Chapter 6.2

Process and Thread Synchronization

Semaphores and Condition Variables

Print Version of Lectures Notes of *Operating Systems*

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

Adrian Coleșa

April 29, 2020

6.2.1

Purpose and Contents

The purpose of this chapter

- Present different synchronization mechanisms: semaphores and condition variables
- Present some classical synchronization patterns (problems): producer / consumer

6.2.2

Bibliography

- A. Tanenbaum, *Modern Operating Systems*, 2nd Edition, 2001, Chapter 2, Processes, p. 100 – 132
- A. Downey, *The Little Book of Semaphores*, 2nd Edition, 2008, p. 1 – 106

6.2.3

Contents

6.2.4

1 Semaphores

Characteristics

- **generalization of locks**
  - more threads could be given permission
  - to enter simultaneously in the critical regions protected by the semaphore
- consists of
  - **a value**
    - number of available permissions to pass the checkpoint
    - only positive values are allowed
  - **two primitives**

    * **P** (**sem_down**): blocks the current process if the value is 0, otherwise decrements the value
    * **V** (**sema_up**): increments the value and wakes up a sleeping process

- main difference from locks
  - any thread is allowed to increase a semaphore value (no of permissions), not only those that previously got one
  - ⇒ could be used both as a (generalized) lock and an event counter

6.2.5

Possible Implementation (uni-processor systems)

```
P() // sema_down()
{
    disable_interrupts();
    while (value == 0) {
        insert(crt_thread, w_queue);
        block_current_thread();;
    }
    value = value - 1;
    enable_interrupts();
}


V()  // sema_up()
{
    disable_interrupts();
    value = value + 1;
    if (! empty(waiting_queue)) {
        th = remove_first(w_queue);
        insert(ready_queue, th);
    }
    enable_interrupts();
}
```

- **interrupts are disabled**, to provide **code atomicity**
  - after critical code is executed, interrupts are enabled back
- "**sleep / wake-up technique**" is used instead of the "*busy waiting*"
  - "**sending to sleep**" (**blocking**) a thread
    * append it to a waiting queue and
    * take the CPU from it
  - "**waking up**" (**unblocking**) a thread
    * remove it from waiting queue
    * append it to the ready queue
    * eventually being given the CPU again

6.2.6

Usage Example

```
#include <pthread.h>

// restrict the number of "persons" in a room
// i.e number of threas in their critical region
sem_t sem;

void three_in_a_room();

main()
{
    // initialize the semaphore!
    // do that before having threads working with it!
    sem_init(&sem, 1, 3);

    // create the competing threads
    for (int i=0; i < 100; i++)
        pthread_create(&t[i], NULL, three_in_a_room, NULL);

    // wait for all threads' termination
    for (int i=0; i < 100; i++)
        pthread_join(t[i], NULL);

    // remove the semaphore
    sem_destroy(&sem);
}


void three_in_a_room()
{
    // ask for a permission, i.e try decrementing the semaphore
    // the thread will be blocked, if no permission is available,
    // i.e. semaphore's value is 0
    sem_wait(&sem);

        // stay in room (critical region) for a while

    // release the obtained permission
    // i.e. increment the semaphore's value
    // this operation NEVER blocks the thread
    sem_post(&sem);
}
```

6.2.7

Practice (1)

- Using semaphores, write the (pseudo)code for the functions in the code below such that to not allow more than 22 players enter simultaneously on the football field. A player is represented by a thread executing the *football_player()* function.

```
void football_player()
{
    enter_football_field();
    play_football();
    exit_the_footbal_field();
}
```

6.2.8

Practice (2)

- Using semaphores, write the (pseudo)code that limits to 5 the number of threads executing simultaneously the critical section in the function below.

```
#define FREE 1
#define BUSY 0
#define MAX_TH 5

int available_unit[MAX_TH] = {FREE, FREE, FREE, FREE, FREE};
int unit[MAX_TH] = {0, 0, 0, 0, 0};

void limited_area()
{
    int pos = -1;

    for (pos = 0; pos < MAX_TH; pos++)
        if (available_unit[pos] == FREE) {
            available_unit[pos] = BUSY;
            break;
        }

    unit[pos]++;

    available_unit[pos] = FREE;
}
```

6.2.9

2 The “Producer/Consumer” Problem

Problem Description

- two types of processes
  - *producers*: produce messages
  - *consumers*: consume messages
- use a bounded buffer
- waiting conditions
  - for producer: “buffer is full”
  - for consumer: “buffer is empty”
- wakeup conditions
  - for producer: “there is space in buffer”
  - for consumer: “there are messages in buffer”

6.2.10

Basic Solution With Race Conditions

```
// Global variables
const N = 100;        // number of slots in buffer
int count = 0;        // number of messages in the buffer


void producer(int item)
{
    // wait for available space
    while (count == N) {} // full buffer

    insert_item(item);

    // wakeup a consumer
    count = count + 1;
}
```

```
int consumer(int *item)
{
    // wait for available message
    while (count == 0) {}  // empty buffer

    *item = remove_item();

    // wakeup a producer
    count = count - 1;
}
```

- **race conditions**
  - uncontrolled concurrent access to variable `count`
  - same message could be consumed more times
  - messages could be overwritten
- **busy waiting**
  - suggests us that **synchronization is needed**
  - **should be avoided** in practice
  - **replace it** with a sleep / wakeup **synchronization mechanism**

6.2.11

Producers/Consumers Problem's Implementation With Semaphores

```
// Global variables and synchronization mechanims
const N = 100;                // number of slots in buffer
Semaphore mutex(1);           // provides mutual exclusion to buffer
Semaphore can_produce_msg(N); // controls producers' access to buffer
Semaphore can_consume_msg(0); // controls consumers' access to buffer


void producer(int item)
{
    // check if buffer is full
    can_produce_msg.P();

    // gets mutual exclusion to buffer
    mutex.P();

    insert_item(item);

    // releases the buffer
    mutex.V();

    // new permission for consumers
    can_consume_msg.V();
}


int consumer(int *item)
{
    // check if buffer is empty
    can_consume_msg.P();

    // gets mutual exclusion to buffer
    mutex.P();

    *item = remove_item();

    // releases the buffer
    mutex.V();

    // new permission for producers
    can_produce_msg.V();
}
```

6.2.12

Remarks On Producers/Consumers Implementation With Semaphores

- there are two ways of using semaphores
  - as a (generalized) lock to provide limited access (mutual exclusion in the example with `mutex` semaphore)
  - as a condition checking and event counter (semaphores `can_consume_msg` and `can_produce_msg`)
- the order of getting the semaphores is important and it can lead to deadlock
  - see the solution with `mutex` semaphore get first

6.2.13

Deadlock In The Producers/Consumers Problem's Implementation With Semaphores

```
// Global variables and synchronization mechanims
const N = 100;                // number of slots in buffer
Semaphore mutex(1);           // provides mutual exclusion to buffer
Semaphore can_produce_msg(N); // controls producers' access to buffer
Semaphore can_consume_msg(0); // controls consumers' access to buffer
```

```
void producer(int item)
{
    // gets mutual exclusion to buffer
    mutex.P();

    // check if buffer is full
    can_produce_msg.P();

    insert_item(item);

    // new permission for consumers
    can_consume_msg.V();

    // releases the buffer
    mutex.V();
}


int consumer(int *item)
{
    // gets mutual exclusion to buffer
    mutex.P();

    // check if buffer is empty
    can_consume_msg.P();

    *item = insert_item();

    // new permission for producers
    can_produce_msg.V();

    // releases the buffer
    mutex.V();
}
```

6.2.14

3 Condition Variables

Condition Variables: Context and Problem

- **used in a mutual exclusion context**
- **a thread** having a lock **must wait for a specific condition** to be fulfilled before going further
  - waited condition reflect the shared resource state
  - lock must be taken before testing / changing the condition
- **should not wait (sleep) keeping the lock**
  - could block the other threads that could change condition to true
- ⇒ waiting thread should **release the lock while waiting** (sleeping)
- ⇒ another thread can take the lock in the meantime and change condition
- that thread **wakes up the sleeping (waiting) thread**

6.2.15

Condition Variables: Possible Deadlock Scenario

```
// Thread waiting
// for a condition
// to be fulfilled
thread_1()
{
  mutex.lock();

  while (cond == FALSE) {
    // e.g. (no_msg == 0)
    // for a consumer
    sleep(1);
  }

  mutex.unlock();
}

// Thread changing
// the waited condition
// to TRUE
thread_2()
{
  mutex.lock();

  cond = TRUE;
  // e.g. no_msg++
  // for a producer
```

```
  mutex.unlock();
}
```

6.2.16

Condition Variables: Functionality

- what is it?
  - **a specialized waiting mechanism**
  - transparently release the lock while blocking the lock holder
  - at wakeup transparently reacquire the lock before resuming the waiting thread
- consists of
  - a waiting queue
  - two (three) primitives
    - **wait**: the calling process is inserted in the waiting queue and suspended
    - **signal**: one sleeping process is removed from the waiting queue and woken up
    - **broadcast**: all sleeping processes are woken up

6.2.17

Condition Variables. Basic Pattern Usage

```
// Thread waiting for
// an event (condition)
// to occur
thread_1()
{
  mutex.lock();

  while (! cond)
    c.wait(mutex);

  mutex.unlock();
}

// Thread generating
// the event
//
thread_2()
{
  mutex.lock();

  cond = TRUE;
  c.signal();

  mutex.unlock();
}
```

6.2.18

Condition Variables: Implementation

```
wait(Lock mutex)
{
  disable_interrupts();

  mutex.unlock();

  insert(crt_th, w_queue);
  block_current_thread();

  mutex.lock();

  enable_interrupts();
}
```

```
signal()
{
  disable_interrupts();


  if (! empty(waiting_queue)) {
    th = remove_first(w_queue);
    insert(ready_queue, th);
  }


  enable_interrupts();
}
```

6.2.19

Condition Variables. General Pattern Usage (Var 1)

```
// Synchronization mechanims: one condition variable
Lock mutex;
Condition c;


// Thread 1's Function
thread_1()
{
    // critical region entrace
    mutex.lock();
    while (! cond_1)
        c.wait(mutex);
    mutex.unlock();

    // thread T1 in its critical region
    ....

    // critical region exit
    mutex.lock();
    cond_2 = TRUE;
    c.broadcast();
    mutex.unlock();
}


// Thread 2's Function
thread_2()
{
    // critical region entrace
    mutex.lock();
    while (! cond_2)
        c.wait(mutex);
    mutex.unlock();

    // thread T2 in its critical region
    ...

    // critical region exit
    mutex.lock();
    cond_1 = TRUE;
    c.broadcast();
    mutex.unlock();
}
```

6.2.20

Condition Variables. General Pattern Usage (Var 2)

```
// Synchronization mechanims: two condition variables
Lock mutex;
Condition c1, c2;


// Thread 1's Function
thread_1()
{
    // critical region entrace
    mutex.lock();
    while (! cond_1)
        c1.wait(mutex);
    mutex.unlock();

    // thread T1 in its critical region
    ....

    // critical region exit
    mutex.lock();
    cond_2 = TRUE;
    c2.signal(); // c2.broadcast();
    mutex.unlock();
}


// Thread 2's Function
thread_2()
{
    // critical region entrace
    mutex.lock();
    while (! cond_2)
        c2.wait(mutex);
    mutex.unlock();

    // thread T2 in its critical region
    ...

    // critical region exit
    mutex.lock();
    cond_1 = TRUE;
    c1.signal();  // c1.broadcast();
    mutex.unlock();
}
```

6.2.21

Condition Variables. Multiple Conditions (Var 1)

```
// Synchronization mechanims: multiple conditions & one condition variable
Lock mutex;
Condition c;


// Thread 1's Function
thread_1()
{
    // critical region entrace
    mutex.lock();
    while (! cond_1 || ! cond_2)
        c.wait(mutex);
    mutex.unlock();

    // T1's critical region
    ....

    // critical region exit
    mutex.lock();
    .....
    mutex.unlock();
}


// Thread 2's Function
thread_2()
{
    ....
    ....
    // critical region exit
    mutex.lock();
    switch (x) {
      case 1:
        cond_1 = TRUE;
        c.broadcast(); // c.signal() ?
        break;
      case 2:
        cond_2 = TRUE;
        c.broadcast(); // c.signal() ?
        break;
    }
    mutex.unlock();
}
```

6.2.22

Condition Variables. Multiple Conditions (Var 2 - BAD)

```
// Synchronization mechanims: multiple conditions & multiple condition variables & one lock
Lock mutex;
Condition c1, c2;


// Thread 1's Function
thread_1()
{
    // critical region entrace
    mutex.lock();
    while (! cond_1)
        c1.wait(mutex);

    while (! cond_2)
        c2.wait(mutex);
    mutex.unlock();

    // T1's critical region
    ....

    // critical region exit
    mutex.lock();
    .....
    mutex.unlock();
}


// Thread 2's Function
thread_2()
{
    ....
    ....
    // critical region exit
    mutex.lock();
    switch (x) {
      case 1:
        cond_1 = TRUE;
        c1.signal(); // c1.broadcast();
        break;
      case 2:
        cond_2 = TRUE;
        c2.signal(); // c2.broadcast();
        break;
    }
    mutex.unlock();
}
```

6.2.23

Condition Variables. Multiple Conditions (Var 2 - OK)

```
// Synchronization mechanims: multiple conditions & multiple condition variables & multiple locks
Lock mutex1, mutex2;
Condition c1, c2;


// Thread 1's Function
thread_1()
{
    // critical region entrace
    mutex1.lock();
    while (! cond_1)
        c1.wait(mutex1);

    mutex2.lock();
    while (! cond_2)
```

```
        c2.wait(mutex2);
    mutex2.unlock();
    mutex1.unlock();

    // T1's critical region
    ....

    // critical region exit
    .....
}


// Thread 2's Function
thread_2()
{
    ....
    // critical region exit

    switch (x) {
    case 1:
        mutex1.lock();
        cond_1 = TRUE;
        c1.signal(); // c1.broadcast();
        mutex1.unlock();
        break;
    case 2:
        mutex2.lock();
        cond_2 = TRUE;
        c2.signal(); // c2.broadcast();
        mutex2.unlock();
        break;
    }
}
```

6.2.24

Condition Variables: Comparison With Semaphores

- *wait* always suspends the calling process
  - while *P* does this only in case the semaphore's value is zero
- a `signal` can be lost (i.e. not seen)
  - while a *V* is not, because it increments the semaphore's value
- safe to use inside a mutual exclusion area
  - while semaphore could lead to deadlock

6.2.25

Rules to Remember About Using Condition Variables

1. **always use them** (`wait` and `signal`) **inside a mutual exclusion area** (protected by a **lock**)
2. the lock does (normally) not protect the shared resource
   - is would simply provide mutual exclusion, but maybe not needed
   - does not support deadlock-free waiting
3. the lock protects the entrance in and exit from critical region
   - where some condition checks or changes are done
   - in a mutual exclusion manner
4. it **could be needed to recheck the condition** after returning from `wait`
   - use *while* instead of *if*
5. **do not use `wait` without checking a condition**
   - it is not sure the thread must wait (we do not control thread scheduling)
   - `signal` has no history, so can be lost

6.2.26

Rules to Remember About Using Condition Variables (review)

1. **eyes** for seeing
2. **ears** for hearing
3. **mind** for thinking (understanding, learning, remembering)
4. see https://www.youtube.com/watch?v=RZxAc3Grkck
5. review previous slide and make use of the "elements" mentioned above

6.2.27

Producers/Consumers Problem's Implementation With Locks and Condition Variables

```
// Global variables and synchronization mechanisms
const N = 100;          // number of slots in buffer
int count = 0;          // number of messages in buffer
Lock mutex;             // provides mutual exclusion to buffer
Condition producers;    // controls producers' access to buffer
Condition consumers;    // controls consumers' access to buffer


void producer(int item)
{
    // gets mutual exclusion to buffer
    mutex.lock();

    // check if buffer is full
    while (count == N)
      producers.wait(mutex);

    insert_item(item);
    count = count + 1;

    // wakes up a consumer
    consumers.signal();

    // releases the buffer
    mutex.unlock();
}


int consumer(int *item)
{
    // gets mutual exclusion to buffer
    mutex.lock();

    // check if buffer is empty
    while (count == 0)
      consumers.wait(mutex);

    *item = remove_item();
    count = count - 1;

    // wakes up a producer
    producers.signal();

    // releases the buffer
    mutex.unlock();
}
```

6.2.28

Practice (3)

- Using locks and condition variables, write the (pseudo)code for the functions in the code below such that to not allow more than 22 players enter simultaneously on the football field. A player is represented by a thread executing the *football_player()* function.

```
void football_player()
{
    enter_football_field();
    play_football();
    exit_the_footbal_field();
}
```

6.2.29

4 Conclusions

What we talked about

- semaphores
  - a generalization of locks ⇒ could allow more threads pass the barrier
  - important to initialize the semaphore
  - "`P()`" and "`V()`" primitives
  - could also be used as an event counter, usually initialized with 0
- condition variables
  - `wait`: provide a specialized way to wait for a condition to be fulfilled
  - `signal`: provide a way to signal that a condition is fulfilled
- producer-consumer synchronization pattern

6.2.30

Lessons Learned

1. locks could be too restrictive
2. semaphores are more flexible, though must be used with care to avoid deadlock
3. condition variables must be used with a lock, i.e. inside a mutual exclusion area

6.2.31