Suyog Varpe

1.To create ‘n’ children. When the children will terminate, display total cumulative time children spent in user and kernel mode.

```
#include<sys/wait.h>
#include<unistd.h>
#include<time.h>
#include<stdio.h>
#include<stdlib.h>
int main(void)
{
    int n, i,status=0;
    time_t currentTime,completionTime;
    double diff_t;
    pid_t pid;
    printf("Enter the value of n children:");
    scanf("%d", &n);
    printf("Creating a %d children",n);
    for(i=0;i<n;i++)
    {
            pid=fork();
            if(pid==0)
            {
                    time(&currentTime);
                    printf("\n For children %d : child Process started at
%s",i+1,ctime(&currentTime));
                    sleep(5);
                    time(&completionTime);
                    printf(" For children %d : child Process ended at
%s",i+1,ctime(&completionTime));
                    diff_t=difftime(completionTime,currentTime);
                    printf("Total cumulative time children spent in user mode to kernel mode for
children %d: =%f\n", i+1,diff_t);
                    exit(0);
            }
            wait(&status);
    }
    printf("\nAll children process has terminated\n");
    return 0;
}
```

2. To generate parent process to write unnamed pipe and will read from it.

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

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

int main(void) {
	int pipefd[2];
	char buffer[5];
	pid_t pid;

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

	// fork the process
	pid = fork();

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

	if (pid == 0) {
	// child process reads from the pipe
	close(pipefd[1]); // close the write end of the pipe

	printf("Child process is reading from the pipe...\n");
	read(pipefd[0], buffer, 5);
	printf("Child process read: %s\n", buffer);

	close(pipefd[0]); // close the read end of the pipe
	_exit(EXIT_SUCCESS);

	} else {
	// parent process writes to the pipe
	close(pipefd[0]); // close the read end of the pipe

	printf("Parent process is writing to the pipe...\n");
	write(pipefd[1], "hello", 5);

	close(pipefd[1]); // close the write end of the pipe
	wait(NULL); // wait for the child process to finish
	exit(EXIT_SUCCESS);
	}
}
```

3. To create a file with hole in it.

```
#include "/home/fymsc57/Downloads/apue.3e/include/apue.h"
#include<fcntl.h>
char buf1[]="Welcome";
char buf2[]="Good Morning";
int main(void)
{
  int fd;
  if((fd=creat("file_hole.txt", O_RDWR))<0)
       printf("Creat error");
  if(write(fd, buf1, 10)!=10)/*Fd Is The File Descriptor,Buf1 Is The Character Array Used To
Hold The Data,The Number Of Bytes To Write From
Buffer*/
             printf("buf1 write error\n");
  if(lseek(fd, 10, SEEK_CUR)== -1)//used to change the location of the read/write pointer of a
file descriptor
       printf("lseek error");
  if(write(fd, buf2, 16) !=16)
       printf("buf2 write error\n");
  exit(0);
}

//hole means create a empty space in the buffer
```

4. Takes multiple files as Command Line Arguments and print their inode numbe

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

int main(int argc, char *argv[]) {
      if (argc < 2) {
      printf("Usage: program file1 file2 ...\n");
      exit(EXIT_FAILURE);
      }

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

      printf("%s has inode number %ld\n", argv[i], (long) st.st_ino);
      }

      return 0;
}
```

5. To handle the two-way communication between parent and child using pipe.

```
#include<stdio.h>
#include<unistd.h>
int main() {
  int pipefds1[2], pipefds2[2];
  int returnstatus1, returnstatus2;
  int pid;
  char pipe1writemessage[20] = "Hi";
  char pipe2writemessage[20] = "Hello";
  char readmessage[20];
  returnstatus1 = pipe(pipefds1);
  if (returnstatus1 == -1) {
        printf("Unable to create pipe 1 \n");
        return 1;
  }
  returnstatus2 = pipe(pipefds2);
  if (returnstatus2 == -1) {
        printf("Unable to create pipe 2 \n");
        return 1;
  }
  pid = fork();

  if (pid != 0){
        close(pipefds1[0]);
        close(pipefds2[1]);
        printf("In Parent: Writing to pipe 1 – Message is %s\n", pipe1writemessage);
        write(pipefds1[1], pipe1writemessage, sizeof(pipe1writemessage));
        read(pipefds2[0], readmessage, sizeof(readmessage));
        printf("In Parent: Reading from pipe 2 – Message is %s\n", readmessage);
  } else {
        close(pipefds1[1]);
        close(pipefds2[0]);
```

```
          read(pipefds1[0], readmessage, sizeof(readmessage));
          printf("In Child: Reading from pipe 1 – Message is %s\n", readmessage);
          printf("In Child: Writing to pipe 2 – Message is %s\n", pipe2writemessage);
          write(pipefds2[1], pipe2writemessage, sizeof(pipe2writemessage));
   }
   return 0;
}
```

6. Print the type of file where file name accepted through Command Line

```
#include "/home/fymsc57/Downloads/apue.3e/include/apue.h"
#include<sys/stat.h>
int main(int argc, char *argv[])
{
    int i;
    struct stat buf;
    char *ptr;
    for(i=1;i<argc;i++)
    {
           printf("%s: ",argv[i]);
           if(lstat(argv[i], &buf)<0)
           {
                     printf("lstat error");
                     continue;
           }
           if(S_ISREG(buf.st_mode))
                     ptr="regular";
           else if(S_ISDIR(buf.st_mode))
                     ptr="directory";
           else if(S_ISCHR(buf.st_mode))
                     ptr="char special";
           else if(S_ISBLK(buf.st_mode))
                     ptr="block special";
           else if(S_ISFIFO(buf.st_mode))
                     ptr="fifo";
           else if(S_ISLNK(buf.st_mode))
                     ptr="symbolic link";
           else if(S_ISSOCK(buf.st_mode))
                     ptr="socket";
           else
                     ptr="***unkown mode***";
           printf("%s\n", ptr);
    }
    exit(0);
}
```

7. To demonstrate the use of atexit() function.

```
#include "/home/fymsc57/Downloads/apue.3e/include/apue.h"
static void my_exit1(void);
static void my_exit2(void);
int main()
{
   if(atexit(my_exit2)!=0)
          printf("can't register my_exit2");
   if(atexit(my_exit1)!=0)
          printf("can't register my_exit1");
   if(atexit(my_exit1)!=0)
          printf("can't register my_exit1");
   printf("main is done\n");
   return 0;
}
static void my_exit1()
{
   printf("first exit handler\n");
}
static void my_exit2()
{
   printf("second exit handler\n");
}
```

8. Open a file goes to sleep for 15 seconds before terminating

```
#include "/home/fymsc57/Downloads/apue.3e/include/apue.h"
#include <sys/types.h>
#include<fcntl.h>

int main(void)
{
   if(open("/home/fymsc57/AOS/Assignment_2/first.c", O_RDWR)<0)
   {
          printf("open error");
   }
   sleep(15);
   printf("\ndone");
   exit(0);
}
```

9. To print the size of the file

```
#include <stdio.h>
long int findSize()
{

    FILE* fp = fopen("/home/fymsc57/AOS/Assignment_2/first.c", "r");
    if (fp == NULL)
    {
            printf("File Not Found!\n");
            return -1;
    }
    fseek(fp, 0, SEEK_END);// The fseek() function changes the read/write position of the file specified by fp
    long int res = ftell(fp);//This function is used to get the total size of file after moving the file pointer at the end of the file
    fclose(fp);
    return res;
}
int main()
{
    long int res = findSize();
    if (res != -1)
            printf("Size of the file is %ld bytes \n", res);
    return 0;
}
```

10. Read the current directory and display the name of the files, no of files in current directory.

```
#include<stdio.h>
#include<dirent.h>
int main (void)
{
    struct dirent *de;
    int count=0;
    DIR *dr;
    dr=opendir(".");
    if(dr==NULL)
    {
            printf("Could not open the current directory");
            return 0;
    }
    while((de=readdir(dr))!=NULL)
    {
            count=count+1;
```

```
          printf("%s\n",de->d_name);
   }
   closedir(dr);
   printf("Number Of Files In current Directory:%d", count);
   return 0;
}
```

11. Write a C program to implement the following unix/linux command (use fork, pipe and exec system call)

ls –l | wc –l

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

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

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

        pid = fork();

        if (pid == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
        } else if (pid == 0) {
        // Child process
        close(pipefd[0]); // Close the read end of the pipe
        dup2(pipefd[1], STDOUT_FILENO); // Redirect stdout to the write end of the pipe
        execlp("ls", "ls", "-l", NULL); // Execute "ls -l"
        perror("execlp");
        exit(EXIT_FAILURE);
        } else {
        // Parent process
        close(pipefd[1]); // Close the write end of the pipe
        dup2(pipefd[0], STDIN_FILENO); // Redirect stdin to the read end of the pipe
        execlp("wc", "wc", "-l", NULL); // Execute "wc -l"
        perror("execlp");
        exit(EXIT_FAILURE);
        }
        return 0;
}
```

Class Code

```
#include<stdio.h>
#include<dirent.h>
int main (void)
{
   struct dirent *de;
   int count=0;
   DIR *dr;
   dr=opendir(".");
   if(dr==NULL)
   {
          printf("Could not open the current directory");
          return 0;
   }
   while((de=readdir(dr))!=NULL)
   {
          count=count+1;
          printf("%s\n",de->d_name);
   }
   closedir(dr);
   printf("Number Of Files In current Directory:%d", count);
   return 0;
}
```

12. Write a C program to 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() {
        struct dirent *de;
        DIR *dir;
        struct stat sb;
        char month[10];
        int month_num;

        // Get the month from the user
```

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

// Convert the month to a number (0-11)
if (strcmp(month, "Jan") == 0) {
month_num = 0;
} else if (strcmp(month, "Feb") == 0) {
month_num = 1;
} else if (strcmp(month, "Mar") == 0) {
month_num = 2;
} else if (strcmp(month, "Apr") == 0) {
month_num = 3;
} else if (strcmp(month, "May") == 0) {
month_num = 4;
} else if (strcmp(month, "Jun") == 0) {
month_num = 5;
} else if (strcmp(month, "Jul") == 0) {
month_num = 6;
} else if (strcmp(month, "Aug") == 0) {
month_num = 7;
} else if (strcmp(month, "Sep") == 0) {
month_num = 8;
} else if (strcmp(month, "Oct") == 0) {
month_num = 9;
} else if (strcmp(month, "Nov") == 0) {
month_num = 10;
} else if (strcmp(month, "Dec") == 0) {
month_num = 11;
} else {
printf("Invalid month\n");
exit(EXIT_FAILURE);
}

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

if (dir == NULL) {
perror("opendir");
exit(EXIT_FAILURE);
}

// Loop through all the files in the directory
while ((de = readdir(dir)) != NULL) {
if (stat(de->d_name, &sb) == -1) {
    perror("stat");
    exit(EXIT_FAILURE);
    }

    // Check if the file was created in the specified month
    struct tm *tm = localtime(&sb.st_ctime);
    if (tm->tm_mon == month_num) {
    printf("%s\n", de->d_name);
    }
    }

    // Close the directory
    closedir(dir);

    return 0;
}
```

13. Write a C program to 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>

int main() {
    struct dirent *de;
    DIR *dir;
    struct stat sb;
    long int size_limit;

    // Get the size limit from the user
    printf("Enter the minimum file size (in bytes): ");
    scanf("%ld", &size_limit);

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

    if (dir == NULL) {
    perror("opendir");
    exit(EXIT_FAILURE);
    }

    // Loop through all the files in the directory
    while ((de = readdir(dir)) != NULL) {
    if (stat(de->d_name, &sb) == -1) {
    perror("stat");
    exit(EXIT_FAILURE);
    }

    // Check if the file size is greater than the specified limit
    if (sb.st_size > size_limit) {
    printf("%s\n", de->d_name);
    }
    }

    // Close the directory
    closedir(dir);

    return 0;
}
```

14.Write a C program to implement the following unix/linux command

i. ls –l > output.txt

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

int main() {
    pid_t pid;
    int fd;

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

    if (pid == -1) {
    perror("fork");
    exit(EXIT_FAILURE);
    }
    else if (pid == 0) {
    // Child process: redirect output to a file
    fd = open("output.txt", O_CREAT | O_WRONLY | O_TRUNC, S_IRUSR | S_IWUSR |
S_IRGRP | S_IROTH);
```

```
        if (fd == -1) {
        perror("open");
        exit(EXIT_FAILURE);
        }

        // Redirect stdout to the file
        dup2(fd, STDOUT_FILENO);

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

        // If execlp returns, there was an error
        perror("execlp");
        exit(EXIT_FAILURE);
        }
        else {
        // Parent process: wait for child to finish
        wait(NULL);
        }

        return 0;
}
```

15.Write a C program which display the information of a given file similar to given by the unix / linux command ls –l <file name>

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

int main(int argc, char *argv[]) {
        struct stat sb;
        struct passwd *pw;
        struct group *gr;
        char *file_mode;
        char date_string[256];

        // Check if a filename was provided
        if (argc != 2) {
```

```
    fprintf(stderr, "Usage: %s <file>\n", argv[0]);
    exit(EXIT_FAILURE);
    }

    // Get the file information
    if (stat(argv[1], &sb) == -1) {
    perror("stat");
    exit(EXIT_FAILURE);
    }

    // Get the file mode as a string
    file_mode = (char *) malloc(11 * sizeof(char));
    snprintf(file_mode, 11, "%o", sb.st_mode & 0777);

    // Get the user and group names
    pw = getpwuid(sb.st_uid);
    gr = getgrgid(sb.st_gid);

    // Convert the last modification time to a string
    strftime(date_string, 256, "%b %d %H:%M", localtime(&sb.st_mtime));

    // Print the file information
    printf("%s %ld %s %s %ld %s %s\n", file_mode, sb.st_nlink, pw->pw_name,
gr->gr_name, sb.st_size, date_string, argv[1]);

    free(file_mode);

    return 0;
}
```

16. Write a C program that behaves like a shell (command interpreter). It has its own prompt say “NewShell$”. Any normal shell command is executed from your shell by starting a child process to execute the system program corresponding to the command. It should additionally interpret the following command.
i) count c <filename> - print number of characters in file
ii) count w <filename> - print number of words in file
iii) count l <filename> - print number of lines in file

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

```
#define MAX_LINE 80
#define MAX_ARGS 10

int main() {
    char *args[MAX_ARGS];
    char *line = NULL;
    size_t len = 0;
    ssize_t read;
    int should_run = 1;

    while (should_run) {
    printf("NewShell$ ");
    fflush(stdout);

    // Read a line of input
    if ((read = getline(&line, &len, stdin)) == -1) {
    // End of file (Ctrl-D) or error
    should_run = 0;
    continue;
    }

    // Strip the newline character from the end of the line
    line[strcspn(line, "\n")] = '\0';

    // Split the line into arguments
    char *arg = strtok(line, " ");
    int i = 0;

    while (arg != NULL) {
    args[i++] = arg;

    if (i == MAX_ARGS - 1) {
        break;
    }

    arg = strtok(NULL, " ");
    }

    args[i] = NULL;

    // Check for built-in commands
    if (strcmp(args[0], "exit") == 0) {
    should_run = 0;
```

```
} else if (strcmp(args[0], "count") == 0 && i == 3) {
// Handle the count command
FILE *fp = fopen(args[2], "r");

if (fp == NULL) {
    printf("Error: Could not open file\n");
} else {
    int count = 0;

    if (strcmp(args[1], "c") == 0) {
    // Count the characters in the file
    fseek(fp, 0L, SEEK_END);
    count = ftell(fp);
    } else if (strcmp(args[1], "w") == 0) {
    // Count the words in the file
    int in_word = 0;
    char c;

    while ((c = fgetc(fp)) != EOF) {
    if (isspace(c)) {
        in_word = 0;
    } else if (!in_word) {
        in_word = 1;
        count++;
    }
    }
    } else if (strcmp(args[1], "l") == 0) {
    // Count the lines in the file
    char c;

    while ((c = fgetc(fp)) != EOF) {
    if (c == '\n') {
        count++;
    }
    }
    } else {
    printf("Error: Invalid count option\n");
    }

    fclose(fp);

    if (count > 0) {
    printf("%d\n", count);
    }
    }
    } else {
    // Run the command as a child process
    pid_t pid = fork();

    if (pid == 0) {
        // Child process
        if (execvp(args[0], args) == -1) {
        printf("Error: Could not execute command\n");
        exit(EXIT_FAILURE);
        }
    } else if (pid < 0) {
        printf("Error: Could not fork process\n");
    } else {
        // Parent process
        int status;
        waitpid(pid, &status, 0);
    }
    }
    }

    free(line);

    return 0;
}
```

17. Write a C program that behaves like a shell (command interpreter). It has its own prompt say “NewShell$”. Any normal shell command is executed from your shell by starting a child process to execute the system program corresponding to the command. It should additionally interpret the following command.
i) list f <dirname> - print name of all files in directory
ii) list n <dirname> - print number of all entries
iii) list i<dirname> - print name and inode of all files

18. Write a C program that behaves like a shell (command interpreter). It has its own prompt say “NewShell$”. Any normal shell command is executed from your shell by starting a child process to execute the system program corresponding to the command. It should additionally interpret the following command.
i) typeline +10 <filename> - print first 10 lines of file
ii) typeline -20 <filename> - print last 20 lines of file
iii) typeline a <filename> - print all lines of file

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

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

#define MAX_COMMAND_LENGTH 100
#define MAX_TOKENS 10

void type_line(char* filename, int start_line, int end_line) {
        FILE* file = fopen(filename, "r");

        if (!file) {
        printf("Failed to open file '%s'\n", filename);
        return;
        }

        char buffer[1024];
        int line_number = 1;

        while (fgets(buffer, sizeof(buffer), file)) {
        if (line_number >= start_line && line_number <= end_line) {
        printf("%s", buffer);
        }

        if (line_number > end_line) {
        break;
        }

        line_number++;
        }

        fclose(file);
}

int main() {
        char command[MAX_COMMAND_LENGTH];

        while (1) {
        printf("NewShell$ ");
        fgets(command, MAX_COMMAND_LENGTH, stdin);

        if (strlen(command) <= 1) {
        continue;
        }
```

```
// Remove newline character from the end of the command
command[strcspn(command, "\n")] = 0;

char* token;
char* tokens[MAX_TOKENS];
int token_count = 0;

token = strtok(command, " ");
while (token != NULL) {
tokens[token_count] = token;
token_count++;

token = strtok(NULL, " ");
}

tokens[token_count] = NULL;

if (strcmp(tokens[0], "exit") == 0) {
break;
}

if (strcmp(tokens[0], "typeline") == 0) {
if (token_count != 3) {
        printf("Invalid arguments\n");
        continue;
}

char* filename = tokens[2];
int line_count;

if (tokens[1][0] == '+') {
        line_count = atoi(tokens[1] + 1);
        type_line(filename, 1, line_count);
} else if (tokens[1][0] == '-') {
        line_count = atoi(tokens[1] + 1);
        type_line(filename, -line_count + 1, 0);
} else if (strcmp(tokens[1], "a") == 0) {
        type_line(filename, 1, INT_MAX);
} else {
        printf("Invalid argument: %s\n", tokens[1]);
}

continue;
```

```
        }

        int pid = fork();

        if (pid == -1) {
        printf("Failed to fork\n");
        exit(1);
        } else if (pid == 0) {
        // Child process
        int file_descriptor;

        if (tokens[token_count - 2] != NULL && strcmp(tokens[token_count - 2], ">") == 0) {
                // Redirect output to a file
                file_descriptor = open(tokens[token_count - 1], O_CREAT | O_WRONLY |
O_TRUNC, 0644);

                if (file_descriptor == -1) {
                printf("Failed to open file '%s'\n", tokens[token_count - 1]);
                exit(1);
                }

                dup2(file_descriptor, STDOUT_FILENO);
                close(file_descriptor);

                tokens[token_count - 2] = NULL;
                tokens[token_count - 1] = NULL;
        }

        if (execvp(tokens[0], tokens) == -1) {
                printf("Failed to execute command\n");
                exit(1);
        }
```

19. Write a C program that behaves like a shell (command interpreter). It has its own prompt say “NewShell$”.Any normal shell command is executed from your shell by starting a child process to execute the system program corresponding to the command. It should

i) additionally interpret the following command.

ii) search f <pattern> <filename> - search first occurrence of pattern in filename

iii) search c <pattern> <filename> - count no. of occurrences of pattern in filename

iv) search a <pattern> <filename> - search all occurrences of pattern in filename

20. 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>

int compare(const void* a, const void* b) {
        struct stat* stat_a = (struct stat*)a;
        struct stat* stat_b = (struct stat*)b;
        return (int)(stat_a->st_size - stat_b->st_size);
}

int main(int argc, char* argv[]) {
        int num_files = argc - 1;
        struct stat* file_stats = (struct stat*)malloc(num_files * sizeof(struct stat));
        if (file_stats == NULL) {
        printf("Error: could not allocate memory for file stats.\n");
        return 1;
        }

        for (int i = 0; i < num_files; i++) {
        if (stat(argv[i+1], &file_stats[i]) != 0) {
        printf("Error: could not get file stats for %s.\n", argv[i+1]);
        return 1;
        }
        }

        qsort(file_stats, num_files, sizeof(struct stat), compare);

        for (int i = 0; i < num_files; i++) {
        printf("%s\n", argv[file_stats[i].st_ino]);
        }

        free(file_stats);
        return 0;
}
```

.21. 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!!!”.

```
// C program to implement sighup(), sigint()
// and sigquit() signal functions
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

// function declaration
void sighup();
void sigint();
void sigquit();

// driver code
void main()
{
   int pid;

   /* get child process */
   if ((pid = fork()) < 0) {
        perror("fork");
        exit(1);
   }

   if (pid == 0) { /* child */
        signal(SIGHUP, sighup);
        signal(SIGINT, sigint);
        signal(SIGQUIT, sigquit);
        for (;;)
               ; /* loop for ever */
   }

   else /* parent */
   { /* pid hold id of child */
        printf("\nPARENT: sending SIGHUP\n\n");
        kill(pid, SIGHUP);

        sleep(3); /* pause for 3 secs */
        printf("\nPARENT: sending SIGINT\n\n");
        kill(pid, SIGINT);

        sleep(3); /* pause for 3 secs */
        printf("\nPARENT: sending SIGQUIT\n\n");
        kill(pid, SIGQUIT);
        sleep(3);
   }
}

// sighup() function definition
void sighup()

{
   signal(SIGHUP, sighup); /* reset signal */
   printf("CHILD: I have received a SIGHUP\n");
}

// sigint() function definition
void sigint()

{
   signal(SIGINT, sigint); /* reset signal */
   printf("CHILD: I have received a SIGINT\n");
}

// sigquit() function definition
void sigquit()
{
   printf("My DADDY has Killed me!!!\n");
   exit(0);
}
```

22. 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/wait.h>

int main() {
    int fds[2]; // file descriptors for the pipe
    pid_t pid1, pid2; // process IDs for the child processes
    int status;

    // create the pipe
    if (pipe(fds) == -1) {
    perror("pipe failed");
    exit(1);
    }

    // block SIGINT and SIGQUIT signals during execution
    sigset_t mask;
    sigemptyset(&mask);
    sigaddset(&mask, SIGINT);
    sigaddset(&mask, SIGQUIT);
    sigprocmask(SIG_BLOCK, &mask, NULL);

    // fork the first child process to execute "ls -l"
    if ((pid1 = fork()) == -1) {
    perror("fork failed");
    exit(1);
    } else if (pid1 == 0) { // child process 1
    close(fds[0]); // close read end of pipe
    dup2(fds[1], STDOUT_FILENO); // redirect stdout to write end of pipe
    execlp("ls", "ls", "-l", NULL); // execute "ls -l"
    perror("execlp for ls failed");
    exit(1);
    }

    // fork the second child process to execute "wc -l"
    if ((pid2 = fork()) == -1) {
    perror("fork failed");
    exit(1);
    } else if (pid2 == 0) { // child process 2
    close(fds[1]); // close write end of pipe
    dup2(fds[0], STDIN_FILENO); // redirect stdin to read end of pipe
    execlp("wc", "wc", "-l", NULL); // execute "wc -l"
    perror("execlp for wc failed");
    exit(1);
    }

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

    // wait for both child processes to finish
    waitpid(pid1, &status, 0);
    waitpid(pid2, &status, 0);

    // unblock SIGINT and SIGQUIT signals
    sigprocmask(SIG_UNBLOCK, &mask, NULL);

    return 0;
}
```

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

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

int main() {
    int fd;
    char *filename = "output.txt";

    // open the output file for writing
    if ((fd = open(filename, O_WRONLY | O_CREAT | O_TRUNC, 0666)) == -1) {
    perror("open failed");
    exit(1);
    }

    // redirect stdout to the output file
    if (dup2(fd, STDOUT_FILENO) == -1) {
    perror("dup2 failed");
    exit(1);
    }
    // close the file descriptor for the output file
    close(fd);

    // print some output to stdout (which will be redirected to the file)
    printf("This is a test output\n");

    // restore stdout to its original state
    if (dup2(STDOUT_FILENO, fd) == -1) {
    perror("dup2 failed");
    exit(1);
    }

    return 0;
}
```

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

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

int main() {
    int pipefd[2];
    pid_t pid;
    char *ls_args[] = {"ls", "-l", NULL};
    char *grep_args[] = {"grep", "txt", NULL};

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

    // fork a child process to run the first command (ls)
    if ((pid = fork()) == -1) {
    perror("fork failed");
    exit(1);
    } else if (pid == 0) {
    // child process: redirect stdout to the write end of the pipe
    close(pipefd[0]);
    dup2(pipefd[1], STDOUT_FILENO);
    close(pipefd[1]);
    // execute the ls command
    execvp(ls_args[0], ls_args);
    perror("execvp failed");
    exit(1);
    }

    // fork another child process to run the second command (grep)
    if ((pid = fork()) == -1) {
    perror("fork failed");
    exit(1);
    } else if (pid == 0) {
    // child process: redirect stdin to the read end of the pipe
    close(pipefd[1]);
    dup2(pipefd[0], STDIN_FILENO);
    close(pipefd[0]);

    // execute the grep command
    execvp(grep_args[0], grep_args);
    perror("execvp failed");
    exit(1);
    }

    // parent process: close both ends of the pipe and wait for both child processes to finish
    close(pipefd[0]);
    close(pipefd[1]);
    wait(NULL);
    wait(NULL);

    return 0;
}
```

25. 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 handle_signal(int sig) {
    // do nothing
}

int main() {
    pid_t pid;
    // install a signal handler for SIGTSTP and SIGCONT
    signal(SIGTSTP, handle_signal);
    signal(SIGCONT, handle_signal);

    // fork a child process
    if ((pid = fork()) == -1) {
    perror("fork failed");
    exit(1);
    } else if (pid == 0) {
    // child process: loop and print a message every second
    while (1) {
    printf("I'm the child process, pid=%d\n", getpid());
    sleep(1);
    }
    }

    // parent process: loop and toggle the child process's state every 5 seconds
    while (1) {
    printf("I'm the parent process, pid=%d\n", getpid());
    sleep(5);

    // send a SIGTSTP signal to suspend the child process
    printf("Suspending child process\n");
    kill(pid, SIGTSTP);
    sleep(5);

    // send a SIGCONT signal to resume the child process
    printf("Resuming child process\n");
    kill(pid, SIGCONT);
    }

    return 0;
}
```

26. Write a C program that illustrates inters process communication using shared memory.

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

#define SIZE 1024

int main() {
    int segment_id;
    char *shared_memory;
    struct shmid_ds shmbuffer;
    int segment_size;
    const int shared_segment_size = SIZE;

    // allocate a shared memory segment
    segment_id = shmget(IPC_PRIVATE, shared_segment_size, IPC_CREAT | IPC_EXCL |
S_IRUSR | S_IWUSR);

    // attach the shared memory segment
    shared_memory = (char *) shmat(segment_id, 0, 0);
    printf("shared memory attached at address %p\n", shared_memory);

    // write some data to the shared memory segment
    sprintf(shared_memory, "Hello, world.");

    // detach the shared memory segment
    shmdt(shared_memory);

    // re-attach the shared memory segment to check its size
    shmid_ds shmid_ds;
    shmat(segment_id, 0, SHM_RDONLY);
    shmctl(segment_id, IPC_STAT, &shmid_ds);
    segment_size = shmid_ds.shm_segsz;

    printf("segment size: %d\n", segment_size);

    // remove the shared memory segment
    shmctl(segment_id, IPC_RMID, 0);

    return 0;
}
```