Vigneshwar

192465007

CSA0428 Operating Systems for UI Design

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.

**Aim:**

To create a new process using fork() and display the process ID (PID) and parent process ID (PPID).

**Algorithm:**

1. Start the program.
2. Use fork() to create a new process.
3. Use getpid() to get PID.
4. Use getppid() to get PPID.
5. Display PID and PPID for both parent and child processes.
6. End the program.

**Code:**

#include <stdio.h>

#include <unistd.h>

int main() {

int pid = fork();

if (pid == 0) {

printf("Child Process - PID: %d, PPID: %d\n", getpid(), getppid());

} else {

printf("Parent Process - PID: %d, Child PID: %d\n", getpid(), pid);

}

return 0;

}

**Sample Output:**

Parent Process - PID: 1234, Child PID: 1235

Child Process - PID: 1235, PPID: 1234

A screenshot of a computer

AI-generated content may be incorrect.

**RESULT:**

Process creation and PID display program executed successfully.

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

**AIM:**

To illustrate file copying using system calls in Linux.

**ALGORITHM:**

1. Open source file in read-only mode
2. Open/create destination file in write mode
3. Read from source and write to destination until EOF
4. Close both files

**CODE:**

#include <stdio.h>

#include <string.h>

int main() {

char source[1000], dest[1000];

printf("Enter source content: ");

fgets(source, sizeof(source), stdin);

strcpy(dest, source); // Simulate file copy

printf("Copied content: %s", dest);

return 0;

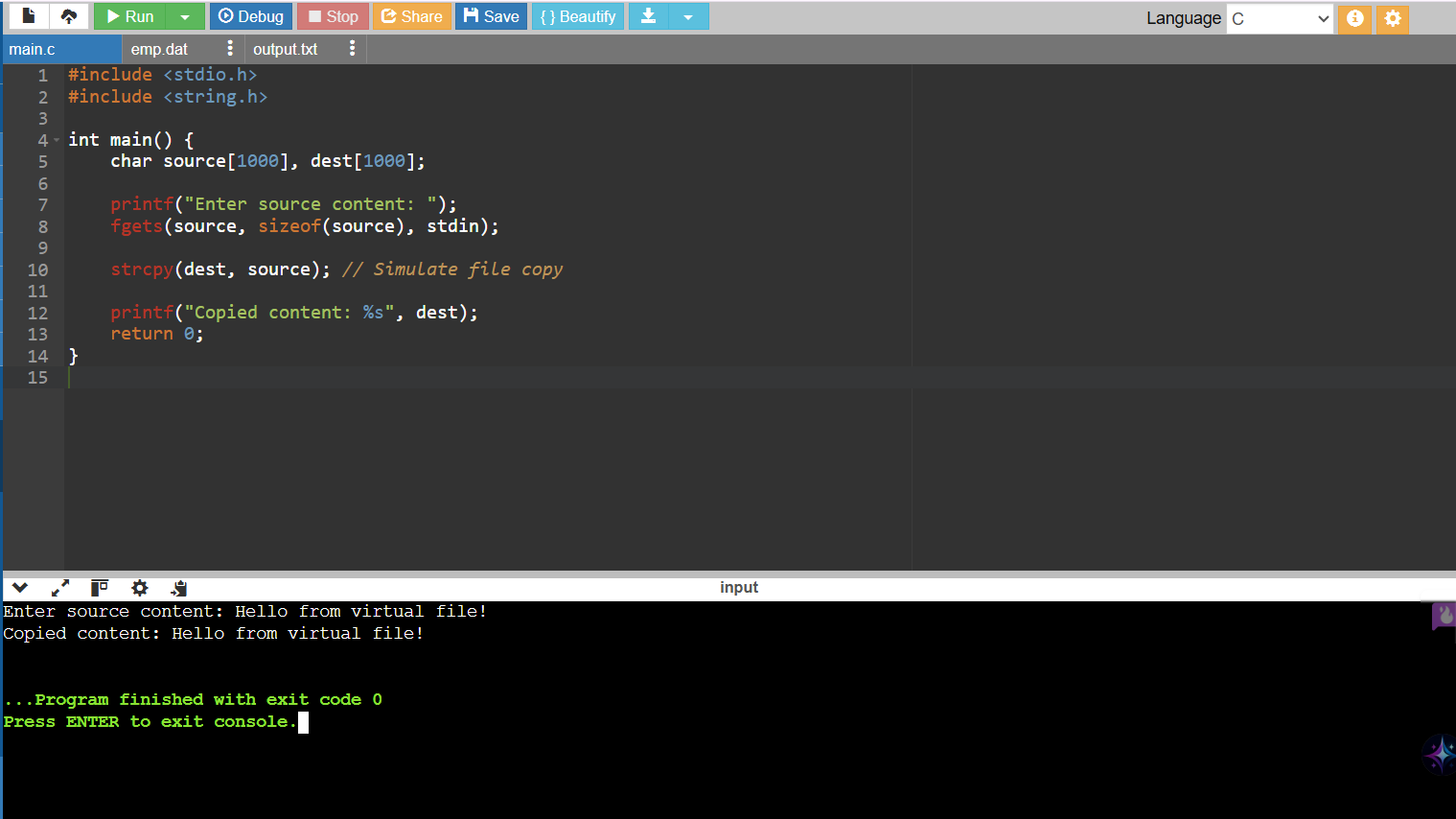
}

**SAMPLE INPUT:**

Enter source content: Hello from virtual file!

**SAMPLE OUTPUT:**

Copied content: Hello from virtual file!

****

**RESULT:**

File copy using system calls executed successfully.

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.

**Aim:**

To implement First Come First Served (FCFS) CPU scheduling algorithm in C.

**Algorithm:**

1. Input number of processes and burst times.
2. Calculate waiting time and turnaround time for each process.
3. Display average waiting time and turnaround time.

**Code:**

#include <stdio.h>

int main() {

int n, i;

printf("Enter number of processes: ");

scanf("%d", &n);

int bt[n], wt[n], tat[n];

printf("Enter burst times:\n");

for(i = 0; i < n; i++)

scanf("%d", &bt[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("P\tBT\tWT\tTAT\n");

for(i = 0; i < n; i++)

printf("%d\t%d\t%d\t%d\n", i+1, bt[i], wt[i], tat[i]);

return 0;

}

**Sample Input:**

3

5 8 12

**Sample Output:**

P BT WT TAT

1 5 0 5

2 8 5 13

3 12 13 25

A screenshot of a computer

AI-generated content may be incorrect.  
**RESULT:**

FCFS CPU scheduling program executed successfully.

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

**Aim:**

To implement non-preemptive Shortest Job First (SJF) scheduling algorithm.

**Algorithm:**

1. Input number of processes and burst times
2. Sort processes based on burst times
3. Calculate waiting time and turnaround time.
4. Display results.

**Code:**

#include <stdio.h>

int main() {

int n, i, j, temp;

printf("Enter number of processes: ");

scanf("%d", &n);

int bt[n], p[n];

for(i = 0; i < n; i++) {

printf("Enter burst time for P%d: ", i+1);

scanf("%d", &bt[i]);

p[i] = i+1;

}

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;

temp = p[i]; p[i] = p[j]; p[j] = temp;

}

int wt = 0, tat = 0, total\_wt = 0, total\_tat = 0;

printf("P\tBT\tWT\tTAT\n");

for(i = 0; i < n; i++) {

tat = wt + bt[i];

printf("P%d\t%d\t%d\t%d\n", p[i], bt[i], wt, tat);

total\_wt += wt;

total\_tat += tat;

wt = tat;

}

return 0;

}

**SAMPLE INPUT:**

Enter number of processes:4

Enter burst time for P1:6

Enter burst time for P2:8

Enter burst time for P3:7

Enter burst time for P4:3

**SAMPLE OUTPUT:**

P BT WT TAT

P4 3 0 3

P1 6 3 9

P3 7 9 16

P2 8 16 24

**A screenshot of a computer

AI-generated content may be incorrect.**

**RESULT:**

SJF scheduling program executed successfully.

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

**Aim:**

To implement non-preemptive Priority Scheduling in C.

**Algorithm:**

1. Input processes with burst time and priority.
2. Sort processes based on priority.
3. Calculate waiting time, turnaround time.
4. Display results.

**Code:**

#include <stdio.h>

struct Process {

int pid, bt, pr;

};

int main() {

struct Process p[10];

int n, i, j;

printf("No. of processes: ");

scanf("%d", &n);

for (i = 0; i < n; i++) {

printf("P%d Burst Priority: ", i + 1);

p[i].pid = i + 1;

scanf("%d %d", &p[i].bt, &p[i].pr);

} for (i = 0; i < n-1; i++)

for (j = i+1; j < n; j++)

if (p[i].pr > p[j].pr) {

struct Process temp = p[i]; p[i] = p[j]; p[j] = temp;

}

int wt = 0, tat;

printf("\nProcess\tBT\tPriority\tWT\tTAT\n");

for (i = 0; i < n; i++) {

tat = wt + p[i].bt;

printf("P%d\t%d\t%d\t\t%d\t%d\n", p[i].pid, p[i].bt, p[i].pr, wt, tat);

wt = tat;

}

}

**Sample Input:**

No. of processes: 3

P1 Burst Priority: 10 2

P2 Burst Priority: 5 1

P3 Burst Priority: 8 3

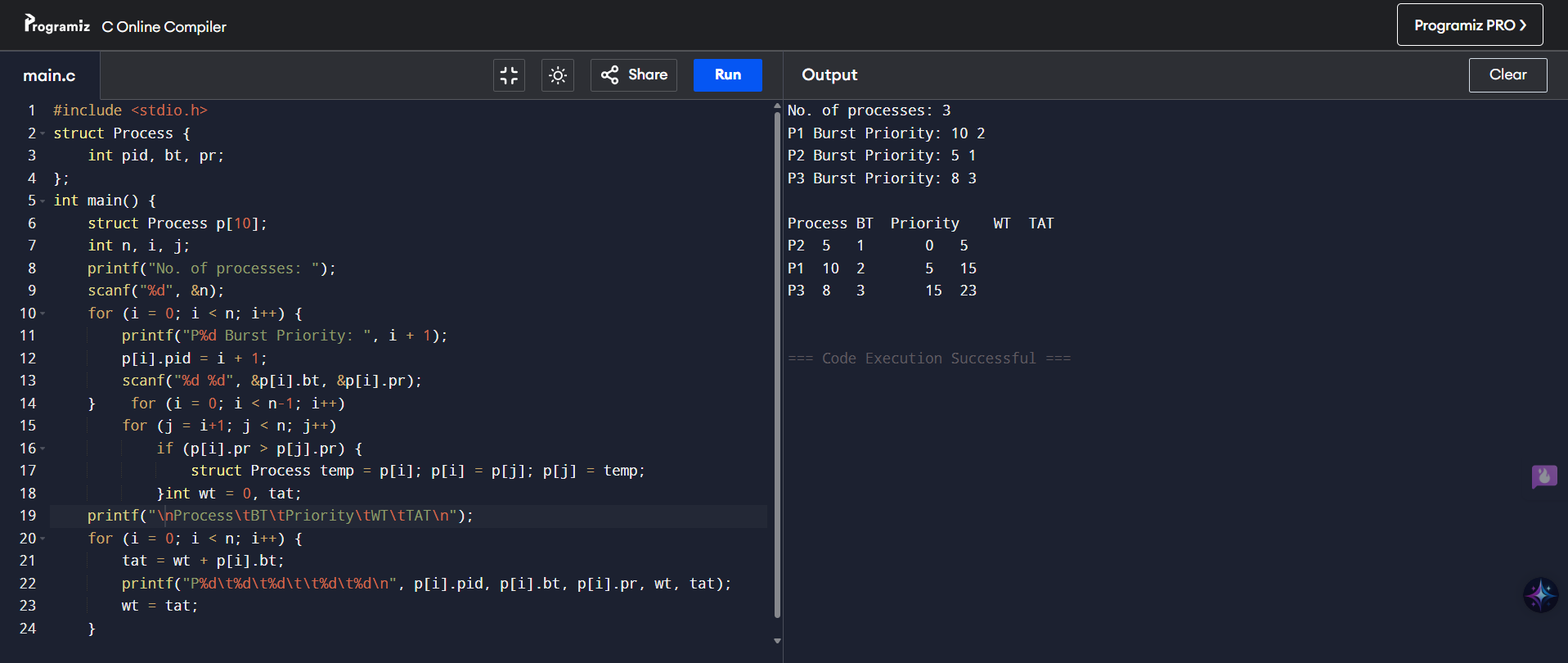
**Sample Output:**

Process BT Priority WT TAT

P2 5 1 0 5

P1 10 2 5 15

P3 8 3 15 23\

****

**RESULT:**

Priority scheduling (non-preemptive) executed successfully.

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

**Aim:**

To implement Preemptive Priority Scheduling algorithm in C.

**Algorithm:**

1. Input processes with arrival time, burst time, and priority.
2. At each unit of time, pick the process with highest priority.
3. If a new process arrives with higher priority, preempt the current process.
4. Calculate Waiting Time (WT) & Turn Around Time (TAT).
5. Display results.

**Code:**

#include <stdio.h>

struct P { int id, bt, pr, rt; } p[10];

int main() {

int n, done = 0;

printf("No. of processes: "); scanf("%d", &n);

for (int i = 0; i < n; i++) {

printf("P%d BT Priority: ", i+1);

p[i].id = i+1; scanf("%d %d", &p[i].bt, &p[i].pr);

p[i].rt = p[i].bt;

}

printf("\nExecution: ");

while (done < n) {

int pos = -1, min = 999;

for (int i=0;i<n;i++)

if (p[i].rt > 0 && p[i].pr < min) min = p[i].pr, pos = i;

if (pos == -1) break;

printf("P%d ", p[pos].id);

p[pos].rt--;

if (p[pos].rt == 0) done++;

}

}

**Sample Input:**

No. of processes: 3

P1 BT Priority: 4 2

P2 BT Priority: 2 1

P3 BT Priority: 3 3

**Sample Output:**

Execution: P2 P2 P1 P1 P1 P1 P3 P3 P3

**A screenshot of a computer

AI-generated content may be incorrect.**

**RESULT:**

Non-preemptive SJF scheduling executed successfully.

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

**AIM:**

To write a C program to implement Non-Preemptive SJF CPU Scheduling.

**Procedure**

1. Enter number of processes and their burst times.
2. Sort processes by burst time (smallest first).
3. Calculate Waiting Time (WT):
4. → First WT = 0, next = previous WT + previous BT.
5. Calculate Turnaround Time (TAT) = WT + BT.
6. Display Process | BT | WT | TAT.
7. End.

**Code**

#include <stdio.h>

int main() {

int n, bt[10], wt[10]={0}, tat[10], i, j, temp;

printf("No. of processes: "); scanf("%d", &n);

for (i=0; i<n; i++) { printf("BT of P%d: ", i+1); scanf("%d", &bt[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; }

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

printf("\nP\tBT\tWT\tTAT\n");

for (i=0; i<n; i++) { tat[i]=wt[i]+bt[i]; printf("P%d\t%d\t%d\t%d\n", i+1, bt[i], wt[i], tat[i]); }

}

**Sample Input:**

No. of processes: 3

BT of P1: 5

BT of P2: 2

BT of P3: 8

**Sample Output:**

P BT WT TAT

P1 2 0 2

P2 5 2 7

P3 8 7 15

A screenshot of a computer

AI-generated content may be incorrect.

**RESULT:**

Non-preemptive SJF scheduling executed successfully.

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

**Aim:**

To implement Round Robin Scheduling algorithm in C.

**Algorithm:**

1. Input processes and burst times.
2. Input time quantum.
3. Execute processes for time quantum repeatedly in a circular manner until completion.
4. Calculate WT & TAT.

**Code**

#include <stdio.h>

int main() {

int n, bt[10], rem[10], tq, time = 0, done;

scanf("%d", &n);

for (int i = 0; i < n; i++) scanf("%d", &bt[i]), rem[i] = bt[i];

scanf("%d", &tq);

while (1) {

done = 1;

for (int i = 0; i < n; i++)

if (rem[i] > 0) {

done = 0;

if (rem[i] > tq) rem[i] -= tq, time += tq, printf("P%d ", i+1);

else time += rem[i], rem[i] = 0, printf("P%d ", i+1);

}

if (done) break;

}

printf("\nTotal Time: %d\n", time);

}

**Sample Input:**

3

5 9 6

3

**Sample Output:**

P1 P2 P3 P1 P2 P3 P2

Total Time: 20



**RESULT:**

Round Robin scheduling program executed successfully.

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

**Aim:**

To demonstrate IPC using shared memory with producer & consumer processes.

**Algorithm:**

1. Create a shared memory segment.
2. Attach shared memory to both parent and child processes.
3. Parent writes data; child reads it.
4. Detach & delete shared memory after communication.

**Code**

#include <stdio.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#include <string.h>

#include <unistd.h>

int main() {

int shmid = shmget(IPC\_PRIVATE, 100, IPC\_CREAT | 0666);

char \*str = (char \*)shmat(shmid, NULL, 0)

if (fork() == 0) {

sleep(3);

printf("Child reads: %s\n", str);

} else {

printf("Parent writes: ");

fgets(str, 100, stdin);

wait(NULL);

shmdt(str);

shmctl(shmid, IPC\_RMID, NULL);

}

return 0;

}

**Sample Input**

Parent writes: Hello from parent

**Sample Output**

Child reads: Hello from parent

A screenshot of a computer

AI-generated content may be incorrect.

**RESULT:**

Shared memory IPC program executed successfully.

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

**Aim:**

To demonstrate IPC using message queue in C.

**Algorithm:**

1. Create a message queue.
2. Parent sends message; child receives it.
3. Display received message.
4. Remove message queue after communication.

**Code**

#include <stdio.h>

#include <sys/ipc.h>

#include <sys/msg.h>

#include <string.h>

struct msg {

long type;

char text[100];

};

int main() {

key\_t key = ftok("progfile", 65);

int msgid = msgget(key, 0666 | IPC\_CREAT);

struct msg m;

if (fork() == 0) {

msgrcv(msgid, &m, sizeof(m.text), 1, 0);

printf("Child received: %s\n", m.text);

} else {

m.type = 1;

printf("Parent sends: ");

fgets(m.text, sizeof(m.text), stdin);

msgsnd(msgid, &m, sizeof(m.text), 0);

wait(NULL);

msgctl(msgid, IPC\_RMID, NULL);

}

return 0;

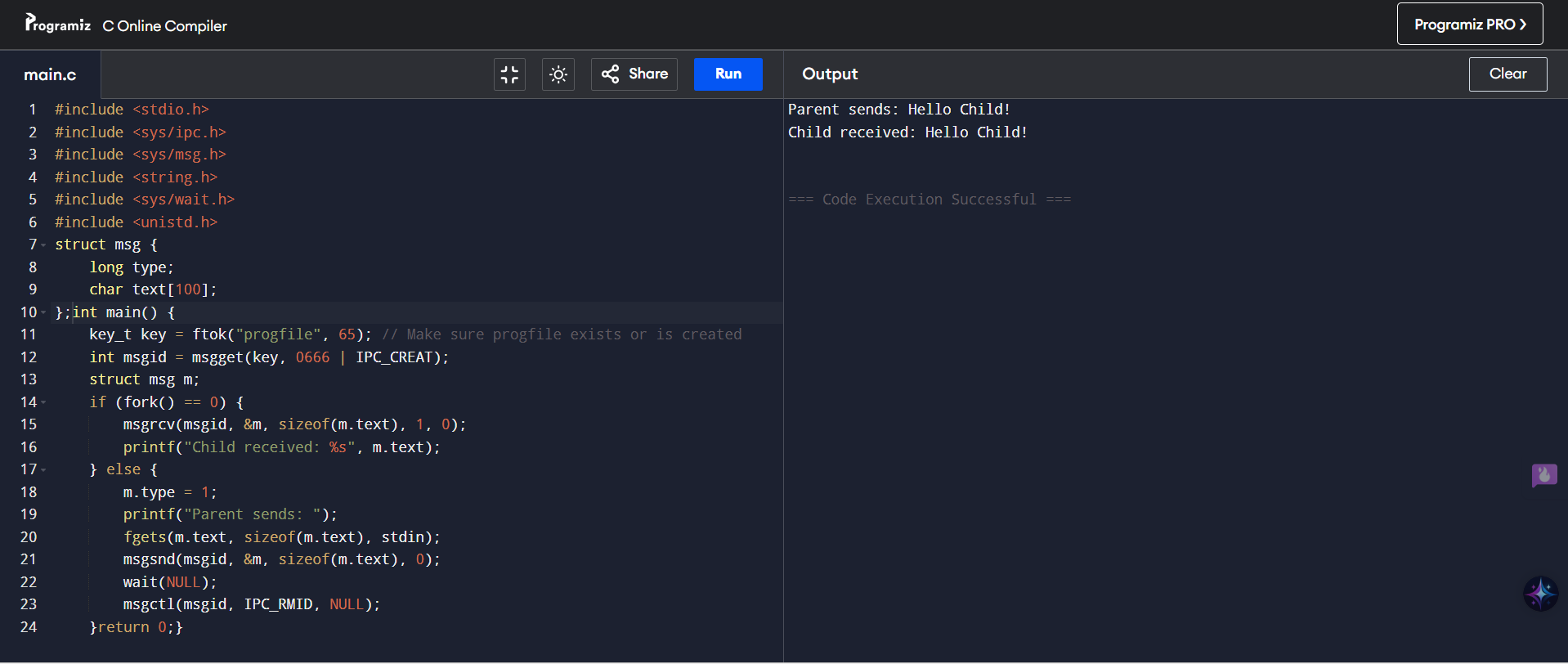
}

**Sample Input**

Parent sends: Hello Child!

**Sample Output**

Child received: Hello Child!



**RESULT:**

IPC using message queue program executed successfully.