Individual Assignment 2

April 2023

1 Instructions

You should submit your solutions through Git. The solutions should include both the code and a PDF-file that clearly answers the questions related to both Problem 1 and Problem 2. Note, that this is an individual assignment and cooperation is not allowed.

2 Problem 1

In our lectures, we have seen how multi-threaded programs might encounter race-conditions when accessing shared resources. To avoid these race-conditions, synchronization primitives in the form of semaphores can be added to the code to protect, for example, critical sections of the code.

However, the introduction of semaphores can lead to deadlocks. For all but the most trivial systems, analyzing whether a system might deadlock is challenging, thus necessitating automated methods that can identify deadlock situations. A simple yet straightforward approach involves identifying situations when a deadlock has occurred. This can be done by constructing a resource allocation graph and determining if the graph contains a cycle. However, while this allows the system to detect when a deadlock has occurred, it does not prevent the deadlock.

A more sophisticated approach involves constructing an automaton or a Petri net model of the code before actually running it. This allows us to identify potential deadlock situations in this model, and possibly also synthesize extended constraints that will ensure the system avoids entering deadlock situations. This approach was developed in the Gadara project [1], where C-code could be automatically analyzed and a Petri net model could be automatically generated. This model is based on the control flow of the code and includes all resource allocations and deallocations. The resources in this case can represent critical sections in the code protected by binary semaphores. By using a synthesis technique, it is possible to generate new Petri net places and transitions in the original model, restricting the system's behavior in a way that prevents deadlocks. The architecture is shown in Figure 1. The fundamental steps in this approach are as follows:

1. The C source code is converted into a Control Flow Graph (CFG) that represents all possible execution paths of the code. The CFG is augmented with locks representing the acquisition and release of a semaphore.
2. The CFG is then translated into a Petri net.
3. A deadlock-free graph is generated through synthesis techniques. This can be treated as a supervisory control problem.
4. The original code is extended with additional guards that ensure that the new code does not contain any deadlocks.

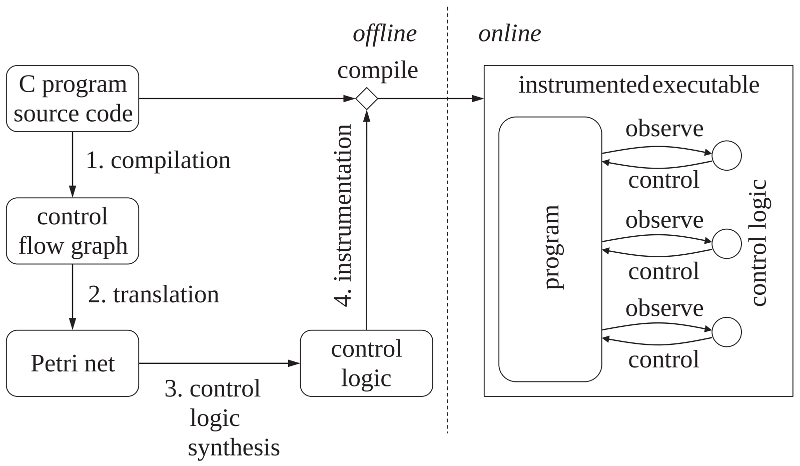

Figure 1: Overview of the Gadara architecture [1].

The developers of Gadara have applied the tool to software that has complex concurrent behavior. One such tool is the BIND software - BIND is a widely used server that handles the Domain Name Systems (DNS) on the 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 acquisition and release of the semaphore in the C-code is represented within the Petri net model. It turns out that the C-code has a potential deadlock situation. Identify this deadlock situation and describe the sequence in which tasks should be scheduled for the deadlock to occur. (1 p)

b) Draw the resource allocation graph for the deadlock situation (see lecture slides for more information about resource allocation graphs). Explain why the resource allocation graph must have a cycle for the system to be in a deadlock situation. (2 p)

c) By manually analyzing the code, suggest modifications by adding additional locks, such that no deadlocks are possible. Submit both the modified C-code and a description of what the modified Petri net would look like after introducing the changes to the C-code that you suggest. (2 p)

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 identifier, 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 become more familiar with the most important scheduling results. Please provide detailed comments with your code.

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. (1 p)

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. Provide your results in the report. Hint: See lecture slides on scheduling. (2 p)

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. (7 p)

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.

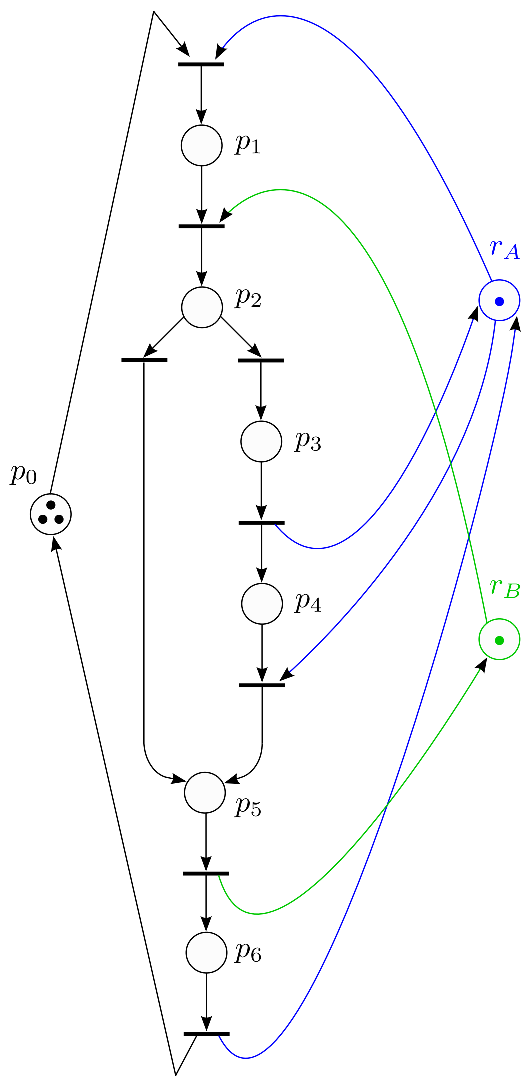

Figure 2: Example of a Petri net built for a part of the BIND software. From [1].

```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_Handle);
    xTaskCreate(the_task, "Task␣2",
       configMINIMAL_STACK_SIZE, NULL, 1, &
       task_b_handle);
    xTaskCreate(the_task, "Task␣3",
       configMINIMAL_STACK_SIZE, NULL, 1, &
       task_c_handle);

    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.