AOS PRACTICAL

** SLIP 1_Q1 : Take multiple files as Command Line Arguments and print their inode numbers and file types

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

void print_file_info(const char *file_path) {
   struct stat file_stat;

   if (stat(file_path, &file_stat) == -1) {
      if (errno == ENOENT) {
         perror("Error");
      } else if (errno == EACCES) {
         printf("Error: Permission denied for %s\n", file_path);
      } else {
         perror("Error");
      }
      return;
   }

   printf("File: %s\n", file_path);
   printf("Inode: %ld\n", (long)file_stat.st_ino);

   if (S_ISDIR(file_stat.st_mode)) {
      printf("Type: Directory\n");
   } else if (S_ISREG(file_stat.st_mode)) {
      printf("Type: Regular file\n");
   } else if (S_ISLNK(file_stat.st_mode)) {
      printf("Type: Symbolic link\n");
   } else {
      printf("Type: Other\n");
   }

   printf("\n");
}

int main(int argc, char *argv[]) {
   if (argc < 2) {
      fprintf(stderr, "Usage: %s <file1> <file2> ...\n", argv[0]);
      return 1;
   }

   for (int i = 1; i < argc; i++) {
      print_file_info(argv[i]);
   }

   return 0;
}
```

** SLIP 1_Q2 : Write a C program to send SIGALRM signal by child process to parent process and parent process make a provision to catch the signal and display alarm is fired.(Use Kill, fork, signal and sleep system call)

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

void handle_alarm(int sig) {
    printf("Alarm fired! Parent process caught the signal.\n");
}

int main() {
    pid_t pid;

    if (signal(SIGALRM, handle_alarm) == SIG_ERR) {
        perror("Error setting up signal handler");
        exit(1);
    }

    pid = fork();

    if (pid < 0) {
        perror("Fork failed");
        exit(1);
    }

    if (pid == 0) {
        sleep(2);
        printf("Child sending SIGALRM to parent...\n");
        kill(getppid(), SIGALRM);
        exit(0);
    } else {
        printf("Parent waiting for signal...\n");
        sleep(5);
        printf("Parent exiting.\n");
    }

    return 0;
}
```

** SLIP 2_Q1 : Write a C program to find file properties such as inode number, number of hard link, Filepermissions, File size, File access and modification time and so on of a given file using stat() system call.

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

```
#include <time.h>
#include <unistd.h>

void print_file_permissions(mode_t mode) {
    printf("Permissions: ");
    printf( (S_ISDIR(mode)) ? "d" : "-");
    printf( (mode & S_IRUSR) ? "r" : "-");
    printf( (mode & S_IWUSR) ? "w" : "-");
    printf( (mode & S_IXUSR) ? "x" : "-");
    printf( (mode & S_IRGRP) ? "r" : "-");
    printf( (mode & S_IWGRP) ? "w" : "-");
    printf( (mode & S_IXGRP) ? "x" : "-");
    printf( (mode & S_IROTH) ? "r" : "-");
    printf( (mode & S_IWOTH) ? "w" : "-");
    printf( (mode & S_IXOTH) ? "x" : "-");
    printf("\n");
}

void print_file_info(const char *filename) {
    struct stat fileStat;

    // Get file information
    if (stat(filename, &fileStat) < 0) {
        perror("stat");
        exit(1);
    }

    // Print file properties
    printf("File: %s\n", filename);
    printf("Inode Number: %ld\n", (long)fileStat.st_ino);
    printf("Number of Hard Links: %ld\n", (long)fileStat.st_nlink);
    printf("File Size: %ld bytes\n", (long)fileStat.st_size);

    // Print file permissions
    print_file_permissions(fileStat.st_mode);

    // Print access, modification, and status change times
    printf("Last Access Time: %s", ctime(&fileStat.st_atime));
    printf("Last Modification Time: %s", ctime(&fileStat.st_mtime));
    printf("Last Status Change Time: %s", ctime(&fileStat.st_ctime));
}

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

    // Print file information
    print_file_info(argv[1]);
    return 0;
}
```

** SLIP 2_Q2 : Write a C program that catches the ctrl-c (SIGINT) signal for the first time and display the appropriate message and exits on pressing ctrl-c again.

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

int sigint_count = 0;

void handle_sigint(int sig) {
    sigint_count++;

    if (sigint_count == 1) {
        printf("Caught SIGINT (Ctrl-C) for the first time. Press Ctrl-C again to exit.\n");
    } else {
        printf("Exiting the program after second Ctrl-C.\n");
        exit(0);
    }
}

int main() {
    if (signal(SIGINT, handle_sigint) == SIG_ERR) {
        perror("Error setting up signal handler");
        exit(1);
    }

    printf("Press Ctrl-C to catch SIGINT...\n");

    while (1) {
        sleep(1); // Keeps the program running and waiting for signals
    }

    return 0;
}
```

** SLIP 3_Q1 : Print the type of file and inode number where file name accepted through Command Line

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

void print_file_info(const char *file_path) {
    struct stat file_stat;

    if (stat(file_path, &file_stat) == -1) {
        perror("Error retrieving file information");
        return;
```

```
   }

   printf("File: %s\n", file_path);
   printf("Inode number: %ld\n", (long)file_stat.st_ino);

   if (S_ISREG(file_stat.st_mode)) {
      printf("Type: Regular file\n");
   } else if (S_ISDIR(file_stat.st_mode)) {
      printf("Type: Directory\n");
   } else if (S_ISLNK(file_stat.st_mode)) {
      printf("Type: Symbolic link\n");
   } else if (S_ISCHR(file_stat.st_mode)) {
      printf("Type: Character device\n");
   } else if (S_ISBLK(file_stat.st_mode)) {
      printf("Type: Block device\n");
   } else if (S_ISFIFO(file_stat.st_mode)) {
      printf("Type: FIFO/pipe\n");
   } else if (S_ISSOCK(file_stat.st_mode)) {
      printf("Type: Socket\n");
   } else {
      printf("Type: Unknown\n");
   }
}

int main(int argc, char *argv[]) {
   if (argc < 2) {
      fprintf(stderr, "Usage: %s <file_name>\n", argv[0]);
      return 1;
   }

   print_file_info(argv[1]);

   return 0;
}
```

** SLIP 3_Q2 : Write a C program which creates a child process to run linux/ unix command or any user defined program. The parent process set the signal handler for death of child signal and Alarm signal. If a child process does not complete its execution in 5 second then parent process kills child process.

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

pid_t child_pid;

void handle_child_termination(int sig) {
   int status;
   waitpid(child_pid, &status, 0);  // Wait for the child process to terminate
```

```
    printf("Child process terminated with status %d.\n", WEXITSTATUS(status));
}

void handle_alarm(int sig) {
    printf("Timeout! Killing child process...\n");
    kill(child_pid, SIGKILL);  // Kill the child process if it exceeds the timeout
    waitpid(child_pid, NULL, 0); // Wait for child to be cleaned up
    exit(1);  // Exit the parent process after killing the child
}

int main(int argc, char *argv[]) {
    if (argc < 2) {
        fprintf(stderr, "Usage: %s <command> [args...]\n", argv[0]);
        exit(1);
    }

    // Set up the signal handler for child termination
    signal(SIGCHLD, handle_child_termination);
    // Set up the signal handler for alarm
    signal(SIGALRM, handle_alarm);

    // Create a child process
    child_pid = fork();
    if (child_pid < 0) {
        perror("Fork failed");
        exit(1);
    }

    if (child_pid == 0) {
        // Child process
        printf("Child process (PID: %d) executing command: ", getpid());
        for (int i = 1; i < argc; i++) {
            printf("%s ", argv[i]);
        }
        printf("\n");

        // Execute the command
        execvp(argv[1], &argv[1]);

        // If execvp fails
        perror("execvp failed");
        exit(1);
    } else {
        // Parent process
        printf("Parent process waiting for child to finish...\n");

        // Set an alarm to kill the child after 5 seconds
        alarm(5);

        // Wait for child process to finish
        pause(); // Parent waits for signals
    }

    return 0;
}
```

** SLIP 4_Q1 : Write a C program to find whether a given files passed through command line arguments are present in current directory or not.

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

int file_exists(const char *file_name) {
    DIR *dir;
    struct dirent *entry;

    // Open the current directory
    dir = opendir(".");
    if (dir == NULL) {
        perror("Error opening directory");
        return 0;
    }

    // Loop through all the entries in the current directory
    while ((entry = readdir(dir)) != NULL) {
        // Compare the file name with the entry name
        if (strcmp(entry->d_name, file_name) == 0) {
            closedir(dir);
            return 1;  // File found
        }
    }

    closedir(dir);
    return 0;  // File not found
}

int main(int argc, char *argv[]) {
    if (argc < 2) {
        fprintf(stderr, "Usage: %s <file1> <file2> ... <fileN>\n", argv[0]);
        return 1;
    }

    for (int i = 1; i < argc; i++) {
        if (file_exists(argv[i])) {
            printf("File '%s' exists in the current directory.\n", argv[i]);
        } else {
            printf("File '%s' does not exist in the current directory.\n", argv[i]);
        }
    }
    return 0;
}
```

** SLIP 4_Q2 : Write a C program which creates a child process and child process catches a signal SIGHUP, SIGINT and SIGQUIT. The Parent process send a SIGHUP or SIGINT signal after every 3 seconds, at the end of 15 second parent send SIGQUIT signal to child and child terminates by displaying message "My Papa has Killed me!!!”.

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

void handle_sighup(int sig) {
    printf("Child: Caught SIGHUP signal\n");
}

void handle_sigint(int sig) {
    printf("Child: Caught SIGINT signal\n");
}

void handle_sigquit(int sig) {
    printf("Child: My Papa has Killed me!!!\n");
    exit(0);  // Child terminates upon receiving SIGQUIT
}

int main() {
    pid_t pid;

    // Create a child process
    pid = fork();

    if (pid < 0) {
        perror("Fork failed");
        exit(1);
    }

    if (pid == 0) {
        // Child process - Set signal handlers
        signal(SIGHUP, handle_sighup);
        signal(SIGINT, handle_sigint);
        signal(SIGQUIT, handle_sigquit);

        printf("Child process (PID: %d) is running and waiting for signals...\n", getpid());

        // Keep child running indefinitely to catch signals
        while (1) {
            pause();  // Wait for signals
        }
    } else {
        // Parent process
        printf("Parent process (PID: %d) is sending signals to the child process...\n", getpid());

        // Send SIGHUP and SIGINT signals alternately every 3 seconds
        for (int i = 0; i < 5; i++) {
            if (i % 2 == 0) {
                kill(pid, SIGHUP);  // Send SIGHUP to child
                printf("Parent: Sent SIGHUP to child\n");
            } else {
                kill(pid, SIGINT);  // Send SIGINT to child
                printf("Parent: Sent SIGINT to child\n");
            }
            sleep(3);  // Wait for 3 seconds
        }

        // After 15 seconds, send SIGQUIT to child to terminate it
        kill(pid, SIGQUIT);
        printf("Parent: Sent SIGQUIT to child, terminating child...\n");

        // Wait for child to terminate
        wait(NULL);
        printf("Parent process exiting...\n");
    }

    return 0;
}
```

** SLIP 5_Q1 : Read the current directory and display the name of the files, no of files in current directory

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

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

    // Open the current directory
    dir = opendir(".");
    if (dir == NULL) {
        perror("Unable to open current directory");
        return 1;
    }

    // Loop through all entries in the directory
    printf("Files in current directory:\n");
    while ((entry = readdir(dir)) != NULL) {
        // Display file name
        printf("%s\n", entry->d_name);
        file_count++;
    }

    // Close the directory
    closedir(dir);

    // Display the number of files
    printf("\nTotal number of files in the current directory: %d\n", file_count);

    return 0;
}
```

** SLIP 5_Q2 : Write a C program to create an unnamed pipe. The child process will write following three messages to pipe and parent process display it.
Message1 = “Hello World”
Message2 = “Hello SPPU”
Message3 = “Linux is Funny”

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

#define MAX_MSG_SIZE 100

int main() {
    int pipefd[2]; // File descriptors for pipe
    pid_t pid;
    char message1[] = "Hello World";
    char message2[] = "Hello SPPU";
    char message3[] = "Linux is Funny";
    char buffer[MAX_MSG_SIZE];

    // Create the pipe
    if (pipe(pipefd) == -1) {
        perror("Pipe creation failed");
        exit(1);
    }

    // Create the child process
    pid = fork();
    if (pid < 0) {
        perror("Fork failed");
        exit(1);
    }

    if (pid == 0) {
        // Child process: Write messages to the pipe
        close(pipefd[0]); // Close read end of the pipe
        write(pipefd[1], message1, strlen(message1) + 1); // Write message 1
        write(pipefd[1], message2, strlen(message2) + 1); // Write message 2
        write(pipefd[1], message3, strlen(message3) + 1); // Write message 3
        close(pipefd[1]); // Close write end of the pipe
        exit(0);
    } else {
        // Parent process: Read messages from the pipe
        close(pipefd[1]); // Close write end of the pipe
        printf("Parent: Reading messages from the pipe:\n");

        // Read and display each message from the pipe
        while (read(pipefd[0], buffer, MAX_MSG_SIZE) > 0) {
            printf("%s\n", buffer);
        }
        close(pipefd[0]); // Close read end of the pipe
        wait(NULL); // Wait for the child process to finish
    }

    return 0;
}
```

** SLIP 6_Q1 : Display all the files from current directory which are created in particular month

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

void list_files_created_in_month(const char *month_name) {
    DIR *dir;
    struct dirent *entry;
    struct stat file_stat;
    char time_str[256];
    struct tm *file_time;

    // Open the current directory
    dir = opendir(".");
    if (dir == NULL) {
        perror("Unable to open current directory");
        return;
    }

    // Loop through all entries in the directory
    while ((entry = readdir(dir)) != NULL) {
        // Get file stats
        if (stat(entry->d_name, &file_stat) == -1) {
            perror("Error getting file stats");
            continue;
        }

        // Convert the file's creation time to a struct tm
        file_time = localtime(&file_stat.st_ctime);  // st_ctime is the creation time
        // Format the time into a string (Month)
        strftime(time_str, sizeof(time_str), "%B", file_time);  // %B gives the full month name

        // Compare if the file was created in the given month
        if (strcmp(time_str, month_name) == 0) {
            printf("File: %s\n", entry->d_name);
        }
    }

    closedir(dir);
}

int main() {
    char month_name[20];

    // Input the month name (e.g., "January", "February", etc.)
    printf("Enter the month name (e.g., January, February, etc.): ");
    scanf("%s", month_name);

    // Convert the input to title case (first letter uppercase, rest lowercase)
    month_name[0] = toupper(month_name[0]);
    for (int i = 1; month_name[i] != '\0'; i++) {
        month_name[i] = tolower(month_name[i]);
    }

    // List files created in the specified month
    list_files_created_in_month(month_name);

    return 0;
}
```

** SLIP 6_Q2 : Write a C program to create n child processes. When all n child processes terminates, Display total cumulative time children spent in user and kernel mode

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

void create_children_and_measure_time(int n) {
    pid_t pid;
    struct tms start_time, end_time;
    clock_t start_clk, end_clk;

    // Get the start time for parent process
    start_clk = times(&start_time);
    for (int i = 0; i < n; i++) {
        pid = fork();
        if (pid < 0) {
            perror("Fork failed");
            exit(1);
        }

        if (pid == 0) {
            // Child process - Sleep for some time to simulate work
            sleep(2);  // Each child sleeps for 2 seconds to simulate work
            exit(0);
        }
    }

    // Parent process waits for all child processes to terminate
    for (int i = 0; i < n; i++) {
        wait(NULL);
    }

    // Get the end time for parent process after all children have terminated
    end_clk = times(&end_time);

    // Calculate the cumulative times for all children
    long total_user_time = end_time.tms_cutime - start_time.tms_cutime;
    long total_kernel_time = end_time.tms_cstime - start_time.tms_cstime;

    // Print the results
    printf("Total user time spent by children: %ld clock ticks\n", total_user_time);
    printf("Total kernel time spent by children: %ld clock ticks\n", total_kernel_time);
}

int main() {
    int n;

    // Get the number of child processes to create
    printf("Enter the number of child processes: ");
    scanf("%d", &n);

    create_children_and_measure_time(n);

    return 0;
}
```

** SLIP 7_Q1 : Write a C Program that demonstrates redirection of standard output to a file

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

int main() {
    // Open the file "output.txt" for writing (create or overwrite)
    FILE *file = freopen("output.txt", "w", stdout);
    // Check if the file opening failed
    if (file == NULL) {
        perror("Error opening file");
        return 1;
    }

    // Print messages to standard output, which is now redirected to the file
    printf("This will be written to the file instead of the console.\n");
    printf("Standard output has been redirected to 'output.txt'.\n");

    // Close the file and reset stdout back to the console
    fclose(file);

    // Restore the standard output to the terminal
    freopen("/dev/tty", "w", stdout);  // On Linux/Unix systems, use "/dev/tty" to restore stdout.

    // Print a message to the console after restoring stdout
    printf("This will be written to the console again.\n");

    return 0;
}
```

** SLIP 7_Q2 : Implement the following unix/linux command (use fork, pipe and exec system call)
ls –l | wc –l

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

int main() {
    int pipefd[2];
    pid_t pid1, pid2;

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

    // Fork the first child process to run ls -l
    pid1 = fork();
    if (pid1 == -1) {
        perror("fork");
        exit(1);
    }

    if (pid1 == 0) {
        // In child process 1 (ls -l)
        // Close the unused read end of the pipe
        close(pipefd[0]);
        // Redirect standard output to the pipe's write end
        dup2(pipefd[1], STDOUT_FILENO);
        close(pipefd[1]);

        // Execute the "ls -l" command
        execlp("ls", "ls", "-l", NULL);

        // If execlp() fails
        perror("execlp");
        exit(1);
    }

    // Fork the second child process to run wc -l
    pid2 = fork();
    if (pid2 == -1) {
        perror("fork");
        exit(1);
    }

    if (pid2 == 0) {
        // In child process 2 (wc -l)
        // Close the unused write end of the pipe
        close(pipefd[1]);

        // Redirect standard input to the pipe's read end
        dup2(pipefd[0], STDIN_FILENO);
        close(pipefd[0]);

        // Execute the "wc -l" command
        execlp("wc", "wc", "-l", NULL);

        // If execlp() fails
        perror("execlp");
        exit(1);
    }

    // Close both ends of the pipe in the parent process
    close(pipefd[0]);
    close(pipefd[1]);

    // Wait for both child processes to finish
    wait(NULL);
    wait(NULL);

    return 0;
}
```

**** SLIP 8_Q1 : Write a C program that redirects standard output to a file output.txt. (use of dup and open system call).**

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

int main() {
   // Open the file output.txt for writing (create it if it doesn't exist, or overwrite it)
   int file_desc = open("output.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);

   // Check if the file was opened successfully
   if (file_desc == -1) {
      perror("Error opening file");
      return 1;
   }

   // Duplicate the file descriptor to STDOUT (file descriptor 1)
   if (dup2(file_desc, STDOUT_FILENO) == -1) {
      perror("Error redirecting stdout");
      return 1;
   }

   // Now, printf will write to output.txt instead of the console
   printf("This message will be written to the file 'output.txt'.\n");
   printf("All subsequent output will also go to the file.\n");

   // Close the file descriptor
   close(file_desc);

   // Optionally, write to stdout again (this would require another redirection)
   printf("This will not be printed to stdout unless we restore stdout.\n");

   return 0;
}
```

** SLIP 8_Q2 : Implement the following unix/linux command (use fork, pipe and exec system call)
ls –l | wc –l

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

int main() {
   int pipefd[2];
   pid_t pid1, pid2;

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

```
// Fork the first child process to run ls -l
pid1 = fork();
if (pid1 == -1) {
    perror("fork");
    exit(1);
}

if (pid1 == 0) {
    // In the first child process (ls -l)

    // Close the unused read end of the pipe
    close(pipefd[0]);

    // Redirect standard output to the write end of the pipe
    dup2(pipefd[1], STDOUT_FILENO);
    close(pipefd[1]);  // Close the original write end

    // Execute the "ls -l" command
    execlp("ls", "ls", "-l", NULL);

    // If execlp() fails
    perror("execlp");
    exit(1);
}

// Fork the second child process to run wc -l
pid2 = fork();
if (pid2 == -1) {
    perror("fork");
    exit(1);
}

if (pid2 == 0) {
    // In the second child process (wc -l)

    // Close the unused write end of the pipe
    close(pipefd[1]);

    // Redirect standard input to the read end of the pipe
    dup2(pipefd[0], STDIN_FILENO);
    close(pipefd[0]);  // Close the original read end

    // Execute the "wc -l" command
    execlp("wc", "wc", "-l", NULL);

    // If execlp() fails
    perror("execlp");
    exit(1);
}
```

```
   // Parent closes both ends of the pipe
   close(pipefd[0]);
   close(pipefd[1]);

   // Parent waits for both child processes to finish
   wait(NULL);
   wait(NULL);

   return 0;
}
```

** SLIP 9_Q1 : Generate parent process to write unnamed pipe and will read from it

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

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

   // Create the pipe
   if (pipe(pipefd) == -1) {
      perror("pipe");
      exit(1);
   }

   // Fork the child process
   pid = fork();
   if (pid == -1) {
      perror("fork");
      exit(1);
   }

   if (pid == 0) {
      // In the child process
      char buffer[100];

      // Close the write end of the pipe in the child process
      close(pipefd[1]);

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

      // Close the read end after reading
      close(pipefd[0]);

      // Exit child process
      exit(0);
   } else {
      // In the parent process
      char *message = "Hello from parent";

      // Close the read end of the pipe in the parent process
      close(pipefd[0]);

      // Write to the pipe
      write(pipefd[1], message, strlen(message) + 1);
      printf("Parent sent message: %s\n", message);

      // Close the write end after writing
      close(pipefd[1]);

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

   return 0;
}
```

** SLIP 9_Q2 : Write a C program to Identify the type (Directory, character device, Block device, Regular file, FIFO or pipe, symbolic link or socket) of given file using stat() system call.

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

void print_file_type(mode_t st_mode) {
    if (S_ISDIR(st_mode)) {
        printf("Directory\n");
    } else if (S_ISCHR(st_mode)) {
        printf("Character device\n");
    } else if (S_ISBLK(st_mode)) {
        printf("Block device\n");
    } else if (S_ISREG(st_mode)) {
        printf("Regular file\n");
    } else if (S_ISFIFO(st_mode)) {
        printf("FIFO or pipe\n");
    } else if (S_ISLNK(st_mode)) {
        printf("Symbolic link\n");
    } else if (S_ISSOCK(st_mode)) {
        printf("Socket\n");
    } else {
        printf("Unknown file type\n");
    }
}

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

    struct stat file_stat;

    // Get the status of the file
    if (stat(argv[1], &file_stat) == -1) {
        perror("stat");
        return 1;
    }

    printf("File type of %s: ", argv[1]);
    print_file_type(file_stat.st_mode);

    return 0;
}
```

** SLIP 10_Q1 : Write a program that illustrates how to execute two commands concurrently with a pipe.

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

int main() {
    int pipefd[2];
    pid_t pid1, pid2;

    // Create the pipe
    if (pipe(pipefd) == -1) {
        perror("pipe");
        exit(1);
    }

    // Fork the first child process to run "ls"
    pid1 = fork();
    if (pid1 == -1) {
        perror("fork");
        exit(1);
    }

    if (pid1 == 0) {
        // In the first child process (ls)

        // Close the read end of the pipe in the first child process
        close(pipefd[0]);

        // Redirect standard output to the write end of the pipe
        dup2(pipefd[1], STDOUT_FILENO);
        close(pipefd[1]);  // Close the original write end

        // Execute the "ls" command
        execlp("ls", "ls", NULL);

        // If execlp() fails
        perror("execlp");
        exit(1);
    }

    // Fork the second child process to run "grep"
    pid2 = fork();
    if (pid2 == -1) {
        perror("fork");
        exit(1);
    }

    if (pid2 == 0) {
        // In the second child process (grep)

        // Close the write end of the pipe in the second child process
        close(pipefd[1]);

        // Redirect standard input to the read end of the pipe
        dup2(pipefd[0], STDIN_FILENO);
        close(pipefd[0]);  // Close the original read end

        // Execute the "grep file" command
        execlp("grep", "grep", "file", NULL);

        // If execlp() fails
        perror("execlp");
        exit(1);
    }

    // Parent process closes both ends of the pipe
    close(pipefd[0]);
    close(pipefd[1]);

    // Parent waits for both child processes to finish
    wait(NULL);
    wait(NULL);

    return 0;
}
```

** SLIP 10_Q2 : Generate parent process to write unnamed pipe and will write into it. Also generate child process which will read from pipe

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

```
#include <unistd.h>

int main() {
    int pipefd[2];  // Array to hold the pipe file descriptors
    pid_t pid;

    // Create the pipe
    if (pipe(pipefd) == -1) {
        perror("pipe");
        exit(1);
    }

    // Fork a child process
    pid = fork();
    if (pid == -1) {
        perror("fork");
        exit(1);
    }

    if (pid == 0) {
        // Child process
        char buffer[100];

        // Close the write end of the pipe in the child process
        close(pipefd[1]);

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

        // Close the read end after reading
        close(pipefd[0]);

        // Exit child process
        exit(0);
    } else {
        // Parent process
        char *message = "Hello from parent";

        // Close the read end of the pipe in the parent process
        close(pipefd[0]);

        // Write to the pipe
        write(pipefd[1], message, strlen(message) + 1);
        printf("Parent sent message: %s\n", message);

        // Close the write end after writing
        close(pipefd[1]);

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

    return 0;
}
```

** SLIP 11_Q1 : Write a C program to get and set the resource limits such as files, memory associated with a process

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

void print_limits(const struct rlimit *limit, const char *resource_name) {
    printf("%s: Soft limit = %ld, Hard limit = %ld\n", resource_name, limit->rlim_cur, limit->rlim_max);
}

int main() {
    struct rlimit rl;

    // Get and print the current limit for the maximum number of open files (RLIMIT_NOFILE)
    if (getrlimit(RLIMIT_NOFILE, &rl) == -1) {
        perror("getrlimit for RLIMIT_NOFILE");
        exit(1);
    }
    print_limits(&rl, "Maximum open files");

    // Set a new soft limit for maximum open files (example: 1024)
    rl.rlim_cur = 1024; // Set the soft limit
    if (setrlimit(RLIMIT_NOFILE, &rl) == -1) {
        perror("setrlimit for RLIMIT_NOFILE");
        exit(1);
    }
    printf("Successfully set new soft limit for open files.\n");

    // Get and print the current memory limit (RLIMIT_AS)
    if (getrlimit(RLIMIT_AS, &rl) == -1) {
        perror("getrlimit for RLIMIT_AS");
        exit(1);
    }
    print_limits(&rl, "Maximum virtual memory (RLIMIT_AS)");

    // Set a new limit for virtual memory (example: 2 GB)
    rl.rlim_cur = 2L * 1024L * 1024L * 1024L; // Set soft limit to 2GB
    if (setrlimit(RLIMIT_AS, &rl) == -1) {
        perror("setrlimit for RLIMIT_AS");
        exit(1);
    }
    printf("Successfully set new soft limit for virtual memory.\n");

    return 0;
}
```

** SLIP 11_Q2 : Write a C program that redirects standard output to a file output.txt. (use of dup and open system call).

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

int main() {
   int fd;

   // Open the file output.txt for writing (create if not exists, truncate it if exists)
   fd = open("output.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
   if (fd == -1) {
      perror("Error opening file");
      exit(1);
   }

   // Redirect standard output (STDOUT) to the file
   if (dup2(fd, STDOUT_FILENO) == -1) {
      perror("Error redirecting stdout");
      close(fd);
      exit(1);
   }

   // Now, anything written to stdout will go into the output.txt file
   printf("This message will be written to output.txt\n");

   // Close the file descriptor
   close(fd);

   return 0;
}
```

** SLIP 12_Q1 : Write a C program that print the exit status of a terminated child process

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

int main() {
   pid_t pid;
   int status;

   pid = fork();  // Create a child process

   if (pid == -1) {
      // If fork() fails
```

```
        perror("fork");
        exit(1);
    } else if (pid == 0) {
        // Child process
        printf("Child process: My PID is %d\n", getpid());
        exit(42);  // Exit with a status code of 42
    } else {
        // Parent process
        // Wait for the child to terminate
        wait(&status);

        // Check if child terminated normally
        if (WIFEXITED(status)) {
            printf("Parent process: Child exited with status %d\n", WEXITSTATUS(status));
        } else {
            printf("Parent process: Child did not terminate normally\n");
        }
    }

    return 0;
}
```

** SLIP 12_Q2 : 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, …)

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

typedef struct {
    char *filename;
    off_t size;
} FileInfo;

// Comparison function for sorting files based on size
int compareFileSize(const void *a, const void *b) {
    FileInfo *fileA = (FileInfo *)a;
    FileInfo *fileB = (FileInfo *)b;
    return (fileA->size - fileB->size);
}

int main(int argc, char *argv[]) {
    if (argc < 2) {
        printf("Usage: %s <file1> <file2> ...\n", argv[0]);
        return 1;
    }

    FileInfo *files = (FileInfo *)malloc((argc - 1) * sizeof(FileInfo));
    if (files == NULL) {
        perror("Memory allocation failed");
        return 1;
    }

    // Get the file sizes using stat()
    for (int i = 1; i < argc; i++) {
        struct stat fileStat;
        if (stat(argv[i], &fileStat) == -1) {
            perror(argv[i]);
            continue;
        }
        files[i - 1].filename = argv[i];
        files[i - 1].size = fileStat.st_size;
    }

    // Sort the files based on their size
    qsort(files, argc - 1, sizeof(FileInfo), compareFileSize);

    // Display the sorted files
    printf("Files sorted by size (ascending):\n");
    for (int i = 0; i < argc - 1; i++) {
        printf("%s: %ld bytes\n", files[i].filename, files[i].size);
    }

    // Free the allocated memory
    free(files);

    return 0;
}
```

** SLIP 13_Q1 : Write a C program that illustrates suspending and resuming processes using signals

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

void child_process() {
    // Child process behavior
    printf("Child process started (PID: %d)\n", getpid());

    // Child process will wait for signals
    while (1) {
        pause();  // Wait for signals
    }
}

int main() {
    pid_t pid;

    pid = fork();  // Create a child process
    if (pid == -1) {
        // Error in forking
        perror("fork failed");
        exit(1);
    }

    if (pid == 0) {
        // Child process
        child_process();
    } else {
        // Parent process
        printf("Parent process (PID: %d), sending SIGSTOP to child (PID: %d)\n", getpid(), pid);

        // Suspend the child process using SIGSTOP
        kill(pid, SIGSTOP);

        // Parent waits for 3 seconds
        sleep(3);

        // Resume the child process using SIGCONT
        printf("Parent process (PID: %d), sending SIGCONT to child (PID: %d)\n", getpid(), pid);
        kill(pid, SIGCONT);

        // Parent waits for the child to terminate
        wait(NULL);
    }

    return 0;
}
```

** SLIP 13_Q2 : 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

```
==>  #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_string>\n", argv[0]);
        return 1;
    }

    const char *prefix = argv[1];
    DIR *dir = opendir(".");  // Open the current directory
    struct dirent *entry;

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

    printf("Files that begin with '%s':\n", prefix);

    // Read the directory contents
    while ((entry = readdir(dir)) != NULL) {
        // Check if the file name starts with the given prefix
        if (strncmp(entry->d_name, prefix, strlen(prefix)) == 0) {
            printf("%s\n", entry->d_name);
        }
    }

    closedir(dir);  // Close the directory

    return 0;
}
```

** SLIP 14_Q1 : Display all the files from current directory whose size is greater that n Bytes Where n is accept from user.

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

int main() {
    long size_threshold;

    // Accept size threshold from user
    printf("Enter the size threshold in bytes: ");
    if (scanf("%ld", &size_threshold) != 1) {
        printf("Invalid input. Please enter a valid number.\n");
        return 1;
    }

    DIR *dir = opendir(".");  // Open the current directory
    struct dirent *entry;

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

    printf("Files in the current directory with size greater than %ld bytes:\n", size_threshold);

    // Read the directory contents
    while ((entry = readdir(dir)) != NULL) {
        struct stat fileStat;

        // Get the file information using stat()
        if (stat(entry->d_name, &fileStat) == -1) {
            perror(entry->d_name);
            continue;
        }

        // Check if the file is regular and its size is greater than the threshold
        if (S_ISREG(fileStat.st_mode) && fileStat.st_size > size_threshold) {
            printf("%s - %ld bytes\n", entry->d_name, fileStat.st_size);
        }
    }

    closedir(dir);  // Close the directory

    return 0;
}
```

** SLIP 14_Q2 : 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.

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

void print_permissions(mode_t mode) {
    char permissions[10] = "---------";

    // Owner permissions
    if (mode & S_IRUSR) permissions[0] = 'r';
    if (mode & S_IWUSR) permissions[1] = 'w';
    if (mode & S_IXUSR) permissions[2] = 'x';

    // Group permissions
    if (mode & S_IRGRP) permissions[3] = 'r';
    if (mode & S_IWGRP) permissions[4] = 'w';
    if (mode & S_IXGRP) permissions[5] = 'x';

    // Other permissions
    if (mode & S_IROTH) permissions[6] = 'r';
    if (mode & S_IWOTH) permissions[7] = 'w';
    if (mode & S_IXOTH) permissions[8] = 'x';

    printf("Permissions: %s\n", permissions);
}

int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf("Usage: %s <filename>\n", argv[0]);
        return 1;
    }
    struct stat file_stat;
    const char *filename = argv[1];

    // Get file information using stat()
    if (stat(filename, &file_stat) == -1) {
        perror("stat");
        return 1;
    }

    // Inode number
    printf("Inode number: %ld\n", (long)file_stat.st_ino);

    // Number of hard links
    printf("Number of hard links: %ld\n", (long)file_stat.st_nlink);

    // File size
    printf("File size: %ld bytes\n", (long)file_stat.st_size);

    // File permissions
    print_permissions(file_stat.st_mode);

    // Last access time
    printf("Last access time: %s", ctime(&file_stat.st_atime));

    // Last modification time
    printf("Last modification time: %s", ctime(&file_stat.st_mtime));

    // Last status change time
    printf("Last status change time: %s", ctime(&file_stat.st_ctime));

    return 0;
}
```

** SLIP 15_Q1 : Display all the files from current directory whose size is greater that n Bytes Where n is accept from user

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

int main() {
    long size_threshold;

    // Accept size threshold from user
    printf("Enter the size threshold in bytes: ");
    if (scanf("%ld", &size_threshold) != 1) {
        printf("Invalid input. Please enter a valid number.\n");
        return 1;
    }
    DIR *dir = opendir(".");  // Open the current directory
    struct dirent *entry;

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

    printf("Files in the current directory with size greater than %ld bytes:\n", size_threshold);

    // Read the directory contents
    while ((entry = readdir(dir)) != NULL) {
        struct stat fileStat;

        // Get the file information using stat()
        if (stat(entry->d_name, &fileStat) == -1) {
            perror(entry->d_name);
            continue;
        }

        // Check if the file is regular and its size is greater than the threshold
        if (S_ISREG(fileStat.st_mode) && fileStat.st_size > size_threshold) {
            printf("%s - %ld bytes\n", entry->d_name, fileStat.st_size);
        }
    }

    closedir(dir);  // Close the directory

    return 0;
}
```

**** SLIP 15_Q2 : Write a C program which creates a child process to run linux/ unix command or any user defined program. The parent process set the signal handler for death of child signal and Alarm signal. If a child process does not complete its execution in 5 second then parent process kills child process**

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

pid_t child_pid;  // To store the child process ID

// Signal handler for child process termination (SIGCHLD)
void handle_child_death(int sig) {
    int status;
    // Wait for the child to terminate and get the exit status
    waitpid(child_pid, &status, 0);
    if (WIFEXITED(status)) {
        printf("Child process finished with exit status %d.\n", WEXITSTATUS(status));
    } else if (WIFSIGNALED(status)) {
        printf("Child process was terminated by signal %d.\n", WTERMSIG(status));
    }
    exit(0); // Exit the parent process after child finishes
}

// Signal handler for alarm (SIGALRM)
void handle_alarm(int sig) {
    printf("Child process did not finish within the time limit. Killing child...\n");
    kill(child_pid, SIGKILL);  // Kill the child if it didn't finish in time
    waitpid(child_pid, NULL, 0);  // Wait for the child to terminate after kill
    exit(1); // Exit the parent process
}

int main(int argc, char *argv[]) {
    if (argc < 2) {
        printf("Usage: %s <command> [args...]\n", argv[0]);
        return 1;
    }

    // Set up signal handlers
    signal(SIGCHLD, handle_child_death);  // Handle child death
    signal(SIGALRM, handle_alarm);        // Handle alarm signal

    // Create a child process using fork
    child_pid = fork();

    if (child_pid == -1) {
        // If fork fails
        perror("Fork failed");
        return 1;
    }

    if (child_pid == 0) {
        // In the child process
        printf("Child process started, executing command...\n");

        // Replace child process with the user-defined command (e.g., ls, or any other program)
        if (execvp(argv[1], &argv[1]) == -1) {
            // If execvp fails
            perror("Exec failed");
            exit(1);
        }
    } else {
        // In the parent process
        printf("Parent process: Waiting for child to complete or time out...\n");

        // Set an alarm after 5 seconds
        alarm(5);
        // Wait for the child to finish
        pause();  // Parent waits for signals (SIGCHLD or SIGALRM)
    }

    return 0;
}
```

**SLIP 16_Q1 : Display all the files from current directory which are created in particular month

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

int main() {
    char month_name[20];
    int year, month;

    // Ask the user for the month and year
    printf("Enter the month (e.g., January, February): ");
    scanf("%s", month_name);
    printf("Enter the year (e.g., 2023): ");
    scanf("%d", &year);

    // Convert the month name to a number (1 for January, 2 for February, etc.)
    if (strcmp(month_name, "January") == 0) {
        month = 1;
    } else if (strcmp(month_name, "February") == 0) {
        month = 2;
    } else if (strcmp(month_name, "March") == 0) {
        month = 3;
    } else if (strcmp(month_name, "April") == 0) {
        month = 4;
    } else if (strcmp(month_name, "May") == 0) {
        month = 5;
    } else if (strcmp(month_name, "June") == 0) {
        month = 6;
    } else if (strcmp(month_name, "July") == 0) {
        month = 7;
    } else if (strcmp(month_name, "August") == 0) {
        month = 8;
    } else if (strcmp(month_name, "September") == 0) {
        month = 9;
    } else if (strcmp(month_name, "October") == 0) {
        month = 10;
    } else if (strcmp(month_name, "November") == 0) {
        month = 11;
    } else if (strcmp(month_name, "December") == 0) {
        month = 12;
    } else {
        printf("Invalid month name!\n");
        return 1;
    }

    DIR *dir = opendir(".");  // Open the current directory
    if (dir == NULL) {
        perror("opendir");
        return 1;
    }

    struct dirent *entry;
    struct stat fileStat;
    struct tm *time_info;

    printf("Files created or modified in %s %d:\n", month_name, year);

    // Read the directory contents
    while ((entry = readdir(dir)) != NULL) {
        // Get the file information
        if (stat(entry->d_name, &fileStat) == -1) {
            perror(entry->d_name);
            continue;
        }

        // Extract the file's modification time
        time_info = localtime(&fileStat.st_mtime);

        // Check if the file's modification time matches the requested month and year
        if (time_info->tm_year + 1900 == year && time_info->tm_mon + 1 == month) {
            printf("%s\n", entry->d_name);  // Print the file name
        }
    }

    closedir(dir);  // Close the directory

    return 0;
}
```

** SLIP 16_Q2 : 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!!!”.

```
==>  #include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
#include <string.h>
#include <time.h>
```

```
// Signal handler for SIGHUP
void handle_sighup(int sig) {
    printf("Child: Received SIGHUP signal\n");
}

// Signal handler for SIGINT
void handle_sigint(int sig) {
    printf("Child: Received SIGINT signal\n");
}

// Signal handler for SIGQUIT
void handle_sigquit(int sig) {
    printf("My DADDY has Killed me!!!\n");
    exit(0);  // Terminate the child process
}

int main() {
    pid_t pid;

    // Fork to create a child process
    pid = fork();

    if (pid == -1) {
        // Error in fork
        perror("Fork failed");
        exit(1);
    }

    if (pid == 0) {
        // Child process: Set up signal handlers
        signal(SIGHUP, handle_sighup);
        signal(SIGINT, handle_sigint);
        signal(SIGQUIT, handle_sigquit);

        // Keep the child process running to catch signals
        while (1) {
            pause();  // Wait for signals
        }
    } else {
        // Parent process: Send signals to the child
        int counter = 0;

        // Send signals every 3 seconds for 30 seconds
        while (counter < 10) {
            if (counter % 2 == 0) {
                // Send SIGHUP to the child every even second (SIGHUP or SIGINT)
                kill(pid, SIGHUP);
                printf("Parent: Sent SIGHUP to child\n");
            } else {
                // Send SIGINT to the child every odd second (SIGHUP or SIGINT)
                kill(pid, SIGINT);
                printf("Parent: Sent SIGINT to child\n");
            }
            sleep(3);
            counter++;
        }

        // After 30 seconds, send SIGQUIT to terminate the child
        kill(pid, SIGQUIT);
        printf("Parent: Sent SIGQUIT to child\n");

        // Wait for the child to terminate
        wait(NULL);
        printf("Parent: Child has terminated\n");
    }

    return 0;
}
```

** SLIP 17_Q1 : Read the current directory and display the name of the files, no of files in current directory

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

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

    // Open the current directory
    dir = opendir(".");
    if (dir == NULL) {
        perror("opendir");
        return 1;
    }

    printf("Files in the current directory:\n");

    // Read the directory entries
    while ((entry = readdir(dir)) != NULL) {
        // Exclude the special . and .. directories
        if (entry->d_name[0] != '.') {
            printf("%s\n", entry->d_name);
            file_count++;
        }
    }

    // Print the total number of files
    printf("\nTotal number of files: %d\n", file_count);

    // Close the directory
    closedir(dir);

    return 0;
}
```

** SLIP 17_Q2 : Write a C program to implement the following unix/linux command (use fork, pipe and exec system call). Your program should block the signal Ctrl-C and Ctrl-\ signal during the execution. i. Ls –l | wc –l

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

void block_signals() {
    sigset_t sigset;
    sigemptyset(&sigset);  // Initialize an empty signal set
    sigaddset(&sigset, SIGINT);  // Add SIGINT (Ctrl-C) to the signal set
    sigaddset(&sigset, SIGQUIT);  // Add SIGQUIT (Ctrl-\) to the signal set
    sigprocmask(SIG_BLOCK, &sigset, NULL);  // Block SIGINT and SIGQUIT
}

int main() {
    int pipefd[2];  // File descriptors for the pipe
    pid_t pid1, pid2;

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

    // Block the SIGINT and SIGQUIT signals
    block_signals();

    // Fork the first child to execute "ls -l"
    pid1 = fork();
    if (pid1 == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    }

    if (pid1 == 0) {
        // Child process 1 (executes "ls -l")
        close(pipefd[0]);  // Close the read end of the pipe

        // Redirect the output to the pipe
        dup2(pipefd[1], STDOUT_FILENO);
        close(pipefd[1]);
        // Execute "ls -l"
        execlp("ls", "ls", "-l", NULL);
        perror("execlp");  // If exec fails
        exit(EXIT_FAILURE);
    }

    // Fork the second child to execute "wc -l"
    pid2 = fork();
    if (pid2 == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    }

    if (pid2 == 0) {
        // Child process 2 (executes "wc -l")
        close(pipefd[1]);  // Close the write end of the pipe

        // Redirect the input to come from the pipe
        dup2(pipefd[0], STDIN_FILENO);
        close(pipefd[0]);

        // Execute "wc -l"
        execlp("wc", "wc", "-l", NULL);
        perror("execlp");  // If exec fails
        exit(EXIT_FAILURE);
    }

    // Parent process: Close the pipe in the parent
    close(pipefd[0]);
    close(pipefd[1]);

    // Wait for both children to finish
    waitpid(pid1, NULL, 0);
    waitpid(pid2, NULL, 0);

    printf("Pipeline executed successfully.\n");
    return 0;
}
```

** SLIP 18_Q1 : Write a C program to find whether a given file is present in current directory or not

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

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

    char *filename = argv[1];
    DIR *dir;
    struct dirent *entry;
    int found = 0;

    // Open the current directory
    dir = opendir(".");
    if (dir == NULL) {
        perror("opendir");
        return 1;
    }

    // Read the directory entries
    while ((entry = readdir(dir)) != NULL) {
        // Compare the filename with the directory entry
        if (strcmp(entry->d_name, filename) == 0) {
            found = 1;
            break;
        }
    }

    if (found) {
        printf("File '%s' found in the current directory.\n", filename);
    } else {
        printf("File '%s' not found in the current directory.\n", filename);
    }

    // Close the directory
    closedir(dir);
    return 0;
}
```

**** SLIP 18_Q2 : Write a C program to create an unnamed pipe. The child process will write following three messages to pipe and parent process display it.**

Message1 = “Hello World”
Message2 = “Hello SPPU”
Message3 = “Linux is Funny”

==>

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

int main() {
    int pipefd[2];  // File descriptors for the pipe
    pid_t pid;
    char buffer[100];

    // Create an unnamed pipe
    if (pipe(pipefd) == -1) {
        perror("pipe");
        exit(EXIT_FAILURE);
    }

    // Fork a child process
    pid = fork();
    if (pid == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    }

    if (pid == 0) {
        // Child process
        close(pipefd[0]);  // Close the read end of the pipe as child will write

        // Messages to be sent to the pipe
        write(pipefd[1], "Hello World\n", 12);
        write(pipefd[1], "Hello SPPU\n", 11);
        write(pipefd[1], "Linux is Funny\n", 15);

        close(pipefd[1]);  // Close the write end after sending all messages
        exit(0);
    } else {
        // Parent process
        close(pipefd[1]);  // Close the write end of the pipe as parent will read

        // Read the messages from the pipe
        while (read(pipefd[0], buffer, sizeof(buffer)) > 0) {
            printf("%s", buffer);
        }

        close(pipefd[0]);  // Close the read end after reading all messages
        wait(NULL);  // Wait for the child to terminate
    }

    return 0;
}
```

** SLIP 19_Q1 : Take multiple files as Command Line Arguments and print their file type and inode number

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

void print_file_info(const char *filename) {
    struct stat file_stat;

    // Get file status using stat()
    if (stat(filename, &file_stat) == -1) {
        perror("stat");
        return;
    }

    // Print inode number
    printf("File: %s\n", filename);
    printf("Inode number: %ld\n", (long)file_stat.st_ino);

    // Determine and print the file type
    if (S_ISREG(file_stat.st_mode)) {
        printf("File type: Regular file\n");
    } else if (S_ISDIR(file_stat.st_mode)) {
        printf("File type: Directory\n");
    } else if (S_ISCHR(file_stat.st_mode)) {
        printf("File type: Character device\n");
    } else if (S_ISBLK(file_stat.st_mode)) {
        printf("File type: Block device\n");
    } else if (S_ISFIFO(file_stat.st_mode)) {
        printf("File type: FIFO (Named pipe)\n");
    } else if (S_ISLNK(file_stat.st_mode)) {
        printf("File type: Symbolic link\n");
    } else if (S_ISSOCK(file_stat.st_mode)) {
        printf("File type: Socket\n");
    } else {
        printf("File type: Unknown\n");
    }

    printf("\n");
}

int main(int argc, char *argv[]) {
    if (argc < 2) {
        fprintf(stderr, "Usage: %s <file1> <file2> ... <fileN>\n", argv[0]);
        return 1;
    }

    // Iterate over each command line argument (file name)
    for (int i = 1; i < argc; i++) {
        print_file_info(argv[i]);
    }

    return 0;
}
```

**** SLIP 19_Q2 : Implement the following unix/linux command (use fork, pipe and exec system call)**

ls –l | wc –l

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

```
int main() {
    int pipefd[2];
    pid_t pid1, pid2;

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

    // Fork the first child process to execute 'ls -l'
    pid1 = fork();
    if (pid1 == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    }

    if (pid1 == 0) {
        // In the first child process (executes 'ls -l')
        close(pipefd[0]);  // Close read end of the pipe
        dup2(pipefd[1], STDOUT_FILENO);  // Redirect stdout to the pipe

        // Execute 'ls -l'
        execlp("ls", "ls", "-l", (char *)NULL);
        perror("execlp");  // If execlp fails
        exit(EXIT_FAILURE);
    } else {
        // Fork the second child process to execute 'wc -l'
        pid2 = fork();
        if (pid2 == -1) {
            perror("fork");
            exit(EXIT_FAILURE);
        }

        if (pid2 == 0) {
            // In the second child process (executes 'wc -l')
            close(pipefd[1]);  // Close write end of the pipe
            dup2(pipefd[0], STDIN_FILENO);  // Redirect stdin from the pipe

            // Execute 'wc -l'
            execlp("wc", "wc", "-l", (char *)NULL);
            perror("execlp");  // If execlp fails
            exit(EXIT_FAILURE);
        } else {
            // In the parent process, close both ends of the pipe
            close(pipefd[0]);
            close(pipefd[1]);

            // Wait for both child processes to terminate
            waitpid(pid1, NULL, 0);
            waitpid(pid2, NULL, 0);
        }
    }

    return 0;
}
```

** SLIP 20_Q1 : Write a C program that illustrates suspending and resuming processes using signals

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

void suspend_and_resume(pid_t child_pid) {
    printf("Parent: Suspending the child process...\n");
    // Send SIGSTOP to suspend the child process
    kill(child_pid, SIGSTOP);

    // Give a little time to observe the suspension
    sleep(2);

    printf("Parent: Resuming the child process...\n");
    // Send SIGCONT to resume the child process
    kill(child_pid, SIGCONT);
}

int main() {
    pid_t pid = fork();

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

    if (pid == 0) {
        // In child process
        while (1) {
            printf("Child: I'm running...\n");
            sleep(1);  // Sleep to simulate some work
        }
    } else {
        // In parent process
        sleep(1);  // Give the child a chance to print something

        suspend_and_resume(pid);

        // Wait for child to terminate
        wait(NULL);
        printf("Parent: Child has terminated.\n");
    }
```

```
   return 0;
}
```

** SLIP 20_Q2 : Write a C program to Identify the type (Directory, character device, Block device, Regular file, FIFO or pipe, symbolic link or socket) of given file using stat() system call.

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

void print_file_type(const char *filename) {
   struct stat file_stat;

   // Get the file information using stat() system call
   if (stat(filename, &file_stat) == -1) {
      perror("stat");
      return;
   }

   // Print the file type based on the file mode
   if (S_ISREG(file_stat.st_mode)) {
      printf("File: %s\nType: Regular file\n", filename);
   } else if (S_ISDIR(file_stat.st_mode)) {
      printf("File: %s\nType: Directory\n", filename);
   } else if (S_ISCHR(file_stat.st_mode)) {
      printf("File: %s\nType: Character device\n", filename);
   } else if (S_ISBLK(file_stat.st_mode)) {
      printf("File: %s\nType: Block device\n", filename);
   } else if (S_ISFIFO(file_stat.st_mode)) {
      printf("File: %s\nType: FIFO (Named pipe)\n", filename);
   } else if (S_ISLNK(file_stat.st_mode)) {
      printf("File: %s\nType: Symbolic link\n", filename);
   } else if (S_ISSOCK(file_stat.st_mode)) {
      printf("File: %s\nType: Socket\n", filename);
   } else {
      printf("File: %s\nType: Unknown\n", filename);
   }
}

int main(int argc, char *argv[]) {
   // Check if at least one file name is passed as argument
   if (argc != 2) {
      fprintf(stderr, "Usage: %s <filename>\n", argv[0]);
      return 1;
   }

   // Call the function to print the file type
   print_file_type(argv[1]);
   return 0;
}
```

** SLIP 21_Q1 : Read the current directory and display the name of the files, no of files in current directory

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

int main() {
   struct dirent *entry;
   DIR *dp;
   int file_count = 0;

   // Open the current directory
   dp = opendir(".");
   if (dp == NULL) {
      perror("opendir");
      return 1;
   }

   printf("Files in the current directory:\n");

   // Read the directory contents
   while ((entry = readdir(dp)) != NULL) {
      // Skip the special entries "." and ".."
      if (strcmp(entry->d_name, ".") != 0 && strcmp(entry->d_name, "..") != 0) {
         printf("%s\n", entry->d_name);
         file_count++;
      }
   }

   // Close the directory
   closedir(dp);

   // Display the number of files
   printf("\nTotal number of files: %d\n", file_count);

   return 0;
}
```

** SLIP 21_Q2 : 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, …)

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

int compare_files(const void *a, const void *b) {
    // Cast the arguments to string pointers (file names)
    const char *file1 = *(const char **)a;
    const char *file2 = *(const char **)b;

    struct stat stat1, stat2;

    // Get the file status using stat()
    if (stat(file1, &stat1) == -1) {
        perror("stat");
        return 0;
    }

    if (stat(file2, &stat2) == -1) {
        perror("stat");
        return 0;
    }

    // Compare file sizes
    if (stat1.st_size < stat2.st_size)
        return -1;
    else if (stat1.st_size > stat2.st_size)
        return 1;
    else
        return 0;
}

int main(int argc, char *argv[]) {
    if (argc < 2) {
        printf("Usage: %s <file1> <file2> <file3> ...\n", argv[0]);
        return 1;
    }

    // Sort file names based on their sizes
    qsort(&argv[1], argc - 1, sizeof(char *), compare_files);

    printf("Files sorted by size (ascending):\n");

    for (int i = 1; i < argc; i++) {
        struct stat stat_buf;
        if (stat(argv[i], &stat_buf) == -1) {
            perror("stat");
            continue;
        }

        printf("%s - Size: %ld bytes\n", argv[i], stat_buf.st_size);
    }

    return 0;
}
```

** SLIP 22_Q1 : Write a C Program that demonstrates redirection of standard output to a file

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

int main() {
   // File pointer and file descriptor
   int fd;

   // Open a file (create if doesn't exist) and redirect standard output to it
   fd = open("output.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
   if (fd == -1) {
      perror("open");
      return 1;
   }

   // Redirect standard output (stdout) to the file
   if (dup2(fd, STDOUT_FILENO) == -1) {
      perror("dup2");
      close(fd);
      return 1;
   }

   // Now, any output written to stdout will go to output.txt
   printf("This message will be written to output.txt\n");
   printf("This is another line in the file.\n");

   // Close the file descriptor
   close(fd);

   return 0;
}
```

** SLIP 22_Q2 : Write a C program to implement the following unix/linux command (use fork, pipe and exec system call). Your program should block the signal Ctrl-C and Ctrl-\ signal during the execution. i. ls –l | wc –l

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

void block_signals() {
   // Block SIGINT (Ctrl-C) and SIGQUIT (Ctrl-\)
   sigset_t set;
   sigemptyset(&set);
   sigaddset(&set, SIGINT);
   sigaddset(&set, SIGQUIT);
```

```
    sigprocmask(SIG_BLOCK, &set, NULL);
}

int main() {
    int pipe_fd[2];
    pid_t pid1, pid2;

    // Create a pipe
    if (pipe(pipe_fd) == -1) {
        perror("pipe");
        exit(1);
    }

    // Block signals before creating processes
    block_signals();

    // Create the first child process to execute 'ls -l'
    pid1 = fork();
    if (pid1 == -1) {
        perror("fork");
        exit(1);
    }

    if (pid1 == 0) {
        // Child process 1 (ls -l)
        // Close unused pipe write end
        close(pipe_fd[0]);

        // Redirect stdout to the pipe
        dup2(pipe_fd[1], STDOUT_FILENO);
        close(pipe_fd[1]);

        // Execute 'ls -l'
        execlp("ls", "ls", "-l", (char *)NULL);
        perror("execlp ls");
        exit(1);
    }

    // Create the second child process to execute 'wc -l'
    pid2 = fork();
    if (pid2 == -1) {
        perror("fork");
        exit(1);
    }

    if (pid2 == 0) {
        // Child process 2 (wc -l)
        // Close unused pipe write end
        close(pipe_fd[1]);

        // Redirect stdin to the pipe
        dup2(pipe_fd[0], STDIN_FILENO);
        close(pipe_fd[0]);

        // Execute 'wc -l'
        execlp("wc", "wc", "-l", (char *)NULL);
        perror("execlp wc");
        exit(1);
    }

    // Parent process
    // Close both ends of the pipe
    close(pipe_fd[0]);
    close(pipe_fd[1]);

    // Wait for both children to finish
    waitpid(pid1, NULL, 0);
    waitpid(pid2, NULL, 0);

    // Allow signals to be handled again after execution
    sigset_t set;
    sigemptyset(&set);
    sigprocmask(SIG_UNBLOCK, &set, NULL);

    return 0;
}
```

** SLIP 23_Q1 : Write a C program to find whether a given file is present in current directory or not

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

int main(int argc, char *argv[]) {
    // Check if a file name is provided
    if (argc != 2) {
        printf("Usage: %s <filename>\n", argv[0]);
        return 1;
    }

    // Get the file name from command-line argument
    const char *filename = argv[1];

    // Check if the file exists and is accessible
    struct stat buffer;
    if (stat(filename, &buffer) == 0) {
        // The file exists
        printf("The file '%s' exists in the current directory.\n", filename);
    } else {
        // The file does not exist or there was an error
        perror("stat");
    }

    return 0;
}
```

** SLIP 23_Q2 : Write a C program to Identify the type (Directory, character device, Block device, Regular file, FIFO or pipe, symbolic link or socket) of given file using stat() system call.

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

void print_file_type(mode_t mode) {
    if (S_ISREG(mode)) {
        printf("Regular file\n");
    } else if (S_ISDIR(mode)) {
        printf("Directory\n");
    } else if (S_ISCHR(mode)) {
        printf("Character device\n");
    } else if (S_ISBLK(mode)) {
        printf("Block device\n");
    } else if (S_ISFIFO(mode)) {
        printf("FIFO/pipe\n");
    } else if (S_ISLNK(mode)) {
        printf("Symbolic link\n");
    } else if (S_ISSOCK(mode)) {
        printf("Socket\n");
    } else {
        printf("Unknown file type\n");
    }
}

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

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

    // Get the file status using stat()
    if (stat(filename, &file_stat) == -1) {
        perror("stat");
        return 1;
    }

    // Identify and print the file type
    print_file_type(file_stat.st_mode);
    return 0;
}
```

** SLIP 24_Q1 : Print the type of file and inode number where file name accepted through Command Line

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

void print_file_type(mode_t mode) {
    if (S_ISREG(mode)) {
        printf("File Type: Regular file\n");
    } else if (S_ISDIR(mode)) {
        printf("File Type: Directory\n");
    } else if (S_ISCHR(mode)) {
        printf("File Type: Character device\n");
    } else if (S_ISBLK(mode)) {
        printf("File Type: Block device\n");
    } else if (S_ISFIFO(mode)) {
        printf("File Type: FIFO/pipe\n");
    } else if (S_ISLNK(mode)) {
        printf("File Type: Symbolic link\n");
    } else if (S_ISSOCK(mode)) {
        printf("File Type: Socket\n");
    } else {
        printf("File Type: Unknown\n");
    }
}

int main(int argc, char *argv[]) {
    // Ensure the program receives the file name as an argument
    if (argc != 2) {
        printf("Usage: %s <filename>\n", argv[0]);
        return 1;
    }

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

    // Use stat() to get the file information
    if (stat(filename, &file_stat) == -1) {
        perror("stat");
        return 1;
    }

    // Print the inode number and file type
    printf("Inode Number: %ld\n", (long)file_stat.st_ino);
    print_file_type(file_stat.st_mode);

    return 0;
}
```

** SLIP 24_Q2 : Write a C program which creates a child process to run linux/ unix command or any user defined program. The parent process set the signal handler for death of child signal and Alarm signal. If a child process does not complete its execution in 5 second then parent process kills child process

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

// Global variable to hold child PID
pid_t child_pid;

// Signal handler for child process termination (SIGCHLD)
void handle_child_termination(int sig) {
    int status;
    waitpid(child_pid, &status, WNOHANG);  // Reap the child
    if (WIFEXITED(status)) {
        printf("Child process terminated successfully with exit status %d.\n", WEXITSTATUS(status));
    } else if (WIFSIGNALED(status)) {
        printf("Child process terminated by signal %d.\n", WTERMSIG(status));
    }
}

// Signal handler for alarm (SIGALRM)
void handle_alarm(int sig) {
    printf("5 seconds passed. Child process is taking too long. Killing child...\n");
    kill(child_pid, SIGKILL);  // Send SIGKILL to child process
}

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

    // Set up signal handlers
    signal(SIGCHLD, handle_child_termination);  // Child termination signal
    signal(SIGALRM, handle_alarm);              // Alarm signal

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

    if (child_pid < 0) {
        perror("fork failed");
        return 1;
    }
    if (child_pid == 0) {
        // Child process: execute the command passed by parent
        printf("Child process: Running command '%s'...\n", argv[1]);

        // Execute the command provided by user (using execvp)
        execlp(argv[1], argv[1], NULL);

        // If exec fails
        perror("exec failed");
        exit(1);
    } else {
        // Parent process: set an alarm for 5 seconds
        alarm(5);  // Set the alarm to go off after 5 seconds

        // Wait for the child to finish or be killed
        printf("Parent process: Waiting for child to complete...\n");
        wait(NULL);  // Wait for the child to terminate
    }

    return 0;
}
```

** SLIP 25_Q1 : Write a C Program that demonstrates redirection of standard output to a file

```
==> #include <stdio.h>
 #include <fcntl.h>
 #include <unistd.h>
int main() {
    // File pointer and file descriptor
    int fd;
    // Open a file (create if doesn't exist) and redirect standard output to it
    fd = open("output.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
    if (fd == -1) {
        perror("open");
        return 1;
    }
    // Redirect standard output (stdout) to the file
    if (dup2(fd, STDOUT_FILENO) == -1) {
        perror("dup2");
        close(fd);
        return 1;
    }
    // Now, any output written to stdout will go to output.txt
    printf("This message will be written to output.txt\n");
    printf("This is another line in the file.\n");
    // Close the file descriptor
    close(fd);
    return 0;
 }
```

** SLIP 25_Q2 : Write a C program that redirects standard output to a file output.txt. (use of dup and open system call).

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

int main() {
    // Open the file "output.txt" in write mode (create or overwrite the file)
    int fd = open("output.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);

    // Check if the file was opened successfully
    if (fd == -1) {
        perror("Error opening file");
        return 1;
    }

    // Redirect the standard output (stdout) to the file
    if (dup2(fd, STDOUT_FILENO) == -1) {
        perror("Error redirecting stdout");
        close(fd);
        return 1;
    }

    // Now, any printf() will be written to "output.txt"
    printf("This message will be written to the file output.txt instead of the terminal.\n");
    printf("Redirection of standard output is successful!\n");

    // Close the file descriptor
    close(fd);

    return 0;
}
```