Operating System Lab

Author: Muhammad Salman

Last Updated: 2023-07-23

Contents

☒ Linux Commands + Basics of C
☒ Structure / Pointer
☐ Process Management
☒ fork(), getpid(), getppid()
☐ wait()
☐ execv()
☐ Scheduling Algo - FCFS

Introduction

C is a programming language developed at AT&T's Bell Laboratories in 1972 by Dennis Ritchie. It is widely known for its reliability and is highly popular among programmers. C is a portable language and is well-suited for structured programming. A C program is typically composed of a collection of functions.

System Requirements

To write and compile C programs, you will need the following:

- Hardware Requirement: Desktop computer or laptop
- Software Requirement: Linux operating system with GCC (GNU Compiler Collection)

GCC (GNU Compiler Collection)

- GCC is a widely used C compiler for Linux-based systems. It is typically operated via the command line. By default, GCC is often included in a Linux installation. To invoke GCC on a source code file, use the following command:
- `gcc filename`
- The default executable output of GCC is named "**a.out**", which can be run by typing `./a.out` in the command line. If you want to specify a different name for the executable output file, you can use the `-o` option followed by the desired output file name. For example:
- `gcc filename -o outputfile`

Linux Basic Commands

Command	Description	Example
help	Used to display the list of built-in commands of bash.	`help`
whereis	It is used to display the path of the package for specific built-in Linux commands.	`whereis ls`
whatis	It is used to display the description of built-in Linux commands.	`whatis grep`
w	`w` is used to display information about currently logged-in users and their activities.	`w`
passwd	`passwd` is used to change the password of the currently logged-in user.	`passwd`
sudo	`sudo` stands for "Superuser Do". It is used to execute a command with administration privileges.	`sudo some_command`
history	`history` is used to display a list of previously executed commands.	`history`
pwd	`pwd` is used to display the current working directory.	`pwd`
ps	`ps` is used to display the information of processes running at that time, including their process ID.	`ps -aux | grep firefox`
htop	`htop` is an interactive process viewer that displays real-time information about processes in a more user-friendly way.	`htop`
kill -9 pid	`kill` is used to send a signal to terminate a process. The `-9` option sends a SIGKILL signal, forcefully terminating the process with the specified process ID (pid).	`kill -9 1234`
df	`df` stands for "disk free". It reports filesystem disk space usage	`df -h`
free	`free` is used to display amount of free and used memory in the system	`free -h`
clear	`clear` is used to clear the terminal screen.	`clear`
shutdown	`shutdown` is used to shut down the system.	`shutdown now`
exit	`exit` is used to exit from the terminal.	`exit`
dir	`dir` is used to display all the folders present in the current working directory.	`dir`
ls	`ls` is used to display the list of files and directories in a given directory.	`ls -a`
cd	`cd` is used to navigate from one folder to another.	`cd folder_name`
cd ..	`cd ..` is used to move one step back from the current folder.	`cd ..`
mkdir	`mkdir` is used to create a directory/folder.	`mkdir folder_name`
rmdir	`rmdir` is used to remove/delete an empty directory/folder.	`rmdir folder_name`
rm	`rm` is used to remove/delete a file or directory/folder.	`rm filename.extension`
touch	`touch` is used to create a file.	`touch filename.extension`
cp	`cp` is used to copy one or more files or folders.	`cp source destination`
mv	`mv` is used to move one or more files or folders.	`mv source destination`
cat	`cat` is used to display the contents of a file.	`cat filename.extension`
tac	`tac` is used to display the contents of a file in reverse order.	`tac filename.extension`
head	`head` is used to display the first few lines of a file.	`head -n 5 filename.extension`
tail	`tail` is used to display the last few lines of a file.	`tail -n 5 filename.extension`
echo	`echo` is used to display a specific message or text on the terminal.	`echo "Some text to be written" > filename.extension`
grep	`grep` is used to search for a pattern/text within one or more files.	`grep "text_to_search" filename.extension`
zip	`zip` is used to create compressed archive files/folders in the ZIP format.	`zip <fileName.zip> <files to be zipped>`
unzip	`unzip` is used to extract the contents of a ZIP archive.	`unzip filename.zip`
date	`date` is used to display the current date and time.	`date`
timedatectl	`timedatectl` is used to view and modify the system's date, time, and time zone.	`timedatectl`
man	`man` command is used to display the documentation for various commands, programs, and system functions.	`man command_name`
strace	`strace` is used to trace system calls and signals of a program.	`strace command_name`
Pipe (|)	The pipe command is used to redirect the output of one command to another command as input.	`cat name.text | wc`
wc	`wc` command is used to count lines, words, and characters in a file or standard input.	`cat filename.extension | wc -l`
Reset History	To clear the terminal history permanently, you can use the following command:	`history -c`

Fundamentals of C

Datatypes

C provides several built-in datatypes to represent different kinds of data. Here are some commonly used datatypes in C:

- **int**: Used to store integer values.
- **float**: Used to store floating-point numbers.
- **char**: Used to store individual characters.
- **double**: Used to store double-precision floating-point numbers.
- ~~**void**: Used to indicate the absence of a data type.~~

Operators in C - Mostly Resembles JAVA Operators (Functionality)

Operator	Description	Example
Arithmetic Operators	Perform mathematical operations	`a + b`, `x * y`, `z / 2`
Relational Operators	Compare values	`x > y`, `a == b`, `z <= 5`
Logical Operators	Perform logical operations	`x && y`, `a || b`, `!flag`
Bitwise Operators	Perform bitwise operations	`x & y` (Bitwise AND)`a | b` (Bitwise OR)
Assignment Operators	Assign values	`x = y`, `a += 5`, `z *= 2`
Conditional Operator	Conditional (ternary) operator	`x > y ? x : y`
Member Access Operators	Access members of structures	`structVar.member` (Member/Dollar Operator)`ptr->member` (SPO - Structure Pointer Operator)
Sizeof Operator	Determine size of a type	`sizeof(int)`, `sizeof(struct)`
Pointer Operators	Manipulate pointers	`&variable` (Address of Operator)`*ptr` (Dereference Operator)

C Input and Printing

- In C programming, you can take input from the user and display output using the `scanf` and `printf` functions, respectively.

Input using `scanf`

- To read input from the user, you can use the `scanf` function. It allows you to receive input and store it in variables. Here's an example:

```c
int number;
printf("Enter a number: ");
scanf("%d", &number);
```

- In the above code, scanf is used to read an integer input from the user and store it in the variable number.

Output using `printf`

- To display output to the console, you can use the printf function. It allows you to print formatted output. Here's an example:

```c
int age = 25;
printf("My age is %d", age);
```

- In the above code, printf is used to print the value of the age variable.

Format Specifiers

- Format specifiers are placeholders that define the type and format of the input/output data.

Data Type	Format Specifier
Integer	`%d`
Floating-Point	`%f`
Double	`%lf`
Character	`%c`
String	`%s`

- You can use these format specifiers within the `scanf` and `printf` functions to correctly handle the input and output.
- Remember to include the `stdio.h` header file at the beginning of your program to use the `scanf` and `printf` functions.

```c
#include <stdio.h>

int main() {
// Code here
return 0;
}
```

Decision Structure

Decision structures in C allow you to control the flow of program execution based on certain conditions. The common decision structures in C are:

1. if Statement - Similar to JAVA

- The `if` statement allows you to execute a block of code if a condition is true. Here's the syntax:

```c
if (condition) {
// Code to be executed if the condition is true
}
```

3. Switch Statement - Similar to JAVA

- The `switch` statement allows you to select one of many code blocks to be executed based on the value of an expression. Here's the syntax:

```c
switch (expression) {
case constant1:
// Code to be executed if expression matches constant1
break;
case constant2:
// Code to be executed if expression matches constant2
break;
default:
// Code to be executed if expression doesn't match any constant
}
```

- The `break` statement is used to exit the switch block.

C Keywords

- typedef

- **Defines a new type alias**

```c
typedef int Age;

int main() {
Age myAge = 25;
return 0;
}
```

- auto

- **Specifies automatic storage duration**

```c
int main() {
auto int x = 5;
return 0;
}
```

- goto

- **Transfers control to a labeled statement**

```c
int main() {
int i = 0;

start:
printf("%d\n", i);
i++;

if (i < 10) {
goto start;
}

return 0;
}
```

- continue

- **Skips the rest of the loop and continues with the next iteration**

```c
int main() {
for (int i = 1; i <= 10; i++) {
if (i % 2 == 0) {
continue;
}
printf("%d\n", i);
}

return 0;
}
```

- sizeof

- **Returns the size of a data type or variable**

```c
int main() {
int x = 10;
size_t size = sizeof(x);

printf("Size of x: %zu bytes\n", size);

return 0;
}
```

- To find the size of an array using `sizeof`, you can divide the size of the whole array by the size of its individual elements. Here's an example:

```c
#include <stdio.h>

int main() {
int numbers[] = {1, 2, 3, 4, 5};
int size = sizeof(numbers) / sizeof(numbers[0]);

printf("Size of the array: %d\n", size);

return 0;
}
```

Loops

for loop - same as JAVA

while loop - same as JAVA

do-while loop - same as JAVA

Functions & Arrays

In C programming, functions and arrays are fundamental concepts. Here's a brief explanation of functions and arrays, their return types, how to return an array, and how to pass an array as a parameter and argument.

- ☐ Function Prototyping - For Future Releases

Function

- A function is a block of code that performs a specific task. It can accept input parameters, perform operations, and optionally return a value.
- The syntax for declaring and defining a function in C is as follows:

```
return_type function_name(parameter_list) {
// Function body
// Code to be executed
// Optional return statement
}
```

- `return_type`: Specifies the data type of the value the function will return (e.g., int, float, void for no return value).
- `function_name`: Name of the function that uniquely identifies it.
- `parameter_list`: Optional list of input parameters (data type and variable name) that the function accepts.

Arrays

- An array is a collection of elements of the same data type stored in contiguous memory locations.
- It allows storing multiple values of the same type under a single variable name. The syntax for declaring and defining an array in C is as follows:
- `data_type array_name[array_size];`

```
#include <stdio.h>

int main() {
// Integer array declaration and initialization
int numbers1[] = {1, 2, 3, 4, 5};
// Character array (string) declaration and initialization
char name1[] = "John";
char name2[6] = {'M', 'a', 'r', 'y', '\0'};
// Double array declaration and initialization
double prices[3] = {9.99, 19.99, 29.99};

// Print the arrays
printf("Integer Array 1: ");
for (int i = 0; i < sizeof(numbers1) / sizeof(numbers1[0]); i++) {
printf("%d ", numbers1[i]);
}
printf("\n");

printf("Character Array 1 (String): %s\n", name1);
printf("Character Array 2 (String): %s\n", name2);

printf("Double Array: ");
for (int i = 0; i < sizeof(prices) / sizeof(prices[0]); i++) {
printf("%.2f ", prices[i]);
}
printf("\n");

return 0;
}
```

Returning an Array from a Function

- In C, it is not possible to directly return an entire array from a function. However, you can return a pointer to the array.

```
#include <stdio.h>

// Function to print an array and return the same array
int* printAndReturnArray(int arr[], int size) {
for (int i = 0; i < size; i++) {
printf("%d ", arr[i]);
}
printf("\n");
return arr;
}
```

```c
int main() {
int numbers[] = {1, 2, 3, 4, 5};
int size = sizeof(numbers) / sizeof(numbers[0]);

// Print the array using the function and assign the returned array to a new array
int* returnedArray = printAndReturnArray(numbers, size);

// Access the elements of the returned array
printf("Returned Array: ");
for (int i = 0; i < size; i++) {
printf("%d ", returnedArray[i]);
}
printf("\n");

return 0;
}
```

Pointers & Structures

- Pointers and structs are important concepts in C programming.

Pointers

- A pointer is a variable that stores the memory address of another variable.
- Allows you to indirectly access and manipulate the value stored in memory.
- Pointers are widely used in C for various purposes, such as dynamic memory allocation, passing arguments by reference, and efficient memory manipulation.

Declaration

- To declare a pointer variable, you use the asterisk (*) symbol before the variable name. For example:

```c
int num = 10;
int *ptr = &num; // Assigns the address of 'num' to 'ptr'
```

- A good convention is to use asterisk(*) before the variable name and not with the datatype during pointer declaration
- To **access the value** stored at the memory location pointed by a pointer, you use the **dereference operator** (*). For example:

```c
// Retrieves the value stored at the memory location pointed by 'ptr'
int value = *ptr;
```

- Pointers are also used in dynamic memory allocation using functions like malloc and free. They provide flexibility in managing memory resources in C.

Common Mistakes

Code Example	Error	Solution
`pc = c;` *Explanation*: Assigning a non-address value to a pointer.	Error: Incompatible types	Assign the address of `c` to `pc` using the `&` operator `pc = &c`

`*pc = &c;`
Explanation: Assigning an address to a non-pointer variable.
	Error: Incompatible types	Assign the value of `c` to content at pointer using the dereference operator
		`pc = &c`
		OR
		`*pc = c`

`pc = &c;` *Explanation*: Assigning the address of variable `c` to `pc`.	No Error	N/A
`*pc = C;` *Explanation*: Assigning the value of `C` to the variable pointed by `pc`.	No Error	N/A

Structures (struct)

- A struct is a user-defined data type that allows you to group different variables of different types under a single name.
- It enables you to create custom data structures to represent complex entities.
- To define a `struct`, you use the `struct` keyword followed by the struct name and a list of member variables enclosed in curly braces. For example:

```c
struct Person {
char name[50];
int age;
float height;
};

// Declaring struct variable
struct Person person1;
// Assigns the value 25 to the 'age' member of 'person1'
person1.age = 25;
```

Returning struct from function

```c
#include <stdio.h>
// Defining Employee Struct
struct Employee {
char name[32];
char address[100];
int salary;
};
// Function Prototyping
struct Employee *inputData(struct Employee *emp, int num);
struct Employee *printData(struct Employee *emp, int num);

int main() {
struct Employee emp1;
struct Employee *inputReturningPointer, *printReturningPointer;

inputReturningPointer = inputData(&emp1, 1);
printReturningPointer = printData(inputReturningPointer, 1);
printData(printReturningPointer, 1);

return 0;
}

struct Employee *inputData(struct Employee *emp, int num) {
printf("\n----------------Inputting Employee #%d Data ----------------\n", num);
printf("Enter the employee name: ");
scanf(" %[^\n]", emp->name);
printf("Enter the employee address: ");
scanf(" %[^\n]", emp->address);
printf("Enter the employee salary: ");
scanf("%d", &(emp->salary));
return emp;
}

struct Employee *printData(struct Employee *emp, int num) {
printf("\n\t\t----------------Printing Employee #%d Data ----------------\n", num);
printf("\t\tEmployee Name: %s\n", emp->name);
printf("\t\tEmployee Address: %s\n", emp->address);
printf("\t\tEmployee Salary: %d\n", emp->salary);
return emp;
}
```

Code	Explanation
`printf("Enter the employee name: ");`	Display a prompt for the user to enter the employee name
`scanf(" %[^\n]", emp->name);`	Read input from the user and store it in the `name` field of `emp`. The format specifier `%[^\n]` is used to accept input till a new line is encountered, including any leading whitespace. This is commonly used with strings. The input is then stored in the `name` field of the `emp` structure.

Process Management

Fork()

- The `fork()` function **creates a new process** by duplicating the existing process. The new process is called the child process, and the original process is called the parent process.
- The `fork()` function **returns twice**: once in the parent process and once in the child process. The return **value in the parent process is the process ID (PID) of the child process**, and the return **value in the child process is 0.**
- After the `fork()` call, the child process inherits a copy of the parent process's memory, file descriptors, and other attributes. However, the child process has its own separate execution flow from that point onward.
- Here's the basic syntax of the `fork()` function:

```c
#include<stdio.h>
#include<unistd.h>      // for fork(), execv()
#include<sys/types.h>   // for pid_t


int main(){

pid_t forkReturn;
forkReturn = fork();

if(forkReturn == -1)
perror("There was an error in process creation");
else if(forkReturn == 0){
printf("Hello from child with PID %d\n", getpid());
// get the parent_process_id using "getpid()"
printf("Hello I'm the child with PID %d and my parent is %d\n", getpid(), getppid());
}else{
// fork() returns child_id to parent
// so forkReturn in this case will've child_id
printf("Hello from the parent of the child with %d\n", forkReturn);
```

```
// current process_id can be get using getpid()
printf("Hello I'm the parent with PID %d\n", getpid());
}

return 0;
}
```

- **Note:** Both processes will run the code that follows the conditional statement. Normally this is undesirable, and by forcing the child to terminate after it has done its "useful work", we can prevent it. Pay attention to the conditional command, you may want to use exit or break. The code shown above can be modified as follows

```
else if(forkReturn == 0){
/* this is the child process */
// Do some work in the child process

// Terminate the child process after its work is done
exit(0);
}
```

fork() summary

- When the `fork()` function is called, it returns a value that determines the behavior of the program in different processes. The return value is:
- -1 if an error occurs during the creation of the child process.
- 0 in the child process, indicating that it is the child process.
- A positive value (process ID) in the parent process, indicating the process ID of the newly created child process.
- The child process is an exact copy of the parent process, including its code, data, and stack. However, the child process has its own unique process ID.
- After the `fork()` call, both the parent and child processes continue executing from the point immediately following the `fork()` call. The operating system determines the order of execution between the parent and child processes.

exec()

- `exec()` is family of functions that're used to replace the current process image with a new process image.
- More precisely, we can say that using exec system call will replace the old file or program from the process with a new file or program. The entire content of the process is replaced with a new program.
- It loads the new program into current process space and runs it from the entry point.

execv()

- `execv()` is a function from the `unistd.h` header file in C, and it is used to replace the current process image with a new process image.
- The `execv()` function requires the path to the executable file as the first argument and an array of strings as the second argument.
- The first element of the array of strings should be the path to the executable file. The last element of the array must be `NULL` to indicate the end of the argument list.
- The executable file should have the appropriate execute permission to be executed using `execv()`.
- When `execv()` is called, the current process is replaced with the new process, and the new process starts executing from the beginning of its code.
- If `execv()` is successful, it does not return. If there is an error, it returns -1, and you can use `perror()` or `errno` to determine the cause of the error.

```
// Demo.c
#include<stdio.h>
#include<unistd.h>

int main(){
printf("%s", "Hello World from Demo File");

return 0;
}
```

Running User Programs using execv()

- Make sure to compile the `Demo.c` file and have the respective binary file that's to be called in other program.

```
// ExecCall.c
#include<stdio.h>
#include<unistd.h>

int main(){
char *path = "./Demo";
char *args[] = {path, NULL};
execv(path, args);

printf("There was an error while replacing the program");

return 0;
}
```

```c
#include<stdio.h>
#include<unistd.h>
#include<errno.h>
#include<string.h>

int main(){
char *path = "/bins/ls";
char *args[] = {path, "-ltr", NULL};

// char *argv_list[] = {"ls", "-lart", "/home", NULL};
// execv("ls", argv_list);
// exit(0);

// The "ls" command is executed with the options "-lart" and the directory "/home".
// The last element of the array must be NULL to indicate the end of the argument list.

execv(path, args);

// Error Handling
if (execv(path, args) == -1) {
perror("Error executing command");
printf("There was an error running the utility\n");
}

return 0;
}
```

Running Command-Line Utility using execv()

`fork()` and `exec()`

- `fork()` starts a new process which is a copy of the one that calls it, while `exec()` replaces the current process image with another (different) one.
- Both parent and child processes are executed simultaneously in case of `fork()` while Control never returns to the original program unless there is an `exec()` error.

wait()

- The `wait()` function is used to **make a parent process wait** until one of its child processes terminates.
- When a child process terminates, its **exit status is available** to the parent process **through the `wait()` function**.
- The `wait()` function suspends the execution of the parent process until a child process terminates.
- If there are no child processes, the `wait()` function returns immediately with an error.
- It can continue its execution without going to the **blocked state**
- The `wait()` function returns the process ID (PID) of the terminated child process.
- By using the `wait()` function, the parent process can ensure proper synchronization and obtain the exit status of the child process.
- If multiple child processes have terminated, the `wait()` function returns the PID of any terminated child process, allowing the parent process to handle them sequentially.
- If the parent process doesn't need to obtain the exit status of the child process, it can use the `wait(NULL)` form of the function.

FoodForThought

- The behavior of the `wait()` function depends on how it is called by the parent process. By default, when the parent process calls `wait()`, it will suspend its execution until any one of its child processes terminates. Once a child process terminates, the parent process will resume its execution and can continue with the remaining code.
- If there are 5 child processes and 2 of them have terminated, the parent process can continue its execution after the first terminated child process is handled by `wait()`. It does not have to wait for all the child processes to terminate before resuming its execution.
- If the parent process needs to wait for all the child processes to terminate before continuing, it can modify the code accordingly. For example, it can maintain a counter to track the number of terminated child processes and use a loop to repeatedly call `wait()` until the counter matches the total number of child processes. This way, the parent process will resume its execution only after all child processes have terminated.

```c
#include<stdio.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/wait.h>

int main(){

// child_pid will hold the process ID of each child process
// wpid will be used to wait for child processes
// status will store the exit status of child processes
// n is the number of child processes to create.
int n = 5, status = 0;
pid_t child_id, wpid;


// parent process creates n child processes
// child process prints its pid, simulate some work
// prints message indicating that it has finished
// exit(0) will terminate the child, essentially
// stopping it from running the remaining code outside
// the conditional statement
for (int id = 0; id < n; id++) {
```

```c
if ((child_pid = fork()) == 0) {
// Child code
printf("Child process %d with PID %d\n", id, getpid());
sleep(1); // Simulating some work
printf("Child process %d with PID %d finished\n", id, getpid());
exit(0);
}
}


// After creating all child processes, the parent process enters a
// while loop with wait(). This loop waits for each child process  to terminate
// The wait() function suspends the execution of the parent process until a child process terminates
// The return value of wait() is the process ID of the terminated child process, and we store it in wpid
// We continue the loop as long as wait() returns a value greater than 0,
// indicating there are still child processes running.
while ((wpid = wait(&status)) > 0);
// The wait() function also takes a pointer to an integer where it stores the
// exit status of the terminated child process.


printf("All child processes have terminated. Parent process continues.\n");

// Parent Process Remaining Code...

return 0;
}
```

exit()

- The `exit()` function is used to terminate the current process and return an exit status to the operating system.
- The exit status value `0` typically indicates successful termination, while a non-zero value indicates an error or abnormal termination.
- The `exit()` function performs various cleanup tasks, including flushing buffered output, closing open files, and releasing allocated memory.
- If the `exit()` function is not explicitly called in the program, the process will exit implicitly when it reaches the end of the `main()` function.