Operating Systems

Semaphores

Hongliang Liang, BUPT

Sep. 2023

Synchronization between Producer/Consumer

Mutual exclusion only solves part of the concurrency problem

Synchronization: With two or more communicating threads, one thread needs to wait for another thread until some condition is true

Our previous implementation: producer

```
send(message msg)
  acquire(buffer_lock)        //
  while in - out == N do
    release(buffer_lock)      //
    acquire(buffer_lock)      //
  buffer[in % N] = msg
  in = in + 1
  release(buffer_lock)        //
  return
```

Our previous implementation: polling

Our previous implementation uses a loop (**polling**) on buffer conditions, in both `send()` and `receive()` — not desirable

Intuitively, it will be nice to have something like —

`sleep()`: suspends a thread by changing its state to BLOCKED, until another wakes it up

`wakeup(thread_id)`: wakes up another thread, by changing its state to READY

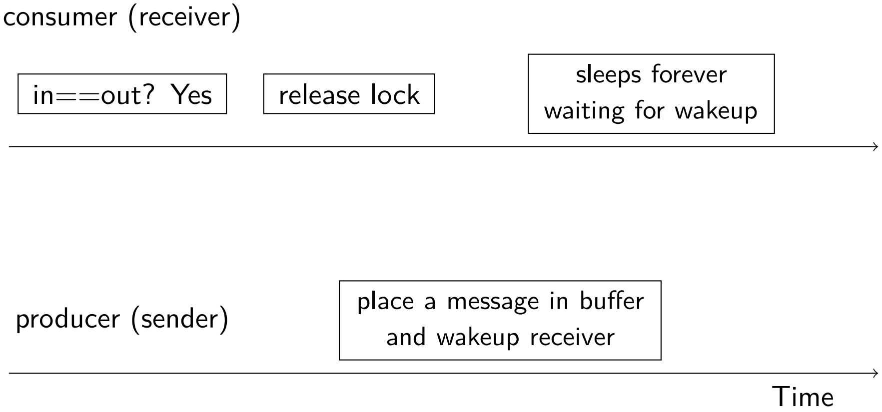
Solving the problem: first try — receive()

```
message receive()
  acquire(buffer_lock)
  while in == out do
    release(buffer_lock)
    sleep()          // add
    acquire(buffer_lock)
  msg = buffer[out % N]
  if in - out == N then
    out = out + 1
    wakeup(senderThread)    // add
  else
    out = out + 1
  release(buffer_lock)
  return msg
```

Solving the problem: first try — send()

```
send(message msg)
  acquire(buffer_lock)
  while in - out == N do
    release(buffer_lock)
    sleep()            // add
    acquire(buffer_lock)
  buffer[in % N] = msg
  if in == out then
    in = in + 1
    wakeup(receiverThread)  // add
  else
    in = in + 1
  release(buffer_lock)
```

The Lost Wakeup Problem



What's causing the problem?

What causes the problem?

The problem is we need to make **two actions before-or-after atomic**:

- Releases the lock
- Calls `sleep()`, which changes the thread state from RUNNING to BLOCKED

We need better synchronization primitives

Intuitively, we need to design a better set of thread synchronization primitives

`sleep()` and `wakeup(thread_id)` does not work well since they do not maintain a "**state**" or "**memory**" about *past* wakeups

Semaphores: maintaining a “table count”

Analogy: the person at the entrance of a restaurant who oversees table assignments

- She needs to maintain a count of unoccupied tables
- When guests arrive, she decrements the table count for each table taken
- When there is no table left, guests will have to wait in a queue
- As tables are freed up, waiting guests are allowed into the restaurant

Semaphores: maintaining a "table count"

Edsger Dijkstra, a 1972 Turing Award winner, proposed Semaphore primitives, **down()** and **up()**, in 1965

Defining semaphores: the first alternative

- A semaphore is a non-negative integer that remembers past wakeups
- `down(semaphore)`: if semaphore > 0, decrement semaphore. Otherwise, wait until another thread increments semaphore, then try to decrement again
- `up(semaphore)`: increment semaphore, and wake up all threads waiting on semaphore

Defining semaphores: the second alternative

The previous definition does not allow a negative count

We can instead allow the count to go negative

- A positive value: it is the number of resources available
- A negative value: its absolute value is the number of threads waiting on available resources
- Just like in a restaurant!

Semantics of Down() and Up()

- `down(semaphore)`: decrement semaphore; if its value is negative, add itself to the waiting queue and change the thread state to BLOCKED
- `up(semaphore)`: increment semaphore, and wake up one of the threads waiting on semaphore

BLITZ semaphores use the second alternative

```
class semaphore
  count: int
  waitingThreads: List [Thread]
up()
  disable interrupts
  count = count + 1
  if count <= 0
    t = waitingThreads.Remove()
    t.status = READY
    readyList.addToEnd(t)
  enable interrupts
down()
  disable interrupts
  count = count - 1
  if count < 0
    waitingThreads.AddToEnd(currentThread)
    currentThread.Sleep()
  enable interrupts
```

Binary semaphores

A binary semaphore: takes on only values of 0 and 1

a binary semaphore can be used as a mutex lock **without the need for polling ("spin lock")**:

- down() corresponds to acquire(),
- up() corresponds to release()

Solving the P-C problem with semaphores

`full`: counting the number of slots that are occupied

- initialized to 0

`empty`: counting the number of slots that are empty

- initialized to the size of the buffer

`mutex`: make sure the sending and receiving threads do not access the shared buffer at the same time

- initialized to 1
- a binary semaphore

Thread synchronization and mutual exclusion

- `mutex` used to solve the mutual exclusion problem
- `full` and `empty` used for thread synchronization

Solving the problem with binary semaphores

```
semaphore mutex = 1, empty = N, full = 0
send(message msg)
  down(mutex)       //*
  down(empty)       //
  buffer[in % N] = msg
  in = in + 1
  up(full)      //
  up(mutex)     //*
message receive()
  down(mutex)       //*
  down(full)        //
  msg = buffer[out % N]
  out = out + 1
  up(empty)     //
  up(mutex)     //*
  return msg
```

Solving the problem with binary semaphores

```
semaphore mutex = 1, empty = N, full = 0
send(message msg)
  down(mutex)        //*
  down(empty)        //
  buffer[in % N] = msg
  in = in + 1
  up(full)       //
  up(mutex)      //*
message receive()
  down(mutex)        //*
  down(full)         //
  msg = buffer[out % N]
  out = out + 1
  up(empty)      //
  up(mutex)      //*
  return msg
```

Correct?

Potential for deadlocks

- `mutex` was decremented before `empty` instead of after it
- If the buffer was completely full, the sender thread will block on `empty`, with `mutex` set to 0 already
- The next time the receiver thread tried to access the buffer, it would do a down on `mutex`
- `mutex` is now 0, so the receiver thread will block, too
- Both threads will be blocked forever

Solving the problem with binary semaphores

```
semaphore mutex = 1, empty = N, full = 0
send(message msg)
  down(empty)
  down(mutex)         //*
  buffer[in % N] = msg
  in = in + 1
  up(mutex)       //*
  up(full)
message receive()
  down(full)
  down(mutex)         //*
  msg = buffer[out % N]
  out = out + 1
  up(mutex)       //*
  up(empty)
  return msg
```

Improving acquire() and release()

- acquire() has been implemented using a TSL instruction in a spin loop
- Spin loops consume processor cycles and should be avoided
- If **acquire()** finds that the lock is LOCKED, a better idea is to put the thread itself to BLOCKED, waiting for another thread to release the lock
- You're asked to implement this improvement in our Lab 2 in BLITZ
  - `waitingThreads`: a list of threads suspended and waiting on the lock
  - `heldBy`: the current state of the lock — which thread is holding the lock
  - Think about race conditions and correctness carefully