Language Vulnerabilities - Let's not forget Concurrency

A. Burns and A.J. Wellings
Real-Time Systems Group
Department of Computer Science
University of York, UK.

1 Introduction

Recent activities, within the auspices of ISO, have focused on the potential vulnerabilities of programming languages in use in high integrity applications. A draft technical report has been produced [5] that unfortunately contains little on the subject of concurrent language features. And yet concurrency is a significant issue in the design and implementation of many systems. Sequential programming languages ignore such issues and assume that the underlying operating system will deal with the management of threads/tasks and their inevitable interactions and synchronisations.

This paper [1] attempts to provide a comprehensive list of language vulnerabilities when concurrency and real-time features are supported. This list could form the basis for an assessment of Ada's tasking features. Such an assessment is likely to indicate that Ada has many advantages over other languages' provisions. Both the full language and the Ravenscar Profile should be addressed within such an assessment.

There are three main motivations for wanting to write concurrent programs and therefore to have the notion of concurrency in the employed programming language [2].

- To model parallelism in the real world - Real-time and embedded programs have to control and interface with real-world entities (robots, conveyor belts, etc.) that are inherently parallel. Reflecting the parallel nature of the system in the structures of the program makes for a more readable, maintainable and reliable application.
- To fully utilize the processor - Modern processors run at speeds far in excess of the input and output devices with which they must interact. A sequential program that is waiting for I/O is unable to perform any other operation.
- To allow more than one processor to solve a problem - A sequential program can only be executed by one processor (unless the compiler has transformed the program into a concurrent one). Modern hardware platforms consists of multiple processors to obtain more powerful execution environments. A concurrent program is able to exploit this true parallelism and obtain faster execution.

Concurrency does however bring with it a number of new vulnerabilities, and language features that support concurrent programming need to be assessed in the same way that sequential features are currently being scrutinized. Note that the decision not to use a concurrent programming language does not remove these vulnerabilities; many will be present in the operating system (OS) and the API used by the sequential program to access to the concurrency features of the OS. Indeed many of the vulnerabilities will be more extreme as they cannot easily be mitigated by the semantic restrictions in the language.

In this short position paper three topics are addressed: concurrency, communication and synchronisation, and scheduling. For each topic a series of issues are considered and possible vulnerabilities and mitigation are identified. Note the concurrent entity that is termed a task, a thread, a process or sometimes an event in the many different concurrent programming languages is called a *task* in the following descriptions.

[1] A version of this paper has been offered to a forthcoming workshop on Language Vulnerabilities

2 Concurrency

Many different concurrency models can be found in programming languages. There are distinguished by issues such as: static or dynamic task creation, hierarchical task structures, and the degree to which one task can influence/interfere with the behaviour of other tasks.

Static task creation

The simplest task structure available is one in which there is a fixed number of tasks that are created at the time of program instantiation. All tasks then exist for the duration of the program, which may be unbounded. Vulnerabilities from this simple model include the following.

1. Not all tasks start their execution (e.g. they may fail during activation).
2. Premature silent termination of a task or tasks.
3. Tasks executing with inappropriate initialisation parameters.
4. Overflow of task-local data (task attributes).

The first two vulnerabilities result in the program executing with only partial functionality. If tasks are relatively independent of each other then this situation may not be apparent to the tasks that are actually executing. A task may fail for a number of reasons including functional problems or execution issues such as stack overflow.

A common pattern for a real-time task is for it to be release periodically (with period `T`) and for its execution urgency to be influenced by its deadline (`D`) – both of these measures being of some appropriate time time. Such a task could be instantiated with inappropriate values for its parameters. The task may still function perfectly, but at the wrong rate (or it may be more likely to miss a deadline).

Where tasks use attributes stored in the TCB (task control block) during execution then an overflow of data may result in another task's TCB being corrupted. Similar issues apply to stack usage.

Language-level mitigation against these vulnerabilities is as follows.

- Signal when a created task fails to become runnable (or to become executable after an initialisation phase).
- Signal when a task terminates.
- Enable all initialisation parameters be be stored as constants in a single program module that can be independently verified.
- Prevent tasks form using attributes stored on the TCB, or ensure that the number of such attributes is statically bound and that each attribute is of a fixed static size.

The term 'signal' here is used in the general sense of a program entity that is either directly executed by the program run-time or is enabled by the run-time and executed under the control of the program. For a concurrent program with static task creation, the signal could be code external to the other tasks or enabled code executed under the control of one of the other tasks.

This signal abstraction will be used in a number of different situations in this treatment.

Dynamic and hierarchical task creation

If tasks can be created during the program's execution then many different program architectures can be constructed. Task termination becomes a normal event, and dependencies between tasks based on creation and termination are possible. Additional vulnerabilities from these features include the following.

5. Memory exhaustion due to dynamic object creation.

6. Memory exhaustion due to memory leakage.

7. Tasks indefinitely waiting for other tasks to terminate.

8. Tasks subject to errors propagating from child task creation.

All dynamically created objects, whether tasks or not, require memory and hence are subject to finite memory constraints. In very dynamic programs where many tasks are recreated and then terminate, it is important to ensure that terminated tasks can (and do) relinquish all memory allocated to them.

A common pattern in dynamic task programs is for one task to create another, and to subsequently wait for its termination. Creating a task may open up a vulnerability, and waiting for another task to terminate will clearly lead to indefinite postponement if that task does not in fact complete.

Language-level mitigation against these vulnerabilities is as follows.

- Allow a program to easily bound the number of active tasks it can contain; and to signal if this bound is violated.
- Allow the run-time to efficiently determine when a terminated task can have all its memory relinquished, allow these conditions to be easily delivered by the program, and require the run-time to actually reuse this memory (could involve the use of garbage collection).
- Provide a time-out feature on awaiting task termination.
- Minimise (or eliminate) dependencies between the created and the creator task, and signal when a task creation fails.

Inter-task influence/interference

Task can usually communication data and synchronise their executions via the language features that are described in section 3. But there are languages features that allow other forms of influence/interference. Some of these are via the scheduling facilities, and are covered in section 4. Here we cover abort, asynchronous exceptions, and asynchronous transfer of control (ATC). All of these are used to get the 'immediate' attention of the designated task. Polling for a state change is inappropriate. Vulnerabilities from these features include the following.

9. Rogue task aborting correctly behaving task (rather than visa versa).

10. Task (or program scope) terminated whilst holding locks/resources.

11. Task being in an inappropriate state to handle ATC or asynchronous exception.

The abort feature is one of the most controversial in that the motivation for its inclusion (to remove a rogue task) is mirrored by its main drawback (rogue task removing others).

All task terminations can cause problems if the task is not in the correct state for termination. But this is especially true when termination is imposed from outside. If the task is not terminating but its control is being influenced from outside then again there is the problem of this influence taking effect when the task is vulnerable, for example while updating a shared complex data structure.

Language-level mitigation against these vulnerabilities is as follows.

- Remove the abort feature.
- Provide a security manager.
- Signal when task terminates, and/or use object 'destructor' methods.
- Define program structures that postpone ATCs or asynchronous exceptions.

Many vulnerabilities can be eliminated by simple banning the associated language feature, but 'abort' is not universally supported and would be a simple feature to ban if the language uses a reserved word to represent this feature (ie. the non use of the reserved word is easily checked).

Knowing when a task has terminated is only the first part of reliable protecting against locks and resources being 'lost' when a task terminates. Recovery action still has to be programmed. To postpone the impact of asynchronous effects is necessary; there is a choice however to either allow code to 'opt in' or 'opt out' of these influences.

3 Communication and Synchronisation

Two general forms of communication are possible: synchronous and asynchronous. Synchronisation can be explicitly supported or programmed. Three combinations are considered in turn.

Asynchronous communication via shared variable

Vulnerabilities from these features include the following.

12. Unintentional use of unprotected shared variables.
13. Mutual update problem.
14. Race conditions.
15. Livelocks.

Shared variable are a well known error-prone language feature. As a result no language relies only on such variables. But some languages do allow their use and require the programmer to ensure that the intended behaviour is delivered.

Language-level mitigation against these vulnerabilities is as follows.

- Remove the shared variable feature.
- Provide adequate synchronisation primitives.

Again if there are appropriate alternatives, shared variables are not necessary (although certain forms of parallel architectures may require them for efficiency reasons). Race conditions and livelocks are difficult to mitigate against, but the number of situations in which they can occur can be reduced by sensible language choices.

Asynchronous communication with synchronisation support

A wide range of support features are available in different languages; for example, semaphores, signals, monitors and protected objects. Vulnerabilities from these features include the following.

16. Race conditions.
17. Deadlocks.
18. Indefinite postponements.
19. Protocol failures.

Once a task can be suspended then deadlocks and unbounded suspension become possible. Parts, or all, of the program can fail to make adequate progress. If a low-level primitive such as a signal or semaphore is used with shared variables to support a protocol such as readers/writers then errors can lead to rare race conditions and protocol failures.

Language-level mitigation against these vulnerabilities is as follows.

- Do not use low-level primitives.
- Use a protocol that is deadlock free - for example the priority ceiling protocol for single processor systems.

Again race conditions, deadlocks and indefinite postponements are difficult to mitigate against.

Synchronous communication (eg. rendezvous)

Simple CSP-like primitives and extended rendezvous are supported in different languages. Vulnerabilities from these features include the following.

20. Race conditions.
21. Deadlocks.
22. Indefinite postponements.

Although a similar list to before, these vulnerabilities are less severe with synchronous communications facilities. Indeed modeling and proof systems can be used to show the absence of these problems within programs, but only if the rendezvous is a simple one.

Language-level mitigation against these vulnerabilities is as follows.

- Do not use the extended versions.
- Restrict what can be done during a rendezvous.

Depending on what the other concurrency features are within the language, the list of restrictions can be quite extensive.

4 Scheduling and Real-Time Issues

In this section there are a number of issues to consider. First we will cover time and clock primitives, then scheduling and related topics.

Clocks and time

For real-time systems it is necessary to have access to a clock, measure time intervals and suspend a task for an interval of time. Vulnerabilities from these features include the following.

23. Drift between system clock and 'real-time'.
24. Drift between clocks on a distributed platform.
25. Inappropriate incorporation (or not) of leap seconds and time zone changes.
26. Mismatch between delay/sleep intent and clock granularity.

These are all well known timing/clock issues [2, 3].

Language-level mitigation against these vulnerabilities is as follows. They all involve the program having more visibility of the underlying implementation of the clock primitives.

- Enable the program to enquire as to the last epoch (synchronisation between system clock and external time base (UTP)) and current maximum drift.
- Enable the program to enquire as to last clock synchronisation and to the current maximum clock drift.
- Support two clocks, one monotonic and one 'wall time', and reduce the potential for use of the wrong clock.
- Enable the programmer to enquire as to the details of the delay/sleep implementation.

Asynchronous and synchronous task control

These language features allow one task to control the executable state of another task. Vulnerabilities from these features include the following.

27. Suspended tasks not being continued subsequently.
28. Tasks being suspended whilst holding locks/ resources.
29. Race conditions.

Similar arguments to those for banning the use of abort can be applied to this level of task control. Being indefinitely suspended is almost the same as being aborted.

Language-level mitigation against these vulnerabilities is as follows.

- Postpone the suspension of a task while it is holding locks or other resources.

Fixed priority scheduling

In this section the most common form of task dispatching is considered. If the language does not directly support such a policy then all scheduling must be under direct user control using the synchronous/asynchronous task control methods discussed above (and the dynamic priority scheme covered below). For fixed priority scheduling the vulnerabilities include the following.

30. Priority inversion.
31. Starvation.
32. Assumptions of scheduling analysis not been met by the program, for example execution times, blocking times, minimum times between sporadic tasks, intensity of interrupts, overheads of run-time, garbage collection overheads etc. requirements.
33. Excessive asynchronous traffic (interrupts/events) generated.

Priority inversion occurs through the use of a synchronisation primitive that does not take priority into account. Starvation occurs when there is not enough processing time available for the low priority tasks to make adequate progress.

Language-level mitigation against these vulnerabilities is as follows.

- Incorporate priority inheritance protocols where appropriate – ideally one that prevents deadlocks.
- Monitor and enforce CPU time usage by tasks.
- Monitor and blocking and interference times.
- Signal when a deadline is missed.
- Signal when interrupts/events occur too often, and be able to disable them for periods of time.

Program control over scheduling parameters or policy

If a scheduling scheme such as fixed priority scheduling is supported by the language then it is usual to allow the program to exercise control over some of the scheduling parameters, such as the assignment (static or dynamic) of priorities to tasks and the periods and deadlines of the tasks. Vulnerabilities from these features include the following.

34. Loss of liveness (some tasks fail to make progress).

35. Loss of timeliness (some task failing to meet a deadline).

Language-level mitigation against these vulnerabilities is as follows.

- Provide high level abstraction for common task 'types' such as periodic or sporadic tasks (ie. prevent program from changing the scheduling parameters).
- Do not allow dynamic priority changes.
- Enable all scheduling parameters be be stored as constants in a single program module that can be independently verified.
- Signal when a task misses a deadline.

5 Conclusions

This short paper has attempted to highlight the many different vulnerabilities that exist with concurrent programming languages. There are a large number (and variety) of language features that support various aspects of concurrent programming. Not all can be used safety and there are many vulnerabilities that the above review has highlighted. For many of these vulnerabilities it is possible to define language-level mitigation. Some are simply to not allow the use of (non-essential) features. Others point to safe usage patterns. As noted in the introduction, the decision not to use a concurrent programming language does not remove these vulnerabilities; many will be present in the operating system (OS) and the API used by the sequential program to access to the concurrency features of the OS.

It is possible to take an extensive set of language features, such as those provided by Ada tasking, and define a subset (and other restrictions) so that a profile is defined that has adequate expressive power and a minimum of vulnerabilities. One candidate for this would be the Ravenscar profile for Ada [1, 4]. Similar profiles need to be defined for other languages. We note that Java has started to undertake this process under the auspices of the Java Community Process (JSR 302).

References

[1] A. Burns, B.Dobbing, and T. Vardanega. Guide for the use of the Ada Ravenscar Profile in high integrity systems. Technical Report YCS-2003-348, University of York, Department of Computer Science, 2003.

[2] A. Burns and A. J. Wellings. *Real-Time Systems and Programming Languages*. Addison Wesley Longman, 4th edition, 2009.

[3] G.F. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems, Concepts and Design*. Addison Wesley, 4th edition, 2005.

[4] ISO/IEC. Information technology - programming languages - guide for the use of the Ada Ravenscar Profile in high integrity systems. Technical Report TR 24718, ISO/IEC, 2005.

[5] ISO/IEC. Information technology - programming languages - guidelines to avoiding vulnerabilities in language selection and use. Technical Report PDTR 24772 – draft, ISO/IEC, 2009.