In the original envisioning of Machine, there is no way for a kernel to be written such that it can guarantee its own safety from other running programs, or the safety of programs from other user-space programs. This document details a plan to add features that will make this possible.

Principles

In designing this system, we want to strive for:

Power

It must be possible to guarantee that user-space programs can be contained (defined precisely below). Additionally, it must be possible to, ignoring physical limits of the size of memory and the size of words, run an arbitrarily large number of programs simultaneously.

Simplicity

We want to add the smallest number of instructions and features as possible.

Abstraction

Programs should be able to be written as if these features had never been added. Obviously this is not completely possible, but the system should stray as little as possible from this ideal.

Definitions

Containment

The kernel should be able to guarantee that a user-space program cannot:

- Run for more than *n* processor cycles, where *n* is not necessarily defined by the kernel (that is, it may be acceptable for the system to impose an arbitrary lower or upper bound on how strict the kernel can be. The degenerate case would be that there is some fixed quantum of scheduling which the kernel can decide to either enforce or not, but whose value the kernel cannot control)
- Access the memory of programs other than itself (including the kernel). The way in which this is achieved may be arbitrarily complex or inefficient, and may impose other requirements in order to be satisfied (for example, the system might be only able to enforce this if memory is allocated to programs in 1M word-chunks), but it must be theoretically possible.
- Read from or write to the input or output devices.
- Halt the machine, including by entering a failure mode

Solution

Features

Modes

The system can be in two modes - *protected* and *user*. At the beginning of execution, the system is in protected mode.

Lookaside Registers

There are 16 extra registers called *lookaside* registers, denoted r' (ie, r'[0], r'[A], etc). When any illegal operation is performed in user mode, all lookaside registers are set such that r'[x] := r[x].

Callback Register

There is a single register called the *callback register*, denoted c. The contents of c are denoted c[]. When any illegal operation is performed in user mode, after the lookaside registers have been set, the system is placed into protected mode, and execution begins from c[].

Fault Register

There is a single register called the *fault register*, denoted f. The contents of f are denoted f[]. When any illegal operation is performed in user mode, the *fault code* (defined below) is placed into f prior to execution resuming from c[] (the order relative to the lookaside registers being set is undefined).

Program Counter Lookaside Register

There is a single register called the *program counter lookaside register*, denoted pc'. The contents of pc' are denoted pc'[]. When any illegal operation is performed in user mode, the value of the program counter prior to the execution of the illegal instruction is placed into pc' prior to execution resuming from c[] (the order relative to the lookaside registers being set and to the fault register being set is undefined).

Virtual Memory Registers

There is a pair of registers called *virtual memory registers*, denoted v (ie, v[0] and v[1]). When a memory access operation is performed in user mode, the physical address, p, that is accessed is defined in relation to the logical address, l, that is coded by the instruction word, by the relation $p = v[0] + l$. If p < v[0] or p > v[1], this operation is an illegal operation.

Program Counter Timer

There is a single register called the *program counter timer*, denoted t. The contents of t are denoted t[]. Prior to the execution of any instruction in user mode, if t[] is not the maximum possible value (that is, t[] + 1 != 0), t is decremented (t[] = t[] - 1). If, prior to this, t is 0, then instead of t being decremented, it is considered an illegal operation, and a fault is triggered. Additionally, if executing the instruction causes any other fault, t is not decremented.

Instructions

These instructions are known as *protected instructions*. Executing any of these instructions in user mode is an illegal operation.

- *enter user mode* - place the system in user mode, set the program counter to r[A], set all registers r[x] := r'[x], and continue execution.
- *lookaside load* - set r[A] := r'[B]
- *lookaside store* - set r'[A] := r[B]
- *set callback* - set c[] := r[A]
- *fault move*- set r[A] := f[]
- *pc lookaside load* - set r[A] := pc'[]
- *set virtual memory low* - set v[0] := r[A]
- *set virtual memory high* - set v[1] := r[A]
- *pc timer load* - set r[A] := t[]
- *pc timer store* - set t[] := r[A]
- *trigger* - no op

Additionally, the following already-defined instructions are now illegal operations if executed in user mode.

- *halt*
- *output*
- *input*

These newly-defined instructions also have their own associated failure modes. As with normal instructions, entering a failure mode will cause the machine to halt.

- Executing *enter user mode* when v[0] + r[A] does not address a word in the range [v[0],v[1]][1]
- Executing *set virtual memory low* or *set virtual memory high* when r[A] does not address an allocated word.[2]

Fault Codes

This is a list of illegal operations which have distinct fault codes. The codes themselves are not defined in this document (and would likely be arbitrary).

- Executing a protected instruction, including *halt*, *output*, and *input*
- Executing the *trigger* instruction
- Executing an instruction when the program counter, t[], is equal to 0
- Executing an instruction outside of the set virtual memory space
- Accessing memory outside of the set virtual memory space
- Attempting to execute an instruction which does not code for a valid operation
- Division by zero

---

[1] This prevents the confusing scenario of a kernel mistake being interpreted as a user-mode mistake, which the kernel would then catch.

[2] This ensures that all illegal memory accesses errors by user-mode programs are virtual memory errors. Without this guarantee, there would be two classes of illegal memory access errors - virtual memory, and physical memory.

Intended Use

The features defined above are designed to be used as follows. The validity of this solution should be evaluated in terms of this intended use, and the extent to which it is technically possible to stray from the intended use (especially to exploit flaws in the design).

As containment is defined at the beginning of this document, it consists of four separate requirements. A kernel must be able to prevent a program from, if it wishes:

- Running for an unbounded number of cycles
- Accessing memory owned by other programs (including the kernel)
- Reading from or writing to input or output devices
- Halting the machine, including by entering a failure mode

The latter two requirements are satisfied simply. The *input, output, and halt* commands trigger faults, and every possible failure mode triggers a fault as well. The former two requirements are where most of the interesting bits are.

In order to prevent user-space programs from running for an unbounded number of cycles, the kernel should set the program counter timer to a reasonable value before executing user-space programs. The fact that setting this counter is a protected instruction prevents programs from extending their own timers and buying themselves more cycles. The fact that setting the callback is a protected instruction prevents programs from continuing to run after the timer has expired. Virtual memory protection should prevent programs from tricking the kernel into running them again after the timer has expired. Note that the kernel may allow unbounded execution by setting the program counter timer to the maximum possible value.

When the timer expires, the registers are all copied into the lookaside registers. The kernel should store the values in the lookaside registers so that they can be restored prior to continuing execution of the program. Storing these registers can be accomplished with *lookaside load*. Restoring them can be accomplished with *lookaside store*. Since *enter user mode* copies all of the lookaside registers into the normal registers, restoring the lookaside registers should be sufficient to guarantee that the state of the world from the point of view of the program hasn't changed since it was last running.

Memory safety is guaranteed by virtual memory. Any accesses of memory (load, store, jump, etc) outside of the allowed region are faults. Since there is a single region, user-space programs' memory should be contiguous (note that it's possible to implement non-contiguous memory through careful handling of faults). Programs may request more memory. Logically, this memory will be appended to their currently-allocated memory. If contiguous physical memory is available, the end of the memory space could simply be grown. However, if physical memory is not available contiguously, another, larger space must be located, and all contents must be copied over. This includes moving the beginning of the memory space backwards, since simply changing the logical beginning without copying would invalidate memory references that the program was holding (that is, virtual addresses would reference different physical addresses than they had previously).

It's possible that programs may not wish to interact with the kernel. In fact, under this model, it should be possible for a program written for the old system (ie, written assuming it was the only program running) to run just as well, so long as the kernel handles faults properly to maintain this illusion. It should even be possible for a program written for the *new* system to run just as well (ie, virtualization), since all of the new instructions that have been added are protected instructions, and will all trigger faults.

However, it's also possible that the program may wish to interact with the kernel. In that case, the *trigger* instruction should be useful. It's a no-op whose sole purpose is to trigger a fault. It's not strictly necessary - kernels could write their interfaces such that particular faults were intentionally caused to signal communication - but it makes this communication much simpler and cleaner. Since the kernel is able to inspect the full state of the running process after a fault, a program wishing to communicate with the kernel should be able to set certain agreed-upon state, execute the *trigger* instruction, and allow the kernel to inspect that state in order to figure out what the requested communication is.

Lastly, since all of the normal failure conditions trigger faults for user-space programs, any error on the part of the program should be able to be reported in detail to the user. Since the program counter is also stored, it should be possible to deduce as much detail as the kernel wishes. This includes details which are not included in the fault code such as which address caused a virtual memory fault, what number was divided by zero, etc.