

实验三：UNIX V6++完整的进程图象

1. 实验目的

结合课程所学知识，通过编写一个简单的 C++代码，并在 UNIX V6++中编译和运行调试。通过查找关键地址单元的值，绘制出整个进程的图象，进而加深对 UNIX 进程图象的理解，特别是对逻辑地址空间与物理地址空间的理解。

2. 实验设备及工具

已配置好 UNIX V6++运行和调试环境的 PC 机一台。

3. 预备知识

- (1) UNIX V6++完整进程图象的构成。
- (2) 关于进程逻辑地址和物理地址的基本概念
- (3) UNIX V6++的运行和调试方法。

4. 实验内容

4. 1. 实验准备

开始本实验之前，做好如下的环境准备和知识准备是必须的。

(1) 可以直接采用实验二中可以运行的可执行程序 showStack.exe，也可以再重新编写一个简单的小程序，并按照实验二中的方法让它可以在 UNIX V6++中运行起来。

(2) 因为本实验要观察进程的页表，因此需将调试对象设置为 Kernel.exe。设置方法在实验二中已经详细介绍，这里不再赘述。

(3) 我们在 Eclipse 环境中调试程序时，看到的所有地址，都是逻辑地址，这一点需要读者理解并牢记。所以，在进行本实验之前，回看一下实验二，理解其中所有的地址值都是程序的逻辑地址将是很有帮助的。

4. 2. 找到进程完整的图象

(1) 调试运行你的可执行程序

因为需要查看进程图象，所以必须在进程执行过程中让它停下来，建议在 Process::Exit() 函数中设置断点，如图 1 所示，此时程序代码已执行完毕，程序马上要结束。

以调试模式启动 UNIX V6++。当 UNIX V6++启动成功，等待调试指令时，点击工具条上的 ，开始调试。UNIX V6++可能会多次暂停执行，可直接点击工具条上的  按钮，直到在 UNIX V6++环境下完成你的可执行程序的运行。以实验一中的的 showStack 程序为例，在键入“cd bin ↵”和“showStack.exe ↵”后，程序执行完输出语句，停在 Process::Exit() 函数中的断点处，如图 2 所示。

```

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_proc->p_pid);
    /* Reset Tracing flag */
    u.u_proc->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++ )
    {
        u.u_signal[i] = 1;
    }
}

```

图 1：设置合适的断点

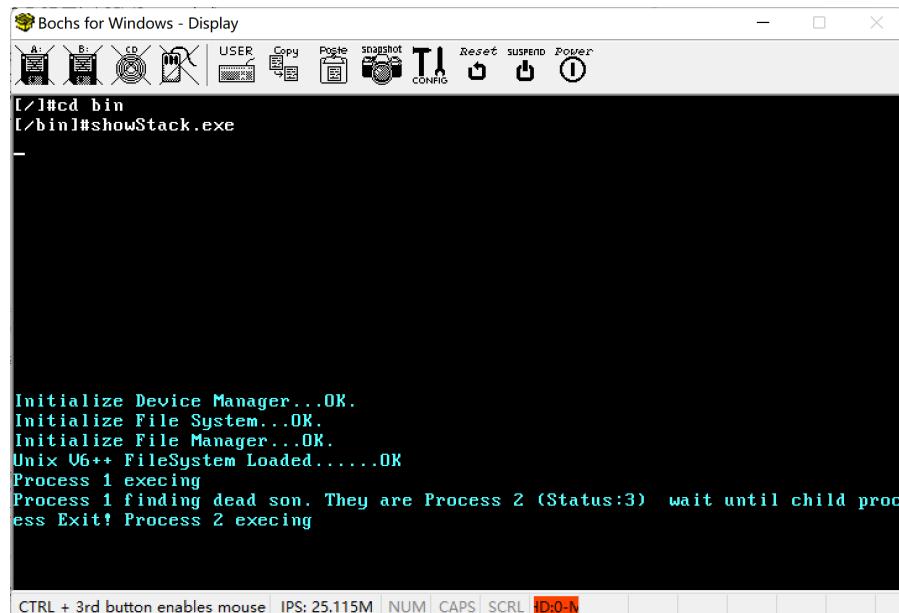


图 2：UNIX V6++停在断点时的状态

(2) 获取进程的 User 结构

我们知道，UNIX V6++的进程 User 结构逻辑地址是固定的，始终位于 3G ~ 3G+4M 部分的最后一页，即：0xC03FF000。这也是为什么 Kernel 中提供的 GetUser 函数通过返回逻辑地址 0xC03FF000 来达到找到当前进程 User 结构的目的（见代码 1）。

```

public:
    static const unsigned long USER_ADDRESS = 0x400000 - 0x1000 + 0xc0000000;
    /* 0xC03FF000 */

    User& Kernel:: GetUser()
    {
        return *(User*)USER_ADDRESS;
    }

```

代码 1

因此，此时利用该逻辑地址，我们可以找到进程的 User 结构。在 Memory 窗口中，使用<Hex integer>模式，可以看到从该地址开始的全部内容(见图 3)。对应 User 结构的定义，就可以确定其全部数据成员的值。如表 1 所示。

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	007FFB0	0000000E	00000000
C03FF040	C03FFFEC	00000000	00000000	00000009
C03FF050	00000000	00000000	00000000	00000000
C03FF060	00000000	00000000	00000000	00000000
C03FF070	00000000	00000000	00000000	00000000
C03FF080	00000000	00000000	00000000	00000000

图 3：在 Memory 中查看 User 结构

表 1 中我们只给出了一小部分与进程图象相关的地址信息的值。其他部分的值有兴趣的读者可尝试自行整理。从中我们可以看到，代码段的起始地址为 0x00401000，这与我们在实验一中看到的 addDemo1 的代码起始位置是符合的。

表 1：User 结构各数据成员的值

变量名称	含义	值
Process* u_procp	proc 结构的逻辑地址	0xC0119600
MemoryDescriptor u_MemoryDescriptor		
PageTable* m_UserPageTableArray	相对映射表首地址	0xC0208000
unsigned long m_TextStartAddress	代码段起始逻辑地址	0x00401000 = 4M+4K
unsigned long m_TextSize	代码段长度	0x00003000 = 12K
unsigned long m_DataStartAddress	数据段起始逻辑地址	0x00404000 = 4M+16k
unsigned long m_DataSize	数据段长度	0x00003000 = 12K
unsigned long m_StackSize	栈段长度	0x00001000 = 4K

(3) 获取进程的 Proc 结构

此外，表 1 中还可以获得的一个重要信息是 u_procp 的值，即进程 proc 结构的逻辑地址：0xC0119600，使用相同的方法，我们可以整理出表 2 中进程 proc 结构中所有数据成员的值。从中我们可以看到进程的 ID 号，父进程的 ID 号，进程的调度状态，进程图象的特征标志等。这里需要特别强调的是，p_addr 中的值是进程的物理地址，所以这里的 0x0040F000 是 User 结构的物理地址，而不是我们前面提到的 User 结构的逻辑地址 0xC03FF000。

(4) 获取进程代码段的 Text 结构

从表 2 中，可以得到的另一个重要的地址信息是 p_textp 指针的值，注意，这里也是该进程代码段 Text 结构的逻辑地址 0xC011AE94。由此，通过 Memory 窗口，我们可以获取到

该 Text 结构的值如表 3 所示。这里同样需要强调的是，`x_caddr` 中是代码段在内存的物理地址，所以这里是 0x0040C000，而不是我们前面提到的 0x00401000。

Address	0 - 3	4 - 7	8 - B	C - F
C0119600	00000000	00000002	00000001	0040F000
C0119610	000005000	C011AE94	00000003	00000001
C0119620	00000065	00000019	00000000	00000000
C0119630	00000000	00000000	C0120DA0	00000000
C0119640	00000000	00000000	FFFFFFFFFF	00000000
C0119650	00000000	00000000	00000000	00000000
C0119660	00000000	00000000	00000000	00000000
C0119670	00000000	00000000	00000000	00000000

图 4: 在 Memory 中查看 Proc 结构

表 2: Proc 结构各数据成员的值

变量名称	含义	值
<code>short p_uid</code>	用户 ID	0
<code>int p_pid</code>	进程标识数, 进程编号	2
<code>int p_ppid</code>	父进程标识数	1
<code>unsigned long p_addr</code>	进程图象可交换部分在内存中的物理地址	0x0040F000
<code>unsigned int p_size</code>	进程图象可交换部分的大小	0x00005000 - 20K
<code>Text* p_textp</code>	指向代码段 Text 结构的指针 (逻辑地址)	0xC011AE94
<code>ProcessState p_stat</code>	进程调度状态	3 = SRUN
<code>int p_flag</code>	进程图象标志	1 = SLOAD
<code>int p_pri</code>	进程优先数	65
<code>int p_cpu</code>	进程使用 CPU 的频繁程度	19
<code>int p_nice</code>		0
<code>int p_time</code>		0
<code>unsigned long p_wchan</code>		0

Address	0 - 3	4 - 7	8 - B	C - F
C011AE90	00010001	00004670	0040C000	00003000
C011AEA0	C011EC00	00010001	00000000	00000000
C011AEB0	00000000	00000000	00000000	00000000
C011AEC0	00000000	00000000	00000000	00000000
C011AED0	00000000	00000000	00000000	00000000
C011AEE0	00000000	00000000	00000000	00000000
C011AEF0	00000000	00000000	00000000	00000000
C011AF00	00000000	00000000	00000000	00000000

图 5: 在 Memory 窗口中观察 Text 结构

表 3: Text 结构数据成员的值

变量名称	含义	值
<code>int x_daddr</code>	代码段在盘交换区的地址 (盘块号)	00004670
<code>unsigned long x_caddr</code>	代码段在内存的起始地址 (物理地址)	0x0040C000
<code>unsigned int x_size</code>	代码段大小	00003000 = 12K

<code>Inode*</code>	<code>x_iptr</code>		
<code>unsigned short</code>	<code>x_count</code>	共享代码段的进程数量	1
<code>unsigned short</code>	<code>x_ccount</code>	共享代码段且图象在内存的进程数量	1

至此，我们找到了进程图象的两大部分：进程图象的可交换部分和代码段在逻辑地址空间和物理地址空间的分布，见表 4。由此表，读者可根据课程所学知识，绘制出进程的相对虚实地址映射表和物理页表，并通过后续实验，验证是否正确。

表 4：进程图象完整信息

变量名称	逻辑地址	物理地址	大小
代码段	0x00401000	0x0040C000	12K
可交换部分	0xC0119600	0x0040F000	20K
ppda 区	0xC0119600	0x0040F000	4K
数据段	0x00404000		12K
堆栈段	0x00001000		4K

4.3. 找到进程完整的页表

(1) 进程的相对虚实地址映射表

通过表 1 中的数据我们知道，进程的相对虚实地址映射表位于 0xC0208000 起始的两个 4K 的页中，注意，这里 0xC0208000 也是逻辑地址。通过在 Memory 窗口中观察这些地址单元中的值，我们将进程相对虚实地址映射表绘制在表 5 中，其中黄色部分请读者补全，并与你在实验 4.2 中绘制的相对虚实地址映射表相比对，看看是否正确。

表 5：进程的相对虚实地址映射表

	地址	含义	值	
		高 20 位页框号	低 12 位标志 (后 3 位 u/s r/w p)	
0#	0xC0208000 ~ 0xC0208003		004 (0000 0000 0100)	
	
1024#	0xC0209000 ~ 0xC0209003		004 (0000 0000 0100)	
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)	
1030#	0xC0209018 ~ 0xC020901B		007 (0000 0000 0111)	
	0xC020901C ~ 0xC020901F		004 (0000 0000 0100)	数据段
	
	0xC0209FF8 ~ 0xC0209FFB		004 (0000 0000 0100)	
2047#	0xC0209FFC ~ 0xC0209FFF		007 (0000 0000 0111)	

(2) 进程的物理页表

进程的物理页表包括页目录，一张核心页表和两张物理页表。请读者首先思考，这四张页表的逻辑地址是什么，再利用 Memory 窗口找到该逻辑地址的相应位置，尝试观察四张页表的全部内容，并与你在实验 4.2 中绘制的物理页表相比对，看看是否正确。

5. 实验报告要求

实验报告需包含以下内容：

- (1) (1 分) 完成实验 4.1~4.2，依照实验指导的步骤，获取进程 User 结构、Proc 结构和 Text 结构的内容，截图或绘制表格说明，并总结出在 UNIX V6++中获取进程的代码段和可交换部分起始位置的逻辑地址和物理地址的方法。
- (2) (1 分) 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。
- (3) (1 分) 完成实验 4.3，获取完整的进程物理页表，仿照表 5，自行绘制表格说明。
- (4) (1 分) 根据实验结果，绘制进程完整的图象，包括进程图象在内存中的位置，相对虚实地址映射表、物理页表在内存的位置和内容，可参考讲义图 4.30。