**CS 344 – Module 1 Notes – Intro**

* **Operating systems** provide convenient abstractions over the hardware resources and standard services that application programmers can use in their programs. A major goal is to provide a convenient software interface to hardware resources.
* Standard services provided by an OS include:
  + **Process Management**
    - An OS is responsible for starting a new program and for ending them
    - Provides this functionality via the process abstraction, which we can think of as a running program
  + **File and Input/Output Management**
    - An OS provides file and directory abstractions, where a file organizes bits into meaning structures and directories organize files into the same
    - Also provides interfaces for reading and writing files as well as interfaces to communicate with external devices
  + **Inter-Process Communication (IPC)**
    - Processes running on the same machine as well as those running on different machines frequently need to communicate with one another
    - Provides support for IPC via various abstractions including signals, pipes, and network sockets
  + **Process Coordination**
    - Different processes running on an OS may need to access shared resources (i.e. files, devices, etc.) Access to these resources need to be coordinated. The OS provides the management for this
* **OS Kernel** – Responsible for providing the standard services that are listed above and is the program that is always running on the computer. It allocates its own memory space called **kernel memory**. To keep an OS running, even when applications fail or act maliciously, **kernel memory** must be protected from direct access by the applications on the OS. **System calls** control access to kernel memory, which is an API provided by the OS.
* There are two primary modes by which users interact with the OS
  + **Command Line Shells**
    - This requires the user to type commands to access OS services. Early OSs only supported command line shells
  + **GUIs**
    - This alleviates the difficulty in learning the various complicated commands and the high learning curve to use them on the user end.
* **Unix** – The most influential OS in history. Unix and its derivatives are the most used OSs in the world. Unix’s roots are in a project called **MULTiplexed Information and Computer Services** (**Multics** for short) and was a new time-sharing OS developed by AT&T Bell Labs, MIT, and GE. It was started in 1964 and was a large project beset by delays
  + Was originally written in assembly code but then switched to C in 1973
* **Unix Derivatives** are **Linux** and **Mobile OS**.
  + **Linus Torvald** is the chief and original developer of Linux in the early 1990s and was a Unix-like OS
  + Linux spawned many other derivatives such as Android
* **POSIX** was created to maintain compatibility between OSs and the **IEEE Computer Society** specified a family of standards called **Portable Operating System Interface** or **POSIX**. It defines standards for software compatibility across Unix variants as well as other OSs. POSIX also deals in standards with system calls
* There are many different shells used with Unix/Linux but we will be using the **BASH** shell (“Bourne-again shell” (/usr/local/bin/bash or /bin/bash))
* **Common Unix Commands**
  + **man** – Accesses the online reference manual
    - **man cp** would give us a detailed description of the “cp” command
  + **pwd** – print working directory (which directory am I in?)
  + **cd** – change directory (Moves your current working directory to a different one, cd HW1)
  + **ls** – displays all files in a given directory (using **ls -a** would show hidden files as well)
  + **mkdir** – create a directory
  + **rmdir** – remove a directory
  + **rm** – remove files (and directories if used recursively)
  + **mv** – move or rename files and directories
  + **cp –** copy files and directories
  + **chmod** – change mode, i.e. change the permissions of the files or directories
* **Commands for File Viewing and Selecting**
  + **cat** – concatenate character data stored in a file with another file (primary use is to dump data to the terminal)
  + **more** – take character data and display one screen-full at a time
  + **less** – similar to “more” command
  + **head** – display the beginning of a text file
  + **tail** – display the (tail) end of a text file
  + **grep** – search a text file
* **Shell Scripting –** All the commands that are accessible from the shell can be placed in a shell “script”. These scripts are executed line by line, as if the lines were being typed in, one by one. They provide the features of high-level programming languages, such as variables, conditional expressions, loops, etc.
  + Scripts are commonly used to automate frequent tasks and to simplify complex commands. For example, they may be written to run nightly backups of data, or to start up servers. They’re also used as a “glue” to connect different programs.
* **System Calls –** Mechanisms by which a user program asks the OS to perform services for it. Shell commands also make system calls to invoke OS services. Below are the 6 categories of system calls
  + **Process Control –** create process, terminate process, load/execute, get/set process attributes, wait for time/wait even/signal event, allocate and free memory
  + **File Management –** create/delete file, open/close, read/write/reposition, get/set file attributes
  + **Device Management –** request/release device, read/write/reposition, get/set device attributes, logically attach to detach devices
  + **Information Maintenance –** get/set date or time, get/set system data, get/set process/file/device attributes
  + **Communication –** create/delete communication connection, send/receive messages, transfer status info, attach/detach remote devices
  + **Protection –** get/set file permissions
* **System Calls for File Management in C**
  + **Creating a File –** We can create a file using the **open** system call and passing it the **O\_CREAT** flag. In this program, we are also passing the flag **O\_TRUNC** which will truncate (shorten) the file if it already exists. (See repl.it example in notes on Canvas)
* **Reading and Writing to a File** – the **open** call has a wider use than just creating a file. We can also use it to open existing files for a variety of purposes e.g. read only, write only, or both, etc. To write to a file, we can use the **write** call, while **read** is to read from a file.
* **System Calls and Library Functions**  - Many times, our programs may invoke C library functions and not involve system calls directly, such as **printf()** that in turn, invokes a system call such as **write()** beind the scenes. Another example is the **fopen()** function to open a file. Internally, this call invokes the **open()** system call to open a file.
  + The most common implementation of the standard C library is on Linux and is the **GNU C library**, commonly called **glibc**. We will be using glibc in this course.
* See [Exploration: Variables & Data Types, Input & Output in C: OPERATING SYSTEMS I (CS\_344\_400\_S2021) (oregonstate.edu)](https://canvas.oregonstate.edu/courses/1810930/pages/exploration-variables-and-data-types-input-and-output-in-c?module_item_id=20734083) for following notes
  + **Variables and Data Types –** In the program, we declare 2 variables: **int denominator;** and **int boundary;**
  + In C, variables must be explicitly declared at compile-time
  + variable type **int** can only store an integer value. C has three other variable types such as:
    - **float –** Can be used to store floating-point numbers, i.e. numbers containing decimal points
    - **double –** similar to float, can be used to store float-point numbers, but has double the precision of float
    - **char –** can be used to store a single character. Character values is specified within single quotes (i.e. ‘A’ or ‘1’ or ‘,’)
  + **The #include directive**
    - This directive tells the program to include code found in another file we created. This is to use functions defined outside of the main program. If we call a **printNums()** function defined in our **functions** file, we would use the line **#include functions.h.** Also called header files.
* **C doesn’t support strings as a basic data type. Strings are arrays of characters which is terminated by the null character.**
* **printf() parameters**
  + **%d** or **%i** – Decimal integer
  + **%x** – Hexadecimal integer
  + **%f** – Floating-point number
  + **%c** – ASCII character
* **Loops** generally have the following set up:
  + **for(initialization statement; loop condition; repeating statement) {**

statement or statements to execute

**}**

* **Pointers –** A pointer in C is a pointer to a specific data type, we can have a pointer to an int, char, float, etc. Pointers can also be used as variables and in expressions. There are two fundamental operators related to pointers
  + **\*p**  = returns the value pointed to by p
  + **&i** = returns the address of variable i
* we can declare pointers in 2 ways:
  + **type\* var;**
  + **type \*var;**
  + Below, type is the type of the variable of whose address the pointer will hold, e.g.
  + **int\***, **char\***, etc.
* In C, an expression with the name of the array is considered the same as a pointer to the first element of the array. For example, if we have an array of ints, **ints numbers [10]**, then the following 2 statements are correct:
  + **int\* ptr = numbers;**
  + **int\* ptr = &numbers[0];**

|  |  |  |  |
| --- | --- | --- | --- |
| Pointers to Arrays & Pointer Arithmetic | | | |
| **Value of the expression** | **Equivalent expression** | **Equivalent expression** | **Equivalent expression** |
| Address of the 1st element of the array | ptr | numbers | &numbers[0] |
| Address of ith element of the array | (ptr +i) | numbers + i | &numbers[i] |
| Value of the 1st element of the array | \*ptr | \*numbers | numbers[0] |
| Value of the ith element of the array | \*(ptr + i) | \*(numbers + i) | numbers[i] |

* **Memory Allocation**
* **Process Memory Layout** - The memory of a process is organized in a few segments. From the lowest address to the highest address the segments are Code, Data, Heap and Stack. Code and Data segment have fixed size. Heap and Stack can grow and shrink during the execution of a process. Typically (particularly on x86 architecture) heap grows towards higher address and stack grows towards lower address. These segments are shown in the figure below:
  + From the lowest address to the highest address, the segments are Code, Data, Heap, and Stack. Code and Data have fixed size. Heap grows towards higher addresses and Stack grows towards lower addresses.
  + The **data segment** contains the memory allocated for global and static variables. This memory is allocated at the start of the program.
  + The **stack segment** contains memory for non-static and non-global variables. They’re maintained in a LIFO order (such as the stack data structure). When a function is called, the stack grows. The arguments of the function are placed on the stack segment, followed by the return address of the calling function and then the local variables of the function itself. This is referred to as the **stack frame (or activation frame)** of the function. When a function returns, the stack frame of the function is removed from the stack segment, and the stack segment shrinks in size. Thus, the stack segment grows and shrinks as the process execution continues.
  + the **heap segment** contains memory that is dynamically allocated. The C functions **malloc** and **calloc** are used to allocate memory on the heap, while the function **free** is used to deallocate/free-up previously allocated memory.
* **Dynamic Memory Allocation** – **malloc** is declared as follows:
  + **void\* malloc(size\_t size);**
  + **malloc** takes one argument which is the size in bytes of the block of memory being requested (**size\_t** type is defined as an **unsigned int**). If malloc is successful, it returns a pointer to the memory allocated. If it fails to allocate a block, it returns **NULL**.
  + **calloc** is declared as follows:
    - **void \*calloc(size\_t numItems, size\_t size);**
  + The first argument to **calloc** specifies how many elements to allocate while the second argument specifies size of each element. **calloc**will try to allocate a block of **numItems** \* size bytes. **calloc** initializes the memory to 0. Just like malloc, **calloc** will return a pointer to the memory it allocated on success, and a NULL on failure.
* C provides a **sizeof** operator that returns the size of its argument in bytes. The argument of **sizeof** can be a data type, variable, expression, etc. It is very commonly used when calling malloc or calloc
  + The **free** function is used as follows:
    - **void free(void \*ptr);**
* **Memory Pitfalls**
  + **Memory Leaks** – Generally occurs when a program allocates memory on the heap, but doesn’t free it even when it is no longer needed. They reduce the amount of heap segment that is available for mem allocation. This can be a severe problem for long running programs such as servers and daemons.
  + **Buffer Flow** – These happen when a program writing data to a buffer in memory overruns the boundary of the buffer. (See buffer example at bottom of this page [Exploration: Memory Allocation: OPERATING SYSTEMS I (CS\_344\_400\_S2021) (oregonstate.edu)](https://canvas.oregonstate.edu/courses/1810930/pages/exploration-memory-allocation?module_item_id=20734101) )
* **Strings –** In C, they are simply an array of characters terminated by the null character. It is not a basic data type unlike many other “newer” languages such as Java or C++.
* Below are 2 different ways of declaring the same string with 10 characters:
  + **char\* myString = “my string”;**
  + **char myString[] = “my string”;**
  + While both complete the same objective, there are important differences in how memory is allocated in each case.
  + Remember, memory allocated on the stack is editable
* The function **strcmp** compares two string arguments, it is used as follows:
  + **int strcmp(const char \*s1, const char \*s2);**
  + Here, we are comparing strings **s1** and **s2**. If both are equal, it returns 0, otherwise it returns a non-zero value. (Note: you mustn’t use the **==** operator to compare two strings since that will compare the values of the two pointers rather than the content of the strings.
  + the related function **strncmp** takes a 3rd argument (**n**) and compares the first **n** characters of the two strings
* The function **strlen** returns the length of the string pointed to by the variable s, it is used as follows:
  + **size\_t strlen(const char \*s);**

**strcpy**

char \*strcpy(char \*dest, const char \*src);

The function strcpy copies the string pointed to by src to the buffer pointed to by dest. It returns the pointer dest. The copied string includes the null character at the end. If the destination buffer isn’t large enough to hold the string, this will cause buffer overflow.

The related function strncpy takes a 3rd argument n and copies the first n characters of src to dest. If src is shorter than n bytes, then strncpy sets the remaining bytes to the null character. If the first n bytes of the source string don’t contain the null character, then dest will not be null terminated.

**strcat**

char \*strcat(char \*dest, const char \*src);

The function strcat appends the string pointed to by src to the string pointed to by dest. The function overwrites the terminating null character in dest and adds a terminating null character at the end of the concatenated string. It returns the pointer dest. The array of characters pointed to by dest must have enough space for the concatenated string, otherwise buffer overflow will occur.

The related function strncat takes a 3rd argument n and concatenates at most n characters of src to dest.

**strdup**

char \*strdup(const char \*s);

The function strdup creates a duplicate of the string s using malloc and returns a pointer to this new string.

**strtok**

char \*strtok(char \*str, const char \*delim);

strtok can break a string into a sequence of nonempty tokens using the characters provided in the delim string as delimiters. The token returned by strtok is a null-terminated string that doesn’t include the character corresponding to the delimiting character using to tokenize the input string. If no more tokens are found then strktok returns NULL.

**strktok\_r**

char \*strtok\_r(char \*str, const char \*delim, char \*\*saveptr);

strktok\_r function is a reentrant version of strtok. We will study the concept of reentrant functions in a later module. But for tokenizing strings, this means that with strtok\_r we can parse different strings concurrently, whereas with strtok we cannot parse different strings concurrently. With strtok\_r, the 3rd argument saveptr is used to maintain context between successive calls to parse a string.