Take me for a spin

Johan Montelius and Amir H. Payberah

1 Introduction

We will first experience that we can not implement any synchronization primitives using regular read and write operations. Then we will implement a spinlock using a `test-and-set` atomic operation. Once we know that we can control the execution, we will use the `pthread` library to implement a concurrent data structure.

2 total-store-order

The only thing ordered with total store order is the store operations. Read operations are a bit free to move and can thus be done before a write operation that we just have issued. As we will see, this prevents us from writing a simple lock using only read and write operations. To see that things break down, let's implement something that should work and then obviously does not.

2.1 Peterson's algorithm

The algorithm is quite simple, but a bit mind-bending before you understand how it works. We will have two variables, one per thread, indicating whether a thread is interesting to enter the critical section. Since both threads can set their flags to *interested* simultaneously, we could have a draw that we somehow need to sort out. We then use a third variable that will decide whose turn it is. The funny thing with the algorithm is that both threads will try to set this to signal that it is the other thread's turn.

We will start by implementing an example and then see how it works and discover that it does not. Our task is to have two threads incrementing a shared counter.

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

volatile int count = 0;

volatile int request[2] = {0,0};
volatile int turn = 0;
```

The two variables used to express interest are grouped in a structure called `request`. The two threads will have identifiers `0` and `1` and can thus

easily access the right flag. To take the lock, a thread will set its own flag to 1 and then wait until either the other thread is not interested or it is its turn - this is the tricky part to understand. Releasing the lock is simply done by resetting the request flag.

```
void lock(int id) {
  request[id] = 1;
  int other = 1-id;
  turn = other;
  while(request[other] == 1 && turn == other) {}; // spin
}

void unlock(int id) {
  request[id] = 0;
}
```

We are now ready to describe the threads. As you know, the procedure we use to start a thread takes one argument, so we have to group the things we want to pass to each thread. We will pass two values, a value we will use to increment the counter and an identifier of the thread.

```
typedef struct args {int inc; int id;} args;

void *increment(void *arg) {
  int inc = ((args*)arg)->inc;
  int id = ((args*)arg)->id;

  for(int i = 0; i < inc; i++) {
    lock(id);
    count++;
    unlock(id);
  }
}
```

The main procedure will start the two threads and wait until they finish before writing out the counter's value.

```
int main(int argc, char *argv[]) {

  if(argc != 2) {
    printf("usage peterson <inc>\n");
    exit(0);
  }

  int inc = atoi(argv[1]);

  pthread_t one_p, two_p;
  args one_args, two_args;

  one_args.inc = inc;
  two_args.inc = inc;

  one_args.id = 0;
  two_args.id = 1;

  pthread_create(&one_p, NULL, increment, &one_args);
  pthread_create(&two_p, NULL, increment, &two_args);
  pthread_join(one_p, NULL);
  pthread_join(two_p, NULL);

  printf("result is %d\n", count);
  return 0;
}
```

When compiling the program, you must include the `pthread` library. This is done using the `-l` flag.

```
 > gcc -o peterson peterson.c -lpthread
```

If everything works, you should be able to run the program like this:

```
 > ./peterson 100
start 0
start 1
result is 200
```

We could have ended this tutorial but try with some higher values to see what is happening. If you are running on a single-core CPU, you will never see something weird but take my word - it does not work on a multi-core CPU. The reason is that each thread has a sequence, write own then read other, and the algorithm is based on setting your own flag before checking the other thread's flag. Total store order does not guarantee that this order is kept - it could be that we see the value of others before our own value has been propagated to memory. This will then allow both threads to think that the other thread is not interested in the lock.

2.2 the atomic `test-and-set`

The savior is a family of atomic operations that will both read and modify a memory location. We will only make small changes to our program, so make a copy and call it `swap.c`. First, we will remove the sequence declaring the two flags and turn variable and only have one global variable that will serve the purpose of a mutual-exclusion lock. We then define the magical procedure that tries to grab this lock. The procedure will compare the content of a memory location, and if it is `0`, unlocked, it will replace it with a `1`, locked. If it succeeds, it will return `0`, and we know that we have obtained the lock.

```
volatile int global = 0;

int try(volatile int *mutex) {
  return __sync_val_compare_and_swap(mutex, 0, 1);
}
```

The locking procedure is changed to operate on a mutex and repeatedly tries to grab this lock until it either succeeds or the end of time. Releasing the lock is simple; we just write a `0`.

```
void lock(volatile int *mutex) {
  while(try(mutex) != 0); // spin
}

void unlock(volatile int *mutex) {
  *mutex = 0;
}
```

To use the new lock, we must adapt the increment procedure and provide the mutex variable as an argument.

```
typedef struct args {int inc; int id; volatile int *mutex;} args;

void *increment(void *arg) {
  int inc = ((args*)arg)->inc;
  int id = ((args*)arg)->id;
  volatile int *mutex = ((args*)arg)->mutex;

  for(int i = 0; i < inc; i++) {
    lock(mutex);
    count++;
    unlock(mutex);
  }
}
```

Fixing the main procedure is left as an exercise to the reader, as it is often phrased when one is tired of writing code. If you are handy, you can compile the program without any errors and are ready to see if it works.

```
> ./swap 10000000
start 2
start 1
result is 20000000
```

- Ahh, (I know this is not a proof) complete success. This is it; we have conquered the total-store-order limitations and can now synchronize our concurrent threads.

3 Spinlocks

The lock we have implemented in `swap.c` is a so-called *spinlock*. It tries to grab the lock and will keep trying until the lock is successfully taken. This is a very simple strategy, and in most (or at least many) cases, it is a perfectly fine strategy. The first downside is if the thread that is holding the lock is busy for a longer period, then the spinning thread will consume CPU resources that could be better spent on other parts of the execution. This performance problem does not prevent the execution from making progress. If this was the only problem, one could use spinlocks and only tackle the problem if there was a performance problem. The second problem is more severe and can put the execution in a deadlock state.

3.1 priority inversion

Assume we have two threads, one low priority, and one high priority. The operating system uses a scheduler that always prioritizes a high-priority thread. Further, assume that we are running on a single core machine so that only one thread can make progress at any given time.

The two threads will live happily together, but the low-priority thread will only get a chance to execute if the high-priority thread is waiting for some external event. What will happen if the low-priority thread takes a lock and starts to execute in the critical section and the high-priority thread wakes up and tries to grab the same lock?

You have to think a bit, and when you realize what will happen, you know what *priority inversion* is. This is a problem, and there is no simple solution, but one way is to boost the priority of threads holding locks. A high-priority thread that finds a lock being held would delegate some of its priority to the thread holding the lock. This sensible solution requires the operating system to know about locks and what thread is doing what. We will avoid this problem simply by saying that it is not up to the lock constructor but to the one who wants to introduce priorities.

3.2 yield

We can solve the performance problem by letting the thread that tries to grab the lock go to sleep if the lock is held by someone else. We will first experiment to see what we are trying to fix by adding a simple print statement in the loop, so a dot will be printed on the screen. Run a few tests and see how often it happens that we end up in a loop. Note that the print statement will take much longer than just spinning, so it is not an exact measurement but indicates the problem.

```
  while(try(mutex) != 0) {
    printf(".");
  }
```

We will get a more accurate number if we keep an internal counter in the spinlock and return the number of loops it took to take it as part of taking the lock. This is a small change to the program. Adapt the `lock()` procedure and then change the `increment()` procedure so it will print the total number of loop iterations it has performed during the benchmark.

```
int lock(volatile int *mutex) {
  int spin = 0;
  while(try(mutex) != 0) {
    spin++;
  }
  return spin;
}
```

Now for the fix - add the statement `sched_yield();` after the increment of the `spin` counter. Does the situation improve?

3.3 wake me up

Going to sleep is only half of the solution; the other one is being told to wake up. The concept we will use to achieve what we want is *futex locks*. GCC does not provide a standard library to work with futex:s, so you will have to build your own wrappers. Take a copy of your program, call it `futex.c` and make the following changes.

First of all, we need to include the proper header files. Then we use the `syscall()` procedure to make calls to the system function `SYS_futex`. You do not have to understand all the arguments, but we are saying that to do a `futex_wait()` we look at the address provided by `futexp` and if it is equal to `1` we suspend. When we call `futex_wake()` we will wake at most `1` thread that is suspended on the futex address. In both cases, the three last arguments are ignored.

```c
#include <unistd.h>
#include <linux/futex.h>
#include <sys/syscall.h>

int futex_wait(volatile int *futexp) {
  return syscall(SYS_futex, futexp, FUTEX_WAIT, 1, NULL, NULL, 0);
}
void futex_wake(volatile int *futexp) {
  syscall(SYS_futex, futexp, FUTEX_WAKE, 1, NULL, NULL, 0);
}
```

Now we have the tools to adapt our `lock()` and `unlock()` procedures. When we find the lock taken, we suspend the lock, and when we release the lock, we wake one (if any) suspended thread. Notice that we use the mutex lock as the address we pass to `futex_wait()`. The call to `futex_wait()` will only be suspended if the lock is still held, and this property will prevent race conditions. When we release the lock, we first set the lock to zero and then call `futex_wake()` - what could happen if we did it the other way around?

```c
int lock(volatile int *lock) {
  int susp = 0;
  while(try(lock) != 0) {
    susp++;
    futex_wait(lock);
  }
  return susp;
}

void unlock(volatile int *lock) {
  *lock = 0;
  futex_wake(lock);
}
```

Now for some experiments, run the swap version and compare it to the futex version. Any difference? Why?

3.4 all fine and dandy

With the atomic swap operation and the operating system support to suspend and wake threads, we have everything we need to write correct concurrent programs that efficiently use the computing resources. In the rest of this exercise, we will use POSIX lock primitives from the `pthread` library, but this is just sugar; we have solved the main problems.

4 A protected list

To experiment with the `pthread` lock primitives, we will build a sorted linked list where we can insert and delete elements concurrently. This first solution will be very simple but only allow one thread to operate on the list at any

given time, i.e., any operation is protected in a critical sector. The more refined solution will allow several threads to operate on the list as long as they do it in different parts of the list - tricky, yes!

Since we will use the POSIX locks, we first install the man pages for those. These man pages are normally not included in a GNU/Linux distribution, and they might actually differ from how it is implemented, but it is a lot better than nothing. If you have `sudo` right to your platform, you can install them using the following command.

```
> sudo apt-get install manpages-posix manpages-posix-dev
```

Now let's start by implementing a simple sorted linked list and operations that will add or remove a given element.

4.1 a sorted list

We will start with an implementation that uses a global mutex to protect the list from concurrent updates. We will use the regular `malloc()` and `free()` procedures to manage allocation of cell structures. Each cell will hold an integer, and the list is ordered with the smallest value first. To benchmark our implementation, we will provide a procedure `toggle()` that takes a value and either insert the value in the list if it is not there otherwise removes it from the list. We can then generate a random sequence of integers and toggle them so that we, after a while, will have a list of half the length of the maximum random value.

Let's go - create a file `list.c`, include some header files that we will need, define a macro for the maximum random value (the values will be from `0` to `MAX -1`), and the definition of the cell structure.

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

/* The list will contain elements from 0 to 99 */

#define MAX 100

typedef struct cell {
  int val;
  struct cell *next;
```

We then do something that is not straightforward, but it will save us a lot of special cases, reduce our code, and make things more efficient. We define two cells called `sentinel` and `dummy`. The *sentinel* is a safety that will prevent us from running past the end of the list without checking. When we toggle a value, we will run down the list until either we find it or we find a

cell with a higher value and then insert a new cell. The sentinel will ensure that we always find a cell with a higher value. The `dummy` element serves a similar purpose and is there to guarantee the list is never empty. We can avoid the special case if the list is empty. We also have a mutex that we will use in our first run to protect the list. We use a macro for the initialization since we are fine with the default values.

```
cell sentinel = {MAX, NULL};
cell dummy = {-1, &sentinel};

cell *global = &dummy;

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

The procedure `toggle()` will take a pointer to the list and a value that it will either add or remove. We use two pointers to run through the list, `this` and `prev`. If we find the element, we can remove the cell since we keep a pointer to the previous cell. We take the mutex lock before we start to prevent conflicts with other operations.

```
void toggle(cell *lst, int r) {
  cell *prev = NULL;
  cell *this = lst;
  cell *removed = NULL;

  pthread_mutex_lock(&mutex);

  while(this->val < r) {
      prev = this;
      this = this->next;
  }
  if(this->val == r) {
    prev->next = this->next;
    removed = this;
  } else {
    cell *new = malloc(sizeof(cell));
    new->val = r;
    new->next = this;
    prev->next = new;
  }
  pthread_mutex_unlock(&mutex);
  if(removed != NULL)  free(removed);
  return;
}
```

Note that we always have at least two cells in the list. The pointer we have as an argument to the procedure will never be NULL, and we will always find a value larger than the value we are looking for. Note that we have moved the call to `free()` to the end where we are outside the critical section. We want to do as little as possible in the critical section to increase concurrency. We could be speculative and also do the call to `malloc()` before

we enter the critical section, but we do not know if we will have to create a new cell, so this might not pay off.

4.2 the benchmark

A benchmark thread will loop and toggle random numbers in the list. We define a structure that holds the arguments to the thread procedure and pass an identifier (we can use if we print statistics), how many times we should loop, and a pointer to the list.

```
typedef struct args {int inc; int id; cell *list;} args;

void *bench(void *arg) {
  int inc = ((args*)arg)->inc;
  int id = ((args*)arg)->id;
  cell *lstp = ((args*)arg)->list;

  for(int i = 0; i < inc; i++) {
    int r = rand() % MAX;
    toggle(lstp, r);
  }
}
```

Now we can tie everything together in a `main()` procedure that starts several threads and let them run a benchmark procedure each. There are no strange things in this code; we only need to allocate some dynamic memory to hold the thread structures and bench argument.

```
int main(int argc, char *argv[]) {

  if(argc != 3) {
    printf("usage: list <total> <threads>\n");
    exit(0);
  }
  int n = atoi(argv[2]);
  int inc = (atoi(argv[1]) / n);

  printf("%d threads doing %d operations each\n", n, inc);

  pthread_mutex_init(&mutex, NULL);

  args *thra = malloc(n * sizeof(args));
  for(int i =0; i < n; i++) {
    thra[i].inc = inc;
    thra[i].id = i;
    thra[i].list = global;
  }

  pthread_t *thrt = malloc(n * sizeof(pthread_t));
  for(int i =0; i < n; i++) {
    pthread_create(&thrt[i], NULL, bench, &thra[i]);
  }

  for(int i =0; i < n; i++) {
    pthread_join(thrt[i], NULL);
  }

  printf("done \n");
  return 0;
}
```

Compile, hold your thumbs and run the benchmark. Hopefully, you will see something like this:

```
 > ./list 100 2
2 threads doing 50 operations each
done
```

5 A concurrent list

The protected list is fine, but we can challenge ourselves by implementing a list where we can toggle several elements simultaneously. It should be perfectly fine to insert an element at the beginning of the list and, at the same time, remove an element somewhere further down the list. We have to be careful, though, since if we do not watch out, we will end up with a very strange list or complete failure. You can take a break here and think about how you would like to solve it. You will, of course, have to take some locks, but what is it that we should protect, and how will we ensure that we don't end up in a deadlock?

5.1 Lorem Ipsum

The title above has nothing to do with the solution, but I did not want to call it "one lock per cell" or something since I just asked you to take a break and figure out your own solution. Hopefully, you have done so now and possibly also arrived at the conclusion that we need one lock per cell (we will later see if we can reduce the number of locks).

The dangerous situation we might end up in is that we are trying to add or remove cells directly next to each other. If we have cells 3-5-6 and at the same time try to add 4 and remove 5, we are in trouble. We could end up with 3-4-5-6 (where 5 will also be freed), or 3-6, or (where 4 is lost forever). We will prevent this by ensuring that we always hold locks of the *previous cell* and the `current cell`. If we have these locks, we can either remove the current cell or insert a new one in between. As we move down the list, we will release and grab locks as we go, ensuring that we always grab a lock in front of the lock we hold.

The implementation requires surprisingly few changes to our code, so make a copy of the code and call it `clist.c`. The cell structure is also changed to include a mutex structure. The initialization of the `dummy` and `sentinel` cells are done using a macro which is fine since we are happy with the default initialization. There is no need for a global lock, so this is removed.

```
typedef struct cell {
  int val;
  struct cell *next;
  pthread_mutex_t mutex;
} cell;

cell sentinel = {MAX, NULL, PTHREAD_MUTEX_INITIALIZER};
cell dummy = {-1, &sentinel, PTHREAD_MUTEX_INITIALIZER};
```

The changes to the `toggle()` procedure are not very large. We only have to think about what we are doing. The first thing is that we need to take the lock of the `dummy` cell (which is always there) and the next cell. The next cell could be the `sentinel`, but we do not know; we only know that there is a cell. Note that you can not trust the pointer `prev->next` unless you have taken the lock of `prev`. Think about this for a while. It could be an easy point on the exam.

```
void toggle(cell *lst, int r) {
  /* This is ok since we know there are at least two cells */
  cell *prev = lst;
  pthread_mutex_lock(&prev->mutex);
  cell *this = prev->next;
  pthread_mutex_lock(&this->mutex);

  cell *removed = NULL;
```

Once we have two locks, we can run down the list to find the position we are looking for. As we move forward, we release a lock behind us and grab the one in front of us.

```
  while(this->val < r) {
    pthread_mutex_unlock(&prev->mutex);
    prev = this;
    this = this->next;
    pthread_mutex_lock(&this->mutex);
  }
```

The code for when we have found the location is almost identical. The only difference is that we have to initialize the mutex.

```
  if(this->val == r) {
    prev->next = this->next;
    removed = this;
  } else {
    cell *new = malloc(sizeof(cell));
    new->val = r;
    new->next = this;
    pthread_mutex_init(&new->mutex, NULL);
    prev->next = new;
  }
```

Finally, we release the two locks that we hold and then we are done.

```
  pthread_mutex_unlock(&prev->mutex);
  pthread_mutex_unlock(&this->mutex);
  if(removed != NULL) free(removed);
  return;
}
```

That was it! We are ready to take our marvel of concurrent programming for a spin.

5.2 do some timings

You are now excited to see if you have managed to improve the execution time of the implementation and to find out how to add some code to measure the time. We will use `clock_gettime()` to get a time-stamp before we start

the threads and another when all threads have finished. Modify both the file `list.c` and `clist.c`. We include the header file `time.h` and add the following code in the `main()` procedure just before creating the threads.

```
struct timespec t_start, t_stop;
clock_gettime(CLOCK_MONOTONIC_COARSE, &t_start);
```

After the segment where all threads have finished, we take the next time stamp and calculate the execution time in milliseconds.

```
clock_gettime(CLOCK_MONOTONIC_COARSE, &t_stop);

long wall_sec = t_stop.tv_sec - t_start.tv_sec;
long wall_nsec = t_stop.tv_nsec - t_start.tv_nsec;
long wall_msec = (wall_sec *1000) + (wall_nsec / 1000000);

printf("done in %ld ms\n", wall_msec);
```

Compile both files and run the benchmarks using, for example, 10 million entries and one to four threads.

5.3 The worst time in your life

This is probably the worst moment in your life - you have successfully completed the implementation of concurrent operations on a list only to discover that it takes more time compared to the stupid solution - what is going on?

If we first look at the single-thread performance, we have a severe penalty since we have to take one mutex lock per cell and not one for the whole list. If we have a list that is approximately 50 elements long, this will mean, on average, 25 locks for each toggle operation.

If we look at the multithreaded execution, the problem is that although we allow several threads to run down the list, they will constantly run into each other and ask the operating system to yield. When they are rescheduled, they might continue a few steps until they run into the thread ahead of them and have to yield again.

If you want to feel better, you can try to increase the `MAX` value. This value sets the limit on the values we insert in the list. If this value is 100, we will have approximately 50 elements in the list (every time we select a new random number, we have a 50/50 chance of adding or removing it). If we change this to 10000, we will, on average, have 5000 elements in the list. This will allow threads to have a larger distance to each other and less frequently run into each other.

When you do a new set of benchmarks, you will see that the execution time increases as the `MAX` value is increased. This is because the list is longer, and the execution time is proportional to the length of the list. To have a

reasonable execution time, you can try to decrease the number of entries as you increase the maximum value. Will your concurrent implementation outperform the simple solution?

5.4 back to basics

Let's stop and think for a while. We have used pthread-mutex locks to implement our concurrent list - why? Because if a lock is held, the thread will suspend to allow other threads to be scheduled. For how long will a lock be held? If the list is n elements long, we will, on average, run-down $n/2$ elements before we find the position where we should toggle an entry. That means the thread in front of us that holds the lock only in 1 out of $n/2$ cases has reached its destination. It is thus very likely that the lock will be released in the next couple of instructions. If the thread in front of us has found its position, it deletes its target, and also this is done in practically no time. This means that in only 1 out of n cases will the lock be held until a thread has created a new cell, something that could take slightly more time (a call to `malloc()`).

This means that if the list has 50.000 cells, it is very likely that a held lock will be released the very next moment. Hmmm, what would happen if we used a spinlock? We know that spinlocks could consume resources if the lock they try to take is held by a thread that is doing some tough work or, in the worst case, has been suspended, but it might be worth the gamble, let's try. We make a new copy of our code and call it `slist.c` and make a few changes to work with the spinlock that we implemented in `swap.c`. Fix the cell, so it holds a `int` that we will use as the mutex variable and patch the initialization of `dummy` and `sentinel`.

```
typedef struct cell {
  int val;
  struct cell *next;
  int mutex;
} cell;

cell sentinel = {MAX, NULL, 0};
cell dummy = {-1, &sentinel, 0};
```

We then need the spinlock version of `try()`, `lock()` and `unlock()` that we copy from `swap.h`. We use the quick and dirty spinlock that does not use `sched_yield()` nor `futex_wait()`. We want to be as aggressive as possible. Patch the code of `toggle()` to use the new locks, and you should be ready to go. Better?

6 Summary

We have seen why we need locks in the first place, that locks have to be built using atomic swap operations, and that it could be nice to use something other than spinlocks. The Linux way of interacting with the operating system is using `futex`, but this is not portable. The POSIX way also provides more abstractions using `pthread` mutex.

We saw that a true concurrent implementation is a bit tricky and does not always pay off to try to parallelize an algorithm. If you want to speed things up, you could try to use spinlocks, but then you have to know what you are doing. A final thought could be that if you are given the task to speed up the concurrent access to a list of 100.000 elements, your first question should be - why do you not use a ....?