Fundamentos de los Sistemas Operativos

Departamento de Informática de Sistemas y Computadoras (DISCA)

Universitat Politècnica de València

fSO

Lab session 8
Memory map analysis (I)

1 Objectives

The main objective of this lab session is to analyze **the memory map of processes,** as well as to understand how it evolves along the process lifetime. We will rely on Linux file ***/proc/PID/maps***.

2 Memory map of a process

Processes are assigned a logical address space. The operating system allocates in the logical space different sections in which process memory is structured: code area, data area, areas for dynamic binding, stack area, libraries and so on. This collection of sections will define the structure of the logical address space and each operating system uses its own structure.

The operating system is responsible for managing the memory map of a process. The memory map of a process may vary during its execution, evolving throughout its lifetime. There is a great similarity or correspondence between the contents of the executable file that supports a process and its initial memory map.

A program's logical address space defines the internal organization of the user code which is determined by the operating system that is the responsible for loading a program into memory to generate a process. In this lab session we will work with the model of logical addresses of the ELF (Executable and Linking Format) standard used in Linux.

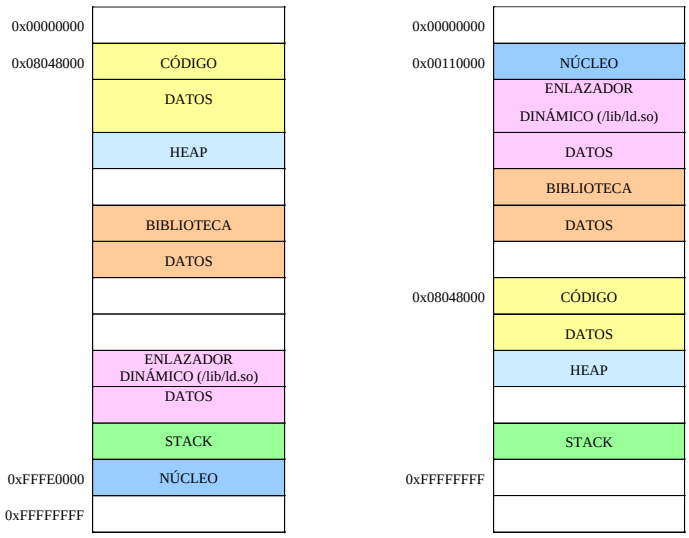


Figure 1. Two possible organizations of a process logical memory for Linux 2.6. x versions on 32-bit PCs.

2.1 32-bit and 64-bit memory maps

Nowadays operating system (i.e. Windows and Linux) are distributed in two versions: 32-bit and 64-bit, depending on the processor architecture where there are going to run. 32-bit processors are able to address logical spaces up to 4GB (2^{32}= 4 GB), but 64-bit processors can address up to 16 EB (2^{64} = 16 ExaByte).

Among the Intel processor family, backward compatibility allows to install a 32-bit operating system on a 64-bit processor and 32-bit processes to run under a 64-bit operating system, but the opposite is not possible: neither a 32-bit operating system can handle 64-bit processes nor 64-bit operating system will run on a 32 bit processor. So, in a 64-bit operating system 64-bit and 32-bit processes can coexist, but 32-bit operating systems only run 32-bit processes.

In Linux the gcc compiler allows you to choose the addressing width for the generated code. Typically, only libraries matching the system address width are available so the code generated will match it, but cross compiling is supported by gcc if the libraries for the target architecture are available. For Intel processors, gcc generates code for 32 bits using the "-m32" option that is the default on 32-bit versions of Linux, but if you want to generate a 32-bit version of your program in a 64-bit Linux system then you have to explicitly set it up with the “-m32” option.

In this lab session we use 32-bit memory maps. The compiling commands found along this lab guide include the “-m32” option because the operating system installed in the lab is 64-bit. If you like, you can optionally try to analyze 64-bit memory maps, if you do it you will find that they are more complex.

2.2 Memory map regions

The memory map of a process consists of several regions, as shown in Figure-1. Each **region** is a **contiguous logical process space zone** characterized by its starting address within the map, its size and a set of properties:

- **Support**. There are two possibilities

  Region with file support: The initial content of the region is got from a file.

  Unsupported region: The region has no initial content.

- **Use type**. It indicates if it is allowed to share the region between processes.

  Private (p): The content of the region is only accessible by the process.

  Shared (s): The content of the region can be accessed by several processes.

- **Protection:** type of access allowed to the region: reading (r), write (w) and execute (x).

- **Fixed or variable size**: variable-sized regions tend to distinguish because they can change their size along process lifetime.

Regions in the **initial memory map** of the process are:

- **Code or text.** Shared region with read(r) and execute(x) permissions, fixed size and supported by the executable file. A region marked as "p" without write permission, in fact it will be shared. In previous versions of the Linux kernel, these regions had the "s" mark.

  r-xp Region code

- **Data with initial value**. Private region (p), each process needs a copy of its variables, it has read (r) and write (w) permissions, fixed size and is supported by the executable file.

  rw-p. Initialized data region when it is supported by a (data) file.

- **Data without initial value**. Private region (p), it has read(r) and write(w), fixed size and not supported since its initial content is irrelevant.

  rw-p. Not initialized data region when it is not supported by a file (BSS segment).

- **Stack**. Private region (p) of variable size with read(r) and write(w) permissions, it supports function local variables, parameters (passed by value or by reference) and returning values. Also supports processor instructions that deal with the stack (push and pop).

The memory map of a process is dynamic with regions that can be added or removed during its execution, some of the new regions that can be created are:

•**Heap.** Private r/w region without support (initially set to zero values) and growing/decreasing according to memory allocation/deallocation by the process. It supports dynamic memory that a process uses at execution time (in C language using the function "malloc/free").

•**Mapped files**. When mapping a file on the process logical space, a region is created for this purpose, its protection is specified by the process when the file mapping is done.

•**Shared memory**. When a shared memory area is created it appears a new shared region in the process memory map.

•**Thread stacks**. Every "thread" needs its own stack.

2.3 The /proc/PID/maps file

The files that are located in the */proc* directory offer information about the system and active processes. To get this information about a process you have to access the directory identified by the process PID. To facilitate a process access to its own information there is a special directory "self" for every process, that is actually implemented using a symbolic link to the directory corresponding to such a process. So when a process with PID 2975 access to */proc/self/*, it is actually accessing the */proc/2975/* directory. This lab session focuses on the information contained in the "maps" file of a process.

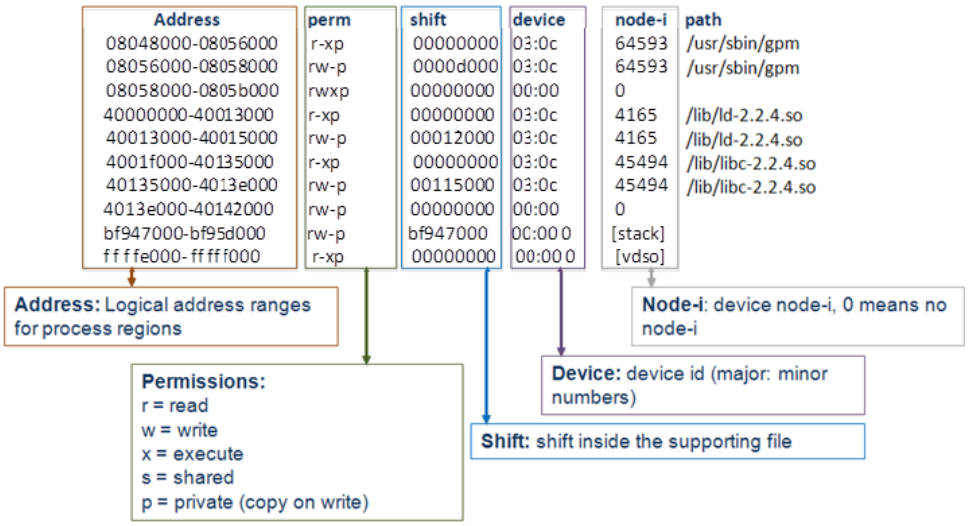


Figure 2. Content of the columns that appear when listing the "maps" file of a process

The following table summarizes the different types of regions observed in Figure 2.

Code regions	Own program code Dynamic library code VDSO region
Data regions	Data region supported by the executable file Non supported data region Stack

Table-1 Some regions that can be observed in a process memory map

The line [vdso] in Figure-2, corresponds to operating system code. In particular, it is the Virtual Dynamically-linked Shared Object, a shared library that allows the process to perform some kernel actions without the overhead of a system call.

2.4 Exercise 1: Experiment with /proc/self

a) Execute twice the following command and explain why you get different results

```
$ ls –l /proc/self
```

a) The command "head -1" shows the first line of a text file on the standard output, execute the following command and explain the result that you get:

```
$ head -1 /proc/self/maps
```

```
$ cat /proc/self/maps | head -1
```

2.5 Tools and commands: system and size

Now we will visualize file */proc/PID/maps*, where PID is the process ID, to analyze the memory map of a LINUX process. During the execution of a process we will use the command *cat* and execute it with *system()* to display the contents of the *maps* file. For this purpose we will build a command with the string *'cat /proc/PID/maps'* and then call *system (command)*. The following lines of code will be included in the program in those points where you want to see the process memory map in order to analyze it.

```
// Create command "path_maps" to show memory map
sprintf(path_maps, "cat /proc/%d/maps",getpid());
// Empty the buffer
fflush(stdout);
// Execute command "path_maps"
system(path_maps);
```

Figure 3. Code to visualize a process memory map

The shell command ***size*** displays the size of every section of an executable file, printing on the screen the size of every region that make up the process, for more information on that command do *$man size*. So, you can try size for instance with */bin/ls*:

```
$ size /bin/ls
   text    data     bss     dec     hex filename
 105182    2044    3424  110650   1b03a  /bin/ls
```

bss (Block Started by Symbol): Size of non-initialized variables region

dec = text + data +bss; hex = 0x(dec)

3 Memory map: local, global variables and function parameters

Program variables can be of the following types:

Global variables: they are defined outside of any function body, and they can be accessed by any function inside the same source file.

Local variables: they are defined inside a function, and they can only be accessed within that function.

Function parameters: they are defined in the function template and are accessible within the function, to read only (passed by value) or read/write (passed by reference).

As we will see these three types of variables do not have to be located in the same region of the process memory map, although they could be. We will work with the file named "map1.c" provided as lab session material in PoliformaT. The file contains the code shown below:

```
/* map1.c code*/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

char path_maps[80]; /* Global variable */
char ni_glob[4095];
long i_glob=20;      /* Initialized global variable */

void f(int param)
{
    printf("Address of funcion f parameter: %p\n",&param);
}

int main() {
    long i_loc=20; /* Initialized local variable */
    long ni_loc; /* Non initialized local variable */
    printf("Process PID: %d\n\n", getpid()); /* Shows process PID */

     /**** ADDRESS VISUALIZATION ***/
    printf("main function address: %p \n", main);
    printf("f function address: %p \n", f);
    printf("Initialiazed global variable i_glob address: %p\n", &i_glob);
    printf("Non initialized global variable ni_glob address \n");
    printf(" 1st ni_glob element address: %p\n", &ni_glob[0]);
    printf(" Last ni_glob element address: %p\n", &ni_glob[4095]);
    printf("Initialized local variable i_loc address: %p\n", &i_loc);
    printf("Non initialized local variable ni_loc address: %p\n", &ni_loc);
    f(40);
    printf("\n PROCESS MEMORY MAP /proc/%d/maps \n", getpid());
    // Create command "path_maps" to show memory map
    sprintf(path_maps, "cat /proc/%d/maps",getpid());
    // Empty the buffer
    fflush(stdout);
    // Execute command "path_maps"
    system(path_maps);
    printf ("            ---------------\n\n");
    return 0;
}
```

Figure 4. Code of source file "map1.c"

3.1 Exercise 2: A basic process memory map

Compile *map1.c* and run it, so do the following:

```
$ gcc -m32 map1.c –o map1
$ ./map1
```

Question 1: Analyze the memory map shown and by referring to both figure 2 and the system manual ($ man proc), answer the following questions:

1. Which is the range of logical addresses occupied by the process "*map1*"?
2. Identify the region of the code and enter its starting and ending addresses. Do the same with the stack region.
3 How many of the studied regions can you identify in the memory map of “map1” process? Identify a supported data, a non supported data region and a library code region.
4 What “map1” regions are supported by physical devices? Justify the need for such support.
5 What i-node number have the non supported regions?
6. What regions of "*map1*" may be shared by multiple processes?
7. Identify regions in table 1, where the function main and the function f are allocated

Question 2: specify in which regions, from those indicated in table 1, the system has allocated each one of the program variables.

Variable type	**Allocated region**
Initialized global variables	

Not initialized global variables:
`path_maps[80]`
`ni_glob [4095]`	

Initialized local variables	
Not initialized local variables	
Function parameters	

3.2 Exercise 3: Memory map of a process with large size local variables

In this exercise we will work with the map2.c file that is provided with lab session material. The code of this file, shown in Figure 5, is almost the same as map1.c (Figure-4), except that there is a new function f2.

```
void f2() {
    int vloc[1024*1024];
    printf("Local variable vloc address: %p\n", vloc);
    printf("\n PROCESS MEMORY MAP (in function f2) \n");
    fflush(stdout);
    system(path_maps);
    printf(" ---------------\n\n");
}
```

Figure 5. Function f2 code, included in “map2.c”, with a large local variable

In the code of this function f2, you define a local variable of large size: int vloc [1024 * 1024]. The map2.c program prints memory map both before calling function f2 and after calling from within that function. The objective is to compare both memory maps, to observe what changes are produced and its relation with variable vloc.

Compile *map2.c* and run it, so do the following:

```
$ gcc -m32 map2.c –o map2
$ ./map2
```

Analyze the memory map shown by map2 and by referring to both figure 2 and the system manual ($ man proc), answer the following questions:

Question 3: Indicate in which region of memory is the local variable vloc.

Question 4: Compare the resulting process maps map1 and map2, and indicate if the size of their respective stacks is different. Indicate which of them is larger and justify it.

4 Memory map: dynamic memory allocation

Dynamic memory allocation is performed by a process during its execution. In Java, dynamic memory allocation is made using the *new* operator, while in C it is done using the *malloc* function. For more information about *malloc* and the dynamic memory allocation in C, see the annex.

Dynamic memory allocation is supported by a region of memory called heap. It is a region of data not supported by file (initialized to zero), which grows as memory is allocated by the process and decreases when it is memory is freed.

4.1 Exercise 4: Process map with dynamic allocation

In this section we will work with the file map3.c provided with lab session material. In this file we can see the following statements and instructions:

```
/* map3.c code */
int *vdin;
/*** Dynamic memory allocation ***/
vdin = (int *) malloc(100*sizeof(int));
……
/** Free previously allocated memory**/
free(vdin);
```

Figure 6. Lines of code in "map3.c" where dynamic memory allocation/deallocation is done

The line with malloc (100 * sizeof (int)) makes a dynamic memory allocation for an array of 100 integers and is equivalent to the Java statement: vdin = new int [100];
Compile "map3.c" and run it, so do the following:

```
$ gcc -m32 map3.c –o map3
$ ./map3
```

The map3 process prints the memory map before and after making the dynamic memory reservation. Our aim will be compare both maps from memory, to observe what changes have happened and its relation with variable *vdin*.

Question 5: Analyze the maps of memory shown by map3 and compare them

1. Indicate whether it appears or disappears some region. Attempt to justify what these changes are due to.	
2. In the case to appear some region, what kind of region is it? What are its permissions?	
3. Indicate in which region is the contents of vector vdin.	

4.2 Exercise 5: Increasing the dynamic allocation

Work with "map3.c" code and increase the dynamic memory allocation as indicated in the following table, you have to compile and execute every time.

Question 6: Discuss what happens with the region where the variable "vdin" is allocated, try to calculate its size.

Dynamic allocation to do	**Effects on the heap region**
`malloc(50000*sizeof(int));`	
`malloc(100000*sizeof(int));`	
`malloc(300000*sizeof(int));`	

5 Memory map: using libraries

Libraries are binary files that contain routines which may come from different environments such as: mathematical libraries, image processing, library to do system calls (OS API) or they can be created by the user. Libraries are a versatile and simple way of making code modular and reusable.

The mechanism that make libraries accessible to applications is called linking and there are two types, static link (cannot be shared) and dynamic link (allows sharing). In Windows, dynamic library files have the extension .DLL (Dynamic Link Library), while the static usually end in .LIB. In Unix and Linux, dynamic libraries have extension .so (Shared Object) and the static .a (Archive).

• **Link static**: the executable is self-contained: includes all the code the application needs, i.e. the own process code and the external functions needed. In case of having two versions of the same library, static and dynamic you should use the compiler option `-static`. The compilation command is:

```
$ gcc -m32 program.c - static - lXYZ - o program
```

where `-lXYZ` indicates linking with library `libXYZ`. For example, `-lm` tells to use the library libm (math library).

•**Dynamic link implicit:** load and mounting of the library is carried out at runtime from the process. It is therefore at runtime when it has to resolve references to constants and functions (symbols) of the library and the relocation of regions program. This is the default option of the compiler (if not otherwise specified) that seeks first to the dynamic version of the library. The build order is as above but without `-static` :

```
$ gcc -m32 program.c - lXYZ - o program
```

As part of the linking process, provided that at least one dynamic library is used, it is included in the executable file the dynamic linking module, that will perform at run time the dynamic loading and linking of libraries used by the program. This option generates an smaller executable file than static linking.

5.1 Exercise 6: Memory map of a process that uses libraries

The file "lib_cos.c" that comes with lab session material, contains the code for a program that uses the cosine function which can be found within the mathematics library. Figure-7 corresponds to that code, whose memory map is printed before and after using cosine function in order to analyze the changes experienced by the process memory map.

```
/* lib_cos.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#define PI 3.14159265
/* Global variables */
char path_maps[80];
char ni_glob[4095];
long i_glob=20;

void f(int param)
{
    printf("Address of funcion f parameter: %p\n",&param);
}
int main()
```

```
{
    /* Local variables */
    long i_loc=20; /* Initialized local variable */
    long ni_loc; /* Non initialized local variable */
    float c;

    printf("Process PID: %d\n\n", getpid()); /* Shows process PID */

    /**** ADDRESS VISUALIZATION ***/
    printf("main function address: %p \n", main);
    printf("f function address: %p \n", f);
    printf("Initialiazed global variable i_glob address: %p\n", &i_glob);
    printf("Non initialized global variable ni_glob address \n");
    printf(" 1st ni_glob element address: %p\n", &ni_glob[0]);
    printf(" Last ni_glob element address: %p\n", &ni_glob[4095]);
    printf("Initialized local variable i_loc address: %p\n", &i_loc);
    printf("Non initialized local variable ni_loc address: %p\n", &ni_loc);
    f(40);

    /** Mathematical operation **/
    c = cos(45*PI/180);
    printf("The mathematical operation result is: %f\n", c);

    printf("\n PROCESS MEMORY MAP /proc/%d/maps \n", getpid());
    sprintf(path_maps,"cat /proc/%d/maps",getpid());
    fflush(stdout);
    system(path_maps);
    printf(" ---------------\n\n");

    return 0;

}
```

Figure 8. Code in *lib_cos.c*

Compile the program, both with the static library and the implicit dynamic linking and execute it redirecting the output to a file:

```
$ gcc -m32  lib_cos.c -static -lm -o lib_static

$ gcc -m32  lib_cos.c -lm -o lib_dynamic

$ ./lib_static > libs_result

$ ./lib_dynamic > libd_result
```

Question 7: Display generated files “libs_result” and “libd_result”, compare them and answer the following questions:

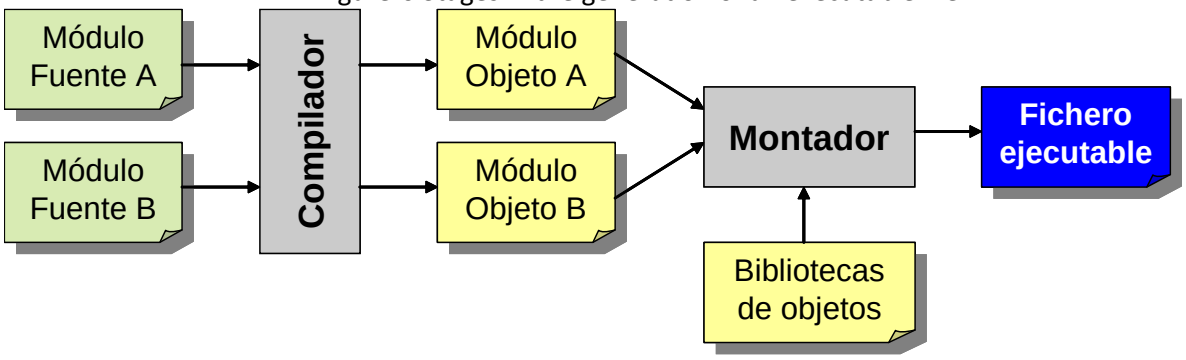
1. What differences do you find when comparing the memory maps generated with the two linking types?	
2 Run the command "ls – lh” in the two executable files and compare their sizes. Try to justify the observed difference.	
3. Apply the command *size* to the resulting executable files and compare the results. Try to justify differences in size of the same regions in both executable files.	

6 Annex

6.1 Generating an executable file

In general, an application will consist of a set of modules which contain source code that must be compiled and linked, as described in Figure-6. The compiler generates the machine code in each source module and the linker generates a single executable file binding all modules and resolving references between them.

Figure-6 Stages in the generation of an executable file



6.2 Executable file format

An executable file is divided into a header and a collection of sections as we can see in Figure-7. The header contains control information that allows to understand the remaining content of the executable file. Every executable file has a set of sections, but at least it will appear three of them: code, data with initial values and data uninitialized. This last section appears in the table of sections of the header, but is not normally stored in the executable since its content is irrelevant.

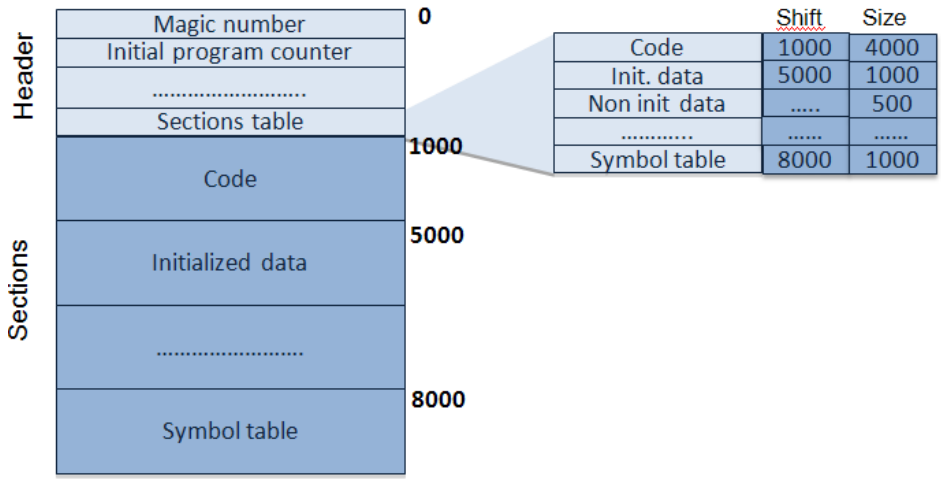


Figure 7. Simplified format of an executable file in LINUX (ELF).

When requesting the execution of a process, several regions of the map are created from the information in the executable file. The initial regions of the process memory map basically correspond to the different sections of the executable file.

6.3 Dynamic memory allocation in C language

Dynamic memory allocation is the one requested by a process at run time. This means that the amount of memory allocation is not specified in the variable declaration. Dynamic memory allocation adds great flexibility to programs because allows the programmer to reserve the exact quantity of memory at the precise moment that it is needed, without having to make a reservation by excess in advance.

The function used to allocate dynamic memory in C is **malloc,** that allocates a contiguous portion of memory. It is defined as:

*void *malloc(size_t size);*

malloc returns a pointer of type void *, that is the memory address from which portion of size *size* is reserved. If you cannot reserve that amount of memory requested malloc returns a pointer set to NULL. A pointer is a variable that contains the address of another object..

The malloc function returns a pointer to *void,* or generic pointer. The C compiler requires a type conversion by using a casting. For example:

```
char *cp;
cp = (char *) malloc(1024);
```

This example attempts to reserve 1024 bytes and the start address is stored in cp. The generic pointer returned by malloc is converted to a pointer of type char through the casting "(char *)".

It is usual to use the sizeof() function to indicate the number of bytes. The sizeof() function can be used to find the size of any type of data, variable or structure, for example:

```
int *ip;
ip = (int *) malloc(1024 * sizeof(int) );
```

In this example you are trying to reserve memory for 1024 integers (not bytes).

When you have finished using a portion of memory you must always free it using the free() function. This function allows that the freed memory be available again for another malloc() call. For instance:

```
free(cp);
free(ip);
```

The free() function takes a pointer as an argument and frees the memory to which the pointer refers.