Operating systems. Lecture 11

Michał Goliński

2018-12-11

Introduction

Recall

- Threads
- `pthreads`
- Threads in the standard libraries
- OpenMP
- Amdahl's law
- BACI

Plan for today

- Real life story
- Critical section problem
- Peterson's algorithm
- Synchronization primitives

Story

Old style bank

Imagine that we run an old style bank (wild west style). We start small. We have just a single teller and multiple safes – accounts. When a customer comes and wants to withdraw or deposit money, the teller would check the ID, go to the appropriate safe and he would either take some money from it or deposit it there.

This scheme runs without problems as long as credentials are checked properly by the teller.

Small improvement

We decide the bank runs too slowly. Therefore we hire a teller's assistant that will check the safes and bring money at the teller's request. This still runs without problems.

Another improvement

Using many safes has become problematic. We build one giant secure vault and replace the safes with a digital ledger accessible only from the vault.

Multiple tellers

Hiring multiple tellers will now bring us a problem. Say two customers want to get a $100 from a common account at the same time with two different tellers. Say the account has a balance of $100. Both tellers send their assistants to check the balance. Both of them come back with the information that money can be withdrawn. Both teller tell their assistants to get a $100 from the vault and to set the account balance to $0. This of course means the bank lost a $100.

Race condition

Of course this probably would be caught in real life, but many error in multithreaded programming look like that. Observe, that there would be no problems if the first assistant finishes the withdrawal before the other one checks the balance. If a result of parallel operation depends on the order of those operation, we speak of a *race condition* (or *race* for short).

Critical section problem

Critical section

In practice the different threads for the most part operate on unrelated data (like in our summing example). But from time to time a thread needs to inform other threads or the main thread of its results. This accessing shared resources is called the *critical section*. Simple errors in multithreaded programming come from wrong protection of the critical section.

Critical section problem

The problem we face is: how to protect the critical section in such a way that at most one thread can enter its critical section at a time.

Critical section problem

A good solution to the critical section problem should have the following properties:

- mutual exclusion – at most one thread can enter its critical section at a time
- progress – if no thread is executing its critical section and some threads wish to enter the critical section, one of them is permitted to do so
- bounded waiting – if a thread makes a request to enter its critical section, there is a bound on other threads entering their critical section before it.

Threads

We will think that threads execute the following code:

```
while (true) {
  enterCriticalsection();
  criticalSection();
  leaveCriticalSection();
  normalCode();
}
```

A thread may exit only when outside of its critical section.

Naive solution 1

We have a global `int lock` initiated to 0.

```
void thread1() {
  int i = 0;
  while (lock == 1);
  lock = 1;
  for(i = 0; i < 10; ++i)
    cout << "A";
  cout << endl;
  lock = 0;
}
```

```
void thread2() {
  int i = 0;
  while (lock == 1);
  lock = 1;
  for(i = 0; i < 10; ++i)
    cout << "B";
  cout << endl;
  lock = 0;
}
```

Full example

```
int lock = 0;

void thread1() {
  int i = 0;
  while (lock == 1);
  lock = 1;
  for(i = 0; i < 10; ++i)
    cout << "A";
  cout << endl;
  lock = 0;
}

void thread2() {
  int i = 0;
  while (lock == 1);
  lock = 1;
```

```
  for(i = 0; i < 10; ++i)
    cout << "B";
  cout << endl;
  lock = 0;
}

main() {
  cobegin {
    thread1();
    thread2();
  }
}
```

Naive solution 1 does not work!

It may happen that the context switch between the threads happens after one of the threads leaves the `while` loop. The other is then permitted to enter its critical section (`lock == 0`), but the first thread also enters the critical section when it resumes operation.

Naive solution 2

This time we create a two element array `request`, initialized to zeros.

```
void thread1() {
  int i = 0;
  while (request[1] == 1);
  request[0] = 1;
  for(i = 0; i < 10; ++i)
    cout << "A";
  cout << endl;
  request[0] = 0;
}
```

```
void thread2() {
  int i = 0;
  while (request[0] == 1);
  request[1] = 1;
  for(i = 0; i < 10; ++i)
    cout << "B";
  cout << endl;
  request[1] = 0;
}
```

Full example

```
int request[2];

void thread1() {
  int i = 0;
  while (request[1] == 1);
  request[0] = 1;
  for(i = 0; i < 10; ++i)
    cout << "A";
  cout << endl;
```

```
  request[0] = 0;
}

void thread2() {
  int i = 0;
  while (request[0] == 1);
  request[1] = 1;
  for(i = 0; i < 10; ++i)
    cout << "B";
  cout << endl;
  request[1] = 0;
}

main() {
  request[0] = 0;
  request[1] = 0;
  cobegin {
    thread1();
    thread2();
  }
}
```

Naive solution 2 does not work!

This fails for exactly the same reason as before: one of the threads can leave its waiting loop and the context may switch then.

Naive solution 3

This time we create a variable `turn`, initialized to zero.

```
void thread1() {
  int j = 0;
  for (j = 0; j < 8; ++ j) {
    int i = 0;
    while (turn != 0);
    for(i = 0; i < 10; ++i)
      cout << "A";
    cout << endl;
    turn = 1;
  }
}
```

```
void thread2() {
  int j = 0;
  for (j = 0; j < 5; ++ j) {
    int i = 0;
    while (turn != 1);
    for(i = 0; i < 10; ++i)
      cout << "B";
    cout << endl;
    turn = 0;
  }
}
```

Full example

```
int turn = 0;

void thread1() {
  int j = 0;
  for (j = 0; j < 8; ++ j) {
    int i = 0;
    while (turn != 0);
    for(i = 0; i < 10; ++i)
      cout << "A";
    cout << endl;
    turn = 1;
  }
}

void thread2() {
  int j = 0;
  for (j = 0; j < 5; ++ j) {
    int i = 0;
    while (turn != 1);
    for(i = 0; i < 10; ++i)
      cout << "B";
    cout << endl;
    turn = 0;
  }
}

main() {
  cobegin {
    thread1();
    thread2();
  }
}
```

This also fails but for different reasons

This actually achieves mutual exclusion, but threads alternate their critical section and one has to wait for the other and will wait indefinitely if the other finishes execution. So no progress happens.

Naive solution 4

Once again we create a two element array `request`, initialized to zeros.

```
void thread1() {
  int j = 0;
  for (j = 0; j < 5; ++ j) {
    int i = 0;
    request[0] = 1;
    while (request[1] == 1) {
      request[0] = 0;
      while (request[1] ==
      ↪  1);
      request[0] = 1;
    }
    for(i = 0; i < 10; ++i)
      cout << "A";
    cout << endl;
    request[0] = 0;
  }
}
```

```
void thread2() {
  int j = 0;
  for (j = 0; j < 8; ++ j) {
    int i = 0;
    request[1] = 1;
    while (request[0] == 1) {
      request[1] = 0;
      while (request[0] ==
      ↪  1);
      request[1] = 1;
    }
    for(i = 0; i < 10; ++i)
      cout << "B";
    cout << endl;
    request[1] = 0;
  }
}
```

Full example

```
int request[2];

void thread1() {
  int j = 0;
  for (j = 0; j < 5; ++ j) {
    int i = 0;
    request[0] = 1;
    while (request[1] == 1) {
      request[0] = 0;
      while (request[1] == 1);
      request[0] = 1;
    }
    for(i = 0; i < 10; ++i)
      cout << "A";
    cout << endl;
    request[0] = 0;
  }
}

void thread2() {
  int j = 0;
  for (j = 0; j < 8; ++ j) {
    int i = 0;
    request[1] = 1;
    while (request[0] == 1) {
      request[1] = 0;
      while (request[0] == 1);
      request[1] = 1;
    }
    for(i = 0; i < 10; ++i)
      cout << "B";
    cout << endl;
    request[1] = 0;
  }
}

main() {
  request[0] = 0;
  request[1] = 0;
  cobegin {
    thread1();
    thread2();
  }
}
```

Naive solution 4 also does not work

This is a bit harder. But one can see that although there is progress and mutual exclusion, there is no bounded waiting.

Peterson's algorithm

We need a global `request` array initialized to zeros and a `turn` variable.

```
void thread1() {
  int j = 0;
  for (j = 0; j < 5; ++ j) {
    int i = 0;
    request[0] = 1;
    turn = 1;
    while (request[1] == 1
           && turn == 1);
    for(i = 0; i < 10; ++i)
      cout << "A";
    cout << endl;
    request[0] = 0;
  }
}
```

```
void thread2() {
  int j = 0;
  for (j = 0; j < 8; ++ j) {
    int i = 0;
    request[1] = 1;
    turn = 0;
    while (request[0] == 1
           && turn == 0);
    for(i = 0; i < 10; ++i)
      cout << "A";
    cout << endl;
    request[1] = 0;
  }
}
```

Full example

```
int request[2];
int turn;

void thread1() {
  int j = 0;
  for (j = 0; j < 5; ++ j) {
    int i = 0;
    request[0] = 1;
    turn = 1;
    while (request[1] == 1
           && turn == 1);
    for(i = 0; i < 10; ++i)
      cout << "A";
    cout << endl;
    request[0] = 0;
  }
}

void thread2() {
  int j = 0;
```

```
  for (j = 0; j < 8; ++ j) {
    int i = 0;
    request[1] = 1;
    turn = 0;
    while (request[0] == 1
           && turn == 0);
    for(i = 0; i < 10; ++i)
      cout << "B";
    cout << endl;
    request[1] = 0;
  }
}

main() {
  request[0] = 0;
  request[1] = 0;
  cobegin {
    thread1();
    thread2();
  }
}
```

Peterson's algorithm works

If one of the threads intends to enter its critical section, but the other is not in the critical section and is not in its entry to the critical section, then the thread will be granted immediate access.

Peterson's algorithm works cont.

If thread1 intends to enter its critical section, but thread2 is in the critical section then it has to wait. When thread2 executes `request[1] = 0`, then thread1 would be permitted to enter if it resumes execution before thread2 reaches its entry section. Otherwise, see the next point.

Peterson's algorithm works cont.

If both threads have their `request` variable set, then the thread that first sets `turn` waits until the other sets `turn` and enters the critical section. The other will enter the critical section next.

Peterson's algorithm and modern hardware

Analyzing the algorithm we have made a tacit assumption, that both threads see the memory operations in the same order. This may seem natural, but for performance reasons modern CPUs can reorder memory operations. To remedy this, processors supply a *memory fence*

instruction: all memory operation before the fence are occcuring to all processors before operations that follow the fence. These instructions are quite expensive in terms of CPU time (hundreds of cycles), but they are sometimes unavoidable.

Busy waiting

In these algorithms we used a busy loop `while(condition);`. This is of course very inefficient. True systems with schedulers should use the `yield` operation: it requests for the current thread to be preempted and a new one scheduled it its place.

Busy waiting may be used if we know they will not run for very long, then it may be faster than yielding to the scheduler.

Synchronization primitives

Definition

Synchronization primitives are the basic building blocks supplied by the language/operating system that allow programmers to make their multithreaded programs behave properly. We will discuss atomics, mutexes (locks) and semaphores.

Atomic instructions

Normally, say, incrementing a memoery location is executed in steps: load the current value, modify it and store it back. Most modern CPUs supply some atomic instructions, i.e., a read-modify-store operation that is visible as one from any other thread.

The test-and-set instruction is a very helpful one: it sets a variable to the given value and returns the old one in a single, indivisible step.

Atomic compare-and-swap

A more complicated one is the compare-and-swap atomic instruction. It takes three arguments: a memory location and two numbers. If the memory is equal to the first number, then it is set to the second argument and the original value is returned. It all happens in one indivisible step. This a very powerful synchronization mechanism, i.e., one can implement other primitives using this instruction.

The x86 processors have the `CMPXCHG` instruction which implements CAS.

Mutex

A mutex (also called a lock) is a very simple, yet powerful synchronization primitive. It lives in two states: locked and unlocked. Any thread may attempt to *acquire* the lock. If the mutex was unlocked, the operation succeeds and the mutex becomes locked. If the mutex was locked, the thread will be blocked and will resume when the mutex becomes available. When the mutex is no longer needed it has to be unlocked.

Mutex is a simple mechanism to protect a critical section: if a thread has to acquire a shared mutex before it enters the critical section and unlocks it afterwards, then we get mutual exclusion.

Mutex in POSIX

POSIX supplies `pthread_mutex_t` and `pthread_spinlock_t`. Both have the same functionality, but the first puts the thread to sleep (preempts it) and the second uses busy waiting.

Semaphore

A semaphore is an integer for which two special operation are available:

- *wait* (also denoted by *P*) – decrements the number, blocks if the number is 0
- *signal* (also denoted by *V*) – increments the number, wake up one of the waiting threads (if any)

With mutexes each thread would first lock, then unlock it. With semaphores this need not be the case: waiting and signaling can be done by different threads.

Binary semaphore

A binary semaphore , i.e. a one that takes only the values 0 and 1 can be used as a mutex, which leads to many confusions.

Primitives in BACI

Semaphore

BACI supplies the `semaphore` type that which is a non-negative integer. Its value can be set on creation or by the `initialsem` procedure:

```
semaphore s = 17;
initialsem(s, 3);
```

Otherwise one can only `wait` or `signal` on a semaphore:

```
void wait(semaphore& s);
void signal(semaphore& s);
```

Binary semaphore

There is also a `binarysem` that can take only the values 0 and 1. Otherwise it works exactly the same.

`atomic`

A function can be declared `atomic` , then the function will be non-preemptive, i.e., it will run without context switches. This gives a way to implement and test other primitives than semaphores.

`suspend` and `revive`

The `suspend` is called, a thread is suspended. It can be woken up using `revive`.

```
void suspend();
void revive(int processNumber);
void which_proc();
```