# Build an Async Runtime for Rust from Scratch

Qi-An Fu
fugoes.qa@gmail.com
https://blog.fugoes.xyz

Tunight at May 9, 2020

# Outline

# Outline

# Long long time ago...

Serve one client with each server process (or thread)

## tcp_server_fork.py

```python
#!/usr/bin/env python3
import os, socket
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
server.bind(('127.0.0.1', 8080)); server.listen(1)
count = 0
while True:
    client, address = server.accept()
    count += 1
    print(f'accepted {address}')
    if os.fork() == 0:
        if client.recv(6) == b'yoo!\r\n':
            client.send(f'yoo {count}!\r\n'.encode())
        client.close(); break
    else:
        client.close()
```

# Long long time ago... (cont'd)

Serve one client with each server process or thread

- Cannot handle large number of clients simultaneously.
  - Stack size for each process (or thread) is 8 MiB.

```
> ulimit -s
8192
```

  - Context switch between processes (or threads) is expensive.

# Long time ago...

Serve many clients with each server process or thread

`tcp_server_select.py`

```
#!/usr/bin/env python3
import os, socket, select
handlers = dict()
count = 0
def accept_handler(x):
    client, address = x.accept()
    global count
    count += 1
    print(f'accepted {address}')
    handlers[client] = client_handler_wrapper(count)
def client_handler_wrapper(c):
    def client_handler(x):
        if x.recv(6) == b'yoo!\r\n':
            x.send(f'yoo {c}!\r\n'.encode())
        x.close(); handlers.pop(x)
    return client_handler
```

# Long time ago... (cont'd)

Serve many clients with each server process or thread

```
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
server.bind(('127.0.0.1', 8080)); server.listen(1)

handlers[server] = accept_handler

while True:
    events, _, _ = select.select(list(handlers.keys()), [], [])
    for x in events:
        handlers[x](x)
```

# Long time ago... (cont'd)

Serve many clients with each server process or thread

- `select()` could watch at most `FD_SETSIZE` handles.

```
int select(int nfds, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, struct timeval *timeout);
```

  - Events are returned in `readfds`, `writefds`, and `exceptfds` directly.

- `poll()`: A `select()` without the `FD_SETSIZE` limitation.

```
int poll(struct pollfd *fds, nfds_t nfds, int timeout);
struct pollfd {
    int   fd;         /* file descriptor */
    short events;     /* requested events */
    short revents;    /* returned events */
};
```

  - Time complexity grows linearly with number of file descriptors to watch.

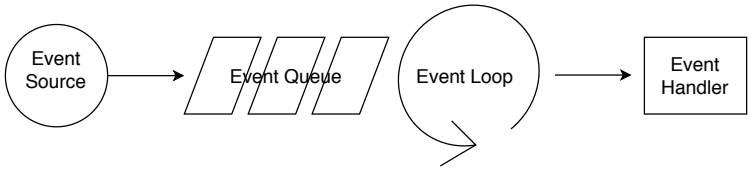
# Long time ago... (cont'd)

Serve many clients with each server process or thread

- epoll: A `poll()` with better time complexity (Linux only).

```
int epoll_create(int size);
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
// op = { EPOLL_CTL_ADD | EPOLL_CTL_MOD | EPOLL_CTL_DEL }
int epoll_wait(int epfd, struct epoll_event *events, int maxevents, int timeout);
```

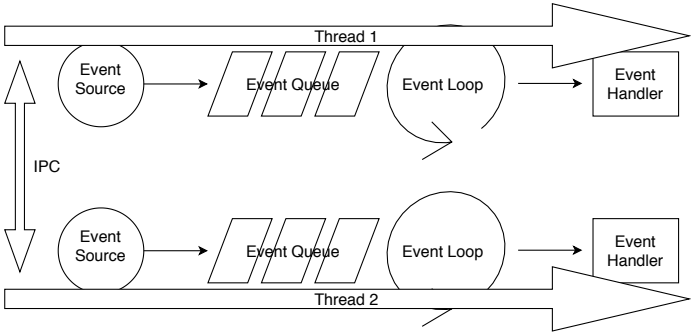
# Event loop




- Event source generates new events, usually it uses some OS event notification mechanism (e.g. `epoll`). New events are pushed into the event queue.
- Event loop keeps popping events from the event queue, and calling their event handlers.
- Event handlers could do IO and calculations, and register new event handlers (a.k.a callbacks).

# Event loop goes multi-threaded

Separate event source, event queue, and event loop for each thread




IPC mechanism are provided as part of the event loop library to notify other threads' event loops.

# Event loop goes multi-threaded (cont'd)

Separate event source, event queue, and event loop for each thread

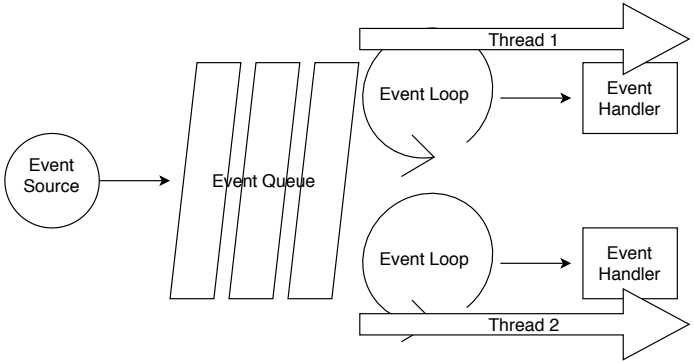
- Library
  - `libev`
    - Reactor library in `C` language.
    - Provide `ev_async` to wake up other threads' event loops.
  - `libuv`
    - Proactor library in `C` language.
    - Provide `uv_async` to wake up other threads' event loops.
- Work imbalance problem between threads.

# Reactor & Proactor

- Reactor
  - Event notification mechanism provided by the OS tells when a file descriptor (e.g. socket, pipe) has events (could read, could write, or exception).
  - The event handler does the IO task.
- Proactor
  - Event notification mechanism provided by the OS does the IO task.
  - When the event handler register new event handlers, it needs to provide IO buffer.
  - Cancellation is hard to implement correctly.
- *nix OS usually provides reactor mechanism, while Windows OS provides proactor mechanism (`IOCP`).
  - Cross-platform library usually follows the proactor model for implementation efficiency and simplicity.

# Event loop goes multi-threaded (cont'd)

Shared event source and event queue, separate event loop for each thread



# Event loop goes multi-threaded (cont'd)

Shared event source and event queue, separate event loop for each thread

- Library
  - `Asio`
    - Proactor library in `C++` language.
    - Run `io_context::run()` for one `io_context` instance in multiple threads.
    - To use threads without explicit locking, use `strand<>`.
- Work is balanced between threads.
  - But introduce overheads for sharing event queue.

# But callback sucks

- Callback hell.
- Code for handling same client is split into different callback functions.
  - One thread (or process) for each client and using blocking IO ease the programmer's life.
- Can we write async programs in blocking style?
  - Yes! We have coroutine.

# Outline

# Not so long time ago...

- Put "operating system" scheduler in user space.
  - Coroutine is just like "threads".
  - When a coroutine doing "system call", instead of trapped into the kernel, control is given back to the user space scheduler.
  - The scheduler would receive events from the event source, and give control back to the coroutine (event loop).
- Coroutine is just cooperative "threads".
- Coroutine has many names (in programming language):
  - Resumable function (`C++17`).
  - Generator (`Python`, `Rust`).

# How to speak generator in Python

https://www.python.org/dev/peps/pep-0255/

```python
def fibonacci(n):
    a, b = 1, 1
    if n == 0: return a
    yield a
    if n == 1: return b
    yield b
    n -= 2
    while True:
        a, b = b, a + b
        if n == 0: return b
        n -= 1
        yield b
g = fibonacci(10)
try:
    for i in range(10 + 1): print(f'fibonacci[{i}] = {next(g)}')
except StopIteration as e:
    print(e.value)
```

# How to speak generator in Python (cont'd)

tcp_server_simple_generator.py

```
#!/usr/bin/env python3
import os, socket, select

handlers = dict()

def server_coroutine(server):
    this = yield

    count = 0
    while True:
        handlers[server] = this
        yield
        client, address = server.accept()
        print(f'accepted {address}')
        count += 1
        g = client_coroutine(client, count); next(g); g.send(g)
```

# How to speak generator in Python (cont'd)

```
def client_coroutine(client, count):
    this = yield

    handlers[client] = this
    yield
    if client.recv(6) == b'yoo!\r\n':
        client.send(f'yoo {count}!\r\n'.encode())
    client.close()
    handlers.pop(client)
```

# How to speak generator in Python (cont'd)

```
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
server.bind(('127.0.0.1', 8080)); server.listen(1)

g_server = server_coroutine(server); next(g_server); g_server.send(g_server)

while True:
    events, _, _ = select.select(list(handlers.keys()), [], [])
    for x in events:
        try:
            next(handlers[x])
        except StopIteration:
            pass
```

# How to speak generator in Python (cont'd)

How to combine generators?
https://www.python.org/dev/peps/pep-0380/

```
r = yield from g
```

is (approximately) equivalent to

```
try:
    for v in g:
        yield v
except StopIteration as e:
    r = e.value
```

# How to speak generator in Python (cont'd)

```python
def wrap(g):
    r = yield from g
    print(r)
    return r

def fibonacci(n):
    a, b = 1, 1
    if n == 0: return a
    yield a
    if n == 1: return b
    yield b
    n -= 2
    while True:
        a, b = b, a + b
        if n == 0: return b
        n -= 1
        yield b

print(list(wrap(fibonacci(10))))
```

# How to speak generator in Python (cont'd)

tcp_server_yield_from.py

```
#!/usr/bin/env python3
import os, socket, select

handlers = dict()

def accept(sock, g):
    handlers[sock] = g
    yield
    handlers.pop(sock)
    return sock.accept()

def recv(sock, n, g):
    handlers[sock] = g
    yield
    handlers.pop(sock)
    return sock.recv(n)
```

# How to speak generator in Python (cont'd)

```python
def recv_exact(sock, n, g):
    r = b''
    while n > 0:
        buf = yield from recv(sock, n, g)
        n -= len(buf)
        r += buf
    return r
```

# How to speak generator in Python (cont'd)

```
def spawn(f, *args, **kwargs):
    g = f(*args, **kwargs)
    next(g)
    g.send(g)

def client_coroutine(client, count):
    this = yield

    msg = yield from recv_exact(client, 6, this)
    if msg == b'yoo!\r\n':
        client.send(f'yoo {count}!\r\n'.encode())
    client.close()
```

# How to speak generator in Python (cont'd)

```
def server_coroutine():
    server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    server.bind(('127.0.0.1', 8080)); server.listen(1)
    this = yield

    count = 0
    while True:
        client, address = yield from accept(server, this)
        print(f'accepted {address}')
        count += 1
        spawn(client_coroutine, client, count)
```

# How to speak generator in Python (cont'd)

```
def block_on(f, *args, **kwargs):
    spawn(f, *args, **kwargs)
    while True:
        events, _, _ = select.select(list(handlers.keys()), [], [])
        for x in events:
            try:
                next(handlers[x])
            except StopIteration:
                pass

block_on(server_coroutine)
```

# How to speak generator in Python (cont'd)

Define a generator as a class by "state machine".

```python
class Fibonacci(object):
    def __init__(self, n):
        self.n, self.i = n, 0
        self.a, self.b = 1, 1

    def __next__(self):
        self.i += 1
        if self.n == self.i - 1: raise StopIteration(self.b)
        if self.i == 1: return self.a
        self.a, self.b = self.b, self.a + self.b
        return self.a

g = Fibonacci(10)
try:
    for i in range(10 + 1): print(f'fibonacci[{i}] = {next(g)}')
except StopIteration as e:
    print(e.value)
```

# Rust's Future API

- raise StopIteration(x) ⇒ Poll::Ready(x), return ⇒ Poll::Pending.
- __next__() ⇒ poll().

```rust
pub trait Future {
    type Output;
    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output>;
}

pub enum Poll<T> { Ready(T), Pending, }

impl<'a> Context<'a> {
    pub fn waker(&self) -> &'a Waker { ... }
}

impl Waker {
    pub fn wake(self) { ... }
}
```

# Rust's Future API (cont'd)

In Python:

```python
def recv(sock, n, g):
    handlers[sock] = g
    yield
    handlers.pop(sock)
    return sock.recv(n)
```

In Rust:

```rust
struct RecvFuture<'a>(bool, usize, &'a SomeSocket);

impl<'a> Future for RecvFuture<'a> {
    type Output = Box<[u8]>;
    fn poll(mut self: Pin<&mut Self>,
            cx: &mut Context<'_>) -> Poll<Self::Output> {
        if self.0 {
            self.0 = false;
            let g = cx.waker();
            register(self.2, g.clone());
            Poll::Pending
        } else {
            deregister(self.2);
            Poll::Ready(self.2.recv(self.1))
        }
    }
}
```

# Rust's Future API (cont'd)

In Python:

```python
def recv_exact(sock, n, g):
    r = b''
    while n > 0:
        buf = yield from recv(sock, n, g)
        n -= len(buf)
        r += buf
    return r
```

In Rust:

```rust
async fn recv_exact(sock: &SomeSocket, n: usize)
                    -> Box<[u8]> {
    let mut n = n;
    let mut r = Vec::new();
    while n > 0 {
        let buf = RecvFuture(true, n, sock).await;
        n -= buf.len();
        r.append(&mut buf.into_vec());
    }
    r.into_boxed_slice()
}
```

- `fn recv_exact()` returns a `Future<Output=Box<[u8]>>`.

# Rust's Future API (cont'd)

In Python:

```python
def spawn(f, *args, **kwargs):
    g = f(*args, **kwargs)
    next(g)
    g.send(g)
```

In Rust:

```rust
fn spawn(coroutine: Future<Output=()>) {
    unimplemented!();
}
```

# Rust's Future API (cont'd)

- Event notification mechanism.
  - Receive events from OS, and call `wake()` for their `Waker`.
  - Provide API for registering and deregistering events with `Waker`.
- `Future` implementation.
  - Do IO.
  - Interact with the event notification mechanism if IO is not ready.
  - Could be combined with the `.await` syntax.
- Executor.
  - Provide the `Waker`'s implementation, so that when `Waker::wake()` is invoked, schedule the coroutine associated with the `Waker` (e.g. push it into a task queue).
  - Provide API for creating and running a coroutine (`fn spawn(future: Future<()>)`).
  - Fetch coroutine from the task queue, and invoke its `poll()` method.

# Outline

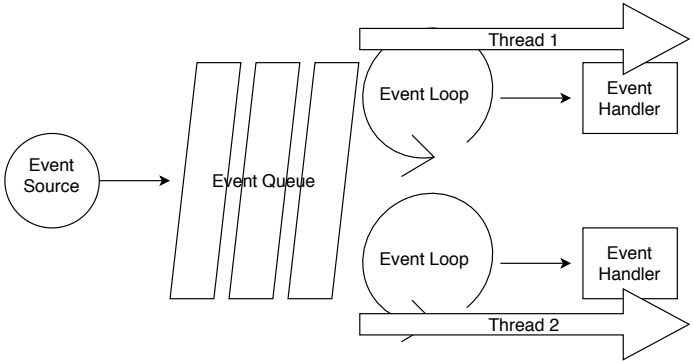
# Overview

https://github.com/Fugoes/yaay-rs/

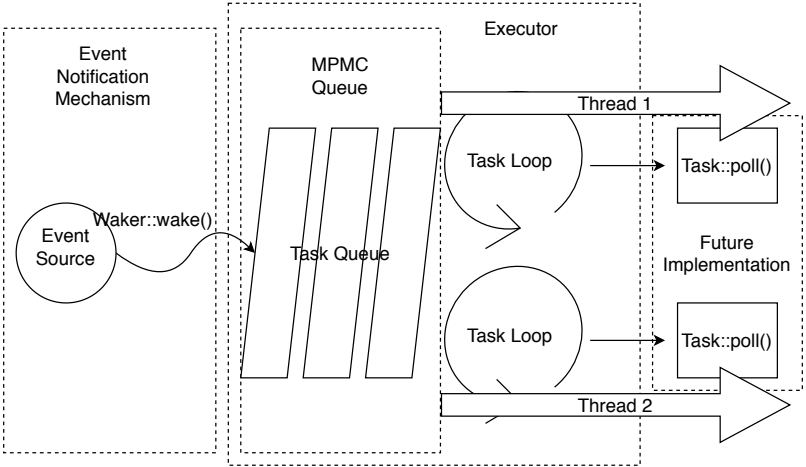
- Focus on implementation of executor (the funniest part).
  - We need a struct to wrap a `Future<Output=()>` type as a top level task ("coroutine"), and do dynamic dispatch to the `poll()` method.
    - ★ `Task` in `yaay-mt-runtime/src/task.rs`
  - We need a fast & scalable MPMC (multiple-producer multiple-consumer) queue to schedule these tasks.
  - Start some threads, each thread keeps popping tasks from the queue and `poll()` these tasks.
- `Future` implementation and event notification mechanism implementation are trivial.
  - `yaay-mio/`

# Event loop goes multi-threaded (revisited)

Shared event source and event queue, separate event loop for each thread



# Multi-threading async runtime



# Future<Output=()> wrapper

- How does Rust do dynamic dispatch?
  - Fat pointer.

```rust
use std::mem::size_of;
trait A { fn a(&self); }
fn main() {
    dbg!(size_of::<&dyn A>());
    dbg!(size_of::<*mut dyn A>());
    dbg!(size_of::<*mut ()>());
}
```

```rust
struct TraitObjectRef {
    data: *const DataType,
    vtable: *const (),
}
```

- How does C++ do dynamic dispatch?

```rust
struct Object {
    void *vtable;
    DataType data;
}
```

# Future<Output=()> wrapper (cont'd)

```rust
struct Task {
    vtable: &'static TaskVTable,
    rc: AtomicUsize,
    ...,
}

#[repr(C)]
struct TaskInner<T>
    where T: Future<Output=()> + Send {
    task: Task,
    future: T,
}
```

```rust
struct TaskVTable {
    fn_poll: unsafe fn(NonNull<Task>,
                       &mut Context) -> Poll<()>,
    fn_drop_in_place: unsafe fn(NonNull<Task>),
    fn_dealloc: unsafe fn(NonNull<Task>),
}
```

- `*mut TaskInner<T>` is not fat pointer, and could be cast to `*mut Task` directly.

# `Future<Output=()>` wrapper (cont'd)

`Task`'s safety guarantee:

- One task shouldn't be polled concurrently by multiple worker threads.
- After a task's `wake()` method is invoked, eventually the task would be polled again.
- When a task is dropped (after polled to ready, or after the runtime is shutdown), the task's `wake()` method should still be safe to invoke.

# Future<Output=()> wrapper (cont'd)

```
struct Task {
    ...,
    status: AtomicU8,
    ...,
}
impl Task {
    const MUTED: u8 = 1;
    const NOTIFIED: u8 = Self::MUTED << 1;
    const DROPPED: u8 = Self::NOTIFIED << 1;
    ...;
}
```

# MPMC queue

## Mutex protected single linked list

```
struct Task {
    ...,
    next: *mut Task,
    ...,
}
```

- No memory allocation when operating the linked list.

# Why mutex? Why not go lock-free?

- Lock-free queues are hard to implement correctly.
- Lock-free does not mean fast.
  - For a mutex protected single linked list, each operation involves only two memory load and store, while a lock-free queue needs to implement its own memory management schema, and each operation is much more expensive.
  - Linked list is memory allocation free, while a fast `crossbeam::queue::SegQueue` involves memory allocation during push. But we are writing a runtime.
- With mutex, more complicate scheduling algorithm could be implemented.

# MPMC queue: naive approach

A shared mutex protected single linked list.

- Thread safety is guaranteed.
- Not scalable.
  - The throughput (lock/unlock operations per second) for the mutex has an upper bound.
  - So that the throughput of the MPMC queue is bounded by the mutex's throughput bound.
  - Cannot improve the upper bound by adding new threads to the runtime.

# MPMC queue: improved naive approach

One mutex protected single linked list per thread as its local queue.

- Thread safety is guaranteed.
- Scalable.
- Work imbalance problem between threads.
  - Solution 0 (Work-sharing): when some thread finds its local queue is too full, share tasks to other threads' local queue.
  - Solution 1 (Work-stealing): when some thread finds its local queue is empty, try stealing tasks from other threads' local queue.

# MPMC queue: Work-sharing vs. Work-stealing

- Work-sharing ✗
  - When all threads find its local queue too full, they would share tasks to other threads' local queue and make things even worse.
- Work-stealing ✓
  - When all threads are busy, no overhead.
  - When some thread local queue is empty, it would consume other threads' task, and improve the total efficiency.

# MPMC queue: termination detection

- What if all threads' local queue is empty?
  - All threads would try stealing tasks from others forever.
  - Need to detect the termination.
- Termination detecting barriers.
  - *The Art of Multiprocessor Programming. Chapter 17, Barriers. Section 17.6, Termination Detecting Barriers.*

```
public interface TDBarrier {
    void setActive(boolean state);
    boolean isTerminated();
}
```

# Termination detecting barriers

```
public class SimpleTDBarrier implements TDBarrier {
  AtomicInteger count;

  public SimpleTDBarrier(int n){
    count = new AtomicInteger(n);
  }

  public void setActive(boolean active) {
    if (active) {
      count.getAndDecrement();
    } else {
      count.getAndIncrement();
    }
  }

  public boolean isTerminated() {
    return count.get() == 0;
  }
}
```

```
public void run() {
  int me = ThreadID.get();
  tdBarrier.setActive(true);
  Runnable task = queue[me].popBottom();
  while (true) {
    while (task != null) {
      task.run();
      task = queue[me].popBottom();
    }
    tdBarrier.setActive(false);
    while (task == null) {
      int victim = random.nextInt() % queue.length;
      if (!queue[victim].isEmpty()) {
        tdBarrier.setActive(true);
        task = queue[victim].popTop();
        if (task == null) {
          tdBarrier.setActive(false);
        }
      }
      if (tdBarrier.isTerminated()) {
        return;
      }
    }
  }
}
```

# Termination detecting barriers (cont'd)

- This termination detecting barriers algorithm is one shot.
  - Key idea: keep an active counter.
- Under the use case of async runtime, there would always be new tasks pushed to these local queues, so we cannot directly adopt the original termination detecting barriers.
- Solution
  - Keep an active counter & an epoch counter.
  - When new events happen, new tasks are pushed into local queues, and the epoch counter are increased.
  - Threads go to sleep only when both:
    1. The active counter as a termination detecting barriers shows termination.
    2. The epoch counter does not change.

# Epoch termination detecting barriers

```rust
#[repr(align(64))]
pub(crate) struct Epoch(AtomicU32);

const ACTIVE_COUNT_BITS: u32 = 9;
const ACTIVE_COUNT_MASK: u32 = !(((!(0 as u32)) >> ACTIVE_COUNT_BITS) << ACTIVE_COUNT_BITS);
const EPOCH: u32 = (1 as u32) << ACTIVE_COUNT_BITS;
const MAX_DURATION: Duration = Duration::from_micros(1_000);

impl Epoch {
    fn active_count(status: u32) -> u32 { status & ACTIVE_COUNT_MASK }
    fn epoch(status: u32) -> u32 { status >> ACTIVE_COUNT_BITS }
}
```

# Epoch termination detecting barriers (cont'd)

```
impl Epoch {
    fn get_instant(&self) -> Instant { Instant::now() }
    fn get_status(&self) -> u32 { self.0.load(SeqCst) }

    fn set_active(&self) -> u32 {
        let status = self.0.fetch_add(1, SeqCst);
        if Epoch::active_count(status) == 0 { self.wake_all_slow(); };
        status + 1
    }
    fn set_inactive(&self) -> u32 { self.0.fetch_sub(1, SeqCst) - 1 }

    fn wait_next_epoch(&self, old_status: u32, old_instant: Instant) {
        if self.0.load(SeqCst) == old_status { self.wait_slow(old_status, old_instant); };
    }
    fn next_epoch(&self) {
        let status = self.0.fetch_add(EPOCH, SeqCst);
        if Epoch::active_count(status) == 0 { self.wake_all_slow(); };
    }
}
```

# `Epoch::wait_slow()` & `Epoch::wake_all_slow()`

- `Epoch::wait_slow(old_status, old_instant)` would:
  1. Check if `self.0` has changed (using CAS), and if not,
  2. Check if the elapsed time from `old_instant` is longer than `MAX_DURATION`, and if not,
  3. Block current thread and wait for wake up using OS API (e.g. `futex` API).
- `Epoch::wake_all_slow()` would wake up all blocking threads (if any) using OS API (e.g. `futex` API).
- Calls to these two function have a single total order (approximately as if the body of these two functions are protected by a same mutex).

# Epoch termination detecting barriers: usage

```
loop {
  // drain the local queue until it is empty          // L0
  let mut old_instant = epoch.get_instant();          // L1
  let mut old_status = epoch.set_inactive();          // L2
  loop {
    if Epoch::active_count(old_status) == 0           // L3
      // recheck the local queue is empty             // L4
      // if not empty, break                          // L5
      epoch.wait_next_epoch(old_status, old_instant); // L6
    } else {
      // try stealing task                            // L7
      // if succeed, break                            // L8
    }
    // try pop from local queue                       // L9
    // if succeed, break                              // L10
    old_instant = epoch.get_instant();                // L11
    old_status = epoch.get_status();                  // L12
  }
  epoch.set_active();                                 // L13
}
```

# Epoch termination detecting barriers: no deadlock

Lemma A: If some threads is blocked on L6, and current `active_count` is not 0, these threads would be unblocked eventually.

```
loop {
  // drain the local queue until it is empty          // L0
  let mut old_instant = epoch.get_instant();          // L1
  let mut old_status = epoch.set_inactive();          // L2
  loop {
    if Epoch::active_count(old_status) == 0           // L3
      // recheck the local queue is empty             // L4
      // if not empty, break                          // L5
      epoch.wait_next_epoch(old_status, old_instant); // L6
    } else {
      // try stealing task                            // L7
      // if succeed, break                            // L8
    }
    // try pop from local queue                       // L9
    // if succeed, break                              // L10
    old_instant = epoch.get_instant();                // L11
    old_status = epoch.get_status();                  // L12
  }
  epoch.set_active();                                 // L13
}
```

Why we could say current `active_count`? Because it is atomic.

*Proof*: Assume some threads are blocked on L6 and `active_count` is not 0 at time $t_0$. According to L3, at some time $t < t_0$, the `active_count` is 0. Assume $t_1$ is the largest $t$ at when the `active_count` is 0, so that between $t_1$ and $t_0$, one thread would call `set_active()` to set `active_count` from 0 to 1. According to the implementation of `set_active()`, it would invoke `wake_all_slow()` when set `active_count` from 0 to 1. So eventually, this `wake_all_slow()` would unblock these threads.

# Epoch termination detecting barriers: no deadlock (cont'd)

Lemma B: If some threads are blocked on `L6` when some one invokes `epoch.next_epoch()`, these threads would be unblocked eventually.

```
loop {
  // drain the local queue until it is empty          // L0
  let mut old_instant = epoch.get_instant();          // L1
  let mut old_status = epoch.set_inactive();          // L2
  loop {
    if Epoch::active_count(old_status) == 0           // L3
      // recheck the local queue is empty             // L4
      // if not empty, break                          // L5
      epoch.wait_next_epoch(old_status, old_instant); // L6
    } else {
      // try stealing task                            // L7
      // if succeed, break                            // L8
    }
    // try pop from local queue                       // L9
    // if succeed, break                              // L10
    old_instant = epoch.get_instant();                // L11
    old_status = epoch.get_status();                  // L12
  }
  epoch.set_active();                                 // L13
}
```

*Proof*:

1. If `epoch.next_epoch()` found `active_count` is not 0, according to lemma A, all worker threads would be unblocked eventually.
2. If `epoch.next_epoch()` found `active_count` is 0, it would invoke `wake_all_slow()` so that these threads would be unblocked eventually.

# Epoch termination detecting barriers: no deadlock (cont'd)

Lemma C: After some one invokes `epoch.next_epoch()`, all worker threads would eventually check its local queue.

```
loop {
  // drain the local queue until it is empty          // L0
  let mut old_instant = epoch.get_instant();          // L1
  let mut old_status = epoch.set_inactive();          // L2
  loop {
    if Epoch::active_count(old_status) == 0           // L3
      // recheck the local queue is empty             // L4
      // if not empty, break                          // L5
      epoch.wait_next_epoch(old_status, old_instant); // L6
    } else {
      // try stealing task                            // L7
      // if succeed, break                            // L8
    }
    // try pop from local queue                       // L9
    // if succeed, break                              // L10
    old_instant = epoch.get_instant();                // L11
    old_status = epoch.get_status();                  // L12
  }
  epoch.set_active();                                 // L13
}
```

*Proof*: According to lemma B, we only need to consider those running worker threads when some one invokes `epoch.next_epoch()`.

1. If a thread is between L5 and L6, it would failed to enter blocking state, so that it would eventually check its local queue.
2. If a thread is not between L5 and L6, it would eventually check its local queue.

# Epoch termination detecting barriers: no deadlock (cont'd)

Lemma D (no deadlock): When all threads are blocked on L6, all local queues are empty.

```
loop {
  // drain the local queue until it is empty          // L0
  let mut old_instant = epoch.get_instant();          // L1
  let mut old_status = epoch.set_inactive();          // L2
  loop {
    if Epoch::active_count(old_status) == 0           // L3
      // recheck the local queue is empty             // L4
      // if not empty, break                          // L5
      epoch.wait_next_epoch(old_status, old_instant); // L6
    } else {
      // try stealing task                            // L7
      // if succeed, break                            // L8
    }
    // try pop from local queue                       // L9
    // if succeed, break                              // L10
    old_instant = epoch.get_instant();                // L11
    old_status = epoch.get_status();                  // L12
  }
  epoch.set_active();                                 // L13
}
```

*Proof*: Assume when all threads are blocked on L6 at $t_0$, and some local queues is not empty at $t_0$, some task called `task` is in some queue `q`. When `push(task)`, `epoch_next_epoch()` is invoked at time $t_1$. So that after $t_1$, all worker threads would eventually check its local queue by lemma C ($t_0 > t_1$). When `q`'s owner thread checking its local queue, it would find `task` and pop it. Contradiction!

Lemma D tells us when some local queues are not empty, some threads are running, so the system could make progress (a.k.a no deadlock).

# Epoch termination detecting barriers: no deadlock (cont'd)

```
loop {
  // drain the local queue until it is empty          // L0
  let mut old_instant = epoch.get_instant();          // L1
  let mut old_status = epoch.set_inactive();          // L2
  loop {
    if Epoch::active_count(old_status) == 0           // L3
      // recheck the local queue is empty             // L4
      // if not empty, break                          // L5
      epoch.wait_next_epoch(old_status, old_instant); // L6
    } else {
      // try stealing task                            // L7
      // if succeed, break                            // L8
    }
    // try pop from local queue                       // L9
    // if succeed, break                              // L10
    old_instant = epoch.get_instant();                // L11
    old_status = epoch.get_status();                  // L12
  }
  epoch.set_active();                                 // L13
}
```

Please note that lemma C is wrong when the `epoch` overflows to exactly same number before `L6` (the proof's (1) would be wrong). This is avoided by checking time elapsed. If the `old_instant` is `MAX_DURATION` earlier, the `wait_next_epoch()` method won't block. For a 23 bits unsigned integer `epoch` to overflow to same number, the `next_epoch()` needs to be invoked 8388608 times, which at least requires 8388608 atomic `fetch_add(EPOCH)` operations. Hopefully it should take longer than 1000 us.

The key for this proof is that, the worker needs to guarantee rechecking local queue before try to `wait_next_epoch()`. Optimization keeping this property could be applied without breaking the proof.

# MPMC queue

- So finally we got a MPMC queue that,
  - Scalable.
  - (Maybe) Correct (Safe & No deadlock).
  - Give up CPU time when no task.
- Should be called “MPMC pool”.
  - Not FIFO.

# How wait_slow() & wake_all_slow() work

```
impl Epoch {
    fn wait_slow(&self, old_status: u32, old_instant: Instant) {
        let key = &self.0 as *const AtomicU32 as usize;
        let validate = move || {
            unsafe {
                let ptr = key as *const AtomicU32;
                let result = (*ptr).compare_exchange_weak(old_status, old_status, SeqCst, Relaxed);
                if result.is_err() { return false; };
                if old_instant.elapsed() > MAX_DURATION { return false; };
                return true;
            }
        };
        unsafe { park(key, validate, || {}, |_, _| {}, ParkToken(0), None) };
    }
    fn wake_all_slow(&self) {
        let key = &self.0 as *const AtomicU32 as usize;
        unsafe { unpark_all(key, UnparkToken(0)) };
    }
}
```

# parking_lot::park() & parking_lot::unpark_all()

- parking_lot is a crate wrapping all platforms' synchronization API. It provides an interface similar to Linux's futex().
  - Linux: futex().
  - Windows: WaitOnAddress().
  - Others: pthread_mutex_t and pthread_cond_t.

# Linux: `futex()`

Two basic operations, `WAIT` and `WAKE`.

- `WAIT(addr, val)`
  If the value stored at the address `addr` is `val`, puts the current thread to block.
- `WAKE(addr, num)`
  Wakes up `num` number of threads waiting on the address `addr`.

# Windows: `WaitOnAddress()`

Similar to `futex()`.

```
BOOL WaitOnAddress(
  volatile VOID *Address,
  PVOID         CompareAddress,
  SIZE_T        AddressSize,
  DWORD         dwMilliseconds
);

void WakeByAddressAll(
  PVOID Address
);

void WakeByAddressSingle(
  PVOID Address
);
```

# Others: `pthread_mutex_t` and `pthread_cond_t`.

A `futex()` like API could be implemented using mutex and condition variable.