Individual Assignment 2

May 2019

1 Instructions

Full solutions should be turned in as a zip-file this involves both the written solutions and C-code. You turn-in the solutions in Ping-Pong. The solutions can be handwritten or generated in LATEX/Word. The solutions should be complete and well explained.

2 Problem 1

We have in the lectures seen how multi-threaded programs might have race-conditions when accessing shared resources. To avoid race-conditions synchronization primitives in the form of semaphores can be added to the code, for example to protect critical sections of the code. However, as a consequence of introducing semaphores deadlocks might occur. For all but trivial systems it is hard to analyze if a system might deadlock and thus automated methods that can identify deadlock situations are needed. A simple but straightforward approach is to identify situations when a deadlock situation has happened, this can be done by building a resource allocation graph and detect if the graph contains a cycle. However, this will only let the system detect that a deadlock has occurred but it will not avoid the deadlock. A more sophisticated approach is to build an automata or Petri net model of the code before actually running the code and to identify potential deadlock situations on this model and possible also synthesize extended constraints that will make sure the system avoids entering deadlock situations. Such an approach was developed in the Gadara project [1] where C-code could be automatically analyzed and a Petri net model automatically generated. The model is based ontrol control flow of the code and contains all resource allocations and deallocations. The resources in this case can represent critical sections in the code that are protected by binary semaphores. By using a synthesis technique it is possible to generated new Petri net places and transitions to the original model that will restrict the behavior of the system in such a way that deadlocks are avoided. The architecture is shown in Figure 1 and the basic steps in the approach are the following.

1. The C source is converted in a Control Flow Graph (CFG) that represent

all code execution paths of the code. The CFG code is augmented with locks representing give and take of a semaphore.

2. The CFG is translated into a Petri net.

3. A deadlock free graph is generated through synthesis techniques, this could be solved as a supervisory control problem.

4. The original code is extended with additional guards that guarantee that the new code does not have any deadlocks.

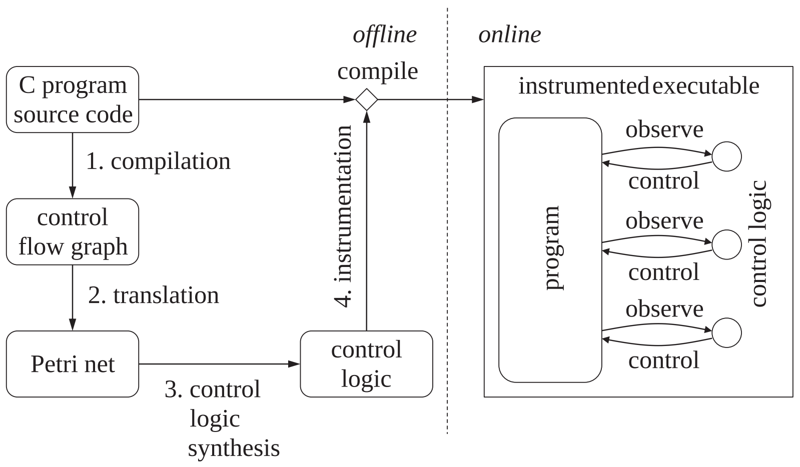

Figure 1: Overview of the Gadara architecture [1].

The developers of Gadara have applied the tool to software that have a complex concurrent behavior. One such tool is the BIND software - BIND is a widely used server to handle the Domain Name Systems (DNS) on internet. C-code (translated to use the FreeRTOS API instead of the original POSIX API for concurrency) for one part of the BIND software is shown in Figure 3. The Petri net model generated from this C-code is shown in Figure 2.

a) Inspect the C-code and the Petri net model and identify how the give and take of the semaphore in the C-code is represented within the Petri net model. It turns out that C-code has a possible deadlock situation. Identify the deadlock situation and describe how the tasks should be scheduled for the deadlock to arise.

b) Draw the resource allocation graph for the deadlock situation (see lecture slides for more information about the resource allocation grah). Motivate why the resource allocation graph must have a cycle for the system to be in a deadlock situation.

c) By manually analyzing the code, suggest how to modify the code, by adding additional locks, such that no deadlocks are possible. You should not restrict the concurrency more than necessary to avoid deadlocks. Submit both the modified C-code and what the modified Petri-net would be after introducing the changes to the C-code that you suggest. For an automated method for restricting the beavior such that deadlocks are avoided you can further study the approach in [].

3 Problem 2

For hard real-time systems it is necessary to determine if the tasks will meet their deadlines. Assuming no interlocks between processes and periodically executing processes this can be done using the Liu-Layland criteria and response-time analysis (see lecture slides). In this task you will implement a C-program that given a set of tasks will determine if it is possible to meet the specified deadlines. Each task is defined by an identity, a period, a worst-case execution time (WCET), and a deadline. The purpose of this task is to practice programming in C, as well as to get more familiar with the most important scheduling results.

To help you get started you can download skeleton code at `http://bit.do/sched_c`. Download and save this file as sched.c.

a) Implement the function `nbr_of_tasks`, by iterating through the linked list containing the tasks and returning the number of tasks in the list. Hint: Check the implementation of `print_tasks` to see how to iterate through the linked list.

b) In function `check_tests` the tasks are added and then a function call to `check_schedulable` is done. To this function call two arguments are provided, the first is the correct result for schedulability analysis using the Liu-Layland criteria and the second is the correct result for the response time analysis. For each analysis the answer might be `SCHED_UNKNOWN`, `SCHED_YES`, or `SCHED_NO`. Analyze the examples for the five test-cases and fill in the correct answer. Hint: See lecture slides on scheduling.

c) Implement the functions `schedulable_Liu_Layland` and `schedulable_response_time_analysis` such that they return the correct response for any given task set, assuming that priorities are unique and at least one task is in the task set.

You can run the code on the Bitcraze virtual Machine or you can use the Eclipse programming environment, see `https://www.eclipse.org/cdt/`.

References

[1] Hongwei Liao, Yin Wang, Hyoun Kyu Cho, Jason Stanley, Terence Kelly, Stephane Lafortune, Scott Mahlke, and Spyros Reveliotis. Concurrency bugs

in multithreaded software: modeling and analysis using petri nets. *Discrete Event Dynamic Systems*, 23(2):157–195, 2013.

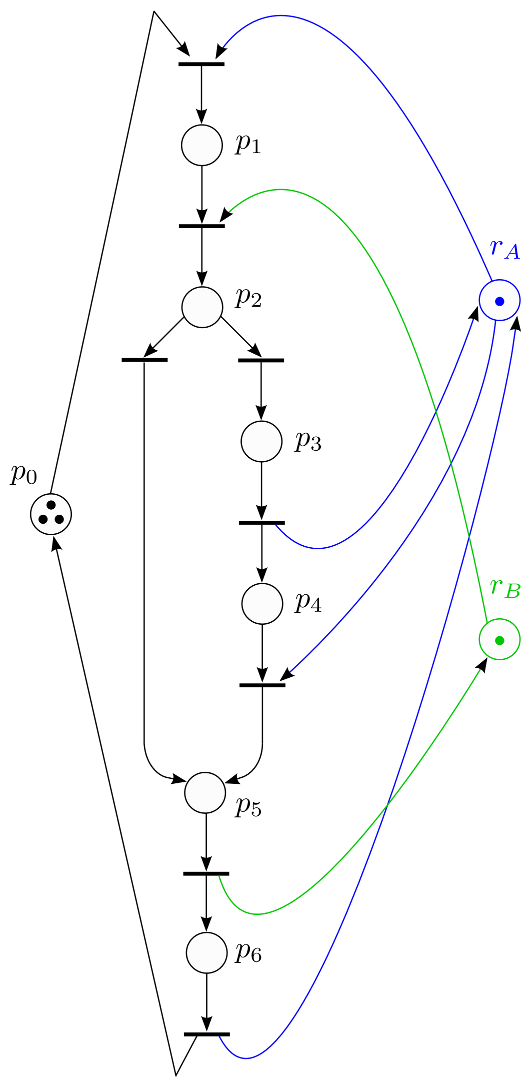

Figure 2: Example of a Petri net built for a part of the BIND software. From [1].

```c
#include <stdio.h>
#include <stdlib.h>

#include ”FreeRTOS.h”
#include ”task.h”
#include ”semphr.h”

xTaskHandle task_a_Handle;
xTaskHandle task_b_Handle;
xTaskHandle task_c_Handle;

SemaphoreHandle_t resource_a;
SemaphoreHandle_t resource_b;

void the_task(void *pvParameters) {
    while (1) {
        xSemaphoreTake(resource_a, portMAX_DELAY);
        xSemaphoreTake(resource_b, portMAX_DELAY);
        if (...) {
                xSemaphoreGive(resource_a);
                xSemaphoreTake(resource_a, portMAX_DELAY);
        }
        xSemaphoreGive(resource_b);
        xSemaphoreGive(resource_a);
    }
}

int main(int argc, char **argv) {
    resource_a = xSemaphoreCreateMutex();
    resource_b = xSemaphoreCreateMutex();
    xTaskCreate(the_task, ”Task␣1”, configMINIMAL_STACK_SIZE, NULL, 1, &task_a_H
    xTaskCreate(the_task, ”Task␣2”, configMINIMAL_STACK_SIZE, NULL, 1, &task_b_h
    xTaskCreate(the_task, ”Task␣3”, configMINIMAL_STACK_SIZE, NULL, 1, &task_c_h

    vTaskStartScheduler();
    for( ;; );
}
```

Figure 3: C-code using the FreeRTOS API for running tasks that have the same behavior is the Petri-net in Figure 2. The code (not executable) can be downloaded from `http://bit.do/bind_c`.