Fundamentos de Sistemas de Operação

MIEI 2017/2018

Laboratory session 6

Overview

Information about Linux memory map of a process. Memory mapped files. Static and dynamic linking.

Memory map of a Linux process

Let's start by running a program (`mem.c`) adapted from one that appears in chapter 2 of the *OSTEP book.*

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

int global;   // global var not inicialized
int vglob = 3;      // inicialized global var
const int cglob = 10;      // global constant

int main(int argc, char *argv[])
{
    long pid;  // local var (on stack)
    int *p;    // local var for pointer

    if (argc != 2) {
       fprintf(stderr, "usage: %s <value>\n", argv[0]);
       exit(1);
    }
    p = (int*)malloc(sizeof(int));  // malloc'd memory is on "heap"
    assert(p != NULL);

    *p = atoi(argv[1]);
    pid = (long)getpid();
    global = vglob = pid; // all vars written with process ID

    printf("(pid:%ld) addr of main:   %lx\n", pid, (unsigned  long) main);
    printf("(pid:%ld) addr of printf: %lx\n", pid, (unsigned  long) printf);
    printf("(pid:%ld) addr of getpid: %lx\n", pid, (unsigned  long) getpid);
    printf("(pid:%ld) addr of p:      %lx\n", pid, (unsigned long) &p);
    printf("(pid:%ld) addr of argv:   %lx\n", pid, (unsigned long) argv);
    printf("(pid:%ld) addr stored in p: %lx\n", pid, (unsigned long) p);
    printf("(pid:%ld) addr of global: %lx\n", pid, (unsigned long) &global);
    printf("(pid:%ld) addr of cglob:  %lx\n", pid, (unsigned long) &cglob);
    printf("(pid:%ld) addr of vglob:  %lx\n", pid, (unsigned long) &vglob);

    while ( *p > 0 ) {
        *p = *p - 1;
        sleep(10);
        printf("(pid:%ld) value of p: %d\n", pid, *p);
    }
    return 0;
}
```

This program exhibits several memory objects that live in different memory segments of a process address space. Run the program several times and observe that virtual addresses are always (more or less) the same[1]. Run several instances of the program simultaneously and see that distinct processes emit the same virtual addresses, although these must correspond to distinct physical addresses (at least some of them, such as the ones holding written variables).

Open another terminal and, while running the process above, give the command `pmap` *<pid>* (whose information comes from the Linux kernel using the pseudofile `/proc/`*<pid>*`/maps`)

[1] Linux kernel uses, by security reasons, a technique called Address Space Layout Randomization (ASLR). In order to prevent an attacker from jumping to know locations in memory, ASLR randomly changes the base address of data and stack segments. According to this, some virtual addresses will not be exactly the same in each process instance, but just similar.

Using the values printed by the program and the output of the command, you can get a table like the next one (a different system can get you a different memory map). Complete or adapt the following table for you case:

begin add	size	permissions	content	prg obj	addresses
08048000	4K	r-x--	(.code)	main, cglobal, etc…	
08049000	4K	rw---	(.data, .bss)	global, vglobal	
08525000	132K	rw---	(heap)	pointed by p	
b75c9000	4K	rw---	[anon]		
b75ca000	1692K	r-x--	libc-2.19.so		
b7771000	8K	r----	libc-2.19.so	*dynamic shared libraries*	
b7773000	4K	rw---	libc-2.19.so		
b7774000	12K	rw---	[anon]		
b7782000	12K	rw---	[anon]		
b7785000	4K	r-x--	(vdso)		
b7786000	8K	r----	(vvar)		
b7788000	128K	r-x--	ld-2.19.so	*dynamic linker loader*	
b77a8000	4K	r----	ld-2.19.so		
b77a9000	4K	rw---	ld-2.19.so		
bfb10000	132K	rw---	[stack]	p, argv	
TOTAL:	2152K				

Identify the several program segments and relate with your C program object's addresses and pages's RWX permissions. Notice the memmaped loader (ld.so) and libc files and respective data segments.

The system call getrlimit

Any program can get information about its several resources limitations. Some resources can be changed within some hard limits imposed by the OS (and its administrator). Consult the manual page of the system call `getrlimit`. Consider the following C program:

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
#include <sys/time.h>
#include <sys/resource.h>

void Getrlimit( int limitType, struct rlimit *r) {
        int res = getrlimit( limitType, r);
        if(res < 0){ perror("getrlimit"); exit(1);}
}

int main(int argc, char *argv[]) {
    struct rlimit r;

    Getrlimit( RLIMIT_AS, &r);
    printf("Maximum size of process virtual addres space; soft limit = %u, hard limit = %u\n",
               (unsigned int)r.rlim_cur, (unsigned int)r.rlim_max);

    Getrlimit( RLIMIT_DATA, &r);
    printf("Maximum size of process's data segment(intialized, uninitiliazed, heap); soft limit =
%u, hard limit = %u\n",
               (unsigned int)r.rlim_cur, (unsigned int)r.rlim_max);

    Getrlimit( RLIMIT_STACK, &r);
    printf("Maximum size of process stack; soft limit = %u, hard limit = %u\n",
               (unsigned int)r.rlim_cur, (unsigned int)r.rlim_max);
    return 0;
}
```

This program prints several of its virtual memory limits. Compile and run it and compare the results obtained with the table of last section.

Try to declare a local array in main function with more than 8MB, like:

`char a[9*1024*1024];` or `char a = alloca(9*1024*1024);`

Can you execute that program? Why? Try to solve your problem by reading *ulimit* and *setrlimit* manuals…

Testing mmap

Read the manual page of *mmap*. The given *mmcat.c* program copies the contents of a file to the process' standard output, similarly to the *cat* command. Try such program.

```
#include <stdlib.h>
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/mman.h>
#include <sys/stat.h>

void fatal_error(char *str){
  perror(str);
  exit(1);
}

/* mmapcopy - uses mmap to copy file fd to stdout
 */
void mmapcopy(int fd, int size) {
    char *bufp; /* ptr to memory mapped VM area */
    int n;

    bufp = mmap(NULL, size, PROT_READ, MAP_PRIVATE, fd, 0);
    if( bufp == MAP_FAILED) fatal_error("mmap ");

    n = write(1, bufp, size);
    if( n != size) fatal_error("write ");
}

int main(int argc, char *argv[]) {
    struct stat stat;
    int fd;

    /* check for required command line argument */
    if (argc != 2) {
        printf("usage: %s <filename>\n", argv[0]);
        exit(1);
    }

    /* copy the input argument to stdout */
    fd = open(argv[1], O_RDONLY, 0);
    if( fd < 0) fatal_error("error in open");

    if( fstat(fd, &stat) != 0) fatal_error("error in fstat");

    mmapcopy(fd, stat.st_size);
    return 0;
}
```

After that, implement a program that copies files (like previous `copy.c`) but without using read/write system calls. You should use *mmap* to transform the copy of files into a copy of memory.

After testing your program, note that there are files of all sizes, and you can't map all of them to your process memory address space. Then, for the final version of you program, take that into account. Try using as argument to your program the maximum size of your memory buffer, like in previous `copy.c` and `fcopy.c`. Your program should be used like this:

`mmcopy` *size file1 file2*

Notice that:

- the used memory buffer must be a multiple of page size for your architecture. Use `sysconf(_SC_PAGE_SIZE)` to get that page size;
- for changes to the mapped memory of the write be saved, the file to write must be resized to the final size. Use `ftruncate` for that;
- and the file must be mapped with `PROT_WRITE` and `MAP_SHARED`.

Compare this program performance with *copy.c* and *fcopy.c*, for the same buffer sizes.

Static and dynamic linking

1. Comparing statically- and dynamically- linked executables

Consider any one of the programs used before. Usually, by default, dynamic linking of shared libraries is used by development tools. Make a copy of the current executable file and compile again the program forcing the use of static linking. Example:

```
gcc -static -o mmcat.static mmcat.c
```

Then compare de programs sizes:

```
size mmcat.dynamic
size mmcat.static
```

Try *man ldd* and then execute the following commands to see the used dynamic libraries:

```
ldd mmcat.dynamic
ldd mmcat.static
```

Why the results are different?

Execute each program seeing its use of system calls with strace[2] command:

```
strace mmcat.dynamic
strace mmcat.static
```

Verify the differences and observe the *mmap* calls done in the dynamic version by the dynamic linker loader to map *libc* library to the process as needed.

2. Generation of static libraries

Consider the several files that simulates a set of usefull functions that we want to build as a binary code library: *util_file.c*, *util_math.c* , *util_net.c*.

Compile all files as usually and after that build a library by creating an archive file with all the compiled files:

```
cc -c util_file.c  util_math.c  util_net.c.
ar -rs libmyutil.a util_file.o util_net.o util_math.o
```

To verify the library contents do:

```
ar -t libmyutil.a
```

and `nm -s libmyutil.a`

Consider now the program `main.c` that uses your library:

```
#include <stdio.h>
#include "myutil.h"

int main()
{
    printf("Inside main()\n");

    /* use a function from each object file that is in the library */
    util_file();
    util_net();
    util_math();
    return 0;
}
```

Compile and link you program using the command:

```
cc -static -o main main.c -L. -lmyutil
```

The option -L. gives the current directory as one that contains libraries and `-lmyutil` requests the linking of the libmyutil library. If you don't use "`-static`", your library will still be linked statically but *libc* dynamically. Run the executable confirming that everything works as expected.

---

[2] If strace is not installed use your system package management tool to install it or, if in a debian based distribution, give one of the following commands: `apt install strace`, `apt-get install strace`.

To see that *cc* is a *compiler driver* that calls the several phases of the compiler and finally the linker do:

```
cc -v –static -o main main.c -L. -lmyutil
```

3. Generation of dynamic libraries

The generation of a dynamic library is similar to creating a static library. Although, there are two important differences:

1. The compiler must generate code that is position independent. As the processes can load the library in different virtual addresses all the references in the code (jumps, accessing variables, etc) must be relative (to current Instruction Pointer register). In LINUX, this is achieved passing the flag *-fPIC* or *–fpic* to the compiler
2. The tool used to create the library is not *ar*. One must use the *ld* tool (directly or through cc). The flag to use is "*-shared"*

You can build both versions of a library but, for now, remove libmyutil.a. The sequence of commands used to create a dynamic (also shared) library is:

```
cc -fpic -c util_file.c util_net.c util_math.c
ld -shared -o libmyutil.so util_file.o util_net.o util_math.o
```

or just: `cc -fpic -shared -o libmyutil.so  util_file.c util_net.c util_math.c`

Run `file` command to verify the file type of `libmyutil.so`.

Compile and link with your library (you can use `–shared`, but that is the default):

```
cc -o main main.c -L. -lmyutil
```

Try to execute and check the dynamic linking by using `ldd main`. It worked?

At execution time, usually the dynamic loader looks for the shared library file in a pre-defined set of directories (*/lib,/usr/lib, /usr/X11R6/lib*, ...). One way is to configure the value of the environment variable `LD_LIBRARY_PATH` in order to extend the search to other places. Now do the following:

```
LD_LIBRARY_PATH=. ./main
```

Now it worked? Why? Execute the command

```
LD_LIBRARY_PATH=.  ldd main
```

and see why.

To know more about shared and dynamic libraries study the **Program Library HOWTO** from

http://www.dwheeler.com/program-librar