**COMPSCI 3SH3 Assignment 2**

1. **Threads (10 Marks)**
2. **Question: 4.10 (2 marks):**

Only heap memory and global variables are shared between threads of a multithreaded process.

1. **Question: 4.13 (1 mark):**

First, I will give a general description of both Concurrency and parallelism. Concurrency is when two or more threads are making independent progress in a multithreaded process. On the other hand, parallelism is when at least two threads are executing simultaneously. Although a single core processor can achieve an **appearance** of parallelism by switching back and forth between the single thread, in general it is **not** possible to have parallelism without concurrency, but it **is** possible to have concurrency without parallelism.

1. **Question 4.14 (3 marks):**

Amdah’ls law is: speedup =

Where p = fraction of code that can be parallelized, s = number of cores.

**For 40 percent parallel:**

1. **8 cores:** p = 0.4, s = 8

speedup =

= 1.53

1. **16 cores:** p = 0.4, s = 16

speedup =

= 1.6

**For 67 percent parallel:**

1. **2 cores:** p = 0.67, s = 2

speedup =

= 1.50

1. **4 cores:** p = 0.67, s = 4

speedup =

= 2.01

**For 90 percent parallel:**

1. **4 cores:** p = 0.9, s = 4

speedup =

= 3.076

1. **8 cores:** p = 0.9, s = 8

speedup =

= 4.70

1. **Question: 4.16 (4 marks):**

i) I would only create 1 thread for performing input and output operations. This is because the number of threads needed depends on the scheduling priority and needs of the application. For input and output, we only need 1 thread, as reading input from a single file and writing output to a single file cannot be sped up using multiple threads; any additional threads would merely be waiting for the I/O to complete.

ii)I will create 4 threads for the CPU-intensive portion of the application. This is because the CPU-intensive portion of an application should be utilising as many threads as there are processing cores. Fewer threads would result in a waste of resources, whereas more than 4 threads would cause inneficiency as it would be unable to run on our operating system architecture.

1. **Synchronization (10 Marks)**
2. **Question: 6.22 (6 marks)**
   1. Race conditions are when two or more threads access a shared variable at the same moment in time. Thus, the race condition in this code is the variable “number\_of\_processes,” which is accessed and modified by both *allocate\_process()* and *release\_process().*
   2. To prevent race conditions from occuring using a mutex lock, we make a call to *acquire()* upon entering either *allocate\_process()* or *release\_process(),* and make a call to *release()* immediately before exiting either function.
   3. This would not help. In *allocate\_process()*, race conditions occur because the variable *number\_of\_processes* is first checked in the conditional statement, and is incremented afterwards based on the conditional. Thus, there can be the case that this variable is 60 at the time of the test, but because of the race condition is set to 61 by another thread before it is incremented by the original thread.
3. **Question: 6.31 (4 marks)**

monitor alarm {

condition c;

int currentTick = 0;

void delay(int ticks) {

int wakeUpTime = currentTick + ticks;

while (currentTick < wakeUpTime) {

delay.wait(wakeUpTime);

}

delay.signal;

}

void tick() {

currentTick = currentTick + 1;

delay.signal;

}

}

1. **Deadlock (10 Marks)**
2. **Question: 8.3 (6 marks):**
   1. 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 |

* 1. First, we note the Available matrix is [1520]. We have to locate a safety sequence so that the following condition is satisfied: Need <= Available

Since Need(T0) <= Available, we select T0(available) = (available) + allocation(P0)

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

Need(T2) <= Aviilable -> Available = [1 5 3 2] + [1 3 5 4] = [2 8 8 6]

Need(T3) <= Aviilable -> Available = [2 8 8 6] + [0 6 3 2] = [2 14 11 8]

Need(T4) <= Aviilable -> Available = [2 14 11 8] + [0014] = [2 14 12 12]

Need(T1) <= Aviilable -> Available = [2 14 12 12] + [1 0 0 0] = [3 14 12 12]

Therefore, the system is in a safe state with the safe sequence beeing {T0, T2, T3, T4, T1}

* 1. Request(T1) = [0 4 2 0]

We first need to check if Request(T1) < Need(T1):

[0 4 2 0] < [0 7 5 0] = true

Next, check if Request(T1) < Available:

[0 4 2 0] < [1 5 2 0] = true

Now, we can 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]

With these updated values, if we verify, we can see that the safe sequence still remains valid, and thus the Request can be granted immediately.

1. **Question: 8.18 (4 Marks):**

These are the graphs that illustrate a deadlock scenario:

Graph d): Cycle is as follows: both T1 and T2 are allocated to R1. R1 has two instances that both T3 and T4 are waiting for. However, T1 is waiting for R2 which also has two instances that are allocated to T3 and T4. Thus, this creates a cycle resulting in a deadlock.

Graph b): Cycle is as follows: T3 is waiting for R1, but R1 is allocated to T1. T1 is waiting for R3, which is allocated to T3. Thus, this creates a cycle resulting in a deadlock

These are the graphs that do not illustrates a deadlock scenario:

Graph c): Order of execution: T2 -> T3 -> T1 OR T3 -> T2 -> T1

Graph a): Order of execution: T2 -> T3 -> T1 OR T2 -> T1 -> T3