|  |  |  |
| --- | --- | --- |
| 南农大  **计算机操作系统课外**  **实验报告**  XH2 | | |
|  | 题 目: | 多级反馈队列进程调度算法在裸机上的实现 |
|  | 姓 名: | 邱日 |
|  | 学 院: | 信息科学与技术学院 |
|  | 专 业: | 计算机科学与技术 |
|  | 班 级: | 计科151 |
|  | 学 号: | 19215116 |
|  | 指导教师: | 姜海燕 职称: 教授 |
|  | 指导助教： | 王睿 |
| 2017年10月 20 日 | | |

**姓名 邱日 学号 19215116 申请成绩等级 A**

**QQ号 3103204417 联系电话 18260068503 E\_mail 3103204417@qq.com**

**申请成绩等级（A：优秀；B：良好；C：一般：D：及格）**

# 一、实验目的

1.1通过程序仿真掌握并发环境、进程 PCB 与控制操作、进程切换以及进程调度算法的原理、过程与实现步骤;

1.2 理解并动手实现进程调度的先来先服务、时间片轮转法、多级反馈队列算法

1.3 强化算法设计和数据结构。

# 二、设计思路与实现内容

1. 设计思路

CPU进程调度可选用的调度策略有：先来先服务调度策略、静态优先级抢占式调度策略、动态优先级抢占式调度策略、时间片轮转调度策略以及多级反馈队列调度策略。其中多级反馈队列调度策略（Multi-Level Feedback Queue,MLFQ）具有较好的性能，能满足各类应用的需要。

本实验实现了先来先服务调度策略、时间片轮转策略以及多级反馈队列调度策略。

实验参照 Linux0.11 task\_struct 的数据结构内容设计将计算机的进程抽象为一个结构体：task\_struct(进程控制块)，内容主要包括：进程编号（proc\_id），进程优先级数（priority）、进程创建时间（start\_time）、进程状态（proc\_state）、进程运行时间（run\_time）、保存进程状态的内部结构体（tss32\_struct）；

其中tss32\_struct对应硬件上intel 32位x86处理器系列一个特殊的用于保存任务状态的结构：任务状态段(task state segment,TSS)。tss32\_struct由硬件决定，共104字节，其中包括进程的通用寄存器（eax等）及段寄存器的值和进程当前状态（PSW）等信息。

首先由用户事先在时钟中断响应函数IntoReady中指定多个任务的开始时间，其初始时刻的优先级等，当系统开始运行时，到达相应时刻将会按照用户之前的要求动态地创建任务。在开时钟中断之前，系统已经创建好三级队列的队列头并分配好空间，其分别为q1,q2,q3，其中q1,q2这上两层队列采用时间片轮转调度，q3为最后一级队列，采用先来先服务调度算法。本实验采用的多级反馈队列算法由上层的时间片轮转算法和最后一层的先来先服务算法构成。

全局布尔变量currentTASKstate表示当前是否有任务，由于并不是每次发生时钟中断都要进行进程调度，每次时钟中断都判断一下是否当前任务在运行并且还将运行，若真，则不调度，直接return,否则依次调度。具体到本实验，采用了三级队列。

**三级反馈队列算法描述(顶层设计)**：

1. 若第一第二级等待队列不全为空，那么对第一第二级队列进行时间片轮转调度
2. 判断是否当前任务在运行并且还将运行，若真，则不调度，直接return,否则若q3不空，取q3队头元素运行。

**第一第二级队列的时间片轮转调度**：

对第一级：

1. 若当前无任务运行，取q1队头等待任务运行
2. 若当前有任务运行，但此时它的运行时间已经超过所需时间片，销毁该任务。
3. 若当前所运行任务优先级为1并且到了它的时间片，判断第二层任务队列是否达到上限，若不满，将当前任务优先级改为2,加入第二级等待队列;否则当前任务修改优先级后加入第三级队列。

对第二级：

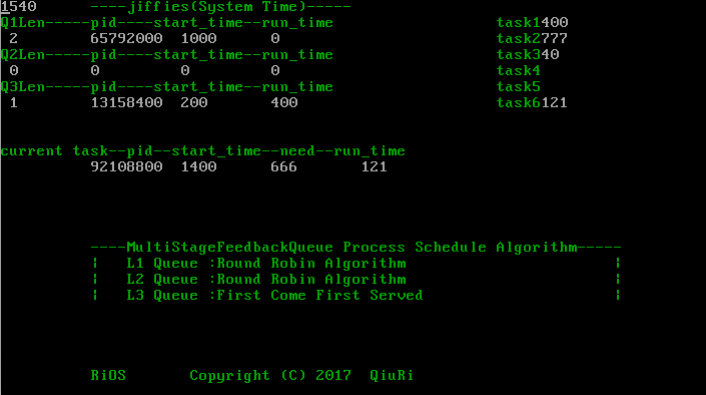
1. 若当前无任务，取q2队头等待任务运行
2. 若当前有任务运行，但此时它的运行时间已经超过所需时间片，销毁该任务。
3. 若当前任务优先级为2并且到了它相应的时间片，若尚未完成，将其优先级改为3放入第三级队列等待。

**最后一级的先来先服务调度算法**：

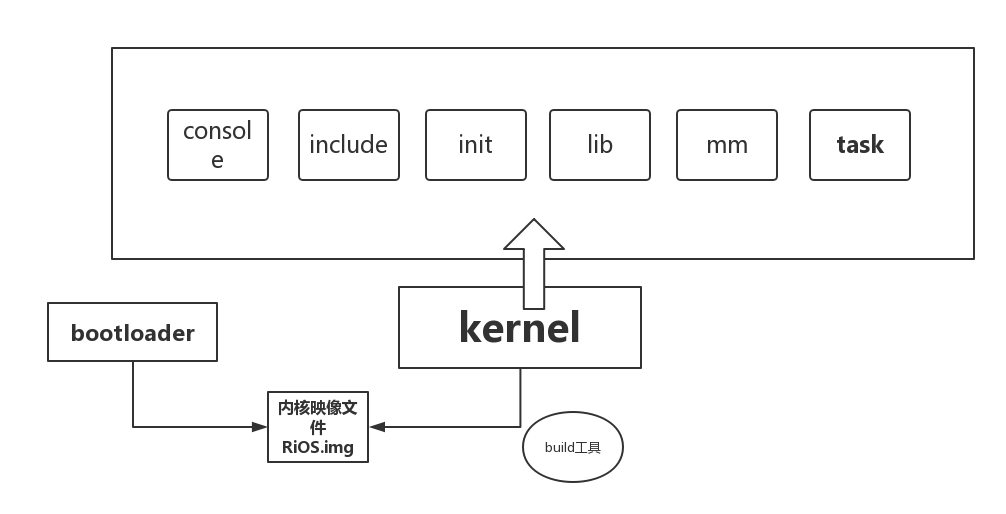
1. 若当前无任务，且q3不空，取q3队头等待任务运行  
    2.若当前有任务运行：但尚未运行完，直接return不调度;否则销毁该任务。若三个队列都为空则return.

（二）实现内容

采用AT&T的x86汇编和C语言作为开发语言，实现了一个可以直接运行在裸机之上的简单内核，目前已经通过测试：在Ubuntu（Linux）系统上利用虚拟机qemu完成测试并录屏;在我的电脑上通过U盘启动，在裸机上也成功运行。两者的效果是一样的，读者如果想要在裸机上测试，我已经编译好我的操作系统内核镜像RiOS.img，放在源码包中，读者可自行烧写到U盘，并从U盘启动进入我的RiOS系统，其过程和拿U盘装windows操作系统是一样的。 下图为运行结果（其实是动态的，这里只是截个图）。



操作系统的启动，首先需要一个512字节的bootloader,在操作系统内核运行之前运行。可以初始化硬件设备、以便为最终调用[操作系统内核](https://baike.baidu.com/item/%E6%93%8D%E4%BD%9C%E7%B3%BB%E7%BB%9F%E5%86%85%E6%A0%B8" \t "/home/curie/Documents\\x/_blank)准备好正确的环境。进程调度算法采用多级反馈队列调度算法（MLFQ），具体实现是三级反馈调度，设三个优先级不同的队列。



RiOS内核系统架构图

我将本人的小操作系统内核命名为RiOS,以上为我的RiOS内核系统架构图，其中task模块为本次实验的重点内容。其实task模块进程调度算法只是我系统的其中一部分，我此次代码的实质是一个可以在裸机上运行的操作系统内核。当然，它还很不完善，没有文件系统等功能，我将在今后的学习中继续写它，在操作系统课程设计时预计将会更加完整。

（1）时钟中断的裸机实现

关于中断：在本次实验中，大部分同学的中断都不是真正的中断，这是很不正确的。操作系统的革命性基础是中断，没有中断，何来并发？没有并发，何来并发环境？在一个非并发环境，即顺序环境写本次实验除了复习数据结构，对于操作系统的学习是没有任何意义的。我阅读并参考Linux0.11（Linus 1991）源代码,通过对可编程计数器/定时器8253和中断控制器8259

发送指令字，实现硬中断。按照实验说明，我每20ms产生一次硬件中断。

通过32位x86处理器的lidt指令，设置idt表（即interrupt descriptor table 中断描述符表）并在表中填入相应记录，实现时钟中断。

1. CPU 与寄存器

不借助Windows或Linux等操作系统，独立运行于裸机，采用真实的CPU和寄存器。本质上是一个小操作系统内核。

# 三、公共数据结构设计与实现（重点）

**（说明：按照实验内容分对象论述。可用文字、示意图、流程图和有注释核心函数伪码来论述。不能只画图，必须用文字说明清楚）**

实验参照 Linux0.11 task\_struct 的数据结构内容设计将计算机的进程抽象为一个结构体：task\_struct(进程控制块)，内容主要包括：进程编号（proc\_id），进程优先级数（priority）、进程创建时间（start\_time）、进程状态（proc\_state）、进程运行时间（run\_time）、保存进程状态的内部结构体（tss32\_struct）；

其中tss32\_struct对应硬件上intel 32位x86处理器系列一个特殊的用于保存任务状态的结构：任务状态段(task state segment,TSS)。tss32\_struct由硬件决定，共104字节，其中包括进程的通用寄存器（eax等）及段寄存器的值和进程当前状态（PSW）等信息。

PCB 控制块的数据结构设计:

typedef struct task\_struct{

long proc\_id;/\*进程编号\*/

long start\_time;/\*进程创建时间\*/

long proc\_state;/\*进程状态\*/

long run\_time;/\*进程运行时间\*/

long priority;/\*优先级\*/

long timepiece\_needed;/\*本进程运行完需要的时间片\*/

int proc\_screen\_x;/\*任务显示位置X坐标\*/

int proc\_screen\_y;/\*任务显示位置Y坐标\*/

tss32\_struct proc\_tss32;

}task\_struct;

各个元素定义如下：

1.proc\_id

进程标识名。long类型,可以唯一标识一个进程。根据当时的系统时间jiffies生成伪随机数（其值形式例如39475200）,作为进程标识。

1. start\_time

进程创建时间。long类型。由系统提前分配，表示进程何时产生。

1. proc\_state

此进程当前状态。其值可以为TASK\_RUNNING运行态、TASK\_READY就绪态、TASK\_BLOCKED阻塞态。

1. run\_time

进程已经运行的时间。long类型,表示进程已经被CPU执行的时间。

1. priority

进程当前的优先级，其值可以动态地改变，取值范围在1、2、3之中，分别对应三级等待队列。

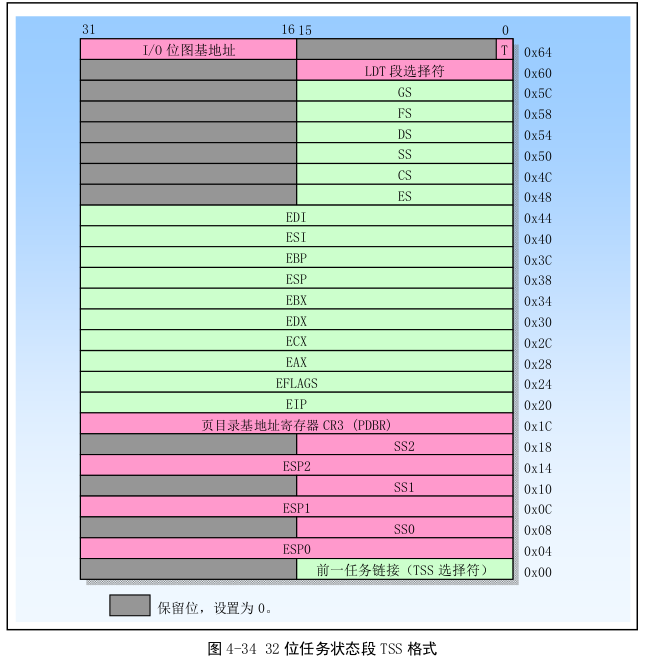
1. timepiece\_needed

本进程运行完需要的时间片。long类型，是完成该任务需要被CPU执行的总时间。

1. proc\_screen\_x

任务显示位置X坐标，与其匹配的还有proc\_screen\_y。proc\_screen\_x与proc\_screen\_y这两个量决定了最终在计算机80×25文本显示模式下显示该任务的x和y坐标。

1. 结构体proc\_tss32

用于恢复一个任务执行的处理器状态信息，对应TSS。自定义的104字节的tss32\_struct类型。在intel x86系列32位CPU中，用于恢复一个任务执行的处理器状态信息被保存在称为任务状态段TSS（task struct segment）PCB结构体task\_struct中 tss32\_struct 类型的内部结构体proc\_tss32对应着它，共104字节，这是由硬件决定的，专门用于任务切换，不可人为更改，图为赵炯著《Linux0.11内核完全注释修正版》第123页32位任务状态段TSS格式。

分为动态字段和静态字段。其中动态字段分为通用寄存器字段、段选择符字段、标志寄存器EFLAGS字段、指令指针EIP字段等，静态字段主要有LDT(local descriptor table)段选择符字段、特权级0、1和2的堆栈指针字段等。这104字节的内容由intel硬件决定，不可人为更改其结构，因此不展开论述。

说明:

★请求运行的并发进程个数可以随机产生[5-10]以内随机整数,也可以事先设定初始值。

★进程创建时间(InTimes):程序运行从 0 开始计时,生成进程的时间间隔控制在 1 分

钟以内,需要保证出现进程并发情景。

★进程运行时间(RunTime):统计记录进程当前已运行了多少时间,此字段开始时为

空,进程运行过程中不断保存和记录。

★进程包含的指令数目(InstrucNum):用[5-20]以内的随机整数产生;

★编号指令(Instruc\_ID):进程所执行指令序号,根据 InstrucNum 数字生成。本试验

假设 CPU 在执行一条机器指令时必须完成,不可以中断。

★每条指令的类型标志(Instruc\_State,0 表示系统调用、1 表示用户态计算操作、2 表

示 PV 操作):用随机数产生 0、1、2。

当 Instruc\_State=0 时,发生系统调用,CPU 进行模式切换,原进程进入阻塞态;

当 Instruc\_State=2 时,发生 PV 请求,进程进入阻塞态;

如果希望简化问题,可以不考虑产生 Instruc\_State=0 的指令;对于学有余力的同学来说,

可以产生 Instruc\_State=0 的指令,进行系统调用。

★指令运行时间(Instruc\_Times):该条指令完成需要的时间,本实验假设 CPU 在执行

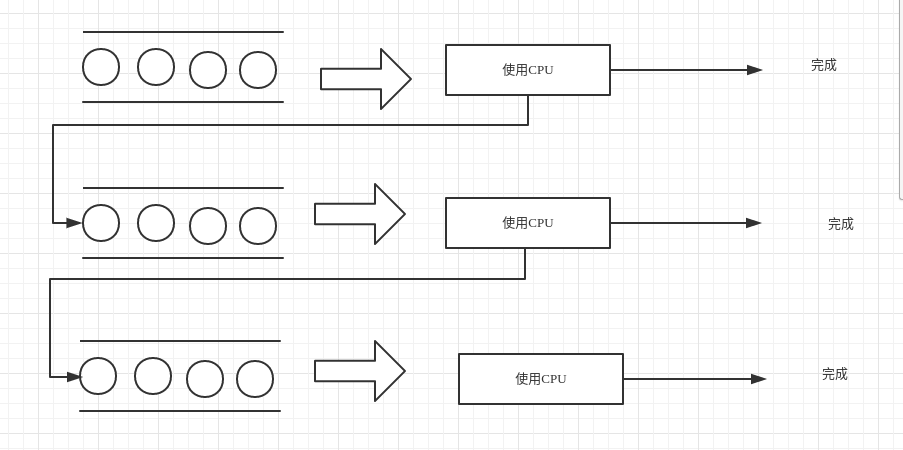
一条机器指令时必须完成,不可以中断。

当 Instruc\_State=0 或 1 时, Instruc\_Times=产生[10-40]之间的随机 10ms 倍数的整数;

当 Instruc\_State=2 时, Instruc\_Times=50,用户进程发生 PV 阻塞请求,并假设完成 I/O

数据通信时间为 50ms,之后可唤醒。

★PSW 中保存该进程当前执行的指令编号。例如,1 表示正在执行第 1 条指令

具体实现上，三级反馈队列调度算法

# 四、程序结构及核心模块的实现（重点）

**（说明：分模块论述。可用文字以及有注释的核心伪码来论述。不能只拷贝代码，必须用文字论述过程、关键变量以及处理操作过程）**

本实验初步实现了一个小kernel,组织结构树如下：

RiOS0.01

├── bootloader.S

├── kernel

│   ├── console

│   │   ├── console.c

│   │   └── console.h

│   ├── include

│   │   ├── asm

│   │   │   ├── 8259\_8253.h

│   │   │   └── x86\_gas.h

│   │   └── rios

│   │   ├── main.h

│   │   └── xwindow.h

│   ├── init

│   │   ├── 8259\_8253.c

│   │   ├── gdt\_idt.c

│   │   ├── gdt\_idt.h

│   │   ├── initialize.c

│   │   ├── initialize.h

│   │   └── interrupt

│   │   ├── interrupt.c

│   │   └── interrupt.h

│   ├── kernel.ld

│   ├── lib

│   │   └── character

│   │   ├── getcharTXT.sh

│   │   ├── RiAlphaNum.txt

│   │   └── RiASCII.txt

│   ├── main.c

│   ├── main\_gas.S

│   ├── makefile

│   ├── mm

│   │   ├── memory.c

│   │   └── memory.h

│   ├── RiOShead.S

│   ├── task

│   │   ├── queue

│   │   │   ├── queue.c

│   │   │   └── queue.h

│   │   ├── task.c

│   │   └── task.h

│   └── xwindow

│   └── xwindow.c

├── makefile

└── RiOS.img

开机后将首先将bootloader.S部分加载到0x7c00处，该部分在获得引导盘基本信息后显示提示信息，跳转到0xc400处去执行RiOShead.S编译后的代码。

RiOShead.S主要的工作有：

1. 调用BiOS功能，进入80\*25的字符文本显示模式。
2. 通过向键盘控制器相应端口写入数据打开A20地址线
3. 令控制寄存器CR0的最低位（bit0）PE位为1，进入32位保护模式;令控制寄存器CR0的最高位（bit31）PG（控制分页管理机制）为0，这个地方非常非常非常重要，由于我禁用了分页机制，进入的是平坦内存模式（flat memory mode），此时线性地址等同于物理地址，这是我接下来编程的基础和前提。（由于分页机制涉及地址映射，我对分页机制尚没有弄懂，为了完成本次进程调度的实验，不分页的平坦内存模式是我目前唯一的选择，内存管理部分由此变得简单）  
    4.设置临时全局描述符表（gdt）,并在表中设置好相应段，然后通过段间长跳转，跳到C语言写的main.c中，执行其中的RiOSmain函数。系统做一些初始化工作后正常运行。

在main.c中将各个模块的内容最后都加载进来，都在RiOSmain函数中调用，接下来对各个模块进行解释：

console模块：这一模块用在系统默认进入的80×25的文本字符显示模式，（全局变量SCREEN\_X,SCREEN\_Y是当前屏幕下一个字符即将输出的位置xy坐标）cosole模块定义了一些常用的用于输出的函数，其中：

Ri\_putchar：用于在屏幕上显示一个ascii码可见字符

xy\_putchar：类似Riputchar,但可以指定绝对位置

putdec:于屏幕上打印某一个数的十进制形式，该函数非常重要，显示各种数全靠它了，类似的还要在绝对位置输出的xy\_putdec

console\_cls\_black:清屏

其他一些函数如endl、tab 、Sleep 分别对应输出一行 、制表符 、睡眠

mm模块：

这部分与我所进入的处理机模式密切相关，本来页式内存管理是非常复杂的，如果开启分页机制，将不得不考虑地址映射与页面共享等问题。但是我禁用了控制寄存器CR0的PG位，进入的是平坦内存模式，采用连续地址分配，基本上不要考虑太多。

为了管理内存，我设了一个全局结构体指针变量RiOSmm，它对应内存管理的记录表，在系统初始阶段，必须先对它进行初始化，保证之后指针正常地分配和回收内存。

task模块:

首先介绍task模块的子模块：queue模块，这里定义了队列的基本操作。

void initQueue (LinkQueue \*q);

void EnQueue(LinkQueue \*q,QElemType e);

void DeQueue(LinkQueue \*q,QElemType \*e);

void pushpQueue(LinkQueue \*q,QueuePtr ptsk);

/\*pushpQueue专为task\_struct所打造，它不是push进一个元素，

\*而是push进一个已经malloc搞好的一坨节点，(不仅仅包括元素)

\*由于不需要再分配空间，因此函数也变得简单了

\*/

void popQueue(LinkQueue \*q);

/\*popQueue和DeQueue功能一样，但不返回被删的东西\*/

QueuePtr getPpopQueue(LinkQueue \*q);

/\*专业回收不要的指针

\*专为task\_struct打造，pop出来的那个指针不是free掉，而是废物利用\*/

void DetroyQueue(LinkQueue \*q);

void Input(LinkQueue \*q);

void PrintQueue(LinkQueue \*q);

void GetTop(LinkQueue \*q,QElemType \*e);

QueuePtr getpTop(LinkQueue \*q);

int QueueLen(LinkQueue \*q);

/\*task\_struct 相关\*/

void print\_taskQueue(LinkQueue \*q);

void print\_One\_task(task\_struct tsk);

task模块的其他部分：

在Linux0.11中，Linus搞了一个空任务，开始就从空任务开始，然后利用一套内联汇编switch\_to完成任务切换，空任务还能完成一些初始化工作。我的进程调度虽然和他完全不同，但也仿制一个全局空任务NULLTASK,它可以用来初始化全局当前任务指针，是十分有用的。

这部分其中一些核心函数

void Create\_task(task\_struct \*mtsk);

/\*在本系统中，先在mkPCB中将task\_struct

\*空间分配好，值都设好，返回给Create\_task一个指针

\*/

void Destroy\_current\_task();

/\*进程撤销原语\*/

void run\_task(task\_struct mtsk);

void sleep\_on(task\_struct \*mtsk\_struct);

void wakeup(task\_struct \*mtsk\_struct);

void schedule();/\*进程调度，将在时钟中断中调用\*/

void MLFQ();/\*Multi-Level Feedback Queue\*/

/\*多级反馈队列调度算法,在本系统中，多级反馈综合了时间片轮转

\*和先来先服务算法,上层用时间片轮转，下层用先来先服务

\*/

void RR\_L1();

void RR\_L2();

void RR\_L1L2();/\*Round Robin\*/

/\*时间片轮转算法,在本系统中，第一级第二级队列用时间片轮转\*/

void FCFS\_L3();/\*First Come First Served\*/

/\*先来先服务算法,在本系统中，最后一级第三级用先来先服务算法\*/

我的任务调度由schedule函数负责。在时钟中断中我调用do\_timer函数，先关中断，再进行任务调度，再开中断。

schedule函数调用MLFQ即多级反馈队列调度

MLFQ函数功能：

1若第一第二级等待队列不全为空，那么对第一第二级队列进行时间片轮转调度，即调用RR\_L1L2函数

2判断是否当前任务在运行并且还将运行，若真，则不调度，直接return,否则若q3不空，取q3队头元素运行。 即调用FCFS\_L3函数

RR\_L1函数功能（对第一级）：

1若当前无任务运行，取q1队头等待任务运行

2若当前有任务运行，但此时它的运行时间已经超过所需时间片，销毁该任务。

3若当前所运行任务优先级为1并且到了它的时间片，判断第二层任务队列是否达到上限，若不满，将当前任务优先级改为2,加入第二级等待队列;否则当前任务修改优先级后加入第三级队列。

RR\_L2函数功能（对第二级）：

1若当前无任务，取q2队头等待任务运行

2若当前有任务运行，但此时它的运行时间已经超过所需时间片，销毁该任务。

3若当前任务优先级为2并且到了它相应的时间片，若尚未完成，将其优先级改为3放入第三级队列等待。

FCFS\_L3（最后一级的先来先服务调度）：

1若当前无任务，且q3不空，取q3队头等待任务运行  
2若当前有任务运行：但尚未运行完，直接return不调度;否则销毁该任务。若三个队列都为空则return.

还有其他模块，限于篇幅不再详述。

原语的伪代码设计：

创建原语：create

create原语定义:

为进程分配唯一标识号id

为进程分配空间

初始化PCB

建立链接，新进程插入到链表

阻塞原语：block

block原语定义:

进程将自身阻塞

加入到阻塞队列

状态变为就绪状态

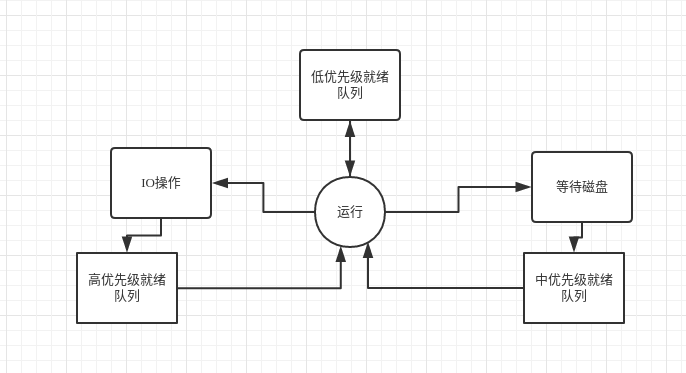
唤醒原语：wakeup

撤销原语 :destroy

Destroy原语定义

一个进程完成工作后

收回他的工作区和进程控制块



三级反馈队列算法示意图

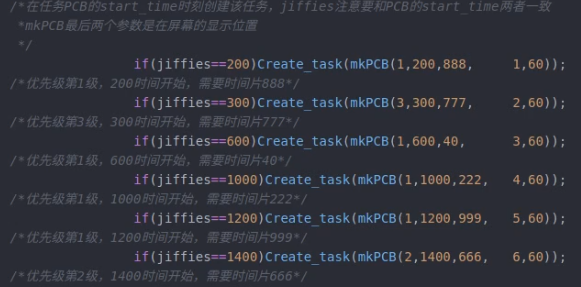
**1**、进程在进入待调度的队列等待时，首先进入[优先级](https://baike.baidu.com/item/%E4%BC%98%E5%85%88%E7%BA%A7/5643121" \t "/tmp/wps-curie/x/_blank)最高的Q1等待。

2、首先调度优先级高的队列中的进程。若高优先级中队列中已没有调度的进程，则调度次[优先级队列](https://baike.baidu.com/item/%E4%BC%98%E5%85%88%E7%BA%A7%E9%98%9F%E5%88%97" \t "/tmp/wps-curie/x/_blank)中的进程。例如：Q1,Q2,Q3三个队列，只有在Q1中没有进程等待时才去调度Q2，同理，只有Q1,Q2都为空时才会去调度Q3。

3、对于同一个队列中的各个进程，按照[时间片轮转](https://baike.baidu.com/item/%E6%97%B6%E9%97%B4%E7%89%87%E8%BD%AE%E8%BD%AC" \t "/tmp/wps-curie/x/_blank)法调度。比如Q1队列的[时间片](https://baike.baidu.com/item/%E6%97%B6%E9%97%B4%E7%89%87/6525414" \t "/tmp/wps-curie/x/_blank)为N，那么Q1中的作业在经历了N个时间片后若还没有完成，则进入Q2队列等待，若Q2的时间片用完后作业还不能完成，一直进入下一级队列，直至完成。

4、在低[优先级](https://baike.baidu.com/item/%E4%BC%98%E5%85%88%E7%BA%A7/5643121" \t "/tmp/wps-curie/x/_blank)的队列中的进程在运行时，又有新到达的作业，那么在运行完这个时间片后，CPU马上分配给新到达的作业。

# 五、测试用例及运行结果分析（重点）



作为一个简易操作系统内核，由于比较底层，算法的可呈现性是比较差的，就算是调度算法完美的Linux0.11（1991）内核也无法生动形象地表现它是如何调度的。

不过，为了本次实验演示的需要，我精心设计了一个方法，让进程调度的演示看起来非常直观。进程必须是动态创建的，我在时钟中断函数中调用一个自定义的IntoReady函数，这个函数的任务就是在系统动态运行的时候，按照实现设好的时间，在相应的时刻将任务PCB插入相应队列，这样就动态创建了进程。

与绝大多数同学不同，我这个演示完全是动态的，不需要人为干预，几个任务的“秒表”不停刷新;本系统是真正运行于裸机之上的，与控制台程序有着根本的区别，在Linux上用qemu虚拟机来展示只是为了录屏方便和观察方便，实际上我自己在用U盘直接进我的RiOS系统时观察到的与在Linux系统中qemu虚拟机窗口中观察到的是一样的，但是速度快了很多，由于真正做到每20ms发生一次硬中断，屏幕刷新的速度人眼已经不太跟的上，“闪瞎了”。

/\*优先级第1级，200时间开始，需要时间片888\*/

if(jiffies==300)Create\_task(mkPCB(3,300,777, 2,60));

/\*优先级第3级，300时间开始，需要时间片777\*/

if(jiffies==600)Create\_task(mkPCB(1,600,40, 3,60));

/\*优先级第1级，600时间开始，需要时间片40\*/

if(jiffies==1000)Create\_task(mkPCB(1,1000,222, 4,60));

/\*优先级第1级，1000时间开始，需要时间片222\*/

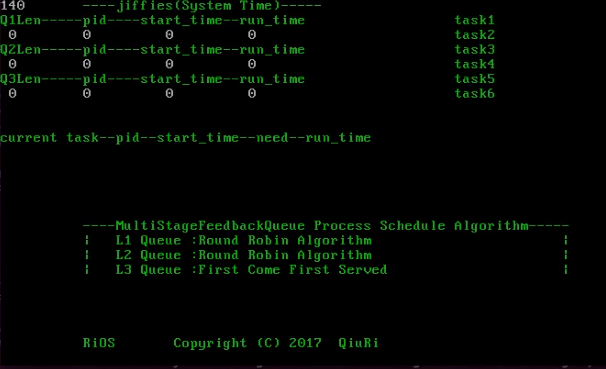
if(jiffies==1200)Create\_task(mkPCB(1,1200,999, 5,60));

/\*优先级第1级，1200时间开始，需要时间片999\*/

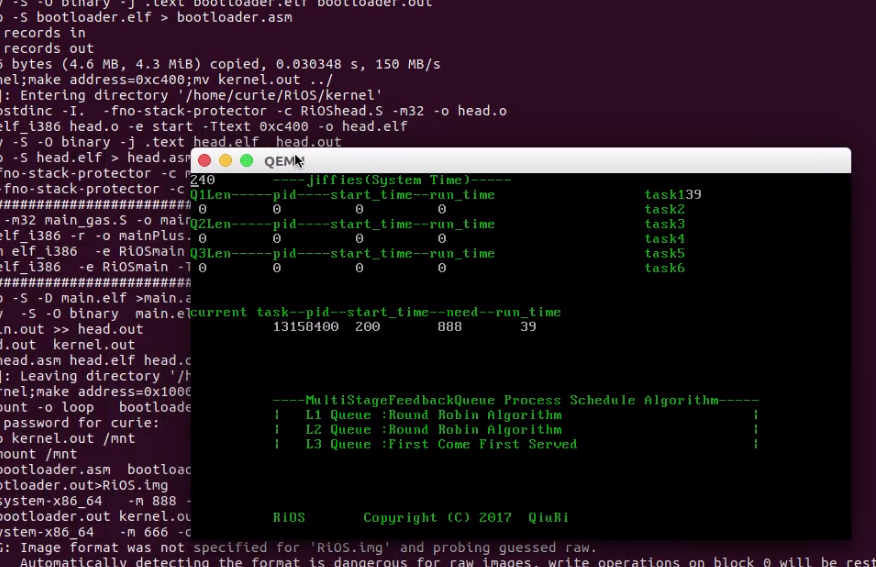
if(jiffies==1400)Create\_task(mkPCB(2,1400,666, 6,60));

/\*优先级第2级，1400时间开始，需要时间片666\*/

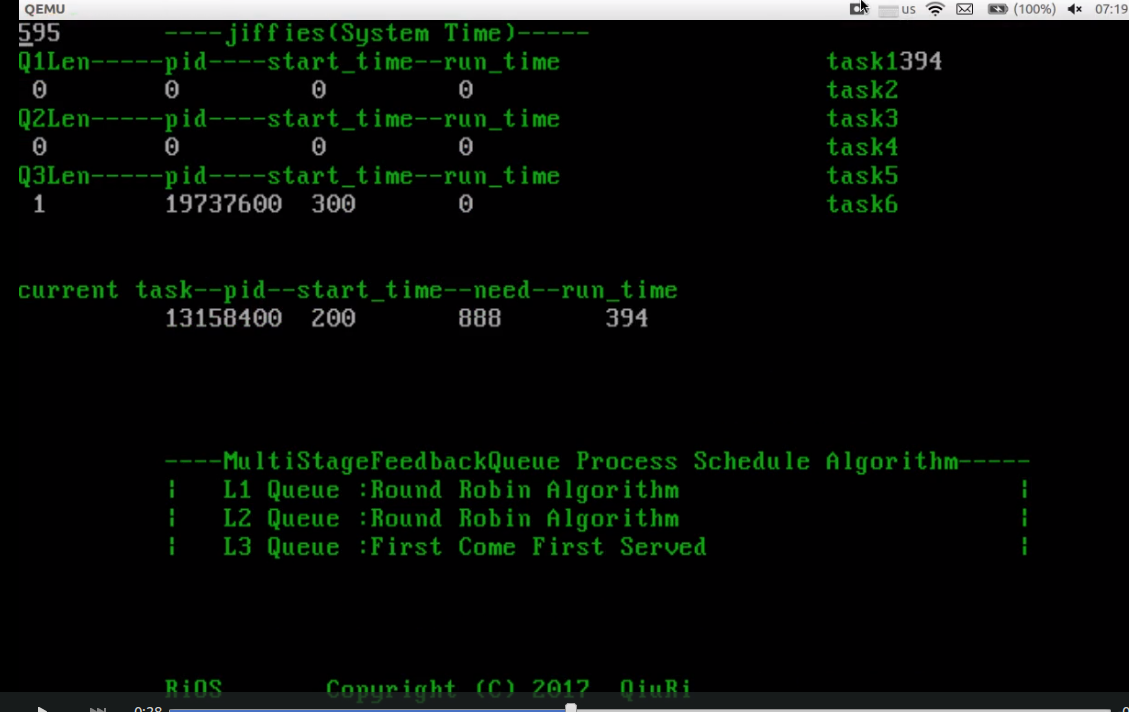
开始没有要等待的任务，所有的任务都是在系统运行过程中添加的，故此时三级队列都是空的。



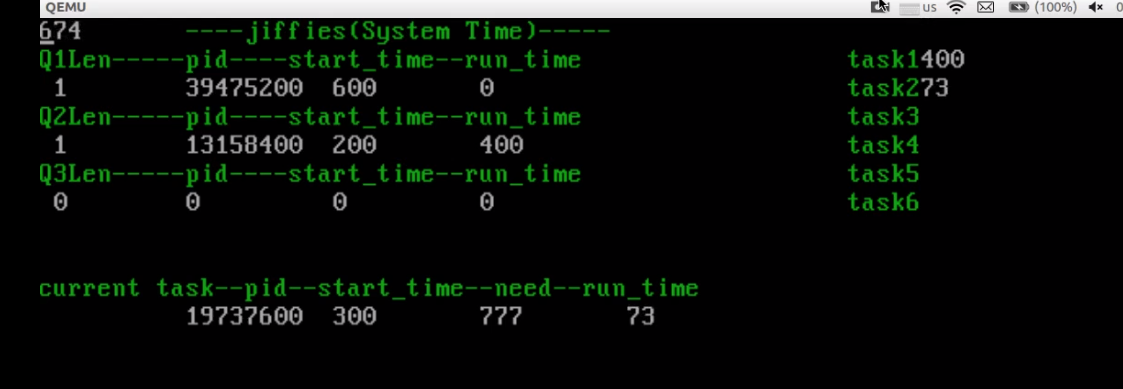
之后新来了一个第一级任务，由于之前没有任务，这个任务直接从第一级等待队列队头中取下，运行。故此时运行者第一个第一级任务，等待队列是空的。

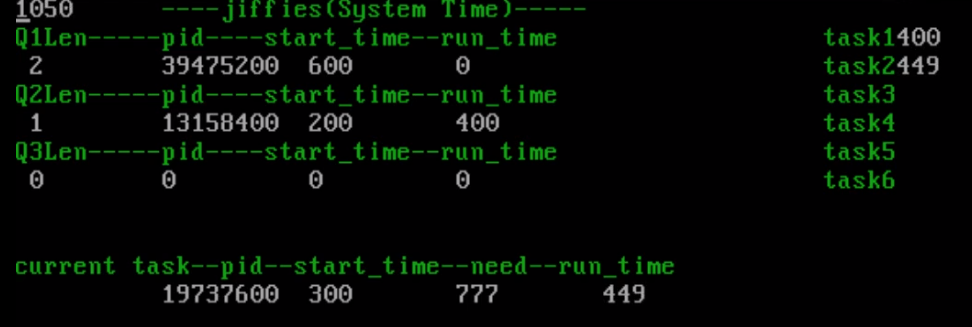


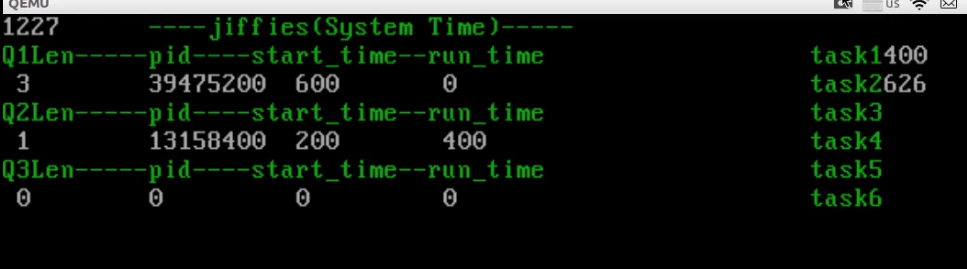
运行着第一级的任务，在第一级时间片用完，从第一级队列下放到第二级，但此时没有其他任务，故系统立即又从第二级拾起它继续运行，此时第一第二级均为空，之后第三级有任务到来，但task1作为第二级时间片尚未用完，继续运行。



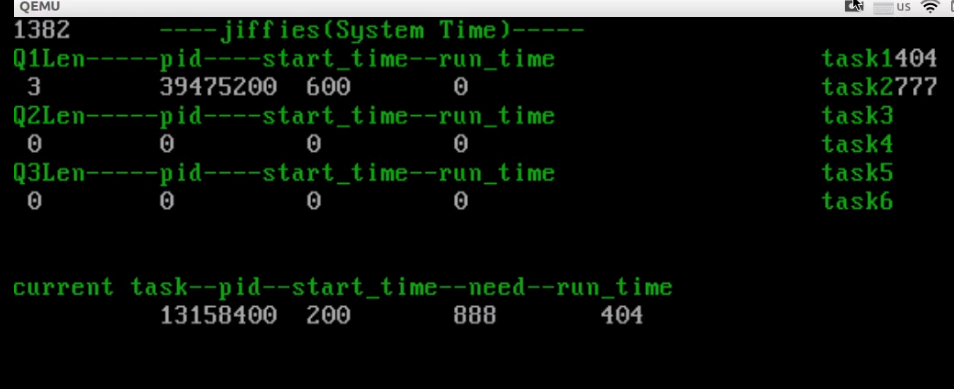
等到task1时间片用完，调入第三级的任务运行，之后有新任务创建并加入到第一级队列

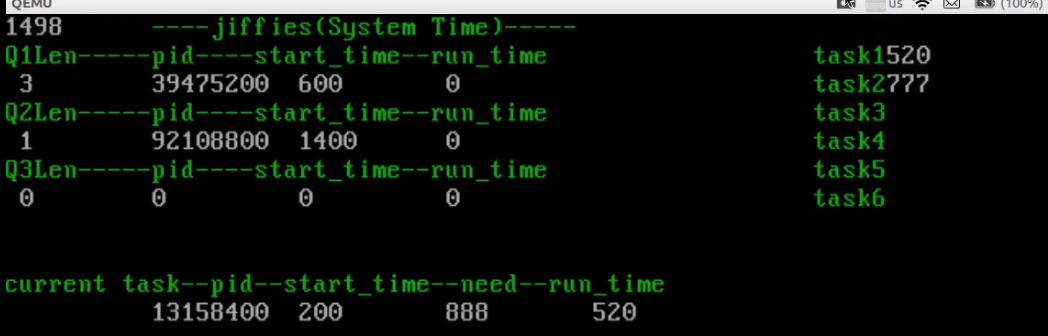


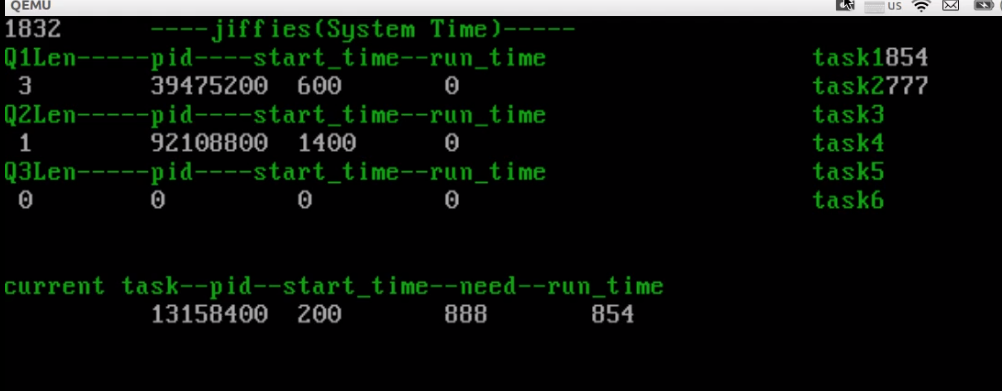




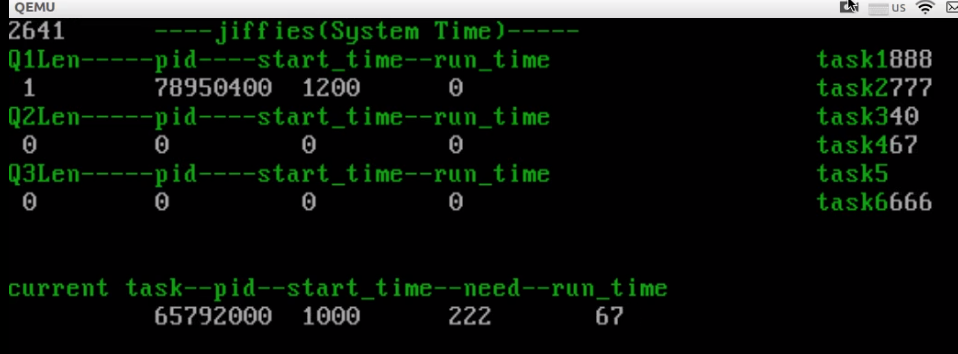
此时第一级队列共有三个任务，当前正在运行的是task1(之前已经掉入第三级队列，这次它将一直运行直到运行完)



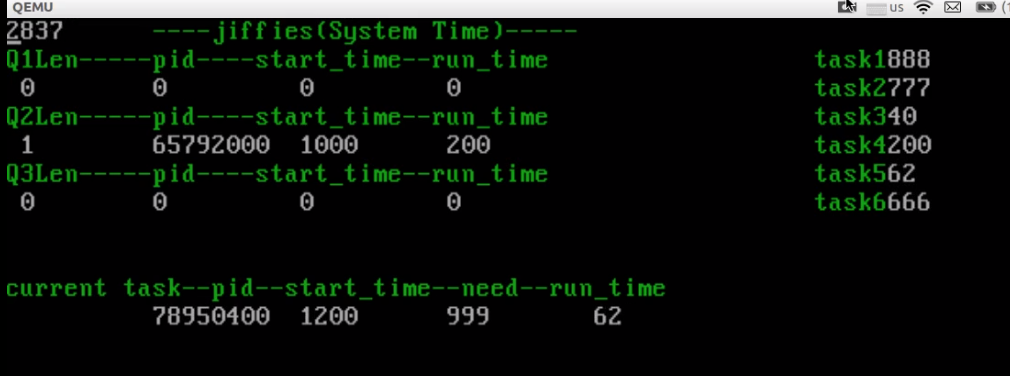


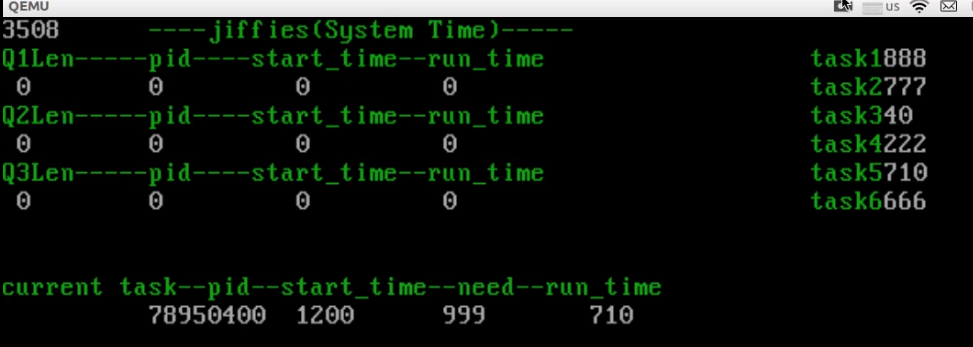


Task3是很快的，它需要的运行时间只要40,故时间片没用完它就结束了。

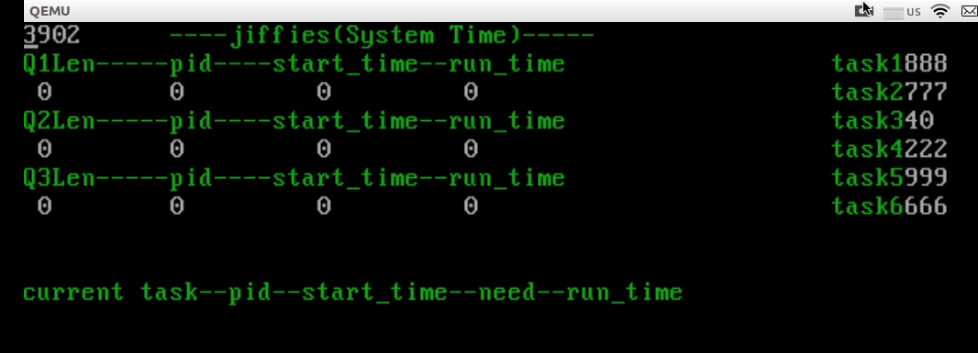


目前在运行的是task5,第二级有一个任务在等待。

  
从第三级取得任务运行，这样三级队列都为空了，只等这个任务运行完



等到这个任务运行完，当前没有任务当前任务指针指向NULL,可以看到系统自动地将相应位置刷新。



测试完成。

实际上我自己的测试是很多的，篇幅有限不再赘述。将虚拟机结果录屏，读者可以看到每一时刻的运行状况;修改IntoReady的一些参数可以轻松创建任务，读者不难验证。

# 六、技术问题及解决方案（重点）

**（具体说明实现过程中技术难题以及如何解决，并总结归纳自己的实验心得，提出自己进一步发展设想）**

1.Makefile的编写与Linux命令

2.GNU风格的汇编语言（gas）与c语言的编译链接

3.从实模式到保护模式（重点：采用键盘控制器端口打开A20地址线）

4.全局描述符表的设定（global descriptor table）

（具体说明实现过程中技术难题以及如何解决，并总结归纳自己的实验心得，提出自己进一步发展设想）

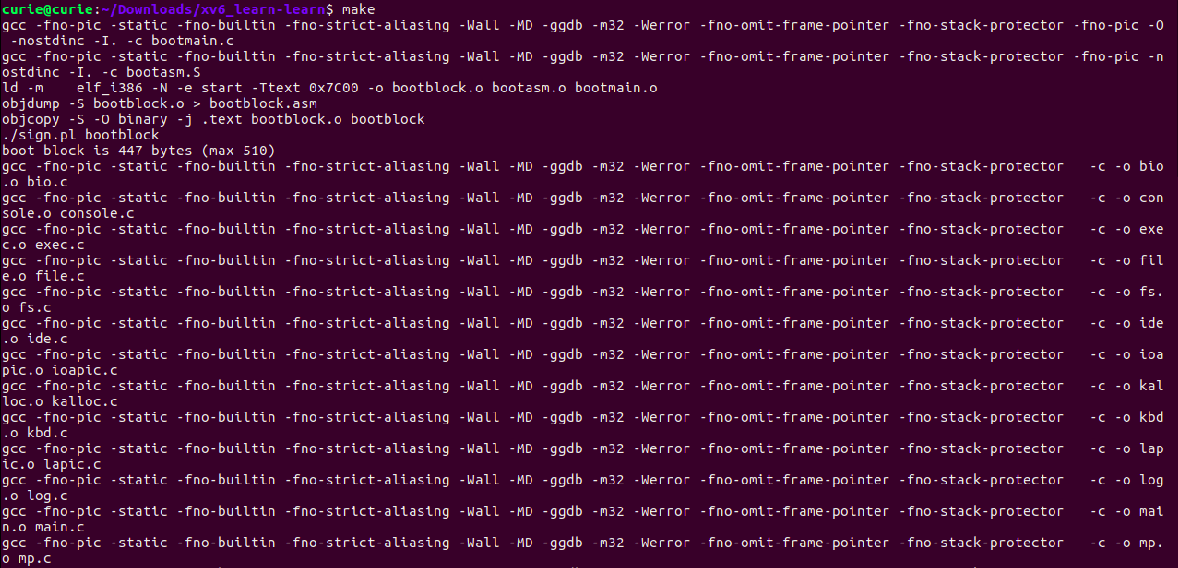
一个偶然的机会，开学时在图书馆看到一本名为《30天自制操作系统》的书，与其他只讲原理的书不一样，该书详细介绍如何从0开始在一张软盘上实现有图形界面的多任务操作系统。该书面向普通读者，内容通俗易懂。我虽然大概知道实现一个玩具操作系统是非常困难的，如果我操作系统课程设计是这种，基本上不可能完成。但我想，如果我每次平时实验都在裸机上完成，学到相应章节实现相应的功能，那么到学期末的时候，我将已经有一些零散的部件了，到做操作系统课程设计时，我再将这些代码组装整合，那么工作量化整为零，或许到最后真的能实现一个“五脏俱全”的小OS。为了实现这个大胆的想法，我从开学就每天阅读此书，这书写于2000年，年代较早，当时的软盘现在早已难觅踪迹，为了尝试在裸机上运行，我使用Windows上的软碟通Ultraso将从书后源码编译得到的镜像写入U盘，但是却启动失败了，电脑一直重复闪出同样的带有ultraso官网地址的提示信息，如图示

![](assets/README-4a267.png)源码里当然没有这些东西，我意识到这个软碟通并不纯净，写入了一些“私货”;平时用Linux比较多，我尝试用Ubuntu上自带的工具Startup disk Creator烧写到U盘，就成功启动了。开发操作系统不可避免地要使用汇编，作者使用的是我们熟悉的intel风格的汇编，用nasm编译器编译，然而作者为了用gcc又自己写了一个工具将nasm编译后的目标文件改为gcc编译后的目标文件，这样才能与那些用gcc编译后的c语言的目标文件链接起来，诸如此类，作者写了一套这样的工具，直接给出编译后的exe文件，说用这些东西来编译就行了。在我看来，其实这些东西根本没有必要，为了在Windows上用Linux下的一些工具搞得这么麻烦，站在作者角度迎合Windows用户是有必要的，但作为读者无疑增加了学习负担。由于他使用了一些自己开发的工具，读作者在Windows平台写的makefile也令人费解，不能直观看出来是如何编译链接的。

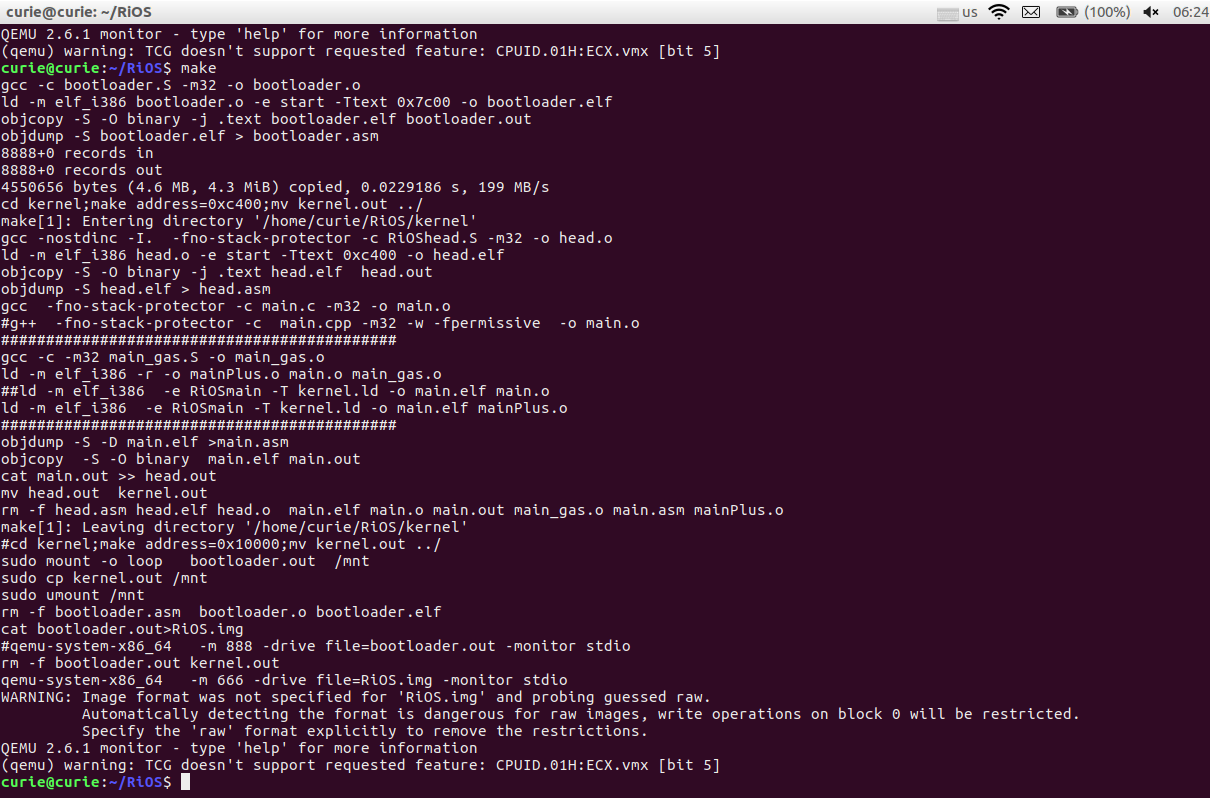
基于以上种种原因，我决定转向Linux平台，使用主流的开发工具。

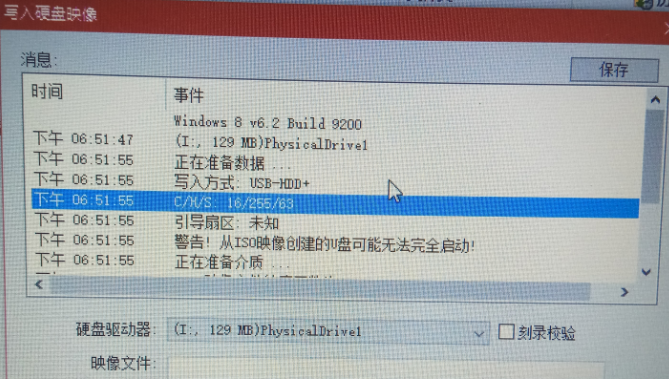
1.《30天自制操作系统》一书的代码只能在Windows用作者给出的一套他自己之前写的工具编译，要移植到Linux继续学习它，首先要将intel风格的汇编改写成AT&T式汇编，这个只是写法的差异，基本上是mov ax,bx改为mov %bx,%ax这种，即操作数位置两者是相反的，AT&T汇编在寄存器操作数前面加了%,其他还有很多细微差别，在实际操作中，从网上查文档基本上可以解决。

2.碰到的问题首先是交叉编译的问题，我的电脑是x86 64位的，但是我开发的这个kernel适用于32位保护模式，哪怕我照着源代码转换为等价的汇编，在用gcc编译的时候也会报Segmentation fault(core dumped)的错误，查阅大量资料后我才知道是交叉编译的问题。如何编译将不能在本机64位系统上运行的32位代码？如果精通Linux命令，通过配置编译参数很容易解决，但尝试了网上的一些方法都没有成功。我想到一个办法，那就是多下载几个类似的Linux下编译的玩具操作系统源码，通过观察命令行编译时的输出信息，看Makefile是如何编写和起作用的。在编译了JOS,XV6,以及一大堆乱七八糟的小操作系统和无数次尝试后，我大概了解Makefile的写法是怎样的。例如ld -m elf\_i386 hello.c -o hello.o 就可以实现编译i386环境的代码，这样就解决了x86\_64的电脑要编译i386代码的问题。后来我意识到，要写好Makefile，除了学习其语法,关键还是要精通Linux命令，对它了解太少将寸步难行，于是有空就拿起本学期选修的Unix课的教科书《UNIX和Linux

大学教程》看看。图为编译xv6时的提示信息

我在深入研究了很多小操作系统的编译过程后，学会了makefile的写法，并对内核代码程序的编译链接产生了更加深刻的理解，针对自己的代码写成了makefile来编译自己的RiOS小操作系统内核，下图为我自己RiOS的编译提示信息，里面的每一步都是我事先写好的Linux命令。看得出来，这是极为复杂的。编写makefile的时候，也是花了我非常大的精力，这也不是一次写成的，而是由少到多，随着代码和代码文件的增加，一点一点写成的。在初始阶段，写makefile对我的困扰甚至超过了写代码本身。



3.因为涉及硬件，我了解了一些关于扇面磁道扇区的基本知识，为了方便我做实验，我特意买了一个128MB的U盘，借助软碟通在格式化时给出的提示信息，了解到U盘C/H/S规格信息：16/255/63，这一点在我启动时将U盘内容复制到内存时用到了(硬编码，默认系统大小不超过128MB)。

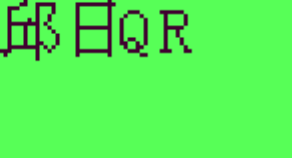
4.虽然《30天自制操作系统》一书带我看到了新世界，但这本书也存在很大问题，例如他一直在讲窗体的制作，图形的绘制，追求让人看起来好像有很大成就感，但是图形界面的制作本不是操作系统根本的一部分，一个kernel甚至根本不需要图形界面;而操作系统的最核心部分，诸如进程调度、存储器管理、文件系统这些他要么语焉不详，要么根本没有写

。这本书实现了日文的字库显示，我结合shell编程也实现了中文字库并显示，但是类似这种事做下去以后，我发现我离操作系统本身越来越远，这本书将我引入歧途。当我意识到这一点时，我赶紧修改mov $0x13,%al;xor %ah,%ah;int $0x10为mov $0x03,%al;xor %ah,%ah;int $0x10,即不再进入320\*200的video mode,

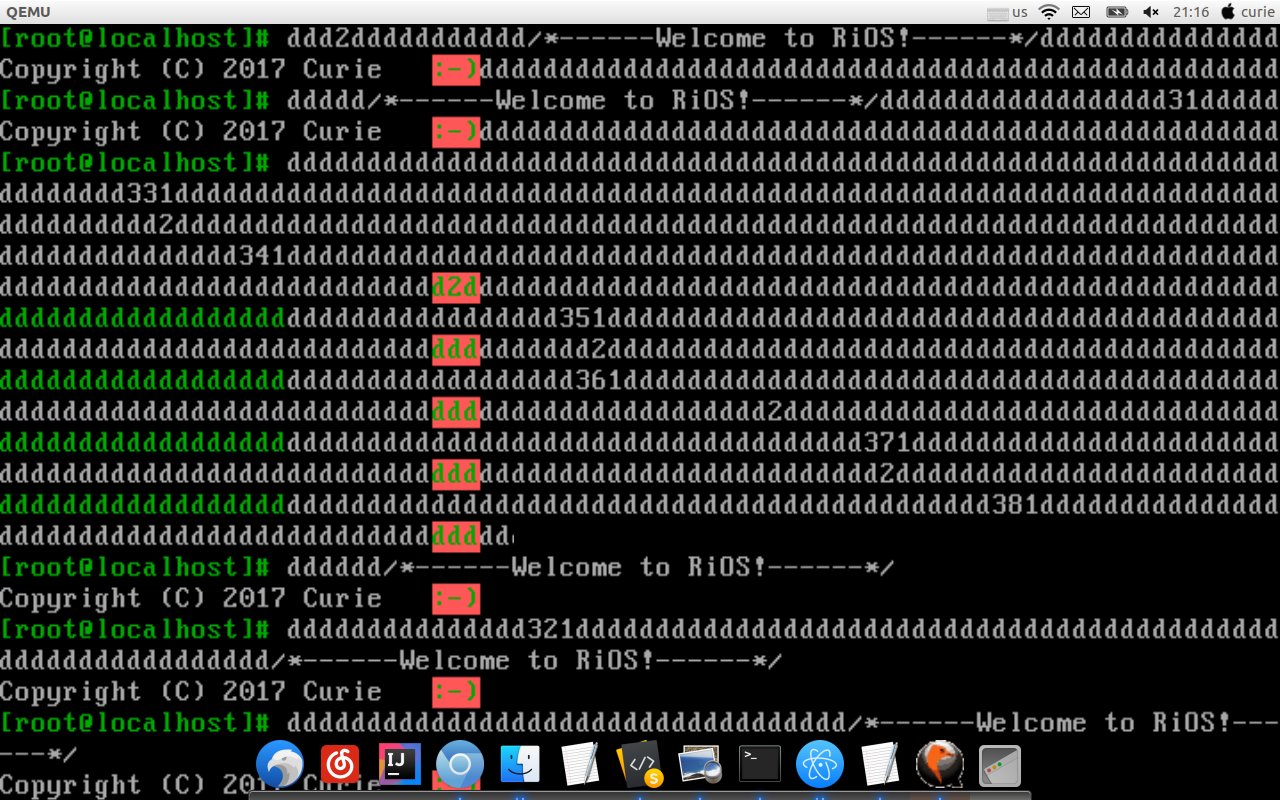
而是进80\*25的字符界面，不再研究图形界面。毕竟写一个简单的小内核，由此了解OS的基本原理才是目的。



图形界面下一开始失败的显示



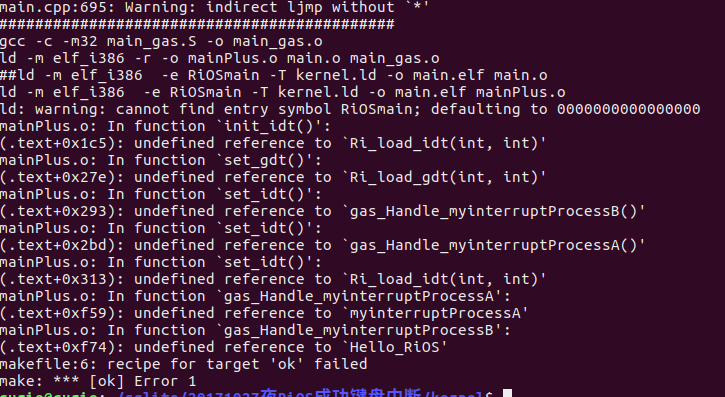
图形界面下字库的测试（成功）



转入字符界面

1. 关于这次操作系统实验，我采用的是纯的C语言，没有用任何C标准库函数，也没有使用任何C++特性的东西。也许读者可能要问：你为什么要拿C语言，为什么不用更为高级的C++来写操作系统内核，你是不是不会用C++才用C语言的？其实开始我也曾试图用C++来写，若能用上其C++特性，或能简化代码。然而实际的开发中，我越来越发现C++在写最底层东西的时候相对于C语言是极其别扭的，我也逐渐理解Linux内核为何采用C语言开发而不使用C++开发。以下我将以我在开发遇到的一个非常困难的问题为例来说明。

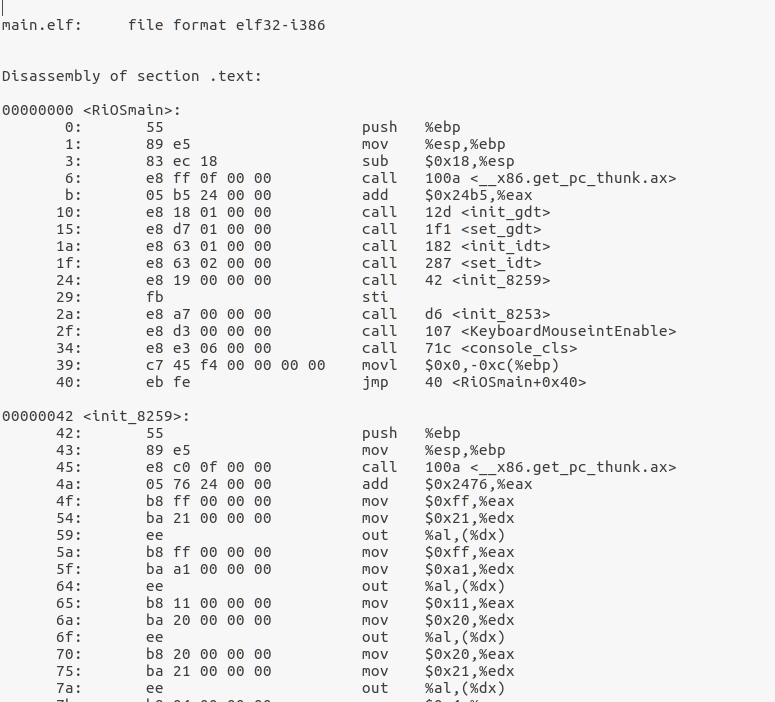
首先问题的背景是我采用C语言开发本系统的过程中，代码没有问题，编译通过可以正常运行的情况下，我将main.c改为main.cpp.当然Makefile里也作相应变化比如用g++来编译main.cpp什么的。但是，却发现一些通过标号调用汇编代码的地方全不行了，报错undefined.



一直找不到原因，我决定深入到机器码，将编译后的机器码反汇编成.asm文件，这样编译中出了什么错便能从源头上找到了。

Linux命令：

objdump -S -D main.elf >main.asm



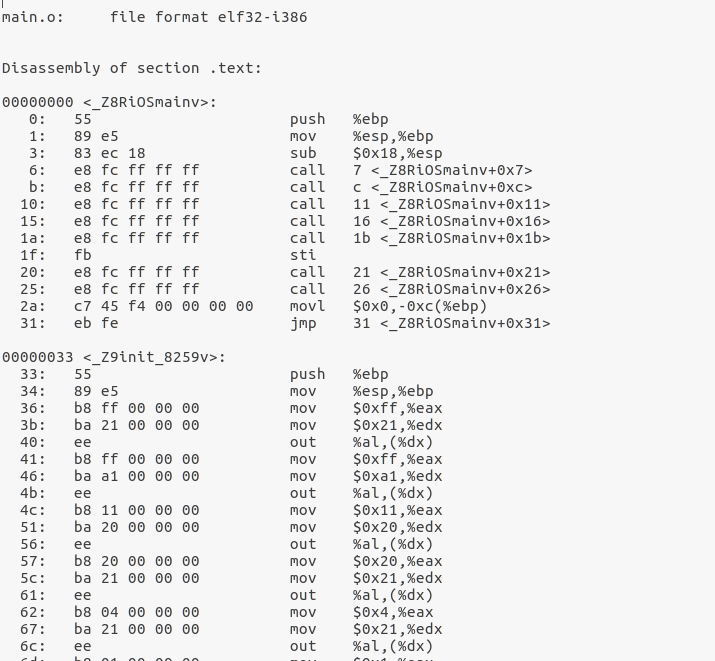
之前编译正常.c文件时的编译结果

可以看到，我的函数名RiOSmain和init\_8259在编译之后是保留原来的字的。

Linux命令：

g++ -nostdinc -I. -fpermissive -fno-stack-protector -c main.cpp -m32 -o main.o

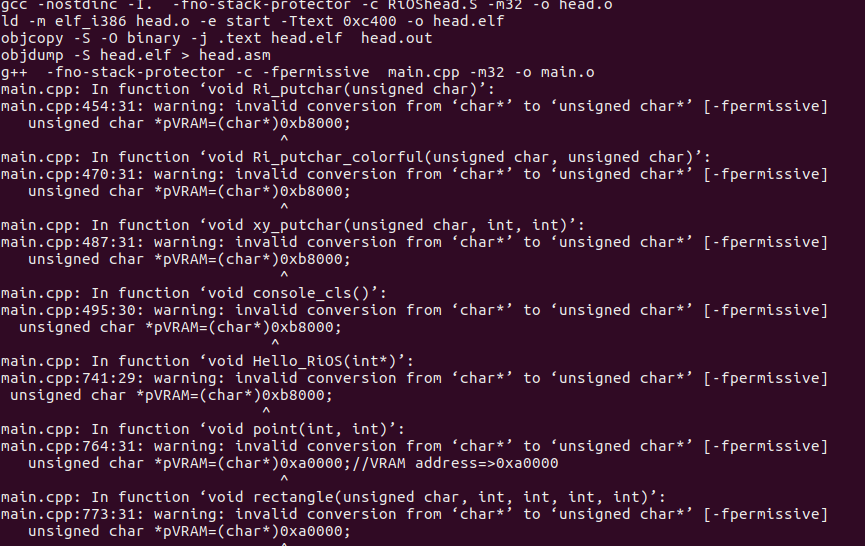
objdump -S main.o >main.asm



编译.cpp文件时的结果

此时我惊讶地发现，在gcc或g++对.cpp文件编译时，原来RiOSmain的函数名变成了\_Z数字RiOSmainv ，即gcc是这样处理的函数到标号的映射是标号：\_Z数字函数名v

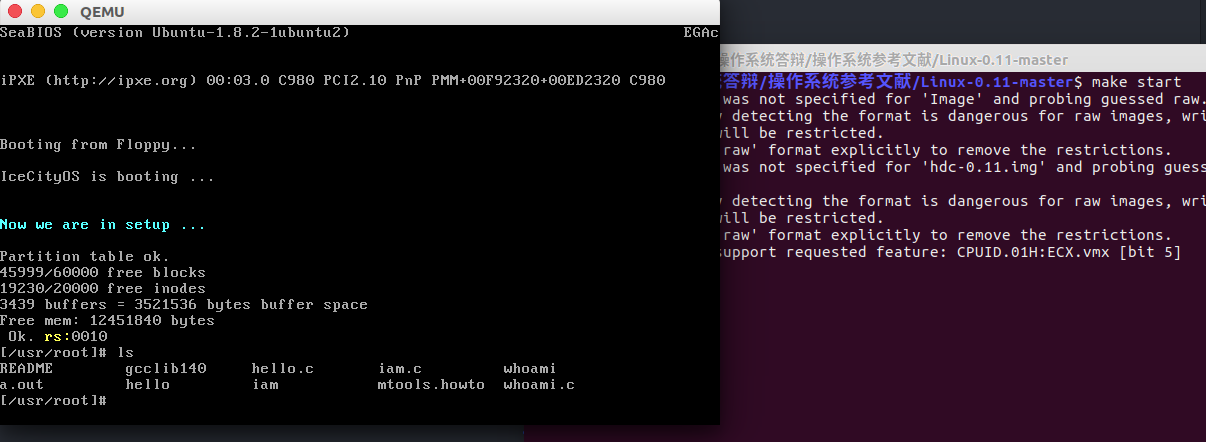
,这个编译器底层的东西我是第一次发现，实际上除了深入源码或是在实验中发现，或许也没有哪本书会讲到。这样处理C++或许会比C安全一些，但对我开发而言就很不方便，在汇编代码中将标号改为这个才能通过标号去访问到C++中的函数。另外，由于C++对指针管理可能比C要严格，一个没有error和warning的.c文件改成.cpp去编译可能就报很多错和警告。



1. 正如老师教导的，操作系统是概念先行，框架先行，其次才是算法。我的知识远远不够支撑我开发哪怕一个非常简单的内核，所以要先读书，通过写代码来验证所学这才是正道。我们所做的，不过是重复发明轮子，伟大的先行者们早已将所有基础的东西完美解决，我们要做的是学习他们的经验。Linus于1991年开发出Linux的最初版本,现在的操作系统都过于庞大而无法学习，阅读Linux早期源码是一个有效的途径，光自己看源码，不能完全看懂，借助赵炯编写的《Linux内核完全注释修正版v3.0》(其书基于Linux0.11)，加深了自己的理解。

Linux0.11由于开发时间较早，所用编译环境与现在有很大不同，后人有将它稍稍修改，到Ubuntu环境下能编译并在qemu虚拟机中运行的版本，下图即是Linux0.11版本在

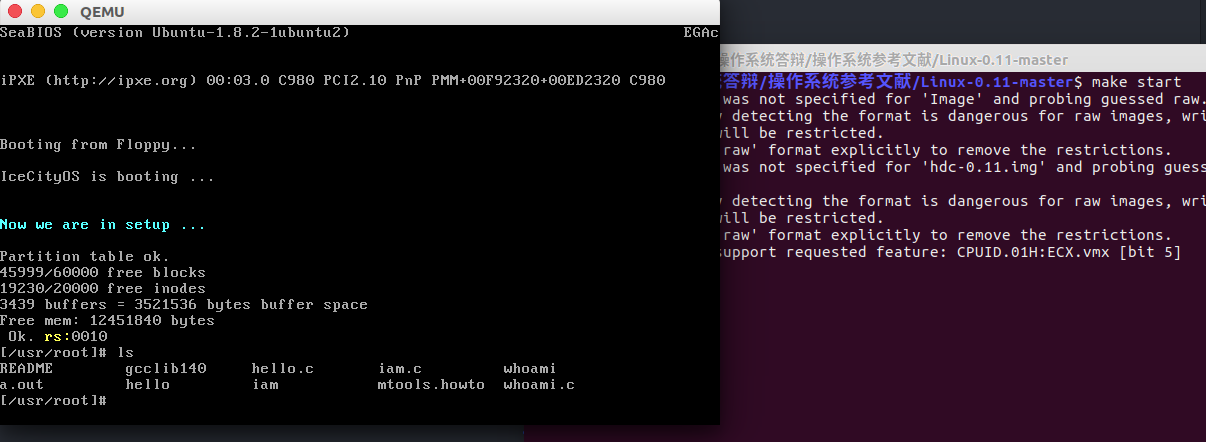
Ubuntu的运行情况。

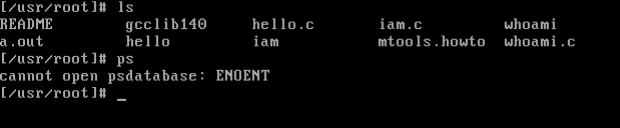


6.在阅读了Linux0.11相应源代码后，我从原理上了解了真正的任务切换的方法，但是实践上有困难。Linux0.11是这样做的，先设置好任务状态段(Task State Segment)TSS,然后使用内联汇编switch\_to一个长跳转就完成了任务切换。我也尝试这种方法，但是每当我进行长跳转时，qemu虚拟机就莫名其妙地黑屏或是所有寄存器全归零直接关闭，在真机上测试时是无限循环地重启。

7.内存管理是操作系统的重要部分，本来我是打算逐步实现操作系统的各个部件的，但是要做完进程调度算法就要malloc和free,就要做好内存管理，而页式内存管理及其复杂，我采用折中方案，处理机模式选择平坦内存模式回避这个问题，使用相对简单的连续内存管理。

8.说实在的，进程调度本身是非常底层的东西，其可呈现性比较差，也就是说比较难让人看到效果。下图为Linux0.11的运行图。





Linux0.11

在Linux中理论上可以用ps命令来查看当前进程运行情况，在Linus于1991年的Linux0.11中，即便它有完美的调度算法，但也没有什么可演示的。我为了完成此次实验，让人看得更直观，作了一点变通，这和调度算法没多大关系，也没有难度，但演示效果是比较好的。

进一步发展设想：

1.本实验可以跑在裸机上，但仅仅完成进程的简单切换还不能称之为操作系统，本学期有多次作业，每次作业我将在前几次实验的基础上加上相应的功能，将我的程序逐步扩展。这样，到我做操作系统课设的时候，我将对本学期的所有实验进行整合和完善，添加功能，使其最终成为一个“五脏俱全”的“麻雀”操作系统。

# 七、用户手册

**（包括：开发环境安装、运行条件、安装步骤、操作步骤、使用说明）**

开发环境安装

1.开发环境：Ubuntu16.10,编译器gcc

2.虚拟机模拟器:qemu-system-x86\_64(模拟64位x86架构处理器的qemu)

3.开发语言：AT&T汇编语言(使用gcc编译器) 、c语言

使用说明：

(1)方法一：从qemu虚拟机运行

1.先确保已经正确安装qemu的相应版本(qemu-system-x86\_64下测试通过)

打开终端，cd到RiOS目录，输入make，编译后在虚拟机运行

2.在上步操作中，请确认已经拔掉所有USB设备，不然会清空其上数据

(2)方法二：在裸机上运行

1.同方法一1步，于RiOS目录下得到制作好的RiOS镜像，插入U盘

在终端输入make write2udisk,写入U盘

2.重启电脑，调整bios启动顺序，将U盘启动作为第一启动项

3.插入U盘，重启电脑从U盘启动，进入RiOS

# 八、备注

**（说明：提交电子资料目录、源程序文件清单及其每个文件的说明，源程序需要注释）**

**源程序文件清单**

我的小kernel:RiOS0.01

**RiOS**

├── bootloader.S

├── kernel

│   ├── console

│   │   ├── console.c

│   │   └── console.h

│   ├── include

│   │   ├── asm

│   │   │   ├── 8259\_8253.h

│   │   │   └── x86\_gas.h

│   │   └── rios

│   │   ├── main.h

│   │   └── xwindow.h

│   ├── init

│   │   ├── 8259\_8253.c

│   │   ├── gdt\_idt.c

│   │   ├── gdt\_idt.h

│   │   ├── initialize.c

│   │   ├── initialize.h

│   │   └── interrupt

│   │   ├── interrupt.c

│   │   └── interrupt.h

│   ├── kernel.ld

│   ├── lib

│   │   └── character

│   │   ├── getcharTXT.sh

│   │   ├── RiAlphaNum.txt

│   │   └── RiASCII.txt

│   ├── main.c

│   ├── main\_gas.S

│   ├── makefile

│   ├── mm

│   │   ├── memory.c

│   │   └── memory.h

│   ├── RiOShead.S

│   ├── task

│   │   ├── queue

│   │   │   ├── queue.c

│   │   │   └── queue.h

│   │   ├── task.c

│   │   └── task.h

│   └── xwindow

│   └── xwindow.c

├── makefile

└── RiOS.img

13 directories, 31 files

所有代码都已经详细注释。

RiOS主要由引导操作系统的引导启动程序bootloader.S及其之后的RiOShead.S，字符设备驱动模块console、 头文件模块include、运行库模块 lib、 mm 、task 、xwindow模块组成、

# 参考文献

[1]赵炯 Linux内核完全注释修正版v3.0

[2]Andrew S. Tanenbaum (荷) 陈向群, 马洪兵等译 现代操作系统[M]著

[3]x86汇编语言从实模式到保护模式 李忠, 王晓波, 余洁著 电子工业出版社 2013.01

[4]郑阿奇 孙承龙 Linux内核精析 [M]电子工业出版社,2013.02

[5]蒲晓蓉 操作系统原理与Linux实例设计[M] 电子工业出版社,2014

[6]于渊 Orange S：一个操作系统的实现[M]电子工业出版社 2009.6.1

[7]William Stallings(美) Operating systems:internals and design principles精髓与设计原理[M]北京:电子工业出版社,2013.07

[8]川合秀实.30天自制操作系统[M].人民邮电出版社:周自恒，李黎明，曾箱江，张文旭,2012.1-710.

参考网络资源

[1]Linus早期Linux内核源码（Linux0.11）https://github.com/yuanxinyu/Linux-0.11

[2]《X86汇编语言-从实模式到保护模式》书后配套代码https://github.com/lichuang/x86-asm-book-source

[3]国外用于教学目的的类Unix操作系统xv6源码https://github.com/leenjewel/xv6\_learn

[4]中文版xv6文档https://github.com/ranxian/xv6-chinese

[5]《30天自制操作系统》配套源码https://github.com/yourtion/30dayMakeOS

[6]《ORANGE’S：一个操作系统的实现》书后源码https://github.com/wlmnzf/oranges

[7]Makefile编写手册 GNU make manual

[N/R]https://www.gnu.org/software/make/manual/make.html

# 附录

RiOS0.01全部代码（除RiAlphaNum.txt与RiASCII.txt）

RiOS代码目录树

**RiOS**

├── bootloader.S

├── kernel

│   ├── console

│   │   ├── console.c

│   │   └── console.h

│   ├── include

│   │   ├── asm

│   │   │   ├── 8259\_8253.h

│   │   │   └── x86\_gas.h

│   │   └── rios

│   │   ├── main.h

│   │   └── xwindow.h

│   ├── init

│   │   ├── 8259\_8253.c

│   │   ├── gdt\_idt.c

│   │   ├── gdt\_idt.h

│   │   ├── initialize.c

│   │   ├── initialize.h

│   │   └── interrupt

│   │   ├── interrupt.c

│   │   └── interrupt.h

│   ├── kernel.ld

│   ├── lib

│   │   └── character

│   │   ├── getcharTXT.sh

│   │   ├── RiAlphaNum.txt

│   │   └── RiASCII.txt

│   ├── main.c

│   ├── main\_gas.S

│   ├── makefile

│   ├── mm

│   │   ├── memory.c

│   │   └── memory.h

│   ├── RiOShead.S

│   ├── task

│   │   ├── queue

│   │   │   ├── queue.c

│   │   │   └── queue.h

│   │   ├── task.c

│   │   └── task.h

│   └── xwindow

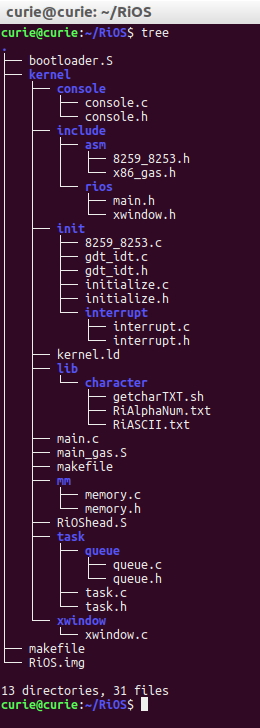
│   └── xwindow.c

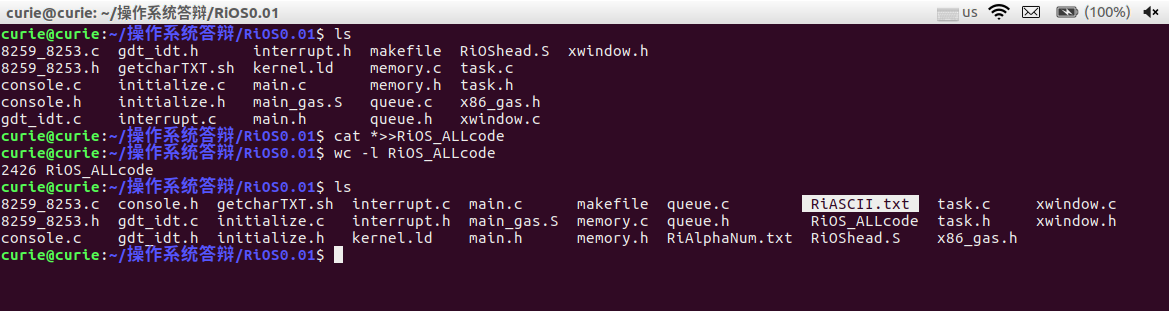
├── makefile

└── RiOS.img

13 directories, 31 files

所有代码都已经详细注释。





我的RiOS代码目录是有层次结构的，这从目录树上可以很清楚地看出来，为了保证RiOS能够正确编译链接，这个目录树不能动，因为Makefile中的编译链接命令都与此相关。但是为了我们人阅读代码查看方便，我就将这些代码都先直接放在同一目录下，然后利用简单的cat命令全部导出到同一个文件RiOS\_ALLcode，（即本系统所有代码，但不包括两个txt文件，那两个txt文件实际上是我用shell编程的方法生成的ASCII字库的C数组的大段声明）可以看到，RiOS\_ALLcode一共有2426行，以下为RiOS\_ALLcode全部内容。

/\*

\* RiOS/init/8259\_8253.c

\*

\* Copyright (C) 2017 Curie （邱日）

\*/

/\*---Programmable Interrupt Controller definition------\*/

void init\_8259()

{

outb(PIC0\_IMR, 0xff);

outb(PIC1\_IMR, 0xff);//向port 0x21 ,0xa1写0xff　　禁止所有的irq中断

//对master pic进行设置

outb(PIC0\_ICW1, 0x11);//设置产生中断的方式为边沿触发　　　 0x20

outb(PIC0\_ICW2, 0x20);//设置中断号的基地址，原来是irq 0x0到0x7,现在基地址是0x20，所以中断号变为int20到int27 0x21

outb(PIC0\_ICW3, 1<<2);//对master pic的bit2进行写１操作，设置slave pic由irq2连接 0x21

outb(PIC0\_ICW4, 0x01);// no buffer mode 0x21

outb(PIC1\_ICW1, 0x11);//edge trigger mode

outb(PIC1\_ICW2, 0x28);//设置中断号的基地址，中断号变为int28 int0x2f 接收

outb(PIC1\_ICW3, 2 );//pic0 bit2

outb(PIC1\_ICW4, 0x01);//no buffer mode

outb(PIC0\_IMR, 0xfb);//1111 1011 打开pic1的中断请求位 port 0x21

outb(PIC1\_IMR, 0xff);//禁止pic１芯片的所有请求　　port 0xa1

return;

}//~

//赵炯《Linux内核完全注释修正版v3.0》

/\*

\* #define \_8253\_Way3bin 0x36 //方式3：方波发生器,二进制计数

\* #define \_8253\_WritePort 0x43// 8253控制字寄存器写端口

//在8086最小系统中，1片8253的通道0、1、2端口地址为1240H、1242H、1244H

\* #define \_8253\_Channel\_0 0x40//8253通道0端口

\*/

void init\_8253()

{

//锁存器latch 赋值11930

#define LATCH 11930 //interrupt request every 10ms

/\*Latch为计数器的初值，1193180/100=11930,频率100HZ,即莓10ms一次中断请求

\*lATCH的值较大11930=0b 0010 1110 1001 1010,要分两次写入端口,每次8位

\*先写低位，再写高位

\*/

/\*

\*outb(\_8253\_WritePort, \_8253\_Way3bin);

\*outb(\_8253\_Channel\_0,LATCH&0xff);//低8位，按位与&符号不能写错

\*按位与：a&b是把a和b都转换成二进制数然后再进行与的运算；

\*逻辑与：a&&b就是当且仅当两个操作数均为 true时，其结果才为 true；只要有一个为零，a&&b就为零。

\*a&b 9&8=1001 & 1000=0b1000,但9&&8=1

\*NOTE!!WARNNING!!!上次我把按位与&错写成逻辑与&&,找了一天bug才找出来

\*outb(\_8253\_Channel\_0,LATCH>>8);//高8位

\*/

outb(PIT\_CTRL, 0x34);

outb(PIT\_CNT0, 0x9c);

outb(PIT\_CNT0, 0x2e);

return;

}//~

void KeyboardMouseintEnable()

{

outb(PIC0\_IMR, 0xf9);//1111 1001 irq 1 2打开 因为keyboard是irq 1

//0xf8是连同PIT也允许，我现在0xf9尚未允许8253的时钟中断，要写0xf8允许8253

outb(PIC1\_IMR, 0xef);//1110 1111 irq 12打开　mouse是irq 12 所以要把pic 1 pic 2的芯片中断响应位打开。

return;

}//~

void \_8253timer\_interrupt\_enable()

{

outb(PIC0\_IMR, 0xf8);

return;

}

/\*

\* RiOS/include/asm/8259\_8253.h

\*

\* Copyright (C) 2017 Curie （邱日）

\*/

void init\_8259();

void KeyboardMouseintEnable();

void init\_8253();

void \_8253timer\_interrupt\_enable();

/\*---init 8259 init 8253 END------------\*/

/\*---Programmable Interrupt Controller------\*/

/\*outb from Linux kernel include/asm/io.h

\*#define outb(value,port) \

\*\_\_asm\_\_ ("outb %%al,%%dx"::"a" (value),"d" (port))

\*WARNNING!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

\*这里要特别注意，GNU format Assembly 和 MASM的操作数位置正好相反

\*linux 里的gas是 outb value,port 即 outb %al,%dx

\*而MASM的汇编语法是 out port,value,即 out dx,al

\*我因此困在这个地方，迟迟不能实现中断。(我把Linux outb 操作数顺序换一换)

\*/

#define outb(port,value) \

\_\_asm\_\_ ("outb %%al,%%dx"::"a" (value),"d" (port))

#define inb(port) ({ \

unsigned char \_v; \

\_\_asm\_\_ volatile ("inb %%dx,%%al":"=a" (\_v):"d" (port)); \

\_v; \

})

/\*---8259 Port address-------------\*/

#define PIC0\_OCW2 0x0020

#define PIC0\_IMR 0x0021

#define PIC0\_ICW1 0x0020

#define PIC0\_ICW2 0x0021

#define PIC0\_ICW3 0x0021

#define PIC0\_ICW4 0x0021

#define PIC1\_OCW2 0x00a0

#define PIC1\_IMR 0x00a1

#define PIC1\_ICW1 0x00a0

#define PIC1\_ICW2 0x00a1

#define PIC1\_ICW3 0x00a1

#define PIC1\_ICW4 0x00a1

/\*---8259 Port address END------------\*/

/\*----8253 timer Port address-------------\*/

#define PIT\_CTRL 0x0043

#define PIT\_CNT0 0x0040

/\*----8253 timer Port address END----------\*/

/\*---init 8259 init 8253-------------\*/

/\*NOTE!!:after the initialization of 8259,8253,

\*We should enable interrupt,otherwise it wont't work.

\*/

/\*---------gas--------------------------------\*/

/\* inline assembly and entern GNU format Assembly Code from maingas.S\*/

//void Ri\_cli();//close interrupt

//void Ri\_sti();//enable interrupt

//inline Assembly Learned from Linux0.11

#define Ri\_sti() \_\_asm\_\_ ("sti"::)//允许中断

#define Ri\_cli() \_\_asm\_\_ ("cli"::)//关中断

/\* my original cli ,sti in C languange

void Ri\_cli(){//close

#define Ri\_cli\_gas() \_\_asm\_\_ ("cli"::)

//这里是单个指令不能加ret,不然导致死循环

Ri\_cli\_gas();

}//~

void Ri\_sti(){//open

#define Ri\_sti\_gas() \_\_asm\_\_ ("sti"::)

//这里是单个指令不能加ret,不然导致死循环

Ri\_sti\_gas();

}//~

\*/

//Alternatively ,I learned this from Linux0.11

//#define Ri\_sti() \_\_asm\_\_ ("sti"::)

//#define Ri\_cli() \_\_asm\_\_ ("cli"::)

/\*查看中断向量表：

\*https://max.book118.com/html/2016/0525/43960342.shtm

\*8259A中断类型码08h时，功能是计时器

\*计时器timer对应8号中断向量,但是那是BIOS功能调用的向量号了

\*但我现在在保护模式，当然不能用BIOS中断，因此也无所谓了

\*正常把时钟中断向量设在0x20h

\*/

/\*

\* RiOS/console/console.c

\*

\* Copyright (C) 2017 Curie （邱日）

\*/

void Ri\_putchar(unsigned char ascii\_char)

{//注意，要传ascii码值，传value过来没有用，要加上48

unsigned char \*pVRAM=(char\*)0xb8000;

if(ascii\_char=='~'){//\n

SCREEN\_X=0;

SCREEN\_Y+=1;

if(SCREEN\_Y%25==0)SCREEN\_Y=0;

}

else if(ascii\_char=='\n'){//\n

SCREEN\_X=0;

SCREEN\_Y+=1;

if(SCREEN\_Y%25==0)SCREEN\_Y=0;

}

else{

pVRAM[(80\*SCREEN\_Y+SCREEN\_X)\*2]=ascii\_char;

SCREEN\_X++; //SCREEN\_X init value=0;

if(SCREEN\_X%80==0)SCREEN\_X=0,SCREEN\_Y++;

if(SCREEN\_Y%25==0)SCREEN\_Y=0;

}

return;

}//~

/\*Ri\_putchar(value2ascii(i));

\*本来应当实现printf的，但实现printf需要先实现sprintf

\*而且还要学习可变参数列表的使用。现阶段先实现一个阉割版的

\*虽然很简单，但也够用了，用putdec打印%d变量的值,Debug比较方便

\*/

void putdec(int value)

{//此函数意义重大，这意味着我可以输出变量的值了

/\*终于能打印变量的值了，之前什么都看不到，这下debug方便多了\*/

/\* putdec 把一个数值转为十进制每个字符依次输出

\*Usage:putdec(1234); 结果屏幕上依次输出1 2 3 4

\*#define value2ascii(n) (n+48)

\*/

if(value==0){Ri\_putchar('0');return;}

int rawvalue=value;

int a[32];

int l=0;

/\*

\*循环获取每一位的值，并存到数组a中。

\*其中l表示已经获取到的位数。

\*由于每次获取的是个位n%10，所以存在数组a中的是各个位数的逆序。

\*/

while(value){

a[l++] = value%10;

value/=10;

}

//反序输出每一位值。

if(rawvalue!=0){

while(--l>=0)

Ri\_putchar(value2ascii(a[l]));// printf("%d ", a[l]);

}

return;

}//~

void xy\_putchar(unsigned char ascii\_char,int row,int col)

{

unsigned char \*pVRAM=(char\*)0xb8000;

pVRAM[(80\*row+col)\*2]=ascii\_char;

return;

}//~

void Ri\_putchar\_colorful(unsigned char ascii\_char,unsigned char color)

{

unsigned char \*pVRAM=(char\*)0xb8000;

if(ascii\_char=='~'){//\n

SCREEN\_X=0;

SCREEN\_Y+=1;

if(SCREEN\_Y%25==0)SCREEN\_Y=0;

}

else if(ascii\_char=='\n'){//\n

SCREEN\_X=0;

SCREEN\_Y+=1;

if(SCREEN\_Y%25==0)SCREEN\_Y=0;

}

else{

pVRAM[(80\*SCREEN\_Y+SCREEN\_X)\*2]=ascii\_char;

pVRAM[(80\*SCREEN\_Y+SCREEN\_X)\*2+1]=color;

SCREEN\_X++; //SCREEN\_X init value=0;

if(SCREEN\_X%80==0)SCREEN\_X=0,SCREEN\_Y++;

if(SCREEN\_Y%25==0)SCREEN\_Y=0;

}

return;

}//~

void xy\_putdec(int value,int row,int col)

{//此函数意义重大，这意味着我可以输出变量的值了

/\*终于能打印变量的值了，之前什么都看不到，这下debug方便多了\*/

/\* putdec 把一个数值转为十进制每个字符依次输出

\*Usage:putdec(1234); 结果屏幕上依次输出1 2 3 4

\*#define value2ascii(n) (n+48)

\*/

if(value==0){xy\_putchar('0',row,col);return;}

int a[32];

int l=0;

/\*循环获取每一位的值，并存到数组a中。

\*其中l表示已经获取到的位数。

\*由于每次获取的是个位n%10，所以存在数组a中的是各个位数的逆序。

\*/

while(value){

a[l++] = value%10;

value/=10;

}

//反序输出每一位值。

while(--l>=0)

xy\_putchar(value2ascii(a[l]),row,col++);// printf("%d ", a[l]);

return;

}//~

void console\_cls()

{

//pVRAM[(80\*SCREEN\_Y+SCREEN\_X)\*2]

unsigned char \*pVRAM=(char\*)0xb8000;

int sc\_x,sc\_y;

for(sc\_x=0;sc\_x<80;sc\_x++){

for(sc\_y=0;sc\_y<25;sc\_y++){

pVRAM[(80\*sc\_y+sc\_x)\*2]='#';

}

}

return;

}//~

void console\_cls\_black()

{

//pVRAM[(80\*SCREEN\_Y+SCREEN\_X)\*2]

unsigned char \*pVRAM=(char\*)0xb8000;

int sc\_x,sc\_y;

for(sc\_x=0;sc\_x<80;sc\_x++){

for(sc\_y=0;sc\_y<25;sc\_y++){

pVRAM[(80\*sc\_y+sc\_x)\*2]=' ';

}

}

return;

}//~

void println(char \*str)

{

int ch,i;

for(i=0;str[i]!='\n';i++){

ch=str[i];//if(ch>=127|ch<32)ch=32;

Ri\_putchar(ch);

}

Ri\_putchar('\n');

return;

}//～

void printlnColorful(char \*str)

{

unsigned char \*pVRAM=(char\*)0xb8000;

#define Black\_greenT 0b00000010

// #ifndef \_setTcolor

// #define \_setTcolor pVRAM[(80\*SCREEN\_Y+SCREEN\_X)\*2+1]=Black\_greenT

// #endif

int ch,i;

for(i=0;str[i]!='\n';i++){

ch=str[i];//if(ch>=127|ch<32)ch=32;

pVRAM[(80\*SCREEN\_Y+SCREEN\_X)\*2+1]=Black\_greenT;

Ri\_putchar(ch);

// \_setTcolor,Ri\_putchar(ch);

}

Ri\_putchar('\n');

return;

}//～

void printnoln(char \*str)

{

int ch,i;

for(i=0;str[i]!='\n';i++){

ch=str[i];//if(ch>=127|ch<32)ch=32;

Ri\_putchar(ch);

}// Ri\_putchar('\n');

return;

}

void endl()

{

Ri\_putchar('\n');

}//~

void tab()

{

int i;

for(i=0;i<4;i++)Ri\_putchar(' ');

}//~

void Sleep(int secs)

{

int i=0,j=0;

for(;i<secs;i++)for(j=0;j<5000000;j++);

}//~

void littleSleep(int secs)

{

int i=0,j=0;

for(;i<secs;i++)for(j=0;j<100000;j++);

}//~

void \_Debug\_showcharFullScreen()

{

unsigned char ascii\_char='A';

int i;

for(i=0;i<300;i++){

Ri\_putchar(ascii\_char);

ascii\_char++;

if(ascii\_char>126)ascii\_char=65;

}

return;

}//~

void resetSCREENYX(int y,int x)

{

SCREEN\_Y=y;

SCREEN\_X=x;

}//~

void \_Debug\_showA2Z()

{//screen filled with A ~Z

int row=0,col=0;

int i=0,j=0;

unsigned char ascii\_char='A';

//Video mode :80\*25

for(j=0;j<25;j++){

for(i=0;i<79;i++)col++,xy\_putchar(ascii\_char,row,col);

ascii\_char++;row++;col=0;

}

}//~

/\*

\* RiOS/console/console.h

\*

\* Copyright (C) 2017 Curie （邱日）

\*/

/\*-----80\*25 Text MODE ------------------

\*全局变量SCREEN\_X,SCREEN\_Y定义在kernel/main.c里面

\* unsigned char SCREEN\_X=0;

\* unsigned char SCREEN\_Y=0;

\*/

#define value2ascii(n) (n+48)

/\*由数值到ascii 码值要加上48

\*$man ascii

\*NOTE!!之前我putchar时没有把value值转成ascii码，导致输不出来，这个把值加48编程ascii码值，这样才能打印

\*/

void Ri\_putchar(unsigned char ascii\_char);

void putdec(int value);//此函数意义重大，这意味着我可以输出变量的值了

/\*终于能打印变量的值了，之前什么都看不到，这下 NOTE!!Debug方便多了

\* putdec 把一个数值转为十进制每个字符依次输出

\*Usage:putdec(1234); 结果屏幕上依次输出1 2 3 4

\*eg putdec(20171028);!!!成功输出数值 2 0 1 7 1 0 2 8 ，离printf又近了一步

\*/

void xy\_putchar(unsigned char ascii\_char,int row,int col);

void Ri\_putchar\_colorful(unsigned char ascii\_char,unsigned char color);

void xy\_putdec(int value,int row,int col);

void console\_cls();

void console\_cls\_black();

void println(char \*str);

void printlnColorful(char \*str);

void printnoln(char \*str);

void resetSCREENYX(int x,int y);

void endl();

void tab();

void Sleep(int secs);

void littleSleep(int secs);

void \_Debug\_showcharFullScreen();

void \_Debug\_showA2Z();

/\*

\* RiOS/init/gdt\_idt.c

\*

\* (C) 2017 Curie （邱日）

\*/

/\*--Global Descriptor Table-&-Global Interrupt Table------\*/

void init\_gdt(void)

{

struct Ri\_GDT\* RiOS\_gdt=(struct Ri\_GDT\*)gdt\_template\_addr;

int i=0;

for(i=0;i<=8192;i++){

setRecord\_gdt(RiOS\_gdt+i,0,0,0);

}

return;

}//~

void init\_idt(void)

{

struct Ri\_IDT \*RiOS\_idt=(struct Ri\_IDT\*)idt\_template\_addr;

int i=0;

for(i=0;i<=256;i++){

setRecord\_idt(RiOS\_idt+i,0,0,0);

}

Ri\_load\_idt(idt\_length, idt\_template\_addr);

/\*for(i=0;i<256;i++){

\*setRecord\_idt(RiOS\_idt+ i,(int)gas\_Handle\_myinterruptProcessB,2<<3,0x8e);

\*/

return;

}//~

void set\_gdt(void)

{

struct Ri\_GDT \*RiOS\_gdt=(struct Ri\_GDT\*)gdt\_template\_addr;

setRecord\_gdt(RiOS\_gdt+0,0x00000000,0x00000000,0x0000);//gdt-0 NULL segment

setRecord\_gdt(RiOS\_gdt+1,0xffffffff,0x00000000,0xc092);//gdt-1 Data Segment:all memory

setRecord\_gdt(RiOS\_gdt+2,0x0000ffff,0x00000000,0xc09a);//gdt-2 RiOShead.S

//setRecord\_gdt(RiOS\_gdt+2,0x000007ff,0x00000000,0xc09a);//gdt-2 RiOShead.S

setRecord\_gdt(RiOS\_gdt+3,0x0000ffff,0x00280000,0xc09a);//gdt-3 main.c

Ri\_load\_gdt(gdt\_length,gdt\_template\_addr);

return;

//access:0x92 preserve for OS,rw\_

//access:0x9a preserve for OS,r\_x

// 0xf2 user,rw\_

// 0xfa user,r\_x

//0xffffffff=>4294967295,4G,all memory

};//~

void set\_idt(void)

{

#define InterruptEnable 0x008e

struct Ri\_IDT \*RiOS\_idt=(struct Ri\_IDT\*)idt\_template\_addr;

setRecord\_idt(RiOS\_idt+ 0x21,(int)gas\_Handle\_myinterruptProcessB,3<<3,InterruptEnable);//Code Segment,segmnet 2

setRecord\_idt(RiOS\_idt+0x27,(int)gas\_Handle\_myinterruptProcessA,3<<3,InterruptEnable);//Code Segment,segmnet 2

/\*--------set 8253 interrupt 20h-------------\*/

//8253 时钟中断中断类型20h，

setRecord\_idt(RiOS\_idt+0x20,(int)gas\_Handle\_8253timer,3<<3,InterruptEnable);

/\*---------------------\*/

Ri\_load\_idt(0x7ff,0x0026f800);

//attribu\_4b\_4nouse\_8b

//----------------------------------

//xxxx 0000 xxxx xxxx |

//0000 0000 1000 1110

//《从实模式到保护模式》P201

//P=1,DPL=00,S=0 TYPE=1110 rwx

//P:segment present

//high12 ---->high23 |

//44b ----->55b |

//----------------------------------

//0x008e=10001110,interrupt is valid

return;

};//~

void setRecord\_gdt(struct Ri\_GDT\* mygdt,int length,int base,int prog\_access)

{

//length<0fffff ,real mode

if(length>0xffff)prog\_access|=0x8000;//G bit =1,段界限以4KB为单位，是32位模式

//错误length/=0x1000;//被这句话坑死了，不停重启 //limit>>16;

//似乎是段长越界这个电脑就重启，底层不太明白

mygdt->length\_low=length&0xffff;//16bits //location:0~15bits

//0~15

mygdt->base\_low=base&0xff;//location:16~31bits

//16~31

mygdt->base\_mid=(base>>16)&0xff; //location:32~39bits(high0~high7)

//32~39

mygdt->access\_right=prog\_access&0xff;//location:40~47(high8~high15)

//access type itself(4bits) uses 40~43bits(high8~high11),eg0x9a,0x

//40~47

mygdt->length\_high=((length>>16)&0x0f)|((prog\_access>>8)&0xf0);//location:(occupy)48~55bits(high16~high23)

//length high itself:(4bits 48~51bits(high16~high19))

//48~55

mygdt->base\_high=(base>>24)&0xff;//location:56~63bits(high24~high31)

//56~63

//prog\_acess:segment access attribute 12bit

//xxxx 0000 xxxx xxxx

//GD:D=1,32mode

return;

}//~

void setRecord\_idt(struct Ri\_IDT\* myidt,int offset,int selector,int attribu\_4b\_4nouse\_8b)

{

///segment attribute

int seg\_attr12b=attribu\_4b\_4nouse\_8b;

//----------------------------------

//xxxx 0000 xxxx xxxx |

//high12 ---->high23 |

//44b ----->55b |

//----------------------------------

// short offset\_low;//0~15;

// short selector;//16~31

// char occupy8bits;//32~39

// char access\_right;//40~48

// short offset\_high;//48~56

myidt->offset\_low=offset&0xffff;

myidt->selector =selector;

myidt->occupy8bits=(seg\_attr12b>>8)&0xff;

myidt->access\_right=(char)(seg\_attr12b&0xff);

//NOTE!!这里之前把&错写成了&&，查了一天才查出来

myidt->offset\_high=(offset>>16)&0xffff;

///!!!!!!!!!!important!!!!!!!!!

//access\_right 1 Bytes

return;

}//~

/\*

\* RiOS/init/gdt\_idt.h

\*

\* Copyright (C) 2017 Curie （邱日）

\*/

extern int gas\_Handle\_myinterruptProcessA();

extern int gas\_Handle\_myinterruptProcessB();

extern int gas\_Handle\_8253timer();

/\*\*gas\_handlers END----\*/

extern void Ri\_load\_gdt(int length,int address);//set Global Descriptor Table's limit and start address

extern void Ri\_load\_idt(int length,int address);//set Interrupt Descriptor Table's limit and start address

extern void Ri\_load\_ldt(int length,int address);//set Local Descriptor Table's limit and start address

//~ in main\_gas.S

/\*--Global Descriptor Table-&-Global Interrupt Table------\*/

//sizeof(short):2 sizeof(char):1

struct Ri\_GDT

{//short 2Bytes,16bits;char 1Bytes,8bits

short length\_low; //0~15

short base\_low;//16~31

char base\_mid;//32~39

char access\_right; //40~47

char length\_high;//48~55

char base\_high; //56~63

};

struct Ri\_IDT

{//short 2Bytes,16bits;char 1Bytes,8bits

short offset\_low;//0~15;

short selector;//16~31

char occupy8bits;//32~39

char access\_right;//40~47

short offset\_high;//48~55

};

#define idt\_template\_addr 0x0026f800 //idt start address

#define idt\_length 0x000007ff //limit 2048=8\*255

//0x270000-0x26f800=2048=256\*8 =>256 interrupt

#define gdt\_template\_addr 0x00270000 //gdt start addrss

#define gdt\_length 0x0000ffff // limit 65536=8\*8192

void init\_gdt(void);

void init\_idt(void);

void set\_gdt(void);

void set\_idt(void);

void setRecord\_gdt(struct Ri\_GDT\* mygdt,int length,int base,int prog\_access);

void setRecord\_idt(struct Ri\_IDT\* myidt,int offset,int selector,int attribu\_4b\_4nouse\_8b);

/\*--Global Descriptor Table-&-Global Interrupt Table END------\*/

// #define FIRST\_TSS\_ENTRY 4

// #define FIRST\_LDT\_ENTRY (FIRST\_TSS\_ENTRY+1)

// #define \_TSS(n) ((((unsigned long) n)<<4)+(FIRST\_TSS\_ENTRY<<3))

// #define \_LDT(n) ((((unsigned long) n)<<4)+(FIRST\_LDT\_ENTRY<<3))

// #define ltr(n) \_\_asm\_\_("ltr %%ax"::"a" (\_TSS(n)))

// #define lldt(n) \_\_asm\_\_("lldt %%ax"::"a" (\_LDT(n)))

##

## RiOS/kernel/getcharTXT.sh

##

## Copyright (C) 2017 Curie （邱日）

##

## a shell script to generate RiOSASCII.txt & RiAlphaNum.txt

###reference:http://blog.csdn.net/weiwang876253631/article/details/8824858

##利用此网站上的取点阵程序，只能得到单个点阵，不能得到我想要的全部ASCII字符数组

##为此写了此脚本批量生成点阵数组（在前面加上符合C语法的char []..申明）重定向到

##txt,再在main函数中把生成的txt include进来就可以用了。

ZiKu=RiAlphaNum.txt

: > $ZiKu

echo "$ZiKu cleaned up."

MyASCII=RiASCII.txt

: > $MyASCII

echo "$MyASCII cleaned up."

#echo "char RiASCII[$alphabet\_charsize]={">>$MyASCII;

#for((i=32;i<=127;i++)) do printf \\$(printf '%03o\t' "$i"); done;printf

#ASCII32是空格

strASCII=" !\"#$%&'() +,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^\_\`abcdefghijklmnopqrstuvwxyz{|}~"

#里面有很多转义字符，搞得很麻烦，还要恢复它们原来的意思

# $man ascii

echo -e $strASCII

#32～126(共95个)是字符(32是空格

echo -e "\n\ngenerating allASCII... \n"

#-e 支持转义，默认不转义

echo -e ".\n\ngenerating allASCII(printable) from ASCII\_32(space) to ASCII\_126(~)... \n"

Ri\_ASCII\_charsize=$[(126-32+1)\*32];

echo "char RiASCII[$Ri\_ASCII\_charsize]={">>$MyASCII;

for i in `seq ${#strASCII}`

do

echo -e ${strASCII:$i-1:1};

echo -e $[($i+31)];

if [ "${strASCII:$i-1:1}" = " " ];then

echo "0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, ">>$MyASCII;

else

echo "/\*" >> $MyASCII;

./zimo a ${strASCII:$i-1:1} |head -n 1| tail -n 1 >>$MyASCII;

echo "\*/" >> $MyASCII;

./zimo a ${strASCII:$i-1:1} |head -n 2| tail -n 1 >>$MyASCII;

fi

done ;

echo "};">>$MyASCII;

echo -e "$MyASCII done.\n"

echo -e "\n\ngenerating alphabetASCII... \n"

#Rialphabet[..]+Rinumber[..] (32\*)

#char Rialphabet[832] 26\*32

#char Rinumber[288] 9\*32

alphabet\_a="a"

alphabet\_temp="a"

alphabet\_ASCIItotal=26;

alphabet\_charsize=$[32\*alphabet\_ASCIItotal];

echo "char Rialphabet[$alphabet\_charsize]={">>$ZiKu;

for alphabet\_temp in {a..z}

do

echo -n $alphabet\_temp

echo "/\*" >> $ZiKu;

./zimo a $alphabet\_temp |head -n 1| tail -n 1 >>$ZiKu;

echo "\*/" >> $ZiKu;

./zimo a $alphabet\_temp |head -n 2| tail -n 1 >>$ZiKu;

#echo $i;

done

echo "};">>$ZiKu;

echo -e ".\n\ngenerating numberASCII... \n"

number\_ASCIItotal=9;

number\_charsize=$[32\*number\_ASCIItotal];

#echo "$ZiKu cleaned up."

echo "char Rinumber[$number\_charsize]={">>$ZiKu;

for((i=0;i<=ASCIItotal+number\_ASCIItotal;i++));

do

echo "/\*" >> $ZiKu;

./zimo a $i |head -n 1| tail -n 1 >>$ZiKu;

echo "\*/" >> $ZiKu;

./zimo a $i |head -n 2| tail -n 1 >>$ZiKu;

echo $i;

done

echo "};">>$ZiKu;

#for((i=1;i<=10;i++));

# do

# echo $(expr $i \\* 3 + 1);

# done

/\*

\* RiOS/init/initialize.c

\*

\* Copyright C) 2017 Curie （邱日）

\*/

int initialize(){

/\*first, initialize gdt(global descriptor table) with zero, and set it(add records and lgdt ect)

\*then, initialize idt(global interrupt table) with zero,and set it(add records and lidt ect)

\*after that,send ICW to 8259 port ,NOTE!!DONOT forget to sti!!! (enable interrupt request)

\* NOTE!! DONOT forget to enable keyboard interrupt Mouse interrupt and 8253 interrupt after the

\*initialization of 8253!!!

\*/

init\_gdt();

set\_gdt();

init\_idt();

set\_idt();

init\_8259();

//gdt,idt,8259 ready.

Ri\_sti();//enable interrupt

init\_8253();//光初始化还没有用，要8259允许8253时钟中断请求

/\*---------TODO:Timer---------------\*/

/\*1.初始化设定8253

\*2.设定idt的一条记录

\*3.编写每次8253中断的处理程序。

\*和键盘中断一样，在8253计时器响应的中断处理程序代码里

\*也必须给外面8259发信号，中断已接受，不然又阻塞。

\*4.必须先允许8253中断

\*/

/\*---------Timer :work done --------\*/

outb(PIC0\_IMR, 0xf9);//1111 1001 irq 1 2打开 因为keyboard是irq 1

/\*由于出现了一些问题，暂时只允许键盘，不允许鼠标\*/

/\* KeyboardMouseintEnable();\*/

//\_8253timer\_interrupt\_enable();//NOTE!! important!!

/\*注意，为了我的task\_struct进程队列别搞乱，我先暂时不开时钟中断，

\*等我的队列头都初始化好了我再开8253时钟中断

\*/

/\*2017/10/25 (现已解决):经过测试，是可以中断了。无论是10s之前还是10s之后，都可以中断

\*但有个问题，就是只能中断一次，想来应该是发出中断请求，收到

\*中断请求后响应，关中断，但接下来没有再把中断打开，所以接下来的中断响应不了了

\*我的8253也要再设置

\*2017/10/27 ADD:微机原理白学了，忘了要 NOTE!!:“中断响应”

\*不然它一直以为还没有得到回应。

\*在中断处理程序中要有outb(PIC0\_OCW2, 0x61); 要把IRQ-01接收信号结束的信息通知给PIC

\*/

init\_mem\_table(RiOSmm);/\*见RiOS/kernel/mm/memory.c\*/

/\*其中RiOSmm是用于内存管理的全局结构体指针，在main.c中定义\*/

return 0;

}

/\*

\* RiOS/init/initialize.h

\*

\* (C) 2017 Curie （邱日）

\*/

int initialize();

/\*

\* RiOS/kernel/init/interrupt/interrupt.c

\*

\* Copyright (C) 2017 Curie （邱日）

\*/

/\*-------interrupt-function----------\*/

/\*TODO :add more Interrupt service functions and add records in interrupt descriptor table\*/

/\*NOTE!! IDT is important \*/

void Hello\_RiOS(int \*esp)

{

//console\_cls();

int keydata;

outb(PIC0\_OCW2, 0x61);//要把IRQ-01接收信号结束的信息通知给PIC

#define KEYDATAPort 0x60

keydata = inb(KEYDATAPort);

/\*一定要把数取走，中断响应

\*两个事情，第一给8259发送接受信号，第二把数取走。

\*这两个只要一个没有做就不行

\*/

// char str[]="/\*------Welcome to RiOS!------\*/~\

// Copyright (C) 2017 Curie ^";

int i=0;//~ => \n ,^ => \0

int ch;

// for(i=0;str[i]!='^';i++){

// ch=str[i];if(ch>=127|ch<32)ch=32;

// Ri\_putchar(ch);

// }

/\*好像puchcar(65)可以，尝试把char转为int,这样就可以输出了

\*中断服务子程序要体现关中断，要给8259一个回应，不然它被阻塞了

\*NOTE!!!不能return;

\*Ri\_putchar\_colorful(95,0b10000000);

\*考虑用写显存的方法控制光标（‘\_’ascii字符95,把它属性设为闪烁看看）

\*80×25 显示效果 王爽《汇编语言》 |7 6 5 4 3 2 1 0

\* |BL(闪烁) R G B 高亮 R G B

\*/

unsigned char \*pVRAM=(char\*)0xb8000;

/\* #define \_setTcolor pVRAM[(80\*SCREEN\_Y+SCREEN\_X)\*2+1]=0b11000010 \*/

// \_setTcolor,Ri\_putchar(58),\_setTcolor,Ri\_putchar(45),\_setTcolor,Ri\_putchar(41);

char str2[]="~[root@localhost]# ^";

/\* #define Black\_greenT 0b00000010\*/

for(i=0;str2[i]!='^';i++){

ch=str2[i];//if(ch>=127|ch<32)ch=32;

Ri\_putchar\_colorful(ch,Black\_greenT);//好像puchcar(65)可以，尝试把char转为int,这样就可以输出了

}

return;

}

/\*我的电脑开机后一直打印A,先把这两个中断给注释掉\*/

void myinterruptProcessA(int \*esp){

// Ri\_putchar('A');

// Sleep(1);

return;

}

/\*我的电脑开机后一直打印A,先把这两个中断给注释掉\*/

void myinterruptProcessB(int \*esp){

//while(1)Ri\_putchar('B');

// outb(PIC0\_OCW2, 0x61);

// int keydata;

// #define KEYDATAPort 0x60

// keydata = inb(KEYDATAPort);

// //一定要把数取走，不然会阻塞

// Ri\_putchar(keydata);

// Sleep(1);

return;

}

/\*

\* RiOS/kernel/init/interrupt/interrupt.h

\*

\* Copyright (C) 2017 Curie （邱日）

\*/

/\*本程序和RiOS/kernel/main\_gas.S联系紧密

\*这里的函数都是中断处理函数，由main\_gas.S中那些汇编语言写的函数调用

\*这里函数的特点 void hello(int \*esp)

\*/

void \_8253timer\_interrupt(int \*esp);//NOTE!!每次8253发生时钟中断，就调用它

//进程调度也在这个部分调度，这是调用频率最高的函数了。

//function defined in main\_gas.S

/\*gas\_handlers---------\*/

/\*在gas写成的汇编函数里call中断服务程序，gas\_handlers主要作用：

\*push ;call InterruptServiceFunc ;pop

\*/

/\*---interrupt service functions----------\*/

/\*first,add a record in idt,define the address of gas\_handler\_myinterrupt1 in main\_gas.S

\*then,in gas\_handler\_myinterrupt1 push,call myinterrupt1,pop

\*finally the CPU will run the code in those interrupt service function.

\*/

/\*TODO :add more Interrupt service functions and add records in interrupt descriptor table\*/

void Hello\_RiOS(int \*esp);/\*显示一些欢迎信息，体现如何写显存\*/

void myinterruptProcessA(int \*esp);

void myinterruptProcessB(int \*esp);

/\*

\* RiOS/kernel/kernel.ld

\*

\* Copyright (C) 2017 Curie （邱日）

\*/

/\*syntax learned from XV6 and JOS.

\*link script for RiOS/main.c

\* reference: https://github.com/leenjewel/xv6\_learn/blob/learn/kernel.ld

\*/

/\*

\*这是一个链接脚本,它指明了RiOS/kernel/main.c的main函数是int RiOSmain()

\*在makefile里面非常重要

\*/

OUTPUT\_FORMAT("elf32-i386", "elf32-i386", "elf32-i386")

OUTPUT\_ARCH(i386)

ENTRY(RiOSmain)/\*main function:RiOSmain\*/

SECTIONS

{

/\* Link the kernel at this address: "." means the current address \*/

. = 0x0000;

.text : {

\*(.text)

}

.rodata : {

\*(.rodata)

}

.data : {

\*(.data)

}

.bss : {

\*(.bss)

}

}

/\*

\* RiOS/kernel/main.c

\*

\* Copyright (C) 2017 Curie （邱日）

\*/

#define NULL ((void \*)0)

/\*注意（char \*）0和（void \*）0的区别，前者只是空指针值，后者可以转化为任意类型的指针\*/

/\*--------- 80\*25 Text MODE------------\*/

unsigned int jiffies=0;/\*从开启8253计时器后的滴答数\*/

unsigned char SCREEN\_X=0;/\*下一个显示字符的x位置\*/

unsigned char SCREEN\_Y=0;/\*下一个显示字符的y位置\*/

int RiOSmain();/\*NOTE!! int RiOSmain():real main function\*/

/\*注意，由于链接的一些问题，在RiOSmain的定义之前只能放声明和.h文件

\*.c文件应当放在RiOSmain的定义之后，不然会出错

\*/

#include "mm/memory.h" //unsigned int Ri\_malloc(unsigned int size);

Mem\_table \* RiOSmm=(struct Mem\_table \*)0x3c0000;/\*用于内存管理的全局结构体指针，在initialize里初始化\*/

/\*在initialize里面调用init\_mem\_table(RiOSmm);\*/

#define Black\_greenT 0b00000010

/\*#define \_setTcolor pVRAM[(80\*SCREEN\_Y+SCREEN\_X)\*2+1]=0b11000010\*/

#define \_setTcolor pVRAM[(80\*SCREEN\_Y+SCREEN\_X)\*2+1]=0b00000010

#include "console/console.h"

#include "include/asm/8259\_8253.h"

#include "init/gdt\_idt.h"

#include "init/initialize.h"

#include "init/interrupt/interrupt.h"

#include "task/task.h"

#include "task/queue/queue.h"

#include "include/rios/xwindow.h"

int currentTASKstate = 0;/\*当前是否有任务，至关重要的全局变量\*/

/\*NOTE!!在RiOSmain函数的定义头上不能放.c文件，只能放.h头文件,不然

\*会报错“Trying to execute code outside RAM or ROM”\*/

/\*---在RiOSmain上面只能放函数的声明，不要把.c文件放在这里，不然出错----\*/

LinkQueue \*q1;

LinkQueue \*q2;

LinkQueue \*q3;

/\*全局队列指针，MLFQ多级反馈队列，这里我们搞三级队列，这里是三个头

\*注意，一定要对这三个LinkQueue \*进行malloc

\*/

typedef QueuePtr tskStruQptr;

tskStruQptr current;

/\*内核全局变量current当前任务指针

\*current->data类型为task\_struct,这里current取从队列中摘下的一个节点

\*/

void showtaskMSG();

int RiOSmain()

{//console main(NOTE!!real main function.defined in RiOS/kernel/kernel.ld)

initialize();/\*-init/initialize.c-\*/

console\_cls\_black();

q1=(LinkQueue\*)Ri\_malloc(sizeof(LinkQueue));

initQueue(q1);/\*千万不要忘记初始化\*/

q2=(LinkQueue\*)Ri\_malloc(sizeof(LinkQueue));

initQueue(q2);/\*千万不要忘记初始化\*/

q3=(LinkQueue\*)Ri\_malloc(sizeof(LinkQueue));

initQueue(q3);/\*千万不要忘记初始化\*/

current=(QueuePtr)Ri\_malloc(sizeof(QNode));

currentTASKstate=0;

showtaskMSG();/\*显示提示信息\*/

\_8253timer\_interrupt\_enable();

/\*三级反馈队列的初始化，初始化完了我再开时钟中断\*/

/\*在main函数里我好像一个任务都没有创建，其实我的任务创建放在中断中

\*任务创建请见RiOS/kernel/task/task.c的IntoReady()函数

\*/

while(1){;}//死循环

return 0;

}

/\*这里的.c文件不要放在RiOSmain函数定义上面，不然会出错\*/

#include "mm/memory.c"

#include "console/console.c"

#include "init/8259\_8253.c"

#include "init/gdt\_idt.c"

#include "init/initialize.c"

#include "init/interrupt/interrupt.c"

#include "task/task.c"

#include "task/queue/queue.c"

/\* unsigned int jiffies=0;\*/

/\*全局变量jiffies是从开机后的滴答数，非常重要我把定义放到开头\*/

void \_8253timer\_interrupt(int \*esp)

{

#define \_8259know 0x60

outb(PIC0\_OCW2 ,\_8259know);/\*NOTE!!中断响应，这句话必须要有\*/

++jiffies;

xy\_putdec(jiffies,0,0);

/\*-------------------在时钟中断内的进程调度\*----------------\*/

do\_timer();/\*定义在RiOS/kernel/task/task.h中，它do\_timer调用schedule函数\*/

return;

}//~

void showtaskMSG()

{

unsigned char \*pVRAM=(char\*)0xb8000;

resetSCREENYX(0,10);

char str0[]="----jiffies(System Time)-----\n";

printlnColorful(str0);

resetSCREENYX(1,0);

char str1[]="Q1Len-----pid----start\_time--run\_time\n";

printlnColorful(str1);

resetSCREENYX(3,0);

char str2[]="Q2Len-----pid----start\_time--run\_time\n";

printlnColorful(str2);

resetSCREENYX(5,0);

char str3[]="Q3Len-----pid----start\_time--run\_time\n";

printlnColorful(str3);

resetSCREENYX(15,10);

char str4[]="----MultiStageFeedbackQueue Process Schedule Algorithm-----\n";

char str5[]="| L1 Queue :Round Robin Algorithm |\n";

char str6[]="| L2 Queue :Round Robin Algorithm |\n";

char str7[]="| L3 Queue :First Come First Served |\n";

printlnColorful(str4);

resetSCREENYX(16,10);printlnColorful(str5);

resetSCREENYX(17,10);printlnColorful(str6);

resetSCREENYX(18,10);printlnColorful(str7);

char infomsg[]="RiOS Copyright (C) 2017 QiuRi \n";

resetSCREENYX(23,10);printlnColorful(infomsg);

resetSCREENYX(9,0);

char str8[]="current task--pid--start\_time--need--run\_time\n";

printlnColorful(str8);

resetSCREENYX(1,55);

char str9[]="task1\n";

printlnColorful(str9);

resetSCREENYX(2,55);

char str10[]="task2\n";

printlnColorful(str10);

resetSCREENYX(3,55);

char str11[]="task3\n";

printlnColorful(str11);

resetSCREENYX(4,55);

char str12[]="task4\n";

printlnColorful(str12);

resetSCREENYX(5,55);

char str13[]="task5\n";

printlnColorful(str13);

resetSCREENYX(6,55);

char str14[]="task6\n";

printlnColorful(str14);

return;

}

/\*---- xwindow 图形界面暂时不做，还是kernel最重要----\*/

#include "lib/character/RiASCII.txt"

#include "lib/character/RiAlphaNum.txt"

#include "xwindow/xwindow.c"

/\*

\* RiOS/kernel/main\_gas.S

\*

\* Copyright (C) 2017 Curie （邱日）

\*/

/\*本程序和RiOS/kernel/init/interrupt/interrupt.h,interrupt.c联系紧密

\*在这里的函数call的那些中断处理函数的形式是void hello(int \*esp),都在上述地方声明和定义

\*/

.global Ri\_load\_gdt

.global Ri\_load\_idt

.global Ri\_load\_ldt

.global gas\_Handle\_8253timer /\*时钟中断处理函数 NOTE!!很重要\*/

.global gas\_ltr

.global gas\_Handle\_myinterruptProcessA

.global gas\_Handle\_myinterruptProcessB

.code32

Ri\_load\_gdt: /\*void Ri\_load\_gdt(int length,int address){\*///stack

mov 4(%esp) ,%ax

mov %ax,6(%esp)

lgdt 6(%esp)

ret//这里用ret,ret 和 iret 还不太一样

Ri\_load\_idt:

mov 4(%esp) ,%ax

mov %ax,6(%esp)

lidt 6(%esp)//之前这地方lidt错写成lgdt了，找了半天

ret

Ri\_load\_ldt:

mov 4(%esp) ,%ax

mov %ax,6(%esp)

lldt 6(%esp)//之前这地方lidt错写成lgdt了，找了半天

ret

#ljmp $section, $offset

#错误！

# Ri\_farjmp: //; void farjmp(int eip, int cs);//远跳转 ljmp

# ljmp \*4(%esp)

# ret

# //; eip, cs //从Linux0.11 sched.h 宏汇编ljmp \*%0...得到启发

# //C语言调用 usage: Ri\_farjmp(eip的值，cs的值); Ri\_farjmp(0,4>>3);

gas\_Handle\_8253timer:

pushw %es

pushw %ds

pushal

movl %esp,%eax

pushl %eax

movw %ss,%ax

movw %ax,%ds

movw %ax,%es//ensure ss=ds=es

//call myinterruptProcessB

call \_8253timer\_interrupt

/\*\_8253timer\_interrupt定义在RiOS/init/interrupt/interrupt.h中

\*时钟中断的处理函数可能是调用频率最高的函数了，它的声明void \_8253timer\_interrupt(int \*esp)

\*此类函数参数列表要注意一下，都是(int \*esp)

\*/

popl %eax

popal

popw %ds

popw %es

iret

gas\_Handle\_myinterruptProcessA:

pushw %es

pushw %ds

pushal //no pushad,but pusha in Ubuntu

//movl %eax,%esp

mov %esp,%eax

pushl %eax

movw %ss,%ax

movw %ax,%ds

movw %ax,%es//ensure ss=ds=es

call myinterruptProcessA//定义在RiOS/init/interrupt/interrupt.h

//call的这类函数参数列表要注意一下，都是(int \*esp)，例如void hello(int \*esp)

popl %eax

popal//popal //popad

popw %ds

popw %es

iret

gas\_Handle\_myinterruptProcessB:

pushw %es

pushw %ds

pushal

movl %esp,%eax

pushl %eax

movw %ss,%ax

movw %ax,%ds

movw %ax,%es//ensure ss=ds=es for Clang

//call myinterruptProcessB

call Hello\_RiOS

popl %eax

popal

popw %ds

popw %es

iret

gas\_ltr://gas\_ltr(int tr)

ltr 4(%esp)

ret

/\*

\* RiOS/rios/main.h

\*

\* Copyright (C) 2017 Curie （邱日）

\*/

laji=head.asm head.elf head.o main.elf \

main.o main.out \

main\_gas.o \

main.asm mainPlus.o

###CFLAGS = -g #gdb 加载符号表

ok:

gcc -nostdinc -I. -fno-stack-protector -c RiOShead.S -m32 -o head.o

ld -m elf\_i386 head.o -e start -Ttext 0xc400 -o head.elf

objcopy -S -O binary -j .text head.elf head.out

objdump -S head.elf > head.asm

gcc -fno-stack-protector -c main.c -m32 -o main.o

#g++ -fno-stack-protector -c main.cpp -m32 -w -fpermissive -o main.o

############################################

gcc -c -m32 main\_gas.S -o main\_gas.o

ld -m elf\_i386 -r -o mainPlus.o main.o main\_gas.o

##ld -m elf\_i386 -e RiOSmain -T kernel.ld -o main.elf main.o

ld -m elf\_i386 -e RiOSmain -T kernel.ld -o main.elf mainPlus.o

############################################

objdump -S -D main.elf >main.asm

objcopy -S -O binary main.elf main.out

cat main.out >> head.out

mv head.out kernel.out

rm -f $(laji)

clean:

rm -f $(laji)

#Notes:

#gcc -nostdinc -I. -fno-stack-protector -c main.c -m32 -o main.o

#-nostdinc 不在标准系统目录中搜索头文件，只在-I指定的目录中搜索

#不加这句 会编译stdio.h啥的，速度会变慢很多，但不加这句 stdio.h time.h没法用

#g++ -nostdinc -I. -fpermissive -fno-stack-protector -c main.cpp -m32 -o main.o

#warning: invalid conversion from ‘int’ to ‘unsigned char\*’ [-fpermissive]

#-fpermissive:error -> warning

#make -s (slient,禁止命令的显示)

# main.c 需要用到 汇编，因此写了main\_gas.S，通过C里的exter外部函数定义在汇编语言中

#先把.c和.S都编译成.o再链接，链接成mainPlus.o

#在gcc编译main\_gas.S时要加-c选项,只编译，不链接;还需要-m elf\_i386 选项，不然i386和x86\_64不兼容

#ld -m elf\_i386 -r -o mainPlus.o main.o maingas.o

#这里必须要加 -r 选项（raw）(直接链接多个.o文件的方法)

/\*

\* RiOS/kernel/mm/memory.c

\*

\* Copyright (C) 2017 Curie （邱日）

\*/

/\*void init\_mem\_table(struct Mem\_table \*Mtable);

\*为整合好后的内存管理模块初始化函数，我把它放在initialize()函数里来调用

\*这个函数用来初始化用于内存管理的全局结构体指针RiOSmm

\*/

void init\_zero\_mem\_table(Mem\_table \*Mtable)

{

Mtable->mem\_record\_total=0;

return;

}//~

int mem\_record\_free(Mem\_table \*Mtable,unsigned int upointer,unsigned int psize)

{

int i,j;

for(i=0;i<Mtable->mem\_record\_total;i++){

if(Mtable->mem\_record[i].mem\_addr>upointer){

break;

}

}/\*此时mem\_record[i]<upointer<mem\_record[i+1]\*/

if(i>0){

/\*@1 准备和前面的空间合并，前面当然不能没有空间\*/

if(Mtable->mem\_record[i-1].mem\_addr+Mtable->mem\_record[i-1].mem\_size==upointer){

/\*现在送过来的这个指针要free掉，而它前面恰好紧挨着一块已经free的内存，

\*那只需要把现在的这个指针所指内存并入这前面的记录即可，size变大

\*但不需要添加新的记录

\*/

Mtable->mem\_record[i-1].mem\_size+=psize;

if(i<Mtable->mem\_record\_total){/\*企图和后面的free内存合并\*/

if(psize+upointer==Mtable->mem\_record[i].mem\_addr){

/\*如果真的可以合并，也就是说此时

\*前面一块free中间是我准备free的pointer,后面又紧挨着一块free内存

\*等于说这三块是连在一块的，那我就要删除一条记录，但把size弄成三个的合体、

\*/

Mtable->mem\_record[i-1].mem\_size+=Mtable->mem\_record[i].mem\_size;

Mtable->mem\_record\_total--;/\*删去一条记录\*/

for(;i<Mtable->mem\_record\_total;i++){/\*后面整个往前挪一格\*/

Mtable->mem\_record[i]=Mtable->mem\_record[i+1];

}

}

}

return 0;/\*free成功\*/

}

}//~@1

if((i<Mtable->mem\_record\_total)&&((upointer+psize)==Mtable->mem\_record[i].mem\_addr)){

/\*@2 若不能和前面的free内存合并

\*但可以和后面的free内存合并\*/

Mtable->mem\_record[i].mem\_addr=upointer;

Mtable->mem\_record[i].mem\_size+=psize;

return 0;/\*free成功\*/

}//～@2

#define MAX\_MEM\_RECORDS 4090

if(Mtable->mem\_record\_total<MAX\_MEM\_RECORDS){

/\*||前面的free\_i|| ||待free的p所指内存|| ||后面的free\_i+1||\*/

/\*@3 若不能和前面的free内存合并

\*也不能和后面的free内存合并\*/

for(j=Mtable->mem\_record\_total;j>i;j--){

/\*均向后挪一格\*/

Mtable->mem\_record[j]=Mtable->mem\_record[j-1];

}

Mtable->mem\_record\_total++;/\*来增加我这新的要free的p的记录(变成第i+1条记录)\*/

Mtable->mem\_record[i].mem\_addr=upointer;

Mtable->mem\_record[i].mem\_size=psize;

return 0;

}//～@3

/\*分配失败\*/

return -1;

}//~

void init\_mem\_table(Mem\_table \*Mtable)

{

init\_zero\_mem\_table(Mtable);

/\*注意在第一次使用Ri\_malloc之前先要对内存管理的部分init一下，

\*而且 NOTE!! 要把不是系统已经用了的地方给free掉

\*要free，不然全是0，没有可用空间

\*/

int memtotal;

/\*前面的0x400000留给我的系统，当然，这是个随便定的值

\*只要别把我的代码区也给malloc了就行

\*/

/\*memtotal=getmemorysize\_bit(0x400000,0xffffffff);

\*本来应该用上面这句话来检测内存大小，但是经过实体机测试，这句话太耗时了

\*那我就来个硬编码，规定memtotal是4G

\*/

memtotal=0xffffffff;

mem\_record\_free(Mtable,0x400000,memtotal-0x400000);

/\*print "total memory: "; putdec(memtotal>>20);print("MB")\*/

return;

}

void \* Ri\_malloc(unsigned int size)

{

/\*注意此处的RiOSmm是个struct Mem\_table \*类型的全局结构体指针变量

\*整个系统只有这一个,专门用来内存管理

\*/

return mem\_record\_alloc(RiOSmm,size);

}

/\*int Ri\_free(unsigned int upointer,int size){\*/

int Ri\_free(void\* upointer,int size)

{

/\*注意此处的RiOSmm是个struct Mem\_table \*类型的全局结构体指针变量

\*整个系统只有这一个,专门用来内存管理

\*/

return mem\_record\_free(RiOSmm,(unsigned int)upointer,size);

}

/\*int mem\_record\_alloc(Mem\_table \*Mtable,unsigned int psize){\*/

void\* mem\_record\_alloc(Mem\_table \*Mtable,unsigned int psize)

{

/\*unsigned int i,getp;\*/

unsigned int i;void \* getp;/\*其实就是unsigned int getp;\*/

for(i=0;i<Mtable->mem\_record\_total;i++){

if(Mtable->mem\_record[i].mem\_size>=psize){

/\*若找到一块大小足够的可用内存\*/

getp=(void\*)Mtable->mem\_record[i].mem\_addr;

Mtable->mem\_record[i].mem\_addr+=psize;

Mtable->mem\_record[i].mem\_size-=psize;

if(Mtable->mem\_record[i].mem\_size==0){

/\*若这块内存恰好分完，没有剩余量，就删去这条记录\*/

Mtable->mem\_record\_total--;

/\*删去这条记录以后，后面的记录都要往前喏一格\*/

for(;i<Mtable->mem\_record\_total;i++){

Mtable->mem\_record[i]=Mtable->mem\_record[i+1];

}

}

return getp;

}

}

putdec(404);/\*以后要注释掉\*/

return 0;/\*run out of memory\*/

}

int getmemorysize\_bit()

{

unsigned int start\_addr=0x666666;//内存检测是从此地址往后检测

/\*因为害怕把代码那一块内存给乱改，于是首地址给的比较大，默认start\_addr前面的内容是已经使用的内存，不检测

\*如果要检测的时候给电脑或者qemu分配的内存比较小，可能就要把start\_addr改小一点了

\*反正只要保证它在我的代码外，但又不超过内存最大地址就行

\*/

unsigned int end\_addr=0xffffffff;/\*最大能检测4G\*/

unsigned int i;unsigned before;

unsigned int word=0x66886688;

unsigned int dorw=0x88668866;

volatile unsigned int \*p;//与硬件相关，提示编译器不要对volatile限定符修饰的对象进行编译器优化

for(i=start\_addr;i<end\_addr;i+=0x1000){

p=(unsigned int \*)i+0x200;

before=\*p;

\*p=word;

if(\*p!=word){

\*p=before;break;

// goto ENDofMEMORY;

}//ENDofMEMORY:

\*p=before;

/\*这里得到的i是bit数，若要返回MB(2^20B)的话要右移20位\*/

}

return i;

}//~

int getmemorysize\_MB()

{ int bits=getmemorysize\_bit();/\*这里得到的i是bit数，若要返回MB(2^20B)的话要右移20位\*/

return (bits>>20);

}//~

/\*

\* RiOS/kernel/mm/memory.h

\*

\* Copyright (C) 2017 Curie （邱日）

\*/

typedef struct Mem\_record

{

unsigned int mem\_addr;

unsigned int mem\_size;

}Mem\_record;

typedef struct Mem\_table

{

unsigned int mem\_record\_total;

Mem\_record mem\_record[4090];

}Mem\_table;

void init\_zero\_mem\_table( Mem\_table \*Mtable);

int mem\_record\_free( Mem\_table \*Mtable,unsigned int upointer,unsigned psize);

void init\_mem\_table(Mem\_table \*Mtable);

void\* Ri\_malloc(unsigned int size);

/\*int Ri\_free(unsigned int upointer,int size);\*/

int Ri\_free(void\* upointer,int size);

/\*int mem\_record\_alloc( Mem\_table \*Mtable,unsigned int psize);\*/

void\* mem\_record\_alloc( Mem\_table \*Mtable,unsigned int psize);

int getmemorysize\_bit();

int getmemorysize\_MB();

/\* void\* 这个类型可以转化为任意类型的指针，其实就是unsigned int

\*之前所有void\*的地方我都用的是unsigned int 类型

\*但是这样直接把一个无符号整型变为指针，不安全，

\*编译器老报WARNING:'makes integer from pointer without a cast'

\*我就把所有unsigned int 指针的地方改成了 void \*

\*/

/\*

\* RiOS/task/queue/queue.c

\*

\* Copyright (C) 2017 Curie （邱日）

\*/

/\*1.注意在第一次使用Ri\_malloc之前先要对内存管理的部分init一下，

\*而且 NOTE!! 要把不是系统已经用了的地方给free掉

\*要free，不然全是0，没有可用空间

\*2.使用这些函数时要注意，要用取地址字符 & NOTE!! ，我这是纯C,不能用C++的引用

\*/

/\*NOTE!! q1=(LinkQueue\*)Ri\_malloc(sizeof(LinkQueue));\*/

/\*NOTE!! q1=(LinkQueue\*)Ri\_malloc(sizeof(LinkQueue));\*/

/\*NOTE!! q1=(LinkQueue\*)Ri\_malloc(sizeof(LinkQueue));\*/

void initQueue (LinkQueue \*q)

{

/\*usage: LinkQueue \*q;q=(LinkQueue\*)Ri\_malloc(sizeof(LinkQueue));

initQueue(q);NOTE!! 先malloc再init\*/

q->front=(QueuePtr)Ri\_malloc(sizeof(QNode));

q->rear=q->front;

if(!q->front ) return;

q->front ->next =NULL;

// q->rear->next=NULL;

return;

}

void EnQueue(LinkQueue \*q,QElemType e)

{

QueuePtr p;

p=(QueuePtr)Ri\_malloc(sizeof(QNode));

if(!p)putdec(404);/\*以后要注释掉，改成其他处理\*/

p->data =e;

p->next =NULL;

q->rear ->next =p;

q->rear =p;

return;

}

void pushpQueue(LinkQueue \*q,QueuePtr ptsk)

{

q->rear ->next =ptsk;

q->rear =ptsk;

return;

}

void DeQueue(LinkQueue \*q,QElemType \*e)

{

/\*usage: QElemType e;e=(QElemType)Ri\_malloc(sizeof(QElemType)); DeQueue(q,e);NOTE!! &\*/

if(q->front == q->rear )return ;

QueuePtr p;

p=q->front ->next ;

\*e=p->data ;/\*NOTE!!在主函数 int \*tmpDel,DeQueue(&q,&tmpDel);putdec(tmpDel)\*/

//e=p->data;

q->front ->next =p->next ;

if(q->rear == p) q->rear = q->front ;

/\* free(p);\*/

Ri\_free((void \*)p,sizeof(p)),p=NULL;

}

void popQueue(LinkQueue \*q)

{

/\*usage: QElemType e;e=(QElemType)Ri\_malloc(sizeof(QElemType)); DeQueue(q,e);NOTE!! &\*/

if(q->front == q->rear )return ;

QueuePtr p;

p=q->front ->next ;

//\*e=p->data ;/\*NOTE!!在主函数 int \*tmpDel,DeQueue(&q,&tmpDel);putdec(tmpDel)\*/

q->front ->next =p->next ;

if(q->rear == p) q->rear = q->front ;

/\* free(p);\*/

Ri\_free((void \*)p,sizeof(p)),p=NULL;

}

QueuePtr getPpopQueue(LinkQueue \*q)

{

if(q->front == q->rear )return NULL;

QueuePtr p;

p=q->front ->next ;

//\*e=p->data ;/\*NOTE!!在主函数 int \*tmpDel,DeQueue(&q,&tmpDel);putdec(tmpDel)\*/

q->front ->next =p->next ;

if(q->rear == p) q->rear = q->front ;

/\* free(p);\*/

/\*Ri\_free((void \*)p,sizeof(p)),p=NULL;\*/

p->next=NULL;

return p;

/\*getPpopQueue与popQueue的区别就在于，pop出来的那个指针不是free掉

\*而是当个宝捡起来，把next置为空，p本身留着，准备插在其他队列的屁股后面

\*此函数专为task\_struct打造

\*/

}

void DetroyQueue(LinkQueue \*q)

{/\*usage： DetroyQueue(q);NOTE!! & \*/

while(q->front){

q->rear =q->front ->next ;

/\* free(q->front );\*/

Ri\_free(q->front,sizeof(q->front)),q->front=NULL;

q->front =q->rear ;

}

return;

}

void GetTop(LinkQueue \*q,QElemType \*e)

{/\*usage: QElemType \*temp;temp=(QElemType)Ri\_malloc(sizeof(QElemType));GetTop(q,temp);注意要加取地址符号\*/

QueuePtr p=q->front->next ;

\*e=p ->data ;

return;

}

QueuePtr getpTop(LinkQueue \*q){

return (q->front->next);

}

int QueueLen(LinkQueue \*q){

int i=0;

QueuePtr p;

p=q->front->next;

while(p){

i++;

p=p->next;

}

return i;

}

/\*注意在第一次使用Ri\_malloc之前先要对内存管理的部分init一下，

\*而且 NOTE!! 要把不是系统已经用了的地方给free掉

\*要free，不然全是0，没有可用空间

\*/

/\*unsigned int Ri\_malloc(unsigned int size){

\* // if(RiOSmm\_initialized)memman\_init(RiOSmm),RiOSmm\_initialized=1;

\* return memman\_alloc(RiOSmm,size);

\*}

\*void Ri\_free(unsigned int upointer,int size){

\*memman\_free(RiOSmm,upointer,size);

\*}

\*/

/\*NOTE!!注意我这是裸机上的纯C，不能用C++的引用\*/

/\*NOTE!!NOTE!!initQueue,千万不要忘记，用队列要先初始化！！

\*先调用initQueue,再去入队出队

\*/

// void queue\_print(LinkQueue \*q){/\*NOTE!!此函数的用于Debug,测试时拿int做,typedef int QElemType;\*/

// QueuePtr p;

// p=q->front->next;

// while(p){

// putdec(p->data),tab();

// /\*这个只能用在data QElemType是int时，当类型变了要重写queue\_print函数\*/

// p=p->next;

// }

// }

// void test\_queue(){/\*NOTE!!此函数的用于Debug,测试时拿int做,typedef int QElemType;\*/

// LinkQueue \*q;

// /\*NOTE!!q1=(LinkQueue\*)Ri\_malloc(sizeof(LinkQueue));

// \*NOTE!!特别注意,LinkQueue q1;q1本身也要malloc，结构体指针自己也要初始化

// \*/

// q=(LinkQueue\*)Ri\_malloc(sizeof(LinkQueue));/\*Very important!\*/

// /\*队列指针本身也要初始化，malloc,不malloc的话

// \*一个队列各种操作没有问题，两个也行，当我搞到第八个队列时就完蛋了

// \*LinkQueue \*指针也要初始化

// \*/

// //q=(LinkQueue\*)Ri\_malloc(sizeof(LinkQueue));/\*Very important!\*/

// //q=(LinkQueue\*)Ri\_malloc(sizeof(LinkQueue));/\*Very important!\*/

// initQueue(q);

// QElemType data1=2;

// QElemType data2=3;

// int i;

// QElemType \*tmpDel;

// QElemType \*tmpTop;

// tmpDel=(QElemType)Ri\_malloc(sizeof(QElemType));

// tmpTop=(QElemType)Ri\_malloc(sizeof(QElemType));

// for(i=0;i<20;i++){

// EnQueue(q,2\*i+1);

// }

// queue\_print(q),endl();

// DeQueue(q,tmpDel);

// queue\_print(q),endl();

// char str1[]="Del:\n";

// printnoln(str1);putdec(\*tmpDel),tab();

// char str2[]="GetTop:\n";

// printnoln(str2),GetTop(q,tmpTop),putdec(\*tmpTop),tab();

// char str3[]="Len:\n";

// printnoln(str3),putdec(QueueLen(q)),tab();

// return;

// }

// void \_int\_test3queues(){/\*NOTE!!此函数的用于Debug,测试时拿int做,typedef int QElemType;\*/

// LinkQueue \*q1;

// q1=(LinkQueue\*)Ri\_malloc(sizeof(LinkQueue));

// initQueue(q1);

// LinkQueue \*q2;

// q2=(LinkQueue\*)Ri\_malloc(sizeof(LinkQueue));

// initQueue(q2);

// LinkQueue \*q3;

// q3=(LinkQueue\*)Ri\_malloc(sizeof(LinkQueue));

// initQueue(q3);

// int i;

// for(i=1;i<=3;i++)

// EnQueue(q1,i);

// for(i=21;i<26;i++)

// EnQueue(q2,i);

// for(i=31;i<39;i++)

// EnQueue(q3,i);

// queue\_print(q1),endl();

// queue\_print(q2),endl();

// queue\_print(q3),endl();

// }

// LinkQueue \* \_MultiLevelIntQueue(){

// LinkQueue \*q1;

// q1=(LinkQueue\*)Ri\_malloc(sizeof(LinkQueue));

// initQueue(q1);

// LinkQueue \*q2;

// q2=(LinkQueue\*)Ri\_malloc(sizeof(LinkQueue));

// initQueue(q2);

// LinkQueue \*q3;

// q3=(LinkQueue\*)Ri\_malloc(sizeof(LinkQueue));

// initQueue(q3);

// int i;

// for(i=1;i<=3;i++)

// EnQueue(q1,i);

// for(i=21;i<26;i++)

// EnQueue(q2,i);

// for(i=31;i<39;i++)

// EnQueue(q3,i);

// char str1[]="Initial:\n";endl(),printnoln(str1),endl();

// queue\_print(q1),endl();queue\_print(q2),endl();queue\_print(q3),endl();

//

// QElemType \*tmpTop;

// QElemType \*tmpDel;

// while(QueueLen(q1)!=0){

// GetTop(q1,tmpTop);DeQueue(q1,tmpDel);

// EnQueue(q2,\*tmpTop);

// }

// char str2[]="Then:\n";endl(),printnoln(str2),endl();

// queue\_print(q1),endl();queue\_print(q2),endl();queue\_print(q3),endl();

//

// while (QueueLen(q2)!=0) {

// GetTop(q2,tmpTop);DeQueue(q2,tmpDel);

// EnQueue(q3,\*tmpTop);

// }

// char str3[]="After:\n";endl(),printnoln(str3),endl();

// queue\_print(q1),endl();queue\_print(q2),endl();queue\_print(q3),endl();

//

// while (QueueLen(q3)!=0) {

// GetTop(q3,tmpTop);DeQueue(q3,tmpDel);

// }

// char str4[]="End:\n";endl(),printnoln(str4),endl();

// queue\_print(q1),endl();queue\_print(q2),endl();queue\_print(q3),endl();

// char str5[]="Now,queue3 Len:\n";

// printnoln(str5),putdec(QueueLen(q3)),tab();

//

//

// };

/\*

\* RiOS/task/queue/queue.h

\*

\* Copyright (C) 2017 Curie （邱日）

\*/

typedef task\_struct QElemType;

/\*typedef int QElemType;\*/

typedef struct QNode

{

QElemType data;

struct QNode \*next;

}QNode ,\*QueuePtr;

typedef struct L

{

QueuePtr front;

QueuePtr rear;

}LinkQueue;

void initQueue (LinkQueue \*q);

void EnQueue(LinkQueue \*q,QElemType e);

void DeQueue(LinkQueue \*q,QElemType \*e);

void pushpQueue(LinkQueue \*q,QueuePtr ptsk);

/\*pushpQueue专为task\_struct所打造，它不是push进一个元素，

\*而是push进一个已经malloc搞好的一坨节点，(不仅仅包括元素)

\*由于不需要再分配空间，因此函数也变得简单了

\*/

void popQueue(LinkQueue \*q);

/\*popQueue和DeQueue功能一样，但不返回被删的东西\*/

QueuePtr getPpopQueue(LinkQueue \*q);

/\*专业回收不要的指针

\*专为task\_struct打造，pop出来的那个指针不是free掉，而是废物利用\*/

void DetroyQueue(LinkQueue \*q);

void Input(LinkQueue \*q);

void PrintQueue(LinkQueue \*q);

void GetTop(LinkQueue \*q,QElemType \*e);

QueuePtr getpTop(LinkQueue \*q);

int QueueLen(LinkQueue \*q);

/\*task\_struct 相关\*/

void print\_taskQueue(LinkQueue \*q);

void print\_One\_task(task\_struct tsk);

/\*

\* RiOS/kernel/bootloader.S

\*

\* Copyright (C) 2017 Curie （邱日）

\*/

#define BootGasStart 0x8000

#define HeadGasStart 0x10000

.code16

.global start

//对照《Linux内核完全注释修正版v3.0》图4.40

start:

xor %ax,%ax

mov %ax,%ds

mov %ax,%es

mov %ax,%ss

/\*------320\*200 VIDEO mode,simple xwindow--\*/

/\* mov $0x13,%al

\* xor %ah,%ah

\* int $0x10 //main.c \_xwindowRiOSmain

\*/

/\*int 0x13 or int 0x03 ? This is a choice. \*/

/\*------80\*25 TEXT mode,default--\*/

mov $0x03,%al

xor %ah,%ah

int $0x10//main.c RiOSmain

/\*------To do:enable address line A20-----\*/

/\*oxff=>port 0x21,port 0xa1\*/

movb $0xff,%al

outb %al,$0x21

nop

outb %al,$0x21

cli //diable interrupt

inb $0x64,%al

//reference:----http://kernelx.weebly.com/a20-address-line.html

/\*-------enable CR0 bit0,Real mode->Protected Mode\*/

lgdt gdt\_info

movl %cr0,%eax

andl $0x7fffffff,%eax

orl $0x01,%eax

movl %eax,%cr0

/\* |PG(bit31) PE(bit0)|

\* PE(BIT0):控制分段管理机制,0处理器运行于实模式,1为保护模式

\* PG(BIT31):控制分页管理机制,0禁用分页管理机制(此时线性地址等同于物理地址),1为启用(此时线性地址通过分页机制转换成物理地址)

\* 实际上我进入的是平坦内存模式 flat memory mode

\*/

ljmp $(2<<3),$Protected32Code

/\*VERY IMPORTANRT!code16->code32 ljmp for align\*/

.code32

Protected32Code:

/\*-----------To Do:flushpipeline------------\*/

movw $(1<<3),%ax

movw %ax,%ds

movw %ax,%es

movw %ax,%gs

movw %ax,%ss

//清空流水线

/\*---load Udisk data(now in memory) to RAM address:0x100000--\*/

//原理图类似赵炯《Linux内核完全注释修正版v3.0》

//图4.40内核代码在物理内存中的移动和分布情况

movl $(BootGasStart+512),%esi

movl $(HeadGasStart+512),%edi

//要加512，bootloader最初是 512Bytes

xor %ecx,%ecx

movb $16,%cl//我的128MBU盘CHS 16/255/63

imul $(16\*255\*63/4) , %ecx //U盘总bit数除以4得到字节数

subl $(512/4), %ecx //启动区大小

movl $main,%esi

movl $0x280000,%edi

//上面已经计算出ecx的大小

//mov $(88888),%ecx //88888

call copy

ljmp $(3<<3),$0x0000

//To C code seg,addr:0x280000:0000 0000

copy:

movl (%esi),%eax

addl $4,%esi

movl %eax,(%edi)

addl $4,%edi

subl $1,%ecx

jnz copy

ret

/\*Real mode 下的 global descriptor table

\*gdt最坑人的地方在于，intel为了兼容16位机和32位机,

\*把gdt的数拆开来放，非常碎，很难有直观感觉

\*gdt具体数位的摆放可参考《从实模式到保护模式》P201的图

\*，也可见赵炯《Linux内核完全注释修正版v3.0》P90分段机制的图

\*/

temp\_gdt:

.word 0x0000,0x0000,0x0000,0x0000 //null segment

.word 0xffff,0x0000,0x9200,0x00cf //data segment

/\*---code segment----------------\*/

.word 0xffff,0x0000,0x9a00,0x00c7 //gas code

.word 0xffff,0x0000,0x9a28,0x00c7 //c code

//0x9a=0x10011010:P(Segment present)1b DPL2b S:1b TYPE:4b

//TYPE 101X read,execute

//见《从实模式到保护模式》P201的图

gdt\_info:

.word 31 /\* 4\*8-1 length0->31\*/

.long temp\_gdt/\*gdt address \*/

main:

//编译链接阶段main.c的主函数RiOSmain就要链接到此处来

/\*Notes:

The 3 methods for enabling the A20 Gate :

1.Keyboard Controller

2.BIOS Function

3.System Port

----http://kernelx.weebly.com/a20-address-line.html

\*/

/\*

\* RiOS/task/task.c

\*

\* Copyright (C) 2017 Curie （邱日）

\*/

/\*多级反馈队列调度算法，第一第二级用时间片轮转，最后一级用先来先服务\*/

void do\_timer()

{

Ri\_cli();/\*关中断\*/

if(currentTASKstate==1){//如果已经有了任务，当前全局任务指针有所指向

if(jiffies>=current->data.start\_time)current->data.run\_time++;

}

if(currentTASKstate==1)run\_task(current->data);

schedule();/\*调度函数，在RiOS/kernel/task/task.h中\*/

show\_task\_queues();

IntoReady();

Ri\_sti();/\*开中断\*/

}//~

void IntoReady()

{

/\*在任务PCB的start\_time时刻创建该任务，jiffies注意要和PCB的start\_time两者一致

\*mkPCB最后两个参数是在屏幕的显示位置

\*/

if(jiffies==200)Create\_task(mkPCB(1,200,888, 1,60));

/\*优先级第1级，200时间开始，需要时间片888\*/

if(jiffies==300)Create\_task(mkPCB(3,300,777, 2,60));

/\*优先级第3级，300时间开始，需要时间片777\*/

if(jiffies==600)Create\_task(mkPCB(1,600,40, 3,60));

/\*优先级第1级，600时间开始，需要时间片40\*/

if(jiffies==1000)Create\_task(mkPCB(1,1000,222, 4,60));

/\*优先级第1级，1000时间开始，需要时间片222\*/

if(jiffies==1200)Create\_task(mkPCB(1,1200,999, 5,60));

/\*优先级第1级，1200时间开始，需要时间片999\*/

if(jiffies==1400)Create\_task(mkPCB(2,1400,666, 6,60));

/\*优先级第2级，1400时间开始，需要时间片666\*/

}//~

void run\_task(task\_struct mtsk)

{

/\*这里仅为了演示\*/

//if(currentTASKstate==1)xy\_putdec(mtsk.run\_time,mtsk.proc\_screen\_x,mtsk.proc\_screen\_y);

xy\_putdec(mtsk.run\_time,mtsk.proc\_screen\_x,mtsk.proc\_screen\_y);

}//~

void schedule()

{/\*进程调度，将在时钟中断中调用\*/

MLFQ();/\*多级反馈队列调度算法\*/

}//~

void MLFQ()

{

/\*Multi-Level Feedback Queue

\*多级反馈队列调度算法

\*前两级采用时间片轮转，最后一级采用先来先服务

\*/

if(QueueLen(q1)>0|QueueLen(q2)>0)RR\_L1L2();

FCFS\_L3();

}//~

void RR\_L1()

{/\*对第一级采用时间片轮转\*/

if(currentTASKstate==0&&QueueLen(q1)!=0){

current=getPpopQueue(q1);

currentTASKstate=1;

}

if(currentTASKstate==1){

if(current->data.run\_time>=current->data.timepiece\_needed){

/\*任务运行完毕\*/

Destroy\_current\_task();

resetcurrent();

currentTASKstate=0;

return;

}

}

if(currentTASKstate==1){

if(current->data.priority==1&&current->data.run\_time>=L1\_timepiece){

if(QueueLen(q2)<MAX\_TASK\_L2){

current->data.priority=2;

pushpQueue(q2,current);

resetcurrent();

}

else{

current->data.priority=3;

pushpQueue(q3,current);

resetcurrent();

}

}

}

}//~

void RR\_L2()

{/\*对第二级采用时间片轮转\*/

if(currentTASKstate==0&&QueueLen(q2)!=0){

current=getPpopQueue(q2);

currentTASKstate=1;

}

if(currentTASKstate==1){

if(current->data.run\_time>=current->data.timepiece\_needed){

/\*任务运行完毕\*/

Destroy\_current\_task();

resetcurrent();

currentTASKstate=0;

return;

}

}

if(currentTASKstate==1){

if(current->data.priority==2&&current->data.run\_time>=L2\_timepiece){

current->data.priority=3;

pushpQueue(q3,current);

resetcurrent();

}

}

}//~

void RR\_L1L2()

{/\*Round Robin\*/

/\*对第一第二级队列用时间片轮转算法\*/

RR\_L1();

RR\_L2();

}

void FCFS\_L3()

{/\*First Come First Served\*/

/\*对第三级队列用先来先服务调度算法\*/

if(currentTASKstate==0&&QueueLen(q3)!=0){

current=getPpopQueue(q3);

currentTASKstate=1;

}

if(currentTASKstate==1){

if(current->data.run\_time<current->data.timepiece\_needed){

return;

}

else {

// currentTASKstate=0;

// resetcurrent();//current->data=NULLTASK;

Destroy\_current\_task();

resetcurrent();

currentTASKstate=0;

return;

}

}

if(QueueLen(q1)==0&&QueueLen(q2)==0&&QueueLen(q3)==0)return;

}

long mkpid()

{/\*pid的生成我采用通过时间jiffies获取伪随机数的方法\*/

if(jiffies==0)return 1;/\*防止pid返回0\*/

return((int)jiffies<<16|jiffies<<8);

}

task\_struct\* mkPCB(long priority,long start\_time, \

long timepiece\_needed, \

int proc\_screen\_x,int proc\_screen\_y )

{

/\*pid的生成我采用通过时间jiffies获取伪随机数的方法\*/

task\_struct \*pnewPCB;

pnewPCB=(task\_struct\*)Ri\_malloc(sizeof(task\_struct));

/\*必须要分配空间\*/

pnewPCB->proc\_id=mkpid();

pnewPCB->start\_time=start\_time;

pnewPCB->proc\_state=TASK\_READY;

/\*创建完就默认ready,这个暂时不要外界赋值\*/

pnewPCB->run\_time=0;

/\*新进程尚未运行\*/

pnewPCB->priority=priority;

pnewPCB->timepiece\_needed=timepiece\_needed;

pnewPCB->proc\_screen\_x=proc\_screen\_x;

pnewPCB->proc\_screen\_y=proc\_screen\_y;

return pnewPCB;

}

void resetcurrent()

{

current=(QueuePtr)Ri\_malloc(sizeof(QNode));

current->data=NULLTASK;

return;

}

void Create\_task(task\_struct \*mtsk)

{/\*进程创建原语\*/

if(mtsk){

if((mtsk->priority==1)&&(QueueLen(q1)<MAX\_TASK\_L1))EnQueue(q1,\*mtsk);

else if((mtsk->priority==2)&&(QueueLen(q2)<MAX\_TASK\_L2))EnQueue(q2,\*mtsk);

else if(mtsk->priority==3)EnQueue(q3,\*mtsk);

/\*入相应的队，注意EnQueue(LinkQueue\*q,QElemType e);\*/

}

return;

}

void Destroy\_current\_task()

{/\*进程撤销原语\*/

Ri\_free((void \*)current,sizeof(QueuePtr)),current=NULL;

return;

}//~

void show\_current()

{

unsigned char \*pVRAM=(char\*)0xb8000;

resetSCREENYX(10,0);

//char str0[]="-----------------------------------------------------\n";

char str0[]=" \n";

println(str0);

/\*每次显示前需要刷新一下，以免有上次画面的残留\*/

if(currentTASKstate==1){

xy\_putdec(current->data.proc\_id,10,10);

xy\_putdec(current->data.start\_time,10,20);

xy\_putdec(current->data.timepiece\_needed,10,30);

xy\_putdec(current->data.run\_time,10,40);

}

return;

}//~

void reflashX\_2\_4\_6()

{/\*刷新屏幕的指定行\*/

unsigned char \*pVRAM=(char\*)0xb8000;

resetSCREENYX(2,0);

//char str0[]="-------------------------------------\n";

char str0[]=" \n";

println(str0);

resetSCREENYX(4,0);

println(str0);

resetSCREENYX(6,0);

println(str0);

}

void show\_task\_queues()

{

reflashX\_2\_4\_6();/\*把要显示的这几行刷新，省得有屏幕残留\*/

show\_queueL1();

show\_queueL2();

show\_queueL3();

show\_current();

//if(currentTASKstate==1)xy\_putdec(q3->front->next->data.start\_time,20,20);

}//~

void show\_queueL1()

{

/\*先展示队列长度\*/

xy\_putdec(QueueLen(q1),2,1);//第一级队列的队列长度

/\*再展示队头的pid\*/

if(QueueLen(q1)>0){

xy\_putdec(q1->front->next->data.proc\_id,2,10);

/\*如果有队头，展示队头的开始时间\*/

if(QueueLen(q1)>0)xy\_putdec(q1->front->next->data.start\_time,2,20);

else xy\_putdec(0,2,20);

/\*展现队头的运行时间\*/

xy\_putdec(q1->front->next->data.run\_time,2,30);

}

else {/\*如果队列里面没有东西，显示也没有意义了\*/

xy\_putdec(0,2,10),xy\_putdec(0,2,20),xy\_putdec(0,2,30);

}

}//~

void show\_queueL2()

{

/\*先展示队列长度\*/

xy\_putdec(QueueLen(q2),4,1);//第二级队列的队列长度

/\*再展示队头的pid\*/

if(QueueLen(q2)>0){

xy\_putdec(q2->front->next->data.proc\_id,4,10);

/\*如果有队头，展示队头的开始时间\*/

if(QueueLen(q2)>0)xy\_putdec(q2->front->next->data.start\_time,4,20);

else xy\_putdec(0,4,20);

xy\_putdec(q2->front->next->data.run\_time,4,30);

}

else {/\*如果队列里面没有东西，显示也没有意义了\*/

xy\_putdec(0,4,10),xy\_putdec(0,4,20),xy\_putdec(0,4,30);

}

}//~

void show\_queueL3()

{

/\*先展示队列长度\*/

xy\_putdec(QueueLen(q3),6,1);//第一级队列的队列长度

/\*再展示队头的pid\*/

if(QueueLen(q3)>0){

xy\_putdec(q3->front->next->data.proc\_id,6,10);

/\*如果有队头，展示队头的开始时间\*/

if(QueueLen(q3)>0)xy\_putdec(q3->front->next->data.start\_time,6,20);

else xy\_putdec(0,6,20);

xy\_putdec(q3->front->next->data.run\_time,6,30);

}

else {/\*如果队列里面没有东西，显示也没有意义了\*/

xy\_putdec(0,6,10),xy\_putdec(0,6,20),xy\_putdec(0,6,30);

}

}

/\*

\* RiOS/task/task.h

\*

\* Copyright (C) 2017 Curie （邱日）

\*/

/\*本系统采用多级反馈队列进程调度算法

\*,本系统中的MLFQ算法对其他调度算法进行了综合，

\*即我采用三级队列，上层队列(前两层)采用时间片轮转

\*最下层(第三层)队列采用先来先服务

\*/

#define L1\_timepiece 200

#define L2\_timepiece 400

/\*#define L3\_timepiece 800\*/

/\*其实在本系统中，不用再定义第三级的时间片了,

\*最下层队列我采用先来先服务，反正一直做，直到该任务完成

\*/

/\*进程的三态模型,对应task\_struct中的proc\_state\*/

#define TASK\_RUNNING 1

/\*running 运行态\*/

#define TASK\_READY 2

/\*ready 就绪态\*/

#define TASK\_BLOCKED 3

/\*blocked 阻塞态 即睡眠态 sleep\*/

/\*三级反馈队列，每个队列最多容纳的任务\*/

#define MAX\_TASK\_L1 3

#define MAX\_TASK\_L2 6

/\*#define MAX\_TASK\_L3 512\*/

//tss结构由硬件决定。

//赵炯《Linux0.11内核完全注释v3.0》P127图4-24 32位任务段tss格式

typedef struct tss32\_struct{//tss结构由硬件决定，固定死了是104个字节。

long back\_link; /\* 16 high bits zero\*/

long esp0;

long ss0; /\* 16 high bits zero\*/

long esp1;

long ss1; /\* 16 high bits zero\*/

long esp2;

long ss2; /\* 16 high bits zero\*/

long cr3;

/\* 32位寄存器 \*/

long eip;

long eflags;

long eax;

long ecx;

long edx;

long ebx;

long esp;

long ebp;

long esi;

long edi;

/\* 16位寄存器 \*/

long es;

long cs;

long ss;

long ds;

long fs;

long gs;

/\*----任务设置部分，任务切换时不会被写入CPU,但不能随意赋值，否则无法正常切换----\*/

long ldt;

long trace\_bitmap;// I/o位图基地址

}tss32\_struct;//~

typedef struct task\_struct{

long proc\_id;/\*进程编号\*/

long start\_time;/\*进程创建时间\*/

long proc\_state;/\*进程状态\*/

long run\_time;/\*进程运行时间\*/

long priority;/\*优先级\*/

long timepiece\_needed;/\*本进程运行完需要的时间片\*/

/\*下面和task\_struct关系不大，只是为了本次实验演示需要\*/

int proc\_screen\_x;

int proc\_screen\_y;

tss32\_struct proc\_tss32;

/\*本来应当包括tss32\_struct\*/

}task\_struct;

task\_struct NULLTASK={

.proc\_id=0,/\*进程编号\*/

.start\_time=0,/\*进程创建时间\*/

.proc\_state=0,/\*进程状态\*/

.run\_time=0,/\*进程运行时间\*/

.priority=0,/\*优先级\*/

.timepiece\_needed=0,/\*本进程运行完需要的时间片\*/

/\*下面和task\_struct关系不大，只是为了本次实验演示需要\*/

.proc\_screen\_x=30,

.proc\_screen\_y=30

};

/\*全局指针current的data要为空的时候，我又不想把current变为NULL

\*那样current可能会有malloc方面的问题，我就搞一个这样的NULLTASK

\*/

void do\_timer();

/\*do\_timer是调用频率非常高的函数，在时钟中断中调用，do\_timer

\*自己调用schedule

\*/

/\*由于需要“手动创建进程，需要事先做好pcb”\*/

task\_struct\* mkPCB(long priority,long start\_time,\

long timepiece\_needed,\

int proc\_screen\_x,int proc\_screen\_y);

/\*pid的生成我采用通过时间jiffies获取伪随机数的方法\*/

void IntoReady();

void resetcurrent();

void show\_current();

long mkpid();

/\*下面定义一些进程的原语\*/

/\*进程原语句主要有：进程的创建、终止、阻塞、唤醒和切换\*/

void Create\_task(task\_struct \*mtsk);

/\*在本系统中，先在mkPCB中将task\_struct

\*空间分配好，值都设好，返回给Create\_task一个指针

\*/

void Destroy\_current\_task();

/\*进程撤销原语\*/

void run\_task(task\_struct mtsk);

/\*show\_task仅仅为展示效果\*/

void reflashX\_2\_4\_6();

void show\_task\_queues();

void show\_queueL1();

void show\_queueL2();

void show\_queueL3();

/\*show\_task这些没有什么技术含量\*/

void sleep\_on(task\_struct \*mtsk\_struct);

void wakeup(task\_struct \*mtsk\_struct);

void schedule();/\*进程调度，将在时钟中断中调用\*/

void MLFQ();/\*Multi-Level Feedback Queue\*/

/\*多级反馈队列调度算法,在本系统中，多级反馈综合了时间片轮转

\*和先来先服务算法,上层用时间片轮转，下层用先来先服务

\*/

void RR\_L1();

void RR\_L2();

void RR\_L1L2();/\*Round Robin\*/

/\*时间片轮转算法,在本系统中，第一级第二级队列用时间片轮转\*/

void FCFS\_L3();/\*First Come First Served\*/

/\*先来先服务算法,在本系统中，最后一级第三级用先来先服务算法\*/

/\*tss32\_struct 32位保护模式，long 32位，4字节

\*tss32\_struct 共 26个数据项，每项4字节，共104字节

\*tss由硬件决定，固定死了是104字节，故它的范围是0～103

\*/

/\*struct 的声明 注意在C和C++里不同

\*在C中定义一个结构体类型要用typedef:

\*C中若光有 struct 甲，那么在别处用甲还是报错：unknown type name 甲

\*所以要用 typedef 来定义这个类型,来取个别名

\*若没有typedef ,其他地方申明变量时 struct student qiuri;//Struct 不能丢

\*若有typedef struct Student{...}Stu; 可以 Stu qiuri;//省略struct

\*而C++若写 struct Student{}; ，在别处是认的,直接 Student qiuri;（C++可以省略struct ,但c语言不可以）

\*/

//struct tss32\_struct;//NOTE!! 注意在用此结构体定义新的结构体时应该：struct tss32\_struct tss32\_mystruct;

/\*

\* RiOS/kernel/include/asm/x86\_gas.c

\*

\* Copyright (C) 2017 Curie （邱日）

\*/

void gas\_set\_eflags(int eflags)

{

\_\_asm\_\_ volatile("pushfl; popl %0":"=r"(eflags));

}

int gas\_get\_eflags(void)

{

\_\_asm\_\_ volatile("popfl %0; popfl"::"r"(eflags));

return eflags;

}

/\*

\* RiOS/xwindow/xwindow.c

\*

\* (C) 2017 Curie （邱日）

\*/

/\*----------xwindow function-----------------\*/

int \_xwindowRiOSmain()

{//xwindow main,no use under console

Draw();

Sleep(3);

while(1);

return 0;

}//~

void point(int x,int y)

{

unsigned char \*pVRAM=(char\*)0xa0000;//VRAM address=>0xa0000

pVRAM[y\*320+x]=Ubuntu\_RED;

return;

}//~

void rectangle(unsigned char color,int left,int top,int right,int bottom)

{

//left x//top y//right x//bottom y

//const int SCREEN\_XSIZE=320;

unsigned char \*pVRAM=(char\*)0xa0000;

int x, y;

for(x=left;x<right;x++){

for(y=top;y<bottom;y++){

pVRAM[y\*320+x]=color;

//320: 0x13 interrupt:VGA graphic mode=>(320\*20)\*8 color

}

}

return;

}//~

void cls()

{

rectangle(WHITE,0,0,320,200);

};//~

void Draw(void)

{//draw ascii chars

rectangle(LIGHT\_GREEN ,0,0,320,200);

//#include "RiASCII.txt"

int i=0;

int x=0,y=0;

int counter=0;

const int LINEMAX=38;

for(i=32;i<126;i++){

Riputch\_int(i,x,y);

x+=8;

counter++;

if(counter%LINEMAX==0){y+=16;x=0;counter%=LINEMAX;}

}

//#include "RiAlphaNum.txt"

//int i=0;

//for(i=0;i<8;i++)Draw\_en(Rialphabet+16\*i,200+8\*i,100);

return;

}//~

void Draw\_en(char \*en32,int pos\_x,int pos\_y)

{

int x,y;

int i,j;

unsigned char hexi=0x80;

for(i=0;i<16;i++){

hexi=0x80;

for(j=0;j<8;j++){

if ((en32[i] & hexi) != 0)point(pos\_x+j, pos\_y+i);

hexi=hexi>>1;

}

}

}//~

void Draw\_zh(char \*zh32,int pos\_x,int pos\_y)

{

int x,y;

char MyWord1,MyWord2;

int i,j;

unsigned char hexi=0x80;

for(i=0;i<16;i++){

MyWord1=zh32[2\*i];

MyWord2=zh32[2\*i+1];

hexi=0x80;

for(j=0;j<8;j++){

if ((MyWord1 & hexi) != 0)point(pos\_x+j,i+pos\_y);//j is pos\_x

if ((MyWord2 & hexi) != 0)point(pos\_x+j+8,i+pos\_y);

hexi=hexi>>1;

}

}

}//~

void Riputch\_int(int ch,int x,int y)

{

//#include "RiASCII.txt"

int i=0,j=0;//if(\*ch==0)Draw\_en(RiASCII+(i-32)\*16,200,100);

Draw\_en(RiASCII+(ch-32)\*16,x,y);

};//~

/\*

\* RiOS/kernel/include/rios/xwindow.c

\*

\* Copyright (C) 2017 Curie （邱日）

\*/

/\*------我目前暂停对xwindow的开发，专注于字符终端，毕竟我搞的是kernel-----------------------------------

\*如果要xwindow 模式，要把RiOS/kernel/RiOShead.S中 mov $0x03,%al;xor %ah,%ah;int $0x10改为mov $0x13,%al;

\* -----------------xwindow-function declaration-----------------

\*/

int \_xwindowRiOSmain();//xwindow main,no use under 80\*25 TEXT MODE

void point(int x,int y);

void rectangle(unsigned char color,int left,int top,int right,int bottom);

void cls();

void Draw(void);

void Draw\_en(char \*en32,int pos\_x,int pos\_y);//8\*16

void Draw\_zh(char \*zh32,int pos\_x,int pos\_y);//16\*16

void Riputch\_int(int ch,int x,int y);

/\*color constant\*/

const unsigned char BLACK = 0;

const unsigned char LIGHT\_GREEN =10;

const unsigned char DARK\_GRAY =20;

const unsigned char WHITE =30;

const unsigned char RED = 40;

const unsigned char MIDLIGHT\_GREEN = 50;

const unsigned char PINK\_RED =60;

const unsigned char YELLOW\_GREEN = 70;

const unsigned char PURPLE = 80;

const unsigned char ORANGE = 90;

const unsigned char LIGHT\_BLUE = 100;

const unsigned char DARK\_RED = 110;

const unsigned char DARK\_GREEN = 120;

const unsigned char DARK\_PURPLE = 130;

const unsigned char DARK\_ORANGE = 140;

const unsigned char DARK\_BLUE = 150;

const unsigned char DARK\_RED\_GRAY = 160;

const unsigned char DARK\_GREEN\_GRAY = 170;

const unsigned char Ubuntu\_RED =181;