

Soongsil Univ.
<http://design.ssu.ac.kr>


Digital
System
Design Lab.

Ch. 07 Synchronization Examples

Chanho Lee
Soongsil University

Copyright 2023. (차세대반도체 혁신공유대학 사업단) all rights reserved

1

Objectives

- Synchronization problems
 - ▶ bounded-buffer, readers-writers, and dining-philosophers
- Specific synchronization tools used by Linux and Windows
 - ▶ POSIX
- Design and develop solutions to process synchronization problems using POSIX APIs.

2


Soongsil Univ.
<http://design.ssu.ac.kr>


Digital
System
Design Lab.

9주차

Copyright 2023. (차세대반도체 혁신공유대학 사업단) all rights reserved

3

7.1 Classic Problems of Synchronization

● The Bounded-Buffer Problem

```
int n; //pool of n buffers, holding one item
semaphore mutex = 1; // mutex lock
semaphore empty = n; // # of empty buffers
semaphore full = 0; // # of full buffers
```

```
while (true) {
    . . .
    /* produce an item in next_produced */
    . . .
    wait(empty);
    wait(mutex);
    . . .
    /* add next_produced to the buffer */
    . . .
    signal(mutex);
    signal(full);
}
```

```
while (true) {
    wait(full);
    wait(mutex);
    . . .
    /* remove an item from buffer to next_consumed */
    . . .
    signal(mutex);
    signal(empty);
    . . .
    /* consume the item in next_consumed */
    . . .
}
```

4

7.1 Classic Problems of Synchronization

● The Readers–Writers Problem

- ▶ Shared database among several concurrent processes
- ▶ Simultaneous access to database by a writer and some other process
 - no reader waiting unless a writer obtained permission to use the shared object → writer starvation
 - writer performs its write as soon as possible: no reading while writing → reader starvation

```
// updating database. used by first entering or last exiting process
semaphore rw_mutex = 1;
// mutual exclusion for updating read_count
semaphore mutex = 1;
int read_count = 0; // counting reading process
```

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

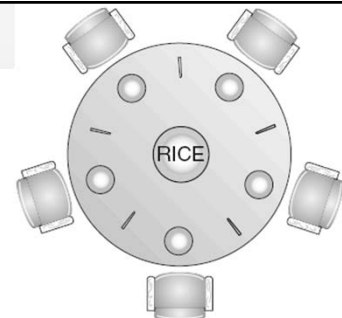
- ▶ Reader-writer lock in read and write mode

```
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);
}
```

7.1 Classic Problems of Synchronization

● The Dining-Philosophers Problem

- ▶ Thinking: no interaction
- ▶ eating: using two chopsticks, picking one at a time
- ▶ example of a large class of concurrency-control problems
- ▶ Semaphore Solution
 - semaphore chopstick[5]; (initialized to 1)
 - Deadlock: possible solutions
 - at most four philosophers to be sitting simultaneously
 - pick up chopsticks only if both chopsticks are available (pick them up in a critical section)
 - asymmetric solution— an odd-numbered left first even-numbered right first
 - deadlock-free ≠ starvation-free



```
while (true) {
    wait(chopstick[i]);
    wait(chopstick[(i+1) % 5]);
    . . .
    /* eat for a while */
    . . .
    signal(chopstick[i]);
    signal(chopstick[(i+1) % 5]);
    . . .
    /* think for a while */
    . . .
}
```

7.1 Classic Problems of Synchronization

► Monitor Solution

- Deadlock-free
- Possible starvation

```
DiningPhilosophers.pickup(i);
...
eat
...
DiningPhilosophers.putdown(i);
```

```
monitor DiningPhilosophers
{
    enum {THINKING, HUNGRY, EATING} state[5];
    condition self[5];
    void pickup(int i) { //before eating
        state[i] = HUNGRY;
        test(i);
        if (state[i] != EATING)
            self[i].wait();
    } //ready to eat
    void putdown(int i) { //after eating
        state[i] = THINKING;
        test((i + 4) % 5);
        test((i + 1) % 5);
    }
    void test(int i) { //when two neighbors are not eating
        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;
    }
}
```

7.2 Synchronization within the Kernel

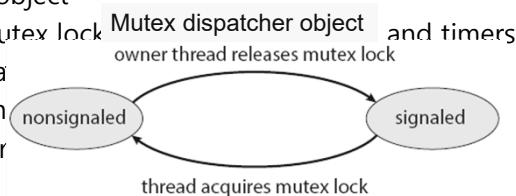
● Synchronization in Windows

► Kernel accesses a global resource

- single-processor system: masks interrupts for all interrupt handlers
- multiprocessor system: protects access to global resources using spinlocks
 - for efficiency, a thread will never be preempted while holding a spinlock

► Outside the kernel: dispatcher object

- threads synchronize using mutex lock and timers
 - System protects shared data
 - Semaphore: behaves as counter
 - Events: notify a waiting thread
 - Timer: notify expired time
 - object in a signaled state is available: acquiring the object (any thread waiting → ready)
 - object in a nonsignaled state is not available: blocked (thread: ready → waiting)
- critical-section object: a user-mode mutex
- acquired and released without kernel intervention
 - Spinlock first → kernel mutex when waiting long



7.2 Synchronization within the Kernel

● Synchronization in Linux

- ▶ Prior to Version 2.6: nonpreemptive kernel
- ▶ Atomic integer: atomic_t → atomic math operation
 - Do not require the overhead of locking mechanisms
 - Useful for a simple synchronization

```
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

- ▶ Mutex lock
 - mutex_lock() and mutex_unlock() function
 - Lock unavailable: sleep state

7.2 Synchronization within the Kernel

▶ Spinlock

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

- preempt_disable() and preempt_enable(): for disabling and enabling kernel preemption

● spinlocks and mutex locks are nonrecursive:

- ▶ no second lock without releasing the first

7.3 POSIX Synchronization

● POSIX API for programmers at the user level

- ▶ not part of any particular OS kernel

● POSIX Mutex Locks

- ▶ `pthread_mutex_t` data type for mutex locks
- ▶ `pthread_mutex_init()` function: mutex is created
 - Parameter: pointer, NULL (default attribute)
- ▶ `pthread_mutex_lock()` / `pthread_mutex_unlock()` functions: mutex is acquired / released
 - calling thread is blocked if the mutex lock is unavailable
- ▶ return a value of 0 with correct operation; nonzero error code

```
#include <pthread.h>
pthread_mutex_t mutex;
/* create and initialize the mutex lock */
pthread_mutex_init(&mutex, NULL);
/* acquire the mutex lock */
pthread_mutex_lock(&mutex);
/* critical section */
/* release the mutex lock */
pthread_mutex_unlock(&mutex);
```

7.3 POSIX Synchronization

● POSIX Semaphores

- ▶ belong to the POSIX SEM extension
- ▶ types of semaphores: named and unnamed (≥ 2.6)
 - Quite similar
 - Differ in how they are created and shared between processes
- ▶ POSIX Named Semaphores (Linux, macOS)
 - multiple unrelated processes can easily use a common semaphore using the semaphore's name

```
#include <semaphore.h>
sem_t *sem;
/* Create the semaphore and initialize it to 1 */
sem = sem_open("SEM", 0_CREAT, 0666, 1); //create and open a POSIX named semaphore, "SEM"
// calls to sem_open() (with the same parameters) return a descriptor to the existing
// semaphore
/* acquire the semaphore */
sem_wait(sem); //wait()
/* critical section */
/* release the semaphore */
sem_post(sem); //signal()
```

7.3 POSIX Synchronization

► POSIX Unnamed Semaphores

1. A pointer to the semaphore
2. A flag indicating the level of sharing
3. The semaphore's initial value

```
#include <semaphore.h>
sem_t sem;
/* Create the semaphore and initialize it to 1 */
sem_init(&sem, 0, 1); //0: shared only by threads belonging to the creator
// non-zero: shared between separate processes in a region of shared memory
/* acquire the semaphore */
sem_wait(&sem);
/* critical section */
/* release the semaphore */
sem_post(&sem);
```

7.3 POSIX Synchronization

● POSIX Condition Variables

► Pthread in C: associating a condition variable with a mutex lock

```
pthread_mutex_t mutex;
pthread_cond_t cond_var;
pthread_mutex_init(&mutex, NULL);
pthread_cond_init(&cond_var, NULL);
pthread_mutex_lock(&mutex); //lock the mutex lock to prevent a possible race condition
while (a != b) // wait for the condition a == b
    pthread_cond_wait(&cond_var, &mutex);
// releases the mutex lock → another thread access the shared data and possibly update its value
pthread_mutex_unlock(&mutex);
```

```
// a thread that modifies the shared data
pthread_mutex_lock(&mutex);
a = b;
pthread_cond_signal(&cond_var); //signaling one thread waiting on the condition variable
pthread_mutex_unlock(&mutex);
```

Summary

- Classic problems of process synchronization
 - ▶ bounded-buffer, readers–writers, and dining-philosophers problems.
- Windows synchronization tools
 - ▶ dispatcher objects as well as events
- Linux
 - ▶ atomic variables, spinlocks, and mutex locks.
- POSIX API
 - ▶ mutex locks, semaphores, and condition variables.
 - ▶ named and unnamed semaphores
- Alternative approaches
 - ▶ transactional memory, OpenMP, and functional languages.

Exercises, problems and projects

- Exercises
 - ▶ 7.4, 7.6
- Programming Problems
 - ▶ 7.17

7.17 Exercise 4.24 asked you to design a multithreaded program that estimated π using the Monte Carlo technique. In that exercise, you were asked to create a single thread that generated random points, storing the result in a global variable. Once that thread exited, the parent thread performed the calculation that estimated the value of π . Modify that program so that you create several threads, each of which generates random points and determines if the points fall within the circle. Each thread will have to update the global count of all points that fall within the circle. Protect against race conditions on updates to the shared global variable by using mutex locks.