卡通画

描述已自动生成

**操作系统实验报告**

**实验题目** 地址映射与内存共享

**学生姓名**  侯腾跃

**学 号**  202221747

**专业班级**  计科22-4班

**指导教师**  田卫东

**完成日期**  2024.11.23

**合肥工业大学 计算机与信息学院**

1. **实验目的和任务要求**

深入理解物理内存的分页管理方式。

 深入理解操作系统的段、页式内存管理。包括理解段表、二级页表，以及逻辑地址、线性地址、物理地址的映射过程。

 编程实现段、页式内存管理上的内存共享，从而深入理解操作系统的内存管理。

1. **实验原理**

地址映射与内存共享是操作系统中重要的概念，通过地址映射，操作系统能够将进程的虚拟地址空间映射到物理内存，实现了虚拟地址到物理地址的转换。这通过页表等机制完成，允许多个进程同时运行且彼此隔离。内存共享则使得多个进程可以访问相同的物理内存区域，从而实现数据的共享与通信。这种共享可以通过进程间通信机制来实现，如共享内存、消息队列等。总的来说，地址映射和内存共享为多任务系统提供了灵活性和效率，使得进程之间可以协同工作，共享资源，提高系统的整体性能。

1. **实验内容**

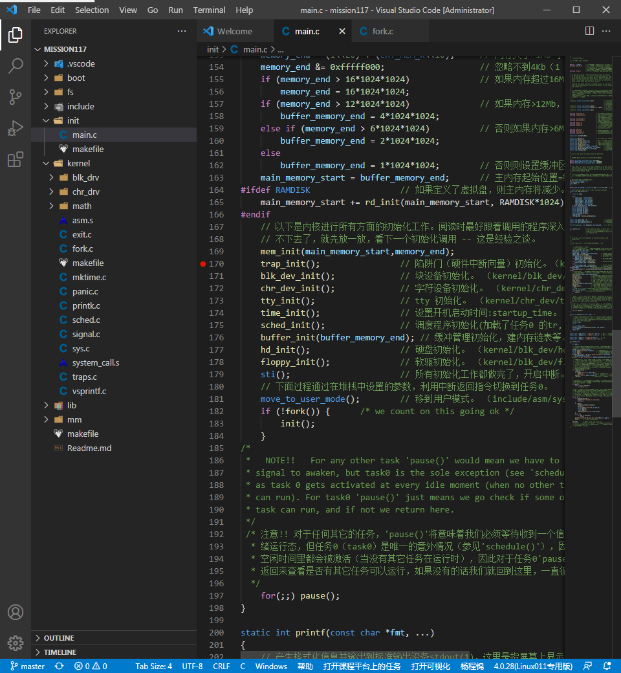
**3.1 分配和释放物理页**

**准备实验**

使用浏览器登录平台领取本次实验对应的任务，从而在平台上创建个人项目（Linux 0.11 内核项目），然后使用VSCode将个人项目克隆到本地磁盘中并打开。

**物理内存的管理**

1. 在init/main.c文件中的第170行添加一个断点，在kernel/fork.c文件中的第99行添加一个断点，按“F5”启动调试。

电脑萤幕画面

描述已自动生成

图 1 添加断点

2. 首先会命中刚刚添加的第一个断点。接下来读者需要按照下面的步骤查看在完成内存初始化后，物理内存的可视化效果。

3. 首先，在VSCode的“View”菜单中选择“Debug Console”，会在VSCode的底部显示出“DEBUG CONSOLE”窗口，在该窗口底部的调试命令编辑器中输入调试命令“-exec call pm()”后按回车。

电脑萤幕画面

描述已自动生成

图 2 添加调试命令

4. 然后，在VSCode的“View”菜单中选择“Command Palette...”，会在VSCode的顶部中间位置显示命令面板，输入“Lab: New Visualizer View”命令后，VSCode会在其右侧弹出一个窗口让读者查看可视化视图。在右侧可视化视图顶部的编辑框中输入命令“#pm”后按回车，可以看到物理内存在刚刚完成初始化后的情况。

5. 此时，物理内存刚刚完成初始化，其中1MB以下的物理内存未参与分页，1M以上的物理内存在完成分页后包含了已使用的物理页和空闲页。此时标记为已使用的物理页是为Linux内核空间保留的物理页，并且已经将这些物理页映射到了逻辑地址空间，确保逻辑地址与物理地址相同，这样Linux内核就可以直接使用物理地址访问这些物理页了，从而为后续Linux内核的初始化过程提供了足够的内存空间以及巨大的便利性。关于Linux是如何将这些物理内存映射到逻辑地址空间的问题，会在本实验后面的练习中进行深入的讨论。

6. 读者也可以将全局变量“mem\_map”添加到“WATCH”窗口查看其内容，如图8-2所示。数组mem\_map的大小是3840，也就是说由mem\_map管理的物理页数量是3840，如果加上不由mem\_map管理的0x100个物理页（即1MB以下的未参与分页的物理内存），则物理页的总数正好是4096，4096个大小为4KB的物理页正好是16MB。

电脑萤幕画面

描述已自动生成

图 3 查看mem\_map

7. 按“F5”继续运行，会在刚刚添加的第二个断点处中断，此时通过执行断点之前的那一行代码，调用get\_free\_page函数第一次申请到了一个空闲的物理页，这一整页物理内存用于存放进程1的进程控制块（关于copy\_process函数已经在实验五中进行了详细的研究，请读者自行回忆相关的内容）。请读者按照前面学习的方法自行操作，在可视化窗口中刷新物理内存的信息，可以看到此时物理页框号最大的那个物理页已经被占用，引用次数变为1。请读者结合get\_free\_page函数的源代码和注释，说明为什么第一次申请的物理页是物理页框号最大的那个。

8. 接下来查看释放物理页的情况。首先删除所有断点，然后在kernel/exit.c文件中的第30行添加一个断点，按“F5”继续调试，会命中刚刚添加的断点，此时将鼠标移动到此行代码中传递给free\_page的参数p上，记录下参数p的值，p指向一个已经结束运行的进程的进程控制块，同样也是一个物理页的基地址，将p的值除以4096就可以得到对应的物理页框号。在可视化窗口中刷新物理内存的信息，可以看到这时p指向的物理页仍然是被占用的状态。按一次“F10”单步调试执行完free\_page函数后，再次在可视化窗口中刷新物理内存的信息，可以看到p指向的物理页已经被释放了，也就是说进程结束后，用于存放其进程控制块信息的那一整页物理内存被系统回收了。

文本

描述已自动生成

图 4 查看参数p

文本

描述已自动生成

图 5 p对应物理页被回收

9. 结束此次调试。

**通过编程的方式练习分配物理页和释放物理页**

1. 打开“学生包”，在本实验对应文件夹下找到mem.c文件，拖动到VSCode中释放，即可打开此文件。将其中的函数physical\_mem复制到Linux 0.11 内核项目下的mm/memory.c文件的末尾处，并且需在include/linux/kernel.h中添加该函数的声明。physical\_mem函数中的代码比较简单，请读者结合其中的注释自行理解。

文本

描述已自动生成

图 6 添加函数声明

2. 添加一个系统调用号为87的系统调用（添加系统调用的方法请参考实验四)，该系统调用的内核函数int dump\_physical\_mem()可以写在 kernel/sys.c 文件的末尾，在此函数中直接调用mm/memory.c文件中的physical\_mem函数即可。

电脑屏幕截图

描述已自动生成

图 7 编写physical\_mem函数

电脑萤幕画面

描述已自动生成文本

描述已自动生成电脑萤幕画面

描述已自动生成

图 8 添加系统调用

3. 生成项目，确保没有语法错误和警告。

4. 按F5启动调试，待Linux 0.11完全启动后，使用vi编辑器新建一个main.c文件，其源代码如下所示：

#define \_\_LIBRARY\_\_

#include <unistd.h>

#define \_\_NR\_dump\_physical\_mem 87

\_syscall0(int, dump\_physical\_mem)

int main()

{

dump\_physical\_mem();

return 0;

}

图形用户界面

描述已自动生成

图 9 编写main.c

5. 保存main.c文件并退出vi编辑器后，依次执行如下命令：

gcc main.c -o mem

sync

mem

应用程序执行后打印输出的物理存储器的信息如图10所示：

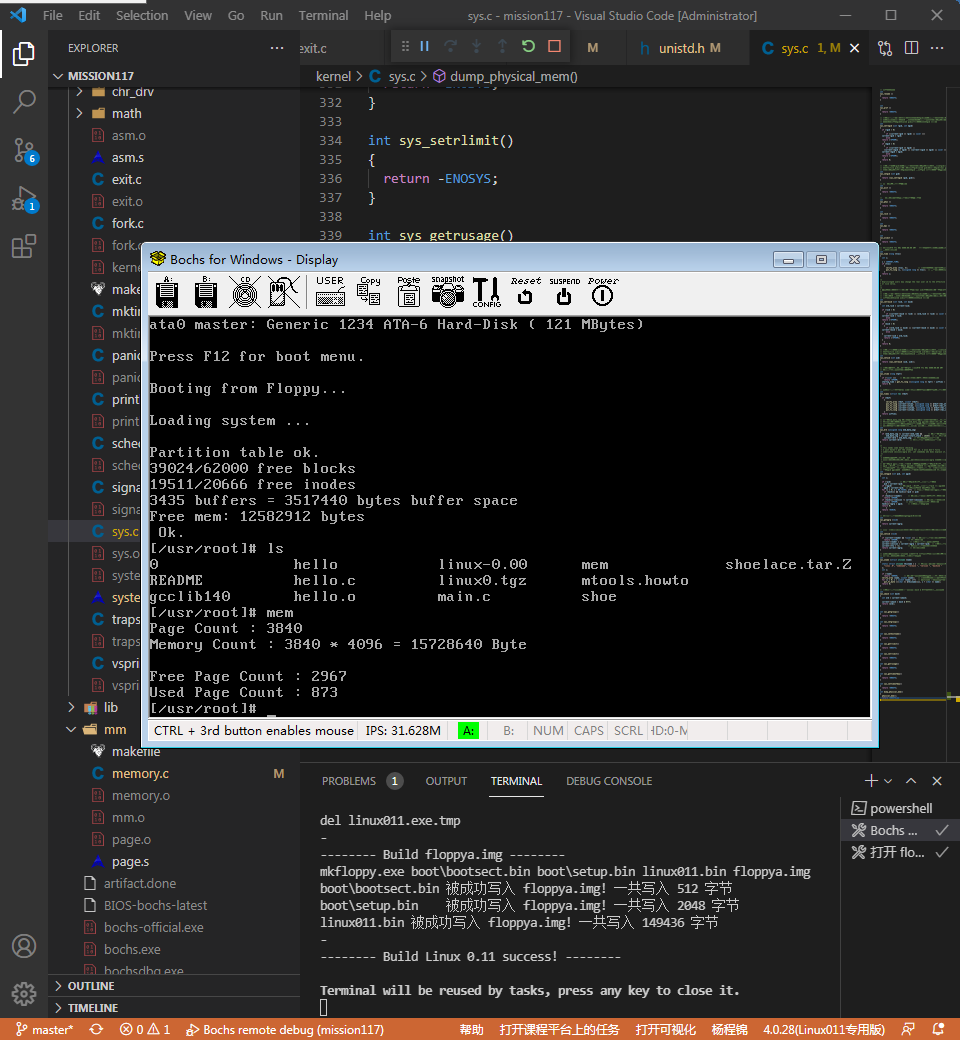


图 10 mem可执行程序结果

对physical\_mem函数进行修改如图11所示。

电脑萤幕的截图

描述已自动生成

图 11 新的physical\_mem函数

输出结果如图12所示。

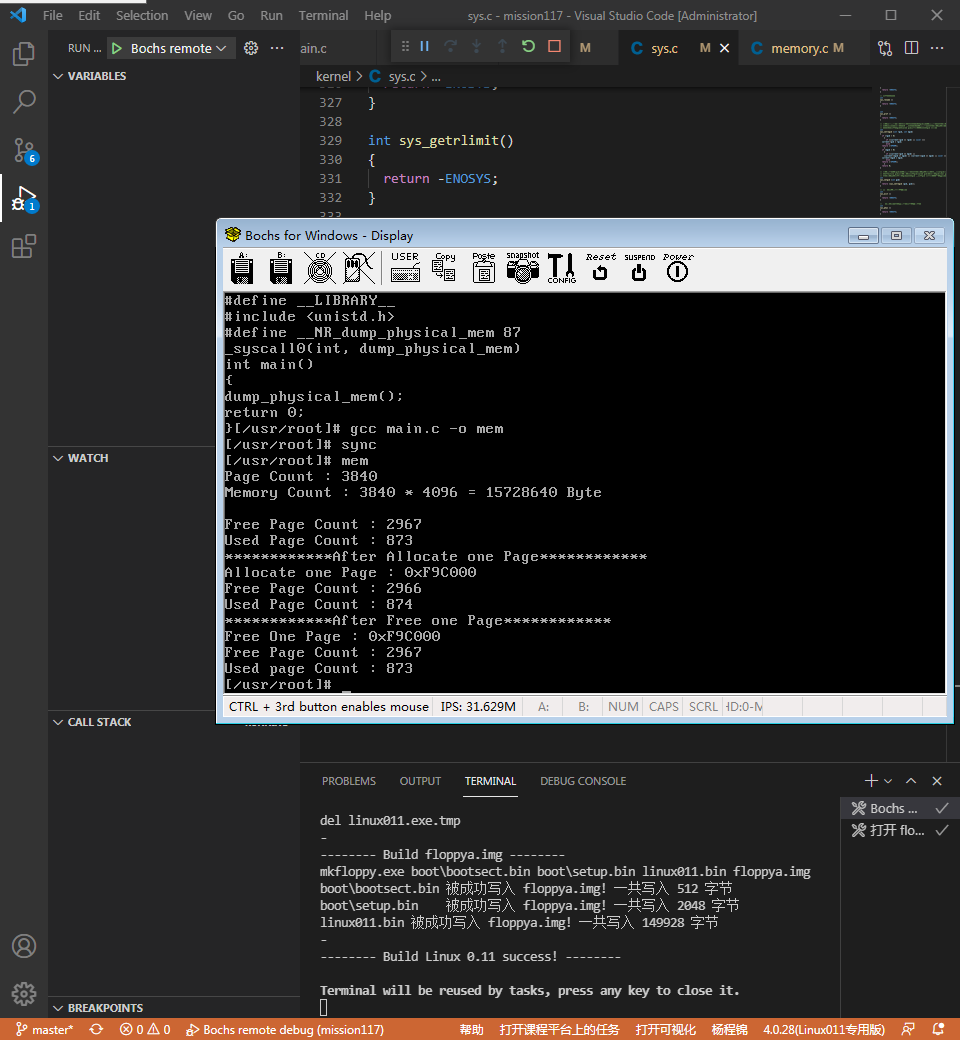


图 12 新的physical\_mem函数输出结果

**3.2跟踪Linux应用程序中的逻辑地址、线性地址、物理地址的映射过程**

**准备实验**

使用浏览器登录平台领取本次实验对应的任务，从而在平台上创建个人项目（Linux 0.11 内核项目），然后使用VSCode将个人项目克隆到本地磁盘中并打开。

**跟踪Linux应用程序中的逻辑地址、线性地址、物理地址的映射过程**

使用Bochs虚拟机提供的Debug调试功能启动Linux 0.11内核项目，然后在Linux终端上运行一个Linux应用程序，此应用程序的main函数中主要包括了一个死循环，源代码如下：

#include <stdio.h>

int i = 0x12345678;

int main(void)

{

printf("The logical/virtual address of i is 0x%08x\n", &i);

fflush(stdout);

while(i)

;

return 0;

}

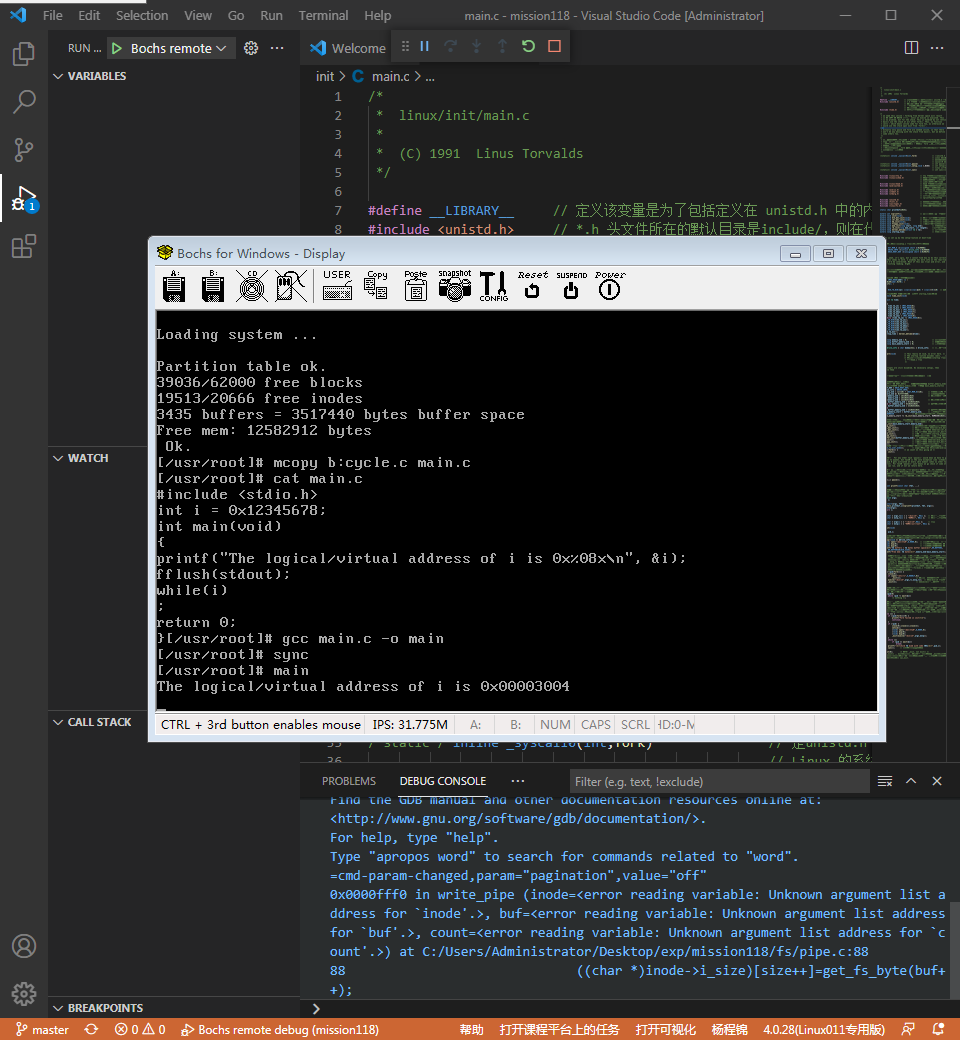
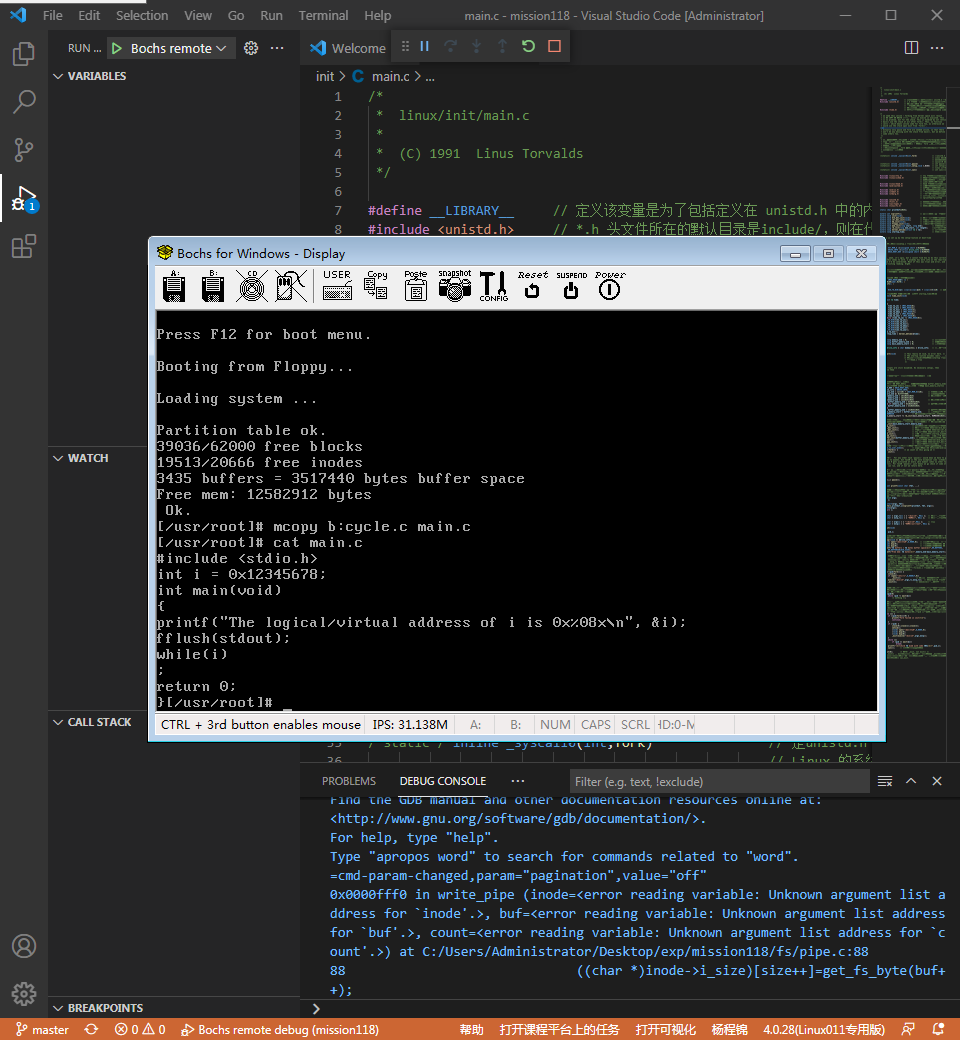


图 13 main可执行文件运行结果

**启动调试Linux内核以及Linux应用程序**

在VSCode的“Terminal”菜单中选择“Run Build Task...”，会在VSCode的顶部中间位置弹出一个可以执行的Task列表，选择其中的“Bochs 命令调试”即可，此时会弹出两个Bochs窗口。标题为“Bochs for windows – Display”的窗口相当于计算机的显示器，显示操作系统的输出。标题为“Bochs for windows – Console”的窗口是Bochs的控制台，用来输入调试命令和输出调试信息。

启动调试后，Bochs在CPU要执行的第一条指令（即BIOS的第一条指令）处中断。此时，Display窗口没有显示任何内容，Console窗口显示将要执行的BIOS的第一条指令，并等待用户输入调试命令，如图14所示。

图形用户界面, 文本

描述已自动生成

图 14 BIOS第一条指令

**调试应用程序并暂停**

接下来请读者按照下面的步骤调试Linux内核以及Linux应用程序，然后使之暂停：

1. 输入命令“c”按回车，Bochs虚拟机会继续运行Linux 0.11操作系统，直到其启动完毕等待用户输入命令。

文本

描述已自动生成

图 15 继续运行

2. 在Linux 0.11中使用vi编辑器新建loop.c文件，并输入之前给出的那个包含死循环的应用程序源代码。

3. 使用GCC将main.c编译为main可执行文件并使用sync保存到硬盘后，运行main，会打印输出如图16所示信息。

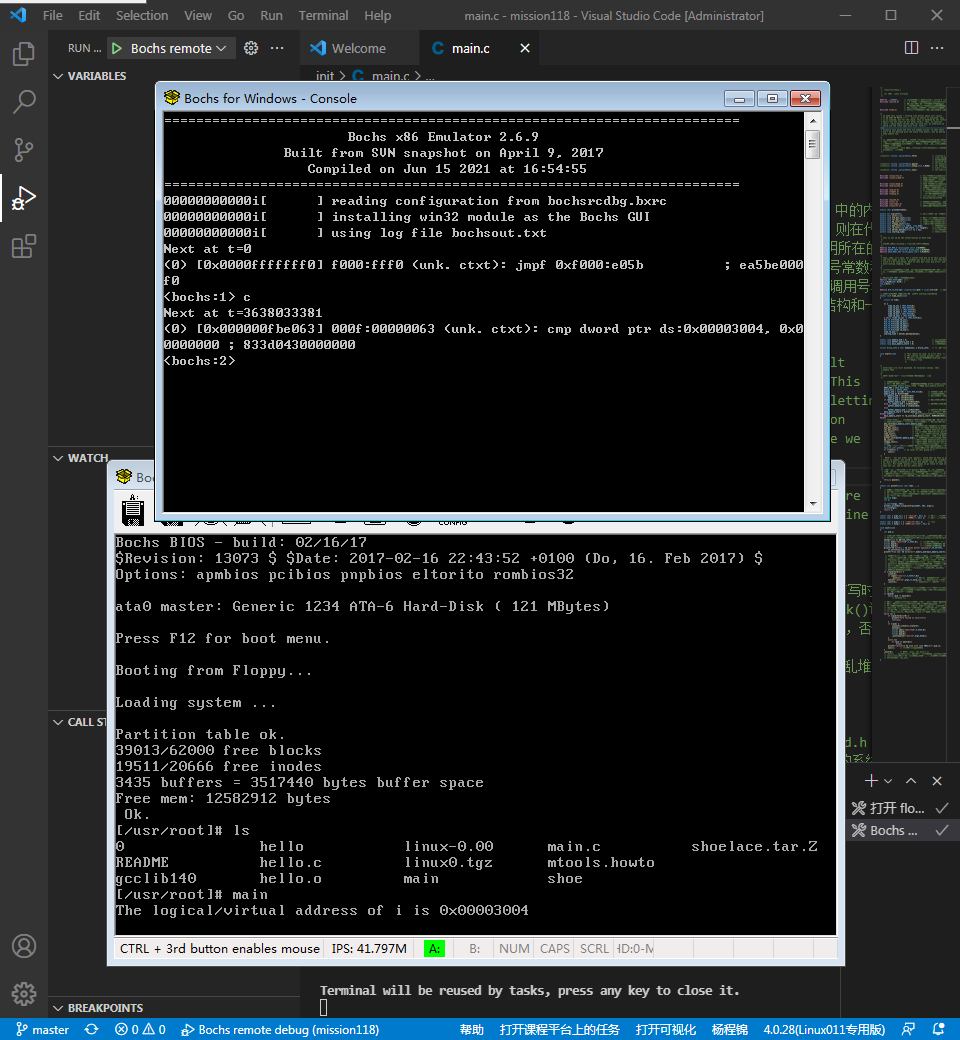


图 16 main输出全局变量i的逻辑地址

只要main应用程序的源代码不发生改变，全局变量i的逻辑地址在任何一个Bochs虚拟机上得到的都是一样的。即使在同一个虚拟机上多次运行，也是一样的。

4. 读者在这里需要先结束此次调试，并关闭Bochs虚拟机。然后重新使用Task列表中的“Bochs 命令调试”启动Bochs虚拟机，待启动Linux 0.11后再运行一次loop应用程序，这样可以保证main应用程序创建的进程PID为6，并且产生的数据与下面描述的一致。

5. 点击Bochs的命令窗口(标题为“Bochs for Windows - Console”)的头部名称位置激活窗口，按“Ctrl+c”，Bochs会暂停运行，等待用户输入调试命令。绝大多数情况下都会在loop应用程序的死循环中暂停，显示如图17所示。

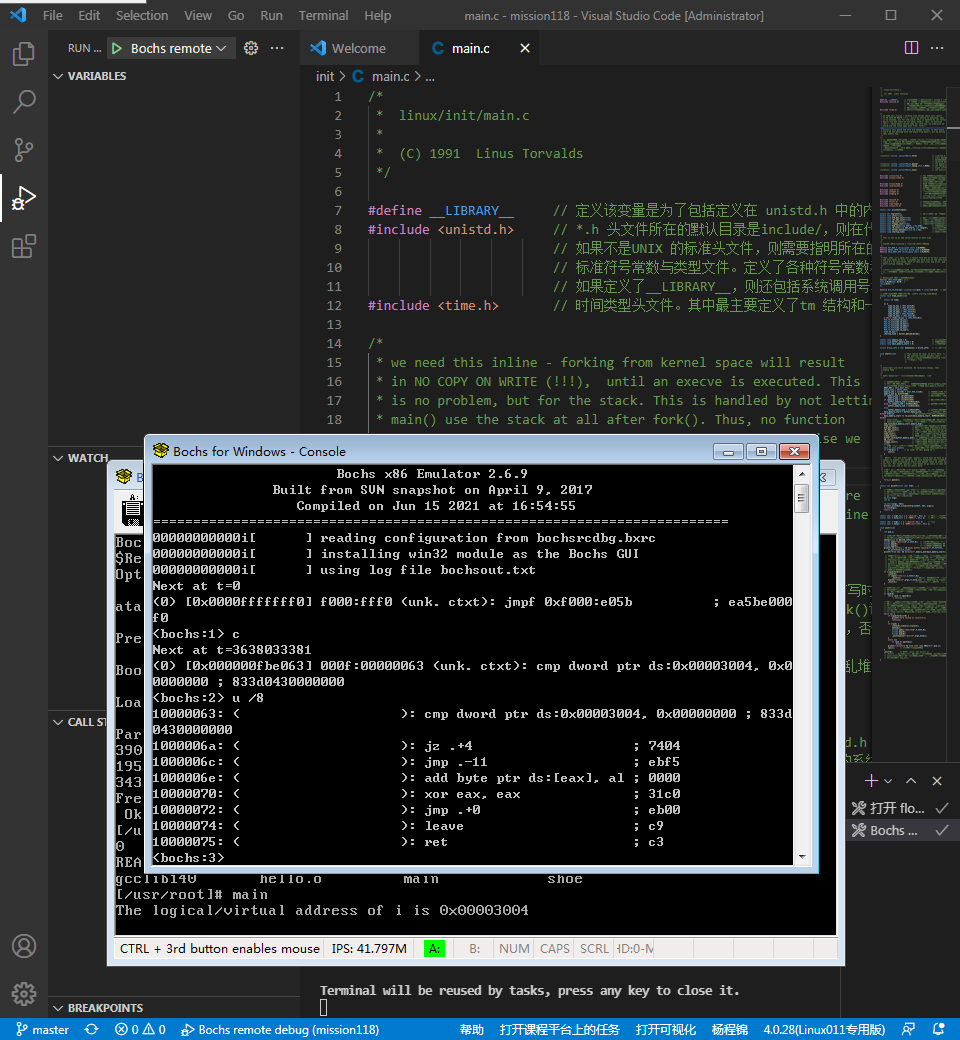


图 17 Bochs暂停后信息

其中冒号左边的“000f”如果是“0008”，则说明是在Linux内核执行时暂停了。那么读者就要在Bochs的命令窗口中输入命令“c”后回车，然后再按“Ctrl+c”，直到其变为“000f”为止，确保是在刚刚编写的应用程序的死循环中暂停。如果显示的将要执行的下一条指令不是“cmp”，就用“n”命令单步运行几步，直到停在“cmp”指令处。

6. 使用命令“u /8”，显示从当前位置开始的8条指令的反汇编代码，如图18所示。

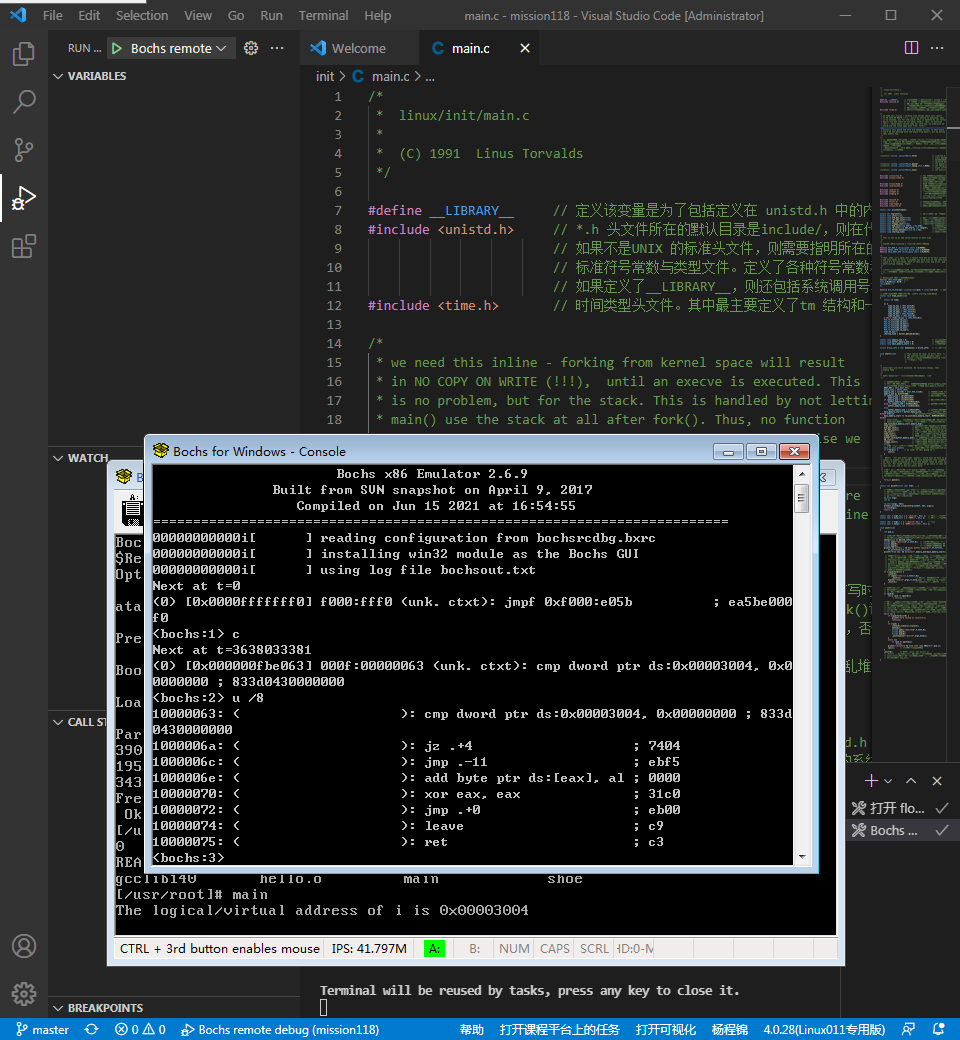


图 18 8条反汇编指令

这些就是main.c中从while语句开始一直到return语句的汇编代码。变量i保存在内存中的ds:0x3004这个地址，并不停地和0进行比较，直到它为0，才会跳出循环。

**处理器通过段表（GDT 和 LDT）将应用程序中的逻辑地址映射为线性地址**

按照下面的步骤进行地址变换：

1. 在Bochs的命令窗口中输入命令“sreg”查看DS寄存器的值，如图19所示。

图形用户界面, 文本

描述已自动生成

图 19 查看DS寄存器值

DS的值为0x0017，换算为16位的二进制为0000000000010111。其中RPL的值为二进制11，十进制为3，是最低的特权级（因为在应用程序中执行）；TI的值为1，表示查找LDT表；使用粗体表示的就是索引值，二进制为10，换算为十进制为2，表示找LDT表中的第3个段描述符（从0开始编号）。

2. 由于LDT表的基址也是由一个段描述符来描述的，而且这个段描述符存储在GDT表中，其索引由ldtr寄存器确定。所以，从上图中也可以得到ldtr的值为0x0068，换算为16位的二进制为0000000001101000，使用粗体表示的就是索引值，二进制为1101，换算为十进制为13，表示LDT表的描述符在GDT的索引为13（第14个描述符）。

3. GDT的起始物理地址存储在寄存器gdtr中，在上图19中显示寄存器gdtr的值为0x00005cb8。所以在Bochs的命令窗口中输入命令“xp /2w 0x00005cb8+13\*8”，就可以查看GDT表中索引值为13的段描述符了，如图20所示。

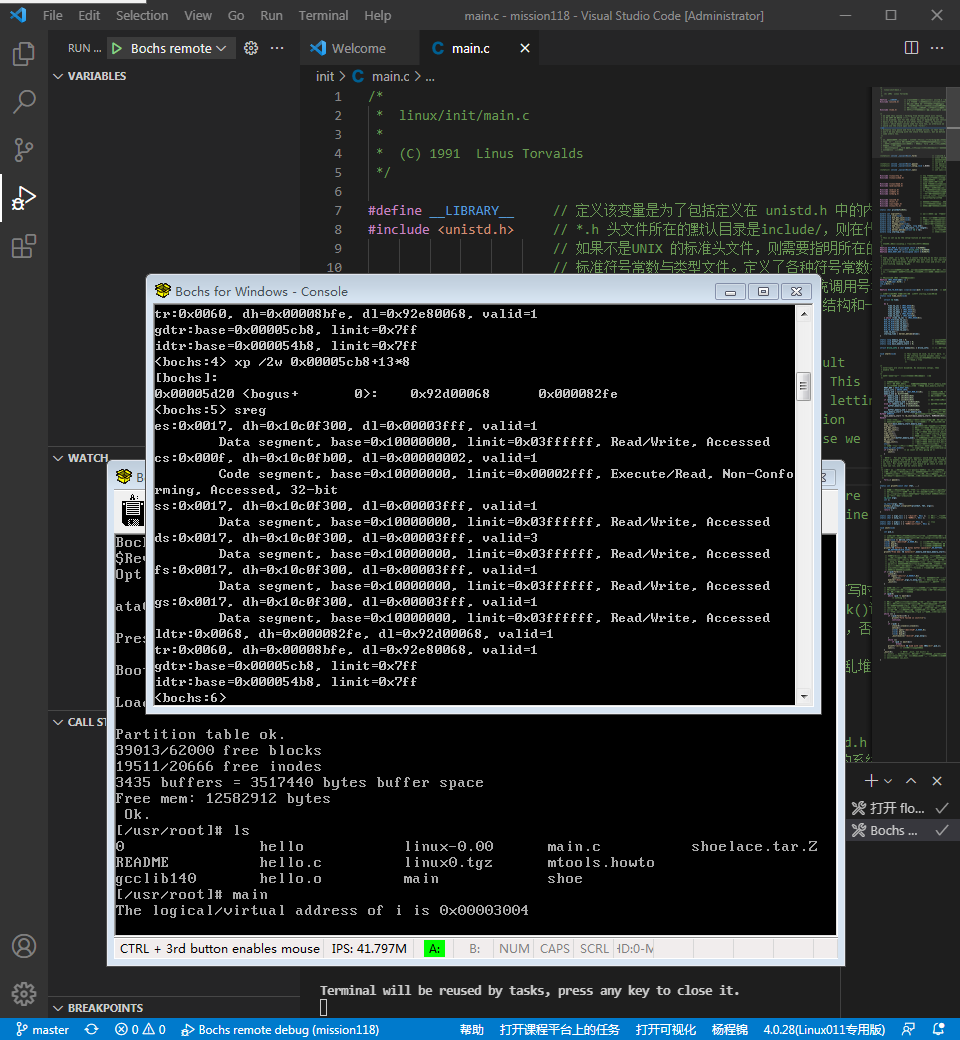


图 20 查看GDT表中索引值为13的段描述符

上面两个命令得到的数值可能和这里给出的示例不一致，这是很正常的。如果读者需要确认自己得到的值是否正确，可以查看“sreg”命令输出内容中的ldtr所在行的dl和dh的值，这两个值是x86处理器为了加快地址映射的速度而缓存的段描述符，读者得到的段描述符必须和它们一致。

4. 将段描述符“0x**92d0**0068 0x**00**0082**fe**”中的加粗部分组合为“0x00fe92d0”，就是LDT表的物理地址了。在Bochs的命令窗口中输入命令“xp /8w 0x00f9d2d0”，可以得到LDT表中前4个段描述符的内容，如图21所示。

图形用户界面, 文本

描述已自动生成

图 21 查看LDT表中前4个段描述符内容

第3个段描述符为“0x00003fff 0x10c0f300”就是ds对应的段描述符了。在“sreg”命令输出的内容中，ds所在行的dl和dh值应该与这个段描述符一致。

段描述符“0x00003fff 0x10c0f300”中加粗部分组合成的“0x10000000”这就是ds段在线性地址空间中的起始地址。用同样的方法也可以计算其它段的基址，都可以得到这个数。接下来使用段基址加上段内偏移就是线性地址了。所以ds:0x3004的线性地址就是0x10000000+0x3004 = 0x10003004。在Bochs的命令窗口中输入命令“calc ds:0x3004”可以查看这个逻辑地址的线性地址，从而可以验证这个结果。

图形用户界面, 文本

描述已自动生成

图 22 验证线性地址

**处理器通过页目录和页表将线性地址映射为物理地址**

按照下面的步骤进行地址变换：

1. 首先需要算出线性地址中的页目录号、页表号和页内偏移，它们分别对应了32位线性地址的高10位+中间10位+低12位，所以0x10003004的页目录号为64，页表号为3，页内偏移为4。

2. 页目录表的起始物理地址由控制寄存器CR3指定。在Bochs的命令窗口中输入命令“creg”可以查看CR3寄存器的值，如图23所示

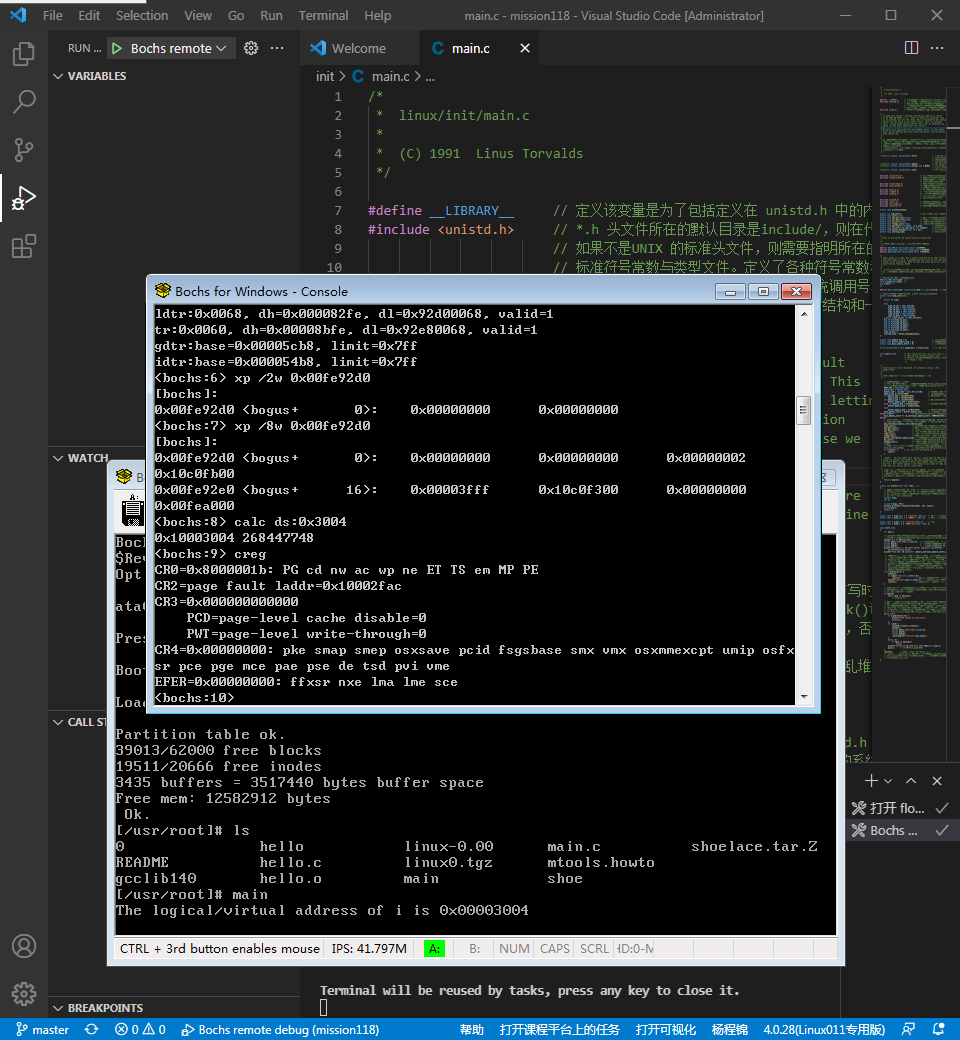


图 23 控制寄存器值

CR3寄存器的值为0，说明页目录的起始物理地址为0。

3. 在Bochs的命令窗口中输入命令“xp /w 0+64\*4”，在页目录中查看页目录号为64的页表项，如图24所示。

文本

描述已自动生成

图 24 页目录号为64的页表项的值

其中027是属性位的值，请读者自己分析这些属性值的意义。页表所在物理页框号为0x00fbd，即页表的起始物理地址为0x00fbd000。

4. 在Bochs的命令窗口中输入命令“xp /w 0x00faa000+3\*4”，在页表中查看页表号为3的页表项，如图25所示。

文本

描述已自动生成

图 25 页表号为3的页表项的值

其中0x00fbb是物理页所在的页框号，所以物理页的起始地址为0x00fbb000。将物理页的起始地址与页内偏移4加在一起得到0x00fbb004就是变量i的物理地址了。

5. 在Bochs的命令窗口中输入命令“xp /w 0x00fa7004”，查看从该物理地址开始的4个字节的值，也就是变量i的值，如图26所示。

文本

描述已自动生成

图 26 变量i的值

可以验证这个值与程序中全局变量i的初值是一样的。

6. 在Bochs的命令窗口中输入命令“setpmem 0x00fbb004 4 0”，将从物理地址0x00fbb004开始的4个字节的值都设为0。然后再使用命令“c”让Bochs继续运行，可以看到应用程序退出了，说明变量i的值在被修改为0后结束了死循环。

文本

描述已自动生成

图 27 程序退出死循环

之前调试了全局变量的地址映射过程，下面请读者自行调试局部变量的地址映射过程。可以将loop应用程序的源代码文件修改为如下的代码，通过物理内存修改局部变量i的值使应用程序结束。

#include <stdio.h>

int main(void)

{

int i = 0x12345678;

printf("The logical/virtual address of i is 0x%08x\n", &i);

fflush(stdout);

while(i)

;

return 0;

}

调试过程下列图所示。

文本

描述已自动生成

图 28 逻辑地址

变量i逻辑地址为：0x03fffef8。

由此判断：页目录：0000001111 页表项：1111111111 页内偏移：111011111000

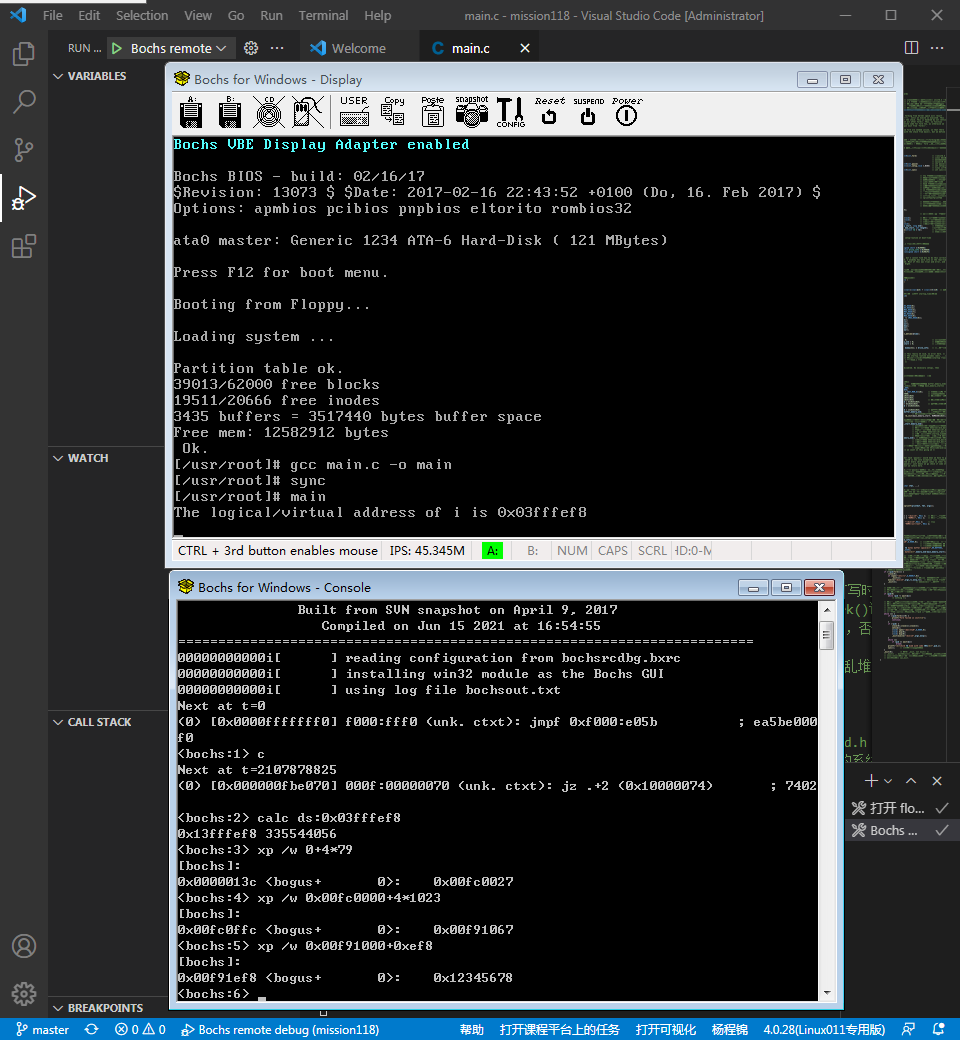


图 29 调试结果

页表起始物理地址：0x00fc0000 物理页所在页框号：0x00f91000 变量i地址：0x00f91ef8

**深入研究Linux 0.11应用程序进程内存的管理方式**

**分段隔离**

1. 按F5启动调试后，使用vi loop.c命令开始编辑loop.c文件，使用下面的源代码修改loop应用程序，并编译生成可执行文件，使之在开始运行的时候就调用fork函数创建一个子进程，这样父进程和子进程就会从同样的位置继续向后并发的运行。

#include <stdio.h>

int i = 0x12345678;

int main(void)

{

fork();

printf("The logical/virtual address of i is 0x%08x\n", &i);

fflush(stdout);

while(i)

;

return 0;

}

2. 使用loop命令运行可执行文件，观察打印输出的信息可以发现父进程和子进程中全局变量i的逻辑地址是相同的，说明父进程和子进程使用了完全相同的逻辑地址空间。

文本

描述已自动生成

图 30 运行loop

3. 结束调试。在kernel/fork.c文件的第171行添加一个条件断点，条件设置为“p->pid == 7”，这样当loop程序创建子进程完毕后就会命中此断点。

文本

描述已自动生成

图 31 添加断点并命中

4. 按F5启动调试，在终端运行loop应用程序后，会命中刚刚添加的条件断点。接下来读者需要按照下面的步骤查看全局描述符表的可视化数据。

5. 首先，在VSCode的“View”菜单中选择“Debug Console”，会在VSCode的底部显示出“DEBUG CONSOLE”窗口，在该窗口底部的调试命令编辑器中输入调试命令“-exec call showgdt()”后按回车。

6. 然后，在VSCode的“View”菜单中选择“Command Palette...”，会在VSCode的顶部中间位置显示命令面板，输入“Lab: New Visualizer View”命令后，VSCode会在其右侧弹出一个窗口让读者查看可视化视图。在右侧可视化视图顶部的编辑框中输入命令“#gdt”后按回车，可以看到全局描述符表的可视化数据。

图形用户界面

描述已自动生成 图形用户界面

描述已自动生成

图 32 全局描述符表

**共享物理页**

1. 仍然使用上面练习中的loop程序和条件断点。

2. 按F5启动调试，在终端运行loop应用程序后，会命中条件断点。接下来读者需要按照下面的步骤查看二级页表的可视化数据。

3. 首先，在VSCode的“View”菜单中选择“Debug Console”，会在VSCode的底部显示出“DEBUG CONSOLE”窗口，在该窗口底部的调试命令编辑器中输入调试命令“-exec call vm2(0x40)”后按回车。

4. 然后，在VSCode的“View”菜单中选择“Command Palette...”，会在VSCode的顶部中间位置显示命令面板，输入“Lab: New Visualizer View”命令后，VSCode会在其右侧弹出一个窗口让读者查看可视化视图。在右侧可视化视图顶部的编辑框中输入命令“#vm2”后按回车，可以看到进程6的完整的二级页表映射关系，其中最左边的是页目录（只显示有效的页目录项），中间的是所有页目录项映射的页表（只显示下标是0x40的页目录项所映射的页表中的有效的页表项，其它的页表中也存在有效的页表项并映射有物理页，但是没有显示出来），最右侧的是一个页表（其页目录项的下标是0x40）映射的所有物理页。记录下最右侧各个物理页的页框号，根据这些物理页的线性地址可以确认这些物理页是进程6所使用的。

图形用户界面

描述已自动生成 图形用户界面, 应用程序, PowerPoint

描述已自动生成

图 33 进程6的完整的二级页表映射关系

5. 使用同样的方法，在“DEBUG CONSOLE”窗口底部的调试命令编辑器中输入调试命令“-exec call vm2(0x50)”后按回车，然后在右侧可视化视图顶部的编辑框中输入命令“#vm2”后按回车，可以看到进程7的完整的二级页表映射关系，根据最右侧物理页的线性地址可以确认这些物理页是进程7所使用的。进程7使用的物理页的页框号与进程6使用的物理页的页框号是完全相同的，说明子进程与父进程共享了相同的物理页，在这些物理页中存储了loop应用程序的指令和数据。

图形用户界面, 文本, 应用程序

描述已自动生成图形用户界面, 应用程序, PowerPoint

描述已自动生成

图 34 loop应用程序的指令和数据

**写时复制(Copy on write)**

1. 在kernel/sched.c文件的第271行添加一个条件断点，条件设置为“task[next]->pid == 7”。

文本

描述已自动生成

图 35 添加条件断点

2. 按F5继续调试，会命中刚刚添加的条件断点。这次命中断点是由于处于就绪状态的子进程刚刚获得处理器，所以此时子进程还没有执行任何指令，当然也就没有访问全局变量i了，所以读者需要再次按F5继续调试，会再次命中刚刚添加的断点。

电脑屏幕截图

描述已自动生成

图 36 命中断点

电脑屏幕截图

描述已自动生成

图 37 命中断点

3. 此时，子进程一定是已经运行过一段时间后才命中的断点。读者使用同样的方法，在“DEBUG CONSOLE”窗口底部的调试命令编辑器中输入调试命令“-exec call vm2(0x50)”后按回车，然后在右侧可视化视图顶部的编辑框中输入命令“#vm2”后按回车，可以看到进程7的完整的二级页表映射关系，使用类似的方法查看进程6的完整的二级页表映射关系，分别查看线性地址0x14004000和0x10004000的物理页的页框号，就会可以发现它们的物理页框号不同了，而且这两个物理页对应的页表项的R/W位已经变为1可写了。

图形用户界面, 应用程序

描述已自动生成图形用户界面, 应用程序, PowerPoint

描述已自动生成

图 38 进程7的完整的二级页表映射关系

**按需加载(Load on demand)**

1. 结束调试，关闭虚拟机。

2. 保留在kernel/sched.c文件的第271行添加的条件断点，删除其它所有断点。

3. 按F5启动调试。在终端运行loop应用程序后会命中条件断点。

4. 在“DEBUG CONSOLE”窗口底部的调试命令编辑器中输入调试命令“-exec call vm2(0x50)”后按回车，然后在右侧可视化视图顶部的编辑框中输入命令“#vm2”后按回车，可以看到进程7的完整的二级页表映射关系，请读者记录下最右侧几个物理页所在的线性地址。这些物理页就是进程7在开始运行之前已经加载到内存中的代码页和数据页。

图形用户界面, 应用程序

描述已自动生成图形用户界面, 应用程序, PowerPoint

描述已自动生成

图 39 进程7的完整的二级页表映射关系

5. 按F5继续运行，会再次命中条件断点。由于此时进程7已经运行过一段时间了，在“DEBUG CONSOLE”窗口底部的调试命令编辑器中输入调试命令“-exec call vm2(0x50)”后按回车，然后在右侧可视化视图顶部的编辑框中输入命令“#vm2”后按回车，可以看到进程7的完整的二级页表映射关系，可以发现有新的物理页被映射到了线性地址空间中，这些物理页就是由于程序运行到了还未加载到内存中的指令，导致缺页异常，从而继续将硬盘上的数据加载到内存中。

图形用户界面

中度可信度描述已自动生成图形用户界面

中度可信度描述已自动生成

图 40 进程7的完整的二级页表映射关系

**内核逻辑地址空间的初始化**

读者可以按照下面的步骤查看Linux 0.11操作系统完成内存初始化后的情况：

1. Linux 0.11启动时，会首先从X86处理器的实模式进入保护模式，然后通过boot/head.s文件中的汇编代码完成保护模式下的初始化工作，即将代码段和数据段的段基址设置为0，将所有物理内存（共16MB）使用二级页表映射到线性地址空间，并且确保内存的物理地址与线性地址一致。最后进入start函数开始进行内核的初始化工作。文件head.s中的源代码请感兴趣的读者自行学习，接下来请读者直接观察一下head.s初始化完毕后的内存情况。

2. 读者可以继续使用之前创建的Linux 0.11内核项目。先删除之前添加的所有断点，然后在init/main.c文件的第135行（start函数开始处）添加一个断点。

3. 按F5启动调试，会在刚刚添加的断点处中断执行。

文本

描述已自动生成

图 41 添加断点并命中

4. 在“DEBUG CONSOLE”窗口底部的调试命令编辑器中输入调试命令“-exec call showgdt()”后按回车，然后在右侧可视化视图顶部的编辑框中输入命令“#gdt”后按回车，可以看到内核代码段描述符和内核数据段描述符的段基址均为0。请读者根据描述符中各个位的值，结合图42说明内核代码段和数据段的段界限大小，以及各个属性位的值分别为多少。



图 42 全局描述表GDT

5. 选择VSCode“View”菜单中的“Run”，在左侧打开调试窗口，在此窗口中展开Registers中的Segs，查看此时CS寄存器和DS寄存器的值。读者结合图43可以看到CS寄存器和DS寄存器中的段选择符分别指向全局描述符表中的代码段描述符和数据段描述符，从而将代码段和数据段的基址都设置为0了。

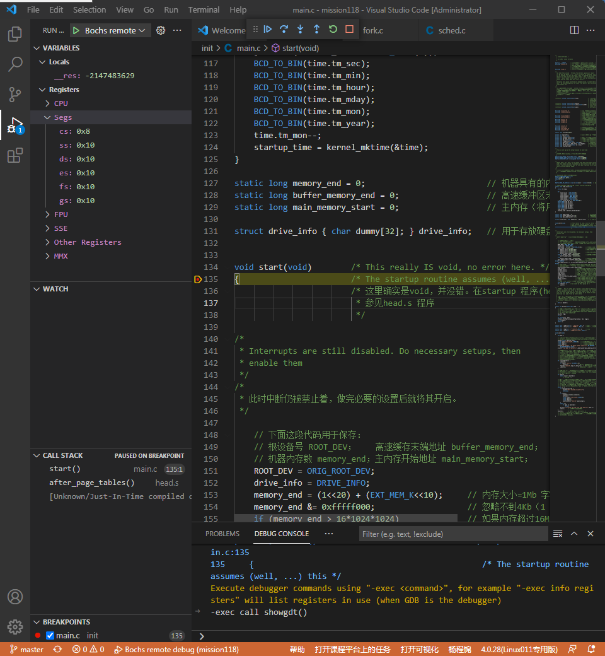


图 43 查看寄存器值

6. 在“DEBUG CONSOLE”窗口底部的调试命令编辑器中输入调试命令“-exec call vm2(0x0)”后按回车，然后在右侧可视化视图顶部的编辑框中输入命令“#vm2”后按回车，可以看到如图44所示的二级页表映射。其中下标是0的页目录项映射的页表中，所有1024个页表项都映射了物理页，并且其映射的任何一个物理页的物理页框号乘以4K后的值（即物理地址）与其所在的线性地址是一致的。请读者继续使用可视化窗口查看其它三个页表中的页表项是否都映射了物理页，从而确认16MB物理内存都映射到了与物理地址相同的线性地址空间。

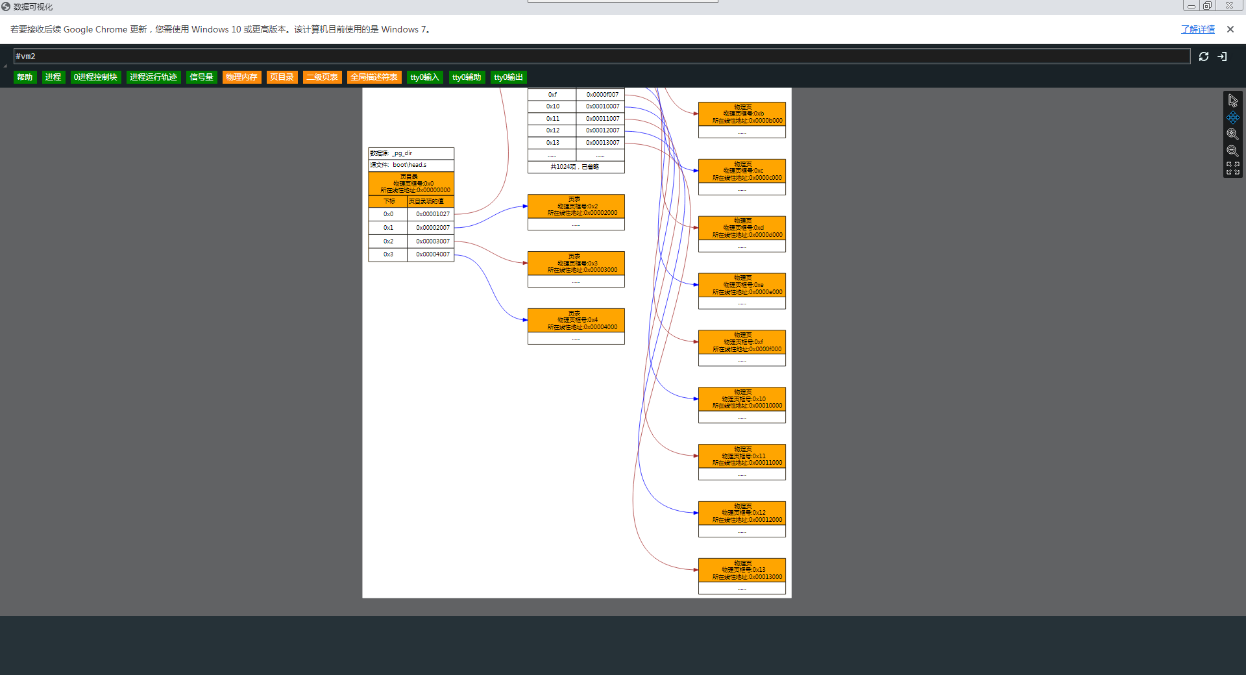
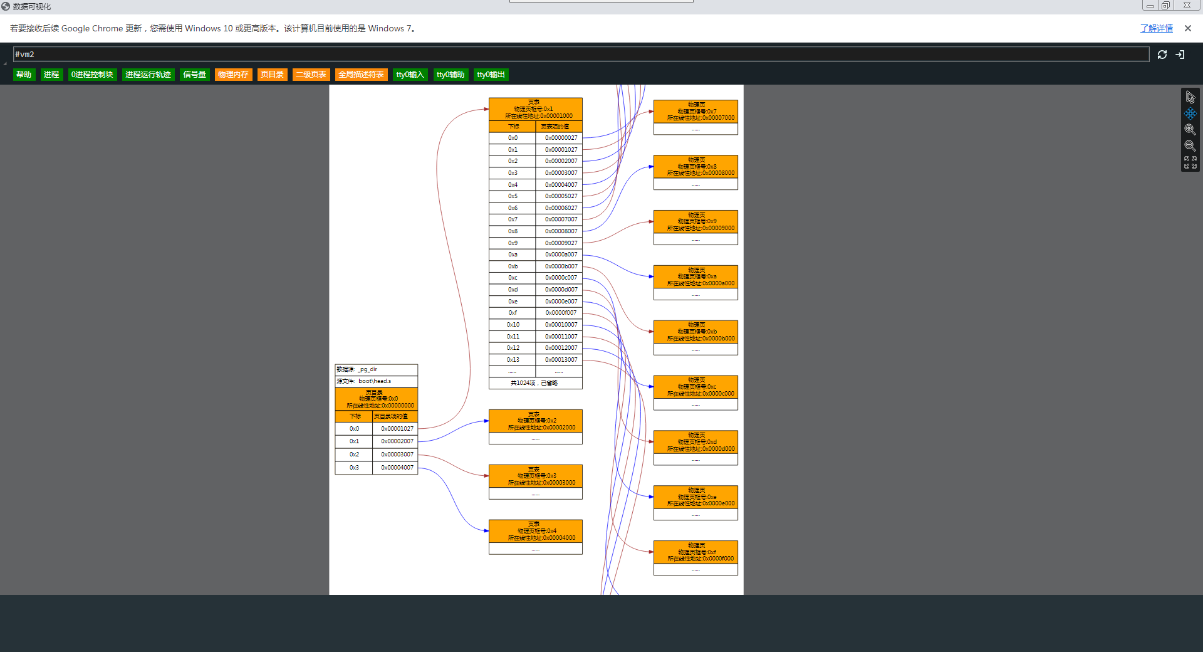
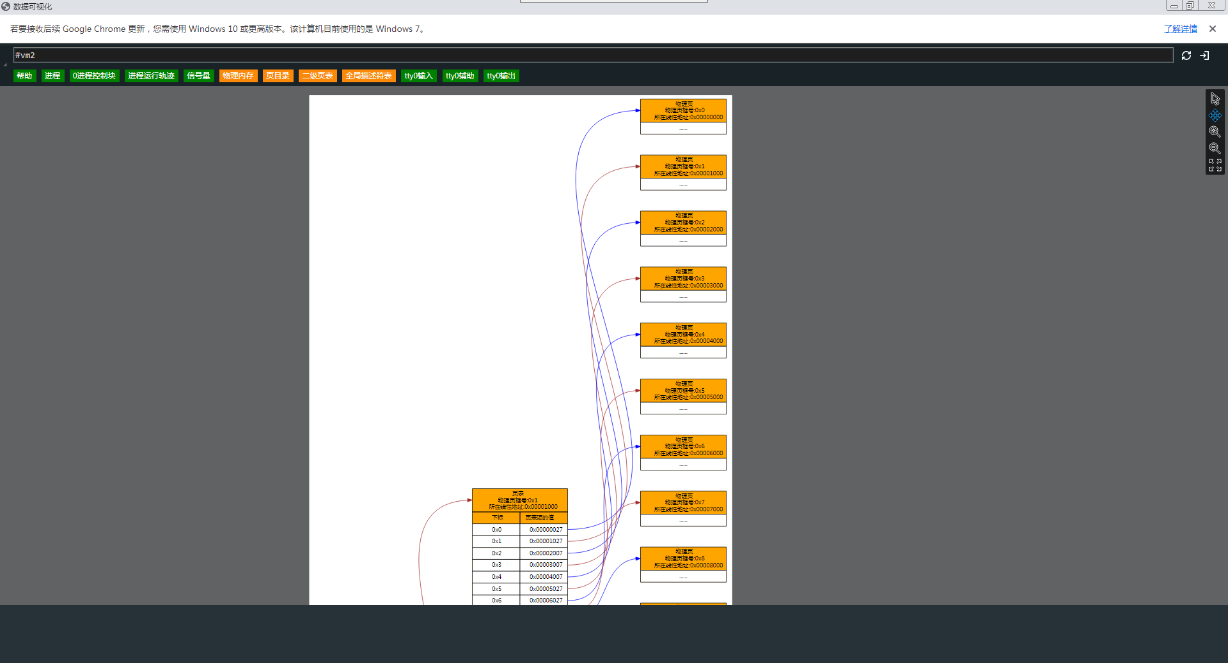


图 44 二级页表映射

**3.3 输出应用程序进程的页目录和页表**

**准备实验**

使用浏览器登录平台领取本次实验对应的任务，从而在平台上创建个人项目（Linux 0.11 内核项目），然后使用VSCode将个人项目克隆到本地磁盘中并打开。

**编写代码输出应用程序进程的页目录和页表**

1. 添加一个系统调用号为87的系统调用（添加系统调用的方法请参考实验四)，该系统调用的内核函数sys\_table\_mapping可以写在kernel/sys.c文件的末尾。

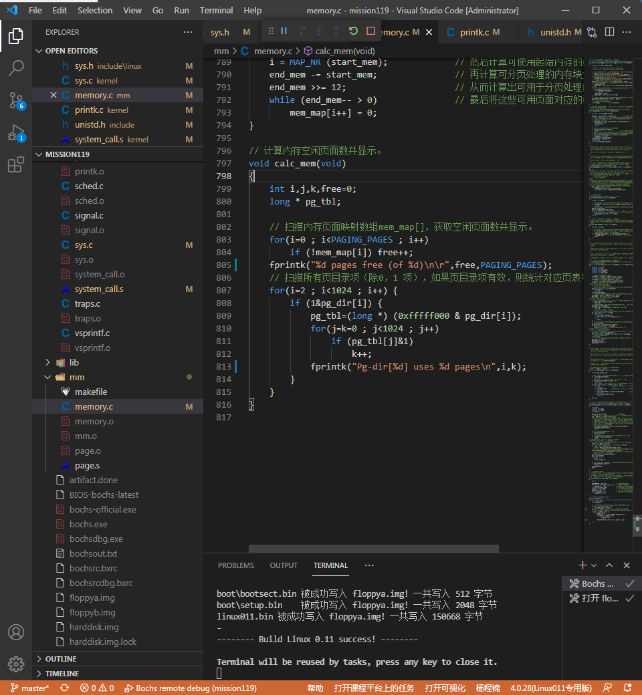


图 45 添加calc\_mem函数

2. 在sys\_table\_mapping函数中调用了一个名称为fprintk的函数用于向标准输出打印信息，而没有使用printk函数，这是由于本实验需要输出的内容较多，需要输出到文件中以便查看，但是printk函数虽然可以在屏幕上进行输出，但是却不能输出到文件中，所以需要实现一个fprintk函数。

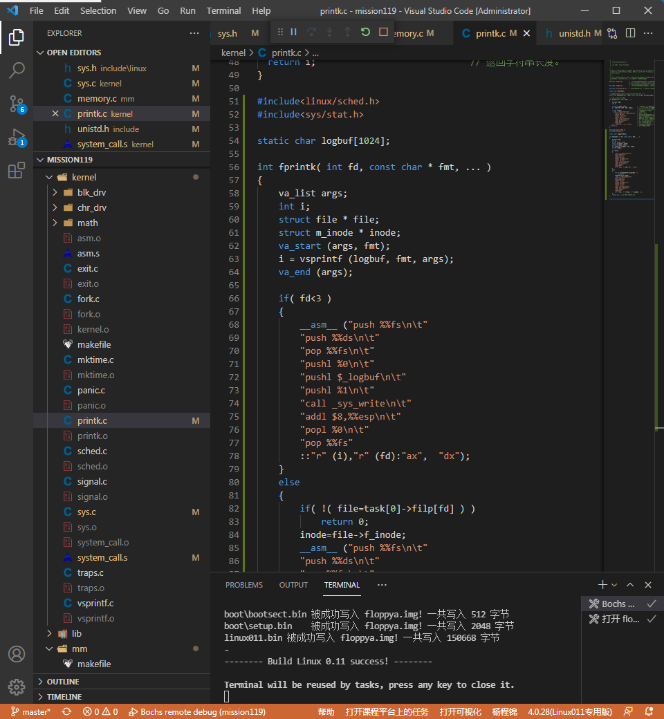


图 46 printk.c中添加fprintk函数

3. 在sys\_table\_mapping函数的开始位置还调用了calc\_mem函数，此函数会计算内存中空闲页面的数量以及各个页表中映射的物理页的数量并显示。在此调用此函数是为了验证sys\_table\_mapping函数输出的页目录和页表的数量是否正确。由于函数calc\_mem中使用的是printk函数，所以还无法将输出内容保存到文件中，所以需要读者将其替换为fprintk函数。

4. 源代码修改完毕后生成项目，确保没有语法错误和警告。

5. 按F5启动调试，待Linux 0.11完全启动后，使用vi编辑器新建一个main.c文件。编辑main.c文件中的源代码如下：

#define \_\_LIBRARY\_\_

#include <unistd.h>

#define \_\_NR\_table\_mapping 87

\_syscall0(int, table\_mapping)

int main()

{

table\_mapping();

return 0;

}

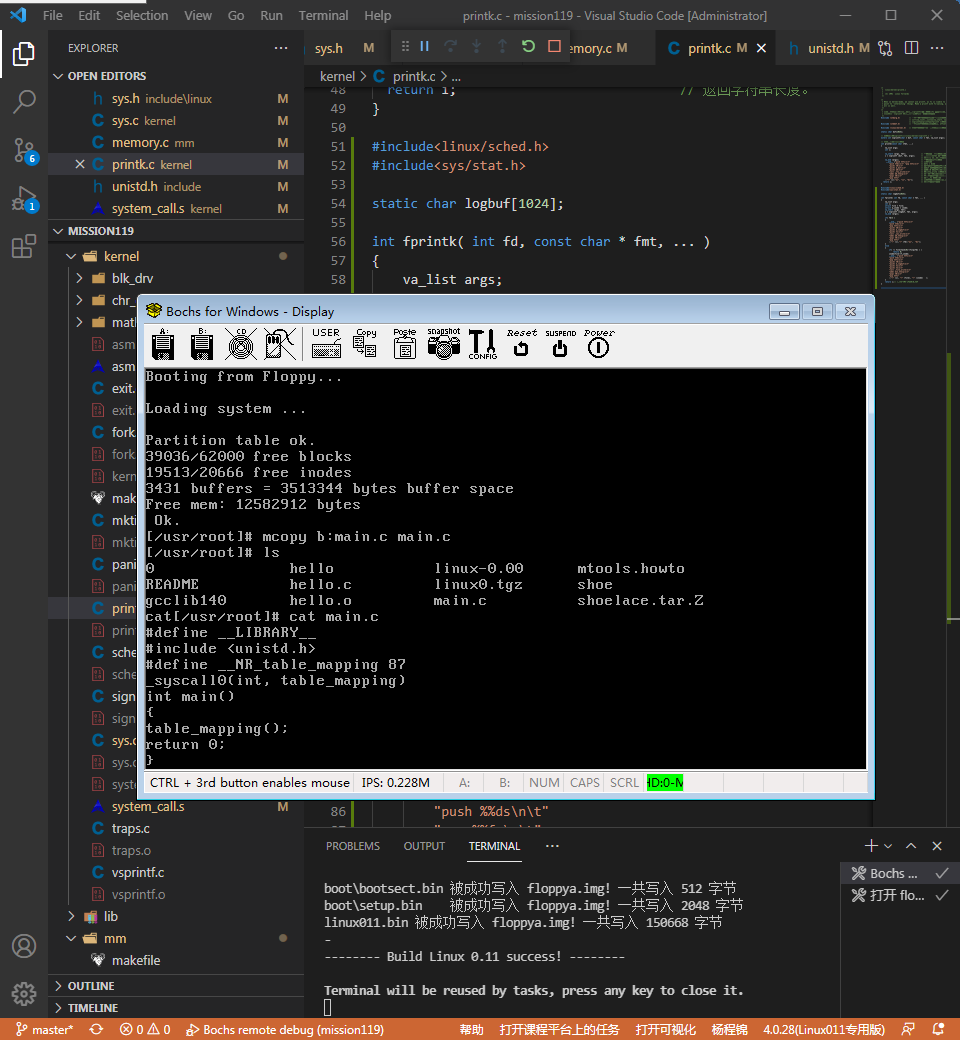


图 47 编写main.c

6. 保存main.c文件后退出vi编辑器，依次执行如下命令：

gcc main.c -o table

sync

table > a.txt

7. 通过命令 vi a.txt 打开vi编辑器查看文本内容，分析输出的结果。

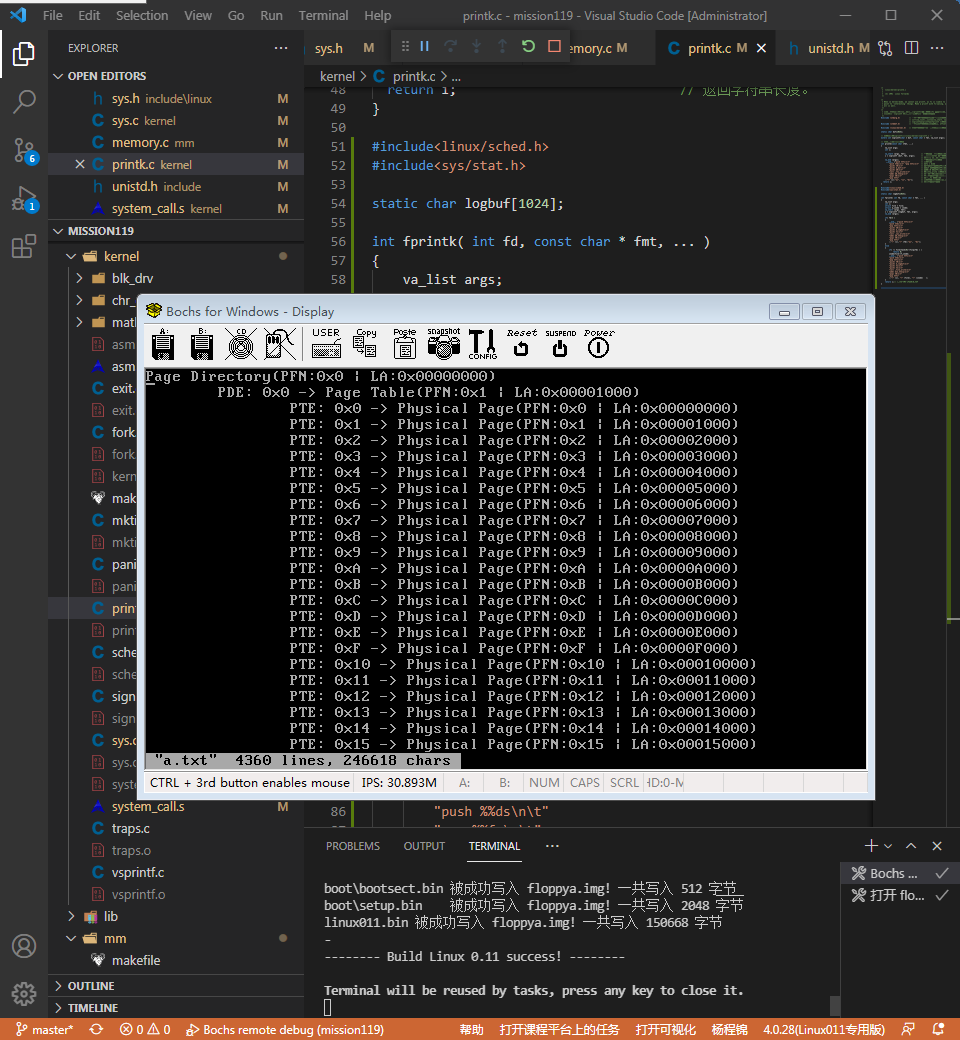


图 48 a.txt输出结果

**3.4用共享内存做缓冲区解决生产者—消费者问题**

**准备实验**

使用浏览器登录平台领取本次实验对应的任务，从而在平台上创建个人项目（Linux 0.11 内核项目），然后使用VSCode将个人项目克隆到本地磁盘中并打开。

**用共享内存做缓冲区解决生产者—消费者问题**

1. 将sem.c文件中的四个信号量的系统调用和两个共享内存的系统调用添加到内核中。



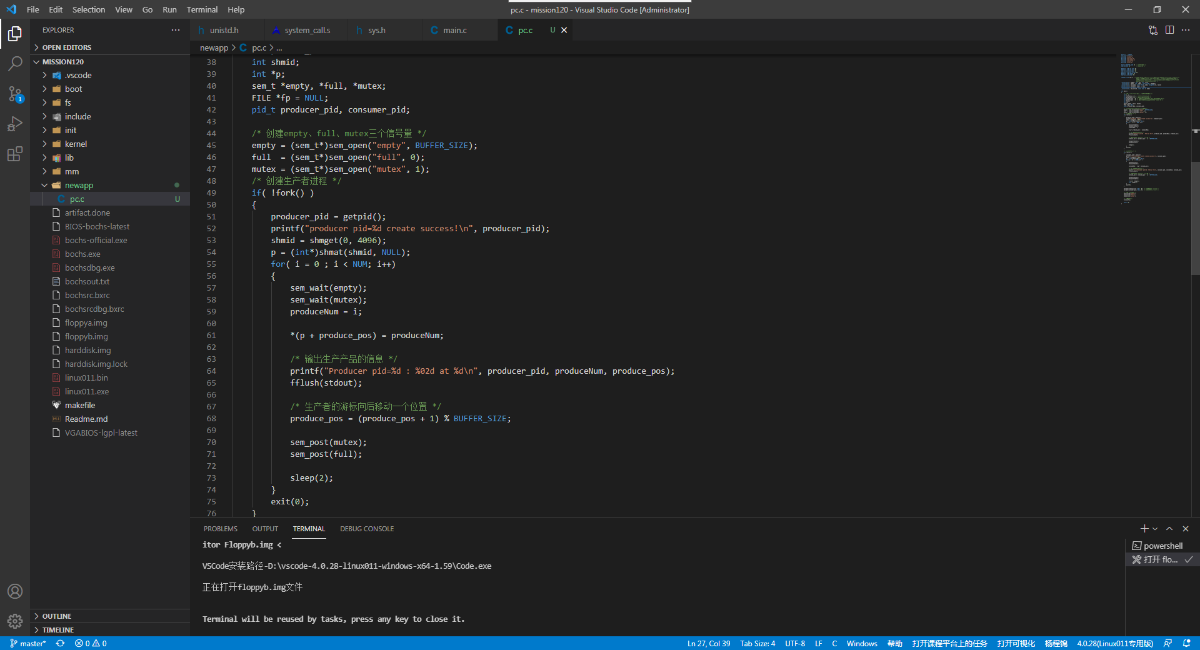
图 49 添加系统调用

2. 生成项目，确保没有语法错误和警告。

3. 将“学生包”本实验文件夹下的pc.c文件放入软盘B中。

4. 按F5启动调试。使用mcopy工具将软盘B中的pc.c文件复制到硬盘的当前目录。

5. 使用命令vi pc.c打开源代码文件，这些源代码仍然是使用文件作为生产者和消费者之间的共享缓冲区，请读者在此基础上将其修改为使用共享内存作为缓冲区。



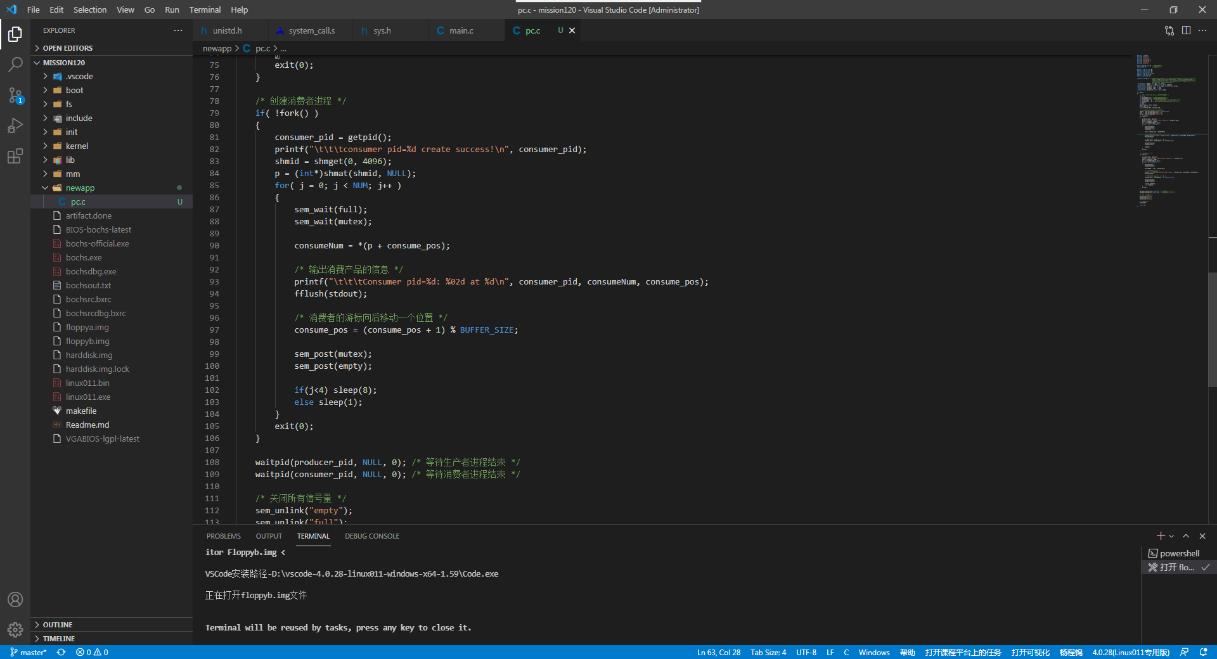


图 50 pc代码修改结果

6. 代码修改完毕后，退出vi编辑器。

7. 在Linux中依次执行下面的命令，运行app：

gcc pc.c -o app

sync

app

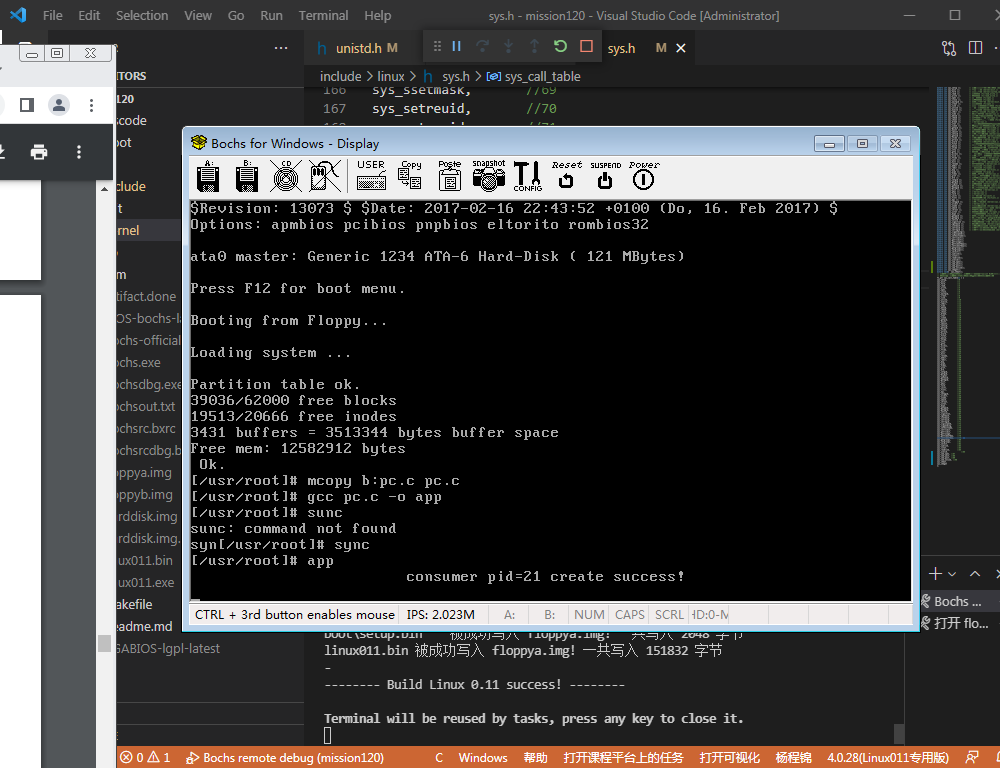


图 51 导入pc并编译

此应用程序执行到最后会出现错误提示“trying to free free page”，说明代码编写成功。

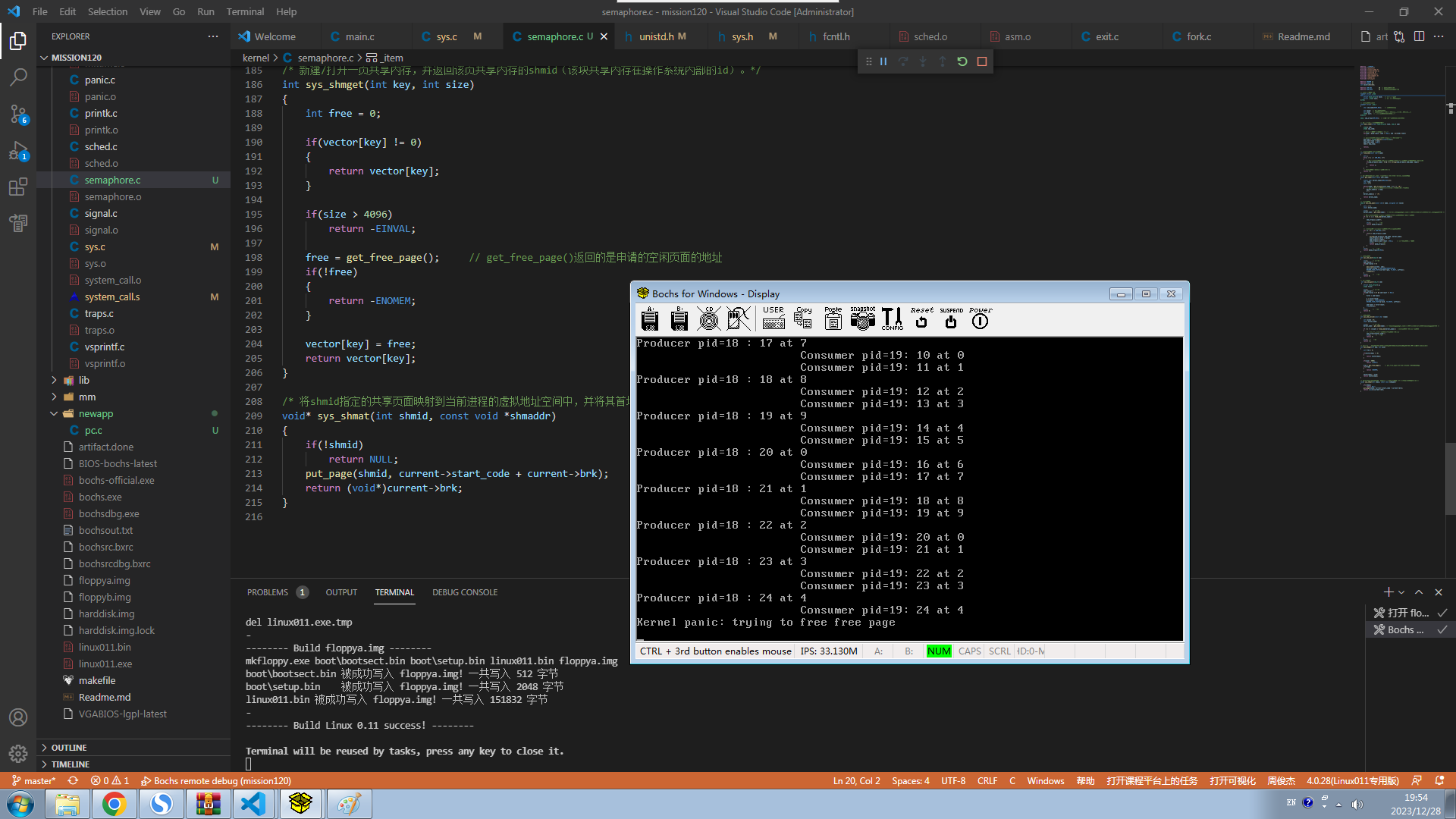


图 52 pc更改成功

1. **实验的思考与问题分析**
2. 答：

（1）逻辑地址到线性地址的映射：

分段： 在分段机制中，逻辑地址由两部分组成，即段选择器和偏移量。首先，通过段选择器在段描述符表中找到对应的段描述符，获得段的基地址和界限。然后，将逻辑地址的偏移量加到段的基地址上，得到线性地址。

分页： 在分页机制中，逻辑地址由两部分组成，即页目录项索引和页内偏移量。通过页目录项索引在页目录表中找到对应的页表基地址，再通过页内偏移量找到对应的物理页帧。最终，将页表基地址和页内偏移量组合成线性地址。

（2）线性地址到物理地址的映射：

分页：在分页机制中，线性地址由两部分组成，即页目录项索引和页内偏移量。通过页目录项索引在页目录表中找到对应的页表基地址，再通过页内偏移量找到对应的物理页帧。最终，将页表基地址和页内偏移量组合成物理地址。

这些步骤在操作系统的内存管理单元（MMU）中完成。分段和分页是常见的地址映射技术，它们可以独立使用，也可以结合使用，形成分段分页的地址映射机制。这样的映射过程可以提供虚拟内存的抽象，使得程序员可以使用逻辑地址而不用关心物理内存的实际位置。这对于实现内存隔离、共享和动态加载等功能是非常重要的。

1. 答：

相同之处：

（1）逻辑地址不变：由于程序代码没有发生改变，变量i的逻辑地址（虚拟地址）在每次运行时应该保持不变。

（2）物理地址不变：由于程序代码和数据存储在相同的物理位置，物理地址在重复运行时也应该保持不变。

不同之处：

（1）实际值可能变化：如果i的值在第一次运行后被修改，那么在第二次运行时，i的实际值可能不同。例如，如果其他程序在第一次运行后修改了i的值，第二次运行时会反映这个新值。

（2）页表和TLB缓存：在第一次运行时，操作系统可能已经将程序的页表加载到内存中，并且页表项被存储在 TLB 缓存中。在第二次运行时，如果页表和 TLB 缓存仍然有效，地址翻译可能更快，而不需要重新加载页表。

总体而言，如果程序代码和数据没有改变，逻辑地址和物理地址应该保持一致。然而，实际值的变化、页表和 TLB 缓存等因素可能导致两次运行时的细微差异。这也强调了在虚拟内存系统中，逻辑地址和物理地址之间的映射是动态变化的，受到多个因素的影响。

1. 答：

在这个实验中，在Linux 0.11内核中添加一个新的系统调用函数 sys\_testg，该函数打印出全局变量 i 的逻辑地址，并进入一个死循环。然后，通过在Linux 0.11的应用程序中调用此函数，并跟踪变量 i 的地址映射过程。

实验步骤

系统调用函数的添加：

在Linux 0.11内核中添加 sys\_testg 函数，该函数打印全局变量 i 的地址并进入死循环。

修改应用程序以调用系统调用：

在应用程序中调用 sys\_testg 函数。

地址映射跟踪：

启动应用程序，并使用Bochs调试工具来跟踪变量 i 的逻辑地址、线性地址、物理地址的映射过程。

修改物理内存中变量 i 的值：

使用Bochs的调试命令修改物理内存中 i 的值为0，从而使应用程序退出死循环。

与第3.2节内容的比较

相同点：

地址映射机制：在两种情况下，地址映射（逻辑地址到线性地址，线性地址到物理地址）的基本过程是相同的。这包括使用段选择符和偏移量确定线性地址，以及使用页表将线性地址映射到物理地址。

调试过程：在两种情况下，都可以使用Bochs工具来跟踪地址映射过程，并通过修改物理内存的值来控制程序流程。

不同点：

内核空间与用户空间：在这个实验中，变量 i 是在内核空间中定义的，而在第3.2节中，变量 i 是在用户空间中的应用程序内定义的。内核空间与用户空间的地址映射可能会有所不同，特别是在权限和访问控制方面。

内存分配策略：内核空间的内存分配策略可能与用户空间不同，这可能影响到变量 i 的物理地址分配。

1. 答：

Linux 0.11 内核的内存布局

在Linux 0.11内核中，内存布局大致分为以下几个部分：

内核空间：内核代码和数据存储在内核空间。这通常位于逻辑地址空间的较低部分。

用户空间：用户程序的代码和数据存储在用户空间。这通常位于逻辑地址空间的较高部分。

全局变量与局部变量的位置差异

全局变量：

存储在内核空间中的固定位置。

通常在内核启动时分配，位于数据段中。

逻辑地址空间中的位置相对固定，不会随函数调用而变化。

局部变量：

存储在函数调用栈上，通常位于内核空间中的栈区。

在函数调用时动态分配，函数返回时释放。

每次函数调用时，局部变量的逻辑地址都可能不同，因为它们取决于栈的当前状态。

1. 答：

定义共享内存结构：定义一个结构来表示共享内存，这应包括缓冲区本身和游标。

struct shared\_memory {

int buffer[BUFFER\_SIZE]; // 缓冲区

int cursor; // 游标

};

初始化共享内存：在内核或应用程序启动时初始化共享内存区域，包括设置游标的初始位置。

生产者逻辑：生产者在共享缓冲区中放置产品，并更新游标。

消费者逻辑：每个消费者读取游标指向的产品，然后更新游标以指向下一个产品。这一步骤需要同步操作，以确保游标的一致性和正确更新。

同步访问：使用锁或其他同步机制来同步对游标的访问。例如，在Linux 0.11中，可以使用信号量或自旋锁来实现同步。

错误处理和边界检查：确保处理缓冲区溢出和游标越界的情况，以及在没有产品可消费时适当地处理消费者的等待状态。

1. 答：

函数原型：

void shmdt(int key, const void\* startaddr);

管理共享内存的引用计数：

定义一个结构体来保存共享内存信息，包括物理页的基址和引用计数。

使用这个结构体定义一个全局共享内存数组。

实现 cancel\_mapping 函数：

函数原型：void cancel\_mapping(const void\* linearaddr);

功能：根据提供的线性地址找到对应的页表项，并将其置为0以取消映射。

在 shmdt 中实现逻辑：

减少共享内存的引用计数。

调用 cancel\_mapping 函数移除共享内存的映射。

转换 startaddr 为线性地址，以便 cancel\_mapping 使用。

如果引用计数大于0，保留物理页；否则，调用 free\_page 释放物理页。

1. 答：

全局描述符表（GDT）

（1）描述符数量：至少需要两个描述符，一个用于代码段（CS），另一个用于数据段（DS）。

（2）描述符值：

描述符由8个字节组成，其中包含段基址、段界限和其他标志（如类型、权限等）。

段基址设定为0，段界限为4GB。

由于x86处理器使用段界限以1或4KB为单位，4GB的段界限需要设置段界限字段为0xFFFFF，并且G位（粒度位）为1，表示界限以4KB为单位。

描述符的类型和权限位需要根据操作系统的要求设置。

示例描述符值（仅为示例，实际值取决于权限和其他标志位）：

段寄存器（CS、DS）

（1）CS和DS的值：这些寄存器应该加载GDT中相应描述符的选择子。

（2）选择子包含描述符在GDT中的索引和其他标志（如特权级）。

（3）假设代码段和数据段分别是GDT中的第一个和第二个描述符，那么CS和DS的值可能是：

CS：0x08（第一个描述符的索引是0，乘以8得到选择子）

DS：0x10（第二个描述符的索引是1，乘以8得到选择子）

二级页表（页目录和页表）

（1）页目录的物理页框号：页目录的起始物理地址（页框号）取决于操作系统如何分配内存。假设从物理地址0开始分配，那么页目录的页框号可能是0。

（2）页目录项的值：由于每个页目录项映射4MB的线性地址空间，4GB的内存需要1024个页目录项。

每个页目录项指向一个页表，页表的物理地址需要填入页目录项。

假设页目录占用一个物理页（4KB），那么紧随其后的页表将开始于物理地址0x1000。

第一个页目录项的值可能是0x00000003（假设页表位于物理地址0x1000，且属性位设置为存在和可写）。

后续页目录项依次增加，每个增加0x004（每个页表占4KB）。

（3）页表项的值：每个页表映射4MB的物理内存，每个页表项映射4KB。

第一个页表映射0到4MB的物理地址，因此其页表项从0开始，每个增加0x001（每个物理页占4KB）。

（4）页目录和页表所占物理页数：

页目录占用1个物理页。

1024个页目录项，每个指向一个页表，每个页表占用1个物理页，共需1024个物理页。

总计：1（页目录）+ 1024（页表）= 1025个物理页。

1. **总结和感想体会**

实验主要围绕Linux 0.11内核的内存管理，特别是地址映射、共享内存和页表映射等核心概念。通过这些实验，我对操作系统的内存管理有了更深的理解。实验涉及到物理内存的分页管理、段页式内存管理、二级页表映射等高级概念，旨在加深对操作系统如何处理和映射内存的理解。通过实验，我更加深入地理解了操作系统的复杂性和优雅。特别是内存管理机制，如何精确地处理每一个字节的存储和访问。总的来说，这些实验是一次宝贵的学习经历，它们不仅加深了我对操作系统内存管理的理解，也为我将来在更高级的系统编程和操作系统研究方面打下了坚实的基础。