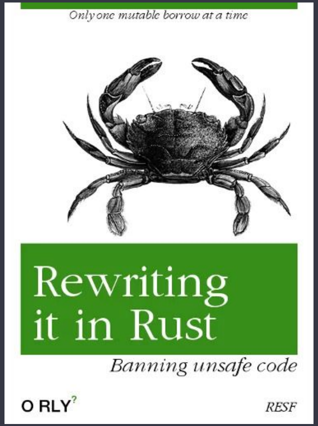
Coroutine Intro with Rust

Zero-cost abstraction of async framework

- By Amanda Tian

Rust Language is ...

- a general purpose language for both system programming and applications, originated from Mozilla in 2006
- **high performance** and memory efficient without runtime or GC
- **Memory and thread safety** with type system and ownership model at compile time
- *Linux 6.1 officially adds support for Rust in kernel*

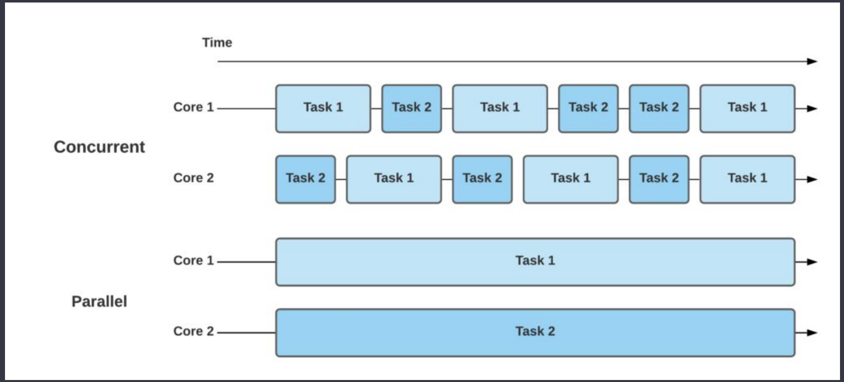


Agenda

- Asynchronous programming basics
- Rust async mechanism
- Runtime & task scheduling
- About stackful coroutine

Part 1
Asynchronous programming basics

Concurrency vs. Parallelism



figure_src

Parallel: **doing** a lot of things at the same time

Concurrency: **dealing** a lot of things at the same time

Asynchronous: describe a language feature to enable concurrent or parallel programming

I/O types - synchronous blocking

Traditional OS threads with one thread per task blocks:

```rust
fn read_parallel() {
    let jh_1 = thread::spawn(|| read("file_A"));
    let jh_2 = thread::spawn(|| read("file_B"));

    jh_1.join();
    jh_2.join();
}
```

Pros:

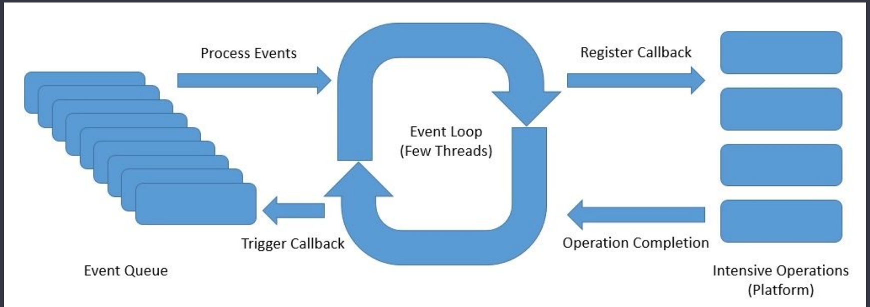
- Simple, straightforward logic
- Free use with kernel's management

Cons:

- Limited number of tasks with large stack mem
- Context switch is bottleneck in high concurrency

I/O types - asynchronous non-blocking

Event driven + I/O multiplexing



figure_src

```
// callback
setTimer(200, () => {
    setTimer(100, () => {
        setTimer(50, () => {
            console.log("I'm the last one");
        });
    });
});


// promise
function timer(ms) {
    return new Promise((resolve) => setTimeout(resolve, ms));
}

timer(200)
.then(() => timer(100))
.then(() => timer(50))
.then(console.log("I'm the last one");
```

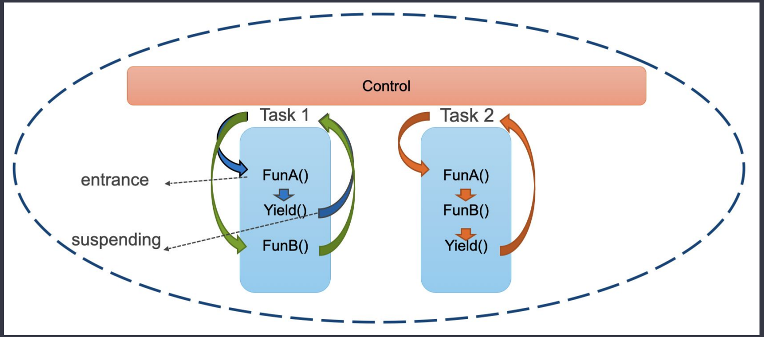
Pros:

- No context switch, relative low mem cost

Cons:

- Callback hell, nested callback chains hard to maintain and understand

I/O types - synchronous non-blocking




```rust
async fn read_parallel_async() {
    let fut_1 = read_file_async("file_A");
    let fut_2 = read_file_async("file_B");

    fut_1.await;
    fut_2.await;
}
```

Coroutines:

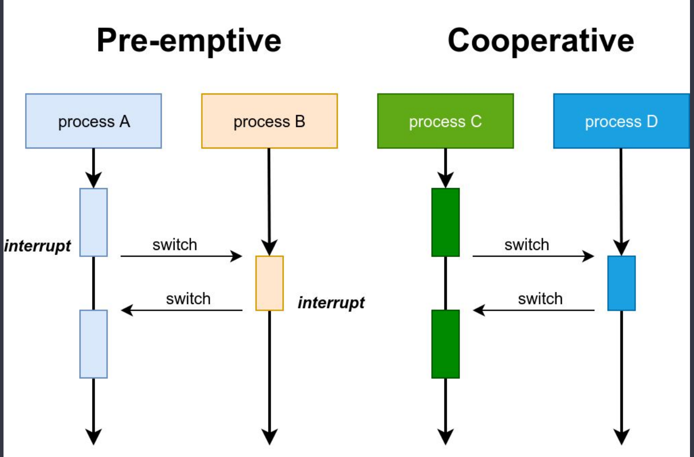
- Pausable cooperative multitask able to yield and resume
- Write async code just in synchronous manner

“Coroutines are computer program components that generalize subroutines for non-preemptive multitasking, by allowing execution to be suspended and resumed.”

- Wikipedia

Variant of functions enable concurrency via cooperative multitasking

Preemptive vs. Cooperative Multitasking



Preemptive:
system forcibly suspend running task and switch to another

Cooperative:
task voluntarily yield control periodically or idle or blocked

Part 2
Rust async mechanism

Rust async mechanism - overview

```rust
fn main() {
    let sum_fut = sum();
    let jh = tokio::spawn(sum_fut);
    block_on(jh);
}

#[inline(never)]
fn get_val() -> impl Future<Output::usize> {
    // do the task asynchronously
    async {1}
}

async fn sum() -> usize {
    get_val().await + 1
}
```

Rust compiler:

- **async** keyword, **.await** syntax

Rust std:

- Basic **Future** trait for pausable task
- **Waker** type to wake up a task

Rust async runtime

Rust async mechanism - async/await

```
// HIR (High-level intermediate representation)
// cargo rustc -- -Z unpretty=hir
#[inline(never)]
async fn get_val()
    ->
    /*impl Trait*/ #[lang = "from_generator"](move |mut _task_context|
    { { let _t = { 1 }; _t } })


async fn sum()
    ->
    /*impl Trait*/ #[lang = "from_generator"](move |mut _task_context|
    {
        {
            let _t =
                {
                    match #[lang = "into_future"](get_val()) {
                        mut pinned =>
                            loop {
                                match unsafe {
                                    #[lang = "poll"](#[lang = "new_unchecked"](&mut pinned),
                                        #[lang = "get_context"](_task_context))
                                } {
                                    #[lang = "Ready"] { 0: result } => break result,
                                    #[lang = "Pending"] {} => { }
                                }
                                _task_context = (yield ());
                            },
                    } + 1
                };
            _t
        }
    })
```

std::ops::generator example:

```
fn main() {
    let generator = || {
        let mut val = 1;
        yield val;
        val += 1;
        yield val;
        val += 1;
        return val;
    };

    assert_eq!(generator.resume(), GeneratorState::Yielded(1));
    assert_eq!(generator.resume(), GeneratorState::Yielded(2));
    assert_eq!(generator.resume(), GeneratorState::Complete(3));
}
```

async/await => generator

Rust async mechanism - async/await

```
// MIR (mid-level intermediate representation)
// cargo rustc -- -Z unpretty=mir
fn sum::{closure#0}(_1: Pin<&mut [static generator@src/main.rs:16:25: 18:2]>, _2: ResumeTy) -> GeneratorState<(), usize> {
    debug _task_context => _19;          // in scope 0 at src/main.rs:16:25: 18:2
    let mut _0: std::ops::GeneratorState<(), usize>; // return place in scope 0 at src/main.rs:16:25: 18:2
    let mut _3: usize;                   // in scope 0 at src/main.rs:17:5: 17:20
    let mut _4: impl std::future::Future<Output = usize>; // in scope 0 at src/main.rs:17:14: 17:20
    let mut _5: impl std::future::Future<Output = usize>; // in scope 0 at src/main.rs:17:5: 17:14
    let mut _6: std::task::Poll<usize>;  // in scope 0 at src/main.rs:17:14: 17:20
    let mut _7: std::pin::Pin<&mut impl std::future::Future<Output = usize>>; // in scope 0 at src/main.rs:17:14: 17:20
    let mut _8: &mut impl std::future::Future<Output = usize>; // in scope 0 at src/main.rs:17:14: 17:20
    let mut _9: &mut impl std::future::Future<Output = usize>; // in scope 0 at src/main.rs:17:14: 17:20
    let mut _10: &mut std::task::Context; // in scope 0 at src/main.rs:17:5: 17:20
    let mut _11: &mut std::task::Context; // in scope 0 at src/main.rs:17:5: 17:20
    let mut _12: std::future::ResumeTy;  // in scope 0 at src/main.rs:17:14: 17:20
    let mut _13: isize;                  // in scope 0 at src/main.rs:17:14: 17:20
    let mut _15: std::future::ResumeTy;  // in scope 0 at src/main.rs:17:14: 17:20
    let mut _16: ();                     // in scope 0 at src/main.rs:17:14: 17:20
    let mut _17: (usize, bool);          // in scope 0 at src/main.rs:17:5: 17:24
    let mut _18: usize;                  // in scope 0 at src/main.rs:16:25: 18:2
    let mut _19: std::future::ResumeTy;  // in scope 0 at src/main.rs:16:25: 18:2
    let mut _20: u32;                    // in scope 0 at src/main.rs:16:25: 18:2
    scope 1 {
        debug pinned => (((*(_1.0: &mut [static generator@src/main.rs:16:25: 18:2])) as variant#3).0: impl std::future::Future<Output = usize>);
        let _14: usize;                  // in scope 1 at src/main.rs:17:5: 17:20
```

async/await => generator => **statemachine** => impl Future

Rust async mechanism - Future trait

A future represents an asynchronous computation obtained by use of async.

```
pub trait Future {
    /// The type of value produced on completion.
    type Output;

    /// Attempt to resolve the future to a final value, registering
    /// the current task for wakeup if the value is not yet available.

    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output>;
}

pub enum Poll<T> {
    /// Represents that a value is immediately ready.
    Ready(T),
    /// represents a value is not ready yet
    Pending,
}
```

Future exposes **poll** method:

- Called by future to drive task execution
- Return Pending when blocked
- Return Ready with output when finished

Poll method defines the statemachine of Future

Rust async mechanism - impl a Future

```rust
#[inline(never)]
fn get_val() -> impl Future<Output::usize> {
    // do the task asynchronously
    async {1}
}

async fn sum() -> SumFuture {
    SumFuture::Initiate
}

enum SumFuture {
    Initiate,
    GetValFirst(GetValFut), // <-- every .await needs a state
    GetValSecond(GetValFut, usize),
    Ready(usize)
}

impl Future for SumFuture {
    type Output = usize;

    fn poll(self: &mut Self, cx: &mut Context) -> Poll<()> {
        let this = self.get_mut();
        loop {   //  <-- drive task state as far as possible
            match this {
                Initiate => {
                    let get_val_fut = get_val();
                    *this = SumFuture::GetValFirst(get_val_fut);
                },
                GetValFirst(fut) => {
                    let pinned = unsafe {Pin::new(unchecked(fut))};
                    match pinned.poll(cx) {
                        Poll::Pending => return Poll::Pending,
                        Poll::Ready(val) => *this = GetValSecond(get_val(), val)
                    }
                },
                GetValSecond(fut, last_val) => {
                    let pinned = unsafe {Pin::new(unchecked(fut))};
                    match pinned.poll(cx) {
                        Poll::Pending => return Poll::Pending,
                        Poll::Ready(val) => *this = Poll::Ready(val + last_val + 1)
                    }
                },
                Ready(val) => {
                    return Poll::Ready(val);
                }
            }
        }
    }
}
```

Futures implementation:

- Leaf future (I/O resource) usually by runtime
- Non-leaf future generated by compiler via async

Memory optimization:

- Zero-cost abstraction allow no heap allocation or dynamic dispatch
- Reuse memory for non-overlap variables

Rust async mechanism - Waker

```rust
pub struct RawWaker {
    data: *const (),
    /// Virtual function pointer table with customized behavior.
    vtable: &'static RawWakerVTable,
}

pub struct Waker {
    waker: RawWaker,
}

impl Waker {
    /// Wake up the task related to this `Waker`.
    pub fn wake(self) {
        let wake = self.waker.vtable.wake;
        let data = self.waker.data;

        unsafe { (wake)(data) };
    }
}

pub struct Context<'a> {
    waker: &'a Waker,
    // marker field could be ignored
    _marker: PhantomData<fn(&'a ()) -> &'a ()>,
}
```

Waker:

- std defined interface to wake up a suspended task when related I/O event ready
- Runtime creates and defines data, vtable, i.e. HOW to wake a task up
- Passed around wrapped in a Context

Part 3
Runtime and task scheduling

Runtime overview

What's runtime?

- Rust std provides minimal primitive: Future trait, async/await for pausable async tasks
- Runtime act as execution context to drive the futures to completion

What runtime consists?

- **Executor**: a task scheduler usually with task queue
- **Reactor**: I/O driver backed by system event queue (mio crate over epoll/kqueue/IOCP)
- **I/O components:** non-blocking APIs interact with Reactor

Rust community runtimes crates:

Runtime	All-time downloads (July 2022)	Description
tokio	59,048,636	An event-driven, non-blocking I/O platform for writing asynchronous I/O backed applications.
async-std	8,002,852	Async version of the Rust standard library
smol	1,491,204	A small and fast async runtime

src

Tokio interfaces

- **#[tokio_main]:** annotate the main function as async
- **block_on**: runtime's entry point, runs a future to completion
- **tokio::spawn:** spawn new future as **Task**s, executed by runtime concurrently
- **JoinHandle**: handle to spawned task to retrieve output on Task finish
- **tokio::spawn_blocking:** runs blocking functions on executor, usually on a separate thread pool from non-blocking tasks
- **tokio::block_in_place**: spawn blocking task and turn current thread to blocking thread, move existing tasks to other worker threads

Task scheduling (single thread)

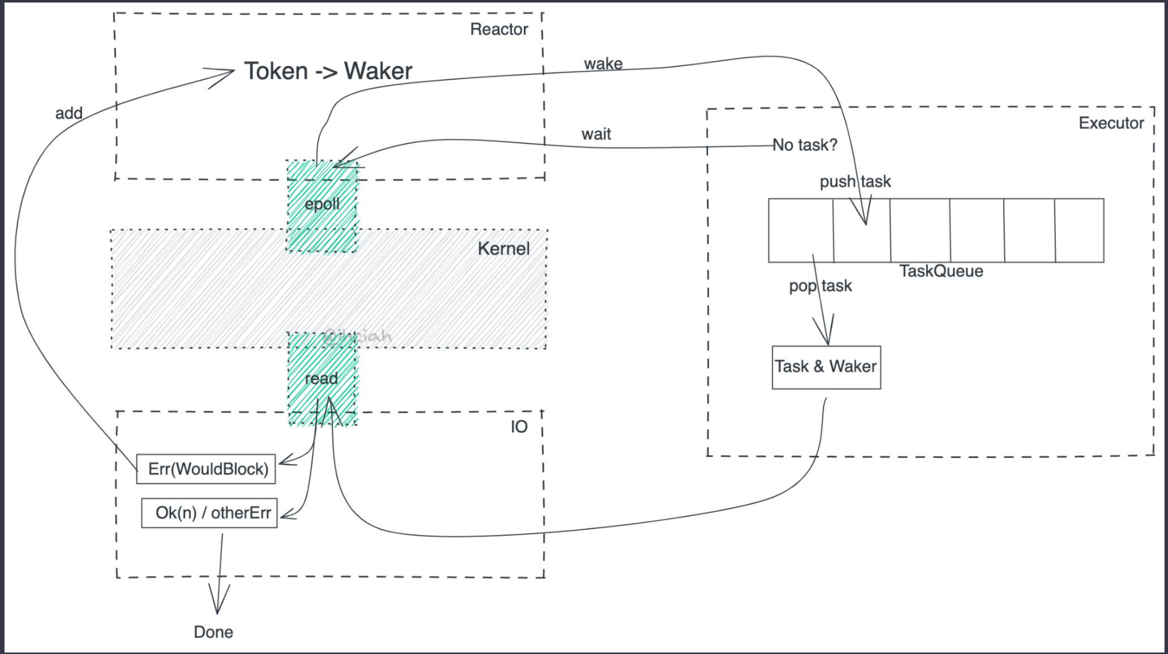


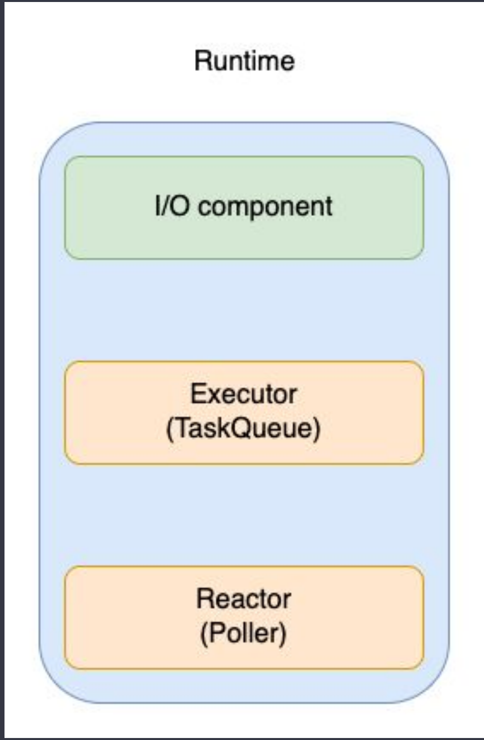
figure src

Executor and reactor form an event-loop, loosely decoupled by Future and Waker

Waker in tokio:

- A reference to the task itself
- Wake pushes task to the queue

Task scheduling (single thread)



I/O component:

1. Provides non-blocking API
2. Register I/O event fd to reactor, with correlated Waker

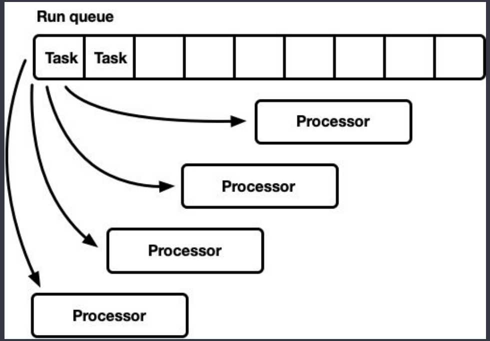
Executor:

1. Poll each task on queue as far as possible
2. Give control to reactor when idle

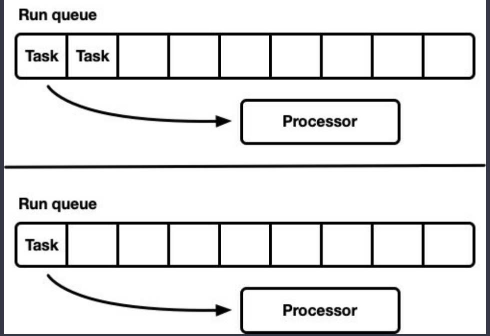
Reactor (underlying Mio - metal I/O):

1. Waiting for I/O event blockingly
2. Wake up the task with event ready
3. Give control back to executor

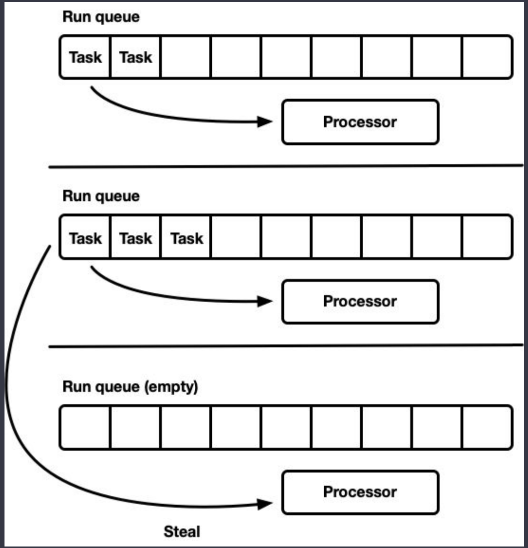
Task scheduling (multi-thread)



Global queue



Separate local queue



Work-stealing model

ref

Handling multiple tasks - join

tokio::join!():
Waits on multiple **concurrent** branches, returning when all branches complete.

```
#[tokio::main]
async fn main() {
    let (tx1, rx1) = oneshot::Receiver::<u32>();
    let (tx2, rx2) = oneshot::Receiver::<u32>();

    tokio::join!(rx1.recv(), rx2.recv());
}

pub struct JoinTask {
    rx1: oneshot::Receiver<u32>,
    rx2: oneshot::Receiver<u32>,
    output: [u32; 2];
}


impl Future for JoinTask {
    type Output = [u32; 2];

    fn poll(mut self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<()> {
        let mut is_pending = false;

        match Pin::new(&mut self.rx1).poll(cx) {
            Poll::Ready(val) => output[0] = val,
            Poll::Pending() => is_pending = true
        }

        match Pin::new(&mut self.rx2).poll(cx) {
            Poll::Ready(val) => output[1] = val,
            Poll::Pending() => is_pending = true
        }

        if is_pending {
            return Poll::Pending();
        }

        Poll::Ready(self.output)
    }
}
```

Handling multiple tasks - join

tokio::select!():
Waits on multiple **concurrent** branches, returning when the first branch completes, **cancelling** the remaining branches.

```rust
#[tokio::main]
async fn main() {
    let (tx1, rx1) = oneshot::Receiver::<u32>();
    let (tx2, rx2) = oneshot::Receiver::<u32>();

    tokio::select!{
        _ = rx1.recv() => {},
        _ = rx2.recv() => {}
    };
}

pub struct SelectTask {
    rx1: oneshot::Receiver<u32>,
    rx2: oneshot::Receiver<u32>,
}


impl Future for SelectTask {
    type Output = u32;

    fn poll(mut self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<()> {
        if let Poll::Ready(val) = Pin::new(&mut self.rx1).poll(cx) {
            println!("rx1 completed first with {:?}", val);
            return Poll::Ready(val);
        }

        if let Poll::Ready(val) = Pin::new(&mut self.rx2).poll(cx) {
            println!("rx2 completed first with {:?}", val);
            return Poll::Ready(val);
        }

        Poll::Pending
    }
}
```

Part 4
About stackful coroutine

More about coroutines...

Coroutine classification:

- asymmetric vs. symmetric
- first-class object vs. constrained construct
- stackful vs. stackless

Stackful coroutine:

- Future state stored as call stack, allocated on heap
- Task switched by context switch
- Also known as fibers, green threads, e.g. Goroutine

Hack with context switch

```rust
struct ThreadContext {
    rsp: u64,
}

fn main() {
    let mut ctx = ThreadContext::default();
    let mut stack = vec![0_u8; SSIZE as usize];

    unsafe {
        let stack_bottom = stack.as_mut_ptr().offset(SSIZE);
        let sb_aligned = (stack_bottom as usize & !15) as *mut u8;
        std::ptr::write(sb_aligned.offset(-16) as *mut u64, hello as u64);
        ctx.rsp = sb_aligned.offset(-16) as u64;
        gt_switch(&mut ctx);
    }
}

fn hello() -> ! {
    println!("I LOVE WAKING UP ON A NEW STACK!");
}

unsafe fn gt_switch(new: *const ThreadContext) {
    asm!(
        "mov rsp, [{0} + 0x00]",
        "ret",
        in(reg) new,
    );
}
```

Figure 3.3: Stack Frame with Base Pointer

Position	Contents	Frame
8n+16(%rbp)	memory argument eightbyte n	Previous
	. . .	
16(%rbp)	memory argument eightbyte 0	
8(%rbp)	return address	Current
0(%rbp)	previous %rbp value	
-8(%rbp)	unspecified	
	. . .	
0(%rsp)	variable size	
-128(%rsp)	red zone	

Stackful coroutine - a toy

```rust
struct Thread {
    id: usize,
    stack: Vec<u8>,
    ctx: ThreadContext,
    state: State,
}

// Registers %rbx,
// %rbp, and %r12-r15 are callee-save registers, meaning that they are saved across function
// calls. Register %rsp is used as the stack pointer, a pointer to the topmost element in the stack.
struct ThreadContext {
    rsp: u64,
    r15: u64,
    r14: u64,
    r13: u64,
    r12: u64,
    rbx: u64,
    rbp: u64
}

impl Thread {
    pub fn new(id: usize) -> Self {
        Thread {
            id,
            stack: vec![0_u8, DEFAULT_STACK_SIZE as u8],
            ctx: ThreadContext::default(),
            state: State::Available
        }
    }
}
```

Thread:

- abstraction of coroutine holds its stack and context with register values

Stackful coroutine - a toy

```rust
pub struct Runtime {
    threads: Vec<Thread>,
    current: usize,
}

impl Runtime {
    pub fn run(&mut self) -> !{
        while self.t_yield() {}
        std::process::exit(0);
    }

    fn t_yield(&mut self) -> bool {
        let mut pos = self.current;
        while self.threads[pos].state != State::Ready {
            pos += 1;

            if pos == self.threads.len() {pos = 0;}

            if pos == self.current {return false;}
        }

        if self.threads[self.current].state != State::Available {
            self.threads[self.current].state = State::Ready;
        }

        self.threads[pos].state = State::Running;
        let old_pos = self.current;
        self.current = pos;

        unsafe  {
            let old: *mut ThreadContext = &mut self.threads[old_pos].ctx;
            let new: *mut ThreadContext = &mut self.threads[pos].ctx;
            asm!("call switch", in("rdi") old, in("rsi") new, clobber_abi("C"));
        }

        self.threads.len() > 0
    }

    pub fn spawn(&mut self, f: fn()) {
        let available = self
            .threads
            .iter_mut()
            .find(|t| t.state == State::Available)
            .expect("no available thread.");

        let size = available.stack.len();
        unsafe {
            let s_ptr = available.stack.as_mut_ptr().offset(size as isize);
            let s_ptr = (s_ptr as usize & !15) as *mut u8;
            std::ptr::write(s_ptr.offset(-16) as *mut u64, guard as u64);
            std::ptr::write(s_ptr.offset(-24) as *mut u64, skip as u64);
```

Runtime:

- API to spawn new threads
- main loop to trigger the execution of threads
- perform context switch when a thread is not Ready

Stackful coroutine - a toy

```
#[naked]
#[no_mangle]
unsafe extern "C" fn switch() {
    asm!(
    "mov [rdi + 0x00], rsp",
    "mov [rdi + 0x08], r15",
    "mov [rdi + 0x10], r14",
    "mov [rdi + 0x18], r13",
    "mov [rdi + 0x20], r12",
    "mov [rdi + 0x28], rbx",
    "mov [rdi + 0x30], rbp",
    "mov rsp, [rsi + 0x00]",
    "mov r15, [rsi + 0x08]",
    "mov r14, [rsi + 0x10]",
    "mov r13, [rsi + 0x18]",
    "mov r12, [rsi + 0x20]",
    "mov rbx, [rsi + 0x28]",
    "mov rbp, [rsi + 0x30]",
    "ret", options(noreturn)
    );
}
```

Context switch:

- Store the current registers to rdi (old thread)
- Load from rsi (new thread) to current registers

Stackful vs. stackless coroutine

Stackless coroutine:

- Lightweight with zero-cost abstraction backed with state machine
- No context switch on task scheduling

Stackful coroutine:

- With call stack stored, capable to yield at any time
- Allow preemptive scheduling on bad actors
- Memory cost on stack growth could be an issue

Last but not least...

Coroutine is powerful, but not suitable in any situations.

Good for:

- Obviously web servers
- UI (wait for user response while doing background tasks)
- Filesystems
- ...

Not best choice:

- CPU intensive computations
- Long running tasks without yielding

Q & A