Marcel ZAUDER 16-124-836
Pascal GERIG 16-104-721

8.1 Several Questions

a) *Why are servers (e.g., web servers) usually structured as thread-per-message gateways?*
Each request leads to the creation of a thread because with this the consumption of OS resources can be minimized which could become a problem for long-duration messages. The most applicable and useful method to implement this is via thread-pools which ensure that at one time only a certain amount of threads can be active.

b) *What are condition objects?*
Condition objects are used in order to encapsulate the waits and notifications for guarded methods. An example for a condition object would be the "lock" object in Java. It can be used to encapsulate specialized synchronization policies.

c) *Why does the SimpleConditionObject from the lecture not need any instance variables?*
There is no need for an instance variable because the access of the condition object is bound to a lock, such that its access is regulated via the *newCondition()* method.

d) *What are "permits" and "latches"?*
A *permit* is very similar to a semaphore in which it can be grabbed by the await method and each signal method call releases it again. An await call causes the invoking process to wait if all permits are already taken.
A *latch* on the other hand is for example a kind of future which is set at one point in time by a signal method call, i.e. it is set to true, and from this point it will always stay true.

8.2 Questions about Futures

a) *Which implementation would you prefer for this kind of problem? Is there any considerable difference at all? Justify your answer!*
Both of the implementations are suitable for this kind of problem. If the cleanup method would take much more time and is more extensive the *Early-Reply* approach would be more useful whereas when each thread would first ask the service for the future and after that can do other extensive things the use of *Futures* would be adviced.
For this example as it is there is no considerable difference observable because both need roughly the same time for the execution.

b) *Write a new class FutureTaskExecDemo.java that uses an ExecutorService implementation to compute the future task and to execute the clients, instead of creating explicit new threads. What is the benefit of using executors?*
The benefit of using an ExecutorService is that a total thread pool can be instantiated in which the number of threads that are available are set. Then each client that wants to get serviced can submit its task to the ExecutorService and as soon as a thread is available the submitted task is handled.

8.3 Nested Monitor

In order to avoid the deadlock, the *synchronized* keywords need to be removed from the *put()* and *get()* method in the *TheNest* buffer. Otherwise only one of each thread can either enter the *put()* or *get()* method, therefore the farmer blocking the access for the hen to put the egg into the nest, but because the farmer is waiting for the hen to notify the farmer that an egg was laid into the buffer, we have a deadlock.
In order to make the solution data race free we can synchronize the access of the nest-buffer itself by creating synchronized blocks around each modification of the buffer.

Marcel ZAUDER 16-124-836
Pascal GERIG 16-104-721

8.4 Thread Speed Evaluation

a) *What amount of processing cores does the CPU in your notebook have and what's the model / manufacturer of it?*
Intel Core i7-9700k, 8-Core Processor

b) *Does the implementation scale well, i.e., more concurrent threads help greatly to reduce the overall calculation time? Please provide concrete runtimes you experienced!*
For each entry there were 3 executions to get the average of the execution time.

Total Amount of Threads	Concurrent Threads	Time [ms]
10000	1	$(731 + 596 + 622)/3 \approx 650$
10000	2	$(654 + 623 + 681)/3 \approx 653$
10000	10	$(704 + 680 + 732)/3 \approx 705$
10000	50	$(686 + 622 + 737)/3 \approx 682$
10000	100	$(623 + 693 + 605)/3 \approx 640$
10000	500	$(672 + 741 + 652)/3 \approx 688$
10000	1000	$(620 + 682 + 751)/3 \approx 684$
10000	5000	$(645 + 693 + 632)/3 \approx 657$
10000	10000	$(670 + 572 + 682)/3 \approx 641$

The number of concurrent threads does not impact the execution time at all. This is because the computation each thread is making is only a simple divition (up to $\frac{1}{2*\text{"Number of Threads"}+1}$) which is done in a matter of milliseconds and therefore concurrent programming is not the most fitting approach for this particular formula.

c) *Depending on your results, why or why not does the solution scale well?*

Total Amount of Threads	Time [ms]	Result (until first error)
10	3	3.0...
50	8	3.12...
100	11	3.13...
500	37	3.13...
1000	74	3.140...
5000	319	3.1413...
10000	666	3.1414...
50000	2896	3.14157...
100000	5577	3.14158...
500000	25992	3.141590...
1000000	54899	3.141591...

It does not scale well, because when we are using 10-times more threads the accuracy is only improving slightly, i.e. with 100'000 thread we get an approximation of Π to fourth decimal point but with 1'000'000 we only get one additional correct decimal point and because of the huge time consumption if more threads are used it takes even more time.

d) *How would you improve the runtime with respect to faster calculations (without changing the algorithm)?*
Each thread can take a fixed amount of denominators and already compute the result of the sum of those. In the end those sums can be added together by the main thread.

e) *Which algorithm would you recommend as drop-in replacement for the Leibniz formula for faster calculation?*
The fastest way to compute Π would be with the Fast Fourier Transform approach discovered by Chudnovsky:

$$S = \sum_{n=0}^{\infty} (-1)^n \frac{(6n)!(k_2 + nk_1)}{(n!)^3 (3n)!(8k_4 k_5)^n}$$

$$\Pi = \frac{k_6 \sqrt{k_3}}{S}$$

Marcel ZAUDER 16-124-836
Pascal GERIG 16-104-721

where,

$$k_1 = 545140134, k_2 = 13591409, k_3 = 640320, k_4 = 100100025, k_5 = 327843840, k_6 = 53360$$

This is used by the current fast application PiFast to compute Π because each iteration leads to approximately 14 correct digits. For a concurrent programm each summand of the sum of S can be computed by one thread (or maybe also splitting this up in nominator and denominator calculations), and additionally the $\sqrt{k_3}$ and $k_6\sqrt{k_3}$ can be computed by individual threads in order to have a much faster, and much more accurate computation of Π.

f) *Why do the runtimes with identical parameters vary so much?*
That could be, because threads are waiting for the CPU to be processed. Especially when other programms are running on the same machine the computation time can increase because many threads want ton be processed.