WHAT IS A DEADLOCK IN JAVA AND HOW TO AVOID IT

Fedoryshyn Dmytro
Student
Kharkiv National University of Radioelectronics, Ukraine

Scientific advisor: Kolesnikov Dmytro
Associate Professor of the Department of Software Engineering, Candidate of Technical Sciences
Kharkiv National University of Radioelectronics, Ukraine

During the development of multithreaded applications, there is often a dilemma: what is more important, reliability or program efficiency. For example, we use synchronization for thread safety, but incorrect synchronization can lead to deadlock [1]. We also use thread pools and semaphores to limit resource usage, but an error in such a design can result in deadlock due to resource shortage.

Deadlock is a situation where two or more processes, while holding certain resources, try to acquire other resources held by other processes, and none of the processes can obtain the resources they need or release the ones they hold.

Deadlock can occur due to several reasons [2]. One reason is mutual exclusion, where a resource can only be used by one thread at a time. Another reason is resource holding and waiting, where a thread holds one resource and waits for another resource held by another thread. Another reason is no preemption, where a resource cannot be taken away from a thread that already holds it but doesn't use it. The final reason is circular wait, where two or more threads hold resources and wait for another resource held by another thread.

To avoid deadlock in Java, it is necessary to avoid the conditions that can potentially lead to deadlock. For example, mutual exclusion can be prohibited for resources that can be safely used by multiple threads simultaneously. It is also important to establish a resource locking order in threads to prevent circular waits. When needing to use a new resource, release all previously held resources before acquiring the new one. Setting time limits for resource waiting and using algorithms for deadlock detection are also necessary.

In conclusion, deadlock is a rather dangerous situation that can occur during the development of multithreaded programs due to various complex reasons. Therefore, when using multithreading in Java, developers need to carefully analyze the code and try to prevent resource leaks and program failures.

References: