Week 8 – Assignments

1. Write a C program for safety Algorithm (A/P).

```
// Banker's Algorithm

#include <stdio.h>

int main()

{

        // P0, P1, P2, P3, P4 are the Process names here


        int n, m, i, j, k;

        n = 5; // Number of processes

        m = 3; // Number of resources

        int alloc[5][3] = { { 0, 1, 0 }, // P0 // Allocation Matrix

                                                { 2, 0, 0 }, // P1

                                                { 3, 0, 2 }, // P2

                                                { 2, 1, 1 }, // P3

                                                { 0, 0, 2 } }; // P4


        int max[5][3] = { { 7, 5, 3 }, // P0 // MAX Matrix

                                                { 3, 2, 2 }, // P1

                                                { 9, 0, 2 }, // P2

                                                { 2, 2, 2 }, // P3

                                                { 4, 3, 3 } }; // P4


        int avail[3] = { 3, 3, 2 }; // Available Resources
```

```
int f[n], ans[n], ind = 0;
for (k = 0; k < n; k++) {
        f[k] = 0;
}
int need[n][m];
for (i = 0; i < n; i++) {
        for (j = 0; j < m; j++)
                need[i][j] = max[i][j] - alloc[i][j];
}
int y = 0;
for (k = 0; k < 5; k++) {
        for (i = 0; i < n; i++) {
                if (f[i] == 0) {

                        int flag = 0;
                        for (j = 0; j < m; j++) {
                                if (need[i][j] > avail[j]){
                                        flag = 1;
                                        break;
                                }
                        }

                        if (flag == 0) {
                                ans[ind++] = i;
                                for (y = 0; y < m; y++)
```

```
                                        avail[y] += alloc[i][y];
                                    f[i] = 1;
                                }
                            }
                        }
}

int flag = 1;

for(int i=0;i<n;i++)
{
if(f[i]==0)
{
        flag=0;
        printf("The following system is not safe");
        break;
}
}

if(flag==1)
{
printf("Following is the SAFE Sequence\n");
for (i = 0; i < n - 1; i++)
        printf(" P%d ->", ans[i]);
printf(" P%d", ans[n - 1]);
}
```

```
        return (0);


        // This code is contributed by Deep Baldha (CandyZack)
}
```

2. Write a C program for resource request algorithm (A/P).

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

int max[100][100];
int alloc[100][100];
int need[100][100];
int avail[100];
int n, r;

void input();

void show();

void cal();

int main() {
   int i, j;
   printf("********** Banker's Algo ************\n");
   input();
   show();
   cal();
   request();
   getch();
   return 0;
}

void input() {
   int i, j;
   printf("Enter the no of Processes\t");
   scanf("%d", &n);
   printf("Enter the no of resources instances\t");
   scanf("%d", &r);
   printf("Enter the Max Matrix\n");
   for (i = 0; i < n; i++) {
      for (j = 0; j < r; j++) {
         scanf("%d", &max[i][j]);
```

```
    }
  }
  printf("Enter the Allocation Matrix\n");
  for (i = 0; i < n; i++) {
    for (j = 0; j < r; j++) {
      scanf("%d", &alloc[i][j]);
    }

  }
  printf("Enter the available Resources\n");
  for (j = 0; j < r; j++) {
    scanf("%d", &avail[j]);
  }
}

void show() {
  int i, j;
  printf("Process\t Allocation\t Max\t Available\t");
  for (i = 0; i < n; i++) {
    printf("\nP%d\t ", i + 1);
    for (j = 0; j < r; j++) {
      printf("%d ", alloc[i][j]);
    }
    printf("\t");
    for (j = 0; j < r; j++) {
      printf("%d ", max[i][j]);
    }
    printf("\t");
    if (i == 0) {
      for (j = 0; j < r; j++)
        printf("%d ", avail[j]);
    }
  }
}

void cal() {
  int finish[100], temp, need[100][100], flag = 1, k, c1 = 0;
  int safe[100];
  int i, j;
  for (i = 0; i < n; i++) {
    finish[i] = 0;
  }
//find need matrix
  for (i = 0; i < n; i++) {
    for (j = 0; j < r; j++) {
      need[i][j] = max[i][j] - alloc[i][j];
    }
  }
  printf("\n");
  while (flag) {
    flag = 0;
    for (i = 0; i < n; i++) {
      int c = 0;
      for (j = 0; j < r; j++) {
        if ((finish[i] == 0) && (need[i][j] <= avail[j])) {
          c++;
          if (c == r) {
            for (k = 0; k < r; k++) {
              avail[k] += alloc[i][j];
```

```
                    finish[i] = 1;
                    flag = 1;
                }
                printf("P%d->", i);
                if (finish[i] == 1) {
                    i = n;
                }
            }
        }
    }
  }
  for (i = 0; i < n; i++) {
    if (finish[i] == 1) {
      c1++;
    } else {
      printf("P%d->", i);
    }

  }
  if (c1 == n) {
    printf("\n The system is in safe state");
  } else {
    printf("\n Process are in dead lock");
    printf("\n System is in unsafe state");
  }
}

void request() {
  int c, pid, request[100][100], B[100][100], i;
  printf("\n Do you want make an additional request for any of the process ? (1=Yes|0=No)");
  scanf("%d", &c);
  if (c == 1) {
    printf("\n Enter process number : ");
    scanf("%d", &pid);
    printf("\n Enter additional request : \n");
    for (i = 0; i < r; i++) {
      printf(" Request for resource %d : ", i + 1);
      scanf("%d", &request[0][i]);
    }
    for (i = 0; i < r; i++) {
      if (request[0][i] > need[pid][i]) {
        printf("\n ******Error encountered******\n");
        exit(0);
      }
    }
    for (i = 0; i < r; i++) {
      avail[i] -= request[0][i];
      alloc[pid][i] += request[0][i];
      need[pid][i] -= request[0][i];
    }
    cal();
    getch();
  } else {
    exit(0);
  }
}
```

3. Write a C program to simulate deadlock between three threads

```
//WRITE A PROGRAM TO SIMULATE DEADLOCK BETWEEN THREE THREADS
#include<stdio.h>
#include<pthread.h>
#include<unistd.h>
void *function1();
void *function2();
void *function3();
pthread_mutex_t first_mutex; //MUTEX LOCK
pthread_mutex_t second_mutex;
pthread_mutex_t third_mutex;

int main()
{
pthread_mutex_init(&first_mutex,NULL); //INITIALIZE THE LOCK
pthread_mutex_init(&second_mutex,NULL);
pthread_mutex_init(&third_mutex, NULL);
pthread_t one, two, three;
pthread_create(&one, NULL, function1, NULL); //CREATE THREAD
pthread_create(&two, NULL, function2, NULL);
pthread_create(&three, NULL, function3, NULL);
pthread_join(one, NULL);
pthread_join(two, NULL);
pthread_join(three, NULL);
```

```
printf("Thread joined\n");
}
void *function1( )
{
pthread_mutex_lock(&first_mutex); //TO ACQUIRE THE RESOURCE/MUTEX LOCK
printf("Thread ONE Acquired first_mutex\n");
sleep(1);
pthread_mutex_lock(&second_mutex);
printf("Thread ONE Acquired second_mutex\n");
pthread_mutex_unlock(&second_mutex); //TO RELEASE THE RESOURCE
printf("Thread ONE Released second_mutex\n");
pthread_mutex_unlock(&first_mutex);
printf("Thread ONE Released first_mutex\n");
}

void *function2( )
{
pthread_mutex_lock(&second_mutex);
printf("Thread TWO Acquired second_mutex\n");
sleep(1);
pthread_mutex_lock(&third_mutex);
printf("Thread TWO Acquired third_mutex\n");
pthread_mutex_unlock(&third_mutex);
printf("Thread TWO Released third_mutex\n");
pthread_mutex_unlock(&second_mutex);
printf("Thread TWO Released second_mutex\n");
```

```
}


void *function3( )
{
pthread_mutex_lock(&third_mutex);
printf("Thread THREE Acquired third_mutex\n");
sleep(1);
pthread_mutex_lock(&first_mutex);
printf("Thread THREE Acquired first_mutex\n");
pthread_mutex_unlock(&first_mutex);
printf("Thread THREE Released first_mutex\n");
pthread_mutex_unlock(&third_mutex);
printf("Thread THREE Released third_mutex\n");
}
```

4. Write a C program to create a deadlock (A/P).

```
#include<stdio.h>
#include<pthread.h>
#include<unistd.h>
void *function1();
 void *function2();
 pthread_mutex_t first_mutex;  //mutex lock
 pthread_mutex_t second_mutex;


int main() {
 pthread_mutex_init(&first_mutex,NULL);  //initialize the lock
 pthread_mutex_init(&second_mutex,NULL);
 pthread_t one, two;
 pthread_create(&one, NULL, function1, NULL);  // create thread
 pthread_create(&two, NULL, function2, NULL);
 pthread_join(one, NULL);
```

```
 pthread_join(two, NULL);
 printf("Thread joined\n");
 }

void *function1( ) {

     pthread_mutex_lock(&first_mutex);  // to acquire the
resource/mutex lock

     printf("Thread ONE acquired first_mutex\n");

     sleep(1);

     pthread_mutex_lock(&second_mutex);

     printf("Thread ONE acquired second_mutex\n");

     pthread_mutex_unlock(&second_mutex); // to release the resource

     printf("Thread ONE released second_mutex\n");

     pthread_mutex_unlock(&first_mutex);

     printf("Thread ONE released first_mutex\n");

}
```

```
void *function2( ) {
     pthread_mutex_lock(&second_mutex);
     printf("Thread TWO acquired second_mutex\n");
     sleep(1);
     pthread_mutex_lock(&first_mutex);
     printf("Thread TWO acquired first_mutex\n");
     pthread_mutex_unlock(&first_mutex);
     printf("Thread TWO released first_mutex\n");
     pthread_mutex_unlock(&second_mutex);
     printf("Thread TWO released second_mutex\n");

}
```

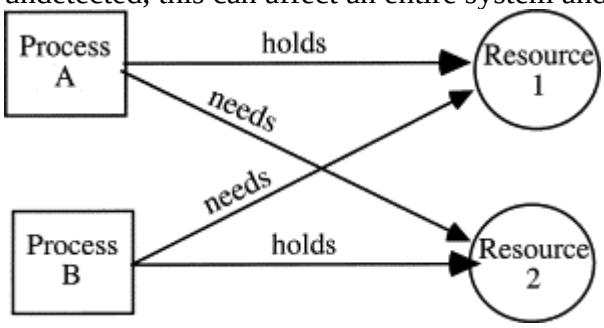
5. Explain in detail about dead lock free condition with appropriate c program

```
// C++ implementation of above program.
#include <bits/stdc++.h>
using namespace std;

// function that calculates
// the minimum no. of resources
int Resources(int process, int need)
{
	int minResources = 0;

	// Condition so that deadlock
	// will not occur
	minResources = process * (need - 1) + 1;

	return minResources;
}

// Driver code
int main()
{
	int process = 3, need = 4;

	cout << "R >= " << Resources(process, need);
	return 0;
}
```

6. Explain the role of ‘All Requests Together’ in deadlocks

Deadlock

Deadlock is best illustrated as in Fig. 6. In this simple case, Process A holds one of two resources it requires to run, while Process B holds the other. Process B needs the resource held by Process A in order to run. Each waits on the other. While this is the simplest form of deadlock, cyclic requests for resources can result in deadlocks involving many processes. If undetected, this can affect an entire system and reach a point of near total deadlock.



Sign in to download full-size image

There are two approaches in operating systems' design to deadlock: *detection* (often left to the operator or user) and *prevention*, or *avoidance*. To avoid the situation shown in Fig. 6, it is necessary to ensure that each process *in turn* is given the resources it needs to execute; once the one which gains the resources completes and consequently releases the resources it holds, the other can go ahead without interference. By allocating resources to a process in a single, uninterruptable action, the operating system can avoid creating situations in which deadlock can arise. Such all-or-nothing approaches to processing requests are referred to as *atomic actions*: they either complete successfully or return the system to the state it was in before the triggering request was acted upon. To ensure that a process can hold a resource, such as a record on a file, for the duration of its requirement without any possibility of destructive interference, the concept of *locking* is used; locks effectively deny access to any process other than the one currently “holding the lock.” Locks can be applied at a number of levels (depending upon system design), from whole files to single records. Where data are likely to be updated, locking ensures that data integrity is maintained because it becomes impossible for two (or more) processes to interleave their updating actions in a destructive way. Imagine two people sharing a bank account with a balance of $100. Without the serialization enforced by locking the bank account record, if one withdraws $50 and the other deposits $25, the actions could become interleaved such that the balance afterwards could be (erroneously) either $50 or $125. By locking the record until either the withdrawal is completed *or* the deposit is completed, then allowing the other action to proceed, the resulting balance will be the correct $75.

Of course, it is necessary to ensure that locks which are set are (eventually) freed, that resources which are allocated are eventually returned to the pool of resources available for allocation by the operating system. Otherwise, it is conceivable that a task can wait indefinitely for a resource that has been locked by another process and has never subsequently been unlocked. It is the responsibility of the operating system to ensure that such “housekeeping” as ensuring that resources are not inadvertently made unavailable to the

computing system as a whole, for example, were a process to lock a resource then fail before it completed its actions and released the lock. Part of the solution to these problems is frequently implemented partially or fully within the hardware.