

Operating Systems Lab



Lab # 14

Memory Management

Instructor: Engr. Muhammad Usman

Email: usman.rafiq@nu.edu.pk

Course Code: CL2006

Department of Computer Science,

National University of Computer and Emerging Sciences FAST

Peshawar Campus

Table of Contents

Memory Management	3
PROC File System	3
Exploring PROC	3
Processes in PROC	4
Overall Memory Usage in PROC	5
System Calls for Memory Handling	5
Introduction	5
Memory Allocation	6
Malloc()	6
Free()	7
Realloc()	8
Calloc()	8
Exercise	9
Recommendations	9

Memory Management

PROC File System

Give the following command without any arguments and press Enter:

```
mount
```

You will see a list of all partitions that are currently mounted on your system. If you look closely, there will be an entry called PROC. PROC is a special file system and is an open door to the Linux kernel. You can peep in here to see what the kernel is doing. The proc file system can be accessed by changing into the /proc directory.

```
cd /proc
```

Here, you will apparently see plenty of files. But these are not files, rather they are parameters, data structures, and statistics collected from the kernel and which appear like files inside the /proc directory. The contents of this directory are created 'on-the-fly' by the kernel whenever we read its contents. The existence of “files” in this directory only appear the instance you request it. Otherwise, the directory is *EMPTY*.

Exploring PROC

So let's start exploring this file system. Type the following command:

```
ls -l /proc/version
```

Note the following about above command:

Q1 What is the file size?

Q2 What is the modification time?

Now let's view the contents of this file. Type the following:

cat /proc/version

You can see version numbers such as the kernel, and the compiler which compiled the kernel.

Processes in PROC

The numbers you see inside the PROC file system are directory representations for each process currently running on your system. These numbers are the process id's. If you give the ls command repeatedly, you will notice that these directory representations are changing dynamically. Choose a process ID and enter that directory using:

```
cd /proc/<number>
```

View the contents. You should be able to see files like cmdline, cwd, environ, exe, fd, stat, statm, status, etc. Of these, fd contains the number of open file descriptors for any given process.

Using the ps command, find out the process id of your current shell.

```
ps
```

Now give the following command:

```
ls -l /proc/<your shell process id>/fd
```

Q3 What open file descriptors can you see from the output?

Q4 What do the arrow operator (->) represent?

Open a new terminal window and type the following:

```
echo "Hello World" >> /proc/ <your shell process id>/fd/1
```

Go back to the previous terminal window.

Q What just happened? We had two terminal windows open. Each terminal window

was one instance of shell process (so 2 processes). We gave a command in one shell and the output appeared in the other shell. Isn't this communication between two processes? The IPC method of communication was through a “file”.

Overall Memory Usage in PROC

Overall memory usage can be seen from

`cat /proc/meminfo`

Q1 Find out the total memory of your machine?

Q2 How much of it is free?

Q3 How much is your swap space?

Q4 How much of that swap space is free from meminfo?

System Calls for Memory Handling

Introduction

For sake of definition, a process is a running program. It means an OS has loaded a program into memory, has arranged some environment for it, and has started running it. Memory is required for the following areas in a process:

- Text Section: The portion of process where the actual executable code resides.
- Data Section: Containing global data that are to be generated during run-time when the program starts.
- Heap: The region from where memory can be dynamically allocated to a process.
- Stack Section: Where local variables (non-static) and function calls are implemented.

To have a general overview of how much size is allocated to a process, do the following simple test.

1. Create a simple Hello World
2. Compile it into an executable called Hello
3. Then, type the command: `ls -l Hello` and note down the size of the executable
4. Then, type the command: `size Hello` and note down the total size of all sections

`ls -l Hello`

`size Hello`

You will notice that a lot of the size given in the `ls` command is nowhere to be seen for the `size` command. Where does it go? Look at option 3: Heap.

Memory Allocation

We have some library functions in C that are used for allocating additional memory to a process. These allocations are done at run-time. Dynamic memory allocation are performed by the calls `malloc()` or `calloc()`. When successful, these will return pointers to the portion of memory just allocated. Once a portion of memory is allocated, we can change its size dynamically using the `realloc()` call. Finally, memory released is freed up using the `free()` call.

1. Malloc()

Initially memory is allocated using `malloc`. The value passed is the total number of bytes requested. If memory is not allocated, `NULL` is returned. If memory is allocated, a pointer to 1st address is returned. Try multiplying `sizeof` by a large number to see if it can allocate that much memory.

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

int main()
{
    struct coord{
        int x, y, z;
```

```
};

struct coord *p;

p = malloc(sizeof(struct coord));

if (p==NULL)

{

    printf("Failed");

}

else

{

printf("\n%d bytes allocated at address %p\n", sizeof(struct
coord), p);

}

return 0;

}
```

2. Free()

Once we are done with memory, we can return it back using the free() function.

Continuing with previous code, we add the following:

```
free(p);
```

```
p = NULL;
```

The first line will free the memory address returned to pointer p. This means it is marked

for use if additional memory is required for the process. However, variables still pointing to it can still use it until the time it is overwritten.

This is known as a dangling pointer. It is good idea to set the pointers to NULL along with

usage of free.

Try free two times and see the output.

3. Realloc()

Using realloc() call, we can dynamically change the size of the block of memory allocated earlier on. It's usage is simple:

```
p = realloc(p, sizeof(struct coord)*10);
```

This will take the current block returned to pointer p, and change the size to whatever we specify as the second parameter.

4. Calloc()

When we allocate space using malloc(), it is not initialized. Calloc() call is just a wrapper around malloc(). Not only does it allocate the required space, it also initializes it to 0.

```
p = calloc(1, sizeof(struct coord));
```

Where, the first parameter is the number of members, and the second parameter is the size required for a member.

Exercise

Study the code below. What is wrong with it?

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

void f(void)
{
    void* s;
    s = malloc(50);
    return;
}

int main(void)
{
    while (1)
    {
        f();
    }
    return 0; }
```

Recommendations

<https://www.geeksforgeeks.org/dynamic-memory-allocation-in-c-using-malloc-calloc-free-and-realloc/>

<https://www.linux.com/news/discover-possibilities-proc-directory/>