Fundamentos de Sistemas de Operação

MIEI 2018/2019

Laboratory session 5

Overview

Information about Linux memory map of a process. Process internal memory management with malloc/free.

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 = %lx, hard limit = %lx\n",
               (unsigned long)r.rlim_cur, (unsigned long)r.rlim_max);

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

    Getrlimit( RLIMIT_STACK, &r);
    printf("Maximum size of process stack; soft limit = %lx, hard limit = %lx\n",
               (unsigned long)r.rlim_cur, (unsigned long)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.

Now 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…

Malloc and System call *brk*

The *heap* is a continuous space of memory (i.e. continuous in terms of virtual addresses) with three bounds:

- a starting point
- a maximum limit, that can be obtained using *getrlimit*
- an the current end point called the break.

The break marks the end of the mapped data memory space, i.e. it is the highest virtual address that can be used by the heap at some point in time. The value of this limit can be changed with the system calls *brk* and *sbrk*.

```
#include <unistd.h>
```

`int brk( void *addr)` – sets the end of data the segment to the value specified by *addr*, if possible; returns 0 on success and -1 on error.

`void * sbrk( intptr_t incr)` – increments the data segment by *incr*. On success, returns the previous program break; if the program´s break was increased, the returned value represents a pointer (address) to the start of the newly allocated memory; on error returns (void *)-1. In the current versions of Linux, `sbrk` is a library call that uses `brk`.

Consult the manual pages of the system calls `brk`, `sbrk` and after that, consider the following program (sbrktest.c):

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

void *Sbrk(unsigned int increment){
       void *res = sbrk( (intptr_t)increment);
       if(res == (void *)-1){
              perror("sbrk");
              exit(1);
       }
       return res;
}

int main(int argc, char *argv[])
{
    unsigned int moreSpace;

    if( argc != 2){
       printf("%s size_of_memory_to_allocate\n", argv[0]);
       exit(1);
    }
    moreSpace = atoi(argv[1]);

    // getting the current break value
    printf("Current program break = %u\n", (unsigned int) Sbrk(0));

    // moving program break
    Sbrk(moreSpace);
    printf("New program break = %u\n", (unsigned int) Sbrk(0));

    return 0;
}
```

Run the above program with increasing arguments until there is an error message. Compare the value that conducts to an error with the limit indicated by *rlimit*.

Malloc/Free 1st implementation [2]

Consider the following version of a tentative *malloc* function (mymalloc1.c):

```
#include <sys/types.h>
```

[2] From now on, this guide uses parts of the report "A Malloc Tutorial" by Marwan Burelle, Laboratoire Système et Sécurité, École Pour l'Informatique et les Techniques Avancées (EPITA), 2009. This guide is also in the clip.

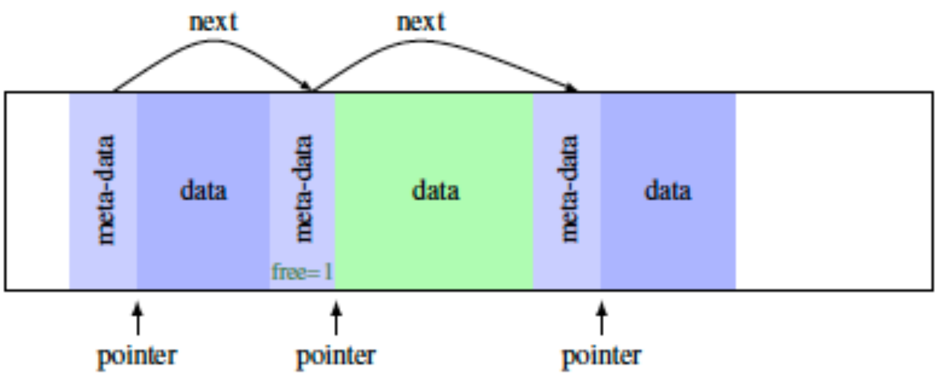
```
#include <unistd.h>

void * myMalloc(size_t size){
    void *p = sbrk(size);
    /* If sbrk fails, we return NULL */
    if ( p == (void*)-1)
        return NULL;
    return p;
}
```

This code works? If the answer is yes, what is the reason why it should not be used?

Malloc/Free 2nd implementation

As you cleverly devised, the code above cannot be used because it is not possible to free the allocated memory in the code through a *free* function. To build usable versions of *malloc / free* the heap can be organised as in the next figure. This is extracted from the report cited in the previous footnote:



As sketched, each allocated block of memory has two parts:

- metadata -- elements for managing the space, including a pointer to the next block;
- the space to be used by the process, returned by malloc(); please note that "pointer" in the picture corresponds to the value returned to the process by malloc().

In C (see mymalloc2.c), we can have a linked list for that metadata, where each record is defined according to the following declaration:

```
typedef struct s_block *t_block;

struct s_block {
  size_t size;   // size of current block
  t_block next;  // pointer to next block
  int free;      // flag indicating that the block is free or occupied; 0 or 1. Occupies 32
                 // bits but the compiler aligns the structure to a multiple of 4
};
```

Finding a chunk of memory to reuse using the First Fit algorithm

When possible, malloc should try to reuse a freed block. Suppose we maintain the following two global pointers:

- *base* which points to the beginning of the block list;
- *last* which points to the last element of the list (this is useful when there is no space available and you need to add a new block to the list).

```
t_block find_block(size_t size){
    t_block b=base;
    while (b!=NULL && !(b->free && b->size >= size)) {
        b = b->next;
    }
    return b;
}
```

The code is easy to follow as we just travel through the list until a block with size equal or bigger to needed *size* is found. If there is no such block, NULL is returned.

Extending the heap

When there is no free block to reuse, it is necessary to extend the heap, the *sbrk()* function is called to get more space that is then added to our list.

```
#define BLOCK_SIZE sizeof(struct s_block)

t_block extend_heap(size_t s){
    t_block b = (t_block) sbrk(BLOCK_SIZE + s);
    if ( b == (void*)-1)
        /* if sbrk fails, return NULL pointer*/
        return NULL;
    b->size = s;
    b->next = NULL;
    b->free = 0;
    if (base==NULL) base=b;
    else last->next = b;
    last = b;
    return b;
}
```

Work to do

Using the code fragments above produce working versions of the following functions:

```
void *myMalloc(unsigned int noBytesToAllocate);
int myFree(void *address);
```

`myMalloc` and `myFree` behaviour correspond to the C library functions *malloc* and *free*.

To test your code add a function

```
void debugBlockList()
```

that dumps the contents of the block list, printing, in a separate line, the metadata associated with each memory block.

Final comment

Function *find_block* above returns to the caller the first (free) block found regardless of its size. Of course, this is a waste of space. If interested, you can consult the report cited in footnote 2 or the section 8.7 (pages 185-188) of the book "***The C programming Language 2nd Ed***"*, B. Kernighan, D. Ritchie, Prentice-Hall 1988* [3]to learn how to build much more efficient (in terms of memory usage) versions of *malloc* and *free*. It is also useful to study chapters 13, 14, and 17 of the course's reference book **Operating Systems: Three Easy Pieces**, R. Arpaci-Dusseau, R. Arpaci-Dusseau, 2015.

[3] This book is not included in the course references but any serious candidate to a degree in Informatics or Computer Science and Engineering must have it in its bookshelf.