COMPSCI 3SH3 Fall, 2022
By: Nathan Agbomedarho
Student ID: 400081762
Date: November 13th 2022

Assignment 2

Threads

a. Question 4.10 Heap memory and global variables are the only things shared between threads of a multithreaded process.

B. Question 4.13 Concurrency occurs when two or more threads are executing in a multithreaded process. Parallelism occurs when at least two thread execute simultaneously. A single core may appear to be capable of parallelism by alternating between a single thread, it's not possible to achieve parallelism in the absence of concurrency, however, you can achieve concurrency without parallelism.

C. Question 4.14

Amdahl's Law = $\frac{1}{((1-p)+p/s)}$ where p = code that can be parallelized & s = number of cores.

a. 8 cores: p = 0.4, s = 8
speedup = $\frac{1}{((1-0.4)+0.4/8} = 1.53$
b.
16 cores: p = 0.4, s = 16
speedup = $\frac{1}{((1-0.4)+0.4/16} = 1.6$

ii. 67% parallel:
a.
2 cores: p = 0.67, s = 2
speedup = $\frac{1}{((1-0.67)+0.67/2} = 1.50$

b.
4 cores: p = 0.67, s = 4
speedup = $\frac{1}{((1-0.67)+0.67/4} = 2.01$

iii. 90% parallel
a.
4 cores: p = 0.9, s = 4
speedup = $\frac{1}{((1-0.9)+0.9/4} = 3.076$

b.
8 cores: p = 0.9, s = 8
speedup = $\frac{1}{((1-0.9)+0.9/8} = 4.70$

D. Question 4.14

i. To achieve this, I would create 1 thread to handle input and output operations, as the number of threads required is dependent on their scheduling priority and the immediate needs of the application. For input and output, only 1 thread is required, as the speed of reading/writing operations on a single file can't be enhanced using multiple threads; any more threads would be relegated to waiting for the completion of the I/O processes.

ii. I would make 4 threads, for the parts of the application that are CPU-intensive; as these parts of the application need to be using as many threads equivalent to the amount of processing cores. Fewer threads are wasteful of the computer's resources, and more would result in inefficient processing and potential instability of the operating system architecture.

Synchronization

Question 6.22

a. Race conditions occur when there are two or more threads attempting to access a shared variable simultaneously. As such, the race condition in this piece of code is: *number_of_processes*, and both *allocate_process()* and *release_process()* access and modify this variable.

b. In order for us to prevent race conditions from occurring, we make use of a mutex lock to call *acquire()* once we enter *allocate_process()* or *release_process* and then call *release()* before exiting either function.

c. This would not prevent the race conditions. In the function *allocate_process*, race conditions happen because *number_of_processes* is verified in a condition and incremented based on the results of the conditional statement. As such, a case can occur wherein this variable is 20 at the conditional, however because of the race condition it is set to 21 by another thread before being incremented by the original thread.

Question 6.31

```
monitor alarm {
    condition c;
    int cTick = 0;

    void delay(int ticks){
        int wakeTime = cTicks + ticks;
```

```
            while(cTick < wakeTime){
                delay.wait(wakeTime);
            }
            delay.signal;
    }
    void tick(){
        cTick += 1;
        delay.signal;
    }
}
```

Deadlock

a. Question 8.3

a. Need Matrix:

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

b. The available matrix is [1520]. Safety sequence condition: Need Available; Need(T0) $\leq$ Available as such we select T0(available) = (available) + allocation(P0).

Available = [1520] + [0012] = [1532]

Need(T2) $\leq$ Available $\rightarrow$ Available = [1 5 3 2] + [1 3 5 4] = [2 8 8 6]
Need(T3) $\leq$ Available $\rightarrow$ Available = [2 8 8 6] + [0 6 3 2] = [2 14 11 8]
Need(T4) $\leq$ Available$\rightarrow$ Available = [2 14 11 8] + [0014] = [2 14 12 12]
Need(T1) $\leq$ Available $\rightarrow$ Available = [2 14 12 12] + [1 0 0 0] = [3 14 12 12]

Therefore, the system is safe with the sequence [T0, T2, T3, T4, T1].

c.
Request(T1) = [0 4 2 0]

Request(T1) < Need(T1): [0 4 2 0] < [0 7 5 0] = true

Request(T1) < Available: [0 4 2 0] < [1 5 2 0] = true

We update the values:
Available = Available – Request(T1) = [1 5 2 0] – [0 4 2 0] = [1 1 0 0]
Allocation = allocation(T1) + Request(T1) = [1 0 0 0] + [0 4 2 0] = [1 4 2 0]
Need(T1) = Need(T1) – Request(T1) = [0 7 5 0] – [0 4 2 0] = [0 3 3 0]

The safe sequence remains valid with these values, therefore the request can be granted immediately.

B. Question 8.18 Graphs that illustrate a deadlock:

Graph (d): The cycle occurs in the following order: T1 and T2 are allocated to R1. T3 and T4 are waiting for R1, which has the instances T1 and T2. T1 is waiting for R2 which has the instances T3 and T4, and this results in a deadlock.

Graph(b): The cycle occurs in the following order: T3 waits for R1, however R1 is allocated to T1. T1 waits for R3, that is allocated to T3, thus creating a deadlock.

Graphs that display a deadlock:

Graph(c): Execution order: T2 → T3 → T1 OR T3 → T2 → T1

Graph(a): T2 → T3 → T1 OR T2 → T1 → T3