Practical assignment 2

Synchronization: thread pool implementation

The purpose of the work

To consider the basic primitives of synchronization and their features, depending on the chosen programming language. Consider approaches to building software using parallelism and familiarize yourself with the classic problem of parallelism in the form of a thread pool.

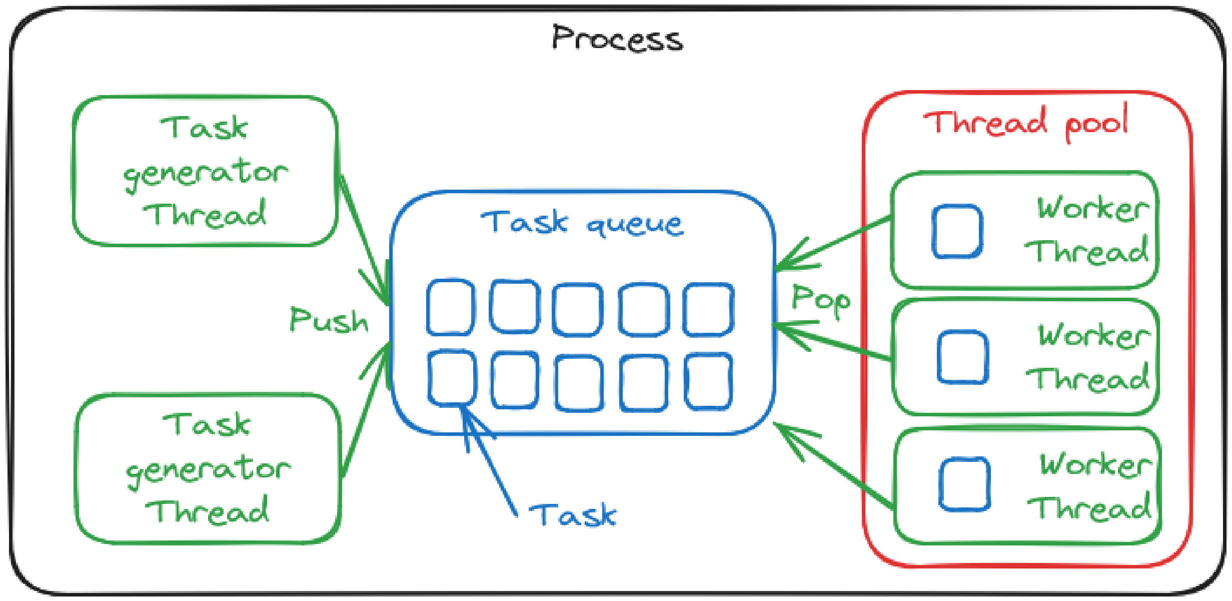

Image 1. A thread pool with 2 generator treads and 3 worker threads

Task

1. Familiarize yourself with the definition of the concept of a thread pool using this lecture materials or third-party sources.
2. Implement your own thread pool with the characteristics specified in the selected option. Among the **common characteristics** the thread pool must:
    a. be written correctly in relation to the selected programming language
    b. be able to complete its work correctly (immediately, with the abandonment of all active tasks, and with the completion of active tasks)
    c. have the possibility of temporarily stopping its work
    d. work with the use of conditional variables (monitors in some languages)
3. The operations of initializing and destroying the pool, adding and removing tasks from the queue must be thread-safe.
4. Create a program that will perform tasks according to the selected option, using the thread pool written by the student. Among the common characteristics: the code responsible for adding tasks to the thread pool, and the thread pool itself must be in different threads of execution.

5. Check and prove the correct operation of the program using the console input/output system. Perform time-limited testing and calculate the number of threads created and the average time a thread is in the waiting state. Determine the average length of each queue and the average task execution time (for unlimited queues). Or define the maximum and minimum time until the queue was filled, the number of rejected tasks (for limited queues).

Theory

To perform this task, it is necessary to consider the main primitives of synchronization, which are present in most programming languages, or can be created with the help of software tools. This list includes such primitives as:

1. **Mutex** is a synchronization primitive, the main task of which is to limit the simultaneous number of execution threads within the process that have access to a certain section of the code. A mutex restricts access to a section of code to only one thread of execution. The classic mutex interface consists of two methods (or functions, depending on the language): lock and unlock. The thread of execution that calls the lock method becomes the owner of the mutex, and until the owner of the mutex calls the unlock method (releases the mutex), calls to the lock method from other threads will cause them to wait for an unlock call from the owning thread. Calling the lock method twice in a row from the same execution thread will in most cases result in a runtime error.
2. **Conditional variable** is a synchronization primitive that blocks the execution of a thread (or several threads) until another thread fulfills a certain condition (condition) and announces it (waiting execution threads will notify about it).

In addition to these synchronization primitives, there are many others. Most of them are used to solve specific tasks or problems. The use of some of them is even considered a sign of "bad" architecture. The list of non-standard/additional primitives includes: spin lock (spin lock), recursive lock (recursive lock), read-write lock (read-write lock, in some sources it is found as shared-unique lock).

One of the main goals of this work is to get acquainted not only with synchronization primitives, but also to get acquainted with such a design pattern as a thread pool. The purpose of using a pattern is to encapsulate the logic of task execution, to simplify the use of parallelism as a tool in general. In reducing the cost of using parallelism as such, due to the leveling of overheads for creating threads of execution (since in a thread pool, each thread of execution is created only once and lives during the life of the entire pool). Stream pools are different in terms of implementation and interfaces. Pools are distinguished mainly by certain characteristics, listed below:

- By the number of task queues (basically this is one execution queue, but there may be more, but these are rare cases inherent in pools with a project-specific or task-specific architecture).
- By the algorithm of selecting a task for further execution (this list of algorithms is almost unlimited, but in most cases, it is the execution of the first task in the list)

- According to the approach to the execution of tasks in a queue (depends on the project architecture, it can be either the execution of all tasks at a certain point in time, at a certain point in the code, or simply execution of the task at once, if there is a free worker flow).
- By the possibility of obtaining the execution result (the question is about the clear possibility of obtaining the execution result using a certain task ID, if the pool clearly does not support this possibility, then this limitation can be bypassed for certain tasks by means of the selected programming language).

The basic scenario of using a thread pool looks like adding a task (a function object or an anonymous function) to the pool and waiting until it is executed asynchronously (without disturbing the work of the main thread) at a certain point in time.

Variants

1. A thread pool is served by 4 worker threads and has one execution queue. Tasks are added immediately to the end of the execution queue. There is a mechanism for obtaining the results of the task (as one of the options – when adding a task, its ID is obtained, and the result of the task and the status of the task can be obtained by this ID). The task is executed from the buffer as soon as there is a free worker thread. The task takes a random time of 5 to 10 seconds.
2. The thread pool is served by 6 worker threads and has one execution queue. The execution queue has a limited size of 20 tasks. Tasks are added immediately to the end of the execution queue, or discarded if they are not placed in the execution queue. The task is executed from the buffer as soon as there is a free worker thread. The task takes a random time of 5 to 10 seconds.
3. A thread pool is served by 4 worker threads and has one execution queue. Tasks are added immediately to the end of the execution queue. The queue of tasks is performed at intervals of 45 seconds (the buffer is filled with tasks for 45 seconds, which are then performed). Tasks added during the execution of the task queue are postponed to the next iteration (double buffering). The task takes a random time between 4 and 10 seconds.
4. A thread pool is served by 4 worker threads and has one execution queue. Tasks are added immediately to the end of the execution queue. The queue of tasks is performed at intervals of 45 seconds (the buffer is filled with tasks for 45 seconds, which are then performed). Tasks added during the execution of the task queue are discarded. The task takes a random time between 6 and 12 seconds.
5. A thread pool contains two queues, each of which is served by 2 worker threads. Tasks are added to the execution queue through one interface (the user does not have explicit access to the execution queues). Tasks are added immediately to the end of the random execution queue. The task takes a random time between 4 and 15 seconds.
6. A thread pool contains two queues, each of which is served by 2 worker threads. Tasks are added to the execution queue through one interface (the user does not have explicit access to the execution queues). Tasks are added immediately to the end of the least loaded in terms of time (a queue with tasks whose total execution time is shorter) of the execution queue. The task takes a random time between 2 and 15 seconds.
7. A thread pool contains two queues, each of which is served by 2 worker threads. Each execution queue has a limited size of 10 tasks. Tasks are added to the execution queue

through one interface (the user does not have explicit access to the execution queues). Tasks are added immediately to the end of the less filled execution queue, or are discarded if they do not fit into the execution queue. The task takes a random time between 4 and 10 seconds.

8. The thread pool is served by 6 worker threads. There is no queue of tasks as such. Tasks are immediately added to the free workflow. If all worker threads are busy, the task is rejected. The task takes a random time of 8 to 12 seconds. (Technically, there is no thread queue, but in this implementation, it is possible to implement 6 separate queues for one task, which, in fact, is the absence of a queue, because a queue is one active task).
9. The thread pool is served by 4 worker threads and has one prioritized execution queue. Tasks are added to the execution queue in the order of "difficulty" (shorter tasks have a higher priority, in this version it is possible that a certain task will never be completed). The task is executed from the buffer as soon as there is a free work thread. The task takes a random time of 5 to 10 seconds.
10. A thread pool contains two queues, the first is served by 3 threads, and the second is served by 1. Tasks are added in the first order of execution through one interface (the user does not have explicit access to the execution queues) in the order of "difficulty" (shorter tasks have a higher priority). If the task from the first queue is not completed within twice the time required for its completion, it is transferred to the second queue. The task is taken up from both queues immediately if there is a corresponding free work flow. The task takes a random time of 5 to 10 seconds.

Control Questions

1. What are the main synchronization primitives that you can list? Which ones are available in your chosen programming language?
2. What additional synchronization primitives can you list? Do you know anything about the additional primitives mentioned in the theoretical material for the work?
3. What is a mutex? Why is using (taking, locking) a mutex considered a hard operation? Specific OS (Windows, Unix) elements and features of synchronization.
4. What is deadlock? How does it arise? With which primitives to synchronize? The easiest way (rule of thumb) to avoid mutual blocking?
5. What is a conditional variable? Is this primitive present in your chosen programming language? Primitive interface, wait and notify methods.
6. Ways to stop and continue a thread in your chosen programming language? Ways of implementation in languages that do not support them?
7. What are the main technical and architectural problems solved by the task pool? Task variants use a metric in the form of task completion time, can this be a real metric in software, what analogue do you think can be used in real software?
8. What do you think is the optimal number of threads for a thread pool and under what conditions?
9. How do you think the situation should be handled when a task from a thread pool tries to add another task to the same thread pool?

Evaluation

Report	2
Synchronization correctness	8
Total	**10**

Absence of Git will lead to 0 points. https://classroom.github.com/a/6pM39_mJ
Incorrect answers on theoretical questions will lead to 0 points.