Ass-1] Create a file with hole in it

Creating a file with a hole in it is a common concept in Unix-like operating systems, where you can create a file that has empty or unwritten space within it. This is achieved using the lseek system call to create a hole in the file. Below is a C program that demonstrates how to create such a file:

This program does the following:

1. Opens a file named "file_with_hole.txt" with write permissions.
2. Uses lseek to seek to a position one byte before the desired file size. This creates a hole in the file.
3. Writes some data at the end of the file.
4. Closes the file.

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>

int main() {
    int fd;
    char filename[] = "file_with_hole.txt";
    int file_size = 1024; // Size of the file in bytes

    // Create a new file with write permissions
    fd = open(filename, O_CREAT | O_WRONLY, S_IRUSR | S_IWUSR);
    if (fd == -1) {
        perror("open");
        exit(EXIT_FAILURE);
    }

    // Seek to a position beyond the end of the file (creating a hole)
    if (lseek(fd, file_size - 1, SEEK_SET) == -1) {
        perror("lseek");
        close(fd);
        exit(EXIT_FAILURE);
    }
```

```
    // Write something at the end of the file
    char data[] = "Hello, this is data at the end of the file.";
    ssize_t bytes_written = write(fd, data, sizeof(data) - 1);
    if (bytes_written == -1) {
        perror("write");
        close(fd);
        exit(EXIT_FAILURE);
    }

    // Close the file
    close(fd);

    printf("File with a hole created: %s\n", filename);

    return 0;
}
```

ASS-2] Take multiple files as Command Line Arguments and print their inode number

You can use the stat system call to retrieve information about files, including their inode numbers. Here's a C program that takes multiple file names as command-line arguments and prints their respective inode numbers:

Here's how this program works:

1. It checks whether there are command-line arguments (file names) provided. If not, it prints a usage message and exits.
2. It iterates through the provided file names one by one.
3. For each file name, it uses the stat function to retrieve file information and stores it in the file_info structure.
4. It prints the file name and its associated inode number.
5. It repeats the process for all the provided files.
6. You can compile this program using a C compiler, and then run it with one or more file names as command-line arguments. It will print the inode numbers of the specified files. For example:

Run ./print_inode file1.txt file2.txt
Replace file1.txt and file2.txt with the actual file names you want to examine.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
    if (argc < 2) {
        fprintf(stderr, "Usage: %s <file1> <file2> ... <fileN>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    for (int i = 1; i < argc; i++) {
        const char *filename = argv[i];
        struct stat file_info;

        if (stat(filename, &file_info) == -1) {
            perror("stat");
            continue; // Skip to the next file on error
        }

        printf("File: %s\n", filename);
        printf("Inode Number: %ld\n", (long)file_info.st_ino);
        printf("---------------\n");
    }

    return 0;
}
```

Assig-3] Write a C program to find file properties such as inode number, number of hard link, File permissions, File size, File access and modification time and so on of a given file using stat() system call.

You can use the stat system call to retrieve various file properties such as inode number, number of hard links, file permissions, file size, access time, and modification time. Here's a C program that demonstrates how to do this:

Here's how this program works:

1. It checks if there is exactly one command-line argument (the filename) provided. If not, it prints a usage message and exits.
2. It uses the stat function to retrieve file information and stores it in the file_info structure.
3. It prints various file properties, including the inode number, number of hard links, file permissions, file size, access time, and modification time.
4. It converts the access and modification times to a human-readable format using strftime.

To use this program, compile it with a C compiler and run it with the filename of the file you want to inspect as a command-line argument, like this:

Run the command ./file_properties_example my_file.txt

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <unistd.h>
#include <time.h>

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <filename>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    const char *filename = argv[1];
    struct stat file_info;
```

```
    if (stat(filename, &file_info) == -1) {
        perror("stat");
        exit(EXIT_FAILURE);
    }

    printf("File Properties for: %s\n", filename);
    printf("Inode Number: %ld\n", (long)file_info.st_ino);
    printf("Number of Hard Links: %ld\n", (long)file_info.st_nlink);
    printf("File Permissions: %o\n", file_info.st_mode & 0777);
    printf("File Size: %ld bytes\n", (long)file_info.st_size);

    // Convert access and modification times to a human-readable format
    char access_time_str[20];
    char mod_time_str[20];
    strftime(access_time_str, sizeof(access_time_str), "%Y-%m-%d %H:%M:%S",
localtime(&file_info.st_atime));
    strftime(mod_time_str, sizeof(mod_time_str), "%Y-%m-%d %H:%M:%S",
localtime(&file_info.st_mtime));

    printf("Access Time: %s\n", access_time_str);
    printf("Modification Time: %s\n", mod_time_str);

    return 0;
}
```

Assign-4] Print the type of file where file name accepted through Command Line
To print the type of file (e.g., regular file, directory, symbolic link) based on the provided file name through the command line, you can use the stat system call to retrieve file information, specifically the st_mode field from the struct stat. Here's a C program to do that:

Here's how this program works:

1. It checks if there is exactly one command-line argument (the filename) provided. If not, it prints a usage message and exits.
2. It uses the stat function to retrieve file information and stores it in the file_info structure.
3. It examines the st_mode field in the struct stat to determine the file type using various S_IS* macros provided by POSIX. These macros check if the file's mode matches the specified type (e.g., regular file, directory, symbolic link, etc.).
4. It prints the type of file based on the determined file type.

To use this program, compile it with a C compiler and run it with the filename of the file you want to inspect as a command-line argument, like this:

Run the Command ./file_type_example my_file.txt
Replace my_file.txt with the actual file you want to determine the type for.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <filename>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    const char *filename = argv[1];
    struct stat file_info;

    if (stat(filename, &file_info) == -1) {
        perror("stat");
        exit(EXIT_FAILURE);
    }

    if (S_ISREG(file_info.st_mode)) {
        printf("%s is a regular file.\n", filename);
    } else if (S_ISDIR(file_info.st_mode)) {
        printf("%s is a directory.\n", filename);
    } else if (S_ISLNK(file_info.st_mode)) {
```

```
        printf("%s is a symbolic link.\n", filename);
    } else if (S_ISFIFO(file_info.st_mode)) {
        printf("%s is a named pipe (FIFO).\n", filename);
    } else if (S_ISSOCK(file_info.st_mode)) {
        printf("%s is a socket.\n", filename);
    } else if (S_ISCHR(file_info.st_mode)) {
        printf("%s is a character special file.\n", filename);
    } else if (S_ISBLK(file_info.st_mode)) {
        printf("%s is a block special file.\n", filename);
    } else {
        printf("%s is of unknown file type.\n", filename);
    }

    return 0;
}
```

Assign-5] Write a C program to find whether a given file is present in current directory or not.

To determine whether a given file is present in the current directory or not, you can use the access or stat system calls in C. Here's a simple C program that uses the access function:
Here's how this program works:

1. It checks if there is exactly one command-line argument (the filename) provided. If not, it prints a usage message and exits.
2. It uses the access function with the F_OK mode to check if the file exists in the current directory. If the function returns 0, the file is present; otherwise, it's not.
3. To use this program, compile it with a C compiler and run it with the filename you want to check as a command-line argument, like this:

Run the Command ./check_file_presence my_file.txt
Replace my_file.txt with the actual filename you want to check. The program will inform you whether the file is present in the current directory or not.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
```

```
int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <filename>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    const char *filename = argv[1];

    if (access(filename, F_OK) == 0) {
        printf("The file '%s' is present in the current directory.\n", filename);
    } else {
        printf("The file '%s' is not present in the current directory.\n", filename);
    }

    return 0;
}
```

6] Write a C program that a string as an argument and return all the files that begins with that name in the current directory. For example > ./a.out foo will return all file names that begins with foo .

Solu:- To achieve this, you can use the opendir, readdir, and closedir functions to read the contents of the current directory, and then check if each file name starts with the provided string argument. Here's a C program that does this:

- Save the code in a file (e.g., list_files.c) and then compile it using a C compiler. Here's how you can compile and run the program:
- Open your terminal.
- Navigate to the directory where you saved list_files.c.
- Compile the program using the following command:

  gcc list_files.c -o list_files
- Run the program with your desired prefix:

  ./list_files foo

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <dirent.h>

int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf("Usage: %s <prefix>\n", argv[0]);
        return 1;
    }
    char *prefix = argv[1];
    int prefix_length = strlen(prefix);

    DIR *dir;
    struct dirent *entry;

    if ((dir = opendir(".")) == NULL) {
        perror("opendir");
        return 1;
    }
    printf("Files starting with '%s' in the current directory:\n", prefix);
    while ((entry = readdir(dir)) != NULL) {
        if (strncmp(entry->d_name, prefix, prefix_length) == 0) {
            printf("%s\n", entry->d_name);
        }
    }

    closedir(dir);
    return 0;
}
```

7] Read the current directory and display the name of the files, no of files in current directory

To read the current directory and display the names of the files along with the number of files in the current directory, you can use the following C program:

- Save this code in a file (e.g., list_files_count.c) and then compile it using a C compiler. Here's how you can compile and run the program:

- Open your terminal.
- Navigate to the directory where you saved list_files_count.c.
- Compile the program using the following command:
  gcc list_files_count.c -o list_files_count
- Run the program:
- ./list_files_count
- The program will list the names of files in the current directory and display the total number of files in that directory.

```
#include <stdio.h>
#include <stdlib.h>
#include <dirent.h>

int main() {
    DIR *dir;
    struct dirent *entry;
    int fileCount = 0;

    if ((dir = opendir(".")) == NULL) {
        perror("opendir");
        return 1;
    }

    printf("Files in the current directory:\n");
    while ((entry = readdir(dir)) != NULL) {
        if (entry->d_type == DT_REG) { // Check if it's a regular file
            printf("%s\n", entry->d_name);
            fileCount++;
        }
    }

    closedir(dir);

    printf("\nNumber of files in the current directory: %d\n", fileCount);

    return 0;
}
```

8] Write a C program which receives file names as command line arguments and display those filenames in ascending order according to their sizes. I) (e.g $ a.out a.txt b.txt c.txt, …)

You can create a C program to receive file names as command line arguments and then display those filenames in ascending order according to their sizes using the following code:

- Save this code in a file (e.g., sort_files_by_size.c) and then compile it using a C compiler. Here's how you can compile and run the program:
- Open your terminal.
- Navigate to the directory where you saved sort_files_by_size.c.
- Compile the program using the following command:
  gcc sort_files_by_size.c -o sort_files_by_size
- Run the program with the desired file names as command line arguments:
  ./sort_files_by_size a.txt b.txt c.txt
- The program will display the filenames in ascending order according to their sizes.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>

// Custom data structure to store filename and file size
struct FileInfo {
    char *filename;
    off_t size;
};
// Custom comparison function for sorting FileInfo structures
int compareFileInfo(const void *a, const void *b) {
    return ((struct FileInfo*)a)->size - ((struct FileInfo*)b)->size;
}
```

```
int main(int argc, char *argv[]) {
    if (argc < 2) {
        fprintf(stderr, "Usage: %s <file1> <file2> ...\n", argv[0]);
        return 1;
    }
    // Create an array of FileInfo structures to store filename and size
    struct FileInfo *fileInfoArray = (struct FileInfo*)malloc((argc - 1) * sizeof(struct
FileInfo));
    if (!fileInfoArray) {
        perror("malloc");
        return 1;
    }
    // Fill the array with filenames and their corresponding sizes
    for (int i = 1; i < argc; i++) {
        struct stat fileStat;
        if (stat(argv[i], &fileStat) == -1) {
            perror("stat");
            return 1;
        }
        fileInfoArray[i - 1].filename = argv[i];
        fileInfoArray[i - 1].size = fileStat.st_size;
    }
    // Sort the FileInfo array in ascending order of file sizes
    qsort(fileInfoArray, argc - 1, sizeof(struct FileInfo), compareFileInfo);

    // Display the filenames in ascending order of sizes
    printf("Filenames in ascending order according to their sizes:\n");
    for (int i = 0; i < argc - 1; i++) {
        printf("%s - %ld bytes\n", fileInfoArray[i].filename, fileInfoArray[i].size);
    }
    // Clean up and free memory
    free(fileInfoArray);
    return 0;
}
```

10] Display all the files from current directory whose size is greater that n Bytes Where n is accept from user.

Solution :- Here's a C program that uses the opendir, readdir, and stat functions to list files in the current directory and its subdirectories based on their size. It accepts the minimum file size (in bytes) from the user and displays the file names and their sizes.
Compile this program using a C compiler, and then run the executable. It will prompt you to enter the minimum file size (in bytes) and list all files in the current directory that are larger than the specified size.

```
#include <stdio.h>
#include <dirent.h>
#include <sys/stat.h>
#include <string.h>

int main() {
    DIR *dir;
    struct dirent *entry;
    struct stat fileStat;
    char path[1000];
    unsigned long long n;

    // Prompt the user for the minimum file size
    printf("Enter the minimum file size (in bytes): ");
    scanf("%llu", &n);

    // Open the current directory
    dir = opendir(".");

    if (dir == NULL) {
        perror("opendir");
        return 1;
    }
```

```
    // Iterate through the directory entries
    while ((entry = readdir(dir)) != NULL) {
        snprintf(path, sizeof(path), "%s", entry->d_name);
        if (stat(path, &fileStat) < 0) {
            perror("stat");
            continue;
        }

        if (S_ISREG(fileStat.st_mode) && fileStat.st_size > n) {
            printf("%s (%llu bytes)\n", path, fileStat.st_size);
        }
    }

    closedir(dir);
    return 0;
}
```

11] Write a C Program that demonstrates redirection of standard output to a file.

Solution:- You can redirect the standard output of a C program to a file in Unix using the freopen function. Here's a simple C program that demonstrates redirection of standard output to a file:
Here's what this program does:

- It uses freopen to redirect standard output (stdout) to a file named "output.txt" in write mode ("w").
- Any subsequent output generated using printf will be written to the "output.txt" file instead of the terminal.
- After the program is finished, it closes the redirected file stream using fclose(stdout).
- To compile and run the program, follow these steps:

- Save the C code to a file, e.g., "redirect_output.c".

Compile the program using a C compiler (e.g., gcc):

```
gcc -o redirect_output redirect_output.c
```

Run the compiled program:
./redirect_output

- After running the program, you'll find the output in the "output.txt" file, and it will no longer be displayed in the terminal.

```
#include <stdio.h>

int main() {
   // Redirect standard output to a file
   freopen("output.txt", "w", stdout);

   // Print to standard output (now redirected to "output.txt")
   printf("This is redirected standard output.\n");

   // Close the redirected file stream
   fclose(stdout);

   return 0;
}
```

12] Write a C program that will only list all subdirectories in alphabetical order from current directory.

Solution:- You can use the opendir and readdir functions from the dirent.h header to list subdirectories in alphabetical order in C. Here's a C program that demonstrates how to do this:

Here's what the program does:

1. It opens the current directory using opendir(".").
2. It reads all directory entries using readdir, filtering out the current directory (".") and parent directory ("..").
3. It stores the names of the subdirectories in an array of strings, subdirs.
4. It uses the qsort function to sort the subdirectory names in alphabetical order.
5. Finally, it prints the sorted list of subdirectories.

Compile and run the program, and it will list the subdirectories in the current directory in alphabetical order.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <dirent.h>

int compare(const void *a, const void *b) {
   return strcmp(*(const char **)a, *(const char **)b);
}
int main() {
   DIR *directory = opendir(".");
   if (!directory) {
      perror("opendir");
      return 1;
   }
   char **subdirs = NULL;
   size_t subdir_count = 0;
   struct dirent *entry;
   while ((entry = readdir(directory)) != NULL) {
      if (entry->d_type == DT_DIR && strcmp(entry->d_name, ".") != 0 &&
strcmp(entry->d_name, "..") != 0) {
         subdirs = (char **)realloc(subdirs, (subdir_count + 1) * sizeof(char *));
         subdirs[subdir_count] = strdup(entry->d_name);
         subdir_count++;
      }
   }

   closedir(directory);
   qsort(subdirs, subdir_count, sizeof(char *), compare);
   printf("Subdirectories in alphabetical order:\n");
   for (size_t i = 0; i < subdir_count; i++) {
      printf("%s\n", subdirs[i]);
```

```
        free(subdirs[i]);
    }
    free(subdirs);
    return 0;
}
```

16] Handle the two-way communication between parent and child processes using pipe.

Solution:- Here's a C program that demonstrates two-way communication between a parent process and a child process using pipes in Unix. In this example, the parent sends a message to the child, and the child modifies the message and sends it back to the parent.

1. In this program:
2. Two pipes, pipe1 and pipe2, are created. pipe1 is used for sending messages from the parent to the child, and pipe2 is used for sending messages from the child to the parent.
3. The parent process sends a message to the child through pipe1, and the child modifies the message and sends it back to the parent through pipe2.
4. Both processes close the ends of the pipes that they are not using to ensure proper communication.
5. The program demonstrates how the parent and child communicate by sending and receiving messages.
   Compile and run the program in a Unix environment to observe the two-way communication between the parent and child processes.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#define BUFFER_SIZE 1024

int main() {
```

```
int pipe1[2], pipe2[2];
char message[BUFFER_SIZE];
pid_t child_pid;

if (pipe(pipe1) == -1 || pipe(pipe2) == -1) {
    perror("pipe");
    exit(EXIT_FAILURE);
}
child_pid = fork();

if (child_pid == -1) {
    perror("fork");
    exit(EXIT_FAILURE);
}
if (child_pid == 0) {
    // Child process
    close(pipe1[1]); // Close write end of pipe1
    close(pipe2[0]); // Close read end of pipe2

    // Read the message from the parent
    read(pipe1[0], message, sizeof(message));
    printf("Child received: %s\n", message);

    // Modify the message
    strcat(message, " (Modified by Child)");

    // Send the modified message back to the parent
    write(pipe2[1], message, strlen(message) + 1);
    close(pipe1[0]);
    close(pipe2[1]);
} else {
    // Parent process
    close(pipe1[0]); // Close read end of pipe1
    close(pipe2[1]); // Close write end of pipe2
```

```
        strcpy(message, "Hello from Parent!");

        // Send the message to the child
        write(pipe1[1], message, strlen(message) + 1);

        // Read the modified message from the child
        read(pipe2[0], message, sizeof(message));
        printf("Parent received: %s\n");

        close(pipe1[1]);
        close(pipe2[0]);
    }
    return 0;
}
```

17] Demonstrate the use of atexit() function.

Solution:- The atexit() function in C is used to register functions that will be automatically called when the program exits, either by returning from the main function or by calling exit(). Here's a simple C program that demonstrates the use of atexit() to register exit functions:

In this program:

1. We define two exit functions, exitFunction1 and exitFunction2, which will be called when the program exits.
2. We use the atexit() function to register these exit functions in the main() function.
3. The main() function prints a message, and then we simulate program exit using exit(0).
4. When the program exits, the registered exit functions (exitFunction1 and exitFunction2) are automatically called in the reverse order of registration.
5. Compile and run this program in a Unix environment, and you will see the output as follows:

//Output

Main function is executing.

Exit function 2 called.

Exit function 1 called.

```
#include <stdio.h>
#include <stdlib.h>

// Exit function 1
void exitFunction1() {
    printf("Exit function 1 called.\n");
}
// Exit function 2
void exitFunction2() {
    printf("Exit function 2 called.\n");
}
int main() {
    // Register exit functions using atexit
    if (atexit(exitFunction1) != 0) {
        fprintf(stderr, "Registration of exit function 1 failed.\n");
        return 1;
    }

    if (atexit(exitFunction2) != 0) {
        fprintf(stderr, "Registration of exit function 2 failed.\n");
        return 1;
    }
    printf("Main function is executing.\n");
    // Simulate program exit
    exit(0);
}
```

18] Write a C program to demonstrates the different behaviour that can be seen with automatic, global, register, static and volatile variables (Use setjmp() and longjmp() system call).

Solution:- In this C program, we'll demonstrate the different behaviors of automatic, global, register, static, and volatile variables using the setjmp() and longjmp() functions. We'll use these functions to create checkpoints in the program's execution and then observe how variable values change as we long jump to different checkpoints.

In this program:

1. We declare and initialize variables of different types: global, automatic, register, static, and volatile.
2. We use setjmp(env) to create two checkpoints. The first checkpoint initializes the variables and modifies their values, and the second checkpoint is used to return to the initial values.
3. We print the variable values at different stages and use longjmp(env, 1) to jump back to the previous checkpoint.
4. You'll observe how the variable values change when we jump between checkpoints using longjmp.

```
#include <stdio.h>
#include <setjmp.h>

jmp_buf env;

// Global variables
int global_var = 0;

int main() {
    int automatic_var = 0;
    register int register_var = 0;
    static int static_var = 0;
    volatile int volatile_var = 0;
```

```
// Set a checkpoint
if (setjmp(env) == 0) {
    // Initial values
    printf("Initial values:\n");
    printf("Global: %d\n", global_var);
    printf("Automatic: %d\n", automatic_var);
    printf("Register: %d\n", register_var);
    printf("Static: %d\n", static_var);
    printf("Volatile: %d\n", volatile_var);

    // Modify the variables
    global_var = 10;
    automatic_var = 20;
    register_var = 30;
    static_var = 40;
    volatile_var = 50;

    // Set another checkpoint
    if (setjmp(env) == 0) {
        printf("\nValues after modification:\n");
        printf("Global: %d\n", global_var);
        printf("Automatic: %d\n", automatic_var);
        printf("Register: %d\n", register_var);
        printf("Static: %d\n", static_var);
        printf("Volatile: %d\n", volatile_var);
    } else {
        printf("\nBack to the first checkpoint using longjmp.\n");
    }

    // Long jump to the first checkpoint
    longjmp(env, 1);
} else {
    printf("Back to the initial checkpoint using longjmp.\n");
}
```

```
    return 0;
}
```

22] Write a C program to get and set the resource limits such as files, memory associated with a process

Solution:- To get and set resource limits for a process in a Unix environment, you can use the getrlimit and setrlimit functions provided by the sys/resource.h header. Resource limits define the maximum amount of resources a process can consume, such as CPU time, file size, and stack size. Here's a C program that demonstrates how to get and set resource limits:

In this program:

1. We include the necessary headers and define the struct rlimit to store resource limit information.
2. We first use getrlimit to retrieve the current resource limits for RLIMIT_NOFILE (maximum open files) and print the soft and hard limits.
3. We then use setrlimit to set new resource limits for RLIMIT_NOFILE. In this example, we set the soft limit to 1024 and the hard limit to 2048 for the maximum number of open files.
4. Compile and run this program in a Unix environment to get and set resource limits for a process. Note that setting resource limits may require superuser privileges in some cases, depending on the resource and current limits.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/resource.h>

int main() {
   struct rlimit limit;
   int resource;

   // Get resource limits for the specified resource (RLIMIT_NOFILE for
maximum number of open files)
   resource = RLIMIT_NOFILE;
```

```
    if (getrlimit(resource, &limit) == 0) {
        printf("Current resource limit for RLIMIT_NOFILE (maximum open
files):\n");
        printf("Soft limit: %lu\n", limit.rlim_cur);
        printf("Hard limit: %lu\n", limit.rlim_max);
    } else {
        perror("getrlimit");
        return 1;
    }

    // Set new resource limits (soft limit and hard limit) for RLIMIT_NOFILE
    limit.rlim_cur = 1024;  // Soft limit: maximum number of open files
    limit.rlim_max = 2048;  // Hard limit

    if (setrlimit(resource, &limit) == 0) {
        printf("New resource limits set for RLIMIT_NOFILE:\n");
        printf("Soft limit: %lu\n", limit.rlim_cur);
        printf("Hard limit: %lu\n", limit.rlim_max);
    } else {
        perror("setrlimit");
        return 1;
    }

    return 0;
}
```

23] Write a program that illustrates how to execute two commands concurrently with a pipe.

You can execute two commands concurrently with a pipe in Unix using the fork, pipe, and exec functions. In this example, we'll demonstrate how to run two simple commands concurrently and pass data from one command to the other through a pipe. Here's a C program to illustrate this:

In this program:

1. We create a pipe using pipe(pipefd). This pipe has two ends: pipefd[0] for reading and pipefd[1] for writing.

2. We fork a child process using fork(). The child process will execute the second command, and the parent process will execute the first command.

3. In the child process, we close the write end of the pipe and redirect its standard input (stdin) to read from the pipe using dup2. Then, we use execlp to execute the second command, e.g., "wc -l" to count lines.

4. In the parent process, we close the read end of the pipe and redirect its standard output (stdout) to write to the pipe using dup2. Then, we use execlp to execute the first command, e.g., "ls" to list files.

5. We wait for the child process to complete using wait(NULL).

6. Compile and run this program in a Unix environment, and it will execute the two commands concurrently with the output of the first command passed to the second command through the pipe.

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

int main() {
    int pipefd[2];
    pid_t child_pid;

    // Create a pipe
    if (pipe(pipefd) == -1) {
        perror("pipe");
        exit(EXIT_FAILURE);
```

```
}

// Fork a child process
child_pid = fork();

if (child_pid == -1) {
    perror("fork");
    exit(EXIT_FAILURE);
}

if (child_pid == 0) {
    // Child process (command 2)
    close(pipefd[1]);  // Close the write end of the pipe

    // Redirect stdin to read from the pipe
    dup2(pipefd[0], 0);

    // Execute the second command (e.g., "wc -l" to count lines)
    execlp("wc", "wc", "-l", NULL);

    perror("execlp");
    exit(EXIT_FAILURE);
} else {
    // Parent process (command 1)
    close(pipefd[0]);  // Close the read end of the pipe

    // Redirect stdout to write to the pipe
    dup2(pipefd[1], 1);

    // Execute the first command (e.g., "ls" to list files)
    execlp("ls", "ls", NULL);

    perror("execlp");
    exit(EXIT_FAILURE);
}

// Close the remaining file descriptors
close(pipefd[0]);
close(pipefd[1]);
```

```
    // Wait for the child process to complete
    wait(NULL);

    return 0;
}
```

24] Write a C program that print the exit status of a terminated child process

You can print the exit status of a terminated child process in Unix by using the waitpid or wait system call. Here's a simple C program that demonstrates how to do this:
In this program:

1. We create a child process using fork().
2. In the child process, we simulate some work (e.g., with sleep(2)) and then terminate it with an exit status using exit(42).
3. In the parent process, we wait for the child process to exit using wait(&status). The status variable will store information about the child's exit status.
4. We use the WIFEXITED macro to check if the child process exited normally, and if so, we print the exit status using WEXITSTATUS.
5. Compile and run the program in a Unix environment. It will create a child process, wait for it to exit, and print the exit status of the child process.

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

int main() {
    pid_t child_pid;
    int status;
```

```
    child_pid = fork();

    if (child_pid == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    }

    if (child_pid == 0) {
        // Child process
        printf("Child process (PID %d) is running.\n", getpid());
        // Simulate some work in the child process
        sleep(2);
        // Terminate the child process with an exit status
        exit(42);
    } else {
        // Parent process
        printf("Parent process (PID %d) is waiting for the child.\n", getpid());
        wait(&status);

        if (WIFEXITED(status)) {
            int exit_status = WEXITSTATUS(status);
            printf("Child process (PID %d) exited with status %d.\n", child_pid,
exit_status);
        } else {
            printf("Child process (PID %d) did not exit normally.\n", child_pid);
        }
    }

    return 0;
}
```

28] Write a C program that illustrates suspending and resuming processes using signals.

You can use signals to suspend and resume processes in Unix. Here's a C program that illustrates how to send the SIGSTOP signal to suspend a process and the SIGCONT signal to resume it:

In this program:

1. We use the signal function to register a custom handler function, suspendHandler, for the SIGTSTP (Ctrl+Z) signal. This handler will be called when the process is suspended.
2. In the main function, we print a message indicating that the process is running and waiting to be suspended with Ctrl+Z.
3. The program enters an infinite loop, which simulates the process continuing to run.
4. When you press Ctrl+Z in the terminal, the suspendHandler function is called, and the message "Process is suspended. Press Enter to resume..." is displayed.
5. To resume the process, you can press Enter, which sends the SIGCONT signal to the process, causing it to resume execution.
6. Compile and run this program in a Unix environment. It will demonstrate how to use signals to suspend and resume a process.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>

void suspendHandler(int signum) {
   printf("Process is suspended. Press Enter to resume...\n");
   fflush(stdout);
}

int main() {
   signal(SIGTSTP, suspendHandler); // Register a handler for SIGTSTP (Ctrl+Z)
to suspend the process
   printf("Process is running. Press Ctrl+Z to suspend...\n");
   fflush(stdout);
   while (1) {
      // Process continues to run until it's suspended
   }
   return 0;
}
```

29] Write a C program which create a child process which catch a signal sighup, sigint and sigquit. The Parent process send a sighup or sigint signal after every 3 seconds, at the end of 30 second parent send sigquit signal to child and child terminates my displaying message “My DADDY has Killed me!!!”.

You can create a C program that spawns a child process, and the child catches SIGHUP, SIGINT, and SIGQUIT signals while the parent sends SIGHUP or SIGINT signals to the child every 3 seconds. After 30 seconds, the parent sends a SIGQUIT signal to terminate the child process. Here's a program that accomplishes this task:

In this program:

1. The parent process forks a child process.
2. The child process sets up signal handlers for SIGHUP, SIGINT, and SIGQUIT.
3. The parent process sends SIGHUP and SIGINT signals to the child every 3 seconds.
4. After 30 seconds, the parent process sends a SIGQUIT signal to the child to terminate it.
5. Compile and run this program in a Unix environment, and you will see the child process handling signals and eventually being terminated by the parent.

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

void child_signal_handler(int signo) {
   if (signo == SIGHUP) {
      printf("Child received SIGHUP\n");
   } else if (signo == SIGINT) {
```

```
        printf("Child received SIGINT\n");
    } else if (signo == SIGQUIT) {
        printf("Child received SIGQUIT\n");
        printf("My DADDY has Killed me!!!\n");
        exit(EXIT_SUCCESS);
    }
}

int main() {
    pid_t child_pid;

    // Fork a child process
    child_pid = fork();

    if (child_pid == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    }

    if (child_pid == 0) {
        // Child process
        signal(SIGHUP, child_signal_handler);
        signal(SIGINT, child_signal_handler);
        signal(SIGQUIT, child_signal_handler);

        while (1) {
            // Child process continues to run
            sleep(1);
        }
    } else {
        // Parent process
        printf("Parent process (PID %d) spawned child (PID %d).\n", getpid(),
child_pid);
        sleep(3);  // Sleep for 3 seconds
```

```
        // Send SIGHUP to child
        kill(child_pid, SIGHUP);
        sleep(3);  // Sleep for 3 seconds

        // Send SIGINT to child
        kill(child_pid, SIGINT);
        sleep(24);  // Sleep for 24 seconds

        // Send SIGQUIT to child to terminate it
        kill(child_pid, SIGQUIT);

        // Wait for the child process to terminate
        wait(NULL);
    }

    return 0;
}
```