COMP3520 – Operating Systems Assignment 1 470425633

1. *Explain the purpose of a condition variable.*

A conditional provides a way in which one thread can wait for a signal from another thread before continuing execution. Some thread T_1 waits for a change in conditional variable C_1 before continuing execution and will wait until it is reawakened by some other thread T_2 signalling the conditional variable C_1. The conditional variable must be used in conjunction with a mutex to prevent undefined behaviour. This allows a programmer to create a thread, do some work, and then wait upon the completion of other work from another thread **without continuously polling the other thread(s).** This is the main point of separation from an ordinary mutex.

2. *Consider the function call pthread_cond_wait(&a_condition_variable, &a_mutex). Assume that a_condition_variable and a_mutex have been correctly initialized. Explain why it is an error to call this function if the mutex is unlocked just before the call.*

Assuming the mutex has been unlocked, this implies the calling thread does not own the mutex. In this scenario, the pthread_cond_wait function will fail as specified in the specification of "The pthread_cond_wait() function will fail if: … The specified mutex was not locked by the calling thread." (Freebsd, 2018) This occurs since the pthread_cond_wait usually releases the mutex and cannot do so if the mutex is unlocked.

3. *In addition to the pthread_cond_wait() function, the pthreads library offers a pthread_cond_timedwait() function that is useful for solving certain types of synchronization problems. Consult the Linux man page for a detailed description of this function. Describe ONE scenario where you would use the pthread_cond_timedwait() function. Justify your answer*

The pthread_cond_timedwait() operates in a similar fashion to the aforementioned pthread_cond_wait() with the additional ***const struct timespec *abstime*** argument. This argument allows the function to have additional behaviour, whereby if the system reaches the time specified in ***abstime*** and the thread re-acquires the lock on the mutex, the thread will unblock. Hence, in comparison to the regular pthread_cond_wait() call, this call offers the thread an alternative mechanism to unblock which may be useful in certain scenarios where the behaviour of other threads is not deterministic. This allows the programmer to regain control of the thread without explicit signalling from another thread.

Example

An example of where this subroutine would be useful can be within a program whereby one thread does work that is non-deterministic (such as interfacing with an external device).

Suppose the program is designed to read a file from disk and place the data into memory, and then print the file.

In this example the main program can create two threads. The first thread, T_1, is used to continuously poll the printer to see if it is available upon which time it will send the data. The second thread, T_2, places disk data into memory for T_1 to read and send to the printer. The pthread_cond_timedwait() function can be used here by both threads to ensure network connectivity and the file read process do not create a deadlock. T_1 would continuously poll the printer and when it has established a connection it will ask T_2 for confirmation that the data has been placed within memory by signalling a conditional variable and then waiting on confirmation. This wait of confirmation from T_1 can be a timedwait(), since if there is any errors within thread T_2 and it does not signal back, T_1 can return back to the main program with an error message. If this is not a timedwait(), T_1 can be blocked until the main program has timed out with no indication of what went wrong.

This can similarly be used by T_2 to wait on T_1 for establishing the connection. If T_1 does not return in an appropriate amount of time T_2 can signal back to the main thread the connection error of the printer.

4. *In English and pseudocode, describe and justify the algorithm that you have used to solve the traffic light synchronization problem.*

This program can be split up into 3 parts. The main program, the vehicle routine, and the traffic light controller routine.

Main
The main program manages the traffic light simulation. When the program is called, there is the optionality of specifying a second argument. If this argument is "debug" the main program initialises a Boolean which allows for print statements which are easier to debug.

Aside from this, the main program initialises 3 mini controller threads and n vehicle threads as supplied by user input. The mini controller threads are fairly simplistic to initialise, however the vehicle threads require a uniform distribution amongst the directions and has the condition that two vehicles travelling in the same direction can be no closer than 1 second apart.

The uniform distribution problem was solved using a function written by a user on Stackoverflow **JXH**. The implementation allowed me to get a fairly uniform distribution across 0-5 for my directions more so than a simple **rand() % 6** would do by using decimals.

The sleep time was allocated using the rand function and controlled vehicles travelling in the same direction as follows:

```
if (! first_vehicle) {
  if (direction == previous_direction && sleep_time == 0) {
    sleep_time = 1
  }
}
```

Following the creation of all vehicle and controller threads, the main program waited on all vehicle threads completion using a join, and then cancelled the controller threads as they would not finish on their own. The finish the program all memory was deallocated, and the conditional variables were destroyed.

Vehicle Thread

The vehicle thread served to allow the direction of the vehicle to wait on a specific controller and direction before proceeding through the intersection. If the said direction is "green", the vehicle proceeds immediately – otherwise it joins a waiting queue. When the vehicle thread is spawned, the function uses the struct pointer to retrieve the vehicle id and direction. The thread then prints that the vehicle has arrived along with an **id** and **direction**. This direction is then used to control the behaviour of the vehicle. For each given direction, there are conditional variables allowing the vehicle to signal to traffic light controller, and signals for the traffic light controller to signal the vehicle back. There is also a counter (protected by a mutex), allowing the traffic light controller to be aware of the number vehicles waiting. In simple terms, this is how it works:

```
if (direction == n2s) {
```

```
if (direction == n2s) {
        // check if the number of vehicles waiting is 0. If it is, signal the controller and proceed, else, wait for controller
        mutex_lock(n2s_mutex_counter);
        n2s_counter++
        mutex_unlock(n2s_mutex_counter);
        mutex_lock(controller_1_mutex_n2s);
        signal(controller_1_waiting_for_vehicles_n2s);
        wait(vehicles_waiting_on_c1_n2s, controller_1_mutex_n2s);
        while (1) {
            //check that it is in fact next in line, otherwise wait
            if (id == last_vehicle_n2s+1) {
                last_vehicle_n2s++;
                print_vehicle_proceeding(id, direction, debug_mode);
                mutex_unlock(controller_1_mutex_n2s);
                return
            } else {
                wait(&vehicles_waiting_on_c1_n2s, &controller_1_mutex_n2s);
            }
        }
}
```

Hence, using the direction of the vehicle, the program signals an appropriate controller. That controller is either then woken, or will notice the increase in the number of vehicles waiting and signal back. When the controller thread then signalled these waiting threads, they would check if they are next in the order of vehicles to proceed and then proceed, otherwise they would wait on the variable again. Its worth noting that all **`previous_in_this_direction`** variables where initialised as -1. This routine thus allowed for:

- For each direction, the thread would wait on the appropriate conditional variable.
- For vehicles arriving at a light whilst it is green, they proceed immediately
- If it is the current threads turn to proceed, it would do so then break the loop.
- Otherwise, the thread would simply wait on the conditional variable again until it's the current threads turn

The print_vehicle_proceeding function was used, allowing the program to print additional details (such as time in seconds) when the program was in debug mode.

The bottom of the thread would then just exit.

Controller Thread

The controller thread functioned in a similar way to the vehicle threads, it functioned dependent on the id of the thread. Since (n2s, s2n) was the default first direction, when this direction thread was called, it automatically identified the lights as green and signalled all vehicles moving in that direction. Following this, it locked the mutex, and signalled for controller 2 that it was their turn. The pseudocode for this was as follows:

```
else if (id == 1) {
        while (1) {
            mutex_lock(green_mutex);
            wait(controller_2, green_mutex);
            mutex_unlock(green_mutex);
            sleep(2);
            print_lights(green , directions, debug_mode);
            create_signal_threads(time_green, min_interval, controller_2_mutex_e2w, controller_2_mutex_w2e,
            vehicles_waiting_on_c2_e2w, vehicles_waiting_on_c2_w2e, controller_2_waiting_for_vehicles_e2w,
            controller_2_waiting_for_vehicles_w2e, e2w_counter, w2e_counter, e2w_mutex_counter, w2e_mutex_counter);
            print_lights(red , directions, debug_mode);

            mutex_lock(&green_mutex);
            cond_signal(&controller_3);
            mutex_unlock(&green_mutex);

        }
    }
```

The main work here is done by an external function which takes in all required parameters to decided how many vehicles to allow to pass, and then to return in the correct amount of time. This function spawned 2 threads each thread responsible for waiting on a given direction, signalling between the controller and the vehicles waiting and to appropriately pass the correct number of vehicles through. The thread spawn function worked as follows:

```
void spawn_threads() {
  pthreads_create(*direction_1_relevant_information_struct)
  pthreads_create(*direction_2_relevant_information_struct)
  sleep(allowed_green_time)
  cancel(thread_1)
  cancel(thread_2)
}
```

This function would take the time the light is allowed to be green, the minimum interval between 2 vehicles travelling in the same direction, the mutex for that set of lights, and the two conditionals corresponding to the two directions vehicles travelled at said lights and handled the signalling and sleeping process. The threads created essentially entered a continuously running loop where it checked if there were any remaining vehicles to pass through that intersection, and signal back to let them through, otherwise wait on a conditional variable for a vehicle to arrive.

```
while (1) {

            mutex_lock(scounter_mutex);
            int vehicles_waiting = counter;
            mutex_unlock(counter_mutex);

            if (vehicles_waiting) {
                mutex_lock(direction_mutex);
                cond_signal(controller_signals);
                mutex_unlock(direction_mutex);
               mutex_lock(counter_mutex);
                counter = counter - 1;
                mutex_unlock(counter_mutex);
                sleep(min_interval);
            } else {
                mutex_lock(direction_mutex);
                cond_wait(controller_waits, direction_mutex);
                cond_signal(controller_signals);
               mutex_unlock(direction_mutex);
                mutex_lock(counter_mutex);
                counter = counter - 1;
                mutex_unlock(counter_mutex);
                sleep(min_interval);
            }
    }
```

Here is some pseudocode describing how this program works. To summarise:

- Checks if there are vehicles waiting, if there are, it signals the vehicle to proceed, then waits for the minimum interval
- If there are not, it goes into a waiting state to be awoken by a vehicle. If it is awoken, it signals back to the vehicle to proceed.

Debugging

In terms of debugging this program there were several features and characteristics which had to be tested individually to ensure the program would function as desired. Due to the nature of what the program did, writing modifiable and flexible test cases was decided to not be the most ideal option. If the program grew in size and complexity code driven test cases could potentially be written, however at this stage manual inspection of the output of the program was deemed suitable.

That being said, the program was written with a feature which allowed for an optional “debug” statement, which altered the way in which the program output. Specifically, this allowed me to display the time in seconds, along with a shorthand version of the original outputs. For example:

```
void print_vehicle_proceeding(id, direction) {
    if (debug_mode) {
       string current_time = get_current_time();
        printf("time:vehicle:direction:direction:proceeding");
    } else {
        printf("Vehicle %d %s is proceeding through the intersection.\n", id, direction);
    }
}
```

This being said, there were ultimately a few conditions which had to be tested to be true. This included:

- Vehicles of a given direction only proceeded once their light was green
- Vehicles going in a certain direction had at least the minimum interval between them.
- Vehicles of a certain direction proceeded through the intersection in their specified order of creation (using their ids)
- Once all vehicles had passed through, the program would terminate.
- The traffic lights iterated correctly in the right order with specified 2 second waiting time between being green.
- The lights were green for their allotted time of being green.
- If a vehicle arrived at an intersection whilst the light was green, and no other were waiting it would proceed immediately

In order the allow for ease of inspection, the output of the program was redirected into a text file:

```
./traffic_lights debug < input.txt > output.txt
```

The input.txt file contained all the given parameters for the program (which was usually asked for as input) allowing the output to be consistently inspected.

In order to inspect for all given characteristics, we can specify what we expect to see, and then inspect our output to assure this is true.

Input 1

```
10  //n_vehicles
2  //arrival rate
2  //min time between 2 vehicles
3  //controller 1 green_time
3  //c_2 green time
3  //c_3 green time
```

Output 1

In this output we expect to see at least 2 pairs of cars proceeding though one direction of intersection once the light as green (2 second wait between pairs) assuming the vehicles had arrived. Hence, we set the arrival rate low, and the green time just under the threshold of 4 (2 times 2 seconds)

```
38:lights:(n2s, s2n):green
39:vehicle:0:n2s:arrive
39:vehicle:0:n2s:proceeding
40:vehicle:0:s2n:arrive
40:vehicle:0:e2w:arrive
40:vehicle:0:s2n:proceeding
40:vehicle:0:n2w:arrive
41:lights:(n2s, s2n):red
41:vehicle:0:s2e:arrive
41:vehicle:0:w2e:arrive
42:vehicle:1:n2s:arrive
43:lights:(e2w, w2e):green
43:vehicle:0:w2e:proceeding
43:vehicle:0:e2w:proceeding
43:vehicle:1:w2e:arrive
```

```
44:vehicle:2:n2s:arrive
45:vehicle:1:w2e:proceeding
45:vehicle:3:n2s:arrive
46:lights:(e2w, w2e):red
48:lights:(n2w, s2e):green
48:vehicle:0:n2w:proceeding
48:vehicle:0:s2e:proceeding
51:lights:(n2w, s2e):red
53:lights:(n2s, s2n):green
53:vehicle:1:n2s:proceeding
55:vehicle:2:n2s:proceeding
56:lights:(n2s, s2n):red
58:lights:(e2w, w2e):green
01:lights:(e2w, w2e):red
03:lights:(n2w, s2e):green
06:lights:(n2w, s2e):red
08:lights:(n2s, s2n):green
08:vehicle:3:n2s:proceeding
Main thread: There are no more vehicles to serve. The simulation will end now.
```

This output was sufficient to demonstrate:

- If a vehicle arrives and the lights are green with no other waiting, the vehicle would proceed.
- If the minimum interval between vehicles is effective
    - If more than 2 vehicles waiting, they proceed immediately, then wait the minimum interval then proceed.
- Vehicles using the same light but travelling in opposite directions operate independently
- 2 second delay between the lights
- That vehicles proceed in the given order of their ids
- That the lights effectively iterate between each other and the program terminates once all the vehicles have passed

Input 2

Another concept to test would be to see how the program handles the lights having a green time less than the minimum interval.

- This allows us to identify that only 1 vehicle would proceed through any given direction in the period the light is green

```
10
2
3
2
2
2
```

Output 2

```
00:lights:(n2s, s2n):green
01:vehicle:0:n2s:arrive
01:vehicle:0:n2s:proceeding
02:lights:(n2s, s2n):red
02:vehicle:0:s2n:arrive
02:vehicle:0:e2w:arrive
```

```
02:vehicle:0:n2w:arrive
03:vehicle:0:s2e:arrive
03:vehicle:0:w2e:arrive
04:lights:(e2w, w2e):green
04:vehicle:0:e2w:proceeding
04:vehicle:0:w2e:proceeding
05:vehicle:1:n2s:arrive
06:vehicle:1:w2e:arrive
06:lights:(e2w, w2e):red
07:vehicle:2:n2s:arrive
08:vehicle:3:n2s:arrive
08:lights:(n2w, s2e):green
08:vehicle:0:n2w:proceeding
08:vehicle:0:s2e:proceeding
10:lights:(n2w, s2e):red
12:lights:(n2s, s2n):green
12:vehicle:1:n2s:proceeding
15:vehicle:0:s2n:proceeding
15:lights:(n2s, s2n):red
17:lights:(e2w, w2e):green
17:vehicle:1:w2e:proceeding
19:lights:(e2w, w2e):red
21:lights:(n2w, s2e):green
23:lights:(n2w, s2e):red
25:lights:(n2s, s2n):green
25:vehicle:2:n2s:proceeding
27:lights:(n2s, s2n):red
29:lights:(e2w, w2e):green
31:lights:(e2w, w2e):red
33:lights:(n2w, s2e):green
35:lights:(n2w, s2e):red
37:lights:(n2s, s2n):green
37:vehicle:3:n2s:proceeding
```

Here we can see the minimum interval and the traffic light green time work together properly and no more than 1 vehicle proceeds through the intersection during a green period.

Input 3

The last test to ensure the program would be to stress test the edge conditions. This included:

- The same minimum interval and green time
- Lots of vehicles (100)
- Quick arrival rate

```
50
1
1
1
1
1
```

Output 3

Disappointingly, the below output has a few issues. At times a vehicle proceeds right after the lights were turned red. Its difficult to identify exactly hat has caused this issue but it can be assumed there is some delays between the controller signalling the vehicle to proceed and the lights turning red.

```
37:vehicle:0:n2s:arrive
37:vehicle:0:s2e:arrive
37:lights:(n2s, s2n):green
37:vehicle:0:e2w:arrive
37:vehicle:0:w2e:arrive
```

```
37:vehicle:0:n2w:arrive
37:vehicle:0:s2n:arrive
37:vehicle:0:n2s:proceeding
37:vehicle:0:s2n:proceeding
38:lights:(n2s, s2n):red
38:vehicle:1:n2s:arrive
38:vehicle:1:w2e:arrive
39:vehicle:2:n2s:arrive
40:lights:(e2w, w2e):green
40:vehicle:0:e2w:proceeding
40:vehicle:0:w2e:proceeding
40:vehicle:3:n2s:arrive
40:vehicle:1:n2w:arrive
40:vehicle:1:s2e:arrive
40:vehicle:2:w2e:arrive
40:vehicle:3:w2e:arrive
40:vehicle:2:n2w:arrive
40:vehicle:3:n2w:arrive
41:lights:(e2w, w2e):red
41:vehicle:1:w2e:proceeding
41:vehicle:4:n2s:arrive
41:vehicle:1:s2n:arrive
41:vehicle:4:n2w:arrive
41:vehicle:1:e2w:arrive
41:vehicle:2:s2e:arrive
41:vehicle:5:n2w:arrive
42:vehicle:5:n2s:arrive
42:vehicle:3:s2e:arrive
42:vehicle:2:e2w:arrive
42:vehicle:3:e2w:arrive
42:vehicle:2:s2n:arrive
43:lights:(n2w, s2e):green
43:vehicle:0:n2w:proceeding
43:vehicle:0:s2e:proceeding
43:vehicle:4:s2e:arrive
43:vehicle:6:n2s:arrive
44:lights:(n2w, s2e):red
44:vehicle:1:s2e:proceeding
44:vehicle:1:n2w:proceeding
44:vehicle:7:n2s:arrive
44:vehicle:4:w2e:arrive
44:vehicle:3:s2n:arrive
44:vehicle:4:e2w:arrive
45:vehicle:8:n2s:arrive
46:lights:(n2s, s2n):green
46:vehicle:1:s2n:proceeding
46:vehicle:9:n2s:arrive
46:vehicle:5:e2w:arrive
46:vehicle:6:e2w:arrive
46:vehicle:5:s2e:arrive
47:vehicle:2:s2n:proceeding
47:lights:(n2s, s2n):red
47:vehicle:1:n2s:proceeding
47:vehicle:6:n2w:arrive
47:vehicle:10:n2s:arrive
48:vehicle:11:n2s:arrive
48:vehicle:5:w2e:arrive
48:vehicle:7:n2w:arrive
48:vehicle:8:n2w:arrive
48:vehicle:6:s2e:arrive
48:vehicle:4:s2n:arrive
48:vehicle:6:w2e:arrive
48:vehicle:7:s2e:arrive
48:vehicle:8:s2e:arrive
48:vehicle:7:e2w:arrive
49:lights:(e2w, w2e):green
```

```
49:vehicle:2:w2e:proceeding
49:vehicle:1:e2w:proceeding
50:vehicle:2:e2w:proceeding
50:vehicle:3:w2e:proceeding
50:lights:(e2w, w2e):red
52:lights:(n2w, s2e):green
52:vehicle:2:n2w:proceeding
52:vehicle:2:s2e:proceeding
53:vehicle:3:s2e:proceeding
53:lights:(n2w, s2e):red
53:vehicle:3:n2w:proceeding
55:lights:(n2s, s2n):green
55:vehicle:2:n2s:proceeding
55:vehicle:3:s2n:proceeding
56:lights:(n2s, s2n):red
56:vehicle:3:n2s:proceeding
56:vehicle:4:s2n:proceeding
58:lights:(e2w, w2e):green
58:vehicle:3:e2w:proceeding
58:vehicle:4:w2e:proceeding
59:lights:(e2w, w2e):red
59:vehicle:4:e2w:proceeding
59:vehicle:5:w2e:proceeding
01:lights:(n2w, s2e):green
01:vehicle:4:n2w:proceeding
01:vehicle:4:s2e:proceeding
02:vehicle:5:n2w:proceeding
02:vehicle:5:s2e:proceeding
02:lights:(n2w, s2e):red
04:lights:(n2s, s2n):green
04:vehicle:4:n2s:proceeding
05:lights:(n2s, s2n):red
05:vehicle:5:n2s:proceeding
07:lights:(e2w, w2e):green
07:vehicle:5:e2w:proceeding
07:vehicle:6:w2e:proceeding
08:vehicle:6:e2w:proceeding
08:lights:(e2w, w2e):red
10:lights:(n2w, s2e):green
10:vehicle:6:s2e:proceeding
10:vehicle:6:n2w:proceeding
11:vehicle:7:s2e:proceeding
11:vehicle:7:n2w:proceeding
11:lights:(n2w, s2e):red
13:lights:(n2s, s2n):green
13:vehicle:6:n2s:proceeding
14:vehicle:7:n2s:proceeding
14:lights:(n2s, s2n):red
16:lights:(e2w, w2e):green
16:vehicle:7:e2w:proceeding
17:lights:(e2w, w2e):red
19:lights:(n2w, s2e):green
19:vehicle:8:n2w:proceeding
19:vehicle:8:s2e:proceeding
20:lights:(n2w, s2e):red
22:lights:(n2s, s2n):green
22:vehicle:8:n2s:proceeding
23:vehicle:9:n2s:proceeding
23:lights:(n2s, s2n):red
25:lights:(e2w, w2e):green
26:lights:(e2w, w2e):red
28:lights:(n2w, s2e):green
29:lights:(n2w, s2e):red
31:lights:(n2s, s2n):green
31:vehicle:10:n2s:proceeding
32:lights:(n2s, s2n):red
```

```
32:vehicle:11:n2s:proceeding
Main thread: There are no more vehicles to serve. The simulation will end no
```

Reference List

- **pthread_cond_wait(3)*. (2019).* Freebsd.org*. Retrieved 8 September 2019, from https://www.freebsd.org/cgi/man.cgi?query=pthread_cond_wait&apropos=0&sektion=3&manpath=FreeBSD+12.0-RELEASE&arch=default&format=html***

- ***JXH (2012).* Generating a uniform distribution of INTEGERS in C*.* Stack Overflow*. Retrieved 8 September 2019, from https://stackoverflow.com/questions/11641629/generating-a-uniform-distribution-of-integers-in-c***