OPERATING SYSTEMS

Synchronization & Semaphores

Operating Systems

School of Data & Computer Science
Sun Yat-sen University

Lecture Notes: os_sysu@163.com
Instructor: Guoyang Cai
email: isscgy@mail.sysu.edu.cn

中山大學
SUN YAT-SEN UNIVERSITY

Process Synchronization

Contents

Synchronization Hardware

Synchronization Hardware

- Overview
  - Many systems provide hardware support for implementing the critical section code.
  - All solutions discussed in this section based on the idea of *Locking*
    - Protecting critical regions via locks.
  - Uniprocessors could disable interrupts to protect critical sections
    - Currently running code would execute without preemption
    - Generally too inefficient on multiprocessor systems
  - Modern machines provide special atomic hardware instructions:
    - *non-interruptible*
    - They either test one memory word and set value at once, or swap contents of two memory words.
    - We abstract the main concepts behind these types of instructions by describing two instructions.
      - test_and_set()
      - compare_and_swap()

Synchronization Hardware

- Interrupt Disabling
  - Process P_i:

```
    repeat
        disable interrupts
        critical section
        enable interrupts
        remainder section
    forever
```

  - On a Uniprocessor
    - Mutual exclusion is preserved but efficiency of execution is degraded: while in critical section, we cannot interleave execution with other processes that are in remainder section.
  - On a Multiprocessor
    - Mutual exclusion is not preserved.
      - Critical section is now atomic but not mutually exclusive (interrupts are not disabled on other processors).

Synchronization Hardware

- **Synchronization Hardware**
  - Special Machine Instructions
    - Normally, access to a memory location excludes other access to that same location.
    - Extension
      - Designers have proposed machine instructions that perform two actions atomically (indivisible) on the same memory location.
        - e.g., reading and writing
      - The execution of such an instruction is also mutually exclusive even on Multiprocessors.

Synchronization Hardware

- **Synchronization Hardware**
  - Solution to Critical Section Problem using Locks
    - The general layout of lock solution is:

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

Synchronization Hardware

- Test_and_Set Synchronization Hardware Instruction
  - Testing and setting (modifying) the content of a memory word *atomically* (a Boolean version):

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

    - It is executed atomically.
    - returns the original Boolean value of the passed parameter
    - set the new value of the passed parameter to "TRUE".
  - The Boolean function represents the essence of the corresponding machine instruction.

Synchronization Hardware

- Test_and_Set Synchronization Hardware Instruction
  - Mutual Exclusion with test_and_set instruction
    - Shared data:

```
boolean lock = FALSE;
```

    - Process P_i:

```
do {
    while (TestAndSet(&lock))
        ;  /* busy waiting */
    critical section
    lock = FALSE;
    remainder section
} while (TRUE);
```

Synchronization Hardware

- Test_and_Set Synchronization Hardware Instruction
  - Another test_and_set Example
    - A description of the machine instruction of test_and_set:

```
bool testset(int &i)
{
    if (i == 0) {
        i = 1;
        return TRUE;
    } else {
        return FALSE;
    }
}
```

Synchronization Hardware

- Test_and_Set Synchronization Hardware Instruction
  - Another test_and_set Example
    - An algorithm that uses test_and_set for Mutual Exclusion:
      - Shared variable b is initialized to 0;
      - Only first P_i who sets b enters critical section.
    - Shared data:

```
int b = 0;
```

    - Process P_i:

```
repeat
    repeat {
        ; /* busy waiting */
    } until testset(b);
    critical section
    b = 0;
    remainder section
forever
```

Synchronization Hardware

- **Synchronization Hardware**
  - Test_and_Set Synchronization Hardware Instruction
    - test_and_set Instruction at Assembly Level

```
enter_region:
TSL REGISTER, LOCK    | copy lock to register and set lock to 1
CMP REGISTER, #0      | was lock zero?
JNE enter_region      | if it was nonzero, lock was set, so loop
RET                   | return to caller, critical region entered

leave_region:
MOVE LOCK, #0         | store a zero in lock
RET                   | return to caller
```

    - TSL: *test and set lock* instruction of atomic.
    - Storage BUS will be locked during its executing time to provide the guarantee of exclusion.

- **Synchronization Hardware**
  - Swap Synchronization Hardware Instruction
    - *Atomically* swap (exchange) two variables:

```
void Swap(boolean *a, boolean *b)
{
    boolean temp = *a;
    *a = *b;
    *b = temp;
}
```

    - This procedure represents the essence of the corresponding machine instruction.

Synchronization Hardware

- Swap Synchronization Hardware Instruction
  - Mutual Exclusion with swap instruction
    - Shared data:

```
boolean lock = FALSE;
```

    - Process P_i:

```
do {
      /* Each process has a local Boolean variable key */
    boolean key = TRUE;
    while (key == TRUE) /* busy waiting */
        swap(&lock, &key);
    critical section
    lock = FALSE;
    remainder section
} while (TRUE);
```

Synchronization Hardware

- **Synchronization Hardware**
  - Swap Synchronization Hardware Instruction
    - swap instruction at Assembly Level

```
enter_region:
MOVE REGISTER, #1     | copy a 1 to the register
XCHG REGISTER,LOCK    | swap the contents of the register and lock
                      | variable
CMP REGISTER, #0      | was lock zero?
JNE enter_region      | if it was nonzero, lock was set, so loop
RET                   | return to caller, critical region entered

leave_region:
MOVE LOCK, #0         | store a zero in lock
RET                   | return to caller
```

    - XCHG: *exchange* instruction of atomic.
    - Storage BUS will be locked during its executing time to provide the guarantee of exclusion.

Synchronization Hardware

- Compare_and_Swap Synchronization Hardware Instruction
  - The compare_and_swap instruction operates on three operands;

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

    if (*value == expected)
        *value = new_value;
    return temp;
}
```

  - executed atomically
  - returns the original value of the passed parameter value
  - set the variable value the value of the passed new_value but only if the comparison “value == expected” is true
    - That is, the swap takes place only under this condition.

Synchronization Hardware

- **Synchronization Hardware**
  - Compare_and_Swap Synchronization Hardware Instruction
    - Mutual Exclusion with compare_and_swap instruction
      - Shared data

```
integer lock = 0;
```

      - Solution:

```
do {
    while (compare_and_swap(&lock, 0, 1) != 0)
        ;  /* busy waiting */
    critical section
    lock = 0;
    remainder section
} while (true);
```

Synchronization Hardware

- **Synchronization Hardware**
  - Disadvantages of Special Machine Instructions
    - *Busy-waiting* is employed, thus while a process is waiting for access to a critical section it continues to consume processor time.
    - *No Progress* (Starvation) is possible when a process leaves critical section and more than one process is waiting.
    - They can be used to provide mutual exclusion but need to be complemented by other mechanisms to satisfy the *bounded-waiting* requirement of the critical section problem.
    - See next slide for an algorithm using the test_and_set instruction that satisfies all the critical-section requirements.

Synchronization Hardware

- Example: Bounded-waiting mutual exclusion with test_and_set
  - Shared data initialized to false:

```
Boolean waiting[n];
integer lock;
```

  - Process P_i:

```
do {
    waiting[i] = TRUE;
    key = TRUE;
    while (waiting[i] && key)
        key = TestAndSet(&lock);
    waiting[i] = FALSE;
    critical section
    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j + 1) % n;
    if (j == i)
        lock = FALSE;
    else
        waiting[j] = FALSE;
    remainder section
} while (TRUE);
```

Synchronization Hardware

- Example: Bounded-waiting mutual exclusion with test_and_set
  - Mutual exclusion
    - Process P_i can enter its critical section only if either waiting[i] == FALSE or key == FALSE.
    - The value of key can become false only if the TestAndSet() is executed. The first process to execute the TestAndSet() will find key == FALSE; all others must wait.
    - The variable waiting[i] can become false only if another process leaves its critical section; only one waiting[i] is set to FALSE, maintaining the mutual-exclusion requirement.
  - Progress
    - The arguments presented for mutual exclusion also apply here, since a process exiting the critical section either sets lock to FALSE or sets waiting[j] to FALSE. Both allow a process that is waiting to enter its critical section to proceed.

Synchronization Hardware

- Example: Bounded-waiting mutual exclusion with test_and_set
  - Bounded-waiting
    - When a process leaves its critical section, it scans the array waiting in the cyclic ordering ($i + 1$, $i + 2$, ..., $n - 1$, 0, ..., $i - 1$). It designates the first process in this ordering that is in the entry section (waiting[j] == TRUE) as the next one to enter the critical section. Any process waiting to enter its critical section will thus do so within $n - 1$ turns.

Mutex Lock

Mutex Locks

- Previous hardware-based solutions to the critical-section problem are complicated and generally inaccessible to application programmers.
- OS designers build software tools to solve critical-section problem.
  - *Mutex Lock* (互斥锁) is the simplest one.
  - It protects a critical section by first acquire() a lock then release() the lock.
  - Boolean variable is used to indicate whether the lock is available or not.
- Calls to acquire() and release() must be atomic.
  - Usually implemented via hardware atomic instructions.

```
acquire() {
    while (!available)
        ; /* busy waiting */
    available = FALSE;
}

release() {
    available = TRUE;
}
```

Mutex Lock

Mutex Locks

- Process P_i:

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

- Mutex Lock solution requires *busy waiting*。
  - While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the call to acquire().
  - This lock therefore called a *spinlock* (自旋锁).

Concept of Semaphore

- Semaphore (信号量) is a more robust synchronization tool that can behave similarly to a mutex lock but can also provide more sophisticated ways for processes to synchronize their activities.
- Logically, a *semaphore* S is a signed integer variable that, apart from initialization, can only be changed through two *atomic and mutually exclusive* operations:
  - wait(S);
    - also denoted as P(S) or down(S)
  - signal(S);
    - also denoted as V(S) or up(S)
- Semaphores were introduced by the Dutch computer scientist *Edsger Dijkstra*, and such, the wait() operation was originally termed P (from the Dutch *proberen*, “to test”); signal() was originally called V (from *verhogen*, “to increment”).

Concept of Semaphore

- Critical Section of n Processes
  - Shared data:

```
semaphore mutex; /* initiallized to 1 */
```

  - Process P_i:

```
do {
    wait(mutex);
    critical section
    signal(mutex);
    remainder section
} while (TRUE);
```

  - Access is via two *atomic* operations defined as:

```
wait(S) {
    while (S <= 0)
        ; /* busy waiting; do nothing */
    S--;
}

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

Semaphores

Concept of Semaphore

- It must guarantee that no two processes can execute wait() and signal() on the same semaphore at the same time.
- Thus, 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.
- To avoid busy waiting, when a process has to wait, it will be put in a blocked queue of processes waiting for the same event.
  - 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

Semaphores

- **Implementation of Semaphore**
  - Define semaphore as a C struct:

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

  - Two operations are assumed:
    - block(): place the process that invokes the operation on the appropriate waiting queue (suspending the process)
    - wakeup(): remove one of processes in the waiting queue and place it in the ready queue (resuming the execution of a blocked process)
  - These two operations are provided by the operating system as basic system calls.

Implementation of Semaphore

- Semaphore operations now defined as

```
wait (semaphore *S) {
    S->value--;          初始化为同类型资源的最大数量
    if (S->value < 0) {
        block(); /* no busy waiting */
        add this process to S->list;
    }
}

signal (semaphore *S) {
    S->value++;
    if (S->value <= 0) { /* S.list not empty */
        remove a process P from S->list;
        wakeup(P);
    }
}
```

- In this implementation, semaphore values may be negative, while its magnitude is the number of processes waiting on that semaphore (the length of waiting list).

Implementation of Semaphore

- It is critical that semaphore operations be executed *atomically*.
  - Recall that we must guarantee that no two processes can execute wait() and signal() operations on the same semaphore at the same time. This is a critical-section problem and may be solved by
    - Inhibiting interrupts in a single-processor environment
    - compare_and_swap() or spinlocks with a multiprocessor
- Two Types of Semaphores
  - Binary semaphore
    - Its integer value can range only between 0 and 1 (really a Boolean);
    - It can be simpler to implement (use waitb() and signalb() operations);
    - same as Mutex Lock
  - Counting semaphore
    - Its integer value can range over an unrestricted domain.

Semaphores

Implementation of Semaphore

- Binary Semaphore Implementation
  - Implementing a binary semaphore S is easy.
  - waitb(S):

```
if (S.value == 1) {
    S.value = 0;
} else {
    block();
    add this process to S.list
}
```

  - signalb(S):

```
if (S.list is empty) {
    S.value = 1;
} else {
    remove a process P from S.list
    wakeup(P);
}
```

Semaphores

Implementation of Semaphore

- Counting Semaphore Implementing
  - We can implement a counting semaphore S using two binary semaphores S1 and S2 (that protect its counter).
  - Data structures:

```
semaphore S;
binary-semaphore S1, S2;
```

  - Initialization:

```
S1.value = 1;
S1.list = NULL;
S2.value = 0;
S2.list = NULL;
S.value = initial value of S;
```

Implementation of Semaphore

- Counting Semaphore Implementing
  - wait(S):

```
waitb(S1);
S.value--;
if (S.value < 0) {
    block();
    add to S.list;
    signalb(S1);
    waitb(S2);
}
signalb(S1);
```

  - signal(S):

```
waitb(S1);
S.value++;
if (S.value <= 0) {
    remove a process P from S.list;
    wakeup(P);
    signalb(S2);
} else
    signalb(S1);
```

Binary Semaphore Implementation

waitb(S):

```
if (S.value == 1) {
    S.value = 0;
} else {
    block();
    add this process to S.list;
}
```

signalb(S):

```
if (S.list is empty) {
    S.value = 1;
} else {
    remove a process P from S.list;
    wakeuo(P);
}
```

Semaphore as a General Synchronization Tool

- Suppose we want to execute a code block B in process P_j only after a code block A executed in process P_i.
  - Shared data:

```
semaphore flag;
```

  - Initialization:

```
flag.value = 0;
flag.list = NULL;
```

  - P_i :

```
…
A
signal(flag);
…
```

  - P_j :

```
…
wait(flag);
B
…
```

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. (we'll talk later)
    - E.g., S and Q be two semaphores initialized to 1.

P_0:	P_1:

`wait(S); /* S-- */`
`wait(Q); /* Q-- */`
`… …`
`signal(S);`
`signal(Q);`
	`wait(Q); /* Q-- */`
	`wait(S); /* S-- */`
	`… …`
	`signal(Q);`
	`signal(S);`

Deadlock may occur here.

- Starvation
  - A process may never be removed from the semaphore queue (say, if LIFO) in which it is suspended.
- Priority Inversion
  - Scheduling problem when lower-priority process holds a lock needed by higher-priority process.

Semaphores

Problems with Semaphores

- Semaphores provide a powerful tool for enforcing mutual exclusion and coordinate processes.
- But wait() and signal() are scattered among several processes. Hence, it is difficult to understand their effects.
  - Usage must be correct in all the processes (correct order, correct variables, no omissions).
- Incorrect use of semaphore operations:
  - signal (mutex) … wait (mutex)
  - wait (mutex) … wait (mutex)
  - Omitting of wait (mutex) or signal (mutex) (or both)
- One bad (or malicious) process can fail the entire collection of processes.

Classical Problems

- **Classical Problems**
  - Three famous problems are used as examples of a large class of concurrency-control problems.
    - Bounded-Buffer
    - Readers and Writers
    - Dining-Philosophers
  - These problems are used for testing nearly every newly proposed synchronization scheme. We use semaphores for synchronization since that is the traditional way to present such solutions. However, actual implementations of these solutions could use Mutex Locks in place of binary semaphores.

The Bounded-Buffer Problem

- Reminder: Producer-Customer problem with race condition

```
#define N 100                                 /* number of slots in the buffer */
int count = 0;                                /* number of items in the buffer */

void producer(void)
{
        int item;

        while (TRUE) {                        /* repeat forever */
                item = produce_item( );       /* generate next item */
                if (count == N) sleep( );     /* if buffer is full, go to sleep */
                insert_item(item);            /* put item in buffer */
                count = count + 1;            /* increment count of items in buffer */
                if (count == 1) wakeup(consumer);  /* was buffer empty? */
        }
}


void consumer(void)
{
        int item;

        while (TRUE) {                        /* repeat forever */
                if (count == 0) sleep( );     /* if buffer is empty, got to sleep */
                item = remove_item( );        /* take item out of buffer */
                count = count − 1;            /* decrement count of items in buffer */
                if (count == N − 1) wakeup(producer);  /* was buffer full? */
                consume_item(item);           /* print item */
        }
}
```

Classical Problems

- **The Bounded-Buffer Problem**
  - Producer-Customer Bounded-Buffer Problem
    - We need three semaphores:
      - A semaphore mutex (initialized to 1) to have mutual exclusion on buffer access.
      - A semaphore full (initialized to 0) to synchronize producer and consumer on the number of consumable items.
      - A semaphore empty (initialized to n) to synchronize producer and consumer on the number of empty spaces.
    - Shared data:

```
semaphore mutex = 1;
semaphore empty = BUFFER_SIZE;
semaphore full = 0;
```

The Bounded-Buffer Problem

- Producer-Customer Bounded-Buffer Problem
  - Producer Process:

```
do {
        …
    produce an item in next_produced
        …
      /* must perform wait(empty) before wait(mutex) */
    wait(empty); } 不可交换顺序
    wait(mutex); }
        …
    add next_produced to buffer
        …
    signal(mutex);
    signal(full);
} while (TRUE);
```

Classical Problems

- **The Bounded-Buffer Problem**
  - Producer-Customer Bounded-Buffer Problem
    - Consumer Process:

```
do {
      /* must perform wait(full) before wait(mutex) */
    wait(full);    } 不可交换顺序
    wait(mutex);
        …
    remove an item from buffer to next_consumed
        …
    signal(mutex);
    signal(empty);
        …
    consume the item in next_consumed
        …
} while (TRUE);
```

The Bounded-Buffer Problem

- Producer-Customer Bounded-Buffer Problem
  - Remarks (from consumer point of view):
    - Putting signal(empty) inside the critical section of the consumer (instead of outside) has no effect since the producer must always wait for both semaphores before proceeding.
    - The consumer must perform wait(full) *before* wait(mutex), otherwise deadlock occurs if consumer enters critical section while the buffer is empty (full.value == 0).
    - The producer must perform wait(empty) *before* wait(mutex), otherwise deadlock occurs if producer enters critical section while the buffer is full (empty.value == 0).
  - Conclusion
    - using semaphores is a difficult art ...

Classical Problems

- **Readers-Writers Problem**
  - A data set (data repository) 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
    - Multiple readers are allowed to read at the same time.
    - Only one single writer can access the shared data at the same time.

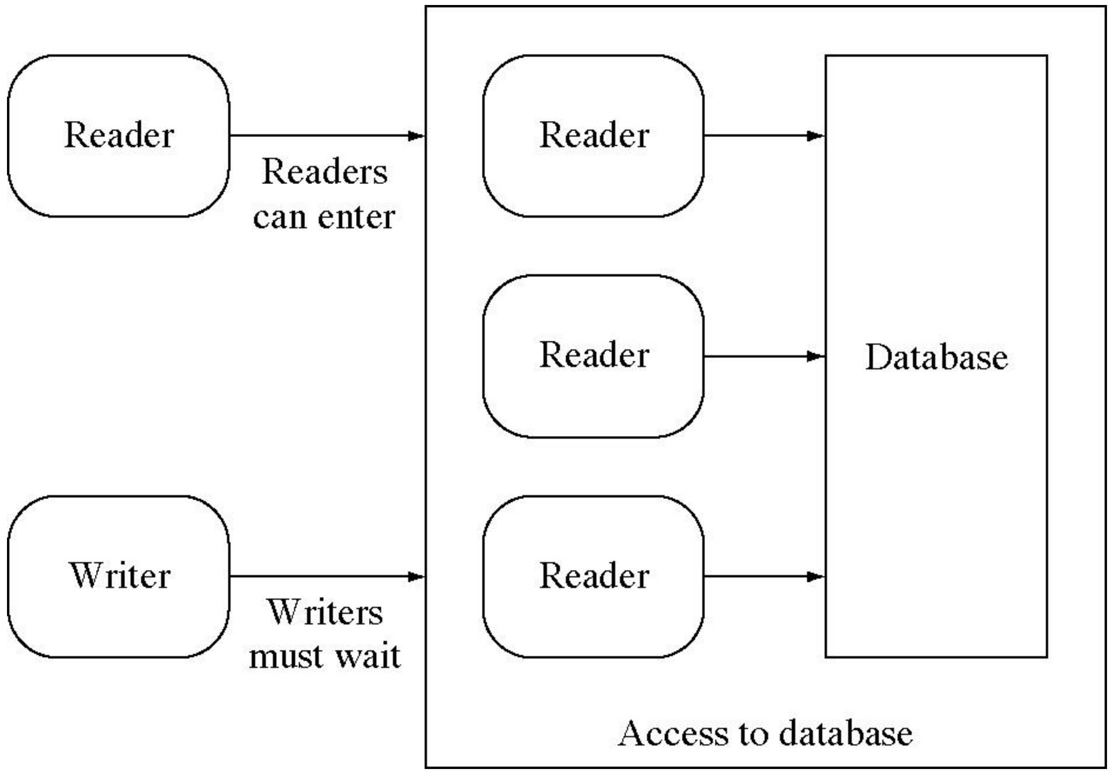
i

Classical Problems

- **Readers-Writers Problem**
  - Readers-Writers Dynamics
    - Any number of reader activities and writer activities are running.
    - At any time, a reader activity may wish to read data.
    - At any time, a writer activity may want to modify the data.
    - Any number of readers may access the data simultaneously.
    - During the time a writer is writing, no other reader or writer may access the shared data.

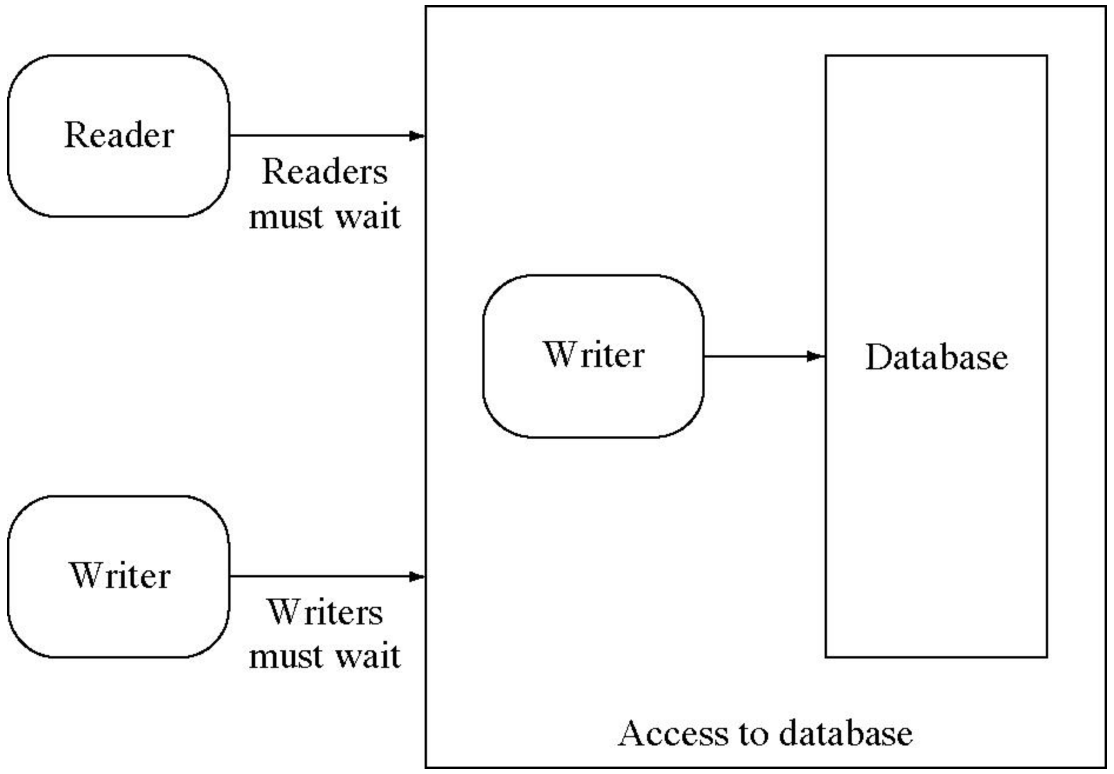
Classical Problems

- **Readers-Writers Problem**
  - Readers-Writers with active readers



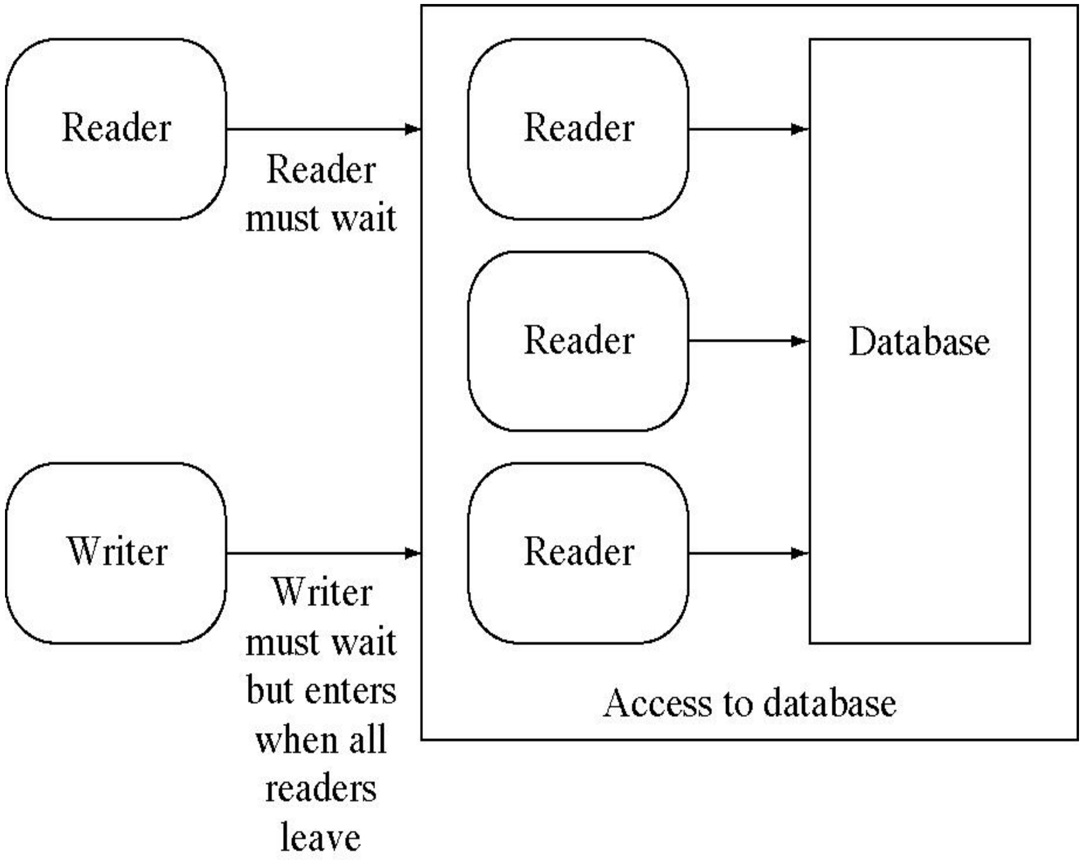
Classical Problems

- **Readers-Writers Problem**
  - Readers-Writers with an active writer



Classical Problems

- **Readers-Writers Problem**
  - Should readers wait for waiting writer?



Classical Problems

- **Readers-Writers Problem**
  - There are various versions with different readers and writers preferences:
    - The First Readers-Writers Problem
      - It is requires that no reader will be kept waiting unless a writer has obtained access to the shared data.
      - In a solution to this case writers may starve.
    - The Second Readers-Writers Problem
      - It is requires that once a writer is ready, no new readers may start reading.
      - In a solution to this case readers may starve.
  - Recap: Starvation
    - A process may never be removed from the semaphore queue in which it is suspended.

Classical Problems

- **Readers-Writers Problem**
  - Solution to the First Readers-Writers Problem
    - read_count (initialized to 0) counter keeps track of how many processes are currently reading.
    - mutex semaphore (initialized to 1) provides mutual exclusion for updating read_count.
    - rw_mutex semaphore (initialized to 1) provides mutual exclusion for the writers; it is also used by the first or last reader that enters or exits the critical section.

Readers-Writers Problem

- Solution to the First Readers-Writers Problem
  - Shared data

```
semaphore mutex = 1;
semaphore rw_mutex = 1;
int read_count = 0;
```

  - Writer Process:

```
do {
    wait(rw_mutex);
        …
    writing is performed
        …
    signal(rw_mutex);
} while(TRUE);
```

Readers-Writers Problem

- Solution to the First Readers-Writers Problem
  - Reader Process:

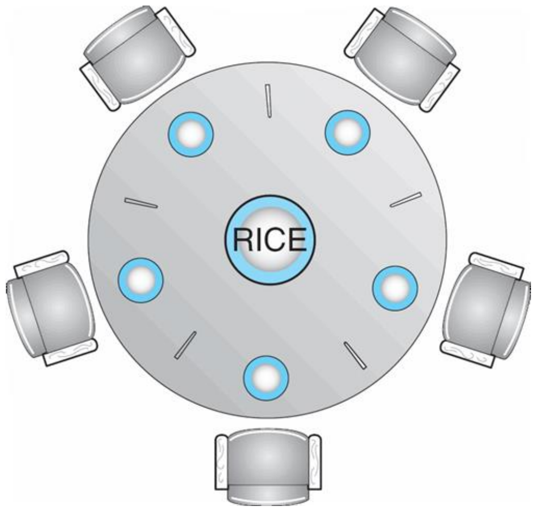
```
do {
    wait(mutex);
    read_count++;
    if (read_count == 1)
        wait(rw_mutex);
    signal(mutex);
        …
    reading is performed
        …
    wait(mutex);
    read_count--;
    if (read_count == 0)
        signal(rw_mutex);
    signal(mutex);
} while(TRUE);
```

Readers-Writers Problem

- Reader–Writer Locks
  - The readers–writers problem and its solutions have been generalized to provide *reader–writer locks* on some systems.
  - Acquiring a reader–writer lock requires specifying the mode of the lock: either read or write access.
    - Only to read shared data requests the lock in read mode, to modify the shared data in write mode.
  - Multiple processes can concurrently acquire a lock in read mode, but only one process may acquire the lock for writing, as exclusive access is required for writers.
- Reader–writer locks are most useful in the following situations:
  - It is easy to identify which processes only read shared data and which processes only write shared data.
  - It has more readers than writers.
    - Reader–writer locks generally require more overhead to establish than semaphores or mutual-exclusion locks. The increased concurrency of multiple readers compensates for the overhead in setting up the reader–writer lock.

Classical Problems

- **Dining-Philosophers Problem**
  - Philosophers spend their lives alternating between thinking and eating. Five philosophers can be seated around a circular table. There is a shared bowl of rice. In front of each one is a plate. Between each pair of philosophers there is a single chopstick, so there are five chopsticks.
  - When a philosopher thinks, she does not interact with her colleagues. From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her.
  - A philosopher may pick up only one chopstick at a time. Obviously, she cannot pick up a chopstick that is already in the hand of a neighbor.
  - When a hungry philosopher has both her chopsticks at the same time, she eats without releasing the chopsticks. When she is finished eating, she puts down both chopsticks and starts thinking again.



Dining-Philosophers Problem

- Dining-Philosophers dynamics for each philosopher

```
do {
    thinks for a while;
    gets the left chopstick;
    gets the right chopstick;
    eats for a while;
    puts the two chopsticks down;
while (TRUE)
```

- The challenge is to grant requests for chopsticks while avoiding deadlock and starvation. This Illustrates the difficulty of allocating resources among process without deadlock and starvation.
- *Deadlock* can occur if everyone tries to get their chopsticks at once. Each gets a left chopstick, and is stuck, because each right chopstick is someone else's left chopstick.
- One simple solution is to represent each chopstick with a semaphore. Each philosopher is a process trying 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.

Dining-Philosophers Problem

- Solution to the Dining-Philosophers Problem
  - Shared data:

```
semaphore chopstick[5];
```

  - Initialization:

```
for (int i = 0; i < 5; i++) {
    chopstick[i].value = 1;
}
```

  - Process P_i:

```
do {
    think for a while
    wait(chopstick[i]); /* acquire the left chopstick */
    wait(chopstick[(i + 1) mod 5]); /* acquire the right
                                       chopstick */
    eat for a while
    signal(chopstick[(i + 1) mod 5]); /* release the right
                                         chopstick */
    signal(chopstick[i]); /* release the left chopstick */
} While (TRUE)
```

Dining-Philosophers Problem

- Although this solution guarantees that no two neighbors are eating simultaneously, it could create a deadlock.
  - Suppose that each of the five philosophers grabs her left chopstick at the same time. All the elements of chopstick will now be equal to 0. Any requirement for right chopstick will be delayed forever.
- Possible solutions to avoid deadlock:
  - Solution 1. Allow at most four philosophers to be sitting at the table that try to eat simultaneously. Then one philosopher can always eat when the other three are holding one chopstick.
  - Solution 2. Allow a philosopher to pick up the chopsticks only if both are available (picking must be done in a critical section).
  - Solution 3. 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.
- Any satisfactory solution to the dining-philosophers problem must guard against the possibility of starvation. A deadlock-free solution does not necessarily eliminate the possibility of starvation.

Dining-Philosophers Problem

- Implementation of *Solution 1*.
  - Use another semaphore T that would limit at four, the number of philosophers "sitting at the table".
  - Shared data:

```
semaphore chopstick[5], T;
```

  - Initialization:

```
for (int i = 0; i < 5; i++)
    chopstick[i].value = 1;
T.value = 4;
```

  - Process P$_i$:

```
do {
      /* think for a while */
    wait(T);
    wait(chopstick[i]);
    wait(chopstick[(i + 1) mod 5]);
      /* eat for a while */
    signal(chopstick[(i + 1) mod 5]);
    signal(chopstick[i]);
    signal(T);
} While (TRUE);
```