Universität
Paderborn
Computer networks
group

Winter term 2020/21

Networked Embedded Systems

Homework 01: : Protothreads

Due date: 2020-11-29 23:59 CET

---

1 Introduction

In this exercise sheet, we will have a close look at Contiki protothreads. Protothreads are the central building blocks of Contiki applications and, also, relevant for future NES exercises and the team project. Being very light-weight, protothreads allow multi-threaded operation even on a constrained platform like our sensor nodes. From a user's perspective, a protothread is a function that keeps state between calls and continues operation at the point where it returned earlier. In the first two parts of this exercise, we will implement such functionality ourselves to better understand what Contiki does behind the scenes. In the last part, we will build firmware that uses Contiki's protothreads.
To pass this homework assignment, you have to successfully submit:

- the implemented source code & Makefile within an archive, e.g., zip.
  and
- a PDF document explaining details of the submitted solution showing screenshots of the correct output when the programs are executed. Please do not put the PDF document within an archive.

Note: Submit the homework assignment via PANDA; all team members have to confirm the submission in PANDA in order that it is counted as valid submission. The submission has to be done until the above outlined due date.

2 Protothread Concept

First, we implement a simple protothread ourselves. **In this part of the exercise, we do not use Contiki, but implement a normal C application for the PC.**

1. Download and unpack the source code from from PANDA[1].
2. Familiarize yourself with `single_thread.c` to understand how the protothread is used in this application.
3. Compile your protothread application with "`make single_thread`". If compilation was successful, run the app with "`./single_thread`".
4. Implement the `protothread()` function to create a thread that is split into four execution steps. In each step, the thread just prints a message indicating the current execution step, and returns. To signal termination, the thread returns `DONE` after the last step.
5. Use the output of the application to assert that
   - the protothread progresses between calls
   - execution terminates
   - the number of execution steps is correct

3 Protothread Scheduling

Next, we implement a simple scheduler to understand how protothreads can be used by the operating system to provide multi-threaded operation. This is again a normal C application for the PC.

1. Familiarize yourself with `multiple_threads.c` to understand how protothreads are used in this application.
2. Compile your protothread application with "`make multiple_threads`". If compilation was successful, run the app with "`./multiple_threads`".
3. Implement the "`protothread1()`" function to create a thread with four execution steps and "`protothread2()`" function to create a thread with two execution steps. In each step, the threads should output a message, indicating the thread id and the execution step.
4. Compile and run your protothread application.
5. Extend one protothread with a local variable (for example an integer that is incremented in each execution step). Output also the current value of the variable in each step. Reflect on the problems that might occur here. Why is the naive approach not working? Try to pinpoint the core of the problem to understand when normal local variables can be used and when that's not possible.

[1] see http://panda.upb.de/

4 Protothreads in Contiki

Next, we have a look at how Contiki uses protothreads. For the first time, we will also modify and extend a Contiki application. You can find the API documentation for our particular Contiki version (3.x) at `/upb/groups/fg-ccs/public/share/nes/2018w/contiki/doc/html/index.html`. Another good source of information is the Contiki's wiki page[2].

1. Extend `protothreads.c` with two more Contiki processes. Both new processes continuously print a message to the serial port. One process should start automatically, while the other one by pressing the *user* button. Helpful function is: "`process_start()`"
2. Run the firmware in the Cooja simulator and assert that both processes get scheduled.
3. You should now be able to answer questions like:
   - How are protothreads different from, for example, Linux threads?
   - Would it be possible to just reimplement Linux threads for our sensor motes?
4. Investigate yourself the functionalities of some common protothread macros/-functions that are used in Contiki processes.

Macros

```
PROCESS_BEGIN();
PROCESS_END();
PROCESS_EXIT();
PROCESS_WAIT_EVENT();
PROCESS_WAIT_EVENT_UNTIL();
PROCESS_YIELD();
PROCESS_WAIT_UNTIL();
PROCESS_PAUSE();
```

This concludes our "Protothreads" exercises.
For questions please visit the Tutorial Session on Thursday 09-11.

Contact

Florian Klingler <florian.klingler@uni-paderborn.de>
Course Website https://panda.uni-paderborn.de/course/view.php?id=16255

[2] see https://github.com/contiki-os/contiki/wiki#Internals