CSE 4/521
Introduction to Operating Systems

Lecture 7 – Process Synchronization II

(Classic Problems of Synchronization, Synchronization Examples)

Summer 2018

Overview

Objective:

1. To examine several classical process-synchronization problems
2. To explore how operating system kernels solve process synchronization problems

- Semaphores (cont.) – Deadlocks and starvation
- Classic Problems of Synchronization
- Synchronization Examples

Recap

- Background
  - Inconsistencies arising from concurrent access to shared data, race condition.
- The Critical-Section Problem
  - Asking permission to enter critical section, 3 guarantees
- Synchronization Hardware
  - `test_and_set()`, `compare_and_swap()`
- Mutex Locks
  - Simple `acquire()` and `release()` of locks, Busy waiting
- Semaphores
  - `Wait()`and `signal()`,implementation with and w/o busy waiting

Questions

1. What are the 3 properties to be guaranteed by any synchronization primitives? (Easy)

2. Any other synchronization primitives apart from what discussed in previous class? (Medium)

3. Does semaphores totally eliminate busy-waiting? If not, then why is it considered better than spinlocks? (Hard)

Overview

- Semaphores (Cont.) – Deadlocks and starvation
- Classic Problems of Synchronization
- Synchronization Examples

Overview

- Semaphores (Cont.) – Deadlocks and starvation
- Classic Problems of Synchronization
- Synchronization Examples

Semaphores – Deadlocks 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

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

P_1

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

Semaphores – 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.

Overview

- Semaphores (Cont.) – Deadlocks and starvation
- Classic Problems of Synchronization
- Synchronization Examples

Classic Problems of Synchronization

- Classical problems used to test newly-proposed synchronization schemes. The 3 problems are:
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem

Classic Problems of Synchronization : 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

Classic Problems of Synchronization : Bounded-Buffer Problem

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);
```

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);
```

Classic Problems of Synchronization : 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.
- Shared Data
  - Semaphore `rw_mutex` initialized to 1
  - Semaphore `mutex` initialized to 1
  - Integer `read_count` initialized to 0

Classic Problems of Synchronization : Readers-Writers Problem

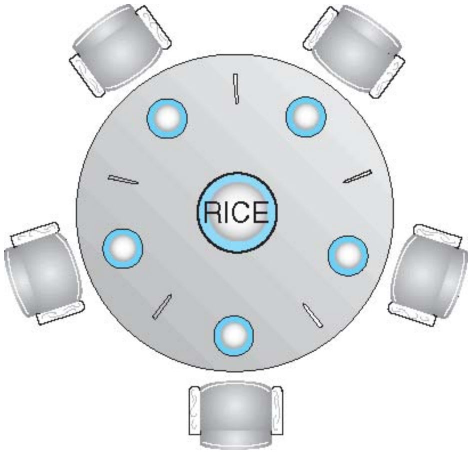
The structure of a reader process

```
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);
```

The structure of a writer process

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

Classic Problems of Synchronization : Dining-Philosophers Problem



- Philosophers do two things - thinking and eating
- When hungry, 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

Classic Problems of Synchronization: Dining-Philosophers Problem

- 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);
```

Any problems in terms of deadlocks with this algorithm?
How would you solve them?

Classic Problems of Synchronization: Dining-Philosophers Problem

- 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.

Overview

- Semaphores (Cont.) – Deadlocks and starvation
- Classic Problems of Synchronization
- Synchronization Examples

Synchronization Examples

- Solaris
- Windows
- Linux
- Pthreads

Synchronization Examples : Solaris

- 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
- 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 priority.

Synchronization Examples : Windows

- 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 as mutexes, semaphores, events, and timers
  - Dispatcher objects either signaled-state (object available) or non-signaled state (thread will block)
- Timers notify one or more thread when time expired

Synchronization Examples : Linux

- 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

Single processor	Multiple Processor
Disable kernel preemption	Acquire spin lock
Enable kernel preemption	Release spin lock

- On single-cpu system, spinlocks replaced by enabling and disabling kernel preemption

Synchronization Examples : Pthreads

- Pthreads API is OS-independent
- It provides:
  - mutex locks
  - Semaphores
  - Etc.
- Non-portable extensions include:
  - read-write locks
  - spinlocks

Pthread Mutex Initialization:

```
#include <pthread.h>
Pthread_mutex_t mutex;
/* create the mutex lock*/
Pthread_mutex_init(&mutex, NULL);
```

Pthread Mutex lock and unlock:

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

/* critical secton */

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

Credits for slides

Silberschatz, Galvin and Gagne