OS02: Interrupts and I/O *

Jens Lechtenbörger

Computer Structures and Operating Systems 2023

1 Introduction

1.1 OS Plan

- OS Overview (Wk 20)
- OS Introduction (Wk 21)
- Interrupts and I/O (Wk 21)
- Threads (Wk 23)
- Thread Scheduling (Wk 24)
- Mutual Exclusion (MX) (Wk 25)
- MX in Java (Wk 25)
- MX Challenges (Wk 25)
- Virtual Memory I (Wk 26)
- Virtual Memory II (Wk 26)
- Processes (Wk 27)
- Security (Wk 28)

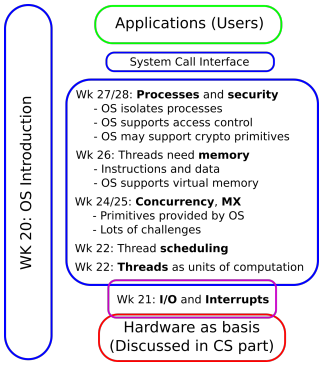

Figure 1: OS course plan, summer 2022

1.2 Today's Core Questions

- Recall keyboard handling in Hack.
  - You used a loop to wait for I/O; this is called **polling**.
    - Fill.asm of project 4.
- How can we improve I/O processing over polling?
  - Why keep CPU busy in loop when nothing happens?

*This PDF document is an inferior version of an OER HTML page; free/libre Org mode source repository.

  - With multitasking such time is **wasted** as other tasks could make better use of the CPU.

- Add **interrupts** as I/O notification mechanism.
  - How to organize I/O then?
  - How much overhead arises?
    * (Overhead: Additional indirect computation time; makes system less efficient.)
    * How to deal with "lots" of I/O events?

1.3 Learning Objectives

- Explain techniques for I/O communication
  - Including sequencing of events under synchronous and asynchronous techniques (polling vs interrupts)
- Discuss dis/advantages of I/O communication techniques
- Explain (interrupt) livelock and mitigation via hybrid technique

1.4 A Note on Literature

- As an exception, this presentation is not based on [Hai19].
  - In [Hai19], Section 2.5 contains some introductory paragraphs on interrupts, while Section 7.3.1 starts with explanations on privilege levels and system calls.
- Chapter 1 of [Sta14] as well of [TB15] contain introductions on interrupts and I/O, while Section 5.1 of [TB15] has additional explanations.

1.5 Retrieval Practice

- Before you continue, answer the following; ideally, without outside help.
  - How does the von Neumann architecture look like?
  - How does I/O processing work with Hack?
  - How to talk to an OS kernel?
  - How are programs, processes, and threads related?
  - What is multitasking? What are its advantages?

1.5.1 Von Neumann and Hack Architecture

1.5.2 Computations

Table of Contents

2 Hack vs Modern Computers

2.1 Recall: Von Neumann Architecture

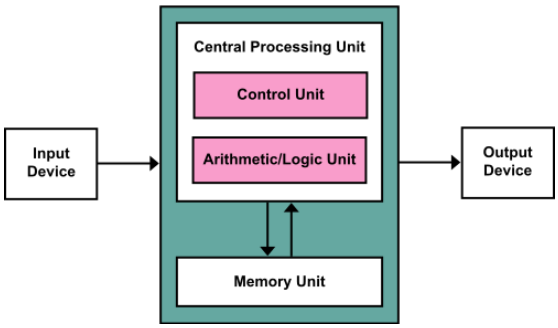

Figure 2: "von Neumann Architecture" by Kapooht under CC BY-SA 3.0; converted from Wikimedia Commons

2.2 Hack vs Modern Computer

	Hack	Modern Computer (e.g., PC, smartphone)
Memory	• RAM for data, ROM for instructions • Physical RAM addresses • No secondary memory	• RAM for data and instructions • Physical and virtual addresses • Memory hierarchy (disks)
CPU	• Single core • Single mode of execution • Neither cache nor MMU	• Multi-core • Multiple protection domains • Caches and MMU
I/O	• No interrupts • Polling for I/O (recall keyboard)	• Interrupts, DMA • Different options for I/O
OS	• Language library • Single thread, no multitasking • No virtual memory	• Real OSs with system calls • Multitasking, scheduling • Virtual memory

This slide highlights key differences between the Hack platform and modern computers. We have already seen that data and instructions are kept separately in Hack, which is called the Harvard architecture, while modern Computers keep data and instructions in RAM, which follows more closely the original von Neumann design. Then you know that in Hack there are physical RAM addresses which are the bits that are actually fed into the memory chips, while modern CPUs also support virtual addresses, which will be the topic of a separate lecture. Also, modern computers come with a memory hierarchy, in particular including stable storage which is absent in the case of Hack. Then modern CPUs typically contain multiple cores; each of which you can think of as a full-fledged Hack CPU. In addition, modern CPUs support multiple protection domains which for example allows to protect the operating system from interference of user applications. Also, modern CPUs contain a memory management unit which is used in the context of virtual addresses and which therefore will also come back in

later lectures. With respect to I/O processing you already saw the programming technique called polling in the Hack platform to access the keyboard and you can also do something like that in modern computers. However, they also support interrupts and so-called direct memory access which requires additional hardware and is used for bulk transfer of data for example in the context of graphics cards. So, modern computers support different types of I/O processing and this will be the topic for today. Finally, with respect to operating systems, Hack does not really have any operating system in a modern sense. If you take a look at the final book chapter, you'll see something called operating system but it's really just a language library and it does not support any of the major features of modern operating systems, such as multitasking or virtual memory.

2.3 CPUs in the Real World

- Known from Hack
  - Registers (addresses, data, control information)
  - Instruction execution cycle (e.g., fetch, decode, execute)
- Additionally
  - Caching
    - Cache = Small, fast memory between CPU and RAM
      - (Usually, multiple levels of caches; L1, L2, . . .)
    - Instructions and data must be in cache before CPU can access them
      - Replacement policies when full (similar to those for memory management)
      - Overhead for context switches, cache pollution
  - Special instructions and modes
    - Access to memory and devices in kernel mode: Subsequent slide
    - Enter idle state when nothing to do (save power)
  - Interrupts: Later slides

(Audio for this slide is split into several audio files, one for each step of the animation. In contrast, these notes contain a transcript of all animation steps. The same is true for other animations.)

Modern CPUs have several characteristics that are important in the contexts of OSs and performance. First, you know that according to the von Neumann architecture, the CPU fetches instructions for execution from RAM. Nowadays, CPUs are equipped with additional, fast but small memory chips called caches, which sit between RAM and processors, and data and instructions are loaded from RAM to a cache before the CPU accesses them. Subsequent accesses to cached data or instructions are much faster than original accesses in RAM, improving performance considerably. (Usually, there is even a hierarchy of multiple caches of decreasing size and increasing speed between RAM and CPU, but this is not important for our purposes. You can find latency estimates on Stack Overflow.)

As caches are small, they come with replacement policies in situations where the cache is full and new contents need to replace old contents. We will see such policies in the context of memory management later on; if you are interested in details, maybe check out cache coherence protocols on Wikipedia.

For now, note that computations are fast if lots of memory accesses can be served from caches instead of from RAM. However, you already learned that context switches happen between user space and kernel space, which as special case includes switching from user space to kernel space with scheduling of a different thread. In all those cases, the new execution context needs data and instructions that differ from the currently cached ones. Thus, context

switches come with overhead in the form of loading new entries from RAM into caches. When returning to the old context, previously cached entries may have been replaced, which is also called *cache pollution* and again requires slow accesses to RAM.

Furthermore, CPUs may support special instructions and execution modes to isolate kernel space from user space and user spaces from each other, for which basics are explained on the next slide. Besides, CPUs may come with special instructions to enter so called *idle states*, which can be executed by the OS to save power when no thread wants to execute any instruction. The link on this slide leads to details for Linux.

Importantly, CPUs have additional input pins or buses on which devices can trigger so-called *interrupts* to signal events that should be processed by the OS. Such interrupt processing is the topic of this presentation.

2.4 Privilege Levels/Rings/Modes

- Hierarchical **protection** domains of CPUs
  - At least **kernel mode** vs. **user mode** (see Sec. 7.3.1 of [Hai19])
    - E.g., **4 rings** since Intel 80286
      - Typically, ring 0 is kernel mode (most privileged), ring 3 is user mode (least privileged)
    - Governed by bit pattern in special register
- **Instruction set restricted** depending on mode/privilege level
  - Special registers protected
  - I/O, memory management protected
- OS starts in **kernel mode**
  - Applications run in **user mode**
  - Interrupts (system calls, traps, ...) lead into kernel mode

Modern processors support protection domains which means that pieces of code can actually be executed at different privilege levels. For our purposes, it would be sufficient to consider just two privilege levels, the so-called kernel mode which has full access to the underlying hardware and the user mode which is restricted and which is the mode in which ordinary application are run. So for example early Intel processors already supported 4 rings, where ring 0 was the kernel mode, the most privileged mode, while ring 3 is the one at which user applications run. A bit pattern in a special register controls in which mode the processor is currently operating.

The key idea of protection domains is to restrict the instruction set which the CPU is currently able to execute. And that restriction depends on the mode or privilege level within the CPU is currently executing. So what instructions to allow or forbid? If you think about it, because the current mode is recorded in a special register, that register of course needs to be protected because otherwise any code running in user mode could just access that register and elevate its own privilege level. So certain special registers are protected. In addition, input-output operations and memory management operations are also protected depending on the current privilege level.

The big picture for the use of different protection domains is as follows. The operating system starts in kernel mode with the highest privilege level, so it is allowed to do whatever it wants to do and has full access over the underlying hardware. At some point in time it starts the first user application and it starts that application in user mode. So the user application is restricted in terms of the instructions that can be executed and if it wants to perform input-output operations, for example, then it needs to perform a system call so that the operating system can perform that operation in its kernel mode on behalf of the user operation. In particular, system calls somehow need to switch the CPU from user mode into kernel mode. Similarly interrupts also need to lead into kernel mode and I'll have more to say about that in the following. As a side remark, I'd like to mention that I'm simplifying here. In particular

CPUs with virtualisation support may also have something like a privilege level -1. The idea there would be that guest operating systems are actually running in a kernel mode but still do not have full control over the hardware, which is reserved to some supervisor component which runs at privilege level -1.

2.5 Big Picture

- I/O devices are components that **interact** with the OS
  - Some **receive requests** and **deliver results**
    - E.g., disk, printer, network card
  - Some **generate events** on their own
    - E.g., timer/clock, keyboard, network card
- Alternative types of I/O
  1. OS **polls** (continuously asks) for events/results
  2. Device triggers **interrupt** when event occurs or result is ready
     - (Historically, special CPU input pins/bits were used to signal interrupts; now, also existing buses are used, e.g., MSI)
     - CPU interrupts current computation and jumps into OS, which **handles** interrupt

2.5.1 Types of I/O

- With polling, I/O is called **synchronous**
  - OS monitors I/O operation for completion
- With interrupts, I/O is called **asynchronous**
  - OS initiates I/O
  - I/O proceeds asynchronously, CPU free to perform other tasks
  - Device triggers interrupt when I/O operation completed
    - CPU interrupted, I/O result handled by OS

3 Polling

3.1 I/O with Polling

I/O happens **synchronously** while OS polls for result

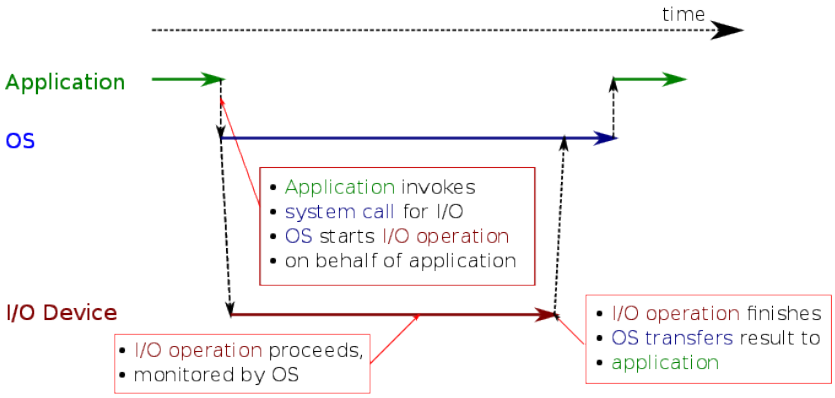

Figure 3: I/O with Polling

Let's see how I/O works with polling. The key characteristic here is that while an I/O operation is ongoing, the CPU does nothing else but synchronously check whether the operation has successfully completed already. So, the key ingredients here are an application that would like to perform some I/O operation, then the operating system which needs to perform the I/O operation on behalf of the application. Recall that the application is run in user mode while the operating system runs in Kernel mode and is thus responsible for I/O operations. And, of course, there is the I/O device, for example a keyboard.

After some normal processing the application invokes a system call to have the operating system perform that I/O operation. So, the operating system will start that operation on behalf of the application at the device.

While the I/O operation proceeds, the operating system continuously monitors its progress and waits for the result.

At some point in time the I/O operation finishes, the operating system obtains the result, and transfers it to the application, which can then continue.

3.2 Polling Observations

- Advantages
  - Simple
  - Fast
    - Result processed as soon as it is available
      - No overhead (compared to interrupts as presented subsequently)
- Disadvantage
  - **Busy waiting** = waiting on CPU for event to occur
    - CPU time wasted if I/O is slow or infrequent
    - Bad idea if wait period is "long"

The question is, how good is polling as an I/O processing technique? It turns out that this question is a hard one, and the entire presentation deals with that question. Let's take a look at advantages and disadvantages of polling.

The major advantage of polling is that it is simple to program. Think of a thread that would like to perform an I/O operation, for example reading a file from disk. Quite likely the thread invokes the I/O operation because it needs the file contents to proceed in a reasonable manner. So the simplest thing to do is to issue that I/O request and then just "to poll," i.e.,

to check continuously whether the data has arrived already and not to do anything else. This is certainly a simple approach.

However, waiting for something that is happening externally looks like a waste of CPU time: The CPU could theoretically do something useful but all that it does is wait for I/O to happen.

An alternative would be to program the thread in such a way that it can do something else *while* disk data is being transferred. Apparently, that would make the program much more complex. Also, polling is faster than this alternative. If a thread actively waits for data from disk, then it can respond to that data as soon as it is available, no overheard involved at all. This point will become much clearer once we've taken a look at the overhead involved in interrupt processing.

Polling is a special instance of a technique called *busy waiting*. Busy waiting is also discussed in the book by Hailperin in the context of scheduling, which is the topic of a subsequent presentation.

3.2.1 Polling Example: Hack Keyboard

- Program waits for user to press key
  - Loop, repeatedly reading keyboard memory location until key pressed
  - Recall Fill.asm
- CPU executes instructions all the time
- Even if Hack had OS with multitasking, nothing else could be executed
  - CPU time in loop is **wasted**

4 Interrupts

4.1 Fundamental Idea

- **Interrupt**: Signal to CPU
  - Generated **externally** to CPU (signal on bus or separate pin)
    - CPU stops doing whatever it did
    - CPU jumps (resets program counter) to **interrupt handler** instead (details on following slides)
- If I/O devices generate **interrupts**, CPU does **not** need to wait for I/O completion
  - OS **initiates** I/O operation at device
    - CPU is free to do something else **asynchronously** during I/O execution
  - At later point, I/O operation completes and **device triggers an interrupt**
    - OS interrupt handler acts accordingly

Interrupts are similar to function calls in the sense that if a function gets called then code elsewhere in memory gets executed. Similarly, when an interrupt gets triggered, the CPU will stop doing whatever it did so far and instead it will jump to an interrupt handler which is provided by the operating system. The major benefit of introducing interrupts is that the operating system no longer needs to actively wait for the completion of potentially long-running I/O operations but instead just needs to initiate I/O operations and then is free to do something else asynchronously, while the I/O operation is ongoing. And if at some later point in time the I/O operation completes, the device triggers or raises an interrupt and the operating system's interrupt handler will act accordingly.

4.2 I/O with Interrupts – Overview

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

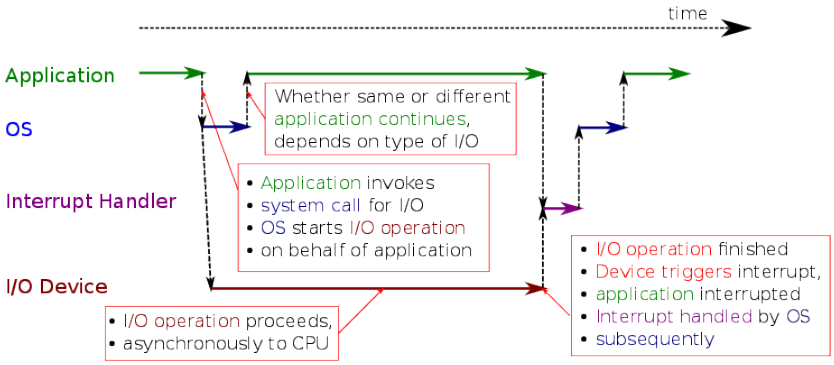

Figure 4: I/O with Interrupts

This slide shows the big picture of I/O processing using interrupts. I really recommend that you contrast this slide with the earlier one shown for I/O processing with polling. So similarly to the polling case, we will be looking at the application, the operating system, and the i/o device. In addition now, we will also take a look at the interrupt handler, which is the specific part of the operating system that is responsible for handling interrupts.

Exactly as in the case of polling an application first may perform arbitrary instructions and then at some point in time invoke the system call to perform an I/O operation. So again, the operating system needs to do that. So again, the operating system is responsible for starting the I/O operation on behalf of the application.

This time however, the operating system only initiates the I/O operation and is then free to do something else while the IO operation proceeds asynchronously.

So after the operating system has performed all housekeeping and management operations that it needs to perform, an application can continue running. Actually whether the previously running application or a different application is going to continue depends on the type I/O and we'll take a look at that separately.

When the I/O operation is finished, the corresponding device triggers an interrupt and that interrupt then leads to execution of the interrupt handler of the operating system, which takes care of necessary management operations.

Finally the operating system allows some application to continue.

4.2.1 Hypothetical Interrupt Example: Hack Keyboard (1/3)

- Suppose Hack had multitasking OS with threads and interrupts (which it does **not**)
  - Multiple programs could run concurrently in separate threads
- Again, wait for user to press key
  - This time, thread invokes (blocking) system call, asking OS for next pressed key
    - (Non-blocking system calls exist as well, discussed later)
  - OS remembers to inform that thread about keyboard input, **blocks** it

        * OS takes that thread aside
        * OS dispatches different thread to execute on CPU

4.2.2 Hypothetical Interrupt Example: Hack Keyboard (2/3)

- Eventually, key gets pressed
  - Keyboard triggers interrupt
    * CPU interrupts whatever it does, jumps to interrupt handler, which interacts with hardware to obtain value representing key
    * OS records that value as return value of system call
      · OS **unblocks** blocked thread
  - Scheduling decision by OS determines what (unblocked) thread to continue next
    * At some point in time, maybe right now, OS chooses thread that invoked the keyboard system call
      · Thread continues with return value from system call

4.2.3 Hypothetical Interrupt Example: Hack Keyboard (3/3)

- Notice: Latency (delay) between system call and processing of its return value
  - Latency between system call and key press
    * Cannot be avoided
    * In contrast to polling, with interrupts something useful can happen on the CPU during this period
  - Between key press (interrupt) and return of key's value into thread
    * Processing of interrupt with **overhead**
      · Discussed subsequently
    * Resulting delay does not exist for polling

4.3 Dijkstra on Interrupts

- "It was a great invention, but also a Box of Pandora. Because the exact moments of the interrupts were unpredictable and outside our control, the interrupt mechanism turned the computer into a nondeterministic machine with a nonreproducible behavior, and could we control such a beast?"
- "When Loopstra and Scholten suggested this feature for the X1, our next machine, I got visions of my program causing irreproducible errors and I panicked."

In retrospect, adding interrupts to computers may sound simple to you given that they are around everywhere, essentially, but this slide shows two quotes by Dijkstra on what he thought when interrupts were about to be added to computers and that tells you that we really need to be careful. If you're interrupted at some point in time, then of course the question is, how can you make sure that whatever you did while you were interrupted, can be resumed later on, once the interrupt has been properly treated? For example, if right now an interrupt occurs raised by your mobile phone because some alert from some social message

or whatever occurs, then it's not really clear that you will be able afterwards to continue the thought, which was interrupted by your phone. Therefore, I suggest that, while you are working your way through these slides, you switch off your mobile phone or at least you place it somewhere, where its interrupts cannot distract you. Actually also CPUs have got a feature to turn off interrupts, which may be used by the operating system to make sure that it does not get interrupted, when it doesn't want to.

4.4 Interrupts, Traps, Faults, Exceptions

- Interruption of ordinary CPU execution
- Hardware-specific, terminology not unified
- Classification
  - **Asynchronous**, triggered externally
    - **Hardware interrupt** (e.g., key pressed, data received)
      - **Timer** (clocks); basic mechanism in scheduling presentation
  - **Synchronous**, triggered internally
    - **Software interrupt**
      - Before execution of instruction, e.g., **page fault** as basic mechanism in virtual memory presentation)
      - After execution of instruction (e.g., overflow)

Here, you see various terms related to interrupts. Interrupts, traps, faults, exceptions. Which is which is not really well defined and varies from textbook to textbook and processor to processor. In all cases we are talking about interruption of the ordinary CPU execution. An accepted classification of those interruptions is into the classes of asynchronous and synchronous interruptions. Note that asynchronous and synchronous are meant in their literal meaning here and they are not directly related to synchronous and asynchronous I/O processing discussed previously. Asynchronous interruptions are those that are triggered externally to the CPU, for example hardware interrupts that I talked about earlier; so, if a key is pressed or if a disk has transferred a block of memory. But also timers can raise interrupts. So we can have interrupts that are triggered periodically and that will be important for scheduling purposes later on. In contrast, synchronous interruptions are those that are triggered internally to the CPU, one example are software interrupts that we'll take a closer look at in a minute and also interruptions that occur either before or after the execution of instructions. So, page faults will become important in the context of virtual memory management later on, while interruptions after instructions will not be important for our purposes.

4.5 Interrupts and CPUs

- OS specifies a **handler** for each type of interrupt and exception
  - Handler = function
  - Type of interrupt determined by number
- E.g., x86 processors
  - Addresses of handlers stored by OS in in-memory table, the **Interrupt Descriptor Table** (IDT) (synonym: Interrupt Vector (Table))
    - Each table entry points to one handler/function
  - CPU (each core) contains an **Interrupt Descriptor Table Register** (IDTR)

        * OS initializes IDTR with start address of IDT

With this and the subsequent slide I am going to explain in a little bit more detail, how interrupts on our PC-CPUs are really processed. From previous explanations, it should already be clear that there are different types of interrupts and exceptions, for example timer interrupts, interrupts when keys are pressed, and also page faults. And now the operating system specifies a handler for each type of interrupt. You should think of such a handler just like a function, and the type of the interrupt is identified by some number. Now the operating system stores addresses of all these functions, all these handlers, in an in-memory table, which is called interrupt descriptor table. Sometimes you also see the term interrupt vector table for that purpose. So, the idea is: this table contains an entry for each handler that tells where that handler resides in main memory. In addition, the CPU has a specific register that keeps the start address of that interrupt descriptor table, the IDTR. So the operating system, which has created the interrupt descriptor table in the first place, places the start address of that table into the IDTR.

4.6 Interrupt Handling

- Upon interrupt of type n:
  - A **context switch** takes place, and (in kernel mode) the CPU
    * saves state of current execution,
    * uses IDTR to access IDT,
    * looks up entry n in IDT, and invokes corresponding handler/function.
    * Afterwards, state is restored, previous execution continued.
- Context switch comes with **overhead**
  - Save/restore state
  - Cache pollution
    * Cache contents unlikely to be useful in new context
      · (In particular, this affects the so-called TLB, to be discussed in virtual memory presentation)
  - Maybe scheduling (later presentation)
    * With setup of new address space (virtual memory presentation)

Whenever an interrupt occurs, a so-called context switch takes place. So that means the CPU gets interrupted in whatever it is currently doing. To resume that interrupted execution, it needs to save the state of the current execution and then it uses the IDTR to figure out where the start address of the interrupt descriptor table is. Based on the type of interrupt that has just occurred it looks at the corresponding entry in the interrupt descriptor table, figures out the start address for that handler, and jumps to that location. And then the interrupt gets handled by the corresponding function. Once that has finished the CPU can restore the saved state and resume the previously interrupted execution. Note that such a context switch always comes with some overhead. So for example the CPU needs to save and restore the state; depending on what exactly happens in the course of interrupt handling, it may also be necessary to set up separate address spaces. Then continue execution of instructions elsewhere always leads to so-called cache pollution. So entries that were previously useful in the CPU cache are no longer useful in the context of the interrupt handler, and therefore they may be evicted from the cache and replaced by an interrupt handling specific data and instructions, so that when later on the real the original functionality resumes its data and instructions will no longer be present in the cache. Similar effects happen for the so-called Translation Lookaside Buffer which deals with virtual address translation, which will come back in a later lecture. And maybe there will be scheduling involved to figure out what to do next. So, all these things may lead to overhead and may actually slow down the system.

4.7 Aside: Interrupts for System Calls

- Recall: System calls = API provided by OS kernel
  - Implementation is OS and hardware specific
- Hardware specific methods to **enter kernel mode**
  - `int 0x80` (generic), `SYSCALL` (AMD), `SYSENTER` (Intel)
- Beyond class: Linux, Intel IA-32, `int 0x80`
  - **Software interrupt** via `int 0x80` leads into kernel mode
  - IDTR contains address of IDT
    - Entry `0x80` points to handler function
      - In the past `system_call`, since 2015 `entry_INT80_32`
    - Initialization during boot (`arch/x86/kernel/traps.c`)

A previous slide contained the term software interrupt as an example for a synchronous interruption of the CPU. Now let's take a look at how this kind of interrupt can actually be used to implement system calls. Recall that system calls are the API, the application programming interface provided by the OS kernel. The question is how could a system call be implemented? Remember that the OS is supposed to run in kernel mode while user applications that invoke system calls run in the CPU's user mode. Essentially, the OS needs to use some mechanism which switches the CPU from user mode to kernel mode, and depending on the CPU architecture there are different hardware specific methods, namely different machine instructions, to enter the kernel mode. The slide names a couple of those. For example, on a 32-bit Intel processor the instruction int 0x80 triggers a software interrupt, which switches the CPU into kernel mode, where the IDTR is used to lookup the start address of the IDT as explained previously. The IDT in turn is then used to lookup the start address of the specific function or handler to execute. In the case of Linux, that function was simply called $\text{system}_{\text{call}}$ in the past, but naming changed in 2015, and you can read more about that change at the URL linked here. Also, you can take a look at the source code file traps.c linked here to see all the details of initialization.

4.8 Self-Study Quiz

This task is available for self-study in Learnweb.

- Consider a networked machine that receives incoming messages. Each of those messages requires about 4 µs CPU time for processing. If interrupts are used, each interrupt introduces a delay of about 6 µs (caused by different types of overhead).
  - How many messages can be processed per second with polling, how many with interrupts?
  - How much time is wasted (waiting for messages to arrive) with polling in the worst case? How much spent on overhead processing with interrupts?

5 Interrupts and I/O Communication

5.1 Recall: I/O with Interrupts

- **Asynchronous** processing of I/O

– External notifications via interrupts

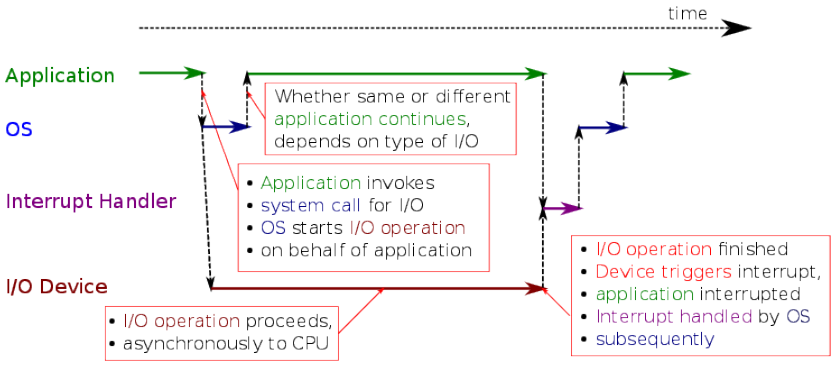

Figure 5: I/O with Interrupts

5.2 Blocking vs Non-Blocking I/O

- Previous slide left open which application continues after I/O system call
  - OS provides blocking **and** non-blocking system calls
- **Blocking** system call
  - Application has to wait (is blocked) until I/O completed
  - However, a different application may continue
    - **Scheduling**, **context switch**, **overhead**
- **Non-blocking** system call
  - OS initiates I/O and returns incomplete result to application
  - Application 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 previous slide showed that after the operating system has initiated an I/O operations - so 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 will continue after the system call or whether maybe the operating system switches to some other application. So, let's now take a look at this question. It turns out that the operating system provides different types of system calls, in particular blocking and nonblocking system calls. If an application invokes a blocking system call, then it has to wait until the corresponding I/O operation is completed. So the operating system blocks that application. In that case the invoking application cannot reasonably continue. However, a different application may continue and that means that the operating system needs to do or perform a scheduling decision and switch to that separate application, which involves a context switch with additional overhead. The alternative is that an application 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 application. Now it is the application'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 will be triggered, that leads to the execution of the corresponding handler and then the operating system will change the incomplete result that was passed to the application and mark that the I/O operation has actually been performed and maybe also return necessary result data to the application. Also it should be clear that if polling is used for I/O processing, I/O can only be blocking.

6 I/O Processing

6.1 Latency Example (1/2)

- Goal: Explain interrupt overhead as serious challenge if interrupts are frequent
- See [Lar+09]
  - 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've seen two different kinds of I/O processing: polling on the one hand and using interrupts on the other hand. So the question is: Which is actually better? If you think about it, polling seems to come with unnecessary CPU wait time. So waiting for slow I/O like networking seems just like a waste of CPU resources. However, I also said that interrupt processing comes with context switches and overhead. The question is, how large is that overhead? Here is an example from the literature. This is about interrupt driven network processing. Two PCs are connected via gigabit ethernet networking. This slide here shows some characteristics of gigabit ethernet. So you see minimum frame sizes and given 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 is: How long does it actually 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 will arrive at a much higher rate.

6.2 Latency Example (2/2)

- Numbers from [Lar+09]
  - Processing of single frame takes **total** of 7.7 µs
  - Latency breakdown according to different sources
    - Hardware: ≈ 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.

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 is bound to overload or even crash the machine if the network load is high.

Let's say every day you're supposed to write an exam but you need 7.7 days to prepare for each exam. Apparently, you should not expect to finish preparations for any exam.

6.3 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

The type of crash that I 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. That means the computations that still take place are mostly related to overhead processing.

Now, if the Livelock is caused by interrupts, then we're talking about interrupt Livelocks. The idea is here that interrupts arrive so fast that they cannot be processed any longer. So, think about individual packets arriving one per microsecond while each individual packet needs 7.7 microseconds processing time. So, the bulk of CPU time will then be wasted with interrupt overhead processing and that means also that other tasks are starved. And as interrupts could be run with high priority, your ordinary tasks will not receive compute time anymore. In addition, there will be lots of context switching, switching between individual processes, cache pollution and that also adds to the overhead. And ultimately nothing useful happens anymore.

A way around this situation is to use a hybrid of polling and interrupts, for example the NAPI (new API) that is discussed on the subsequent slide.

6.3.1 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
- Starvation revisiting in later presentations on scheduling and challenges for mutual exclusion

6.4 NAPI

- Linux "New API" for networking (2001), see [SOK01]
- Hybrid scheme
  - Use interrupts under low load
    - Utilize CPUs better
      - Avoid polling for devices without data
  - Switch to polling under high load
    - Avoid I/O overhead
      - Data will be available anyways

NAPI or new API is the name of a technique of the Linux kernel which was introduced in 2001. So actually you see that it's not a good idea to name something "new- something". Anyways it is a hybrid scheme that combines polling and interrupt processing to make most use of the available resources. The drawback of polling was that we are wasting 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. So if only few packets arrive, then each of these packets will trigger an interrupt and that avoids polling in situations where no data is available. However, if lots of network packets arrive, then the system switches to polling to avoid the I/O overhead that we just discussed. And the assumption there is, that under high load, new data will be available anyways, so whenever the operating system looks for new data, it will be there.

7 Outlook

7.1 When to Poll?

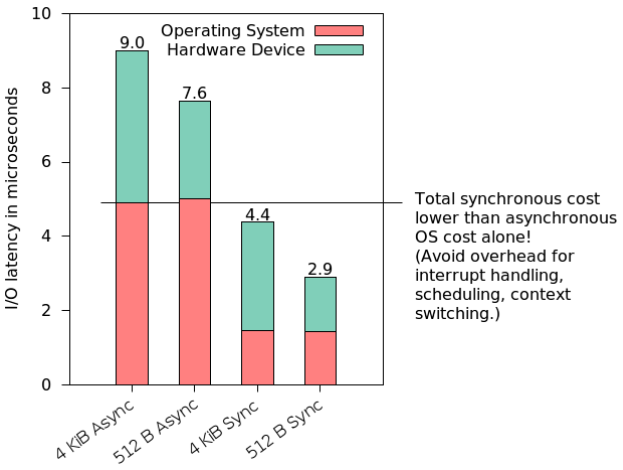

Figure 6: Measurements for DRAM-based storage prototype (data from [YMH12])

This figure shows data from research paper [YMH12] 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 I/O devices such as hard disks, polling is preferable for low-latency I/O devices.

7.2 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/SCMs are fast, see [Nan+16]
  - 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

7.3 Call for Research

- [Bar+17]: 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 [Jon+18])
  - Millisecond latency (disk I/O) is hidden by OS
    - Multitasking
  - What about microseconds of new generation of fast I/O devices?
    - E.g., Gbps networking, flash memory
    - Paper describes datacenter challenges experienced at Google

8 Conclusions

8.1 Summary

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

Bibliography

[Bar+17] Luiz Barroso et al. "Attack of the Killer Microseconds". In: *CACM* 60.4 (2017), pp. 48–54. URL: https://dl.acm.org/citation.cfm?id=3015146.

[Hai19] Max Hailperin. *Operating Systems and Middleware – Supporting Controlled Interaction*. revised edition 1.3.1, 2019. URL: https://gustavus.edu/mcs/max/os-book/.

[Jon+18] Christopher Jonathan et al. "Exploiting Coroutines to Attack the "Killer Nanoseconds"". In: *Proc. VLDB Endow.* 11.11 (2018), pp. 1702–1714. ISSN: 2150-8097. DOI: 10.14778/3236187.3236216. URL: https://doi.org/10.14778/3236187.3236216.

[Lar+09] Steen Larsen et al. "Architectural Breakdown of End-to-End Latency in a TCP/IP Network". In: *Int. J. Parallel Prog.* 37.6 (2009), pp. 556–571. URL: http://link.springer.com/article/10.1007/s10766-009-0109-6.

[Nan+16] Mihir Nanavati et al. "Non-volatile Storage – Implications of the Datacenter's Shifting Center". In: *ACM Queue* 13.9 (2016). URL: https://queue.acm.org/detail.cfm?id=2874238.

[SOK01] Jamal Hadi Salim, Robert Olsson, and Alexey Kuznetsov. "Beyond Softnet". In: *Proceedings of the 5th Annual Linux Showcase & Conference*. Oakland, California: USENIX Association, 2001. URL: https://www.usenix.org/publications/library/proceedings/als01/full_papers/jamal/jamal.pdf.

[Sta14] William Stallings. *Operating Systems – Internals and Design Principles*. 8th ed. Pearson, 2014.

[TB15] Andrew S. Tanenbaum and Herbert Bos. *Modern Operating Systems*. 4th ed. Pearson, 2015.

[YMH12] Jisoo Yang, Dave B. Minturn, and Frank Hady. "When Poll is Better than Interrupt". In: *FAST 2012*. 2012. URL: https://www.usenix.org/conference/fast12/when-poll-better-interrupt.

License Information