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
- Semaphore **`full`** initialized to the value 0
- Semaphore **`empty`** initialized to the value n

Bounded Buffer Problem (Cont.)

- The structure of the producer process

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

Semaphore **mutex** = 1
Semaphore **full** = 0
Semaphore **empty** = n

Bounded Buffer Problem (Cont.)

- The structure of the consumer process

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

Semaphore **mutex** = 1
Semaphore **full** = 0
Semaphore **empty** = n

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
- 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

```
while (true) {
      wait(rw_mutex);

         ...
      /* writing is performed */

         ...

   signal(rw_mutex);

}
```

Semaphore **rw_mutex** = 1

Semaphore **mutex** = 1

Integer **read_count** = 0

Readers-Writers Problem (Cont.)

- The structure of a reader process

Semaphore **rw_mutex** = 1

Semaphore **mutex** = 1

Integer **read_count** = 0

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

        signal(rw_mutex);
}
```

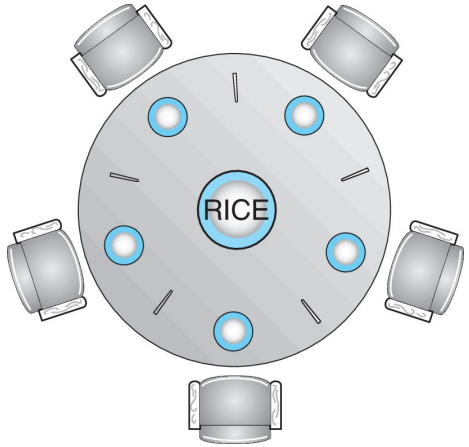
```
while (true){
        wait(mutex);
        read_count++;
        if (read_count == 1)
        wait(rw_mutex);
        signal(mutex);
          ...    /* reading is performed */
          ...
        wait(mutex);
                read count--;
                if (read_count == 0)
                signal(rw_mutex);
        signal(mutex);

}
```

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

- Semaphore Solution
- 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?

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);
   }
}
```

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;
     }
}
```

Solution to Dining Philosophers (Cont.)

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

```
DiningPhilosophers.pickup(i);

    /** EAT **/

DiningPhilosophers.putdown(i);
```

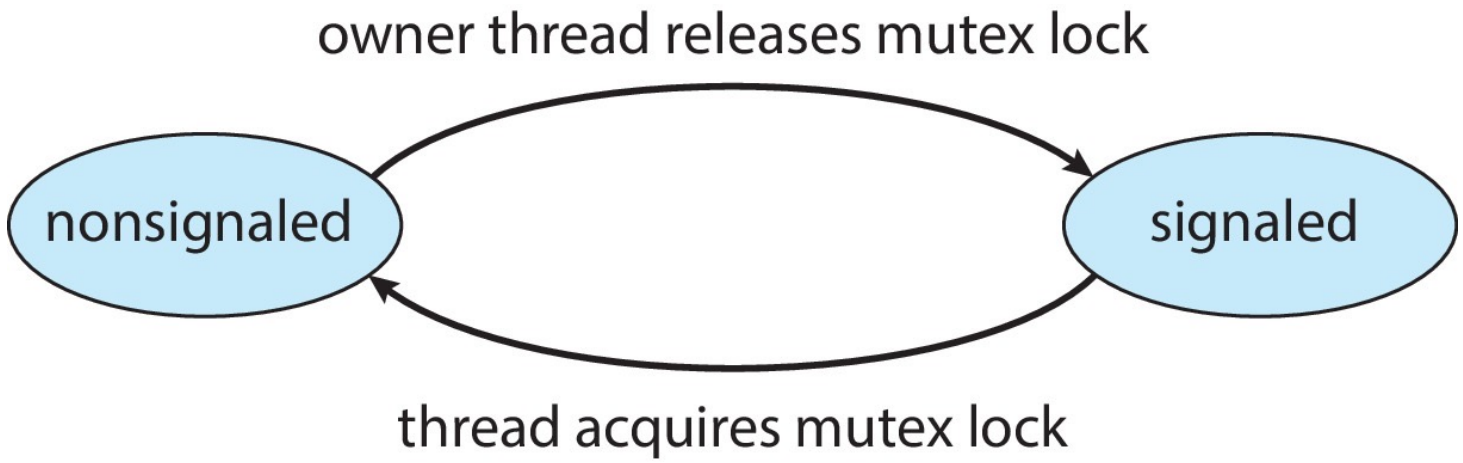
- No deadlock, but starvation is possible

Kernel Synchronization - Windows

- Uses interrupt masks to protect access to global resources on uniprocessor systems
- Uses *spinlocks* on multiprocessor systems
  - Spin locking-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)

Kernel Synchronization - Windows

- Mutex dispatcher object



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

- Atomic variables
  - `atomic_t` is the type for atomic integer
- Consider the variables

```
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`

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

- 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/C++ and these languages do 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);
```

Condition variable in POXIS thread library

- CV: condition variable
- pthread_cond_t cond; //declare an instance of CV.
- pthread_mutex_t mutex; //declare an instance of mutex
- pthread_cond_init(&cond, NULL); //initialize a CV
- pthread_cond_signal(&cond); //signal a CV: unblock a thread that is blocked on this CV. If no thread is blocked on the CV, the signal is ignore.
- pthread_cond_broadcast(&cond); //unblock all threads that are blocked on this CV.
- pthread_cond_wait(&cond,&mutex): Before calling this function, the mutex must be locked by the calling thread. When calling this function, it unlocks the mutex and blocks on the CV.
  - When the CV is signaled and the calling thread unblocks, pthread_cond_wait reacquires a lock on mutex.

Condition variable … (cont)

```
void initialize_flag (){
   pthread_mutex_init (&thread_flag_mutex, NULL);
   pthread_cond_init (&thread_flag_cv, NULL);
   thread_flag = 0;
}
```

```
void set_thread_flag (int flag_value) {
     pthread_mutex_lock (&thread_flag_mutex);
          thread_flag = flag_value;
          pthread_cond_signal (&thread_flag_cv);
     pthread_mutex_unlock (&thread_flag_mutex);
}
```

```
void* thread_function (void* thread_arg){
      while (1) {
          pthread_mutex_lock (&thread_flag_mutex);
          while (!thread_flag)
               pthread_cond_wait (&thread_flag_cv, &thread_flag_mutex);
         pthread_mutex_unlock (&thread_flag_mutex);
         do_work ();
}
      return NULL;

}
```

Summary

- Classic problems of process synchronization include the bounded-buffer, readers–writers, and dining-philosophers problems. Solutions to these problems can be developed using the tools presented in Chapter 6, including mutex locks, semaphores, monitors, and condition variables.
- Windows uses dispatcher objects as well as events to implement process synchronization tools.
- Linux uses a variety of approaches to protect against race conditions, including atomic variables, spinlocks, and mutex locks.
- The POSIX API provides mutex locks, semaphores, and condition variables. POSIX provides two forms of semaphores: named and unnamed. Several unrelated processes can easily access the same named semaphore by sim- ply referring to its name. Unnamed semaphores cannot be shared as easily, and require
placing the semaphore in a region of shared memory.

Summary (Cont.)

- Java has a rich library and API for synchronization. Available tools include monitors (which are provided at the language level) as well as reentrant locks, semaphores, and condition variables (which are supported by the API).
- Alternative approaches to solving the critical-section problem include transactional memory, OpenMP, and functional languages. Functional languages are particularly intriguing, as they offer a different programming paradigm from procedural languages. Unlike procedural languages, functional languages do not maintain state and therefore are generally immune from race conditions and critical sections.

End of Chapter 7