Principles for Reliable Operating Systems

Peter J. Denning
Naval Postgraduate School

1/28/03

Back in the summer of 1975, Dorothy Denning and I, then still newlyweds, spent a month at Cambridge. During that time Roger Needham and I met daily to discuss topics in the design of operating systems. We were searching for fundamental principles for reliable systems. One fruit of those discussions was my paper, "Fault tolerant operating systems," in ACM *Computing Surveys*, December 1976. Two topics of our discussions have stuck in my mind for all these years because the principles were sound and relevant to real systems. They are interrupts and capability addressing.

Interrupts

Roger and I were concerned about the considerable variation in the interpretations of the purpose and operation of interrupt systems, which had been a part of operating systems since the Atlas at University of Manchester in 1959. We saw no clear consensus on their design principles. The Atlas team called them *interrupts* because they were used to interrupt normal processing to allow calls to operating system functions. Other operating systems called them *traps* -- a metaphorical reference to a mousetrap springing in response to a pre-set condition. In describing the Burroughs and Multics operating systems, Elliot Organick called them *unexpected procedure calls*. In their seminal paper, "Programming semantics for multiprocess computations," Jack Dennis and Earl Van Horn (DVH) called them *exceptional conditions* and linked them to the protected entry of any routine providing a function for a class of objects. IBM referred to interrupts as *exceptions*. By 1975 several leading language designers believed that every procedure call, whether to the OS or not, should provide both a normal return and an exception return. The common features of these interpretations were that interrupts gave safe access to supervisory functions of the operating system, stopped programs that encountered error conditions, enabled the operating system to divert to high priority functions, and relied on the procedure calling mechanism. Roger and I were specifically interested in a uniform interpretation of interrupt systems that accommodated these common features and gave clear guidance on how the interrupt hardware and software should be designed for reliability of the whole operating system.

In a nutshell, our conclusions were:

- Interrupt system is at a low kernel level, just above the procedure mechanism.
- The interrupt vector, which points to the routines for handling each type of exception, should encode not only the handler entry points, but their proper supervisor state and interrupt mask settings. The procedure mechanism should set the specified supervisor state and masks on call, and restore them on return.
- Hardware condition detectors notify the dispatcher of faults and external device signals. The detectors for faults could generally be synchronized with the system clock but the detectors for external conditions could not.
- Failure to realize that external condition signals could occur simultaneously led to interrupt dispatchers prone to arbitration failures.

The interrupt system itself consisted of detectors, a dispatcher, a mask, and a vector (list) of interrupt handler routines. The detectors were hardware devices that monitored for pre-set conditions and raised a signal when one occurred. The dispatcher, a combination of hardware and microcode, selected one of the unmasked, raised conditions and invoked a procedure call on the corresponding handler. The mask told which signals to respond to. The vector listed the interrupt handler routines.

One of the open questions concerned the placement of the interrupt system in the functional hierarchy of the operating system. Following the principle of layering, which was gaining popularity since Edsger Dijkstra used it successfully in the THE system, we concluded that the interrupt system belonged in the kernel just above the procedure mechanism, which was just above the instruction set. The interrupt system had to be higher than the procedure mechanism since the dispatcher calls procedures. It had to be lower than everything else, since all other OS functions could define exceptional conditions.

Another open question was how to get the dispatcher to safely enter the CPU supervisor mode when it invoked an interrupt handler, and restore user mode upon return. Entry into the supervisor state had to be coupled tightly to interrupt dispatching lest a separate mechanism become a back door for intruders. We borrowed from the DVH capability idea to describe a clean way to do this. The entries in the interrupt vector would encode the entry point address, the target supervisor mode, and the target interrupt mask. Procedure call would load the instruction pointer, mode, and mask registers simultaneously from these data. Procedure return would restore the former values.

Still another open question was what kinds of conditions should be handled by the interrupt system. Real systems recognized two categories of conditions: faults and external signals. A fault condition meant that the running program could not continue until the detected error was corrected; examples were memory parity, arithmetic, addressing, protection, illegal instructions. An external signal meant that a peripheral device (such as disk) needed an OS action before a deadline; examples were disk completion, receipt of network packet, clock interruption. We did not see any good alternative for separating these two kinds of conditions. Yet there was a crucial difference between them. Errors

could be detected in the CPU between instruction cycles; therefore, the dispatcher always saw a stable set of error condition signals. In contrast, external signals were unconstrained by the CPU clock; therefore, the dispatcher could witness simultaneously arriving device signals and suffer arbitration failures. Arbitration failures are a serious threat to reliability.

David Wheeler and other colleagues had documented arbitration failures that occur when the dispatch circuit is unable to select, within a clock cycle, exactly one of several simultaneously occurring incoming signals. Wheeler argued persuasively that, although the probability of an arbitration failure might appear small (e.g., 1 in 100,000), it is only a matter of a few days before enough interrupts have been processed that a failure is nearly certain. When the failure occurs, the CPU mysteriously hangs up, losing data and requiring a complete cold-restart. Wheeler designed a threshold flipflop (TFF) for the interrupt system that would pause the CPU clock until the TFF indicated it had reached a decision, thereby averting the arbitration failure in exchange for an occasional delay of more than one clock cycle until the TFF correctly registered an interrupt.

Capability Addressing

Roger and I also discussed capability addressing and the structure of capability-based operating systems. Invented by Dennis and Van Horn in 1966, capabilities were long, protected, globally unique addresses for objects. Robert Fabry built a prototype capability machine two years later. Within a few more years the Plessey Company built the System 250, a telephone switching computer that used capability addressing; they reported ultra-high reliability, security, and resistance to software errors. Roger and his colleagues were in the middle of a project to build CAP, a general-purpose capability machine and operating system. Their own preliminary experiments had suggested that such a system would be extremely reliable because errors could not spread outside the local address space in which they occurred.

Roger was extremely worried about the complexity of the CAP operating system. It appeared that the requirement that capabilities be hardware protected from alteration could only be met by partitioning the memory of the machine into separate data and capability parts, which then precipitated a similar partition of the operating system and its data structures into separate data and capability parts. There was a significant problem of maintaining consistency between data and their corresponding capabilities. The complexity was further aggravated by the rigid interpretation of capabilities as “access tickets” for objects. File owners seemed to find it more natural to control access to their files with access control lists than to set up a daemon process to hand out capabilities on request to qualified users. Roger and I discussed possible ways to reduce the complexity to be competitive with other operating systems.

We concluded that the principle of hardware-protected capabilities was the source of much complexity. If we could relax that principle, we could preserve the good features of capability addressing without the cost of special memory or of partitioning. One way to do this would be to use type-checking in compilers to verify that capability arguments passed to system routines were in fact

capabilities. The integrity of capabilities could be guaranteed if the set of OS programs that used capabilities (all layers up through the directory level) were all part of a trusted set assembled and verified by experienced programmers. This might not prevent a determined hacker from penetrating the kernel and modifying capabilities, but it would guarantee the proper use of capabilities for all normal users. Unfortunately, the CAP hardware was already committed to memory partitioning and the OS design was too far along for this to be a realistic option. Besides, compiler technology had not evolved to the point where the required type-checking could be trusted.

We also developed a hybrid access-control method that would combine features of access control lists and access tickets. We observed that an access control list is permanent and persists as long as the file exists. In contrast, a capability list can be a temporary structure that lasts only as long as the associated computational process. After a process is created, its capability list can be loaded (on demand) with capabilities dynamically constructed from the access lists attached to the files holding the objects addressed by the process. This hybrid generalized the standard virtual memory: the mapping tables contain capabilities constructed on the fly from access control lists attached to files. This hybrid was of great interest both to Roger and to Maurice Wilkes. But again the CAP project was too far along to retrofit this.

In their 1979 follow-on book about the CAP operating system, Roger and Maurice lamented that they were unable to reduce the complexity of the system enough to make it competitive with more conventional operating systems. The main benefit, reliable and secure object addressing and sharing, had too large a cost.

Was that the end for these ideas? For designing a system with the reliability of a capability system at the cost of a conventional system? Far from it. These ideas are the backbone of modern object oriented programming systems. The compilers use “handles” to refer to objects -- handles are like software capabilities -- and type checking to assure that handles are passed only to functions authorized to receive them. Objects can be dynamically loaded from external files, to which conventional access lists control access. Although these ideas did not make it in CAP, Roger can nonetheless take pleasure in seeing the technology he helped to develop become a mainstay in computing.