TRIBHUVAN UNIVERSITY

Paschimanchal Campus
Lamachaur 16, Pokhara

Department of Computer Engineering

Lab Report On: Operating System

Submitted By
Name: Anish Karki
Roll No: PAS080BCT007

Submitted To: SRP Sir
Date of submission: 13/08/2025

LAB 1: Write programs using the following system calls of UNIX operating system: fork, exec, getpid, exit, wait, close, stat.

Introduction:

System calls in UNIX provide an interface between user programs and the operating system kernel. Here’s what each one does:

Process Management Calls

- fork() → Creates a new process (child process). The child gets a copy of the parent’s memory space.
- exec() family → Replaces the current process image with a new program.
- getpid() → Returns the process ID of the calling process.
- exit() → Terminates the calling process.
- wait() → Parent process waits until its child process finishes.
- close() → Closes a file descriptor.

ALGORITHM:

1. Start the program.
2. Read the input from the command line.
3. Use fork() system call to create process, getppid() system call used to get the parent process ID and getpid() system call used to get the current process ID
4. execvp() system call used to execute that command given on that command line argument 5. execlp() system call used to execute specified command.
6. Open the directory at specified in command line input.
7. Display the directory contents.
8. Stop the program.

SOURCE CODE:

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

int main(int argc, char *argv[]) {
    if (argc < 3) {
        printf("Usage: %s <file> <command> [args...]\n", argv[0]);
        exit(EXIT_FAILURE);
    }
```

```
    pid_t pid = fork();

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

    if (pid == 0) {
        printf("Child process running command: %s\n", argv[2]);
        execvp(argv[2], &argv[2]);
        perror("execvp failed");
        exit(EXIT_FAILURE);
    } else {
        printf("Parent process (PID %d) waiting for child to finish...\n", getpid());
        wait(NULL);
        printf("Child finished. Now reading file: %s\n", argv[1]);

        FILE *file = fopen(argv[1], "r");
        if (!file) {
            perror("Error opening file");
            exit(EXIT_FAILURE);
        }

        char ch;
        while ((ch = fgetc(file)) != EOF) {
            putchar(ch);
        }

        fclose(file);
    }

    return EXIT_SUCCESS;
}
```

CONCLUSION:
Hence, fork, exec, getpid, exit, wait, close is implemented.

LAB2: To write a program for implementing Directory management using the following system calls of UNIX operating system: opendir, readdir.

ALGORITHM:

1.Start the program.
2. Open the directory at specified in command line input.
3. Display the directory contents.
4. Stop the program

SOURCE CODE:

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

int main(int argc, char *argv[]) {
    DIR *folder;
    struct dirent *entry;
    int count = 0;

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

    folder = opendir(argv[1]);
    if (!folder) {
        perror("Could not open directory");
        return 1;
    }

    printf("\nItems in directory:\n");
    while ((entry = readdir(folder)) != NULL) {
        printf("%s\n", entry->d_name);
        count++;
    }
    closedir(folder);

    printf("\nTotal items: %d\n", count);

    return 0;
}
```

LAB 3: Write a programs to simulate UNIX commands like ls, grep, etc.

Algorithm:

1. Start the program.
2. Read the input through command line.
3. Open the specified file.
4. Options (c & i) are performed.
5. Stop the program.

SOURCE CODE:

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

int main(int argc, char *argv[]) {
    if (argc != 3) {
        printf("Usage: %s <word> <file>\n", argv[0]);
        return 1;
    }

    char *word = argv[1];
    char *filename = argv[2];
    char line[256];

    FILE *file = fopen(filename, "r");
    if (!file) {
        perror("Could not open file");
        return 1;
    }
while (fgets(line, sizeof(line), file)) {
        if (strstr(line, word)) {
            printf("%s", line);
        }
    }

    fclose(file);
    return 0;
}
```

RESULT:

Thus the program to stimulate the UNIX commands was written and successfully executed.

LAB 4: Write a program using the I/O system calls of UNIX operating system (open,read, write, etc)

Algorithm:

1. Start the program.
2. Read the input from user specified file.
3. Write the content of the file to newly created file.
4. Show the file properties (access time, modified time, & etc,.)
5. Stop the program.

SOURCE CODE:

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

int main(int argc, char *argv[]) {
    int fd_in, fd_out;
    char buffer[100];
    ssize_t n;
    struct stat file_info;

    if (argc != 3) {
        printf("Usage: %s <input_file> <output_file>\n", argv[0]);
        return 1;
    }

    fd_in = open(argv[1], O_RDONLY);
    if (fd_in < 0) {
        perror("Error opening input file");
        return 1;
    }

    fd_out = creat(argv[2], 0644);
    if (fd_out < 0) {
        perror("Error creating output file");
        close(fd_in);
        return 1;
    }

    while ((n = read(fd_in, buffer, sizeof(buffer))) > 0) {
```

```
        if (write(fd_out, buffer, n) != n) {
            perror("Error writing to output file");
            close(fd_in);
            close(fd_out);
            return 1;
        }
    }

    if (stat(argv[1], &file_info) == 0) {
        printf("Input file size: %ld bytes\n", file_info.st_size);
    }

    close(fd_in);
    close(fd_out);

    printf("File copied successfully!\n");
    return 0;
}
```

RESULT:

Thus the program to stimulate the UNIX commands was written and successfully executed.

LAB5: (a) Write a program for implementing the FCFS Scheduling algorithm

ALGORITHM:

1. Start the process.
2. Declare the array size.
3. Get the number of elements to be inserted.
4. Select the process that first arrived in the ready queue
5. Make the average waiting the length of next process.
6. Start with the first process from it's selection as above and let other process to be in queue. 7. Calculate the total number of burst time.
8. Display the values.
9. Stop the process.

SOURCE CODE:

```
#include <stdio.h>

int main() {
    int n, i;
    printf("Enter number of processes: ");
    scanf("%d", &n);

    int bt[n], at[n], wt[n], tat[n];
    char pname[n];

    for (i = 0; i < n; i++) {
        pname[i] = 'A' + i;
        printf("\nProcess %c\n", pname[i]);
        printf("Burst Time: ");
        scanf("%d", &bt[i]);
        printf("Arrival Time: ");
        scanf("%d", &at[i]);
    }

    wt[0] = 0;
    for (i = 1; i < n; i++) {
        wt[i] = wt[i-1] + bt[i-1];
    }

    for (i = 0; i < n; i++) {
        tat[i] = wt[i] + bt[i];
    }
```

```
    printf("\nProcess\tBT\tAT\tWT\tTAT\n");
    for (i = 0; i < n; i++) {
        printf("%c\t%d\t%d\t%d\t%d\n", pname[i], bt[i], at[i], wt[i], tat[i]);
    }

    float avg_wt = 0, avg_tat = 0;
    for (i = 0; i < n; i++) {
        avg_wt += wt[i];
        avg_tat += tat[i];
    }
    avg_wt /= n;
    avg_tat /= n;

    printf("\nAverage Waiting Time: %.2f\n", avg_wt);
    printf("Average Turnaround Time: %.2f\n", avg_tat);

    return 0;
}
```

RESULT:

Thus the program for implementing FCFs scheduling algorithm was written and successfully executed.

(b) Write a program for simulation of SJF Scheduling algorithm

ALGORITHM:

1.Start the process.
2. Declare the array size.
3. Get the number of elements to be inserted.
4. Select the process which have shortest burst will execute first.
5. If two process have same burst length then FCFS scheduling algorithm used.
6. Make the average waiting the length of next process.
7. Start with the first process from it's selection as above and let other process to be in queue. 8. Calculate the total number of burst time.
9. Display the values.
10. Stop the process.

SOURCE CODE:

```
#include <stdio.h>

int main() {
    int n, i, j;
    printf("Enter number of processes: ");
    scanf("%d", &n);

    int bt[n], at[n], wt[n], tat[n], temp;
    char pname[n];

    for (i = 0; i < n; i++) {
        pname[i] = 'A' + i;
        printf("\nProcess %c\n", pname[i]);
        printf("Burst Time: ");
        scanf("%d", &bt[i]);
        printf("Arrival Time: ");
        scanf("%d", &at[i]);
    }
    for (i = 0; i < n - 1; i++) {
        for (j = i + 1; j < n; j++) {
            if (bt[i] > bt[j]) {
                temp = bt[i];
                bt[i] = bt[j];
                bt[j] = temp;

                char tname = pname[i];
                pname[i] = pname[j];
```

```
            pname[j] = tname;

            temp = at[i];
            at[i] = at[j];
            at[j] = temp;
        }
    }
}
wt[0] = 0;
for (i = 1; i < n; i++) {
    wt[i] = wt[i - 1] + bt[i - 1];
}

for (i = 0; i < n; i++) {
    tat[i] = wt[i] + bt[i];
}

printf("\nProcess\tBT\tAT\tWT\tTAT\n");
for (i = 0; i < n; i++) {
    printf("%c\t%d\t%d\t%d\t%d\n", pname[i], bt[i], at[i], wt[i], tat[i]);
}

float avg_wt = 0, avg_tat = 0;
for (i = 0; i < n; i++) {
    avg_wt += wt[i];
    avg_tat += tat[i];
}
avg_wt /= n;
avg_tat /= n;

printf("\nAverage Waiting Time: %.2f\n", avg_wt);
printf("Average Turnaround Time: %.2f\n", avg_tat);

return 0;
}
```

RESULT:

Thus the program for implementing SJFs scheduling algorithm was written and successfully executed.

LAB6: Implement the Producer – Consumer problem using semaphores

ALGORITHM:

1. Start the process
2. Initialize buffer size
3. Consumer enters, before that producer buffer was not empty.
4. Producer enters, before check consumer consumes the buffer.
5. Stop the process.

SOURCE CODE:

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

int mutex = 1;
int full = 0;
int empty = 3;
int x = 0;

int wait(int s) {
    return --s;
}

int signal(int s) {
    return ++s;
}

void producer() {
    mutex = wait(mutex);
    empty = wait(empty);
    full = signal(full);
    x++;
    printf("Producer produces item %d\n", x);
    mutex = signal(mutex);
}

void consumer() {
    mutex = wait(mutex);
    full = wait(full);
    empty = signal(empty);
    printf("Consumer consumes item %d\n", x);
    x--;
```

```
    mutex = signal(mutex);
}

int main() {
    int choice;

    while (1) {
        printf("\n1. PRODUCER\n2. CONSUMER\n3. EXIT\nEnter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                if (mutex == 1 && empty != 0)
                    producer();
                else
                    printf("Buffer is FULL\n");
                break;

            case 2:
                if (mutex == 1 && full != 0)
                    consumer();
                else
                    printf("Buffer is EMPTY\n");
                break;

            case 3:
                exit(0);

            default:
                printf("Invalid choice! Try again.\n");
        }
    }

    return 0;
}
```

RESULT:

Thus the program for Implement the Producer – Consumer problem using semaphores was written and successfully executed.