Advanced Operating Systems Practical
Slip 1

Q.1

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

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
void printInodeAndType(const char *filename)
 {
   struct stat fileStat;
   if (stat(filename, &fileStat) == -1)
  {
      perror("Error getting file information");
      return;
   }
   printf("File: %s\n", filename);
   printf("Inode Number: %ld\n", (long)fileStat.st_ino);
   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_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_ISLNK(fileStat.st_mode))
{
        printf("File Type: Symbolic Link\n");
}
 else if (S_ISSOCK(fileStat.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 EXIT_FAILURE;
    }
    for (int i = 1; i < argc; i++)
```

```
  {
      printInodeAndType(argv[i]);
   }
   return EXIT_SUCCESS;
}
```

Q.2
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 <signal.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
void alarmHandler(int signum)
{
   if (signum == SIGALRM)
  {
      printf("Alarm is fired!\n");
   }
}

int main()
{
   pid_t childPid;
   // Register the signal handler for SIGALRM
   if (signal(SIGALRM, alarmHandler) == SIG_ERR)
 {
      perror("Failed to register signal handler");
      return EXIT_FAILURE;
```

```
    }

    // Fork a child process
    if ((childPid = fork()) < 0)
  {
        perror("Fork failed");
        return EXIT_FAILURE;
    }
else if (childPid == 0)
{
        // Child process
        // Sleep for 2 seconds and then send SIGALRM to the parent
        sleep(2);
        kill(getppid(), SIGALRM);
        exit(EXIT_SUCCESS);
    }
else
{
        // Parent process
        // Wait for the child to complete
        waitpid(childPid, NULL, 0);
    }
    return EXIT_SUCCESS;
}
```

Slip 2

Q.1

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/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <time.h>
void printFileProperties(const char *filename)
{
   struct stat fileStat;
   // Use the stat system call to get file properties
   if (stat(filename, &fileStat) == -1)
{
      perror("Error getting file information");
      return;
   }

   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 Permissions: %o\n", fileStat.st_mode & 0777);
   printf("File Size: %ld bytes\n", (long)fileStat.st_size);

   // Convert access and modification times to a human-readable format
   printf("Last Access Time: %s", ctime(&fileStat.st_atime));
   printf("Last Modification Time: %s", ctime(&fileStat.st_mtime));
   printf("\n");
}

int main(int argc, char *argv[])
{
   if (argc != 2)
```

```
  {
      fprintf(stderr, "Usage: %s <filename>\n", argv[0]);
      return EXIT_FAILURE;
   }
   printFileProperties(argv[1]);
   return EXIT_SUCCESS;
}
```

Q.2

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>
// Global variable to track the number of times Ctrl-C is pressed
int ctrlCCount = 0;
// Signal handler function
void sigintHandler(int signum)
{
   if (signum == SIGINT)
{
      if (ctrlCCount == 0)
{
         printf("Ctrl-C pressed. Press again to exit.\n");
         ctrlCCount++;
      }
 else
{
         printf("Exiting...\n");
         exit(EXIT_SUCCESS);
      }
   }
```

```
}

int main() {
   // Register the signal handler for SIGINT
   if (signal(SIGINT, sigintHandler) == SIG_ERR)
{
       perror("Failed to register signal handler");
       return EXIT_FAILURE;
   }
   printf("Press Ctrl-C to trigger the signal handler.\n");
   // Infinite loop to keep the program running
   while (1)
 {
       // Do some work or sleep here
   }
   return EXIT_SUCCESS;  // Note: This line will not be reached due to the
infinite loop
}
```

Slip 3

Q.1

Print the type of file and inode number where file name accepted through Command Line

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
void printInodeAndType(const char *filename)
 {
   struct stat fileStat;
```

```
    if (stat(filename, &fileStat) == -1)
   {
        perror("Error getting file information");
        return;
    }
    printf("File: %s\n", filename);
    printf("Inode Number: %ld\n", (long)fileStat.st_ino);
    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_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_ISLNK(fileStat.st_mode))
{
        printf("File Type: Symbolic Link\n");
```

```
}
 else if (S_ISSOCK(fileStat.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 EXIT_FAILURE;
    }
    for (int i = 1; i < argc; i++)
  {
        printInodeAndType(argv[i]);
    }
    return EXIT_SUCCESS;
}
```

Q.2

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 <signal.h>
```

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
void childHandler(int signum)
{
   if (signum == SIGCHLD)
{
      printf("Child process has terminated.\n");
   }
}
void alarmHandler(int signum)
{
   if (signum == SIGALRM)
{
      printf("Timeout: Child process took too long. Killing the child.\n");
      exit(EXIT_FAILURE);
   }
}
int main() {
   pid_t childPid;
   // Register the signal handler for SIGCHLD
   if (signal(SIGCHLD, childHandler) == SIG_ERR)
{
      perror("Failed to register SIGCHLD handler");
      return EXIT_FAILURE;
   }
   // Register the signal handler for SIGALRM
   if (signal(SIGALRM, alarmHandler) == SIG_ERR)
{
      perror("Failed to register SIGALRM handler");
```

```
        return EXIT_FAILURE;
    }
    // Fork a child process
    if ((childPid = fork()) < 0)
{
        perror("Fork failed");
        return EXIT_FAILURE;
    } else if (childPid == 0)
{
        // Child process
        // Execute the command (replace "ls" with your desired command)
        execlp("ls", "ls", "-l", NULL);
        perror("execlp failed");
        exit(EXIT_FAILURE);
    } else {
        // Parent process
        // Set an alarm for 5 seconds
        alarm(5);
        // Wait for the child to complete
        waitpid(childPid, NULL, 0);
        // Disable the alarm
        alarm(0);
        printf("Parent process exiting.\n");
    }
    return EXIT_SUCCESS;
}
```

Slip 4

Q.1

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[])
{
   // Check if at least one filename is provided
   if (argc < 2)
 {
       fprintf(stderr, "Usage: %s file1 file2 ... fileN\n", argv[0]);
       return EXIT_FAILURE;
   }
   // Loop through command line arguments starting from index 1
   for (int i = 1; i < argc; i++)
{
       // Use access() to check if the file exists in the current directory
       if (access(argv[i], F_OK) == 0) {
           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 EXIT_SUCCESS;
}
```

Q.2

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 <signal.h>
#include <sys/types.h>
#include <unistd.h>
pid_t childPid;
void sighupHandler(int signum)
 {
    if (signum == SIGHUP) {
        printf("Child process received SIGHUP signal.\n");
    }
}

void sigintHandler(int signum)
{
    if (signum == SIGINT)
 {
        printf("Child process received SIGINT signal.\n");
    }
}
void sigquitHandler(int signum)
{
    if (signum == SIGQUIT)
 {
        printf("Child process received SIGQUIT signal. My Papa has Killed
me!!!\n");
        exit(EXIT_SUCCESS);
    }
}
int main() {
    // Fork a child process
    if ((childPid = fork()) < 0)
```

```
{
        perror("Fork failed");
        return EXIT_FAILURE;
    } else if (childPid == 0)
 {
        // Child process
        // Register signal handlers for SIGHUP, SIGINT, and SIGQUIT
        signal(SIGHUP, sighupHandler);
        signal(SIGINT, sigintHandler);
        signal(SIGQUIT, sigquitHandler);
        // Infinite loop to keep the child process running
        while (1)
    {
            // Do some work or sleep here
        }
    }
else
{
        // Parent process
        // Send SIGHUP or SIGINT every 3 seconds for 15 seconds
        for (int i = 0; i < 5; i++)
{
            sleep(3);
            if (i % 2 == 0)
{
                printf("Sending SIGHUP to child.\n");
                kill(childPid, SIGHUP);
            }
else
{
```

```
            printf("Sending SIGINT to child.\n");
            kill(childPid, SIGINT);
        }
    }
    // After 15 seconds, send SIGQUIT to terminate the child
    sleep(3);
    printf("Sending SIGQUIT to child.\n");
    kill(childPid, SIGQUIT);
    // Wait for the child to complete
    waitpid(childPid, NULL, 0);
    printf("Parent process exiting.\n");
  }
  return EXIT_SUCCESS;
}
```

Slip 5

Q.1

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;
  // Open the current directory
  dir = opendir(".");
  // Check if the directory can be opened
  if (dir == NULL)
{
```

```
        perror("Error opening directory");
        return EXIT_FAILURE;
    }
    int fileCount = 0;
    // Read each entry in the directory
    while ((entry = readdir(dir)) != NULL)
{
        if (entry->d_type == DT_REG)
{  // Check if it's a regular file
            printf("%s\n", entry->d_name);
            fileCount++;
        }
    }
    // Close the directory
    closedir(dir);
    printf("Number of files in the current directory: %d\n", fileCount);
    return EXIT_SUCCESS;
}
```

Q.2

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>
#define MSG_SIZE 100
int main()
{
    int pipe_fd[2];
```

```
    pid_t child_pid;
    // Create a pipe
    if (pipe(pipe_fd) == -1)
{
        perror("Pipe creation failed");
        return EXIT_FAILURE;
    }
    // Fork a child process
    if ((child_pid = fork()) < 0)
{
        perror("Fork failed");
        return EXIT_FAILURE;
    }
    if (child_pid == 0)
 {
        // Child process
        // Close the read end of the pipe in the child
        close(pipe_fd[0]);
        // Messages to be written to the pipe
        char message1[] = "Hello World";
        char message2[] = "Hello SPPU";
        char message3[] = "Linux is Funny";
        // Write messages to the pipe
        write(pipe_fd[1], message1, MSG_SIZE);
        write(pipe_fd[1], message2, MSG_SIZE);
        write(pipe_fd[1], message3, MSG_SIZE);

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

```
    }
else
{
        // Parent process
        // Close the write end of the pipe in the parent
        close(pipe_fd[1]);
        // Buffer to read messages from the pipe
        char buffer[MSG_SIZE];
        // Read and display messages from the pipe
        printf("Messages from child process:\n");
        // Read message 1
        read(pipe_fd[0], buffer, MSG_SIZE);
       printf("%s\n", buffer);
        // Read message 2
        read(pipe_fd[0], buffer, MSG_SIZE);
        printf("%s\n", buffer);
        // Read message 3
        read(pipe_fd[0], buffer, MSG_SIZE);
        printf("%s\n", buffer);
        // Close the read end of the pipe in the parent
        close(pipe_fd[0]);
    }
   return EXIT_SUCCESS;
}
```

Slip 6

Q.1

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>
int main()
{
   DIR *dir;
   struct dirent *entry;
   struct stat fileStat;
   // Open the current directory
   dir = opendir(".");
   // Check if the directory can be opened
   if (dir == NULL)
{
       perror("Error opening directory");
       return EXIT_FAILURE;
   }
   // Get the current month
   time_t t = time(NULL);
   struct tm *tm_info = localtime(&t);
   int currentMonth = tm_info->tm_mon + 1; // Month is zero-based
   // Read each entry in the directory
   while ((entry = readdir(dir)) != NULL)
 {
       // Use stat to get information about the file
       if (stat(entry->d_name, &fileStat) == -1)
{
           perror("Error getting file information");
           continue;
       }
       // Check if the file was created in the current month
```

```
        struct tm *fileTime = localtime(&fileStat.st_ctime);
        if (fileTime->tm_mon + 1 == currentMonth)
 {
            printf("%s\n", entry->d_name);
        }
    }
    // Close the directory
    closedir(dir);
    return EXIT_SUCCESS;
}
```

Q.2

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) {
        fprintf(stderr, "Usage: %s <number_of_children>\n", argv[0]);
        return EXIT_FAILURE;
    }
    int n = atoi(argv[1]);
    int i;
    for (i = 0; i < n; i++)
{
        pid_t child_pid = fork();
        if (child_pid < 0)
```

```
{
            perror("Fork failed");
            return EXIT_FAILURE;
        } else if (child_pid == 0)
{
            // Child process
            // Perform some work in the child (e.g., sleep for a short duration)
            sleep(1);
            exit(EXIT_SUCCESS);
        }
    }
   // Parent process
    // Wait for all child processes to terminate
    for (i = 0; i < n; i++)
{
        wait(NULL);
    }
    // Get resource usage information
    struct rusage usage;
    if (getrusage(RUSAGE_CHILDREN, &usage) == -1)
{
        perror("Error getting resource usage information");
        return EXIT_FAILURE;
    }
    // Display total cumulative time spent in user and kernel mode
    printf("Total User Time: %ld seconds %ld microseconds\n",
usage.ru_utime.tv_sec, usage.ru_utime.tv_usec);
    printf("Total Kernel Time: %ld seconds %ld microseconds\n",
usage.ru_stime.tv_sec, usage.ru_stime.tv_usec);
    return EXIT_SUCCESS;
}
```

Slip 7

Q.1

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

```
#include <stdio.h>
int main()
{
   // Open a file for writing
   FILE *file = freopen("output.txt", "w", stdout);
   if (file == NULL) {
      perror("Error opening file");
      return 1;
   }
   // Redirect standard output to the file
   printf("This is redirected to a file.\n");
   // Close the file (optional, as fclose will also close stdout)
   fclose(file);
   // Now standard output is restored to the original state
   // Output to the console
   printf("This is printed to the console.\n");
   return 0;
}
```

Q.2

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 child_pid;
    // Create a pipe
    if (pipe(pipe_fd) == -1)
{
        perror("Pipe creation failed");
        return EXIT_FAILURE;
    }
    // Fork a child process
    if ((child_pid = fork()) < 0)
{
        perror("Fork failed");
        return EXIT_FAILURE;
    }
    if (child_pid == 0)
{
        // Child process
        // Close the read end of the pipe in the child
        close(pipe_fd[0]);
        // Redirect stdout to the write end of the pipe
        dup2(pipe_fd[1], STDOUT_FILENO);
        // Close the unused write end of the pipe
        close(pipe_fd[1]);
        // Execute the "ls -l" command
        execlp("ls", "ls", "-l", (char *)NULL);
        // If execlp fails, print an error message
        perror("execlp failed");
        exit(EXIT_FAILURE);
    } else {
        // Parent process
```

```
        // Close the write end of the pipe in the parent
        close(pipe_fd[1]);
        // Redirect stdin to the read end of the pipe
        dup2(pipe_fd[0], STDIN_FILENO);
        // Close the unused read end of the pipe
        close(pipe_fd[0]);
        // Execute the "wc -l" command
        execlp("wc", "wc", "-l", (char *)NULL);
        // If execlp fails, print an error message
        perror("execlp failed");
        exit(EXIT_FAILURE);
    }
    return EXIT_SUCCESS;
}
```

Slip 8

Q.1

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 file_fd;
    // Open or create a file for writing (truncate if it already exists)
    if ((file_fd = open("output.txt", O_WRONLY | O_CREAT | O_TRUNC, 0666)) == -1)
{
        perror("Error opening file");
        return EXIT_FAILURE;
```

```
    }
    // Duplicate file descriptor to point to standard output (stdout)
    if (dup2(file_fd, STDOUT_FILENO) == -1)
{
        perror("Error redirecting standard output");
        close(file_fd);
        return EXIT_FAILURE;
    }
    // Close the original file descriptor
    close(file_fd);
    // Now, standard output is redirected to "output.txt"
    // Print to the redirected standard output
    printf("This is redirected to a file.\n");
    // Close standard output (optional)
    // close(STDOUT_FILENO);
    return EXIT_SUCCESS;
}
```

Q.2

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 child_pid;
    // Create a pipe
    if (pipe(pipe_fd) == -1)
```

```
{
        perror("Pipe creation failed");
        return EXIT_FAILURE;
    }
    // Fork a child process
    if ((child_pid = fork()) < 0)
{
        perror("Fork failed");
        return EXIT_FAILURE;
    }
    if (child_pid == 0)
{
        // Child process
        // Close the read end of the pipe in the child
        close(pipe_fd[0]);
        // Redirect stdout to the write end of the pipe
        dup2(pipe_fd[1], STDOUT_FILENO);
        // Close the unused write end of the pipe
        close(pipe_fd[1]);
        // Execute the "ls -l" command
        execlp("ls", "ls", "-l", (char *)NULL);
        // If execlp fails, print an error message
        perror("execlp failed");
        exit(EXIT_FAILURE);
    } else {
        // Parent process
        // Close the write end of the pipe in the parent
        close(pipe_fd[1]);
        // Redirect stdin to the read end of the pipe
        dup2(pipe_fd[0], STDIN_FILENO);
```

```
        // Close the unused read end of the pipe
        close(pipe_fd[0]);
        // Execute the "wc -l" command
        execlp("wc", "wc", "-l", (char *)NULL);
        // If execlp fails, print an error message
        perror("execlp failed");
        exit(EXIT_FAILURE);
    }
    return EXIT_SUCCESS;
}
```

Slip 9

Q.1

Generate parent process to write unnamed pipe and will read from it

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

#define MSG_SIZE 100

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

    // Create a pipe
    if (pipe(pipe_fd) == -1) {
        perror("Pipe creation failed");
        return EXIT_FAILURE;
    }
```

```
// Fork a child process
if ((child_pid = fork()) < 0) {
    perror("Fork failed");
    return EXIT_FAILURE;
}
if (child_pid == 0) {
    // Child process
    // Close the write end of the pipe in the child
    close(pipe_fd[1]);
    // Buffer to read message from the pipe
    char buffer[MSG_SIZE];
    // Read the message from the pipe
    read(pipe_fd[0], buffer, MSG_SIZE);
    printf("Child process received message: %s\n", buffer);
    // Close the read end of the pipe in the child
    close(pipe_fd[0]);
    exit(EXIT_SUCCESS);
} else {
    // Parent process
    // Close the read end of the pipe in the parent
    close(pipe_fd[0]);
    // Message to be written to the pipe
    char message[] = "Hello from the parent process!";
    // Write the message to the pipe
    write(pipe_fd[1], message, sizeof(message));
    // Close the write end of the pipe in the parent
    close(pipe_fd[1]);
    // Wait for the child to complete
    wait(NULL);
```

```
        printf("Parent process exiting.\n");
    }

    return EXIT_SUCCESS;
}
```

Q.2

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 identifyFileType(const char *filename) {
    struct stat fileStat;

    // Use the stat system call to get file information
    if (stat(filename, &fileStat) == -1) {
        perror("Error getting file information");
        return;
    }

    // Check file type using st_mode field in the struct stat
    if (S_ISDIR(fileStat.st_mode)) {
        printf("%s is a directory.\n", filename);
    } else if (S_ISCHR(fileStat.st_mode)) {
        printf("%s is a character device.\n", filename);
    } else if (S_ISBLK(fileStat.st_mode)) {
        printf("%s is a block device.\n", filename);
```

```
    } else if (S_ISREG(fileStat.st_mode)) {
        printf("%s is a regular file.\n", filename);
    } else if (S_ISFIFO(fileStat.st_mode)) {
        printf("%s is a FIFO or pipe.\n", filename);
    } else if (S_ISLNK(fileStat.st_mode)) {
        printf("%s is a symbolic link.\n", filename);
    } else if (S_ISSOCK(fileStat.st_mode)) {
        printf("%s is a socket.\n", filename);
    } else {
        printf("%s is of an unknown type.\n", filename);
    }
}

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

    identifyFileType(argv[1]);

    return EXIT_SUCCESS;
}
```

Slip 10

Q.1

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 pipe_fd[2];
    pid_t child1_pid, child2_pid;

    // Create a pipe
    if (pipe(pipe_fd) == -1) {
        perror("Pipe creation failed");
        return EXIT_FAILURE;
    }

    // Fork the first child process
    if ((child1_pid = fork()) < 0) {
        perror("Fork for child 1 failed");
        return EXIT_FAILURE;
    }

    if (child1_pid == 0) {
        // Child process 1

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

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

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

```
    // Execute the first command (e.g., "ls")
    execlp("ls", "ls", (char *)NULL);

    // If execlp fails, print an error message
    perror("execlp for child 1 failed");

    exit(EXIT_FAILURE);
}

// Fork the second child process
if ((child2_pid = fork()) < 0) {
    perror("Fork for child 2 failed");
    return EXIT_FAILURE;
}

if (child2_pid == 0) {
    // Child process 2

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

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

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

    // Execute the second command (e.g., "wc -l")
    execlp("wc", "wc", "-l", (char *)NULL);
```

```
        // If execlp fails, print an error message
        perror("execlp for child 2 failed");

        exit(EXIT_FAILURE);
    }

    // Close the unused ends of the pipe in the parent
    close(pipe_fd[0]);
    close(pipe_fd[1]);

    // Wait for both child processes to complete
    waitpid(child1_pid, NULL, 0);
    waitpid(child2_pid, NULL, 0);

    printf("Parent process exiting.\n");

    return EXIT_SUCCESS;
}
```

Q.2

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>

#define MSG_SIZE 100

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

```
pid_t child_pid;

// Create a pipe
if (pipe(pipe_fd) == -1) {
    perror("Pipe creation failed");
    return EXIT_FAILURE;
}

// Fork a child process
if ((child_pid = fork()) < 0) {
    perror("Fork failed");
    return EXIT_FAILURE;
}

if (child_pid == 0) {
    // Child process

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

    // Buffer to read message from the pipe
    char buffer[MSG_SIZE];

    // Read the message from the pipe
    read(pipe_fd[0], buffer, MSG_SIZE);

    printf("Child process received message: %s\n", buffer);

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

```
        exit(EXIT_SUCCESS);
    } else {
        // Parent process

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

        // Message to be written to the pipe
        char message[] = "Hello from the parent process!";

        // Write the message to the pipe
        write(pipe_fd[1], message, sizeof(message));

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

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

        printf("Parent process exiting.\n");
    }

    return EXIT_SUCCESS;
}
```