OPERATING SYSTEMS

APPLICATIONS OF THREADS AND PROCESS SYNCHRONIZATION

FACULTY: SHAIK NASEERA

Group Members:

REG.NO	NAME	EMAIL ID
17BCI0108	K. NAVEEN KUMAR REDDY	naveen.kumarreddy2017@vitstudent.ac.in
17BCE0566	NAVEEN AADITYA.CH	naveen.aaditya2017@vitstudent.ac.in
17BCI0117	SAI NIKHIL RAGA	ragasai.nikhil2017@vitstudent.ac.in
17BCE2248	SARTHAK SINGH	sarthak.singh2017@vitstudent.ac.in

ABSTRACT:

In this project, APPLICATIONS OF THREADS and PROCESS SYNCHRONIZATION, we are going to know about the functionalities of Threads and thereby understanding the process of Multithreading and we are going to examine and understand some existing problems and how these problems can be solved using threads and take a new problem and solve it using the concept of Multithreading.

The problems that are taken to examine are:

- ➢ Reader/Writer Problem
- ➢ Dining Philosopher Problem
- ➢ Producer and Consumer Problem
- ➢ Doctor-patient problem
- ➢ Cigarette- Smokers Problem

These five problems give us a good idea on how to use threads and how these problems are approached.After getting a good idea on how to solve these problems using Multithreading and Process Synchronization, we will take three other problems and solve them with the knowledge that we have gained by understanding how existing problems are solved.

The problems that are taken to solve are:

- ➢ Bus Ticket Reservation
- ➢ Course Registration
- ➢ Simultaneous usage of Joint Account

INTRODUCTION:

Thread:

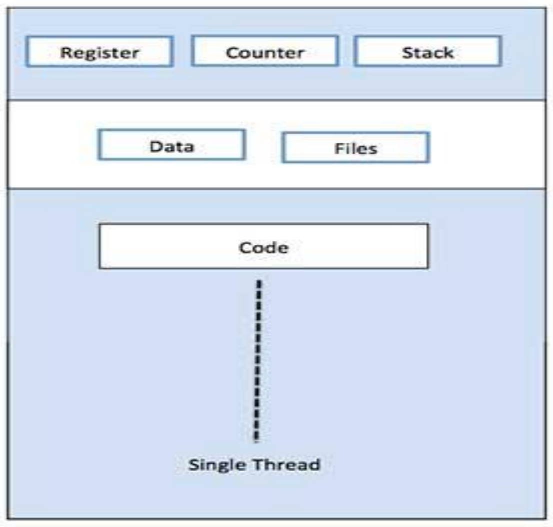
A Thread is defined as the execution of the minimal sequence of programmed instructions which a scheduler can manage independently, which is typically a part of the operating system. A thread is considered as a placeholder information which is associated with single use of a program that can be used to handle multiple concurrent users. A thread is a lightweight process. Threads provide a way to improve application performance through parallelism. Threads represent a software approach to improve the performance of the operating system by reducing overhead thread is equivalent to classical process.

MultiThreading:

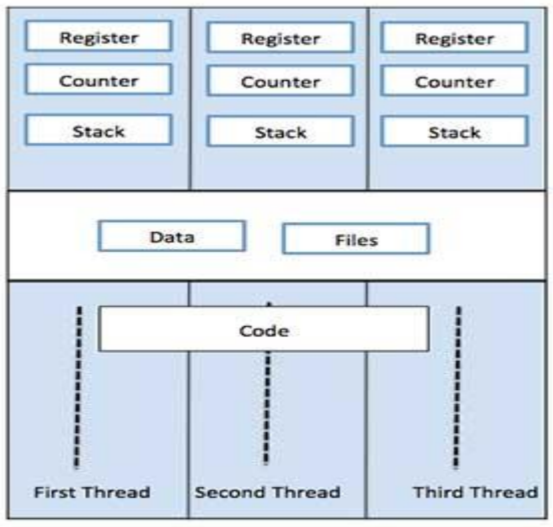
MultiThreading is defined as the ability of Central Processing Unit to execute multiple threads or processes concurrently or simultaneously, supported by the Operating System. In a Multithreaded application, the threads or the processes shares common resources of single or multiple cores which includes the CPU cache, Transition Look Aside Buffer and also the computing units. The main aim of MultiThreading is to increase the utilization of single core by using Thread-level Parallelism.

Advantages of MultiThreading:

- If in a process which consists of multiple threads, one thread gets a lot of cache misses , the other threads can take advantage of the unused computing resources which leads to the faster execution of overall process as the other resources would have been left idle if the process had been executed with a single thread.
- If a thread does not require all the computing resources of the Central Processing Unit then use of multiple threads may prevent the other resources from being idle.



Single Process P with single thread



Single Process P with three threads

Process Synchronization:

Process Synchronization is a mechanism which ensures that two or more processes or threads do not execute the program segment critical section simultaneously. Access of critical section to different processes can be controlled by using synchronization techniques. When one thread is being executed in the critical section then all other threads have to wait until the thread in the critical section finishes its execution. If proper synchronization techniques are not used , then it may lead to a condition known as race condition where the values of shared variables will be unpredictable and vary at the time of context switch of the processes.

Description of Existing Problems:

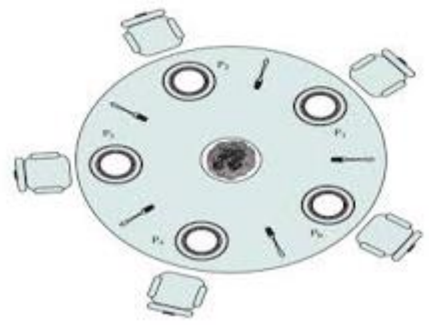
Reader/Writer problem:

When one writer is writing into the data area, another writer or reader can't access the data area for its purpose. When one reader is reading from the data area, other readers can also read but writer can't write.

	R	W
R	T	F
W	F	F

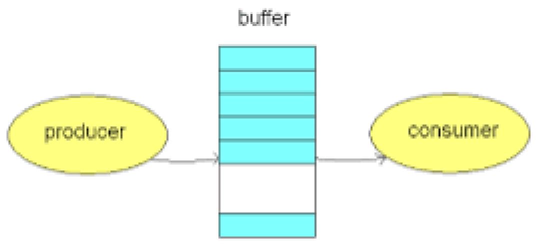
Dining Philosopher Problem:

There are 5 philosophers, who share a round table, which has one chair for each person. There are 5 chopsticks placed in between them and a bowl of rice in middle of the table. When a philosopher gets hungry, he grabs the nearest two chopsticks and starts eating, if the person beside him is already eating, then he has to wait until he gets the chopstick from that person.



Producer Consumer Problem:

A producer produces the item which the consumer consumes. When the producer has not yet produced any item, the consumer cannot consume it and when the consumer does not consume atleast one item in the buffer, the producer has to wait until he gets one empty place in the buffer in order to produce an item.



Doctor Patient problem:

In clinics, patients are given chairs where they can wait until they meet the doctor.If the number of patients are more, then some patients have to stand and wait until a patient comes out of doctor’s room and a patient enters the doctor’s room.Later, when a chair is left empty, the patient who is waiting can take the seat.

Cigarette smoker’s problem:

There are three requirements to make the cigarette, and each smoker will carry any one ingredient of his choice.Later, two random requirements out of three are kept on the table and the lucky person who has the other ingredient can make the cigarette and smoke. Within that time, another two ingredients can be kept on the table and the process continues on.

Alltheabove discussed problems are being solved by using the concept of multithreading and by using locks for process synchronization.

Description of the problems that we are going to solve:

Bus Ticket Reservation:

Case-1:

Normally while booking a ticket, if there is only one thread then if one user starts booking the ticket, all other users should wait until the first user books or cancels the ticket.

To avoid this, the concept the multithreading can be used. By using multiple threads, two or more users can book the tickets simultaneously.

Case-2:

But still there would be a problem if two users try to book same ticket simultaneously, they might end up in booking the same ticket which should not happen.

This problem can be solved by using the concept of Process Synchronization.

Course Registration:

Case-1:

During the time of Course Registration, if there is only one thread allotted to that process, if one student tries to register the course , others should wait until that student selects or drops a course.

This problem can be solved by using the concept of multithreading. By using multiple threads, many students can register the course simultaneously.

Case-2:

But still there would be a problem if there is only one seat in the course and two students try to register that one seat, then they end up in registering the course which should not happen.

This problem can be solved by using the concept of Process Synchronization.

Simultaneous Usage of Joint Account:

Case-1 :

If there is a single thread allotted to this process and if one user is trying to do a transaction then all the other users of same account have to wait until that user completes his transactions.

This problem can be solved by using the concept of multithreading. By using multiple threads, two or more users can do transaction simultaneously.

Case-2:

But still there would be a problem if two users try to withdraw the amount from the account simultaneously, then the total amount in the account would be inconsistent.

This problem can be solved by using the concept of Process Synchronization.

LITERATURE SURVEY

Normally to achieve maximum performance in message passing programs, the communication and calculation need to be overlapped. But the transformations required in the program add significant complexity and also error-prone. The authors argue that the overlapping of calculation and communication can be achieved easily and consistently by using the concept of MULTITHREADING and executing multiple threads of control on each processor.They also present timing data for a typical message passing application to demonstrate the advantages of the scheme[1].

The authors used detailed simulation studies to evaluate the performance of several scheduling strategies and also explored the trade offs between the use of busy waiting and blocking synchronization primitives and their interactions with the scheduling strategies.Their results show that in situations where the number of processors is less than the number of processes , regular priority based scheduling in conjunction with busy waiting synchronization primitives results in extremely poor utilization of processors. In such conditions, the use of blocking synchronization can increase the performance significantly[2].

The main aim of the Workflow management systems is to automate the execution of business processes by allowing the concurrent execution of multiple instances of a process. But neither Semaphores nor Monitors are appropriate for the workflow applications. So they proposed a method to guarantee mutual exclusion and to enforce correct inter-leavings as defined by the user between current workflow processes. They proposed a protocol that takes the advantage of semantic constructs associated with workflow management to solve some complex problems such as dealing with the coarse granularity of workflow specifications and inherited restrictions[3].

The C Threads package allows parallel programming in C under the Mach operating system. The package provides multiple threads of control within a single shared address space, mutual exclusion locks for protection of critical regions, and condition variables.[4]

It is often desirable, for reasons of clarity, portability, and efficiency, to write parallel programs in which the number of processes is independent of the number of available processors. Several modern operating systems support more than one process in an address space, but the overhead of creating and synchronizing kernel processes can be high. Many runtime environments implement lightweight processes (threads) in user space, but this approach usually results in second-class status for threads, making it difficult or impossible to perform scheduling operations at appropriate times (e.g. when the current thread blocks in the kernel). In addition, a lack of common assumptions may also make it difficult for parallel programs or library routines that use dissimilar thread packages to communicate with each other, or to synchronize access to shared data.[5]

Split-C provides a global address space with a clear concept of locality and unusual assignment operators. These are used as tools to reduce the frequency and cost of remote access. The language allows a mixture of shared memory, message passing, and data parallel programming styles while providing efficient access to the underlying machine. The authors demonstrate the basic language concepts using regular and irregular parallel programs and give performance results for various stages of program optimization.[6]

Methods are provided for thread-agile script execution.If the web browser has a stored state object for the script, thread-agile script execution may be carried out by retrieving the state object; preparing a thread available from a thread pool for execution of the instance of the script in dependence upon the state object and an execution context for the instance of the script; providing, to an execution engine for executing the script, the state object and the prepared thread; and passing the message to the execution engine.[7]

A system for providing cross-exception event handling is provided. The system allows a source thread to throw an event (e.g., exception) as part of structured event handling of a programming language that specifies a target thread. When the event is thrown, the source thread starts a handler thread to handle the event in a current context of the target thread. The handler thread is passed an indication of the event and the target thread and sets its context to be consistent with that of handling events in the target thread.[8]

A card combination which is issued to a single cardholder can be reprogrammed by the cardholder for use by a sub-user to a desired extent with regard to value and time. The cardholder uses a master enabling code to access the programming mechanism. He/she assigns a sub-PIN for use by the sub-user and opens a subordinate account for the allowed credit value within card's total credit value. This subordinate account can be accessed using the sub-PIN. A limited term can be selected during which any transaction using the sub-PIN and using the subordinate account can be conducted. After the expiration of that term the sub-PIN is automatically erased and any balance in the subordinate account is re-credited to the main credit account of the card. Such a multi-user card can also be used in conjunction with a program for varying the value of units stored in the credit account. [9]

Although multi thread libraries have been implemented to provide threads—a unit of concurrent/parallel execution—at user level, there is no thread library that provides both parallelism and portability. We designed and implemented the PPL (Parallel Pthread Library) with the following two requirements in mind: (1) It should permit parallel execution according to the system, and (2) it should provide a common thread environment for many operating systems and be highly portable. We employed a multi-routine approach to realize parallelism, separating the internal structure of PPL into two parts to expedite portability: namely, a virtual processor dependent module and a virtual processor-independent module.[10]

Light-weight processes, threads, are fast vehicles for concurrent/parallel execution in a single program. There are two thread models: kernel-level thread model; and user-level one. Although the kernel-level threads are more lightweight than UNIX processes; it have observed that they are less light-weight than we expected. Therefore, the user-level thread model has attracted attention of researchers. There have been many user-level thread libraries.Parallel Pthread Library (PPL), we are developing, aims at supporting the both. In this paper, we describe PPL.[11]

In many environments, multi-threaded code is written in a language that was originally designed without thread support (e.g. C), to which a library of threading primitives was subsequently added. There appears to be a general understanding that this is not the right approach. We provide specific arguments that a pure library approach, in which the compiler is designed independently of threading issues, cannot guarantee correctness of the resulting code.We first review why the approach almost works, and then examine some of the surprising behavior it may entail. We further illustrate that there are very simple cases in which a pure library-based approach seems incapable of expressing an efficient parallel algorithm.Our discussion takes place in the context of C with Pthreads, since it is commonly used, reasonably well specified, and does not attempt to ensure type-safety, which would entail even stronger constraints. The issues we raise are not specific to that context.[12]

Computer-implemented systems and methods for handling access to one or more resources. Executable entities that are running substantially concurrently provide access requests to an operating system (OS). One or more traps of the OS are avoided to improve resource accessing performance through use of information stored in a shared locking mechanism. The shared locking mechanism indicates the overall state of the locking process, such as the number of processes waiting to retrieve data from a resource and/or whether a writer process is waiting to access the resource.[13]

However, a **lock** /unlock icon replaces the semaphore value, and the owner of the **lock** is shown when the **lock** is **locked**. The two tags for **mutex locks** are llW for **mutex** wait and HI/ **mutex** unlock.[14]

Lint is a command which examines C source programs, detecting a number of bugs and obscurities. It enforces the type rules of C more strictly than the C compilers. It may also be used to enforce a number of portability restrictions involved in moving programs between different machines and/or operating systems. Another option detects a number of wasteful, or error prone, constructions which nevertheless are, strictly speaking, legal. This document discusses the use of lint, gives an overview of the implementation, and gives some hints on the writing of machine independent C code.[15]

The content of this thesis is analysis and transformation of C programs. We develop several analyses that support the transformation of a program into its generating extension. A generating extension is a program that produces specialized programs when executed on parts of the input.[16]

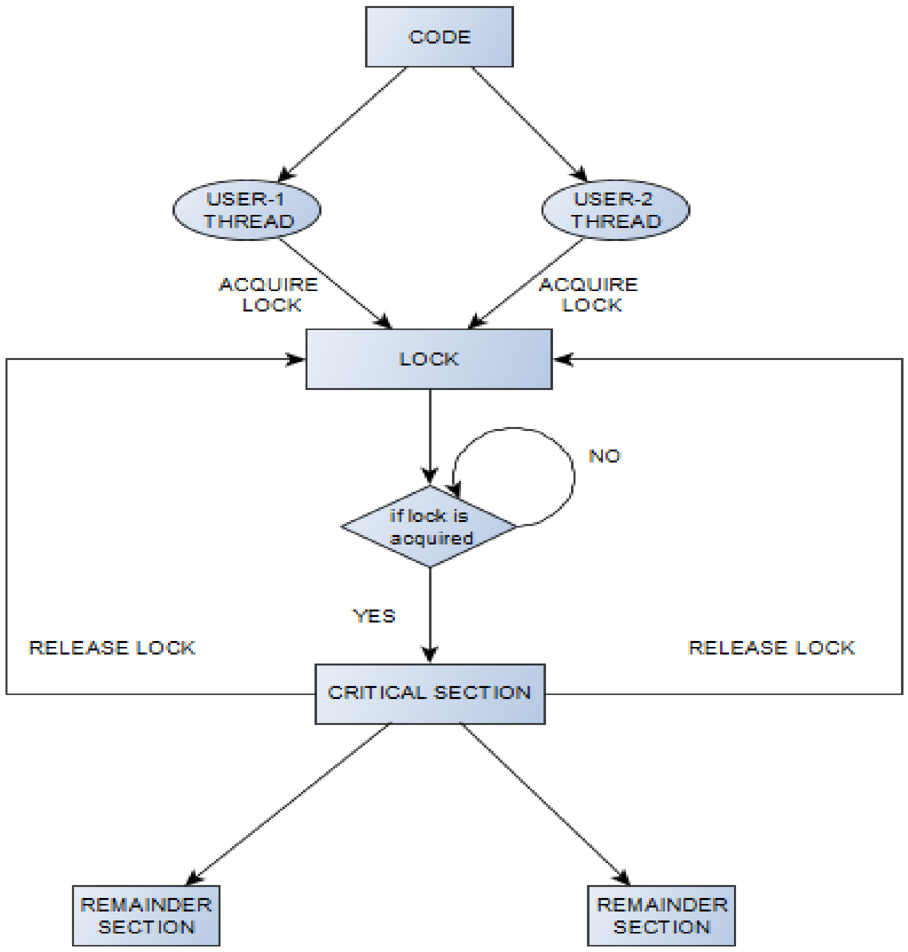
This paper describes a pragmatic but portable fallback approach for creating and dispatching between the machine contexts of multiple threads of execution on Unix systems that lack a dedicated user-space context switching facility. Such a fallback approach for implementing machine contexts is a vital part of a user-space multithreading environment, if it has to achieve maximum portability across a wide range of Unix flavors.[17]

The problem of multiprogram scheduling on a single processor is studied from the viewpoint of the characteristics peculiar to the program functions that need guaranteed service. It is shown that an optimum fixed priority scheduler possesses an upper bound to processor utilization which may be as low as 70 percent for large task sets. It is also shown that full processor utilization can be achieved by dynamically assigning priorities on the basis of their current deadlines. A combination of these two scheduling techniques is also discussed.[18]

In this work, we present a survey of the different parallel programming models and tools available today with special consideration to their suitability for high-performance computing. Thus, we review the shared and distributed memory approaches, as well as the current heterogeneous parallel programming model. In addition, we analyze how the partitioned global address space (PGAS) and hybrid parallel programming models are used to combine the advantages of shared and distributed memory systems. The work is completed by considering languages with specific parallel support and the distributed programming paradigm. In all cases, we present characteristics, strengths, and weaknesses.[19]

The C Threads package allows parallel programming in C under the Mach operating system. The package provides multiple threads of control within a single shared address space, mutual exclusion locks for protection of critical regions, and condition variables.[20]

System Design and Architecture:



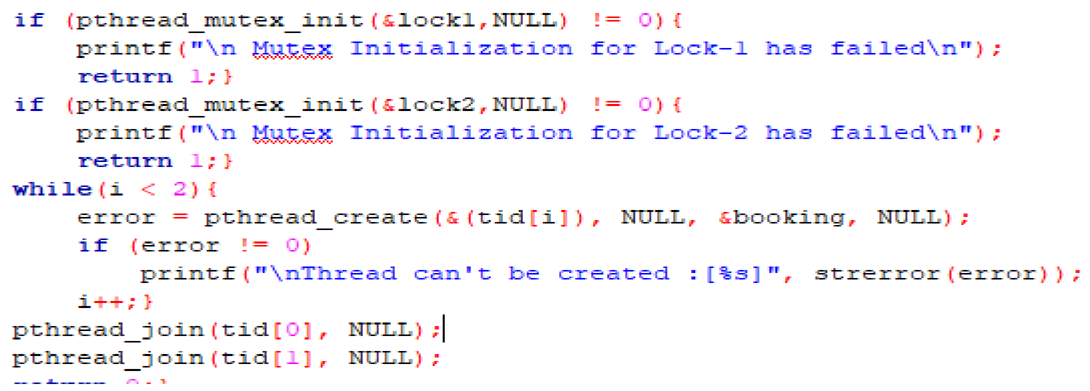
Proposed Algorithms:

Bus Ticket Reservation:

In the process of Bus Ticket Reservation, we have used two threads to represent two different users and we have used two locks, one when taking input from the users and the other when the user proceeds for transaction.

Step-1:

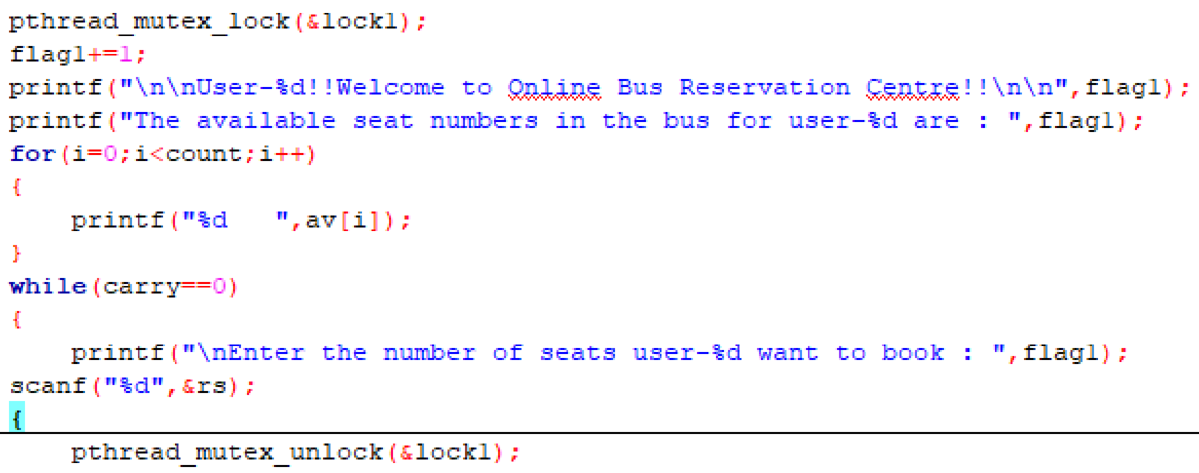
Two threads are created one each for two users.



```
if (pthread_mutex_init(&lock1,NULL) != 0){
    printf("\n Mutex Initialization for Lock-1 has failed\n");
    return 1;}
if (pthread_mutex_init(&lock2,NULL) != 0){
    printf("\n Mutex Initialization for Lock-2 has failed\n");
    return 1;}
while(i < 2){
    error = pthread_create(&(tid[i]), NULL, &booking, NULL);
    if (error != 0)
        printf("\nThread can't be created :[%s]", strerror(error));
    i++;}
pthread_join(tid[0], NULL);
pthread_join(tid[1], NULL);
```

Step-2:

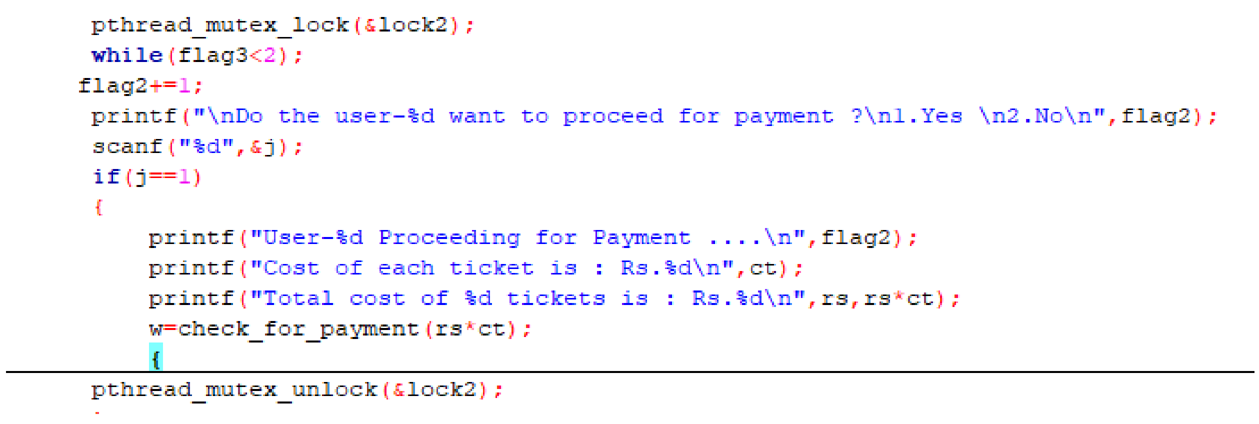
User-1 starts the process first by using locking the lock-1 and enters the input of number of seats required by him and the same number of seat numbers. After entering all these details, user one unlocks the lock-1 and locks the lock-2 and enters the transaction process. While lock-1 is locked, user-2 waits for the lock to be free and enters the process immediately when user-1 unlocks the lock-1.



```
pthread_mutex_lock(&lock1);
flag1+=1;
printf("\n\nUser-%d!!Welcome to Online Bus Reservation Centre!!\n\n",flag1);
printf("The available seat numbers in the bus for user-%d are : ",flag1);
for(i=0;i<count;i++)
{
    printf("%d   ",av[i]);
}
while(carry==0)
{
    printf("\nEnter the number of seats user-%d want to book : ",flag1);
scanf("%d",&rs);
{
    pthread_mutex_unlock(&lock1);
```

Step-3:

As user-1 has acquired the lock-2 for transcation process, user-1 will be in his transaction process and user-2 will be entering the details, after entering the details user-2 releases the lock-1 and waits for lock-2 to be free and enters the transaction process immediately when the user-1 releases the lock-2. And then user-2 completes the process of transaction and releases the lock-2.



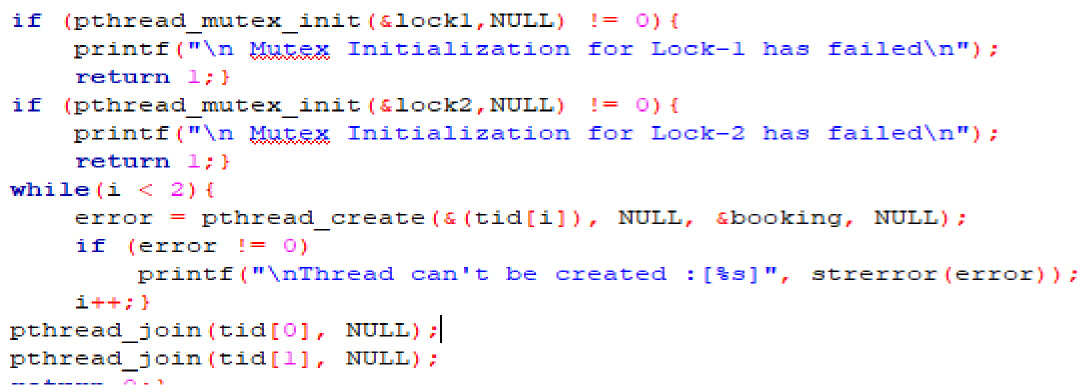
```
    pthread_mutex_lock(&lock2);
    while(flag3<2);
   flag2+=1;
    printf("\nDo the user-%d want to proceed for payment ?\n1.Yes \n2.No\n",flag2);
    scanf("%d",&j);
    if(j==1)
    {
        printf("User-%d Proceeding for Payment ....\n",flag2);
        printf("Cost of each ticket is : Rs.%d\n",ct);
        printf("Total cost of %d tickets is : Rs.%d\n",rs,rs*ct);
        w=check_for_payment(rs*ct);
        {
    pthread_mutex_unlock(&lock2);
```

<u>Course Registration:</u>

In the process of Course Registration, we have used two threads to represent two different users and we have used two locks, one when taking input from the users whether to continue for course registration or not and the other to proceed for course registration.

Step-1:

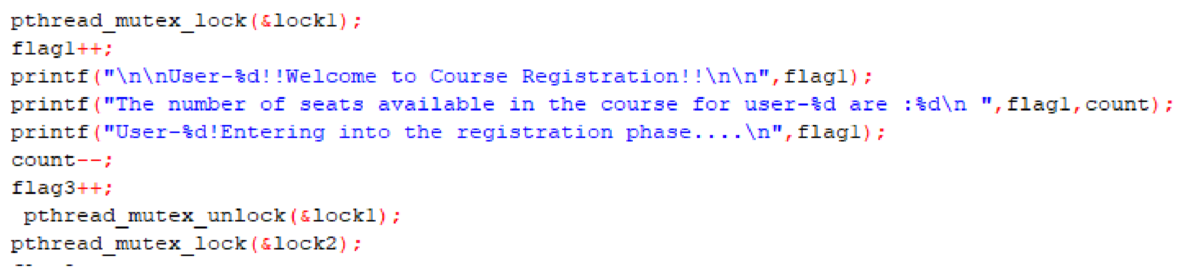
Two threads are created one each for two users.



```
if (pthread_mutex_init(&lock1,NULL) != 0){
    printf("\n Mutex Initialization for Lock-1 has failed\n");
    return 1;}
if (pthread_mutex_init(&lock2,NULL) != 0){
    printf("\n Mutex Initialization for Lock-2 has failed\n");
    return 1;}
while(i < 2){
    error = pthread_create(&(tid[i]), NULL, &booking, NULL);
    if (error != 0)
        printf("\nThread can't be created :[%s]", strerror(error));
    i++;}
pthread_join(tid[0], NULL);
pthread_join(tid[1], NULL);
```

Step-2:

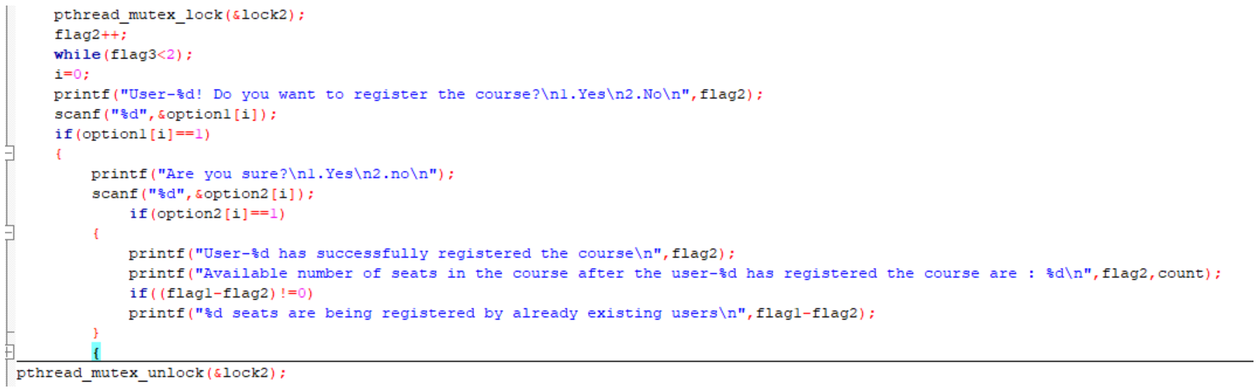
User-1 starts the process by acquiring lock-1 and enters if he wants to continue for course registration and then releases the lock-1 and acquires lock-2. While lock-1 is acquired by the user-1 , user-2 waits until the lock-1 is released and enters the process immediately.



```
pthread_mutex_lock(&lock1);
flag1++;
printf("\n\nUser-%d!!Welcome to Course Registration!!\n\n",flag1);
printf("The number of seats available in the course for user-%d are :%d\n ",flag1,count);
printf("User-%d!Entering into the registration phase....\n",flag1);
count--;
flag3++;
 pthread_mutex_unlock(&lock1);
pthread_mutex_lock(&lock2);
```

Step-3:

As user-1 has acquired the lock-2 for registering the course, he will be in that process and releases the lock-2 after the completion of that process while user-2 will be in lock-1 and releases the lock-1 and waits for the lock-2 to be released and enters the registration process as soon as he acquires the lock-2.And then he completes the registration process and releases the lock-2.



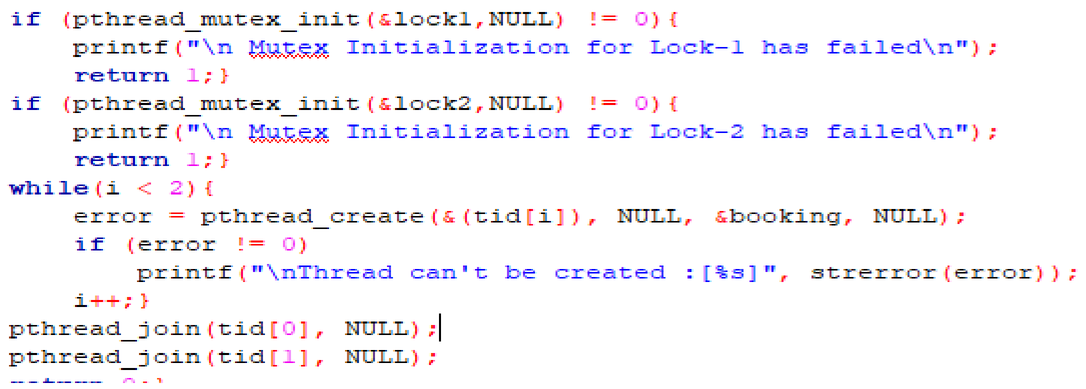
```
    pthread_mutex_lock(&lock2);
    flag2++;
    while(flag3<2);
    i=0;
    printf("User-%d! Do you want to register the course?\n1.Yes\n2.No\n",flag2);
    scanf("%d",&option1[i]);
    if(option1[i]==1)
    {
        printf("Are you sure?\n1.Yes\n2.no\n");
        scanf("%d",&option2[i]);
            if(option2[i]==1)
        {
            printf("User-%d has successfully registered the course\n",flag2);
            printf("Available number of seats in the course after the user-%d has registered the course are : %d\n",flag2,count);
            if((flag1-flag2)!=0)
            printf("%d seats are being registered by already existing users\n",flag1-flag2);
        }
        {
pthread_mutex_unlock(&lock2);
```

Simultaneous Usage of Joint Account:

In this process, we have used two threads to represent two different users and we have used two locks, one when taking input from the users whether to continue for transaction or not and the other to proceed for transaction.

Step-1:

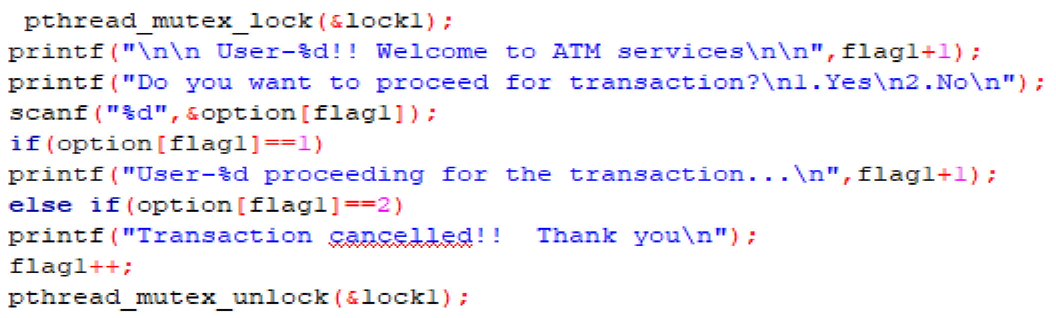
Two threads are created one each for two users.



```
if (pthread_mutex_init(&lock1,NULL) != 0){
    printf("\n Mutex Initialization for Lock-1 has failed\n");
    return 1;}
if (pthread_mutex_init(&lock2,NULL) != 0){
    printf("\n Mutex Initialization for Lock-2 has failed\n");
    return 1;}
while(i < 2){
    error = pthread_create(&(tid[i]), NULL, &booking, NULL);
    if (error != 0)
        printf("\nThread can't be created :[%s]", strerror(error));
    i++;}
pthread_join(tid[0], NULL);
pthread_join(tid[1], NULL);
```

Step-2:

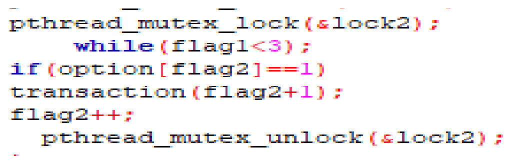
User-1 starts the process by acquiring lock-1 and enters if he wants to continue for transaction and then releases the lock-1 and acquires lock-2. While lock-1 is acquired by the user-1, user-2 waits until the lock-1 is released and enters the process immediately.



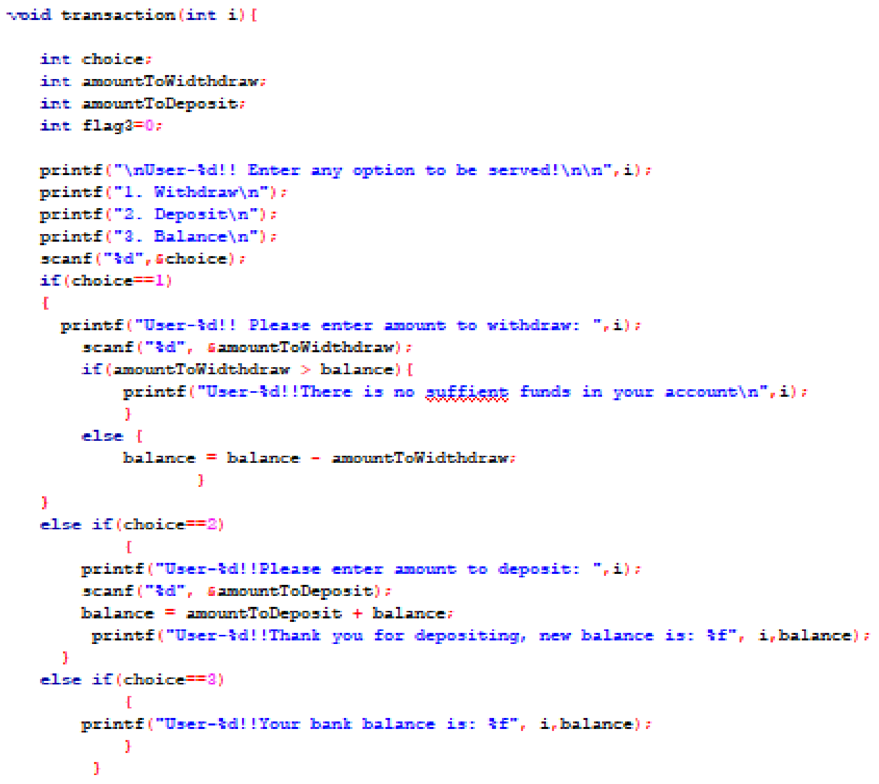
```
 pthread_mutex_lock(&lock1);
printf("\n\n User-%d!! Welcome to ATM services\n\n",flag1+1);
printf("Do you want to proceed for transaction?\n1.Yes\n2.No\n");
scanf("%d",&option[flag1]);
if(option[flag1]==1)
printf("User-%d proceeding for the transaction...\n",flag1+1);
else if(option[flag1]==2)
printf("Transaction cancelled!!  Thank you\n");
flag1++;
pthread_mutex_unlock(&lock1);
```

Step-3:

As user-1 has acquired the lock-2 for transaction, he will be in that process and releases the lock-2 after the completion of that process while user-2 will be in lock-1 and releases the lock-1 and waits for the lock-2 to be released and enters the transaction process as soon as he acquires the lock-2. And then he completes the transaction process and releases the lock-2.



```
pthread_mutex_lock(&lock2);
    while(flag1<3);
if(option[flag2]==1)
transaction(flag2+1);
flag2++;
  pthread_mutex_unlock(&lock2);
```



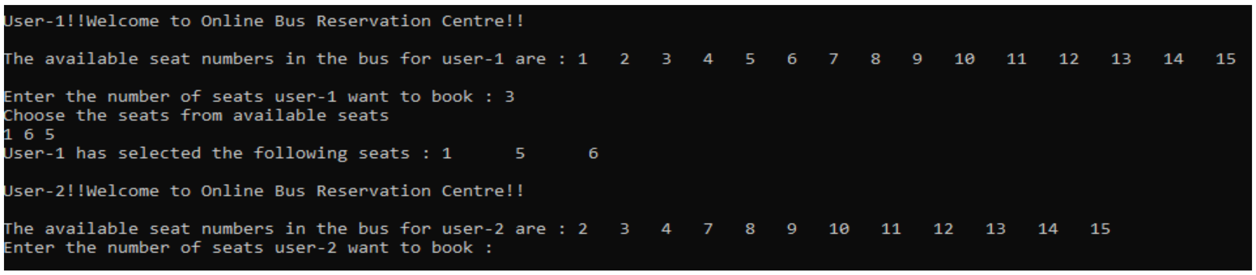
```
void transaction(int i){

   int choice;
   int amountToWidthdraw;
   int amountToDeposit;
   int flag3=0;

   printf("\nUser-%d!! Enter any option to be served!\n\n",i);
   printf("1. Withdraw\n");
   printf("2. Deposit\n");
   printf("3. Balance\n");
   scanf("%d",&choice);
   if(choice==1)
   {
     printf("User-%d!! Please enter amount to withdraw: ",i);
       scanf("%d", &amountToWidthdraw);
       if(amountToWidthdraw > balance){
           printf("User-%d!!There is no suffient funds in your account\n",i);
           }
       else {
           balance = balance - amountToWidthdraw;
                  }
   }
   else if(choice==2)
           {
       printf("User-%d!!Please enter amount to deposit: ",i);
       scanf("%d", &amountToDeposit);
       balance = amountToDeposit + balance;
        printf("User-%d!!Thank you for depositing, new balance is: %f", i,balance);
     }
   else if(choice==3)
           {
       printf("User-%d!!Your bank balance is: %f", i,balance);
           }
        }
```

Results (Comparative Study):

Bus Ticket Reservation:

Condition: Two users i.e User-1 and User-2 wants to book a same ticket.
Case-1 (Without Synchronization):
If there is no Process Synchronization, then the two users end up in booking the same ticket which leads to inconsistency.
If there is no Process Synchronization, then the transactions made by them might lead to inconsistency.

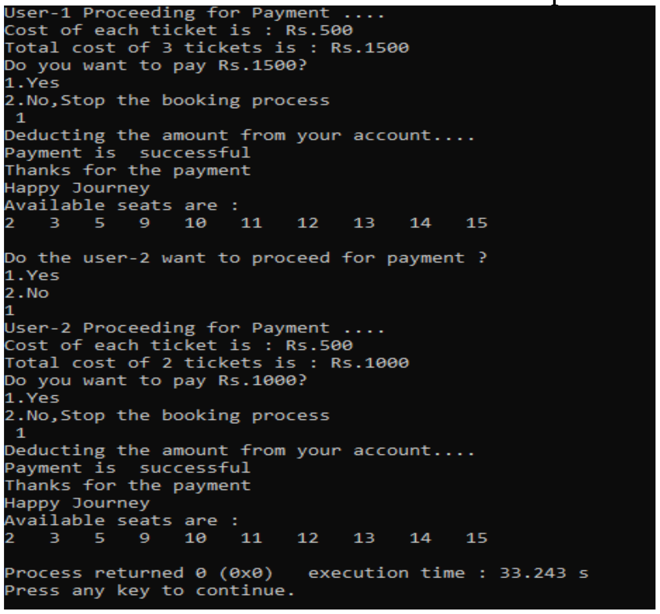
Case-2 (With Synchronization):
In our program, we used two different locks, one for taking the input of number of seats and seat numbers and other for transaction process. So if one user acquires the lock and gives the input, the other user should wait for the lock. So the seats selected by the user-1 will not be available for user-2 ,thereby avoiding the inconsistency.



```
User-1!!Welcome to Online Bus Reservation Centre!!

The available seat numbers in the bus for user-1 are : 1   2   3   4   5   6   7   8   9   10   11   12   13   14   15

Enter the number of seats user-1 want to book : 3
Choose the seats from available seats
1 6 5
User-1 has selected the following seats : 1      5      6

User-2!!Welcome to Online Bus Reservation Centre!!

The available seat numbers in the bus for user-2 are : 2   3   4   7   8   9   10   11   12   13   14   15
Enter the number of seats user-2 want to book :
```

In the above case, User-1 selected the seats 1,5 and 6.
So those seats would not be available for another users, thereby avoiding the inconsistency.

In our process, there is a lock even for transaction process, so when one user is making a transaction, the other users have to wait until the current user completes his transaction.



```
User-1 Proceeding for Payment ....
Cost of each ticket is : Rs.500
Total cost of 3 tickets is : Rs.1500
Do you want to pay Rs.1500?
1.Yes
2.No,Stop the booking process
 1
Deducting the amount from your account....
Payment is  successful
Thanks for the payment
Happy Journey
Available seats are :
2   3   5   9   10   11   12   13   14   15

Do the user-2 want to proceed for payment ?
1.Yes
2.No
1
User-2 Proceeding for Payment ....
Cost of each ticket is : Rs.500
Total cost of 2 tickets is : Rs.1000
Do you want to pay Rs.1000?
1.Yes
2.No,Stop the booking process
 1
Deducting the amount from your account....
Payment is  successful
Thanks for the payment
Happy Journey
Available seats are :
2   3   5   9   10   11   12   13   14   15

Process returned 0 (0x0)   execution time : 33.243 s
Press any key to continue.
```

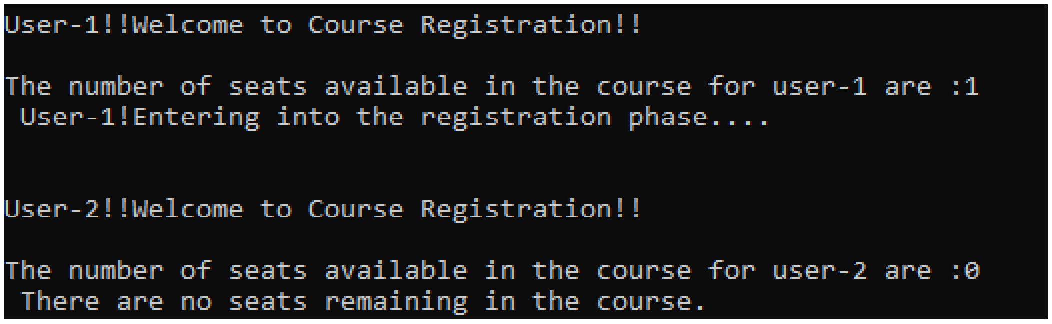
<u>Course Registration:</u>

Condition: There is only one seat remaining in the course and two users, user-1 and user-2 tries to register the same seat.

Case-1 (Without Synchronization):
If there is no Process Synchronization, then the two users end up in registering the same seat which leads to inconsistency.

Case-2 (With Synchronization):

In our program, as we used locks when the first user acquires the lock and tries to register the seat, this seat is no longer made available to other users, there by avoiding the inconsistency.



```
User-1!!Welcome to Course Registration!!

The number of seats available in the course for user-1 are :1
 User-1!Entering into the registration phase....


User-2!!Welcome to Course Registration!!

The number of seats available in the course for user-2 are :0
 There are no seats remaining in the course.
```

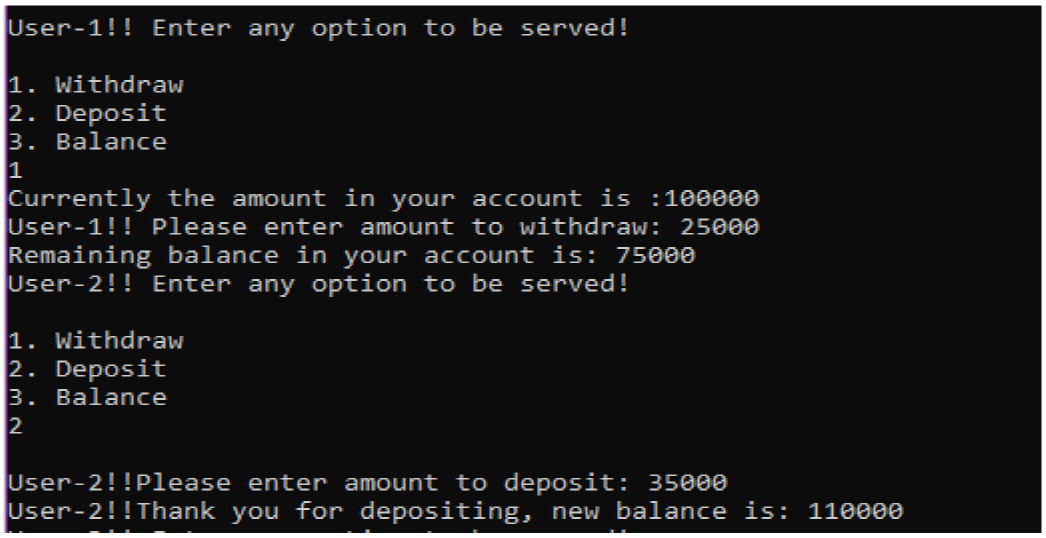
<u>Simultaneous Usage of Joint Account:</u>

Condition: User-1 tries to withdraw amount from the account and the user-2 tries to deposit the amount into the account.

Case-1 (Without Synchronization):
If there is no Process Synchronization, then the total amount in the account will be inconsistent.

Case-2 (With Synchronization):

In our program, as we used locks when the first user acquires the lock and tries to withdraw the amount from the account, user-2 waits for the lock to be released. User-2 performs his transaction i.e depositing amount into the account only after user-1 releases the lock , thereby avoiding inconsistency.



```
User-1!! Enter any option to be served!

1. Withdraw
2. Deposit
3. Balance
1
Currently the amount in your account is :100000
User-1!! Please enter amount to withdraw: 25000
Remaining balance in your account is: 75000
User-2!! Enter any option to be served!

1. Withdraw
2. Deposit
3. Balance
2

User-2!!Please enter amount to deposit: 35000
User-2!!Thank you for depositing, new balance is: 110000
```

Conclusion:

By working on this project about APPLICATIONS OF THREADS AND PROCESS SYNCHRONIZATION, we got a clear idea on how to solve different problems in different methods by using the methods of MULTITHREADING and PROCESS SYNCHRONIZATION. We have clearly gone through the method of solving existing problems i.e Reader/Writer problem, Dining Philosopher Problem , Producer and Consumer Problem , Doctor- Patient problem, Cigarette- Smokers problems and solved some more problems such as Bus Ticket Reservation, Course Registration and Simultaneous Usage of Joint Account. Our proposed algorithms for the three above mentioned problems achieves considerably better consistency when compared to the algorithms which does not include the concepts of MultiThreading and Process Synchronization.

REFERENCES

[1] Felten, E. W., & McNamee, D. (1992, April). Improving the performance of message-passing applications by multithreading. In *Scalable High Performance Computing Conference, 1992. SHPCC-92, Proceedings*. (pp. 84-89). IEEE.

[2] Gupta, A., Tucker, A., & Urushibara, S. (1991, April). The impact of operating system scheduling policies and synchronization methods of performance of parallel applications. In *ACM SIGMETRICS Performance Evaluation Review* (Vol. 19, No. 1, pp. 120-132). ACM.

[3] Alonso, G., Agrawal, D., & El Abbadi, A. (1996, October). Process synchronization in workflow management systems. In *Parallel and Distributed Processing, 1996., Eighth IEEE Symposium on* (pp. 581-588). IEEE.

[4] Cooper, E. C., & Draves, R. P. (1988). *C threads* . Carnegie-Mellon University. Department of Computer Science.

[5] Marsh, B. D., Scott, M. L., LeBlanc, T. J., & Markatos, E. P. (1991, September). First-class user-level threads. In *ACM SIGOPS Operating Systems Review* (Vol. 25, No. 5, pp. 110-121). ACM.

[6] Culler, D. E., Dusseau, A., Goldstein, S. C., Krishnamurthy, A., Lumetta, S., Von Eicken, T., & Yelick, K. (1993, November). Parallel programming in Split-C. In *Supercomputing '93. Proceedings* (pp. 262-273). IEEE.

[7] Batres, S. R., Kizer, G. M., Seth, G., & Silver, A. K. (2014). *U.S. Patent No. 8,694,961*. Washington, DC: U.S. Patent and Trademark Office.

[8] Hildebrandt, T. H. (2017). *U.S. Patent No. 9,830,206*. Washington, DC: U.S. Patent and Trademark Office.

[9] Dethloff, J., & Hinneberg, C. (1989). *U.S. Patent No. 4,837,422*. Washington, DC: U.S. Patent and Trademark Office.

[10] Sakamoto, C., Miyazaki, T., Kuwayama, M., Saisho, K., & Fukuda, A. (1998). Design and implementation of a parallel pthread library (ppl) with parallelism and portability. *Systems and Computers in Japan*, *29*(2), 28-35.

[11] Miyazaki, T., Sakamoto, C., Kuwayama, M., Saisho, K., & Fukuda, A. (1994, November). Parallel pthread library (PPL): user-level thread library with parallelism and portability. In *Computer Software and Applications Conference, 1994. COMPSAC 94. Proceedings., Eighteenth Annual International* (pp. 301-306). IEEE.

[12] Boehm, H. J. (2005, June). Threads cannot be implemented as a library. In *ACM Sigplan Notices* (Vol. 40, No. 6, pp. 261-268). ACM.

[13] Shorb, C. S. (2008). *U.S. Patent No. 7,380,073*. Washington, DC: U.S. Patent and Trademark Office.

[14] Bedy, M., Carr, S., Huang, X., & Shene, C. K. (2000, May). A visualization system for multithreaded programming. In *ACM SIGCSE Bulletin* (Vol. 32, No. 1, pp. 1-5). ACM.

[15] Johnson, S. C. (1977). *Lint, a C program checker* . Murray Hill: Bell Telephone Laboratories.

[16] Andersen, L. O. (1994). *Program analysis and specialization for the C programming language* (Doctoral dissertation, University of Copenhagen).

[17] Engelschall, R. S. (2000, June). Portable multithreading: The signal stack trick for user-space thread creation. In *Proceedings of the annual conference on USENIX Annual Technical Conference* (pp. 20-20). USENIX Association.

[18] Liu, C. L., & Layland, J. W. (1973). Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, *20*(1), 46-61.

[19] Diaz, J., Munoz-Caro, C., & Nino, A. (2012). A survey of parallel programming models and tools in the multi and many-core era. *IEEE Transactions on parallel and distributed systems*, *23* (8), 1369-1386.

[20] Cooper, E. C., & Draves, R. P. (1988). *C threads* . Carnegie-Mellon University. Department of Computer Science.