Real-Time Embedded Systems, 7.5 hp
Assignment 3

Before you start the Assignment

Check the "Practical Assignments and Project Seminar" learning module on the course page on Blackboard. It includes general information about the assignments and the Lab Kit.

After you complete Assignment

Once you have completed the tasks for Assignment 3, one student in the group will submit the results via the course page on Blackboard.

The assignment submission form is found within the learning module "**Submissions**".

Grading and Deadline

- **Grading Session**: Check the date and time in Time edit for "Grading Ass. 3".
- **Deadline**: By the day before the grading session at **11PM**.

Objectives

- Strengthen your skills in working with the Raspberry Pi, particularly concerning concurrent programming.
- Understand, use, and modify the internals of a small multi-threading kernel called tinythreads.
- Implement cooperative multitasking[1] (aka non-preemptive multitasking) using a lightweight multi-threading kernel implemented with non-local gotos.
- This assignment also emphasizes the real-time implications of cooperative multitasking, where tasks must voluntarily yield control. Unlike preemptive systems, a single long task can delay others, which has direct consequences in embedded real-time systems (e.g., LED blinking or display refresh)

Content

Assignment 3 includes three parts:

- Part 1 involves understanding the Tinythreads library, a lightweight multi-threading kernel implemented using non-local gotos. The original tinythreads library has been modified and ported to RPi 3 (ARMv8) for this assignment.
- Part 2 is about testing your understanding of Tinythreads.
- Part 3 encompasses implement cooperative multitasking using TinyThreads. Cooperative multitasking allows multiple tasks to execute concurrently and voluntarily yield control to another task.

Q&A

Supervision sessions will address questions, and students are encouraged to share relevant queries or comments on the Discussions forum. **Note that posting source code is not allowed.**

[1] https://en.wikipedia.org/wiki/Cooperative_multitasking

Required Equipment and Software:

- Lab kit, including the PiFace Control and Display, along with a breadboard circuit for the LED.
- Raspberry Pi OS (if you are compiling code on it).
- For debugging purposes, consider using a serial console software like PuTTY, which works on Windows, macOS, and Linux.

References

- C reference[2] (Recommended by Wagner)

[2] https://en.cppreference.com/w/c

PART 1 - Understanding Tinythreads

Tinythreads is a lightweight multi-threading kernel implemented using non-local gotos. In Assignment 3, the original Tinythreads library (tinythreads.h and tinythreads.c) was modified and ported to RPi 3 (ARMv8).

The objective of Part 1 is to gain a comprehensive understanding of how the Tinythreads library works, including the modifications made specifically for the DT8025 course. **This is a crucial part of the assignment and should not be underestimated**.

Assignment Details

1. Start by downloading ´a3p1.zip´file and extracting its contents.
2. Familiarize yourself with Tinythreads and understand how it works.

Technical notes on the implementation (By Fredrik Bengtsson @ LTU.SE)

- The thread implementation in tinythreads follows the typical pattern found in threading libraries. Threads share global memory and other resources while using private stacks. Execution using alternative stacks is achieved through a platform-dependent trick in the spawn() function. Once the newly created thread's execution context is initialized by `setjmp`[3] (return value 0), the stored stack-pointer is overwritten with an appropriate address within a separate memory block. The exact location within a `jmp_buf`[4] where the stack-pointer is stored may differ between platforms. It's important to set the initial value close to the highest address of the designated memory block, especially on architectures where the stack grows towards lower addresses. For Assignment 3, the SETSTACK(buf, a) macro has already been modified to work with the RPi 3 (ARMv8). Thus, the preprocessor macro `SETSTACK(buf, a)` may need modification if you intend to use tinythreads.c on another platform than the RPi 3 (ARMv8).
- Understanding this thread library primarily involves understanding and using `setjmp` and `longjmp`, for example, which registers/state are saved, how stacks are initialized, and why SETSTACK macro is needed for ARMv8. Hence, consider working through a sequence of `spawn` and `yield` calls as presented in the attached TheoryA2.pdf file. Notice how the underlying "thread of execution" jumps from context to context.
- Regarding memory management, the stack space for a thread is allocated as part of the control block for that thread. A fixed amount of thread control blocks (`thread_block`) is defined in a global array (`threads`), initially organized as a linked list of free blocks. Memory can be allocated from this pool when a new thread is spawned. While allocation and deallocation of thread memory are handled by the same operations that organize queues of threads in other parts of the implementation, this may be somewhat of an overkill. In practice, allocation and deallocation could be streamlined. However, it is important to note that the stack spaces and the thread block array have fixed sizes, which could be limiting. In the current implementation, execution halts if dequeuing is attempted on an empty queue. In contrast, a shortage of stack space will manifest as random memory corruption caused by out-of-bounds memory writes.
    - In a production system, a more dynamic memory management scheme would be desirable, ideally coupled with an analysis method to predict the memory needs of a set of threads statically.

[3] https://en.cppreference.com/w/c/program/setjmp
[4] https://en.cppreference.com/w/c/program/jmp_buf

3. Add C Libraries

- Considering the results from Assignments 1 and 2, please copy the following files into a3p1/lib/:
    i. piface.*
    ii. expstruct.*
    NOTE: Use the files from a2p2, which include the iexp() function without the LED blinking interleaving.

4. Generating a kernel using Tinythreads

- The `a3p1.c` file contains an example of a program executing two tasks concurrently. One task computes and displays the power of a number indefinitely, while the other computes and displays prime numbers indefinitely. Each task is assigned to a different "thread" to execute; one is spawned while the other runs as the main thread. Compile the code and boot the RPi using the newly created kernel, i.e., `a3p1.img`.
- A successful kernel creation and installation should enable you to visualize the outcomes of two threads executing the power function, similar to `a3p1.expected.img`.

Overall, Part 1 focuses on understanding the Tinythreads library, adding necessary C libraries, and generating a kernel that demonstrates concurrent task execution. Make sure to carefully follow the instructions for successful completion of Part 1, as it forms the foundation for the rest of the assignment.

Deliverables

For Assignment 3 Part 1, there are no deliverables, i.e., nothing to submit.

PART 2 - Testing your understanding of Tinythreads

The objective of Part 2 is to challenge your understanding of how the modified Tinythreads library works. To achieve this, consider the Tinythreads source code and the following code excerpt from `a3p1.c`.

```
1  int main() {
2      piface_init();
3      spawn(computePower, 0);
4      computePrimes(1);
5  }
```

Assignment Details

Answer the provided questions in a file named "AnswersA3P2.txt."

1. What is the purpose of the `spawn` function? How does it work?
2. What is the purpose of the `dispatch` function? How does it work?
3. What is the purpose of the `yield` function? How does it work?
4. When is the code `current->function(current->arg);` within the `spawn` function executed?
5. After finishing the execution of line 3, describe the content of `readyQ`.
    i. Note: you might want to draw a diagram to visualize the queue.
6. After finishing the execution of line 3, describe the content of `freeQ`.
    - Note: you might want to draw a diagram to visualize the queue.
7. Which task, i.e., computePower and `computePrimes`, executes first, and why?
8. Although functions `computePower` and `computePrimes` never return, they execute concurrently in the `a3p1.img` kernel. How is this achieved?
9. In a hypothetical scenario where `computePower` and `computePrimes` do return, which implies that the threads assigned to execute the tasks will terminate, you may need to track which threads terminated. One approach is to keep information about the terminated threads in a list called `doneQ`. Where in `lib/tinythreads.c` would you add a thread that terminated into `doneQ`?

Deliverables

For Assignment 3 Part 2, one student in the group must upload the following individual files:

- `AnswersA3P2.txt`

Note:

- All students in the group are equally responsible for the submitted source code.
- The group ensures that the submitted code does not include cheating and plagiarism issues.

PART 3 - Cooperative multitasking using Tinythreads

The objective of Part 3 is to implement cooperative multitasking using TinyThreads. In cooperative multitasking, multiple tasks are executed concurrently and voluntarily yield control to the scheduler or to other tasks. A task executes until it explicitly releases control or enters a waiting state (e.g., I/O operation). This approach contrasts with manual interleaving, where the programmer manually schedules tasks in the code.

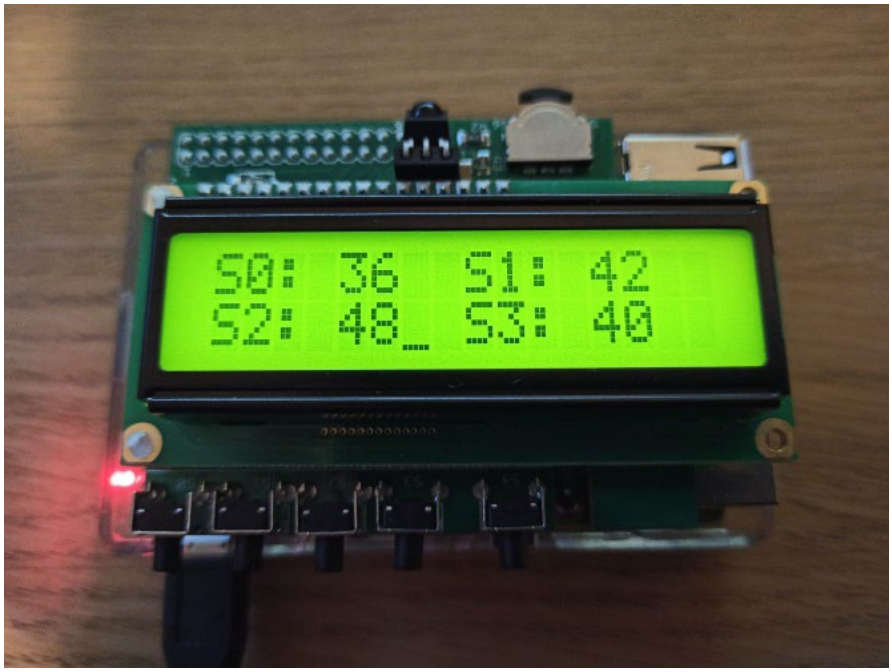
Assignment Details

1. Make a copy of the a3p1 folder/directory and rename it to a3p3.
2. In a3p3, rename `a3p1.c` to `a3p3.c`.
3. Edit the `Makefile` and change `MAINFILE = a3p3`.
4. Edit a3p3.c and finish the implementation of `void computeExponential(int seg)`

- This function will compute and display the result of the iexp function in a particular segment on the LED. If `seg` is odd, the function displays the fraction part of iexp; otherwise, it displays the integer part.
- The functions `void computePrimes(int seg)`, `void computePower(int seg)`, and `void computeExponential(int seg)` represent tasks that cooperate.

5. Edit lib/piface.c and implement the following functions (already declared):

- `void print_at_seg(int seg, int num)`: The LCD can be divided into 4 segments. Each segment (top_left, top_right, bottom_left, and bottom_right) can be used by a thread to present the output result of its running function. See the example illustrated below.



- To position the cursor on the PiFace Display, you might find it useful to use `void piface_set_cursor(uint8_t col, uint8_t row)` in `lib/piface`. This function was inspired by the piface library created by Thomas Preston <thomas.preston@openlx.org.uk>.

6. Edit `a3p3.c` and replace the calls to `PUTTOLDC("T%i: %i", seg, value)` with `print_at_seg(seg, value)`.

- For example, considering the `computePower` function, replace the instruction `PUTTOLDC("T%i: %i", pos, n*n)` with `print_at_seg(pos, n*n)`.

7. To test, compile `a3p3.c` and boot the Raspberry Pi using the newly generated kernel, i.e., `a3p3.img`. Make sure it works before moving to the next step.
8. Now, let's further experiment with cooperative multitasking by adding more task to execute concurrently.
    - **Remember**: cooperative multitasking only guarantees fairness if tasks yield frequently. In real-time systems, this means long-running tasks (e.g., computing large exponentials) can block time-critical tasks like LED toggling unless carefully managed.
9. Edit a3p3.c and change the main file to:

```
int main() {
    piface_init();
    piface_puts("DT8025 - A3P3");
    RPI_WaitMicroSeconds(2000000);
    piface_clear();
    spawn(computePower, 0);
    spawn(computePrimes, 1);
    spawn(computeExponential, 2);
    computeExponential(3);
}
```

10. Compile the code and boot the RPi using the newly created kernel, i.e., `a3p3.img`. A successful kernel creation and installation should enable you to visualize the cooperative multitasking outcome on the PiFace Display. It should be similar to `a3p1.expected.img`.
11. Now, edit `a3p3.c` again and change the main file to:

```
int main() {
    piface_init();
    piface_puts("DT8025 - A3P3");
    RPI_WaitMicroSeconds(2000000);
    piface_clear();
    spawn(computePower, 0);
    spawn(computePower, 1);
    spawn(computePrimes, 2);
    spawn(computePrimes, 3);
    spawn(computeExponential, 4);
    spawn(computeExponential, 5);
    computeExponential(6);
}
```

12. Compile the code and boot the RPi using the newly created kernel, i.e., `a3p3.img`. Does it work? Create a file called ` AnswersA3P3.txt` and provide an explanation for why this new kernel is not working as expected.

Deliverables

For Assignment 3 Part 3, one student in the group must upload the following individual files:

- `AnswersA3P3.txt`
- `a3p3.c`
- `piface.c`

Note:

- All students in the group are equally responsible for the submitted source code.
- The group ensures that the submitted code does not include cheating and plagiarism issues.