6.CGT Concurrency - Termination [CGT]

6.CGT.0 Terminology

Abort: The completion and shut down of a thread, where the thread is not permitted any execution after the command to abort has been received by the thread, or by the runtime services that control the thread. In particular, the thread will not be able to release any locks that it has explicitly acquired, and may not release any OS provided locks or data structures.

Termination: The completion and orderly shutdown of a thread, where the thread is permitted to make data objects consistent, return any heap-acquired storage, and notify any dependent threads that it is terminating.

Terminating Thread: The thread that is being halted from any further execution.

Termination Directing Thread: The thread (including the OS) that is causing the Terminated Thread to terminate.

Master of a Thread: Any thread which must wait for the terminating thread before it can take further execution steps (including termination of itself).

Child Thread: Any thread for which the terminating thread is a master. Note: Master-Child relationships may be static or dynamic, and may be a result of the master having created the child or have been given “master” status by other mechanisms.

6.CGT.1 Description of Application Vulnerability

There are a number of steps in the termination of a thread.

- The termination of execution of the thread, including termination of any synchronous communication;
- The finalisation of the local objects of the thread;
- Waiting for any threads that may depend on the thread to terminate;
- Finalisation of any state associated with dependent threads;
- Notification of outer scopes that finalisation is complete, including possible notification of the activating task;
- Removal and cleanup of thread control blocks and any state accessible by the thread by possibly threads in outer scopes.

Depending upon the multithreading model, some of these steps may be combined, may be explicitly programmed, or may be missing.

Thread termination vulnerabilities happen because of a failure in the termination protocol itself, or because of implicit dependencies threads that are outside of the termination dependency have on the terminating thread(s).

A thread may complete normal execution and terminate, may terminate due to a local error condition, or may be terminated by another thread or by the underlying runtime. A thread may also fail to terminate because it depends upon other threads that fail to complete their work and terminate. Early termination will result in the application not completing its task or result in the failure of the application. Late termination or failure to termination will cause non progress of the application, or will cause the application to deliver no results, or wrong results.

6.CGT.2 Cross References

Hoare C.A.R., "Communicating Sequential Processes", Prentice Hall, 1985
Holzmann G., "The SPIN Model Checker: Principles and Reference Manual"., Addison Wesley Professional. 2003

Larsen, Peterson, Wang, "Model Checking for Real-Time Systems"., Proceedings of the 10th International Conference on Fundamentals of Computation Theory, 1995

The Ravenscar Tasking Profile, specified in ISO/IEC 8652:1995 Ada with TC 1:2001 and AM 1:2007

6.CGT.3 Mechanism of Failure

If the terminated thread terminates prematurely, and there is no visibility to its runtime state from other threads sharing a

communication protocol or a termination protocol, then those threads will be unaware of the termination (unless they make a specific operation or request that makes them aware). Threads that depend upon direct actions from the terminating task (in the sense of waiting exclusively for a specific action before continuing) will wait forever.

If a dependent thread depends on the terminating thread, but the dependent thread ignores the termination notification, then a protocol failure will occur in the dependent thread. For asynchronous termination events, an unexpected event may cause immediate transfer of control from the execution place of dependent thread to anther (possible unknown), resulting in corrupted objects or resources; or may cause termination in the master thread, and an expected propagation of failures.

These conditions can result in

- premature shutdown of the system;
- corruption or arbitrary execution of code;
- livelock;
- deadlock;

depending upon how other threads handle the termination errors.

If a thread is aborted by another thread, there is nothing that can be done within the aborted thread to prepare data for return to master tasks, except possibly the management thread or OS notifies others that the event occurred. Any held locks may be left in a locked state resulting in waiting threads never being released. If the aborted thread was holding resources or performing active updates when aborted, then any direct access by other threads to such locks, resources or memory may result in corruption of those threads or of the complete system, up to and including arbitrary code execution.

Arbitrary execution of random code is distinct possibility from some kinds of termination errors, but arbitrary execution of known code is not likely since it is hard to determine where nonterminating threads will be in their execution when the terminating thread notification is delivered.

6.CGT.4 Applicable Language Characteristics

Languages that permit concurrency within the language, or support libraries and operating systems (such as POSIX-compliant OSs or Windows) that provide hooks for concurrency control.

6.CGT.5 Avoiding the Vulnerability or Mitigating its Effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- Use a language that provides a complete concurrency mechanism.
- Use mechanisms of the language or system to determine that necessary threads are still operating. Such mechanisms may be direct communication, runtime-level checks, explicit dependency relationships, or progress counters in shared communication code to verify progress
- Handle events and exceptions from termination events
- Program fall-back handlers to report or recover from premature termination failures.
- Provide manager threads to monitor progress and to collect and recover from improper terminations or abortions of threads.

6.CGT.6 Implications for Standardisation

In future standardisation activities, the following items should be considered:

- Provide a mechanism (either a language mechanism or a service call) to preclude the abort of a thread from another thread during critical pieces of code. Some languages (eg Ada) provide a notion of an abort-deferred region.
- Provide a mechanism (either a language mechanism or a service call) to signal another thread (or an entity that can be queried by other threads) when a thread terminates.
- Provide a structure within the concurrency service (either a language mechanism or a service call) that defers the delivery of asynchronous exceptions or asynchronous transfers of control