Low-Level Thread Dispatching on the x86

Hello, and welcome to this lesson on Low-Level Thread Dispatching on the x86.

Dispatching is the operation of transferring the execution from one thread to another.

We say that the currently running thread dispatches another thread when the current thread suspends itself and starts the other thread in its place. The dispatcher is typically called by the scheduler to transfer execution from one thread to another in the system. We can think of the scheduler as making the decision about which thread to go next, and of the dispatcher doing the legwork of transferring the execution.

The steps that we will use to switch from one thread to another are very similar to those used in exception handling. We will briefly re-visit the implementation of the exception handling on the x86.

And use the exception dispatching framework to implement a thread dispatcher.

We then have a brief look at a possible implementation of the thread control block in a C++ class of type THREAD.

We will then walk through the details of the context switch from the current thread to the another.

Before we can dispatch to a thread, we need to set up its stack correctly. We will also learn how to do that.

We will then learn about the execution lifecycle of a thread from the moment it gets created until its thread function terminates, and the thread terminates as well.

By the end of this lesson you understand how thread dispatching works for many CPU architectures. You will be able to read, understand, and modify in an informed fashion the thread dispatching code in a simple operating system.

Some of you may know that the x86 has special support for task management, using the Task State Segment, in short TSS. Most operating system designers ignore this support. We will do the same.

Let's begin!

Low-Level Thread Dispatching

Let's assume that the low-level context switch function is provided by the function "dispatch to", which is called by the current thread and which transfer the control over to the argument thread.

In order to successfully dispatch to the new thread we need to do three things.

First, we need to save the state of the current thread. This state will be needed again when the current thread resumes execution at a later time. At this low level we are concerned exclusively with the processor state, primarily its registers. We assume that higher levels of the system software take care of other aspects, such as address spaces and other things.

Second we need to load the state of the new thread into the registers of the CPU.

Finally we continue with the execution of the new thread

This of course requires us to solve three problems:

First, how do we save the state of the current threat.

Second how do we load the state of the new thread

And third how do we continue the execution of the new thread?

This looks like a difficult set of problems. But we have seen something similar before.

Where?

Right, we have encountered a similar problem when we looked at how exceptions and interrupts are handled.

Therefore we will be implementing thread dispatching somewhat similarly to the way of how we implemented the handling of exceptions.

RECAP: Exception Handling

Let's revisit the implementation of exception handling in our system.

Let us assume that an exception occurs say exception number zero. The hardware stores a number of registers on the stack and calls the appropriate interrupt service routine in this case ISR 0 to ISR 31. At this point we have the instruction pointer on the stack as well as the code segment register and the contents of the eflags register.

That exception specific routine stores the interrupt number and if needed the error code on the stack.

Here we see same process for a double fault exception

Once the interrupt number and the error code are stored on the stack a common stub function is called. This function stores the remaining registers on the stack and calls the high level exception handler.

At the moment when the high-level exception handler is called, the content of the top of the stack can be represented as the following struct, which give access to all CPU registers.

The high level exception handler is called from here, and once the exception handler returns the previously saved register state is loaded from the stack into the registers and we return from the exception or from the interrupt using the IRET instruction

RECAP: Saving and Loading State

Let's highlight how the processor state is saved and how it is loaded again when an exception occurs and is being handled.

When the exception occurs, the program counter, the code segment, and the eflags registers are saved on the stack in hardware. In some cases the error code is saved on the stack by the hardware as well. We then push the error_code – if missing – and the interrupt number on the stack and then save the remaining registers in the common stub.

This is where we save the state of the processor.

After the exception returns, we pop the previously stored state from the stack into the registers.

This is where we load the state from the stack back into the registers again. The program counter, code segment register, and the execution flags register is load at the very end by the IRET instruction, and the processor continues executing from where it left off as if nothing had happened.

Saving and Restoring the State

Let's further simplify this picture. Here we see a thread executing. (Time goes from top to bottom.)

At some point in time an exception occurs, and a low-level exception dispatcher is called. This dispatcher saves the state of the processor (in the red box).

In the meantime the thread is suspended and is waiting for the exception handler to return in order to resume execution.

The low level dispatcher in turn makes a function call to the high-level exception handler.

The latter does its thing and returns to the low-level exception handler.

This one now loads the previously stored state back into the processor registers and returns from the exception using the IRET instruction, which stands for RETURN FROM INTERRUPT.

The thread now resume execution as if nothing had happened.

The question now is: "What does this have to do with thread dispatching?" This becomes clearer when we – somewhat counter-intuitively – get rid of the exception.

Since we don't have an exception, we don't need an exception dispatcher either. The state saving (the red block) is now invoked by the thread as a regular function, and not as an exception.

Since we don't have an exception, we can get rid of the exception handler. This one would have nothing to do anyway.

Things become a bit more compact now, and it becomes clear that we are first saving the processor state (red box) and then load the same state again (blue box). This looks a bit silly.

How about we load another thread's state instead?

If we do that, we continue the execution with another thread. We have successfully transferred control from the first thread to the second.

Dispatching, Example

Let's draw this – again – slightly differently.

Let's assume that we have two treads, Thread A and Thread B. Thread A is running on the CPU. Thread B is suspended.

At some point, Thread A make a call to **Thread::dispatch_to** to pass the CPU to Thread B.

Inside the function dispatch_to, Thread A saves the processor state in its stack. This is done in code marked by the red box. In the code in the blue box the Thread locates the state for Thread B and loads it into the CPU registers.

When the IRET instruction is called, the control is passed to Thread B, which now is executing.

The first thing Thread B does is to return from the previously-called function **dispatch_to**. We make a simplifying assumption here that both threads have been running before. We will address how threads are started and terminated later.

At some point, Thread B in this example switches back control to Thread A by calling the function **dispatch_to** with A as parameter. This time, Thread B's state is saved on B's stack and Thread A's state is loaded onto the CPU from A's stack.

When we – so-to-say – return from the interrupt using the IRET instruction, Thread A continues from where it had left off earlier, and Thread B is now suspended.

Now Thread A finally returns from the previously called function **dispatch_to**.

Here we mark in green the fact that Thread B loads the state onto the CPU that Thread A had saved earlier. Of course the state that Thread A loads onto the CPU when dispatching over to Thread B earlier was saved by Thread B sometimes earlier during its execution. This figure does not show how this happened.

RECAP: Structure of a Thread

Before we continue with the detailed implementation, the implementation of the Thread Control Block, or TCB. The TCB contains all the management information about the thread, such as thread id, priority, maybe some bookkeeping information.

It also contains a pointer to the stack of the thread.

A second pointer indicates the current top of the stack.

We implement the TCB as an object of class Thread. This object contains all the information that we mentioned earlier, such as **thread_id** and so on.

An important implementation detail is that we store the top-of-the stack pointer at the very beginning of the object. In this way the address of the stack pointer is the same as the address of the thread object, and we can use the two interchangeably. We will se later how this simplifies the assembly code.

Dispatching Step-by-Step: 0. Start

Let look now in detail at the four steps necessary to switch control from one thread to another.

Let's assume that the current thread just started the dispatching procedure by calling the function **dispatch_to**. More precisely, we assume that we are inside the assembly function **threads_low_switch_to**.

On the stack, we now have the return address from where the current thread called the function **dispatch_to**, and the new thread as argument. The code to push these two items onto the stack is generated by the compiler.

Dispatching Step-by-Step: 1. Fix Stack

This low-level function must perform four steps to switch to the new thread.

First, need to fix the content of the stack. Currently, the stack looks like it does during a function call. We do not want to return from this function, however. Rather, we want to return from an exception.

The stack therefore has to look a bit differently. Rather than having the return address only on the stack, we need the return address and the associated code segment. We also need the eflags. The thread argument stays.

Dispatching Step-by-Step: 2. Save State

Now we can proceed to save the rest of the state.

This code save the error code and the interrupt number. These values are not needed. We just keep them for compatibilities sake with the exception handler.

This code now save the general purpose registers.

All eight of them with a single instruction.

The segment registers need to be saved explicitly.

Same as in our exception handler code.

Now we store the stack pointer in the TCB of the current thread. We have a global variable "current thread", which points to the TCB of the current thread, and which we can use to store the stack pointer.

The code says to store the stack pointer in the location pointed to by the "current_thread" TCB pointer.

We previously made sure that the top-of-the-stack pointer in the TCB is located such that it has the same address as the TCB itself. This is why this simple code works.

Dispatching Step-by-Step: 3. Locate State

Now that we have saved the state of the current thread, we proceed to load the state of the new thread.

First we have to find where the state of the new thread is located. This is not difficult once we recall that the state of that thread was save very much in the same way as the state of the current thread. Moreover, the pointer to the TCB (and therefore the pointer to the location of the top-of-the-stack pointer) of the new thread was passed as argument. It is therefore easy to find on the stack.

Specifically, this argument is 68 Bytes away from the current top-of-the stack.

We update the "current thread" TCB to point to the new thread.

As pointed out multiple times before, the first entry in the TCB is the top-of-the stack pointer.

We therefore dereference the TCB pointer and load the value into the stack register.

The stack pointer now points to the top of the stack of the new thread.

This is the stack that we will be using to load the state into the registers.

We don't need the old stack until we switch back the old thread.

Dispatching Step-by-Step: 4. Load State

Now we are ready to load the state of the new thread from its stack into the registers.

We start with the segment registers.

We continue with the general purpose registers.

This is done with a single instruction.

We skip the interrupt number and the error code.

As pointed out earlier, maintaining these two numbers is not necessary. We keep it to keep the code similar to the exception handling code.

Now it is time to continue the execution where the new thread left off.

All the information is stored in the remaining part of the stack: The return address, code segment, and eflags registers were stored on the stack when the saved the state of the new thread onto the stack when that thread called the dispatching function. We remember from when we described how the stack gets fixed at the beginning of the dispatching that we store the return address from the call to the function **dispatch_to**.

By calling the instruction IRET, we continue the execution at the return address, that is, where the function **dispatch_to** of the new thread has been called from.

By continuing execution at the return address, we rather unceremoniously exit the function **dispatch_to**.

The calling function just sees that we returned from **dispatch_to**, and it clears the argument off the stack. (The code to do this is generated by the compiler.) Now there is nothing from the context switch left on the stack, and the calling function of the new thread continues executing as if nothing had happened.

Creating a Thread

In the description above we have assumed that all threads have had their state put on the stack by an earlier invocation of the function **dispatch_to**.

Now we need to address how to handle threads that have just been created; that is, they have not had a chance to call **dispatch_to**.

No worries, instead of waiting for the function **dispatch_to** to save the state on the stack, we save an initial state on the thread's stack when we create the thread.

This is done as part of the function **setup_context**, which is called by the thread constructor. This function carefully pushes an initial state onto the stack of the new thread. Note that we need to carefully distinguish between – on one hand - the execution stack, which is pointed to by the stack pointer register of the CPU, and on the other hand the thread stack of the newly created thread. This stack is not used by the CPU until the thread is running. The top-of-the thread stack is pointed to by the top-of-the stack pointer in the TCB of the newly created thread. Let's see how we set up the state for the new thread.

First, we push onto the thread stack a pointer to the shutdown function, which is called when the thread function returns and the threads needs to be shut down.

Just to be clear, we push onto the thread stack by explicitly manipulating the top-of-the stack pointer in the TCB of the thread. We store a 32-bit value by decrementing the stack pointer by 4 byte and then storing the value in the location pointed to by the top-of-the stack pointer.

We now push the pointer to the thread function onto the thread stack.

Again, keep in mind that this is not the execution stack of the CPU until the new thread actually gets to execute.

Now we start creating the part of the thread stack that is expected by the **dispatch_to** function.

During the **dispatch_to** we stored the return address. Now we store the start address for the thread. This start address is not the address of the thread function, but rather the address of a static function in the Thread class. The role of this function is to do anything that is needed to start the thread function correctly. For example, one may need to enable interrupts because we are storing a zero value for the EFLAGS register, which means that the thread will start with interrupts disabled. If we want to not bother the user to enabled interrupts inside the thread function, we can add this to the thread start function.

We continue by storing dummy values for error code and interrupt number.

As pointed out many times before, we will not be using these values.

We continue by pushing the general-purpose registers.

All eight of them.

And we wrap it up with the four segment registers.

The stack now looks like one that has been created by the first part of the **dispatch_to** function.

Execution Lifecycle of a Thread

How does this all work together? Let's assume that we just created a new thread, with the stack as depicted here.

Somebody, maybe the scheduler, dispatches this thread for the first time.

As part of this, the thread state that we so carefully constructed during thread creation, is loaded from the stack into the registers.

The last instruction in dispatch_to is the return-from-interrupt instruction IRET, which interprets the top three values on the stack as new program counter, code segment, and eflags register. The CPU will therefore continue execution with the function **thread_start**. This function does all the generic thread start management.

The instruction IRET pops the program counter, the segment register and the EFLAGS register off the stack and continues with the function **thread_start**.

When the function **thread_start** returns, the CPU will continue executing at whatever address it finds on the stack. (The code to do so has been generated by the compiler.) On the top of the stack we have the address of the thread function.

So the CPU starts executing the thread function. During the execution of the thread function, the thread may yield and require the CPU multiple times through the **dispatch_to** function.

Once the thread function returns, the the CPU will again continue executing at whatever address it finds on the stack. (The code to do so – again - has been generated by the compiler.) On the top of the stack we have the address of the thread shutdown function. This function does all the generic cleanup needed to terminate a thread.

The CPU starts executing the thread shutdown function.

When the thread shutdown starts executing, the thread stack is empty.

Low-Level Dispatching on the x86: Summary

With this we have come to the conclusion of this lesson. Let's summarize what we learned.

At the beginning we drew similarities between the implementation of exception handlers and that of a thread dispatcher. The use of the "return from interrupt" instruction to switch between threads may not be the best way to implement threads on a x86 system, but it is very general and quite portable to other architectures, such as the MIPS or the ARM.

We looked at how thread control blocks – TCBs – can be implemented as objects of a C++ class, say **Thread**.

We then looked in detail at how a low level thread dispatching function goes about to store the processor state of the current thread, then locate the state of the new thread, and finally load the state of the new thread and continue executing as the new thread from where it left off as if nothing had happened.

In order for a thread to start correctly, it initial state must be loaded into the registers by the low-level thread dispatch function. This initial state must be carefully crafted during thread creation. We looked at how this is done.

Finally, we looked in detail how the thread executes from its creation until its termination. We looked at how we leverage the function call code generated by the compiler to force the thread to invoke first the generic thread start function, then the thread function, and finally the generic thread shutdown function.

I sincerely hope that you enjoyed this lesson on Low-Level Dispatching on the x86. We went through a lot of low-level code, but you certainly have gained a thorough understanding of how thread management works.

Thank you for watching!