


Join GitHub today

GitHub is home to over 40 million developers working together to host and review code, manage projects, and build software together.

Sign up

Dismiss

A blazing fast and lightweight C asymmetric coroutine library



#c #coroutine #coroutine-library #high-performance #lightweight

 80 commits

 4 branches

 7 releases

 7 contributors






















 Apache-2.0

Branch: master

New pull request

Find File

Clone or download

 hnes	fix the osx platform in the testing matrix (the Homebrew had some inc... ..	Latest commit e3058b3 on 16 Feb
 img	images for readme.md	last year
 .travis.yml	fix the osx platform in the testing matrix (the Homebrew had some inc...	7 months ago
 LICENSE	Initial commit	last year
 README.md	fix a mistake in the README.md and its corresponding in README_zh.md	last year
 README_zh.md	fix a mistake in the README.md and its corresponding in README_zh.md	last year
 aco.c	add type cast operation after the return of malloc function	last year
 aco.h	prepare the drafting of release v1.2.4	last year
 aco_assert_override.h	add angle brackets to the email address	last year
 acosw.S	add support for MacOS.	last year
 make.sh	add some helpful error messages if the output directory doesn't exist...	8 months ago
 test.sh	add angle brackets to the email address	last year
 test_aco_benchmark.c	change `asm` to `__asm__` to support cc's `--std=c99` flag (from issue ...	last year
 test_aco_synopsis.c	add angle brackets to the email address	last year
 test_aco_tutorial_0.c	add angle brackets to the email address	last year
 test_aco_tutorial_1.c	add angle brackets to the email address	last year
 test_aco_tutorial_2.c	add angle brackets to the email address	last year
 test_aco_tutorial_3.c	add angle brackets to the email address	last year
 test_aco_tutorial_4.c	add angle brackets to the email address	last year
 test_aco_tutorial_5.c	add angle brackets to the email address	last year
 test_aco_tutorial_6.c	add angle brackets to the email address	last year

README.md

Name

libaco - A blazing fast and lightweight C asymmetric coroutine library.

The code name of this project is Arkenstone

Asymmetric COroutine & Arkenstone is the reason why it's been named `aco`.




Currently supports Sys V ABI of Intel386 and x86-64.

Here is a brief summary of this project:

- Along with the implementation of a production-ready C coroutine library, here is a detailed documentation about how to implement a *fastest* and *correct* coroutine library and also with a strict [mathematical proof](#);
- It has no more than 700 LOC but has the full functionality which you may want from a coroutine library;
- The [benchmark](#) part shows that a context switch between coroutines only takes about *10 ns* (in the case of standalone stack) on the AWS c5d.large machine;
- User could choose to create a new coroutine with a *standalone stack* or with a *shared stack* (could be shared with others);
- It is extremely memory efficient: *10,000,000* coroutines simultaneously to run cost only *2.8 GB* physical memory (run with tcmalloc, each coroutine has a *120B* copy-stack size configuration).

The phrase "*fastest*" in above means the fastest context switching implementation which complies to the Sys V ABI of Intel386 or AMD64.

[build](#) [passing](#) [release](#) [v1.2.4](#) [license](#) [Apache-2.0](#) [doc](#) [en](#) + [中文](#) [Tweet](#)

Issues and PRs are welcome   

Note: Please use [releases](#) instead of the `master` to build the final binary.

Table of Contents

- [Name](#)
- [Table of Contents](#)
- [Status](#)
- [Synopsis](#)
- [Description](#)
- [Build and Test](#)
 - [CFLAGS](#)
 - [Build](#)
 - [Test](#)
- [Tutorials](#)
- [API](#)
 - [aco_thread_init](#)
 - [aco_share_stack_new](#)
 - [aco_share_stack_new2](#)
 - [aco_share_stack_destroy](#)
 - [aco_create](#)
 - [aco_resume](#)
 - [aco_yield](#)
 - [aco_get_co](#)
 - [aco_get_arg](#)
 - [aco_exit](#)
 - [aco_destroy](#)
 - [MACROS](#)
- [Benchmark](#)
- [Proof of Correctness](#)
 - [Running Model](#)
 - [Mathematical Induction](#)
 - [Miscellaneous](#)
 - [Red Zone](#)
 - [Stack Pointer](#)
- [Best Practice](#)
- [TODO](#)
- [CHANGES](#)
- [Donation](#)

- [Copyright and License](#)

Status

Production ready.

Synopsis

```
#include "aco.h"
#include <stdio.h>

// this header would override the default C `assert`;
// you may refer the "API : MACROS" part for more details.
#include "aco_assert_override.h"

void foo(int ct) {
    printf("co: %p: yield to main_co: %d\n", aco_get_co(), *((int*)(aco_get_arg())));
    aco_yield();
    *((int*)(aco_get_arg())) = ct + 1;
}

void co_fp0() {
    printf("co: %p: entry: %d\n", aco_get_co(), *((int*)(aco_get_arg())));
    int ct = 0;
    while(ct < 6){
        foo(ct);
        ct++;
    }
    printf("co: %p: exit to main_co: %d\n", aco_get_co(), *((int*)(aco_get_arg())));
    aco_exit();
}

int main() {
    aco_thread_init(NULL);

    aco_t* main_co = aco_create(NULL, NULL, 0, NULL, NULL);
    aco_share_stack_t* sstk = aco_share_stack_new(0);

    int co_ct_arg_point_to_me = 0;
    aco_t* co = aco_create(main_co, sstk, 0, co_fp0, &co_ct_arg_point_to_me);

    int ct = 0;
    while(ct < 6){
        assert(co->is_end == 0);
        printf("main_co: yield to co: %p: %d\n", co, ct);
        aco_resume(co);
        assert(co_ct_arg_point_to_me == ct);
        ct++;
    }
    printf("main_co: yield to co: %p: %d\n", co, ct);
    aco_resume(co);
    assert(co_ct_arg_point_to_me == ct);
    assert(co->is_end);

    printf("main_co: destroy and exit\n");
    aco_destroy(co);
    co = NULL;
    aco_share_stack_destroy(sstk);
    sstk = NULL;
    aco_destroy(main_co);
    main_co = NULL;

    return 0;
}

# default build
$ gcc -g -O2 acosw.S aco.c test_aco_synopsis.c -o test_aco_synopsis
$ ./test_aco_synopsis
main_co: yield to co: 0x1887120: 0
co: 0x1887120: entry: 0
co: 0x1887120: yield to main_co: 0
```

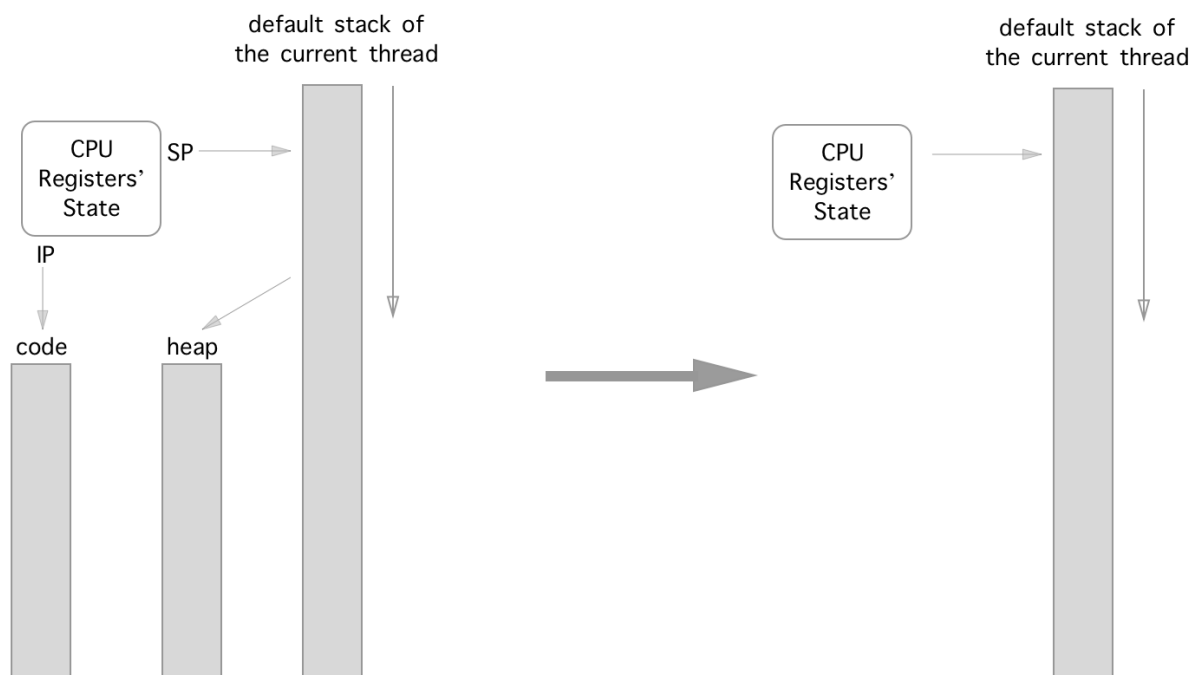
```

main_co: yield to co: 0x1887120: 1
co: 0x1887120: yield to main_co: 1
main_co: yield to co: 0x1887120: 2
co: 0x1887120: yield to main_co: 2
main_co: yield to co: 0x1887120: 3
co: 0x1887120: yield to main_co: 3
main_co: yield to co: 0x1887120: 4
co: 0x1887120: yield to main_co: 4
main_co: yield to co: 0x1887120: 5
co: 0x1887120: yield to main_co: 5
main_co: yield to co: 0x1887120: 6
co: 0x1887120: exit to main_co: 6
main_co: destroy and exit
# i386
$ gcc -g -m32 -O2 acosw.S aco.c test_aco_synopsis.c -o test_aco_synopsis
# share fpu and mxcsr env
$ gcc -g -D ACO_CONFIG_SHARE_FPU_MXCSR_ENV -O2 acosw.S aco.c test_aco_synopsis.c -o test_aco_synopsis
# with valgrind friendly support
$ gcc -g -D ACO_USE_VALGRIND -O2 acosw.S aco.c test_aco_synopsis.c -o test_aco_synopsis
$ valgrind --leak-check=full --tool=memcheck ./test_aco_synopsis

```

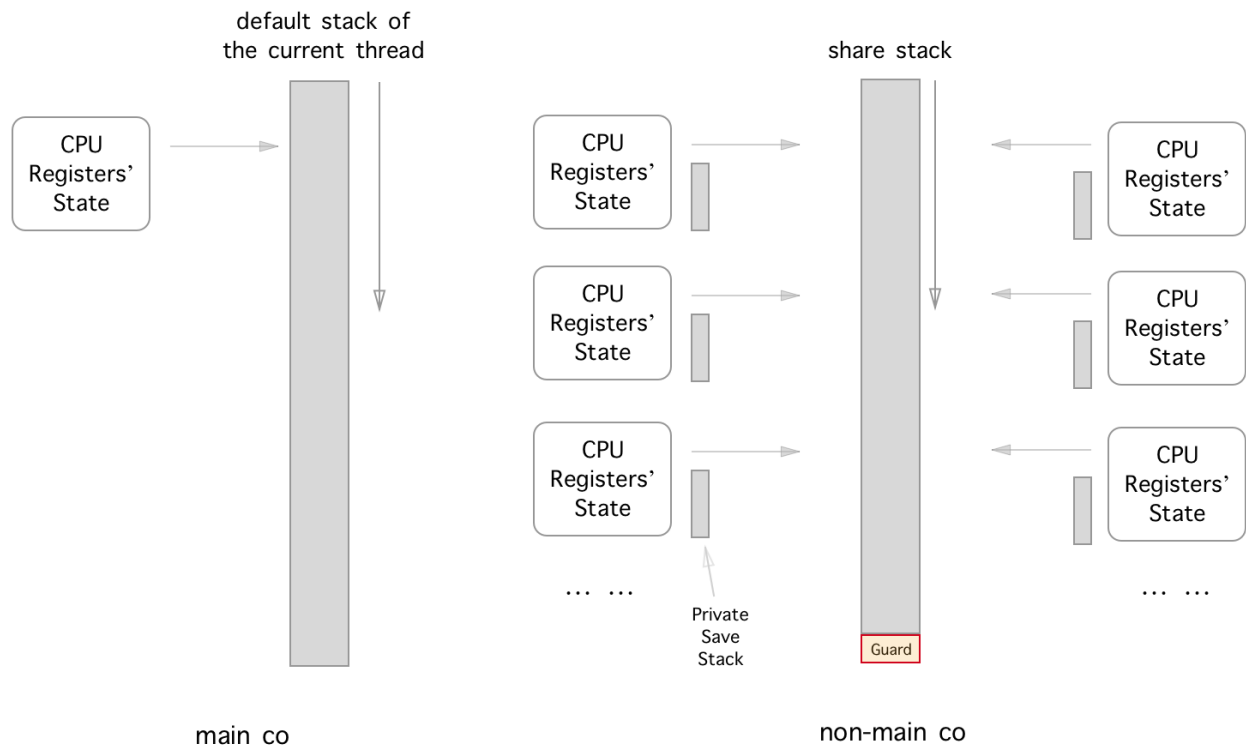
For more information you may refer to the "[Build and Test](#)" part.

Description



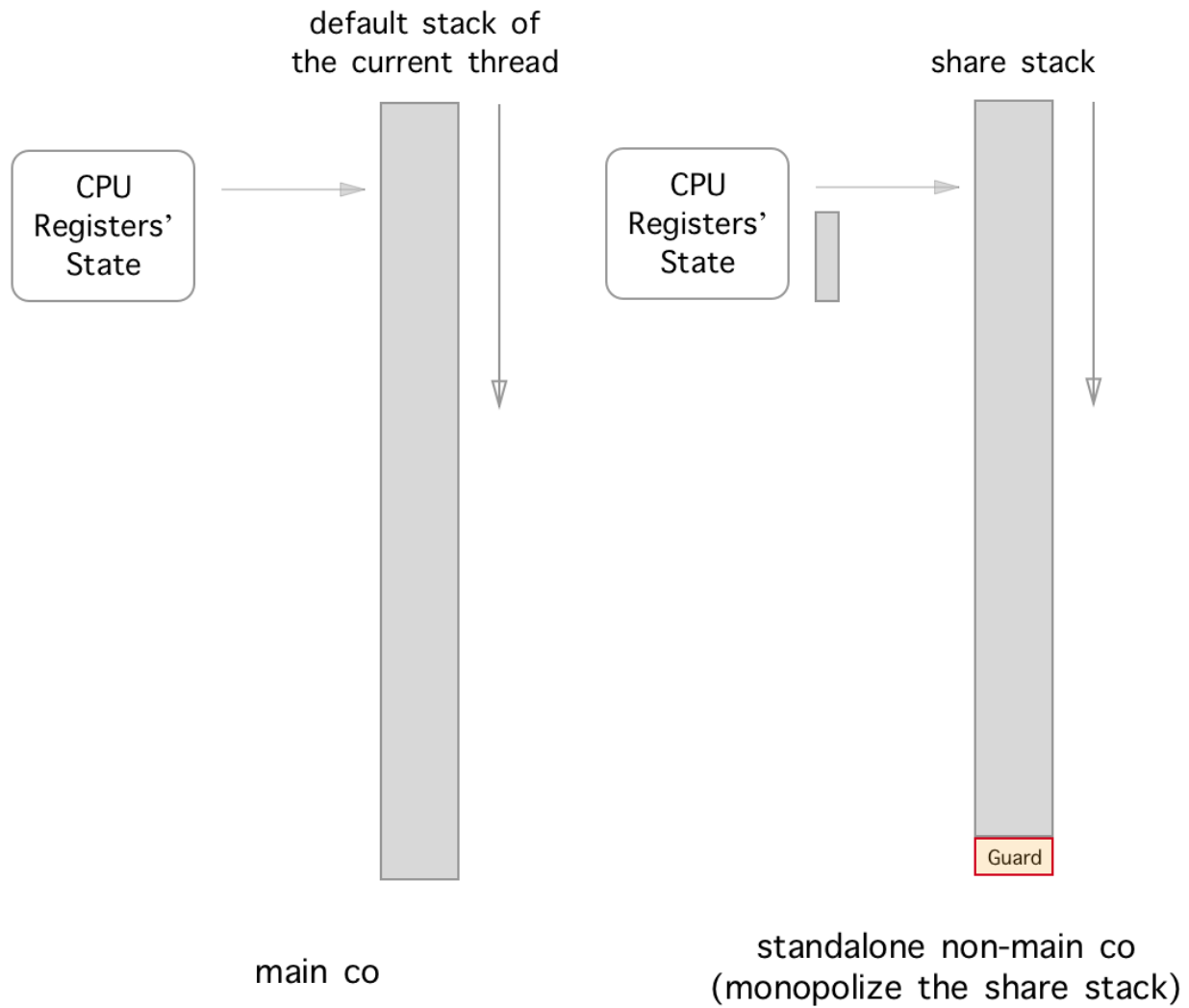
There are 4 basic elements of an ordinary execution state: {cpu_registers, code, heap, stack}.

Since the code information is indicated by $\{E|R\}IP$ register, and the address of the memory allocated from heap is normally stored in the stack directly or indirectly, thus we could simplify the 4 elements into only 2 of them: {cpu_registers, stack}.

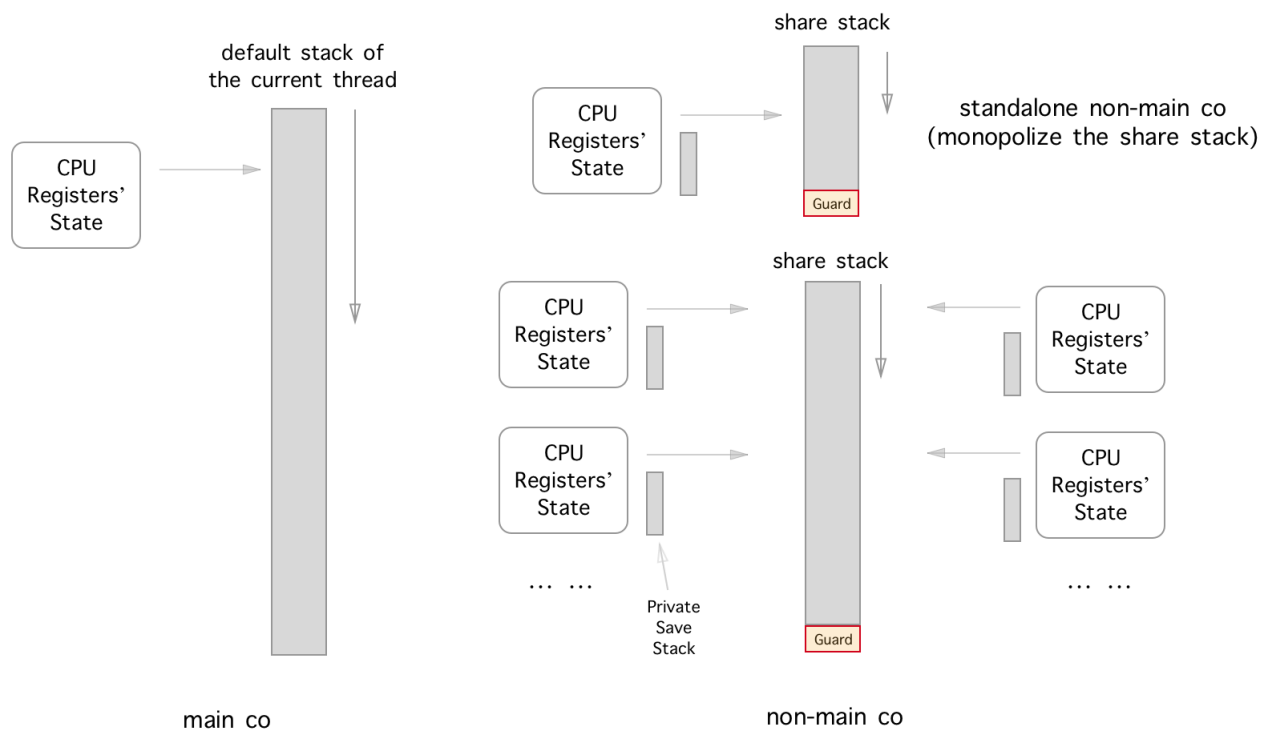


We define the `main co` as the coroutine who monopolizes the default stack of the current thread. And since the `main co` is the only user of this stack, we only need to save/restore the necessary cpu registers' state of the `main co` when it's been yielded-from/resumed-to (switched-out/switched-in).

Next, the definition of the `non-main co` is the coroutine whose execution stack is a stack which is not the default stack of the current thread and may be shared with the other `non-main co`. Thus the `non-main co` must have a `private save stack` memory buffer to save/restore its execution stack when it is been switched-out/switched-in (because the succeeding/preceding `co` may would/had use/used the share stack as its execution stack).



There is a special case of non-main co, that is `standalone non-main co` what we called in libaco: the share stack of the non-main coroutine has only one co user. Thus there is no need to do saving/restoring stuff of its private save stack when it is been switched-out/switched-in since there is no other co will touch the execution stack of the standalone non-main co except itself.



Finally, we get the big picture of libaco.

There is a "[Proof of Correctness](#)" part you may find really helpful if you want to dive into the internal of libaco or want to implement your own coroutine library.

It is also highly recommended to read the source code of the tutorials and benchmark next. The [benchmark](#) result is very impressive and enlightening too.

Build and Test

CFLAGS

- `-m32`

The `-m32` option of gcc could help you to build the i386 application of libaco on a x86_64 machine.

- C macro: `ACO_CONFIG_SHARE_FPU_MXCSR_ENV`

You could define the global C macro `ACO_CONFIG_SHARE_FPU_MXCSR_ENV` to speed up the performance of context switching between coroutines slightly if none of your code would change the control words of FPU and MXCSR. If the macro is not defined, all the co would maintain its own copy of the FPU and MXCSR control words. It is recommended to always define this macro globally since it is very rare that one function needs to set its own special env of FPU or MXCSR instead of using the default env defined by the ISO C. But you may not need to define this macro if you are not sure of it.

- C macro: `ACO_USE_VALGRIND`

If you want to use the tool memcheck of valgrind to test the application, then you may need to define the global C macro `ACO_USE_VALGRIND` to enable the friendly support of valgrind in libaco. But it is not recommended to define this macro in the final release build for the performance reason. You may also need to install the valgrind headers (package name is "valgrind-devel" in centos for example) to build libaco application with C macro `ACO_USE_VALGRIND` defined. (The memcheck of valgrind only works well with the standalone co currently. In the case of the shared stack used by more than one non-main co, the memcheck of valgrind would generate many false positive reports. For more information you may refer to "[test_aco_tutorial_6.c](#)".)

- C macro: `ACO_USE_ASAN`

The global C macro `ACO_USE_ASAN` would enable the friendly support of [Address Sanitizer](#) in libaco (support both gcc and clang).

Build

To build the test suites of libaco:

```
$ mkdir output
$ bash make.sh
```

There is also some detailed options in make.sh:

```
$bash make.sh -h
Usage: make.sh [-o <no-m32|no-valgrind>] [-h]
```

Example:

```
# default build
bash make.sh
# build without the i386 binary output
bash make.sh -o no-m32
# build without the valgrind supported binary output
bash make.sh -o no-valgrind
# build without the valgrind supported and i386 binary output
bash make.sh -o no-valgrind -o no-m32
```

In short, using `-o no-valgrind` if you have no valgrind headers installed, `-o no-m32` if you have no 32-bit gcc development tools installed on a AMD64 host.

On MacOS, you need to [replace](#) the default `sed` and `grep` commands of MacOS with the GNU `sed` and `grep` to run `make.sh` and `test.sh` (such requirement would be removed in the future):

```
$ brew install grep --with-default-names
$ brew install gnu-sed --with-default-names
```

Test

```
$ cd output
$ bash ../test.sh
```

Tutorials

The `test_aco_tutorial_0.c` in this repository shows the basic usage of libaco. There is only one main co and one standalone non-main co in this tutorial. The comments in the source code is also very helpful.

The `test_aco_tutorial_1.c` shows the usage of some statistics of non-main co. The data structure of `aco_t` is very clear and is defined in `aco.h`.

There are one main co, one standalone non-main co and two non-main co (pointing to the same share stack) in `test_aco_tutorial_2.c`.

The `test_aco_tutorial_3.c` shows how to use libaco in a multithreaded process. Basically, one instance of libaco is designed only to work inside one certain thread to gain the maximum performance of context switching between coroutines. If you want to use libaco in multithreaded environment, simply to create one instance of libaco in each of the threads. There is no data-sharing across threads inside the libaco, and you have to deal with the data competition among multiple threads yourself (like what `gl_race_aco_yield_ct` does in this tutorial).

One of the rules in libaco is to call `aco_exit()` to terminate the execution of the non-main co instead of the default direct C style `return`, otherwise libaco will treat such behaviour as illegal and trigger the default protector whose job is to log the error information about the offending co to stderr and abort the process immediately. The `test_aco_tutorial_4.c` shows such "offending co" situation.

You could also define your own protector to substitute the default one (to do some customized "last words" stuff). But no matter in what case, the process will be aborted after the protector was executed. The `test_aco_tutorial_5.c` shows how to define the customized last word function.

The last example is a simple coroutine scheduler in `test_aco_tutorial_6.c`.

API

It would be very helpful to read the corresponding API implementation in the source code simultaneously when you are reading the following API description of libaco since the source code is pretty clear and easy to understand. And it is also recommended to read all the [tutorials](#) before reading the API document.

It is strongly recommended to read the [Best Practice](#) part before starting to write the real application of libaco (in addition to describing how to truly release libaco's extreme performance in your application, there is also a notice about the programming of libaco).

Note: The version control of libaco follows the spec: [Semantic Versioning 2.0.0](#). So the API in the following list have the compatibility guarantee. (Please note that there is no such guarantee for the API no in the list.)

aco_thread_init

```
typedef void (*aco_cofuncp_t)(void);
void aco_thread_init(aco_cofuncp_t last_word_co_fp);
```

Initializes the libaco environment in the current thread.

It will store the current control words of FPU and MXCSR into a thread-local global variable.

- If the global macro `ACO_CONFIG_SHARE_FPU_MXCSR_ENV` is not defined, the saved control words would be used as a reference value to set up the control words of the new co's FPU and MXCSR (in `aco_create`) and each co would maintain

its own copy of FPU and MXCSR control words during later context switching.

- If the global macro `ACO_CONFIG_SHARE_FPU_MXCSR_ENV` is defined, then all the co shares the same control words of FPU and MXCSR. You may refer the "[Build and Test](#)" part of this document for more information about this.

And as it said in the `test_aco_tutorial_5.c` of the "[Tutorials](#)" part, when the 1st argument `last_word_co_fp` is not NULL then the function pointed by `last_word_co_fp` will substitute the default protector to do some "last words" stuff about the offending co before the process is aborted. In such last word function, you could use `aco_get_co` to get the pointer of the offending co. For more information, you may read `test_aco_tutorial_5.c`.

aco_share_stack_new

```
aco_share_stack_t* aco_share_stack_new(size_t sz);
```

Equal to `aco_share_stack_new2(sz, 1)`.

aco_share_stack_new2

```
aco_share_stack_t* aco_share_stack_new2(size_t sz, char guard_page_enabled);
```

Creates a new share stack with a advisory memory size of `sz` in bytes and may have a guard page (read-only) for the detection of stack overflow which is depending on the 2nd argument `guard_page_enabled`.

To use the default size value (2MB) if the 1st argument `sz` equals 0. After some computation of alignment and reserve, this function will ensure the final valid length of the share stack in return:

- `final_valid_sz >= 4096`
- `final_valid_sz >= sz`
- `final_valid_sz % page_size == 0` if the `guard_page_enabled != 0`

And as close to the value of `sz` as possible.

When the value of the 2nd argument `guard_page_enabled` is 1, the share stack in return would have one read-only guard page for the detection of stack overflow while a value 0 of `guard_page_enabled` means without such guard page.

This function will always return a valid share stack.

aco_share_stack_destroy

```
void aco_share_stack_destroy(aco_share_stack_t* sstk);
```

Destory the share stack `ssstk`.

Be sure that all the co whose share stack is `ssstk` is already destroyed when you destroy the `ssstk`.

aco_create

```
typedef void (*aco_cofuncp_t)(void);
aco_t* aco_create(aco_t* main_co, aco_share_stack_t* share_stack,
    size_t save_stack_sz, aco_cofuncp_t co_fp, void* arg);
```

Create a new co.

If it is a `main_co` you want to create, just call: `aco_create(NULL, NULL, 0, NULL, NULL)`. Main co is a special standalone coroutine whose share stack is the default thread stack. In the thread, main co is the coroutine who should be created and started to execute before all the other non-main coroutine does.

Otherwise it is a non-main co you want to create:

- The 1st argument `main_co` is the main co the co will `aco_yield` to in the future context switching. `main_co` must not be NULL;

- The 2nd argument `share_stack` is the address of a share stack which the non-main co you want to create will use as its executing stack in the future. `share_stack` must not be NULL;
- The 3rd argument `save_stack_sz` specifies the init size of the private save stack of this co. The unit is in bytes. A value of 0 means to use the default size 64 bytes. Since automatical resizing would happen when the private save stack is not big enough to hold the executing stack of the co when it has to yield the share stack it is occupying to another co, you usually should not worry about the value of `sz` at all. But it will bring some performance impact to the memory allocator when a huge amount (say 10,000,000) of the co resizes their private save stack continuously, so it is very wise and highly recommended to set the `save_stack_sz` with a value equal to the maximum value of `co->save_stack.max_cpsz` when the co is running (You may refer to the "[Best Practice](#)" part of this document for more information about such optimization);
- The 4th argument `co_fp` is the entry function pointer of the co. `co_fp` must not be NULL;
- The last argument `arg` is a pointer value and will set to `co->arg` of the co to create. It could be used as a input argument for the co.

This function will always return a valid co. And we name the state of the co in return as "init" if it is a non-main co you want to create.

aco_resume

```
void aco_resume(aco_t* co);
```

Yield from the caller main co and to start or continue the execution of `co`.

The caller of this function must be a main co and must be `co->main_co`. And the 1st argument `co` must be a non-main co.

The first time you resume a `co`, it starts running the function pointing by `co->fp`. If `co` has already been yielded, `aco_resume` restarts it and continues the execution.

After the call of `aco_resume`, we name the state of the caller — main co as "yielded".

aco_yield

```
void aco_yield();
```

Yield the execution of `co` and resume `co->main_co`. The caller of this function must be a non-main co. And `co->main_co` must not be NULL.

After the call of `aco_yield`, we name the state of the caller — `co` as "yielded".

aco_get_co

```
aco_t* aco_get_co();
```

Return the pointer of the current non-main co. The caller of this function must be a non-main co.

aco_get_arg

```
void* aco_get_arg();
```

Equal to `(aco_get_co()->arg)`. And also, the caller of this function must be a non-main co.

aco_exit

```
void aco_exit();
```

In addition do the same as `aco_yield()`, `aco_exit()` also set `co->is_end` to 1 thus to mark the `co` at the status of "end".

aco_destroy

```
void aco_destroy(aco_t* co);
```

Destroy the `co`. The argument `co` must not be NULL. The private save stack would also been destroyed if the `co` is a non-main `co`.

MACROS

Version

```
#define ACO_VERSION_MAJOR 1
#define ACO_VERSION_MINOR 2
#define ACO_VERSION_PATCH 4
```

These 3 macros are defined in the header `aco.h` and the value of them follows the spec: [Semantic Versioning 2.0.0](#).

aco_assert_override.h

```
// provide the compiler with branch prediction information
#define likely(x)          aco_likely(x)
#define unlikely(x)        aco_unlikely(x)

// override the default `assert` for convenience when coding
#define assert(EX)         aco_assert(EX)

// equal to `assert((ptr) != NULL)`
#define assertptr(ptr)      aco_assertptr(ptr)

// assert the successful return of memory allocation
#define assertalloc_bool(b) aco_assertalloc_bool(b)
#define assertalloc_ptr(ptr) aco_assertalloc_ptr(ptr)
```

You could choose to include the header `"aco_assert_override.h"` to override the default C `"assert"` in the libaco application like `test_aco_synopsis.c` does (this header including should be at the last of the include directives list in the source file because the C `"assert"` is a C macro definition too) and define the 5 other macros in the above. Please do not include this header in the application source file if you want to use the default C `"assert"`.

For more details you may refer to the source file [aco_assert_override.h](#).

Benchmark

Date: Sat Jun 30 UTC 2018.

Machine: [c5d.large on AWS](#).

OS: RHEL-7.5 (Red Hat Enterprise Linux 7.5).

Here is a brief summary of the benchmark part:

- One time of the context switching between coroutines takes only about **10.29 ns** (in the case of standalone stack, where x87 and mxcsr control words are shared between coroutines);
- One time of the context switching between coroutines takes only about **10.38 ns** (in the case of standalone stack, where each coroutine maintains their own x87 and mxcsr control words);
- It is extremely memory efficient: it only costs **2.8 GB** of physical memory to run **10,000,000** coroutines simultaneously (with `tcmalloc`, where each coroutine has a **120 bytes** copy-stack size configuration).

```
$ LD_PRELOAD=/usr/lib64/libtcmalloc_minimal.so.4 ./test_aco_benchmark.no_valgrind.shareFPUenv
+build:x86_64
+build:-DACO_CONFIG_SHARE_FPU_MXCSR_ENV
+build:share fpu & mxcsr control words between coroutines
+build:undefined ACO_USE_VALGRIND
```

```
+build:without valgrind memcheck friendly support
```

```
sizeof(aco_t)=152:
```

comment	task_amount	all_time_cost	ns_per_op	speed
aco_create/init_save_stk_sz=64B op/s	1	0.000 s	230.00 ns/op	4347824.79
aco_resume/co_amount=1/copy_stack_size=0B op/s	20000000	0.412 s	20.59 ns/op	48576413.55
-> acosw op/s	40000000	0.412 s	10.29 ns/op	97152827.10
aco_destroy op/s	1	0.000 s	650.00 ns/op	1538461.66
aco_create/init_save_stk_sz=64B op/s	1	0.000 s	200.00 ns/op	5000001.72
aco_resume/co_amount=1/copy_stack_size=0B op/s	20000000	0.412 s	20.61 ns/op	48525164.25
-> acosw op/s	40000000	0.412 s	10.30 ns/op	97050328.50
aco_destroy op/s	1	0.000 s	666.00 ns/op	1501501.49
aco_create/init_save_stk_sz=64B op/s	2000000	0.131 s	65.50 ns/op	15266771.53
aco_resume/co_amount=2000000/copy_stack_size=8B op/s	20000000	0.666 s	33.29 ns/op	30043022.64
aco_destroy op/s	2000000	0.066 s	32.87 ns/op	30425152.25
aco_create/init_save_stk_sz=64B op/s	2000000	0.130 s	65.22 ns/op	15332218.24
aco_resume/co_amount=2000000/copy_stack_size=24B op/s	20000000	0.675 s	33.75 ns/op	29630018.73
aco_destroy op/s	2000000	0.067 s	33.45 ns/op	29898311.36
aco_create/init_save_stk_sz=64B op/s	2000000	0.131 s	65.42 ns/op	15286937.97
aco_resume/co_amount=2000000/copy_stack_size=40B op/s	20000000	0.669 s	33.45 ns/op	29891277.59
aco_destroy op/s	2000000	0.080 s	39.87 ns/op	25084242.29
aco_create/init_save_stk_sz=64B op/s	2000000	0.224 s	111.86 ns/op	8940010.49
aco_resume/co_amount=2000000/copy_stack_size=56B op/s	20000000	0.678 s	33.88 ns/op	29515473.53
aco_destroy op/s	2000000	0.067 s	33.42 ns/op	29922412.68
aco_create/init_save_stk_sz=64B op/s	2000000	0.131 s	65.74 ns/op	15211896.70
aco_resume/co_amount=2000000/copy_stack_size=120B op/s	20000000	0.769 s	38.45 ns/op	26010724.94
aco_destroy op/s	2000000	0.088 s	44.11 ns/op	22669240.25
aco_create/init_save_stk_sz=64B op/s	10000000	1.240 s	123.97 ns/op	8066542.54
aco_resume/co_amount=10000000/copy_stack_size=8B op/s	40000000	1.327 s	33.17 ns/op	30143409.55
aco_destroy op/s	10000000	0.328 s	32.82 ns/op	30467658.05
aco_create/init_save_stk_sz=64B op/s	10000000	0.659 s	65.94 ns/op	15165717.02
aco_resume/co_amount=10000000/copy_stack_size=24B op/s	40000000	1.345 s	33.63 ns/op	29737708.53
aco_destroy op/s	10000000	0.337 s	33.71 ns/op	29666697.09
aco_create/init_save_stk_sz=64B op/s	10000000	0.654 s	65.38 ns/op	15296191.35
aco_resume/co_amount=10000000/copy_stack_size=40B op/s	40000000	1.348 s	33.71 ns/op	29663992.77

op/s				
aco_destroy	10000000	0.336 s	33.56 ns/op	29794574.96
op/s				
aco_create/init_save_stk_sz=64B	10000000	0.653 s	65.29 ns/op	15316087.09
op/s				
aco_resume/co_amount=10000000/copy_stack_size=56B	40000000	1.384 s	34.60 ns/op	28902221.24
op/s				
aco_destroy	10000000	0.337 s	33.73 ns/op	29643682.93
op/s				
aco_create/init_save_stk_sz=64B	10000000	0.652 s	65.19 ns/op	15340872.40
op/s				
aco_resume/co_amount=10000000/copy_stack_size=120B	40000000	1.565 s	39.11 ns/op	25566255.73
op/s				
aco_destroy	10000000	0.443 s	44.30 ns/op	22574242.55
op/s				
aco_create/init_save_stk_sz=64B	2000000	0.131 s	65.61 ns/op	15241722.94
op/s				
aco_resume/co_amount=2000000/copy_stack_size=136B	20000000	0.947 s	47.36 ns/op	21114212.05
op/s				
aco_destroy	2000000	0.125 s	62.35 ns/op	16039466.45
op/s				
aco_create/init_save_stk_sz=64B	2000000	0.131 s	65.71 ns/op	15218784.72
op/s				
aco_resume/co_amount=2000000/copy_stack_size=136B	20000000	0.948 s	47.39 ns/op	21101216.29
op/s				
aco_destroy	2000000	0.125 s	62.73 ns/op	15941559.26
op/s				
aco_create/init_save_stk_sz=64B	2000000	0.131 s	65.49 ns/op	15270258.18
op/s				
aco_resume/co_amount=2000000/copy_stack_size=152B	20000000	1.069 s	53.44 ns/op	18714275.17
op/s				
aco_destroy	2000000	0.122 s	61.05 ns/op	16378678.85
op/s				
aco_create/init_save_stk_sz=64B	2000000	0.132 s	65.91 ns/op	15171336.62
op/s				
aco_resume/co_amount=2000000/copy_stack_size=232B	20000000	1.190 s	59.48 ns/op	16813230.99
op/s				
aco_destroy	2000000	0.123 s	61.26 ns/op	16324298.25
op/s				
aco_create/init_save_stk_sz=64B	2000000	0.131 s	65.68 ns/op	15224361.30
op/s				
aco_resume/co_amount=2000000/copy_stack_size=488B	20000000	1.828 s	91.40 ns/op	10941133.56
op/s				
aco_destroy	2000000	0.145 s	72.56 ns/op	13781182.82
op/s				
aco_create/init_save_stk_sz=64B	2000000	0.132 s	65.80 ns/op	15197461.34
op/s				
aco_resume/co_amount=2000000/copy_stack_size=488B	20000000	1.829 s	91.47 ns/op	10932139.32
op/s				
aco_destroy	2000000	0.149 s	74.70 ns/op	13387258.82
op/s				
aco_create/init_save_stk_sz=64B	1000000	0.067 s	66.63 ns/op	15007426.35
op/s				
aco_resume/co_amount=1000000/copy_stack_size=1000B	20000000	4.224 s	211.20 ns/op	4734744.76
op/s				
aco_destroy	1000000	0.093 s	93.36 ns/op	10711651.49
op/s				
aco_create/init_save_stk_sz=64B	1000000	0.066 s	66.28 ns/op	15086953.73
op/s				
aco_resume/co_amount=1000000/copy_stack_size=1000B	20000000	4.222 s	211.12 ns/op	4736537.93
op/s				
aco_destroy	1000000	0.094 s	94.09 ns/op	10627664.78
op/s				
aco_create/init_save_stk_sz=64B	100000	0.007 s	70.72 ns/op	14139923.59
op/s				
aco_resume/co_amount=100000/copy_stack_size=1000B	20000000	4.191 s	209.56 ns/op	4771909.70

op/s				
aco_destroy	100000	0.010 s	101.21 ns/op	9880747.28
op/s				
aco_create/init_save_stk_sz=64B	100000	0.007 s	66.62 ns/op	15010433.00
op/s				
aco_resume/co_amount=100000/copy_stack_size=2024B	20000000	7.002 s	350.11 ns/op	2856228.03
op/s				
aco_destroy	100000	0.016 s	159.69 ns/op	6262129.35
op/s				
aco_create/init_save_stk_sz=64B	100000	0.007 s	65.76 ns/op	15205994.08
op/s				
aco_resume/co_amount=100000/copy_stack_size=4072B	20000000	11.918 s	595.90 ns/op	1678127.54
op/s				
aco_destroy	100000	0.019 s	186.32 ns/op	5367189.85
op/s				
aco_create/init_save_stk_sz=64B	100000	0.006 s	63.03 ns/op	15865531.37
op/s				
aco_resume/co_amount=100000/copy_stack_size=7992B	20000000	21.808 s	1090.42 ns/op	917079.11
op/s				
aco_destroy	100000	0.038 s	378.33 ns/op	2643225.42
op/s				

```
$ LD_PRELOAD=/usr/lib64/libtcmalloc_minimal.so.4 ./test_aco_benchmark..no_valgrind.standaloneFPUenv
```

```
+build:x86_64
+build:undefined ACO_CONFIG_SHARE_FPU_MXCSR_ENV
+build:each coroutine maintain each own fpu & mxcsr control words
+build:undefined ACO_USE_VALGRIND
+build:without valgrind memcheck friendly support
```

```
sizeof(aco_t)=160:
```

comment	task_amount	all_time_cost	ns_per_op	speed
aco_create/init_save_stk_sz=64B	1	0.000 s	273.00 ns/op	3663004.27
op/s				
aco_resume/co_amount=1/copy_stack_size=0B	20000000	0.415 s	20.76 ns/op	48173877.75
op/s				
-> acosw	40000000	0.415 s	10.38 ns/op	96347755.51
op/s				
aco_destroy	1	0.000 s	381.00 ns/op	2624672.26
op/s				
aco_create/init_save_stk_sz=64B	1	0.000 s	212.00 ns/op	4716980.43
op/s				
aco_resume/co_amount=1/copy_stack_size=0B	20000000	0.415 s	20.75 ns/op	48185455.26
op/s				
-> acosw	40000000	0.415 s	10.38 ns/op	96370910.51
op/s				
aco_destroy	1	0.000 s	174.00 ns/op	5747123.38
op/s				
aco_create/init_save_stk_sz=64B	2000000	0.131 s	65.63 ns/op	15237386.02
op/s				
aco_resume/co_amount=2000000/copy_stack_size=8B	20000000	0.664 s	33.20 ns/op	30119155.82
op/s				
aco_destroy	2000000	0.065 s	32.67 ns/op	30604542.55
op/s				
aco_create/init_save_stk_sz=64B	2000000	0.131 s	65.33 ns/op	15305975.29
op/s				
aco_resume/co_amount=2000000/copy_stack_size=24B	20000000	0.675 s	33.74 ns/op	29638360.61
op/s				
aco_destroy	2000000	0.067 s	33.31 ns/op	30016633.42
op/s				
aco_create/init_save_stk_sz=64B	2000000	0.131 s	65.61 ns/op	15241767.78
op/s				
aco_resume/co_amount=2000000/copy_stack_size=40B	20000000	0.678 s	33.88 ns/op	29518648.08
op/s				
aco_destroy	2000000	0.079 s	39.74 ns/op	25163018.30
op/s				

aco_create/init_save_stk_sz=64B op/s	2000000	0.221 s	110.73 ns/op	9030660.30
aco_resume/co_amount=2000000/copy_stack_size=56B op/s	2000000	0.684 s	34.18 ns/op	29253416.65
aco_destroy op/s	2000000	0.067 s	33.40 ns/op	29938840.64
aco_create/init_save_stk_sz=64B op/s	2000000	0.131 s	65.60 ns/op	15244077.65
aco_resume/co_amount=2000000/copy_stack_size=120B op/s	2000000	0.769 s	38.43 ns/op	26021228.41
aco_destroy op/s	2000000	0.087 s	43.74 ns/op	22863987.42
aco_create/init_save_stk_sz=64B op/s	1000000	1.251 s	125.08 ns/op	7994958.59
aco_resume/co_amount=1000000/copy_stack_size=8B op/s	4000000	1.327 s	33.19 ns/op	30133654.80
aco_destroy op/s	1000000	0.329 s	32.85 ns/op	30439787.32
aco_create/init_save_stk_sz=64B op/s	1000000	0.674 s	67.37 ns/op	14843796.57
aco_resume/co_amount=1000000/copy_stack_size=24B op/s	4000000	1.354 s	33.84 ns/op	29548523.05
aco_destroy op/s	1000000	0.339 s	33.90 ns/op	29494634.83
aco_create/init_save_stk_sz=64B op/s	1000000	0.672 s	67.19 ns/op	14882262.88
aco_resume/co_amount=1000000/copy_stack_size=40B op/s	4000000	1.361 s	34.02 ns/op	29393520.19
aco_destroy op/s	1000000	0.338 s	33.77 ns/op	29609577.59
aco_create/init_save_stk_sz=64B op/s	1000000	0.673 s	67.31 ns/op	14857716.02
aco_resume/co_amount=1000000/copy_stack_size=56B op/s	4000000	1.371 s	34.27 ns/op	29181897.80
aco_destroy op/s	1000000	0.339 s	33.85 ns/op	29540633.63
aco_create/init_save_stk_sz=64B op/s	1000000	0.672 s	67.24 ns/op	14873017.10
aco_resume/co_amount=1000000/copy_stack_size=120B op/s	4000000	1.548 s	38.71 ns/op	25835542.17
aco_destroy op/s	1000000	0.446 s	44.61 ns/op	22415961.64
aco_create/init_save_stk_sz=64B op/s	2000000	0.132 s	66.01 ns/op	15148290.52
aco_resume/co_amount=2000000/copy_stack_size=136B op/s	2000000	0.944 s	47.22 ns/op	21177946.19
aco_destroy op/s	2000000	0.124 s	61.99 ns/op	16132721.97
aco_create/init_save_stk_sz=64B op/s	2000000	0.133 s	66.36 ns/op	15068860.85
aco_resume/co_amount=2000000/copy_stack_size=136B op/s	2000000	0.944 s	47.20 ns/op	21187541.38
aco_destroy op/s	2000000	0.124 s	62.21 ns/op	16073322.25
aco_create/init_save_stk_sz=64B op/s	2000000	0.131 s	65.62 ns/op	15238955.93
aco_resume/co_amount=2000000/copy_stack_size=152B op/s	2000000	1.072 s	53.61 ns/op	18652789.74
aco_destroy op/s	2000000	0.121 s	60.42 ns/op	16551368.04
aco_create/init_save_stk_sz=64B op/s	2000000	0.132 s	66.08 ns/op	15132547.65
aco_resume/co_amount=2000000/copy_stack_size=232B op/s	2000000	1.198 s	59.88 ns/op	16699389.91
aco_destroy op/s	2000000	0.121 s	60.71 ns/op	16471465.52

aco_create/init_save_stk_sz=64B op/s	2000000	0.133 s	66.50 ns/op	15036985.95
aco_resume/co_amount=2000000/copy_stack_size=488B op/s	2000000	1.853 s	92.63 ns/op	10796126.04
aco_destroy op/s	2000000	0.146 s	72.87 ns/op	13723559.36
aco_create/init_save_stk_sz=64B op/s	2000000	0.132 s	66.14 ns/op	15118324.13
aco_resume/co_amount=2000000/copy_stack_size=488B op/s	2000000	1.855 s	92.75 ns/op	10781572.22
aco_destroy op/s	2000000	0.152 s	75.79 ns/op	13194130.51
aco_create/init_save_stk_sz=64B op/s	1000000	0.067 s	66.97 ns/op	14931921.56
aco_resume/co_amount=1000000/copy_stack_size=1000B op/s	2000000	4.218 s	210.90 ns/op	4741536.66
aco_destroy op/s	1000000	0.093 s	93.16 ns/op	10734691.98
aco_create/init_save_stk_sz=64B op/s	1000000	0.066 s	66.49 ns/op	15039274.31
aco_resume/co_amount=1000000/copy_stack_size=1000B op/s	2000000	4.216 s	210.81 ns/op	4743543.53
aco_destroy op/s	1000000	0.094 s	93.97 ns/op	10641539.58
aco_create/init_save_stk_sz=64B op/s	100000	0.007 s	70.95 ns/op	14094724.73
aco_resume/co_amount=100000/copy_stack_size=1000B op/s	2000000	4.190 s	209.52 ns/op	4772746.50
aco_destroy op/s	100000	0.010 s	100.99 ns/op	9902271.51
aco_create/init_save_stk_sz=64B op/s	100000	0.007 s	66.49 ns/op	15040038.84
aco_resume/co_amount=100000/copy_stack_size=2024B op/s	2000000	7.028 s	351.38 ns/op	2845942.55
aco_destroy op/s	100000	0.016 s	159.15 ns/op	6283444.42
aco_create/init_save_stk_sz=64B op/s	100000	0.007 s	65.73 ns/op	15214482.36
aco_resume/co_amount=100000/copy_stack_size=4072B op/s	2000000	11.879 s	593.95 ns/op	1683636.60
aco_destroy op/s	100000	0.018 s	184.23 ns/op	5428119.00
aco_create/init_save_stk_sz=64B op/s	100000	0.006 s	63.41 ns/op	15771072.16
aco_resume/co_amount=100000/copy_stack_size=7992B op/s	2000000	21.808 s	1090.42 ns/op	917081.56
aco_destroy op/s	100000	0.038 s	376.78 ns/op	2654073.13

Proof of Correctness

It is essential to be very familiar with the standard of [Sys V ABI of intel386 and x86-64](#) before you start to implement or prove a coroutine library.

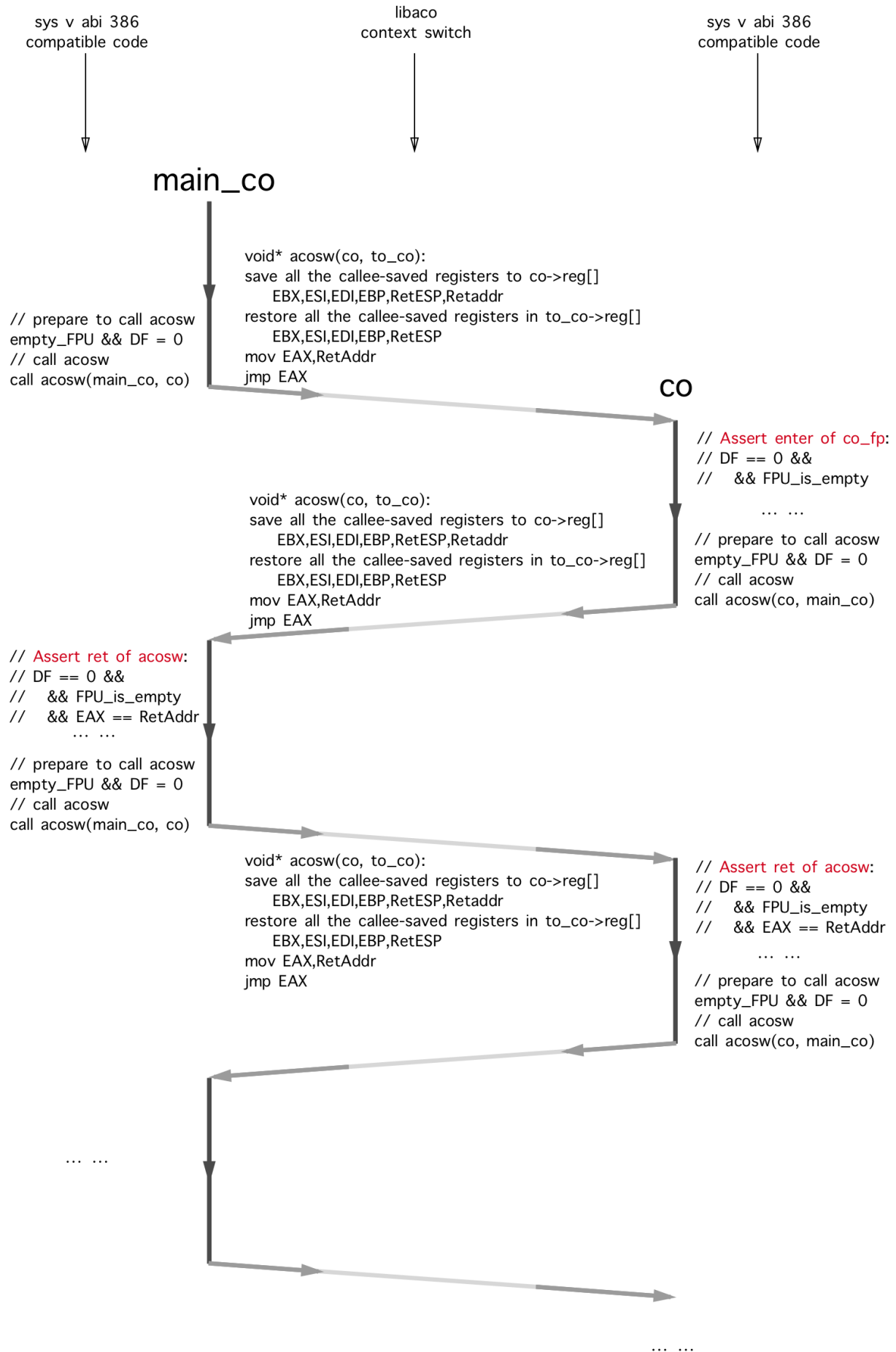
The proof below has no direct description about the IP (instruction pointer), SP (stack pointer) and the saving/restoring between the private save stack and the share stack, since these things are pretty trivial and easy to understand when they are compared with the ABI constraints stuff.

Running Model

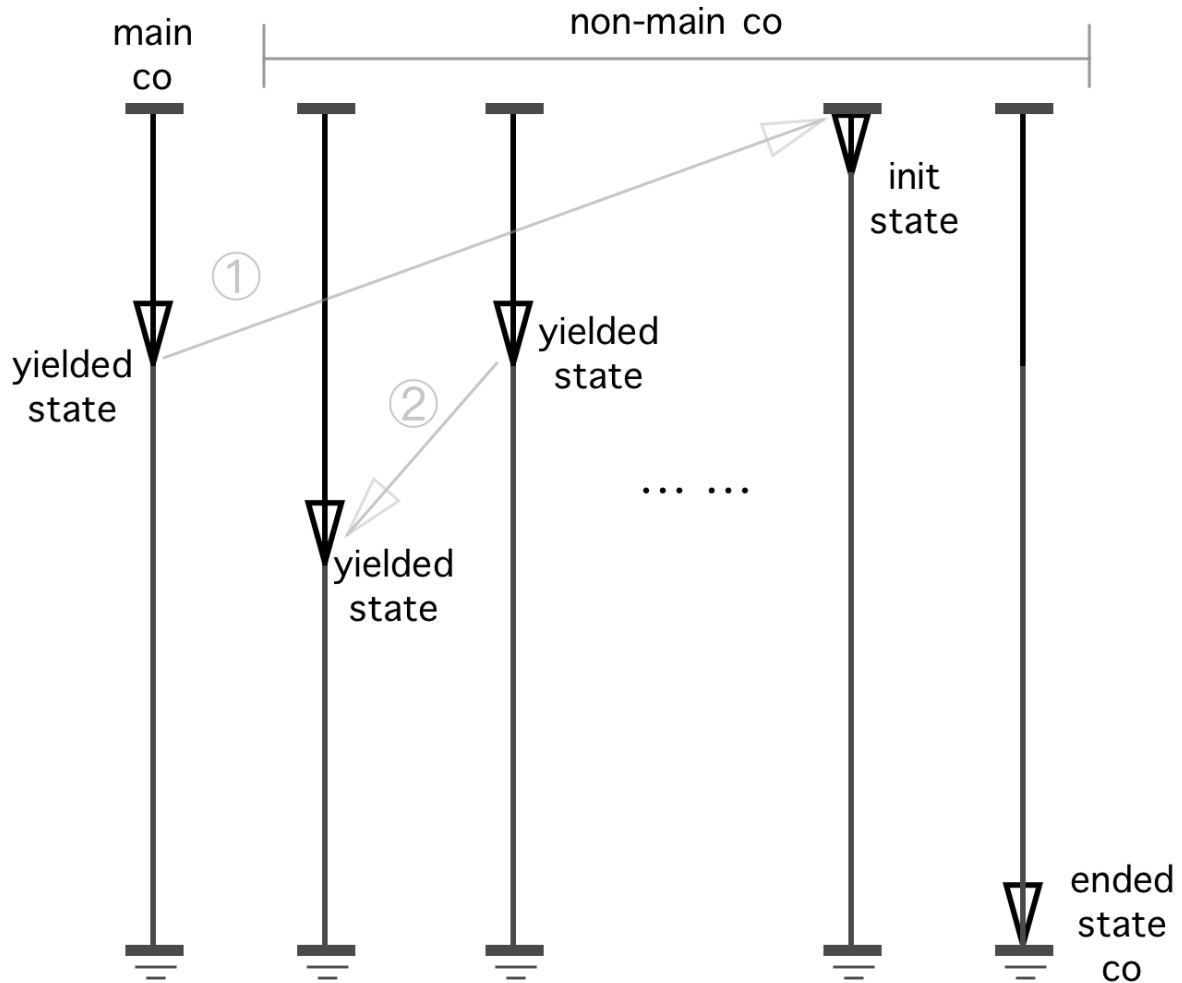
In the OS thread, the main coroutine `main_co` is the coroutine who should be created and started to execute first, before all the other non-main coroutines do.

The next diagram is a simple example of the context switching between `main_co` and `co`.

In this proof, we just assume that we are under Sys V ABI of intel386 since there is no fundamental differences between the Sys V ABI of intel386 and x86-64. We also assume that none of the code would change the control words of FPU and MXCSR.



The next diagram is actually a symmetric coroutine's running model which has an unlimited number of non-main co-s and one main co. This is fine because the asymmetric coroutine is just a special case of the symmetric coroutine. To prove the correctness of the symmetric coroutine is a little more challenging than of the asymmetric coroutine and thus more fun it would become. (libaco only implemented the API of asymmetric coroutine currently because the semantic meaning of the asymmetric coroutine API is far more easy to understand and to use than the symmetric coroutine does.)



Since the main co is the 1st coroutine starts to run, the 1st context switching in this OS thread must be in the form of `acosw(main_co, co)` where the 2nd argument `co` is a non-main co.

Mathematical Induction

It is easy to prove that there only exists two kinds of state transfer in the above diagram:

- yielded state co → init state co
- yielded state co → yielded state co

To prove the correctness of `void* acosw(aco_t* from_co, aco_t* to_co)` implementation is equivalent to prove all the co constantly comply to the constraints of Sys V ABI before and after the call of `acosw`. We assume that the other part of binary code (except `acosw`) in the co had already comply to the ABI (they are normally generated by the compiler correctly).

Here is a summary of the registers' constraints in the Function Calling Convention of Intel386 Sys V ABI:

Registers' usage in the calling convention of the Intel386 System V ABI:

caller saved (scratch) registers:

C1.0: EAX

At the entry of a function call:
could be any value

After the return of ``acosw``:
hold the return value for ``acosw``

C1.1: ECX, EDX

At the entry of a function call:
could be any value

After the return of ``acosw``:
could be any value

C1.2: Arithmetic flags, x87 and mxcsr flags
 At the entry of a function call:
 could be any value
 After the return of `acosw`:
 could be any value

C1.3: ST(0-7)
 At the entry of a function call:
 the stack of FPU must be empty
 After the return of `acosw`:
 the stack of FPU must be empty

C1.4: Direction flag
 At the entry of a function call:
 DF must be 0
 After the return of `acosw`:
 DF must be 0

C1.5: others: xmm*,ymm*,mm*,k*...
 At the entry of a function call:
 could be any value
 After the return of `acosw`:
 could be any value

callee saved registers:
 C2.0: EBX,ESI,EDI,EBP
 At the entry of a function call:
 could be any value
 After the return of `acosw`:
 must be the same as it is at the entry of `acosw`

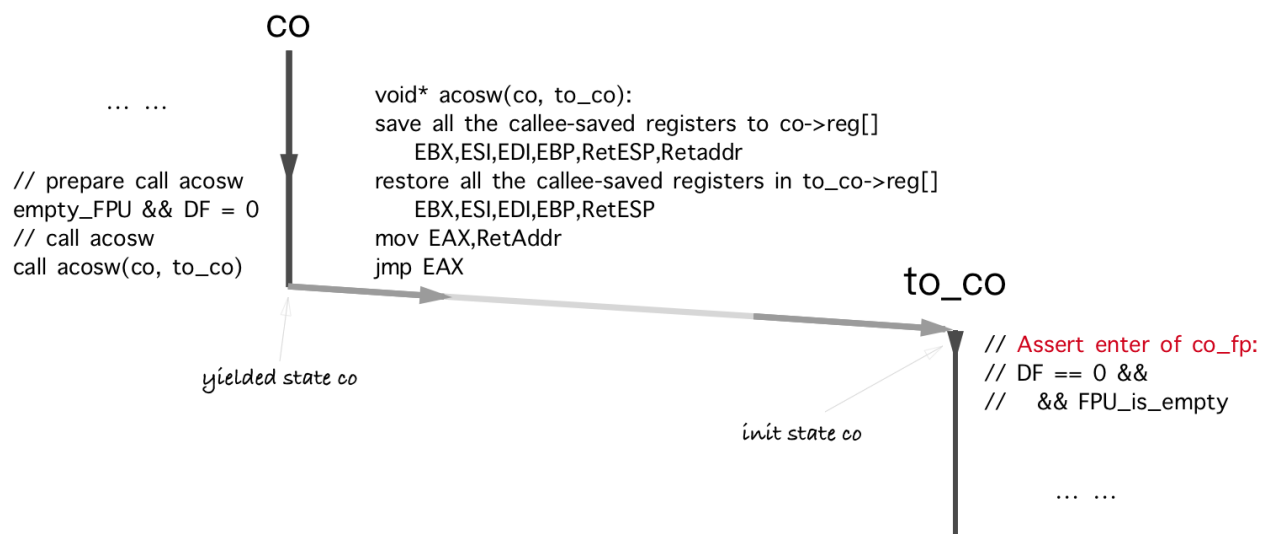
C2.1: ESP
 At the entry of a function call:
 must be a valid stack pointer
 (alignment of 16 bytes, retaddr and etc...)
 After the return of `acosw`:
 must be the same as it is before the call of `acosw`

C2.2: control word of FPU & mxcsr
 At the entry of a function call:
 could be any configuration
 After the return of `acosw`:
 must be the same as it is before the call of `acosw`
 (unless the caller of `acosw` assume `acosw` may \
 change the control words of FPU or MXCSR on purpose \
 like `fesetenv`)

(For Intel386, the register usage is defined in the "P13 - Table 2.3: Register Usage" of [Sys V ABI Intel386 V1.1](#), and for AMD64 is in "P23 - Figure 3.4: Register Usage" of [Sys V ABI AMD64 V1.0](#).)

Proof:

1. yielded state co -> init state co:



The diagram above is for the 1st case: "yielded state co -> init state co".

Constraints: C 1.0, 1.1, 1.2, 1.5 (*satisfied* ✓)

The scratch registers below can hold any value at the entry of a function:

```
EAX, ECX, EDX
XMM*, YMM*, MM*, K*...
status bits of EFLAGS, FPU, MXCSR
```

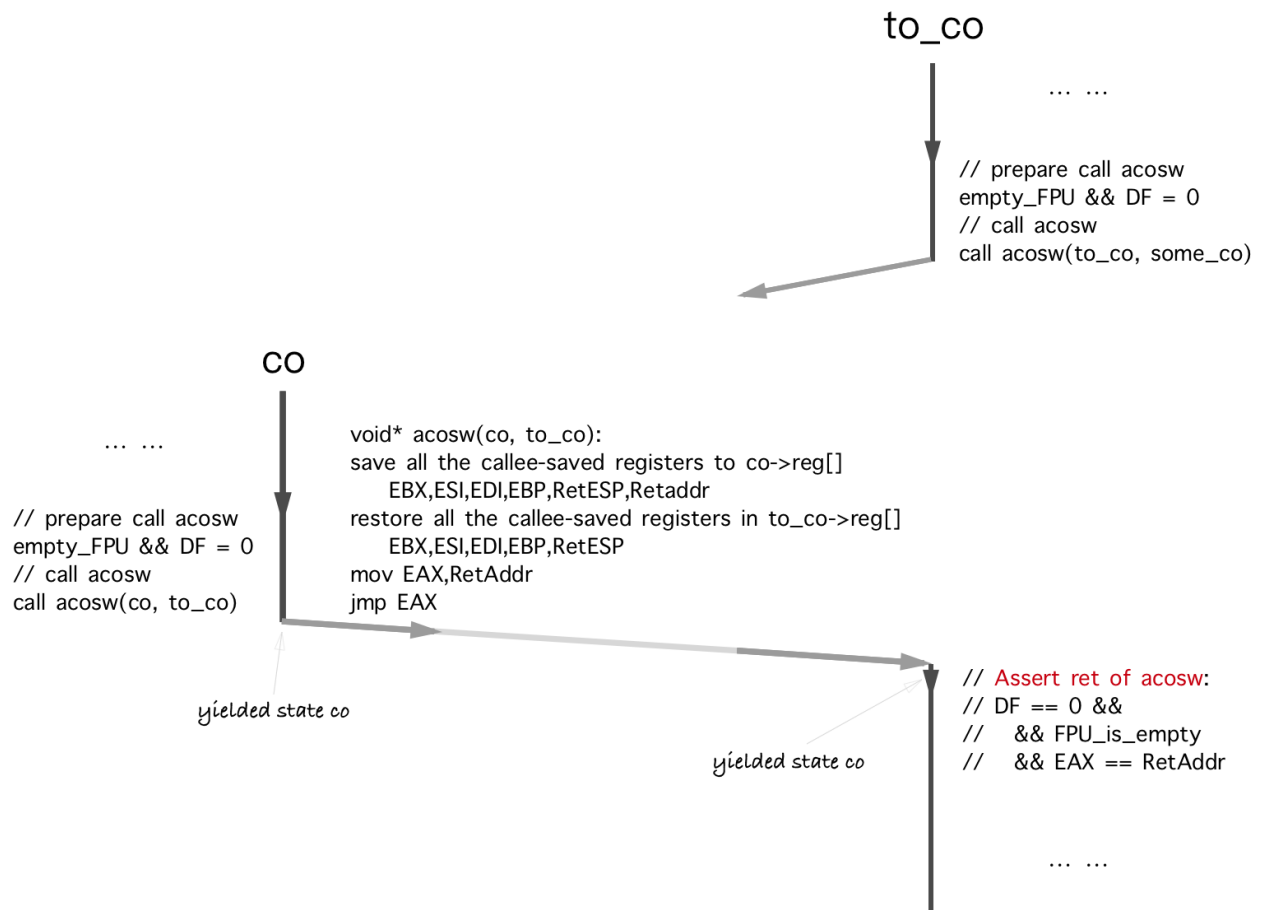
Constraints: C 1.3, 1.4 (*satisfied* ✓)

Since the stack of FPU must already be empty and the DF must already be 0 before `acosw(co, to_co)` was called (the binary code of `co` is already complied to the ABI), the constraint 1.3 and 1.4 is complied by `acosw`.

Constraints: C 2.0, 2.1, 2.2 (*satisfied* ✓)

C 2.0 & 2.1 is already satisfied. Since we already assumed that nobody will change the control words of FPU and MXCSR, C 2.2 is satisfied too.

2. yielded state `co` -> yielded state `co`:



The diagram above is for the 2nd case: yielded state `co` -> yielded state `co`.

Constraints: C 1.0 (*satisfied* ✓)

EAX already holding the return value when `acosw` returns back to `to_co` (resume).

Constraints: C 1.1, 1.2, 1.5 (*satisfied* ✓)

The scratch registers below can hold any value at the entry of a function and after the return of `acosw`:

```
ECX, EDX
XMM*, YMM*, MM*, K*...
status bits of EFLAGS, FPU, MXCSR
```

Constraints: C 1.3, 1.4 (*satisfied* ✓)

Since the stack of FPU must already be empty and the DF must already be 0 before `acosw(co, to_co)` was called (the binary code of `co` is already complied to the ABI), the constraint 1.3 and 1.4 is complied by `acosw`.

Constraints: C 2.0, 2.1, 2.2 (*satisfied* ✓)

C 2.0 & 2.1 is satisfied because there is saving & restoring of the callee saved registers when `acosw` been called/returned. Since we already assumed that nobody will change the control words of FPU and MXCSR, C 2.2 is satisfied too.

3. Mathematical induction:

The 1st `acosw` in the thread must be the 1st case: yielded state `co` -> init state `co`, and all the next `acosw` must be one of the 2 case above. Sequentially, we could prove that "all the `co` constantly comply to the constraints of Sys V ABI before and after the call of `acosw`". Thus, the proof is finished.

Miscellaneous

Red Zone

There is a new thing called [red zone](#) in System V ABI x86-64:

The 128-byte area beyond the location pointed to by `%rsp` is considered to be reserved and shall not be modified by signal or interrupt handlers. Therefore, functions may use this area for temporary data that is not needed across function calls. In particular, leaf functions may use this area for their entire stack frame, rather than adjusting the stack pointer in the prologue and epilogue. This area is known as the red zone.

Since the red zone is "not preserved by the callee", we just do not care about it at all in the context switching between coroutines (because the `acosw` is a leaf function).

Stack Pointer

The end of the input argument area shall be aligned on a 16 (32 or 64, if `__m256` or `__m512` is passed on stack) byte boundary. In other words, the value `(%esp + 4)` is always a multiple of 16 (32 or 64) when control is transferred to the function entry point. The stack pointer, `%esp`, always points to the end of the latest allocated stack frame.

— Intel386-psABI-1.1:2.2.2 The Stack Frame

The stack pointer, `%rsp`, always points to the end of the latest allocated stack frame.

— Sys V ABI AMD64 Version 1.0:3.2.2 The Stack Frame

Here is a [bug example](#) in Tencent's `libco`. The ABI states that the `(E|R)SP` should always point to the end of the latest allocated stack frame. But in file `cctx_swap.S` of `libco`, the `(E|R)SP` had been used to address the memory on the heap.

By default, the signal handler is invoked on the normal process stack. It is possible to arrange that the signal handler uses an alternate stack; see `sigaltstack(2)` for a discussion of how to do this and when it might be useful.

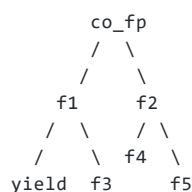
— man 7 signal : Signal dispositions

Terrible things may happen if the `(E|R)SP` is pointing to the data structure on the heap when signal comes. (Using the `breakpoint` and `signal` commands of `gdb` could produce such bug conveniently. Although by using `sigaltstack` to change the default signal stack could alleviate the problem, but still, that kind of usage of `(E|R)SP` still violates the ABI.)

Best Practice

In summary, if you want to gain the ultra performance of `libaco`, just keep the stack usage of the non-standalone non-main `co` at the point of calling `aco_yield` as small as possible. And be very careful if you want to pass the address of a local variable from one `co` to another `co` since the local variable is usually on the **share** stack. Allocating this kind of variables from the heap is always the wiser choice.

In detail, there are 5 tips:



1. The stack usage of main co has no direct influence to the performance of context switching between coroutines (since it has a standalone execution stack);
2. The stack usage of standalone non-main co has no direct influence to the performance of context switching between coroutines. But a huge amount of standalone non-main co would cost too much of virtual memory (due to the standalone stack), so it is not recommended to create huge amount of standalone non-main co in one thread;
3. The stack usage of non-standalone (share stack with other coroutines) non-main co when it is been yielded (i.e. call `aco_yield` to yield back to main co) has a big impact to the performance of context switching between coroutines, as already indicated by the benchmark results. In the diagram above, the stack usage of function `f2`, `f3`, `f4` and `f5` has no direct influence over the context switching performance since there are no `aco_yield` when they are executing, whereas the stack usage of `co_fp` and `f1` dominates the value of `co->save_stack.max_cpsz` and has a big influence over the context switching performance.

The key to keeping the stack usage of a function as low as possible is to allocate the local variables (especially the big ones) on the heap and manage their lifecycle manually instead of allocating them on the stack by default. The `-fstack-usage` option of gcc is very helpful about this.

```
int* gl_ptr;

void inc_p(int* p){ (*p)++; }

void co_fp0() {
    int ct = 0;
    gl_ptr = &ct; // line 7
    aco_yield();
    check(ct);
    int* ptr = &ct;
    inc_p(ptr); // line 11
    aco_exit();
}

void co_fp1() {
    do_sth(gl_ptr); // line 16
    aco_exit();
}
```

4. In the above code snippet, we assume that `co_fp0` & `co_fp1` shares the same share stack (they are both non-main co) and the running sequence of them is "`co_fp0 -> co_fp1 -> co_fp0`". Since they are sharing the same stack, the address holding in `gl_ptr` in `co_fp1` (line 16) has totally different semantics with the `gl_ptr` in line 7 of `co_fp0`, and that kind of code would probably corrupt the execution stack of `co_fp1`. But the line 11 is fine because variable `ct` and function `inc_p` are in the same coroutine context. Allocating that kind of variables (need to share with other coroutines) on the heap would simply solve such problems:

```
int* gl_ptr;

void inc_p(int* p){ (*p)++; }

void co_fp0() {
    int* ct_ptr = malloc(sizeof(int));
    assert(ct_ptr != NULL);
    *ct_ptr = 0;
    gl_ptr = ct_ptr;
    aco_yield();
    check(*ct_ptr);
    int* ptr = ct_ptr;
    inc_p(ptr);
    free(ct_ptr);
    gl_ptr = NULL;
    aco_exit();
}

void co_fp1() {
    do_sth(gl_ptr);
    aco_exit();
}
```

TODO

New ideas are welcome!

- Add a macro like `aco_mem_new` which is the combination of something like `p = malloc(sz); assertalloc_ptr(p)`.
- Add a new API `aco_reset` to support the reusability of the coroutine objects.
- Support other platforms (especially arm & arm64).

CHANGES

v1.2.4 Sun Jul 29 2018

Changed ``asm`` to ``__asm__`` in `aco.h` to support compiler's ``--std=c99`` flag (Issue #16, proposed by Theo Schlossnagle @postwait).

v1.2.3 Thu Jul 26 2018

Added support for MacOS;

Added support for shared library build of libaco (PR #10, proposed by Theo Schlossnagle @postwait);

Added C macro `ACO_REG_IDX_BP` in `aco.h` (PR #15, proposed by Theo Schlossnagle @postwait);

Added global C config macro `ACO_USE_ASAN` which could enable the friendly support of address sanitizer (both gcc and clang) (PR #14, proposed by Theo Schlossnagle @postwait);

Added `README_zh.md`.

v1.2.2 Mon Jul 9 2018

Added a new option ``-o <no-m32|no-valgrind>`` to `make.sh`;

Correction about the value of macro `ACO_VERSION_PATCH` (issue #1 kindly reported by Markus Elfring @elfring);

Adjusted some noncompliant naming of identifiers (double underscore ``__``) (issue #1, kindly proposed by Markus Elfring @elfring);

Supported the header file including by C++ (issue #4, kindly proposed by Markus Elfring @elfring).

v1.2.1 Sat Jul 7 2018

Fixed some noncompliant include guards in two C header files (issue #1 kindly reported by Markus Elfring @elfring);

Removed the "pure" word from "pure C" statement since it is containing assembly codes (kindly reported by Peter Cawley @corsix);

Many updates in the `README.md` document.

v1.2.0 Tue Jul 3 2018




Provided another header named ``aco_assert_override.h`` so user could choose to override the default ``assert`` or not;

Added some macros about the version information.

v1.1 Mon Jul 2 2018

Removed the requirement on the GCC version (`>= 5.0`).

v1.0 Sun Jul 1 2018

The v1.0 release of libaco, cheers   

Donation

I'm a full-time open source developer. Any amount of the donations will be highly appreciated and could bring me great encouragement.

- Paypal

[paypal.me link](https://paypal.me/hnes)

- Alipay (支付(宝|寶))



- Wechat (微信)



Copyright and License

Copyright (C) 2018, by Sen Han 00hnes@gmail.com.

Under the Apache License, Version 2.0.

See the [LICENSE](#) file for details.