Chapter 6.3

Process and Thread Synchronization

Classical Synchronization Patterns

Print Version of Lectures Notes of *Operating Systems*

Technical University of Cluj-Napoca (UTCN)
Computer Science Department

Adrian Coleșa

May 6, 2020

6.3.1

Purpose and Contents

The purpose of this chapter

- Present some classical synchronization patterns (problems): readers / writers, barrier, philosophers etc.

6.3.2

Bibliography

- A. Tanenbaum, *Modern Operating Systems*, 2nd Edition, 2001, Chapter 2, Processes, p. 100 – 132
- A. Downey, *The Little Book of Semaphores*, 2nd Edition, 2016, p. 1 – 115

6.3.3

Contents

6.3.4

1 Implementing Synchronization Mechanisms With Other Synchronization Mechanisms

Semaphores Using Locks And Condition Variables

```
// Internal Variables
int value = initValue;
Lock mutex;
Condition permission;


// decrement the semaphore by 1
P()
{
    mutex.lock();

    while (value == 0)
        permission.wait(mutex);

    value--;

    mutex.unlock();
}
```

```
// increment the semaphore by 1
V()
{
    mutex.lock();

    value++;

    permission.signal();

    mutex.unlock();
}

// decrement the semaphore by N
P(int n)
{
    mutex.lock();

    while (value < n)
        permission.wait(mutex);

    value -= n;

    mutex.unlock();
}

// increment the semaphore by N
V(int n)
{
    mutex.lock();

    value += n;

    permission.broadcast();

    mutex.unlock();
}
```

6.3.5

Locks Using Semaphores

```
// internal variables
Semaphore s(1);
int lockHolder = -1;

// Acquire the lock
lock()
{
    s.P();
    lockHolder = gettid();

}

// Release the lock
unlock()
{
    if (lockHolder = gettid()) {
        lockHolder = -1;
        s.V();
    }
}
```

6.3.6

Condition Variables Using Semaphores

- implementation

```
// internal variables
List<Semaphore> semList;

// wait until signaled
// releasing the lock
wait(Lock *mutex)
{
    // creates a new 0 sem
    Semaphore s(0);

    // add teh sem to waiting list
    semList.add(s);

    // release the lock
    mutex->unlock();

    // go to sleep
    s.P();

    // re-acquire the lock
    mutex->lock();
}

// wake up a waiting thread
// supposes to be called when
//   the lock is held!!!
signal()
{
    Semaphore s;

    if (! semList.isEmpty()) {
        s = semList.removeFirst();
        s.V();
    }
}
```

.

- usage

```
Lock mutex;
Condition c;
int ok = 0;


// Thread T1
mutex.lock();
while (!ok)
    c.wait(&mutex);
mutex.unlock();


// Thread T2
mutex.lock();
ok = 1;
c.signal();
mutex.unlock();
```

6.3.7

Practice (1)

You are given the three functions below, executed by three different threads. You are required to use

1. semaphores
2. locks and condition variables

to make sure that the string displayed on the screen is always “**1 + 2 + 3 + 4 = 10**”, no matter how the threads are scheduled.

```
thread_function_1()
{
  printf("1 + ");
  printf("3 + ");
}
```

```
thread_function_2()
{
  printf("2 + ");
  printf("4 =");
}
```

```
thread_function_3()
{
  printf("10\n");

}
```

6.3.8

2 Classical Synchronization Patterns

Strict Alternation Using Semaphores

```
// Semaphores initialization
Semaphores s[N];

// consider initial turn is for thread with ID = 0
s[0] = 1;

// it is not the turn of other threads
for (i=1; i<N; i++)
    s[i] = 0;


// Threads' function
thread_function(int th_id)
{
    s[th_id].P();

    printf("It is my turn now! Nobody can take it from me.\n");

    // ... until I give it voluntarily to next
    s[(th_id+1)%N].V();
}
```

6.3.9

Strict Alternation Using Locks and Condition Variables

```
// Global variables
Lock mutex;
Condition c[N];
int turn = 0;


// Threads' function
thread_function(int th_id)
{
    // ENTRY in critical region
    mutex.lock();

    while (turn != th_id)
        c[thId].wait(mutex);

    mutex.unlock();

    // INSIDE the critical region
    printf("It is my turn now! Nobody can take it from me.\n");

    // EXIT from critical region
    mutex.lock();

    // ... until I give it voluntarily to next
    turn = (turn + 1) % N;
    c[turn].signal();

    mutex.unlock();
}
```

6.3.10

“Unfair” Rendezvous

```
// Global variables
Semaphore is_friend_1 = 0;
Semaphore is_friend_2 = 0;;


// function of friend_1 threads
friend_1()
{
    // announce its own presence
    is_friend_1.V();

    // check for its partner's presence
    is_friend_2.P();
}


// function of friend_2 threads
friend_2()
{
    // announce its own presence
    is_friend_2.V();

    // check for its partner's presence
    is_friend_1.P();
}
```

6.3.11

“Fair” Rendezvous

```
// Global variables
Semaphore is_friend_1 = 0;
Semaphore access_to_meeting_point_1 = 1;

Semaphore is_friend_2 = 0;
Semaphore access_to_meeting_point_2 = 1;


// function of friend_1 threads
friend_1()
{
    // get exclusive access to the meeting
    access_to_meeting_point_1.P();

    // announce its own presence
    is_friend_1.V();

    // check for its partner's presence
    is_friend_2.P();

    // let another of the same type enter
    access_to_meeting_point_1.V();
}


// function of friend_2 threads
friend_2()
{
    // get exclusive access to the meeting
    access_to_meeting_point_2.P();

    // announce its own presence
    is_friend_2.V();

    // check for its partner's presence
    is_friend_1.P();

    // let another of the same type enter
    access_to_meeting_point_2.V();
}
```

6.3.12

“Deadlocked” Rendezvous

```
// Global variables
Semaphore is_friend_1 = 0;
Semaphore is_friend_2 = 0;


// function of friend_1 threads
friend_1()
{
    // check for its partner's presence
    is_friend_2.P();

    // announce its own presence
    is_friend_1.V();
}


// function of friend_2 threads
friend_2()
{
    // check for its partner's presence
    is_friend_1.P();

    // announce its own presence
    is_friend_2.V();
}
```

6.3.13

One-Time Usage Barrier (Meeting of N Processes)

```
// Global variables
Semaphore mutex = 1;
Semaphore barrier = 0;
int count = 0;  // count how many threads arrived at meeting point
```

```
// function called by threads to wait for meeting
meeting_point()
{
    // get exclusive access
    // when updating and checking count
    mutex.P();

    count++;

    if (count == N)      // if the last one
        barrier.V();     // open the barrier

    // let the others enter the checking point
    mutex.V();

    barrier.P();         // here is the barrier
    barrier.V();         // let it open for the next
}
```

6.3.14

Reusable Barrier (By The Same Set of N Processes)

```
// Global variables
Semaphore mutex = 1;
Semaphore barrier1 = 0; // for stoping thread at entrence of meeting point
Semaphore barrier2 = 1; // for stopping threads at exit from meeting point

int count = 0;          // count how many threads
                        // arrived and inside the meeting point


// function called by threads to wait for meeting
meeting_point()
{
  // get exclusive access
  // at entrace checkpoint
  mutex.P();

  count++;

  if (count == N) { // if the last one entering
      barrier2.P(); // close the exiting barrier
      barrier1.V(); // open the entering barrier
  }

  // let the others enter the checking point
  mutex.V();

  barrier1.P();      // the entering barrier
  barrier1.V();      // let it open for the next

  .....


  ......

  // get exclusive access
  // at exit checkpoint
  mutex.P();
  count--;

  if (count == 0) { // if the last one exiting
      barrier1.P(); // close the entering barrier
      barrier2.V(); // open the exiting barrier
  }

  // let the others enter the checking point
  mutex.V();

  barrier2.P();      // the exiting barrier
  barrier2.V();      // let it open for the next
}
```

6.3.15

Readers/Writers Problem. Description

- models accesses to a shared database (DB)
- there are two types of threads
  - *readers*: just read, do not modify the shared resources
  - *writers*: modify the shared resource
- synchronization (access) rules are
  - multiple readers allowed simultaneously, but not in the same time with a writer
  - when a writer accesses the shared resource, no other process can accesses it

6.3.16

Readers/Writers Problem. Implementation With Locks and Condition Variables

```
// Global variables
int WR = 0; // waiting readers
int AR = 0; // active readers on the DB; AR >= 0
int WW = 0; // waiting writers
int AW = 0; // active writers on the DB, 0 <= WR <=1

Lock mutex;
Condition okToRead, okToWrite;
```

```
// Reader's function
Reader()
{
    mutex.Acquire();
    while (AW + WW > 0) {
        WR++;
        okToRead.WAIT(&mutex);
        WR--;
    }
    AR++;
    mutex.Release();

    //  -----------> read DB

    mutex.Acquire();
    AR--;
    if (AR == 0 && WW > 0)
        okToWrite.SIGNAL();
    mutex.Release();




}


// Writer's function
// writers get preference over readers
Writer()
{
    mutex.Acquire();
    while (AR + AW > 0) {
        WW++;
        okToWrite.WAIT(&mutex);
        WW--;
    }
    AW++;
    mutex.Release();

    //  -----------> write DB

    mutex.Acquire();
    AW--;
    if (WW > 0) // favor writers
      okToWrite.SIGNAL();
    else
      if (WR > 0)
        okToRead.BROADCAST();
    mutex.Release();
}
```

6.3.17

Readers/Writers Problem. Implementation With Semaphores (1)

```
Semaphore permissions = MAX_READERS;


// Reader's function
// readers get preference over writers
Reader()
{
    permissions.P(1);

    //  -----------> read DB

    permissions.V(1);
}


// Writer's function
Writer()
{
    permissions.P(MAX_READERS);

    //  -----------> write DB

    permissions.V(MAX_READERS);
}
```

6.3.18

Readers/Writers Problem. Implementation With Semaphores (2)

```
Semaphore permissions = 1;
Semaphore mutex = 1;
int readers = 0; // numer of readers in critical region


// Readres' function
// readers get preference over writers
Reader()
{
    mutex.P();
    readers++;

    if (readers == 1)
      permissions.P();
    mutex.V();

    //  -----------> read DB

    mutex.P();
    readers--;

    if (readers == 0)
      permissions.V();
    mutex.V();
}


// Writers' function
Writer()
{

    permissions.P();
```

```
    //  -----------> write DB

    permissions.V();

}
```

6.3.19

Readers/Writers Problem. A Particular Case: Single Favored Reader

```
Semaphore permissions(1);
Semaphore writerBarrier(1);


// Single reader's function
Reader()
{


    permissions.P();

    //  -----------> read DB

    permissions.V();


    }


// Writers' function
Writer()
{
    writersBarrier();

    permissions.P();

    //  -----------> write DB

    permissions.V();

    writersBarrier.V();
}
```

6.3.20

Dining Philosophers. Description

- proposed by Dijkstra in 1965
- five philosophers are seated around a circular table
- each philosopher has a plate with spaghetti
- between each pair of plates is one fork
- a philosopher needs two forks to eat
- a philosopher eats and thinks
- only one philosopher can hold a fork at a time
- **deadlock** and **starvation** should be avoided

6.3.21

Dining Philosophers. Implementation with Deadlock

```
// Global variables
const int N = 5;

// Utility functions
int right(id) { return id; }
int left(id) { return (id+1) % N; }

// Synchronization mechanisms
Semaphores forks[N];
for (i=0; i<N; i++)
    forks[i] = 1;


void philosopher(int id)
{
    while (TRUE) {
        think();
        take_forks(id);
        eat();
        put_forks(id);
    }
}


void take_forks(int id)
{
    forks[right(id)].P();
    forks[left(id)].P();
}

void put_forks(int id)
{
    forks[right(id)].V();
    forks[left(id)].V();
}
```

6.3.22

Dining Philosophers. Solution 1

```
// Global variables
const int N = 5;

// Utility functions
int right(id) { return id; }
int left(id) { return (id+1) % N; }

// Synchronization mechanisms
Semaphore limit = 4;
Semaphores forks[N];
for (i=0; i<N; i++)
    forks[i] = 1;


void philosopher(int id)
{
    while (TRUE) {
        think();
        take_forks(id);
        eat();
        put_forks(id);
    }
}


void take_forks(int id)
{
    limit.P();
    forks[right(id)].P();
    forks[left(id)].P();
}

void put_forks(int id)
{
    forks[right(id)].V();
    forks[left(id)].V();
    limit.V();
}
```

6.3.23

Dining Philosophers. Solution 2

```
// Global variables
const int N = 5;
typedef enum {THINKIG, HUNGRY, EATING} STATE;
STATE state[N];

// Utility functions
int right(id) { return id; }
int left(id) { return (id+1) % N; }

Semaphore mutex = 1;
Semaphores permission[N];
for (i=0; i<N; i++)
    permission[i] = 0;


void philosopher(int id)
{
    while (TRUE) {
        think();
        take_forks(id);
        eat();
        put_forks(id);
    }
}

void take_forks(int id)
{
    mutex.P();
    state[id] = HUNGRY;
    test(id);
    mutex.V();
    permission[id].P();
}


void put_forks(int id)
{
    mutex.P();
    state[id] = THINKING;
    test(left(id));
    test(right(id));
    mutex.V();
}

void test(int id)
{
    if (state[id] == HUNGRY &&
        state[left(id)] != EATING &&
        state[right(id)] != EATING) {
        state[id] = EATING;
        permission[id].V();
}
```

6.3.24

3 Conclusions

What we talked about

- implementing some synchronization mechanisms with other synchronization mechanisms
  - semaphores with locks and condition variables
  - locks with semaphores

- condition variables with semaphores

- some common synchronization patterns
  - rendezvous
  - barrier
  - readers / writers
  - dining philosophers

6.3.25

Lessons Learned

1. synchronization problems are complex
2. readers / writers synchronization pattern is a sort of "relaxed lock"

6.3.26