Process Synchronization

Chapter 5: Process Synchronization

- Background
- The Critical-Section Problem
- Peterson’s Solution
- Dekker’s algorithm
- Synchronization Hardware
- Mutex Locks
- Semaphores
- Classic Problems of Synchronization
- Monitors
- Synchronization Examples
- Alternative Approaches

Objectives

- To present the concept of process synchronization.
- To introduce the critical-section problem, whose solutions can be used to ensure the consistency of shared data
- To present both software and hardware solutions of the critical-section problem
- To examine several classical process-synchronization problems
- To explore several tools that are used to solve process synchronization problems

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

- `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 `register1 = counter` {register1 = 5}
S1: producer execute `register1 = register1 + 1` {register1 = 6}
S2: consumer execute `register2 = counter` {register2 = 5}
S3: consumer execute `register2 = register2 - 1` {register2 = 4}
S4: producer execute `counter = register1` {counter = 6 }
S5: consumer execute `counter = register2` {counter = 4}

Critical Section Problem

- Consider system of ***n*** processes {$p_0, p_1, \ldots p_{n-1}$}
- 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);
```

Algorithm for Process P_i

```
do {

        while (turn == j);

                critical section

        turn = j;

                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

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

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 for Process P_i

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

Peterson's Solution (Cont.)

- Provable that the three CS requirement are met:
  1. Mutual exclusion is preserved

     $\mathbf{P_i}$ enters CS only if:

     either **`flag[j] = false`** or **`turn = i`**
  2. Progress requirement is satisfied
  3. Bounded-waiting requirement is met

Dekker’s algorithm

- **Dekker's algorithm** is the first known correct solution to the problem in concurrent programming.
- The solution is attributed to Dutch mathematician Th. J. Dekker by Edsger W. Dijkstra in an unpublished paper on sequential process descriptions and his manuscript on cooperating sequential processes.
- It allows two threads to share a single-use resource without conflict, using only shared memory for communication.
- It avoids the strict alternation of a naïve turn-taking algorithm, and was one of the first mutual exclusion algorithms to be invented.

Dekker’s Algorithm

```
//flag[] is boolean array; and turn is an integer
flag[0] = false
flag[1] = false
turn    = 0   // or 1
```

```
P0:
   flag[0] = true;
   while (flag[1] == true) {
      if (turn ≠ 0) {
         flag[0] = false;
         while (turn ≠ 0) {
           // busy wait
         }
         flag[0] = true;
      }
   }

   // critical section
   ...
   turn    = 1;
   flag[0] = false;
   // remainder section
```

```
P1:
   flag[1] = true;
   while (flag[0] == true) {
      if (turn ≠ 1) {
         flag[1] = false;
         while (turn ≠ 1) {
           // busy wait
         }
         flag[1] = true;
      }
   }

   // critical section
   ...
   turn    = 0;
   flag[1] = false;
   // remainder section
```

Peterson’s V/s Dekker

```
Peterson's: "I want to enter."                   flag[0]=true;
            "You can enter next."                turn=1;
            "If you want to enter and            while(flag[1]==true&&turn==1){
            it's your turn I'll wait."           }
            Else: Enter CS!                      // CS
            "I don't want to enter any more."    flag[0]=false;

Dekker's:   "I want to enter."                   flag[0]=true;
            "If you want to enter                while(flag[1]==true){
             and if it's your turn                 if(turn!=0){
             I don't want to enter any more."        flag[0]=false;
            "If it's your turn                       while(turn!=0){
             I'll wait."                             }
            "I want to enter."                       flag[0]=true;
                                                   }
                                                 }
            Enter CS!                            // CS
            "You can enter next."                turn=1;
            "I don't want to enter any more."    flag[0]=false;
```

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);
```

Hardware Solution

```
boolean TestAndSet(boolean *lockValue)
{
        boolean RValue = *lockvalue;
        *lockvalue = true;
        return RValue;
}
/*Code for process Pi Here 'i' can be 0 to n-1 , where 'n' is the number of
concurrently executing processes.*/

boolean lock = false;
//Entry Section
do
{
        while(TestAndSet(&lock))
        {
                ; // do nothing as another process is in its critical region.... (wait.)
        }

                <Critical section>

  //Exit section
  lock = false; // Let another process execute in their critical section

  }while(1);
```

Statement Number			TestAndSet	
		lock	lockValue	RValue
1	Initially	false	-	-
2	Entry Section	false	-	-
3	*TestAndSet Function*	false	false	-
4	*TestAndSet Function*		true	false
5	So P_i enters in its critical section and value of *lock* =*true*			
6	Meanwhile P_k also wants to enter in its critical section			
7	Entry Section	true	-	-
8	*TestAndSet Function*	true	true	-
9	*TestAndSet Function*		true	true
10	P_k could not enter in its critical section as value of *lock* =*true*			
11	Exit section for P_i	false		
	Again P_k wants to enter in its critical section			
12	Entry Section	false	-	-
13	*TestAndSet Function*	true	true	-
14	*TestAndSet Function*		true	false
15	P_k could now enter in its critical section			

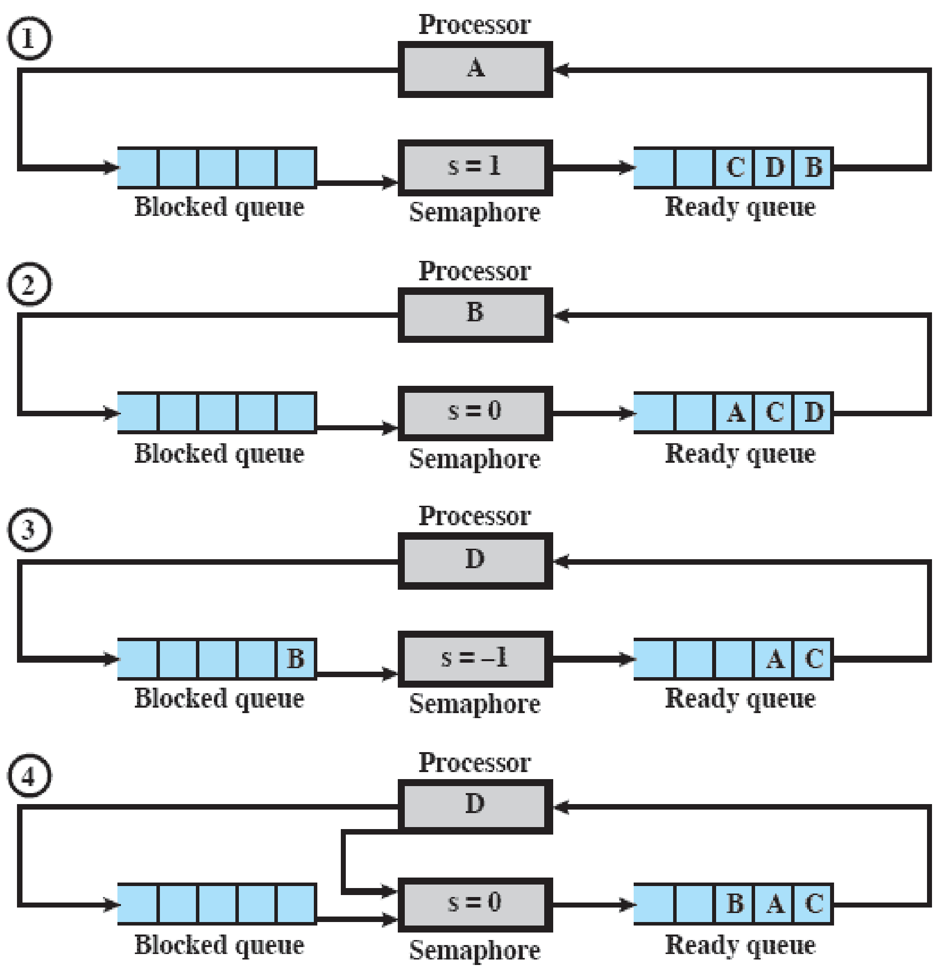
Mutex Locks

- 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:
    S1;
    signal(synch);
P2:
    wait(synch);
    S2;
```

- Can implement a counting semaphore ***S*** as a binary semaphore
- So, Semaphores can be used as Mutex for Crtical section problem solution, ordering the execution of events/statement, managing the multiple instances of same resources as counting semaphores

Examples

1. Let S and Q be two semaphores initialized to 1, where P0 and P1 processes the following statements. What does the situation depicts?

P0	P1
wait(S);	wait(Q);
wait(Q);	wait(S);
signal(S);	signal(Q);
signal(Q);	signal(S);

2. Consider the methods used by processes P1 and P2 for accessing their critical sections whenever needed, as given below. The initial values of shared boolean variables S1 and S2 are randomly assigned.

Method used by P1	**Method used by P2**
while (S1 == S2) ;	while (S1 != S2) ;
critical section	critical section
S1 = S2;	S2 = not (S1);

Which critical section requirement has fulfilled by the above execution? Justify

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 as General Synchronization Tool

- **Counting** semaphore – integer value can range over an unrestricted domain
- **Binary** semaphore – integer value can range only between 0 and 1; can be simpler to implement
  - Also known as **mutex locks**
- Provides mutual exclusion

```
Semaphore mutex;    //  initialized to 1
do {
    wait (mutex);
      // Critical Section
    signal (mutex);
        // remainder section
} while (TRUE);
```

- While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the entry code.
- Busy waiting wastes CPU cycles that some other process might be able to use productively.
- This type of semaphore is also called a spinlock because the process "spins" while waiting for the lock.
- Spinlocks do have an advantage in that no context switch is required when a process must wait on a lock, and a context switch may take considerable time
- To overcome the need for busy waiting, we can modify the definition of the wait() and signal() semaphore operations.
- Rather than engaging in busy waiting, the process can *block* itself. The block operation places a process into a waiting queue associated with the semaphore

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;
  ```
  ```

Semaphore


1
Processor
A
s = 1
Blocked queue
Semaphore
C D B
Ready queue
2
Processor
B
s = 0
Blocked queue
Semaphore
A C D
Ready queue
3
Processor
D
B
s = –1
Blocked queue
Semaphore
A C
Ready queue
4
Processor
D
s = 0
Blocked queue
Semaphore
B A C
Ready queue

Implementation with no Busy waiting (Cont.)

Example of Semaphore Mechanism

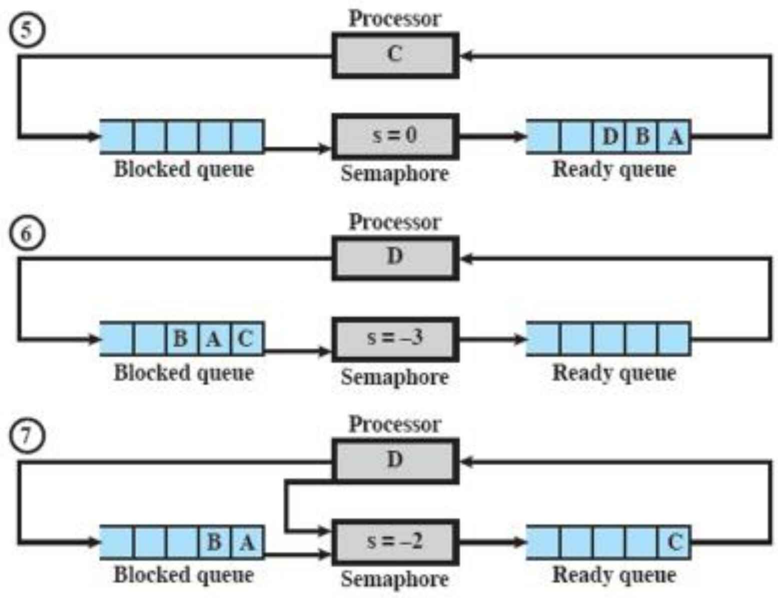


Figure 5.5 Example of Semaphore Mechanism

There are 5 P operations and 5 V operations on a semaphore S which is initialized to 3. At the end of execution what will be the value of S?

Example

Initial Value of counting semaphore S=3

Process	Function call	Semaphore value	S->list (Blocked list)
P1	wait( )	S=2 (3→2)	Empty list
P2	wait( )	S=1(2→1)	Empty list
P3	wait( )	S=0 (1→0)	Empty list
P4	wait ( )	S=-1 (0→-1)	P4 **[if S is negative, then the magnitude of semaphore value represents number of process in blocked list]**
P5	wait ( )	S=-2 (-1→-2)	P4, P5 **[if S is negative, then the magnitude of semaphore value represents number of process in blocked list]**
P1	signal( )	S=-1 (-2+1=-1)	P4 is removed from blocked list, and allowed to enter into critical section
P2	signal( )	S=0 (-1+1=0)	P5 is removed from blocked list, and allowed to enter into critical section
P3	signal( )	S=1(0+1=1)	Empty list
P4	signal( )	S=2 (1+1=2)	Empty list
P5	signal( )	S=3 (2+1=3)	Empty list

Example:

A shared variable x, initialized to zero, is operated on by four concurrent processes W, X, Y, Z as follows. Each of the processes W and X reads x from memory, increments by one, stores it to memory, and then terminates. Each of the processes Y and Z reads x from memory, decrements by two, stores it to memory, and then terminates. Each process before reading x invokes the P operation (i.e., wait) on a counting semaphore S and invokes the V operation (i.e., signal) on the semaphore S after storing x to memory. Semaphore S is initialized to two. What is the maximum possible value of x after all processes complete execution? Justify your answer.

Given Scenario

W	X	Y	Z
Wait(S)	Wait(S)	Wait(S)	Wait(S)
R(X)	R(X)	R(X)	R(X)
X=X+1	X=X+1	X=X-2	X=X-2
W(X)	W(X)	W(X)	W(X)
Signal(S)	Signal(S)	Signal(S)	Signal(S)

S=2, what is the maximum value of X after completing the execution?

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

S=1, F=0 , E=n

Void producer()
While (T)
{
Produce()

Wait(E)
Wait(S)

Append()

Signal(S)
Signal(F)
}

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);
```

```
Void consumer()

While(T)
{ wait(F)
  wait(S)

  take()


  Signal(S)
  Signal(E)

Use()

}
```

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 (Cont.)

- The structure of a writer process

```
do {
        wait(rw_mutex);

           ...
        /* writing is performed */

           ...
    signal(rw_mutex);
} while (true);
```

Readers-Writers Problem (Cont.)

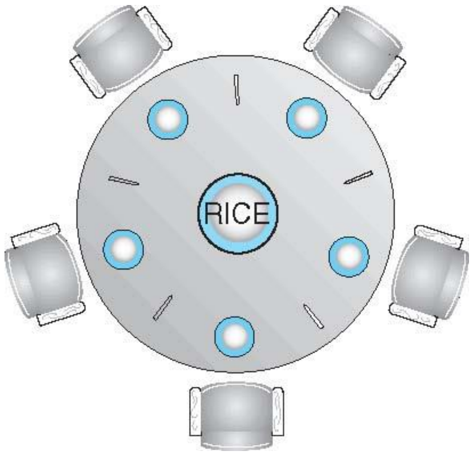
- The structure of a reader process

```
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);
```

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 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);
```

-

Problems

- Case I: If Po, P1 takes their left chopstick first
- Case II : If all the philosophers take their left chopstick first
- Case III: How to come out from case II?
- Case IV: Can P1 and P3 (independent) start having the food concurrently?
- Case V: can P2 and P4 (independent) start having the food concurrently?
- Case VI: Can odd numbered processes are allowed concurrently?

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.

Problems with Semaphores

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