Session 6: Memory

On completion of these activities you should be able to:

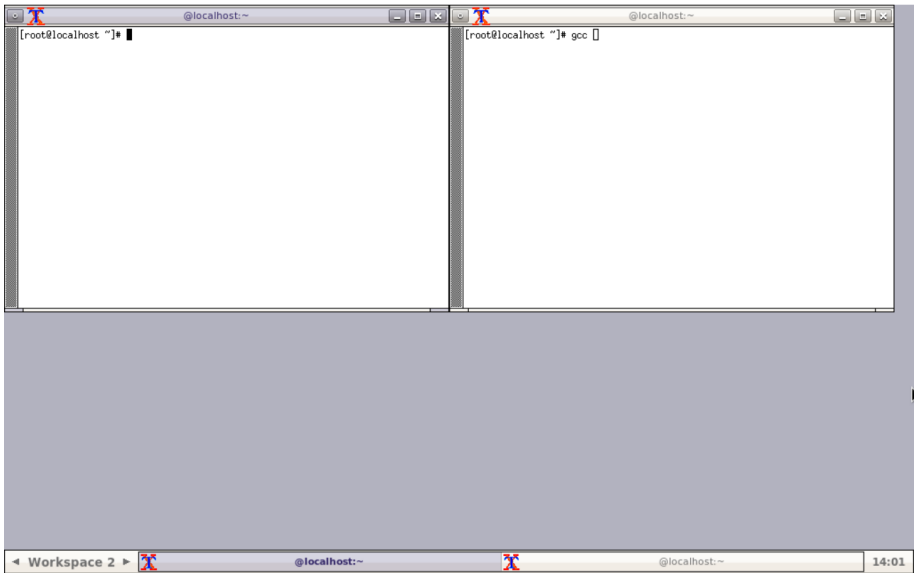
- Use system tools to monitor the memory behaviour of Linux processes.
- Understand and make minor modifications to simple C-programs which allocate memory using malloc.
- Explain some aspects of Linux process behaviour in terms of Virtual Memory management

Size units for memory

Much documentation referring to memory uses the abbreviations KB, MB, GB to mean 2^{10}, 2^{20}, 2^{30} bytes, respectively. This is technically *incorrect*, since the SI prefixes K, M, G actually mean 10^3, 10^6, 10^9. Note, for example, that 10^9 (= 1,000,000,000) is *smaller* than 2^{30} (= 1,073,741,824). This is why disk vendors use GB with its technically *correct* meaning: it allows them to put a bigger number on the box. This document uses the abbreviations KiB, MiB, GiB for 2^{10}, 2^{20}, 2^{30}.

Task 0

Start a unix environment in JSLinux (https://bellard.org/jslinux/). Select the Fedora X-Windows environment that should look something like the figure below.



Task 1

The primary Linux source for summary system information about memory is provided by the special file **`/proc/meminfo`**. To see a snapshot of the current state of memory, use the following command:

```
cat /proc/meminfo | less
```

(the pipe through less may not be necessary if your shell window is tall enough). Find the following fields in the output (note that everywhere meminfo says “kB”, it really means “KiB”):

- **MemTotal**: the total amount of physical RAM (as opposed to virtual memory) which is potentially available for use.
- **MemFree**: the amount of physical memory currently free.
- **SwapTotal** & **SwapFree**: the total amount of potentially available swap space and the total amount of currently free swap space, respectively.
- **Slab** & **PageTables**: the sum of these is a lower bound on the amount of memory needed by the kernel (look up “slab allocation” in the index of [OSC]).

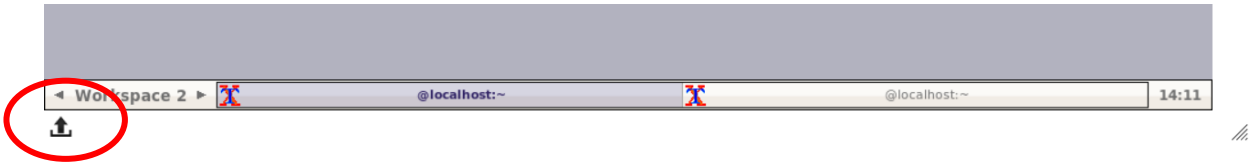
If you are curious about all the other fields, you can check the pages of RedHat and Linus Torvalds:

https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/deployment_guide/s2-proc-meminfo

https://github.com/torvalds/linux/blob/master/Documentation/filesystems/proc.rst

Download

The following tasks use simple C programmes to allocate memory. (If you are unfamiliar with C, don’t panic: you won’t need to write any new code for these exercises.) You need to download the source code of the files allocate.c and eatmymemory.c from Moodle. Once you have them in your computer, you can upload them to the unix environment with the upload button (black arrow pointing up) at the bottom of the screen:



If you are running this in a unix computer or a virtual box, you can download the files directly using wget like this

```
wget http://www.staff.city.ac.uk/~sbbk034/allocate.c
```

```
wget http://www.staff.city.ac.uk/~sbbk034/eatmymemory.c
```

Task 2

Study the source code `allocate.c`. This program will allocate memory for a certain number of *ints*. When you run it, it will ask you for a number *n* from the command line and then requests *n* times the amount of memory allocated to an int. Identify the part of the code which uses the function `malloc` to allocate memory. Note that the parameter passed to `malloc` is a number of bytes.

Compile the program with the following command:

```
gcc –o all1 allocate.c
```

This will create an executable called **all1** (use the **ls** command to check this). Run the executable like this

```
./all1
```

How many bytes have been allocated for **one** int? Was this the amount that you were expecting? Run **all1** again and increase the number. Eventually, there will be a problem and there will not be enough memory. What is the memory limit at which the programme outputs an error? Open a second terminal and run again **cat /proc/meminfo | less.** Does your previous result matches the memory shown in meminfo? We will explore further in the next task.

Task 3

Compile the program with the following command:

```
gcc –o emm eatmymemory.c
```

This will create an executable called **emm** (use the **ls** command to check this).

Study the source code **eatmymemory.c**. This program simply reads a number *n* from the command line and then requests *n* MiB of memory. Identify the part of the code which uses the function **malloc** to allocate memory. Note that the parameter passed to malloc is a number of bytes, so we have to multiply *n* by 1024x1024 (local variable mem). Note also that part of the code is commented-out (the code between /* and */). In a later task you will uncomment this code but leave it as it is for now.

Open a window and run **cat /proc/meminfo | less.** Leave the window open. Now, in a new shell window, run emm as follows:

```
./emm 300
```

The program simply requests 300 MiB of memory then immediately goes to sleep without actually using the memory. You may have an error code (… malloc FAILED) and you will have to request less memory. Open a new window and run again **cat /proc/meminfo | less.** You could run `emm` once more until you reach the limit of the memory. The total amount of virtual memory which has been allocated to these processes will depend on how much you were able to request. Say it is roughly 120 MiB (=122,880 KiB). Does this match with the numbers of total/free/available amounts of physical RAM available shown in your `/proc/meminfo`? How is this possible?

Task 4

A plausible explanation for the above behaviour is that many pages have been paged out to swap space. Use the information from `/proc/meminfo` to calculate the total amount of memory space available to processes on your environment: total physical RAM *plus* total swap space. Is there enough space available to explain what you saw in the previous task? In fact, Linux is being optimistic: it assumes that processes generally ask for more memory than they actually use. This is known as *over-committing*. If you think about how virtual memory works, overcommitment doesn't require any special steps to implement in a virtual memory system: the OS will always delay allocating a physical frame until the first attempt by the process to access the virtual page; when the page fault happens, physical memory can be allocated for the page; if the page is never used, the page fault never happens and no physical memory is allocated.

Look at the total amount of free memory being reported; rather little *physical* memory has actually been allocated to each `emm` process (how much?), because `eatmymemory.c` never uses *any* of the memory that it requests, so Linux's optimism was justified. But what happens when memory is over-committed and processes *do* try to use it? Let's find out.

You are going to modify `eatmymemory` so that it actually uses the memory it requests. First, kill all the currently running `emm` processes (the simplest way is just to enter Ctrl-C in each of the shell windows where the processes were launched; alternatively, you can use the `kill` command). Edit `eatmymemory.c` and uncomment the commented-out code by deleting /* and */. This code accesses the requested memory by writing the value 0 to the first memory location in each page (look at the for-loop and its use of the `memset` function). Recompile the program (see Task 2 above), call this emm2 and run it again. As before, run `emm` in separate shell windows and request a certain amount, say 100 MiB. Note that this version of `eatmymemory` prompts the user to press Enter to continue: **don't** press Enter just yet. At this point, the situation should be much the same as before, you can verify this by looking at `/proc/meminfo`. Now press `Enter` in the `emm` window and look again at `/proc/meminfo`. You should see that the amount of physical memory allocated to the process (in Active) jumps by approximately 100 MiB and the total amount of free memory drops by the same amount.

Task 5 [optional]

The kernel process which kills user processes when physical memory is exhausted is known as the OOM (Out Of Memory) killer. Some Linux users view memory over-commitment and the use of the OOM process to be a design flaw. Read one such view, together with an explanation of how to change the over-commit behaviour, here:

http://www.win.tue.nl/~aeb/linux/lk/lk-9.html#ss9.6.