IT University of Copenhagen

Bachelor: Operating Systems and C, Handin for Group A

Multi-programming

Multithreaded Sum, Multithreaded FIFO buffer, Producer-consumer, Banker's Algorithm

Anders Edelbo Lillie
aedl@itu.dk

Andreas Bjørn Hassing Nielsen
abhn@itu.dk

Markus Thomsen
matho@itu.dk

October 26, 2016

Contents

1 Introduction and background

This paper reports a project regarding multi-programming in the C language, performed in October, 2016, in the ITU course "Operating Systems and C". The project is split in 4 tasks, each one addressing different aspects of threads and process synchronization. The purpose is to understand how POSIX threads, semaphores and mutex locks work, as well as how these are implemented in C on a Linux platform. The 4 tasks are briefly described below, and the complete project description is attached in appendix 7.1.

1. **Multi-threaded sum**
   Use threads to work concurrently on sum computations on a multi-core processor, yielding higher performance than when run on a single processor.

2. **Multi-threaded FIFO buffer as a linked list**
   Implement a linked list in C, which is thread safe for parallelism. A code skeleton for this problem is given beforehand, and lacks only the implementation of add- and remove functions of nodes in the list, as well as `mutex` locks.

3. **Producer-Consumer with bounded buffer**
   Implement the Producer-Consumer problem with `Pthread`s in C with the use of counting-semaphores.

4. **Banker's algorithm for deadlock prevention**
   Implement Banker's algorithm in C. Banker's algorithm is a deadlock prevention algorithm, which ensures that any process running in the system, that also states its maximum claim on available resources at entry, can never get more resources than what is available.

2 Problem analysis

In this section, it is discussed how solutions to the different tasks can be constructed, such that they fulfill the requirements listed in the formerly mentioned project description. Each task is discussed on its own in the subsections below.

Multi-threaded sum

This is a trivial exercise to get started with multi-threaded software development. The sum function to be run multi-threaded is: $\sum_{i=1}^{N} \sqrt{i}$.

The first part of the exercise requests that the multi-threaded version of the sum function runs faster than in a single thread. It is assumed that N should not be too small, otherwise the overhead of threading will overshadow the performance gains, thus a limit for N for the function to be run in multi-threaded mode should be found.

With sum being a function that depends on addition, the commutative law of addition can be used to split the work between multiple threads by taking a range of i's, computing the square root of each of them, and then summing them together as a single final operation, without worrying about the order of operations.

The second part of the exercise dictates that performance measurements must be made for the single- and multi-threaded sum functions. The `time` command could be used for process time measuring purposes. The size of N to be used for testing should range from around 1e+05 until the single-threaded function takes a substantial amount of time (around 30 seconds). The observed results in the range of N's can be used to create a speedup graph, comparing a single-threaded version with a multi-threaded one.

Multi-threaded FIFO buffer as a linked list

As a FIFO linked list inherently has a way of queuing items, there are not many alternatives as to how items should be added or removed. An add-function should make a link to the item as the last item in the list, and a remove-function should remove the link to the first item added, and preserve the list structure. The "tricky" part however is to preserve this list structure so the links will never be broken and there will be no null-pointers (except for the last element). The simplest way of doing this is to keep track of the list length. The add- and remove functions should thereby always do as described above, but check if the list is empty first, ensuring that the first and last elements are always known. This could obviously be implemented with if-else statements. In this case however, having a root note in the list, simplifies the operations, such that the only cases necessary to take into account are when removing an element and the list is empty or only contains one element. This way there is always a first and a last element, as a handle, but it is not represented as an actual element in the list.

The second aspect of the task is to make the list thread safe. Using `mutex` locks from the `pthread` library makes it somewhat easy to do this. A lock should make it impossible for race conditions to occur when adding and removing items to and from the FIFO list. One could simply introduce just one lock, which is requested as the first thing in these operations, and then release it again when finished. This would make all add/remove operations atomic, regardless of the number of threads. One could think two different locks for add and remove respectively would make the program more efficient, but this would introduce a race condition, when one thread adds an element, while another thread removes an element in the list.

Producer-Consumer with bounded buffer

The Producer-Consumer problem is a classic process synchronization problem. The problem is concerned with the sharing of memory between processes, which is rife with opportunity for deadlocks, starvation and memory overwrites. In the Producer-Consumer problem, one or more producer processes, produces resources for the consumption of one and more consumer processes. The processes run in parallel, which necessitates some kind of locked communication between the processes. To serve as a communication medium, some kind of bounded

buffer is used. The bounded buffer will store the products produced by the producer processes. In this case, the `Multi-threaded FIFO linked list` described earlier will be used. Two counting semaphores are to be employed, to keep consumer threads from consuming products from the bounded buffer, if it's empty and keep producer threads from producing if it's full. In this specific instance of the Producer-Consumer problem, there's also a need to limit the amount of products produced, to a parameter set when running the program. To solve this, some kind of locking mechanism is necessary. A mutex-lock can be employed to make decrementing the amount of products left to be produced atomic. Another solution could be to use a counting semaphore to keep track of the amount of products left to be produced, the threads calling it's `wait()`, each time they're ready to produce.

The program needs to be able to terminate when the last product has been consumed. This can be accomplished by a busy wait in the `main` function, letting the parent process return, as soon as the last product has been consumed. Another way to lock the `main` function, untill it's ready to return, is to initiate a counting semaphore with a count value equal double the amount of the products to be produced. The parent process would call the count semaphores `wait()` function, and the consumer and producer processes, would call it's `signal()` function every time a product is consumed or produced.

Banker's algorithm for deadlock prevention

Banker's algorithm ensures that no deadlock will ever happen (when implemented correctly). The algorithm enforces an equilibrium between requested resources, resources available and resources released from each process.

Banker's algorithm is described as a recipe in the course book[1], and a code skeleton with a proper order of operations has been handed out through the course page.

It is hinted, that starting out with implementing the function to check if a state is safe or not is a good idea. This makes sense, as it is one of the main components in the algorithm. When an `is_safe` function has been implemented, the input test files from the handout can be tested (See section 7.5.5 for these files).

From the project description, it is expected that the results for `input.txt` is for the state to be safe, and `input2.txt` should be "unstable". With these files, the validity of the `is_safe` function can be verified. Both will be safe, but `input2.txt` will be harder to generate requests for, as much of the resources will already be allocated at program start.

3 Examples

In this section, the functionality of the programs are described and illustrated through some examples.

Multi-threaded sum

This program's execution is simple, as it expects a single integer input and prints out the computed square sum. Three examples are shown below, illustrating the program run on a quad-core processor, which means, 4 threads work concurrently:

```
$ ./sqrtsum.out 10
sqrt sum = 22.47
$ ./sqrtsum.out 100
sqrt sum = 671.46
$ ./sqrtsum.out 100000000
sqrt sum = 666666671666.46
```

For comparison the same operations on a single thread is illustrated below:

[1] Silberschatz p. 330, 7.5.3 Banker's Algorithm

```
$ ./sqrtsum_single.out 10
sqrt sum = 22.47
$ ./sqrtsum_single.out 100
sqrt sum = 671.46
$ ./sqrtsum_single.out 100000000
sqrt sum = 666666671666.57
```

As can be seen above, `sqrtsum_single` deviates slightly from the correct result found in the multi-threaded version `sqrtsum` for $N = 100000000$, a qualified guess for the reason behind this can be found in section 4 where the performance difference of the two versions is also investigated.

Multi-threaded FIFO buffer as a linked list

The following sequence of operations

```
add(a)
add(b)
add(c)
remove(a)
add(d)
remove(b)
```

with 2 threads will do as illustrated in below terminal output. Above operations are written in pseudo-code for clarity, but corresponds to `node_new_str, list_add, list_remove`.
The terminal output depicts the linked list's structure after each operation. The order of nodes are annotated with letters, and the number is the thread, which created the node.

```
root -> ('a1') -> ('a2') -> end
root -> ('a1') -> ('a2') -> end
root -> ('a1') -> ('a2') -> ('b2') -> ('b1') -> end
root -> ('a1') -> ('a2') -> ('b2') -> ('b1') -> end
root -> ('a1') -> ('a2') -> ('b2') -> ('b1') -> ('c2') -> ('c1') -> end
root -> ('a1') -> ('a2') -> ('b2') -> ('b1') -> ('c2') -> ('c1') -> end
root -> ('b2') -> ('b1') -> ('c2') -> ('c1') -> end
root -> ('b2') -> ('b1') -> ('c2') -> ('c1') -> end
root -> ('b2') -> ('b1') -> ('c2') -> ('c1') -> ('d2') -> ('d1') -> end
root -> ('b2') -> ('b1') -> ('c2') -> ('c1') -> ('d2') -> ('d1') -> end
root -> ('c2') -> ('c1') -> ('d2') -> ('d1') -> end
root -> ('c2') -> ('c1') -> ('d2') -> ('d1') -> end
List length: 4
Final structure:
root -> ('c2') -> ('c1') -> ('d2') -> ('d1') -> end
```

As can be seen above, the operations are executed atomically with no race conditions or unexpected behavior. Nodes are added to the end of the list and removed from the start, conforming to the FIFO-structure. The test is performed with `sleep(0.5)` in between the operations, as the sequence would otherwise be *more* random.

Producer-Consumer with bounded buffer

When running the program, it takes an input in the form of 4, non-positive integer values:

```
$ ./pcp.out "producers" "consumers" "buffer size" "products"
```

Where producers are the number of producer threads to run. Consumers are the number of consumer threads. Buffer size is the amount of products, the buffer can hold, and products is the number of products to produce. An example input could be:

```
$ ./pcp.out 3 2 4 6
```

Which would produce something like the following output:

```
Producers: 3
Consumers: 2
Buffer size: 4
Products: 6
Starting producer thread: 0
Starting producer thread: 1
Starting producer thread: 2
Starting consumer thread: 0
Starting consumer thread: 1
Producer 2 produced item 4867. Items in buffer: 1 out of 4.
Consumer 1 consumed item 4867. Items in buffer: 0 out of 4.
Producer 1 produced item 2222. Items in buffer: 1 out of 4.
Consumer 0 consumed item 2222. Items in buffer: 0 out of 4.
Producer 0 produced item 4344. Items in buffer: 1 out of 4.
Producer 1 produced item 7325. Items in buffer: 2 out of 4.
Producer 0 produced item 9055. Items in buffer: 3 out of 4.
Consumer 0 consumed item 4344. Items in buffer: 2 out of 4.
Consumer 1 consumed item 7325. Items in buffer: 1 out of 4.
Consumer 0 consumed item 9055. Items in buffer: 0 out of 4.
Producer 2 produced item 6528. Items in buffer: 1 out of 4.
Consumer 1 consumed item 6528. Items in buffer: 0 out of 4.
Exiting program.
```

If the program is run with too few parameters, it will produce the following error message:

```
Error: Wrong amount of parameters.
Example usage: ./pcp.out <n_producers> <n_comsumers> <buffer_size> <n_products>
```

If the program is run with non-positive parameters, it will produce the following error message:

```
Producers: 3
Consumers: 2
Buffer size: 4
Products: -1
Error: Non-positive input parameters.
```

Banker's algorithm for deadlock handling

To run Banker's algorithm, you need to define m: the amount of threads to run, n: the amount of distinct resources the system contains, then a resource vector, defining how much of each resources that can be allocated at any given time, a maximum matrix which defines the maximum amount of resources each process can hold and a allocation matrix, to start the system in a particular state, or zeroed if you wish to start the program in an empty state.

A sample of using the program, using a file as input, is seen below:

```
$ ./banker.out < input.txt
Number of processes: Number of resources: Resource vector: Enter max matrix: Enter allocation matrix:
Need matrix:
R1 R2 R3
3  2  2
6  1  3
3  1  4
4  2  2
Availability vector:
R1 R2 R3
9  3  6
The initial state is safe!
Process 0: Requesting resources.
Process 1: Requesting resources.
Process 3: Requesting resources.
Process 0: Releasing resources.
...
```

The first line in the example above looks weird, but this is because the program can also be run without a setup file, which requires user input to the command line. See an example of a such a usage below:

```
$ ./banker.out
Number of processes: 4
Number of resources: 3
Resource vector: 1 2 3
Enter max matrix:
1 0 0
1 2 1
0 1 3
0 1 1
Enter allocation matrix:
0 0 0
0 0 0
0 0 0
0 0 0
Need matrix:
R1 R2 R3
1  0  0
1  2  1
0  1  3
0  1  1
Availability vector:
R1 R2 R3
1  2  3
The initial state is safe!
Process 0: Requesting resources.
Process 1: Requesting resources.
Process 2: Requesting resources.
Process 0: Releasing resources.
...
```

4 Technical description of the program

In this section, the specified functionality is described with a technical description, both covering how they work, and how they are implemented in the source code.

4.1 Multi-threaded sum

The multi-threaded square root sum function is implemented without the use of a `mutex`.

Mutexes is avoided by dividing the work between threads, such that each thread can work in distinct parts of the memory, and will therefore never interfere with each other. Upon completion of all the work in each thread, the threads are joined. Then the main thread sums the results together and prints the final sum to the screen.

To properly explain how the work is subdivided between threads, some constants must be established; N is the max of the sum-range and P is the amount of processors available to the program. Each process with a idx receives work corresponding to its ID, thus summing from $N/P * idx + 1$ to $N/P * (idx + 1)$. For example if we have 3 processors and are trying to get $\sum_{i=1}^{1e+08} \sqrt{i}$:
$thread_{idx=0}$ will do the following work: $\sum_{i=1}^{33,333,333} \sqrt{i}$. Then $thread_{idx=1}$ will perform: $\sum_{i=33,333,334}^{66,666,666} \sqrt{i}$, and $thread_{idx=2}$ will do: $\sum_{i=66,666,667}^{100,000,000} \sqrt{i}$. The distribution of work in this division is not uniform, as the threads that receive the final amounts of work have to perform the square root function on larger values of i, thus resulting in more work for those threads. The extra work comes from the implementation of the function `sqrt` in `math.h`, which most likely is implemented by making guesses closer and closer to the actual value, and larger values of i requires more guessing.

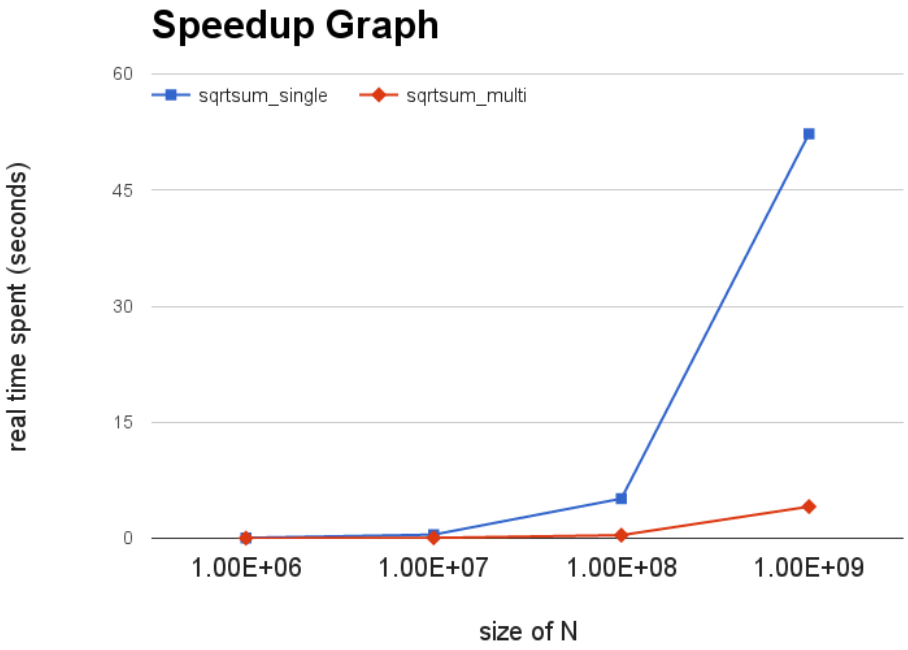
The threads are informed of their range via a struct that contains `from` and `to`, and the `id` of the thread.

The value 100 was selected for `MIN_N_MULTITHREADED`. This constant is added to avoid spending energy on subdividing tasks with small values of N, as the overhead of creating threads and separating the work would overshadow the performance gains from multi-threading.

As mentioned section 3, `sqrtsum_single` generates erroneous values above some high values of N (for instance, 100000000 generates 666666671666.57, which is off by 0.11 from the *more* correct value 666666671666.46). A qualified guess on why this happens, is that the sum is calculated with a long list of doubles that are added together one by one in one for loop, whereas with the multi-threaded version, the square root sums are split into several segments, and then summed together at the end, possibly eliminating some of the floating point precision errors.

See figure 4.1 for a performance comparison of running the square root sum function in a single thread, versus running it in several threads[2]. The graph states, as expected, that for larger values of N, the multi-threaded version outperforms the single-threaded one by far.

[2]In this case 8 threads, as it is run on a quad-core i7 CPU with hyper threading



4.2 FIFO-buffer

The linked list is constructed through the `list_new()` function, which initialized the "empty" root-node and sets the length counter to 0. This `struct` keeps track of the list length, the first element, and the last element. A node (or element) is created through either `node_new_str()`, `node_new_int()` or `node_new()`, and contains a pointer to the next node as well as a possible element(`void*`-type for generality). When a node is added to the list, the last node in the list (which is initially the root-element) is set to point to this, and the given node is assigned as being the last.

When a node is removed from the list, the root node is set to point to the second non-root node in the list, thereby "deleting" the link to the first node. If the list only contains one element the root node is assigned as being the last. If the list is empty, the function returns `NULL`. Both `list_add()` and `list_remove()` operations initially requests to lock the mutex contained within the given list `struct`, and releases it upon completion.

No race conditions can occur, which is depicted in figure 4.2 below. This image illustrates the sequence of two `list_add()` functions, each executed by a seperate thread. The left example is with no locks implemented, while the right on has the mutex lock described above implemented. Furthermore a `list_delete()` function is implemented to free up allocated memory used by the program. This, and a node removed by `list_remove()` is the caller's responsible to handle. A simple function `node_destroy()` is however also implemented to free memory of a single node.

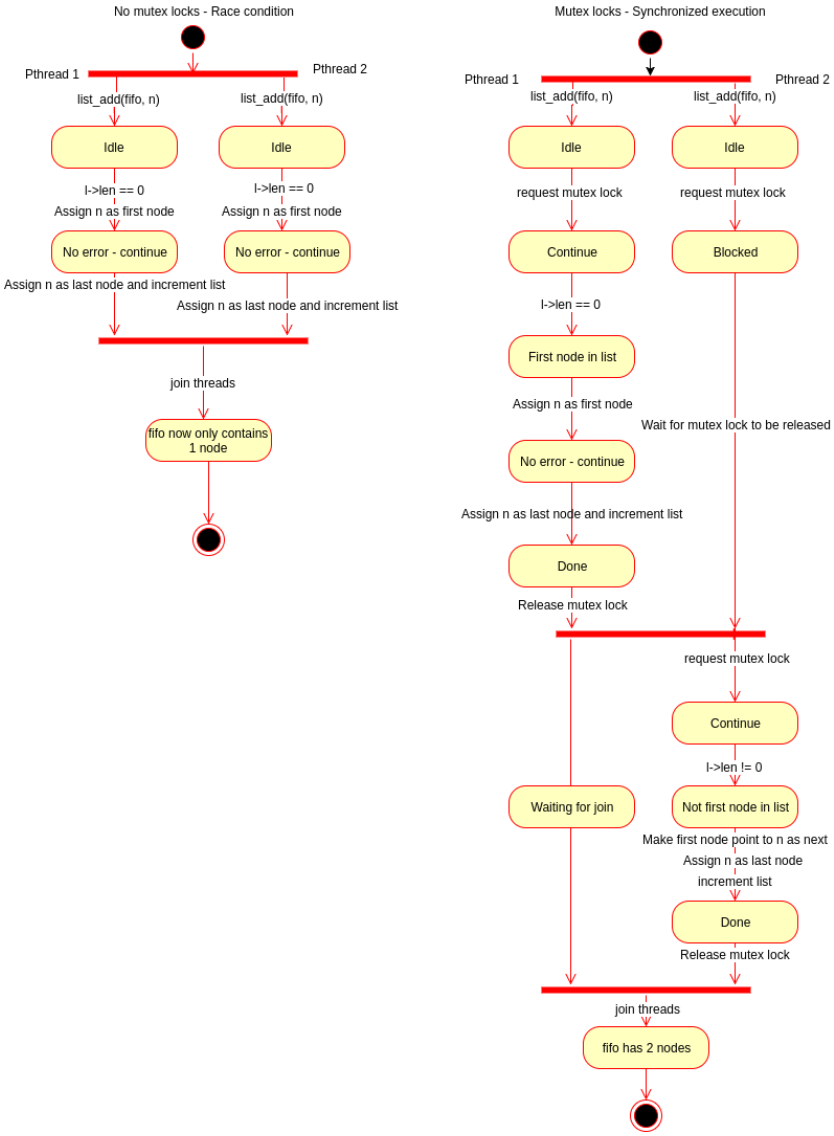


Figure 1: Illustration of a possible race condition (left) and thread-safe execution (right).

4.3 Producer-Consumer problem

The Program has three main components: The main process, the producer processes and the consumer processes. When the program is started, the main process first initiates the two counting semaphores.

It gives the `full` semaphore a value of 0, and the `empty` semaphore a value equal to the `buff_size` parameter. It proceeds to initiate the Producer POSIX threads, and Consumer POSIX threads, before entering a busy-wait while-loop. When the busy-wait releases. the process will return 0. The busy-wait will release as soon as the `products` variable equals 0, the counting value of the `full` semaphore equals 0, and the `empty` semaphore equals the buffer size.

The Producer threads call a lock function on the `product_lock` mutex. This is done to prevent edge cases, where the main thread might return, before the child threads a finished doing their work. After the lock call, the process checks if there remains any products to be produced, returning if there's none, and calling the `sem_wait(&empty)` function if there is. This semaphore will have the process wait, until there is room in the bounded buffer for a product. The process produces a random number, and inserts it into the thread-safe `buffer` variable. Once the Producer thread is finished producing, it signals the `full` semaphore, with the `sem_post(&full)` call. When the Producer thread is finished with it's critical work, or terminates, it will unlock the `product_lock` mutex.

The Consumer threads start by waiting on the full semaphore, by calling the `sem_wait(&full)`. The thread will wait for there to be a product for it to consume, at which point it will consume the product, and print the program state. Once done, the Consumer thread will signal the `empty` semaphore, with a `sem_post(&empty)` call. Following is a simplified version of the programs control flow, and how the different threads communicate.

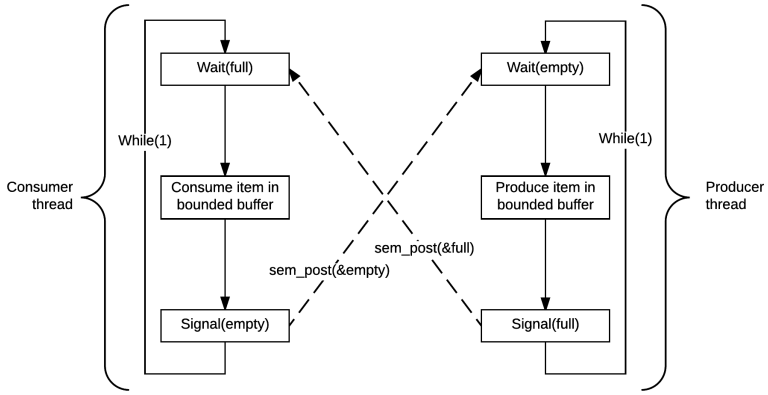


Figure 2: Simplified Illustration of how the Producer and Consumer threads communicate.

4.4 Banker's algorithm

Section 3 properly described how the program works, as well as how to run it. In this subsection it is described how the functionality is achieved in the implementation.

As mentioned in section 2, the algorithm is documented as a "recipe", which the implementation is heavily based upon. When the current state is given as input, the need- and availability matrices are calculated. When this is done, the safety-algorithm is run to check whether the initial state is safe or not. The `is_safe` function compares the available resources with the needed resources for every process, to make sure the requested resources do not exceed the actual available ones.

In the safety algorithm, it was decided to use a single `for` loop construct, that resets (sets it to -1, such that it becomes 0 on the next run) the index counter every time $Finish[i] == false \wedge Need_i <= Work$. This was done to avoid having to use an enclosing `while` loop, which would've further complicating the code.

In section 2 it was described that an error was found in the generation of requests and releases. The error specifically, is that the following code will be stuck in an endless loop if a process has a need- or allocation vector with a sum less than or equal to 0:

```
void generate_request(int i, int *request)
{
    int j, sum = 0;
    while (!sum) // we will never escape this, if sum(s->need[i]) <= 1
    {
        for (j = 0; j < resource_count; j++)
        {
            request[j] = s->need[i][j] * ((double)rand()) / (double)RAND_MAX;
            sum += request[j];
        }
    }
}
```

The error was corrected by using a modulus operation instead of the typecasting and divisions, see 7.5.1 for the specific implementation information.

A problem was found in the generation of the need matrix. When a process requests is allocated too many resources from the get-go, negative numbers will appear in the need matrix, and this cascades into a lot of fun times. The issue was solved by checking upon creation of the need matrix, by making sure no numbers in it would ever be less than 0.

5 Test

Having described how the programs could be implemented, their functionality through examples, as well as their technical descriptions, finally this section shows to what extend the programs solves the given problems.

The examples from section 3 are proper tests for functionality of the programs, as they make sure no deadlocks and race conditions occur, which a some of the major obstacles in multi-threaded programming.

The multi-threaded sum program is simple in the sense that not much can be tested other than several I/O use case tests, and assert this with expected computation results[3]. This has been done a number of times throughout the project with a various number of threads and sum limits. While testing the program, a flaw was found in the subdivision of work; integer division cut away information prior to a multiplication, which made the `from` and `to` variables receive incorrect values.

The FIFO-buffer is implemented as a linked list with adhering add- and remove functions as requested by problem 2.1[4]. Testing whether the list is thread-safe have been done throughout the project by running several sequences of add and remove functions by a number of threads and printing out the list structure to compare expected behavior with actual behavior. Whenever the program has crashed with `Segmentation fault` a primitive debugging has been conducted to isolate the issue.

The Producer-Consumer problem has been tested with the examples where it's given too few parameters, negative and zero parameters, and parameters with the wrong datatypes, all of these are handled gracefully. The only test that the program couldn't handle gracefully, is started with an unreasonable amount of threads, 4000+ in the case of the authors machine. We found out that unnamed semaphores and semaphore_getvalue, from the semaphore.h library, were deprecated on Darwin systems (OSX) by testing on multiple machines. This isn't important, as our program is only meant to be used on Linux systems, but it would have been relevant if our code was meant to be portable. This shows that the program is capable of handling a large amount of consumer and producer threads, albeit only on Linux based systems.

The Banker's algorithm implementation correctly works as a deadlock avoidance. Testing for deadlocks can be hard, and it is an assumption that the final implementation is a flawless version of the algorithm.

When running `banker.out`, if a deadlock has occurred, it will seem like as if the program has stalled on requesting or releasing resources, but this has yet to happen in the final version.

While testing the with random inputs, a bug in the handout code was found. The functions to generate resource requests and releases were faulty, in that they could end up in endless while loops, thus trapping the threads at arbitrary times depending on the request generated. We fixed this by updating the arithmetic in the two functions, for more information see section 4. The updated generate request and release functions have been published[5] on the LearnIT forums[6] for other students to grab, in case they got stuck on the same issue.

[3] `http://wolframalpha.com` was used to verify results from the program with what was expected.
[4] See appendix 7.1.
[5] `https://gist.github.com/AndreasHassing/a715548a21f299274fe1ab44cfbd228c`
[6] `https://learnit.itu.dk/mod/forum/discuss.php?d=11408#p32972`

6 Conclusion

Working with threads and synchronization in the C programming language can be rather cumbersome, which have been a topic throughout the project. The 4 tasks; sum, FIFO-buffer, Producer-Consumer problem, and Banker's algorithm have all been implemented with the required functionality from the project description (which can be seen in appendix 7.1).

The first two tasks, sum and FIFO-buffer, were clearly the tasks which were the easiest to get started with, as they did not encapsulate much functionality, and thereby not much to be implemented. Using the `pthread` library a lot of functionality and tools, including `mutex` locks, are given to the developer. The Producer-Consumer problem was rather trivial to implement the base functionality in, as there was a great project skeleton in the end of Silberchatz Chapter 5. However, keeping track of how many products was produced proved more difficult, as this information had to be shared between processes. Making the program terminate, after the last product had been consumed and printed, also proved difficult, and while the problem was solved, the solution seemed less than ideal. Preferably the Producer and Consumer threads would have terminated themselves, necessitating only a *join* call to each of the sub-threads from the parent, before it could terminate. The Banker's algorithm yielded a lot of trouble, in that bits and bytes of the code handout was seemingly broken. A benefit of this is that the debugging tool `lldb` has seen a lot more usage. Furthermore, it was assumed that the input from `input2.txt` should fail the initial state safety check, but this was a faulty assumption, and cost a lot of time.

Retrospective

In regards to the FIFO-buffer task, more time should have been used to actually understand why a root node was implemented for the list structure. It did not cause any problems, but would have saved some time in development. The program was furthermore initially implemented with two locks, but only discovered late why this could cause race conditions. More testing early on to discover this would have been optimal.

The development of the Producer-Consumer program could have benefited from a more incremental development approach, focusing on solving each smaller problem in order, instead of trying to solve them all at once.

In general, it would've been a great idea to sit down and run each algorithm by hand before implementing, to get a better idea of how, why and when they work. This piece of advice to our future selves, is greatly empowered by the amount of time spent, re-asserting over and over that the safety algorithm was properly implemented.

Extensions and improvements

Improvements to the FIFO-buffer would be to make it "clean up" after itself instead of making it the caller's responsibility. Having e.g. a list with 1 billion nodes and forgetting to free up memory would be a rather bad buffer-list. The Producer-Consumer program could have been extended, by allowing it to take a function as parameter in the Producer and Consumer function. With a little re-writing, the program could be turned into a library, for Producing and Consuming any kind of item. with the function passed to the Producer function, being the function capable of producing the item, and the function passed to the Consumer function, being the function meant for consuming the item.

7 Appendix

7.1 Project description

Intentional blank space, as a separate PDF file is included in the report, and begins on the next page.

IT-Universitetet i København 10. oktober 2016

Operativsystemer og C

Obligatorisk opgave 2

Rapporten - som **pdf-fil** - samt kildekode skal pakkes og uploades til learnit senest onsdag den 26. oktober, klokken 23:59

Denne obligatoriske opgave består af flere separate del-opgaver. Alle opgaverne skal afleveres og indgår som del af rapporten.

Baggrund

Målet med denne obligatoriske opgave er at forstå hvordan POSIX tråde, semaforer og mutex låse fungerer, samt hvordan disse implementeres i C på en Linux platform.

Historisk set, har det været hardwareproducenten som implementerede sine egne trådbiblioteker hvilket gjorde det svært at portere flertrådede programmer mellem forskellige platforme. For familien af UNIX operativsystemer, herunder Linux, findes der dog et stadardiseret portabelt trådbibliotek specificeret ved den såkaldte IEEE POSIX 1003.1c standard. Trådbiblioteker, som anvender denne standard, kaldes POSIX tråde (eller Pthreads) og er i C defineret i header filen `<pthread.h>`.

POSIX trådbiblioteket tilbyder funktioner for `pthread_t` trådobjekter. Fx til at skabe, samle og terminere findes følgende funktioner:

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine)(void *), void *arg);
int pthread_join(pthread_t thread, void **value_ptr);
void pthread_exit(void *value_ptr);
```

POSIX trådbiblioteket tilbyder også funktioner til synkronisering af tråde, så som mutex låse og betingelsesvariabler:

```
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr)
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex)
int pthread_cond_destroy(pthread_cond_t *cond);
```

Semaforer er også en del af POSIX 1003.1 standarden men defineres for sig i header filen `<semaphores.h>`. Semaforer er objekter af typen `sem_t` og anvendes fx gennem:

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
int sem_post(sem_t *sem);
int sem_getvalue(sem_t *sem, int *sval);
int sem_destroy(sem_t *sem);
```

Du kan bruge `man` til at få flere detaljer om alle de ovenstående funktioner.

Opgave 1. Multitrådet sum

Målet med denne opgave er, at du kan

- anvende tråde til at samarbejde om beregninger på en multicore processorer.

I Silberschatz på side 161 i 8. udgave eller side 171 i 9. udgave vises koden til et C program, der bruger Pthreads API til at starte en tråd som beregner sum-funktionen:

$$sum = \sum_{i=0}^{N} i.$$

Dette program illustrerer anvendelsen af Pthreads men har derudover ingen mærkbar effekt for brugeren af programmet.

1.1) Omskriv programmet sådan at det rent faktisk kører hurtigere (på en multicore maskine) end det ellers ville have gjort uden brug af tråde. Du skal dog gøre det for en funktion der (i stedet for *sum*) beregner summen af kvadratrødder:

$$sumsqrt = \sum_{i=0}^{N} \sqrt{i}.$$

Denne funktion tager lidt længere tid at beregne og kan gå til højere N uden overflow.

1.2) Du skal også måle udførselstiden af dit program ved forskellige antal tråde (f.eks. ved at bruge `time`) og lave en tilhørende speed-up graf. Brug f.eks. $N = 10000000$, der er tilpas stort til at beregningen tager ca. 1-5 sek. med én tråd.

Hints:

- Du skal starte flere tråde (1-2-4-8 etc.) til at samarbejde om beregningsarbejdet.
- Du skal desuden ændre fra at lave summen for integers (typen `int`) til at lave dem i flydende tal (typen `double`). Dette er nødvendigt da $\sqrt{i}$ ikke er et heltal.
- For at kunne bruge `sqrt()` skal du inkludere `<math.h>` samt linke med `-lm`.

- Du skal lave en `struct` med parametre til trådene, der fortæller hvilket arbejde tråden skal lave.
- Du kan med fordel antage, at N divideret med antallet af tråde er et heltal.
- Husk at sikre dig at din CPU har flere cores (`cat /proc/cpuinfo`). Du kan med fordel logge ind på `ssh.itu.dk` når du tester.

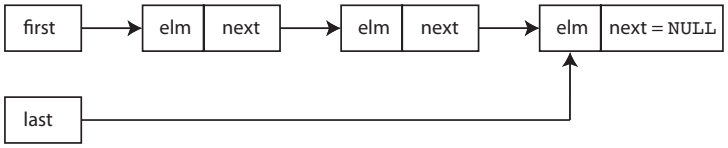
Opgave 2. Multitrådet FIFO buffer som kædet liste

Målet med denne opgave er, at du kan

- implementere en kædet liste i C.
- trådsikre dine programmer så de kan anvedes af flere tråde i parallel.

I denne opgave skal du implementere en FIFO buffer som en kædet liste i C til flertrådede programmer.

Kig på koden i de uploadede filer `list.c` og `list.h`, der viser en ufuldstændig udgave af en kædet liste i C uden brug af tråde.



2.1) Du skal starte med at færdiggøre den ufuldstændige 1-tråds udgave af en kædet liste i `list.c`, sådan at den kan anvendes som FIFO buffer. Bemærk, at bufferens indhold er af typen `void` for at den kan indeholde alle objekttyper (fx strenge).

Dette kræver, at du implementerer de to funktioner:

```
void list_add(List *l, Node *n);

Node *list_remove(List *l);
```

2.2) Antag nu, at vi anvender implementationen af en kædet liste i et program hvor flere tråde tilgår den samme liste med funktionerne implementeret i `list.c`.

Beskriv de problemer, der kan opstå.

2.3) Brug én eller flere mutex låse til at lave en version af `list.c`, der kan bruges i flertrådede programmer, samt et testprogram til at sikre at den virker.

Hints:

- Se kommentarerne i kildekoden for mere detaljerede beskrivelser.
- Bemærk, at der i listen altid indsættes et første element (her kaldes det `root`), som ikke må fjernes. Dvs. at `first` pointeren peger på dette første element og `first->next` peger på det første rigtige element indsat i listen (eller `NULL` hvis der ikke er noget). Når du fjerner et element skal du huske at overholde dette.

Opgave 3. Producer-Consumer med bounded buffer

Målet med denne opgave er, at du kan

- implementere Producer-Consumer problemet med Pthreads i C.
- anvende tælle-semaforer.

3.1) Implementér ved brug af POSIX tråde et producer-consumer system, som beskrevet i Silberschatz afsnit 6.7.1 i 9.. Programmet skal anvende en kædet liste fra forrige opgave til at realisere den delte buffer. Elementerne i bufferen kan for eksempel være tekststrenge. Som beskrevet i bogen skal bufferen have begrænset kapacitet, og man skal bruge to tælle-semaforer (`empty` og `full`) for at holde styr på hvor mange pladser der er frie, og hvor mange pladser der er i brug.

Når bufferen er tom skal consumer-trådene sættes til at vente indtil der kommer data i bufferen, og når den er fuld skal producer-trådene vente indtil der er blevet plads i bufferen til de nye data.

Dit producer-consumer program skal som minimum kunne følgende:

- Input (fx fra kommandolinjen) skal være antallet af producers, antallet af consumers, størrelsen af den delte buffer, samt antallet af produkter der ialt skal produceres.
- Hver producer eller consumer skal køre i sin egen tråd - der må kun være én producer-funktion og én consumer-funktion.
- Hver gang et produkt produceres eller konsumeres, skal tråden „sove“ i et tidsinterval med tilfældig længde. Til dette anvendes en tilfældighedsgenerator og funktionen `sleep()` fra `<sys/time.h>`.
- Programmet må ikke gå i baglås eller risikere at tråde udsultes.
- Output skal være information fra producers og consumers hver gang de producerer eller konsumerer et produkt samt hvor mange produkter der er i bufferen på dette tidspunkt. Fx noget i stil med:

```
[hbrs@cypher opg3]$ ./prodcons 6 6 10
Producer 5 produced Item_0. Items in buffer: 1 (out of 10).
Producer 2 produced Item_1. Items in buffer: 2 (out of 10).
Consumer 0 consumed Item_0. Items in buffer: 1 (out of 10).
Producer 0 produced Item_2. Items in buffer: 2 (out of 10).
Producer 1 produced Item_3. Items in buffer: 3 (out of 10).
Producer 3 produced Item_4. Items in buffer: 4 (out of 10).
Producer 0 produced Item_5. Items in buffer: 5 (out of 10).
Consumer 1 consumed Item_1. Items in buffer: 4 (out of 10).
Producer 2 produced Item_6. Items in buffer: 5 (out of 10).
```

```
Producer 4 produced Item_7. Items in buffer: 6 (out of 10).
Producer 5 produced Item_8. Items in buffer: 7 (out of 10).
Consumer 3 consumed Item_2. Items in buffer: 6 (out of 10).
Consumer 1 consumed Item_3. Items in buffer: 5 (out of 10).
Consumer 2 consumed Item_4. Items in buffer: 4 (out of 10).
...
```

- Sørg for at programmet afslutter når alle produkter er produceret og konsumeret (dette kan med fordel implementeres til sidst).

Hints:

- Et opgave der ligner - dog med en anderledes implementeret bufffer - er Silberschatz s. 303 projekt 3.
- En simpel måde til at få en tråd til at sove i et tilfældigt tidsrum kan skrives:

```
/* Random sleep function */
void Sleep(float wait_time_ms)
{
  wait_time_ms = ((float)rand())*wait_time_ms / (float)RAND_MAX;
  usleep((int) (wait_time_ms * 1e3f)); // convert from ms to us
}
```

  Dvs. at `Sleep(1000)` vil sove i gennemsnit 1 sekund.

  Da `rand()` er en pseudo-tilfældighedsgenerator, skal den have et skiftende seed:

```
// seed the random number generator
struct timeval tv;
gettimeofday(&tv, NULL);
srand(tv.tv_usec);
```

  (ellers giver den de samme „tilfældige“ tal hver gang programmet køres).

Opgave 4. Banker's algorithm til håndtering af deadlock

Opgaven er baseret på Silberschatz s. 339 Banker's Algorithm Projekt med pthreads.

Der uploadet en ufuldstændig programkode `banker.c`, som kan bruges som udgangspunkt. Koden er baseret på tilstandsstrukturen (gennemgået ved forelæsningen);

```
typedef struct state {
  int *resource;
  int *available;
  int **max;
```

```
  int **allocation;
  int **need;
} State;
```

bestående af to vektorer (`resource` og `available`) og tre matricer (`max`, `allocation` og `need`). Disse er også beskrevet i lærebogen afsnit 7.5.

Begyndelsestilstanden specifices ved hjælp af en input `.txt` fil, fx givet ved formatet

```
<m>
<n>

<resource vector>

<max matrix>

<allocation matrix>
```

som så inlæses via stdin når programmet køres med `./banker < input.txt`. Her er `m` antallet af processer og `n` antallet af resurser.

Et eksempel på `input.txt`:

```
4
3

9 3 6

3 2 2
6 1 3
3 1 4
4 2 2

0 0 0
0 0 0
0 0 0
0 0 0
```

Værdierne for `<allocation matrix>` kan sættes forskellig fra 0 for at afprøve koden i bestemte kritiske situationer (prøv f.eks. `input2.txt`, som er den „ustabile“ tilstand vi så på til forelæsningen).

Efter at input er indlæst, vil den ufuldstændige kode `banker.c` starte m tråde, der simulerer de m processer, og skiftevis forespørger på resurser og frigiver resurser uden at afslutte. Antallet af resurser der forespørges eller frigives er implementeret tilfældige. Det gøres ved at sikre sig, at der på intet tidspunkt frigives flere resurser end allokeret til processen eller forespørges på flere resurser end der maksimalt er tilladt for processen.

En forespørgsel fra en proces skal kun tildeles hvis tildelingen vil resultere i en sikker tilstand. Til dette skal banker's algoritme implementeres. Ellers skal processen vente til senere (`Sleep(100)`) og prøve igen med samme forespørgsel.

Du skal (som minimum) implementere følgende punkter i den ufuldstændige kode:

- Allokere hukommelse dynamisk til de vektorer og matricer, der skal anvendes.
- Banker's safety-algoritme bruges til at afgøre om en tilstand er sikker eller ej og dermed om en forespørgsel skal tildeles eller ej. Dette gøres i funktionen;

  ```
  int resource_request(int i, int *request) {}
  ```

- Frigivelse af resourcer skal implementeres i:

  ```
  void resource_release(int i, int *request) {}
  ```

- Du skal sikre dig at begyndelsestilstanden er sikker før trådene startes.
- Du skal sikre dig at al tilgang til delt hukommelse undgår race conditions.

Hints:

- En måde at allokere matricer i C er vist i eksemplet med matrix multiplikation.
- Begynd med at lave en funktion, der tager en tilstand som input og returnerer om den er sikker eller usikker.
- Du kan med fordel udskrive `available` vektoren hver gang den ændres.

Rapport

Du skal igen lave en fuld rapport der kan læses alene. I rapporten skal du beskrive hvordan du har implementeret punkterne der spørges til i opgaverne. Heruden skal du redegøre for hvordan du har testet at disse fungerer efter hensigten. Al relevant kildekode (+evt. Makefile) skal inkluderes i et appendix.

7.2 Code: Multi-threaded sum

7.2.1 sqrtsum.c

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

double *sqrtsums;          // this data is shared by the thread(s)
void *runner(void *param); // threads call this function
struct threadargs {
        int index;
        int from;
        int to;
};

// The lowest N, for which the program will be run multithreaded.
//
// If N is very low, the overhead of running the sqrtsum function
// multithreaded will be significant, and will result in loss of performance.
// If N is very high, running the sqrtsum function in a single thread
// will perform worse than when run with multiple threads.
// We picked 100, it is a good number, it is also a decent hipshot.
#define MIN_N_MULTITHREADED 100

// calculates `sum sqrt(i), i=a to b`
double sqrt_sum_from_to(int a, int b);
// calculates `sum sqrt(i), i=1 to n`
double sqrt_sum_of(int n);

// calculates the sum of an array of doubles
double sum_of_doubles(double doubles[], int arraysize);

// prints the sum result to the console
void print_result(double result);

int main(int argc, char *argv[]) {
        pthread_attr_t attr;      // set of thread attributes
        pthread_attr_init(&attr); // get the default attributes

        int processorcount = sysconf(_SC_NPROCESSORS_ONLN);
        sqrtsums = malloc(sizeof(double) * processorcount);

        if (argc != 2) {
                fprintf(stderr, "usage: %s <integer value>\n", argv[0]);
                return -1;
        }
        if (atoi(argv[1]) < 0) {
```

```
                fprintf(stderr, "%d must be >= 0\n", atoi(argv[1]));
                return -1;
        }

        int N = atoi(argv[1]);
        if (N < MIN_N_MULTITHREADED) {
                // we do not want to go multithreaded on small N's,
                // see the comment at the definition of MIN_N_MULTITHREADED
                print_result(sqrt_sum_of(N));
                return 0;
        }

        pthread_t tid[processorcount]; // thread identifiers
        struct threadargs args[processorcount]; // args for the threads
        for (int i = 0; i < processorcount; i++) {
                // should also work for odd numbered processorcounts
                args[i] = (struct threadargs) {
                        .index = i,
                        // move multiplicative operands to the numerator to avoid
                        // that the integer division cuts away too much information
                        .from = (N * i) / processorcount + 1,
                        .to = (N * (i+1)) / processorcount
                };

                // create the thread
                pthread_create(&tid[i], &attr, runner, &args[i]);
        }

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

        print_result(sum_of_doubles(sqrtsums, processorcount));

        return 0;
}

void *runner(void *param) {
        struct threadargs *args = (struct threadargs*)param;
        sqrtsums[args->index] = 0;

        for (int i = args->from; i <= args->to; i++) {
                sqrtsums[args->index] += sqrt(i);
        }

        pthread_exit(0);
}
```

```
double sqrt_sum_of(int n) {
        return (sqrt_sum_from_to(1, n));
}

double sqrt_sum_from_to(int a, int b) {
        double sqrtsum = 0;
        for (int i = a; i <= b; i++) {
                sqrtsum += sqrt((double)i);
        }

        return sqrtsum;
}

double sum_of_doubles(double doubles[], int arraysize) {
        double sum = 0;
        for (int i = 0; i < arraysize; i++) {
                sum += doubles[i];
        }

        return sum;
}

void print_result(double result) {
        printf("sqrt sum = %.2f\n", result);
}
```

7.2.2 sqrtsum_single.c

```c
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
        if (argc != 2) {
                fprintf(stderr, "usage: %s <integer value>\n", argv[0]);
                return -1;
        }
        if (atoi(argv[1]) < 0) {
                fprintf(stderr, "%d must be >= 0\n", atoi(argv[1]));
                return -1;
        }

        double sqrtsum = 0;
        for (int i = 0; i <= atoi(argv[1]); i++) {
                sqrtsum += sqrt(i);
        }

        printf("sqrt sum = %.2f\n", sqrtsum);
}
```

7.2.3 Makefile

```
all: sqrtsum sqrtsum_single

UNAME := $(shell uname)

# Compiler arguments
CC=gcc
CCFLAGS=-Wall -Werror

# Linker arguments
LDFLAGS=
LDLIBS=-lm

ifneq (£(UNAME), Darwin)
        LDFLAGS=-pthread
endif

sqrtsum: sqrtsum.o
        $(CC) $(CCFLAGS) $(LDFLAGS) -o $@.out sqrtsum.o $(LDLIBS)

sqrtsum.o: sqrtsum.c
        $(CC) $(CCFLAGS) -c sqrtsum.c

sqrtsum_single: sqrtsum_single.o
        $(CC) $(CCFLAGS) -o $@.out sqrtsum_single.o $(LDLIBS)

sqrtsum_single.o: sqrtsum_single.c
        $(CC) $(CCFLAGS) -c sqrtsum_single.c

.PHONY: clean

clean:
        rm -rf *.o *.gch *.out

debug: CCFLAGS += -DDEBUG -g
debug: clean all
```

7.3 Code: FIFO-buffer

7.3.1 list.h

```
/*****************************************************************************
list.h

Header file with definition of a simple linked list.

*****************************************************************************/
#include <pthread.h>

#ifndef _LIST_H
#define _LIST_H

/* structures */
typedef struct node {
        void *elm; /* use void type for generality; we cast the element's type to void type */
        struct node *next;
} Node;

typedef struct list {
        int len;
        Node *first;
        Node *last;
        pthread_mutex_t mutex;
} List;

/* functions */
List *list_new(void);            /* return a new list structure */
void list_delete(List *l);       /* delete a list structure */
void list_add(List *l, Node *n); /* add node n to list l as the last element */
Node *list_remove(List *l);      /* remove and return the first element from list l*/
Node *node_new(void);            /* return a new node structure */
Node *node_new_str(char *s);     /* return a new node structure, where elm points to new copy of string
Node *node_new_int(int *num);    /* return a new node structure, where elm points to a copy of  num */
void node_destroy(Node *n);      /* destroy a node and reclaim memory  */
void print_structure(List *l);   /* Print graphical structure of list to stdout */

#endif
```

7.3.2 list.c

```
/*****************************************************************************
list.c

Implementation of simple linked list defined in list.h.

*****************************************************************************/

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

#include "list.h"

/* list_new: return a new list structure */
List *list_new(void)
{
        List *l;

        l = (List *) malloc(sizeof(List));
        l->len = 0;

        /* insert root element which should never be removed */
        l->first = l->last = (Node *) malloc(sizeof(Node));
        l->first->elm = NULL;
        l->first->next = NULL;
        return l;
}

/* Clean up */
void list_delete(List *l) {
        Node *n;
        n = l->first;
        while (n != NULL) {
                free(n->elm);
                Node *next;
                next = n->next;
                free(n);
                n = next;
        }
        free(l);
}

/* list_add: add node n to list l as the last element */
void list_add(List *l, Node *n)
{
        pthread_mutex_lock(&(l->mutex));
```

```
        l->last->next = n;
        l->last = n; //Assign new node as last in list
        l->len++;
        pthread_mutex_unlock(&(l->mutex));
}

/* list_remove: remove and return the first (non-root) element from list l */
Node *list_remove(List *l)
{
        Node *n;
        if (l->len == 0) {
                printf("The list is empty. \n");
                return NULL;
        } else {
                pthread_mutex_lock(&(l->mutex));
                n = l->first->next; //The first (non-root) node
                if (l->len == 1) {
                        /* Empty the list */
                        l->last = l->first; //Root node
                }
                l->first->next = n->next; //Point to NULL if len == 1
                l->len--;
                pthread_mutex_unlock(&(l->mutex));
        }
        return n; //Memory allocated for this should be freed by caller
}

/* Free memory for a node */
void node_destroy(Node *n) {
        free(n->elm);
        free(n);
}

/* node_new: return a new node structure */
Node *node_new(void)
{
        Node *n;
        n = (Node *) malloc(sizeof(Node));
        n->elm = NULL;
        n->next = NULL;
        return n;
}

/* node_new_str: return a new node structure, where elm points to new copy of s */
Node *node_new_str(char *s)
{
        Node *n;
        n = (Node *) malloc(sizeof(Node));
```

```
        n->elm = (void *) malloc((strlen(s)+1) * sizeof(char));
        strcpy((char *) n->elm, s);
        n->next = NULL;
        return n;
}

/* node_new_str: return a new node structure, where elm points a to an integer num */
Node *node_new_int(int *num)
{
        Node *n;
        n = (Node *) malloc(sizeof(Node));
        n->elm = (void *) malloc(sizeof(int*));
        n->elm = num;
        n->next = NULL;
        return n;
}

/* Prints the structure of the FIFO linked list */
void print_structure(List *l) {
        Node *n;
        for (int i = 1; i <= l->len; i++) {
                if (i == 1) {
                        n = l->first->next;
                        printf("root -> ");
                } else {
                        n = n->next;
                }
                printf("('%s') -> ", (char*) n->elm);
                if (i == l->len) {
                        printf("end\n");
                }
        }
}
```

7.3.3 main.c

```
/*****************************************************************************
main.c

Implementation of a simple FIFO buffer as a linked list defined in list.h.

*****************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include "list.h"

#define N_THREADS 2

// FIFO list;
List *fifo;

// Threads
pthread_t tid[N_THREADS];

/* Test the synchronization of threads, using sleep
to avoid one thread to finish its job instantly */
void* test(void* args) {
        /* Convert thread number to char */
        int i = (int)(long)args;
        char tid[3] = {'a', i+'0', '\0'}; //"String"

        sleep(0.5);
        list_add(fifo, node_new_str(tid));
        sleep(0.5);
        print_structure(fifo);
        sleep(0.5);
        tid[0] = 'b';
        list_add(fifo, node_new_str(tid));
        sleep(0.5);
        print_structure(fifo);
        sleep(0.5);
        tid[0] = 'c';
        list_add(fifo, node_new_str(tid));
        sleep(0.5);
        print_structure(fifo);
        sleep(0.5);
        list_remove(fifo);
        sleep(0.5);
        print_structure(fifo);
        sleep(0.5);
```

```
        tid[0] = 'd';
        list_add(fifo, node_new_str(tid));
        sleep(0.5);
        print_structure(fifo);
        sleep(0.5);
        list_remove(fifo);
        sleep(0.5);
        print_structure(fifo);
}

int main(int argc, char* argv[])
{
        fifo = list_new();

        /* Start worker threads */
        int i = 0;
        while (i < N_THREADS) {
                pthread_create(&(tid[i]), NULL, &test, (void *)(long)++i);
        }

        /* Wait for threads to finish work */
        int j = 0;
        while (j < N_THREADS) {
                pthread_join(tid[j], NULL);
                j++;
        }

        printf("List length: %i\nFinal structure:\n", fifo->len);
        print_structure(fifo);

        /* Clean up */
        list_delete(fifo);

        return 0;
}
```

7.3.4 Makefile

```
all: fifo

CC=gcc
OBJS = list.o
LIBS= -pthread

fifo: list.o
	gcc -o $@ main.c ${OBJS} $(LIBS) -std=c99

list.o:
	$(CC) -c list.h list.c ${LIBS}

clean:
	rm -rf *o fifo
```

7.4 Code: Producer-Consumer problem

7.4.1 main.c

```c
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <unistd.h>

#include "../list/list.h"

sem_t full;
sem_t empty;
List *buffer;
pthread_mutex_t product_lock;
int consumers;
int producers;
int buff_size;
int products;

/* Random Sleep function from assignment.*/
void Sleep(float wait_time_ms)
{
        wait_time_ms = ((float)rand())*wait_time_ms / (float)RAND_MAX;
        usleep((int) (wait_time_ms * 1e3f));
}

/* init_semaphores: Set full semaphore to 0 and empty semaphore to buffer size.*/
void init_semaphores(int buffer_size)
{
        sem_init(&full, 1, 0);
        sem_init(&empty, 1, buffer_size);
}

/* insert_item: Insert integer Item into the FIFO buffer.*/
void insert_item(int item)
{
        int* item_pointer = malloc(sizeof(int*));
        *item_pointer = item;
        list_add(buffer, node_new_int(item_pointer));
}

/*remove_item: Remove first item from the FIFO buffer.*/
void remove_item(int *item)
{
        Node *n = list_remove(buffer);
        *item = *(int*)(n->elm);
}
```

```
/*producer: Function run by producer threads. Will keep producing items,
                        and inserting them in the buffer, as long as the buffer isn't full
                        and it hasn't run out of products.*/
void* producer(void *arg)
{
        int id = *(int*)arg;
        while(1)
        {
                //Lock the global variabel check and decrement.
                pthread_mutex_lock(&product_lock);
                if(products > 0)
                {
                        products--;
                        int item;

                        //Produce products. The thread remains lock to avoid edge case pre-mature termi
                        item = rand() % 10000;
                        sem_wait(&empty); //Wait for space to produce, if buffer is full.
                        insert_item(item);

                        //Record amount of products in full semaphore
                        int buffer_val;
                        sem_getvalue(&full, &buffer_val);
                        pthread_mutex_unlock(&product_lock);

                        //Print program state and signal full semaphore
                        printf("Producer %d produced item %d. Items in buffer: %d out of %d.\n",
                                id, item, buffer_val+1, buff_size);
                        sem_post(&full);
                        Sleep(500.0);
                } else {
                        //If there are no products left, we still need to unlock the mutex.
                        pthread_mutex_unlock(&product_lock);
                        return NULL;
                }
        }
}

/*consumer: Function run by consumer threads. Will keep consuming items,
                        removing them from the FIFO buffer, as long as it contains products.*/
void* consumer(void *arg)
{
        int id = *(int*)arg;
        while(1)
        {
                //Wait for there to be consumable items.
                sem_wait(&full);
```

```
                //Consume items
                int item;
                remove_item(&item);
                int buffer_val;

                //Signal empty semaphore and print thread status.
                sem_getvalue(&full, &buffer_val);
                printf("Consumer %d consumed item %d. Items in buffer: %d out of %d.\n",
                        id, item, buffer_val, buff_size);
                sem_post(&empty);
                Sleep(500.0);
   }
}

/*main: Initiaites variables, and starts producer and consumer threads.*/
int main(int argc, char *argv[])
{
        setbuf(stdout,NULL);
        //Seeding random generator
        struct timeval tv;
        gettimeofday(&tv, NULL);
        srand(tv.tv_usec);

        if(argc != 5)
        {
                printf("Error: Wrong amount of parameters.\n");
                printf("Example usage: %s <n_producers> <n_comsumers> <buffer_size> <n_products>\n", ar
                exit(EXIT_FAILURE);
        }

        producers = atoi(argv[1]);
        printf("Producers: %d\n", producers);
        consumers = atoi(argv[2]);
        printf("Consumers: %d\n", consumers);
        buff_size = atoi(argv[3]);
        printf("Buffer size: %d\n", buff_size);
        products = atoi(argv[4]);
        printf("Products: %d\n", products);
        if( (producers < 1) || (consumers < 1) || (buff_size < 1) || (products < 1))
        {
                printf("Error: Non-positive input parameters.\n");
                exit(EXIT_FAILURE);
        }

        //Initiating buffer
        buffer = list_new();

        //Initiating semaphores
```

```
        init_semaphores(buff_size);

        //Start producer and consumer threads.
        pthread_t producer_threads[producers];
        pthread_t consumer_threads[consumers];
        for(int index = 0; index < producers; ++index)
        {
                printf("Starting producer thread: %d\n", index);
                int *id = malloc(sizeof(int*));
                *id = index;
                pthread_create(&producer_threads[index], NULL, producer, (void*) id);
        }

        for(int index = 0; index < consumers; ++index)
        {
                printf("Starting consumer thread: %d\n", index);
                int *id = malloc(sizeof(int*));
                *id = index;
                pthread_create(&consumer_threads[index], NULL, consumer, (void*) id);
        }

        //Once every product is produced and the buffer is empty, the program terminates
        int full_val = 1;
        int empty_val = 0;
        while(products > 0 || full_val > 0 || empty_val < buff_size){
                Sleep(200.0);
                sem_getvalue(&full, &full_val);
                sem_getvalue(&empty, &empty_val);
        }
        printf("Exiting program.\n");
        return 0;
}
```

7.4.2 Makefile

```
all: pcp

CC=gcc
OBJS = list.o
LIBS= -pthread

pcp: list.o
	gcc -o $@.out main.c -Wno-deprecated-declarations ${OBJS} $(LIBS) -std=c99

list.o: ../list/list.h ../list/list.c
	$(CC) -c ../list/list.h ../list/list.c ${LIBS} -std=c99

clean:
	rm -rf *.out *.gch *o pcp
```

7.5 Code: Banker's algorithm

7.5.1 banker.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <unistd.h> // for usleep
#include <string.h> // for memcpy
#include <pthread.h>

#include "array_helpers.h"

typedef struct state
{
        int *resource;
        int *available;
        int **max;
        int **allocation;
        int **need;
} State;

// Global variables
int process_count, resource_count;
State *s = NULL;

// Mutex for access to state.
pthread_mutex_t state_mutex;

// Run the Safety Algorithm on the state `s`
int is_safe() {
        int safe = 1; // assume the state is safe, then try to disprove it

        int *work = malloc(resource_count * sizeof(int));
        memcpy(work, s->available, resource_count * sizeof(int));
        // use calloc for `finish`, to initialize elements
        // in the array to `0` which is also `False`
        int *finish = calloc(process_count, sizeof(int));

        for (int i = 0; i < process_count; i++) {
                if (!finish[i] && array_lte(s->need[i], work, resource_count)) {
                        array_add(work, s->allocation[i], resource_count);
                        finish[i] = 1;
                        i = -1; // reset the index to start from 0 on the next run
                }
        }

        for (int i = 0; i < process_count; i++) {
                if (!finish[i]) { safe = 0; break; }
        }
```

```
        free(work);
        free(finish);

        return safe;
}

/* Random sleep function */
void Sleep(float wait_time_ms)
{
        // add randomness
        wait_time_ms = ((float)rand()) * wait_time_ms / (float)RAND_MAX;
        usleep((int)(wait_time_ms * 1e3f)); // convert from ms to us
}

// Run a request against the state, deducting
// the request from available resources, adding
// the request to allocation for the process and
// deducting the request from the needed resources
// of the process.
void run_request(int i, int *request) {
        array_subtract(s->available, request, resource_count);
        array_add(s->allocation[i], request, resource_count);
        array_subtract(s->need[i], request, resource_count);
}

// Reverts a request. See the documentation for `run_request`,
// and revert the process horizontally and vertically.
void revert_request(int i, int *request) {
        array_add(s->need[i], request, resource_count);
        array_subtract(s->allocation[i], request, resource_count);
        array_add(s->available, request, resource_count);
}

/* Allocate resources in request for process i, only if it
   results in a safe state and return 1, else return 0 */
int resource_request(int i, int *request)
{
        if (!array_lte(request, s->need[i], resource_count)) {
                printf("Process %d has exceeded its maximum claim.\n", i);
                exit(EXIT_FAILURE);
        }
        pthread_mutex_lock(&state_mutex);
        if (!array_lte(request, s->available, resource_count)) {
                pthread_mutex_unlock(&state_mutex);
                return 0; // we can't request those resources yet!
        }
```

```
        run_request(i, request);

        int state_is_safe = is_safe();

        if (!state_is_safe) {
                revert_request(i, request);
#ifdef DEBUG
                printf("Process %d: request is unsafe.\n  request: [%d %d %d]\n",
                    i, request[0], request[1], request[2]);
#endif
        }

        pthread_mutex_unlock(&state_mutex);
        return state_is_safe;
}

/* Release the resources in request for process i */
void resource_release(int i, int *request)
{
        pthread_mutex_lock(&state_mutex);
        revert_request(i, request);
        pthread_mutex_unlock(&state_mutex);
}

/* Generate a request vector */
void generate_request(int i, int *request)
{
        int j, sum = 0;
        while (!sum)
        {
                for (j = 0; j < resource_count; j++)
                {
                        request[j] = rand() % (s->need[i][j] + 1);
                        sum += request[j];
                }
        }
        printf("Process %d: Requesting resources.\n", i);
}

/* Generate a release vector */
void generate_release(int i, int *request)
{
        int j, sum = 0;
        while (!sum)
        {
                for (j = 0; j < resource_count; j++)
                {
                        request[j] = rand() % (s->allocation[i][j] + 1);
```

```
                        sum += request[j];
                }
        }
        printf("Process %d: Releasing resources.\n", i);
}

/* Threads starts here */
void *process_thread(void *param)
{
        /* Process number */
        int i = (int)(long)param;
        /* Allocate request vector */
        int *request = malloc(resource_count * sizeof(int));
        while (1)
        {
                /* Generate request */
                generate_request(i, request);
                while (!resource_request(i, request))
                {
                        /* Wait */
                        Sleep(100);
                }
                /* Generate release */
                generate_release(i, request);
                /* Release resources */
                resource_release(i, request);
                /* Wait */
                Sleep(1000);
        }
        free(request);
}

// Allocate resources to the state structure.
void allocate_state() {
        s = (State *) malloc(sizeof(State));
        s->resource = malloc(resource_count * sizeof(int));
        s->available = malloc(resource_count * sizeof(int));

        s->max = malloc(process_count * sizeof(int*));
        s->allocation = malloc(process_count * sizeof(int*));
        s->need = malloc(process_count * sizeof(int*));
        for (int i = 0; i < process_count; i++) {
                s->max[i] = malloc(resource_count * sizeof(int));
                s->allocation[i] = malloc(resource_count * sizeof(int));
                s->need[i] = malloc(resource_count * sizeof(int));
        }
}
```

```
// Free dynamically allocated resources from
// the State structure and its members. Then
// NULL the State variable `s`.
void free_state() {
        for (int i = 0; i < process_count; i++) {
                free(s->max[i]);
                free(s->allocation[i]);
                free(s->need[i]);
        }
        free(s->max);
        free(s->allocation);
        free(s->need);

        free(s->resource);
        free(s->available);
        free(s);
        s = NULL;
}

int main(int argc, char *argv[])
{
        /* Get size of current state as input */
        int i, j;
        printf("Number of processes: ");
        scanf("%d", &process_count); // n
        printf("Number of resources: ");
        scanf("%d", &resource_count); // m

        /* Allocate memory for state */
        if (s == NULL)
        {
                allocate_state();
        };

        /* Get current state as input */
        printf("Resource vector: ");
        for (i = 0; i < resource_count; i++)
                scanf("%d", &s->resource[i]);
        printf("Enter max matrix: ");
        for (i = 0; i < process_count; i++)
                for (j = 0; j < resource_count; j++)
                        scanf("%d", &s->max[i][j]);
        printf("Enter allocation matrix: ");
        for (i = 0; i < process_count; i++)
                for (j = 0; j < resource_count; j++)
                        scanf("%d", &s->allocation[i][j]);
        printf("\n");
```

```
/* Calcuate the need matrix */
for (i = 0; i < process_count; i++) {
        for (j = 0; j < resource_count; j++) {
                s->need[i][j] = s->max[i][j] - s->allocation[i][j];
                if (s->need[i][j] < 0) {
                        printf("Too many resources of type %d are allocated for process %d.\n",
                        exit(EXIT_FAILURE);
                }
        }
}

/* Calcuate the availability vector */
for (j = 0; j < resource_count; j++)
{
        int sum = 0;
        for (i = 0; i < process_count; i++)
                sum += s->allocation[i][j];
        s->available[j] = s->resource[j] - sum;
}

/* Output need matrix and availability vector */
printf("Need matrix:\n");
for (i = 0; i < resource_count; i++)
        printf("R%d ", i + 1);
printf("\n");
for (i = 0; i < process_count; i++)
{
        print_array("%d  ", s->need[i], resource_count);
        printf("\n");
}
printf("Availability vector:\n");
for (i = 0; i < resource_count; i++)
        printf("R%d ", i + 1);
printf("\n");
print_array("%d  ", s->available, resource_count);
printf("\n");

/* If initial state is unsafe then terminate with error */
if (!is_safe()) {
        fprintf(stderr, "The initial state is unsafe!\n");
        exit(EXIT_FAILURE);
}
printf("The initial state is safe!\n");

/* Seed the random number generator */
struct timeval tv;
gettimeofday(&tv, NULL);
srand(tv.tv_usec);
```

```
        /* Create process_count threads */
        pthread_t *tid = malloc(process_count * sizeof(pthread_t));
        for (i = 0; i < process_count; i++)
                pthread_create(&tid[i], NULL, process_thread, (void *)(long)i);

        /* Wait for threads to finish */
        pthread_exit(0);
        free(tid);

        /* Free state memory */
        free_state();
}
```

7.5.2 array_helpers.h

```c
#include <stdlib.h>

// Compares elements from a with elements in b, and returns
// true if a[i] <= b[i] for all i = 0..n-1.
int array_lte(int* a, int* b, size_t elements);

// Prints an array with a format string.
void print_array(char *format, int *array, size_t elements);

// Adds each element from array b onto each element in
// array a. Array a is modified as a side-effect.
void array_add(int* a, int* b, size_t elements);

// Subtracts each element in array b from each element
// in array a. Array a is modified as a side-effect.
void array_subtract(int* a, int* b, size_t elements);
```

7.5.3 array_helpers.c

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

// Compares elements from a with elements in b, and returns
// true if a[i] <= b[i] for all i = 0..n-1.
int array_lte(int* a, int* b, size_t elements) {
        for (int i = 0; i < elements; i++) {
                if (a[i] > b[i]) return 0;
        }

        return 1;
}

// Prints an array with a format string.
void print_array(char *format, int *array, size_t elements) {
        for (int i = 0; i < elements; i++) {
                printf(format, array[i]);
        }
}

// Adds each element from array b onto each element in
// array a. Array a is modified as a side-effect.
void array_add(int* a, int* b, size_t elements) {
        for (int i = 0; i < elements; i++) {
                a[i] = a[i] + b[i];
        }
}

// Subtracts each element in array b from each element
// in array a. Array a is modified as a side-effect.
void array_subtract(int* a, int* b, size_t elements) {
        for (int i = 0; i < elements; i++) {
                a[i] = a[i] - b[i];
        }
}
```

7.5.4 Makefile

```
all: banker

UNAME := $(shell uname)

# Compiler arguments
CC=gcc
CCFLAGS=-Wall -Werror

# Linker arguments
LDFLAGS=

ifneq ($(UNAME), Darwin)
        LDFLAGS=-pthread
endif

banker: banker.o array_helpers.o
        $(CC) $(CCFLAGS) $(LDFLAGS) -o $@.out banker.o array_helpers.o

banker.o: banker.c
        $(CC) $(CCFLAGS) -c banker.c

array_helpers.o: array_helpers.h array_helpers.c
        $(CC) $(CCFLAGS) -c array_helpers.h array_helpers.c

.PHONY: clean

clean:
        rm -rf *.o *.gch *.out

debug: CCFLAGS += -DDEBUG -g
debug: clean all
```

7.5.5 Input files

input.txt

```
4
3

9 3 6

3 2 2
6 1 3
3 1 4
4 2 2

0 0 0
0 0 0
0 0 0
0 0 0
```

input2.txt

```
4
3

9 3 6

3 2 2
6 1 3
3 1 4
4 2 2

1 0 0
5 1 1
2 1 1
0 0 2
```