# CSc 332 (L) - Operating Systems

Lab – Spring 2018

Instructor: Michael Iannelli, email: miannelli@gradcenter.cuny.edu

## Memory Management

April 27, 2018

### Introduction

One of the most-demanded resources in computers (other than the CPU) is system memory. Every process needs it since a process' code, stack, heap (dynamically-allocated structures), and data (variables) must all reside in memory. A processor reads instructions from memory and reads/writes data from/to memory. The functional interface of memory is:

- `value = read(address)`: read the contents of memory location at address
- `write(address, value)`: write value to the memory location at address

The memory manager is the part of the operating system that is responsible for allocating this resource to processes.

### Kernel vs User Memory

*Kernel Memory* comprises of :

- Text – where only the read-only parts of the program are stored. This is usually the actual instruction code of the program. Several instances of the same program can share this area of memory.
- Static Data – the area where preknown memory is allocated. This is generally for global variables and static C++ class members. The operating system allocates a copy of this memory area for each instance of the program.
- Memory Arena (also known as break space) – the area where dynamic runtime memory is stored. The memory arena consists of the heap and unused memory. The heap is where all user-allocated memory is located. The heap grows up from a lower memory address to a higher memory address.
- Stack – whenever a program makes a function call, the current function's state needs to be saved onto the stack. The stack grows down from a higher memory address to a lower memory address. A unique memory arena and stack exists for each instance of the program.

*User Memory* resides in the memory heap and is called by memory routines such as `malloc(), realloc(), free(), calloc()`

### Linking

A program is a bunch of code and data (variables, which may be initialized or uninitialized). For a program to execute as a process, it has to be loaded into memory via a program loader. The executable file that contains the program consists of several sections that identify different regions of the program and where they should be loaded into memory. Common regions include executable code ("text"), initialized variables ("data"), and zero-filled variables ("bss"). Some executable files also contain sections for storing an application icon, language-specific strings, and digital signatures to validate that the file has not been

modified. In addition to making room for these memory-resident sections, the program loader will have to allocate an area of memory for the program's stack, which stores return addresses of called functions, saved registers, and local variables and define an area for a heap, which stores dynamic requests for memory (e.g., malloc/free).

Complete programs are rarely compiled from a single file of code. In most cases, the program makes use of functions and variables that are external to the code in the file. For example, in C one is likely to use a printf function or system calls within a program. Neither of these are implemented by the user. Instead, they are located in libraries. If a program is compiled from several files, one file may refer to functions in another file and the other file may, for example, make references to global variables in the first file. The separately-compiled files are linked together to create the final executable file. Note that an executable program does not use symbol names; everything is be referred to by a memory address.

To handle the fact that an executable program will be assembled from several files, the compiler generates a a symbol table along with the compiled code for each file that it compiles. This symbol table contains textual names of functions and variables, their values (e.g., memory location or offset), if known, and a list of places in the code where the symbol is referenced. The job of a linker is to match up references to unknown symbols from compiled files (object file) with their known counterparts in another object file and generate a final executable file. The linker may also be provided with a library file, which is simply an archive (bundle, or collection) of object files that were generated from multiple source program files. The linker then searches through the archive to find the specific object modules that contain the symbols that need to be resolved (for example, the definition of printf). This entire process is called *static linking*. All symbols are resolved by the linker and any needed components from libraries are copied into the final executable file.

There are a few problems with static linking. One is that if a library changes (due to a bug, for example), then the version in the executable file is outdated. The second problem is that because many programs have a copy of common library elements (e.g., printf), the size of every program file is that much bigger. This is also reflected in the executable process: even if multiple processes use the same code, each must have its own copy. The desire to remedy this led to dynamic linking and shared libraries.

*Dynamic linking* is just-in-time linking. Instead of relying on a linker to put everything together into one executable program, the linking (cross-referencing of known and unknown symbols and placement in memory) is deferred to program load time. The static linking phase resolves symbols against a stub library instead of one that contains the actual library code. On first reference, the stub function in the linked library calls the operating system loader to load the actual function from the dynamic library file into memory, relacing the stub reference with the actual function.

Dynamic linking does not solve the issue of every process' use of a library consuming system memory. If twenty processes each use the printf function, each one will have its own copy of the code that implements it. Shared libraries add text (code) sharing onto dynamic linking. If a program needs to use code from a dynamic library, the stub function checks whether the needed library component is in memory. If not, the stub requests that the operating system loader load it. As with dynamic libraries, the stub is then replaced with the address of the library. If the shared library is already in another process' memory, the operating system creates a read-only shared memory region among the processes so that each process will have that code in its address space. To share this memory among multiple processes means that we will need to reserve a region of memory for shared libraries. Because different libraries may be loaded in a different order, we cannot predict what specific location a library will be loaded into. For this reason, it is important that shared libraries are compiled with position-independent code, containing only relative memory references instead of absolute addresses.

**Memory Usage**

It is often important to check memory usage and memory used per process on servers so that resources do not fall short and users are able to access the server. For example, if you are running a webserver, then the server must have enough memory to serve the visitors to the site. If not, the site would become very slow or even go down when there is a traffic spike, simply because memory would fall short. Its just like what happens on your desktop PC.

*Linux commands to check memory usage*

- `free` **command**: The `free` command shows a nicely summarized table containing information about the memory on your computer. Different columns are given to show information about Total memory, Used memory and Free memory. Typing the command as `free -m -t` shows you the information in MB units and a row at the bottom giving the Total amount of memory of each column.

- `/proc/meminfo`: The next way to check memory usage is to read the /proc/meminfo file. Know that the /proc file system does not contain real files. They are rather virtual files that contain dynamic information about the kernel and the system. Check the values of MemTotal, MemFree, Buffers, Cached, SwapTotal, SwapFree. They indicate same values of memory usage as the free command.

- `vmstat`: The `vmstat` command shows summary information about memory, processes, interrupts, paging and block I/O information. We're only interested in the information about memory. Typing the command as `vmstat -S M` will show the memory sizes in MB form, which is easier to read and understand. The Free column shows how much free memory we have. The "buff" column shows the amount of buffered memory, and "cache" shows the amount of cached information in the memory.

- `top` **command**: The "top" command is one of the most important utilities available for any user in Linux. It is just like the Task Manager in Windows and displays a list of the running tasks and various details about each application. The output is automatically updated often. The output of top contains the following fields:

  `PID, User, PR, NI, VIRT, RES, SHR, S, %CPU, %MEM, TIME+ and COMMAND.`

  While some of the fields are not important for us, the column you should be paying attention to is %MEM. This column displays the percentage of total memory that the application is using. For example, if the Xorg command is shown with 10% of memory on a computer running 2GB of total memory (physical + swap memory), then it is using around 200MB of memory.

- `pmap` **command**: reports memory map of a process

  Usage: `pmap [-x|-d] [-q] pid1, pid2, ...`; -x: Show the extended format; -d: Show the device format; -q: Do not display some header/footer lines.

***