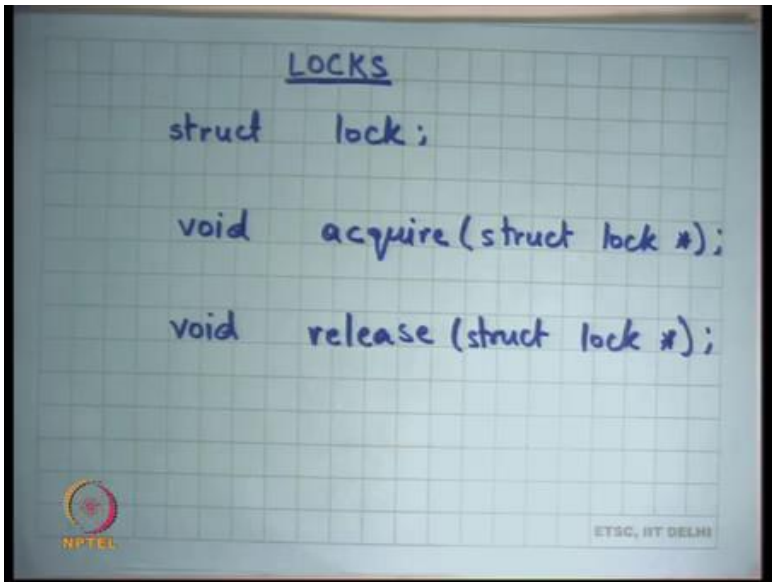
Operating Systems

Prof. Sorav Bansal
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

Lecture – 22
Fine-grained locking and its challenges

Welcome to Operating Systems lecture - 22. So, last time we were discussing locks.

(Refer Slide Time: 00:29)



LOCKS

```
struct lock;

void acquire(struct lock *);

void release(struct lock *);
```

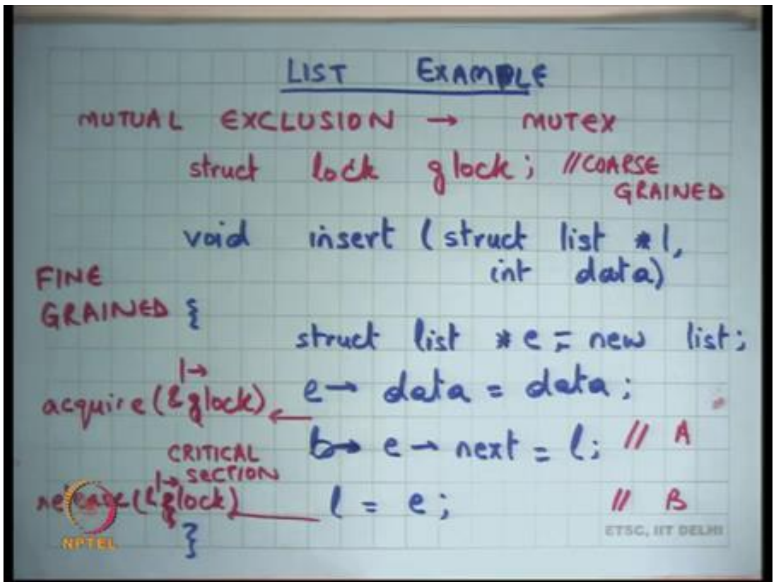
And just to revise, locks is an abstraction, the abstraction has a type called struct lock so you can declare any variable of this type struct lock. And this type has two functions, acquire and release right. And the semantics of acquire and release are that only, the lock can only be acquired once at any time.

If another, if somebody if two calls to acquire a function are made before any release then you know, if two simultaneous calls to acquire are made or two calls are made without a release then one of them will not be able to acquire and only the other one will be. So, one if two threads try to acquire the same lock one thread will be able to acquire it and the other will have to wait, and the other will have to wait till the first thread calls release right. So, that is the semantics of a lock.

And we also saw last time how a lock can potentially be implemented. Basically, we use some sort of hardware support for an atomic instruction in the case of x 86, where we are using the exchange instruction to implement the lock right. We also saw there are two ways to implement the lock, you could either spin wait, or you could either block right. Spinning is useful if you have a multiprocessor and you know they have critical sections are small.

So, you know spinning maybe the better option to do, but if you are on a uniprocessor or if your critical sections are large then blocking may be a more efficient thing to do, right. Because after all blocking is not free, blocking involves executing many instructions also and a spinning is going to finish very soon then maybe spinning is better right.

(Refer Slide Time: 02:05)



LIST EXAMPLE

MUTUAL EXCLUSION → MUTEX

```
        struct lock glock;  //COARSE
                                GRAINED
        void  insert (struct list *l,
                          int data)
FINE
GRAINED {
              struct list *e = new list;
    l→
acquire(&glock) ← e→ data = data;
       CRITICAL
        SECTION
              e → next = l;  // A
    l→
release(&glock) ─ l = e;          // B
        }
```

ETSC, IIT DELHI

And then we were looking at this example of a list and we said look let us say there is a list there is a shared list l and there is this function called insert, that is going to insert this data element into this l. And here is this code which is just going to you know allocate a new list element and initialize its data and its next pointer and update the list right.

And if multiple threads are doing the simultaneously there is a problem because this area, let us say this is statement A and the statement B these are these need to be atomic executed atomically right. So, if two threads are executing these and the inter and the execution interleave on these two instructions then very bad things can happen and we saw some bad things that can happen.

And so of course, locks are meant to solve these kinds of problems and what you are going to do is you are going to put an acquire call here, and the release call here right. And because a lock cannot be acquired twice simultaneously only one thread can be active in this particular region right.

So, this region is also called a critical section right, and the and the lock and this act of actually using locks to ensure that the thread only one thread is active in the critical section is also called mutual exclusion ok. So, locks are a way to ensure mutual exclusion such and in fact, the locks themselves another name for the locks because of this word is basically mutex write that. So, a lock is also called a mutex in many scenarios alright. The other thing is you know you can actually because you know you can actually choose your lock variables.

So firstly, if you want to have mutual exclusion between multiple threats, does the lock needs to be visible to both the threats yes right. So, can the lock be a local variable here of course, not right because local variable is only said private. So, it has to be a global variable clearly.

But even if it is a global variable you know you have a choice you can either have a single, you know you can have a global lock, let us say struct lock g lock global lock and you can use acquire g lock and release g lock ok. So, you can do that, this is this perfectly correct code except that you know you are using one lock for all these insert operations.

So, if there are multiple lists, and you are doing in multiple threads are doing inserts on different lists simultaneously, they will become these operations will become mutually exclusive. So, even though there are multiple lists they did not need to be mutually exclusive, because you are using a global lock you have made them mutually exclusive. And so, this is called coarse grained locking right.

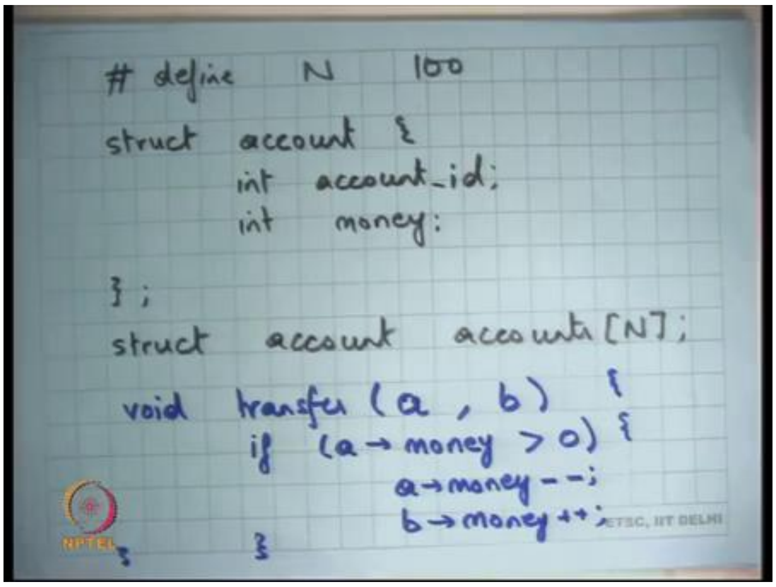
So, you are taking locks at a very coarse granularity, you are basically saying these are all the lists and I have one lock for all the lists right. So, irrespective of which list I am on you are going to take the same lock and you are serializing accesses to all the lists which is an over you know overkill you do not really need all that. So, what is the better option?

Student: (Refer Time: 05:25).

Have a lock per list right. So, we discussed this time last time. So, we can have a lock per list instead of a g lock let us say I have a lock variable inside the structure itself, and I can just basically take that and this is fine grained locking right. In general, coarse grained locking is easier to get correct, but its less performant, finer grain locking is a little harder to reason about, but it has potentially more performance, more concurrency.

So, let us take some more examples to understand this tradeoff between coarse grained locking and fine-grained locking alright. So, let us take this code and let us say you know let us say I am a bank and I am holding these accounts.

(Refer Slide Time: 06:05)



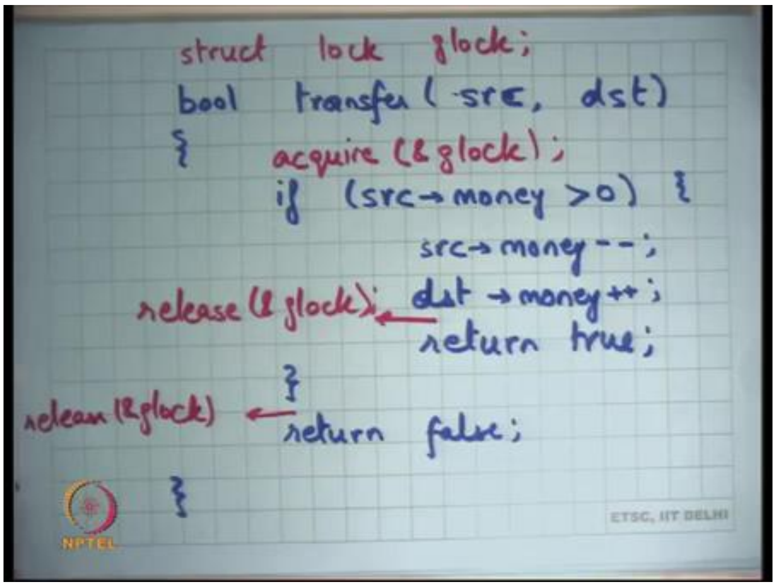
```
# define   N   100
struct  account {
            int  account_id;
            int   money;
};
struct   account   accounts[N];
void   transfer (a , b)   {
           if  (a → money > 0) {
                      a→money --;
                      b→money ++;
}          }
```

So, you know let us say I define this type called struct account, and let us say each account has an account id or account number right and it how much money there is in the account right ok. And let us say you know I define some number let us say its 100 and I have these accounts and let us say this account is also as is, so these accounts are declared as this global array right.

So, these are accounts that I am holding, and each account is having some money and I am a bank and I want to provide this service to my customers called transfer right. So, I should be able to say void transfer, you know let us say account a to account b, I am writing some syntax here. And let us say what it is going to do is if a dot money is greater than 0, then a dot money minus minus b dot money plus plus and that is alright.

So, and let us say let me just write this in a different slide. So, you know let us say I want to provide this transfer functionality, that is going to transfer money from one account to another ok. Let us let me rewrite this on new slide because its little running out of space here.

(Refer Slide Time: 07:53)



```
                     struct  lock  glock;
                     bool  transfer( src,  dst)
                     {     acquire(&glock);
                           if  (src->money >0)  {
                                      src->money--;
           release(&glock); <-        dst->money++;
                                      return  true;
                           }
relean(&glock)  <-         return  false;
                     }
```

And let us also say that this transfer can either succeed or fail. So, let us say this is a bool transfer you know source to destination let me also rename the variable. So, that its clearer, and say if source dot money is greater than 0 source dot money minus minus destination dot money plus plus say let us say I return true.

So, the intent of the transfer function is to transfer let us say one unit of money from the source account to the destination account and it can only transfer if the money was actually non negative right are actually positive and otherwise it is just returns false let us say right. So, let us say this is my transfer function. Now clearly, I mean so assuming that this function is correct if you are running it in a single thread, but if multiple threads are running this code then bad things can happen. What are some bad things that can happen ok?

So, one bad thing that can happen is let us say initially an account, the source account had 1 rupee or 1 unit of money in it. So, two threads come in, they both check that the money is greater than 0, because its 1 and they both try to decrement it and at the end of the day what you will have is that one account the account will have actually money equal to minus 1 right.

And so, two people would have been paid and the you know, and it does not make sense because money should not be less than 0 right. So, you are you are dealing with a nonnegative number here. So, there is a race condition here. So, what is happening is the between the check and update there is a race condition right, what you would have wanted is that the check and the update are atomic which means you would want to use some kind of a lock here right.

The other thing that can of course, happen is you know these statements themselves are not atomic right, minus minus and plus plus involves at least two or three instructions because this going to read it in to register two or three operations, read it in to register incremented and then write it. And we have seen this before that if these instructions that interlude bad things can happen. We call hits variable in the web server that we discussed before right.

So, you know these here are these problems with this ok. So, what will you do? Right, you will use a lock of course, and let us say one way to use a lock is I define lock, struct lock g lock once again it is a global lock and I just say acquire g lock and release g lock right and do I need this, this or do I need to do something else.

Student: (Refer Time: 10:52).

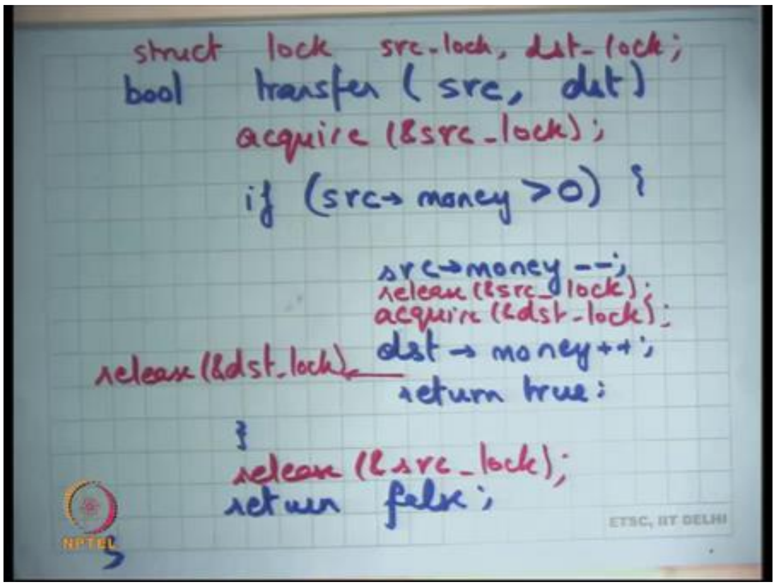
One more. So, I need to put a release g lock here I need to make sure that along all the parts of my program the lock is actually released by before I exit the function right of course, right. So, this is fine this code is correct ok.

But is this the best way to do things well no because let us say you know I have a hundred accounts I want to transfer money to you and you know another person wants to transfer money to somebody else, you know we both of us will get serialized. Only one thread, one transfer instance we can work at any time it is not a very scalable solution, if your bank was you know this function is running inside an inside a server which is you know connected to 180 atms it is not a very scalable solution right. So, what will you do? Right. So, you will.

Student: (Refer Time: 11:40).

So, I got two answers, one is put a lock at the source and the destination separately and the other is put a lock per account let us say let us try the first one the second the first one first right. So, clearly, we need to use fine grained locks, one lock is not good enough. We want to use more locks, so that there is more than concurrency right. So, let us say the first thing I am going to try to do is I am going to say lock the source separately lock the destination separately, alright.

(Refer Slide Time: 12:05)



```
struct lock src-lock, dst-lock;
bool transfer (src, dst)
    acquire (&src_lock);
    if (src→ money > 0) {
            src→money --;
            release (&src_lock);
            acquire (&dst-lock);
            dst → money++;
release (&dst_lock)
            return true;
    }
    release (&src_lock);
    return false;
```

So, let us say bool transfer, source destination if source dot money 0 ok. So, let us say I rewrite this code and let us say I have two locks now right. So, I have struct lock, source lock and destination lock let us say I tried to do this and let us say I say acquire source lock.

So, I am going to say here I am accessing the source. So, I am going to use a different lock to protect all operations on source and I am going to use a different lock to protect operations on destination right. So, what I am going to do is I am going to say acquire source lock and I am going to say release source lock right and let us say acquire right yeah give me a minute. Destination lock let us say release destination, lock alright. And what do I need to do here?

Student: Release.

Release source lock right ok. So, let us say I do this. So, what have I done? I have made sure that all operations which are of this nature which is between source and source itself you are serial you are making them atomic and these operations are make are atomic alright ok.

So, what are some things that cannot happen? It is not possible that there is a race condition within the statement, it is also not possible that you check and you know; so these two statements cannot be concurrent with these two statements themselves, but is this correct?

Student: No.

No, why?

Student: Because it may happen that the source, the source of the parts would have been decremented, but the destination has not got the money now.

Ok.

Student: But now somebody may want that either off source or destination whoever has the money he can take; he will not be able to take.

So, here is one answer, he is saying that look you know it is possible at this point there is no lock held right. So, what can happen is at this point you have released the source locks, you know let us say I am transferring money to you my lock has been released I am about to take your lock, but let us say there is another thread which is just checking the total amount of money in the bank. At this point you will see you know the bank has one look one unit of money less right, that is not a and that may violate some invariants ok.

So, that is a valid point you know, because if there is a thread which is checking how much total money there is you know this transform operation is not atomic anymore. The whole transfer operation is not atomic anymore because you know there is a point where somebody can observe the state of the bank and he will see an inconsistent state right ok. So, agreed, transfer is not atomic. Is there anything else that is wrong with this?

Student: (Refer Time: 15:48).

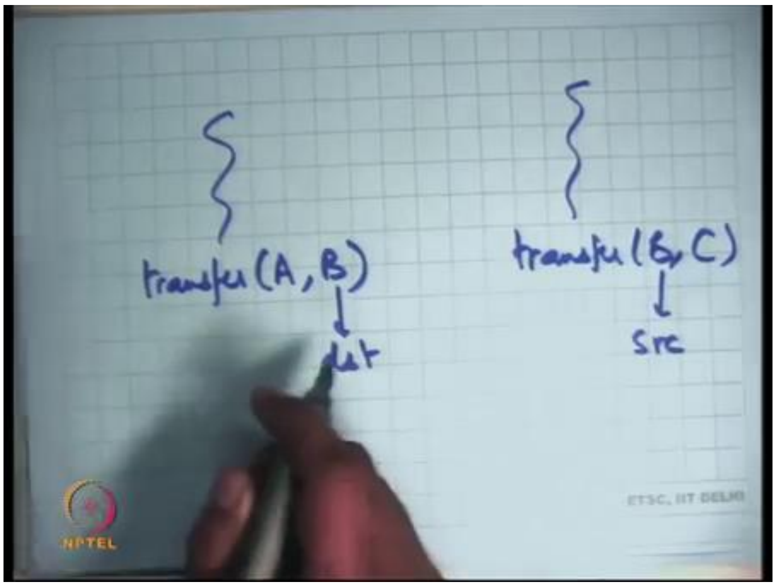
Right. So, let us just talk about that, let us just say there is a transfer in this function in this code in system. So, what if you know I want you know there is a thread. So, I wanted to one thread is trying to transfer money from me to you and there is another thread that is trying to transfer money from you to your friend right.

So, one thread has me as a source and another thread has and you as a destination and another thread has you as the source right. So, one thread has you as a destination and another thread has you as the source. So, what can happen? One of the threads could be operating on your account at this point and another thread could be operating at on your account at this point, is this greater is this I mean do you see a correctness problem?

Student: Yes.

Yes, because there is a race condition between one thread here and another thread here and the race condition exists because you are using different locks right. So, both threads are holding some lock, but they are not all in the same lock and so they can concurrently be accessing different, you know concurrently accessing the same data; however, right.

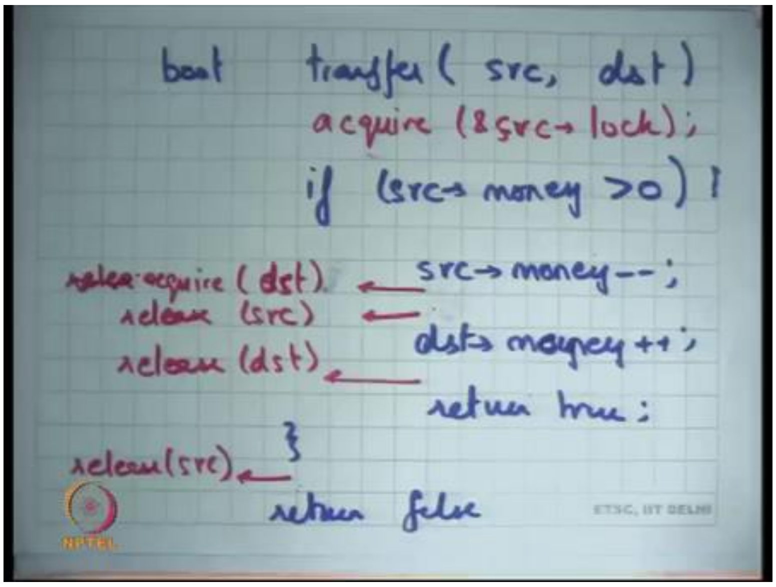
(Refer Slide Time: 16:57)



So, the problem is if one thread says transfer A to B, and another thread says transfer B to C; here B is the destination and here B is the source right. And this code does not ensure mutual exclusion between this code and this code right. And so, there is a problem, you wanted to also ensure mutual exclusion between this code and this code right, if it is possible for these accounts to be the same right.

So, the problem really occurred because I was trying to have a lock per argument and the arguments could alias right. So, it is possible that one person's argument is another person source argument is another person's destination argument. So, the better thing is probably to have a lock per account right ok. So, let us see let us rewrite this. So, let me just rewrite this.

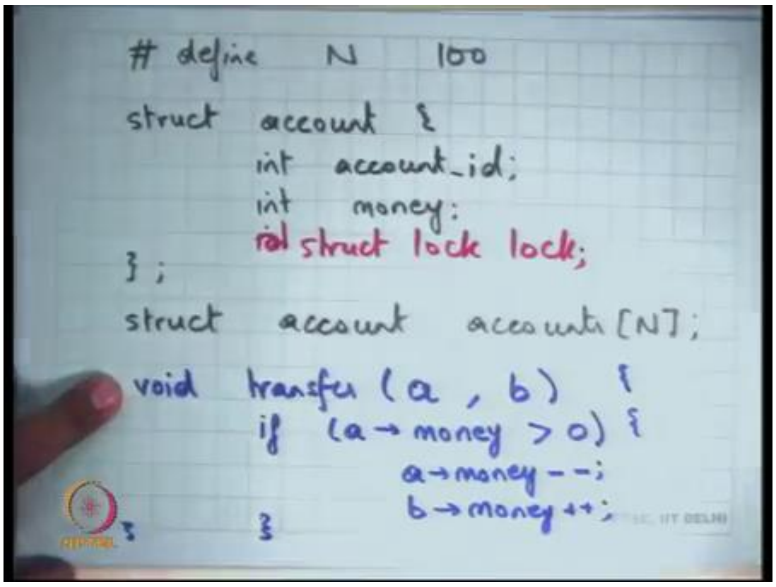
(Refer Slide Time: 17:57)



```
bool    transfer ( src, dst )
            acquire (&src-> lock);
            if (src-> money >0) {
~~relea~~ acquire ( dst)  <---  src-> money--;
      release (src)      <---
      release (dst)  <---        dst-> money++;
                                 return true;
                       }
release(src) <---
                 return false
```

Bool transfer, source destination and now let me use right. So, now, what I am going to do here is I am going to say acquire, source struct lock ok. So, I now have an account a lock per account right.

(Refer Slide Time: 18:51)



```
# define   N    100
struct  account {
        int  account_id;
        int   money;
        ~~ral~~ struct lock lock;
};
struct   account   accounts [N];
void  transfer ( a , b )   {
      if  (a → money > 0) {
               a → money --;
               b → money ++;
}     }
```

So, what I what, the other thing I have done here is in this structure account I have another field called struct lock and so I am going to say acquire source struct lock. Where do I release it?

Student: Before the source release; after the source release.

So clearly you know I could alright. So, I could release it here, I could not release it here, I could release it here or I could release it here. I need to acquire another lock right I need to acquire the destination lock. Where should I acquire it?

Student: Before releasing; Sir acquire before both of them before the transaction starts and then release both of them before that.

So, one answer is acquire both of them here, and release both of them here and release both of them here.

Student: (Refer Time: 19:40).

Right. So, acquire everything in the beginning and release everything at the end just before exiting and you know so the somebody says that is a bad thing because no it seems its it may be correct, but it is not the best that you can do right.

Student: What we can do we can acquire source lock, then we can after source money minus minus we can acquire the destination lock and then after that we release the source lock and then after we have.

All right. So, there are many options, even I mean in the small code you can see there are you know many different sort of options and it is a little confusing, but one thing you can do is you can acquire the source lock here, you can release the source lock then you can acquire the destination lock, then release the destination lock and then release the source lock here right.

So, acquire, release, acquire, release. This has a problem that the transfer is no longer atomic right, because you have released the source lock then you acquired the destination lock, in the middle if somebody observes the state he will see an inconsistent overall state. It is you know you may or may not care about atomicity of the whole transfer function right.

Sometimes you care about the atomicity of the whole operation sometimes you do not care about the atomicity of the transfer function. If the only service that the bank is providing is transfer, perhaps you do not care about the atomicity right. How does it matter?

If for some time somebody sees 1 rupee less in the account right, at the end of the day probably I am going to see I mean this everything correct right, but of course, you know typically you would want something like transfer to be atomic because you know there will be sum let us say you know sum thread which is let us say the sum thread which will just sum all the accounts and that may be running in parallel and so you may care about the atomicity.

(Refer Slide Time: 21:15)

void sum

So, the second option is that you let us say you care about atomicity of transfer. So, the second option was suggested was that you acquire source lock, but then before you release the source lock you acquired the destination lock, then you acquire the source lock then you release the source lock and then you release the destination lock. So, here is the; here is the suggestion you release no sorry.

Student: Acquire.

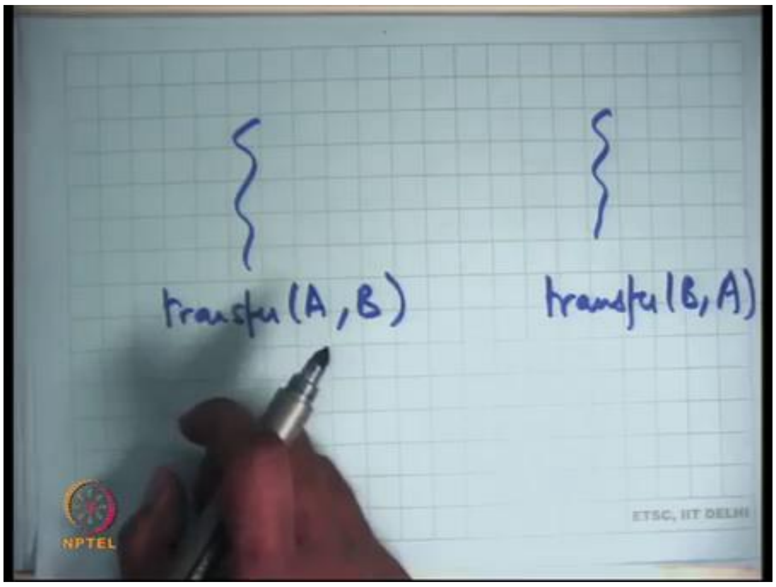
You acquire let me just shortcut it shorthand it, so you acquire destination you release source and then you release destination and what do you do here? Release source all right. So, notice that our code has become slightly, slightly tricky because at this point I have acquired one lock and I am trying to acquire the other lock right and I am I have to do that because this operation which involves accessing multiple sort of accounts needs to I have both, you know for atomicity I need to hold at least hold on to at least one lock I cannot just release both locks alright. So, let us see what can happen.

Student: Like one second.

So, if the source and destination are same then you can see a problem. You know you acquire the same lock and you try to acquire the same lock and what is happens? The thread deadlocks right it will never be able to acquire the same lock twice, that is abstraction the lock. The same thread calls require twice it just dead locks it will never be able to proceed ok.

So, that you know you may say that is you know that you can check you can just say if source is not equal to dead then do this otherwise you do not do anything right. So, that is a very interesting observation, but let us just take this observation a little further let us say one thread is transferring account from me to you and another thread is transferring account from you to me right. So, what will happen? The first so let me just write it before I discuss.

(Refer Slide Time: 23:37)



Let us say this is doing transfer A to B, and this is doing transfer B to A alright. So, what will happen? The first thread will acquire as lock and it will try and may and it may be somewhere here and its possible at the second threads run simultaneously and it requires B’s lock right.

Student: (Refer Time: 24:03).

Now, the first I will try to acquire B’s lock and it will wait for the second thread to release B’s lock. And the second thread will try to acquire A’s lock and it will wait for the first thread to release A’s lock. What do you have?

Student: Dead lock.

Dead lock because the first thread has acquired A’s lock and is waiting for B’s lock to get free. The second thread has acquired B’s lock and is waiting for as lock to get free, both the threads are waiting for the other thread to release a lock, but they will never be able to none of the threads will be able to release any lock because they are both stuck at an acquire right and so, you have a deadlock right alright.

So, this is a big problem with fine grained locking, we you have you know big recall that coarse grained locking had had no such problem you just had one big lock you know, but now because you now you what you did was you somehow said you know maybe per account locks are better. So, you had per account locks, then you had some operations which involved multiple accounts.

So, when you have multiple accounts you need to take multiple locks at the same time when you have to take multiple locks at the same time then you have these problems of cyclic dependency right. There is a cyclic dependency because a there is a cycle because I am holding my lock and I am trying to waiting for another lock and that guy is holding the lock that I am waiting for and he is waiting for my the lock I am holding. So, there is a cyclic dependency in the lock acquisition order alright ah.

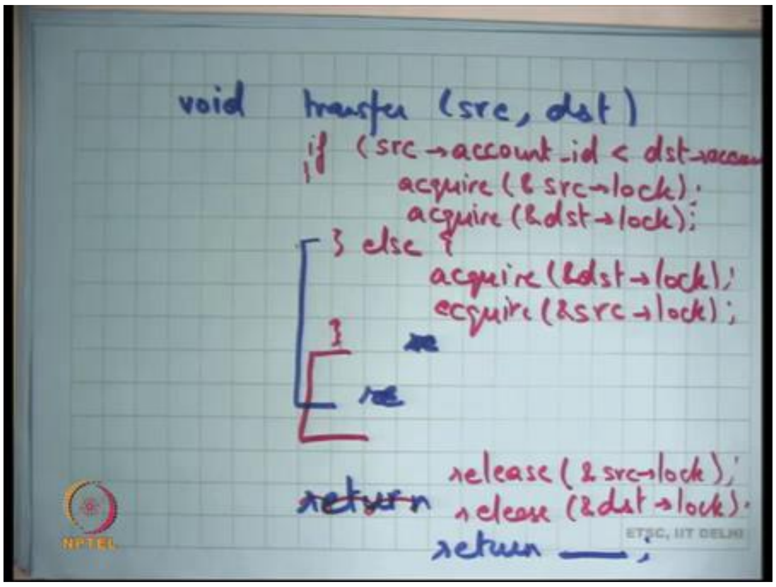
So, how do you solve something like this? So, clearly this code is not correct it is from a data race freedom standpoint. So, there are no races in this code transfer function appears atomic, but it has a problem that the function can actually deadlock, the system can actually deadlock. So, what is this; what is the solution?

The solution that is typically taken in an operating system and so these the deadlock is a very common problems seen in many different kind in scenarios, but in an operating system the typical solution to this kind of a problem is to have an order on the acquisition of locks.

So, you always say that if you ever have to do an operation which involves multiple entities and which involves taking multiple locks, then you will take those locks in a certain order right. So, one option is that you say that I am going to take. So, you and that order can be completely you know arbitrary, you can decide whatever order you want, but assuming that all if you are; if you are taking multiple locks and you are taking them in a certain order then they will never be a cyclic dependency right.

So, let us say one thread is transferring money from you to me to you and another thread is transferring money from you to me, then in both threads there will be an order right. Depending on what the order is either my lock will be taken before lock you, or your lock will be taken up before me. But it will not, never happen that one thread is taking my lock before you and another thread is taking your lock before me right. So, let us say I order it on account id right. So, if I order it an account id the correct version of this code may look like something like this.

(Refer Slide Time: 27:11)



```
void  transfer (src, dst)
        if (src->account_id < dst->accou
            acquire (& src->lock);
            acquire (&dst->lock);
        } else {
              acquire (&dst->lock);
              acquire (&src->lock);
        }

~~return~~   release (& src->lock);
             release (&dst->lock);
             return ___;
```

Void transfer, source, destination and let us say you know I am let say this is my this is all my code which has all returned somewhere it has a return. So, at the end of it has return. So, I am not going to write the whole code, but let us just look at the; look at the locking behavior.

So, I may want to do something like this, if source dot account id is less than dst dot account id, then acquire source dot lock and then acquire destination dot lock; else what? Acquire destination dot lock and then acquire source dot lock and then after you have all these locks you can do whatever you like here, because you know these operations will be atomic with respect to each other and then at the end of the day you can say release, do the release need to have an order.

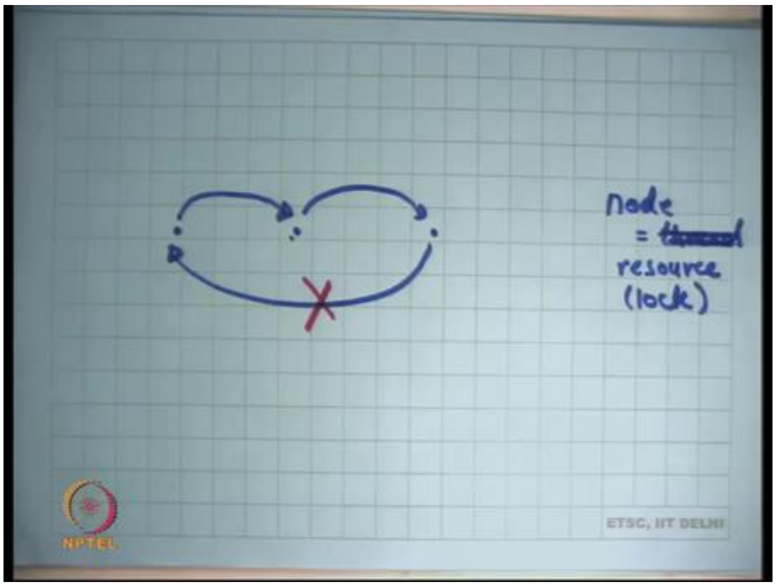
No, they do not have release does not need to have an order release is just a non blocking operation, you just say you know any order is fine and then you return. Now, whatever value you want to return here ok.

So, this code is correct, its fine grained each lock, each account has one lock and so and you have made sure that any operation that involves multiple accounts and needs to be atomic takes all the locks, all the corresponding locks and you have made sure that those locks are in a certain global order right. So, this order needs to be a global order and all threads respect that global order right.

So, let us say; let us say let us consider the same situation, let us say I wanted to transfer money to you and you know in one thread, and then there is another thread that wants to transfer money from you to me then in I in both cases let us say you know my account id was smaller than yours then both cases my lock will be taken before yours.

So, let us say the first thread takes my lock and then he is the second thread will also try to take my lock first and so it will block right there right. So, the second lock will not be taken, I mean so there will not be a cyclic dependency in other words you know basically a deadlock occurred.

(Refer Slide Time: 30:33)



So, let us say each of the, each let us say if I were to draw a graph and each thread. So, and each node in this graph is equal to a thread right, and let us say let me draw edges in this graph, if I hold I am I need a resource if I can potentially request for a resource that is held by the other thread right. So, I draw an edge between two nodes if I can request for a resource that is held by the destination node.

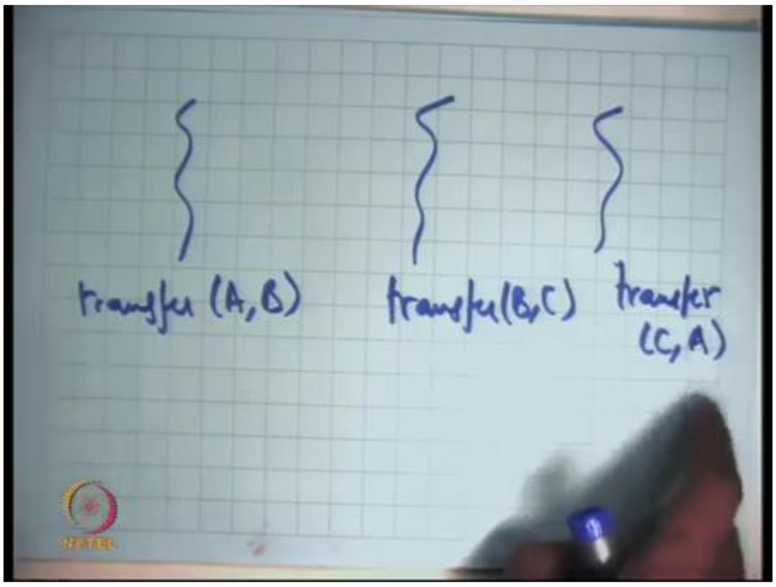
So, if the source thread can request for a source that is held by the destination thread, I draw an edge and the deadlock could only occur if let us say. So, let us say this could request for this and let us say this could request for this right. So, deadlock can only occur if there is cycle in this graph. What I have made sure is that these edges can only be forward edges because I have imposed a total order on the resources.

So, I can never be holding a resource. So, let us say these nodes are now resources right. So, the other way to look at it is a node is a resource, and resources you know I am using the word resource for a lock in this case. So, I draw an edge from one resource to another resource, if it is possible for me to hold a resource and then request for the other resource alright. So, it is possible for one thread to hold one resource and request one other resource as an edge this is the directed edge from that. So, and a deadlock can occur if there is a cycle in this graph right.

So, in this in the example in our accounts a deadlock could occur because it is possible that one thread is wait holding account x and requesting for account y and another thread is holding account y and requesting for account x, but if I put a total order on these resources, on these locks and I say that it is never and I make sure that such backward edges never exist right.

So, if I disallow backward edges, basically saying then there can be never be a cycle. So, that is basically why there can never be a deadlock if you completely order all these sources. So, you make sure that your resources are acquired in a certain global order, you will never have a backward edge in your resource acquisition graph right. And so there will never be a deadlock. So, you can you know you can extend this example to three threads.

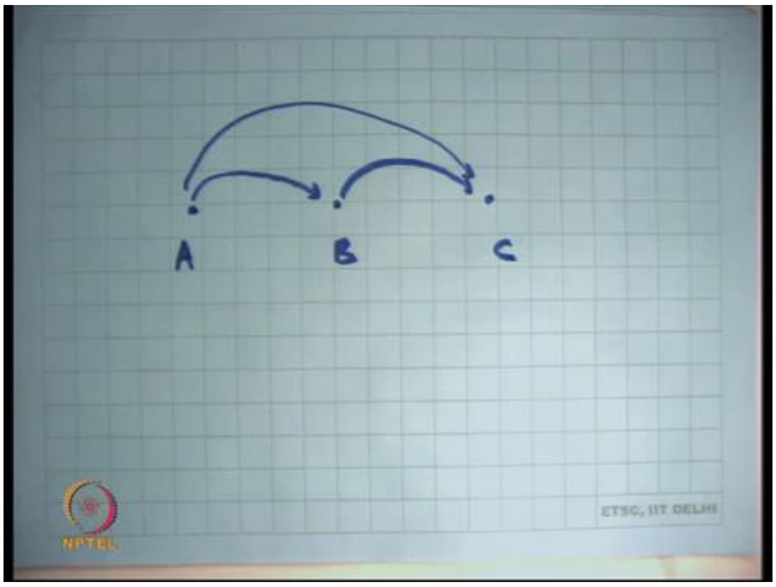
(Refer Slide Time: 33:11)



You know one saying transfer A, B; transfer B C and this one saying transfer C A right. So, once again you know if you do not have a total order in your locks by account id then a deadlock can occur right, same thing I hold account A, wait for account B the other one holds for account B waits for account C and the third thread completes the cycle, he holds the account C waits for account A and all three threads; now dead lock right.

But if you have a total order then there will be no edge from C to A, there will only be edges from A to C right. So, if I were to draw the resource.

(Refer Slide Time: 33:55)


$$\begin{tikzcd} A \arrow[r, bend left] \arrow[rr, bend left] & B \arrow[r, bend left] & C \end{tikzcd}$$

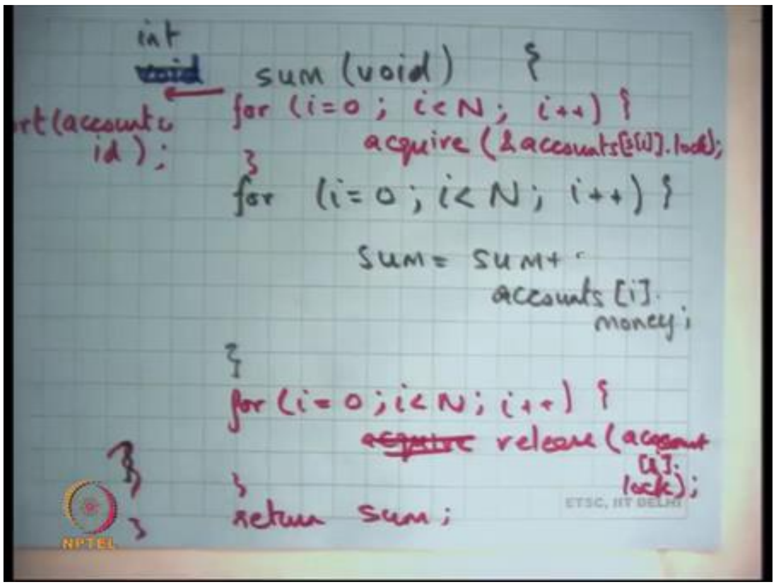
The let us say these are the locks and this is A’s lock this is B’s lock and this is C’s lock this is possible and let us say A is less than B is less than C, this is possible this is possible, but this is not possible; instead I made sure I have written my code such that only this is possible right.

So, because I have ordered my locks, I can only have forward edges in my resource dependency graph and so they can be noted locks alright. So, let us see notice that the code that we had come up with was that will acquire source lock, then you know very carefully we will acquire destination release source lock after we will acquire destination lock and we will try to acquire the destination lock as late as possible and all that thing right.

But you know it looks very nice it would be nice if we could do this because it you know as you have you have probably thought that it is going to increase concurrency. But in practice it is not possible because you also need to order these locks in a certain way. So, you know basically we came back to the same solution that acquired both the locks in the beginning, right because you also need to check the order and then acquire the locks in that way right.

So, even though the most concurrent way to do things would have been acquired locks as you need them, but because you had to have a total order on the locks you could not do that, you just had to take all the locks in the beginning and in fact, you have to take those locks in a certain order right. So, let us take another example, let us say in the same system there is another thread running that is called sum all right and let us say this right.

(Refer Slide Time: 35:27)



```
int
~~void~~    sum (void)    {
rt(accounts      for (i=0 ; i<N ;  i++) {
    id );              acquire (&accounts[i].lock);
                 }
                 for  (i=0 ; i< N ; i++) {

                         sum= sum+
                                accounts [i].
                                        money;

                 }
                 for (i=0 ; i<N ; i++) {
                         ~~acquire~~ release (account
                                              [i].
}                                             lock);
                 }
                 return  sum;
       }
```

So, let us say the sum returns an integer and it just what it does is it just goes over all the accounts. So, for i is equals 0, i is less than N, i plus plus sum is equal to sum plus accounts, I bought money right. So, that is it this is my code and of course, this function is also being done by a thread concurrently with other threads that may be running the same function sum or that may be running the transfer function alright.

And let us say you know I am going to make sure that you know I want to make sure that the sum stays constant. So, the total money in my account in my bank is should remain constant if we just we are transferring money across each other and so make to making. So, clearly if you want that to be true then the transfer function should have been atomic, and we have made that atomic itself. But can I just write the sum function like this? Would that be ok? No, why? What can happen?.

Student: (Refer Time: 36:48).

The sum function could run concurrently with the transfer thread and the transfer thread could have decremented some money from some account and may not have incremented the money in some account and so the sum may be incorrect right. So, what do you need to do? You want to make the sum operation mutually exclusive with the transfer operations and but so, in a coarse grained world it would have been very easy you would have had just one lock and both this transfer function and the sum function would have taken that lock and that would have made everything completely mutually exclusive. Now in the fine-grained world which locks did you take and when?

Student: All the locks, all the.

So, one answer is let us take locks for all accounts everybody agrees?

Student: We think is this two in order.

So yes, I think we need to take locks of all accounts because that if we miss even one account then you know our total sum will be wrong. Now, where should I take these locks?

Student: (Refer Time: 37:49).

Should I take the lock on demand around this?

Student: No, it has to be taken around (Refer Time: 37:55).

You, if you take the locks around this then you know you are violating the order right. So, here is an operation that just is not operating on two accounts in operating on n accounts right and it and this operation also needs to be atomics with respect to other threads. So, what are you going to do? You are going to take all the locks apriori then you are going to do the function and then go to release all the locks after that you cannot take the locks on demand because you need a certain order on the global order on the locks right.

So, what you?

Student: (Refer Time: 38:29) take the way that their accounts i dot id is always lesser than accounts i plus 1 dot id.

Sure. So, let us talk yeah. So, let us first look at what we can do. So, let us say so, the right thing to do would be for i is equal to 0, i is less than N, i plus plus acquire. Accounts i dot lock this is not correct either right because I am not obeying any order on the lock, I should have really ordered it by. So, you know what we will need to do is let us say you have another array which sorts the accounts by id right and so and that sorted order is what we use here right.

So, you just take the locks in a certain sorted order and the sorted out for order is let us say in the order of the account id ordered by the account id and similarly you are going to just release it here. Alright here you do not care about the order right and return sum right. So, notice that the array itself is not necessarily sorted by the account id right, you just generate a permutation which is the sorted permutation s which is you know it sorts the accounts by this account id and then you basically do that right and so, you need to take them apriori.

Of course, now you can you know you may say, but you know what perhaps the better thing would have been that you know just arrange your accounts in a certain order apriori in the order of account id or allocate the accounts in order of the account id. And so that way you know you do not need to do the sort operation and you can probably even take the accounts in on demand in that case sure.

Student: So, that even in once we have sorted that now we can take the locks on demand (Refer Time: 40:52).

We have no t sorted the accounts we have just you just got a permutation. So, notice that I am just saying accounts si. So, s is a permutation, which is a sorted permutation now which basically says what will be the sorted order of these accounts.

Student: Sir (Refer Time: 41:27) we can also.

So, what you could do is you could say you know accounts si here and so then you can take the they take the.

Student: We have to make sure that we do not release we always hold on to one.

Right you just have to make sure that you always hold on to at least one lock right. So, that is true ok. So, the question is cannot we just say acquire account i and release account i right here ok. So, I see some head saying no. So, why?

Student: That is it we have to transfer from A to B (Refer Time: 41:44).

Right.

Student: And when A is less than I and B is greater than I then we have already added the amount of the previous amount of A into the sum, but now there is a transfer (Refer Time: 00:00) also then.

Right. So, basically you know let us say there is a transfer going on from account A to account B and now sum takes lock on account A and you know let us say before the transfer started and it got the lock and it will you know read some value then transfer happens then it tries to get an account lock on account B and then you know it gets it and so, it will get a wrong sum it will probably get it can even get you know something which is bigger than your actual amount right ok.

So, in general figuring out where to put the locks, how to put the locks, whether to put the locks or not is you know hard. In fact, lock is just one way of ensuring correctness right. Notice that locks are just basically what we did we said, this code is not correct if executed concurrently, let us make this code mutually exclusive. One way to make things mutually exclusive is locks and then let us use locks and then we said where to put the locks that itself seems to be a hard problem.

In fact, if there are other ways to make sure that the code is correct which does not involve taking locks where you can just structure your code in a very careful way such that without taking locks where you can actually make sure that code is correct alright and we are going to discuss some as the as we go along the course.

So, in general ensuring correctness and concurrency is hard, but and you know you would typically want to avoid things that are hard to understand because they are you know not just its hard to code, but it is hard to maintain overtime. Typically, programmers follow a discipline, and that is the locking discipline.

And the locking discipline basically says that firstly, you should think about your system as a whole and figure out where do you want to put the locks or how do you want to associate the locks. You want to associate a lock per account do you want to associate a lock per pair of accounts, do you want to associate a global lock with the whole all whole area of accounts these are all possibilities and you can you can choose one on them.

Once you have done that once you have decided that then you ensure that for all shared regions for all regions which can access this shared data or access the region that is predicted by that lock that lock is always held before executing that region. So, if there is any code that can access a shared region then the lock corresponding to that shared region should be held, by that lock right. And if there is an operation that acquire that needs to touch multiple shared regions and needs to be atomic then locks of all these accounts should be acquired apriori or all these regions should be acquired apriori and released after that. So, this is so, and they should be acquired in a certain global order right.

So, these are this is basically the discipline that if you follow you will get correctness, may not be the best way to get correct correctness, but it is a reasonably good way of getting correctness. So, in the example that we saw there was transfer and there was some right in transfer also we did the same thing we said we are going to operate on two accounts.

So, let us take the accounts let us take that locks for both the accounts apriori then do the operation and then release both the locks. Similarly, in the sum operation we said you have to touch all these shared objects. So, let us take plot for all these shared objects apriori, touch all these shared objects and then release all these shared objects after that right. So, in both cases we were just using the locking discipline.

Student: Sir.

Yes.

Student: Is eventual consistence tolerable in OS?

Is eventual consistency tolerable in OS? So, what is eventual consistency you know these are. So, in general eventual consistency. So, eventual consistency has been is something that you know distributed systems people talk about, where you say that let us say the two people who are accessing the same thing eventually it will get consistent, but sometimes you may see inconsistent values right and you know so, let us so you know when you are designing a system you have to worry about what does consistency mean and what kind of consistency guarantees you need to provide to the user of your system right. Let us just talk in concrete terms before we know.

So, those are those become sort of much more sort of high-level discussions. Let us just talk in concrete terms it looks let us look at these examples and see you know what you need. So, in this case you know this is the consistency level you need, and this is how you do use it.

So, since we are talking about a bank you know if there is some thread is going to run, the sum thread is going to have to take all the locks and basically; that means, that all the transfer threads basically gets stopped for that amount of time right till the sum gets computed. Or, in fact, till the other locks get taken are there better ways to do that? Yes, there are better ways to do that. In fact, banks are not, you know bank accounts are not implemented in memory I am just using a toy example right you would typically implement bank accounts in a database and database has other ways to do concurrency control right because.

So, let us understand, I will just give you a brief understanding of why databases con the way the database does the concurrency control is different from how an operating system needs to do concurrency controls a database needs to acts you know operates on disk blocks because it also cares about persistence you know data should remain across reboots.

Disk accesses are much much slower than memory accesses right and so the overhead to do concurrency control is much smaller in a database. And so you can do more fancier things to do concurrency control in database and typical way of doing the concurrency control database of what is called transactions right.

So, we are going to discuss some of this later. So, databases do transaction con concurrency to a different way, an operating system cannot do transactions because you know it is too costly to do transactions for memory accesses right. And in fact, you know more recently there are there you know modern hardware is coming up with what is called transactional memory that allows you to do some of this.

And you know these are all advanced topics that we may have time to cover later in the course ok. But, by and large today locks is the way to do concurrency control and you are right I mean, if there is something like this, then the sum thread is actually you know basically bringing everything else to sort of halt for that amount of time. But that is basically what the price you will have to pay anyways.

Good; no more questions alright. Let us stop and continue the discussion next time.