

Process Synchronization

Background

- ❑ Processes can execute concurrently
 - ❑ May be interrupted at any time, partially completing execution
- ❑ Concurrent access to shared data may result in data inconsistency
- ❑ Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- ❑ Illustration of the problem:

Suppose that we wanted to provide a solution to the consumer-producer problem that fills **all** the buffers. We can do so by having an integer **counter** that keeps track of the number of full buffers. Initially, **counter** is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

Producer

```
while (true) {  
    /* produce an item in next produced */  
  
    while (counter == BUFFER_SIZE) ;  
        /* do nothing */  
    buffer[in] = next_produced;  
    in = (in + 1) % BUFFER_SIZE;  
    counter++;  
}
```

Consumer

```
while (true) {  
    while (counter == 0)  
        ; /* do nothing */  
    next_consumed = buffer[out];  
    out = (out + 1) % BUFFER_SIZE;  
    counter--;  
    /* consume the item in next consumed */  
}
```

Race Condition

- A situation where several processes accesses and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place is called **race condition**.
- This condition can be avoided using the technique **process synchronization**
- Here, **ensure only one process manipulate shared data at a time.**

Race Condition

- `counter++` could be implemented as

```
register1 = counter
register1 = register1 + 1
counter = register1
```

- `counter--` could be implemented as

```
register2 = counter
register2 = register2 - 1
counter = register2
```

- Consider this execution interleaving with “count = 5” initially:

S0: producer execute	<code>register1 = counter</code>	{register1 = 5}
S1: producer execute	<code>register1 = register1 + 1</code>	{register1 = 6}
S2: consumer execute	<code>register2 = counter</code>	{register2 = 5}
S3: consumer execute	<code>register2 = register2 - 1</code>	{register2 = 4}
S4: producer execute	<code>counter = register1</code>	{counter = 6}
S5: consumer execute	<code>counter = register2</code>	{counter = 4}

Critical Section Problem

- Consider system of n processes $\{p_0, p_1, \dots, p_{n-1}\}$
- Each process has **critical section** segment of code
 - Process may be changing common variables, updating table, writing file, etc
 - When one process in critical section, no other may be in its critical section
- **Critical section problem** is to design protocol to solve this
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

Critical Section

- General structure of process P_i

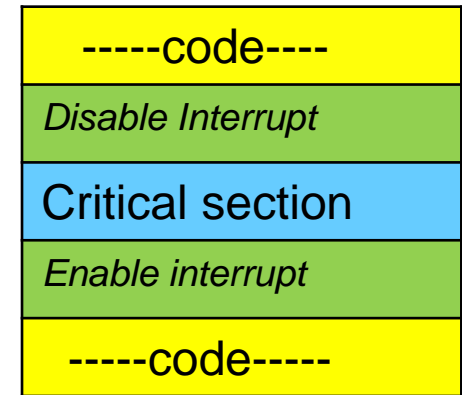
```
do {  
    entry section  
    critical section  
    exit section  
    remainder section  
} while (true);
```


Solution to Critical-Section Problem

1. **Mutual Exclusion** - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
2. **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
3. **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
 - Assume that each process executes at a nonzero speed
 - No assumption concerning **relative speed** of the n processes

Hardware Solution

- **Entry section – first action is “disable interrupts”**
- **Exit section – last action is “enable interrupts”**
- Must be done by OS. Why?
- Implementation issues:
 - **Uniprocessor systems**
 - Currently running code would execute without preemption
 - **Multiprocessor systems**
 - Generally too inefficient on multiprocessor systems
 - Operating systems using this not broadly scalable
- **Is this an acceptable solution?**
 - This is impractical if the critical section code is taking long time to execute



Critical-Section Handling in OS

Two approaches depending on if kernel is preemptive or non-preemptive

- **Preemptive** – allows preemption of process when running in kernel mode
- **Non-preemptive** – runs until exits kernel mode, blocks, or voluntarily yields CPU
 - ▶ Essentially free of race conditions in kernel mode

Algorithm 1: Software Solution for Process P_i

Keep a variable “turn” to indicate which process next

```
do {  
    while (turn != i);  
    critical section  
    turn = j;  
    remainder section  
} while (true);
```

- Algorithm is correct. **Only one process at a time in the critical section.**
- What if $\text{turn} = j$, P_i wants to enter in critical section and P_j does not want to enter in critical section?

Algorithm 1: Software Solution for Process P_i

```
do {  
    while (turn != i);  
        critical section  
    turn = j;  
        remainder section  
} while (true);
```

- What if $\text{turn} = j$, P_i wants to enter in critical section and P_j does not want to enter in critical section?

Algorithm 2 for Process P_i

- 2 Process solution using **Flag variable**
- Flag is Boolean variable with value T & F

```
While (1)
```

```
{
```

```
    Flag[i]=T    //Pi wish to go in CS
```

```
    while(Flag[j]); Check weather Pj wants to go to CS
```

```
        CS
```

```
    Flag[i]=F
```

```
}
```

CS	P0	P1
	F	F
P0	T	F
P1	F	T
	F	F
P0	T	F
	F	F
P0	T	F
	F	F
	T	T

Algorithm 3: Peterson's Solution

- Good algorithmic description of solving the problem
- Two process solution
- Assume that the **load** and **store** machine-language instructions are atomic; that is, cannot be interrupted
- The two processes share two variables:
 - `int turn;`
 - `Boolean flag[2]`
- The variable `turn` indicates whose turn it is to enter the critical section
- The `flag` array is used to indicate if a process is ready to enter the critical section.
- `flag[i] = true` implies that process P_i is ready!

Algorithm 3: Algorithm for Process P_i

```
do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
        critical section  
    flag[i] = false;  
        remainder section  
} while (true);
```


Algorithm 3: Peterson's Solution (Cont.)

□ Provable that the three CS requirement are met:

1. Mutual exclusion is preserved

P_i enters CS only if:

either `flag[j] = false` or `turn = i`

2. Progress requirement is satisfied

3. Bounded-waiting requirement is met

•But results in **busy waiting**.

Synchronization Hardware

- Many systems provide hardware support for implementing the critical section code.
- All solutions below based on idea of **locking**
 - Protecting critical regions via locks
- Uniprocessors – could disable interrupts
 - Currently running code would execute without preemption
 - Generally too inefficient on multiprocessor systems
 - ▶ Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
 - ▶ **Atomic** = non-interruptible
 - Either **test memory word and set value**
 - Or **swap contents of two memory words**

Solution to Critical-section Problem Using Locks

```
do {  
    acquire lock  
        critical section  
    release lock  
        remainder section  
} while (TRUE);
```

test_and_set Instruction

Definition:

```
boolean test_and_set (boolean *target)
{
    boolean rv = *target;
    *target = TRUE;
    return rv;
}
```

1. Executed atomically
2. Returns the original value of passed parameter
3. Set the new value of passed parameter to “TRUE”.

Solution using test_and_set()

- Shared Boolean variable lock, initialized to FALSE
- Solution:

```
do {  
    while (test_and_set(&lock)); /* do nothing */  
    /* critical section */  
    lock = false;  
    /* remainder section */  
} while (true);
```

compare_and_swap Instruction

Definition:

```
int compare_and_swap(int *value, int expected, int new_value) {  
    int temp = *value;  
  
    if (*value == expected)  
        *value = new_value;  
    return temp;  
}
```

1. Executed atomically
2. Returns the original value of passed parameter “value”
3. Set the variable “value” the value of the passed parameter “new_value” but only if “value” == “expected”. That is, the swap takes place only under this condition.

Solution using compare_and_swap

- Shared integer “lock” initialized to 0;
- Solution:

```
do {  
    while (compare_and_swap(&lock, 0, 1) != 0)  
        ; /* do nothing */  
    /* critical section */  
    lock = 0;  
    /* remainder section */  
} while (true);
```

Mutex Locks

- ❑ Previous solutions are complicated and generally inaccessible to application programmers
- ❑ OS designers build software tools to solve critical section problem
- ❑ Simplest is mutex lock
- ❑ Protect a critical section by first **acquire()** a lock then **release()** the lock
 - ❑ Boolean variable indicating if lock is available or not
- ❑ Calls to **acquire()** and **release()** must be atomic
 - ❑ Usually implemented via hardware atomic instructions
- ❑ But this solution requires **busy waiting**
 - ❑ This lock therefore called a **spinlock**

acquire() and release()

```
□ acquire() {  
    while (!available); /* busy wait */  
    available = false;;  
}  
  
□ release() {  
    available = true;  
}  
  
□ do {  
    acquire lock  
    critical section  
    release lock  
    remainder section  
} while (true);
```

Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.
- Semaphore **S** – integer variable
- Can only be accessed via two indivisible (atomic) operations

- **wait()** and **signal()**

- ▶ Originally called **P()** and **V()**

- Definition of the **wait()** operation

```
wait(S) {  
    while (S <= 0); // busy wait  
    S--;  
}
```

- Definition of the **signal()** operation

```
signal(S) {  
    S++;  
}
```

Semaphore Usage

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
 - Same as a **mutex lock**
- Can solve various synchronization problems
- Consider P_1 and P_2 that require S_1 to happen before S_2
Create a semaphore “**synch**” initialized to 0
P1 :
 S_1 ;
 signal (synch) ;
P2 :
 wait (synch) ;
 S_2 ;
- Can implement a counting semaphore S as a binary semaphore

Semaphore Implementation

- ❑ Must guarantee that no two processes can execute the **wait()** and **signal()** on the same semaphore at the same time
- ❑ Thus, the implementation becomes the critical section problem where the **wait** and **signal** code are placed in the critical section
 - ❑ Could now have **busy waiting** in critical section implementation
 - ▶ But implementation code is short
 - ▶ Little busy waiting if critical section rarely occupied
- ❑ Note that applications may spend lots of time in critical sections and therefore this is not a good solution

Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated waiting queue
- Each entry in a waiting queue has two data items:
 - value (of type integer)
 - pointer to next record in the list
- Two operations:
 - **block** – place the process invoking the operation on the appropriate waiting queue
 - **wakeup** – remove one of processes in the waiting queue and place it in the ready queue
- `typedef struct{
 int value;
 struct process *list;
} semaphore;`

Implementation with no Busy waiting (Cont.)

```
wait(semaphore *S) {  
    S->value--;  
    if (S->value < 0) {  
        add this process to S->list;  
        block();  
    }  
}  
  
signal(semaphore *S) {  
    S->value++;  
    if (S->value <= 0) {  
        remove a process P from S->list;  
        wakeup(P);  
    }  
}
```

Deadlock and Starvation

- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let S and Q be two semaphores initialized to 1

P_0
`wait(S) ;`
`wait(Q) ;`
`...`
`signal(S) ;`
`signal(Q) ;`

P_1
`wait(Q) ;`
`wait(S) ;`
`...`
`signal(Q) ;`
`signal(S) ;`

- **Starvation** – **indefinite blocking**
 - A process may never be removed from the semaphore queue in which it is suspended
- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
 - Solved via **priority-inheritance protocol**

Classical Problems of Synchronization

- Classical problems used to test newly-proposed synchronization schemes
 - Bounded-Buffer Problem
 - Readers and Writers Problem
 - Dining-Philosophers Problem

Bounded-Buffer Problem

- 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

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);
```

Bounded Buffer Problem (Cont.)

- 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);
```

Readers-Writers Problem

- ❑ Suppose that a database is to be shared among several concurrent processes.
- ❑ Some of these processes may want only to read the database, whereas others may want to update (that is, to read and write) the database.
- ❑ If two readers access the shared data simultaneously, no adverse effects will result.
- ❑ However, if a writer and some other process (either a reader or a writer) access the database simultaneously, chaos may ensue.
- ❑ To ensure that these difficulties do not arise, we require that the writers have exclusive access to the shared database while writing to the database. This synchronization problem is referred to as the **readers–writers problem**

Readers-Writers Problem

- The readers–writers problem has several variations, all involving priorities.
 - The simplest one, referred to as the **first** readers–writers problem, requires that no reader be kept waiting unless a writer has already obtained permission to use the shared object. In other words, no reader should wait for other readers to finish simply because a writer is waiting.
- The **second** readers–writers problem requires that, once a writer is ready, that writer perform its write as soon as possible. In other words, if a writer is waiting to access the object, no new readers may start reading.
- A solution to either problem may result in starvation. In the first case, writers may starve; in the second case, readers may starve.

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
 - ❑ Data set
 - ❑ Semaphore **rw_mutex** initialized to 1
 - ❑ Semaphore **mutex** initialized to 1
 - ❑ Integer **read_count** initialized to 0

Readers-Writers Problem

- The mutex semaphore is used to ensure mutual exclusion when the variable read count is updated.
 - The **read count** variable keeps track of **how many processes are currently reading the object**.
- The semaphore rw mutex functions as a mutual exclusion semaphore for the writers.
 - It is also used by the first or last reader that enters or exits the critical section.
 - It is not used by readers who enter or exit while other readers are in their critical sections.

Readers-Writers Problem (Cont.)

- The structure of a writer process

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


The structure of a writer process

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

$Mutex = 1$

$rw_Mutex = 1$

$rc = 0$

Case 1. Writer comes first

$rw_Mutex = 0$ { wait(rw_mutex) reduces
the rw_mutex & writer
get access to data for
update }

Writing

The structure of a writer process

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

Mutex = 1

$\pi W_Mutex = \cancel{1} \bigcirc$

$\pi C = 0$

1) If other writer comes $\{ W-W \rightarrow \text{not possible} \}$
 $\text{wait}(\pi W_mutex);$ loop as $\pi W_mutex = 0$

\Rightarrow Only one Writer

The structure of a writer process

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

Mutex = ~~1~~ 1
 $\pi w_mutex = \cancel{1} 0$
 $\pi c = \cancel{0} 1$

ii) If Reader comes { W-R → Not possible }

wait(mutex);

$\pi c++$;

If ($\pi c == 1$)

wait(πw_mutex);

⇒ Only One writer at a time.

Reader will loop here as $\pi w_mutex = 0$

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);
```

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);
```

$rw_mutex = 1$ ○

$mutex = 1$ ○ 1

$rc = 0$ 1

Case 2: Reader comes first

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);
```

$rw_mutex = 1$ 

$mutex = 1$ 

$rc = 1$ 

Case 1: If writer comes ($r-w \rightarrow$ Not allowed)
wait(rw_mutex); Writer loop as
 $rw_mutex = 0$

Readers-Writers Problem (Cont.)

- The structure of a reader process

do {

wait(mutex); \downarrow mutex

read_count++; \uparrow rc

if (read_count == 1) false

wait(rw_mutex); \downarrow not exe since if is false

signal(mutex); \uparrow mutex

...

/* reading is performed */ Reader 2 start reading

...

wait(mutex);

read_count--;


if (read_count == 0)

signal(rw_mutex);

signal(mutex);

} while (true);

Case 2: If other reader comes (R-R \Rightarrow allowed)

$\neg rw_mutex = 1$ 
 $mutex = 1$ \emptyset \pm \emptyset 1
 $rc = 0$ \pm 2

Readers-Writers Problem (Cont.)

- The structure of a reader process

do {

```
wait(mutex);  $\downarrow$  mutex  
read_count++;  $\uparrow$  rc  
if (read_count == 1) false
```

`wait(rw_mutex);` Not exe since it is false

```
signal(mutex); ↑ mutex
```

...

```
...
/* reading is performed */ Reader 2 start reading
```

• • •

```
wait(mutex); ↓ mutex
```

```
read count--;
```

```
read_count--;  
if (read_count == 0)
```

```
signal(rw_mutex);
```

```
signal(mutex); ↑ mutex
```

```
} while (true);
```

```
    } while (true);
```

First reader completed its reading then

$\pi w_mutex = 1$ 

$\text{mutex} = \{ \emptyset, \perp, \emptyset, \perp, \emptyset \}$

$$\pi_C = 0 \quad 1 \quad 2 \quad 1 \quad 1$$

Readers-Writers Problem (Cont.)

- The structure of a reader process

```
do {  
    wait(mutex);  
    read_count++;  $\uparrow$   $rc$   
    if (read_count == 1)  
        wait(rw_mutex);  
    signal(mutex);  
  
    ...  
    /* reading is performed */  
  
    ...  
    wait(mutex);  $\downarrow$   $mutex$   
    read_count--;  $\downarrow$   $rc$   
    if (read_count == 0)  $0 = 0 \checkmark$  True  
    signal(rw_mutex);  $\uparrow$   $rw\_mutex$   
    signal(mutex);  $\uparrow$   $mutex$   
} while (true);
```

$rw_mutex = \emptyset \downarrow$

$mutex = \emptyset \downarrow$

$rc = 0$

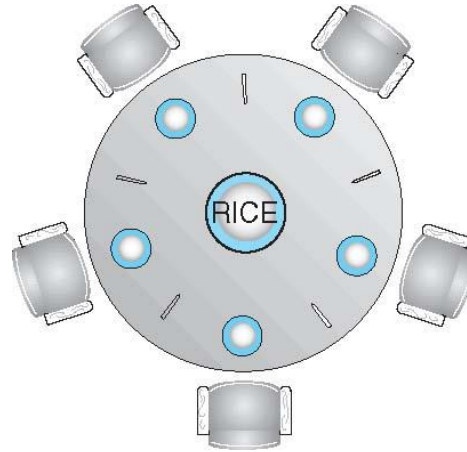
Last reader is completing its reading \Rightarrow Exit sec

When all readers complete reading ;
last reader will signal the rw-mutex
⇒ So, Writer will be able to write.

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

Dining-Philosophers Problem



- ❑ Philosophers spend their lives alternating thinking and eating
- ❑ Don't 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
 - ❑ Shared data
 - ▶ Bowl of rice (data set)
 - ▶ Semaphore **chopstick** [5] initialized to 1

Dining-Philosophers Problem

- It is an example of a large class of concurrency-control problems
- It is a simple representation of the need to allocate several resources among several processes in a **deadlock-free and starvation-free manner**
- One simple solution is to represent each chopstick with a semaphore.
- A philosopher tries to grab a chopstick by executing a wait() operation on that semaphore.
- She releases her chopsticks by executing the signal() operation on the appropriate semaphores.
- Thus, the shared data are
 - semaphore chopstick[5];

Dining-Philosophers Problem Algorithm

- The structure of Philosopher i :

```
do {  
    wait (chopstick[i] );  
    wait (chopstick[ (i + 1) % 5] );  
        // eat  
    signal (chopstick[i] );  
    signal (chopstick[ (i + 1) % 5] );  
        // think  
} while (TRUE);
```

- What is the problem with this algorithm?
- This solution guarantees that no two neighbors are eating simultaneously, it nevertheless must be rejected because it could create a deadlock.
 - Suppose that all five philosophers become hungry at the same time and each grabs her left chopstick.
 - All the elements of chopstick will now be equal to 0. When each philosopher tries to grab her right chopstick, she will be delayed forever.

Dining-Philosophers Problem Algorithm (Cont.)

- Deadlock handling
 - Allow at most 4 philosophers to be sitting simultaneously at the table.
 - Allow a philosopher to pick up the forks only if both are available (picking must be done in a critical section).
 - Use an asymmetric solution -- an odd-numbered philosopher picks up first the left chopstick and then the right chopstick. Even-numbered philosopher picks up first the right chopstick and then the left chopstick.
- Note, however, that any satisfactory solution to the dining-philosophers problem must guard against the possibility that one of the philosophers will starve to death. A deadlock-free solution does not necessarily eliminate the possibility of starvation

Problems with Semaphores

- ❑ Incorrect use of semaphore operations:
 - ❑ `signal (mutex) wait (mutex)`
 - ❑ `wait (mutex) ... wait (mutex)`
 - ❑ Omitting `wait (mutex)` or `signal (mutex)` (or both)
- ❑ Deadlock and starvation are possible.

Summary

- ❑ Given a collection of cooperating sequential processes that share data, mutual exclusion must be provided to ensure that a critical section of code is used by only one process or thread at a time.
- ❑ Hardware based solutions are too complicated for most developers to use.
- ❑ Mutex locks and semaphores overcome this obstacle. Both tools can be used to solve various synchronization problems and can be implemented efficiently, especially if hardware support for atomic operations is available.
- ❑ Various synchronization problems (such as the bounded-buffer problem,
- ❑ the readers–writers problem, and the dining-philosophers problem) are important mainly because they are examples of a large class of concurrency-control problems.
- ❑ These problems are used to test nearly every newly proposed synchronization scheme.