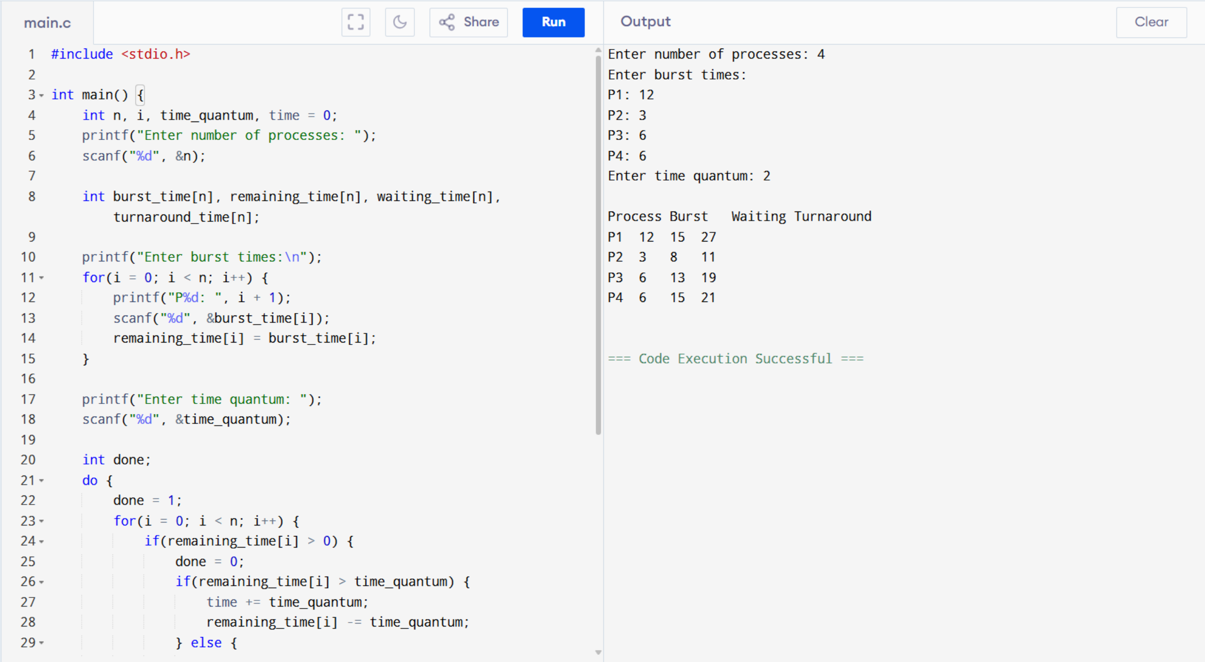
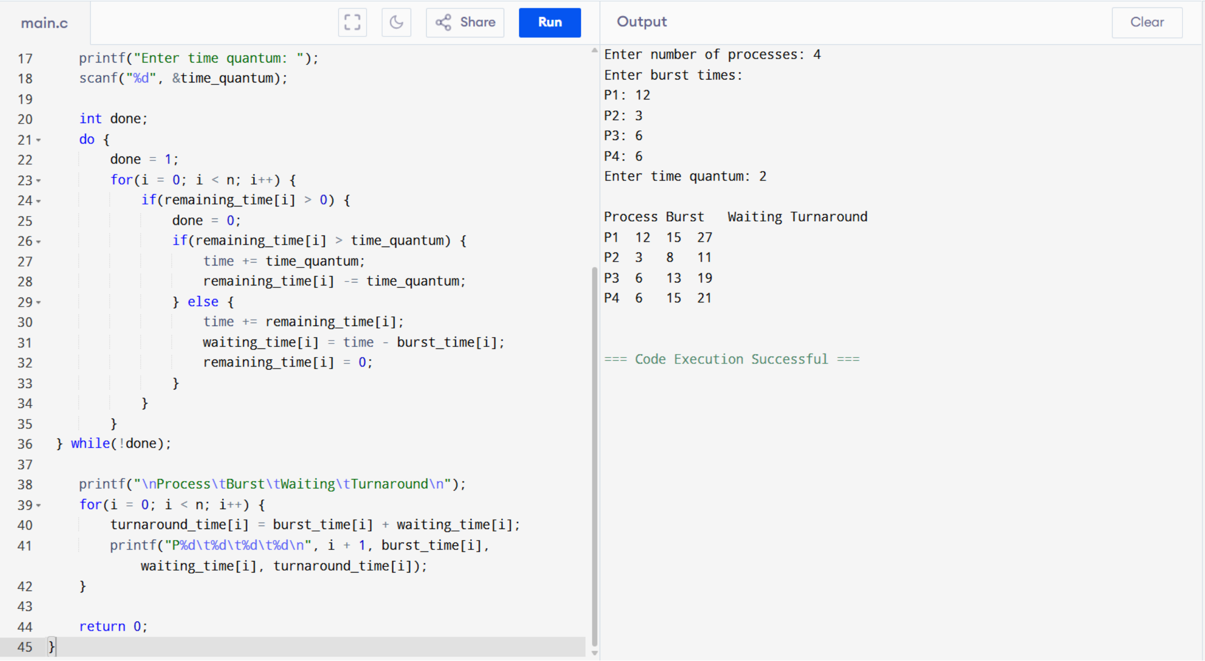


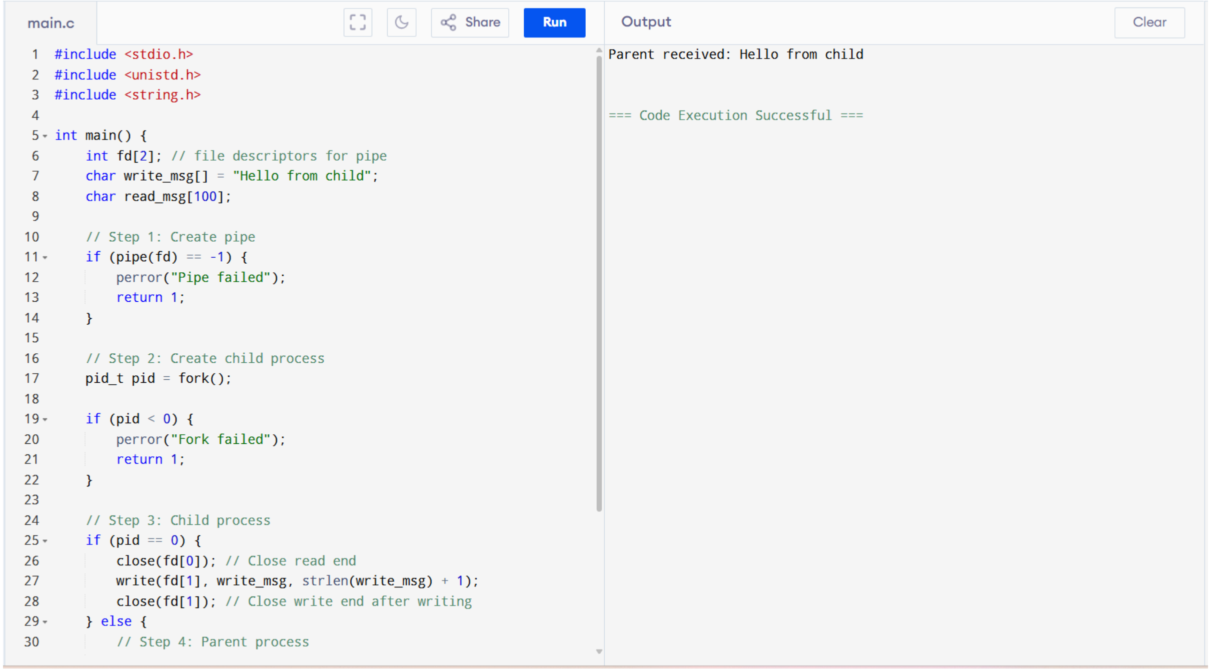
```c
#include <stdio.h>

int main() {
    int n, i, time_quantum, time = 0;
    printf("Enter number of processes: ");
    scanf("%d", &n);

    int burst_time[n], remaining_time[n], waiting_time[n], turnaround_time[n];

    printf("Enter burst times:\n");
    for(i = 0; i < n; i++) {
        printf("P%d: ", i + 1);
        scanf("%d", &burst_time[i]);
        remaining_time[i] = burst_time[i];
    }

    printf("Enter time quantum: ");
    scanf("%d", &time_quantum);

    int done;
    do {
        done = 1;
        for(i = 0; i < n; i++) {
            if(remaining_time[i] > 0) {
                done = 0;
                if(remaining_time[i] > time_quantum) {
                    time += time_quantum;
                    remaining_time[i] -= time_quantum;
                } else {
```

```
Enter number of processes: 4
Enter burst times:
P1: 12
P2: 3
P3: 6
P4: 6
Enter time quantum: 2

Process Burst   Waiting Turnaround
P1   12   15   27
P2   3    8    11
P3   6    13   19
P4   6    15   21


=== Code Execution Successful ===
```



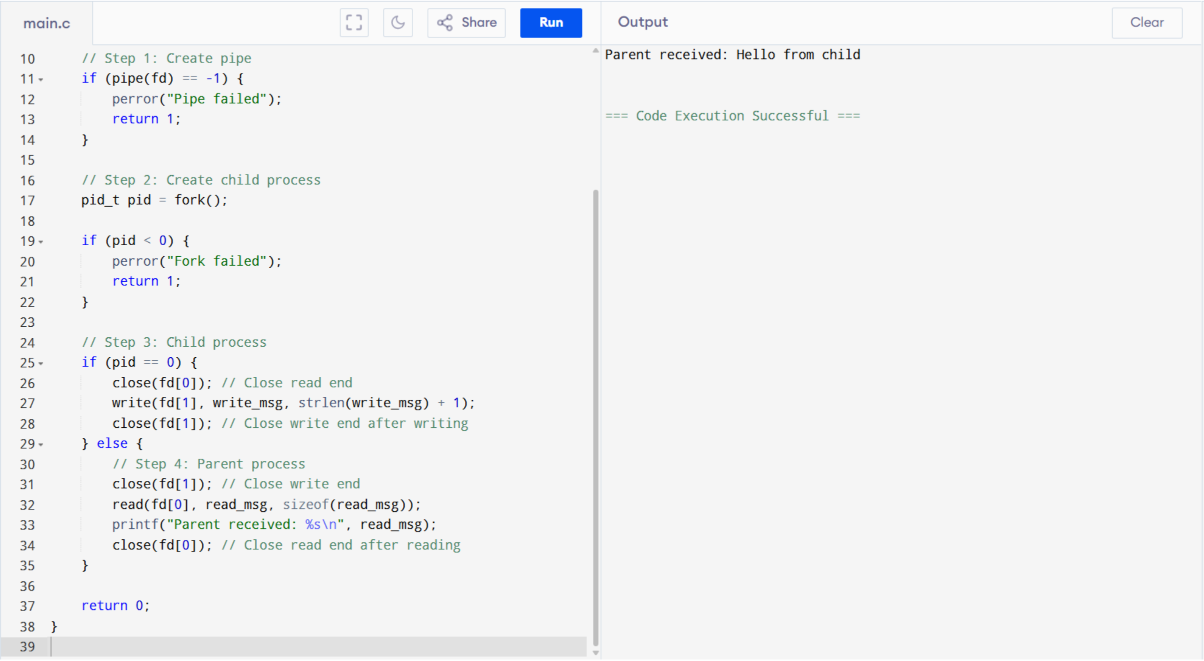
```c
    printf("Enter time quantum: ");
    scanf("%d", &time_quantum);

    int done;
    do {
        done = 1;
        for(i = 0; i < n; i++) {
            if(remaining_time[i] > 0) {
                done = 0;
                if(remaining_time[i] > time_quantum) {
                    time += time_quantum;
                    remaining_time[i] -= time_quantum;
                } else {
                    time += remaining_time[i];
                    waiting_time[i] = time - burst_time[i];
                    remaining_time[i] = 0;
                }
            }
        }
 } while(!done);

    printf("\nProcess\tBurst\tWaiting\tTurnaround\n");
    for(i = 0; i < n; i++) {
        turnaround_time[i] = burst_time[i] + waiting_time[i];
        printf("P%d\t%d\t%d\t%d\n", i + 1, burst_time[i], waiting_time[i], turnaround_time[i]);
    }

    return 0;
}
```

```
Enter number of processes: 4
Enter burst times:
P1: 12
P2: 3
P3: 6
P4: 6
Enter time quantum: 2

Process Burst   Waiting Turnaround
P1   12   15   27
P2   3    8    11
P3   6    13   19
P4   6    15   21


=== Code Execution Successful ===
```



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

int main() {
    int fd[2]; // file descriptors for pipe
    char write_msg[] = "Hello from child";
    char read_msg[100];

    // Step 1: Create pipe
    if (pipe(fd) == -1) {
        perror("Pipe failed");
        return 1;
    }

    // Step 2: Create child process
    pid_t pid = fork();

    if (pid < 0) {
        perror("Fork failed");
        return 1;
    }

    // Step 3: Child process
    if (pid == 0) {
        close(fd[0]); // Close read end
        write(fd[1], write_msg, strlen(write_msg) + 1);
        close(fd[1]); // Close write end after writing
    } else {
        // Step 4: Parent process
```

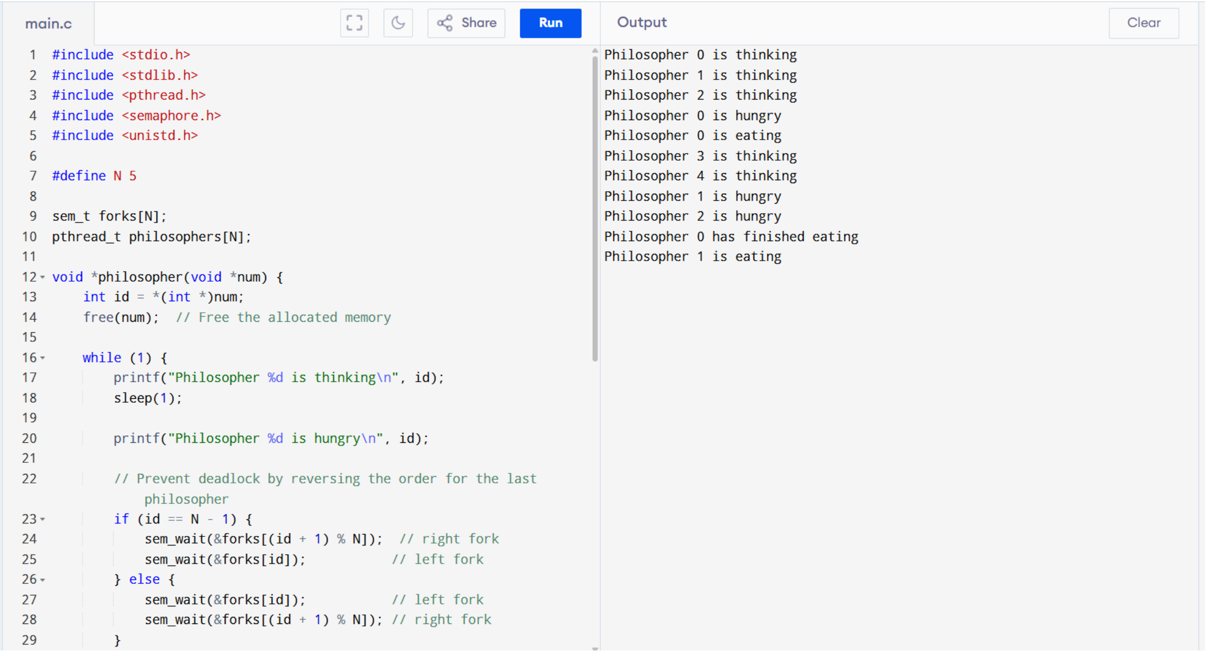
```
Parent received: Hello from child


=== Code Execution Successful ===
```



```c
    // Step 1: Create pipe
    if (pipe(fd) == -1) {
        perror("Pipe failed");
        return 1;
    }

    // Step 2: Create child process
    pid_t pid = fork();

    if (pid < 0) {
        perror("Fork failed");
        return 1;
    }

    // Step 3: Child process
    if (pid == 0) {
        close(fd[0]); // Close read end
        write(fd[1], write_msg, strlen(write_msg) + 1);
        close(fd[1]); // Close write end after writing
    } else {
        // Step 4: Parent process
```

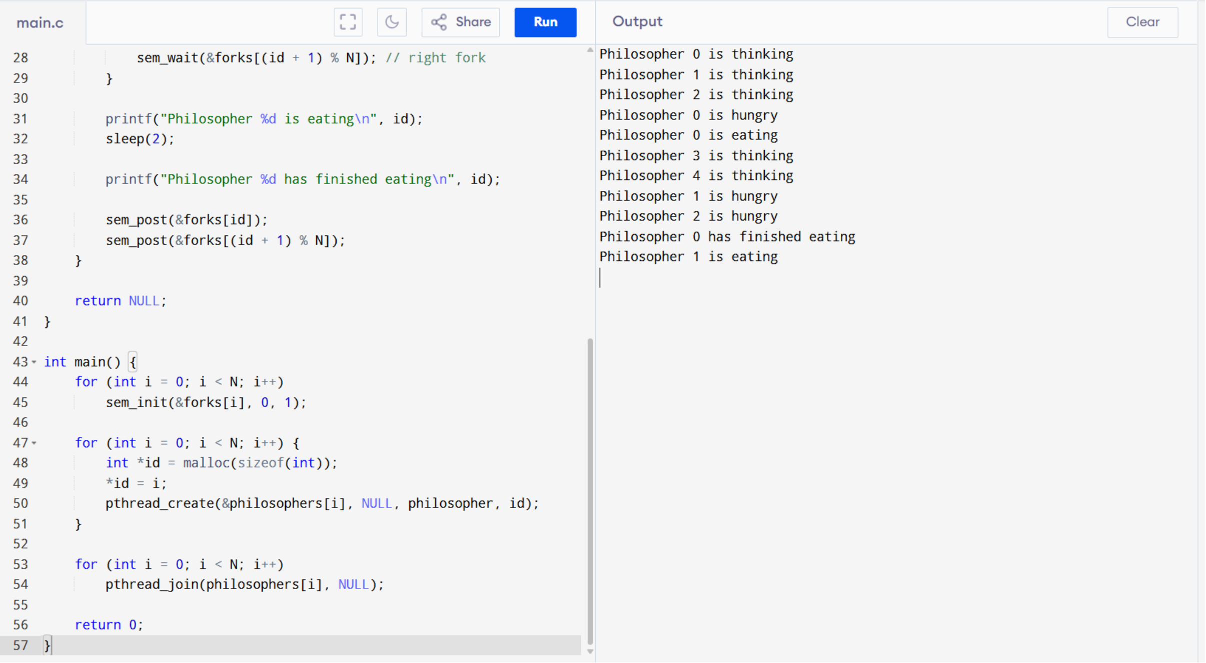
```c
        close(fd[1]); // Close write end
        read(fd[0], read_msg, sizeof(read_msg));
        printf("Parent received: %s\n", read_msg);
        close(fd[0]); // Close read end after reading
    }

    return 0;
}

```

```
Parent received: Hello from child


=== Code Execution Successful ===
```



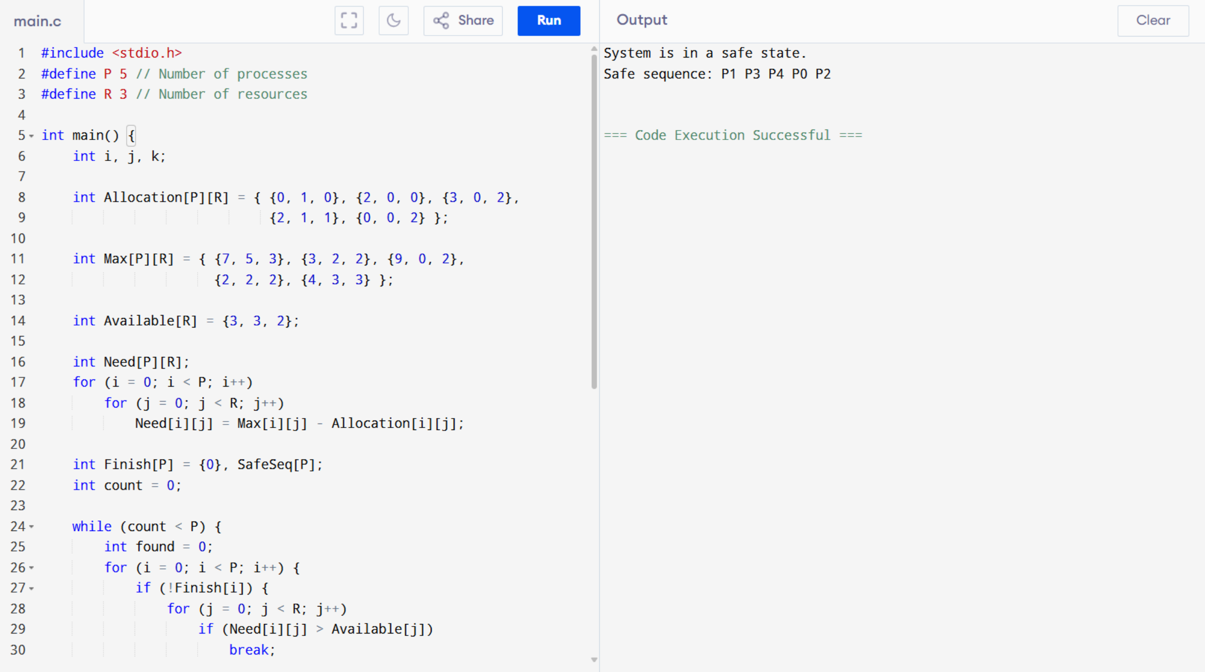
```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

#define N 5

sem_t forks[N];
pthread_t philosophers[N];

void *philosopher(void *num) {
    int id = *(int *)num;
    free(num);  // Free the allocated memory

    while (1) {
        printf("Philosopher %d is thinking\n", id);
        sleep(1);

        printf("Philosopher %d is hungry\n", id);

        // Prevent deadlock by reversing the order for the last philosopher
        if (id == N - 1) {
            sem_wait(&forks[(id + 1) % N]);  // right fork
            sem_wait(&forks[id]);           // left fork
        } else {
            sem_wait(&forks[id]);           // left fork
            sem_wait(&forks[(id + 1) % N]); // right fork
        }
```

```
Philosopher 0 is thinking
Philosopher 1 is thinking
Philosopher 2 is thinking
Philosopher 0 is hungry
Philosopher 0 is eating
Philosopher 3 is thinking
Philosopher 4 is thinking
Philosopher 1 is hungry
Philosopher 2 is hungry
Philosopher 0 has finished eating
Philosopher 1 is eating
```



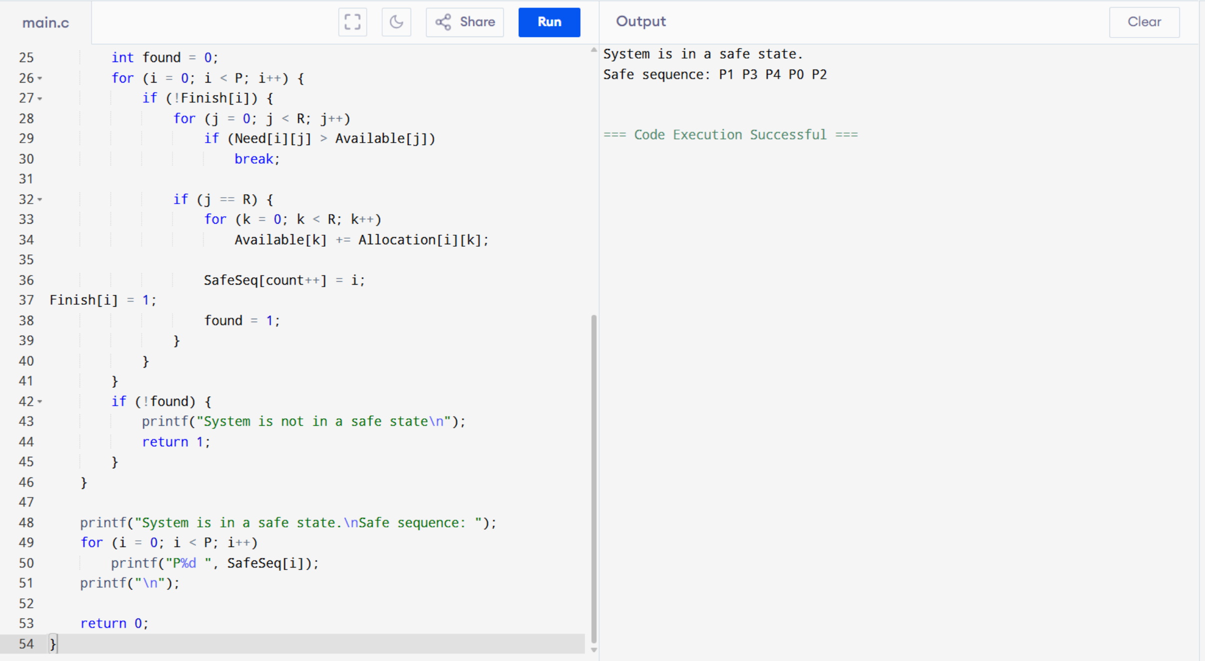
```c
            sem_wait(&forks[(id + 1) % N]); // right fork
        }

        printf("Philosopher %d is eating\n", id);
        sleep(2);

        printf("Philosopher %d has finished eating\n", id);

        sem_post(&forks[id]);
        sem_post(&forks[(id + 1) % N]);
    }

    return NULL;
}

int main() {
    for (int i = 0; i < N; i++)
        sem_init(&forks[i], 0, 1);

    for (int i = 0; i < N; i++) {
        int *id = malloc(sizeof(int));
        *id = i;
        pthread_create(&philosophers[i], NULL, philosopher, id);
    }

    for (int i = 0; i < N; i++)
        pthread_join(philosophers[i], NULL);

    return 0;
}
```

```
Philosopher 0 is thinking
Philosopher 1 is thinking
Philosopher 2 is thinking
Philosopher 0 is hungry
Philosopher 0 is eating
Philosopher 3 is thinking
Philosopher 4 is thinking
Philosopher 1 is hungry
Philosopher 2 is hungry
Philosopher 0 has finished eating
Philosopher 1 is eating
```



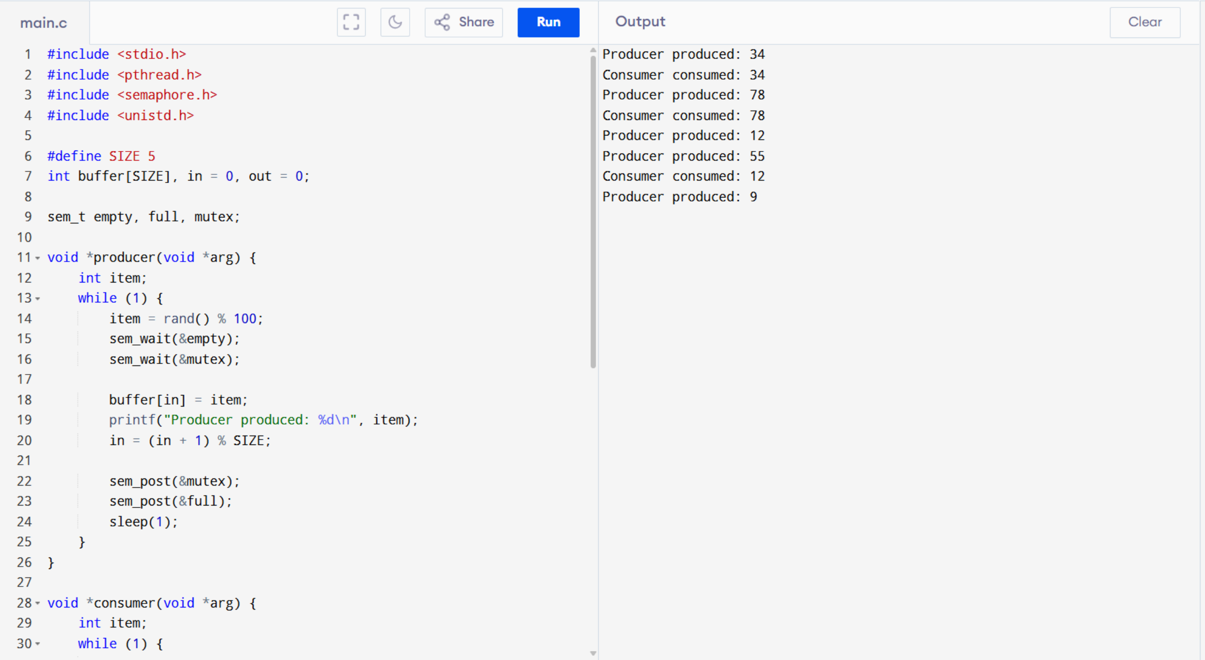
```c
#include <stdio.h>
#define P 5 // Number of processes
#define R 3 // Number of resources

int main() {
    int i, j, k;

    int Allocation[P][R] = { {0, 1, 0}, {2, 0, 0}, {3, 0, 2},
                             {2, 1, 1}, {0, 0, 2} };

    int Max[P][R] = { {7, 5, 3}, {3, 2, 2}, {9, 0, 2},
                      {2, 2, 2}, {4, 3, 3} };

    int Available[R] = {3, 3, 2};

    int Need[P][R];
    for (i = 0; i < P; i++)
        for (j = 0; j < R; j++)
            Need[i][j] = Max[i][j] - Allocation[i][j];

    int Finish[P] = {0}, SafeSeq[P];
    int count = 0;

    while (count < P) {
        int found = 0;
        for (i = 0; i < P; i++) {
            if (!Finish[i]) {
                for (j = 0; j < R; j++)
                    if (Need[i][j] > Available[j])
                        break;
```

```
System is in a safe state.
Safe sequence: P1 P3 P4 P0 P2


=== Code Execution Successful ===
```



```c
        int found = 0;
        for (i = 0; i < P; i++) {
            if (!Finish[i]) {
                for (j = 0; j < R; j++)
                    if (Need[i][j] > Available[j])
                        break;

                if (j == R) {
                    for (k = 0; k < R; k++)
                        Available[k] += Allocation[i][k];

                    SafeSeq[count++] = i;
Finish[i] = 1;
                    found = 1;
                }
            }
        }
        if (!found) {
            printf("System is not in a safe state\n");
            return 1;
        }
    }

    printf("System is in a safe state.\nSafe sequence: ");
    for (i = 0; i < P; i++)
        printf("P%d ", SafeSeq[i]);
    printf("\n");

    return 0;
}
```

```
System is in a safe state.
Safe sequence: P1 P3 P4 P0 P2


=== Code Execution Successful ===
```



```c
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

#define SIZE 5
int buffer[SIZE], in = 0, out = 0;

sem_t empty, full, mutex;

void *producer(void *arg) {
    int item;
    while (1) {
        item = rand() % 100;
        sem_wait(&empty);
        sem_wait(&mutex);

        buffer[in] = item;
        printf("Producer produced: %d\n", item);
        in = (in + 1) % SIZE;

        sem_post(&mutex);
        sem_post(&full);
        sleep(1);
    }
}

void *consumer(void *arg) {
    int item;
    while (1) {
```

```
Producer produced: 34
Consumer consumed: 34
Producer produced: 78
Consumer consumed: 78
Producer produced: 12
Producer produced: 55
Consumer consumed: 12
Producer produced: 9
```