# 启动代码，将其他代码加载到内存

Bootsect.s

1. 被放到Image的最开始位置，该文件编译后不能大于512字节（Image看到软盘，就是在第一个扇区），不足的部分有tools/Build.c程序补0。
2. Bootsect被bios加载到内存的0x7c00处（共512字节），bios将程序跳转到0x7c00处执行，也就是bootsect开始执行
3. Bootsect首先将自己移动到0x90000处
4. Bootsect然后在将Image的512--2560读取到0x9200-0x99c00
5. Bootsect最后读取system到0x10000-0x90000。并且将system从0x10000-0x90000移动到0x00000-0x80000处
6. 将代码跳转到0x9200处（也就是setup.s的开头）

# 设置代码：通过BIOS获取系统参数，保存到内存。并跳转到32位保护模式

setup.s

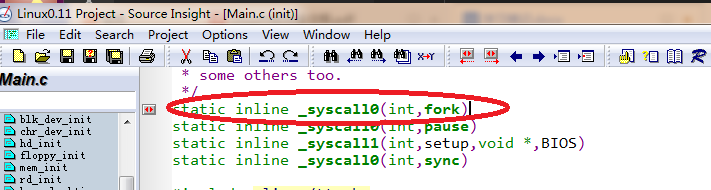
# 头部代码：32位代码，为转c代码做准备

head.s

# Sysem代码：内核代码

## fork()函数的定义

### 函数定义



### 宏展开

\_syscall0是一个宏， 展开后， 就是int fork(){}

#define \_syscall0(type,name) \

type name(void) \

{ \

long \_\_res; \

\_\_asm\_\_ volatile ("int $0x80" \

: "=a" (\_\_res) \

: "0" (\_\_NR\_##name)); \

if (\_\_res >= 0) \

return (type) \_\_res; \

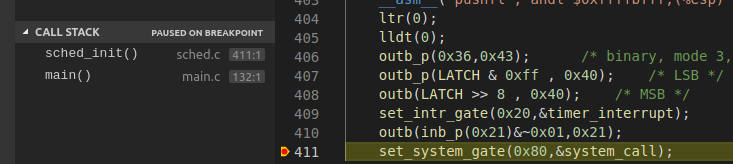
errno = -\_\_res; \

return -1; \

}

设置AX寄存器=\_\_NR\_fork， 然后调用0x80中断

### 0x80中断的响应函数，在下面的代码设置的， 是system\_call



### system\_call的定义

\_system\_call:

//eax如果超出nr\_system\_calls，则没有对应的处理函数，所以直接转到bad\_sys\_cal

cmpl $nr\_system\_calls-1,%eax

ja bad\_sys\_call

//将寄存器压栈保存

push %ds

push %es

push %fs

pushl %edx

//为调用函数\_sys\_call\_table[eax]，准备参数

pushl %ecx # push %ebx,%ecx,%edx as parameters

pushl %ebx # to the system call

movl $0x10,%edx # set up ds,es to kernel space

mov %dx,%ds

mov %dx,%es

movl $0x17,%edx # fs points to local data space

mov %dx,%fs

//调用函数

call \_sys\_call\_table(,%eax,4)

//返回后， 判断进程状态，如果是系统调用还未成功，需要等待，子调用reschedule

pushl %eax

movl \_current,%eax

cmpl $0,state(%eax) # state

jne reschedule

//如果当前进程的时间片已经用完，也调用reschedule

cmpl $0,counter(%eax) # counter

je reschedule

ret\_from\_sys\_call:

movl \_current,%eax # task[0] cannot have signals

cmpl \_task,%eax

je 3f

cmpw $0x0f,CS(%esp) # was old code segment supervisor ?

jne 3f

cmpw $0x17,OLDSS(%esp) # was stack segment = 0x17 ?

jne 3f

movl signal(%eax),%ebx

movl blocked(%eax),%ecx

notl %ecx

andl %ebx,%ecx

bsfl %ecx,%ecx

je 3f

btrl %ecx,%ebx

movl %ebx,signal(%eax)

incl %ecx

pushl %ecx

call \_do\_signal

popl %eax

3: popl %eax

popl %ebx

popl %ecx

popl %edx

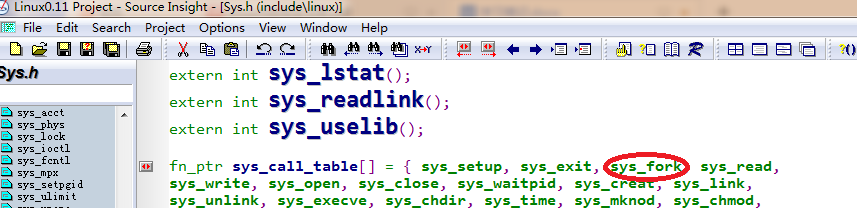
pop %fs

pop %es

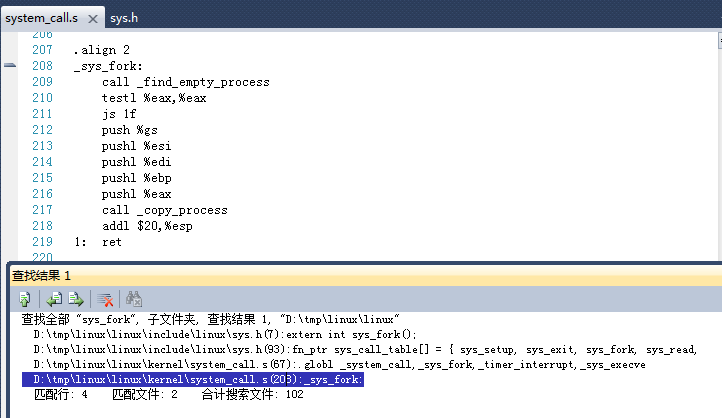
pop %ds

Iret

### \_sys\_call\_table的初始化

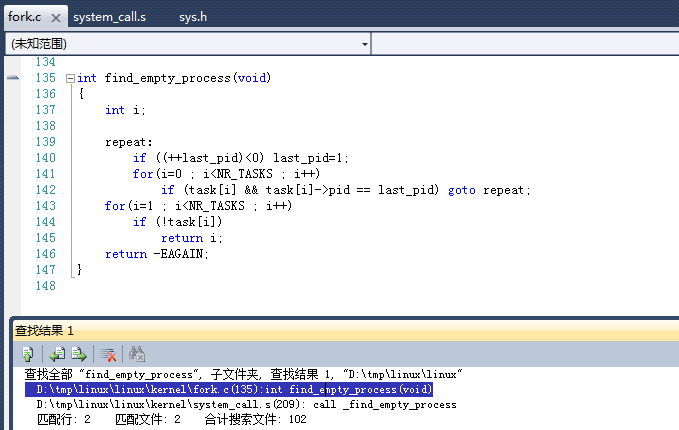


### sys\_fork函数

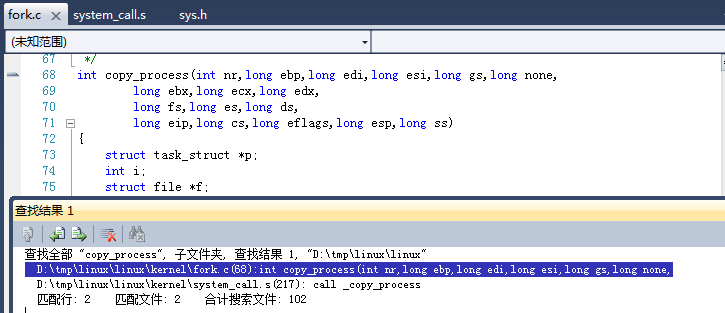


### find\_empty\_process

1. 遍历所有的任务， 找到未用过的最小的进程号（pid），存在last\_pid中
2. 遍历所有的任务， 找到空的任务，返回空任务的index



### copy\_process



1. 找到空闲的内存页get\_free\_page

输入：

%1(ax=0) - 0；

%2(LOW\_MEM)//#define LOW\_MEM 0x100000；

%3(cx=PAGING PAGES)=0xf00；

%4(di=mem\_map+PAGING\_PAGES-1)=0x19350(mem\_map)+0xf00(PAGING\_PAGES)-1=0x1a24f

从edi开始，想后找cx个字段，不等ax的，

返回FFF000， 最后一页可用（将改业标记为占用）

1. 代码

int copy\_process(int nr,long ebp,long edi,long esi,long gs,long none,

long ebx,long ecx,long edx,

long fs,long es,long ds,

long eip,long cs,long eflags,long esp,long ss)

{

struct task\_struct \*p;

int i;

struct file \*f;

//将返回的可用的页，作为task的指针

p = (struct task\_struct \*) get\_free\_page();

if (!p)

return -EAGAIN;

//到全局任务数组中， 记录新任务的指针

task[nr] = p;

//将当父进程任务（当前任务）的属性全部拷贝到新任务

\*p = \*current; /\* NOTE! this doesn't copy the supervisor stack \*/

//设置新任务的与父进程任务（当前任务）不一样的属性

p->state = TASK\_UNINTERRUPTIBLE;

p->pid = last\_pid;

p->father = current->pid;

p->counter = p->priority;

p->signal = 0;

p->alarm = 0;

p->leader = 0; /\* process leadership doesn't inherit \*/

p->utime = p->stime = 0;

p->cutime = p->cstime = 0;

p->start\_time = jiffies;

p->tss.back\_link = 0;

p->tss.esp0 = PAGE\_SIZE + (long) p;

p->tss.ss0 = 0x10;

p->tss.eip = eip;

p->tss.eflags = eflags;

p->tss.eax = 0;

p->tss.ecx = ecx;

p->tss.edx = edx;

p->tss.ebx = ebx;

p->tss.esp = esp;

p->tss.ebp = ebp;

p->tss.esi = esi;

p->tss.edi = edi;

p->tss.es = es & 0xffff;

p->tss.cs = cs & 0xffff;

p->tss.ss = ss & 0xffff;

p->tss.ds = ds & 0xffff;

p->tss.fs = fs & 0xffff;

p->tss.gs = gs & 0xffff;

p->tss.ldt = \_LDT(nr);

p->tss.trace\_bitmap = 0x80000000;

if (last\_task\_used\_math == current)

\_\_asm\_\_("clts ; fnsave %0"::"m" (p->tss.i387));

//复制进程的所有内存

if (copy\_mem(nr,p)) {

task[nr] = NULL;

free\_page((long) p);

return -EAGAIN;

}

//将父进程的文件句柄的引用计数+1

for (i=0; i<NR\_OPEN;i++)

if (f=p->filp[i])

f->f\_count++;

if (current->pwd)

current->pwd->i\_count++;

if (current->root)

current->root->i\_count++;

if (current->executable)

current->executable->i\_count++;

//设置tss，ldt

set\_tss\_desc(gdt+(nr<<1)+FIRST\_TSS\_ENTRY,&(p->tss));

set\_ldt\_desc(gdt+(nr<<1)+FIRST\_LDT\_ENTRY,&(p->ldt));

p->state = TASK\_RUNNING; /\* do this last, just in case \*/

return last\_pid;

}

1. copy\_mem

//计算新任务的代码段，数据段的新的基址

new\_data\_base = new\_code\_base = nr \* 0x4000000; // 新基址=任务号\*64Mb(任务大小)。

p->start\_code = new\_code\_base;

set\_base(p->ldt[1],new\_code\_base); // 设置代码段描述符中基址域。

set\_base(p->ldt[2],new\_data\_base); // 设置数据段描述符中基址域。

//复制元任务的内存中的所有页

if (copy\_page\_tables(old\_data\_base,new\_data\_base,data\_limit)) { // 复制代码和数据段。

free\_page\_tables(new\_data\_base,data\_limit); // 如果出错则释放申请的内存。

return -ENOMEM;

}

1. copy\_page\_tables复制所有页
2. 我页表分配一页空闲的页
3. 将新任务的pg\_dir执行空闲页
4. 遍历当前任务中，的页表的每一页，复制到新任务的页表中

指向相同的页

1. 对于大于1M的内存页，需要到mem\_map中增加引用计数

## schedule函数

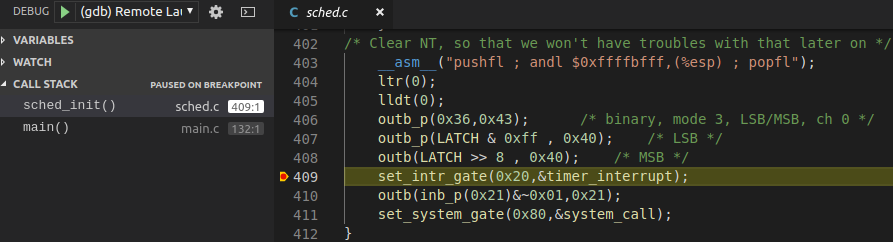
Fork完成后， 系统中就有2个进程。其中0号进程进入无限循环的pause（for(;;) pause();

）. 而新fork出的1号进程，就一直在等待被调度，运行

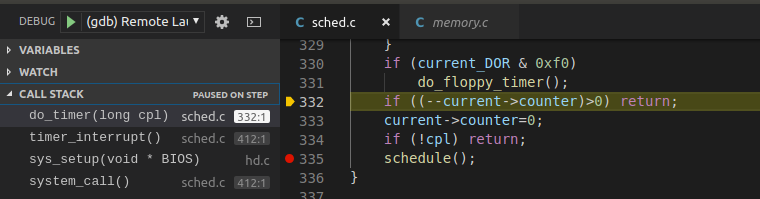
一直等到schedule函数被调用，进程1才被调度运行。

### 注册时钟中断函数

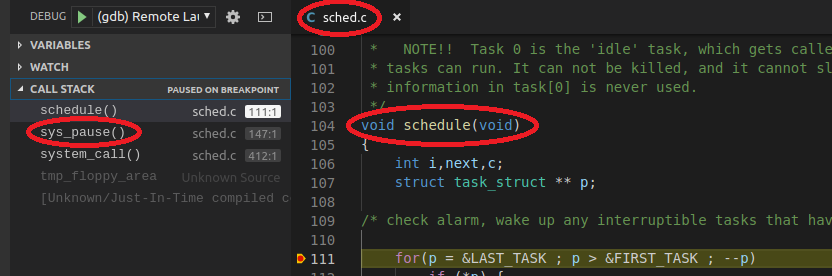
在sched.c的sched\_init函数中，注册0x20号中断的入口函数是timer\_interrupt



### schedule函数被钟中断函数timer\_interrupt调用

、

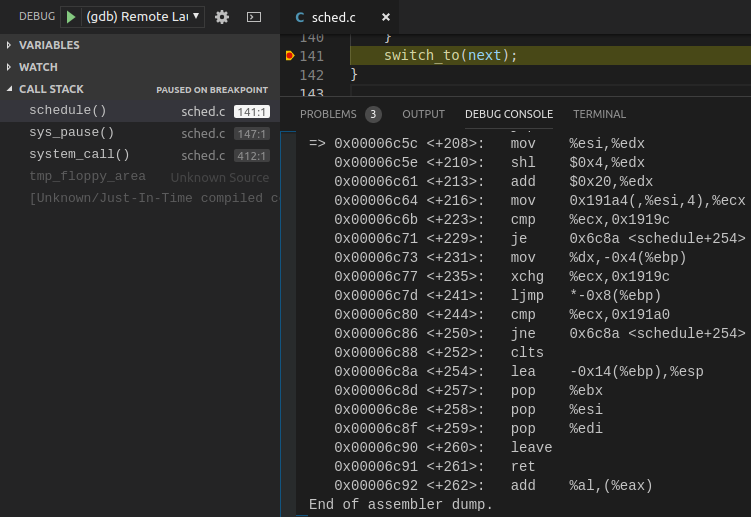
### schedule函数被sleep，wake, pause等函数调用



### schedule函数的处理流程

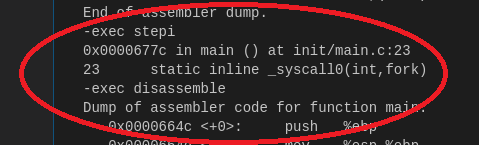
1. 遍历所有的进程任务，看看是否有收到信号量，如果是则将进程的状态从中断状态修改为运行状态
2. 遍历所有的进程任务，寻找时间片计数counter最大的任务。如果所有任务的剩余间片计数counter都小于0，则将所有任务的时间片计数counter都重增加“任务自己的优先级priority”，再遍历进程任务，找到时间片计数counter最大的任务
3. 将时间片计数counter最大的任务，调用起来（switch\_to）

### switch\_to函数



其中0x00006c7D的长跳转， 将会跳转到新任务的上次中断的代码中去， 本任务在这里被中断。

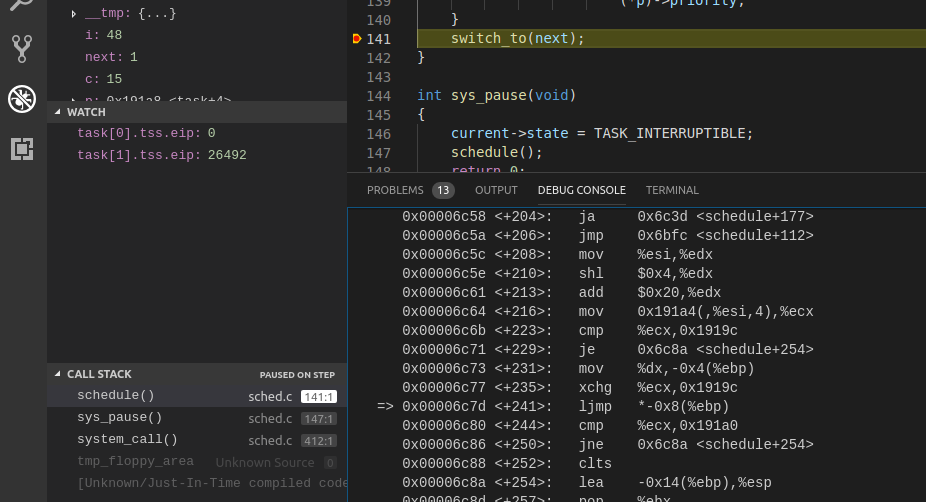
长跳转后， 调到了0x0000677c(task[1].tss.eip)去了



### TSS.eip

1. 在fork时，程序就填写task[1].tss.eip=eip
2. 所以switch\_to，在ljmp做任务跳转时，cpu发现是跳转到tss，就会将老任务的现场记录到TR寄存器指向的tss中，并且从跳转到的TSS中加载场景（各种寄存器），并将TR执行新的tss. 最后执行新的tss的代码。这么多的步骤，都是这一条ljmp完成的。

Ljmp前：



Ljmp之后

