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 fd[2];  // pipe file descriptors
   pid_t pid1, pid2;
   // Create a pipe
   if (pipe(fd) == -1) {
      perror("pipe");
      exit(1);
   }
   // First child process (producer)
   pid1 = fork();
   if (pid1 == -1) {
      perror("fork");
      exit(1);
   }
   if (pid1 == 0) {
      // Child 1: execute "ls"
      dup2(fd[1], STDOUT_FILENO);  // redirect stdout to pipe write-end
      close(fd[0]); // close unused read end
      close(fd[1]);
      execlp("ls", "ls", "-l", NULL);
      perror("execlp"); // only runs if execlp fails
      exit(1);
   }
```

```
    // Second child process (consumer)
    pid2 = fork();
    if (pid2 == -1) {
        perror("fork");
        exit(1);
    }
    if (pid2 == 0) {
        // Child 2: execute "grep .c"
        dup2(fd[0], STDIN_FILENO);  // redirect stdin to pipe read-end
        close(fd[1]); // close unused write end
        close(fd[0]);
        execlp("grep", "grep", ".c", NULL);
        perror("execlp");
        exit(1);
    }
    // Parent process closes both ends of pipe
    close(fd[0]);
    close(fd[1]);
    // Wait for both children
    waitpid(pid1, NULL, 0);
    waitpid(pid2, NULL, 0);
    return 0;
}
```

Q2. Generate parent process to write unnamed pipe and will write into it. Also generate childprocess which will read from pipe.

->

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

```
int main() {
    int fd[2];        // pipe file descriptors
    pid_t pid;
    char write_msg[] = "Hello from Parent!";
    char read_msg[100];

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

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

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

    if (pid > 0) {
        // Parent process
        close(fd[0]);  // close unused read end
        write(fd[1], write_msg, strlen(write_msg) + 1);
        close(fd[1]);  // close write end after writing
    } else {
        // Child process
        close(fd[1]);  // close unused write end
        read(fd[0], read_msg, sizeof(read_msg));
        printf("Child received: %s\n", read_msg);
        close(fd[0]);  // close read end
    }
```

```
    return 0;
}
```

Slip11

Q.1) 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_limit(int resource, const char *name) {
    struct rlimit limit;
    if (getrlimit(resource, &limit) == 0) {
        printf("%s -> soft: %ld, hard: %ld\n",
               name,
               (long)limit.rlim_cur,
               (long)limit.rlim_max);
    } else {
        perror("getrlimit");
    }
}

int main() {
    printf("Before setting limits:\n");
    print_limit(RLIMIT_FSIZE, "Max file size");
    print_limit(RLIMIT_DATA,  "Max data size");
    print_limit(RLIMIT_STACK, "Max stack size");
    print_limit(RLIMIT_NOFILE,"Max open files");

    // Example: change file size limit to 1 MB
    struct rlimit new_limit;
    new_limit.rlim_cur = 1024 * 1024;   // soft limit = 1 MB
    new_limit.rlim_max = RLIM_INFINITY; // keep hard limit unchanged
```

```
    if (setrlimit(RLIMIT_FSIZE, &new_limit) == -1) {
        perror("setrlimit");
    } else {
        printf("\nAfter setting new file size limit:\n");
        print_limit(RLIMIT_FSIZE, "Max file size");
    }

    return 0;
}
```

Q.2) 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 (or create) file for writing, truncate if exists
    fd = open("output.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
    if (fd < 0) {
        perror("open");
        exit(1);
    }

    // Duplicate file descriptor onto STDOUT (fd = 1)
    if (dup2(fd, STDOUT_FILENO) == -1) {
        perror("dup2");
        exit(1);
    }
```

```
    close(fd); // fd no longer needed, since STDOUT now refers to file

    // Any printf now goes to output.txt
    printf("This line will be written into output.txt\n");
    printf("Hello from redirected stdout!\n");

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

int main() {
    pid_t pid;
    int status;

    pid = fork();

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

    if (pid == 0) {
        // Child process
        printf("Child process running (PID = %d)\n", getpid());
        exit(7); // Child terminates with exit status 7
```

```
    } else {
        // Parent process waits
        wait(&status);

        if (WIFEXITED(status)) {
            printf("Child exited normally with status = %d\n", WEXITSTATUS(status));
        } else if (WIFSIGNALED(status)) {
            printf("Child terminated by signal = %d\n", WTERMSIG(status));
        } else {
            printf("Child terminated abnormally\n");
        }
    }

    return 0;
}
```

Q.2) Write a C program which receives file names as command line arguments and display thosefilenames 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 info
struct FileInfo {
    char name[256];
    off_t size;
};

// Comparison function for qsort
int compare(const void *a, const void *b) {
    struct FileInfo *f1 = (struct FileInfo *)a;
    struct FileInfo *f2 = (struct FileInfo *)b;
```

```
    if (f1->size < f2->size) return -1;
    else if (f1->size > f2->size) return 1;
    else return 0;
}

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

    struct FileInfo files[argc-1];
    struct stat st;
    int count = 0;

    // Collect file names and sizes
    for (int i = 1; i < argc; i++) {
        if (stat(argv[i], &st) == 0) {
            strcpy(files[count].name, argv[i]);
            files[count].size = st.st_size;
            count++;
        } else {
            perror(argv[i]);
        }
    }

    // Sort by size
    qsort(files, count, sizeof(struct FileInfo), compare);

    // Display sorted filenames
    printf("Files sorted by size (ascending):\n");
    for (int i = 0; i < count; i++) {
        printf("%s (%ld bytes)\n", files[i].name, (long)files[i].size);
```

```
    }

    return 0;
}
```

Slip 13

Q.1) Write a C program that illustrates suspending and resuming processes using signals [10 Marks]

Q.2) 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 [20 Marks

Q.1) Write a C program that illustrates suspending and resuming processes using signals
[10 Marks]
Q.2) 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 [20 Marks

Q.1) 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>
```

```
int main() {
    pid_t pid;

    pid = fork();

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

    if (pid == 0) {
        // Child process: runs in a loop
        for (int i = 1; i <= 10; i++) {
            printf("Child working... iteration %d (PID = %d)\n", i, getpid());
            sleep(1);
        }
        exit(0);
    } else {
        // Parent process
        sleep(3);  // let child run a bit

        printf("\nParent: Suspending child (SIGSTOP)\n");
        kill(pid, SIGSTOP);   // suspend child
        sleep(5);

        printf("\nParent: Resuming child (SIGCONT)\n");
        kill(pid, SIGCONT);   // resume child

        wait(NULL); // wait for child to finish
        printf("\nParent: Child process finished.\n");
    }
```

```
    return 0;
}
```

Q.2) 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) {
        fprintf(stderr, "Usage: %s <prefix>\n", argv[0]);
        exit(1);
    }

    char *prefix = argv[1];
    DIR *d;
    struct dirent *dir;

    d = opendir(".");
    if (d == NULL) {
        perror("opendir");
        exit(1);
    }

    printf("Files beginning with \"%s\":\n", prefix);
    while ((dir = readdir(d)) != NULL) {
        if (strncmp(dir->d_name, prefix, strlen(prefix)) == 0) {
            printf("%s\n", dir->d_name);
        }
    }
```

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

Example Run

```
$ ls
foo1.txt  foo_bar.c  bar.txt  main.c
$ ./a.out foo
Files beginning with "foo":
foo1.txt
foo_bar.c
```

Slip 14

Q.1) Display all the files from current directory whose size is greater that n Bytes Where n is accept from user. [10 Marks]

Q.2) 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

Q.1) 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() {
    DIR *d;
    struct dirent *dir;
    struct stat st;
    long n;

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

    d = opendir(".");
    if (d == NULL) {
        perror("opendir");
        exit(1);
    }

    printf("Files greater than %ld bytes:\n", n);
    while ((dir = readdir(d)) != NULL) {
        if (stat(dir->d_name, &st) == 0) {
            if (S_ISREG(st.st_mode) && st.st_size > n) {
                printf("%s (%ld bytes)\n", dir->d_name, st.st_size);
            }
        }
    }
```

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

Q.2) 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 <pwd.h>
#include <grp.h>
#include <unistd.h>

void print_permissions(mode_t mode) {
    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" : "-");
}

int main() {
    char filename[256];
    struct stat st;

    printf("Enter the filename: ");
    scanf("%s", filename);
```

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

    printf("File: %s\n", filename);
    printf("Inode number: %ld\n", (long)st.st_ino);
    printf("Number of hard links: %ld\n", (long)st.st_nlink);
    printf("File size: %ld bytes\n", (long)st.st_size);
    printf("File permissions: ");
    print_permissions(st.st_mode);
    printf("\n");

    printf("Owner UID: %d, GID: %d\n", st.st_uid, st.st_gid);
    printf("Last access time: %s", ctime(&st.st_atime));
    printf("Last modification time: %s", ctime(&st.st_mtime));
    printf("Last status change time: %s", ctime(&st.st_ctime));
    return 0;
}
```

Slip 15

Q.1) Display all the files from current directory whose size is greater that n Bytes Where n is accept from user [10 Marks]

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

Q.1) 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() {
    DIR *d;
    struct dirent *dir;
    struct stat st;
    long n;
```

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

    d = opendir(".");
    if (d == NULL) {
        perror("opendir");
        exit(1);
    }

    printf("Files greater than %ld bytes:\n", n);
    while ((dir = readdir(d)) != NULL) {
        if (stat(dir->d_name, &st) == 0) {
            if (S_ISREG(st.st_mode) && st.st_size > n) {
                printf("%s (%ld bytes)\n", dir->d_name, st.st_size);
            }
        }
    }

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

Q.2) Write a C program which creates a child process to run linux/ unix command or any userdefined 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>

pid_t child_pid = 0;

void child_handler(int sig) {
    int status;
```

```
    pid_t pid = wait(&status);  // collect child exit status
    if (pid == child_pid) {
        printf("\nParent: Child %d terminated with status %d\n", pid, WEXITSTATUS(status));
        child_pid = 0;
    }
}

void alarm_handler(int sig) {
    if (child_pid != 0) {
        printf("\nParent: Child did not finish in 5 seconds. Killing child %d\n", child_pid);
        kill(child_pid, SIGKILL);
    }
}

int main() {
    // Set up signal handlers
    signal(SIGCHLD, child_handler);
    signal(SIGALRM, alarm_handler);

    child_pid = fork();
    if (child_pid < 0) {
        perror("fork");
        exit(1);
    }

    if (child_pid == 0) {
        // Child process: run a command (example: sleep 10)
        execlp("sleep", "sleep", "10", NULL); // change to any command
        perror("execlp failed");
        exit(1);
    } else {
        // Parent process
        printf("Parent: Monitoring child process %d\n", child_pid);
```

```
        alarm(5);  // set 5-second alarm
        pause();   // wait for signals
        return 0;
    }
}
```

Slip 16

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 *d;
    struct dirent *dir;
    struct stat st;
    char month[20];
    struct tm *tm_info;

    printf("Enter the month name (e.g., Jan, Feb): ");
    scanf("%s", month);

    d = opendir(".");
    if (d == NULL) {
        perror("opendir");
        exit(1);
    }

    printf("Files created/modified in %s:\n", month);
    while ((dir = readdir(d)) != NULL) {
        if (stat(dir->d_name, &st) == 0) {
```

```
            tm_info = localtime(&st.st_mtime);
            char file_month[4];
            strftime(file_month, sizeof(file_month), "%b", tm_info); // get month abbreviation
            if (strcasecmp(file_month, month) == 0) {
                printf("%s\n", dir->d_name);
            }
        }
    }

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

Q.2) 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 <time.h>

void signal_handler(int sig) {
    if (sig == SIGHUP) {
        printf("Child: Received SIGHUP\n");
    } else if (sig == SIGINT) {
        printf("Child: Received SIGINT\n");
    } else if (sig == SIGQUIT) {
        printf("My DADDY has Killed me!!!\n");
        exit(0);
    }
}

int main() {
```

```
pid_t pid = fork();

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

if (pid == 0) {
    // Child process: set signal handlers
    signal(SIGHUP, signal_handler);
    signal(SIGINT, signal_handler);
    signal(SIGQUIT, signal_handler);

    // Keep child alive waiting for signals
    while(1) {
        pause(); // wait for signals
    }
} else {
    // Parent process
    printf("Parent: Sending signals to child %d\n", pid);
    for (int i = 0; i < 10; i++) { // 10 times * 3 sec = 30 sec
        sleep(3);
        if (i % 2 == 0) {
            kill(pid, SIGHUP);
        } else {
            kill(pid, SIGINT);
        }
    }

    // After 30 sec send SIGQUIT
    kill(pid, SIGQUIT);
    wait(NULL); // wait for child termination
    printf("Parent: Child terminated.\n");
```

```
    }
    return 0;
}
```

Slip 17

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 *d;
    struct dirent *dir;
    int count = 0;

    d = opendir(".");
    if (d == NULL) {
        perror("opendir");
        exit(1);
    }

    printf("Files in current directory:\n");
    while ((dir = readdir(d)) != NULL) {
        // Skip "." and ".."
        if (dir->d_name[0] != '.') {
            printf("%s\n", dir->d_name);
            count++;
        }
    }

    printf("\nTotal number of files: %d\n", count);
    closedir(d);
    return 0;
```

}

Q.2) Write a C program to implement the following unix/linux command (use fork, pipe and execsystem call). Your program should block the signal Ctrl-C and Ctrl-\ signal during theexecution. i. Ls –l | wc –l

->

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

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

    // Block Ctrl-C (SIGINT) and Ctrl-\ (SIGQUIT)
    sigset_t set;
    sigemptyset(&set);
    sigaddset(&set, SIGINT);
    sigaddset(&set, SIGQUIT);
    sigprocmask(SIG_BLOCK, &set, NULL);

    if (pipe(fd) == -1) {
        perror("pipe");
        exit(1);
    }

    pid1 = fork();
    if (pid1 < 0) {
        perror("fork");
        exit(1);
    }

    if (pid1 == 0) {
```

```
    // First child: ls -l
    dup2(fd[1], STDOUT_FILENO); // redirect stdout to pipe
    close(fd[0]);
    close(fd[1]);

    execlp("ls", "ls", "-l", NULL);
    perror("execlp ls");
    exit(1);
}

pid2 = fork();
if (pid2 < 0) {
    perror("fork");
    exit(1);
}

if (pid2 == 0) {
    // Second child: wc -l
    dup2(fd[0], STDIN_FILENO); // redirect stdin from pipe
    close(fd[1]);
    close(fd[0]);

    execlp("wc", "wc", "-l", NULL);
    perror("execlp wc");
    exit(1);
}

// Parent process closes pipe
close(fd[0]);
close(fd[1]);

// Wait for both children
waitpid(pid1, NULL, 0);
```

```
    waitpid(pid2, NULL, 0);

    return 0;
}
```

Slip 18

Q.1) 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() {
    char filename[256];
    DIR *d;
    struct dirent *dir;
    int found = 0;

    printf("Enter the filename to search: ");
    scanf("%s", filename);

    d = opendir(".");
    if (d == NULL) {
        perror("opendir");
        exit(1);
    }

    while ((dir = readdir(d)) != NULL) {
        if (strcmp(dir->d_name, filename) == 0) {
```

```
            found = 1;
            break;
        }
    }

    closedir(d);

    if (found) {
        printf("File '%s' is present in the current directory.\n", filename);
    } else {
        printf("File '%s' is NOT present in the current directory.\n", filename);
    }
    return 0;
}
```

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>
#include <string.h>
int main() {
    int fd[2];
    pid_t pid;
    char *messages[] = {"Hello World", "Hello SPPU", "Linux is Funny"};
    char buffer[100];
    if (pipe(fd) == -1) {
        perror("pipe");
        exit(1);
    }
    pid = fork();
    if (pid < 0) {
        perror("fork");
        exit(1);
```

```
    }
    if (pid == 0) {
        // Child process: write messages to pipe
        close(fd[0]); // close unused read end
        for (int i = 0; i < 3; i++) {
            write(fd[1], messages[i], strlen(messages[i]) + 1); // include '\0'
        }
        close(fd[1]);
        exit(0);
    } else {
        // Parent process: read messages from pipe
        close(fd[1]); // close unused write end
        for (int i = 0; i < 3; i++) {
            read(fd[0], buffer, sizeof(buffer));
            printf("Parent received: %s\n", buffer);
        }
        close(fd[0]);
    }
    return 0;
}
```

Slip 19

Q.1) 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>

void print_file_type(mode_t mode) {
    if (S_ISREG(mode)) printf("Regular File");
    else if (S_ISDIR(mode)) printf("Directory");
    else if (S_ISCHR(mode)) printf("Character Device");
    else if (S_ISBLK(mode)) printf("Block Device");
    else if (S_ISFIFO(mode)) printf("FIFO/Pipe");
```

```
    else if (S_ISLNK(mode)) printf("Symbolic Link");
    else if (S_ISSOCK(mode)) printf("Socket");
    else printf("Unknown");
}

int main(int argc, char *argv[]) {
    struct stat st;

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

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

        printf("File: %s\n", argv[i]);
        printf("Inode number: %ld\n", (long)st.st_ino);
        printf("File type: ");
        print_file_type(st.st_mode);
        printf("\n\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 fd[2];
    pid_t pid1, pid2;

    if (pipe(fd) == -1) {
        perror("pipe");
        exit(1);
    }

    pid1 = fork();
    if (pid1 < 0) {
        perror("fork");
        exit(1);
    }

    if (pid1 == 0) {
        // First child: ls -l
        dup2(fd[1], STDOUT_FILENO); // redirect stdout to pipe
        close(fd[0]);
        close(fd[1]);
        execlp("ls", "ls", "-l", NULL);
        perror("execlp ls");
        exit(1);
    }

    pid2 = fork();
    if (pid2 < 0) {
        perror("fork");
        exit(1);
    }

    if (pid2 == 0) {
```

```
        // Second child: wc -l
        dup2(fd[0], STDIN_FILENO); // redirect stdin from pipe
        close(fd[1]);
        close(fd[0]);
        execlp("wc", "wc", "-l", NULL);
        perror("execlp wc");
        exit(1);
    }
    // Parent process closes pipe
    close(fd[0]);
    close(fd[1]);
    // Wait for both children
    waitpid(pid1, NULL, 0);
    waitpid(pid2, NULL, 0);
    return 0;
}
```

Slip 20

Q.1) 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>

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

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

```
    if (pid == 0) {
        // Child process: run a loop
        for (int i = 1; i <= 10; i++) {
            printf("Child working... iteration %d (PID = %d)\n", i, getpid());
            sleep(1);
        }
        exit(0);
    } else {
        // Parent process: suspend and resume child
        sleep(3); // Let child run a few iterations
        printf("\nParent: Suspending child (SIGSTOP)\n");
        kill(pid, SIGSTOP); // suspend child
        sleep(5);
        printf("\nParent: Resuming child (SIGCONT)\n");
        kill(pid, SIGCONT); // resume child

        wait(NULL); // Wait for child to finish
        printf("\nParent: Child process finished.\n");
    }

    return 0;
}
```

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/stat.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 type\n");
}

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

    struct stat st;
    if (stat(argv[1], &st) == -1) {
        perror("stat");
        return 1;
    }

    printf("File: %s\n", argv[1]);
    printf("File type: ");
    print_file_type(st.st_mode);

    return 0;
}
```