Operating systems. Lecture 12

Michał Goliński

2018-12-18

Introduction

Recall

- Critical section problem
- Peterson's algorithm
- Synchronization primitives
  - Mutexes
  - Semaphores

Plan for today

- Classical problems of multithreaded programming
  - Producer and consumer
  - Sleeping barber
  - Readers and writers
  - Dining philosophers
- Deadlocks

Producer and consumer

Problem statement

We must synchronize two processes: one is produces some data, the other consumes it. Both share a finite size buffer that should not be overfilled. If the buffer becomes empty, the consumer should be put to sleep. If the buffer is full, the producer should be put to sleep. Of course the threads should resume work when space or data become available.

Solution

The simplest solution uses two semaphores to synchronize the producer and consumer. One of the semaphores (`put`) is responsible for letting the consumer to write data to the buffer, the other (`take`) is responsible for letting the consumer take the data from the buffer.

The `take` semaphore starts with zero, the `put` semaphore starts with the number of empty places in the buffer.

Code

```
void producer() {
  while(true) {
    wait(put);
    putIntoBuffer(item);
    signal(take);
  }
}
```

```
void consumer() {
  while(true) {
    wait(take);
    item = takeFromBuffer();
    signal(put);
  }
}
```

Observe that the code is symmetrical. The producer produces items, but the consumer "produces" empty places in the buffer.

Note

In this solution both process access the buffer, potentially at the same time. One must make sure that the implementation allows that (the usual implementation with a cyclic buffer is safe in this regard).

Variations

A common variation has multiple consumers and producers, not just two. Then two semaphores are probably not enough. The easiest solution is to use a mutex that will make sure that at most one process is changing the buffer. A more sophisticated solution could use two mutexes: one for producers and one for consumers.

Sleeping barber

Problem statement

This problem is very similar to the previous one. This time we have one barber who gives haircut to the clients. If there are no waiting customers, the barber goes to sleep (he should wake up when a customer shows up and give him a haircut). The customers can wait in a finite size waiting room. If there are not enough space in the waiting room, the customer leaves.

The solution

Solution is left as a task for classes.

Readers and writers

Problem statement

Imagine we have a large library. There are two types of users of this library: readers and writers. Multiple readers may use the library at the same time. But a writer must me alone in the library: no other readers nor writers may access the library while the writer is inside.

Solution one

We will use one binary semaphores `library` (to synchronize readers and writers) and one mutex: `mutex`.

Code

```
void reader() {
  lock(mutex);
  ++noOfReaders;
  if (noOfReaders == 1)
    wait(library);
  unlock(mutex);
  readLibrary();
  lock(mutex);
  --noOfReaders;
  if (noOfReaders == 0)
    signal(library);
  unlock(mutex);
}
```

```
void writer() {
  wait(library);
  writeLibrary();
  signal(library);
}
```

Discussion

Observe that this solution has a strong preference for readers: if one reader is in the library, the next reader does not even look at the `library` semaphore. Hence, if there are new readers entering the library, the writer may wait indefinitely to enter the library.

Solution two

We will use two binary semaphores `library` (to synchronize readers and writers) and `readRequest` and two mutexes: `rmutex` and `wmutex`.

Code

```
void reader() {
  wait(readRequest);
  lock(rmutex);

  ++noOfReaders;
  if (noOfReaders == 1)
    wait(library);

  unlock(rmutex);
  signal(readRequest);
  readLibrary();

  lock(rmutex);
  --noOfReaders;
  if (noOfReaders == 0)

    signal(library);
  unlock(rmutex);
}
```

```
void writer() {
  lock(wmutex);
  ++noOfWriters;

  if (noOfWriters == 1)
    wait(readRequest);
  unlock(wmutex);

  wait(library);
  writeLibrary();
  signal(library);

  lock(wmutex);
  --noOfWriters;
  if (noOfWriters == 0)

    signal(readRequest);
  unlock(wmutex);
}
```

Discussion

Observe that this time we have a preference for writers. This time a reader might be starved when writers keep showing up.

Read-write lock

This reader-writer access mechanism happens often in real-life systems. Because of that languages and libraries supply so called read-write locks: a mutex-like lock that can be used to solve this problem.

A read-write lock can be either *locked* – this works like for a normal mutex and can be done by just one thread at a time (intended for writers) or it can be *read-locked* – this allows for many threads to read from a shared resource ensuring that no writer can acquire the lock concurrently.

Read-write lock cont.

The C++17 standard library defines the `std::shared_mutex` class which implements this scheme, similarly, the pthreads library has the `pthread_rwlock_t` type and corresponding functions.

Read-copy-update (RCU)

The Linux kernel uses another solution for its readers-writer problems. The kernel has many linked lists which are often traversed by many readers. The are not changed often, but still this has to be done from time to time. Using a writer lock on the list would be detrimental to performance, hence another idea is used.

If a list element needs to be changed or a new is to be added, it is constructed off the existing list, only as the last step the `next` pointer of an existing element is atomically set to the new element. This way all th readers traversing the list see either the old or the new list.

Deleting in RCU

If an element needs to be deleted, the `next` pointer of the preceeding element is manipulated atomically to skip the requested element. This way for new readers the element is inaccessible. The writer needs only to wait for all the old readers to traverse off the deleted element and it can be deallocated from memory.

Deadlock

Problem

Mutexes and semaphores usually give an easy way to solve the different mutual exclusion problems. The biggest problem is threads waiting for each other. We talk about a **deadlock** when each thread from a group of threads waits for another thread to take some action (usually to release a lock). This is a very serious problem in concurrent programming but it can also happen in other distributed systems.

Simple example of a deadlock

```
binarysem mutex;

void thread() {
  wait(mutex);
}

void main() {
  initialsem(mutex, 1);
  cobegin {
    thread();
    thread();
  }
}
```

Another example of a deadlock

```
binarysem mutex[2];

void thread1() {
  for(;;) {
    wait(mutex[0]);
    wait(mutex[1]);
    signal(mutex[1]);
    signal(mutex[0]);
  }
}

void thread2() {
  for(;;) {
    wait(mutex[1]);
    wait(mutex[0]);
    signal(mutex[0]);
    signal(mutex[1]);
  }
}

void main() {
  initialsem(mutex[0], 1);
  initialsem(mutex[1], 1);
  cobegin {
    thread1();
    thread2();
  }
}
```

Deadlock in C

```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>

pthread_mutex_t mutex[2];

void *th1(void *arg) {
    pthread_mutex_lock(mutex);
    sleep(1);
    pthread_mutex_lock(mutex+1);
    pthread_mutex_unlock(mutex+1);
    pthread_mutex_unlock(mutex);
```

```
    return NULL;
}

void *th2(void *arg) {
    pthread_mutex_lock(mutex+1);
    sleep(1);
    pthread_mutex_lock(mutex);
    pthread_mutex_unlock(mutex);
    pthread_mutex_unlock(mutex+1);
    return NULL;
}

int main() {
    pthread_t thread[2];
    pthread_attr_t attr;
    void *res;
    pthread_create(thread, NULL, &th1, NULL);
    pthread_create(thread+1, NULL, &th2, NULL);
    printf("Created threads\n");
    pthread_join(thread[0], NULL);
    pthread_join(thread[1], NULL);
    return 0;
}
```

Necessary conditions

In order for a deadlock to occur four conditions must happen at the same time:

- mutual exclusion – a resource has to be non-sharable
- hold and wait – a process is holding a resource and waiting to acquire another resource
- no preemption – the resource can only be released voluntarily
- circular wait – there is a cycle of dependency: P_1 waits for P_2, P_2 waits for P_3, …, P_N waits for P_1.

Deadlock handling

Most operating systems do not try to detect deadlocks. At least not in user programs. It is assumed they do not occur as frequently to make their detection worthwhile. Moreover, any such system is not able to detect all deadlocked programs (similarly to the halting problem in computability theory). It is assumed the user will kill the erratic program.

Some OS try to detect and handle deadlocks in the kernel. A hardware watchdog is often necessary to do that. The solution is often a reboot.

Deadlock avoidance

Because handling a deadlock is often impossible (short of killing the affected processes), one should try to avoid them altogether. The most well known is the Dijkstra's resource hierarchy solution: one should impose a strict ordering on resources (and corresponding locks) and **always** acquire the locks according to this order in any threads (releasing them in reverse order). This is a sure method to avoid a deadlock, unfortunately it is not always easy to do.

Lock-free programs

Another method to avoid deadlock is to write lock-free programs, usually by employing atomic operations. Unfortunately this also is not easy to do, although usually gives a much faster solution than code riddled with locks. An application of this method is the RCU mechanism for modifying linked lists.

Dining philosophers

Problem statement

Five philosophers sit around a round table. At the center of the table there is a pot of rice and between each two philosophers there is a chopstick. Philosophers for the most time think about the nature of the universe, but at irregular intervals they become hungry. In that moment a philosopher takes the two chopsticks that are most adjacent, eats some rice and puts the chopsticks down. Philosophers do not speak to each other.

Create the scheme that allows the philosophers to not starve to death.

Wrong solution

We have an array of 5 semaphores called `chopstick`. Philosophers take first the left, then the right chopstick.

```
binarysem chopstick[5];

void philosopher(int i) {
  int j;
  j = (i+1)%5;
  for(;;) {
    cout << "Philosopher " << i << " is hungry" << endl;
    wait(chopstick[i]);
    wait(chopstick[j]);
    cout << "Philosopher " << i << " eats..." << endl;
    signal(chopstick[j]);
    signal(chopstick[i]);
  }
```

```
}

void main() {
  initialsem(chopstick[0], 1);
  initialsem(chopstick[1], 1);
  initialsem(chopstick[2], 1);
  initialsem(chopstick[3], 1);
  initialsem(chopstick[4], 1);
  cobegin {
    philosopher(0);
    philosopher(1);
    philosopher(2);
    philosopher(3);
    philosopher(4);
  }
}
```

Dijkstra's solution

The problem has a certain symmetry. Dijkstra's solution requires to break it. We number the chopsticks and each philosopher is required to first take the chopstick with the lower number. This prevents the deadlock to occur.