Operating Systems
Jacobs University Bremen
Dr. Jürgen Schönwälder

Course: CO20-320202
Date: 2018-10-18
Due: 2018-10-25

OS 2018 Problem Sheet #3

Problem 3.1: *Linux completely fair scheduler* (1+1+1+1 = 4 points)

The Completely Fair Scheduler (CFS) was introduced with the Linux kernel 2.6.23 and it was revised in kernel version 2.6.24. Read about the design of the (revised) CFS scheduler and answer the following questions:

a) What does fairness mean in the context of the CFS?

b) How does the CFS scheduler select tasks to run? What is the data structure used to maintain the necessary information and why was it chosen?

c) Does the CFS scheduler use time-slices? Are there parameters affecting CFS time calculations?

d) How do priorities (nice values) affect the selection of tasks?

Problem 3.2: *student running group* (6 points)

After an intense weekend, a group of students decide to do something for their fitness and their health. They form a running group that meets regularly to go for a run together. Unfortunately, the students are very busy and hence not always on time for the running sessions. So they decide that whoever arrives first for a running session becomes the leader of the running session. The running session leader waits for a fixed time for other runners to arrive. Once the time has passed, all runners that have arrived start running. Since runners may run at different pace, not all runners complete the run together. But the runners show real team spirit and they always wait until every runner participating in the running session has arrived before they leave to study again.

In order to increase the pressure to attend running sessions, the students decide that whoever has missed five running sessions has to leave the running group. In addition, it occasionally happens that the leader of a running session has to run alone if nobody else shows up. This is, of course, a bit annoying and hence runners who did run ten times alone leave the running group voluntarily.

Write a C program to simulate the running group. Every runner is implemented as a separate thread. The time between running sessions is modeled by sleeping for a random number of microseconds. The time the session leader waits for additional runners to arrive is also randomized. Runners arriving too late simply try to make the next running session.

Your program should use pthread mutexes and condition variables and timed wait functions. Do not use any other synchronization primitives. Your program should implement the option `-n` to set the number of runners initially participating in the running group (default value is one runner).

While you generally have to handle all possible runtime errors, it is acceptable for this assignment to assume that calls to lock or unlock mutexes or calls to wait or signal condition variables generally succeed. (This rule hopefully keeps your code more readable and makes it easier to review your solutions.) In general, try to write clean and well organized code. Only submit your source code files, do not upload any object code files or executables or any other files that are of no value for us.

Below is the non-debugging output of a solution for this problem:

```
./runner -n 10 2>/dev/null
r0 stopped after  17 run(s):   2 run(s) missed,  10 run(s) alone
r1 stopped after   8 run(s):   5 run(s) missed,   0 run(s) alone
r2 stopped after   4 run(s):   5 run(s) missed,   1 run(s) alone
```

```
r3 stopped after   8 run(s):   5 run(s) missed,   0 run(s) alone
r4 stopped after  12 run(s):   5 run(s) missed,   4 run(s) alone
r5 stopped after  17 run(s):   1 run(s) missed,  10 run(s) alone
r6 stopped after  21 run(s):   4 run(s) missed,  10 run(s) alone
r7 stopped after  12 run(s):   5 run(s) missed,   2 run(s) alone
r8 stopped after   6 run(s):   5 run(s) missed,   3 run(s) alone
r9 stopped after  13 run(s):   5 run(s) missed,   3 run(s) alone
```

A template of the program is provided below and on the course web page so that you can concentrate on filling in the missing parts that implement the logic of the student running group.

```c
/*
 * p3-runner/runner-template.c --
 *
 *      Runners meeting regularly (if they manage). See Operating
 *      Systems '2018 assignment #3 for more details.
 */

#define _REENTRANT
#define _DEFAULT_SOURCE

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <getopt.h>
#include <assert.h>
#include <pthread.h>
#include <errno.h>
#include <sys/time.h>

#define LATE_THRESHOLD          5
#define LONELY_THRESHOLD        10

#define GROUP_SLEEPING          0x01
#define GROUP_ASSEMBLING        0x02
#define GROUP_RUNNING           0x03
#define GROUP_FINISHING         0x04

#define RUNNER_SLEEPING         0x01
#define RUNNER_LEADING          0x02
#define RUNNER_ASSEMBLING       0x03
#define RUNNER_RUNNING          0x04
#define RUNNER_WAITING          0x05

typedef struct group {
    unsigned        state;      /* the state of the running group */
    unsigned        arriving;   /* number of runnners arriving */
    unsigned        running;    /* number of runnners running */
} group_t;

typedef struct runner {
    unsigned        id;         /* identity of the runner */
    pthread_t       tid;        /* thread identifier */
    unsigned        state;      /* state of the runner */
    unsigned        late;       /* number of runs missed (late arrival) */
    unsigned        lonely;     /* number of runs without any other runners */
    unsigned        runs;       /* number of runs completed */
    group_t        *group;      /* the group this runner belongs to */
} runner_t;

static const char *progname = "runner";
```

```
52
53  static void*
54  runners_life(void *data)
55  {
56      runner_t *runner = (runner_t *) data;
57      group_t *group = runner->group;
58
59      assert(runner && runner->group);
60
61      while (runner->late < LATE_THRESHOLD
62             && runner->lonely < LONELY_THRESHOLD) {
63
64          runner->state = RUNNER_SLEEPING;
65          (void) fprintf(stderr, "r%d sleeping\n", runner->id);
66          /* not very random but good enough here */
67          usleep(172000+random()%10000);
68
69          /* add additional logic to model the runners here */
70
71          /* the session leader is expected to wait for some time for
72             additional runners to arrive and join the session */
73          usleep(3600+random()%100);
74      }
75
76      return NULL;
77  }
78
79  int
80  main(int argc, char *argv[])
81  {
82      int c, n = 1;
83      runner_t *runner = NULL;
84      int rc = EXIT_SUCCESS;
85
86      group_t group = {
87          .state      = GROUP_SLEEPING,
88          .arriving   = 0,
89          .running    = 0,
90      };
91
92      while ((c = getopt(argc, argv, "n:h")) >= 0) {
93          switch (c) {
94          case 'n':
95              if ((n = atoi(optarg)) <= 0) {
96                  (void) fprintf(stderr, "number of runners must be > 0\n");
97                  exit(EXIT_FAILURE);
98              }
99              break;
100         case 'h':
101             (void) printf("Usage: %s [-n runners] [-h]\n", progname);
102             exit(EXIT_SUCCESS);
103         }
104     }
105
106     runner = calloc(n, sizeof(runner_t));
107     if (! runner) {
108         (void) fprintf(stderr, "%s: calloc() failed\n", progname);
109         rc = EXIT_FAILURE;
110         goto cleanup;
111     }
112
113     for (int i = 0; i < n; i++) {
114         runner[i].id = i;
```

```
115             runner[i].state = RUNNER_SLEEPING;
116             runner[i].late = 0;
117             runner[i].runs = 0;
118             runner[i].group = &group;
119 
120             /* XXX create thread for runner[i] XXX */
121 
122         }
123 
124         for (int i = 0; i < n; i++) {
125 
126             /* XXX join thread for runner[i] XXX */
127 
128         }
129 
130         for (int i = 0; i < n; i++) {
131             (void) printf("r%d stopped after %3d run(s): "
132                           "%3d run(s) missed, %3d run(s) alone\n",
133                           runner[i].id, runner[i].runs,
134                           runner[i].late, runner[i].lonely);
135         }
136 
137     cleanup:
138 
139         if (runner) { (void) free(runner); }
140 
141         return rc;
142     }
```