The following paper was originally published in the
Proceedings of the USENIX 1996 Annual Technical Conference
San Diego, California, January 1996

Eliminating Receive Livelock in an Interrupt-driven Kernel

Jeffrey Mogul, DEC Western Research Laboratory
K. K. Ramakrishnan, AT&T Bell Laboratories

For more information about USENIX Association contact:

1. Phone: 510 528-8649
2. FAX: 510 548-5738
3. Email: office@usenix.org
4. WWW URL: http://www.usenix.org

DOCKET ALARM
Find authenticated court documents without watermarks at docketalarm.com.

Eliminating Receive Livelock in an Interrupt-driven Kernel

Jeffrey C. Mogul
Digital Equipment Corporation Western Research Laboratory
K. K. Ramakrishnan
AT&T Bell Laboratories

Abstract

Most operating systems use interface interrupts to schedule network tasks. Interrupt-driven systems can provide low overhead and good latency at low offered load, but degrade significantly at higher arrival rates unless care is taken to prevent several pathologies. These are various forms of ***receive livelock***, in which the system spends all its time processing interrupts, to the exclusion of other necessary tasks. Under extreme conditions, no packets are delivered to the user application or the output of the system.

To avoid livelock and related problems, an operating system must schedule network interrupt handling as carefully as it schedules process execution. We modified an interrupt-driven networking implementation to do so; this eliminates receive livelock without degrading other aspects of system performance. We present measurements demonstrating the success of our approach.

1. Introduction

Most operating systems use interrupts to internally schedule the performance of tasks related to I/O events, and particularly the invocation of network protocol software. Interrupts are useful because they allow the CPU to spend most of its time doing useful processing, yet respond quickly to events without constantly having to poll for event arrivals.

Polling is expensive, especially when I/O events are relatively rare, as is the case with disks, which seldom interrupt more than a few hundred times per second. Polling can also increase the latency of response to an event. Modern systems can respond to an interrupt in a few tens of microseconds; to achieve the same latency using polling, the system would have to poll tens of thousands of times per second, which would create excessive overhead. For a general-purpose system, an interrupt-driven design works best.

Most extant operating systems were designed to handle I/O devices that interrupt every few milliseconds. Disks tended to issue events on the order of once per revolution; first-generation LAN environments tend to generate a few hundred packets per second for any single end-system. Although people understood the need to reduce the cost of taking an interrupt, in general this cost was low enough that any normal system would spend only a fraction of its CPU time handling interrupts.

The world has changed. Operating systems typically use the same interrupt mechanisms to control both network processing and traditional I/O devices, yet many new applications can generate packets several orders of magnitude more often than a disk can generate seeks. Multimedia and other real-time applications will become widespread. Client-server applications, such as NFS, running on fast clients and servers can generate heavy RPC loads. Multicast and broadcast protocols subject innocent-bystander hosts to loads that do not interest them at all. As a result, network implementations must now deal with significantly higher event rates.

Many multi-media and client-server applications share another unpleasant property: unlike traditional network applications (Telnet, FTP, electronic mail), they are not flow-controlled. Some multi-media applications want constant-rate, low-latency service; RPC-based client-server applications often use datagram-style transports, instead of reliable, flow-controlled protocols. Note that whereas I/O devices such as disks generate interrupts only as a result of requests from the operating system, and so are inherently flow-controlled, network interfaces generate unsolicited receive interrupts.

The shift to higher event rates and non-flow-controlled protocols can subject a host to congestive collapse: once the event rate saturates the system, without a negative feedback loop to control the sources, there is no way to gracefully shed load. If the host runs at full throughput under these conditions, and gives fair service to all sources, this at least preserves the possibility of stability. But if throughput decreases as the offered load increases, the overall system becomes unstable.

Interrupt-driven systems tend to perform badly under overload. Tasks performed at interrupt level,

DOCKET ALARM
Find authenticated court documents without watermarks at docketalarm.com.

by definition, have absolute priority over all other tasks. If the event rate is high enough to cause the system to spend all of its time responding to interrupts, then nothing else will happen, and the system throughput will drop to zero. We call this condition *receive livelock*: the system is not deadlocked, but it makes no progress on any of its tasks.

Any purely interrupt-driven system using fixed interrupt priorities will suffer from receive livelock under input overload conditions. Once the input rate exceeds the reciprocal of the CPU cost of processing one input event, any task scheduled at a lower priority will not get a chance to run.

Yet we do not want to lightly discard the obvious benefits of an interrupt-driven design. Instead, we should integrate control of the network interrupt handling sub-system into the operating system's scheduling mechanisms and policies. In this paper, we present a number of simple modifications to the purely interrupt-driven model, and show that they guarantee throughput and improve latency under overload, while preserving the desirable qualities of an interrupt-driven system under light load.

2. Motivating applications

We were led to our investigations by a number of specific applications that can suffer from livelock. Such applications could be built on dedicated single-purpose systems, but are often built using a general-purpose system such as UNIX®, and we wanted to find a general solution to the livelock problem. The applications include:

- *Host-based routing*: Although inter-network routing is traditionally done using special-purpose (usually non-interrupt-driven) router systems, routing is often done using more conventional hosts. Virtually all Internet ''firewall'' products use UNIX or Windows NT™ systems for routing [7, 13]. Much experimentation with new routing algorithms is done on UNIX [2], especially for IP multicasting.
- *Passive network monitoring*: network managers, developers, and researchers commonly use UNIX systems, with their network interfaces in ''promiscuous mode,'' to monitor traffic on a LAN for debugging or statistics gathering [8].
- *Network file service*: servers for protocols such as NFS are commonly built from UNIX systems.

These applications (and others like them, such as Web servers) are all potentially exposed to heavy, non-flow-controlled loads. We have encountered livelock in all three of these applications, have solved or mitigated the problem, and have shipped the solutions to customers. The rest of this paper concentrates on host-based routing, since this simplifies the context of the problem and allows easy performance measurement.

3. Requirements for scheduling network tasks

Performance problems generally arise when a system is subjected to transient or long-term input overload. Ideally, the communication subsystem could handle the worst-case input load without saturating, but cost considerations often prevent us from building such powerful systems. Systems are usually sized to support a specified design-center load, and under overload the best we can ask for is controlled and graceful degradation.

When an end-system is involved in processing considerable network traffic, its performance depends critically on how its tasks are scheduled. The mechanisms and policies that schedule packet processing and other tasks should guarantee acceptable system *throughput*, reasonable *latency* and *jitter* (variance in delay), *fair* allocation of resources, and overall system *stability*, without imposing excessive overheads, especially when the system is overloaded.

We can define throughput as the rate at which the system delivers packets to their ultimate consumers. A consumer could be an application running on the receiving host, or the host could be acting as a router and forwarding packets to consumers on other hosts. We expect the throughput of a well-designed system to keep up with the offered load up to a point called the *Maximum Loss Free Receive Rate* (MLFRR), and at higher loads throughput should not drop below this rate.

Of course, useful throughput depends not just on successful reception of packets; the system must also transmit packets. Because packet reception and packet transmission often compete for the same resources, under input overload conditions the scheduling subsystem must ensure that packet transmission continues at an adequate rate.

Many applications, such as distributed systems and interactive multimedia, often depend more on low-latency, low-jitter communications than on high throughput. Even during overload, we want to avoid long queues, which increases latency, and bursty scheduling, which increases jitter.

When a host is overloaded with incoming network packets, it must also continue to process other tasks, so as to keep the system responsive to management and control requests, and to allow applications to make use of the arriving packets. The scheduling subsystem must fairly allocate CPU resources among packet reception, packet transmission, protocol

DOCKET ALARM
Find authenticated court documents without watermarks at docketalarm.com.

processing, other I/O processing, system housekeeping, and application processing.

A host that behaves badly when overloaded can also harm other systems on the network. Livelock in a router, for example, may cause the loss of control messages, or delay their processing. This can lead other routers to incorrectly infer link failure, causing incorrect routing information to propagate over the entire wide-area network. Worse, loss or delay of control messages can lead to network instability, by causing positive feedback in the generation of control traffic [10].

4. Interrupt-driven scheduling and its consequences

Scheduling policies and mechanisms significantly affect the throughput and latency of a system under overload. In an interrupt-driven operating system, the interrupt subsystem must be viewed as a component of the scheduling system, since it has a major role in determining what code runs when. We have observed that interrupt-driven systems have trouble meeting the requirements discussed in section 3.

In this section, we first describe the characteristics of an interrupt-driven system, and then identify three kinds of problems causes by network input overload in interrupt-driven systems:

- *Receive livelocks* under overload: delivered throughput drops to zero while the input overload persists.
- Increased *latency* for packet delivery or forwarding: the system delays the delivery of one packet while it processes the interrupts for subsequent packets, possibly of a burst.
- *Starvation* of packet transmission: even if the CPU keeps up with the input load, strict priority assignments may prevent it from transmitting any packets.

4.1. Description of an interrupt-driven system

An interrupt-driven system performs badly under network input overload because of the way in which it prioritizes the tasks executed as the result of network input. We begin by describing a typical operating system's structure for processing and prioritizing network tasks. We use the 4.2BSD [5] model for our example, but we have observed that other operating systems, such as VMS™, DOS, and Windows NT, and even several Ethernet chips, have similar characteristics and hence similar problems.

When a packet arrives, the network interface signals this event by interrupting the CPU. Device interrupts normally have a fixed Interrupt Priority Level (IPL), and preempt all tasks running at a lower IPL; interrupts do not preempt tasks running at the same IPL. The interrupt causes entry into the associated network device driver, which does some initial processing of the packet. In 4.2BSD, only buffer management and data-link layer processing happens at ''device IPL.'' The device driver then places the packet on a queue, and generates a software interrupt to cause further processing of the packet. The software interrupt is taken at a lower IPL, and so this protocol processing can be preempted by subsequent interrupts. (We avoid lengthy periods at high IPL, to reduce latency for handling certain other events.)

The queues between steps executed at different IPLs provide some insulation against packet losses due to transient overloads, but typically they have fixed length limits. When a packet should be queued but the queue is full, the system must drop the packet. The selection of proper queue limits, and thus the allocation of buffering among layers in the system, is critical to good performance, but beyond the scope of this paper.

Note that the operating system's scheduler does not participate in any of this activity, and in fact is entirely ignorant of it.

As a consequence of this structure, a heavy load of incoming packets could generate a high rate of interrupts at device IPL. Dispatching an interrupt is a costly operation, so to avoid this overhead, the network device driver attempts to *batch* interrupts. That is, if packets arrive in a burst, the interrupt handler attempts to process as many packets as possible before returning from the interrupt. This amortizes the cost of processing an interrupt over several packets.

Even with batching, a system overloaded with input packets will spend most of its time in the code that runs at device IPL. That is, the design gives absolute priority to processing incoming packets. At the time that 4.2BSD was developed, in the early 1980s, the rationale for this was that network adapters had little buffer memory, and so if the system failed to move a received packet promptly into main memory, a subsequent packet might be lost. (This is still a problem with low-cost interfaces.) Thus, systems derived from 4.2BSD do minimal processing at device IPL, and give this processing priority over all other network tasks.

Modern network adapters can receive many back-to-back packets without host intervention, either through the use of copious buffering or highly autonomous DMA engines. This insulates the system from the network, and eliminates much of the rationale for giving absolute priority to the first few steps of processing a received packet.

DOCKET ALARM
Find authenticated court documents without watermarks at docketalarm.com.

4.2. Receive livelock

In an interrupt-driven system, receiver interrupts take priority over all other activity. If packets arrive too fast, the system will spend all of its time processing receiver interrupts. It will therefore have no resources left to support delivery of the arriving packets to applications (or, in the case of a router, to forwarding and transmitting these packets). The useful throughput of the system will drop to zero.

Following [11], we refer to this condition as *receive livelock*: a state of the system where no useful progress is being made, because some necessary resource is entirely consumed with processing receiver interrupts. When the input load drops sufficiently, the system leaves this state, and is again able to make forward progress. This is not a deadlock state, from which the system would not recover even when the input rate drops to zero.

A system could behave in one of three ways as the input load increases. In an ideal system, the delivered throughput always matches the offered load. In a realizable system, the delivered throughput keeps up with the offered load up to the *Maximum Loss Free Receive Rate* (MLFRR), and then is relatively constant after that. At loads above the MLFRR, the system is still making progress, but it is dropping some of the offered input; typically, packets are dropped at a queue between processing steps that occur at different priorities.

In a system prone to receive livelock, however, throughput decreases with increasing offered load, for input rates above the MLFRR. Receive livelock occurs at the point where the throughput falls to zero. A livelocked system wastes all of the effort it puts into partially processing received packets, since they are all discarded.

Receiver-interrupt batching complicates the situation slightly. By improving system efficiency under heavy load, batching can increase the MLFRR. Batching can shift the livelock point but cannot, by itself, prevent livelock.

In section 6.2, we present measurements showing how livelock occurs in a practical situation. Additional measurements, and a more detailed discussion of the problem, are given in [11].

4.3. Receive latency under overload

Although interrupt-driven designs are normally thought of as a way to reduce latency, they can actually increase the latency of packet delivery. If a burst of packets arrives too rapidly, the system will do link-level processing of the entire burst before doing any higher-layer processing of the first packet, because link-level processing is done at a higher priority. As a result, the first packet of the burst is not delivered to the user until link-level processing has been completed for all the packets in the burst. The latency to deliver the first packet in a burst is increased almost by the time it takes to receive the entire burst. If the burst is made up of several independent NFS RPC requests, for example, this means that the server's disk sits idle when it could be doing useful work.

One of the authors has previously described experiments demonstrating this effect [12].

4.4. Starvation of transmits under overload

In most systems, the packet transmission process consists of selecting packets from an output queue, handing them to the interface, waiting until the interface has sent the packet, and then releasing the associated buffer.

Packet transmission is often done at a lower priority than packet reception. This policy is superficially sound, because it minimizes the probability of packet loss when a burst of arriving packets exceeds the available buffer space. Reasonable operation of higher level protocols and applications, however, requires that transmit processing makes sufficient progress.

When the system is overloaded for long periods, use of a fixed lower priority for transmission leads to reduced throughput, or even complete cessation of packet transmission. Packets may be awaiting transmission, but the transmitting interface is idle. We call this *transmit starvation*.

Transmit starvation may occur if the transmitter interrupts at a lower priority than the receiver; or if they interrupt at the same priority, but the receiver's events are processed first by the driver; or if transmission completions are detected by polling, and the polling is done at a lower priority than receiver event processing.

This effect has also been described previously [12].

5. Avoiding livelock through better scheduling

In this section, we discuss several techniques to avoid receive livelocks. The techniques we discuss in this section include mechanisms to control the rate of incoming interrupts, polling-based mechanisms to ensure fair allocation of resources, and techniques to avoid unnecessary preemption.

DOCKET ALARM
Find authenticated court documents without watermarks at docketalarm.com.

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.