Exceptions, Interrupts and Processes

David May: November 17, 2019

Background

We want to make instruction emulation and system calls very efficient.

We want to make interrupts very efficient.

We want to make it impossible for an untrusted process - either miscompiled or malevolant - to cause system failure.

We are probably expecting to implement and supply a preferred version of concurrency, communication and input-output along with our implementations of C#, Python - but still be able to implement others efficiently.

Contexts

Almost all implementations of processes (and interrupts) require context switches in which most of the registers are saved and restored. We don't have a lot of registers and it seems likely that they could all be saved or restored in a few cycles - maybe even just one cycle. Our architecture should make it possible to provide implementations that do this, whilst also allowing simple, less optimised implementations.

Our context can be represented in an object which holds 15 words. This requires 16 words in total, so there is the possibility of aligning these context objects in memory so as to enable single cycle access. It would also be practical to provide a separate memory for the contexts.

If we provide an architectural *run list* of processes that are ready to execute, it is possible for these to be pre-loaded from memory into one or two context buffers ready for execution; similarly, it is possible for them to be stored in buffers whilst they are being written back to memory. This allows the process context switching to proceed in parallel with process execution.

So a context could look like this:

General purpose registers	r0 ... r9
Program counter	pc
Workspace pointer	wp
Environment pointer	ep
Link register	lr
Run link	rl

This context structure can also be used for the exception and interrupt contexts.

It is important that exceptions and interrupts can schedule normal processes (*deferred handlers*) to handle non-urgent tasks. This includes emulation of complex instructions. The excepting context and excepting instruction are stored in registers r0 and r1 of the exception-handling context; these are used to access the operands and can be passed on to registers r0 and r1 of the deferred handler.

Operation

We have three execution states, each with its own register set holding a context. These are:

NORMAL	used for a collection of concurrent processes
EXCEPTION	used when a process gives rise to an exception or *system call*
INTERRUPT	used for a call from a hardware device or from a process

There is also an IDLE state.

There are flags to record the state - one set for NORMAL, one set for EXCEPTION, one set for INTERRUPT. The NORMAL register set is used if neither the exception flag nor the interrupt flag is set. The EXCEPTION register set is used if the exception flag is set and the interrupt flag is not set. The INTERRUPT register set is used if the interrupt flag is set.

Exceptions - calls, unimplemented instructions and exceptions will set the exception flag. The excepting context and excepting instruction are stored in registers r0 and r1 of the EXCEPTION context and can be used to access the opcode and operands of the excepting instruction using GETOP and PUTOP instructions. These instructions will access the registers in the excepting context.

Interrupts - hardware and software will set the interrupt flag. They can not occur in INTERRUPT state. The reason for the interrupt is supplied in register r0 and can be used to determine what action to take. For a software interrupt, the reason is supplied as an instruction operand; for a hardware interrupt, it is supplied by the hardware device from a register that can be set by software when the device is configured/enabled.

There is an EXIT instruction that clears the flag associated with the register set in use.

NORMAL mode is used to execute a collection of processes. There are instructions to move to the next process (NEXT) to pause and re-schedule the current process (PAUSE) and to schedule a process (RUN). This enables the processor to optimise process scheduling by *context flow*, pipelining the process contexts through the normal register set. These instructions can be executed in EXCEPTION and INTERRUPT mode. The NEXT instruction only takes effect when the processor returns to NORMAL mode.

There are instructions to create a new process; these get an object for the process context and supply initial values for the ep and the pc. GETP creates an (untrusted) process; GETTP creates a trusted process. DEFER creates a trusted process, copying the r0 and r1 register values of the EXCEPTION context and enabling the deferred handler to access the operands of the excepting instruction from memory.

Only trusted processes can execute NEXT, PAUSE, RUN, DEFER, GETOP and PUTOP instructions.

Garbage Collector

There are objects in memory for the exception and interrupt contexts, along with a collection of objects for normal processes.

At the point that the garbage collector starts, there will be an interrupt context and an exception context in the associated registers; there may also be a normal context in the normal registers. The first action is to save these to their associated objects in memory. As soon as this action starts, and for the rest of the marking phase, when any pointer is loaded, its associated object will be marked.

The garbage collector operates by marking the contexts in memory, starting with the interrupt and exception contexts and continuing with the contexts on the run list. It maintains a list of the objects waiting to be marked (which may include context objects). It uses this list only when it has reached the end of the run list. This means that it prioritises marking of contexts waiting to run.

If the context switcher encounters a context that has not been marked when it is writing back a context (for example, as a result of executing a NEXT instruction), it waits until the context has been marked before continuing. This ensures that the original pointers in the object (which may have been stored and overwritten during execution) will be marked, along with any new pointers that have been loaded.

A consequence of this is the that garbage collector can be implemented entirely as part of the memory.