A Pattern For Device Drivers

Peter Seymour
peteralanseymour@hotmail.com

February 9, 2020

1 Wait-free Message Passing To Devices

When registers of the Breathe system are synchronised the single path of execution of the runtime moves into an external module. This creates a problem should the external module fail or take a long time to perform its task. If the runtime is simulating threads then all threads will be blocked which would impose unacceptable overheads and fragility especially as the number of external modules increases. The system specification implies that all synchronisations must complete. Some modules perform very simple tasks which in their nature take only a small bounded amount of time and pose no problems. However, for arbitrary devices such as writing to a concurrently running window or accessing a network this is not the case. Below I outline a simple pattern that all modules can follow to ensure that in the worst case a known and configurable amount of time is spent synchronising.

2 The Pattern

The device runs concurrently to the runtime and is started when the module (driver) is loaded. Should the device fail or block the runtime is unaffected. A message queue is established to asynchronously send messages to the device. Message delivery is not guaranteed but further higher level protocols can be established with messages coming back from the device on another queue. This is in keeping with the specification that views registers as monitored by the outside world only. Ideally this model would be sufficient since it takes only a bounded amount of time to post the message. However, that is only true if the queue is allowed to grow indefinitely which is not a desirable property. Should the device be particularly slow some mechanism to alert the producer is required to throttle its behavour and redirect it to more useful tasks,

Four registers are created `spare`, `timeout`, `increase` and `signal`. The first three are non-negative integers and the last a boolean value.

Initially the queue has a non-zero size and timeout configured. The register `spare` is read only and contains the number of spare slots in the message queue. At any time after synchronising to this register but before posting a message

more slots may become empty but never less. The register `timeout` contains the maximum timeout period which may be updated. The register `increase` is used to increase the maximum queue size. Notice that once more slots are allocated they cannot be reclaimed. Finally the `signal` register contains a read/write status flag indicating whether a signal will be generated if a message is dropped. Initially this is set to false.

The queue is implemented as in [1] which leads to the following behaviour:

Entry	Max Wait	Exit
Spare > 0	O(1)	Success
Spare $= 0$	O(timeout)	Success
		Failure

Here success and failure refer only to the act of posting a message and say nothing of its receipt. There are three key use cases for this system. Firstly a simple application can blindly send messages to the device. In the worst case the queue fills up and messages are dropped. The cost of the dropped messages is the runtime blocking for a period equal to `timeout` per message. In the second case the application can read `spare` and determine if posting a message will take almost no time or possibly fail. If failure is possible then the application can avoid posting the message and in turn avoid the timeout penalty. Ideally the application will be smart and either increase the queue size, throttle itself yielding to other processes or do other work itself. The third case is similar except that instead of reading `spare` the signal is activated and messages are posted. Should the queue fill then a message will be dropped but the application will be alerted via the signal and can take action. Either resending or adapting the queue making it larger for future use.

The control flow for the second case is shown below:

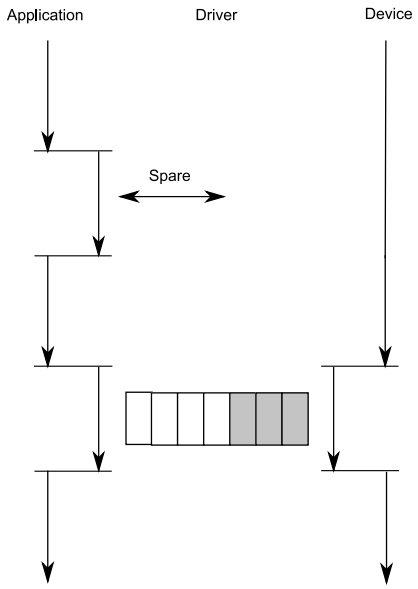

Figure 1: Control flow passing a message to a device

3 Conclusion

The pattern presented can be widely used and gives wait-free behaviour but allows smart applications to monitor device communication and take action. The default configuration allows drivers to be created for simple use so long as dropped messages are tolerable. In other cases applications can take pre-emptive or remedial action to acheive efficient long-term stability between itself and external devices.

References

[1] "An Efficient Wait-free Queue Implementation", P. Seymour, 2007.