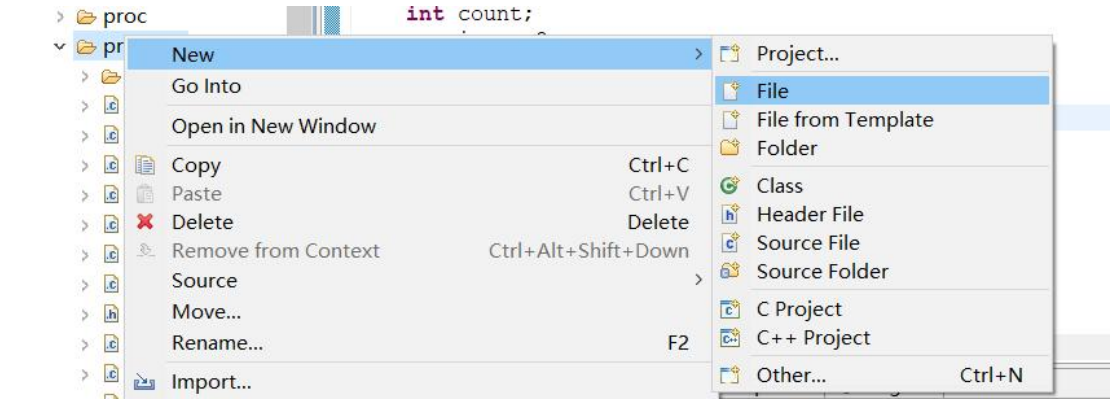
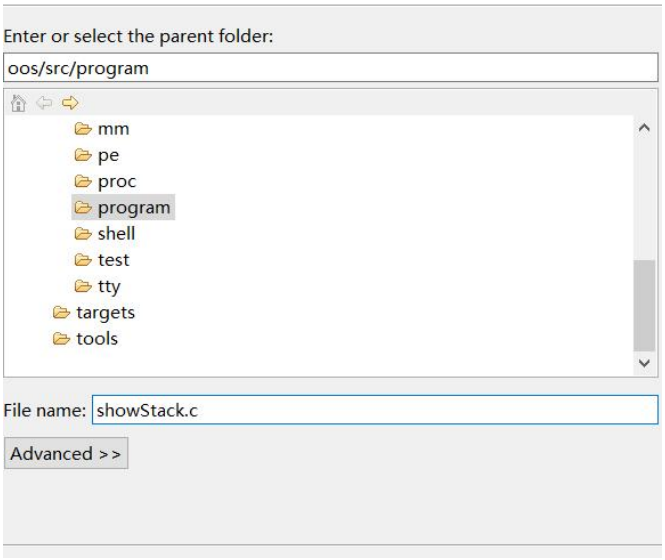
# P03：UNIX V6++完整的进程图像

2152118 史君宝

## 一、 完成实验 4.1-4.2，回答问题

（1） 在 program 文件加入一个新的 c 语言文件




showStack.c Makefile

```c
#include <stdio.h>

int version = 1;

main1()
{
    int a, b, result;
    a = 1;
    b = 2;
    result = sum(a, b);
    printf("result = %d\n", result);
}

int sum(var1, var2)
{
    int count;
    version = 2;
    count = var1 + var2;
    return(count);
}
```

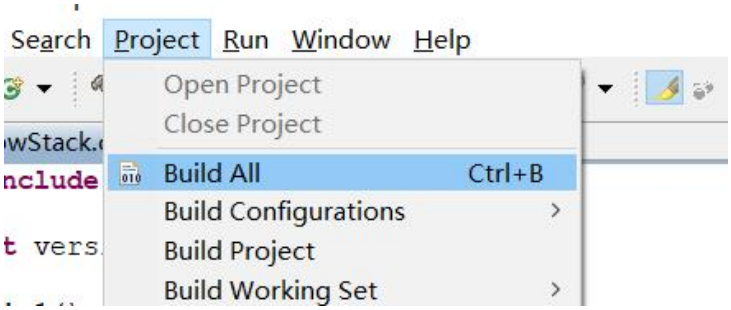
（2） 修改编译需要使用的 Makefile 文件

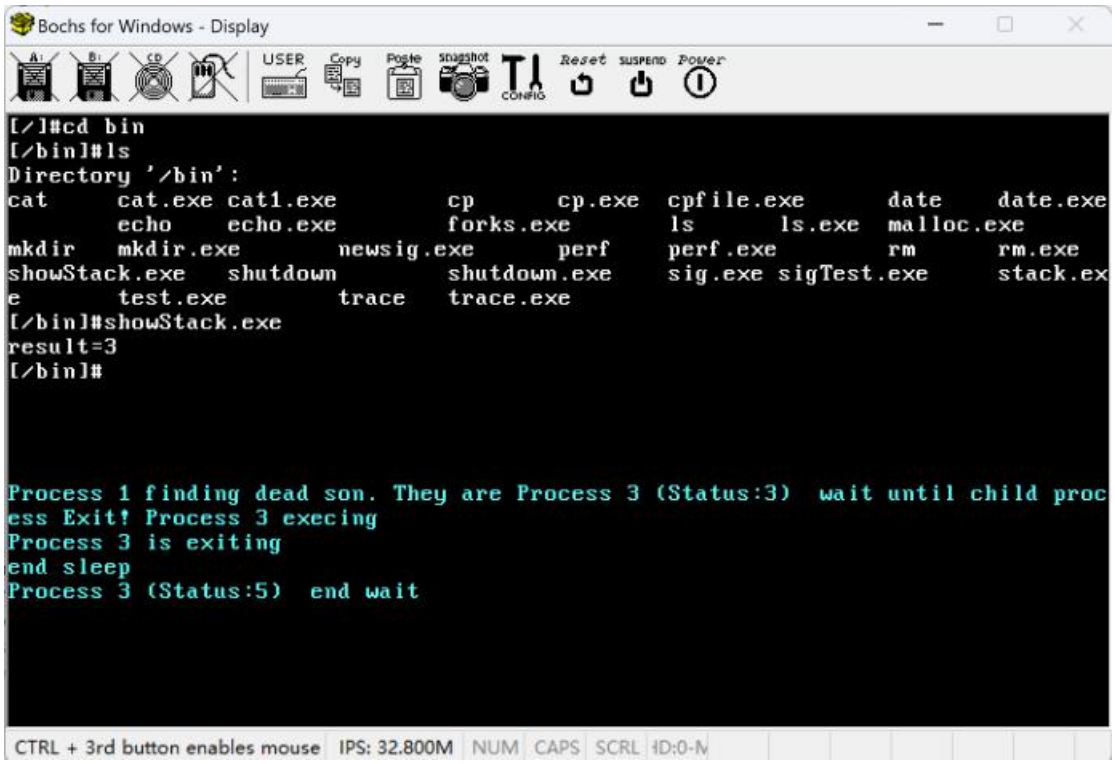
```
$(TARGET)\stack.exe \
$(TARGET)\malloc.exe\
$(TARGET)\showStack.exe
```

```
$(TARGET)\malloc.exe :      malloc.c
    $(CC) $(CFLAGS) -I"$(INCLUDE)" -I"$(LIB_INCLUDE)"  $< -e _main1 $(V6++LIB) -o $@
    copy $(TARGET)\malloc.exe $(MAKEIMAGEPATH)\$(BIN)\malloc.exe

$(TARGET)\showStack.exe :       showStack.c
    $(CC) $(CFLAGS) -I"$(INCLUDE)" -I"$(LIB_INCLUDE)"  $< -e _main1 $(V6++LIB) -o $@
    copy $(TARGET)\showStack.exe $(MAKEIMAGEPATH)\$(BIN)\showStack.exe
```

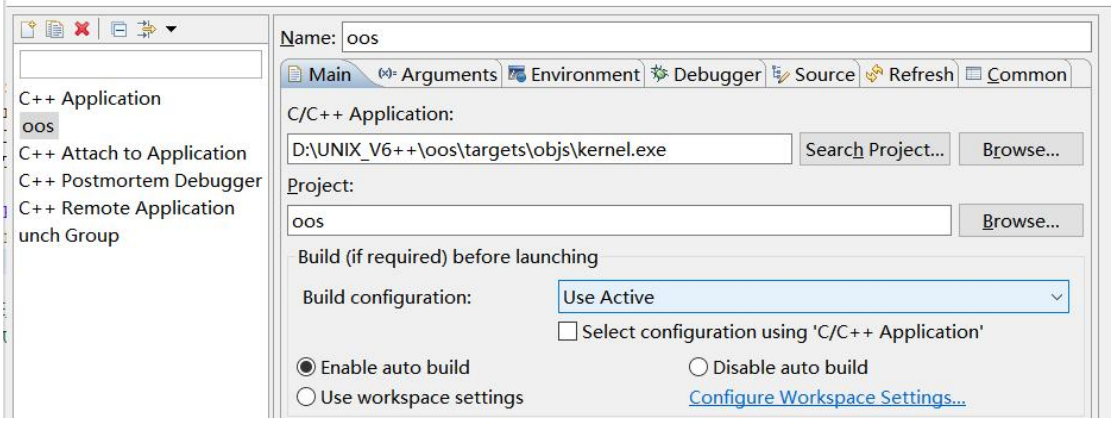
（3） 重新编译运行 UNIX V6++代码

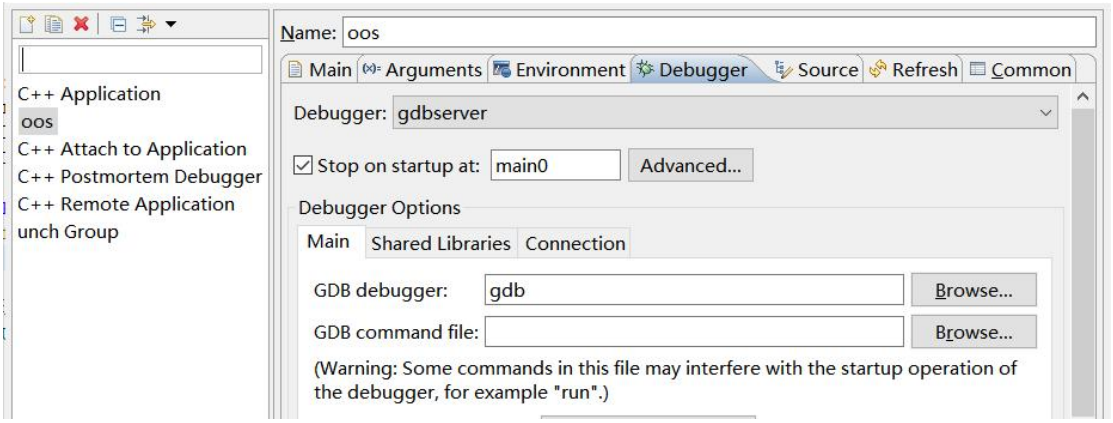



（4） 设置调试对象和调试起点

在实验二中我们需要具体观察 showStack.c 的具体的执行过程，所以调试对象设置的为 showStack.exe。而在本实验中我们要观察进程的页表，应将调试对象修改为 Kernel.exe。




（5） 设置断点，并开始调试

```
void Process::Exit()
{
    int i;
    User& u = Kernel::Instance().GetUser();
    ProcessManager& procMgr = Kernel::Instance().GetProcessManager();
    OpenFileTable& fileTable = *Kernel::Instance().GetFileManager().m_OpenFileTable;
    InodeTable& inodeTable = *Kernel::Instance().GetFileManager().m_InodeTable;

    Diagnose::Write("Process %d is exiting\n",u.u_procp->p_pid);
    /* Reset Tracing flag */
    u.u_procp->p_flag &= (~Process::STRC);

    /* 清除进程的信号处理函数，设置为1表示不对该信号作任何处理 */
    for ( i = 0; i < User::NSIG; i++ )
```

```
[/]#cd bin
[/bin]#showStack.exe
result = 3
[/bin]#showStack.exe
```

（6） 观察 User 的文件，查看信息：

Use.h：

```
public:
    unsigned long u_rsav[2];    /* 用于保存esp与ebp指针 */
    unsigned long u_ssav[2];    /* 用于对esp和ebp指针的二次保护 */
    Process* u_procp;           /* 指向该u结构对应的Process结构 */
    /* 新添加变量，用于替代原有的变量
     * int u_uisa[16]
     * int u_uisd[16]
     * u_tsize
     * u_dsize
     * u_ssize
     */
    MemoryDescriptor u_MemoryDescriptor;
```

| Address | 0 - 3 | 4 - 7 | 8 - B | C - F |
|---|---|---|---|---|
| C03FF000 | C03FFF8C | C03FFFA4 | 00000000 | 00000000 |
| C03FF010 | C0119600 | C0208000 | 00401000 | 00003000 |
| C03FF020 | 00404000 | 00003000 | 00001000 | C03FFFDC |
| C03FF030 | 00000000 | 007FFBB0 | 0000000B | 00000000 |
| C03FF040 | C03FFFEC | 00000000 | 00000000 | 00000008 |
| C03FF050 | 00000000 | 00000000 | 00000000 | 00000000 |

其中 0XC03FF000 开始的两个四字节是 u_rsav，之后的两个四字节是 u_ssav，0XC0119600 是 u_procp。之后便是 u_MemoryDescriptor
我们查找具体的代码，可以知道：
MemoryDescriptor.h：

```
public:
    PageTable*      m_UserPageTableArray;
    /* 以下数据都是线性地址 */
    unsigned long   m_TextStartAddress; /* 代码段起始地址 */
    unsigned long   m_TextSize;         /* 代码段长度 */

    unsigned long   m_DataStartAddress; /* 数据段起始地址 */
    unsigned long   m_DataSize;         /* 数据段长度 */

    unsigned long   m_StackSize;        /* 栈段长度 */
    //unsigned long m_HeapSize;         /* 堆段长度 */
};
```

| | | |
|---|---|---|
| u_rsav | 用于保存 esp 与 eb 指针 | 0XC03FFF8C　0XC03FFFA4 |
| u_ssav | 用于对 esp 和 ebp 指针的二次保护 | |
| u_procp | 指向 process 结构 | 0XC0119600 |
| u_MemoryDescriptor | | |
| m_UserPageTa | 相对映射段首地址 | 0XC0208000 |

| | | |
|---|---|---|
| bleArray | | |
| m_TextStartAddress | 代码段首地址 | 0x00401000=4M+4K |
| m_TextSize | 代码段长度 | 0x00003000=12K |
| m_DataStartAddress | 数据段首地址 | 0x00404000=4M+16K |
| m_DataSize | 数据段长度 | 0x00003000=12K |
| m_StackSize | 栈段长度 | 0x00001000=4K |

（7）进程的 proc 结构，获得信息

Process.h：

```
public:
    /* 用于标识进程的标识 */
    short p_uid;        /* 用户ID */
    int p_pid;          /* 进程标识数，进程编号 */
    int p_ppid;         /* 父进程标识数 */

    /* 进程内存中图像信息位置 */
    unsigned long   p_addr; /* TBD user结构即ppda区在物理内存中的地址，用于替代页表中的某一项 */
    unsigned int    p_size; /* 除共享正文段的长度，以字节单位 */
    Text*   p_textp;        /* 指向该进程所运行的代码段的描述符 */

    /* 进程调度状态 */
    ProcessState    p_stat; /* 进程当前状态 */
    int p_flag;         /* 进程标志位，可以将多个状态组合 */
```

对应的 Memory 信息为：

0XC0119600 <Hex> | 0XC0119600 : 0xC0119600 <Hex Integer> | New Renderings...

| Address | 0 - 3 | 4 - 7 | 8 - B | C - F |
|---|---|---|---|---|
| C01195F0 | C01195C0 | 00000000 | C0120DA0 | 00000000 |
| C0119600 | 00000000 | 00000002 | 00000001 | 0040F000 |
| C0119610 | 00005000 | C011AE94 | 00000003 | 00000001 |
| C0119620 | 00000065 | 00000018 | 00000000 | 00000000 |
| C0119630 | 00000000 | 00000000 | C0120DA0 | 00000000 |
| C0119640 | 00000000 | 00000000 | FFFFFFFF | 00000000 |

可以知道 00000000 是用户 ID，00000002 是进程编号，00000001 是父进程标识数,0X0040F000 是 PPDA 区在物理内存之中的地址,0X00005000 是 p_size，而 0XC011AE94 是指向 Text 表中的 Text 对象的指针。

因此我们可以对 Process 结构的内容画出下面的图：

| 变量名称 | 含义 | 值 |
| --- | --- | --- |
| short p_uid | 用户 ID | 0 |
| int p_pid | 进程标识数 | 2 |
| int p_ppid | 父进程标识数 | 1 |
| unsigned long p_addr | user 结构即 ppda 区的物理地址 | 0x0040F000 |
| unsigned int p_size | 除共享正文段的长度，以字节单位 | 0x00005000=20K |
| Text* p_textp | 指向代码段 Text 结构的逻辑地址 | 0xC011AE94 |
| ProcessState p_stat | 进程调度状态 | 3=SRUN |
| int p_flag | 进程标志位 | 1=SLOAD |
| int p_pri | 进程优先数 | 65 |
| int p_cpu | cpu 值，用于计算 p_pri | 19 |
| int p_nice | 进程优先数微调参数 | 0 |
| int p_time | 进程在盘上(内存内)驻留时间 | 0 |
| unsigned long p_wchan | 进程睡眠原因 | 0 |

(8)获得 Text 结构，观察信息：

```
public:
    int             x_daddr;    /* 代码正文段在盘交换区上的地址 */
    unsigned long   x_caddr;    /* 代码正文段在物理内存中的起始地址，以字节为单位 */
    unsigned int    x_size;     /* 代码段长度，以字节为单位 */
    Inode*          x_iptr;     /* 内存inode地址 */
    unsigned short  x_count;    /* 共享正文段的进程数 */
    unsigned short  x_ccount;   /* 共享该正文段且图像在内存的进程数 */
};
```

之后我们根据前面获得的 Text 表中的 Text 对象逻辑地址 0XC011AE94 打开 Memory 对应位置：

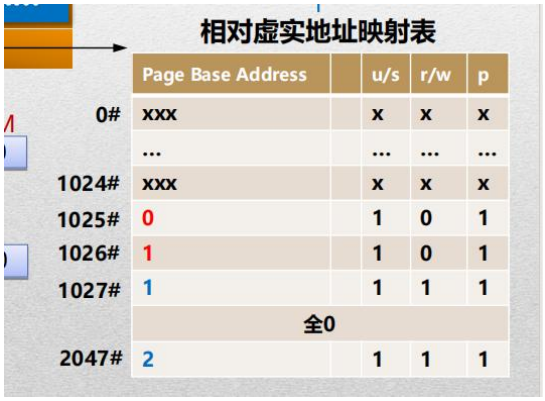
| Address | 0 - 3 | 4 - 7 | 8 - B | C - F |
| --- | --- | --- | --- | --- |
| C011AE90 | 00010001 | 00004670 | 0040C000 | 00003000 |
| C011AEA0 | C011ECD0 | 00010001 | 00000000 | 00000000 |
| C011AEB0 | 00000000 | 00000000 | 00000000 | 00000000 |
| C011AEC0 | 00000000 | 00000000 | 00000000 | 00000000 |
| C011AED0 | 00000000 | 00000000 | 00000000 | 00000000 |
| C011AEE0 | 00000000 | 00000000 | 00000000 | 00000000 |

因此可以得到：

| 变量名称 | 含义 | 值 |
| --- | --- | --- |
| int x_daddr | 代码段在盘交换区上的地址 | 0x00004670 |
| unsigned long x_caddr | 代码段起始地址（物理地址） | 0x0040C000 |
| unsigned int x_size | 代码段长度，以字节为单位 | 0x00003000 = 12K |
| Inode* x_iptr | 内存 inode 地址 | 0xC011ECD0 |
| Unsigned short x_count | 共享正文段的进程数 | 1 |
| Unsigned short x_ccount | 共享该正文段且图像在内存的进程数 | 1 |

（9）绘制相对虚实地址映射表和物理页表。

我们查询课本的知识，绘制相对虚实地址映射表：



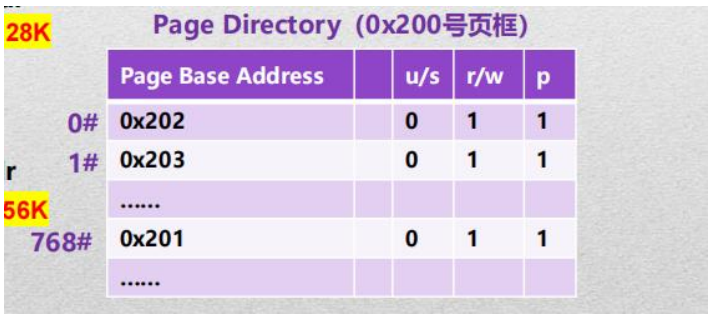

其中可以知道的是：


因此可以绘制：

| 页号 | 地址 | 页框号 | 低 12 位 |
|---|---|---|---|
| 0# | 0XC0208000-0XC0208003 | / | / |
| …… | …… | …… | …… |
| 1024# | 0XC0209000-0XC0209003 | / | / |
| 1025# | 0XC0209004-0XC0209007 | 0(0X00000) | 0X005 |
| 1026# | 0XC0209008-0XC020900B | 1(0X00001) | 0X005 |
| 1027# | 0XC020900C-0XC020900F | 2(0X00002) | 0X005 |
| 1028# | 0XC0209010-0XC0209013 | 1(0X00001) | 0X007 |
| 1029# | 0XC0209014-0XC0209017 | 2(0X00002) | 0X007 |
| 1030# | 0XC0209018-0XC020901B | 3(0X00003) | 0X007 |
| 1031# | 0XC020901C-0XC020901F | 0(0X00000) | 0X004 |
| …… | …… | …… | …… |
| 2047# | 0XC0209FFC-0XC0209FFF | 4(0X00004) | 0X007 |

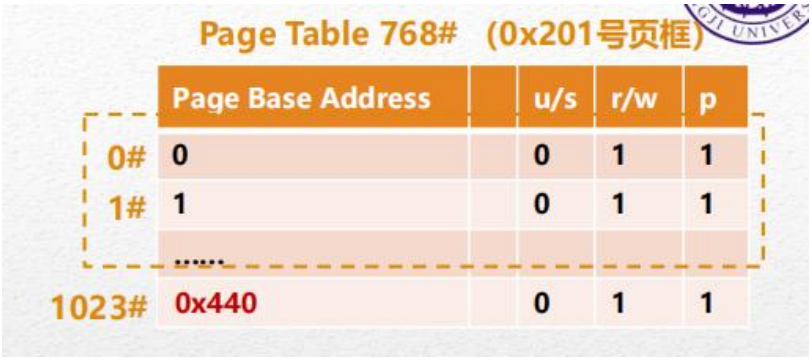
物理页表为：

目录页表 0X200#



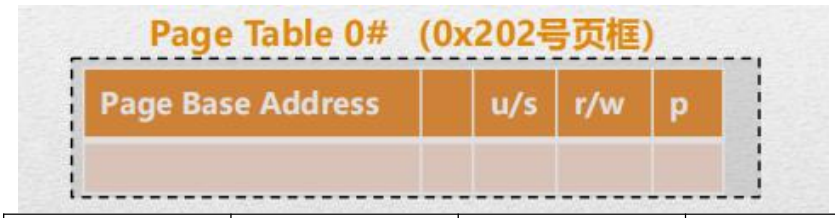
| 页号 | PBA | U/S | R/W | P |
|---|---|---|---|---|
| 0# | 0X202 | 0 | 1 | 1 |
| 1# | 0X203 | 0 | 1 | 1 |
| …… | …… | …… | …… | …… |
| 768# | 0X201 | 0 | 1 | 1 |
| …… | …… | …… | …… | …… |

目录页表 0X201#




| 页号 | PBA | U/S | R/W | P |
|---|---|---|---|---|
| 0# | 0X000 | 0 | 1 | 1 |
| 1# | 0X001 | 0 | 1 | 1 |
| …… | …… | …… | …… | …… |
| 1023# | 0X40F(p_addr>>12) | 0 | 1 | 1 |

目录页表 0X202#




| 页号 | PBA | U/S | R/W | P |
|---|---|---|---|---|
| / | / | / | / | / |

目录页表 0X203#

| 页号 | PBA | U/S | R/W | P |
|---|---|---|---|---|
| 0# | / | / | / | / |
| 1# | 0X40C(x_caddr>>12) | 1 | 0 | 1 |
| 2# | 0X40D(x_caddr>>12+1) | 1 | 0 | 1 |
| 3# | 0X40E(x_caddr>>12+2) | 1 | 0 | 1 |
| 4# | 0X410(p_addr>>12+1) | 1 | 1 | 1 |
| 5# | 0X411(p_addr>>12+2) | 1 | 1 | 1 |
| 6# | 0X412(p_addr>>12+3) | 1 | 1 | 1 |
| …… | …… | …… | …… | …… |
| 1023# | 0X413(p_addr>>12+4) | 1 | 1 | 1 |

（10）总结：获得进程代码段和 PPDA 去起始地址的方法。

由于进程 User 结构的逻辑地址固定，恒为 0xC03FF000，这也是 PPDA 区的逻辑地址。我们打开其对应位置 Memory，可知 proc 结构在逻辑空间中的地址，且 u_MemoryDescriptor 中的 m_TextStartAddress 就是代码段的逻辑地址。我们打开 proc 的 Memory 可以从 p_addr 中知道 PPDA 区的物理地址，并从 p_textp 可以知道 text 结构的逻辑地址，打开对应 Memory 从 p_caddr 就可以知道代码段的物理地址了。

表 4：进程图象完整信息

| 名称 | 逻辑地址 | 物理地址 | 大小 |
|---|---|---|---|
| 代码段 | 0x00401000 | 0x0040C000 | 12K |
| 可交换部分 | 0xC03FF000 | 0x0040F000 | 20K |
| PPDA 区 | 0xC03FF000 | 0x0040F000 | 4K |
| 数据段 | 0x00404000 | 0x00410000 | 12K |
| 堆栈段 | | 0x00413000 | 1K |

## 二、 完成实验 4.3，获取完整的进程相对虚实地址映射表，补齐表 5。

（1）找到对应 Memory，补齐表五：

| Address | 0 - 3 | 4 - 7 | 8 - B | C - F |
|---|---|---|---|---|
| C0208000 | 00000004 | 00000004 | 00000004 | 00000004 |
| C0208010 | 00000004 | 00000004 | 00000004 | 00000004 |
| C0208020 | 00000004 | 00000004 | 00000004 | 00000004 |
| C0208030 | 00000004 | 00000004 | 00000004 | 00000004 |
| C0208040 | 00000004 | 00000004 | 00000004 | 00000004 |
| C0208050 | 00000004 | 00000004 | 00000004 | 00000004 |

表 5：进程的相对虚实地址映射表

| 页号 | 地址 | 值 | | |
|---|---|---|---|---|
| | | 高 20 位页框号 | 低 12 位标志位（u/s r/w p） | |
| 0# | 0xC0208000~0xC0208003 | | | |
| | …… | | | |
| 1024# | 0xC0208000~0xC0208003 | | | |
| 1025# | 0xC0209004~0xC0209007 | | 005（0000 0000 0101） | |
| 1026# | 0xC0209008~0xC020900B | | 005（0000 0000 0101） | 代码段 |
| 1027# | 0xC020900C~0xC020900F | | 005（0000 0000 0101） | |
| 1028# | 0xC0209010~0xC0209013 | | 007（0000 0000 0111） | |
| 1029# | 0xC0209014~0xC0209017 | | 007（0000 0000 0111） | 数据段 |
| 1030# | 0xC0209018~0xC020901B | | 007（0000 0000 0111） | |
| 1031# | 0xC020901C~0xC020901F | | 004（0000 0000 0100） | |
| | …… | …… | …… | |
| 2047# | 0xC0209FFC~0xC0209FFF | | 007（0000 0000 0111） | 堆栈段 |

由上可知，在上述表中的 0#到 1024#的位置都是 0X0000 0004
之后我们找到 1025# 的位置：

| | | | | |
|---|---|---|---|---|
| C0208FF0 | 00000004 | 00000004 | 00000004 | 00000004 |
| C0209000 | 00000004 | 00000005 | 00001005 | 00002005 |
| C0209010 | 00001007 | 00002007 | 00003007 | 00000004 |
| C0209020 | 00000004 | 00000004 | 00000004 | 00000004 |
| C0209030 | 00000004 | 00000004 | 00000004 | 00000004 |
| C0209040 | 00000004 | 00000004 | 00000004 | 00000004 |

可以看到 1024#的位置是 0X00000004,之后我们需要补齐 1025#-1027#，
它们分别是 0X00000005，0X00001005，0X00002005；对应的物理页框
就是 0X00000,0X00001,0X00002：
与所绘制的表是相同的
在 1028#-1030#，它们分别是 0X00001007，0X00002007，0X00003007；
对应的物理页框就是 0X00001,0X00002,0X00003：
与所绘制的表是相同的
如图：

| | | | |
|---|---|---|---|
| 1025# | 0XC0209004-0XC0209007 | 0(0X00000) | 0X005 |
| 1026# | 0XC0209008-0XC020900B | 1(0X00001) | 0X005 |
| 1027# | 0XC020900C-0XC020900F | 2(0X00002) | 0X005 |
| 1028# | 0XC0209010-0XC0209013 | 1(0X00001) | 0X007 |
| 1029# | 0XC0209014-0XC0209017 | 2(0X00002) | 0X007 |
| 1030# | 0XC0209018-0XC020901B | 3(0X00003) | 0X007 |

所以完整的相对虚实映射表：

| 页号 | 地址 | 页框号 | 低 12 位 |
|---|---|---|---|
| 0# | 0XC0208000-0XC0208003 | / | / |
| …… | …… | …… | …… |
| 1024# | 0XC0209000-0XC0209003 | / | / |
| 1025# | 0XC0209004-0XC0209007 | 0(0X00000) | 0X005 |
| 1026# | 0XC0209008-0XC020900B | 1(0X00001) | 0X005 |
| 1027# | 0XC020900C-0XC020900F | 2(0X00002) | 0X005 |
| 1028# | 0XC0209010-0XC0209013 | 1(0X00001) | 0X007 |
| 1029# | 0XC0209014-0XC0209017 | 2(0X00002) | 0X007 |
| 1030# | 0XC0209018-0XC020901B | 3(0X00003) | 0X007 |
| 1031# | 0XC020901C-0XC020901F | 0(0X00000) | 0X004 |
| …… | …… | …… | …… |
| 2047# | 0XC0209FFC-0XC0209FFF | 4(0X00004) | 0X007 |

## 三、 回答四张页表的逻辑地址，并绘制。

在物理地址上，四张页表占据 2M——2M+16k 的空间，在逻辑地址上需要平移 3G，应该在 3G+2M 之后的四个页表。
我们在 Memory 中找到这些页表，可以看到：

| Address | 0 - 3 | 4 - 7 | 8 - B | C - F |
|---|---|---|---|---|
| C0200000 | 00202027 | 00203027 | 00000000 | 00000000 |
| C0200010 | 00000000 | 00000000 | 00000000 | 00000000 |
| C0200020 | 00000000 | 00000000 | 00000000 | 00000000 |
| C0200030 | 00000000 | 00000000 | 00000000 | 00000000 |
| C0200040 | 00000000 | 00000000 | 00000000 | 00000000 |
| C0200050 | 00000000 | 00000000 | 00000000 | 00000000 |

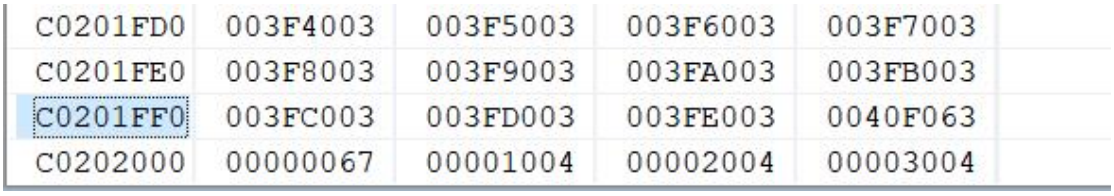
可以知道用户页表 0#的页框号是 0X202，1#的页框号是 0X203

| Address | 0 - 3 | 4 - 7 | 8 - B | C - F |
|---|---|---|---|---|
| C0200C00 | 00201023 | 00000000 | 00000000 | 00000000 |
| C0200C10 | 00000000 | 00000000 | 00000000 | 00000000 |
| C0200C20 | 00000000 | 00000000 | 00000000 | 00000000 |
| C0200C30 | 00000000 | 00000000 | 00000000 | 00000000 |
| C0200C40 | 00000000 | 00000000 | 00000000 | 00000000 |
| C0200C50 | 00000000 | 00000000 | 00000000 | 00000000 |

内核页表 768#的页框号是 0X201。
之后我们依次验证三个页表
0X201：

| 页号 | PBA | U/S | R/W | P |
|---|---|---|---|---|
| 0# | 0X000 | 0 | 1 | 1 |
| 1# | 0X001 | 0 | 1 | 1 |
| …… | …… | …… | …… | …… |
| 1023# | 0X40F(p_addr>>12) | 0 | 1 | 1 |

Memory 图像为：

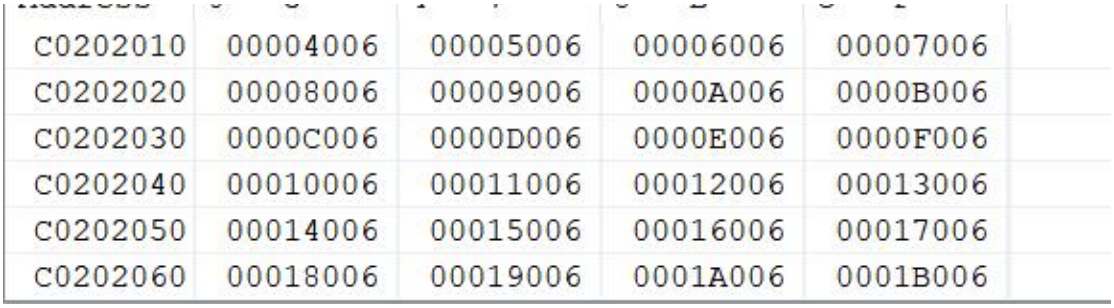


| | | | | |
|---|---|---|---|---|
| C0201FD0 | 003F4003 | 003F5003 | 003F6003 | 003F7003 |
| C0201FE0 | 003F8003 | 003F9003 | 003FA003 | 003FB003 |
| C0201FF0 | 003FC003 | 003FD003 | 003FE003 | 0040F063 |
| C0202000 | 00000067 | 00001004 | 00002004 | 00003004 |

可以看到 0X201 的最后一项是 0X40F。

0X202：

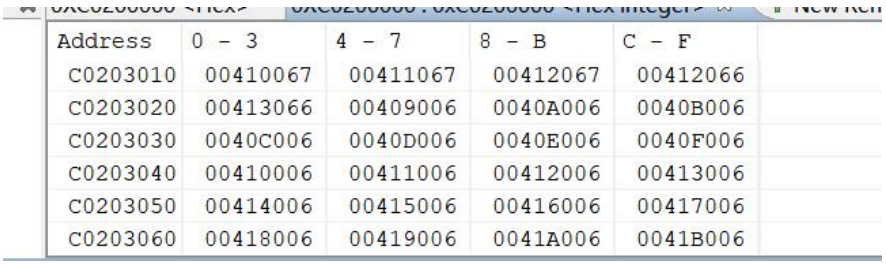
| 页号 | PBA | U/S | R/W | P |
|---|---|---|---|---|
| / | / | / | / | / |



| | | | | |
|---|---|---|---|---|
| C0202010 | 00004006 | 00005006 | 00006006 | 00007006 |
| C0202020 | 00008006 | 00009006 | 0000A006 | 0000B006 |
| C0202030 | 0000C006 | 0000D006 | 0000E006 | 0000F006 |
| C0202040 | 00010006 | 00011006 | 00012006 | 00013006 |
| C0202050 | 00014006 | 00015006 | 00016006 | 00017006 |
| C0202060 | 00018006 | 00019006 | 0001A006 | 0001B006 |

0X203：

| 页号 | PBA | U/S | R/W | P |
|---|---|---|---|---|
| 0# | / | / | / | / |
| 1# | 0X40C(x_caddr>>12) | 1 | 0 | 1 |
| 2# | 0X40D(x_caddr>>12+1) | 1 | 0 | 1 |
| 3# | 0X40E(x_caddr>>12+2) | 1 | 0 | 1 |
| 4# | 0X410(p_addr>>12+1) | 1 | 1 | 1 |
| 5# | 0X411(p_addr>>12+2) | 1 | 1 | 1 |
| 6# | 0X412(p_addr>>12+3) | 1 | 1 | 1 |
| …… | …… | …… | …… | …… |
| 1023# | 0X413(p_addr>>12+4) | 1 | 1 | 1 |



| Address | 0 - 3 | 4 - 7 | 8 - B | C - F |
|---|---|---|---|---|
| C0203010 | 00410067 | 00411067 | 00412067 | 00412066 |
| C0203020 | 00413066 | 00409006 | 0040A006 | 0040B006 |
| C0203030 | 0040C006 | 0040D006 | 0040E006 | 0040F006 |
| C0203040 | 00410006 | 00411006 | 00412006 | 00413006 |
| C0203050 | 00414006 | 00415006 | 00416006 | 00417006 |
| C0203060 | 00418006 | 00419006 | 0041A006 | 0041B006 |

也是相同的。

## 四、 内存图像：


| | |
|---|---|
| 0-1M | 保留区 |
| 1M-1.5M | 内核 |
| 1.5M-2M | 内核堆对象 |
| 2M-2M+4K | 0X200#物理页面 |
| 2M+4K-2M+8K | 0X201#物理页面 |
| 2M+8K-2M+12K | 0X202#物理页面 |
| 2M+12K-2M+16K | 0X203#物理页面 |
| …… | 其他页表 |
| 4M+48K-4M+60K | 代码段 |
| 4M+60K-4M+64K | PPDA 区 |
| 4M+64K-4M+76K | 数据段 |
| 4M+76K-4M+80K | 堆栈段 |
| …… | …… |