

2020 春

LINUX OPERATING SYSTEM

YANG

LINUX操作系统（双语）



番外篇 5 进程内存空间

一、逻辑地址和物理地址

1. 先编写一个完整而简单的不能再简单的代码

```
1 //sample.c
2 int sum(int x, int y){
3     return x+y;
4 }
5
6 int main(){
7     sum(2,3);
8     return 0;
9 }
```

2. 使用gcc进行编译和汇编但不链接成可执行文件，得到目标代码sample.o。

```
1 $ gcc -g -c sample.c
```

3. 使用objdump对sample.o进行反编译查看其汇编代码。

```
1 $ objdump -d sample.o
```

在我的Deepin15.11环境下，显示结果如下所示：

```
1 sample.o:      文件格式 elf64-x86-64
2
3 Disassembly of section .text:
4
5 0000000000000000 <sum>:
6 int sum(int x, int y){
7     0:  55                push    %rbp
8     1:  48 89 e5          mov     %rsp,%rbp
9     4:  89 7d fc          mov     %edi,-0x4(%rbp)
10    7:  89 75 f8          mov     %esi,-0x8(%rbp)
11    return x+y;
12    a:  8b 55 fc          mov     -0x4(%rbp),%edx
13    d:  8b 45 f8          mov     -0x8(%rbp),%eax
14   10:  01 d0            add     %edx,%eax
15 }
16   12:  5d                pop     %rbp
17   13:  c3                retq
18
19 0000000000000014 <main>:
20 int main(){
21   14:  55                push    %rbp
```

物理地址=基址+逻辑地址 物理地址
0x2000

```

22      15:  48 89 e5                mov     %rsp,%rbp
23      sum(2,3);
24      18:  be 03 00 00 00          mov     $0x3,%esi
25      1d:  bf 02 00 00 00          mov     $0x2,%edi
26      22:  e8 00 00 00 00          callq   27 <main+0x13>
27      return 0;
28      27:  b8 00 00 00 00          mov     $0x0,%eax
29      2c:  5d                      pop     %rbp
30      2d:  c3                      retq

```

4. 请特别关注上述结果中从第7行开始的 绿色 的数字，从0开始递增，直到2d为止。这些数字即为程序的 逻辑地址 (Logical Address)。这种地址也称 虚拟地址 (Virtual Address)，对每一个程序 (Program) 而言总是从0开始，每一个地址对应一条指令，我们把第7~10行单独拿出来分析一下 (所有的数字都是以16进制[即4个二进制]书写)：

```

1      0:  55                      push    %rbp
2      1:  48 89 e5                mov     %rsp,%rbp
3      4:  89 7d fc                mov     %edi,-0x4(%rbp)
4      7:  89 75 f8                mov     %esi,-0x8(%rbp)

```

- 0是刚才我们提到的每个程序的起始逻辑地址；
- 0x55是 `push %rbp` 汇编语句对应的机器指令，正好1个字节，因此第2条指令的逻辑地址等于第1条指令逻辑地址+1=1；
- 0x48 89 e5这3个字节的机器指令对应的汇编语句是 `mov %rsp, %rbp`，我不打算去解释这些汇编语句的作用，我要你们现在观察的逻辑地址的递增规律：因为这条指令占了3个字节，所以下一条指令的逻辑地址等本条指令逻辑地址+3=4；
- 那么后面的 规律你应该get到了。

5. 逻辑地址0x0~0x13对应的是sum函数的所有指令，0x14~0x2d是main函数的所有指令，在逻辑地址0x22处执行了 `call` 指令，是要调用sum函数了，但此处并没有指明sum的逻辑地址，因为 这份目标代码还未link，所以很多的地址有可能还会发生变化。那么我们去掉-g-c参数，用gcc对sample.c执行编译链接形成可执行文件sample，再用objdump对其反编译。

```

1  $ gcc sample.c -o sample
2  $ objdump -d sample
3  #输出部分结果如下
4  0000000000000060 <sum>:
5  660:  55                      push    %rbp
6  661:  48 89 e5                mov     %rsp,%rbp
7  664:  89 7d fc                mov     %edi,-0x4(%rbp)
8  667:  89 75 f8                mov     %esi,-0x8(%rbp)
9  66a:  8b 55 fc                mov     -0x4(%rbp),%edx
10 66d:  8b 45 f8                mov     -0x8(%rbp),%eax
11 670:  01 d0                  add     %edx,%eax
12 672:  5d                      pop     %rbp
13 673:  c3                      retq
14
15 0000000000000074 <main>:
16 674:  55                      push    %rbp
17 675:  48 89 e5                mov     %rsp,%rbp
18 678:  be 03 00 00 00          mov     $0x3,%esi

```

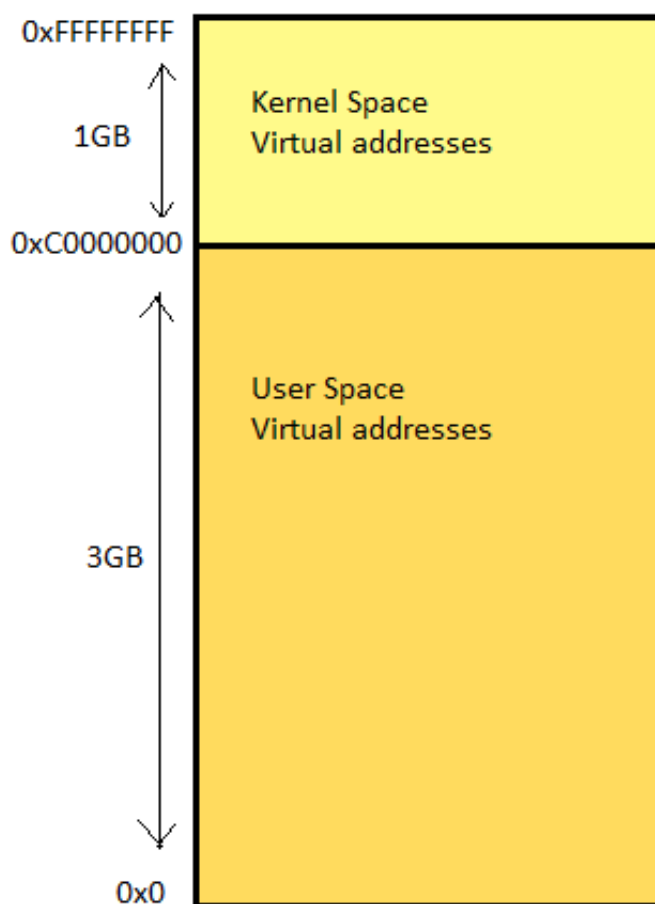
19	67d:	bf 02 00 00 00	mov \$0x2,%edi
20	682:	e8 d9 ff ff ff	callq 660 <sum>
21	687:	b8 00 00 00 00	mov \$0x0,%eax
22	68c:	5d	pop %rbp
23	68d:	c3	retq
24	68e:	66 90	xchg %ax,%ax
25			

不难发现，sum的起始逻辑地址变为了0x660，main的起始逻辑地址变成了0x674，在0x682处，我们找到了 `call 660<sum>` 指令，旨在调用sum函数，即跳转到sum的起始地址。

6. **物理地址**是内存单元看到的地址，是指令和数据真实的内存地址，而**逻辑地址**是面向程序而言的。CPU在执行指令的时候，比如上面提到的 `call 660`，会先将逻辑地址经一个MMU (Memory Managment Unit) 的硬件设备转换成物理地址，然后再进行跳转。

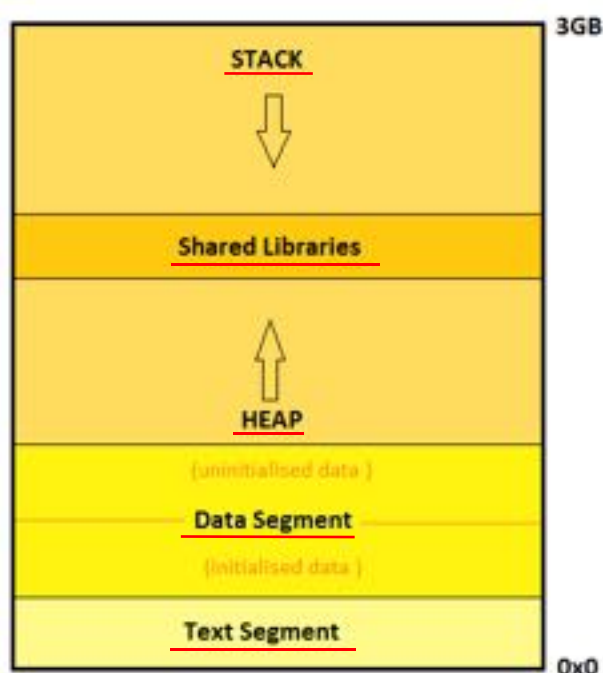
二、进程的内存映像

以 **32 位机器**为例，地址总线是32位，可寻址的最大内存空间是 2^{32} Bytes，即4GBBytes。每一个运行的进程都可以获得一个4GB的逻辑地址空间，这个空间被分成两个部分：**内核空间**和**用户空间**，其中用户空间分配到从0x00000000到0xC0000000共3GB的地址，而内核空间分配了0xC0000000到0xFFFFFFFF高位的1GB地址（如下图所示）。



而用户进程的虚拟空间内容就是我们之前学过的进程内存映像，它一共包括四个部分的内容（下图）：

- text代码段：是只读的，存放要执行的指令，你们在上面已经看到了，自己找一下text section字眼；
- data数据段：存放全局或静态变量；
- heap堆：运行时（Run time）分配的内存（如用malloc函数申请内存）；
- stack栈：存放局部变量和函数返回地址。



因为堆栈段的内容要程序运行起来才能观察到，我们先看一下data段的情况，我们再编写一个sample2.c，代码如下：

```

1  //sample2.c
2  int global_var = 5;
3
4  int main(){
5      static int static_var = 6;
6      return 0;
7  }
```

然后进行一波操作：

```

1  $ gcc -g -c sample2.c
2  $ objdump -s -d sample2
3  #部分结果如下
4  sample.o:      文件格式 elf64-x86-64
5
6  Contents of section .text:
7  0000 554889e5 b8000000 005dc3      UH.....].
8  Contents of section .data:
9  0000 05000000 06000000      .....
10 ...
11 ...
12 ...
13 Disassembly of section .text:
14
```

```

15  0000000000000000 <main>:
16      0:   55                push   %rbp
17      1:  48 89 e5             mov     %rsp,%rbp
18      4:  b8 00 00 00 00       mov     $0x0,%eax
19      9:   5d                pop     %rbp
20     a:   c3                retq

```


section.data是data段，里面已经写好了global_var和static_var的初始值；**One more thing**: 你们再看一看section.text那一串16进制数和下面main函数的一串16进制数是不是很匹配？上面的就是main函数指令集合在text段中的存放形式，下方的是objdump为了方便我们阅读而重新排了版。

我们向sample2.c再添加一个局部变量，并使用malloc在运行时申请一段内存空间，另存为sample3.c，代码如下：

```

1  //sample3.c
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <stdio.h>
5
6  int global_var = 5;
7
8  int main(){
9      static int static_var = 6;
10     int local_var = 7;           //from STACK
11     int*p = (int*)malloc(100);    //from HEAP 100 bytes
12
13     //Be careful: we must use %lx to show a 64bits address!!
14     printf("the global_var address is %lx\n", &global_var);
15     printf("the static_var address is %lx\n", &static_var);
16     printf("the local_var address is %lx\n", &local_var);
17     printf("the address which the p points to%lx\n", p);
18
19     free(p);
20     sleep(1000); //We need watch the process state, so let it sleep deeply.
21     return 0;
22 }

```



然后对其进行正常编译链接，再运行它，它很快会进入sleep状态，

```

1  $ gcc sample3.c -o sample3
2  $ ./sample3
3  #输出结果如下
4  the global_var address is 55b3c179d048
5  the static_var address is 55b3c179d04c
6  the local_var address is 7ffdb0f3704
7  the address which the p points to55b3c2937010

```

我们重新打开一个终端窗口，执行如下指令

```

1  $ ps -e
2  18288 pts/2    00:00:00 sample3
3  18294 pts/1    00:00:00 ps
4  ##我们把sample3的pid: 18288记下来，每个机器的pid都不一样，请注意这一点！

```

```

5  $ cat /proc/18288/maps
6  ##输出结果如下
7  55b3c159c000-55b3c159d000 r-xp 00000000 08:01 833578      text segment
   /home/youngyt/Desktop/L12.MemoryManagement/sample3
8  55b3c179c000-55b3c179d000 r--p 00000000 08:01 833578
   /home/youngyt/Desktop/L12.MemoryManagement/sample3
9  55b3c179d000-55b3c179e000 rw-p 00001000 08:01 833578      data segment
   /home/youngyt/Desktop/L12.MemoryManagement/sample3
10 55b3c2937000-55b3c2958000 rw-p 00000000 00:00 0          [heap]
11 7f2b0f9c8000-7f2b0fb5d000 r-xp 00000000 08:01 922700 /usr/lib/x86_64-linux-
   gnu/libc-2.24.so
12 7f2b0fb5d000-7f2b0fd5d000 ---p 00195000 08:01 922700 /usr/lib/x86_64-linux-
   gnu/libc-2.24.so
13 7f2b0fd5d000-7f2b0fd61000 r--p 00195000 08:01 922700 /usr/lib/x86_64-linux-
   gnu/libc-2.24.so
14 7f2b0fd61000-7f2b0fd63000 rw-p 00199000 08:01 922700 /usr/lib/x86_64-linux-
   gnu/libc-2.24.so
15 7f2b0fd63000-7f2b0fd67000 rw-p 00000000 00:00 0
16 7f2b0fd67000-7f2b0fd8a000 r-xp 00000000 08:01 922248 /usr/lib/x86_64-linux-
   gnu/ld-2.24.so
17 7f2b0ff69000-7f2b0ff6b000 rw-p 00000000 00:00 0
18 7f2b0ff6b000-7f2b0ff6d000 rw-p 00000000 00:00 0
19 7f2b0ff6d000-7f2b0ff6f000 r--p 00023000 08:01 922248 /usr/lib/x86_64-linux-
   gnu/ld-2.24.so
20 7f2b0ff6f000-7f2b0ff71000 rw-p 00024000 08:01 922248 /usr/lib/x86_64-linux-
   gnu/ld-2.24.so
21 7f2b0ff71000-7f2b0ff73000 rw-p 00000000 00:00 0
22 7ffdab0d5000-7ffdab0f7000 rw-p 00000000 00:00 0          [stack]
23 7ffdab13b000-7ffdab13e000 r--p 00000000 00:00 0          [vvar]
24 7ffdab13e000-7ffdab140000 r-xp 00000000 00:00 0          [vdso]
25 ffffffff600000-ffffffff601000 r-xp 00000000 00:00 0          [vsyscall]
26

```

每列的标题分别是 **Address**, **Permissions**, **Offset**, **Device**, **iNode**, **Pathname**。

- ✓ Address: the start address and the end address in the process that the mapping occupies.
- ✓ Permissions: *readable, writable, executable, private, shared*.
 - Offset: the offset in the file where the mapping begins. Not every mapping is mapped from a file, so the value of offset is zero.
 - Device: In the form “major number : minor number” of the associated device, as in, the file from which this mapping is mapped from. Again, the mappings which are not mapped from any file, value is 00:00.
 - iNode: i-node of the related file.
- ✓ Pathname: The path of the related file. Its blank in case there is no related file.

现在你们的任务是找到这个进程在内存中四个部分所在的地址区间（我以行号标注）：

- text：第7行
- data：第9行
- heap：第10行
- stack：第22行

A little task: 从sample3的执行输出中，请将代码中定义的全局/静态变量、局部变量和动态内存申请的内存地址分别核实他们来自进程的哪个内存部分。

The End