Operating Systems (Code: COMP2006)

CURTIN UNIVERSITY
Department of Computing

Multitasked Sudoku Solution Validator

Due Date: **4PM, Monday May 8, 2017**

Objective

The objective of this programming assignment is to give you some experiences in using multiple processes and multiple threads and their inter-process and inter-thread communications respectively. You will learn how to create processes and threads and solve the critical section problems. Note that the project is a modified version of the Sudoku Solution Validator described in Chapter 4 of the unit's textbook: "Operating System Concepts" by Silberschatz, et al, pp. 197-199.

Assignment Description

Briefly, the Sudoku puzzle considers a (9 x 9) grid, and its correct/valid solution must have the following properties:

- Each column must contain all of the digits 1 to 9,
- Each row must contain all of the digits 1 to 9, and
- Each of the nine (3 x 3) sub-grid must contain all of the digits 1 to 9.

The following figure (**from Fig. 4.19 of the textbook on page 198**) shows one possible valid solution of Sudoku.

6	2	4	5	3	9	1	8	7
5	1	9	7	2	8	6	3	4
8	3	7	6	1	4	2	9	5
1	4	3	8	6	5	7	2	9
9	5	8	2	4	7	3	6	1
7	6	2	3	9	1	4	5	8
3	7	1	9	5	6	8	4	2
4	9	6	1	8	2	5	7	3
2	8	5	4	7	3	9	1	6

The Sudoku Solution Validator determines if a solution to the Sudoku puzzle is valid. One possible way of multitasking the validator is to create eleven tasks (i.e., processes or threads for multi-processes or multi-threads implementation, respectively) that together check the following Sudoku properties:

- Group-1: Nine tasks, each of which checks the validity of each of the nine rows.
- Group-2: One task to check the validity of each of the nine columns.

- Group-3: One task to check the validity of all the nine (3 x 3) sub-grids.

In this project, you are asked to write a program in C language that implements a Multitasked Sudoku Solution Validator (**mssv**) using the described eleven **child** tasks. Specifically, given a solution of the Sudoku puzzle, the **parent** task and the child tasks do the following steps.

Parent task

1) Create shared memory **Buffer1**, **Buffer2**, and **Counter**.

   Buffer1 is to store the (9 x 9) Sudoku solution from which each child task reads its assigned input. Note that the parent task fills in **Buffer1** from a given **input file**.

   Buffer2 is a (1 x 11) array of integers that is used by all child tasks to store the total number of valid sub-grids they have found. For the example, Buffer2[1] = 1 and Buffer2[10] = 9.

   Counter is an integer (initialized to 0) that contains the total number of valid sub-grids. E.g., **Counter** = 27 if nine rows, nine columns, and nine (3 x 3) sub-grids are all valid.

2) Create eleven **child** tasks.

3) Assign each child task a **region** of the Sudoku solution to validate. Note that for each child task in Group-1, a region is each row, while for both tasks in Group-2 and Group-3, the region is the (9 x 9) grid. For this step, you need to use a data structure that represents each region, and make the parent process to pass the data structure to each child task.

4) Wait for the results from the eleven child tasks. The parent task **blocks** while waiting.

5) Print all validation results (i.e., valid or invalid), and the total number of "valid" results to screen.

   For the previous example, the parent task shows the following (ID-*i*, refers to the ID of process *i*) on the screen:

   Validation result from process ID-1: row 1 is valid
   Validation result from process ID-2: row 2 is valid
   Validation result from process ID-3: row 3 is valid
   Validation result from process ID-4: row 4 is valid
   Validation result from process ID-5: row 5 is valid
   Validation result from process ID-6: row 6 is valid
   Validation result from process ID-7: row 7 is valid
   Validation result from process ID-8: row 8 is valid
   Validation result from process ID-9: row 9 is valid
   Validation result from process ID-10: 9 of 9 columns are valid
   Validation result from process ID-11: 9 of 9 sub-grids are valid

   There are 27 valid sub-grids, and thus the solution is valid.

6) Remove all created resources, e.g., semaphores, shared memory, child tasks, etc.

7) Terminate.

Child task

Each child task *i* performs the following steps after receiving its assigned region.

1) Check all sub-grids in the assigned region. You may need to include some time delay to be able to observe any synchronization issue that can generate inconsistent results. So, you need to include a *sleep* (*delay*) at the end of this step, where delay is a random number between 1 to **maxdelay**. Note that you provide the value of **maxdelay** when you run your program; its suggested value is 5.

2) Store all sub-grids that it found invalid into a **log** file.

   For the above example, no child task stores any invalid sub-grids into the log file. However, if row 4, columns 5 and 8, and sub grid [1..3, 4..6] are invalid, the log file should contain:

   process ID-4: row 4 is invalid
   process ID-10: column 5, 8 are invalid
   process ID-11: sub-grid [1..3, 4..6] *is valid*

3) Store the total number of valid sub-grids it has found in **Buffer2 [*i*]**; Let *#valid* be the number.

4) Update **Counter**, i.e., **Counter = Counter +** *#valid*

   E.g., the task that checks the nine columns should increase **Counter** by 9 (or 7) if it found 9 (or 7) valid columns.

5) The last task that updates **Counter** should **wake up** the parent task.

6) Terminate.

Notice that access to **Buffer2** and **Counter** can be considered as the **producer-consumer problem**, i.e., the parent is the consumer while the children are the producers.

Let us call your executable program as **mssv**. Your program should be run as:

msvv input1 maxdelay

where input1 is the file that contains the Sudoku Solution to be validated.

Part A: Implementation using processes (50%)

Since processes do not share memory, the parent process needs to create **Buffer1**, **Buffer2**, and **Counter**.

Please read Chapter 3 of the textbook (Operating System Concepts by Silberschatz, et al.) to learn how to create shared memory, and Chapter 5 on how to use POSIX semaphores. Make sure that the parent process waits for the termination of all child processes and remove the shared memory, and semaphores.

Part B: Implementation using pthreads (30%)

As threads by default share memory, for this part you need not create shared memory. However, you must address the same synchronization issues. Use pthread_mutex_lock(), pthread_mutex_unlock(), pthread_cond_wait(), and pthread_cond_signal() in your program. Chapter 4 shows how to write a multithread C program, while Chapter 5 provides a discussion on Pthread synchronization.

Instruction for submission

1. Assignment submission is **compulsory**. Students will be penalized by a deduction of ten percent per calendar day for a late submission. **An assessment more than seven calendar days overdue will not be marked and will receive a mark of 0**.

2. You must (i) submit a hard copy of your assignment report to the unit box, (ii) submit the soft copy of the report to the unit Blackboard (**in one zip file**), and (iii) put your program files i.e., mssv.c, makefile, and other files, e.g., test input, in your home directory, under a directory named **OS/assignment**.

3. Your assignment report should include:

   - A signed cover page that includes the words "Operating Systems Assignment", and your name in the form: family, other names. Your name should be as recorded in the student database.

   - Software solution of your assignment that includes (i) all source code for the programs with proper in-line and header documentation. Use proper indentation so that your code can be easily read. Make sure that you use meaningful variable names, and delete all unnecessary comments that you created while debugging your program; and (ii) readme file that, among others, explains how to compile your program and how to run the program.

   - Detailed discussion on how any mutual exclusion is achieved and what processes / threads access the shared resources.

   - Description of any cases for which your program is not working correctly or how you test your program that make you believe it works perfectly.

- Sample inputs and outputs from your running programs.

Your report will be assessed (worth 20% of the overall assignment mark).

4. Due dates and other arrangements may only be altered with the consent of the majority of the students enrolled in the unit and with the consent of the lecturer.

5. Demo requirements:

   - You may be required to demonstrate your program and/or sit a quiz during tutorial sessions (to be announced).

   - For demo, you MUST keep the source code of your programs in your home directory, and the source code MUST be that submitted. The programs should run on any machine in the department labs.

Failure to meet these requirements may result in the assignment not being marked.