

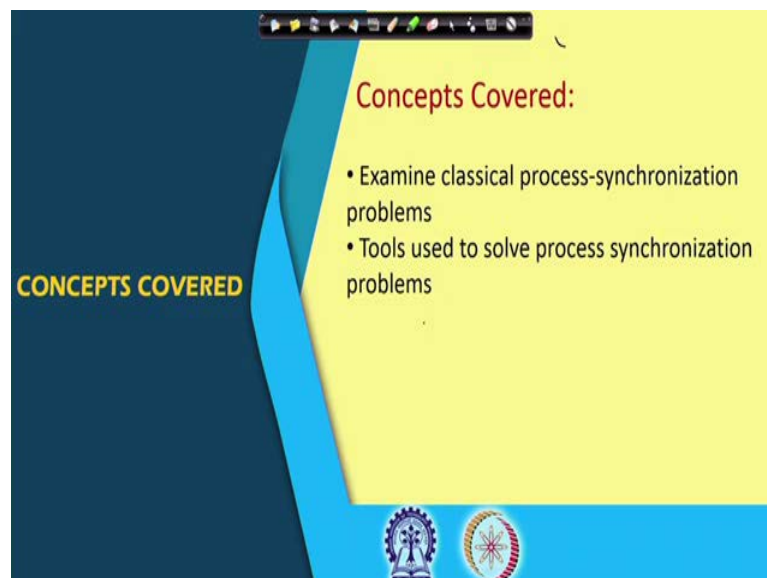
**Operating System Fundamentals**  
**Prof. Santanu Chattopadhyay**  
**Department of Electronics and Electrical Communication Engineering**  
**Indian Institute of Technology, Kharagpur**

**Lecture – 35**  
**Synchronization Examples**

So, in our last class we were looking into some Synchronization primitives like using some techniques like algorithmic techniques, then some semaphore based techniques, then some hardware assisted techniques etcetera. And ultimately we have seen that at a higher level we can have the semaphore and monitor as the two synchronization primitive that is that are available for general programmers. So, in next few classes we are going to look into some examples of this synchronization problem and in our last class we have also seen how we can do some process synchronization using semaphores, how to solve this mutual exclusion problem and so on.

So, today we will be looking into some classical synchronization problems that is some problems which are been acknowledged worldwide by operating system fraternity that these are some fundamental problems that can occur in many situation in an operating system. So, they need to be solved. So, if we know the solution to these problems using say semaphore or monitor, then we can use it for may all those types of problems. So, these classical problems that we will be looking into.

(Refer Slide Time: 01:43)



The slide features a dark blue background on the left with the text 'CONCEPTS COVERED' in yellow. The right side has a light yellow background with the text 'Concepts Covered:' in red, followed by a bulleted list. At the bottom, there are two circular logos: the Indian Institute of Technology Kharagpur logo on the left and the Indian National Flag on the right.

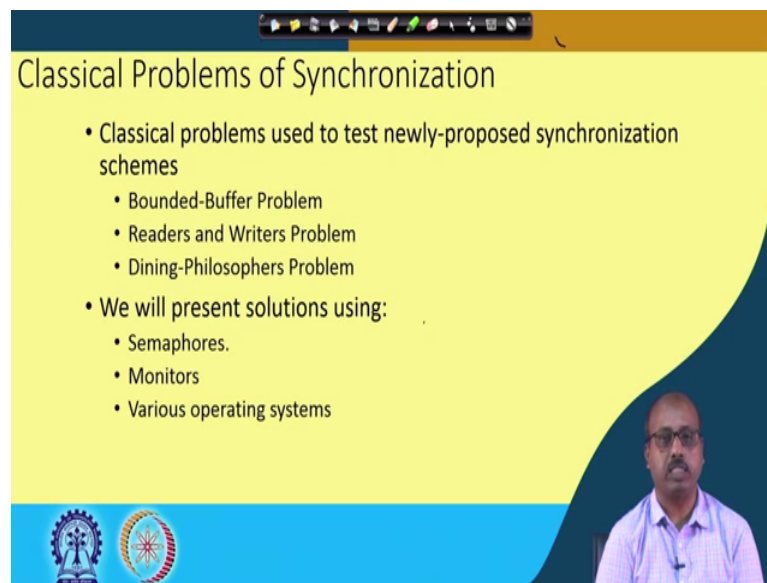
**CONCEPTS COVERED**

**Concepts Covered:**

- Examine classical process-synchronization problems
- Tools used to solve process synchronization problems

So, there are mainly some tools that are the problems that are producer consumer problem, then readers writers problem so on. So, we will examine classical process synchronization problems and we will see the how the tools can be used to solve this synchronization problems.

(Refer Slide Time: 01:58)



The slide is titled "Classical Problems of Synchronization" and features a yellow background with a blue wave-like shape on the right side. At the top, there is a navigation bar with various icons. The main content is a bulleted list:

- Classical problems used to test newly-proposed synchronization schemes
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem
- We will present solutions using:
  - Semaphores.
  - Monitors
  - Various operating systems

In the bottom right corner, there is a video inset showing a man with glasses and a mustache, wearing a light blue shirt, speaking. At the bottom left of the slide, there are two logos: one of a gear and a person, and another of a gear and a sun.

So, classical synchronization problems that we have the first one is known as the bounded buffer problem. So here we have got two or more processes some of the processes they are writing on to the buffer and some other processes they are trying to read from the buffer. So, relative speed of these producers and consumers that we have, so they are speeded that will matter. So some if the buffer is a finite size, then after some time with the producers are fast then the buffer will become full. So, consumers until and unless some consumer consume some items from the buffer producer should not be allowed to produce for the writing.

Similarly, if the buffer is empty in that case that consumers should not be allowed to read from the buffer or take items from the buffer because there is no item in the buffer. So, if you if the buffer size that we have is in finite then of course, we do not have this synchronization problem, but if the buffer is of a limited size some fixed size then in that case the synchronization has to be ensured.

Some other variants of this classical synchronization problems like readers writers problems. So, here we have got another very another synchronization problem where we

have got several processes. Some of the processes they just try to read some data item whereas, the writer processes are there who want to modify the data items. So, naturally if some writer is writing or modifying the data item, so at that time no reader process should be allowed also no more writer process should be allowed at that point, so writers access is mutually exclusive.

On the other hand if we have no difficulty if multiple readers are allowed to read the data items simultaneously. So, that way we have got this readers writers problem. So, its a different its different from the producer bounded buffer producer consumer problem. Then we will see another problem which is known as dining philosophers problem this is again something they are related to shared resources or synchronization between the processes are known. So, we will see them slowly.

So, we will solve solutions they will look into the solutions based on semaphores, monitors and you can also look into how in various operating systems the problems are solved, some operating systems they provide primitives. So, that these problems can be solved at that level at the waist level itself.

(Refer Slide Time: 04:25)

The slide is titled "Bounded-Buffer Problem" and contains the following text:

- $n$  buffers, each can hold one item
- Semaphore mutex initialized to the value 1
- Semaphore full initialized to the value 0
- Semaphore empty initialized to the value  $n$

The diagram shows a central buffer represented as a horizontal cylinder divided into  $n$  segments. On the left side, three producer processes labeled  $P_1$ ,  $P_2$ , and  $P_3$  have arrows pointing towards the buffer. On the right side, two consumer processes labeled  $C_1$  and  $C_2$  have arrows pointing away from the buffer. The buffer segments are numbered 1 through  $n$  from left to right.

So, first we look into the semaphore based solutions and in that we will look into the bounded buffer problem. So, as I was telling in a bounded buffer problem what happens is that, we have got some process which are there are a number of processes which are known as the producer processes. So, we call them say if there are three producer

processes P 1, P 2 and P 3 now and there are a number of consumer processes say suppose I have got two consumer processes c 1 and c 2.

Now, producer processes they are producing items and the items are not specific for a consumer like. So, it is, so we may be the item produced by any of the producer processes may be consumed by any of the consumer processes. So, there is no distinction between the data item produced by one producer from the data item produced by another producer they are similar. Similarly as far as consumption is concerned, so that is also similar. So, any consumer can consume any item from the buffer.

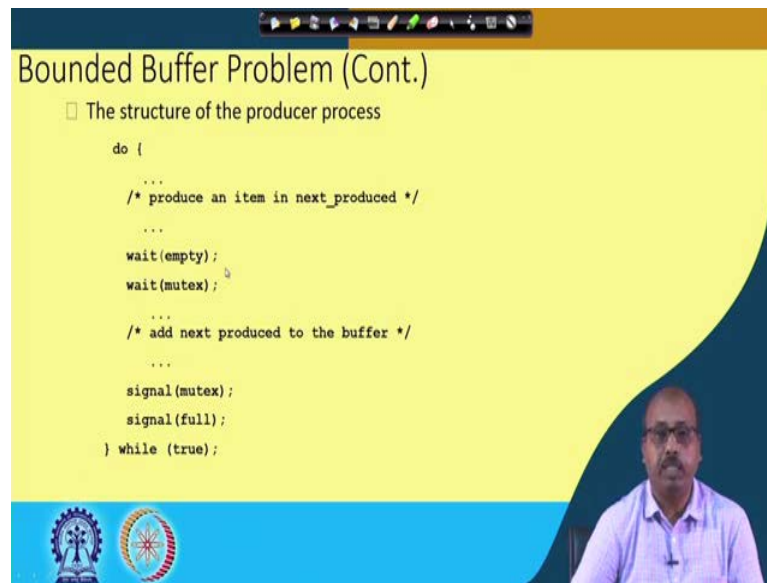
So, to solve this speed mismatch problem, so we have to; we have to solve this speed mismatch problem, we have to put a buffer in between and that buffer is say this one. So, this is a finite size buffer that we have in between. So, this buffer has got 1, 2, 3, 4, 5, 6 locations. So, when this producer, so producer processes they will right on to these buffer and the consumer processes they will consume from this buffer. So, the speed becomes an important issue, now if there are.

So, in this case this n that is the buffer size, so n is equal to 6 and we use a number of semaphores mutex, full and empty. So, mutex is for mutually exclusive access to the buffer. So, any producer or consumer willing to access the buffer should do it in a mutually exclusive fashion. So, for that purpose this mutex semaphore has been used. So, mutex this semaphore value is initialized to 1. So that the first process that wants to enter into the critical section to modify this buffer, so that will be allowed.

And initially this buffer has got no item in it. So, we say that the semaphore full is initialized to 0. So, initially there is nothing in the buffer. So, this full semaphore value, so this is counting semaphore. So, this indicates the number of items that are at presently there in the buffer. And we have got the semaphore, so this counts the number of empty slots that are there in the buffer at present. So, initially all the n slots are empty, so this empty semaphore is initialized to n.

Now, you can understand that I can allow the producer processes to produce further items as long as that the empty semaphore value is greater or greater than 0. On the other hand this consumer process it should be allowed to consume items from the buffer as long as this full value is greater than 0. So, with this concept, so we can see like how this problem can be solved using semaphores.

(Refer Slide Time: 07:54)



Bounded Buffer Problem (Cont.)

- The structure of the producer process

```
do {  
    ...  
    /* produce an item in next_produced */  
    ...  
    wait(empty);  
    wait(mutex);  
    ...  
    /* add next produced to the buffer */  
    ...  
    signal(mutex);  
    signal(full);  
} while (true);
```

So, this is the structure of the producer problem. So, this producer process is assumed to be an infinite loop. So, all the producer processes the code is similar ok. So, it is that is why it is not written separately for different producers. So, what it is doing? So, initially it is doing some processing which is internal to the producer, so at this time it is not going to modify the buffer. So, it is producing an item in some temporary variable say next produced. So, in the next produced variable the item has been produced. Now, it has to seek permission and try to see whether there is any empty slot in the buffer or not. So, this wait on empty, so this will be doing the thing.

So, see as you can as you remember that this empty semaphore was initialized to n because all the slots are empty. So, first time a producer comes, so this wait empty. So, this will be through and the empty variable will become n minus 1. So, now, after so the there are empty slots, so this producer can proceed to ride the item on to the buffer, but before writing or modifying the buffer it has to have mutually exclusive access to the buffer. So, for that it does wait on mutex and then after that it is free to modify the buffer. So, if it is through this wait on mutex now it is free to modify the buffer. So, it does the action, so that the next produced item is put into the buffer.

Once the buffer modification is over it will signal mutex that other processes can enter into their critical sections and then it signals full. So, full semaphore was initialized to 0. Now, one item has been produced, so by this signal statement the full variable will

become equal to 1. And of also you remember from the semaphore semantics if it is a way if it is a blocking type of semaphore that is the if a process was had called wait on full and it was waiting there then the signal full will invoke the process it will inform the process and take it up for execution.

So, this loop goes on in an infinite fashion. So, as long as the producers are there, so they can produce item and these producers they will produce item as long as there are empty slots in the buffer. So, this is the producer process structure.

(Refer Slide Time: 10:27)

```
□ The structure of the consumer process
do {
    wait(full);
    wait(mutex);
    ...
    /* remove an item from buffer to next_consumed */
    ...
    signal(mutex);
    signal(empty);
    ...
    /* consume the item in next consumed */
    ...
} while (true);
```

Next the consumer process structure, so this is like this. So, first the consumer process it first needs to access the buffer read it read the next element from the buffer and then use it in some internal computation. So, how is it doing? First it is doing a wait on full. So, the full variable contains the number of items that are there in the buffer at present and as I said that the full may be allowed or the consumer may be allowed if full value is greater than 0. So, initially full value was equal to 0 after some producers have produced some items. So, full value will become nonzero it will increase to some non zero value.

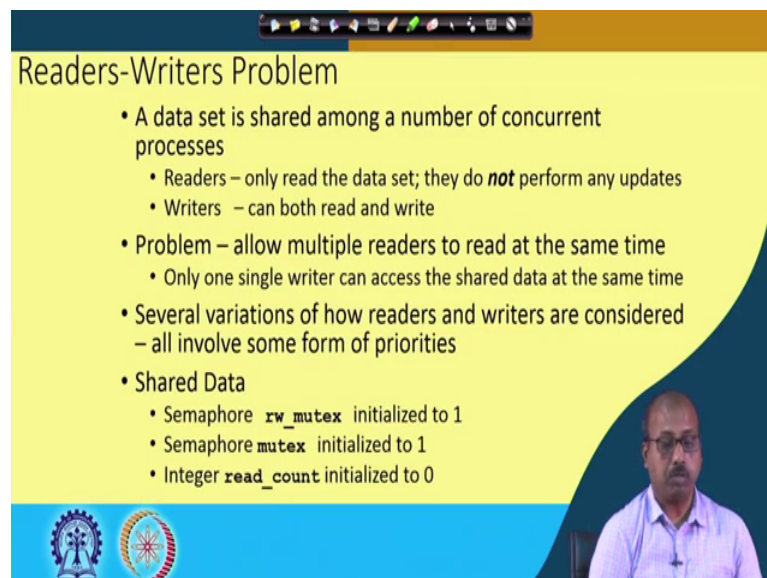
So, suppose producer has producers have produced 3 items, so full value is currently 3. So, this wait on full it will be through at that time, then it will wait on mutex ok. So, after getting permission that I should be able to there are some after get doing the check that there is some item in the buffer, this wait on mutex it will seek permission to access the buffer and after that it executes the code that will remove an item from buffer into some

local variable next consumed. Then it will signal mutex, so buffer modification has been done.

So, this mutex signal mutex it will take come out of the critical section and then signal empty. Now, one empty slot has been created because this consumer has read one item from the buffer. So, if there are some producer processes waiting for some slots to be available in the buffer, so this signal empty. So, this will inform that producer that yes some slots are available now , so you can try again to write. So, the signal empty is increasing the value of the empty variable.

After that it goes into local processing where it will consume the next item a the item that was produced in next consumed variable into its local processing and then while true. So, this way this producer process that the consumer processes will work. So, what essentially happens? The both producer and consumer processes they use this mutex semaphore whenever it is trying to modify the buffer either reading or writing the buffer. And the producer process informs the consumer process that something has been written by signaling on the full for full semaphore and this consumer process informs the producer process that some slot is empty by doing a signaling on the empty semaphore.

(Refer Slide Time: 13:06)



**Readers-Writers Problem**

- A data set is shared among a number of concurrent processes
  - Readers – only read the data set; they do **not** perform any updates
  - Writers – can both read and write
- Problem – allow multiple readers to read at the same time
  - Only one single writer can access the shared data at the same time
- Several variations of how readers and writers are considered – all involve some form of priorities
- Shared Data
  - Semaphore `rw_mutex` initialized to 1
  - Semaphore `mutex` initialized to 1
  - Integer `read_count` initialized to 0

The slide features a yellow background with a blue and orange header. A video inset in the bottom right corner shows a man with glasses and a mustache, wearing a light blue shirt, speaking. At the bottom left, there are two circular logos: one with a gear and a person, and another with a gear and a sun.

So, this is how this whole thing works. Now, you can very easily check that there is at no point of time if the producers are fast compared to the consumers then also after some time the producer will be made to wait. Similarly, if there is no item in the buffer

consumers are faster than producers, then this consumers will be made to wait because there is no item in the buffer, so the consumers are made to wait. So, this type of situation can occur in many real in a real situation in an operating system.

For example, suppose we have got one process which is reading from the keyboard and another process which is some user program which is running the running some code. So, from the values for the program variables are entered to be are to be entered by the user from the keyboard. So, this keyboard reading process and this program under execution, so, these two are two separate processes. Now, there can be several readers there can be there can be several processes which are reading from the keyboard.

So, as a result we can have a number of producers for that produces these key stroke sequences and the consumers may be the all these programs that are running. So, that way we can have a number of producers and consumers created in the scenario. So, this way this producer consumer problem this happens to be one of the very fundamental issues that we have in the operating system.

Next we look into another very common synchronization problem that is there in operating system known as readers-writers problem. So, as the statement of the problem it goes like this that a data set is shared among a number of concurrent processes and there are some processes which are reader in nature. So, they are they just try to read the data set and do not perform any updates they are just reading from the data set. And there are some writers that can both read and write actually read is redundant here for the writer. So, this is basically the right operation, but since the right is there, so it can also read it, so there is no harm in that.

So, the problem that we have is we can we should we should allow multiple readers to read at the same time. So, because there is no point in allowing one reader at a time because the readers are not going to modify, so there is no question of the data getting corrupted one when if multiple readers access them simultaneously. So, multiple readers can be allowed simultaneously; however, as far as the writer is concerned only a single writer can access the shared data at the same time.

So, I cannot allow multiple writers to access that data simultaneously because they will all this writers they will try to modify the data and as we have seen in our discussion on critical section problems that multiple modifications if they are carried out



simultaneously there is a chance of inconsistency in the final value that is there in the data item. So, we should not allow multiple writers simultaneously only one writer should be allowed at one point of time.

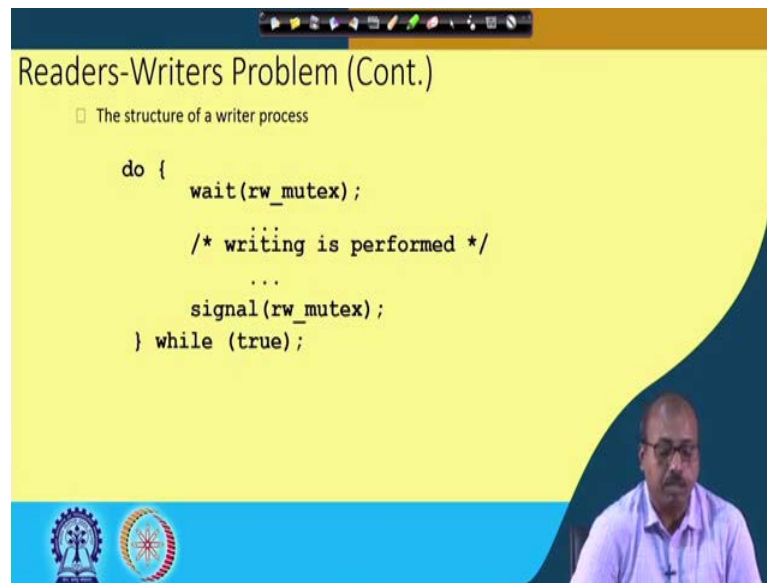
And naturally I cannot when I cannot allow any more reader also when a writer is accessing. So, writer access is totally mutually exclusive, but these readers are shared and multiple readers can be allowed, but when a writer is accessing only one writer should be allowed no more writer or reader should be allowed at that point of time. Now, there are several variations of how readers and writers are considered. So, all involved some form of priorities like, if there is a; if there is a continuous flow of readers and they are, so if we say that readers are allowed simultaneously.

So, whenever a reader comes we allow it assuming that there was no writer at that point of time, but if now a writer comes, so then the writer has to wait for this entire flow to be over. So, that way in this case the readers have got priorities over writers. So, this situation, so there can be different variants of the problem, but, so we discussing on all those variations that is out of scope. So, we will be looking into some solution were the readers are having priorities over the writers.

So, for doing this operation about this implementing this solution, so we have will be using some shared variable. Now, semaphore read right mutex, so that is initialized to 1 and we will be using another semaphore mutex which is initialized to 1 and there is a integer variable read count, so that will be initialized to 0. So, this read count variable, so it will hold at present how many readers are there in the system.

So, when read count is 0, then only some writer if it has arrived, so it should it can be allowed. So, similarly when a reader come, so reader can check this read count value and this reader will understand if reading is going on by this read count becoming greater than 0, then it will understand that some readers are there in the system. So, it should also be able to proceed.

(Refer Slide Time: 18:35)



Readers-Writers Problem (Cont.)

- The structure of a writer process

```
do {
    wait(rw_mutex);

    /* writing is performed */
    ...
    signal(rw_mutex);
} while (true);
```

The slide features a yellow background with a blue and orange header. At the bottom, there are two logos on the left and a small video inset of a man speaking on the right.

So, we will see how this readers and writers are coded. So, structure of a writer processes very simple. So, this read right mutex that we have, so the that was initialized to 1 here and so this is for the writer access ok. So, writer can do both read and write operation that is why we named the variable as read write mutex, if you think that I will only write operation not read operation then you can conveniently rename this variable as right mutex, but anyway that is not a big issue.

So, this wait on this read write mutex, so that is; so if that it was initialized to 1. So, if this is the first writer that has arrived then this check will be through and once this check is through this mutex value will become 0 and after that no more writer will be allowed to enter it will be waiting at that wait statement. So, this way this writer process is done. So, first it does a wait on read write mutex, then it will be writing some writing will be performed and then after the writing is over it will signal the read write mutex. So, this will go on in an infinite loop.

(Refer Slide Time: 19:51)

Readers-Writers Problem (Cont.)

□ The structure of a reader process

```
do {
    wait(mutex);
    read count++;
    if (read count == 1)
        wait(rw_mutex);
    signal(mutex);
    ...
    /* reading is performed */
    ...
    wait(mutex);
    read count--;
    if (read count == 0)
        signal(rw_mutex);
    signal(mutex);
} while (true);
```

On the other hand this reader process, so this is a bit complex. So, first what we have to do is that whenever a reader comes it has to increment the read count variable. So, for the time being I will ask you to consider do not look into this wait mutex statement. So, you just see this statement that a reader has arrived and accordingly the read count has been implemented. Now, if this is the first reader process then read counts value was initially 0 now it has become 1. So, if the read count is 1; that means, this is the first reader the no more reader is there in the system now.

Now, there can be two situations; one situation is some writer is now doing the writing or second other situation is that there is no writer also and to take that condition whether in a writer is in progress or not. So, you have to check this read write mutex. So, if read count is 1 that is the first reader, then it will wait on this read write mutex otherwise it will be otherwise it can proceed to reading. And in between so this read count variables. So, these becomes another shared variable because suppose I have got two reader processes that are there. So, R 1 R 2; so R 1 and R 2, two reader processes are there. So, both the reader processes will try to modify this read count variable.

So, this read count plus plus statement is executed by both the readers and we have seen in our critical section discussion that whenever a shared variable is modified by more than one process. So, that becomes a critical section of code. So, these read count plus

plus this implement operation, so this is a critical section code. So, that has to be protected by some semaphore. So, for that purpose this mutex semaphore has been used.

So, R 1 comes R 1 does a wait on mutex, so of if it is through then it will be doing this modification, while doing this modification if R 2 comes R 2 will be stuck at this wait on mutex. So, it will not be; it will not be able to do anything, it will not be able to access the read count variable. So, that is the purpose. So, this wait on this mutex semaphore, so this is used for protecting the read count variable.

Now, if read count is equal to 1 then it will be waiting on this read write mutex otherwise it will signal mutex that is I am done with the updating the read counts. So, any other reader process coming, so they can check this they can be through and modify this read count. And if read count is equal to 1 and there is some writer currently writing, then this read write mutex value was 0. So, this wait on read write mutex, so this will make the reader to wait and at that time it is not signaling mutex because that is; that is; that is not necessary because if this reader is waiting then the successive readers who will arrive. So, they will also have to wait until and unless this writing operation is the writing operation is over by the currently progressing writer.

So, when that writer will finish, so it will be the this read write mutex value will become greater than 0 and then this reader will be through, then it will be a signal mutex to tell other readers that you can proceed now. Whatever be the situation, so it is once this part is through the reader is allowed to proceed allowed to do the reading. So, reading will be performed, after reading is performed now I have to decrement the reader count ok.

So, this again this read count is a shared variable as I said. So, this becomes a critical section code this discriminating the read counts value. So, read count minus minus is done and if read count becomes equal to 0; that means, there is no more reader process, so this was the last reader process ok. So, for a process after doing read count minus minus, so if read count becomes 0; that means, this was a; this was the last reader process. So, there is no more reader process in the system. So, in that case it has to tell, it has to see whether any writer is waiting or not and if some writer is waiting it needs to signal the writer that you can proceed now, so that is done at this point.

So, if read count is 0 in that case it is doing signal read write mutex, so it will be telling the writer process that you can proceed now. Whatever be the case ultimately this read

count modification is over. So, this it comes out of this critical section of code and it does it signal mutex where this mu this it comes out of this shared code of these read count access. So, that way it comes out. So, that and then while through do, so this way it will it will go on doing these reading operations.

So, reader process structure is slightly complex compared to the writer process structure. Now, you see as I was telling that we are favoring multiple readers because the first reader comes and if it is somehow through this read count update and it has come to the reading stage, then any more writer any more reader coming you see that it will find that read count is not equal to 1. So, it will go into signal mutex and go into the writing, so it will not wait for that.

(Refer Slide Time: 25:23)

Readers-Writers Problem (Cont.)

□ The structure of a reader process

```
do {
    wait(mutex);
    read_count++;
    if (read_count == 1)
        wait(&rw_mutex);
    signal(mutex);

    /* reading is performed */

    wait(mutex);
    read_count--;
    if (read_count == 0)
        signal(&rw_mutex);
    signal(mutex);
} while (true);
```

*Handwritten notes:*  $R_1, R_2, R_3 \dots$  (readers),  $W_1, W_2, W_3 \dots$  (writers). Favoring readers. Writer may starve.

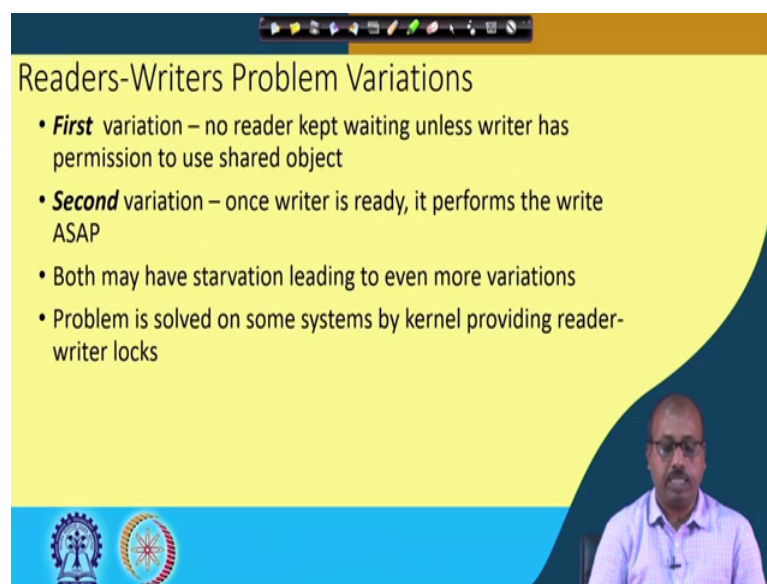
So, if the for the first reader that we have it is the first reader may have to wait for some currently progressing writer, but the successive readers R 2, R 3 etcetera as long as this reader 1 is continuing, so these R 2, R 3 all of them, so they need not wait for the writer. Now, suppose the situation was that this writer one came first, then came reader 1, then while reader 1 is progressing then writer 2 came, but still reader 1 is continuing at that time this writer reader 2, reader 3 came like that. So, maybe I can write; so this is a reader 1 arrives. So, this is these are the arrival time for reader 2, reader 3 this is the arrival time of writer 2 like that, maybe at this point the reader 1 finishes. So, reader 1 finishes at this point.

Now, what will happen after this reader 1 has arrived and this writer 1 is over, so, reader 1 will be allowed to enter into this reading part, now this writer 2 it will see that. So, each it will find that these read write mutex value is 0, so it will be made to wait. So, this reader this writer process will be made to wait, but these R 2, R 3, so they though they came after writer 2. So, they will be allowed to proceed because it will only find that read count is not equal to 1, so it will be through this and it will go into the reading phase. So, this writer will have to wait till all the readers that are there in the system are over. So, if there is a continuous flow of readers, then this will happen quite late that the writer is allowed to write onto the data item.

So, this particular solution it favors; it favors readers. So, if there is a multiple if there is a continuous flow of readers then the writer may starve; so writer may starve if there is a continuous flow of readers. So, there are other solutions where we can make the solution to be favoring the writers. So, if there is a writer then all the readers who arrived after that, so they will be made to wait and that and then the writer should be allowed and then only the readers will continue.

So, that type of solution is also possible, but that will also favor the writers not the readers. So, that way we can have different situation, so this is rather asymmetric solution unlike that producer consumer problem which was more or less a symmetric solution. So, this particular solution is a bit asymmetric.

(Refer Slide Time: 28:15)



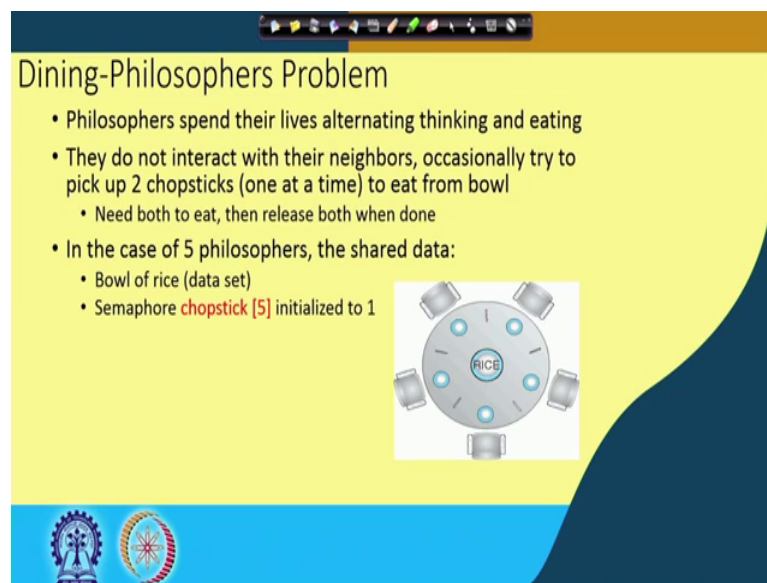
### Readers-Writers Problem Variations

- **First** variation – no reader kept waiting unless writer has permission to use shared object
- **Second** variation – once writer is ready, it performs the write ASAP
- Both may have starvation leading to even more variations
- Problem is solved on some systems by kernel providing reader-writer locks

So, the variations that we have, the first variant is that no reader kept waiting unless writer has permission to use the shared object, so this is the first variation that we have seen. And second variation is that once the writer is ready it performs the write as soon as, so both the variants are there. Both may have starvation leading to even more variation like; so first variation when the reader is no reader will be kept waiting. So, in that case the writer may starve and the second variation when the writer is ready it performs right as soon as possible. So, in that case if there is a continuous flow of writers, then these readers will be make to wait. So, that in that case the reader processes they will start.

So, both the versions they have got the starvation problem and this problems can be solved on some systems by kernel providing reader writer locks some operating systems they will provide at a kernel level, some reader writer lock and if it gets the lock then that will ensure that the access is made in that sequence only.

(Refer Slide Time: 29:23)



**Dining-Philosophers Problem**

- Philosophers spend their lives alternating thinking and eating
- They do not interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
  - Need both to eat, then release both when done
- In the case of 5 philosophers, the shared data:
  - Bowl of rice (data set)
  - Semaphore **chopstick [5]** initialized to 1

The diagram shows a circular table with five chairs. In the center is a bowl labeled 'RICE'. There are five pairs of chopsticks, one pair between each adjacent pair of chairs.

So, this is the readers writers problem that we have. So, next we will be looking into another very interesting problem known as dining philosophers problem. So, as the name of the problems are just, so this is something related to philosophers they are eating on a table and all, but this has got similarity with some real situation that can happen in a computer system where there are multiple shared resources and the processes they

require more than one resource for doing certain operation. So, we will see into this dining philosophers problem in the next class.