Context

Oliver Kowalke

Table of Contents

Overview

Boost.Context is a foundational library that provides a sort of cooperative multitasking on a single thread. By providing an abstraction of the current execution state in the current thread, including the stack (with local variables) and stack pointer, all registers and CPU flags, and the instruction pointer, a *fcontext_t* instance represents a specific point in the application's execution path. This is useful for building higher-level abstractions, like *coroutines*, *cooperative threads (userland threads)* or an equivalent to C# keyword *yield* in C++.

A *fcontext_t* provides the means to suspend the current execution path and to transfer execution control, thereby permitting another *fcontext_t* to run on the current thread. This state full transfer mechanism enables a *fcontext_t* to suspend execution from within nested functions and, later, to resume from where it was suspended. While the execution path represented by a *fcontext_t* only runs on a single thread, it can be migrated to another thread at any given time.

A context switch between threads requires system calls (involving the OS kernel), which can cost more than thousand CPU cycles on x86 CPUs. By contrast, transferring control among them requires only fewer than hundred CPU cycles because it does not involve system calls as it is done within a single thread.

In order to use the classes and functions described here, you can either include the specific headers specified by the descriptions of each class or function, or include the master library header:

```
#include <boost/context/all.hpp>
```

which includes all the other headers in turn.

All functions and classes are contained in the namespace *boost::context*.

Requirements

Boost.Context must be built for the particular compiler(s) and CPU architecture(s)s being targeted. **Boost.Context** includes assembly code and, therefore, requires GNU AS for supported POSIX systems, MASM for Windows/x86 systems and ARMasm for Windows/arm systems.

Note

MASM64 (ml64.exe) is a part of Microsoft's Windows Driver Kit.

Important

Please note that `address-model=64` must be given to bjam command line on 64bit Windows for 64bit build; otherwise 32bit code will be generated.

Important

For cross-compiling the lib you must specify certain additional properties at bjam command line: `target-os`, `abi`, `binary-format`, `architecture` and `address-model`.

Context

Each instance of *fcontext_t* represents a context (CPU registers and stack space). Together with its related functions *jump_fcontext()* and *make_fcontext()* it provides a execution control transfer mechanism similar interface like ucontext_t. *fcontext_t* and its functions are located in *boost::context* and the functions are declared as extern "C".

Warning

If *fcontext_t* is used in a multi threaded application, it can migrated between threads, but must not reference *thread-local storage*.

Important

The low level API is the part to port to new platforms.

Note

If *fiber-local storage* is used on Windows, the user is responsible for calling *::FlsAlloc()*, *::FlsFree()*.

Executing a context

A new context supposed to execute a *context-function* (returning void and accepting intptr_t as argument) will be created on top of the stack (at 16 byte boundary) by function *make_fcontext()*.

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

// creates and manages a protected stack (with guard page)
ctx::guarded_stack_allocator alloc;
void * sp( alloc.allocate(ctx::minimum_stacksize()));
std::size_t size( ctx::guarded_stack_allocator::minimum_stacksize());

// 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));
```

Calling *jump_fcontext()* invokes the *context-function* in a newly created context complete with registers, flags, stack and instruction pointers. When control should be returned to the original calling context, call *jump_fcontext()*. The current context information (registers, flags, and stack and instruction pointers) is saved and the original context information is restored. Calling *jump_fcontext()* again resumes execution in the second context after saving the new state of the original context.

```
namespace ctx = boost::context;

ctx::fcontext_t fcm, * fc1, * fc2;

void f1( intptr_t)
{
        std::cout << "f1: entered" << std::endl;
        std::cout << "f1: call jump_fcontext( fc1, fc2, 0)" << std::endl;
        ctx::jump_fcontext( fc1, fc2, 0);
        std::cout << "f1: return" << std::endl;
        ctx::jump_fcontext( fc1, & fcm, 0);
}

void f2( intptr_t)
{
        std::cout << "f2: entered" << std::endl;
        std::cout << "f2: call jump_fcontext( fc2, fc1, 0)" << std::endl;
        ctx::jump_fcontext( fc2, fc1, 0);
        BOOST_ASSERT( false && ! "f2: never returns");
}

int main( int argc, char * argv[])
{
        ctx::guarded_stack_allocator alloc;
        void * sp1( alloc.allocate(ctx::minimum_stacksize()));
        std::size_t size( ctx::guarded_stack_allocator::minimum_stacksize());

        fc1 = ctx::make_fcontext( sp1, size, f1);
        fc2 = ctx::make_fcontext( sp2, size, f2);

        std::cout << "main: call jump_fcontext( & fcm, fc1, 0)" << std::endl;
        ctx::jump_fcontext( & fcm, fc1, 0);

        std::cout << "main: done" << std::endl;

        return EXIT_SUCCESS;
}

output:
    main: call jump_fcontext( & fcm, & fc1, 0)
    f1: entered
    f1: call jump_fcontext( & fc1, & fc2, 0)
    f2: entered
    f2: call jump_fcontext( & fc2, & fc1, 0)
    f1: return
    main: done
```

First call of *jump_fcontext()* enters the *context-function* `f1()` by starting context fc1 (context fcm saves the registers of `main()`). For jumping between context's fc1 and fc2 `jump_fcontext()` is called. Because context fcm is chained to fc1, `main()` is entered (returning from *jump_fcontext()*) after context fc1 becomes complete (return from `f1()`).

Warning

Calling *jump_fcontext()* to the same context from inside the same context results in undefined behaviour.

Important

The size of the stack is required to be larger than the size of fcontext_t.

Note

In contrast to threads, which are preemtive, *fcontext_t* switches are cooperative (programmer controls when switch will happen). The kernel is not involved in the context switches.

Transfer of data

The third argument passed to *jump_fcontext()*, in one context, is passed as the first argument of the *context-function* if the context is started for the first time. In all following invocations of *jump_fcontext()* the intptr_t passed to *jump_fcontext()*, in one context, is returned by *jump_fcontext()* in the other context.

```
namespace ctx = boost::context;

ctx::fcontext_t fcm, * fc;

typedef std::pair< int, int >   pair_t;

void f( intptr_t param)
{
    pair_t * p = ( pair_t *) param;

    p = ( pair_t *) ctx::jump_fcontext( fc, & fcm, ( intptr_t) ( p->first + p->second) );

    ctx::jump_fcontext( fc, & fcm, ( intptr_t) ( p->first + p->second) );
}

int main( int argc, char * argv[])
{
    ctx::guarded_stack_allocator alloc;
    void * sp( alloc.allocate(ctx::minimum_stacksize()));
    std::size_t size( ctx::guarded_stack_allocator::minimum_stacksize());

    pair_t p( std::make_pair( 2, 7) );
    fc = ctx::make_fcontext( sp, size, f);

    int res = ( int) ctx::jump_fcontext( & fcm, fc, ( intptr_t) & p);
    std::cout << p.first << " + " << p.second << " == " << res << std::endl;

    p = std::make_pair( 5, 6);
    res = ( int) ctx::jump_fcontext( & fcm, fc, ( intptr_t) & p);
    std::cout << p.first << " + " << p.second << " == " << res << std::endl;

    std::cout << "main: done" << std::endl;

    return EXIT_SUCCESS;
}

output:
    2 + 7 == 9
    5 + 6 == 11
    main: done
```

Exceptions in *context-function*

If the *context-function* emits an exception, the behaviour is undefined.

Important

context-function should wrap the code in a try/catch block.

Preserving floating point registers

Preserving the floating point registers increases the cycle count for a context switch (see performance tests). The fourth argument of *jump_fcontext()* controls if fpu registers should be preserved by the context jump.

Important

The use of the fpu controlling argument of *jump_fcontext()* must be consistent in the application. Otherwise the behaviour is undefined.

Stack unwinding

Sometimes it is necessary to unwind the stack of an unfinished context to destroy local stack variables so they can release allocated resources (RAII pattern). The user is responsible for this task.

Struct `fcontext_t` and related functions

```
struct stack_t
{
    void    *   sp;
    std::size_t size;
};

struct fcontext_t
{
    < platform specific >

    stack_t  fc_stack;
};

intptr_t jump_fcontext( fcontext_t * ofc, fcontext_t const* nfc, intptr_t vp, bool pre↵
serve_fpu = true);
fcontext_t * make_fcontext( void * sp, std::size_t size, void(* fn)(intptr_t) );
```

`sp`

Member: Pointer to the beginning of the stack (depending of the architecture the stack grows downwards or upwards).

`size`

Member: Size of the stack in bytes.

`fc_stack`

Member: Tracks the memory for the context's stack.

`intptr_t jump_fcontext( fcontext_t * ofc, fcontext_t * nfc, intptr_t p, bool preserve_fpu = true)`

Effects: Stores the current context data (stack pointer, instruction pointer, and CPU registers) to `*ofc` and restores the context data from `*nfc`, which implies jumping to `*nfc`'s execution context. The intptr_t argument, `p`, is passed to the current

context to be returned by the most recent call to `jump_fcontext()` in the same thread. The last argument controls if fpu registers have to be preserved.

Returns: The third pointer argument passed to the most recent call to `jump_fcontext()`, if any.

```
fcontext_t * make_fcontext( void * sp, std::size_t size, void(*fn)(intptr_t))
```

Precondition: Stack `sp` function pointer `fn` are valid (depending on the architecture `sp` points to the top or bottom of the stack) and `size` > 0.

Effects: Creates an fcontext_t at the beginning of the stack and prepares the stack to execute the *context-function* `fn`.

Returns: Returns a pointer to fcontext_t which is placed on the stack.

Stack allocation

A *fcontext_t* requires a stack which will be allocated/deallocated by a *StackAllocator* (examples contain an implementation of simple_stack_allocator).

Note

The implementation of a *StackAllocator* might include logic to protect against exceeding the context's available stack size rather than leaving it as undefined behaviour.

Note

The stack is not required to be aligned; alignment takes place inside *make_fcontext()*.

Note

Depending on the architecture *StackAllocator* returns an address from the top of the stack (grows downwards) or the bottom of the stack (grows upwards).

Performance

Performance of **Boost.Context** was measured on the platforms shown in the following table. Performance measurements were taken using `rdtsc`, with overhead corrections, on x86 platforms. In each case, stack protection was active, cache warm-up was accounted for, and the one running thread was pinned to a single CPU. The code was compiled using the build options, 'variant = release cxxflags = -DBOOST_DISABLE_ASSERTS'.

Applying `-DBOOST_USE_UCONTEXT` to cxxflags the performance of `ucontext` will be measured too.

The numbers in the table are the number of cycles per iteration, based upon an average computed over 10 iterations.

Table 1. Performance of context switch

Platform	ucontext_t	fcontext_t with fpu	fcontext_t without fpu	boost::function
AMD Athlon 64 Dual-Core 4400+ (32bit Linux)	846 cycles	65 cycles	43 cycles	15 cycles
Intel Core2 Q6700 (64bit Linux)	1481 cycles	172 cycles	63 cycles	25 cycles

Rationale

No inline-assembler

Some newer compiler (for instance MSVC 10 for x86_64 and itanium) do not support inline assembler. [1]. Inlined assembler generates code bloating which his not welcome on embedded systems.

fcontext_t

Boost.Context provides the low level API fcontext_t which is implemented in assembler to provide context swapping operations. fcontext_t is the part to port to new platforms.

Note

Context switches do not preserve the signal mask on UNIX systems.

Because the assembler code uses the byte layout of *fcontext_t* to access its members *fcontext_t* must be a POD. This requires that *fcontext_t* has only a default constructor, no visibility keywords (e.g. private, public, protected), no virtual methods and all members and base classes are PODs too.

Protecting the stack

Because the stack's size is fixed -- there is no support for split stacks yet -- it is important to protect against exceeding the stack's bounds. Otherwise, in the best case, overrunning the stack's memory will result in a segmentation fault or access violation and, in the worst case, the application's memory will be overwritten. `stack_allocator` appends a guard page to the stack to help detect overruns. The guard page consumes no physical memory, but generates a segmentation fault or access violation on access to the virtual memory addresses within it.

Other APIs

setjmp()/longjmp()

C99 defines `setjmp()/longjmp()` to provide non-local jumps but it does not require that *longjmp()* preserves the current stack frame. Therefore, jumping into a function which was exited via a call to *longjmp()* is undefined [2].

ucontext_t

Since POSIX.1-2003 `ucontext_t` is deprecated and was removed in POSIX.1-2008! The function signature of `makecontext()` is:

```
void makecontext(ucontext_t *ucp, void (*func)(), int argc, ...);
```

The third argument of `makecontext()` specifies the number of integer arguments that follow which will require function pointer cast if `func` will accept those arguments which is undefined in C99 [3].

The arguments in the var-arg list are required to be integers, passing pointers in var-arg list is not guaranteed to work, especially it will fail for architectures where pointers are larger than integers.

`ucontext_t` preserves signal mask between context switches which involves system calls consuming a lot of CPU cycles (ucontext_t is slower by perfomance_link[factor 13x] relative to `fcontext_t`).

[1] MSDN article 'Inline Assembler'
[2] ISO/IEC 9899:1999, 2005, 7.13.2.1:2
[3] ISO/IEC 9899:1999, 2005, J.2

Windows fibers

A drawback of Windows Fiber API is that `CreateFiber()` does not accept a pointer to user allocated stack space preventing the reuse of stacks for other context instances. Because the Windows Fiber API requires to call `ConvertThreadToFiber()` if `SwitchFiber()` is called for a thread which has not been converted to a fiber. For the same reason `ConvertFiberToThread()` must be called after return from `SwitchFiber()` if the thread was forced to be converted to a fiber before (which is inefficient).

```
if ( ! is_a_fiber() )
{
    ConvertThreadToFiber( 0);
    SwitchToFiber( ctx);
    ConvertFiberToThread();
}
```

If the condition `_WIN32_WINNT >= _WIN32_WINNT_VISTA` is met function `IsThreadAFiber()` is provided in order to detect if the current thread was already converted. Unfortunately Windows XP + SP 2/3 defines `_WIN32_WINNT >= _WIN32_WINNT_VISTA` without providing `IsThreadAFiber()`.

x86 and floating-point env

i386

"The FpCsr and the MxCsr register must be saved and restored before any call or return by any procedure that needs to modify them ..." [4].

x86_64

Windows

MxCsr - "A callee that modifies any of the non-volatile fields within MxCsr must restore them before returning to its caller. Furthermore, a caller that has modified any of these fields must restore them to their standard values before invoking a callee ..." [5].

FpCsr - "A callee that modifies any of the fields within FpCsr must restore them before returning to its caller. Furthermore, a caller that has modified any of these fields must restore them to their standard values before invoking a callee ..." [6].

"The MMX and floating-point stack registers (MM0-MM7/ST0-ST7) are preserved across context switches. There is no explicit calling convention for these registers." [7].

"The 64-bit Microsoft compiler does not use ST(0)-ST(7)/MM0-MM7". [8].

"XMM6-XMM15 must be preserved" [9]

SysV

"The control bits of the MxCsr register are callee-saved (preserved across calls), while the status bits are caller-saved (not preserved). The x87 status word register is caller-saved, whereas the x87 control word (FpCsr) is callee-saved." [10].

[4] 'Calling Conventions', Agner Fog
[5] MSDN article 'MxCsr'
[6] MSDN article 'FpCsr'
[7] MSDN article 'Legacy Floating-Point Support'
[8] 'Calling Conventions', Agner Fog
[9] MSDN article 'Register Usage'
[10] SysV ABI AMD64 Architecture Processor Supplement Draft Version 0.99.4, 3.2.1

Reference

ARM

- AAPCS ABI: Procedure Call Standard for the ARM Architecture
- AAPCS/LINUX: ARM GNU/Linux Application Binary Interface Supplement

MIPS

- O32 ABI: SYSTEM V APPLICATION BINARY INTERFACE, MIPS RISC Processor Supplement

PowerPC32

- SYSV ABI: SYSTEM V APPLICATION BINARY INTERFACE PowerPC Processor Supplement

PowerPC64

- SYSV ABI: PowerPC User Instruction Set Architecture, Book I

X86-32

- SYSV ABI: SYSTEM V APPLICATION BINARY INTERFACE, Intel386TM Architecture Processor Supplement
- MS PE: Calling Conventions

X86-64

- SYSV ABI: System V Application Binary Interface, AMD64 Architecture Processor Supplement
- MS PE: x64 Software Conventions

Todo

- provide support for SPARC, SuperH (SH4), S/390
- support split-stack feature from gcc/gold linker

Acknowledgments

I'd like to thank Adreas Fett, Artyom Beilis, Daniel Larimer, David Deakins, Fernando Pelliccioni, Giovanni Piero Deretta, Gordon Woodhull, Helge Bahmann, Holger Grund, Jeffrey Lee Hellrung (Jr.), Keith Jeffery, Martin Husemann, Phil Endecott, Robert Stewart, Sergey Cheban, Steven Watanabe, Vicente J. Botet Escriba, Wayne Piekarski.