**任务十 分配和释放物理页**

**1.实验目的**

(1)深入理解物理内存的分页管理方式。

(2)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。

(3)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。

**2.实验内容**

分配和释放物理页

**3.实验步骤**

**3.1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。

**3.2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。

②首先会命中刚刚添加的第一个断点。

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。

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，如图8-1所示。

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。

⑥将全局变量“mem\_map”添加到“WATCH”窗口查看其内容，如图8-2所示。

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。

⑧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。

⑨结束此次调试。

**3.3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Linux0.11 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。

②添加一个系统调用号为87的系统调用。

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;

}

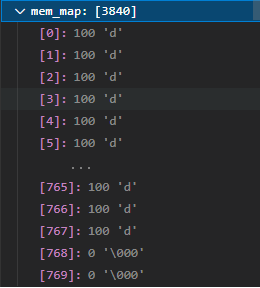
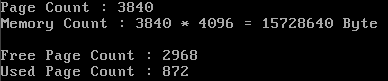
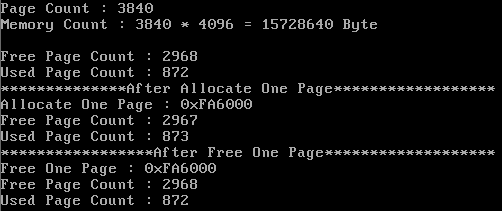
⑤保存main.c文件并退出vi编辑器后，依次执行如下命令：

gccmain.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8-3所示：

⑥****对physical\_mem函数中的源代码进行修改，运行结果如下：

⑦在Linux 0.11的终端使用下面的命令将刚刚编写的文件main.c复制到软盘B中。

mcopymain.c b:main.c

⑧关闭Bochs虚拟机。

⑨在VSCode左侧的“文件资源管理器”窗口顶部点击“New Folder”按钮，新建一个名称为newapp的文件夹。在“文件资源管理器”窗口中的newapp文件夹节点上点击鼠标右键，选择菜单中的“Reveal in File Explorer”，可以使用Windows的资源管理器打开此文件夹所在的位置，双击打开此文件夹。

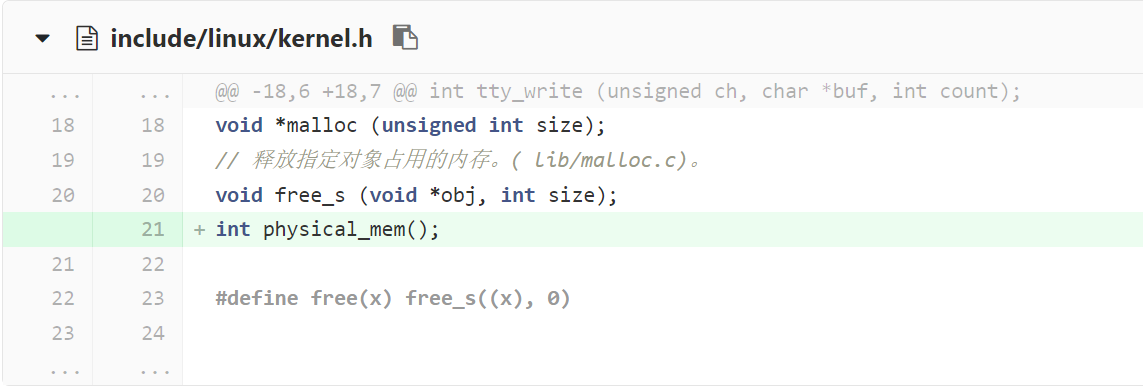
⑩在VSCode的“Terminal”菜单中选择“Run Build Task...”，会在VSCode的顶部中间位置弹出一个可以执行的Task列表，选择其中的“打开 floppyb.img”后会使用Floppy Editor工具打开该项目中的floppyb.img文件，查看软盘镜像中的文件列表，确保刚刚编写的文件已经成功复制到软盘镜像文件中。在文件列表中选中main.c文件，并点击工具栏上的“复制”按钮，然后粘贴到Windows的资源管理器打开的newapp文件夹中。

**3.4 提交作业**

实验结束后先使用VSCode左侧的“源代码版本控制窗口”查看文件变更详情，确认无误后再将本地项目提交到平台的个人项目中。

**4.运行结果**

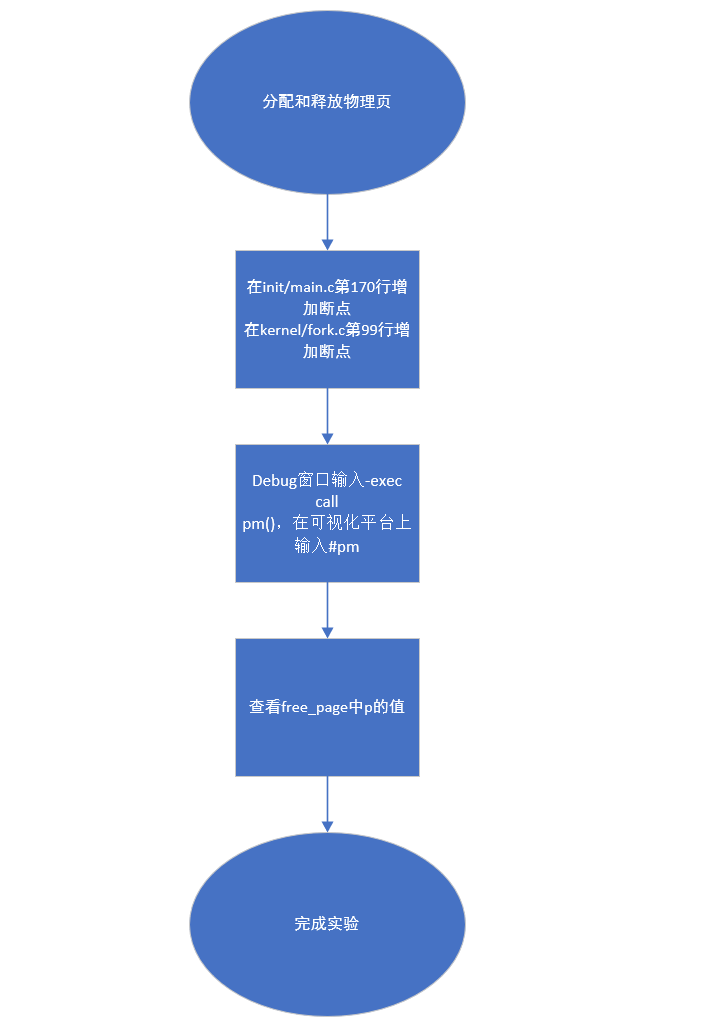
共修改8个文件







**5.流程图**



**6.实验体会**

理解物理内存的分页管理方式和操作系统的段、页式内存管理非常重要。物理内存的分页管理方式是将物理内存划分为若干个大小相等的页框。虚拟内存和物理内存之间的映射关系，是通过页表来完成的。在实现页式内存管理时，需要使用段表和二级页表来完成逻辑地址、线性地址和物理地址之间的映射关系。

**7.思考与练习**

无

**任务十一** **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**

**1.实验目的**

(1)深入理解物理内存的分页管理方式。

(2)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。

(3)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。

**2.实验内容**

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

**3.实验步骤**

**3.1 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。

**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**

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。

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loop.c编译为loop可执行文件并使用sync保存到硬盘后，运行loop，会打印输出如下信息：

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loop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。

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

在读者自行将逻辑地址转换为线性地址之前，需要先掌握段选择符和段描述符的格式。

段选择符

⑦在Bochs的命令窗口中输入命令“sreg”查看DS寄存器的值

⑧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）。

⑨GDT的起始物理地址存储在寄存器gdtr中，在上图中显示寄存器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。

3.3 页目录和页表

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，如下：

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，如下图：

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，如下图：

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，如下图：

⑥在Bochs的命令窗口中输入命令“setpmem 0x00fa7004 4 0”，将从物理地址0x00fa7004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。

**3.4 深入研究Linux 0.11应用程序进程内存的管理方式**

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。

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。

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。

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。

⑦在全局描述符表的可视化数据中可以看到（注意，左侧是全局描述符表GDT，但是为了方便读者查看，在右侧同时显示了全局描述符表项映射的任务状态段TSS和局部描述符表LDT），进程6（父进程）的局部描述符LDT表中代码段描述符和数据段描述符所使用的段基址均为0x10000000，而进程7（子进程）的局部描述符LDT表中代码段描述符和数据段描述符所使用的段基址均为0x14000000，也就是说，虽然父进程和子进程使用了相同的逻辑地址（通过打印输出全局变量i的逻辑地址确定的），但是当两个进程中的逻辑地址转换为线性地址时，还需要再加上各自的段基址，这样父进程和子进程中的相同的逻辑地址就会映射到不同的线性地址空间，从而完成了分段隔离。

**3.5 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。

⑤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。

**3.6 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”。

②按F5继续调试，会命中刚刚添加的条件断点

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。

**3.7 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。

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。

**3.8 提交作业**

当代码编写完成并运行成功后，为了将刚刚编写的文件提交到平台，需要按照下面的步骤将这些文件从Linux 0.11操作系统中复制到软盘B，然后再复制到本地。

①在Linux 0.11的终端使用下面的命令将刚刚编写的文件loop.c复制到软盘B中。

mcopyloop.c b:loop.c

②关闭Bochs虚拟机。

③在VSCode左侧的“文件资源管理器”窗口顶部点击“New Folder”按钮，新建一个名称为newapp的文件夹。在“文件资源管理器”窗口中的newapp文件夹节点上点击鼠标右键，选择菜单中的“Reveal in File Explorer”，可以使用Windows的资源管理器打开此文件夹所在的位置，双击打开此文件夹。

④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“打开 floppyb.img”后会使用Floppy Editor工具打开该项目中的floppyb.img文件，查看软盘镜像中的文件列表，确保刚刚编写的文件已经成功复制到软盘镜像文件中。在文件列表中选中loop.c文件，并点击工具栏上的“复制”按钮，然后粘贴到Windows的资源管理器打开的newapp文件夹中。

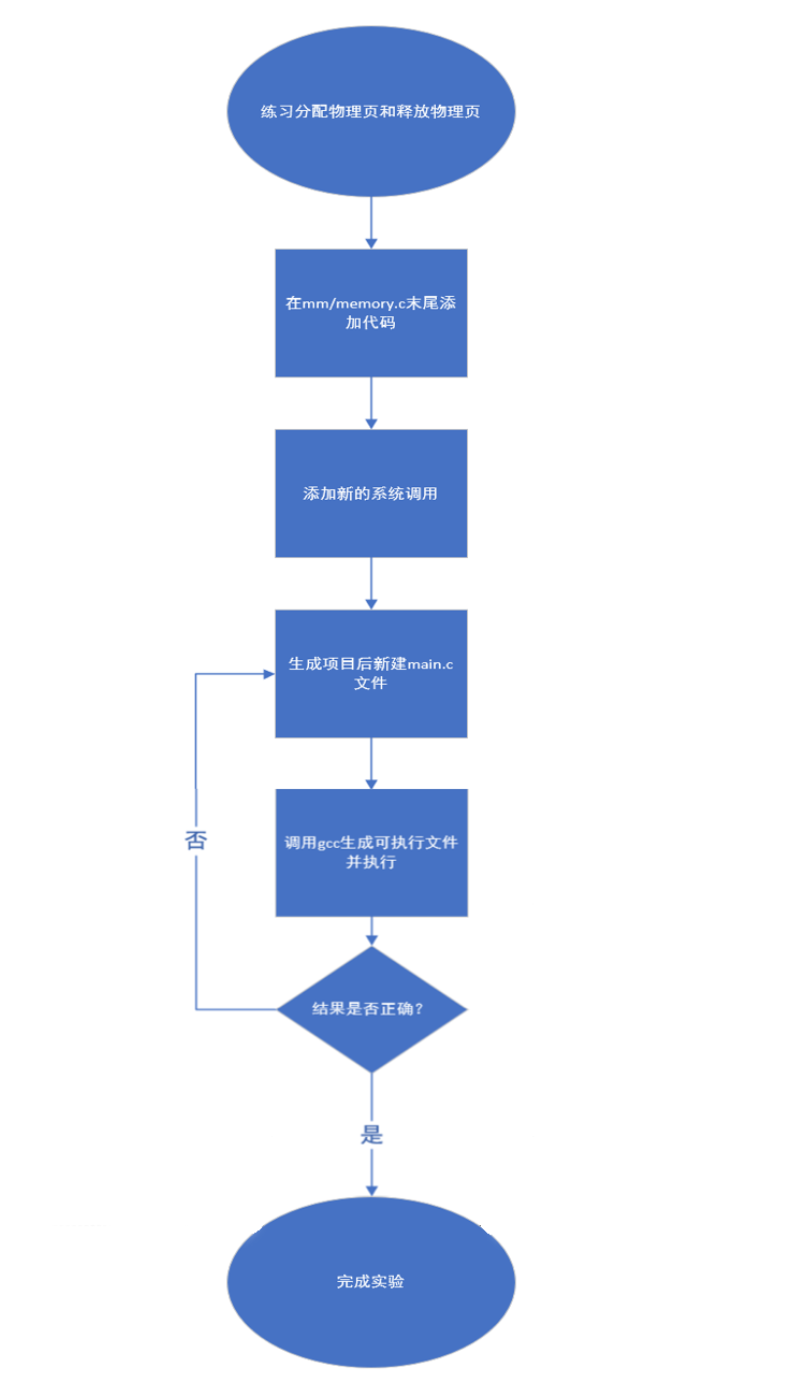
⑤实验结束后先使用VSCode左侧的“源代码版本控制窗口”查看文件变更详情，确认无误后再将本地项目提交到平台的个人项目中。

**4.运行结果**

共修改一个文件



**5.流程图**



**6.实验体会**

在操作系统中，每个进程都有自己的段表和二级页表。逻辑地址、线性地址和物理地址的映射过程，是通过段表和二级页表的查找过程来完成的。内存共享允许多个进程共享同一块物理内存区域。在实现内存共享时，需要特殊处理页表项，以确保多个进程共享同一块物理内存区域时，它们都能够正确地读写该内存区域。

**7.思考与练习**

无

**任务十二 输出应用程序进程的页目录和页表**

**1.实验目的**

(1)深入理解物理内存的分页管理方式。

(2)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。

(3)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。

**2.实验内容**

输出应用程序进程的页目录和页表

**3.实验步骤**

**3.1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。

**3.2 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。其源代码可以参见“学生包”中本实验对应文件夹下的table.c文件。

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，

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。

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Linux011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;

}

⑥保存main.c文件后退出vi编辑器，依次执行如下命令：

gccmain.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。

**3.3 提交作业**

①在Linux 0.11的终端使用下面的命令将刚刚编写的文件main.c复制到软盘B中。

mcopymain.c b:main.c

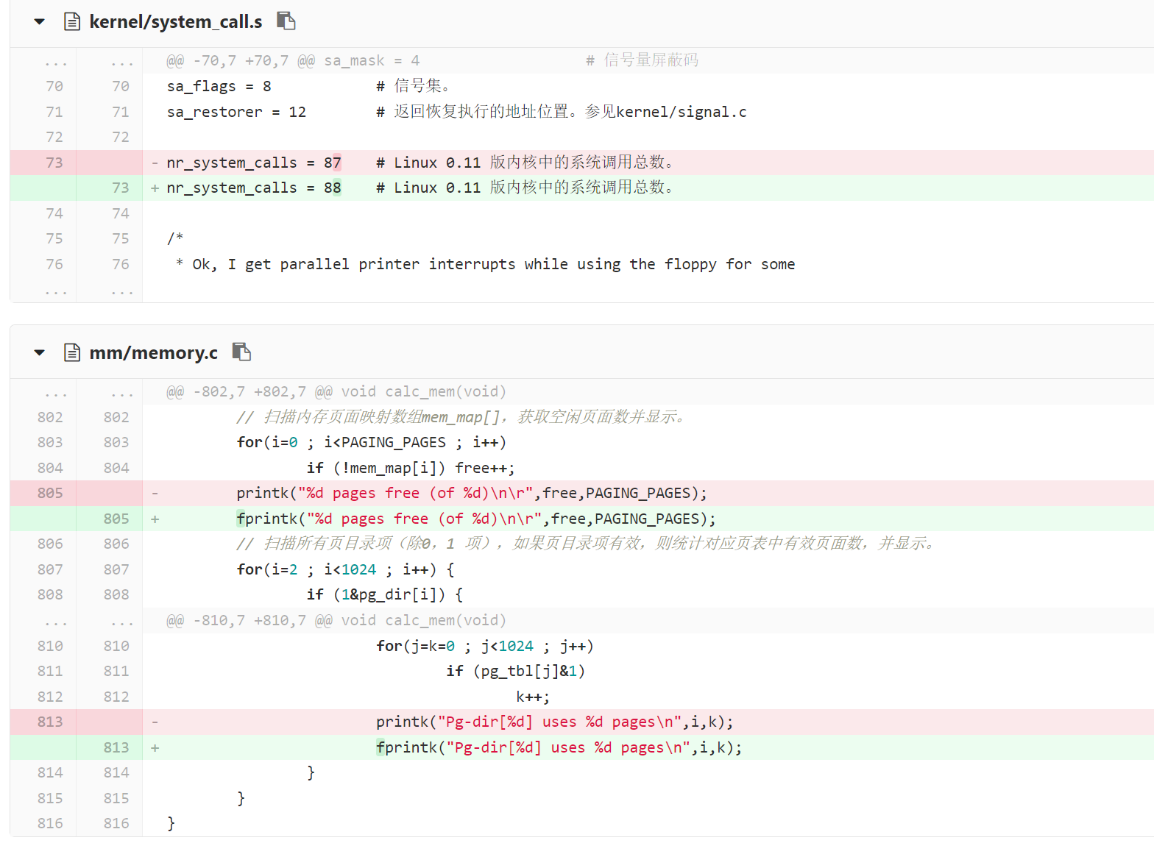
②关闭Bochs虚拟机。

③在VSCode左侧的“文件资源管理器”窗口顶部点击“New Folder”按钮，新建一个名称为newapp的文件夹。在“文件资源管理器”窗口中的newapp文件夹节点上点击鼠标右键，选择菜单中的“Reveal in File Explorer”，可以使用Windows的资源管理器打开此文件夹所在的位置，双击打开此文件夹。

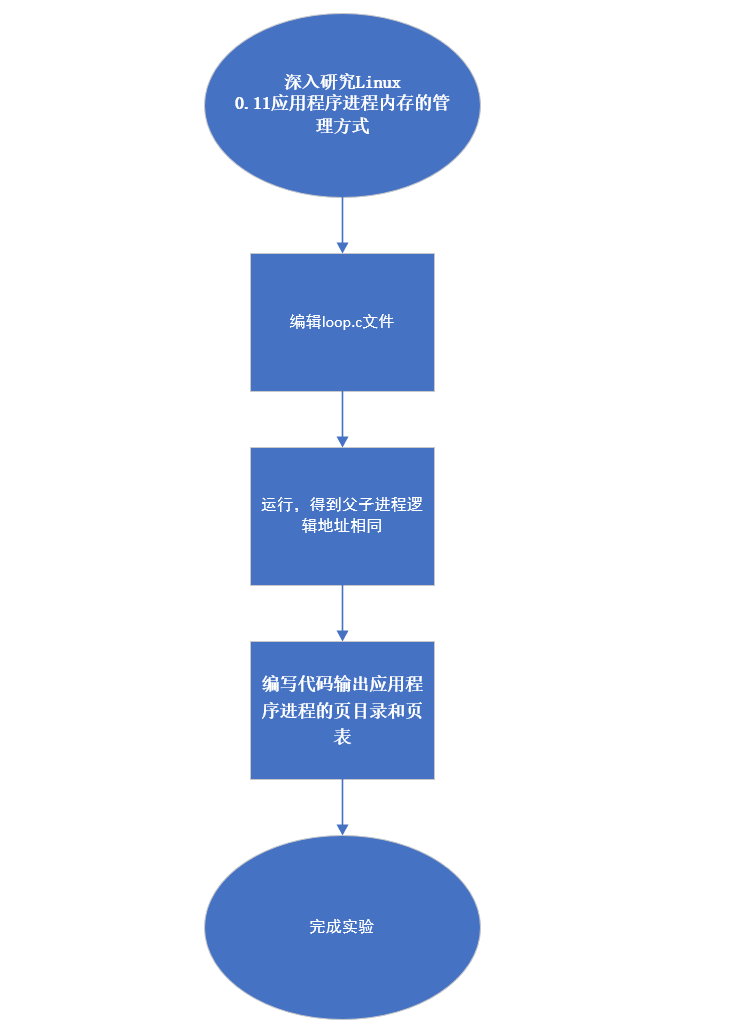
④在VSCode的“Terminal”菜单中选择“Run Build Task...”，会在VSCode的顶部中间位置弹出一个可以执行的Task列表，选择其中的“打开 floppyb.img”后会使用Floppy Editor工具打开该项目中的floppyb.img文件，查看软盘镜像中的文件列表，确保刚刚编写的文件已经成功复制到软盘镜像文件中。在文件列表中选中main.c文件，并点击工具栏上的“复制”按钮，然后粘贴到Windows的资源管理器打开的newapp文件夹中。

⑤实验结束后先使用VSCode左侧的“源代码版本控制窗口”查看文件变更详情，确认无误后再将本地项目提交平台的个人项目中。

**4.运行结果**

共修改10个文件

**5.流程图**



**6.实验体会**

通过该实验的学习，编程实现段、页式内存管理上的内存共享，我深入理解了操作系统的内存管理机制。在实现内存共享时，我需要特殊处理页表项，确保多个进程共享同一块物理内存区域时，它们都能够正确地读写该内存区域。这个过程让我更加深入地理解了操作系统的内存管理机制，同时也更加熟悉了操作系统的开发流程。

**7.思考与练习**

无

**任务十三** **用共享内存做缓冲区解决生产者—消费者问题**

**1.实验目的**

(1)深入理解物理内存的分页管理方式。

(2)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。

(3)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。

**2.实验内容**

用共享内存做缓冲区解决生产者—消费者问题

**3.实验步骤**

**3.1 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。

**3.2 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。添加系统调用的方法可以参考实验四和实验七，具体步骤这里不再详细说明。

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。

提示：

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：

gccpc.c-o app

sync

app

**3.3 提交作业**

①在Linux 0.11的终端使用下面的命令将刚刚编写的文件pc.c复制到软盘B中。

mcopypc.c b:pc.c

②关闭Bochs虚拟机。

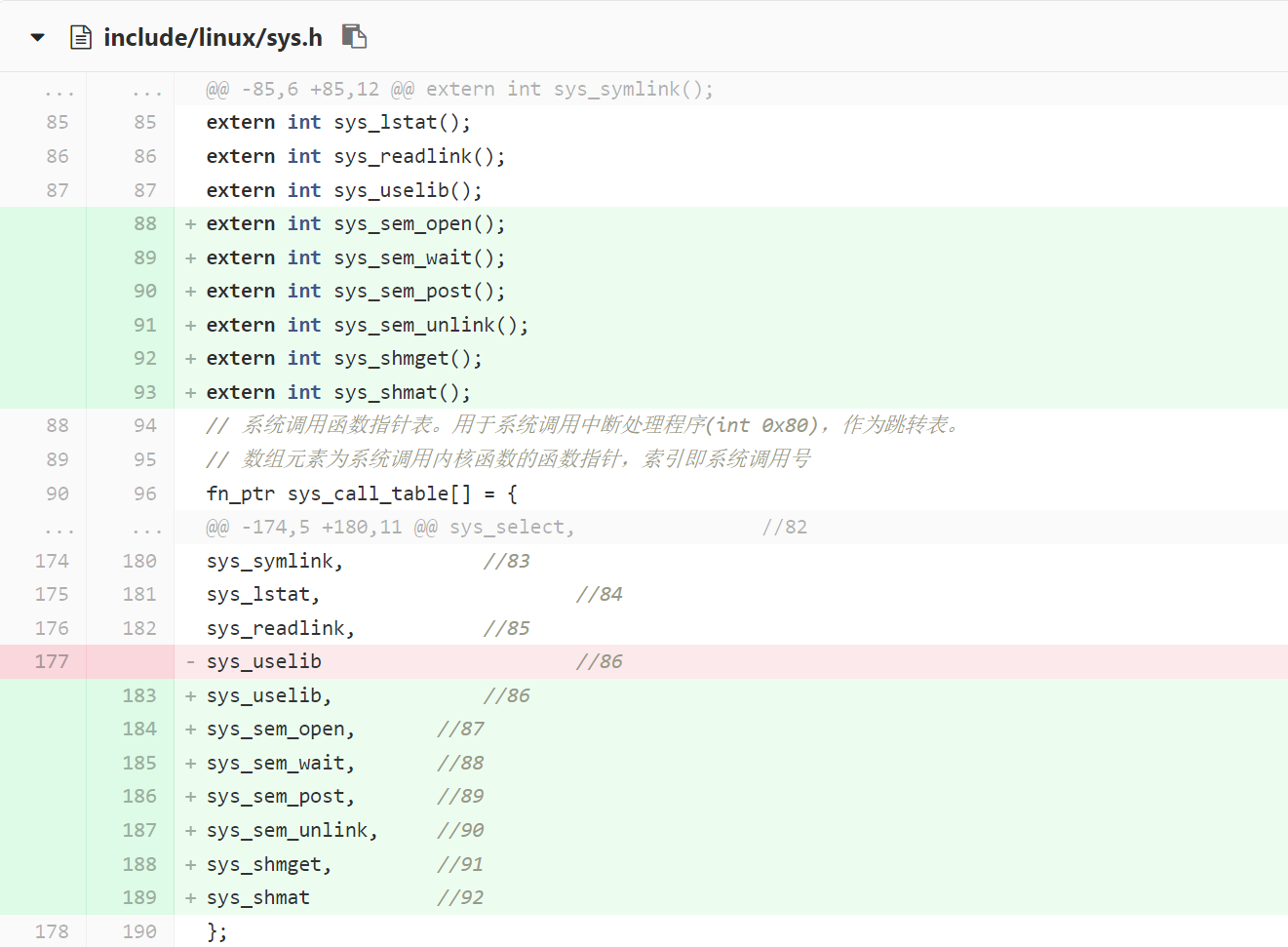
③在VSCode左侧的“文件资源管理器”窗口顶部点击“New Folder”按钮，新建一个名称为newapp的文件夹。在“文件资源管理器”窗口中的newapp文件夹节点上点击鼠标右键，选择菜单中的“Reveal in File Explorer”，可以使用Windows的资源管理器打开此文件夹所在的位置，双击打开此文件夹。

④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“打开 floppyb.img”后会使用Floppy Editor工具打开该项目中的floppyb.img文件，查看软盘镜像中的文件列表，确保刚刚编写的文件已经成功复制到软盘镜像文件中。在文件列表中选中pc.c文件，并点击工具栏上的“复制”按钮，然后粘贴到Windows的资源管理器打开的newapp文件夹中。

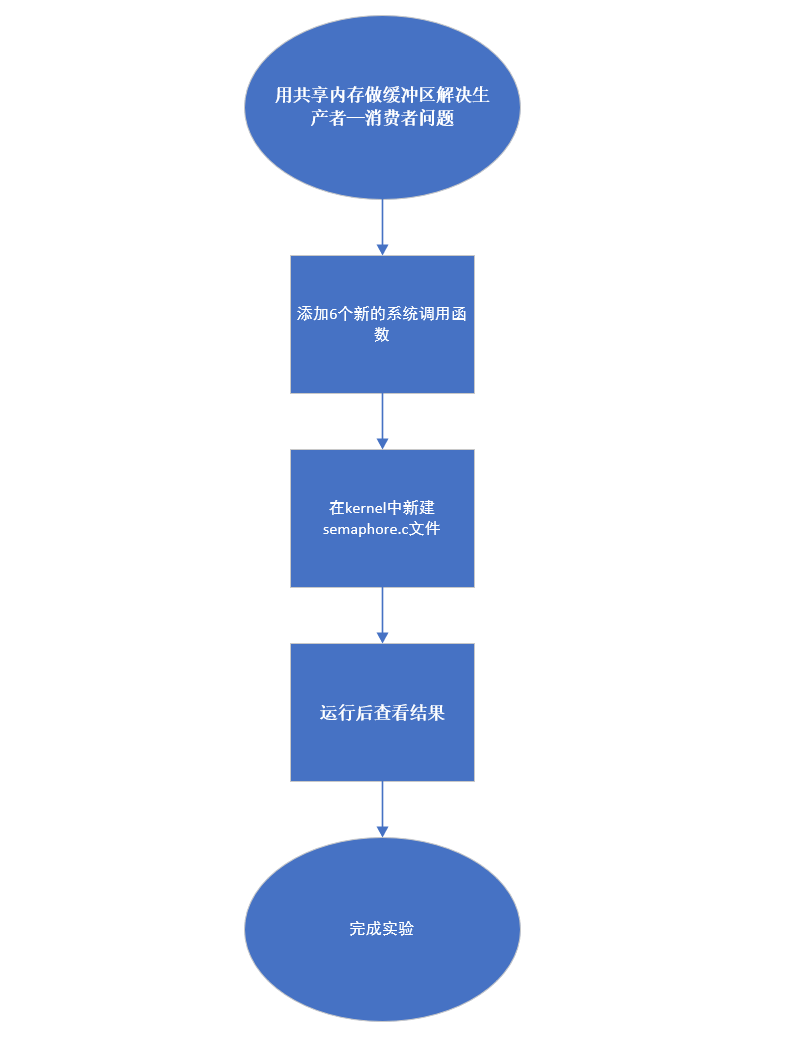
⑤实验结束后先使用VSCode左侧的“源代码版本控制窗口”查看文件变更详情，确认无误后再将本地项目提交到平台的个人项目中，方便教师通过平台查看读者提交的作业。

**4.运行结果**

共修改6个文件



**5.流程图**



**6.实验体会**

我了解了内存页式段式的格式，通过大量的练习熟悉了分配内存的基本步骤，深刻体会到了内存管理在Linux中的重要作用，同时熟悉了在本地编写分配内存文件后通过软盘b拷贝到虚拟机中的相关操作。

**7.思考与练习**

1. 请读者认真体会本实验中关于将逻辑地址映射为线性地址，线性地址映射为物理地址的过程，尝试列出映射过程中最为重要的几步。

分段和分页是将逻辑地址映射为线性地址的关键步骤。在分段过程中，通过将逻辑地址划分为段号和段内偏移量两部分，可以通过查找段表来获取该段的基地址。在分页过程中，将段内偏移量划分为页号和页内偏移量两部分，可以通过查找页表来获取该页的物理地址。最后，将页表中获取到的物理地址与页内偏移量相加，得到最终的物理地址。

2. 在本实验第3.2节的loop程序退出后，如果读者接着再运行一次，并再次进行地址跟踪，会发现有哪些异同？尝试说明原因？

在第一次运行loop程序时，操作系统会将程序加载到内存中并创建一个进程来执行该程序。在程序执行期间，操作系统会为该进程分配一段连续的内存空间。当程序执行完毕并退出时，操作系统会回收该进程所占用的内存空间，并将该进程从系统中删除。

如果接着再次运行loop程序，并进行地址跟踪，会发现程序的执行过程与第一次运行时的执行过程基本相同。因为在第二次运行loop程序时，操作系统会重新将程序加载到内存中，并创建一个新的进程来执行该程序。由于程序代码、数据等信息的地址在内存中是随机分配的，因此在第二次运行时，这些地址可能会与第一次运行时的地址不同，但程序的执行过程和结果应该是一致的。

3. 使用下面的代码在Linux 0.11内核中写一个系统调用函数，并在Linux 0.11的应用程序中调用此函数，然后仿照本实验第3.2节的内容跟踪变量i的逻辑地址、线性地址、物理地址的映射过程，最后通过将物理内存中变量i的值修改为0的方式使应用程序退出。体会Linux内核地址映射过程与3.2节中Linux应用程序地址映射过程的异同。

volatile int i = 0x12345678;

void sys\_testg()

{

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

while(i)

;

}

注意，在定义变量i时使用“volatile”关键字目的是防止编译器对其进行优化，否则i的值会被拷贝到寄存器中，而通过Bochs调试命令修改的是内存中i的值，不是寄存器中的值，就会导致程序无法退出死循环。

内核空间和用户空间的划分不同。在Linux内核中，内核空间和用户空间是分开的，内核空间的地址范围是0xC0000000到0xFFFFFFFF，用户空间的地址范围是0x00000000到0xBFFFFFFF。在3.2节中，内核空间和用户空间是共享的，它们的地址范围都是0x00000000到0xFFFFFFFF。

内核代码和数据的映射方式不同。在Linux内核中，内核代码和数据是直接映射到物理地址空间中的，在3.2节中，内核代码和数据是通过段机制映射到逻辑地址空间中的。

地址空间的划分方式不同。在Linux内核中，地址空间是通过页表机制划分的，每个进程都有自己的页表，用于将逻辑地址映射到物理地址。在3.2节中，地址空间是通过段机制划分的，每个进程都有自己的段描述符表，用于将逻辑地址映射到物理地址。

4. 将上一个练习中的变量i修改为sys\_testg函数内的局部变量，仍然通过将物理内存中变量i的值修改为0的方式使应用程序退出。请读者结合Linux 0.11内核的逻辑地址空间的内存布局，体会内核中的全局变量与局部变量在逻辑地址空间中的位置有什么不同。

在Linux 0.11内核中，全局变量和局部变量在逻辑地址空间中的位置是不同的。全局变量存储在内核数据段中，而局部变量存储在内核栈中。内核数据段在逻辑地址空间中的位置是固定的，通常是从0x00001000开始的一段连续的地址空间。而内核栈的位置是动态的，它的大小是在内核初始化时确定的，通常是从高地址向低地址扩展。

5. 在本实验3.4中遇到了共享的物理页被多个进程重复释放，导致操作系统报告错误并中断运行的问题，暂时是通过在申请共享的物理页时增加物理页在mem\_map中的引用计数来解决的。请读者换一个思路，通过实现一个简化版本的关闭共享内存的系统调用函数shmdt来解决此问题，其函数原型可以为：

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

参数key是共享内存的键值，即为共享内存数组的下标；参数startaddr是共享内存的逻辑地址。在每个生产者进程和消费者进程退出前，都必须调用此函数关闭共享内存，最终确保无论是一个生产者与一个消费者同步运行，还是一个生产者与多个消费者同步运行都不会再出现应用程序退出时操作系统报告错误的情况。

提示：

1) 读者需要实现一个将共享的物理页从当前进程的二级页表映射中移除的内核函数,其原型可以为：

void cancel\_mapping(const void\* linearaddr)

参数linearaddr是需要取消映射的物理页的线性地址，根据此线性地址找到对应的页表项，然后将页表项置为0即可。

2) 之前的源代码中只是用全局的vector数组保存了共享内存的物理页的基址，现在需要为共享内存定义一个结构体，其中除了保存物理页的基址外，还需要保存共享内存的引用计数（可以参考信号量的引用计数），再使用此结构体定义一个全局的共享内存数组。当进程调用shmdt函数关闭共享内存时，首先将共享内存的引用计数减1，然后调用cancel\_mapping函数将共享的物理页从二级页表映射中移除。由于shmdt的第二个参数startaddr是共享内存的逻辑地址，需要为其加上current->start\_code转换为线性地址后，才能作为cancel\_mapping函数的参数。最后，判断如果共享页的引用计数的值大于0，说明仍然有其他进程在使用此共享内存，就结束（不释放物理页）；否则，需要调用free\_page函数释放物理页（只释放这一次）。

先定义一个共享内存的结构体，包含物理页的基址和引用计数两个成员变量。再定义一个全局的共享内存数组，用于保存所有的共享内存，在申请共享内存时，需要遍历这个数组，找到一个空闲的位置，将物理页的基址和引用计数保存到数组中。

6. 假设一台使用x86处理器的物理裸机有4GB的物理内存，如果读者需要设计一个操作系统，要求在操作系统初始化的时候使用全局描述符表将代码段和数据段的段基址设定为0，段界限为4GB，然后将所有物理内存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。请读者说明在完成初始化后，全局描述符表中有几个描述符，各个描述符的值是多少？CS、DS寄存器的值是多少？页目录的物理页框号是多少？页目录中1024个页表项的值可能是多少？第一个页表中1024个页表项的值可能是多少？页目录和页表一共占用了多少个物理页？

代码段：基址为0，段界限为0xFFFF FFFF，访问权限为可执行、非一致性代码段，DPL为0。

数据段：基址为0，段界限为0xFFFF FFFF，访问权限为可读写向上扩展的数据段，DPL为0。

CS寄存器的值为0，DS寄存器的值为8。

每个页目录和页表都有1024个页表项，所以页目录中1024个页表项的值可能是：

指向页表的物理地址，访问权限为可读写，存在位为0（表示该页表不存在）或1（表示该页表存在）。

第一个页表中1024个页表项的值可能是：

指向物理页的物理地址，访问权限为可读写，存在位为1（表示该物理页已经被映射）。

所以页目录和页表一共占用了1 + 1024 = 1025个物理页。

**任务十四 页面置换算法**

**1.实验目的**

掌握OPT、FIFO、LRU、LFU、Clock等页面置换算法；

掌握可用空间表及分配方法；

掌握边界标识法以及伙伴系统的内存分配方法和回收方法。

**2.实验内容**

页面置换算法

**3.实验步骤**

**3.1 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（Windows控制台程序），然后使用VSCode将个人项目克隆到本地磁盘中并打开。

**3.2 查看最佳页面置换算法(OPT)和先进先出页面置换算法(FIFO)的执行过程**

**最佳页面置换算法**

当需要换出一个页面时，淘汰那个以后不再需要的或最远的将来才会用到的页面。

**先进先出页面置换算法**

当需要换出一个页面时，淘汰那个最先调入主存的页面，或者说在主存中驻留时间最长的那一页。

①运行task“生成项目(make)”生成应用程序项目，确保没有语法上的错误。

②按 Ctrl+Shift+C，在项目所在目录启动控制台CMD窗口。

③输入命令chcp 65001调整编码格式，防止输出中文时出现乱码。Win7用户还需要在控制台窗口中右键点击窗口名称，选择“设置”—》“字体”tab，设置字体为“Lucida Console”。

④输入可执行文件exe的名字后按下回车运行程序，查看OPT和FIFO的执行过程，其中，“#”表示未引用的内存块。执行结果如下图所示：



**3.3 完成最近最久未使用页面置换算法(LRU)**

**最近最久未使用页面置换算法**

当需要换出一个页面时，淘汰那个在最近一段时间里较久未被访问的页面。它是根据程序执行时所具有的局部性来考虑的，即那些刚被使用过的页面可能马上还要被使用，而那些在较长时间里未被使用的页面一般可能不会马上使用。

仔细阅读main.c中的源代码，并仿照已经实现的OPT和FIFO算法来实现最近最久未使用页面置换算法(LRU)。正确实现LRU算法后的执行结果应该如下图所示。



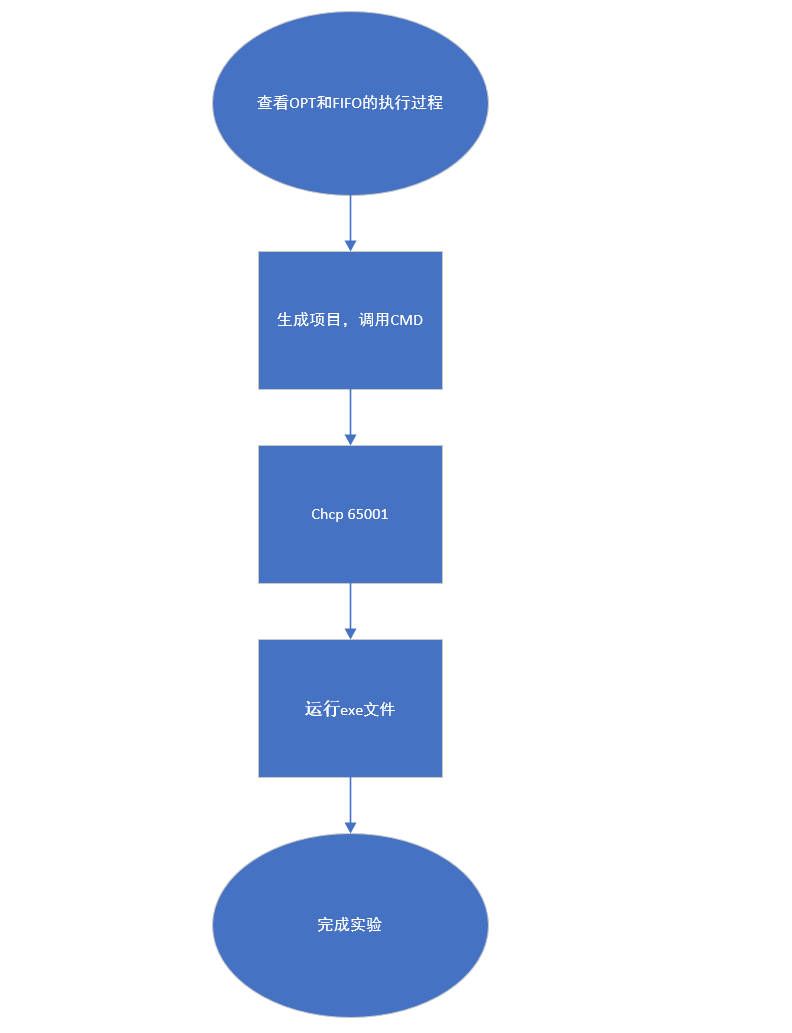
**图9-2: LRU页面置换算法的执行结果**

**3.4提交作业**

实验结束后先使用VSCode左侧的“源代码版本控制窗口”查看文件变更详情，确认无误后再将本地项目提交到平台的个人项目中，方便教师通过平台查看读者提交的作业。

**4.运行结果**

共修改一个文件

**5.流程图**

**6.实验体会**

通过本次实验，我了解了内存的控制方法。通过占用回收的方式模拟内存控制，调用各类函数实现内存块的增删。

**7.思考与练习**

无

**任务十五 动态内存分配 - 边界标识法**

**1.实验目的**

掌握OPT、FIFO、LRU、LFU、Clock等页面置换算法；

掌握可用空间表及分配方法；

掌握边界标识法以及伙伴系统的内存分配方法和回收方法。

**2.实验内容**

动态内存分配-边界标识法

**3.实验步骤**

**3.1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（Windows控制台程序），然后使用VSCode将个人项目克隆到本地磁盘中并打开。

**3.2 边界标识法的设计实现**

边界标识法是操作系统中用以进行动态分区分配的一种存储管理方法。系统将所有的空闲块链接在一个双重循环链表结构的可利用空间表中；分配可按首次拟合进行，也可按最佳拟合进行。系统的特点在于：在每个内存区的头部和底部两个边界上分别设有标识，以标识该区域为占用块或空闲块，使得在回收用户释放的空闲块时易于判别在物理位置上与其相邻的内存区域是否为空闲块，以便将所有地址连续的空闲存储区组合成一个尽可能大的空闲块。

1. **分配算法**

可以采用首次拟合法进行分配，即从表头指针所指节点起，在可利用空间表中进行查找，找到第一个容量不小于请求分配的存储量的空闲块时，即可进行分配。

1. **回收算法**

一旦作业释放占用块，系统需立即回收以备新的请求产生时进行再分配。为了使地址相邻的空闲块结合成一个尽可能大的节点，则首先需要检查刚释放的占用块的左、右邻居是否为空闲块。若释放块的左、右邻居均为占用块，则处理最为简单，只要将此新的空闲块作为一个结点插入到可利用空闲表中即可；若只有左邻居是空闲块，则应与左邻区合并成一个结点；若只有右邻居是空闲块，则应与右邻区合并成一个结点；若左右邻居都是空闲块，则应将3块合起来成为一个结点留在可利用空间表中。

1. **实现**

运行task“生成项目(make)”生成应用程序项目，确保没有语法错误。

①按 Ctrl+Shift+C，在项目所在目录启动控制台CMD窗口。

②输入命令chcp 65001调整编码格式，防止输出中文时出现乱码。Win7用户还需要在控制台窗口中右键点击窗口名称，选择“设置”—》“字体”tab，设置字体为“Lucida Console”。

③输入可执行文件exe的名字后按下回车运行程序，查看运行结果。其提供的功能项如下图所示：



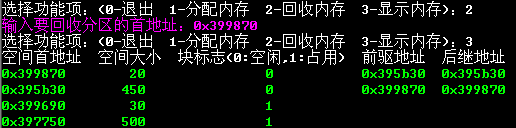
**图9-3: 边界标识法的功能项**

程序启动后会停留在此界面等待用户输入功能项序号0-3中的一个。系统会根据用户输入的序号执行相应的功能，然后继续停留在此界面等待用户的输入，直到用户按0退出应用程序。

例如，首先使用功能3查看当前内存的分配情况，然后使用功能1，输入内存长度20来分配内存，接着分配长度为30和500的内存，然后使用功能3查看内存的分配情况，如图9-4所示。如果已分配的内存达到内存的最大值时，再次分配内存时，系统会提示“无法继续分配内存”。当然还可以使用功能2来回收内存（注意：在释放内存时，首地址的输入使用十六进制）。如果要回收的内存左右有空闲区，就会与之合并，否则就作为一个结点插入到可利用空间表中，如图9-5所示。读者可在使用这些功能的同时，加深对程序的理解。



**图9-4: 分配内存**



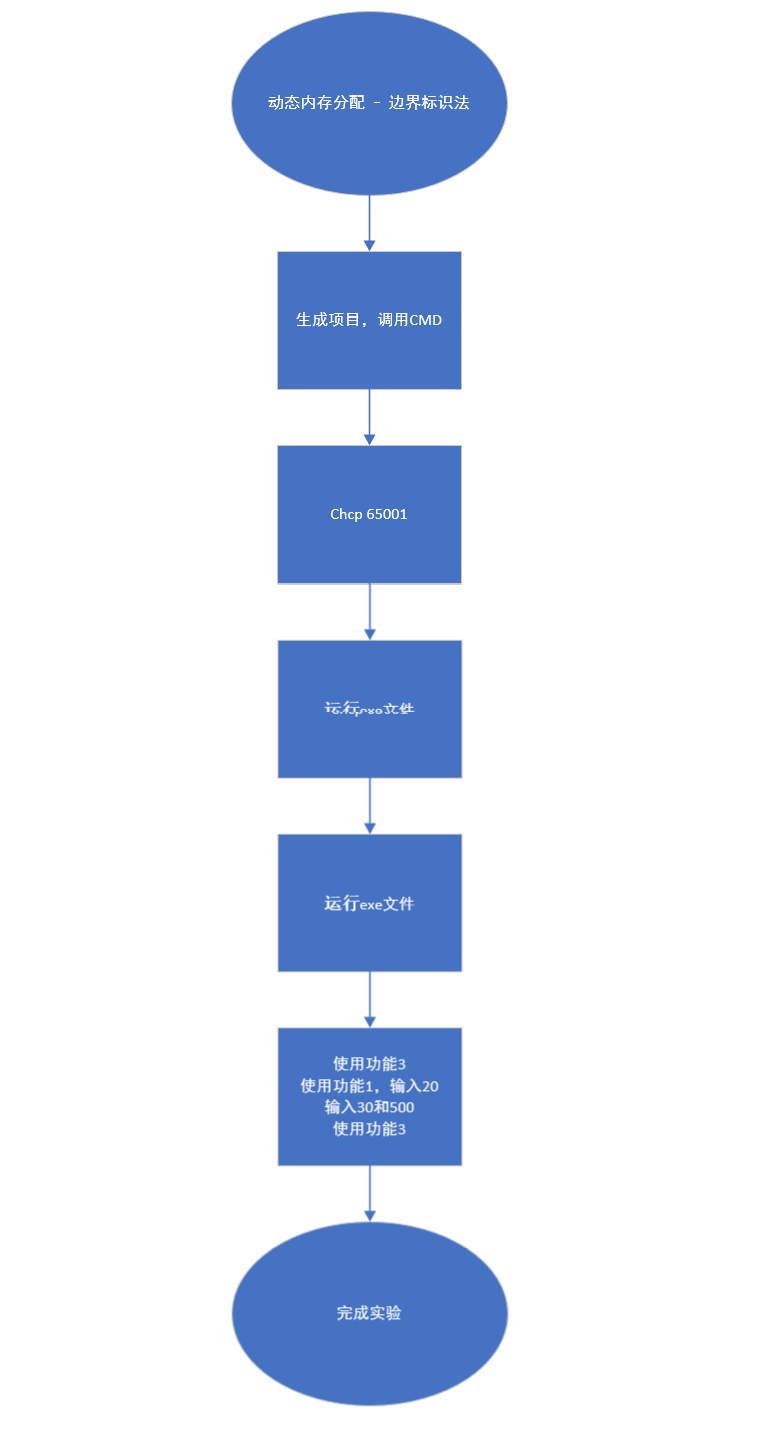
**图9-5: 回收内存**

**3.3 提交作业**

实验结束后先使用VSCode左侧的“源代码版本控制窗口”查看文件变更详情，确认无误后再将本地项目提交到平台的个人项目中，方便教师通过平台查看读者提交的作业。

**4.运行结果**

**5.流程图**



**6.实验体会**

掌握不同的页面置换算法，如OPT、FIFO、LRU、LFU、Clock等，可以更好地优化内存管理，提高系统的性能和效率。OPT算法是一种理论上最优的页面置换算法，但实际上很难实现。FIFO算法是一种简单而常用的页面置换算法，但是它没有考虑页面的访问频率和重要性，可能会导致性能下降。LRU算法是一种比较常用的页面置换算法，它根据页面最近的访问时间来判断哪些页面应该被置换出去。LFU算法是一种根据页面访问频率来判断哪些页面应该被置换出去的算法。

**7.思考与练习**

**任务十六 动态内存分配 -伙伴系统**

**1.实验目的**

**2.实验内容**

**3.实验步骤**

**3.1 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（Windows控制台程序），然后使用VSCode将个人项目克隆到本地磁盘中并打开。

**3.2 伙伴系统的设计实现**

伙伴系统是操作系统中用到的另一种动态存储管理方法。它和边界标识法类似，当用户提出申请时，分配一块大小“恰当”的内存区给用户；反之，在用户释放内存区时即回收。所不同的是，在伙伴系统中，无论是占用块或空闲块，其大小均为2的k次幂（k为某个正整数）。例如：当用户申请n个字节的内存区时，分配的占用块大小为2k个字节（2k-1＜n≤2k）。由此，在可利用空间表中的空闲块大小也只能是2的k次幂。

1. **分配算法**

当用户提出大小为n个字节的内存请求时，首先在可利用表上寻找节点大小与n相匹配的子表，若此子表非空，则将子表中任意一个结点分配之即可；若此子表为空，则需从更大的非空子表中去查找，直至找到一个空闲块，则将其中一部分分配给用户，而将剩余部分插入相应的子表中。

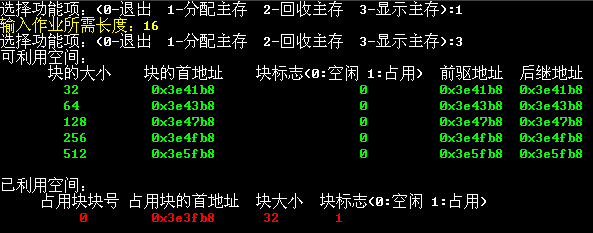
1. **回收算法**

在回收空闲块时，应首先判断其伙伴是否为空闲块，若否，则只要将释放的空闲块简单插入在相应子表中即可；若是，则需在相应子表中找到其伙伴并删除之，然后再判断合并后的空闲块的伙伴是否是空闲块。依此重复，直到归并所得空闲块的伙伴不是空闲块时，再插入到相应的子表中去。

1. **实现**



**图9-6: 伙伴系统运行的结果**

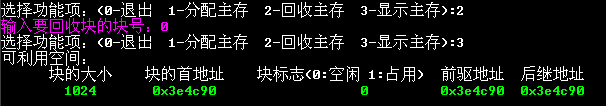
****

**图9-7: 申请内存块后的内存分布情况**

1. **练习**

编程实现伙伴系统的内存回收算法。完成算法后，可以按照下面的案例进行测试：

1. 先分配一个大小为20的内存块，然后使用功能3可以看到刚刚分配的内存块在已利用空间中的块号为0，接下来回收块号为0的内存块，会导致所有的空闲块归并为一个空闲块。此时查看内存的显示情况，如图9-8所示，在可利用空间显示了一个内存块。



**图9-8: 回收内存块后的内存分布情况**

1. 连续分配5次块大小为20的内存块，查看内存的显示情况，然后按照下面的顺序回收内存块，回收块号为0的内存块、回收块号为2的内存块、回收块号为1的内存块、回收块号为3的内存块、回收块号为4的内存块，并在每次回收之后查看内存的显示情况，如图9-9和图9-10所示，完成所有的内存块回收之后，可利用空间恢复为一个内存块。



**图9-9: 回收块号为2的内存块后的分布情况**



**4.运行结果**

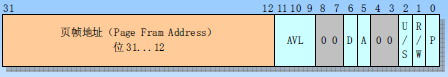
**5.流程图**

无

**6.实验体会**

除了页面置换算法，还有可用空间表及分配方法、边界标识法以及伙伴系统的内存分配方法和回收方法等内存管理技术。可用空间表及分配方法可以帮助我们更好地管理内存空间，避免内存碎片的产生。边界标识法是一种简单而常用的内存分配方法，它可以将内存空间分割成大小相等的块，并使用边界标识来记录每个块的状态。伙伴系统是一种高效的内存分配方法，它可以将内存空间分割成大小相等的块，并使用二叉树来管理这些块，从而实现高效的内存分配和回收。

**7.思考与练习**

****1.在Linux011 内核项目中，打开lib/malloc.c文件，仔细阅读其中的源代码及注释，查看一下linux 0.11系统动态分配和回收内存的方法。其使用的方法与本实验介绍的边界标识法和伙伴系统法不同。请读者尝试分别用边界标识法和伙伴系统算法来修改malloc函数和free函数来修改系统的动态分配内存和回收内存的方法。

使用边界标识法来实现malloc和free函数，需要修改内存块的数据结构，以记录每个空闲块的起始位置和大小。malloc函数则需要遍历空闲块链表，找到一个大小满足要求的空闲块，并将其分配出去。free函数则需要将一个已分配的块插入到空闲块链表中，并合并相邻的空闲块。

使用伙伴系统算法来实现，则需要将内存划分为大小为2的幂次方的块，并使用一个伙伴系统来管理这些块。malloc函数则需要在伙伴系统中找到一个大小满足要求的块，并将其分配出去。free函数则需要将一个已分配的块插入到伙伴系统中，并合并相邻的空闲块。

2.x86平台为实现改进型的Clock页面置换算法提供了硬件支持，它实现了二级映射机制，它定义的页表项的格式如下：

其中位5是访问标志，当处理器访问页表项映射的页面时，页表表项的这个标志就会置为1。位6是页面修改标志，当处理器对一个页面执行写操作时，该项就会被置为1。

在Linux 0.11的内核中写一个系统调用函数来实现改进型的Clock页面置换算法的模拟。参考实验八的第3.5节，在此函数中将二级映射表打印出来，然后实现一个算法来根据页表项中的访问位和修改位计算出应该置换出的页面。在此函数中不必完全实现Clock页面置换算法，只要能判断出应该置换的页面即可。再编写一个Linux 0.11应用程序，在应用程序中调用此系统调用函数。

在系统调用表中添加一个新的系统调用号后，具体的实现如下：

// 系统调用号

#define \_\_NR\_CLOCK\_PAGE\_SWAP 333

// 页表项

typedef struct {

unsigned int present : 1;

unsigned int rw : 1;

unsigned int user : 1;

unsigned int accessed : 1;

unsigned int dirty : 1;

unsigned int unused : 7;

unsigned int frame : 20;

} page\_table\_entry;

// 页目录项

typedef struct {

unsigned int present : 1;

unsigned int rw : 1;

unsigned int user : 1;

unsigned int unused : 9;

unsigned int page\_table\_base : 20;

} page\_directory\_entry;

// 页目录表

page\_directory\_entry \*page\_directory = (page\_directory\_entry \*)0x1000;

// 页表

page\_table\_entry \*page\_table = (page\_table\_entry \*)0x2000;

// 二级映射表

unsigned char \*second\_level\_map = (unsigned char \*)0x3000;

// 系统调用函数

int sys\_clock\_page\_swap(void) {

int i, j;

unsigned int address;

unsigned char access\_bit, dirty\_bit;

unsigned int page\_frame\_to\_swap = 0;

// 打印二级映射表

for (i = 0; i < 1024; i++) {

if (page\_directory[i].present == 1) {

for (j = 0; j < 1024; j++) {

if (page\_table[i \* 1024 + j].present == 1) {

address = (i << 22) | (j << 12);

printf("Virtual address: %x, Physical address: %x\n", address, address | (page\_table[i \* 1024 + j].frame << 12));

}

}

}

}

// 遍历页表，计算每个页面的访问位和修改位

for (i = 0; i < 1024; i++) {

if (page\_directory[i].present == 1) {

for (j = 0; j < 1024; j++) {

if (page\_table[i \* 1024 + j].present == 1) {

access\_bit = second\_level\_map[i \* 1024 + j] & 0x1;

dirty\_bit = (second\_level\_map[i \* 1024 + j] >> 1) & 0x1;

// 根据访问位和修改位计算出应该置换出的页面

if (access\_bit == 0 && dirty\_bit == 0) {

page\_frame\_to\_swap = page\_table[i \* 1024 + j].frame;

break;

}

}

}

}

}

// 返回应该置换出的页面的页框号

return page\_frame\_to\_swap;

}

测试的应用程序实现如下：

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/syscall.h>

#define \_\_NR\_CLOCK\_PAGE\_SWAP 333

int main() {

int page\_frame\_to\_swap;

// 调用系统调用函数

page\_frame\_to\_swap = syscall(\_\_NR\_CLOCK\_PAGE\_SWAP);

printf("Page frame to swap: %d\n", page\_frame\_to\_swap);

return 0;

}