LOVELY PROFESSIONAL UNIVERSITY

CSE 316 - OPERATING SYSTEMS

Question No 03

REPORT FILE ON QUESTION NO 03

SUBMITTED BY :
Ashutosh Pandey - RK21GP62

Guided by:
Mrs Cherry Khosla Mam

Table of Contents

Your paragraph text

Question

Write a multithreaded program that implements the banker's algorithm. Create n threads that request and release resources from the bank. The banker will grant the request only if it leaves the system in a safe state. It is important that shared data be safe from concurrent access. To ensure safe access to shared data, you can use mutex locks.

Ensure:
1. The program should be dynamic such that the threads are created at run time based on the input from the user.
2. The resources must be displaced after each allocation.
3. The system state should be visible after each allocation.

Introduction

The Banker's algorithm is a classic example of a resource allocation algorithm used to prevent deadlock in computer systems. It is an algorithm that is used to manage and allocate resources among multiple processes in a system.

In this algorithm, the system keeps track of the resources that are available and the resources that are currently being used by each process. The algorithm determines whether a request for resources by a process can be safely granted or not, based on the current state of the system and the future requests that are expected to be made by other processes.

Implementing the Banker's algorithm requires careful management of shared data to ensure that multiple processes can access the system's resources without causing conflicts or deadlock. Multithreading and mutex locks are used to ensure that shared data is accessed safely by multiple threads.

In this task, we will create a multithreaded program that implements the Banker's algorithm, which creates n threads that request and release resources from the bank. The program will grant the request only if it leaves the system in a safe state. The program will be dynamic in such a way that the threads are created at run time based on the input from the user. The resources must be displaced after each allocation, and the system state should be visible after each allocation.

Through this exercise, we will gain a better understanding of the Banker's algorithm and how to manage shared data in a multithreaded environment.

Brief Description

The banker's algorithm is a resource allocation and deadlock avoidance algorithm used in operating systems. It is designed to determine if allocating requested resources to a process leaves the system in a safe state, i.e., it avoids deadlock, and ensures that the system can continue to allocate resources to other processes.

To implement the banker's algorithm in a multithreaded program, we can create n threads that request and release resources from the bank. The program should be dynamic, such that the number of threads is determined at runtime based on user input. We will need to use mutex locks to ensure safe access to shared data.

After each allocation, the program should display the resources that are currently available in the bank, as well as the system state. This will help us ensure that the algorithm is working correctly and that the system remains in a safe state.

To implement the algorithm, we can follow these steps:

1. Create a data structure to represent the current state of the system. This should include the total number of resources available, the number of resources allocated to each process, and the number of resources needed by each process.
2. Create n threads, where n is the number of processes in the system. Each thread should represent a process and should request and release resources from the bank.
3. When a thread requests resources, check if the request can be granted without leaving the system in an unsafe state. If the request can be granted, update the system state to reflect the allocation and display the new state of the system.
4. When a thread releases resources, update the system state to reflect the release and display the new state of the system.
5. Use mutex locks to ensure safe access to shared data.

By following these steps, we can implement a multithreaded program that uses the banker's algorithm to allocate resources and avoid deadlock in a dynamic system.

Logic of the Code

This C program is an implementation of the banker's algorithm for deadlock avoidance in an operating system. The banker's algorithm is a resource allocation and deadlock avoidance algorithm used to manage resources and avoid deadlock situations in a computer system.

The program first prompts the user for the number of processes and resources in the system. It then prompts for the current availability of each resource type, followed by the resource allocation and maximum resource request for each process.

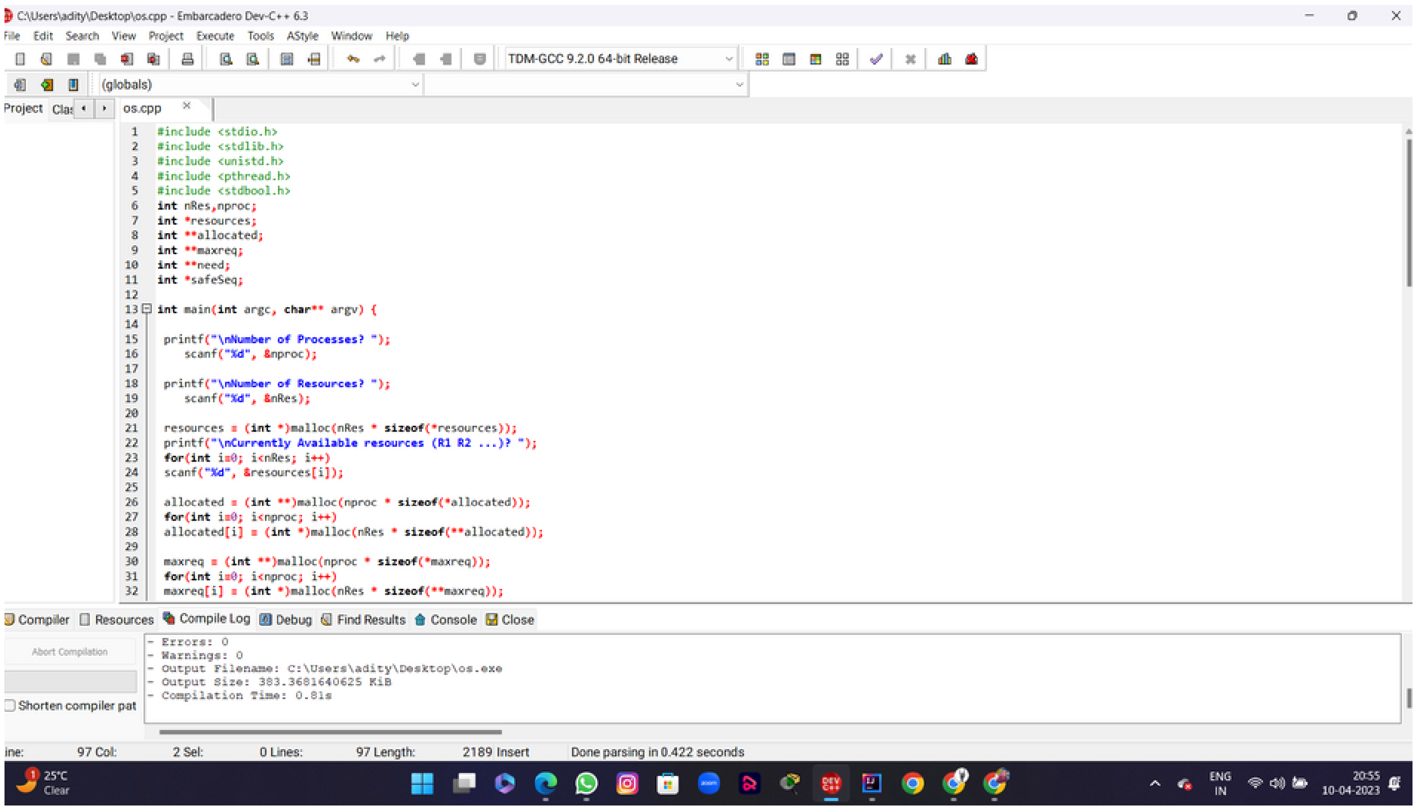
The program calculates the resource need for each process by subtracting the allocated resources from the maximum resource request. It then initializes an array for the safe sequence of processes.

The getSafeSeq function attempts to find a safe sequence of processes that will allow them to complete without deadlock. It does this by checking if each process can complete its resource need based on the available resources. If so, the process is added to the safe sequence, and its allocated resources are released. If not, the process is skipped, and the algorithm moves on to the next process.

If a safe sequence is found, it is returned as an array of process IDs. If a safe sequence is not found, the function returns false.

The program does not implement actual thread synchronization or multi-processing capabilities, but it can serve as a useful starting point for further development of a fully functional deadlock avoidance algorithm

SCREENSHOTS

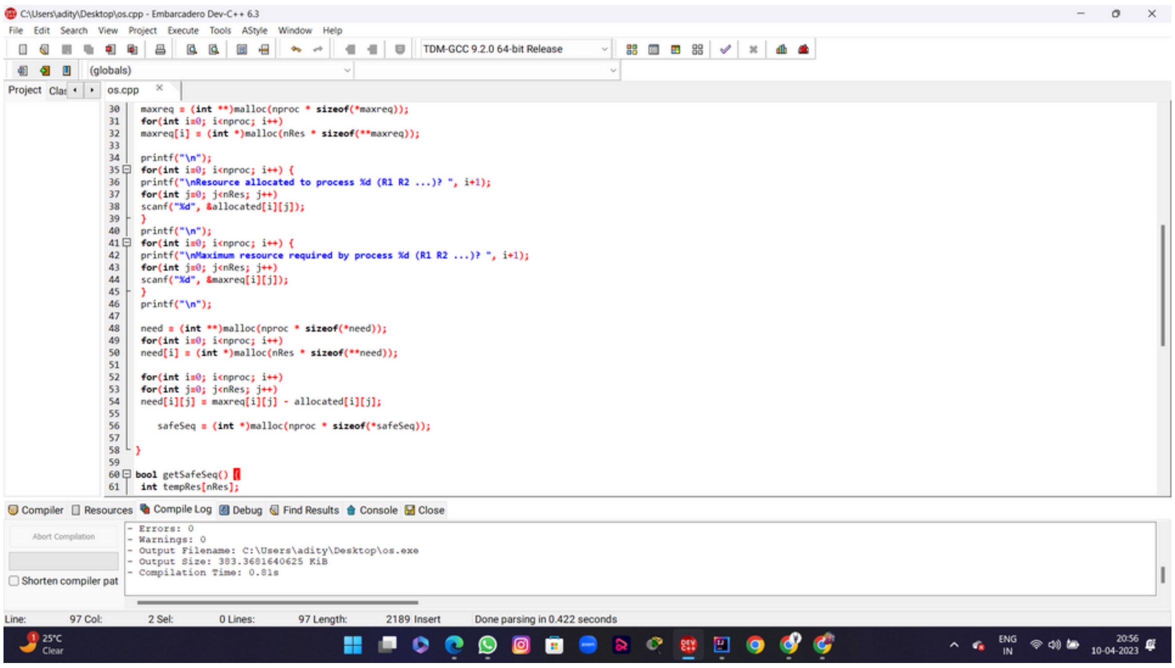


C:\Users\adity\Desktop\os.cpp - Embarcadero Dev-C++ 6.3
File Edit Search View Project Execute Tools AStyle Window Help
TDM-GCC 9.2.0 64-bit Release
(globals)
Project Clas
os.cpp

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <stdbool.h>
int nRes,nproc;
int *resources;
int **allocated;
int **maxreq;
int **need;
int *safeSeq;

int main(int argc, char** argv) {

 printf("\nNumber of Processes? ");
    scanf("%d", &nproc);

 printf("\nNumber of Resources? ");
    scanf("%d", &nRes);

 resources = (int *)malloc(nRes * sizeof(*resources));
 printf("\nCurrently Available resources (R1 R2 ...)? ");
 for(int i=0; i<nRes; i++)
 scanf("%d", &resources[i]);

 allocated = (int **)malloc(nproc * sizeof(*allocated));
 for(int i=0; i<nproc; i++)
 allocated[i] = (int *)malloc(nRes * sizeof(**allocated));

 maxreq = (int **)malloc(nproc * sizeof(*maxreq));
 for(int i=0; i<nproc; i++)
 maxreq[i] = (int *)malloc(nRes * sizeof(**maxreq));
```

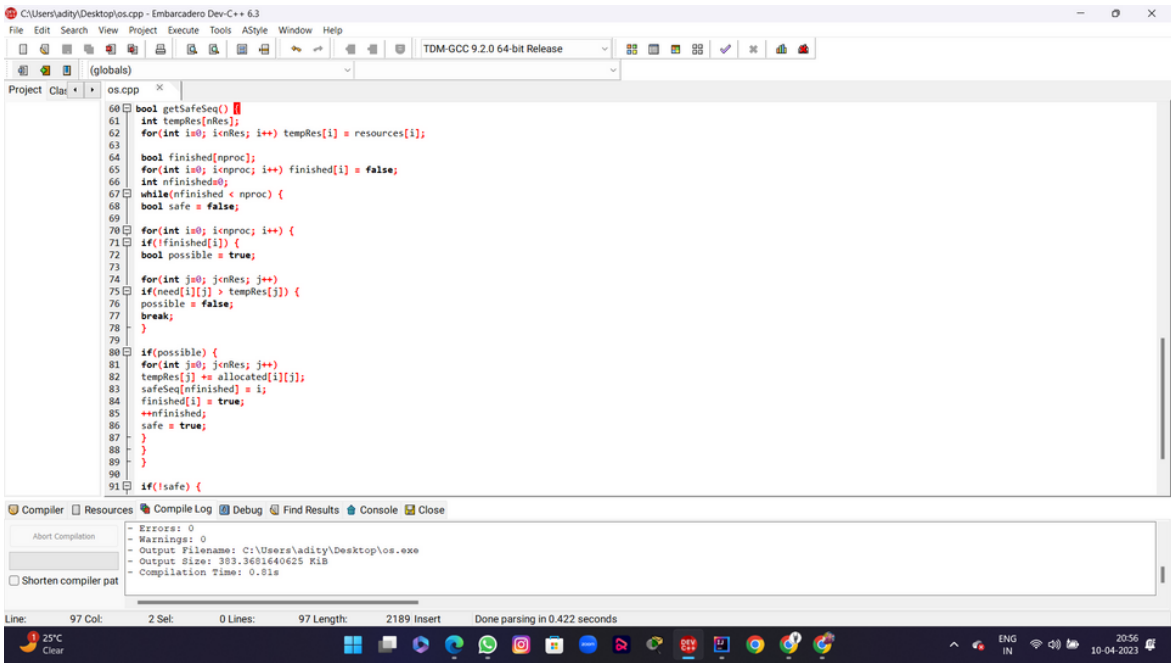
```
- Errors: 0
- Warnings: 0
- Output Filename: C:\Users\adity\Desktop\os.exe
- Output Size: 383.3681640625 KiB
- Compilation Time: 0.81s
```

ine: 97 Col: 2 Sel: 0 Lines: 97 Length: 2189 Insert
Done parsing in 0.422 seconds
25°C
Clear
ENG
IN
20:55
10-04-2023



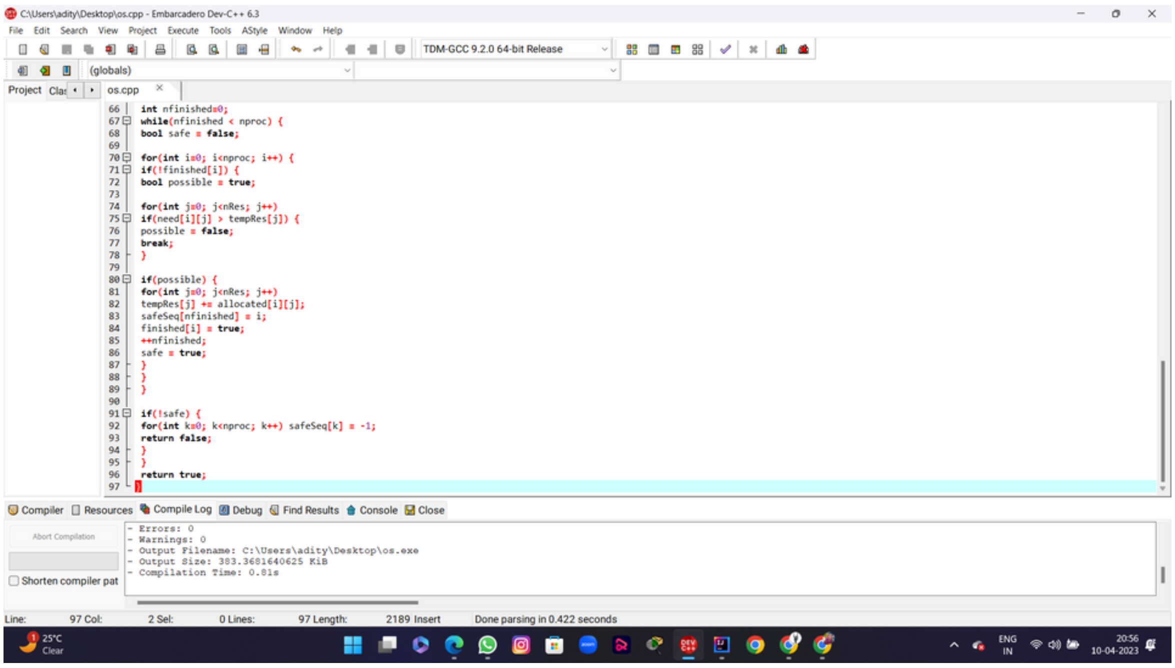
```
maxreq = (int **)malloc(nproc * sizeof(*maxreq));
for(int i=0; i<nproc; i++)
maxreq[i] = (int *)malloc(nRes * sizeof(**maxreq));

printf("\n");
for(int i=0; i<nproc; i++) {
printf("\nResource allocated to process %d (R1 R2 ...)? ", i+1);
for(int j=0; j<nRes; j++)
scanf("%d", &allocated[i][j]);
}
printf("\n");
for(int i=0; i<nproc; i++) {
printf("\nMaximum resource required by process %d (R1 R2 ...)? ", i+1);
for(int j=0; j<nRes; j++)
scanf("%d", &maxreq[i][j]);
}
printf("\n");

need = (int **)malloc(nproc * sizeof(*need));
for(int i=0; i<nproc; i++)
need[i] = (int *)malloc(nRes * sizeof(**need));

for(int i=0; i<nproc; i++)
for(int j=0; j<nRes; j++)
need[i][j] = maxreq[i][j] - allocated[i][j];

   safeSeq = (int *)malloc(nproc * sizeof(*safeSeq));

}

bool getSafeSeq() {
 int tempRes[nRes];
```

```
- Errors: 0
- Warnings: 0
- Output Filename: C:\Users\adity\Desktop\os.exe
- Output Size: 383.3681640625 KiB
- Compilation Time: 0.81s
```



```
bool getSafeSeq() {
 int tempRes[nRes];
 for(int i=0; i<nRes; i++) tempRes[i] = resources[i];

 bool finished[nproc];
 for(int i=0; i<nproc; i++) finished[i] = false;
 int nfinished=0;
 while(nfinished < nproc) {
 bool safe = false;

 for(int i=0; i<nproc; i++) {
 if(!finished[i]) {
 bool possible = true;

 for(int j=0; j<nRes; j++)
 if(need[i][j] > tempRes[j]) {
 possible = false;
 break;
 }

 if(possible) {
 for(int j=0; j<nRes; j++)
 tempRes[j] += allocated[i][j];
 safeSeq[nfinished] = i;
 finished[i] = true;
 ++nfinished;
 safe = true;
 }
 }
 }

 if(!safe) {
```

```
- Errors: 0
- Warnings: 0
- Output Filename: C:\Users\adity\Desktop\os.exe
- Output Size: 383.3681640625 KiB
- Compilation Time: 0.81s
```


C:\Users\adity\Desktop\os.cpp - Embarcadero Dev-C++ 6.3
File Edit Search View Project Execute Tools AStyle Window Help
TDM-GCC 9.2.0 64-bit Release
(globals)
Project Clas
os.cpp
Compiler Resources Compile Log Debug Find Results Console Close
Abort Compilation
Shorten compiler pat
25°C Clear
ENG IN
20:56 10-04-2023

```
int nfinished=0;
while(nfinished < nproc) {
bool safe = false;

for(int i=0; i<nproc; i++) {
if(!finished[i]) {
bool possible = true;

for(int j=0; j<nRes; j++)
if(need[i][j] > tempRes[j]) {
possible = false;
break;
}

if(possible) {
for(int j=0; j<nRes; j++)
tempRes[j] += allocated[i][j];
safeSeq[nfinished] = i;
finished[i] = true;
++nfinished;
safe = true;
}
}
}

if(!safe) {
for(int k=0; k<nproc; k++) safeSeq[k] = -1;
return false;
}
}
return true;
}
```

```
- Errors: 0
- Warnings: 0
- Output Filename: C:\Users\adity\Desktop\os.exe
- Output Size: 383.3681640625 KiB
- Compilation Time: 0.81s
```

Line: 97 Col: 2 Sel: 0 Lines: 97 Length: 2189 Insert Done parsing in 0.422 seconds