## Chapter 7: Synchronization Examples



**Operating System Concepts – 10th Edition** 

Silberschatz, Galvin and Gagne ©2018

# Chapter 7: Synchronization Examples

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





- 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);
```





- □ 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);
    What is the problem with this algorithm?
П
```





```
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);
}
```



## 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;
     }
```



}



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



**Operating System Concepts – 10th Edition** 

# A Monitor to Allocate Single Resource

```
monitor ResourceAllocator
{
   boolean busy;
   condition x;
   void acquire(int time) {
            if (busy)
               x.wait(time);
            busy = TRUE;
   void release() {
           busy = FALSE;
            x.signal();
initialization code() {
    busy = FALSE;
}
```





### **Synchronization Examples**

- □ Solaris
- Windows
- Linux
- Pthreads





## **Solaris Synchronization**

- Implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing
- Uses adaptive mutexes for efficiency when protecting data from short code segments
  - Starts as a standard semaphore spin-lock
  - If lock held, and by a thread running on another CPU, spins
  - If lock held by non-run-state thread, block and sleep waiting for signal of lock being released
- Uses condition variables
- Uses readers-writers locks when longer sections of code need access to data
- Uses turnstiles to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock
  - Turnstiles are per-lock-holding-thread, not per-object
- Priority-inheritance per-turnstile gives the running thread the highest of the priorities of the threads in its turnstile



## **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()
{
    /* read/write memory */
}
```





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

```
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.



## **End of Chapter 7**



**Operating System Concepts – 10th Edition**