The University of Queensland
School of Information Technology and Electrical Engineering
Semester Two, 2012

# COMP3301/COMP7308
# Assignment 1 — User-space scheduler

**Due: 8pm on Friday 24th August 2012**
**Weighting: 25% for COMP3301 students (100 marks total)**

Version 1.0 — 18 July 2012

## Introduction

The goal of this assignment is to give you a practical knowledge of operating system schedulers and the experience in modifying them in a defined way.

As part of the assessment you must also answer some short–response questions that will test your understanding of the concepts that you have implemented.

You will complete this assignment on the student Linux server, `moss`. More information on this server is available from `http://student.eait.uq.edu.au/infrastructure/remote-access/`.

This assignment may be completed **individually** or in **self-selected groups of two**. Please read the section on group work if you decide to work in groups of two. Although you may share code with your team member (if working in groups of two), it is still considered cheating to look at another student's code or allow your code to be seen or copied. You should be aware that all code submitted may be subject to automated checks by plagiarism-detection software. You should read and understand the school's policy on student misconduct, which is available in the course profile.

## Pth—GNU Portable Threads

Pth is a portable userland threading library that can be linked against to provide threading capabilities to a process. You should be familiar with this library from the practicals. It is suggested you review these before starting the assignment.

The default Pth scheduler operates in a cooperative manner — threads must yield the CPU for other threads to run. Pth has been modified for you to include a preemptive scheduler, that can interrupt a thread if it runs longer than the fixed scheduling quantum. All further references to Pth refer to this preemptive version, and you will complete this assignment using that version. You may download Preemptive Pth from the course website. Do not download from any other source, as this will be the unmodified version.

## Scheduling behaviour

The current Pth scheduler implements a round-robin algorithm where each thread is given a time slice to run in turn. If the thread does not yield by the end of this fixed time slice then it is preempted, and the next ready thread is run. It does not provide any mechanism for a thread to specify a deadline.

Your task is to modify the Pth scheduler to implement the **earliest deadline first** scheduling algorithm. You should be familiar with this algorithm from lectures, so refer to the corresponding lecture notes if you need more information. The new algorithm will replace the current round-robin one.

Threads will specify their deadline on creation through two new attributes: `PTH_ATTR_DEADLINE_C` (execution time) and `PTH_ATTR_DEADLINE_T` (period). Both deadlines are strictly positive integer multiples of the time slice (that is, both are strictly $> 0$). You must create these new attributes and expose them through `pth.h` (hint: do not add these directly to `pth.h` as this is automatically generated). Ensure you can set and get the attributes from a test program. For instance, a thread may specify its deadline as follows:

`PTH_ATTR_DEADLINE_C` $= 1$
`PTH_ATTR_DEADLINE_T` $= 5$

The above example means that a thread has a deadline of once every 5 time slices.

New threads should inherit the deadlines of their parent threads, but may be overridden by calls to `pth_attr_set()`. If not specified (and not inherited), the attributes should default to the following:

| | |
|---|---|
| `PTH_ATTR_DEADLINE_C` | 1 |
| `PTH_ATTR_DEADLINE_T` | 10 |

When creating a thread, the scheduler should validate the deadline attributes and see if the system is still schedulable with the addition of the new thread. If either of these conditions fail, the scheduler must deny the thread's creation (by returning the error specified below). You do not have to handle the case where a thread's deadline is not an integer multiple of the time slice. Recall that the schedulability test of earliest deadline first is:

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1$$

On each scheduling decision, the thread with the earliest deadline should be run. If a clear decision cannot be made, the system should enter a tie break as follows:

1. If one of the tied threads ran in the last scheduling period, continue running it (see figure 1 for an example)
2. If neither thread ran in the last scheduling period, pick the oldest thread (oldest is defined as having being created before the other)

Once a thread has reached its deadline, it should be removed from consideration until the next multiple of its deadline is reached. For instance a thread specifying a deadline of once every 5 seconds will be runnable at $t = 0$, $t = 500$, $t = 1000$, *etc.* (where $t$ is in milliseconds.)

For simplicity, if a thread yields via `pth_yield()` or otherwise makes a call into Pth's POSIX replacement library (e.g. `write(2)`), you can consider it finished for the current scheduling slice and the scheduler should enter an idle loop until the next scheduling slice.

You should ensure that you do not disrupt Pth's event handling mechanism or its queue management functions (such as moving a thread between priority queues).

## Sample thread system

The following sample thread system are provided so you can visualise the concepts and behaviour specified. **Note that it is a simplified and contrived example. The scheduling period is shown as 1 second, whereas the Pth period is 100 ms.**

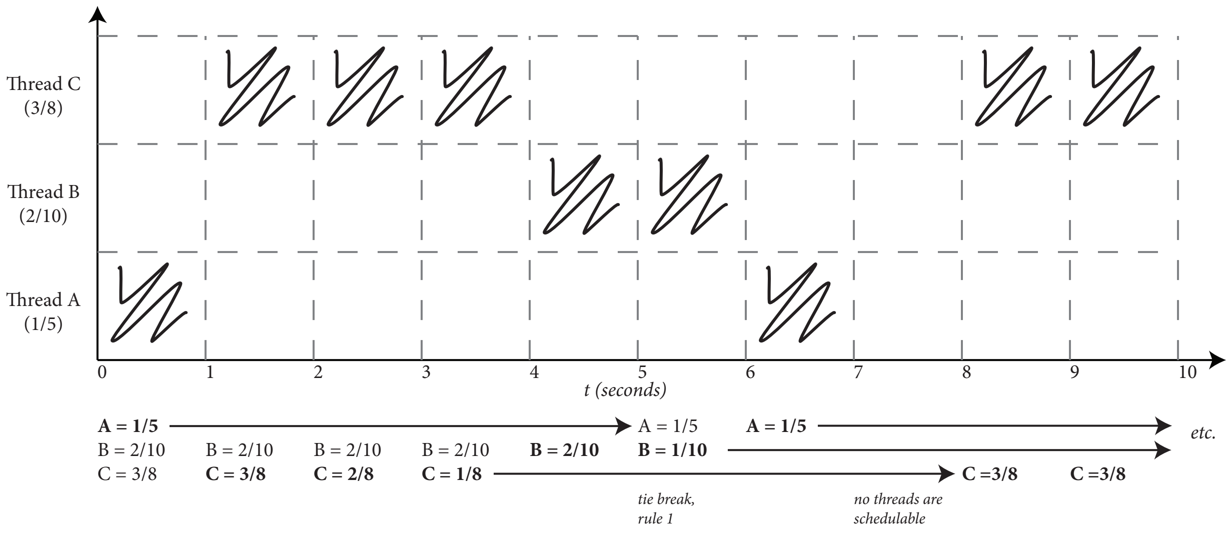

Figure 1: Sample thread system where scheduling period is 1 second

## Thread creation errors

Some additional errors may be returned if a new thread cannot be created. These should be returned via the `pth_error` function.

| `errno` **code** | **Reason for error** |
|---|---|
| `ERANGE` | if `PTH_ATTR_DEADLINE_C` $>$ `PTH_ATTR_DEADLINE_T` |
| `EAGAIN` | if schedulability test fails |

## Scheduling logging

As part of the assignment, you must also modify the Pth scheduler to output a log file with information on each thread's running time, as shown below. The output must be exactly as described as it will be automatically marked. This logging will give a visual representation of the scheduling decisions your scheduler is making, such as the order in which threads are picked, and how long they run for. It may be useful to start the assignment with this logging.

The behaviour of this logging is as follows:

On initialisation, the scheduler shall open a file called `sched.log` in the current working directory. If an existing file by this name exists, it should be overridden. Any errors in opening the file (such as permissions) should cause the initialisation to fail and a descriptive error to be printed to *standard error* (consider using `perror(3)`). Each time a user-created thread is created, a header row will be written to the file in the following format (followed by a newline):

`t` $T_i$ $T_{i+1}$ $\ldots$ $T_n$

where $T_i$ is the name of the first user-created thread, $T_{i+1}$ is each consecutive thread (if any), up to the last thread, $T_n$. Each column should be padded to 8 spaces, with at least one space between each (you may wish to consider using the `printf` padding specifier for this). Therefore if a thread name is longer than 7 characters, it should be truncated (7 visible characters and 1 space make up each column heading). There should be no leading or trailing spaces on any line, nor should there be any blank lines. The order of columns should be in order of thread creation, and should never change order. Do not include the scheduler or main threads. For example, a valid header row may be:

```
t       nice    greedy
```

Every time a new user-created thread is created, the header row should be updated and written to the file again. For example, adding a new thread to the above example may cause the following header row to be written:

```
t       nice    greedy  greedy2
```

If multiple threads are created before any run, then multiple header rows will be present in the log file.

For every scheduling decision, the time since epoch (rounded to the nearest whole millisecond) of when the thread started running should be written in the first column (under `t`), padded to 8 spaces. Note that this means if a thread yields in under 1 millisecond, the start time of it and the next thread will be the same. This is expected behaviour. For the purposes of this assignment, the epoch should be fixed to when the first user-created thread begins running.

Three `+` characters should then be written in the corresponding column for the thread that just ran, directly followed by a `P` if the thread was preempted, or a `Y` if the thread yielded. See above at the bottom of the "Scheduling behaviour" section for information on threads yielding.

If no thread ran in the given period (see Figure 1 at $t = 7$ for instance), the written line should only contain the time since epoch.

You will need to make sure you insert enough padding so the first character of the `+` lines up with the first character of the thread's name in the header row. For example, assuming the threads in the above example were all started in the main function, a possible log file output may be:

```
t       nice    greedy  greedy2
0       +++Y
0               +++P
100                     +++P
200             +++P
300                     +++P
400     +++Y
t       nice    greedy  greedy2 greedy3
400                             +++P
```

This output (assumed complete) reveals the following about the test program:

- The order of running was: `nice`, `greedy`, `greedy2`, `greedy`, `greedy2` and `nice`
- The `nice` thread yielded cooperatively back to the scheduler in under 1 millisecond
- The greedy threads were preempted after 100 milliseconds each, which reveals that the scheduling time slice is set to 100 milliseconds
- The `nice` thread was either waiting on an event or sleeping for at least 1 second
- At approximately 400 milliseconds since program start a new thread `greedy3` was created

- The program exited after the `greedy3` thread was preempted, which based on previous observations can be computed to half a second after the first thread was run

- The scheduler used to produce these outputs implements the round-robin algorithm, **not** the algorithm specified in this assignment

You should try to print log outputs as "close" to scheduling decisions as possible to ensure scheduling overhead does not creep into the time column (for instance if the scheduler spends 5 milliseconds doing housekeeping after a thread returns but before writing to the log file, you should ensure this 5 milliseconds isn't included in the time column).

It is expected that you may see some small drift in the times for a long running program due to factors outside of your control, however each time slice should always be 100 milliseconds. Acceptable drift between each thread is in the range of $\pm 5$ milliseconds. If you are unsure if the drift in your system is acceptable, please demonstrate your system to a tutor.

More examples of valid log outputs will be made available on the course website in due time (though the code used to produce them will not be). Some example log outputs are available on the course website. Please note they are contrived examples and do not necessarily show a complete solution.

## Short-response questions

1. What is the practical difference between earliest-deadline first and rate monotonic scheduling? Consider utilisation, schedulability, etc.

2. For simplicity, this assignment specifies that if a thread yields or makes a blocking system call, the scheduler should idle until the next scheduling slice. Explain why this is undesirable in real systems, and propose a solution that could be implemented into the Pth library to remedy this.

3. Show a sample system of threads that is *not* schedulable, and explain why not and how this could be fixed. Make sure you show each thread's deadlines in a table.

4. Show a sample system of threads that is schedulable in earliest-deadline first, but not in rate-monotonic scheduling (if any). Explain your answer.

## Code compilation

As the Pth source tree already contains a `configure` script and a `Makefile` to compile the library, you do not need to provide any additional build infrastructure. Your modifications must compile with the vanilla Pth Makefile system.

To mark your modified source, your assignment subdirectory will be checked out from Subversion on *moss*, and the latest version of the `build-pth.sh` script (as provided on the course website) will be run in your `a1` subdirectory (one up from the Pth source directory you imported), as follows:

```
./build-pth.sh
```

Note that it will **not** be built with debugging support. For more information on the `build-path.sh` script, see the Pth practicals.

Please ensure your assignment compiles successfully using the build script before submitting.

If the marker cannot mark your assignment due to compilation errors, they may attempt to fix the problem at their own discretion. A penalty of up to 10 marks may be applied if the marker has to modify your code in any way.

## Coding style

Your modifications should follow the existing style of the Pth source code around your code. We are aware that there are inconsistencies in the Pth source — you do not need to reformat any existing code.

## Group work

If you decide to work in groups of two, **one** member of the group should e-mail one of the tutors (or if unavailable the lecturer) **no later than seven days before the due date**. After this cutoff it will be assumed that you are working individually and you will be marked as such. Both members of a group will receive the same mark, and any complaints or problems should be directed to the lecturer who will treat each case confidentially.

The tutors and lecturer's e-mail addresses can be found on the course website.

## Submission and Version Control

The due date for this assignment is **8pm on Friday 24th August 2012**. Submission made after this time will incur a 10% penalty per day late (weekends are counted as 1 day). Any submissions more than 4 days late will receive 0 marks. No extensions will be given without supporting documentation (i.e. medical certificate or family emergency)—should such a situation occur you should e-mail the course coordinator as soon as possible.

This assignment must be submitted through ITEE's Subversion system. The repository URL for this assignment is (where *s4123456* is your student number):

```
https://svn.itee.uq.edu.au/repo/comp3301-s4123456/a1
```

If you are working in groups of two, only one student should use their repository. When submitting group membership to the tutor, you should nominate the student that will host the repository for the group. Permissions will be then set accordingly so you can use your own account. This means both students will checkout and commit to the same repository.

Before starting this assignment you should import the Pth source tree into a `pth-2.0.7-preempt` subdirectory in the above repository URL. This can be done with the `svn import` command, for example:

```
svn import pth-2.0.7-preempt \
    https://svn.itee.uq.edu.au/repo/comp3301-s4123456/a1/pth-2.0.7-preempt
```

Please do not commit a compiled copy of the source tree as it can be very large. Note that after importing, you will need to update your working copy to download the source tree from the server.

Submissions will be retrieved from your repository when the due date has been reached. Your submission time will be taken as the most recent revision in the above repository directory.

You are required to make regular commits to your repository as a demonstration of your work. Subversion history will be considered in marking. Do not submit your assignment with a single, large commit. You will be penalised for doing so.

For information regarding ITEE's Subversion system, see `http://student.eait.uq.edu.au/software/subversion/`.

Answers to the short-response questions should be provided in a `responses.txt` file in the specified repository directory (*not* the `pth-2.0.7-preempt` subdirectory).

Please ensure your solution builds and runs on *moss*, as this is where it will be marked.

# Assessment Criteria

| Grade band | Scheduler modifications (40 marks) | | Scheduler logging (25 marks) | | Short-response answers (20 marks) | | Coding style and comments (10 marks) | | Version control (5 marks) | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Excellent** | Correct implementation of scheduling algorithm with no errors. Clear understanding of scheduler concepts. | 40 | Scheduler log file created and populated correctly when test program is run. Output validates scheduler modifications as correct. | 25 | Clear explanations and descriptions of answers given. No incorrect or misleading explanations. Student clearly grasps the assignment concepts. | 20 | Coding style applied consistently and without any error. Code has meaningful comments where appropriate, with no complex sections left uncommented. | 10 | Evidence of continual progress through version control history. | 5 |
| **Very good** | Correct implementation of scheduling algorithm with very few to no errors. Clear understanding of scheduler concepts, with only minor issues. | 36<br>32 | Scheduler log file created and populated with few to no errors when test program is run. Output validates scheduler modifications as correct. | 23<br>20 | Clear explanations and descriptions of answers given. Very few incorrect or misleading explanations. Student clearly grasps the assignment concepts. | 18<br>16 | Coding style applied consistently with few errors. Code has meaningful comments where appropriate, with some complex sections left uncommented. | 9<br>8 | Evidence of good progress through version control history. | 4 |
| **Good** | Correct implementation of scheduling algorithm with few errors. Good understanding of scheduler concepts, with some issues present. | 28<br>24 | Scheduler log file created and populated with few errors. Output mostly validates scheduler modifications. | 18<br>15 | Explanations and descriptions of answers given. Some incorrect or misleading answers. Student has a fair grasp of the assignment concepts. | 14<br>12 | Coding style applied with some consistency and with some errors. Code has a fair amount of meaningful comments where appropriate, with some complex sections left uncommented. | 7<br>6 | Evidence of some progress through version control history. | 3 |
| **Satisfactory** | Partial implementation of scheduling algorithm with some errors. Basic understanding of scheduler concepts. | 20 | Scheduler log file created and populated with errors. Output cannot be used to validate scheduler modifications. | 12 | Some answers given with some incorrect or misleading answers. Basic explanations given. Student has a basic understanding of assignment concepts. | 10 | Fair attempt to apply coding style, but some errors or not much consistency. A basic attempt to place meaningful comments throughout code. | 5 | Little evidence of progress through version control history. | 2 |
| **Poor** | Basic attempt to implement scheduling algorithm but has significant errors. Little evidence of an understanding of scheduler concepts. | 16<br>12<br>8<br>4 | Scheduler log file created but not populated, or populated with many errors. Output does not correspond to scheduler modifications. | 9<br>7<br>5<br>3 | Very few answers given. Many incorrect or misleading answers. Student has a poor grasp of assignment concepts. | 8<br>6<br>4<br>2 | Basic attempt to apply coding style, with many errors or no consistency. Very few meaningful comments in code. | 4<br>3<br>2<br>1 | Very little to no evidence of progress through version control history. | 1 |
| **Very poor** | No attempt made to implement scheduling algorithm. | 0 | No logging output written to specified file. | 0 | No attempt made at short-response questions. | 0 | No attempt made to apply coding style standards or comment code. | 0 | No evidence of progress through version control history. | 0 |

| **Penalty:** | **Comments:** |
|---|---|
| **Total mark:** /100 | |