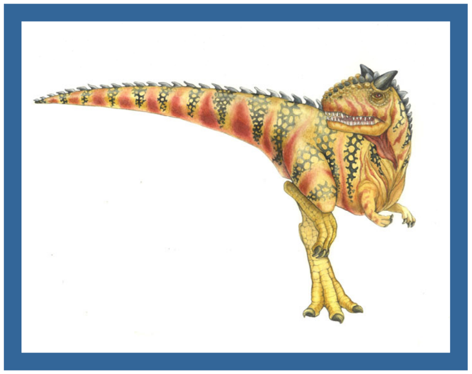
Operating System Concepts

Tenth Edition

Silberschatz, Galvin and Gagne

Chapter 7

Synchronization Examples



Chapter 7: Synchronization Examples

- Classic Problems of Synchronization
- Synchronization within the Kernel
- POSIX Synchronization
- Synchronization in Java
- Alternative Approaches

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 [1]

- ***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 2

- 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 3

- 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);
```


Another Solution?

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

Readers-Writers Problem [1]

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

Readers-Writers Problem [2]

- The structure of a writer process

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

Readers-Writers Problem [3]

- 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 4

- The structure of a reader process

```
do {
        wait(mutex);
        read_count++;
        if (read_count == 1) // first reader
           wait(rw_mutex);
        signal(mutex);
            ...
        /* reading is performed */
            ...
        wait(mutex);
        read_count--;
        if (read_count == 0) // last reader
           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

Readers-Writers without Starvation [1]

- The structure of a writer process

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

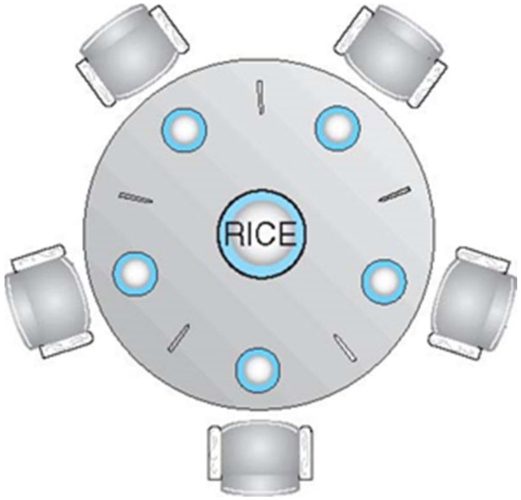
Readers-Writers without Starvation 2

- The structure of a reader process

```
do {
        wait (fairness);
        wait(r_mutex);
        read_count++;
        if (read_count == 1)  //first reader
             wait(rw_mutex);
        signal(r_mutex);
        signal (fairness);
             ...
        /* reading is performed */
             ...
        wait(r_mutex);
        read_count--;
        if (read_count == 0)  //last reader
             signal(rw_mutex);
        signal(r_mutex);
   } while (true);
```

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 [1]

- 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?

Dining-Philosophers Problem Algorithm 2

- <u>Deadlock handling</u>
  - Allow at most 4 philosophers to be sitting simultaneously at the table.
  - Allow a philosopher to pick up the chopsticks only if both are available (picking must be done in a critical section).
  - Works, but reduces parallelism
  - 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.

Monitor Solution to Dining Philosophers [1]

```
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 [2]

```
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;
}
}
```

Monitor Solution to Dining Philosophers 3

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

```
DiningPhilosophers.pickup(i);

EAT

DiningPhilosophers.putdown(i);
```

- Is a deadlock possible? Is starvation possible?

Synchronization Examples

- Windows
- Linux
- Pthreads

Windows Synchronization

- Uses interrupt masks to protect access to global resources on uniprocessor systems
- Uses **spinlocks** on multiprocessor systems
  - Spinlocking-thread will never be preempted
- Also provides **dispatcher objects** user-land which may act mutexes, semaphores, events, and timers
  - **Events**
    - An event acts much like a condition variable
  - **Timers** notify one or more thread when time expired
  - Dispatcher objects either **signaled-state** (object available) or **non-signaled state** (thread will block)

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

Pthreads Synchronization

- Pthreads API is OS-independent
- It provides:
  - mutex locks
  - condition variable
- Non-portable extensions include:
  - read-write locks
  - spinlocks

Alternative Approaches

- Transactional Memory
- OpenMP
- Functional Programming Languages

Transactional Memory

- A **memory transaction** is a sequence of read-write operations to memory that are performed atomically.

```
void update()
{
        atomic
        {
                /* read/write memory */
        }
}
```

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.
- 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.

Multiple-Choice Question

- In the structure of the producer process shown in Slide 5, what happen if `wait(empty)` is replaced with `signal(empty)` and `signal(full)` is replaced with `wait(full)`?

  A) Producer will remain blocked after adding an item in the buffer.

  B) Consumer will remain blocked after taking out an item from the buffer.

  C) Producer and consumer may access the buffer at the same time.

  D) All of the above.

Multiple-Choice Question 2

- In the solution provided for readers-writers problem, if a writer is in the critical section, and multiple readers and writers are waiting,

  A) all waiting readers will be allowed to enter the critical section when the writer in the critical section exits.

  B) all waiting writers will be allowed to enter the critical section when the writer in the critical section exits.

  C) exactly one of the waiting writers will be allowed to enter the critical section when the writer in the critical section exits.

  D) either all waiting readers or exactly one writer will be allowed to enter the critical section.

Multiple-Choice Question 3

- POSIX named semaphores

  A) can easily be used by multiple unrelated processes.

  B) can be initialized during creation time.

  C) uses sem_wait( ) and sem_post( ) to acquire and release a semaphore respectively.

  D) All of the above.

Essay Questions

- Explain the difference between the first readers–writers problem and the second readers–-writers problem.

- Describe a scenario when using a reader–writer lock is more appropriate than another synchronization tool such as a semaphore.

- In the solution for dining philosophers problem using monitors, provide a scenario in which a philosopher may starve to death.