CompSci 3SH3, Winter 2021
Student Name: Jatin Chowdhary
Student Number: 400033011
Mac ID: Chowdhaj
Date: March 21st, 2021

ASSIGNMENT #2

THREADS

1. a) Which of the following components of program state are shared across threads in a multithreaded process?

i) Register values: Not shared
ii) Heap memory: Shared
iii) Global variables: Shared
iv) Stack memory: Not shared

1. b) Is it possible to have concurrency but not parallelism? Explain.

Yes, it is possible to have concurrency but not parallelism. This is because concurrency and parallelism are two different things. Concurrency is executing multiple threads on the same core/processor, and parallelism is executing multiple threads on different processors at the same time; each thread is executed on its own core/processor. Parallelism is only possible on multiple cores/processors; you need at least 2 CPUs to parallelize something. As the name implies, threads run in parallel. Therefore, it is possible to have concurrency, but not parallelism. Furthermore, concurrency is possible on a single core/processor due to switching. The CPU is able to switch between threads so fast that it appears as if the threads are simultaneously making progress.

1. c) Using Amdahl's Law, calculate the speedup gain for the following applications:

i) 40 percent parallel with
(a) eight processing cores

N = 8
P = 0.4
S = 1 - P
  = 1 - 0.4
  = 0.6

Speedup <= 1 / (S + ((1 - S) / N))
        <= 1 / (0.6 + ((1 - 0.6) / 8))
        <= 1 / (0.6 + (0.4 / 8))
        <= 1 / (0.6 + 0.05)
        <= 1 / (0.65)

```
        <= 1.538461538

The speedup is about 1.54

(b) sixteen processing cores

N = 16
P = 0.4
S = 1 - P
  = 1 - 0.4
  = 0.6

Speedup <= 1 / (S + ((1 - S) / N))
        <= 1 / (0.6 + ((1 - 0.6) / 16))
        <= 1 / (0.6 + (0.4 / 16))
        <= 1 / (0.6 + 0.025)
        <= 1 / (0.625)
        <= 1.6

The speedup is about 1.60

ii) 67 percent parallel
(a) two processing cores

N = 2
P = 0.67
S = 1 - P
  = 1 - 0.67
  = 0.33

Speedup <= 1 / (S + ((1 - S) / N))
        <= 1 / (0.33 + ((1 - 0.33) / 2))
        <= 1 / (0.33 + (0.67 / 2))
        <= 1 / (0.33 + 0.335)
        <= 1 / (0.665)
        <= 1.503759398

The speedup is about 1.50

(b) four processing cores

N = 4
P = 0.67
S = 1 - P
  = 1 - 0.67
  = 0.33

Speedup <= 1 / (S + ((1 - S) / N))
        <= 1 / (0.33 + ((1 - 0.33) / 4))
        <= 1 / (0.33 + (0.67 / 4))
```

```
        <= 1 / (0.33 + 0.1675)
        <= 1 / (0.4975)
        <= 2.010050251
```

The speedup is about 2.01

iii) 90 percent parallel
(a) four processing cores

```
N = 4
P = 0.9
S = 1 - P
  = 1 - 0.9
  = 0.1

Speedup <= 1 / (S + ((1 - S) / N))
        <= 1 / (0.1 + ((1 - 0.1) / 4))
        <= 1 / (0.1 + (0.9 / 4))
        <= 1 / (0.1 + 0.225)
        <= 1 / (0.325)
        <= 3.076923077
```

The speedup is about 3.08

(b) eight processing cores

```
N = 8
P = 0.9
S = 1 - P
  = 1 - 0.9
  = 0.1

Speedup <= 1 / (S + ((1 - S) / N))
        <= 1 / (0.1 + ((1 - 0.1) / 8))
        <= 1 / (0.1 + (0.9 / 8))
        <= 1 / (0.1 + 0.1125)
        <= 1 / (0.2125)
        <= 4.705882353
```

The speedup is about 4.71

1.d)

i) How many threads will you create to perform the input and output? Explain.

To handle the input and output of this program, I only need 1 thread, in total. A single thread will handle both input and output. This is because input and output are serial actions. Input cannot occur after output, and output cannot occur before input. These two actions must

be done sequentially; input -> output. Assuming the data is processed in a dependent manner, the input and output actions cannot be done at the same time (i.e. Process 1 input value, and then output it). Therefore, I will use a single thread to handle both input and output. It does not make sense to have two different single threads to handle input and output, when the actions are done sequentially, and cannot be parallelized.

ii) How many threads will you create for the CPU-intensive portion of the application? Explain.

For the CPU-intensive portion of the application, I will create as many threads as possible based on two factors:

A) How well the application can be parallelized
B) Maximum number of threads that can be created without adversly affecting the entire system.

To make this problem easier, let's assume that the application can be completely parallelized, and that the maximum number of threads that can be created to process the application is 4; identical to the number of cores on the system. Then, I will create 3 additional threads to speedup the application. In total there will be 4 threads working on the application. For example:

```
Thread 1) Input -> Process Data -> Output
Thread 2)          Process Data
Thread 3)          Process Data
Thread 4)          Process Data
```

The first thread handles input, output, and it processes the data along with the other 3 threads. In total there are 4 threads; 1 for every core the system has.

Furthermore, if the CPU cores support hyper-threading, then I will create 7 additional threads in addition to the first thread. For example:

```
Thread 1) Input -> Process Data -> Output
Thread 2)          Process Data
Thread 3)          Process Data
Thread 4)          Process Data
Thread 5)          Process Data
Thread 6)          Process Data
Thread 7)          Process Data
Thread 8)          Process Data
```

SYNCHRONIZATION

2. a. i) Identify the race condition(s)

A race condition can occur due to the following lines:

```
    A) ++number_of_processes;
    B) --number_of_processes;
```
A race condition is caused when this variable (number_of_processes) is mutated; incremented/decremented.

Furthermore, the following line can also be problematic:
```
    C) if (number_of_processes == MAX_PROCESSES)
```
Assume that `MAX_PROCESSES` is 254. If two threads are created, and the first thread stops executing just before the increment statement, and the second thread executes to the end of 'allocate_process()', then the first thread will increment the counter, and cause it to exceed the max.

2. a. ii) Assume you have a mutex lock named mutex with the operations acquire() and release(). Indicate where the locking needs to be placed to prevent the race condition(s).

The 'acquire()' and 'release()' functions can be placed before and after the variable `number_of_processes` is mutated. For example:

```
     ...
    acquire();
    ++number_of_processes;
    release();
     ...
    acquire();
    --number_of_processes;
    release();
     ...
```

To combat the potential read race condition caused by the if-statement, the 'acquire()' and 'release()' functions can be placed at the beginning and end of the function(s). For example:

```
     ...
    int allocate_process() {
        acquire();
         ...
        release();
    }
     ...
```

2. a. iii) Could we replace the integer variable:
`int number of processes = 0`
with the atomic integer:
`atomic_t number of processes = 0`
to prevent the race condition(s)?

Yes, using the atomic integer can prevent the race conditions that occur when the variable 'number_of_processes' is mutated. The atomic

integer will ensure indivisibility via the hardware, and prevent 'number_of_processes' from being written simultaneously.
However, in certain situations, this may not work (i.e. If multiple processes are created/terminated in succession, and all of them want to operate on `number_of_processes`). But for this question, this scenario is out of context.

2. b) Design an algorithm (using pseudocode in Listing 2) for a monitor that implements an alarm clock that enables a calling program to delay itself for a specified number of time units (ticks).

```
monitor alarm {

    condition c;

    void delay(int ticks) {
        // TODO: Implement delay
        /*
         * Pseudo-code:
         * - Create a new integer called 'i'
         * - Iterate over the parameter 'ticks'
         * - For every iteration, call the 'tick()'
         *   function
         * - Iterate until 'i' == 'ticks'
         *
         * OR
         *
         * - Invoke the 'tick()' function until the
         *   condition 'c' is met
         */
    }

    void tick() {
        // TODO: Implement tick
        /*
         * Pseudo-code:
         * - Call the hardware method 'tick()' to
         *   stall the program
         * - Return to caller once operation is complete
         */
    }
}
```

DEADLOCK

3. a. i) What is the content of the matrix Need?

The need for a thread is given by the following equation:
Need[Thread] = Max[Thread] - Allocation[Thread]

The content of the matrix Need is:

	Need			
	A	B	C	D
T0	0	0	1	2
T1	0	7	5	0
T2	1	0	0	2
T3	0	0	2	0
T4	0	6	4	2

3. a. ii) Is the system in a safe state?

No, the system is in an un-safe state. This is because only 'T3's request for resources can be satisfied. The other threads' needs cannot be met with the current available resources.
T0's needs cannot be met because it needs 2 D's and 0 are available
T1's needs cannot be met because it needs 7 A's and 0 are available
T2's needs cannot be met because it needs 4 D's and 0 are available
T4's needs cannot be met because it needs 4 D's and 0 are available
Therefore, the system is in an un-safe state.

3. a. iii) If a request from thread T1 arrives for (0,4,2,0), can the request be granted immediately?

Yes, this request can be immediately granted because the availability is (1,5,2,0), and the request is for (0,4,2,0). Subtracting these two gives us: (1,5,2,0) - (0,4,2,0) = (1,1,0,0). Thus, this request can be immediately satisfied.

3. b) Which of the four resource-allocation graphs shown in Figure 1 illustrate deadlock? For those situations that are deadlocked, provide the cycle of threads and resources. Where there is not a deadlock situation, illustrate the order in which the threads may complete execution.

The graphs (B) & (D) are deadlocked.
- The cycle in graph (B) is: T1 -> R3 -> T3 -> R1 -> T1
  (T1 -> T3 -> T1)
- The cycles in graph (D) are: R1 -> T1 -> R2 -> T3 -> R1
  (T1 -> T3 -> T1)
  R1 -> T2 -> T2 -> T4 -> R1
  (T2 -> T4 -> T2)

The graphs (A) & (C) are NOT deadlocked.
- In graph (A), T2 can finish because it has R2. Once it finishes, it will release R2, and T3 or T1 can acquire R2 and use it to finish. If T3 acquires it first, it will finish, and then T1 will acquire R2, and use it to finish along with R1.
- In graph (C), T2 and T3 can finish because both of them have 1 instance of R1 and R2. They do not require any more resources, thus they can finish. Once one of {T2, T3} gives up their instance of R1 and R2, T1 can acquire those instances and use it to finish.