AOS Practical Slips Solution

MSc(computer Science) (Savitribai Phule Pune University)

AOS PRACTICAL SLIPS SOLUTION

Slip no:-1,3,19,24 (10 m)

Take multiple files as Command Line Arguments and print their inode numbers and file types in c

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

int main(int argc, char *argv[]) {
   // Check if at least one file is provided as an argument
   if (argc < 2) {
      printf("Usage: %s <file1> <file2> ...\n", argv[0]);
      return 1;
   }

   // Loop through each provided argument (skip the program name)
   for (int i = 1; i < argc; i++) {
      struct stat fileStat;

      // Attempt to retrieve file status
      if (stat(argv[i], &fileStat) == -1) {
         perror("stat");
```

```
    continue;
}

// Print the file name
printf("File: %s\n", argv[i]);

// Print the inode number
printf("Inode Number: %ld\n", (long)fileStat.st_ino);

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

```
        } else {
            printf("File Type: Unknown\n");
        }

        printf("\n");
    }

    return 0;
}
```

Slip no:-2,14 (10 and 20 m)

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 <sys/stat.h>
#include <time.h>

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

```
    }

    struct stat fileStat;
    if (stat(argv[1], &fileStat) < 0) {
        perror("stat");
        return 1;
    }

    printf("File: %s\n", argv[1]);
    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);
    printf("File permissions: ");
    printf((S_ISDIR(fileStat.st_mode)) ? "d" : "-");
    printf((fileStat.st_mode & S_IRUSR) ? "r" : "-");
    printf((fileStat.st_mode & S_IWUSR) ? "w" : "-");
    printf((fileStat.st_mode & S_IXUSR) ? "x" : "-");
    printf((fileStat.st_mode & S_IRGRP) ? "r" : "-");
    printf((fileStat.st_mode & S_IWGRP) ? "w" : "-");
    printf((fileStat.st_mode & S_IXGRP) ? "x" : "-");
    printf((fileStat.st_mode & S_IROTH) ? "r" : "-");
```

```
    printf((fileStat.st_mode & S_IWOTH) ? "w" : "-");
    printf((fileStat.st_mode & S_IXOTH) ? "x" : "-");
    printf("\n");
    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));

    return 0;
}
```

Slip no:-7,22,25 (10 m)

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

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

int main() {
    // File to which the output will be redirected
    const char *filename = "output.txt";

    // Redirect stdout to the specified file
```

```
    FILE *file = freopen(filename, "w", stdout);
    if (file == NULL) {
        perror("freopen");
        return 1;
    }

    // Now any printf statements will write to "output.txt"
instead of the console
    printf("This line will be written to the file.\n");
    printf("Redirecting standard output in C is easy!\n");

    // Optional: restore stdout to console (if needed)
    fflush(stdout);
    freopen("/dev/tty", "w", stdout);
    printf("This line will be printed to the console.\n");

    return 0;
}
```

Slip no:-8,11,25 (10 and 20 m)

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 <fcntl.h>
#include <unistd.h>

int main() {
   // Open the file output.txt in write-only mode, create it if it doesn't exist, and set file permissions to 0644
   int file = open("output.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
   if (file < 0) {
      perror("open");
      exit(EXIT_FAILURE);
   }

   // Duplicate the file descriptor for stdout (file descriptor 1)
   if (dup2(file, STDOUT_FILENO) < 0) {
      perror("dup2");
      close(file);
      exit(EXIT_FAILURE);
   }
```

```
    // Close the original file descriptor as it's no longer
needed
    close(file);

    // From this point, any output to stdout will go to
output.txt
    printf("This will be written to output.txt\n");

    return 0;
}
```

Slip no:3,15,24 (20 m)

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 = 0; // Global variable to store child process ID

// Signal handler for SIGCHLD (child termination)
void handle_sigchld(int sig) {
    int status;
    pid_t pid = waitpid(child_pid, &status, WNOHANG);
    if (pid > 0) {
        printf("Child process %d terminated.\n", pid);
        exit(0); // Exit the parent process after child termination
    }
}

// Signal handler for SIGALRM (alarm signal)
void handle_sigalrm(int sig) {
    printf("Child process took too long. Killing child process %d.\n", child_pid);
    kill(child_pid, SIGKILL); // Terminate the child process
    waitpid(child_pid, NULL, 0); // Clean up zombie process
    exit(0); // 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 handlers
   signal(SIGCHLD, handle_sigchld); // Handle child
termination
   signal(SIGALRM, handle_sigalrm); // Handle alarm
timeout

   // Fork a child process
   child_pid = fork();
   if (child_pid < 0) {
      perror("fork failed");
      exit(1);
   } else if (child_pid == 0) {
      // Child process: Execute the command
      execvp(argv[1], &argv[1]);
      perror("execvp failed"); // Exec failed if this line runs
      exit(1);
```

```
    } else {
        // Parent process: Set an alarm for 5 seconds
        alarm(5);

        // Wait for the child process to complete or alarm to
trigger
        pause(); // Suspend until a signal arrives
    }

    return 0;
}
```

Slip no:-4,18,23(10 m)

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 <unistd.h>

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

```
    return 1;
  }

  for (int i = 1; i < argc; ++i) {
    if (access(argv[i], F_OK) == 0) {
      printf("File %s is present in the current directory.\n",
argv[i]);
    } else {
      printf("File %s is NOT present in the current
directory.\n", argv[i]);
    }
  }

  return 0;
}
```

Slip no:-9,20,23 (20 m)

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>
```

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

int main(int argc, char *argv[]) {
```

```
    if (argc != 2) {
        printf("Usage: %s <file_path>\n", argv[0]);
        return 1;
    }

    struct stat file_stat;
    if (stat(argv[1], &file_stat) != 0) {
        perror("stat");
        return 1;
    }

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

    return 0;
}
```

Slip no:-7,8,17,19,22 (20 m)

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 pipe_fd[2];
    pid_t pid1, pid2;

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

    // Create the first child process
    if ((pid1 = fork()) == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    }

    if (pid1 == 0) {
        // First child process
        // Redirect stdout to the write end of the pipe
        close(pipe_fd[0]);
```

```
    dup2(pipe_fd[1], STDOUT_FILENO);
    close(pipe_fd[1]);

    // Execute the `ls -l` command
    execlp("ls", "ls", "-l", (char *)NULL);
    perror("execlp");
    exit(EXIT_FAILURE);
}

// Create the second child process
if ((pid2 = fork()) == -1) {
    perror("fork");
    exit(EXIT_FAILURE);
}

if (pid2 == 0) {
    // Second child process
    // Redirect stdin to the read end of the pipe
    close(pipe_fd[1]);
    dup2(pipe_fd[0], STDIN_FILENO);
    close(pipe_fd[0]);
```

```
        // Execute the `wc -l` command
        execlp("wc", "wc", "-l", (char *)NULL);
        perror("execlp");
        exit(EXIT_FAILURE);
    }

    // Parent process
    close(pipe_fd[0]);
    close(pipe_fd[1]);

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

    return 0;
}
```

Slip no:-12,21(20 m)

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>

// Structure to hold file name and size
struct FileInfo {
    char name[256];
    off_t size;
};

// Comparator function to sort FileInfo structures by size
int compare(const void *a, const void *b) {
    struct FileInfo *file1 = (struct FileInfo *)a;
    struct FileInfo *file2 = (struct FileInfo *)b;
    return (file1->size - file2->size);
}

int main(int argc, char *argv[]) {
    // Check if at least one file name is provided
    if (argc < 2) {
        fprintf(stderr, "Usage: %s <file1> <file2> ...\n", argv[0]);
        exit(1);
```

```
    }

    // Array to hold FileInfo structures for each file
    struct FileInfo files[argc - 1];

    // Loop through each file provided as a command-line argument
    for (int i = 1; i < argc; i++) {
        struct stat fileStat;

        // Get file information using stat()
        if (stat(argv[i], &fileStat) == -1) {
            perror("stat");
            continue; // Skip this file if there's an error
        }

        // Store file name and size in the FileInfo structure
        strncpy(files[i - 1].name, argv[i], sizeof(files[i - 1].name) - 1);
        files[i - 1].name[sizeof(files[i - 1].name) - 1] = '\0'; // Ensure null-terminated
        files[i - 1].size = fileStat.st_size;
    }
```

```
    // Sort the files array by size in ascending order
    qsort(files, argc - 1, sizeof(struct FileInfo), compare);

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

    return 0;
}
```

Slip no:-5,17,21 (10 m)

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) {
    // Ignore "." and ".." entries
    if (entry->d_name[0] != '.') {
        printf("%s\n", entry->d_name);
        file_count++;
    }
}

// Close the directory
```

```
    closedir(dir);

    printf("Total number of files: %d\n", file_count);

    return 0;
}
```

Slip no:-13,20(10 m)

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>

pid_t child_pid;

void handle_sigusr1(int sig) {
    printf("Received SIGUSR1, suspending the child process...\n");
    kill(child_pid, SIGSTOP);
}
```

```
void handle_sigusr2(int sig) {
   printf("Received SIGUSR2, resuming the child
process...\n");
   kill(child_pid, SIGCONT);
}

int main() {
   struct sigaction sa_usr1, sa_usr2;

   // Set up handler for SIGUSR1
   sa_usr1.sa_handler = handle_sigusr1;
   sigemptyset(&sa_usr1.sa_mask);
   sa_usr1.sa_flags = 0;
   sigaction(SIGUSR1, &sa_usr1, NULL);

   // Set up handler for SIGUSR2
   sa_usr2.sa_handler = handle_sigusr2;
   sigemptyset(&sa_usr2.sa_mask);
   sa_usr2.sa_flags = 0;
   sigaction(SIGUSR2, &sa_usr2, NULL);
```

```
    if ((child_pid = fork()) == 0) {
        // Child process
        while (1) {
            printf("Child process is running...\n");
            sleep(1);
        }
    } else {
        // Parent process
        sleep(3);  // Let the child process run for a few seconds
        kill(getpid(), SIGUSR1);  // Send SIGUSR1 to suspend the child process
        sleep(3);  // Wait for a few seconds
        kill(getpid(), SIGUSR2);  // Send SIGUSR2 to resume the child process
        sleep(3);  // Let the child process run for a few more seconds
        kill(child_pid, SIGKILL);  // Terminate the child process

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

    return 0;
```

}

Slip no:-5,18 (20 m)

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>

int main() {
    int pipe_fd[2];
    pid_t pid;
    char buffer[128];

    // Messages to be written by the child process
    const char *message1 = "Hello World\n";
    const char *message2 = "Hello SPPU\n";
    const char *message3 = "Linux is Funny\n";

    // Create the pipe
```

```
if (pipe(pipe_fd) == -1) {
    perror("pipe");
    exit(EXIT_FAILURE);
}

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

if (pid == 0) {
    // Child process
    close(pipe_fd[0]); // Close the read end of the pipe

    // Write messages to the pipe
    write(pipe_fd[1], message1, strlen(message1));
    write(pipe_fd[1], message2, strlen(message2));
    write(pipe_fd[1], message3, strlen(message3));

    close(pipe_fd[1]); // Close the write end of the pipe
```

```
    } else {
        // Parent process
        close(pipe_fd[1]); // Close the write end of the pipe

        // Read messages from the pipe and display them
        printf("Parent process reading messages from the
pipe:\n");
        while (read(pipe_fd[0], buffer, sizeof(buffer)) > 0) {
            printf("%s", buffer);
        }

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

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

    return 0;
}
```

Slip no:-6,16(10 m)

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 print_files_by_month(int target_month) {
    DIR *dir;
    struct dirent *entry;
    struct stat file_stat;
    struct tm *file_time;
    char time_str[100];

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

    printf("Files created in month %d:\n", target_month);
```

```
    // Read the directory entries
    while ((entry = readdir(dir)) != NULL) {
        // Get file information using stat()
        if (stat(entry->d_name, &file_stat) == 0) {
            // Get the time information of the file
            file_time = localtime(&file_stat.st_ctime);

            // Check if the file was created in the target month
            if ((file_time->tm_mon + 1) == target_month) {
                strftime(time_str, sizeof(time_str), "%Y-%m-%d
%H:%M:%S", file_time);
                printf("%s (created on: %s)\n", entry->d_name,
time_str);
            }
        }
    }

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

```
int main() {
   int month;

   printf("Enter the month (1-12) to list files created in that month: ");
   scanf("%d", &month);

   if (month < 1 || month > 12) {
      printf("Invalid month. Please enter a value between 1 and 12.\n");
      return 1;
   }

   print_files_by_month(month);

   return 0;
}
```

Slip no:4,16 (20 m)

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 <unistd.h>
#include <signal.h>
#include <sys/wait.h>

void handle_signal(int sig) {
    switch (sig) {
        case SIGHUP:
            printf("Child received SIGHUP signal\n");
            break;
        case SIGINT:
            printf("Child received SIGINT signal\n");
            break;
        case SIGQUIT:
            printf("My DADDY has Killed me!!!\n");
            exit(0);
    }
}

int main() {
    pid_t pid;
```

```
// Create a child process
if ((pid = fork()) == 0) {
    // Child process
    signal(SIGHUP, handle_signal);
    signal(SIGINT, handle_signal);
    signal(SIGQUIT, handle_signal);

    // Run an infinite loop to keep the child process alive
    while (1) {
        pause(); // Wait for signals
    }
} else {
    // Parent process
    for (int i = 0; i < 10; i++) {
        sleep(3);
        if (i % 2 == 0) {
            kill(pid, SIGHUP);
        } else {
            kill(pid, SIGINT);
        }
    }
```

```
    // After 30 seconds, send SIGQUIT signal to terminate the child process
    kill(pid, SIGQUIT);

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

  return 0;
}
```

Slip no:-9,10 (10 and 20 m)

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>
#include <string.h>

int main() {
  int pipe_fd[2];
  pid_t pid;
```

```
char write_msg[] = "Hello from parent process!";
char read_buffer[128];

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

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

if (pid == 0) {
    // Child process
    close(pipe_fd[1]); // Close the write end of the pipe

    // Read message from the pipe
    read(pipe_fd[0], read_buffer, sizeof(read_buffer));
```

```
        printf("Child process received: %s\n", read_buffer);

        close(pipe_fd[0]); // Close the read end of the pipe
    } else {
        // Parent process
        close(pipe_fd[0]); // Close the read end of the pipe

        // Write message to the pipe
        write(pipe_fd[1], write_msg, strlen(write_msg) + 1);
        close(pipe_fd[1]); // Close the write end of the pipe

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

    return 0;
}
```

Slip no:-14,15 (10 m)

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 <dirent.h>
#include <sys/stat.h>

void list_files_greater_than_size(off_t size_threshold) {
    DIR *dir;
    struct dirent *entry;
    struct stat file_stat;

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

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

    // Read the directory entries
    while ((entry = readdir(dir)) != NULL) {
        // Get file information using stat()
        if (stat(entry->d_name, &file_stat) == 0) {
```

```
            // Check if the file size is greater than the threshold
            if (file_stat.st_size > size_threshold) {
                printf("%s (size: %ld bytes)\n", entry->d_name,
file_stat.st_size);
            }
        }
    }

    // Close the directory
    closedir(dir);
}

int main() {
    off_t size_threshold;

    printf("Enter the size threshold (in bytes): ");
    scanf("%ld", &size_threshold);

    list_files_greater_than_size(size_threshold);

    return 0;
}
```

Slip no:-13 (20 m)

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[]) {
   DIR *dir;
   struct dirent *entry;

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

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

```
        return 1;
    }

    printf("Files that begin with '%s':\n", argv[1]);

    // Read the directory entries
    while ((entry = readdir(dir)) != NULL) {
        // Check if the file name starts with the given prefix
        if (strncmp(entry->d_name, argv[1], strlen(argv[1])) ==
0) {
            printf("%s\n", entry->d_name);
        }
    }

    // Close the directory
    closedir(dir);

    return 0;
}
```

Slip no:-11(10 m)

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_resource_limits() {
   struct rlimit rl;

   // Get and print the maximum number of open files
   if (getrlimit(RLIMIT_NOFILE, &rl) == 0) {
      printf("Max number of open files: Soft limit = %lu, Hard limit = %lu\n",
            rl.rlim_cur, rl.rlim_max);
   } else {
      perror("getrlimit RLIMIT_NOFILE");
   }

   // Get and print the maximum size of virtual memory
   if (getrlimit(RLIMIT_AS, &rl) == 0) {
      printf("Max virtual memory size: Soft limit = %lu bytes, Hard limit = %lu bytes\n",
            rl.rlim_cur, rl.rlim_max);
```

```
    } else {
        perror("getrlimit RLIMIT_AS");
    }
}

void set_resource_limits() {
    struct rlimit rl;

    // Set new limits for the maximum number of open files
    rl.rlim_cur = 1024; // New soft limit
    rl.rlim_max = 2048; // New hard limit
    if (setrlimit(RLIMIT_NOFILE, &rl) != 0) {
        perror("setrlimit RLIMIT_NOFILE");
    }

    // Set new limits for the maximum size of virtual memory
    rl.rlim_cur = 512 * 1024 * 1024; // New soft limit (512
MB)
    rl.rlim_max = 1024 * 1024 * 1024; // New hard limit (1
GB)
    if (setrlimit(RLIMIT_AS, &rl) != 0) {
        perror("setrlimit RLIMIT_AS");
```

```
    }
}

int main() {
    printf("Current resource limits:\n");
    print_resource_limits();

    printf("\nSetting new resource limits...\n");
    set_resource_limits();

    printf("\nUpdated resource limits:\n");
    print_resource_limits();

    return 0;
}
```

Slip no:-12 (10 m)

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

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

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

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

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

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

  return 0;
}
```

Slip no:-6 (20 m)

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/wait.h>
#include <sys/resource.h>

int main(int argc, char *argv[]) {
```

```
if (argc != 2) {
    printf("Usage: %s <number_of_children>\n", argv[0]);
    return 1;
}

int n = atoi(argv[1]);
pid_t pid;
struct rusage usage;
struct timeval total_user_time = {0, 0};
struct timeval total_system_time = {0, 0};

for (int i = 0; i < n; ++i) {
    pid = fork();
    if (pid == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    }

    if (pid == 0) {
        // Child process
        printf("Child process %d: PID = %d\n", i + 1, getpid());
        // Simulate some work
```

```
            for (int j = 0; j < 1000000; ++j);
            exit(0);
        }
    }

    // Parent process waits for all child processes to terminate
    for (int i = 0; i < n; ++i) {
        wait4(-1, NULL, 0, &usage);
        timeradd(&total_user_time, &usage.ru_utime, &total_user_time);
        timeradd(&total_system_time, &usage.ru_stime, &total_system_time);
    }

    printf("Total cumulative time spent by child processes:\n");
    printf("User mode: %ld.%06ld seconds\n", (long)total_user_time.tv_sec, (long)total_user_time.tv_usec);
    printf("Kernel mode: %ld.%06ld seconds\n", (long)total_system_time.tv_sec, (long)total_system_time.tv_usec);
```

```
    return 0;
}
```

Slip no:-1 (20 m)

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/wait.h>

void handle_sigalrm(int sig) {
    printf("Alarm is fired!\n");
}

int main() {
    pid_t pid;

    // Set up handler for SIGALRM
    signal(SIGALRM, handle_sigalrm);
```

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

    if (pid == 0) {
        // Child process
        sleep(3); // Wait for 3 seconds
        kill(getppid(), SIGALRM); // Send SIGALRM signal to parent process
        exit(0);
    } else {
        // Parent process
        printf("Parent process waiting for SIGALRM signal...\n");
        pause(); // Wait for signals

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

```
    }

    return 0;
}
```

Slip no:-2 (20 m)

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 <unistd.h>
#include <signal.h>

volatile sig_atomic_t sigint_count = 0;

void handle_sigint(int sig) {
    sigint_count++;
    if (sigint_count == 1) {
        printf("Caught SIGINT (Ctrl-C). Press Ctrl-C again to exit.\n");
    } else {
        printf("Caught SIGINT again. Exiting...\n");
```

```
        exit(0);
    }
}

int main() {
    // Set up handler for SIGINT
    signal(SIGINT, handle_sigint);

    // Run an infinite loop
    while (1) {
        pause(); // Wait for signals
    }

    return 0;
}
```