Chapter 7: Synchronization Examples

Chapter 7: Synchronization Examples

- Explain the bounded-buffer, readers-writers, and dining philosophers synchronization problems.
- Describe the tools used by Linux and Windows to solve synchronization problems.
- Illustrate how POSIX and Java can be used to solve process synchronization problems.

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
  - mutex binary semaphore provides mutual exclusion for accesses to the buffer pool.
- Semaphore **`full`** initialized to the value 0
  - Semaphore full counts the number of full buffers
- Semaphore **`empty`** initialized to the value n
  - Semaphore empty counts the number of empty buffers

Bounded Buffer Problem - Producer Process

Initially ‘n’ empty buffers, 0 full buffers.

- The structure of the producer process

```
while (true) {
        ...
         /* produce an item in next_produced */
        ...
      wait(empty);      // wait if no buffer is empty
      wait(mutex);      // wait if mutex is 0; then decrement
         ...
        /* add next produced to the buffer */
         ...
      signal(mutex);    // increment mutex
      signal(full);     // increment the filled buffer count
 }
```

Bounded Buffer Problem - Consumer Process

- The structure of the consumer process.
- Note the symmetry between the producer and the consumer

```
while (true) {
    wait(full);      // wait if no buffer is full
    wait(mutex);
        ...
       /* remove an item from buffer to next_consumed */
        ...
    signal(mutex);
    signal(empty);   // increment the empty buffer count
        ...
       /* consume the item in next consumed */
        ...
    }
```

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, read and write) the database
- 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

Readers-Writers Problem Variations

- 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.
  - For this reason, other variants of the problem have been proposed.

Readers-Writers Problem - Semaphores

- In the solution to the first readers–writers problem, the reader processes share the following data structures:

  semaphore rw_mutex = 1;

  semaphore mutex = 1;

  int read_count = 0;

The binary semaphores mutex and rw_mutex are initialized to 1;

Read_count is a counting semaphore initialized to 0.

- 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 is common to both reader and writer processes.
  - It 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 that enter or exit while other readers are in their critical sections.

Readers-Writers Problem - Writer Process

- The structure of a writer process of the first readers–writers problem.

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

Readers-Writers Problem - Reader Process

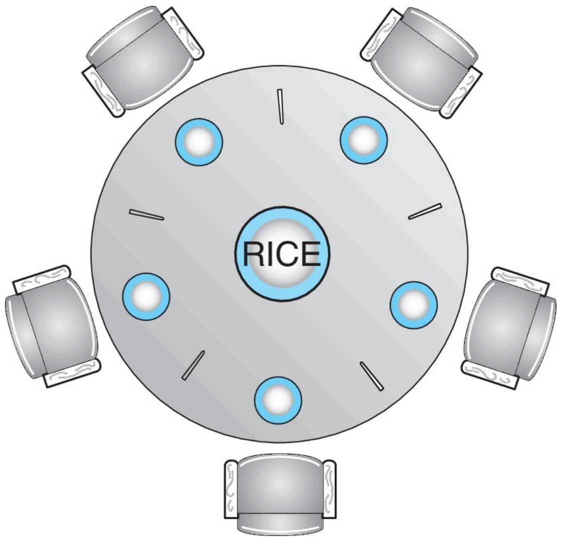
- The structure of a reader process of the first readers–writers problem.

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

Dining-Philosophers Problem

- ✓ The **dining-philosophers problem** is considered a classic synchronization problem because 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

- Five 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
- Shared data:
    - Bowl of rice (data set)
    - Semaphore chopstick [5] initialized to 1

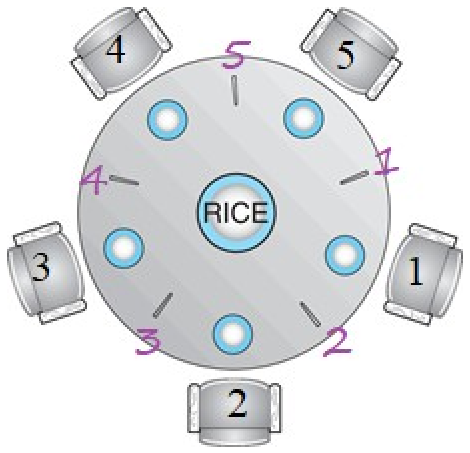


Dining-Philosophers Problem Algorithm

- Semaphore Solution:
  Represent each chopstick with a semaphore.
- The structure of Philosopher *i*:

```
    while  (true){
         wait (chopstick[i] );
         wait (chopStick[ (i + 1) % 5] );

          /* eat for awhile */

         signal (chopstick[i] );
         signal (chopstick[ (i + 1) % 5] );

          /* think for awhile */

    }
```

- What is the problem with this algorithm?
  Could create a deadlock.



e.g., right & left chopsticks:

i =1 (right)
(i+1)% 5 = (1+1)%5 = 2 (left)

i =2 (right)
(i+1)% 5 = (2+1)%5 = 3 (left)

i =3 (right)
(i+1)% 5 = (3+1)%5 = 4 (left)

i =4 (right)
(i+1)% 5 = (4+1)%5 = 5 (left)

i =5 (right)
(i+1)% 5 = (5+1)%5 = 1 (left)

Suggested Solutions

Several possible remedies to the deadlock problem:

- Allow at most four philosophers to be sitting simultaneously at the table.
- Allow a philosopher to pick up her chopsticks only if both chopsticks are available (to do this, she must pick them up in a critical section).
- Use an asymmetric solution—that is, an odd-numbered philosopher picks up first his left chopstick and then his right chopstick, whereas an even numbered philosopher picks up his right chopstick and then his left chopstick.

.

Monitor Solution to Dining Philosophers

- This solution imposes the restriction that a philosopher may pick up his chopsticks only if both of them are available.
- We need to distinguish among three states in which we may find a philosopher. For this purpose, we introduce the following data structure:

  enum {THINKING, HUNGRY, EATING} state[5];
- Philosopher *i* can set the variable state[i] = EATING only if his two neighbors are not eating: (state[(i+4) % 5] != EATING) and(state[(i+1) % 5] != EATING).
- We also need to declare

  condition self[5];

  This allows philosopher *i* to delay himself when he is hungry but is unable to obtain the chopsticks he needs.
- The distribution of the chopsticks is controlled by the monitor DiningPhilosophers

Monitor Solution to the Dining Philosophers (Cont.)

- Each philosopher *i* invokes the operations **`pickup()`** and **`putdown()`** in the following sequence:

```
DiningPhilosophers.pickup(i);

     /** EAT **/

DiningPhilosophers.putdown(i);
```

- No deadlock, but starvation is possible
- 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

Monitor Solution to Dining Philosophers

```
monitor DiningPhilosophers
{
   enum { THINKING; HUNGRY, EATING) state [5] ;
   condition self [5];

   void pickup (int i) {
          state[i] = HUNGRY;
          test(i);
          if (state[i] != EATING)
                 self[i].wait;
   }

   void putdown (int i) {
          state[i] = THINKING;
           // test left and right neighbors
           test((i + 4) % 5);
           test((i + 1) % 5);
   }
```

Monitor Solution to Dining Philosophers (Cont.)

```
    void test (int i) {
            if ((state[(i + 4) % 5] != EATING) &&
            (state[i] == HUNGRY) &&
            (state[(i + 1) % 5] != EATING) )
              {
                   state[i] = EATING ;
                   self[i].signal () ;
            }
    }

    initialization_code() {
           for (int i = 0; i < 5; i++)
            state[i] = THINKING;
    }
}
```

Kernel Synchronization - Windows

- Uses interrupt masks to protect access to global resources on uniprocessor systems
- Uses **spinlocks** on multiprocessor systems
  - Spinlocking-thread will never be preempted

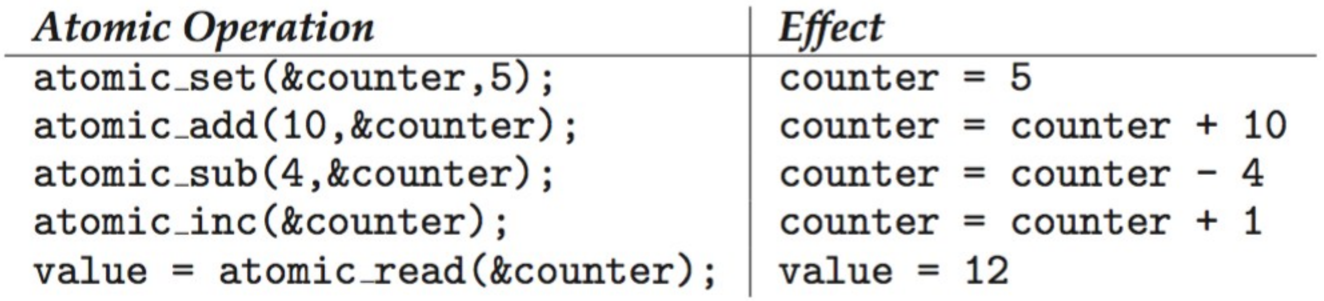
Linux Synchronization

- Linux:
  - Prior to kernel Version 2.6, disables interrupts to implement short critical sections
  - Version 2.6 and later, fully preemptive
- Linux provides:
  - Semaphores
  - atomic integers
  - spinlocks
  - reader-writer versions of both
- On single-cpu system, spinlocks replaced by enabling and disabling kernel preemption

Linux Synchronization

- The simplest synchronization technique within the Linux kernel is an atomic integer, which is represented using the opaque data type atomic_t
- Atomic variables
  `atomic_t` is the type for atomic integer
- The following code illustrates declaring an atomic integer counter and then performing various atomic operations:

```
atomic_t counter;
int value;
```



Atomic Operation	*Effect*
`atomic_set(&counter,5);`	`counter = 5`
`atomic_add(10,&counter);`	`counter = counter + 10`
`atomic_sub(4,&counter);`	`counter = counter - 4`
`atomic_inc(&counter);`	`counter = counter + 1`
`value = atomic_read(&counter);`	`value = 12`

Data type whose concrete data structure is not defined in an interface. This enforces information hiding.

POSIX Synchronization

- POSIX API provides
  - mutex locks
  - semaphores
  - condition variable
- Widely used on UNIX, Linux, and macOS

POSIX Mutex Locks

- Creating and initializing the lock

```
#include <pthread.h>

pthread_mutex_t mutex;

/* create and initialize the mutex lock */
pthread_mutex_init(&mutex,NULL);
```

- Acquiring and releasing the lock

```
/* acquire the mutex lock */
pthread_mutex_lock(&mutex);

/* critical section */

/* release the mutex lock */
pthread_mutex_unlock(&mutex);
```

POSIX Semaphores

- POSIX provides two versions – **named** and **unnamed**.
- Named semaphores can be used by unrelated processes, unnamed cannot.

POSIX Named Semaphores

- Creating an initializing the semaphore:

```
#include <semaphore.h>
sem_t *sem;

/* Create the semaphore and initialize it to 1 */
sem = sem_open("SEM", O_CREAT, 0666, 1);
```

- Another process can access the semaphore by referring to its name **`SEM`**.
- Acquiring and releasing the semaphore:

```
/* acquire the semaphore */
sem_wait(sem);

/* critical section */

/* release the semaphore */
sem_post(sem);
```

POSIX Unnamed Semaphores

The sem init() function is passed three parameters:
1. A pointer to the semaphore
2. A flag indicating the level of sharing. 0 means no sharing.
3. The semaphore's initial value

- Creating an initializing the semaphore:

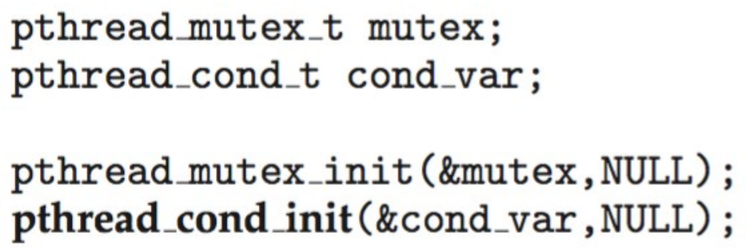
```
#include <semaphore.h>
sem_t sem;

/* Create the semaphore and initialize it to 1 */
sem_init(&sem, 0, 1);
```

- Acquiring and releasing the semaphore:

```
/* acquire the semaphore */
sem_wait(&sem);

/* critical section */

/* release the semaphore */
sem_post(&sem);
```

POSIX Condition Variables

- Since POSIX is typically used in C and C does not provide a monitor, POSIX condition variables are associated with a POSIX mutex lock to provide mutual exclusion:
- Creating and initializing the condition variable:



```
pthread_mutex_t mutex;
pthread_cond_t cond_var;

pthread_mutex_init(&mutex,NULL);
pthread_cond_init(&cond_var,NULL);
```

POSIX Condition Variables

- Thread waiting for the condition **`a == b`** to become true:

```
pthread_mutex_lock(&mutex);
while (a != b)
     pthread_cond_wait(&cond_var, &mutex);

pthread_mutex_unlock(&mutex);
```

- Thread signaling another thread waiting on the condition variable:

```
pthread_mutex_lock(&mutex);
a = b;
pthread_cond_signal(&cond_var);
pthread_mutex_unlock(&mutex);
```

Java Synchronization

- Java provides rich set of synchronization features:
  - Java monitors
  - Reentrant locks
  - Semaphores
  - Condition variables

Java Monitors

- Every Java object has associated with it a single lock.
- If a method is declared as `synchronized`, a calling thread must own the lock for the object.
- If the lock is owned by another thread, the calling thread must wait for the lock until it is released.
- Locks are released when the owning thread exits the `synchronized` method.

Bounded Buffer – Java Synchronization

```
public class BoundedBuffer<E>
{
   private static final int BUFFER_SIZE = 5;

   private int count, in, out;
   private E[] buffer;

   public BoundedBuffer() {
      count = 0;
      in = 0;
      out = 0;
      buffer = (E[]) new Object[BUFFER_SIZE];
   }

   /* Producers call this method */
   public synchronized void insert(E item) {
      /* See Figure 7.11 */
   }

   /* Consumers call this method */
   public synchronized E remove() {
      /* See Figure 7.11 */
   }
}
```

Java Synchronization

- A thread typically calls wait() when it is waiting for a condition to become true.
- How does a thread get notified?
- When a thread calls **`notify()`**:

1. An arbitrary thread T is selected from the wait set
2. T is moved from the wait set to the entry set
3. Set the state of T from blocked to runnable.

- T can now compete for the lock to check if the condition it was waiting for is now true.

Bounded Buffer – Java Synchronization

```
/* Producers call this method */
public synchronized void insert(E item) {
   while (count == BUFFER_SIZE) {
      try {
         wait();
      }
      catch (InterruptedException ie) { }
   }

   buffer[in] = item;
   in = (in + 1) % BUFFER_SIZE;
   count++;

   notify();
}
```

Bounded Buffer – Java Synchronization

```
/* Consumers call this method */
public synchronized E remove() {
   E item;

   while (count == 0) {
      try {
         wait();
      }
      catch (InterruptedException ie) { }
   }

   item = buffer[out];
   out = (out + 1) % BUFFER_SIZE;
   count--;

   notify();

   return item;
}
```

Java Reentrant Locks

- Similar to mutex locks
- The **`finally`** clause ensures the lock will be released in case an exception occurs in the **`try`** block.

```
Lock key = new ReentrantLock();

key.lock();
try {
   /* critical section */
}
finally {
   key.unlock();
}
```

Java Semaphores

- Constructor:

```
Semaphore(int value);
```

- Usage:

```
Semaphore sem = new Semaphore(1);

try {
   sem.acquire();
   /* critical section */
}
catch (InterruptedException ie) { }
finally {
   sem.release();
}
```

Java Condition Variables

- Condition variables are associated with an **`ReentrantLock`**.
- Creating a condition variable using **`newCondition()`** method of **`ReentrantLock`**:

```
Lock key = new ReentrantLock();
Condition condVar = key.newCondition();
```

- A thread waits by calling the **`await()`** method, and signals by calling the **`signal()`** method.

Java Condition Variables

- Example:
- Five threads numbered 0 .. 4
- Shared variable **`turn`** indicating which thread's turn it is.
- Thread calls **`doWork()`** when it wishes to do some work. (But it may only do work if it is their turn.
- If not their turn, wait
- If their turn, do some work for awhile ……
- When completed, notify the thread whose turn is next.
- Necessary data structures:

```
Lock lock = new ReentrantLock();
Condition[] condVars = new Condition[5];

for (int i = 0; i < 5; i++)
   condVars[i] = lock.newCondition();
```

Java Condition Variables

```
/* threadNumber is the thread that wishes to do some work */
public void doWork(int threadNumber)
{
  lock.lock();

  try {
    /**
     * If it's not my turn, then wait
     * until I'm signaled.
     */
    if (threadNumber != turn)
      condVars[threadNumber].await();

    /**
     * Do some work for awhile ...
     */

    /**
     * Now signal to the next thread.
     */
    turn = (turn + 1) % 5;
    condVars[turn].signal();
  }
  catch (InterruptedException ie) { }
  finally {
    lock.unlock();
  }
}
```

Alternative Approaches

- Transactional Memory
- OpenMP
- Functional Programming Languages

Transactional Memory

- Consider a function update() that must be called atomically. One option is to use mutex locks:

```
void update ()
{
   acquire();

   /* modify shared data */

   release();
}
```

- A **memory transaction** is a sequence of read-write operations to memory that are performed atomically. A transaction can be completed by adding **`atomic{S}`** which ensure statements in **`S`** are executed atomically:

```
void update ()
{
   atomic {
      /* modify shared data */
   }
}
```

OpenMP

- OpenMP is a set of compiler directives and API that support parallel progamming.

```
void update(int value)
   {
         #pragma omp critical
         {
                count += value
         }
    }
```

The code contained within the **`#pragma omp critical`** directive is treated as a critical section and performed atomically.

Functional Programming Languages

- Functional programming languages offer a different paradigm than procedural languages in that they do not maintain state and therefore are generally immune from race conditions and critical sections.
- Variables are treated as immutable and cannot change state once they have been assigned a value.
- There is increasing interest in functional languages such as Erlang and Scala for their approach in handling data races.

End of Chapter 7