Boost.Context s390x Implemenatation - Doubts - Need help

3 messages

Naveen Naidu <naveennaidu479@gmail.com> 10 May 2019 at 17:01
To: Oliver Kowalke <oliver.kowalke@gmail.com>

Hello Oliver,

I'm sorry for disturbing you from your busy schedule. But I wouldn't have messaged you, had I really did not feel the need to. I have been going through the fcontext and I have been reading and re reading the ABI's and have many online resources, yet I was unable to find the answers, and that is the reason why I decided to mail you my list of questions.

I will be writing my understanding of how *make_fcontext()* and *jump_fcontext()* work. I will be using the example of x86_64 assembly files to explain my understanding. The SYSV ELF ABI for the architecture is present here

Boost.context library provides the users with fibers which are used to control the flow of program execution. These fibers, in their bare implementation uses *fcontext (an alternative to ucontext)* to switch between contexts using the functions *make_fcontext()* and *jump_fcontext().* These two functions are low level assembly files which are written by hand for different architectures.

Assumption: *There are two different kind of stacks maintained. One is the* ***execution_stack****, which is the usual stack maintained by operating systems to manage between different functions. Another is a* ***context_stack****. The context_stack is used to store the registers and context_functions of each context we make via make_fcontext(). These stacks are stored independent of execution_stack. Each stack frame in context_stack belongs to one of the fcontext_t we make using make_fcontext(). And whenever we switch between context, it is to these context-stack-frames that we save the registers and switch to the new context-stack-frame.* ***Please let me know, if this is a wrong assumption****.*

----

1. make_fcontext( )

The aim of the make_fcontext() is to prepare the stack for the first invocation of jump_fcontext(). Let's take the following code as an example to explain my understanding. Before we go on, there is one assumption that I have made while understanding it.

```
// context-function
void f(intptr);

// creates a new stack
std::size_t size = 8192;
void* sp(std::malloc(size));

// context fc uses f() as context function
// fcontext_t is placed on top of context stack
// a pointer to fcontext_t is returned
```

```
fcontext_t fc(make_fcontext(sp,size,f))
```

The following are the steps that occur in *make_fcontext()*. The assembly file for *make_x86_64_sysv_elf_gas* is here . We first create the context function and creates
a stack using malloc. The context_function and stack_pointer to the newly created stack is passed to the *make_fcontext()*. Note that when *make_fcontext()* is called,
the assembly code are initiated. The parameters passed to the *make_fcontext()* are stored in the registers.

1. The register ***%rdi*** holds the first argument of *make_fcontext*() which is the starting address of the context stack which had been calculated using malloc() . The first thing we need to make sure is that the starting address is aligned with the ELF requirements. Since for x86, the address in quad word aligned. We shift the starting address of the context_stack to the lower 16 byte boundary and hence making it quad word aligned.

2. Once the address is aligned. We then create some space for the context data on the context-stack.

3. The third argument which is the address of the context-functions and is present in the register *%rdx* is saved onto the context-stack. And after that we store all the inital registers that we might need to use when there is a first time jump to this context.

4. We then create two functions/labels. *trampoline* and *finish.* Trampoline is entered when jump_context is called for the first time. This function pushes the current Base pointer onto the stack and invokes the context-function. I'm a little confused about this. *Why should this trampoline function be called only once*, is it because - The context-function of a particular context is only called once and we are not supposed to the call the context-function of the context from within the same context?

5. Another function *finish* is executed when the context-function completes it's execution. This basically sets the exit code to zero and exits from the function and clears the stack frame.

6. It is important to note that the absolute address of the *trampoline* and *finish* are stored at the appropriate locations on the context-stack. These address are used to invoke the functions.

7. It then returns the starting address of this context stack.

In short, make_fcontext() is responsible for creating the inital context-stack and initializing it with data that is necessary for the context switching.

----

2. jump_fcontext()

This file is used to jump between fibers/fiber_contexts. The following are the steps that occur:

1. Save the registers of the current ongoing functions.
2. Move to the context_stack frame of the context we want to switch to.
3. Restore the values of register from the context_stack.
4. Jump to the context_function.

This is the function which I'm unable to understand clearly. My doubts are as follows. Please see the jump_fcontext() file from here for reference.

1. `leaq -0x38(%rsp), %rsp /* prepare stack */` (*line 33*)

   What does the RSP actually point to here. Does RSP point to the lowest most address of the current ongoing context. If so, does it mean that whenever we switch from one context to another we will always reduce the stack pointer making and then store the values of the calling function in this stack before we switch to next context in the context-stack.

   Suppose we shift from *main( )* function to a context-fuction *fc1( )* , so does the RSP point to the address where the *jump_context( )* is called in the *main( )* function? I don't think this is the right understanding of mine, because the above line is executed after we shift into the *jump_fcontext( ).* If that's the case, the RSP should point to the lowest most address of the context of the current function. But how does RSP directly shift to the lowest most address of the context-stack as soon as we call jump_fcontext and also how does it know, which context-stack-frame it should shift to.

2. `movq %rdi, %rsp` (*line 51*)

   Doesn't ***%rdi*** point to the first argument of jump_fcontext(). And the first argument of jump_context is the address of the current context. Then we are using that addres to restore the values into the register. If that is the case, how is that that we are restoring the registers from fromt calling context. But shouldn't we be using the second argument which points to the context to which we want to jump.

   And according to the ABI ***%rsi*** points to the second argument, which points to the address of the context we have to switch.

3. `leaq 0x40(%rsp), %rsp /* prepare stack */` (*line 67*)

   Why are we preparing another stack here? Is this stack for the context-function.

4.
   ```
    /* return transfer_t from jump */
   /* RAX == fctx, RDX == data */
   movq %rsi, %rdx
   /* pass transfer_t as first arg in context function */
   /* RDI == fctx, RSI == data */
   movq %rax, %rdi
   ```

   Can you please explain what is actually hapenning here. Or maybe just point me to few references reading which I can decipher what this actually means.

5. I also had the doubt in the way *jump_fcontext* is called. The call of *jump_fcontext* is like below:

   ```
   jump_fcontext( & fc1, fc2, 0)
   ```

   Why are we passing the address which stores the variable fc1 but directly pass the reference to fc2. Shouldn't we be passing the address of fc2 i.e (***&fc2).***

6. Also, when a context-function starts it's execution does it use the context-stack-frame to save it's local variables and stuff or does it use the execution stack.

Thank you very much for your patience in reading such a long email. And also, I am really sorry for typing up such a long list of questions. But I have been at it for quite a few days now. Trying to read as many implementations as I can. I was able to understand what *make_fcontext()* is. But I was not able to understand *jump_fcontext.* I know the answer is right in front of me, but if these points were cleared up, I can really get all my abstract thougts to a clear understanding.

Thanks again and sorry for all the inconvenience caused.

Oliver Kowalke <oliver.kowalke@gmail.com> 10 May 2019 at 18:47
To: Naveen Naidu <naveennaidu479@gmail.com>

Am Fr., 10. Mai 2019 um 13:31 Uhr schrieb Naveen Naidu <naveennaidu479@gmail.com>:

> ***Assumption:*** *There are two different kind of stacks maintained. One is the* ***execution_stack****, which is the usual stack maintained by operating systems to manage between different functions. Another is a* ***context_stack****. The context_stack is used to store the registers and context_functions of each context we make via make_fcontext(). These stacks are stored independent of execution_stack. Each stack frame in context_stack belongs to one of the fcontext_t we make using make_fcontext(). And whenever we switch between context, it is to these context-stack-frames that we save the registers and switch to the new context-stack-frame.* ***Please let me know, if this is a wrong assumption****.*

correct - applications stack (stack containing main()) + stack used by threads are assigned and managed by the operating system
[Quoted text hidden]
only required for some architectures - it's used on intel architecture because the return address is stored at the stack (not in a cpu registers as other architectures do)
I guess that s390x doesn't require this -> see ARM64 or MIPS

> 1.
> 2. Another function *finish* is executed when the context-function completes it's execution. This basically sets the exit code to zero and exits from the function and clears the stack frame.
>
> 3. It is important to note that the absolute address of the *trampoline* and *finish* are stored at the appropriate locations on the context-stack. These address are used to invoke the functions.
>
> 4. It then returns the starting address of this context stack.
>
> In short, make_fcontext() is responsible for creating the inital context-stack and initializing it with data that is necessary for the context switching.

correct

> --
>
> ## 2. jump_fcontext()
>
> This file is used to jump between fibers/fiber_contexts. The following are the steps that occur:
>
> 1. Save the registers of the current ongoing functions.
> 2. Move to the context_stack frame of the context we want to switch to.
> 3. Restore the values of register from the context_stack.
> 4. Jump to the context_function.

> This is the function which I'm unable to understand clearly. My doubts are as follows. Please see the jump_fcontext() file from here for reference.
>
> 1. `leaq -0x38(%rsp), %rsp /* prepare stack */` *(line 33)*
>
>   What does the RSP actually point to here. Does RSP point to the lowest most address of the current ongoing context.

yes

> 1. If so, does it mean that whenever we
>   switch from one context to another we will always reduce the stack pointer making and then store the values of the calling function in this stack before we switch to next context in the context-stack.

yes, the stack of the current context (whic hwill be suspended) is used as storage for the content of the CPU registers

> 1.
>   Suppose we shift from *main( )* function to a context-fuction *fc1( )* , so does the RSP point to the address where the *jump_context( )* is called in the *main( )* function?

no - you have to distinghuis between local data and instructions
local data are stored at teh stack and are structured/managed by the stack frames -> RSP == stack pointer, gives access to the local data/stack content/stack frame
instruction pointer/program counter -> RIP == pointer to the sequence of instructions in the program

so invoking jump_fcontext() in main():
- jump_fcontext() is entered (you are inside the assembler code -> use gdb for debugging)-
- on entry of jump_fcontext() RSP points to the stack of main() and RIP points to the instruction comming AFTER jump_fcontext() in main()

> 1. I don't think this is the right understanding of mine, because the above line is executed after we shift into the *jump_fcontext( ).* If that's the case, the RSP should point to the lowest most address of the context of the current function. But how does RSP directly shift to the lowest most address of the context-stack as soon as we call jump_fcontext and also how does it know, which context-stack-frame it should shift to.

see above

> 1. `movq %rdi, %rsp` *(line 51)*
>
>   Doesn't ***%rdi*** point to the first argument of jump_fcontext(). And the first argument of jump_context is the address of the current context. Then we are using that addres to restore the values into the register. If that is the case, how is that that we are restoring the registers from fromt calling context. But shouldn't we be using the second argument which points to the context to which we want to jump.

/* restore RSP (pointing to context-data) from RDI */
RDI contains the fcontext_t poitner of the cotnext we want to switch to

> 1.
>   And according to the ABI ***%rsi*** points to the second argument, which points to the address of the context we have to switch.

RSI points to the transfer_t pointer -> transfer of data between context'

> 1.
> 2. `leaq 0x40(%rsp), %rsp /* prepare stack */` *(line 67)*
>
>   Why are we preparing another stack here? Is this stack for the context-function.

prepare room in order to store the registers of the current context

> 1.
> 2.
> ```
>  /* return transfer_t from jump */
> /* RAX == fctx, RDX == data */
> movq %rsi, %rdx
> /* pass transfer_t as first arg in context function */
> /* RDI == fctx, RSI == data */
> movq %rax, %rdi
> ```
>   Can you please explain what is actually hapenning here. Or maybe just point me to few references reading which I can decipher what this actually means.

transfer of data:
RAX is used to return data from functions -> return suspended context (== address of ist lowest stack address) from jump_fcontext()

two cases:
1. resume of a completly new context -> pass the suspended context as argument to the entry function -> hence pass it as RDI and RSI (it is a struct with two parameters)
2. resume of a older context that has been resumed at least one time ago -> pass suspended context as return value -> hence pass it as RAX and RDX (it is a struct with two elements)

-> you need to figure out how x390x ABI (callign convetion) passes a struct (transfer_t) from and to a function
[Quoted text hidden]

Naveen Naidu <naveennaidu479@gmail.com> 11 May 2019 at 09:16
To: Oliver Kowalke <oliver.kowalke@gmail.com>

Thanks a lot for taking your time to reply to the email. This mail has provided me with proper directions to look into. I again apologize for the inconvenience caused.