CSA04 – OPERATING SYSTEMS

LIST OF PROGRAMS

1. Create a new process by invoking the appropriate system call. Get the process identifier of the currently running process and its respective parent using system calls and display the same using a C program.

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

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

   if (child_pid < 0) {
      perror("Fork failed");
      return 1;
   }
   if (child_pid == 0) {
      printf("Child Process:\n");
      printf("PID: %d\n", getpid());
      printf("Parent's PID: %d\n", getppid());


   } else {
      printf("Parent Process:\n");
      printf("PID: %d\n", getpid());
 }


   return 0;
}
```

```
Parent process: PID = 1234
Child process: PID = 1235, Parent PID = 1234
```

2. Identify the system calls to copy the content of one file to another and illustrate the same using a C program

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
FILE *fptr1,*fptr2;
char filename[100],c;
printf("Enter the filename to open for reading \n");
scanf("%s", filename);
fptr1 = fopen(filename,"r");
if (fptr1==NULL)
{
printf("Cannot open file %s \n",filename);
exit(0);
}
printf("Enter the filename to open for writing \n");
scanf("%s",filename);
fptr2 = fopen(filename,"w");
if (fptr2 == NULL)
{
printf("Cannot open file %s \n",filename);
exit(0);
}
c = fgetc(fptr1);
while (c!=EOF)
{
fputc(c,fptr2);
c = fgetc(fptr1);
}
printf("\nContents copied to %s",filename);
fclose(fptr1);
fclose(fptr2);
return 0;
```

}

```
$ ./file_copy source.txt destination.txt
File copy successful
```

3. Design a CPU scheduling program with C using First Come First Served technique with the following considerations. a. All processes are activated at time 0. b. Assume that no process waits on I/O devices

```
#include<stdio.h>

int main()
{
int n,bt[20],wt[20],tat[20],i,j; float avwt=0,avtat=0;
printf("Enter total number of processes(maximum 20):");
scanf("%d",&n);
printf("\nEnter Process Burst Time\n");
for(i=0;i<n;i++)
{
printf("P[%d]:",i+1);
scanf("%d",&bt[i]);
}
wt[0]=0;
for(i=1;i<n;i++)
{
wt[i]=0;
for(j=0;j<i;j++)
wt[i]+=bt[j];
}
printf("\nProcess\t\tBurst Time\tWaiting Time\tTurnaround Time");
for(i=0;i<n;i++)
{
 tat[i]=bt[i]+wt[i]; avwt+=wt[i];
 avtat+=tat[i];printf("\nP[%d]\t\t%d\t\t%d\t\t%d",i+1,bt[i],wt[i],tat[i]);
}
 avwt/=i; avtat/=i;
printf("\n\nAverage Waiting Time:%.2f",avwt);
printf("\nAverage Turnaround Time:%.2f",avtat);
```

```
return 0;
}
```

```
Enter total number of processes(maximum 20):3

Enter Process Burst Time
P[1]:4
P[2]:5
P[3]:6

Process         Burst Time      Waiting Time    Turnaround Time
P[1]            4               0               4
P[2]            5               4               9
P[3]            6               9               15

Average Waiting Time:4.33
Average Turnaround Time:9.33
--------------------------------
Process exited after 6.243 seconds with return value 0
Press any key to continue . . .
```

4.Construct a scheduling program with C that selects the waiting process with the smallest execution time to execute next.

```
#include <stdio.h>

struct Process {
    int id;
    int arrival_time;
    int burst_time;
    int completion_time;
};

void swap(struct Process* a, struct Process* b) {
    struct Process temp = *a;
    *a = *b;
    *b = temp;
}

void sortProcesses(struct Process processes[], int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n - i - 1; j++) {
```

```
            if (processes[j].burst_time > processes[j + 1].burst_time) {
                swap(&processes[j], &processes[j + 1]);
            }
        }
    }
}

void calculateCompletionTimes(struct Process processes[], int n) {
    int current_time = 0;
    for (int i = 0; i < n; i++) {
        current_time += processes[i].burst_time;
        processes[i].completion_time = current_time;
    }
}

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

    struct Process processes[n];

    for (int i = 0; i < n; i++) {
        processes[i].id = i + 1;
        printf("Enter arrival time for process %d: ", i + 1);
        scanf("%d", &processes[i].arrival_time);
        printf("Enter burst time for process %d: ", i + 1);
        scanf("%d", &processes[i].burst_time);
    }

    sortProcesses(processes, n);

    calculateCompletionTimes(processes, n);

    printf("Process\tArrival Time\tBurst Time\tCompletion Time\n");
    for (int i = 0; i < n; i++) {
```

```
        printf("%d\t\t%d\t\t%d\t\t%d\n", processes[i].id,
    processes[i].arrival_time, processes[i].burst_time,
    processes[i].completion_time);
      }

      return 0;
    }
```

```
Enter the number of processes: 3
Enter arrival time for process 1: 5
Enter burst time for process 1: 6
Enter arrival time for process 2: 4
Enter burst time for process 2: 2
Enter arrival time for process 3: 6
Enter burst time for process 3: 8
Time 0: Idle
Time 1: Idle
Time 2: Idle
Time 3: Idle
Time 4: Executing process 2
Time 6: Executing process 1
Time 12: Executing process 3

--------------------------------
Process exited after 6.103 seconds with return value 0
Press any key to continue . . .
```

5.Construct a scheduling program with C that selects the waiting process with the highest priority to execute next.

```
#include <stdio.h>

struct Process {
   int id;
  int priority;
   int burst_time;
   int completion_time;
};

void swap(struct Process* a, struct Process* b) {
```

```
    struct Process temp = *a;
    *a = *b;
    *b = temp;
}
void sortProcesses(struct Process processes[], int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (processes[j].priority > processes[j + 1].priority) {
                swap(&processes[j], &processes[j + 1]);
            }   }
    }
}

void calculateCompletionTimes(struct Process processes[], int n) {
    int current_time = 0;
    for (int i = 0; i < n; i++) {
        current_time += processes[i].burst_time;
        processes[i].completion_time = current_time;
    }
}

int main() {
    int n;
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    struct Process processes[n];
```

```
    for (int i = 0; i < n; i++) {
        processes[i].id = i + 1;
        printf("Enter priority for process %d: ", i + 1);
        scanf("%d", &processes[i].priority);
        printf("Enter burst time for process %d: ", i + 1);
        scanf("%d", &processes[i].burst_time);
    }
  sortProcesses(processes, n);
   calculateCompletionTimes(processes, n);
   printf("Process\tPriority\tBurst Time\tCompletion Time\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\t\t%d\n", processes[i].id, processes[i].priority,
processes[i].burst_time, processes[i].completion_time);
    }
    return 0;
```

```
Enter the number of processes: 3
Enter priority for process 1: 44
Enter burst time for process 1: 5
Enter priority for process 2: 3
Enter burst time for process 2: 2
Enter priority for process 3: 4
Enter burst time for process 3: 54
Process Priority        Burst Time      Completion Time
2       3               2               2
3       4               54              56
1       44              5               61

--------------------------------
Process exited after 7.602 seconds with return value 0
Press any key to continue . . .
```

```
}
```

6.Construct a C program to implement pre-emptive priority scheduling algorithm.

```
#include<stdio.h>
struct Process {
   int id;
   int priority;
   int burst_time;
   int remaining_time;
};
void priorityScheduling(struct Process processes[], int n) {
   int currentTime = 0;
   int totalExecutionTime = 0;
   for (int i = 0; i < n; i++) {
      totalExecutionTime += processes[i].burst_time;
   }
   while (currentTime < totalExecutionTime) {
      int highestPriority = -1;
      int selectedProcess = -1;
      for (int i = 0; i < n; i++) {
         if (processes[i].remaining_time > 0 && processes[i].priority >
highestPriority) {
            highestPriority = processes[i].priority;
            selectedProcess = i;
         }
      }
      if (selectedProcess != -1) {
```

```
            printf("Executing process %d at time %d\n",
processes[selectedProcess].id, currentTime);
            processes[selectedProcess].remaining_time--;
            currentTime++;
        } else {
            currentTime++;
        }
    }
}
int main() {
    int n;
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    struct Process processes[n];
    for (int i = 0; i < n; i++) {
        processes[i].id = i + 1;
        printf("Enter burst time for process %d: ", i + 1);
        scanf("%d", &processes[i].burst_time);
        printf("Enter priority for process %d: ", i + 1);
        scanf("%d", &processes[i].priority);
        processes[i].remaining_time = processes[i].burst_time;
    }
    priorityScheduling(processes, n);
    return 0;
}
```

```
Enter the number of processes: 3
Enter burst time for process 1: 4
Enter priority for process 1: 5
Enter burst time for process 2: 3
Enter priority for process 2: 2
Enter burst time for process 3: 5
Enter priority for process 3: 6
Executing process 3 at time 0
Executing process 3 at time 1
Executing process 3 at time 2
Executing process 3 at time 3
Executing process 3 at time 4
Executing process 1 at time 5
Executing process 1 at time 6
Executing process 1 at time 7
Executing process 1 at time 8
Executing process 2 at time 9
Executing process 2 at time 10
Executing process 2 at time 11

--------------------------------
Process exited after 6.099 seconds with return value 0
Press any key to continue . . .
```

7.Construct a C program to implement non-pre emptive SJF algorithm.

```
#include <stdio.h>
struct Process {
    int id;
    int arrival_time;
    int burst_time;
};
void shortestJobFirst(struct Process processes[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (processes[j].burst_time > processes[j + 1].burst_time) {
```

```
            struct Process temp = processes[j];
            processes[j] = processes[j + 1];
            processes[j + 1] = temp;
        }
    }
  }
  int currentTime = 0;
  for (int i = 0; i < n; i++) {
    while (currentTime < processes[i].arrival_time) {
      printf("Time %d: Idle\n", currentTime);
      currentTime++;
    }
    printf("Time %d: Executing process %d\n", currentTime, processes[i].id);
    currentTime += processes[i].burst_time;
  }
}
int main() {
  int n;
  printf("Enter the number of processes: ");
  scanf("%d", &n);
  struct Process processes[n];
  for (int i = 0; i < n; i++) {
    processes[i].id = i + 1;
    printf("Enter arrival time for process %d: ", i + 1);
    scanf("%d", &processes[i].arrival_time);
    printf("Enter burst time for process %d: ", i + 1);
    scanf("%d", &processes[i].burst_time);
```

```
    }
    shortestJobFirst(processes, n);
   return 0;
}
```

```
Enter the number of processes: 3
Enter arrival time for process 1: 4
Enter burst time for process 1: 5
Enter arrival time for process 2: 2
Enter burst time for process 2: 3
Enter arrival time for process 3: 5
Enter burst time for process 3: 2
Time 0: Idle
Time 1: Idle
Time 2: Idle
Time 3: Idle
Time 4: Idle
Time 5: Executing process 3
Time 7: Executing process 2
Time 10: Executing process 1

--------------------------------
Process exited after 5.293 seconds with return value 0
Press any key to continue . . .
```

8. Construct a C program to simulate Round Robin scheduling algorithm with C

```
#include <stdio.h>
struct Process {
    int id;
    int burst_time;
    int remaining_time;
};
void roundRobin(struct Process processes[], int n, int time_quantum) {
    int currentTime = 0;
    while (1) {
```

```
        int allProcessesDone = 1;
        for (int i = 0; i < n; i++) {
            if (processes[i].remaining_time > 0) {
                allProcessesDone = 0;
                int executionTime = (processes[i].remaining_time < time_quantum) ?
processes[i].remaining_time : time_quantum;
                processes[i].remaining_time -= executionTime;
                currentTime += executionTime;
                printf("Time %d: Executing process %d (Remaining Time: %d)\n",
currentTime, processes[i].id, processes[i].remaining_time);
            }
        }
        if (allProcessesDone)
            break;
    }
}

int main() {
    int n;
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    int time_quantum;
    printf("Enter the time quantum for Round Robin: ");
    scanf("%d", &time_quantum);
    struct Process processes[n];
    for (int i = 0; i < n; i++) {
        processes[i].id = i + 1;
        printf("Enter burst time for process %d: ", i + 1);
```

```
        scanf("%d", &processes[i].burst_time);
        processes[i].remaining_time = processes[i].burst_time;
    }
    roundRobin(processes, n, time_quantum);
    return 0;
}
```

```
Enter the number of processes: 3
Enter the time quantum for Round Robin: 2
Enter burst time for process 1: 4
Enter burst time for process 2: 5
Enter burst time for process 3: 1
Time 2: Executing process 1 (Remaining Time: 2)
Time 4: Executing process 2 (Remaining Time: 3)
Time 5: Executing process 3 (Remaining Time: 0)
Time 7: Executing process 1 (Remaining Time: 0)
Time 9: Executing process 2 (Remaining Time: 1)
Time 10: Executing process 2 (Remaining Time: 0)

--------------------------------
Process exited after 6.951 seconds with return value 0
Press any key to continue . . .
```

9.Illustrate the concept of inter-process communication using shared memory with a C program

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#define SHM_SIZE 1024
struct SharedData {
    int counter;
    char message[256];
};
```

```
int main() {
    int shmid;
    key_t key = 1234;
    if ((shmid = shmget(key, sizeof(struct SharedData), IPC_CREAT | 0666)) <
0) {
        perror("shmget");
        exit(EXIT_FAILURE);
    }
    struct SharedData *shared_data = (struct SharedData *)shmat(shmid, NULL,
0);
    if ((int)shared_data == -1) {
        perror("shmat");
        exit(EXIT_FAILURE);
    }
    shared_data->counter = 0;
    snprintf(shared_data->message, sizeof(shared_data->message), "Hello from
the parent process!");


    pid_t child_pid = fork();


    if (child_pid < 0) {
        perror("fork");
        exit(EXIT_FAILURE);
    } else if (child_pid == 0) {
        printf("Child process: Counter = %d, Message = %s\n",
shared_data->counter, shared_data->message);
        shared_data->counter += 10;
        snprintf(shared_data->message, sizeof(shared_data->message), "Hello
from the child process!");
```

```
        shmdt((void *)shared_data);
    } else {
        sleep(1);
        printf("Parent process: Counter = %d, Message = %s\n",
shared_data->counter, shared_data->message);

        shared_data->counter += 5;
        snprintf(shared_data->message, sizeof(shared_data->message), "Hello
from the parent process!");

        wait(NULL);
        shmdt((void *)shared_data);
        shmctl(shmid, IPC_RMID, NULL);
    }

    return 0;
}
```

```
Parent process: Counter = 0, Message = Hello from the parent process!
Child process: Counter = 0, Message = Hello from the parent process!
Parent process: Counter = 15, Message = Hello from the child process!
```

10.Illustrate the concept of inter-process communication using message queue with a C program

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

#include <string.h>
```

```
// Structure to represent a message
struct Message {
    long mtype;      // Message type (must be greater than 0)
    char mtext[256];  // Message text
};

int main() {
    key_t key = ftok("/tmp", 'A');
    int msqid;

    if ((msqid = msgget(key, IPC_CREAT | 0666)) < 0) {
        perror("msgget");
        exit(EXIT_FAILURE);
    }
    pid_t child_pid = fork();

    if (child_pid < 0) {
        perror("fork");
        exit(EXIT_FAILURE);
    } else if (child_pid == 0) {
        struct Message msg;
        if (msgrcv(msqid, &msg, sizeof(msg.mtext), 1, 0) < 0) {
            perror("msgrcv");
            exit(EXIT_FAILURE);
        }
```

```
        printf("Child process received message: %s\n", msg.mtext);
    } else {
        struct Message msg;

        msg.mtype = 1;
        snprintf(msg.mtext, sizeof(msg.mtext), "Hello from the parent process!");

        if (msgsnd(msqid, &msg, sizeof(msg.mtext), 0) < 0) {
            perror("msgsnd");
            exit(EXIT_FAILURE);
        }
        wait(NULL);
        if (msgctl(msqid, IPC_RMID, NULL) < 0) {
            perror("msgctl");
            exit(EXIT_FAILURE);
        }
    }
    return 0;
}
```

```
Parent (Sender) Process sent: Hello from the Parent (Sender) Process!
Child (Receiver) Process received: Hello from the Parent (Sender) Process!
```