1. sigaction函数

sigaction函数的功能是检查或修改与指定信号相关联的处理动作（可同时两种操作）。

他是POSIX的信号接口，而signal()是标准C的信号接口(如果程序必须在非POSIX系统上运行，那么就应该使用这个接口)

int sigaction(int signum, const struct sigaction \*act,

struct sigaction \*oldact);

signum参数指出要捕获的信号类型，act参数指定新的信号处理方式，oldact参数输出先前信号的处理方式（如果不为NULL的话）。如果参数oldact不是NULL指针，则原来的信号处理方式会由此结构sigaction返回。函数成功返回0，出错则返回-1，错误原因存于error中。

其中比较重要的是结构体sigaction的定义。

1. struct sigaction定义

struct sigaction {

void (\*sa\_handler)(int);

void (\*sa\_sigaction)(int, siginfo\_t \*, void \*);

sigset\_t sa\_mask;

int sa\_flags;

void (\*sa\_restorer)(void);

}

sa\_handler此参数和signal()的参数handler相同，代表新的信号处理函数

sa\_mask 用来设置在处理该信号时暂时将sa\_mask 指定的信号集搁置

sa\_flags 用来设置信号处理的其他相关操作，下列的数值可用。

SA\_RESETHAND：当调用信号处理函数时，将信号的处理函数重置为缺省值SIG\_DFL

SA\_RESTART：如果信号中断了进程的某个系统调用，则系统自动启动该系统调用

SA\_NODEFER ：一般情况下， 当信号处理函数运行时，内核将阻塞该给定信号。但是如果设置了 SA\_NODEFER标记， 那么在该信号处理函数运行时，内核将不会阻塞该信号

SA\_SIGINFO 提供附加信息，一个指向siginfo结构的指针以及一个指向进程上下文标识符的指针

当设置SA\_SIGINFO时，void \*sa\_sigaction项将作为一个替代的信号处理程序使用。

本次作业中对该结构体的赋值较为简单，只需要初始化一个结构体，修改其sa\_handler为我们将用来替换的处理函数入口即可。

1. mprotect函数

在Linux中，mprotect()函数可以用来修改一段指定内存区域的保护属性。

#include <unistd.h>

#include <sys/mmap.h>

int mprotect(const void \*start, size\_t len, int prot);

mprotect()函数把自start开始的、长度为len的内存区的保护属性修改为prot指定的值。

prot可以取以下几个值，并且可以用“|”将几个属性合起来使用：

1）PROT\_READ：表示内存段内的内容可读

2）PROT\_WRITE：表示内存段内的内容可写

3）PROT\_EXEC：表示内存段中的内容可执行

4）PROT\_NONE：表示内存段中的内容无法访问

需要指出的是，指定的内存区间必须包含整个内存页（4K）。区间开始的地址start必须是一个内存页的起始地址，并且区间长度len必须是页大小的整数倍。

如果执行成功，则返回0；如果执行失败，则返回-1，并且设置errno变量，说明具体因为什么原因造成调用失败。错误的原因主要有以下几个：

1. EACCES 该内存不能设置为相应权限。
2. EINVAL start 不是一个有效的指针，指向的不是某个内存页的开头。

3）ENOMEM内核内部的结构体无法分配。

4）ENOMEM进程的地址空间在区间 [start, start+len] 范围内是无效，或者有一个或多个内存页没有映射。

我们最需要关心的是，如果调用进程内存访问行为侵犯了这些设置的保护属性，内核会为该进程产生 SIGSEGV （Segmentation fault，段错误）信号，并且终止该进程。所以我们需要用sigaction捕获的信号量是SIGSEGV。

1. 内存分配

题目中要求分配整页整页的连续内存。但是使用malloc分配的内存有两个问题：第一，地址大概率不连续。第二，基地址大概率不为页面的开头。

方法：使用 mmap 系统调用来绕过 malloc 并直接从 Linux 内核中分配页面对齐内存。mmap通过映射 /dev/zero 来分配内存页。内存将被初始化为可读和可写模式。代码如下：

int fd = open (“/dev/zero”, O\_RDONLY);

int\* memory = mmap (NULL, page\_size, PROT\_READ | PROT\_WRITE, MAP\_PRIVATE, fd, 0);

close (fd);

之后可调用mprotect把该内存区域设置为只读。

五、代码

#include <fcntl.h>

#include <signal.h>

#include <stdio.h>

#include <string.h>

#include <sys/mman.h>

#include <sys/stat.h>

#include <sys/types.h>

#include <unistd.h>

static int alloc\_size;

static int\* memory;

int time= 1;

void segv\_handler (int signal\_number)

{

printf("memory accessed!\n");

printf(" This time addr 0x%x is accessed \n",memory+1024\*(time-1));

mprotect(memory, getpagesize()\*time, PROT\_READ | PROT\_WRITE);

time++;

}

int main ()

{

int fd,i;

struct sigaction sa;

/\* 初始化segv\_handler为SIGSEGV的句柄。\*/

memset(&sa,0,sizeof(sa));

sa.sa\_handler = &segv\_handler;

sigaction(SIGSEGV, &sa, NULL);

/\* 使用映射/dev/zero分配内存页。最初映射的内存为只写。\*/

alloc\_size = 64\*getpagesize();//256KB

fd = open("/dev/zero", O\_RDONLY);

memory = mmap(NULL, alloc\_size, PROT\_WRITE, MAP\_PRIVATE, fd, 0);

close(fd);

/\* 写页来获得一个私有复制。 \*/

memory[0] = 0;

/\* 使内存为read-only。 \*/

mprotect(memory, alloc\_size, PROT\_READ);

/\* 写分配内存区域。 \*/

//memory[0] = 1;

for(i=0;i<64;i++)

memory[1024\*i] = i;

/\* 所有工作都结束；unmap内存映射。 \*/

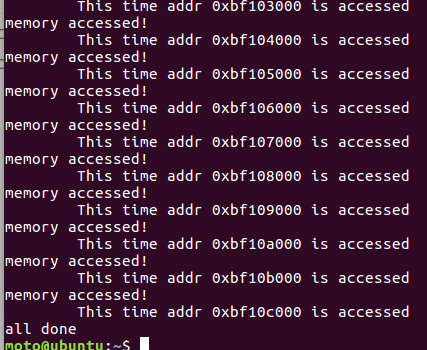
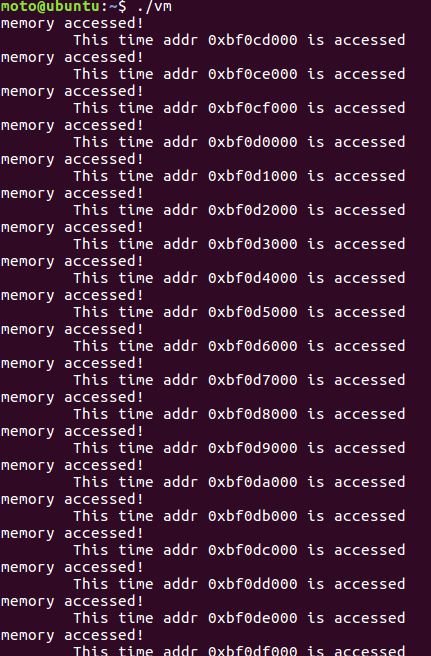
printf("all done\n");

munmap(memory, alloc\_size);

return 0;

}

六、测试结果



截图省略了中间一些部分的打印。

总体来说我们可以看到，程序打印了64次对被设置为只读内存进行写操作的访问地址，开始地址为0xbf0cd000，结束地址为bf10c000，跨度为63个页面，实现了对64个页面的访问。

另外需要说明的是，当程序触发访问只读保护内存的写操作时，如果在处理程序中不将该访问地址修改为可读可写的话，程序返回到执行位置时会继续陷入触发SIGSEGV的死循环。