Fault Tolerance and Avoidance in Biomedical Systems

Shane Stephens & Gernot Heiser
School of Computer Science and Engineering
University of New South Wales

Abstract

It is important for a variety of reasons that biomedical systems execute without errors. One useful approach towards error-free software is to design a range of fault tolerant properties into applications software. In addition, by restricting the behaviour of an application and requiring explicit allocation of resources such as memory, errors can be caught while an application is still being written, rather than once an application has been released. This paper investigates how an operating system can support biomedical applications using these approaches.

1 Introduction

A biomedical system is one which interfaces with humans in a medical context. Examples include life-support devices such as pace makers, diagnostic and monitoring devices such as electrocardiographs, and prosthetic limbs and organs.

Fault tolerance in this context refers to recovery from a fault in such a way that the faulty service can recover, and continues to be offered to the user. Fault avoidance refers to increasing the stringency of a system in such a way that bugs are easier to find.

While fault tolerance and avoidance is important in all systems, it is especially important to ensure that biomedical devices work as intended. In some cases, the patient is unable to survive without the assistance of the device, while in others, the patient relies upon the correctness of information the device provides. Therefore the methodologies used in conventional software are inappropriate for Biomedical software.

One approach to fault tolerance is formal verification of code [3]. The author is currently involved in an attempt to verify the L4 microkernel [4], and we are confident that our approach will succeed. Another feature of L4 that sets it apart from traditional embedded kernels is that it implements memory protection. Memory protection is useful in this context as it limits the damage that can occur due to malfunctioning code.

However, executing on a verified kernel is not sufficient protection for biomedical systems - even in the presence of a perfect kernel, user applications can fail. One solution to this problem is to require that the user applications themselves be verified in a similar manner to the kernel. Given the magnitude of effort required to verify code, however, this may not be a practical approach: there are many more user applications than kernels.

Fault tolerance and avoidance should therefore be examined as an alternative in cases where full verification of user code is not a feasible option. It is a central thesis of this paper that an appropriately designed operating system can support and aid programmers of user applications who wish to write fault tolerant code.

A prototype version of such an operating system, Biomedical Operating System (BiOS), has been written, and further research is currently in progress. Some key aspects of this operating system are presented below.

2 The BiOS Design

BiOS provides a small set of user-level services, implemented on top of L4. These services include a pager, a system-call server, and a packet server. Given the size and modularity of these services, verification should be possible.

The domain of embedded biomedical applications has several important properties that have influenced the design of BiOS. These properties are:

- that embedded biomedical applications typically require only a small number of concurrent threads of execution. A general purpose system that allows arbitrary execution of multiple applications is not required.
- that biomedical devices will typically not be required to run code which was not produced by the developer of the device. In general, faults will occur because of programming bugs, not malicious code.

- that typical biomedical applications involve the continuous or semi-continuous processing of packetised data. This is exploited by the provision of a highly optimised streams abstraction which lies at the base of many of the fault-tolerant properties of BiOS.
- that because of limitations in human perception and the relatively slow rate of events within the human body, most biomedical applications have a data acquisition/production frequency in the order of only tens of Hertz. In addition, the human body itself adapts gracefully to delayed deadlines on the milliseconds scale - jerky video streams are still watchable, and a delay between action and effect can often be adapted to. Hence hard real-time guarantees are not required in general.

Given these properties, a decision was made to base BiOS inter-process communication around a streams abstraction. Although this abstraction is quite different to existing UNIX abstractions, BiOS is not intended to be a general-purpose operating system. In addition, provision of this abstraction allows biomedical developers to think about streams-based problems in a more natural manner. Finally, soft real-time schedulers for streams exist (see for instance Löser et. al. [5]), and adaptation of an existing scheduler to the BiOS system should be possible if soft real-time is required.

BiOS streams connect several participating threads together in an ordered fashion. When a stream is created, it is initialised with a fixed number of packets that can be passed along the stream. At any given time, each packet may only belong to one thread (or "stream element"). Adjacent stream elements communicate by transferring ownership of a packet from one element to another.

This promotes a user view of an application as a set of communicating, modular stream elements. Ideally, each element performs a single logical action, and each logical action is distinct in its execution from the rest of the system.

BiOS enforces this abstraction by providing only a streams interface to the system drivers. This also increases the efficiency of the system - BiOS streams are designed to provide a zero-copy communications mechanism.

3 Modularity

Providing applications developers with a system interface that promotes modularity has several advantages. Firstly, the task of writing an application is simplified, as the design approach essentially consists of identifying candidate stream elements, designing an interface between adjacent elements, and writing each element.

Secondly, modular applications are easier to debug, as accidental memory accesses are more likely to cause a protection fault than a side-effect.

Finally, modular applications are easier to verify [2] (if verification is considered absolutely necessary), as the application is already split into a set of orthogonal sections that communicate via a well-defined interface.

4 Protection vs Performance

To provide an efficient zero-copy mechanism for streams, BiOS must place all packets in globally shared memory. However, this weakens interprocess protection, because stream elements can write to packets that they do not own.

To solve this problem without sacrificing performance, BiOS provides two completely separate implementations of the streams interface. The first implementation enforces protection by manipulation of virtual memory using an L4 memory primitive known as "grant". Grant operates on one or more contiguous pages of memory, and can be thought of as a transfer of the underlying frames from one address space to another.

The safe streams interface provides an operating system service known as the "packet server". When a new stream is created, that stream's packets are initialised within the packet server's address space, one per page, and are only granted to participating threads as required. Similarly, when a thread decides to send a packet, this packet is granted back to the packet server. In this manner, illegal accesses within the region of memory containing the packets are detected by the system, and the developer is notified.

However, due to the relatively high cost of page granting, this implementation is quite slow. Given the nature of many biomedical applications, this limitation may not be significant. However, if more efficient communication is required, BiOS provides a second implementation of the streams interface. This implementation provides a permanently mapped region of memory for the stream. Packets reside in this region, and illegal accesses are not caught.

This interface is fast for three reasons. Firstly, expensive virtual memory operations are not required. Secondly, because the user applications are given a pointer to a buffer rather than supplying one, the stream can be used to implement zero-copy transfer of data all the way along the stream (including to and from operating system drivers). Finally, the operating system does not play a heavy role in the streams mechanism (being involved only in blocking stream elements that are waiting on packets which have not been sent), which reduces execution time substantially.

Because the two implementations provide exactly the same interface, switching from the safe implementation to the fast implementation simply requires toggling an initialisation flag. As a result, user code which executes safely on the slow interface can still be considered safe when running on the fast interface.

It is evident that carefully written malicious code could seem to execute correctly on the protected implementation, yet perform illegal accesses on the high-performance implementation. However, the purpose of the BiOS dual streams implementation is not to protect against malicious code, but instead to detect accidentally programmed bugs. This restriction explicitly excludes consideration of Byzantine failures.

5 Fault Recovery

The programmer's view of BiOS applications is that of a cooperating system of stream elements. This view allows the programmer to implement several fault recovery mechanisms at user level with a minimum of difficulty.

BiOS can be configured to restart a task when an exception is raised by that task. Rather than using the 'main' entry point, BiOS will start the task at an additional, user-defined entry point (much like a light-weight version of UNIX signals). All stream memory and mappings in the task are preserved, and the user code must then determine what error occurred and handle the error appropriately.

A simple mechanism for dealing with an error may be simply to discard the most recent packet of data and request the next one. Alternatively, the user module may simply be restarted with all of its state re-initialised. More complicated mechanisms may simply attempt to process the faulty packet with an alternative algorithm, or execute an internal consistency check before continuing.

The user can also insert stream elements which perform explicit bounds-checking at various points of the stream. These elements can be registered with BiOS as additional exception-generators, and can be programmed to trigger if packets are detected with erroneous or nonsensical data.

Such stream elements could look for signs of malfunction such as packets that contain unexpected values (for instance, negative values in a frequency field); or an unreasonable time without a new packet becoming available. Other user-defined signs could also be implemented if required.

This approach provides mechanisms by which users can write fault-tolerant code, rather than dictating operating-system level fault-tolerant procedures to the programmer.

6 N-Version programming

N-Version programming is a popular existing technique for writing fault-tolerant software where proof of an algorithm is impractical. Essentially, the approach consists of processing data with several implementations of the same algorithm, and attempting to find a consensus of the results [1].

This approach can readily be implemented with little overhead using BiOS streams. Several alternate implementations of the required algorithm can be implemented as separate threads or processes, and registered in a stream. A demultiplexing stream element can then make several copies of a packet and pass a copy to each implementation. Finally, a consensus element could collect each implementation's result, and use any of the existing approaches to choose an acceptable outcome based upon the results gathered.

7 Conclusion

Provision of a reliable system is the responsibility of both the operating system provider and the application writer. This paper has examined some operating system features that can aid the application writer in construction of a fault-tolerant application.

References

[1] A. Avizienis. The methodology of n-version programming. In *Software Fault Tolerance*, pages 23–46, 1995.

[2] K. Havelund and J. Skakkebaek. Practical application of model checking in software verification. In *Proceedings of the 7th Workshop on the SPIN Verification System*, Sept. 1999.

[3] C. A. R. Hoare. An axiomatic approach to computer programming. *Commun. ACM*, 12:576–580, 1969.

[4] M. Hohmuth, H. Tews, and S. G. Stephens. Applying source-code verification to a microkernel — the VFiasco project. Technical Report TUD–FI02–02–März 2002, Dresden University of Technology, 2002. Available from URL: `http://os.inf.tu-dresden.de/vfiasco/`.

[5] J. Löser, H. Härtig, and L. Reuther. A streaming interface for real-time interprocess communication. Technical Report TUD–FI01–09–August 2001, Dresden University of Technology, 2001.