



CBS1006
PRINCIPLES OF
OPERATING SYSTEMS
LAB ASSESSMENT 5

27-April-2021

Siddhartha Purwar
[19BBS0072]

1. Code the Banker's algorithm in C and test the working of it with arbitrary inputs.

```
//Siddhartha Purwar 27-April-2021
//Banker algorithm
//Safety algorithm and Request algorithm
#include<stdio.h>
#include<string.h>

#define MAX_RES 100//maximum number of resource type
#define MAX_PRO 100//maximum number of process

int resource_type = 0, process_total = 0;
int seq[MAX_PRO], finish[MAX_PRO], available[MAX_RES], temp_work[MAX_RES], work[MAX_RES], resou
rce[MAX_RES];
int process_alc[MAX_PRO][MAX_RES], process_max[MAX_PRO][MAX_RES], need[MAX_PRO][MAX_RES];

void input_data();
void safety_algorithm();
void request_algorithm();

int main(){
    printf("Banker's Algorithm\n\n");
    input_data();
    safety_algorithm();
    request_algorithm();
    return 0;
}

void input_data(){
    printf("Input the following data\n");
    printf("Enter total type of resource\n");
    scanf("%d", &resource_type);
    for(int i = 0; i < resource_type; i++){
        printf("Total number of instances for resource %d\n", i + 1);
        scanf("%d", &resource[i]);
        available[i] = resource[i];
    }
    printf("Enter total number of process\n");
    scanf("%d", &process_total);
    for(int i = 0; i < process_total; i++){
        printf("\nEnter max resource for PROCESS %d\n", i + 1);
        for(int j = 0; j < resource_type; j++){
            scanf("%d", &process_max[i][j]);
        }
    }
    for(int i = 0; i < process_total; i++){
        printf("\nEnter allocated resource for PROCESS %d\n", i + 1);
        for(int j = 0; j < resource_type; j++){
            scanf("%d", &process_alc[i][j]);
            available[j] -= process_alc[i][j];
            need[i][j] = process_max[i][j] - process_alc[i][j];
        }
    }
}
```

```

    }
}

memcpy(work, available, sizeof(work)); //work = available
printf("Total available resources are \n");
for(int i = 0; i < resource_type; i++){
    printf("%d\n", work[i]);
}
for(int i = 0; i < process_total; i++){
    printf("Need for process %d-->\t", i + 1);
    for(int j = 0; j < resource_type; j++){
        printf("%d\t", need[i][j]);
    }
    printf("\n");
}
}

void safety_algorithm(){
    int flag[process_total], h = 0, l = process_total;
    int temp_work[MAX_RES];
    memcpy(temp_work, work, sizeof(temp_work));
    for(int i = 0; i < process_total; i++){
        finish[i] = -1; // using -1 to indicate false
    }
    for(int a = 0; a < l; a++){
        int i = (a % process_total);
        for(int j = 0; j < resource_type; j++){
            if(need[i][j] > temp_work[j]){
                flag[i] = -1;
                break;
            }
            flag[i] = 1;
        }
        if(finish[i] == 1){
            continue;
        }
        if(flag[i] == -1){
            continue;
        }
        for(int j = 0; j < resource_type; j++){
            temp_work[j] += process alc[i][j];
        }
        printf("\ntemp_work -->\t");
        for(int q = 0; q < resource_type; q++){
            printf("%d\t", temp_work[q]);
        }
        l += process_total;
        finish[i] = 1;
        seq[h++] = i;
    }
    for(int i = 0; i < process_total; i++){
        if(finish[i] == -1){
            printf("\nDEADLOCK EXISTS\n");
            return;
        }
    }
}

```

```

    }
}

printf("\nNO DEADLOCK EXIT\nSequence of execurion can be\n");
for(int i = 0; i < process_total; i++){
    printf("Process %d\t", seq[i] + 1);
}
printf("\n");
return;
}

void request_algorithm(){
    int reques_id = -1, request[resource_type];
    printf("\nEnter the Requesting Process Number (%d....%d)\t", 0, process_total - 1);
    scanf("%d", &reques_id);
    printf("\nEnter requested resources\n");
    for(int i = 0; i < resource_type; i++){
        scanf("%d", &request[i]);
        if(request[i] > need[reques_id][i]){
            printf("\nCannot accept the request [Error: REQUEST is more than NEED]");
            return;
        }
        if(request[i] > available[i]){
            printf("\nCannot accept the request [Error: REQUEST is more than AVAILABLE]");
            return;
        }
    }
    for(int i = 0; i < resource_type; i++){
        available[i] -= request[i];
        process_alc[reques_id][i] += request[i];
        need[reques_id][i] -= request[i];
    }
    memcpy(work, available, sizeof(work));
    safety_algorithm();
}

}

```

OUTPUT:

```
siddhartha@siddhartha-VirtualBox:/media/sf_D_DRIVE/c_programming/OS/a5$ ./a.out
Banker's Algorithm

Input the following data
Enter total type of resource
3
Total number of instances for resource 1
3 14 11
Total number of instances for resource 2
Total number of instances for resource 3
Enter total number of process
4

Enter max resource for PROCESS 1
0 0 1

Enter max resource for PROCESS 2
1 7 5

Enter max resource for PROCESS 3
2 3 5

Enter max resource for PROCESS 4
0 6 5

Enter allocated resource for PROCESS 1
0 0 1

Enter allocated resource for PROCESS 2
1 0 0

Enter allocated resource for PROCESS 3
1 3 5

Enter allocated resource for PROCESS 4
0 6 3
Total available resources are
1
5
2
```

```
Need for process 1--> 0 0 0
Need for process 2--> 0 7 5
Need for process 3--> 1 0 0
Need for process 4--> 0 0 2

temp_work --> 1 5 3
temp_work --> 2 8 8
temp_work --> 2 14 11
temp_work --> 3 14 11
NO DEADLOCK EXIT
Sequence of execution can be
Process 1 Process 3 Process 4 Process 2

Enter the Requesting Process Number (0....3) 1

Enter requested resources
0 5 2

temp_work --> 1 0 1
temp_work --> 2 3 6
temp_work --> 2 9 9
temp_work --> 3 14 11
NO DEADLOCK EXIT
Sequence of execution can be
Process 1 Process 3 Process 4 Process 2
siddhartha@siddhartha-VirtualBox:/media/sf_D_DRIVE/c_programming/OS/a5$
```

2. Write a C program to provide a solution for the classical synchronization problem namely the Producer Consumer problem using POSIX Semaphores (Do not use the same approach as given in the material).

```
#include <pthread.h>
#include <semaphore.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>/>for using sin function for generating buffer data

#define total_producer 2
#define total_consumer 2
#define Buffer 5

sem_t empty; //keep track of the number of empty spots
sem_t full; //keep track of the number of full

int in = 0; //buffer[in%Buffer] is the first empty position
int out = 0; //buffer[out%Buffer] is the first empty position
int buffer[Buffer]; //shared buffer
pthread_mutex_t mutex; //for ensuring mutual exclusion to shared data

void *producer(void *arg){
    int data;
    for(int i = 0; i < Buffer; i++, in = (in + 1) % Buffer){
        data = (int)((sin((i + 9) * (i * 11))) * 100); // Produce an random integer
        sem_wait(&empty);
        pthread_mutex_lock(&mutex);
        //critical section starts
        buffer[in] = data;
        printf("Producer writing data %d at position%d\n", buffer[in],in);
        //ctitical section ends
        pthread_mutex_unlock(&mutex);
        sem_post(&full);
    }
    pthread_exit(NULL);
}

void *consumer(void *arg){
    for(int j = 0; j < Buffer; j++, out = (out + 1) % Buffer){
        sem_wait(&full);
        pthread_mutex_lock(&mutex);
        //critical section starts
        int item = buffer[out];
        printf("\t\t\t\t\tConsumer reading data %d from position %d\n",item, out);
        //critical section ends
        pthread_mutex_unlock(&mutex);
        sem_post(&empty);
    }
    pthread_exit(NULL);
}
```

```

int main(){
    sem_init(&full, 0, 0);
    sem_init(&empty, 0, Buffer);
    pthread_t prod[total_producer], cons[total_consumer];
    pthread_mutex_init(&mutex, NULL);
    for(int p = 0; p < total_producer; p++){
        if(pthread_create(&prod[p], NULL, (void *)producer, NULL)){
            perror("Producer thread");
            exit(-1);
        }
    }
    for(int c = 0; c < total_consumer; c++){
        if(pthread_create(&cons[c], NULL, (void *)consumer, NULL)){
            perror("Consumer thread");
            exit(-1);
        }
    }
    for(int p = 0; p < total_producer; p++){
        pthread_join(prod[p], NULL);
    }
    for(int c = 0; c < total_consumer; c++){
        pthread_join(cons[c], NULL);
    }
    pthread_mutex_destroy(&mutex);
    sem_destroy(&empty);
    sem_destroy(&full);
    return 0;
}

```

OUTPUT

```

siddhartha@siddhartha-VirtualBox:/media/sf_D_DRIVE/c_programming/OS/a5$ gcc q2.c -lpthread -lrt -lm
siddhartha@siddhartha-VirtualBox:/media/sf_D_DRIVE/c_programming/OS/a5$ ./a.out
Producer writing data 0 at position0
Producer writing data -4 at position1
Producer writing data -9 at position2
Producer writing data 15 at position3

Producer writing data 0 at position4
Producer writing data -4 at position0
Producer writing data -9 at position1
Producer writing data 15 at position2

Producer writing data 22 at position3
Producer writing data 22 at position4

Consumer reading data 0 from position 0
Consumer reading data -4 from position 1
Consumer reading data -9 from position 2
Consumer reading data 15 from position 3

Consumer reading data 0 from position 4
Consumer reading data -4 from position 0
Consumer reading data -9 from position 1
Consumer reading data 15 from position 2
Consumer reading data 22 from position 3
Consumer reading data 22 from position 4

siddhartha@siddhartha-VirtualBox:/media/sf_D_DRIVE/c_programming/OS/a5$

```

3. Write a program to study the allocation of memory by applying the following memory allocation strategies.

- FIRST FIT
- BEST FIT
- WORST FIT

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

#define MAX_PARTITION 1000
#define MAX_PROCESS 1000

int total_part, total_proc, part_size[MAX_PARTITION], process_size[MAX_PROCESS];

void input();
void first_fit();
void best_fit();
void worst_fit();

int main(){
    int choice = -1;
    input();
    printf("\nSelect the method:\n1-) First fit\n2-) Best fit\n3-) Worst fit\n");
    scanf("%d", &choice);
    printf("\n");
    switch(choice){
        case 1:
            first_fit();
            break;
        case 2:
            best_fit();
            break;
        case 3:
            worst_fit();
            break;
        default:
            printf("INVALID CHOICE\n");
            break;
    }
    return 0;
}

void input(){
    printf("Enter total number of PARTITION\n");
    scanf("%d", &total_part);
    for(int i = 0; i < total_part; i++){
        printf("Enter size of PARTITION %d\t", i + 1);
        scanf("%d", &part_size[i]);
    }
    printf("Enter total number of PROCESS\n");
    scanf("%d", &total_proc);
}
```

```

        for(int i = 0; i < total_proc; i++){
            printf("Enter size of PROCESS %d\t", i + 1);
            scanf("%d", &process_size[i]);
        }
    }

void first_fit(){
    int j = 0;
    int flag = 1;
    for(int i = 0; i < total_proc; i++){
        flag = 1;
        for(j = 0; j < total_part; j++){
            if(process_size[i] <= part_size[j] && flag == 1){
                part_size[j] = part_size[j] - process_size[i];
                printf("Process %d is allocated in partition %d\n", i + 1, j + 1);
                printf("Memory space available\n");
                for(int k = 0; k < total_part; k++){
                    printf("Partition %d --> %dkb\n", k + 1, part_size[k]);
                }
                flag = -1;
                break;
            }
        }
        if(j == total_part){
            printf("Not sufficient Memory space to accomodate PROCESS %d\n", i + 1);
            return;
        }
    }
}

void worst_fit(){
    int j = 0;
    for(int i = 0; i < total_proc; i++){
        int flag = -1;
        int temp[total_part];
        memcpy(temp, part_size, sizeof(temp));
        int max = -1;
        for(j = 0; j < total_part; j++){
            if(process_size[i] <= temp[j]){
                temp[j] = temp[j] - process_size[i];
                if(max < temp[j]){
                    max = temp[j];
                    flag = j;
                }
            }
        }
        if(flag == -1){
            printf("Not sufficient Memory space to accomodate PROCESS %d\n", i + 1);
            return;
        }
        else{
            printf("Process %d is allocated in partition %d\n", i + 1, flag + 1);
            part_size[flag] = part_size[flag] - process_size[i];
            printf("Memory space available\n");
        }
    }
}

```

```

        for(int k = 0; k < total_part; k++){
            printf("Partition %d --> %dkb\n", k + 1, part_size[k]);
        }
    }

}

void best_fit(){
    int j = 0;
    for(int i = 0; i < total_proc; i++){
        int flag = -1;
        int temp[total_part];
        memcpy(temp, part_size, sizeof(temp));
        int min = 99999;
        for(j = 0; j < total_part; j++){
            if(process_size[i] <= temp[j]){
                temp[j] = temp[j] - process_size[i];
                if(min >= temp[j]){
                    min = temp[j];
                    flag = j;
                }
            }
        }
        if(flag == -1){
            printf("Not sufficient Memory space to accomadate PROCESS %d\n", i + 1);
            return;
        }
        else{
            printf("Process %d is allocated in partition %d\n", i + 1, flag + 1);
            part_size[flag] = part_size[flag] - process_size[i];
            printf("Memory space available\n");
            for(int k = 0; k < total_part; k++){
                printf("Partition %d --> %dkb\n", k + 1, part_size[k]);
            }
        }
    }
}

```

OUTPUT(First_fit) (when all process cannot accommodate)

```
siddhartha@siddhartha-VirtualBox:/media/sf_D_DRIVE/c_programming/OS/a5$ ./a.out
Enter total number of PARTITION
3
Enter size of PARTITION 1      100
Enter size of PARTITION 2      80
Enter size of PARTITION 3      500
Enter total number of PROCESS
5
Enter size of PROCESS 1 75
Enter size of PROCESS 2 100
Enter size of PROCESS 3 120
Enter size of PROCESS 4 450
Enter size of PROCESS 5 50

Select the method:
1-) First fit
2-) Best fit
3-) Worst fit
1

Process 1 is allocated in partition 1
Memory space available
Partition 1 --> 25kb
Partition 2 --> 80kb
Partition 3 --> 500kb
Process 2 is allocated in partition 3
Memory space available
Partition 1 --> 25kb
Partition 2 --> 80kb
Partition 3 --> 400kb
Process 3 is allocated in partition 3
Memory space available
Partition 1 --> 25kb
Partition 2 --> 80kb
Partition 3 --> 280kb
Not sufficient Memory space to accomodate PROCESS 4
```

OUTPUT(Best_fit) (when all process can be accommodate)

```
siddhartha@siddhartha-VirtualBox:/media/sf_D_DRIVE/c_programming/OS/a5$ ./a.out
Enter total number of PARTITION
3
Enter size of PARTITION 1      100
Enter size of PARTITION 2      80
Enter size of PARTITION 3      500
Enter total number of PROCESS
5
Enter size of PROCESS 1 75
Enter size of PROCESS 2 100
Enter size of PROCESS 3 120
Enter size of PROCESS 4 200
Enter size of PROCESS 5 50

Select the method:
1-) First fit
2-) Best fit
3-) Worst fit
2

Process 1 is allocated in partition 2
Memory space available
Partition 1 --> 100kb
Partition 2 --> 5kb
Partition 3 --> 500kb
Process 2 is allocated in partition 1
Memory space available
Partition 1 --> 0kb
Partition 2 --> 5kb
Partition 3 --> 500kb
Process 3 is allocated in partition 3
Memory space available
Partition 1 --> 0kb
Partition 2 --> 5kb
Partition 3 --> 380kb
Process 4 is allocated in partition 3
Memory space available
Partition 1 --> 0kb
Partition 2 --> 5kb
Partition 3 --> 180kb
Process 5 is allocated in partition 3
Memory space available
Partition 1 --> 0kb
Partition 2 --> 5kb
Partition 3 --> 130kb
siddhartha@siddhartha-VirtualBox:/media/sf_D_DRIVE/c_programming/OS/a5$
```

OUTPUT(Worst_fit) (when all process can be accommodate)

```
siddhartha@siddhartha-VirtualBox:~/media/sf_D_DRIVE/c_programming/OS/a5$ gcc q5.c
siddhartha@siddhartha-VirtualBox:/media/sf_D_DRIVE/c_programming/OS/a5$ ./a.out
Enter total number of PARTITION
3
Enter size of PARTITION 1      100
Enter size of PARTITION 2      80
Enter size of PARTITION 3      500
Enter total number of PROCESS
5
Enter size of PROCESS 1 75
Enter size of PROCESS 2 100
Enter size of PROCESS 3 120
Enter size of PROCESS 4 200
Enter size of PROCESS 5 50

Select the method:
1-) First fit
2-) Best fit
3-) Worst fit
3

Process 1 is allocated in partition 3
Memory space available
Partition 1 --> 100kb
Partition 2 --> 80kb
Partition 3 --> 425kb
Process 2 is allocated in partition 3
Memory space available
Partition 1 --> 100kb
Partition 2 --> 80kb
Partition 3 --> 325kb
Process 3 is allocated in partition 3
Memory space available
Partition 1 --> 100kb
Partition 2 --> 80kb
Partition 3 --> 205kb
Process 4 is allocated in partition 3
Memory space available
Partition 1 --> 100kb
Partition 2 --> 80kb
Partition 3 --> 5kb
Process 5 is allocated in partition 1
Memory space available
Partition 1 --> 50kb
Partition 2 --> 80kb
Partition 3 --> 5kb
```