Module 1, Lesson 2 "Architectural Support"

Hello, and welcome to this lesson on architectural support for operating systems.

In this lesson we will explore what the operating system needs from the underlying architecture in order to do its thing.

If you have taken a computer organization class or an introductory architecture class before, none of this material will be new to you. We will present it from the perspective of an operating system designer.

Here we will pick and focus on four general areas, namely:

- Support for asynchronous events, that is, handling of exceptions and interrupts.
- Hardware protection, with privilege levels, privileged instructions, I/O protections and so on.
- Support for address spaces and memory protection and finally,
- Timers.

Note that there are other areas where the CPU has facilities that can be used to support the operating system. For example, atomic operations in support of synchronization, direct memory access, or the Task State Segment to support thread management in the x86. Some of these we will visit later in the course.

1. Support for Asynchronous Events

Let's talk first about support for asynchronous events.

- We start from the observation that operating systems have to handle many asynchronous events, such as events from devices, for example when data becomes ready, or when an error is detected.
- Other asynchronous events can be when the user provides some sort of input, either by pressing a key, or by moving or clicking a mouse, or by moving or tapping the smartphone.
- Finally, timers can fire, which is treated as asynchronous events as well.
- How should the system overall, but in particular the operating system, handle such asynchronous events?

As we will explore more in detail later, the operating system can do this in one of two ways.

- It can periodically check for the event and handle it if the event occurred since the last time it was checked. Constantly checking for events is clearly not an efficient method, and so it is used only in very special situations, typically in real-time systems.
- A more common way is to handle events in a so-called interrupt-fashion, where the occurrence of an event triggers a so-called interrupt, and the CPU "interrupts" what it is doing to call an interrupt handler, which in turn handles the event.
- Most operating systems today (except for some real-time systems) handle asynchronous events using interrupts.

Most modern OS's are Interrupt-Driven

We said that most operating systems are interrupt driven, so what is the benefit? From a system design perspective interrupts have two advantages.

First, interrupts allow for the easy separation between event handling on one hand and the main computation activities on the other. One simply installs an interrupt handler, which gets called whenever a particular event occurs. This happens largely invisibly to the main computation. The system designer and programmer

can now partition the problem and thus simplify the design process by dealing with the main computation separately from the event handling.

The second advantage of interrupt-driven handling of events is multiprogramming, where the CPU continues with the main computation while a request, maybe a read operation, is being handled by an I/O device.

Let's look how this works using a simple example, with a CPU and two I/O devices, say a disk and a keyboard.

The CPU is busy doing some computation, while the two devices are idle.

At some point the CPU issues a read operation on the disk, and instead of waiting for the disk to come back, it takes advantage of multiprogramming by executing something else, maybe another thread.

The disk now is busy serving the request. If this is a spinning hard drive, this may take several milliseconds.

To make things a bit more interesting, before the disk returns with the data, the user presses a key on the keyboard.

The key press triggers an interrupt, and the CPU calls the interrupt handler that handles keyboard press events. The handler copies the data from the keyboard into memory maybe, and then returns. The CPU resumes whatever it was doing.

At some time later the disk is finally done with the request, and the disk controller triggers an interrupt.

Again, the CPU drops what it was doing and calls the interrupt handler to take care of the data ready event. This handler may copy the data from the disk controller to the main memory of the computer, for the CPU to be able to access the data.

Once the handler is returns, whoever was reading, maybe a thread, now is ready to continue execution by returning from the read function.

We notice that at no time was the CPU idle. In general, as long a there is work for the CPU, the multiprogramming enabled by interrupt-driven handling of events allows for high utilization of the computing resources.

Interrupts

From previous computer architecture or computer organization classes we recall, at least at high level, what happens when an interrupt occurs. Let's summarize it here.

When an interrupt or an exception occurs, the CPU stops execution on the current thread and save the state of at least some of the registers. It then typically changes into supervisor mode or privileged mode. This allows the CPU to execute so-called privileged instructions, which cannot be executed during normal operation. And then the CPU branches to a predefined location in memory and continues execution there.

The question is: How does the CPU know where to branch to?

Let's work our way through the handling of an interrupt. Let's say the CPU is just executing some instruction, say, to point the stack to some new location.

An interrupt occurs, maybe triggered by a device. Let's say that the number of the interrupt is X. The CPU stops, saves the state, and changes into supervisor mode. Now it must determine the predefined location where to branch to, that is, the address of the interrupt service routine.

For this, the CPU looks up the address of this routine in a specific area of the memory, which we call the interrupt vector area. This area is a vector of addresses of interrupt service routines, one for each different type of interrupt.

The CPU indexes into this vector using the interrupt number, in this case X, and looks up the address of the routine, in this case Hex ABAB.

Hex ABAB is the address of the interrupt service routine for interrupt number X and the CPU therefore branches to location Hex ABAB to continue execution there.

Interrupt service routines are a bit special in that they to not return using a return-from-subroutine instruction, but rather a return-from-interrupt. As part of this instruction, the state of the CPU is restored, and the CPU continues execution from where it left off. This is how an interrupt or an exception gets handled.

Let's re-state a few observations:

- First, the address of the interrupt service routine responsible for handling a particular interrupt is stored in the interrupt vector area.
- Second the return-from-interrupt instruction automatically restores the state of the CPU and loads the program counter to continue execution where the interrupt happened. In practice, an interrupt-handling framework needs to save and restore additional registers in order for to this all to work, but the hardware takes care of the basics.

Finally, interrupts or exceptions are typically triggered by asynchronous events, such as device state changes or timers, or various unexpected errors such as division-by-zero or invalid memory references. Interrupts can also be triggered synchronously, for example for the invocation of system calls. We will look at this in one of the next lessons.

2. Hardware protection

The underlying hardware architecture also has support for hardware protection.

We remember that in very early systems, the user owned the machine, and there was no form of operating system running at all.

Other than for convenience, there was no perceived need to protect one software from another.

With the advent of operating systems, multiple programs had to co-habit on the same computer, at the very least the operating system and one or more user programs.

As a result, the various programs had to be protected from each other so that they could not adversely affect each other.

The question of course is, How could programs affect each other? We will look at four ways.

First, each CPU has a set of instructions that should not be indiscriminately invoked. A good example is the HALT instruction. If a user program calls the HALT instruction, the entire machine comes to a sudden stop, including all the other user programs and the operating system. Pedestrian users should therefore not be allowed to call such instructions.

Second, user programs could access or modify data on devices, such as storage devices.

Similarly, a user program could access or overwrite data or executable code in the memory either of the operating system or another user program, with predictable consequences.

Finally, and maybe a little more subtly, a user program could end up in an infinite loop and so not be able to give up the CPU for other user programs or the operating system to make progress. The machine becomes largely useless until it is shut off.

We note that such problems could be the result of bugs in the program, or they could have been planted with malicious intent.

So, how can the hardware help protect the system in these four situations? In the case of illegal instructions, most CPUs operate in two or more modes.

For the simple case of two modes, we distinguish between user mode and supervisor or kernel mode. The user mode allows execution of only so-called non-privileged instructions such as load/store, or arithmetic operations, and others.

Most system-related instructions, such as our HALT instruction, or masking and unmasking of interrupts and others, are so-called privileged, and can only be executed when the CPU is in the supervisor or kernel mode that we mentioned earlier.

How do we prevent unauthorized programs to read/write data to or from devices?

Most CPUs support I/O protection they do this by having all I/O operations be privileged. As we just learned, privileged instructions can only be executed when the CPU is in supervisor mode. In case of the x86 processor family, data is written or read to or from I/O devices using some variation of the out or in instructions, which are privileged.

Next is memory protection.

What are the ways for the hardware to protect, for example, the interrupt vector, or the code for the interrupt service routines, or memory segments of other user programs?

This is typically done by the memory management unit of the CPU by defining some form of legal address range that the CPU can refer to on behalf of the user program.

How does this work? Assume that we have a CPU on one side, and the memory on the other.

And there is a legal address range, which is delimited by a base address and by a limit, which is the size of the range. Valid addresses range from base address to base + limit. All other addresses are invalid, and trigger an "invalid memory reference" exception.

When the CPU issues an address, the memory management unit (MMU) checks whether the address is larger equal to the base address, if it is, it checks whether it is smaller than the base + limit, and if so, the memory reference is allowed to occur.

If not, an "invalid memory reference" exception occurs. The exception handling routine associate with this exception is invoked, which allows the operating system to take over.

Finally, most CPUs have extensive support for timers, which can be used to force control of the CPU away from user programs.

A timer can be set to fire at some particular time instance in the future. When the timer fires, a "timer interrupt" is triggered, and the timer interrupt service routine is invoked, which gives control back to the operating system.

3. Support for Address spaces

After interrupts and hardware protection, support for **address spaces** is another area where the operating system needs help from the underlying architecture.

To visualize the problem, let's go back to the scenario that we had in the introductory lecture, where we have multiple user programs in the system, here called Job 1 to Job 4, in addition to the operating system.

Let's highlight two of these user programs, and call them "My Program" and "Your Program." These two programs are stored each in a different location in memory.

If we look at how the users of the two programs see the memory, we notice that the user of "My Program" sees a memory region that starts at zero and goes to, say Hex FFFFFF. The user of "Your Program" sees memory in the same way, that is, ranging from zero to Hex FFFFFF. We call the way the program sees its memory the "address space" of the program.

How can we fit two address spaces with the identical address ranges (that is, from zero to Hex FFFFFF) to different regions of memory in a way that is transparent to the users and to the user programs?

This is where the memory management unit of the CPU comes in handy again, except that we use it slightly differently from when we implemented memory protection.

Let's take again the CPU on the left and the memory on the right, with the two regions of memory for my program and your program. Let's assume that my program is running, and the base and the limit registers are set up to translate CPU memory reference to my program's address region.

Whenever the CPU issues an address, which is between zero and Hex FFFFFF, the memory management unit adds the base address to it.

The memory management unit then checks whether the memory reference is valid and if so allows it to happen.

In order to support multiple programs in memory, the memory management unit has a table of base addresses and limits, one for each program. At any given time, one entry from this table is used, in this case the one for my program.

When the operating system switches to another program, in this case your program, the memory management unit loads a new memory mapping by using your programs entries for base and limit. In this way the CPU's memory references are translated to your program's memory.

4. Timers

Finally, the fourth area of CPU support to the operating system that we briefly explore are timers.

We remember from a few minutes ago that timers can be set, and an interrupt gives control to the OS when the timer fires.

This can be used for prevent user programs from hogging the system, but there are many other applications of timers as well.

For example, timers are used to implement round robin schedulers, which are really handy when implementing time sharing.

Timers are also used to implement time-of-day clocks.

Summary: What does the Operating System need?

With this we have come to the end of this lesson. To summarize, we explored together a few points where the operating system needs specialized support from the underlying architecture. We identified four areas where this is the case.

The first was about support for handling asynchronous events. We looked at how interrupts greatly simplify the handling of such events, and how the operating system designer can take advantage of interrupt handling to increase the CPU utilization through multiprogramming.

We then looked at what is needed to protect the operation of the computing system from buggy or malicious software. We listed dual-mode operation, where some instructions can only be executed when the CPU operates in privileged mode. Similarly, I/O operations must be privileged as well to avoid user programs accessing devices directly.

Mechanisms must be in place to protect critical memory sections, such as operating system tables, or to protect the memory of one process from another processes execution. Timers are used to take over control when either a user program or a device driver hogs the CPU and so threatens to freeze the system.

Next we discussed the problem of multiplexing the memory across processes, and creating the illusion that each process owns the memory. This is achieved by the Memory Management Unit of the CPU, which supports per-process memory relocation. We will be discussing the details of this when we explore memory management in more detail.

Finally, hardware timers are critical in supporting the smooth operation of the system. As mentioned earlier, they are used as time-out indicators for the system to take over. But they can also be used to support time-sharing, to implement a wall-clock, and others.

We did leave out a few areas. For example, we did not talk about instructions that implement atomic operations. We will explore these when we talk about synchronization later in the course. We did not talk about direct memory access (DMA).

We also did not talk about more complex support, such the Task-State-Segments, which support thread switching on the x86. Task-State Segments are a good example of support that was well-intended but rarely used. When we will explore thread implementations, we will learn that most of today's operating systems implement thread switching using very basic instructions.

In the next lessons we will learn about the structure of operating systems and about operating system interfaces, such as the system call interface for the user and the device driver interface for devices.

Thank you for watching!