Part V

Memory Management 59

Chapter 9

Memory

9.1 PROC File System

Give the following command without any ar guments and press Enter:

```
mount
```

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

```
cd /proc
```

Here, you will apparently see plenty of les. But these are not les, rather they are param eters, data structures, and statistics collected from the kernel and which appear like les in side the /proc directory. The contents of this directory are created 'on-the- y' by the kernel whenever we read its contents. The existence of les in this directory only appear the in stance you request it. Otherwise, the directory is EMPTY.

9.1.1 Exploring PROC

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

ls -l /proc/version

Note the following about above

command: Q1 What is the le size?

Q2 What is the modi cation time?

Q3 Now check the current time on your watch?

Do you notice that the modi cation time is the same as the time you entered the command. If not, wait for 1 minute and run the command again. You will notice that the modi cation time has changed yet you haven't even touched the le so far. This proves that the information appears on the y and that nothing exists in the directory as such.

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

```
cat /proc/version
```

You can see version numbers such as the kernel, and the compiler which compiled the kernel. Give the ls command again. Note the following again:

Q What is the le size? What is the modi - cation time? How come so much informa tion was just showed as contents of the le yet the le size is zero? Again, this is con rmation of previous point.

9.1.2 Processes in PROC

The numbers you see inside the PROC le sys tem are directory representations for each pro cess 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 dynam ically. Choose a process ID and enter that di rectory using:

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


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

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

```
ps
```

Now give the following command:

• Size of shared libraries

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

Q What open le descriptors can you see from the output?

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

Open a new terminal window and type the fol lowing:

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

Go back to the previous terminal window.

Q What just happened? We had two termi nal windows open. Each terminal win dow 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 com munication was through a le .

9.1.3 Process Memory in PROC

Look at the output of the statm le. It should

• Memory used by the process for its stack

• Number of dirty pages (pages of memory that have been modi ed by the program)

Check these information for your currently logged in shell.

```
cat /proc/<your shell process id>/statm
cat /proc/<your shell process id>/status
```

9.1.4 Overall Memory Usage in PROC

Overall memory usage can be seen from cat /proc/meminfo

Q Find out the total memory of your machine

Q How much of it is free

Q How much is your swap space

Q How much of that swap space is free from meminfo display a list of seven numbers separated by spaces. Each number is the number of page frames of memory used by a process in a par ticular category. The categories are:

- Total process size
- Size of process currently resident in main memory
- Size of memory shared with other pro cesses
- Text section size of a process

9.2 Exploring Memory

9.2.1 Logical and Physical Ad dresses

Write the following simple C program.

```
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>
int main()
{
   int fd, bw, br;
   char *buffer=(char*)calloc(NULL, BUFSIZ); fd =
   open("/tmp/foo.txt",O_CREAT|O_RDWR);   br  =
   read(0,buffer, BUFSIZ);
   bw = write(fd,buffer,BUFSIZ);
   close(fd);
   return 0;
}
```

You can see usage of I/O system calls such as open(), read(), write() and close(). You can also see a system call called calloc(). Calloc() is allocating un-used space for an array of n number of elements. We will see description of calloc() and its sister system calls shortly.

9.2.1.1 Logical Addresses

Compile this code using:

```
gcc code.c -c
```

and then

```
gcc code.c -o code -g
```

The -g is for adding debug symbols.

Q What is the size of the executable le with and without the -g argument?

We are now going to use the objdump utility to disassemble (-d) the program and view the information about the object le.

Run the following:

```
objdump -D code.o | less
```

Note: If the output is too much, you can enter

```
objdump -D code.o >
   /location/of/directory/code.txt
```

You can then view the contents of the output by opening code.txt in any text editor. Find out the following (You will have to learn to scroll the huge list of addresses up and down and learn to catch addresses):

Q The address on which your rst instruction of your program starts. Note it down.

Q The address at which the main() function starts. Note this down as well.

Q Identify the instructions for calloc(), open(), read(), write() and close() calls and note down the addresses on which these instructions appear. This should be in the main function.

Q What do you think? Are the addresses to the left logical or physical addresses?

Q Find out where and in what section the write() call is referring to? Go to that sec tion. How many lines of code are in that section (making a rough estimate)?

Q Do you think these lines of code are the actual implementation of the write call? Write() call is from the unistd.h library. So do you think we are currently looking at the function from the unistd.h library?

9.2.1.2 Physical Addresses

Now exit and recompile your code with the static option.

```
gcc code.c -o code -static
```

Q What is the size of the executable le with and without the -g argument?

Use Objdump again.

```
objdump -D code | less
```

Try to answer the following and compare with the information you noted down earlier.

Q What is the starting point of your rst in struction again?

Q What is the address of the main function?

Q Do you think this time it is a logical or a physical address?

Q At what instruction addresses are the di er ent calls located (calloc, open, read, write, close, etc)

Q In which section is the write() call referring to? Where is it referring to in memory address? Go to that address !!!

Q Do you think this is the actual implemen tation area of the write() call?

9.3 System Calls for Mem ory Handling

9.3.1 Introduction

For sake of de nition, a process is a running program. It means an OS has loaded a pro gram into memory, has arranged some environ ment for it, and has started running it. Mem ory is required for the following areas in a pro cess:

- 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 imple mented.

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

1. Create a simle 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.

9.3.2 Memory Allocation

We have some library functions in C that are used for allocating additional memory to a pro cess. These allocations are done at

run-time. Dynamic memory allocation are performed by the calls malloc() or calloc(). When success full, 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.

9.3.2.1 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 %d\n",
                  sizeof(struct coord), p);
  }
  return 0;
}
```

9.3.2.2 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 rst line will free the memory address re turned to pointer p. This means it is marked for use if addtional 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.

9.3.2.3 Realloc()

Using realloc() call, we can dynamically change the size of the block of memory allocated ear lier 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. There is, however, a problem with the above statement .. can you identify it?

9.3.2.4 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 rst parameter is the number of members, and the second parameter is the size required for a member.

9.3.3 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;
}
```