# 02335 Operating Systems, Fall 2024
# Instructions for Assignment 3: Thread-safe Alarm Queue — v.1

October 7, 2024

In this assignment, you will implement a simple message queue with prioritized alarms which might be used, e.g. in an embedded system.

The queue must be *thread safe*, i.e. it must be possible to call any operations concurrently from different threads.

The queue must be provided as a *static library* to be included by other programs when linked.

You will first develop a sequential, non thread-safe version of the library. This may be tested with the provided `demo` build target. Your sequential library may then be made thread-safe and tried out using the provided `test` target which exercises the library from a concurrent program. You will also have to develop further concurrent tests yourself.

## 1 Learning objectives

During this assignment you will be working towards the following learning objectives:

- You can explain how the compiler, assembler and linker are used to create executables.
- You can apply standard programming methodologies and tools such as test-driven development, build systems, debuggers.
- You can explain the role of each component of a compilation tool-chain used in system programming and how the components interact.
- You can generate a static library and use it for static linking
- You can use synchronization primitives to establish thread safe data structures.
- You can use synchronization primitives to implement blocking operations.
- You can use concurrent threads to test thread-safe and blocking data structures.

## 2 Reference material

When reading literature, it is helpful to remember that the AMD64 architecture is known under many synonyms, including AMD64, x86-64, x86_64, EM64T, IA-32e and Intel 64.

See Assignment 1 for reference material on C, Unix and development tools.

For this assignment, you should consult:

- Section 2.4 Tanenbaum and Bos' book, with emphasis on sections 2.4.5 and 2.4.6.

## 3 Programming Environment

For all the assignments, you are supposed to be programming for a Unix environment. This may in particular be a Linux environment, but any Posix-compliant system (like MacOS) should be feasible.

On DTU Learn you will find instructions for establishing and testing a Unix environment on your platform.

## 4 Source code tree

For this assignment, you are provided with a source code file `assignment_3.tgz` which is organized into various `.c` and `.h` files and a Makefile for building and testing your program.

The files your are given in this assignment are:

- `aq.h`

  Contains the functions prototypes and definitions for the alarm queue

- `aq_skel.c`

  A skeleton file providing dummy implementations of the prototype functions.

- `aq_demo.c`

  Contains an example of a simple sequential program that tests the basic functionality of the sequential library version.

- `aq_test.c`

  Contains an example of a simple concurrent program testing the blocking behaviour of the an alarm queue.

- `Makefile`

  A makefile for building a sequential demo target `demo` and an concurrent test target `test`. There should be no need to modify this file, but you may do so if needed.

### 4.1 Testing the environment

To test the environment, first make sure you are in the directory where you unpacked the tar file and then type:

```
cp aq_skel.c aq_seq.c
cp aq_skel.c aq_tsafe.c
make
```

This will produce two binaries (programs), `demo_aq` and `test_aq`. After successful compilation type:

```
./demo_aq
```

to run the sequential test and

```
./demo_aq
```

to run the concurrent test. Of course, both will fail as they use the skeleton implementation only.

## 5 Working on the assignment

You are going to implement a message queue to be used by concurrent senders and receivers.

Messages may be of two kinds: *Normal messages* and *alarm messages*. The system should give priority to alarm messages but will only be able to hold a single alarm message at at time.

The actual messages are supposed to be held in memory blocks allocated by the sender and to be freed by the receiver. Messages are passed to and from the queue using pointers to these blocks using the universal `void *` type. Therefore, a sender should not access the message block after sending it. The actual form of the messages is to be agreed by the senders and receivers using the queue, casting the memory blocks pointed at to the appropriate types.

The interface to alarm queues is defined by the header file `aq.h` providing constants, types and function prototypes for using alarm queues.

An alarm queue is created using the function `aq_create()`. If successful this returns a handle to the queue in the form of a pointer to an opaque type. That the type is opaque means that the user of the queue does not know anything about the structure pointed at. To reflect this, the type `AlarmQueue` is defined as `(void *)`. Any number of alarm queues may be created. If anything fails, the `aq_create` function returns `NULL`.

For a given queue `q`, messages are sent using the operation `aq_send(q,msg,kind)`, where `msg` is a non-null pointer to a message object, and `kind` indicates its kind using one of the two constants `AQ_NORMAL` and `AQ_ALARM`. It must always be possible to send a normal message, i.e. the the call of `aq_send` must never block. This calls for a dynamic datastructure for the alarm queue.

An alarm queue can hold at most one alarm message. If an alarm message has been sent to the queue, but not yet received, further calls of `aq_send` with an alarm message must block until the first alarm message has been received. Once this happens, a new alarm message may be inserted into the queue.

Messages on an alarm queue `q` are received using the operation `aq_recv(q,pmsg)` where `pmsg` is a pointer to a variable where the pointer to the received message is to be placed. If no messages are held in the queue, the operation must block until a message is available. When this is the case, the operation sets the variable at `pmsg` to point to the message block, removes the message from the queue, and returns the kind of the message.

Normal messages must be received in the order sent. Alarm messages take priority over normal messages. At most one alarm message may be held in the queue at any time.

The queue interface also provide functions `size(q)` and `alarms(q)` which return the current number of messages (of both kinds) and the number of alarm messages (0 or 1) respectively. These may be used for testing and debugging purposes.

The operations of the queue must be *thread-safe*, i.e. it must be possible to call the operations concurrently from different threads at the same time without the queue becoming inconsistent. Furthermore the operations must be *atomic*, i.e. any concurrent calls of the operations must work as is executed in some sequential order.

In order to achieve thread safety, the queue should be implemented as a *monitor* using mutex'es and conditions.

## 5.1 Task 1: A sequential alarm queue

Before dealing with the thread safety, you should in this task implement a version of the alarm queue which is to be used on in sequential, non-concurrent program environment. This should help you getting the data structure right.

For such a queue, blocking does not make sense (as it would block the whole program). Therefore the queue behaviour is to be modified as follows:

- If there are no messages (of either kind) present when `aq_recv` is called, it should return immediately with error code `AQ_NO_MSG`.
- If the queue already holds an alarm message when `aq_send` is called with an alarm message, it should return immediately with the error code `AQ_NO_ROOM`

The sequential queue should be developed in a file named `aq_seq.c` which is used when generating the sequential version of the library which again is used by the sequential test program `demo_aq`.

Note that the `create` operation must generate distinct queues for each call. I.e. it must allocate a struct which holds all the information about the data structure of the new queue.

### Reporting for Task 1

In the report section for this task, you should address the following points:

1. Describe and illustrate the data structure you have chosen for the queue. In particular you should explain how normal and alarm messages are represented.
2. Study the `Makefile` and explain briefly what the sequential queue library is called, how it is generated, and how the demo program gets access to it.

3. Explain briefly what the `aq_demo.c` (using `aux.c`) program does and which properties of the sequential queue is tests.

## 5.2 Task 2: Making the alarm queue thread-safe

In this task you should elaborate on the sequential alarm queue in order to make it thread-safe as described above.

The thread-safe alarm queue should be developed in the file named `aq_tsafe.c`. Its contents should be as close to the sequential version as possible, but using mutex'es and conditions in order to achieve thread-safety and proper blocking of operations.

If implemented properly in Task 1, the `create` operation should allocate a distinct structure for each queue. Hence this operation should not need further protection itself. However, for each queue, appropriate mutex and condition instances must be allocated and associated with the queue created.

### Reporting for Task 2

In the report section for this task, you should address the following points:

1. Describe the purpose of the mutex and condition instances you are using for each queue and how you have used them.

## 5.3 Task 3: Testing the thread-safe alarm queue

In this tasks you should develop your own tests of the thread-safe alarm queue. Following the idea of *test-driven development*, this task may done before or concurrently with tasks 1 and 2.

When testing programs to run in a concurrent setting, you would often like operations called from different threads to occur in a specific order. However, the actual execution order is determined by the OS scheduler.

Therefore, in order to test specific execution scenarios, you may (try to) overrule the scheduling by running the program in *slow motion* inserting sleeps are various positions. Although this does not guarantee that a particular execution scenario is always achieved, in most cases it gives sufficient confidence. The durations of the sleep operations should be in order of tens or hundreds milliseconds.

The execution scenarios may be illustrated by (UML-like) *sequence diagrams* which shows in which order various operations are called and when they return.

### Reporting for Task 3

In the report section for this task, you should address the following points:

1. Draw a sequence diagram showing how the queue operations are executed in the `aq_test.c` program. Explain which properties of the alarm queue, are tested by this.

2. Device a test which shows that a call of `aq_send` with an alarm message will block if an alarm message is already present and that it will unblock when the previous alarm has been received. Draw a sequence diagram of this test and implement it in a file called `test1.c`.

   Run the test and describe the result.

   Hint: You should use three threads in order to detect when the blocking ends. This could be done by sending a normal message afterwards and compare its position to other normal messages.

3. Provide at least one more test of your own demonstrating some specific property of the alarm queue. These tests should occur in files named `test2.c` etc.

## 6 Hand-in and reporting

On Assignment 3 on DTU Learn, you must hand in the following:

- A file named `group_`*`NN`*`_assign_3.tgz` containing your solution files `aq_seq.c`, `aq_tsafe.c` , and your test files `test1.c`, `test2.c`, . . . . If you have changed or added other files, these should be included also. As suggested by the suffix, the format of the file must be a gzipped tar-archive.
- A file named `group_`*`NN`*`_assign_3.pdf` with a report for the assignment. See below.

In the above file names, *NN* is your (two-digit) group number, e.g. `group_05_assign_3.tgz` .

### 6.1 Report requirements and rules

The report may be written in Danish or English. It must be a pdf file in A4 format but otherwise the layout is free.

The report should have a front page with the following information:

- **Who:** Group no. + name and study no. of each participant
- **What:** Assignment name and no.
- **Where:** University, department, course number and course name
- **When:** Date of hand-in

In the report, each task should be addressed in order and the reporting points properly answered.

There is no page limit, but the report should be kept concise and to the point. You may use code snippets to illustrate your text.

References to used material and solutions may be given as footnotes or by a reference list. Any use of generative AI must be declared: What did you use it for and how?

By handing in a report with all your names, all members of the group implicitly confirm that all work has been done by the group itself and that each member has made a significant contribution to the work.

Also note:

- Any collaboration with other groups on smaller parts of the assignments must be declared and clearly identified. Collaboration on major parts is not acceptable.
- Any undeclared use or adaption of others' work is *strictly prohibited*