Slide 1

Part 4: Process Synchronization

- **Race Condition & Critical-Section**
- User-level Solutions
- OS-level Solutions
  - Synchronization Hardware
  - Semaphores
- Classical Problems of Synchronization

* Important but difficult ☺

how animations

Slide 2

concurrent processes dont necessarily run in parallel
(exception: running in two cores)

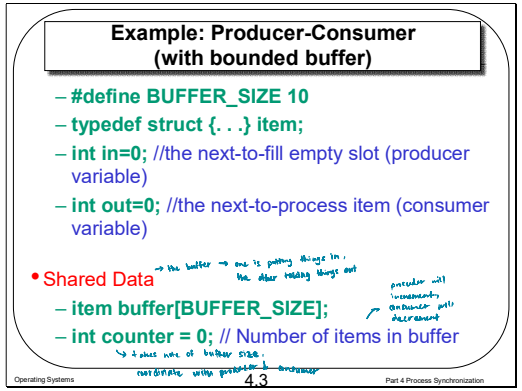
Background

- Access to shared data from concurrent processes may result in data inconsistency
  - Inconsistent Data: Data value depends on the order of instruction executions from concurrent processes
  - That is, data value depends on when context switches occur between the concurrent processes
    ↳ a problem to be fixed
- Maintaining data consistency requires mechanisms to ensure the orderly execution of concurrent processes
  - Causal ordering (sequencing) of reads and writes to the shared data from concurrent processes

Why we need process synchronization? Remember that cooperating/concurrent processes usually share some data. Concurrent access to shared data may result in data inconsistency. We should consider context switches in any place of process execution in order to consider all the possible interleaving orders.

We need some mechanisms to ensure the orderly execution of cooperating processes.

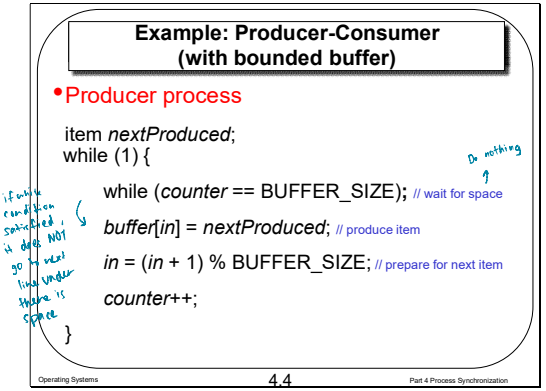
Slide 3



Let's look at one example. Suppose we have two processes in the producer and consumer example: producer and consumer. They share a variable counter to keep track of the number of items in the shared buffer.

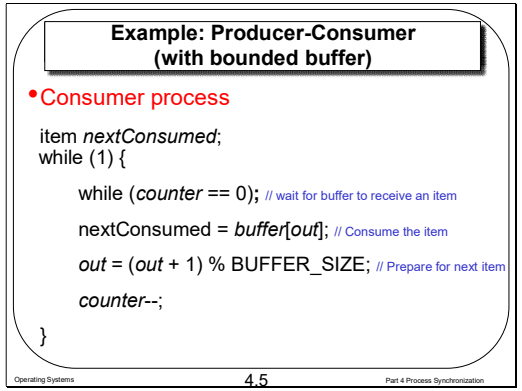
Buffer is the shared buffer with BUFFER_SIZE size.
in and out are pointers.
in: the current empty slot in the buffer that the producer wants to put in.
out: the item in the buffer that the consumer wants to take from.
counter is the number of items in the buffer.
in and out are initialized with zero, and counter is also equal to zero.

Slide 4



If the buffer is full, wait (pay attention to the semicolon). Otherwise, we put the item into the buffer and increase the counter by one.
We use (in+1) mod BUFFER_SIZE, meaning that the buffer is a circular buffer. When the in value reaches the end of the buffer, it goes back to zero.

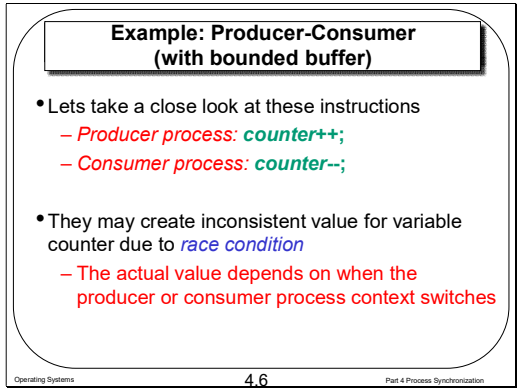
Slide 5



Consumer: we check whether the buffer is empty. If it is empty, the consumer waits.

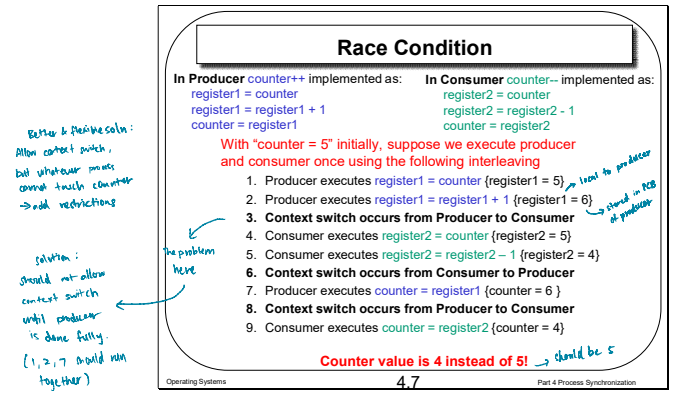
So, now, let's see what are the shared variables in those two processes?
Between producer and consumer:
Counter is the shared variable.
Buffer is the address shared by both processes. However, it is read-only.

Slide 6



The two statements counter++ and counter– may create inconsistent value due to race condition. Race condition means that the actual value depends on the order of process executions. Let's see the details with an example.

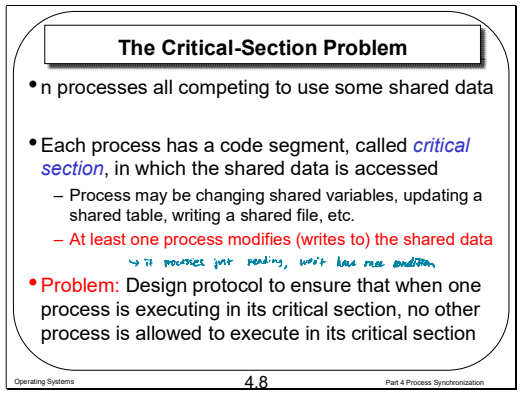
Slide 7



@one animation in the additional slide.

This example shows that different execution orders of multiple processes may result in data inconsistency. The actual value of the shared variable depends on the order of process executions. We need to consider all possible context switch places to check whether inconsistency is possible.

Slide 8

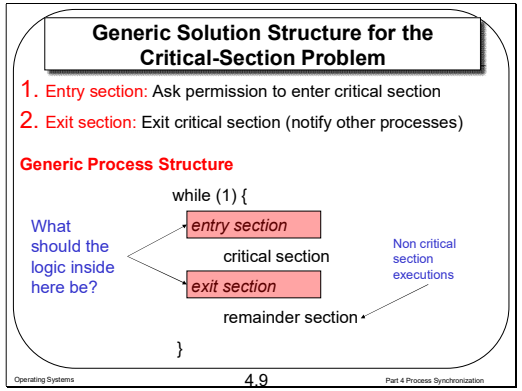


Now we define the critical section problem. In this problem, we have multiple processes all competing to use some shared data. Each process has a code segment called critical section, where the shared data is accessed. At least one of the processes is updating the data.

So, the critical section problem is to design a protocol to ensure that when a process is executing in its critical section, no other process is allowed to execute in its critical section. That means, at any time, there is at most one process in the critical section.

In the previous example, counter++ and counter-- are critical section code. If at any time, there is at most one process in its critical session → no context switch in the middle of the critical session · data consistency is achieved.

Slide 9

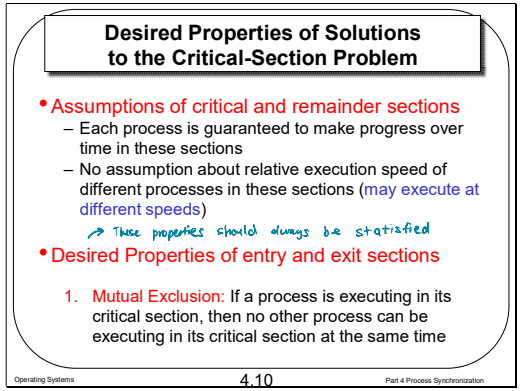


We model each process with a program structure. Each process must ask permission to enter the critical section in entry section, then enter the critical section to access the shared variables, and exit it in the exit section. In the exit section it indicates that the other processes can go into the critical section. Remainder sections are for other (non critical section) logic.

The loop is to allow you to explore all possible orders of executions (different places to perform context switches). Our implementation of the entry section/exit section should be correct for all the possible orders of executions.

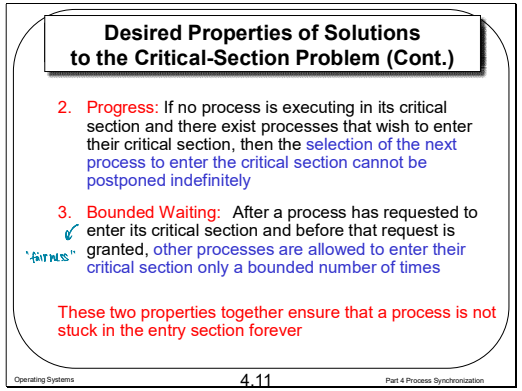
We will now offer some solutions in implementing entry section and exit section in order to solve the critical section problem.

Slide 10



Mutual exclusion means the solution must guarantee that, at any time, there is at most one process in the critical section.

Slide 11

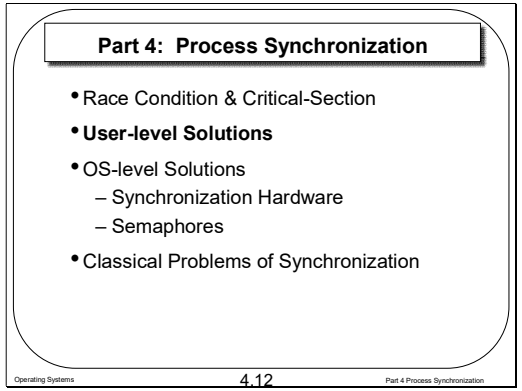


Progress: If there is no process in the critical section, a process requesting to enter the critical section will eventually enter it. Progress means that a process that is not in the critical section cannot block other processes from entering the critical section. Bounded waiting means a process can eventually get the chance to enter the critical section.

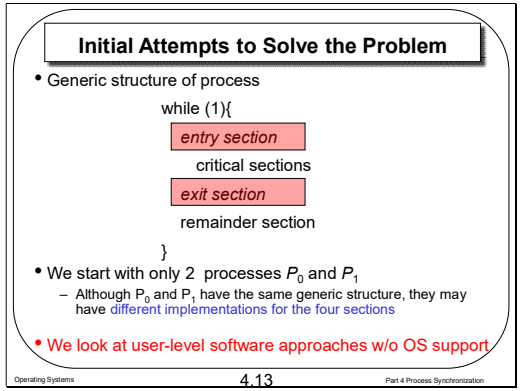
A correct solution to a critical section problem should satisfy all the three requirements. If any of the three requirements is violated, the solution is wrong.

There is a circumstance called live-lock (corresponding to deadlock), in which two or more processes are organized as a process group, all the processes can acquire or release lock, which satisfies bounded waiting, but the process group cannot progress (or why we call it live-lock. :-)).

Slide 12



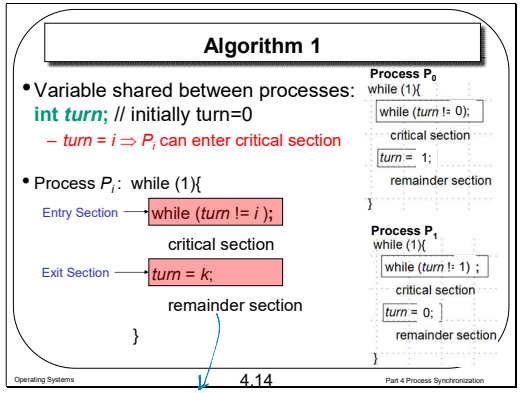
Slide 13



Before we talk about operating system support for the critical section problem, we look at software solutions without operating system support.
Two points:
1. We learn how to analyze whether a solution can satisfy the three requirements.
2. These solutions are implemented by users, without any special support from OS.

Here, we want to use these examples to show you: these software approaches are difficult to write, even for two processes. Thus, it is necessary for OS to provide support.

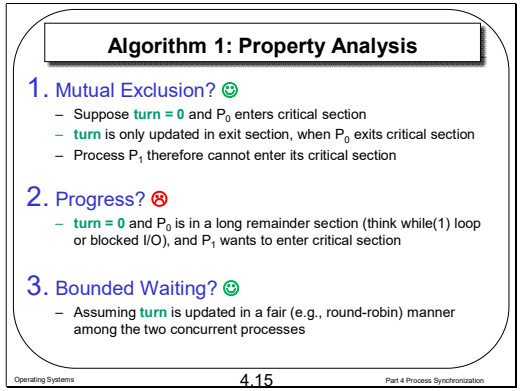
Slide 14



P1 give turn, but P0 doesn't want
P1 wants the turn, but can't go in cause its not its turn!

First try: Algorithm 1. We define one shared variable: turn. If turn=0, P0 can enter its critical section, otherwise P1 can enter its critical section.

Slide 15



****Best practices for analyzing BMP (Bounded Waiting, Mutual Exclusion, Progress)*****

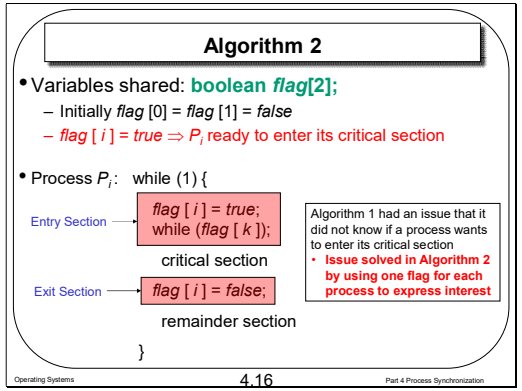
- Mutual exclusion: we assume that one process is in the critical session, and then see if the other process can enter the critical section. If yes, mutual exclusion is violated. Otherwise, mutual exclusion is satisfied in this scenario. we can check the other way around.

- Progress: we consider the scenario if no process is in the critical section. Let's consider if a process is in the entry/exit section, will it stop other processes to go into the critical section? If yes, progress is violated.

- Bounded wait: whether it allows infinite cutting in. usually no problem.

Slide 16

issue when both process flag is up.



Shortcoming of Algorithm 1: The single shared variable is not sufficient to remember the state of each process. It only records the process that is allowed to enter its critical section. To address this problem, we use two shared variables. Flag[0] and flag[1].

In the entry section, we set our flag to be true. Then, we wait for flag[k] to be false, that means pk has left the critical section.

In the exit section, we set flag[i] to be false, meaning that Pi has left the critical section.

Slide 17

Algorithm 2: Property Analysis

1. Mutual Exclusion? ☺
   - Suppose P_0 enters critical section, then **flag [1] = false** and **flag [0] = true**
   - **flag [0]** is only updated to false in the exit section of P_0
   - Process P_1 therefore cannot enter its critical section

   ↗ hard to satisfy

2. Progress? ☹ Consider the following interleaving:
   1. P_0 executes **flag [0] = true;**
   2. **Context switch to P_1**
   3. P_1 executes **flag [1] = true;**

   Both P_0 and P_1 are stuck in an infinite while loop now!

3. Bounded Waiting? ☺ → not directly related to 2
   - Once P_0 sets **flag [0] = true**, process P_1 cannot enter its critical section

Slide 18

if both "raise hand", turn breaks the tie

Algorithm 3

- Variables are a combination of Algorithms 1 and 2
  - *flag [i] = true* ⇒ P_i ready to enter its critical section
  - *turn = i* ⇒ P_i can enter critical section

  one process need to keep track of flags from all other processes to should examine it

- Proc. P_i: while(1){

  any other process

Entry Section →
flag [i] = true;
turn = k;
while (*flag [k]* and *turn = k*);

critical section

Exit Section →
flag [i] = false;

remainder section

}

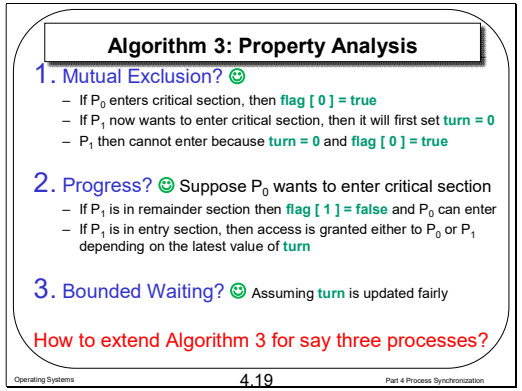
Algorithm 2 could not choose between two processes that wanted to enter their critical section at the same time
- **Issue solved in Algorithm 3 by using turn variable to denote whose turn it is**

why is bounded waiting for algo 2?
- so other weak process that called the critical section its flag became false. Then other process can enter critical section

This algorithm combines the ideas of algorithms 1 and 2.

This algorithm satisfies the three requirements of a solution to the critical section problem. Let's see how it solves the problem of algorithms 1 and 2.
In the entry section: we indicate the intension of entering the critical section by setting flag[i]=true, and also set turn to be k. If flag[k]=true and turn=k, that means process Pk has expressed intent to enter the critical section and it is its turn. If either condition is false, we can enter the critical section.

Slide 19

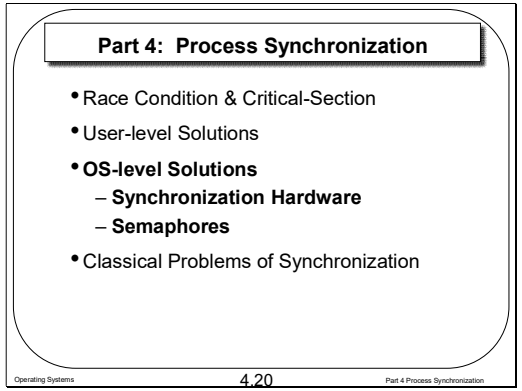


Algorithm 3 meets all the three requirements. It solves the critical section problem for two processes. Please refer the textbook for the proof. The proof is just for your reference.

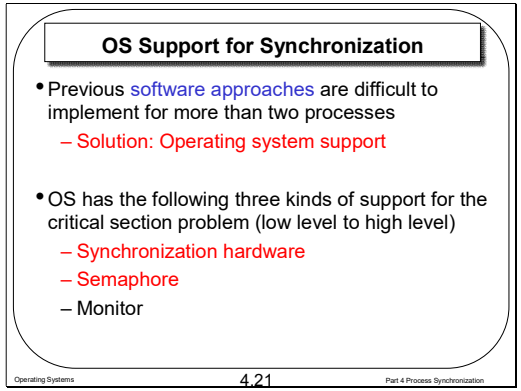
The algorithm is developed by a famous Microsoft researcher named Leslie Lamport. I have shown you how difficult it is to implement a solution with software approaches (via shared variable).

In summary, we have learnt different implementations for the critical section problem. Here, the important point is to show you how to analyze whether an implementation can solve the problem. You are not required to remember the detailed algorithm, for example, in our exam, we will *not* examine you the detailed algorithm. We will not ask you what algorithm 1 is, what algorithm 2 is… the important thing is that, given an algorithm/example, you should be able to analyze whether it satisfies the three requirements. There is one question in the tutorial. Please attempt it.

Slide 20



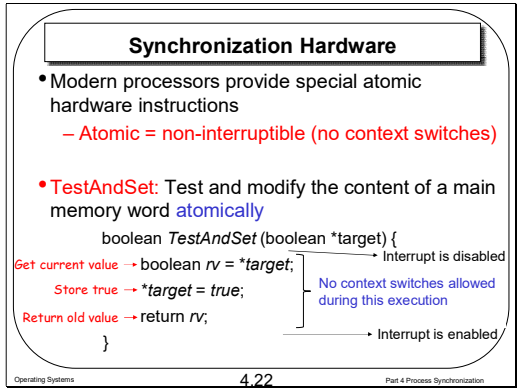
Slide 21



@let's review what we have learnt in our last lecture. We started with talking about the critical section problem. there are three requirements: mutual exclusion, progress and bounded waiting. You can refer to the previous slides for the detailed definition of each requirement. Then, we talk about software approaches to solve this problem, and those approaches are without operating system support. There are three algorithms for solving the critical section problem of two processes. Algorithm 1, 2, and 3. Algorithms 1 and 2 have progress violations, and Algorithm 3 satisfies all the three requirements. However, software approaches are too difficult to implement for multiple processes. Thus, we need the support from operating system.

Operating system has the following three kinds of support for critical section: synchronization hardware, semaphore and monitor (we focus on the first two). They are from low level to high level. We will discuss it one by one. For each kind, we first look at how operating system implements the process synchronization mechanisms. Then we talk about how we can use the mechanisms to implement process synchronization.

Slide 22



Modern hardware usually offer some atomic operations or instructions. An atomic operation means that the operation is executed in order without preemption. The execution of an atomic operation is non-interruptible, meaning that there is no context switch from the first line till the last line.

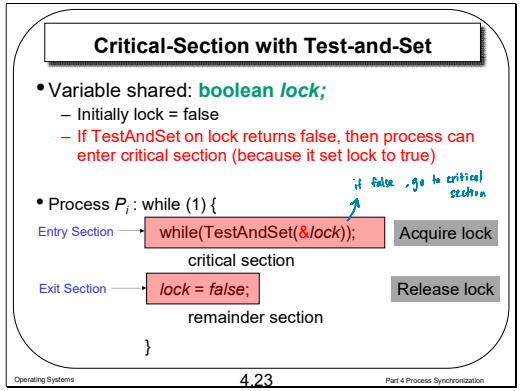
TestAndSet is a common one. It reads a value and changes the value of the target word atomically. The return value is the old value of the target word. Note * denotes a pointer in c/c++.

Interrupt is disabled -> OS will disable the context switch before the first line is executed. OS will enable the context switch after the last line is executed.

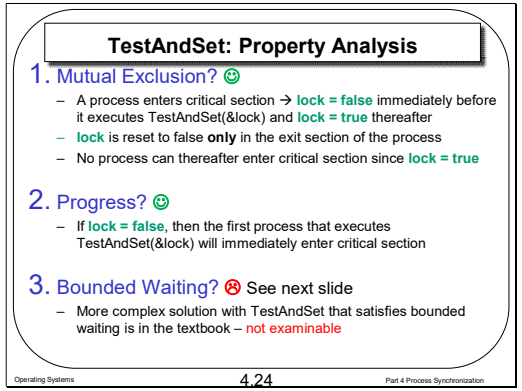
Slide 23

bounded waiting not satisfied → no record of failing to get lock

so test and set will return false , when it received the lock as false, but will update and set the lock to true , then proceed to run critical section code, hence other testandset will not be able to acquire the lock

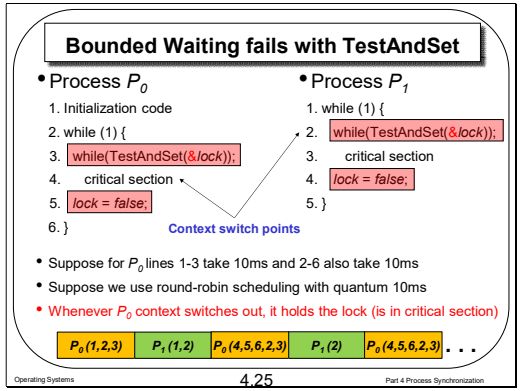


Slide 24

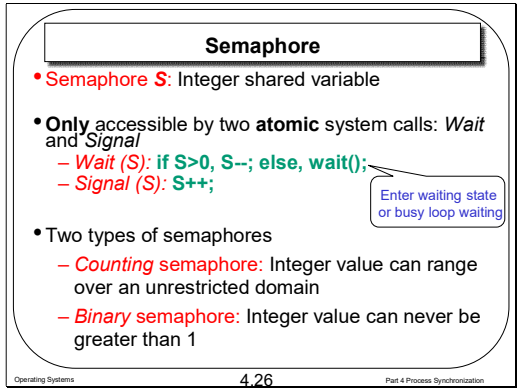


We can use TestAndSet to implement mutual exclusion. We define a lock boolean variable. Initially, it is false. & is the address of the variable in c/c++.

Slide 25



Slide 26



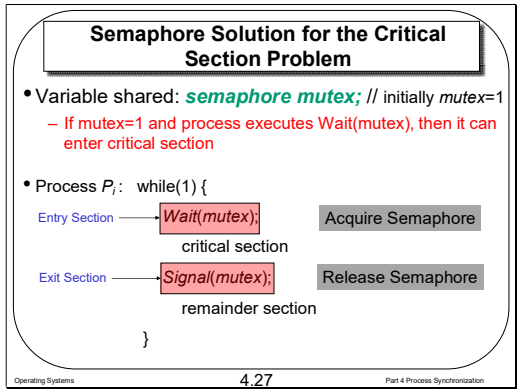
The hardware solutions are too low level, and are complicated to use. So, operating system provides a higher level synchronization mechanism called semaphore. Semaphore is very important, not only because we have questions on this part in every past exam paper, but also because semaphore is a widely used process synchronization mechanism.

Semaphore is an integer shared variable. Operating system offers two atomic operations to manipulate this shared variable: Wait and Signal. We will talk about the details of implementation in later slides.

When we execute Wait, if S value is one, we can finish waiting and go to the next step. If S value is zero, we have to wait. Its value can only be changed by these two atomic operations. You can not directly change the value.

There are two types of semaphore, depending on the values it can take. For a counter semaphore, its integer value can range over an unrestricted domain. This is for the scenario that the shared resource has many instances, for example, there can be multiple printers in the office. The second kind is called binary semaphore whose integer value can either be 0 or 1. It is usually used for a lock for mutual exclusion.

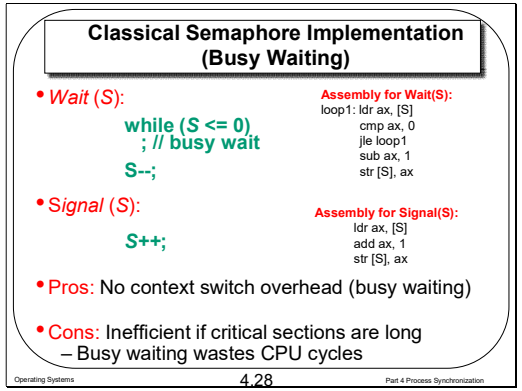
Slide 27



With the semaphore, we can easily develop a solution to the critical section problem. The mutex value is initialized with 1, a binary semaphore. You can see, this implementation is simpler than the previous solution using atomic instructions. There is no while-loop, because it is already implemented in wait. We will see more details later.

Slide 28

Impossible to use for signal core



Now let's see how semaphore is implemented in OS. The classical implementation for Wait and Signal uses busy waiting.

Why the implementation is called busy waiting? Because if S<=0, the process needs to busy wait in a while loop, and keeps polling the value of S. Busy waiting has the advantage that no context switches occur. That is particularly good for a short waiting time. However, the constraint is that busy waiting is inefficient if the critical section is long, because the while loop will consume too many CPU cycles.

Note, if Wait(S) is not implemented in an atomic manner, mutual exclusion is violated. See next slide and the tutorial question on this.

Slide 29

allow context switch, but don't touch S

Busy Waiting Semaphore (Property Analysis)

1. To avoid race condition, S++ and S-- must be atomic
2. Wait(S) must be atomic to ensure mutual exclusion
   - Consider the following with S>0 initially
     1. Process P_0 executes **while (S <= 0);** and exits busy wait
     2. **Context switch from P_0 to P_1**
     3. Process P_1 executes **while (S <= 0);** and **S--;**
     4. **Context switch from P_1 to P_0**
     5. Process P_0 executes **S--;**
     6. *Mutual Exclusion?* ☹

Assembly for Wait(S):

```
loop1: ldr ax, [S]   ┐
       cmp ax, 0     ├ while (S <= 0);
       jle loop1     ┘
       sub ax, 1     ┐
       str [S], ax   ├ S--;
```

Assembly for Signal(S):

```
       ldr ax, [S]   ┐
       add ax, 1     ├ S++;
       str [S], ax   ┘
```


Slide 30

Current Semaphore Implementation (Blocking)

- Define a semaphore as a record

```
typedef struct {
    int value;
    struct process *L;
} semaphore;
```

L is a queue that stores the processes waiting on this semaphore (in the form of PCB list)

- Need two simple operations
  - *block( )*: blocks the current process
  - *wakeup( )*: resumes the execution of a blocked process in list L

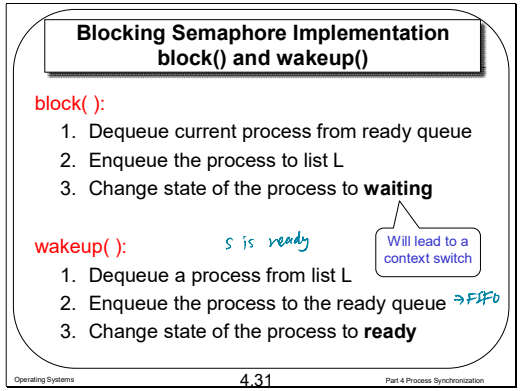
putting process in list

→ taking process out of list

Because the busy waiting implementation is not feasible on a single processor system, current operating systems have another implementation: blocking. The basic idea is that when a process enters its critical section, other processes that want to enter their critical section will be blocked. A process is blocked, meaning it will be put into the waiting state. So, the process waiting to enter the critical section does not consume CPU cycles.

Before we go into the implementation details, we define the relevant data structure and two operations.

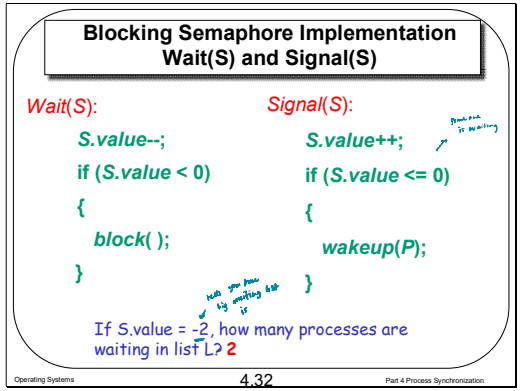
Slide 31



Let's take a closer look at those two operations.

Block: first dequeue current process from ready queue (we suppose that the PCB of the running process is also stored in the head of the ready queue), and then enqueue current process to list L. The process state changes from running to waiting.
Wakeup: dequeue a process P from list L. We can have different policies here, just like process scheduling. Then enqueue P to the ready queue. The process state changes from waiting to ready.

Slide 32

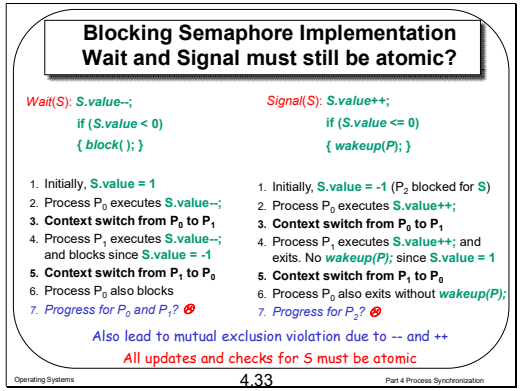


no while loop

Wait(S): We first decrease the value by one. If S.value<0, then it means we need to block because the value before we decreased it was already <=0. Hence we block the process.

Signal(S): We first increase the value by one. If S.value is smaller than or equal to zero, we wakeup a process from list L. Why we need to check if(s.value<=0) and not if(s.value<0)? Consider s.value=-1 (only one process waiting in list L), before executing signal. After S.value++, what is the value? S.value=0. That is why we need to consider s.value==0 as well for waking up processes.

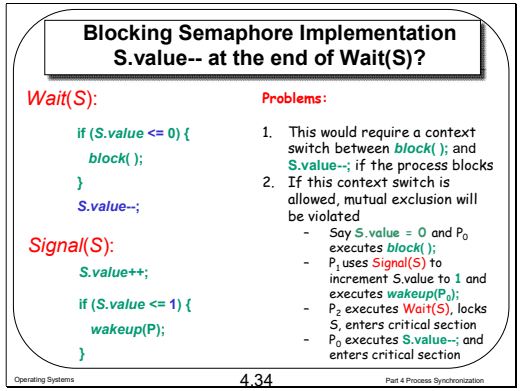
Slide 33



Both Wait() and Signal() must be executed atomically (at least all updates and checks for semaphore S), even if the blocking semaphore implementation is used. If these updates and checks are not atomic, then progress can be violated as shown in the slide. It can also lead to violation of mutual exclusion because of the update to S (same issue as with the counter variable in the producer-consumer problem in Slide 4.7).

There is some possibility to allow limited context switches inside Wait() and Signal(). More details about this are in the tutorial.

Slide 34



Why should we update the value of S before blocking? If we update the value after blocking, then this means we need to allow for a context switch inside Wait(). The process that is blocked cannot execute, because it is in the Waiting state. This means another process must be allowed to execute, otherwise the process that is currently holding the semaphore cannot release it.

If we allow context switch between block() and S.value-- inside Wait(), this can lead to violation of mutual exclusion. Suppose a process P0 executes block() and moves to the Waiting state (semaphore value currently is 0). Later, process P1 which is currently holding the semaphore, increments its value to 1 and wakes up P0 (P0 moves to Ready state). While P0 is in Ready state, another process P2 executes Wait(), locks semaphore by decrementing its value to 0 and enters its critical section. While P2 is in the critical section, P0 starts running, decrements the value of the semaphore to -1 and enters critical section. This is a violation of mutual exclusion. Note, P0 will not check again whether the value of semaphore is <=0, because it has already executed block() and will now resume its execution from S.value--.

Slide 35

Part 4: Process Synchronization

- The Critical-Section Problem
- User-level Solutions
- OS-level Solutions
  - Synchronization Hardware
  - Semaphores
- **Classical Problems of Synchronization**

Slide 36

Best Practices for Semaphores

- Step 1: Understand the application scenario and identify the following:
  - Shared variables/data
  - Shared resources
- Step 2: Protect the shared variables
  - Identify the critical section
  - Use binary semaphore, and add entry section (Wait) and exit section (Signal)
- Step 3: Protect the shared resources
  - Identity each kind of resource; one semaphore for one kind
  - Initial value: number of resource instances
  - When the resource is requested/consumed: Wait
  - When the resource is released: Signal

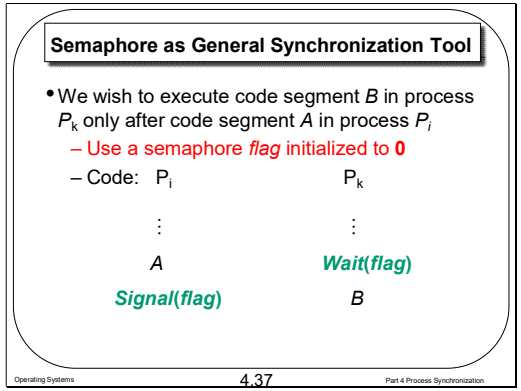
Step 1, 2, 3 for semaphores. Those are the three typical steps for solving process synchronization problems with semaphores.

Shared variables is logical: counter in producer/consumer.
Shared resources are physical: Number of printers.

If you remove all the Wait/Signal, the *logic* of the code is still correct in sequential executions. However, it is not correct in concurrent executions.

The semaphore value represents the number of instances for that kind of resource (>0). Otherwise, represents the number of processes waiting in the waiting queue (list L).

Slide 37

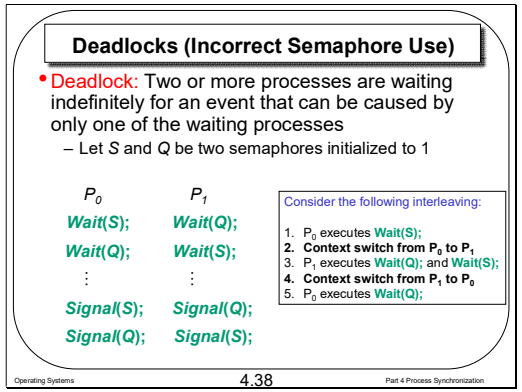


@Let's review what we have learnt so far. We start with talking about atomic operations. There is no context switches from the first line to the last line of the execution of an atomic operation. We give one example of showing how to implement process synchronization with atomic operation TestAndSet. Semaphore is implemented as a shared integer variable. It supports two atomic operations: Wait and Signal. With the semaphore, it is easy to implement the process synchronization among multiple processes. Then, we have talked about two implementations of semaphores in operating systems. Busy waiting and blocking. Busy waiting may waste CPU cycles when the critical section is long. Blocking results in context switches. The process waiting to enter the critical section is put to the waiting queue of the semaphore (list L).

After talking about the implementation of semaphore in OS, we now talk about the usage of semaphores. We will talk about the first issue: semaphore as general process synchronization tool.

Semaphore can be used as a general synchronization tool (to enforce some execution order). In the above example, think what happens for different interleavings. It is guaranteed that B will execute after A, because Wait(flag) will only complete after Signal(flag) is executed.

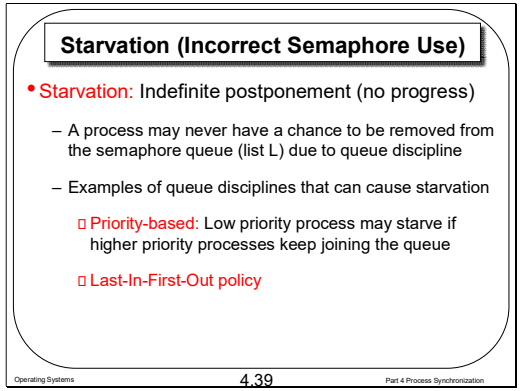
Slide 38



However, incorrect usage of semaphore can cause problems. We will talk about two problems: deadlocks and starvation.

Deadlock means two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes. Simply speaking, for two processes, A and B, a deadlock occurs if A waits for B, and B waits for A. That means, none of them can make any progress.

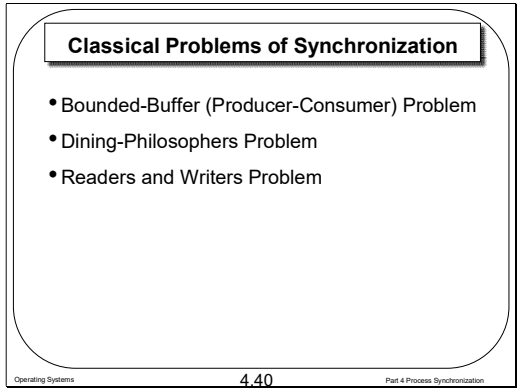
Slide 39



Starvation– a process may never have a chance to be removed from the semaphore queue (list L). For example, if the dequeue method is priority based, or LIFO (Last in first out).

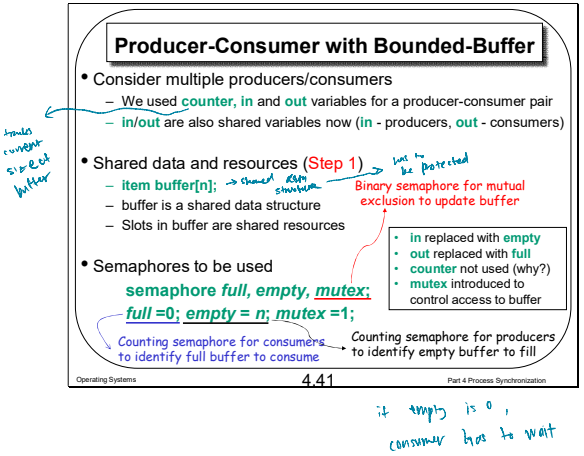
What is the solution? Aging.

Slide 40



We now talk about three classical problems of synchronization. We see how developers can use semaphore to implement the process synchronization in those three problems.
Bounded buffer problem, dining philosopher problem, and readers and writer problems.
After learning those examples, we should learn two things: firstly we should know how to write a process synchronization with semaphore; Secondly, we should be able to analyze the correctness of the program with semaphore.

Slide 41



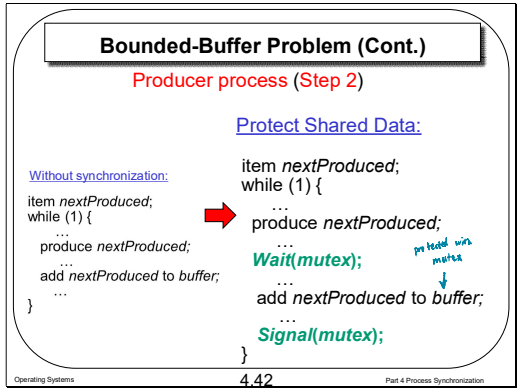
In general, we use counting semaphore to represent shared resources that requires process synchronization, and use binary semaphore to protect shared variable for mutual exclusions.

Mutex for mutual exclusion on the buffer (semaphore mutex). Any access to the buffer must be protected by this semaphore.

What are the shared resources that we need to protect? What are the counting semaphores that we need to define? Consider if the system has multiple producers and consumers.

- The number of items in the buffer (semaphore full). They are the same resource among multiple consumers. We must make sure the orderly consumption on the items.
- The number of empty slots in the buffer (semaphore empty). They are the same resource among multiple producers. We must make sure the orderly execution on taking the slots.

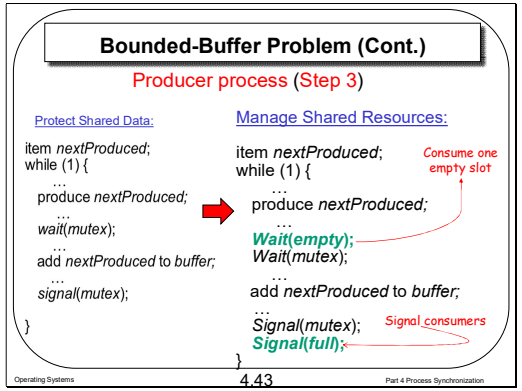
Slide 42



Original code: no synchronization.

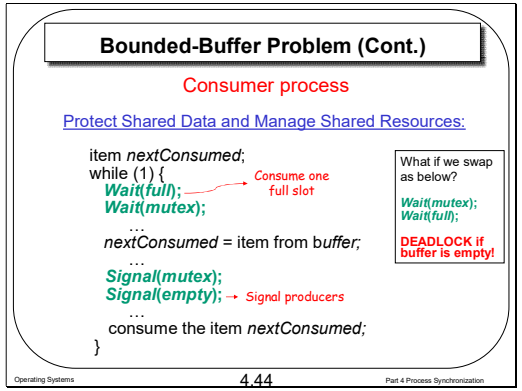
First, we protect the critical section, i.e., buffer update with the binary semaphore mutex. Say, we need to increment the number of items in the buffer, and manipulate the buffer. "add *nextProduced* to *buffer*," is a critical section.

Slide 43



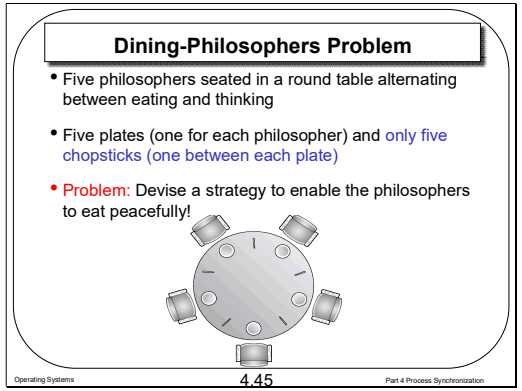
Step 3: protect the shared resources.

Slide 44



You can think of different scenarios to see that they are correctly synchornized. For example, when the buffer is full. consumer can go in; but producer cannot. You can consider other cases as well.
Now, let's consider one question: the order of Wait(full) and Wait(mutex) cannot be swapped. Why?

Slide 45

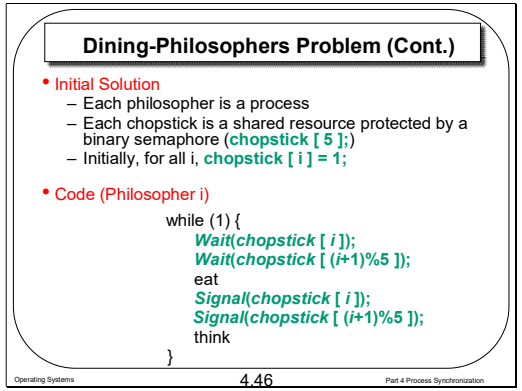


What is the five philosopher problem?
Each philosopher is either eating or thinking. They share a circular table with five chairs. A philosopher sits on a chair. There are only five chopsticks. When a philosopher becomes hungry, he will try to pick up the two chopsticks (on his left and right hand side). A philosopher can only pick the chopstick one at a time. He cannot take the chopsticks from others. He can eat only when he picks both chopsticks. When he finishes eating, he puts down the chopsticks.

What are the shared data here? Five chopsticks. So, we can define five semaphores, one semaphore for each chopstick.

Each chopstick is one semaphore. Why? Each chopstick is shared by two different processes.

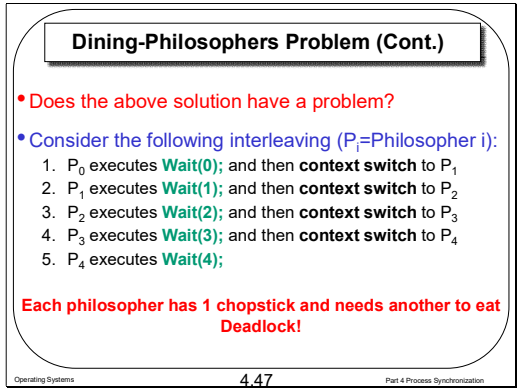
Slide 46



For philosopher i (0<=i<5): He first picks up his left chopstick, and then the right chopstick. This is done by calling the Wait operation on the corresponding semaphore. If both operations can succeed, the philosopher can eat. After eating, he releases the left chopstick and then the right one. After that, the philosopher will think, and then repeat the process.

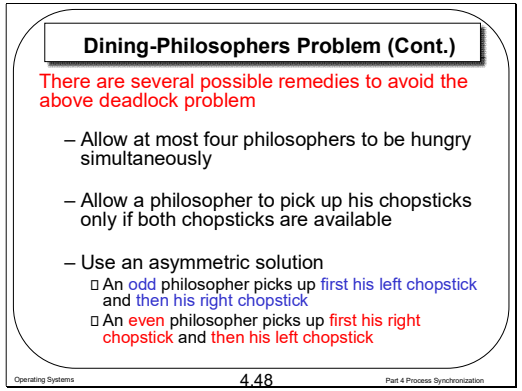
Does this solution have any problem? Each philosopher executes the first line and a context switch occurs. → deadlock.

Slide 47



The problem occurs when all five philosophers become hungry simultaneously, and each grabs his left chopstick. Now all the semaphores of chopstick will be equal to 0. When each philosopher tries to grab his right chopstick, he will be delayed forever. That is called deadlock. You can consider there is a context switch after the philosopher process executes the first line. You can see, incorrect use of semaphores can easily lead to the deadlock problem.

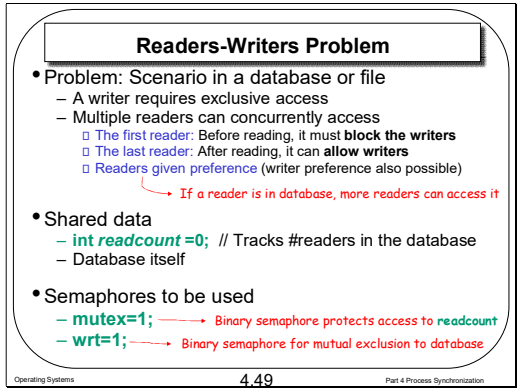
Slide 48



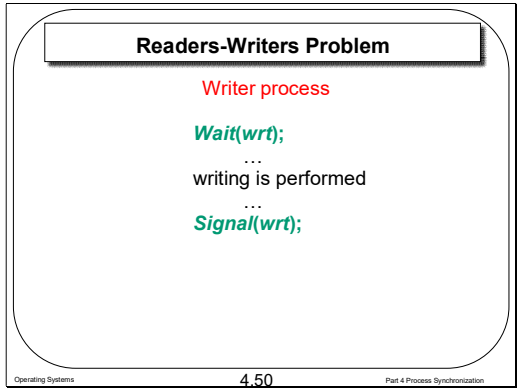
Can you think of a solution for this problem? There are many possible solutions. We give some examples here. We will talk more in our next chapter. Here, you just need to know that, those solutions will make sure: even when all the five philosophers are hungry at the same time, at least one philosopher can eat.

First, we allow at most four philosophers to be hungry at the same time. In this case, at least one philosopher can pick up both chopsticks.
Second, we allow a philosopher to pick up his chopsticks only if both are available. There is no deadlock, because one philosopher can take both chopsticks.
Third, we use an asymmetric solution: give an id to each philosopher. an odd philosopher picks up first his left chopstick and then his right one, whereas the even philosophers pick up his right chopstick first and then his left chopstick. We make sure there is at least one philosopher who gets both chopsticks.

Slide 49



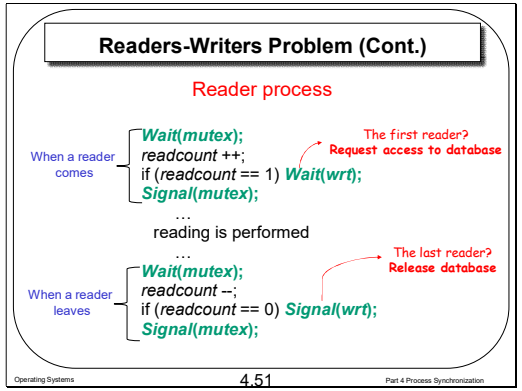
Slide 50



In database system, or file systems, we need to synchronize the reads and writes to shared data. The difference here is that we allow two readers to concurrently read the data. But if there is a writer, the access is exclusive and we require that writer has exclusive access to the shared data.

For system throughput, we try to maximize the probability of concurrent accesses from readers. Suppose a reader is accessing the database. During this period, if another reader comes to the database, we allow multiple readers to access and block the writers. Only after the last reader has exited the database, we allow the writers to access the database. That means, the first reader should block the writer before reading, and the last reader should signal the writer after finishing access to the database.

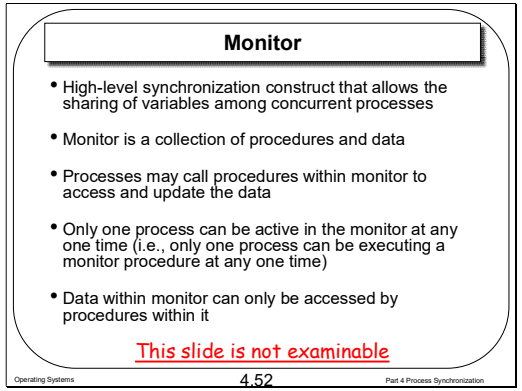
Slide 51



The first reader holds the wrt semaphore, and blocks the writer. The last reader signals on this semaphore.

Readcount is the shared variable. So we need to use a mutex for synchronization.

Slide 52



We have already seen that semaphore can solve the process synchronization problem, but also induce other problems such as deadlock. OS actually introduces a higher level construct for developers named monitor. With monitor, it is even easier to implement a solution for process synchronization.

Monitor is a higher level abstraction that allows the sharing of variable among concurrent processes. In its definition, it is a collection of procedures and data. You can consider it as a java class. Processes call the procedures within monitor to access the shared data. The operating system ensures that only one process can be active in the monitor at any time. That is, only one process can be executing a monitor procedure at any one time. That property is very important.