# Interrupts and I/O II [1][2]

IT Systems, Summer Term 2026

Dr. Matthes Elstermann

This is the second presentation on interrupts and I/O processing.

## 1 Introduction

- Part 1
  - Introduction
  - Hack vs Modern Computers
  - Polling
  - Interrupts
- Break for self-study
- Part 2
  - I/O Processing Variants
  - Outlook
  - Conclusions

In part 1, we looked at polling and interrupts for I/O processing. Now, we revisit interrupt processing under high load and investigate latency implications, which are addressed by a hybrid scheme. An outlook and conclusions end the presentation.

## 2 I/O Processing Variants

Let us see some variants of I/O processing.

### 2.1 Recall: I/O with Interrupts

- Recall
  - **Asynchronous** processing of I/O
  - External notifications via interrupts

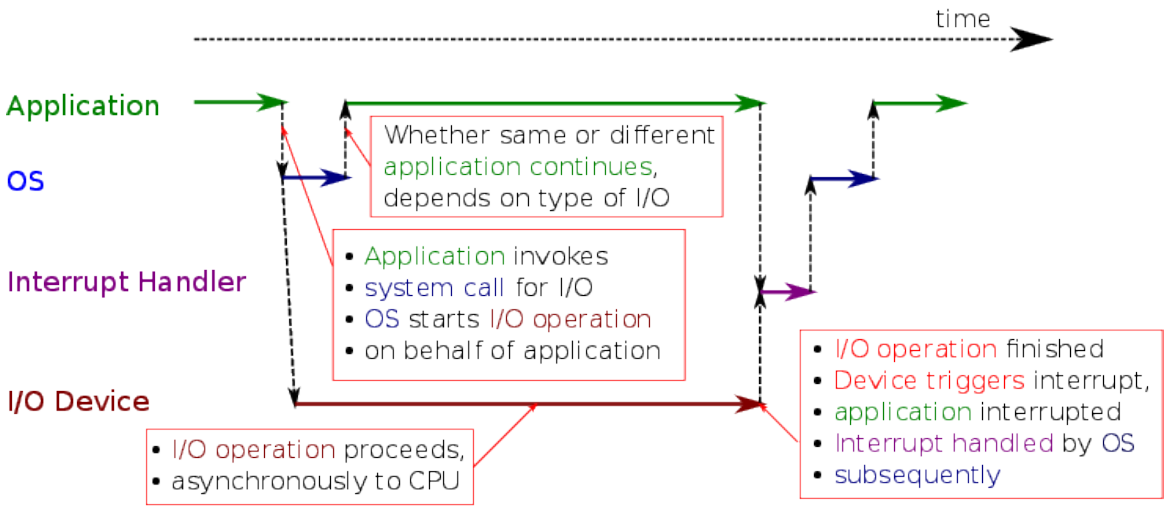


Figure 1: I/O with Interrupts

Recall the process visualized here.

[1] This PDF document is an inferior version of an OER in HTML format; free/libre Org mode source repository.
[2] Material created by Jens Lechtenbörger; see end of document for license information.

## 2.2 Blocking vs Non-Blocking I/O

- Previous slide left open which application continues after I/O system call

  The previous slide showed that after the operating system has initiated an I/O operation, i.e., after it has finished the system call, some application may continue to run asynchronously to the I/O operation. However, the previous slide left open whether the same application that did the system call continues after the system call or whether maybe the operating system switches to some other application.

  First, recall that the OS really schedules threads as part of processes that represent running applications.

  - OS provides blocking **and** non-blocking system calls

    Operating systems provide different types of system calls, in particular blocking and non-blocking ones.
  - **Blocking** system call
    - Thread invoking system call has to wait (is blocked) until I/O completed
    - However, a different thread may continue
      - **Scheduling**, **context switch**, **overhead**

    If a thread invokes a blocking system call, then it has to wait until the corresponding I/O operation is completed. As explained earlier, the operating system blocks that thread. In that case, the invoking thread cannot reasonably continue. In this situation, the OS performs scheduling to choose a different thread for execution. Clearly, scheduling and the context switch to a different thread cause overhead.
  - **Non-blocking** system call
    - OS initiates I/O and returns incomplete result to thread
    - Thread continues (and is informed of or needs to check for I/O completion at later point in time)
    - (Notice: This is impossible with polling)

The alternative is that a thread invokes a non-blocking system call. In that case, the operating system initiates the I/O operation and immediately returns an incomplete result to the calling thread. Now it is the thread's responsibility to figure out if, and when, the I/O operation has actually been completed.

As explained before, when the I/O operation is completed, an interrupt is triggered, which leads to the execution of the corresponding handler. In that handler, the OS changes the previously incomplete result that was passed to the thread: It indicates that the I/O operation has been completed, potentially including the result or a pointer to it.

As a side remark, note that polling is always a blocking form of I/O.

## 2.3 Latency Example (1/2)

- Goal: Explain interrupt overhead as serious challenge if interrupts are frequent
- See (Larsen et al. 2009)
  - Two PCs with Intel Xeon processors (2.13 GHz)
  - 1 Gbps Ethernet networking cards connected via PCIe
  - $1-2$ frames may arrive per 1 µs (1 µs = one millionth second)
    - For the curious
      - Ethernet's unit of transfer: frame with minimum size of 512 b
      - At 1 Gbps, 1000 b need 1 µs for transfer, plus propagation and queueing delays
      - Thus, $1-2$ frames may arrive per 1 µs
  - Interrupt per frame arrival!?
    - What about 10 Gbps networking?

So far, we have seen two different kinds of I/O processing: polling on the one hand and using interrupts on the other. Let us investigate whether one is better than the other.

As mentioned several times, polling seems to come with unnecessary CPU wait time. Indeed, waiting for slow or irregular I/O like networking seems just like a waste of CPU resources. However, interrupt processing comes with context switches and overhead. The question is, how large is that overhead?

Here is an example from the literature, which is about interrupt driven network processing: Two PCs are connected via gigabit ethernet networking, for which some characteristics are shown. Given the minimum frame size and the transfer speed of one gigabit per second, we can expect that in the worst case one or two frames may arrive every microsecond.

The question now is: How long does it take to process those frames if one or two are arriving per microsecond? Will that work? And if you think about even faster networks like 10 gigabit networking, then of course frames may arrive at a much higher rate.

## 2.4 Latency Example (2/2)

- Numbers from (Larsen et al. 2009)
  - Processing of single frame takes **total** of 7.7 µs
  - Latency breakdown according to different sources
    - Hardware: $\approx 0.6$ µs
    - Interrupt processing: $> 3$ µs
    - Processing of data: $> 3$ µs
- If one or two frames arrive per 1 µs and each frame needs 7.7 µs processing time, something is seriously wrong
  - Network data will be **dropped** because it arrives too fast
    - The system could even crash
  - Interrupt per arrival does **not** work

The paper cited here shows that each network frame needs a processing time of 7.7 microseconds. That total of 7.7 microseconds is broken down in great detail in the paper, which you can check out yourself if you are curious.

Obviously, when frames arrive at high speed, up to the maximum of 1 or 2 frames per microsecond, this timing is asking for disaster. Raising an interrupt for every packet arrival on a gigabit network leads to the loss of network data as data cannot be processed as fast as it arrives. In fact, the machine might become unresponsive or even crash.

## 2.5 Interrupt Livelocks

- **Livelock**: Situation in which computations take place but (almost) no progress is made
  - Computation time is mostly wasted on overhead
- Interrupt livelock
  - Interrupts arrive so fast that they cannot be processed any longer
    - Also, not enough CPU time left for other tasks
      - Interrupts served with high priority
    - Context switching, cache pollution
    - Nothing useful happens any more
  - Prevent by hybrid of polling and interrupts
    - E.g., NAPI (New API)

The type of crash mentioned on the previous slide has a technical term, namely livelock. A livelock is a situation in which computation still takes place, but almost no progress is made. In other words, computations that still take place are mostly related to overhead processing.

Now, if the livelock is caused by interrupts, then we are talking about interrupt livelocks. In this case, interrupts arrive so fast that they cannot be processed any longer.

Think about individual packets arriving one per microsecond while each individual packet needs 7.7 microseconds processing time. In this case, the bulk of CPU time is wasted with interrupt overhead processing, which also implies that very little CPU time is left for other tasks. If interrupts are processed with high priority, ordinary tasks will not receive compute time anymore. In addition, there will be lots of context switching and cache pollution, all of which adds to the overhead. Ultimately, nothing useful happens anymore.

A way around this situation is to use a hybrid of polling and interrupts, for example the NAPI, to be discussed shortly.

### 2.5.1 Starvation

- Interrupt livelock is special case of starvation
- **Starvation** = continued denial/lack of resource
  - Under interrupt livelock, threads do not receive resource CPU (in sufficient quantities for progress) as long as "too many" interrupts are triggered

An interrupt livelock is a special case of so-called starvation. Starvation means that some thread does not receive some requested resource for a longer period of time. In that case, the thread cannot make any progress, it starves.

In an interrupt livelock, threads do not receive the resource CPU in sufficient quantities for progress as long as "too many" interrupts are triggered.

- Starvation revisited in later presentations on scheduling and challenges for mutual exclusion

Starvation is an important concept in OSs, which will be revisited in later presentations on scheduling and challenges for mutual exclusion

## 2.6 NAPI

- NAPI: Linux "New API" for networking
  - See (Salim, Olsson, and Kuznetsov 2001)
- Hybrid scheme
  - Use interrupts under low load
    - Utilize CPUs better
      - Avoid polling for devices without data

    NAPI is an acronym for a network processing technique of the Linux kernel, which was introduced a long time ago as "New API". Obviously, it is not a good idea to name something "new something". In fact, kernel documentation clarifies: "The name NAPI no longer stands for anything in particular".

    In any case, NAPI is a hybrid scheme that combines polling and interrupt processing to make most use of the available resources.

    The drawback of polling is that we waste CPU cycles if we are polling for something and that something is not ready yet. In contrast, under NAPI, interrupts are used under low load: If only few packets arrive, then each of these packets will trigger an interrupt, which avoids polling in situations where little data is available.

  - Switch to polling under high load
    - Avoid interrupt overhead
      - Data will be available anyways

    However, if lots of network packets arrive, then the system switches to polling, which avoids the I/O overhead that we just discussed. The assumption is that, under high load, new data will be available anyways, so whenever the operating system looks for new data, it will be there.

    The basic mechanism of NAPI is as follows: Once a network interrupt occurs, the kernel disables this interrupt temporarily and enters polling mode. The kernel then retrieves all available network packets with polling and only enables interrupts again when no more network data is available.

  - See (Cai and Karsten 2023) for performance improvement proposed in 2023
    - Merged into kernel in 2024
    - May reduce energy consumption in datacenters by 30%

    As observed in the research paper cited here, the kernel may enable interrupts too soon: Although the kernel may not have any network data left for processing, the corresponding application thread may not have completed its processing of previously received network data. In this situation, newly arriving network data triggers an interrupt, interrupting the application thread, causing interrupt overhead and context switching.

    To prevent this overhead, the solution proposed in the paper and integrated into the Linux kernel is as follows: Interrupts are only enabled again when the application thread has completed its processing, i.e., when the thread would be blocked by the kernel because it asks for more network data when none is available.

    As reported in the paper, this simple change may lead to impressive performance improvements of 30%, saving considerable amounts of energy in datacenters.

    Note that the implemented technique is somewhat more complex than described here, e.g., it includes timeouts as safety mechanism to enable interrupts again when applications fail.

# 3 Outlook

There is more to I/O processing, with newer hardware raising new challenges. Beyond learning objectives, this section mentions some trends.

## 3.1 When to Poll?

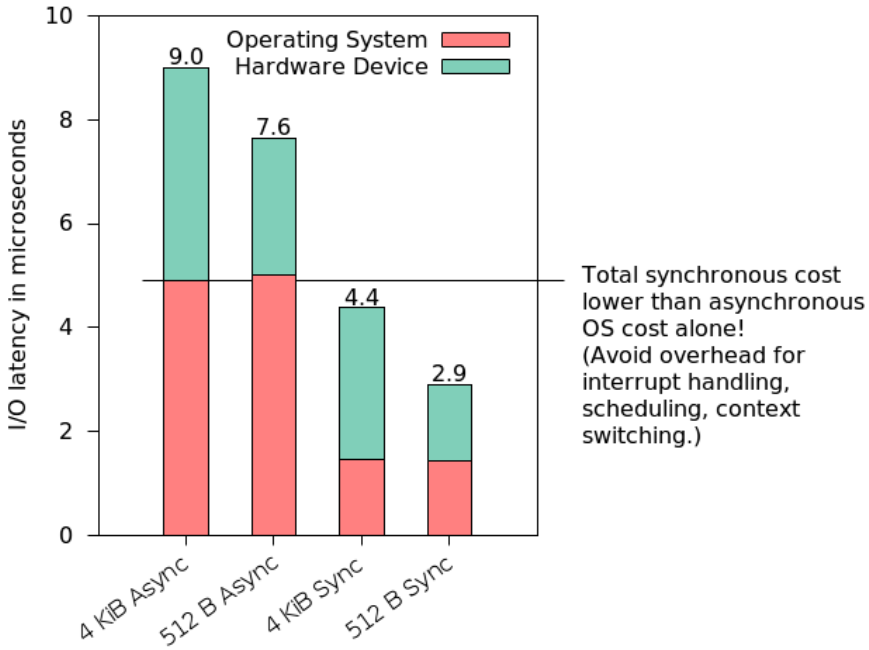


Figure 2: Measurements for DRAM-based storage prototype (data from Yang, Minturn, Hady (2012))

(Source for numbers: (Yang, Minturn, and Hady 2012))

This figure shows data from a research paper dating back to 2012 when non-volatile memory devices gained traction as alternative to slow disks. Some details for such devices are mentioned on the next slide. Essentially, these devices deliver data so fast, both with low latency and high bandwidth, that polling is more efficient than interrupt-driven processing.

For such devices, the two bars to the left show interrupt-driven processing times, while the two bars to the right show times for polling. Note that the total time for polling is smaller than the OS overhead for interrupts alone.

Thus, while interrupts are useful for high-latency devices such as hard disks, polling is preferable for low-latency devices.

## 3.2 I/O Processing – Then and Now

- Then: Disks are slow
  - Mechanical devices
  - Delivered data is processed immediately by CPU
  - Latency before data arrives → Interrupts beneficial
- Now: Nonvolatile memory is fast, see (Nanavati et al. 2016)
  - Mechanics eliminated
  - Operation at network/bus speed (PCIe)
  - Data can be delivered faster than processed → Polling beneficial
  - Need to rethink previous techniques
    - Balancing, scheduling, scaling, tiering

The thoughts from the previous slide are revisited in the paper cited here. Basically, this paper explores the change in data centers from slow hard disks, around which software architectures were built for decades, to fast nonvolatile memory. The authors argue that many layers of the software infrastructure require innovations to build efficient systems that take advantage of fast nonvolatile memory.

## 3.3 Call for Research

- (Barroso et al. 2017): Attack of the Killer Microseconds
  - Nanosecond latency (DRAM access when data not in CPU cache) is hidden by CPU hardware
    - Out-of-order execution, branch prediction, multithreading (two threads per core)
    - (However, also ongoing research to address Killer Nanoseconds (Jonathan et al. 2018))

In the context of the memory hierarchy, you saw that RAM is slow compared to CPU operations, introducing latency on an order measured in nanoseconds. This type of latency can be hidden with caching and CPU optimizations shown here.

- Millisecond latency (disk I/O) is hidden by OS
  - Multitasking

  The relatively large latency in the range of milliseconds of hard disks is hidden by the OS with multitasking: While one thread waits for a slow I/O operation to complete, the OS schedules another thread for execution.
- What about microseconds of new generation of fast I/O devices?
  - E.g., Gbps networking, flash memory
  - Paper describes datacenter challenges experienced at Google

  However, gigabit networking and flash memory introduce latency at the scale of microseconds. This scale is too large to be hidden by CPU optimizations and too small to outweigh the overhead of context switching for multitasking. The paper cited here calls for research into infrastructure stacks that are optimized for microsecond latency.

# 4 Conclusions

Let us conclude.

## 4.1 Summary

- Interrupt handling is major OS task
  - I/O processing
  - Timers, to be revisited for scheduling
  - System call implementation
- Polling vs interrupt-driven I/O
  - Efficiency trade-off
  - Interrupt livelocks and NAPI

Interrupt handling is a major OS task with applications for I/O processing, timers, and the implementation of system calls. Notably, timers will be revisited for scheduling.

Polling and interrupt-driven input/output processing come with complementary strengths and weaknesses. To avoid interrupt livelocks, hybrid schemes can be used.

### Bibliography


Barroso, Luiz, Mike Marty, David Patterson, and Parthasarathy Ranganathan. 2017. "Attack of the Killer Microseconds." *Cacm* 60 (4): 48–54. https://dl.acm.org/citation.cfm?id=3015146.

Cai, Peter, and Martin Karsten. 2023. "Kernel Vs. User-Level Networking: Don't Throw out the Stack with the Interrupts." *Proc. Acm Meas. Anal. Comput. Syst.* 7 (3). https://doi.org/10.1145/3626780.

Jonathan, Christopher, Umar Farooq Minhas, James Hunter, Justin Levandoski, and Gor Nishanov. 2018. "Exploiting Coroutines to Attack the 'Killer Nanoseconds'." *Proc. Vldb Endow.* 11 (11): 1702–14. https://doi.org/10.14778/3236187.3236216.

Larsen, Steen, Parthasarathy Sarangam, Ram Huggahalli, and Siddharth Kulkarni. 2009. "Architectural Breakdown of End-to-End Latency in a TCP/IP Network." *Int. J. Parallel Prog.* 37 (6): 556–71. http://link.springer.com/article/10.1007/s10766-009-0109-6.

Nanavati, Mihir, Malte Schwarzkopf, Jake Wires, and Andrew Warfield. 2016. "Non-Volatile Storage – Implications of the Datacenter's Shifting Center." *Acm Queue* 13 (9). https://queue.acm.org/detail.cfm?id=2874238.

Salim, Jamal Hadi, Robert Olsson, and Alexey Kuznetsov. 2001. "Beyond Softnet." In *Proceedings of the 5th Annual Linux Showcase & Conference*. Oakland, California: USENIX Association. https://www.usenix.org/publications/library/proceedings/als01/full_papers/jamal/jamal.pdf.

Yang, Jisoo, Dave B. Minturn, and Frank Hady. 2012. "When Poll Is Better than Interrupt." In *Fast 2012*. https://www.usenix.org/conference/fast12/when-poll-better-interrupt.


The bibliography contains references used in this presentation.

# License Information