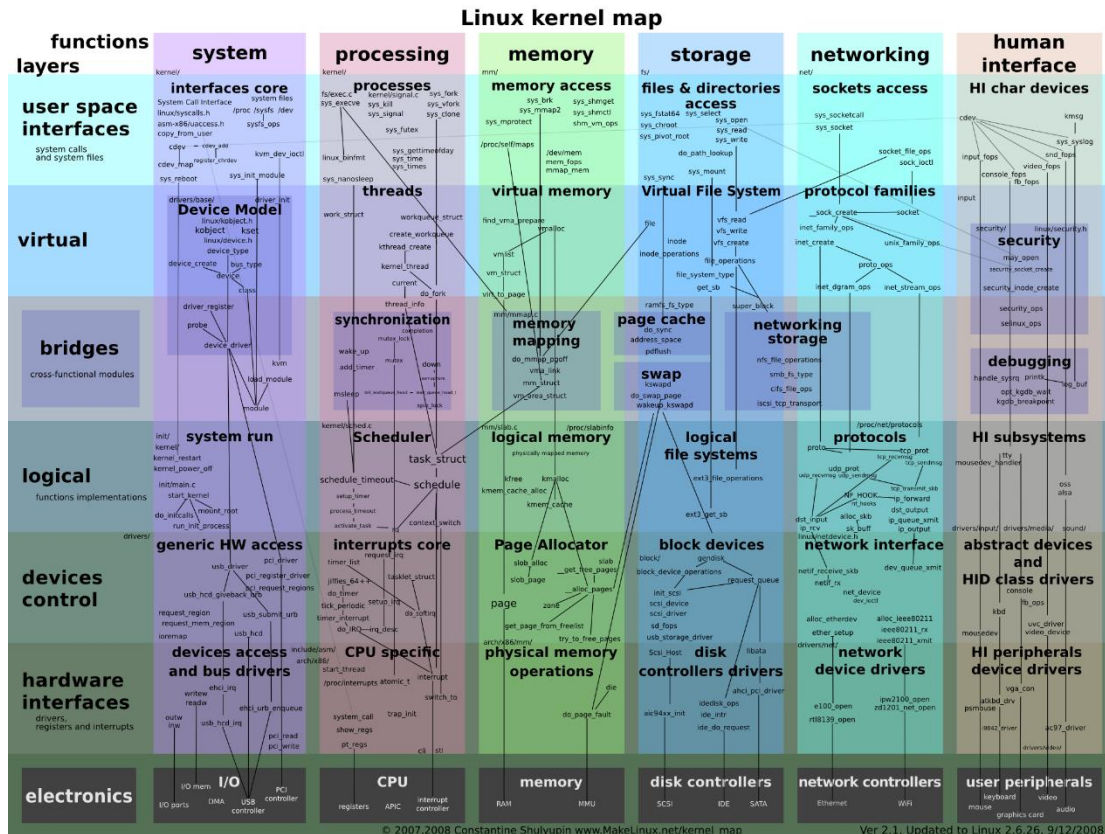
IV . 实验流程

一、代码阅读

内核代码图：



内核函数关系图：



主要阅读的代码：

与进程相关的部分代码：/kernel/fork.c /kernel/exit.c /init/main.c

/kernel/system_call.s /kernel/signal.c 等

因为涉及内存也阅读了内存的部分代码：/mm/memory.c

因为初始化的部分设计开机，阅读了开机部分代码：/boot/bootsect.s

/boot/head.s /boot/setup.s

因为进程通信设计文件系统，阅读了文件系统部分代码：/fs/pipe.c /fs/inode.c

/kernel/signal.c

开机过程：

开机过程主要涉及的是三个文件 bootsect.s、head.s 和 setup.s，下面一一叙述

/boot/bootsect.s

bootsect 位于启动盘的第一个扇区，由 BIOS 自动加载到内存的 0x7c00 的位置，且只加载第一个扇区，共 512 字节。加载后将跳到 0x7c00 来执行代码，此时 CS = 0x7c0，IP = 0，即指向第一条指令代码。

bootsect 首先将自身移动到 0x90000 的位置，然后跳到 0x90000 的 go 标号处执行，重新设置 DS = ES = SS = 0x9000、SP = 0xff00，栈顶在 0x9ff00 处。

接着使用 BIOS 0x13 号中断，将 setup 共 4 个扇区的代码加载到 0x90200 开始的位置。

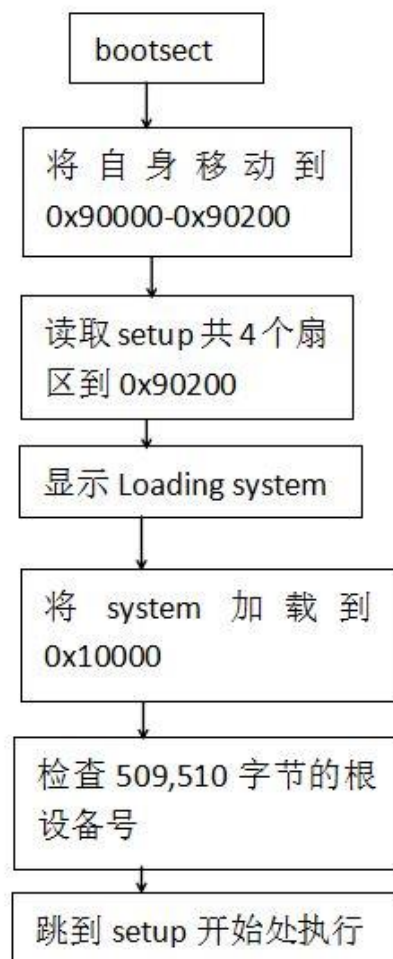
然后读取软盘（第 83 行）的每个磁道的最大扇区数，并填到标号 sectors 两个字节的内存中。这个量可以给读取 system 使用。接着读取光标位置，在屏幕显示标号 msg1 的信息：



再读取 system 的代码到 0x10000 中，注意 system 的大小不会超过 192KB，所以末端为 0x40000。将第 6 个扇区（从 1 开始）开始的读取 0x30000 个字节到内存 0x10000 中。

接着对第 509 和 510 字节值进行检测，如果值不为 0，则跳到 setup 处（CS = 0x9020，IP = 0，139 行）执行。事实上，509 和 510 字节中的初始值为 0x306，也就是第二个硬盘第一个分区。但我们可以在 build 时改变它的值。

程序流程图



/boot/setup.s

setup 利用 BIOS 0x10 中断，读取光标的位置，扩展内存的大小，显示卡参数，以及两个硬盘参数表信息到起始内存 0x90000 中，也就是 bootsect 的代码

被覆盖, 其中硬盘参数表与硬盘分区表不一样, 且起始位置为 0x90080 和 0x90090, 每个都有 16 个字节, 第二个硬盘参数表不存在的话, 初始化为 0。而且 0x901FC 保存的根设备号没有被覆盖。

接着将内核代码 system 从 0x10000 移动到 0 开始的位置, 即 0x10000 ~ 0x90000 的内容移动到 0x0 ~ 0x80000, 每次移动 0x10000 字节, 即 64KB, 共 8 次。

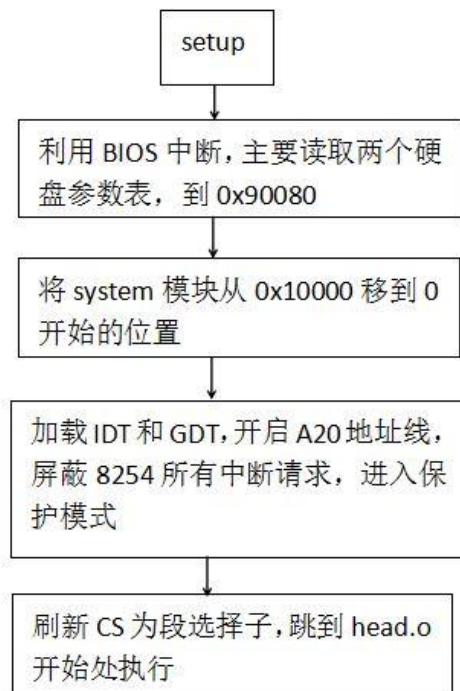
加载中断描述符的段长&地址和全局描述符的段长&地址 (共 6 字节) 到相应寄存器中, 然后开启 A20 地址线, 屏蔽 8254 主从芯片的所有请求, 并将保护模式置位 (CR0 寄存器中的最低位), 此时中断标志没有置位, 也就是没有开启中断。自此从实模式进入保护模式, 但分页未开启。

跳到 CS = 8, IP = 0 (第 193 行), 即地址为 0 的地方开始执行, 也就是 system 的 head.o 开始执行。此时选择子 RPL = 0。

注意: GDT 位于 setup.s 的 205 行 (地址为: 0x90200 + gdt), 共有 3 项, 每项 8 字节。第一项不用, 默认为 0。第二项是代码段, 基地址是 0, 长度为 8MB, 可读执行。第三项为数据段, 基地址为 0, 长度为 8MB, 可读写。这个 GDT 是临时的, 提供给内核启动使用而已。

上述是为执行内核代码作准备。

程序流程图



/boot/head.s

head.s 格式是 AT&T 汇编语言格式, 使用的是 gcc 编译器, 因为它最终要与其他用 C 语言写的模块进行连接。

由于刚开启保护模式, 这时候实模式下的段地址已经不能使用。故 head.s 首先将各个数据段重新设置为段选择子, 且为 0x10, RPL = 0。设置堆栈为 stack_start (位于 sched.c 的 69 行), 使用的堆栈起始是 user_stack, 共一页内存。这个堆栈即为内核初始化时使用的系统堆栈。然后重新设置 IDT, 使用(0x8, ignore_int)作为所有中断发生的入口地址, 初始化 idt 开始的 2KB 内存, 最后 LIDT 加载到寄存器中。ignore_int 仅仅打印一行 "Unknown interrupt!\n\nr"

使用 LGDT 重新设置 GDT, 第一项依旧不用, 第二项为代码段 0x08, 地址为 0, 长度为 16MB, 只可以执行。第三项为内核数据段 0x10, 地址为 0, 长度为 16MB, 可读写。第四项置为 0, 不用, 剩下的 252 项全部初始化为 0。注意 DPL = 0。由于 GDT 重新设置, 缓存无效, 必须重新更新段寄存器。

接着检查 A20 地址线是否开启。检查的方式是向 0x10000(1MB)逐渐写入 1,2,3,4...然后比较内存 0 处是否为该值, 如果是则不断循环, 否则说明已经开启, 放入 0x10000 的值不会放入 0 处。

然后跳到 after_page_tables 标号处执行 (135 行)。布置 setup_paging 执行后执行 main 函数的环境:

物理地址	内容
user_stack + PAGE_SIZE	0
user_stack + PAGE_SIZE - 4	0
user_stack + PAGE_SIZE - 8	0
user_stack + PAGE_SIZE - 12	标号 L6: 死循环
user_stack + PAGE_SIZE - 16	main 函数的入口地址

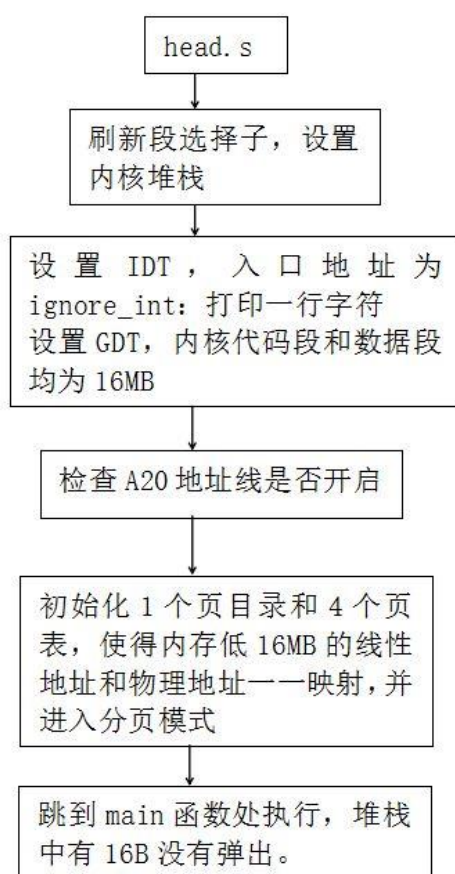
当前栈顶指向 main, 即 $SP = user_stack + PAGE_SIZE - 16$ 。然后执行 `jmp setup_paging`, 跳到 `setup_paging` 处执行, 注意该函数有 `ret`, 即最后会跳到 main 函数处执行。

`setup_paging` 主要是初始化一个页目录和 4 个页表, 共有 16MB 字节的内存, 完成低 16MB 内存的线性地址和物理地址的一一映射, 也就是内核态下物理地址和线性地址是一样的。注意这是从高地址到低地址完成的 (方向位 `std`)。刚好与上述设置的内核代码段和内核数据段的段长一致。然后初始化 CR3 寄存器为页目录地址 0, CR0 的第 31 位 (最高位) 置 1, 开启分页模式。

`setup_paging` 这段代码执行后, 内存地址 0 处的 `head.o` 部分代码将被页目录覆盖。低 5 页内存映像如下:

物理地址（字节）	内容
0	0x1007
4	0x2007
页目录：8	0x3007
16	0x4007
20 ~ 4092	填 0
0x1000	7
0x1004	0x1007
0x1008	0x2007
0x100c	0x3007
	...
0x4ff4	
0x4ff8	
0x4ffc	0xfff007 (起始地址：16MB - 4KB)

程序流程图



进程初始化过程:

进程初始化指的是从进程 0 之前的准备工作，以及进程 0,1,2,3,4 的创建及运行状态和过程，下面详细叙述

main 函数启动任务 0,1

前面主要涉及到获取硬件参数，进入保护模式，开启分页模式，初始化中断描述符表和全局描述符等工作，所以用了汇编语言来写。main 函数位于/init/main.c 中，是用 C 语言写的。注意在用 gcc 编译时，要将 main 改名，这样才能让 heas.s 位于 system 模块的开头，否则 gcc 会认为 main 才是入口。

main 主要是设置中断时执行的函数，块设备和字符设备的初始化，tty 初始化，以及内存缓冲区链表的初始化，系统开机时间的初始化，硬盘的初始化，以及任务 0 的初始化，允许中断处理，然后将任务 0 移动到用户态下执行，启动任务 1 (init 进程)，进入无休止的睡眠。任务 1 挂载根文件系统，设置标准输入输出和错误，并创建 shell 进程，最后循环等待所有子进程退出，回收僵尸进程。下面的工作事实上都是由任务 0 完成的，按照 main 函数调用次序组织。

页框的初始化

main 中对应的函数调用是 **mem_init(main_memory_start,memory_end)**

其中，对于 bochs 来说，main_memory_start = 4MB，memory_end = 16MB，第二个值为 1MB + 从 BIOS 获得的扩展内存的大小，但不超过 16MB。显然主内存通常是 1MB 以上的内存，也就是扩展内存。这个内存主要是用来规划页框。

具体中断的初始化

trap_init():主要是设置了 0~47 号中断的入口地址到 IDT 中，其中 32~47 这 16 个中断对应的是 8259 芯片的中断，并开放了 8259 的两个中断请求。剩下的 8259 中断，将在各个初始化函数中设置。

块设备的初始化

blk_dev_init(): 主要是对 32 个请求项进行初始化, 表示没被使用。

struct request: 定义了完整的请求信息, 包括哪个设备读或写请求哪几个扇区, 然后将扇区读到哪个缓冲区中, 或写哪个缓冲区, 同时还有等待当前项的进程链表。request 请求是一个链表, 通过 next 连接, linux 电梯调度也在这里发生, 涉及到底层 IO 操作。

字符设备的初始化

chr_dev_init(): 该函数为空。

tty_init(): 主要设置串口 1 和串口 2 的中断处理函数, 同时初始化串口 1 和串口 2 的一些硬件属性。

con_init(): 这个程序主要完成显示屏和键盘的初始化, 在显示屏显示显卡的类型, 设置键盘中断的入口函数。

系统时间的初始化

time_init(): 读取当前系统启动时的详细时间, 如 2016.12.05 20:13:14, 但是以 1970.01.01 00:00:00 为起点表示的秒。而 jiffies 表示的系统运行时间, 单位是 10ms, 每 10ms 发生一次日时钟中断, 而该变量会加一, 该变量是计算机世界的“时间”。

start_up + jiffies / 100 表示的将是实际的时间。

任务 0 的初始化

sched_init(): 利用任务 0 的任务状态段和局部描述符段的偏移地址对 GDT 描述符进行设置, 同时选择子加载到相应的寄存器中, 剩余的 63 个任务初始化为空, 描述符也为空。最后设置时钟中断(32 号)的入口地址, 并开启。设置 128 号系统调用中断号的入口地址。其实, 一开始的内核代码执行流就是任务 0 在执行。

内存高速缓冲区

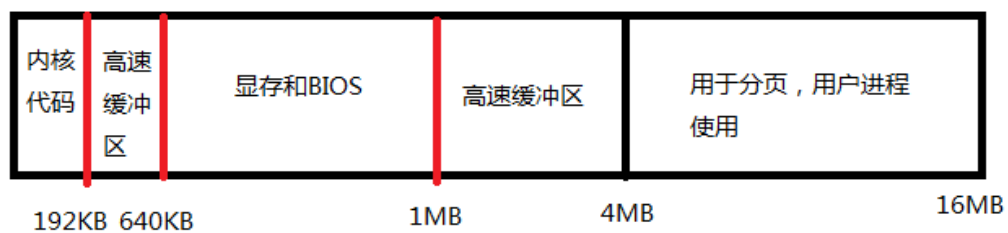
由于在 bochs 中，内存超过 16MB，而 linux 0.11 最大支持 16MB 的内存。故 `buffer_memory_end = 4MB`。注意高速缓冲区必须跳过显存区域 640KB~1MB。

此时的高速缓冲区为内核代码末端~640KB，1MB~4MB。主内存区则为 4MB~16MB。

`buffer_init(buffer_memory_end)`：这个函数首先确定高速缓冲区的位置，内核代码结束的地方是高速缓冲区的开始。跳过显存 640KB~1MB。

注意这里缓冲头从高速缓冲区的起始开始分配，而缓冲块则从后往前，从高速缓冲区的末端开始分配，构成一一对应的管理关系。每个缓冲头指向一块高速缓冲区，缓冲头前后项相互指向，构建空闲内存双向环形链表。最后 `free_list` 指向第一项，初始化整个 `hash_table` 为空。

综合上面内核代码的移动，分页，以及这里的高速缓冲区，内存的映像如下：



硬盘的初始化

`hd_init()`:主要是设置请求函数，设置 46 号中断，即硬盘中断的处理函数，同时将主 8254 的 `int2` 开放，允许从片发出中断。复位硬盘中断 `IRQ14` 屏蔽码。硬盘的主设备号是 `MAJOR_NR = 3`。

软盘的初始化

floppy_init(): 设置软盘中断的设置软盘中断的处理函数, 将 IRQ6 的软盘中断开放。软盘的主设备号是 MAJOR_NR = 2。

任务 0 切换到用户态下执行

```
sti();
```

```
move_to_user_mode();
```

这两个嵌入式汇编宏均在 include/asm/system.h(p389):

在切换到用户态时, 中断已经开启。该宏布置了中断返回现场, 即 SS = 0x17, ESP = 原来的的系统内核栈(user_stack)栈顶(跳转到 main 时有 16 字节没退出), EFLAGS, CS = 0xf, EIP = 标号 1 处的地址。也就是, 系统的内核栈, 经切换之后变为了任务 0 用户态下的栈, 特权级变为 3, 切换后使用的是 LDT 的地址, 然而对应的段基地址仍为 0, 考虑到页目录的基地址为 0, 故任务 0 执行的仍然是内核代码。

接着创建出 init 进程, 然后任务 0 进入无休止的睡眠。

任务 0 的流程图



任务 1(init 进程)的执行过程

init 进程主要是挂载文件系统, 设置标准输入输出错误句柄, 同时创建出进程 2, 运行 shell。init 进程作为所有孤儿进程的父进程, 最后不断取回用户进程的退出码, 回收僵尸进程, 撤销进程表项。此时所有的进程都处在用户态下运行。

挂载根文件系统

`setup((void*)&drive_info)`

drive_info 主要来源于 bios 对硬盘参数表的读取, 两个硬盘共 32 字节。

init 是由任务 0 fork 出来的, 而 drive_info 本来就在编译后的内核数据段中, 类似的还有 argv_rc, envp_rc, argv, envp 等, 这些东西最后都是被共享了, 虚拟地址空间不同而已, 引用的物理页地址是一样的, 都是在上述一开始建立页目录和 4 个页表时建立的。注意 fork 的机制, 只需重新复制页表就可以了, 物理页是一样的, 只要不对数据进行写, 则不会发生重新建立物理内存页的现象 (写时复制)。

该函数对应的系统调用为 sys_setup, 这个函数利用从 BIOS 中读取的 32 个字节 (保存在 0x90080), 获取到两个硬盘驱动的柱面数、磁头数、每磁道扇区数、控制字等, 然后我们可以计算出每个硬盘的扇区总数, hd[0],hd[5]分别代表第一、二块硬盘, 存储起始扇区为 0, 以及整块硬盘的扇区总数。如果第二块硬盘参数中有出现 0, 则表示不存在, 设置 NR_HD = 1。我们可以在 linclude/linux/config.h 中注释掉 HD_TYPE, 这样就可以自定义两块硬盘的上述参数。

然后通过读取每块硬盘的第一个扇区, 读取硬盘分区表, 初始化每个分区的起始扇区和扇区数, 0x301~0x304, 0x306~0x309。注意, 如果硬盘不存在, 则不会执行这个步骤。

执行 rd_load(), 如果我们没有定义虚拟硬盘, 则不会以虚拟硬盘作为根设备启动。最后执行 mount_root(), 函数中的 ROOT_DEV 定义在 fs/super.c 第 29 行, 然而它会在 main.c 中第 110 行被重新赋值, 其值取自 0x901fc, 也就是启动扇区的 508,509 字节, 这个值编译结束后是固定的。对于以软盘作为根设备而言, 一般是第二个软盘, 即 0x21d, 系统启动后会提示: Insert root floppy and press ENTER"

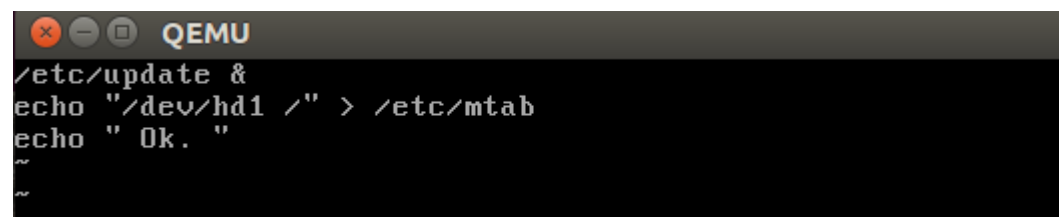
这个函数首先设置全局文件描述符表为未使用，然后初始化超级块数组。读取根设备的超级块，取超级块数组的第一项，超级块位于第三四扇区，即第二数据块，且读取超级块时会把 i 节点位图和数据块位图都读到高速缓冲区来。然后读取该超级块的第一个节点，作为根节点，同时将该超级块代表的文件系统的挂载点设置为根节点。然后读取该超级块中文件系统的数据块空闲块数，空闲 i 节点数，并打印出统计信息。

该函数执行结束后，会打印出 `NR_BUFFERS * BLOCK_SIZE`，表示可用的缓冲区字节数，不包含缓冲头。再打印出主内存数，也就是用于分页的内存。

```
a51263/62000 free blocks
20000/20666 free inodes
3416 buffers = 3497984 bytes buffer space
Free mem: 12582912 bytes
```

启动任务 2 (shell 进程)

进程 2 是任务 1 使用 `fork` 创建的，继承了进程 1 的文件句柄，故它首先将标准输入关闭，并以 `/etc/rc` 作为标准输入，然后使用环境变量和参数(`argv_rc` 和 `envp_rc`)，执行 `/bin/sh` 可执行文件，启动 shell 进程。由于这里的 `sh` 是非交互的（参数前面没有“-”符号），也就是说它只是作为一个命令程序来运行 `/etc/rc` 文件，所以执行完 `rc` 文件中命令之后就会立刻退出，进程 2 也随之结束。`/etc/rc` 文件其实就是 shell 脚本，该脚本中的内容如下：



```
QEMU
/etc/update &
echo "/dev/hd1 /" > /etc/mtab
echo " Ok. "
```

首先执行 `update` 程序，后面单一个 `&` 符号（放在完整指令列的最后端）表示将该指令程序放入后台中工作，这是一个系统里带的二进制文件，至于它了干

了什么就不得而知了。然后执行两个 echo 命令，第一个输出重定向到/etc/mtab 文件，第二个输出到终端界面上。

进程 3

进程 3 就是上面提到的由 shell 指令后台运行 update 程序，这个程序以二进制文件形式存在于系统中，也无法的得知做了什么，把它注释掉发现对于系统没有任何影响。

启动进程 4 (shell 进程)

进程 4 也是任务 1 使用 fork 创建的，继承了进程 1 的文件句柄，故它首先将标准输入关闭，并以/etc/rc 作为标准输入，然后使用环境变量和参数(argv 和 envp 注意这里的参数和进程 2 的参数是不一样的)，执行/bin/sh 可执行文件，启动 shell 进程。因为参数的不同会使得此 shell 进程是可交互的，并且会嵌套在一个大的死循环中，及如果 shell 进程退出后会，重新创建另一个 shell 进程。

在这里关于/bin/sh 程序，这里再说几点，首先对于一个操作系统来说，我们要使用它必然要涉及到交互，shell 就作为用户与内核进行交互的接口，也可以说 shell 就是系统的用户界面。而 shell 本身也是一个程序，只不过它比较特别，操作系统启动之后就一直在运行，等待用户输入，然后解释用户输入的参数，这里的参数大部分都是独立的命令程序名，比如 mkdir，mkfs 等，shell 会将这种命令程序调入内存执行，因此这些命令程序都是 shell 的子程序，完成对应的功能之后就会退出(当然有些在后台程序运行)，然后 shell 又会继续等待用户输入参数。但是 shell 的有一个参数比较特殊，再回看上面进程 2 和进程 4，可以发现两个地方执行/bin/sh 程序，但参数不一样，区别在于“/bin/sh”参数前面是否带“-”符

号，不带“-”表示以非交互方式执行，也就是说带“-”表示以交互方式执行，什么意思呢？实际测试下。

首先去掉 argv 参数的“-”，运行 linux0.11 的情况如下图所示：

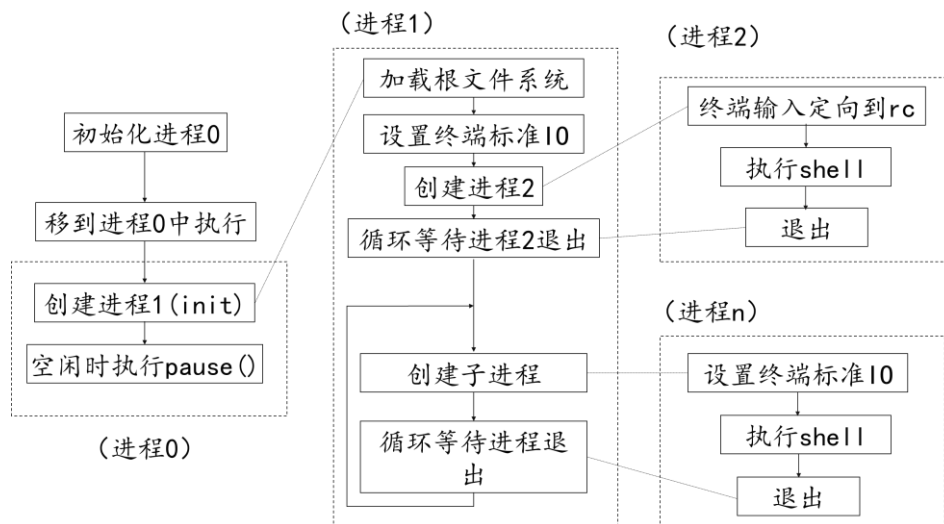
```
37/1440 free blocks
410/480 free inodes
3458 buffers = 3540992 bytes buffer space
Free mem: 12582912 bytes
bash#
bash# ls
bin  dev  etc  root  tmp  usr
bash# cd /
bash# pwd
/
bash# cd /bin
bash# ls
mkfs  mknod  mkswap  mount  sh  umount  vi
bash#
```

可以看到 linux 一样能运行，但是这样用起来似乎不对劲，可以发现命令提示符无法根据当前目录进行相应的显示，想知道当前目录必须要 pwd 一下，所以缺乏了与用户的交互。

加上“-”符号就会像我们普通的 shell 操作一样了，如下图所示：

```
37/1440 free blocks
410/480 free inodes
3458 buffers = 3540992 bytes buffer space
Free mem: 12582912 bytes
[/usr/rootl#
[/usr/rootl# ls
a.out  hello.c
[/usr/rootl# cd /
[/l# ls
bin  dev  etc  root  tmp  usr
[/l# cd dev
[/devl#
```

进程之间关系图



进程间通信

进程通信一共可以有三种方式，一种是通过共享内存方式，一种是通过管道机制，一种是通过信号量，下面主要叙述管道机制和信号量机制，共享内存方式在上一级晁大任组已经做过了，就不赘述了。

管道机制

```
fs --- pipe.c --- sys_pipe() --- 在 file_table[64]中申请两个空闲项
|                               | 分别为 f[0]和 f[1]
|                               |- 在 current 的 filp[20]中找到两个空闲项
|                               | filp[fd[0]]和 filp[fd[1]], 分别指向
|                               | f[0]和 f[1]
|                               |- inode=get_pipe_inode()
|                               |- f[0]->f_inode = f[1]->f_inode = inode
|                               | //指向同一个 indoe
|                               |- f[0]->f_mode = 1//读
|                               |- f[1]->f_mode = 2//写
|
|-inode.c --- get_pipe_inode() --- inode = get_empty_inode()
|                               |- inode->i_size=get_free_page()
|                               |- inode->i_count = 2
|                               |- PIPE_HEAD(*inode) = PIPE_TAIL(*inode) = 0
|                               |- inode->i_pipe = 1
```

管道的操作

假设首先进行读进程，此时管道中还没有数据，size=0，读进程会被挂起，切换到写进程中执行

写进程执行，str1[1024]开始被写入管道。当写完一次后，说明此时管道中已经有数据可以供读取，此时唤醒读进程

读进程虽然被唤醒，但是写进程还没有退出，所以写进程继续执行

```
//尾指针用于读，头指针用于写
fs --- pipe.c --- read_pipe() --- size=PIPE_SIZE(*inode)
|                                     | //size 表示还有多少未读数据
|                                     |- 若 size=0 表示全部读完，唤醒写进程，本进程休眠
|                                     |
|                                     |- chars = PAGE_SIZE-PIPE_TAIL(*inode)
|                                     | //chars 表示管道中还剩余的字节数
|                                     |- if (chars > count)
|                                     |     chars = count;
|                                     | //剩余字节数大于需读取的字节数
|                                     | //取需读取的字节数
|                                     |- if (chars > size)
|                                     |     chars = size;
|                                     | //要读的数据大于管道剩余未读的数据
|                                     | //取剩余未读数据
|                                     |- count -= chars
|                                     |- PIPE_TAIL(*inode) += chars
|                                     | //移动尾指针
|                                     |- PIPE_TAIL(*inode) &= (PAGE_SIZE-1)
|                                     | //指针移动到 4095 以外，回滚到页首
|                                     |- 拷贝内容到用户空间
|                                     |- 唤醒写进程
|- write_pipe() --- size=(PAGE_SIZE-1)-PIPE_SIZE(*inode)
|                                     | //size 表示管道中还有多少空间可供写入
|                                     |- size=0 表示管道中没有空间了,唤醒读进程,本进程休眠
|                                     |- chars = PAGE_SIZE-PIPE_HEAD(*inode)
|                                     | //chars 表示管道中还剩余的字节数
|                                     |- if (chars > count)
|                                     |     chars = count;
|                                     |- if (chars > size)
|                                     |     chars = size;
|                                     |- count -= chars
|                                     |- PIPE_HEAD(*inode) += chars
|                                     |- PIPE_HEAD(*inode) &= (PAGE_SIZE-1)
```



```
| - 将数据写入管道
| - 唤醒读进程
```

假设写管道的过程中发生了时钟中断，写进程的时间片会被削减，但是只要大于零，就不会退出。

```
kernel --- sched.c --- do_timer() --- if ((--current->counter)>0) return
| - schedule()
```

写进程一直写数据到管道中，直到管道写满，会挂起本进程，切换到读进程中执行

```
fs --- pipe.c --- read_pipe() --- size=PIPE_SIZE(*inode)
| - while(!size){
|   wake_up(&inode->i_wait);
|   sleep_on(&inode->i_wait);
| }
```

读进程执行，读出一次管道数据后，管道中有空间可供写入，唤醒写进程

写进程虽被唤醒，但是读进程还没发生调度，因此读进程继续读取数据。

假设期间发送时钟中断，会削减读进程时间片，时间片削减为 0 后切换到写进程中执行。

写进程的尾指针调度之前指向 4095 的位置，此时唤醒后仍然为 4095，进入下一次循环，由于尾指针的位置变了，因此 size 不为 0，chars=1, HEAD=4096，此时与上 4095，HEAD=0，回滚到页首，重新开始写入。

写进程一直写数据，直到管道再次被写满，重新挂起，切换到读进程。

此时读进程时间片为 0，因此，进入 schedule()函数后会重新分配时间片。

```
--- kernel --- sched.c --- schedule() --- for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
|   if (*p)
|     (*p)->counter = ((*p)->counter >> 1) +
|     (*p)->priority;
```

读进程继续执行，直到头指针和尾指针重合，再次切换到写进程。

写进程和读进程轮流执行，直到数据交互完毕。

信号机制

示例进程 processsig

```
#include <stdio.h>
#include <signal.h>
void sig_usr(int signo)
{
    if(signo == SIGUSR1){
        printf("received SIGUSER1\n");
    }else{
        printf("NOT SIGUSER1\n");
    }
    signal(SIGUSR1, sig_usr);
}

int main(int argc, char **argv)
{
    signal(SIGUSR1, sig_usr); // 绑定信号处理函数
    while(1){
        pause();
    }
    return 0;
}
```

示例进程 sendsig

```
#include <stdio.h>
int main(int argc, char **argv)
{
    int pid, ret, signo;
    int i;

    if(argc != 3){
        printf("Usage <signo> <pid>\n");
        return -1;
    }
    signo = atoi(argv[1]);
    pid = atoi(argv[2]);

    ret = kill(pid, signo);
    for(i=0; i<1000000; i++){
        if(ret != 0){
            printf("send signal error\n");
        }
    }
}
```

```
    return 0;
}
```

首先执行进程 processsig 进程,然后执行 ./sendsig 10 160 给 processsig 发送信号

processsig 进程执行，其中 restorer()函数由内核指定

```

--- kernel --- signal.c --- sys_signal() --- tmp.sa_handler = handler//设置信号处理函数
| - tmp.sa_restorer = restorer//设置恢复现场函数
| - current->sigaction[signum-1] = tmp
|   //设置当前进程的信号

```

processsig 进程进入可中断等待状态，切换到 sendsig 中执行

```
--- kernel --- sched.c --- sys_pause() --- current->state = TASK_INTERRUPTIBLE
                                     |- schedule()
```

send_sig 会执行 `ret=kill(pid, signo)` 这一行代码，对应 `sys_kill()` 函数

```

--- kernel --- exit.c --- sys_kill() --- 找到需要 kill 的进程 p
      |                               |- send_sig(sig,*p,0)//发送信号 0
      |                               |- send_sig() --- p->signal |= (1<<(sig-1))//设置信号位图

```

发送完成后，sendsig 进程退出，进入进程 0，进程 0 中执行 schedule()函数，发

现 processsig 进程接收到信号，于是唤醒 processsig 进程

```

---kernel---sched.c---schedule()---for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
    |   if (((*p)->signal&
    |   ~(_BLOCKABLE & (*p)->blocked))
    |   && (*p)->state==TASK_INTERRUPTIBLE)
    |       (*p)->state=TASK_RUNNING;
|-再次遍历 task_struct[32],发现只有 processsig 处就绪态
|- switch_to(next)

```

processsig 进程开始执行，会继续在循环中执行 pause()函数，这是一个系统调

用, 当系统调用返回时, 会执行 `ret_from_sys_call` 标号处, 最终会调用 `do_signal()`

函数

```

--- kernel --- system_call.s --- ret_from_sys_call --- do_signal()
    |
    |- signal.c --- do_signal() --- old_eip=eip
        | //备份 eip
        | sa = current->sigaction + signr - 1
        | //获取接收到的信号
        |- sa_handler = (unsigned long) sa->sa_handler
        | //获取信号处理函数

```

```
| - *(&eip) = sa_handler  
| //返回后跳转到处理函数执行  
| - old_eip, eflags, edx, ecx, eax,  
| signr, sa->sa_restorer 依次压到用户栈  
| - 返回, 跳转到 sa_handler 处执行
```

信号处理函数结束后, 会执行 ret 指令, 该指令会将栈顶的内容放到 eip 中, 最终跳转到 eip 执行, 此时栈顶是 sa->sa_restorer, 因此跳转到 restorer 处执行。restorer 会依次将 eax, ecx, edx 和 eflags 恢复, 然后执行 ret, 此时栈顶是 old_eip, 因此跳转回到 processsig 函数继续执行

二、数据提取

数据的提取部分将只会简述数据是如何提取的, 并给出一定的例子, 数据的含义都体现在上述对过程描述, 数据则会存在于相应的文件夹中。

开机部分的数据提取, 一是以为数据大部分存在于寄存器中, 比较难以提取, 二是很多东西都是写死在代码里的, 可以从代码一步步解析出来, 所以穿插在了上面对代码解读部分。

在 main 时, 切进入进程 0 之前仍处于内核态, 此时可以使用 log 函数, 也提取了一部分数据, 举例如下:

```
{'module': 'porcess', 'event': 'schedule_initial', 'provider': 'zt', 'data': {'FIRST_TSS_ENTRY': 4}  
{'module': 'porcess', 'event': 'schedule_initial', 'provider': 'zt', 'data': {'firstl_a': 0, 'b': 0}  
{'module': 'porcess', 'event': 'schedule_initial', 'provider': 'zt', 'data': {'secondl_a': 0, 'b': 0}
```

在初始几个进程的一部分数据因为已经转入用户态了, 很多数据的提取学长的 log 已经无法使用的, 使用了 c 自带的 printf 函数, 输出到屏幕上, 然后手动记录。代码修改举例:

```
printf("pid: %d ,task struct: father:%d", pid, task[pid]->father);
```

```
Booting from Floppy...  
Loading system ...  
Partition table ok.  
51280/62000 free blocks  
20007/20666 free inodes  
3435 buffers = 3517440 bytes buffer space  
Free mem: 12582912 bytes  
Ok.  
pid: 3pidwait: 3[/usr/root]#
```

但也有一些数据可以用 log 函数输出，举例如下：

```
{'module':'porcess','event':'find_empty_process','provider':'zt','data':('The input parameter:3 67108216 4092 1 23 30523 99578 12 309952 23 23 23 235695 15 582 67108204 23, 'state':2,'pid':3,'father':2,'counter':15,'start_time':6)}
```

管道机制数据展示：

```
{'module':'porcess','event':'schedule_read_pipe','provider':'zt','data':{'size:0,chars:0,count=0}}
{'module':'porcess','event':'schedule_write_pipe','provider':'zt','data':{'size:1024,chars:1,HEAD:4096}}
```

信号量机制：

```
{'module':'porcess','event':'schedule_do_signal','provider':'zt','data':{'old_eip:0x7D00,sa_handler:3}}
```

三、可视化部分

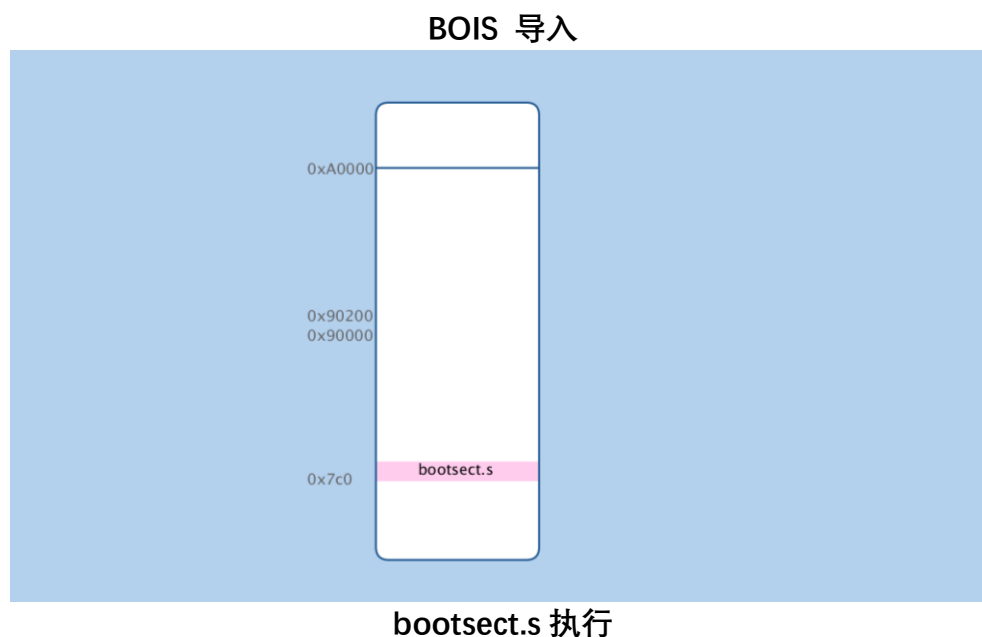
可视化工具选择的是 processing，这一被称为最适合设计师的编程软件。

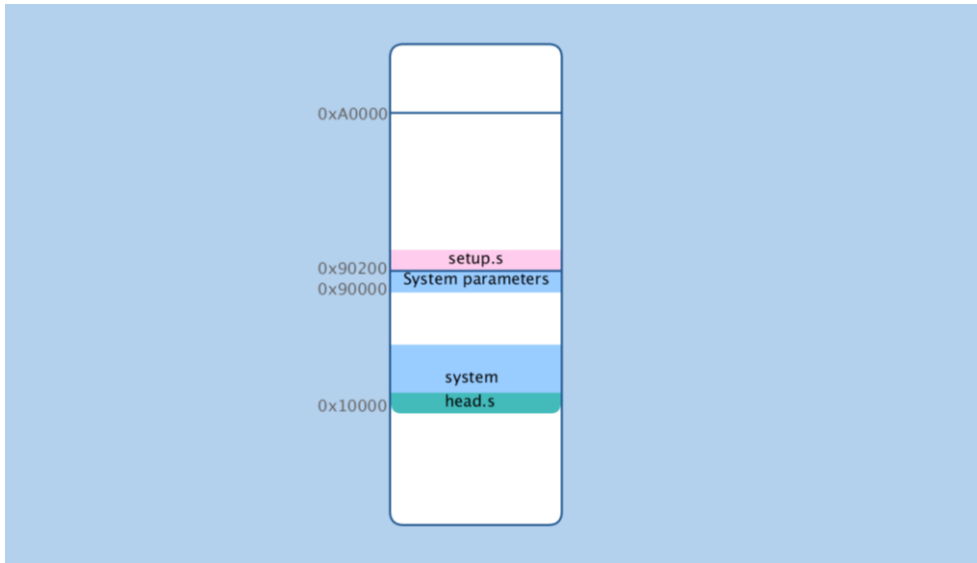
可视化部分采用了时间管理机制，一个个关键帧时间在代码中处于不同的时间，确定每个函数应该执行的时间段，利用 draw 函数不断的重画整张画布的特点，不断的刷新画布来实现动画。最后我们最后渲染出了一个五分三十七秒的视频，视频见对应文件夹中。

因为时间原因，进程通信部分没有来的实现可视化。

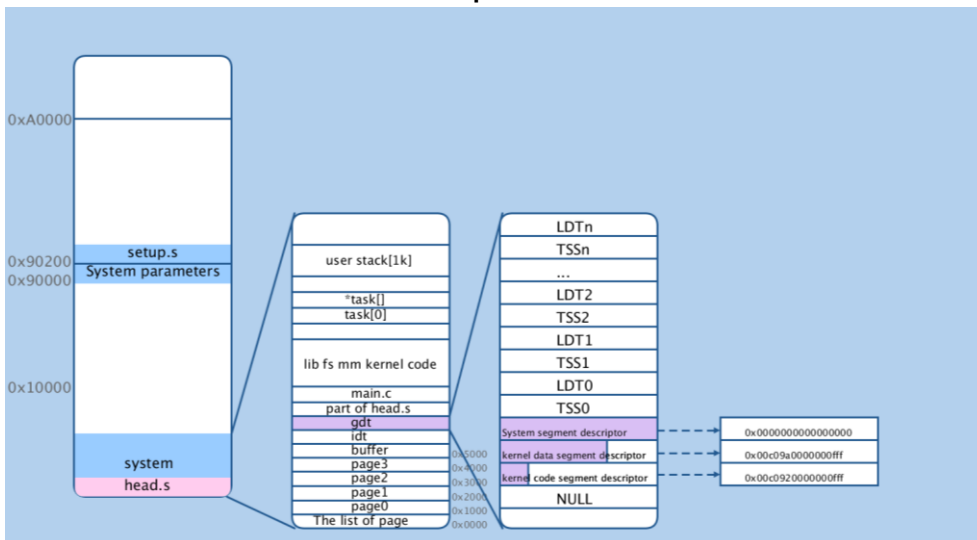
可视化代码共计 803 行。

关键帧展示如下：

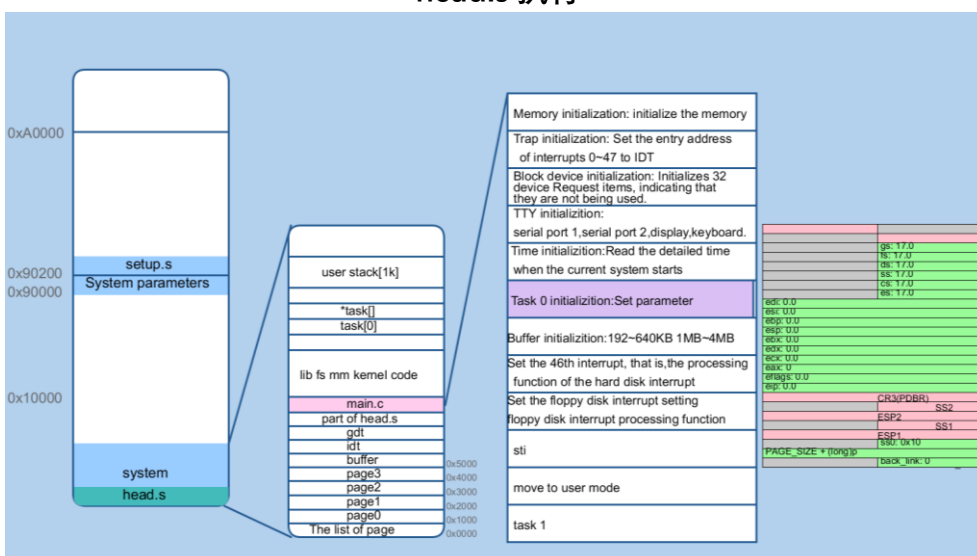




setup.s 执行



head.s 执行



初始进程创建的父亲关系

pid = 0

pid = 1

pid = 2

pid = 3

chosen

state 0

state 1

state 2

state 3

Command Line

INFORMATION

EXIT PROCEDURE 2

ss

esp

cs

eip

ds

es

fs

edx

ecx

ebx

eax

signr

pid = 0

pid = 1

pid = 2

pid = 3

chosen

state 0

state 1

state 2

state 3

Command Line

INFORMATION

Schedule is called by sys_pause.
Procedure 1's counter is 13.

ss

esp

cs

eip

ds

es

fs

edx

ecx

ebx

eax

signr

pid = 0

pid = 1

pid = 3

pid = 4

chosen

state 0

state 1

state 2

state 3

Command Line

INFORMATION

Schedule is called by sys_waitpid.
Procedure 4's counter is 15.

ss

esp

cs

eip

ds

es

fs

edx

ecx

ebx

eax

signr

三、总结

这一次对 Linux 0.11 源代码的阅读及可视化，让我近距离感受了 Linus 这一伟大人物代码的书写风格，以及 Linux 这一伟大的操作系统的运行过程。并且因为选择的是进程模块，我们也通过进程角度看到了操作系统是如何从开机到稳定运行起来，以及进程如何完成创建，销毁，调度，通信这一系列的动作。在可视化方面，及时的从 pyqt 转到 processing 也是根据选用了最适合的软件实现最贴切的功能。

对下一级的建议则是，环境的选择时比较重要的，有了一个好的实验环境可能会省很多事，比如如果早一些有陈宇翔的环境，我的数据提取就可以统一的使用陈宇翔同学的环境提取了。

最后感谢杨兴强老师这一学期的指导与督促。