C++ Zero Cost Conf */

Асинхронные фреймворки

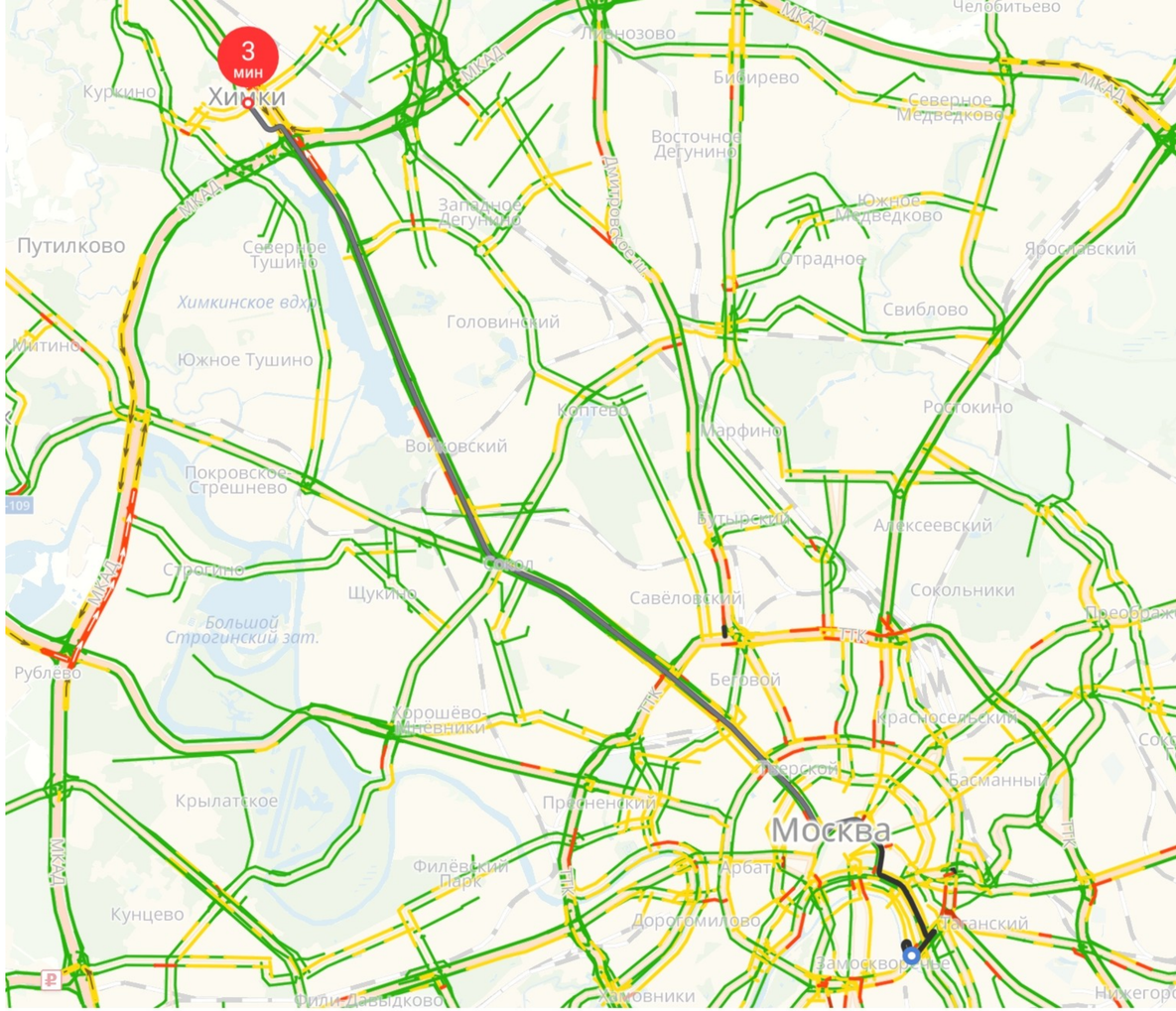
Анатомия

Полухин Антон
Эксперт разработчик C++

Yandex for developers *//>

Содержание

- IO bound архитектуры
  - 1 поток на запрос
  - callbacks
  - корутины
- Устройство движка
- Хитрости / Хардкор
  - стек вместо «кучи»
  - неблокирующие мьютексы



Пишем синхронный сервер

Синхронный сервер

```
void naive_accept() {
  for (;;) {
    auto new_socket = accept(listener);

    std::thread thrd([socket = std::move(new_socket)] {
      auto data = socket.receive();
      process(data);
      socket.send(data);
    });

    thrd.detach();
  }
}
```

Синхронный сервер

```
void naive_accept() {
  for (;;) {
    auto new_socket = accept(listener);

    std::thread thrd([socket = std::move(new_socket)] {
      auto data = socket.receive();
      process(data);
      socket.send(data);
    });

    thrd.detach();
  }
}
```

Синхронный сервер

```
void naive_accept() {
  for (;;) {
    auto new_socket = accept(listener);

    std::thread thrd([socket = std::move(new_socket)] {
      auto data = socket.receive();
      process(data);
      socket.send(data);
    });

    thrd.detach();
  }
}
```

Синхронный сервер

```
void naive_accept() {
  for (;;) {
    auto new_socket = accept(listener);

    std::thread thrd([socket = std::move(new_socket)] {
      auto data = socket.receive();
      process(data);
      socket.send(data);
    });

    thrd.detach();
  }
}
```

Синхронный сервер

```
void naive_accept() {
  for (;;) {
    auto new_socket = accept(listener);

    std::thread thrd([socket = std::move(new_socket)] {
      auto data = socket.receive();
      process(data);
      socket.send(data);
    });

    thrd.detach();
  }
}
```

Синхронный сервер

```
void naive_accept() {
  for (;;) {
    auto new_socket = accept(listener);

    std::thread thrd([socket = std::move(new_socket)] {
      auto data = socket.receive();
      process(data);
      socket.send(data);
    });

    thrd.detach();
  }
}
```

Синхронный сервер

```
void naive_accept() {
  for (;;) {
    auto new_socket = accept(listener);

    std::thread thrd([socket = std::move(new_socket)] {
      auto data = socket.receive();
      process(data);
      socket.send(data);
    });

    thrd.detach();
  }
}
```

Синхронный сервер

```
void naive_accept() {
  for (;;) {
    auto new_socket = accept(listener);

    std::thread thrd([socket = std::move(new_socket)] {
      auto data = socket.receive();
      process(data);
      socket.send(data);
    });

    thrd.detach();
  }
}
```

Синхронный сервер

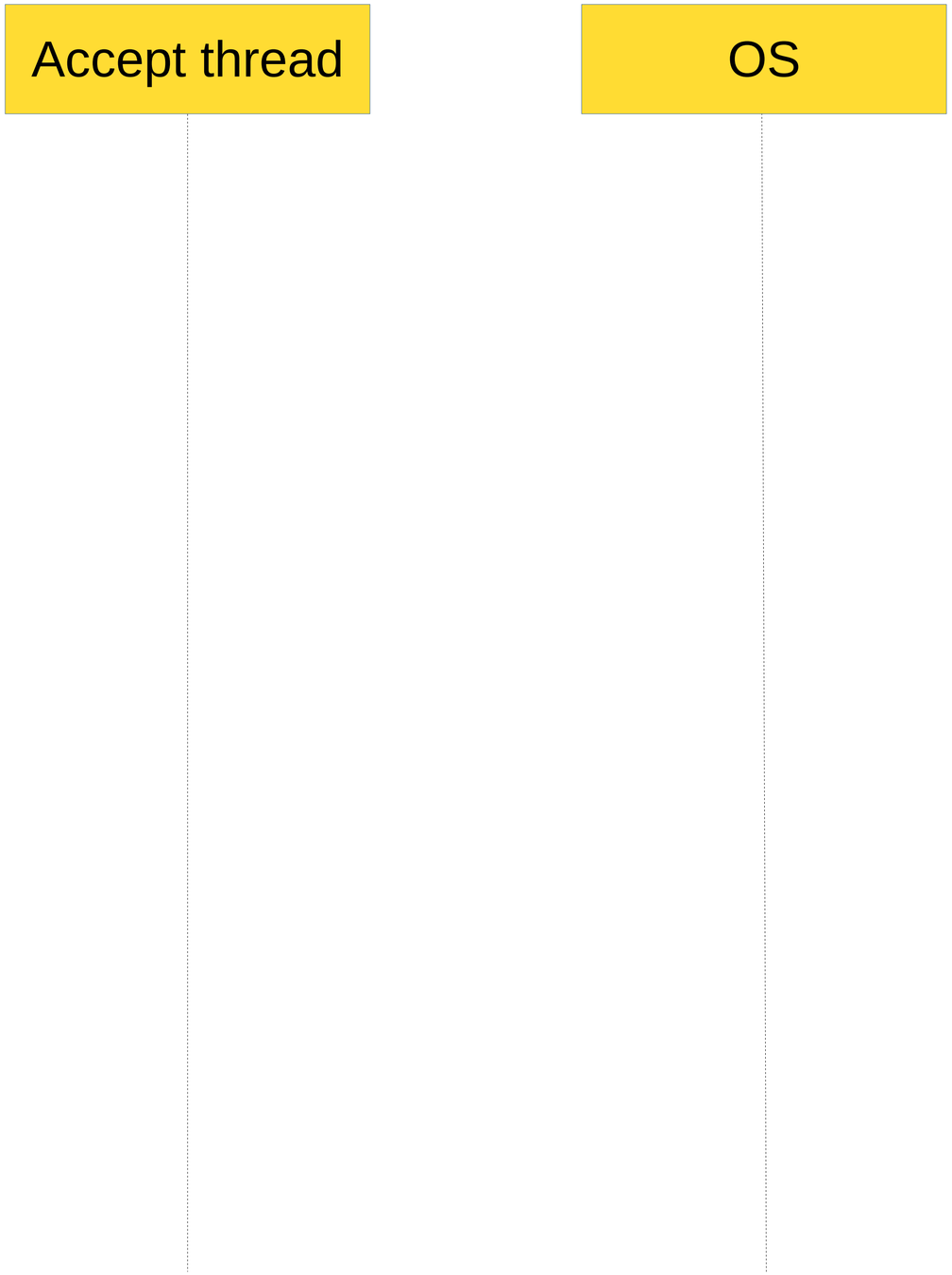
```
void naive_accept() {
  for (;;) {
    auto new_socket = accept(listener);

    std::thread thrd([socket = std::move(new_socket)] {
      auto data = socket.receive();
      process(data);
      socket.send(data);
    });

    thrd.detach();
  }
}
```

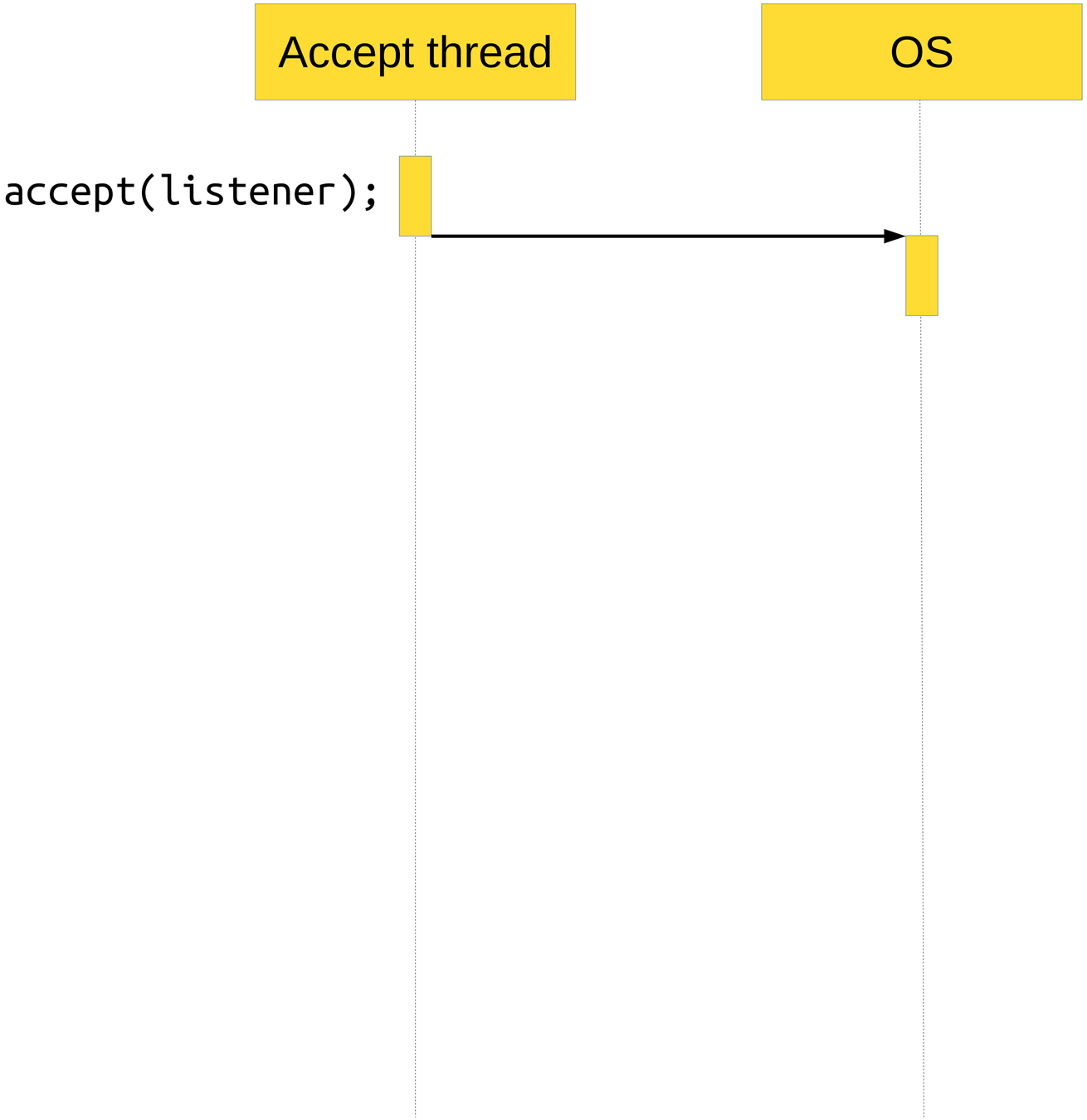
Синхронный сервер

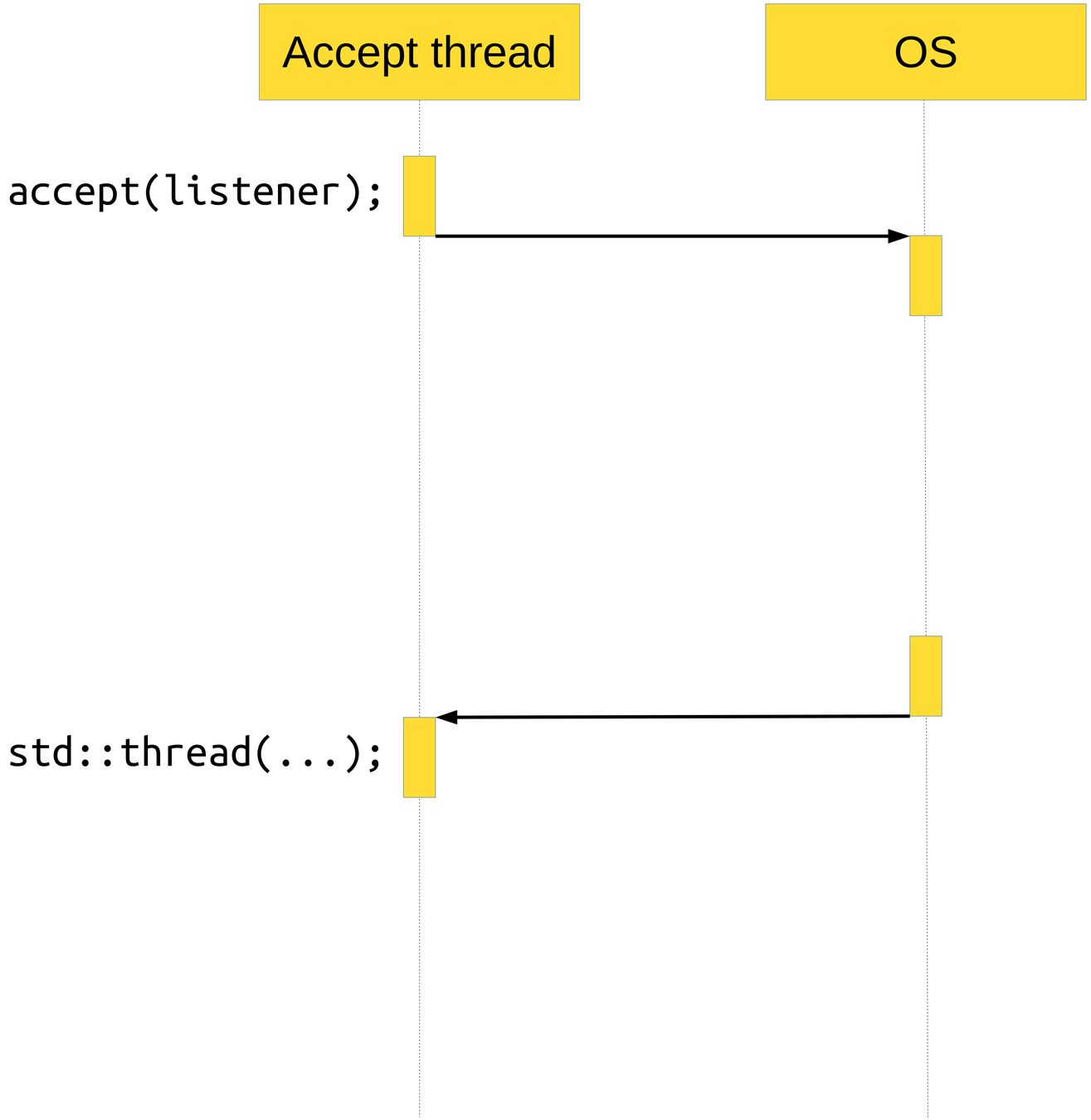
```
void naive_accept() {
  for (;;) {
    auto new_socket = accept(listener);

    std::thread thrd([socket = std::move(new_socket)] {
      auto data = socket.receive();
      process(data);
      socket.send(data);
    });

    thrd.detach();
  }
}
```

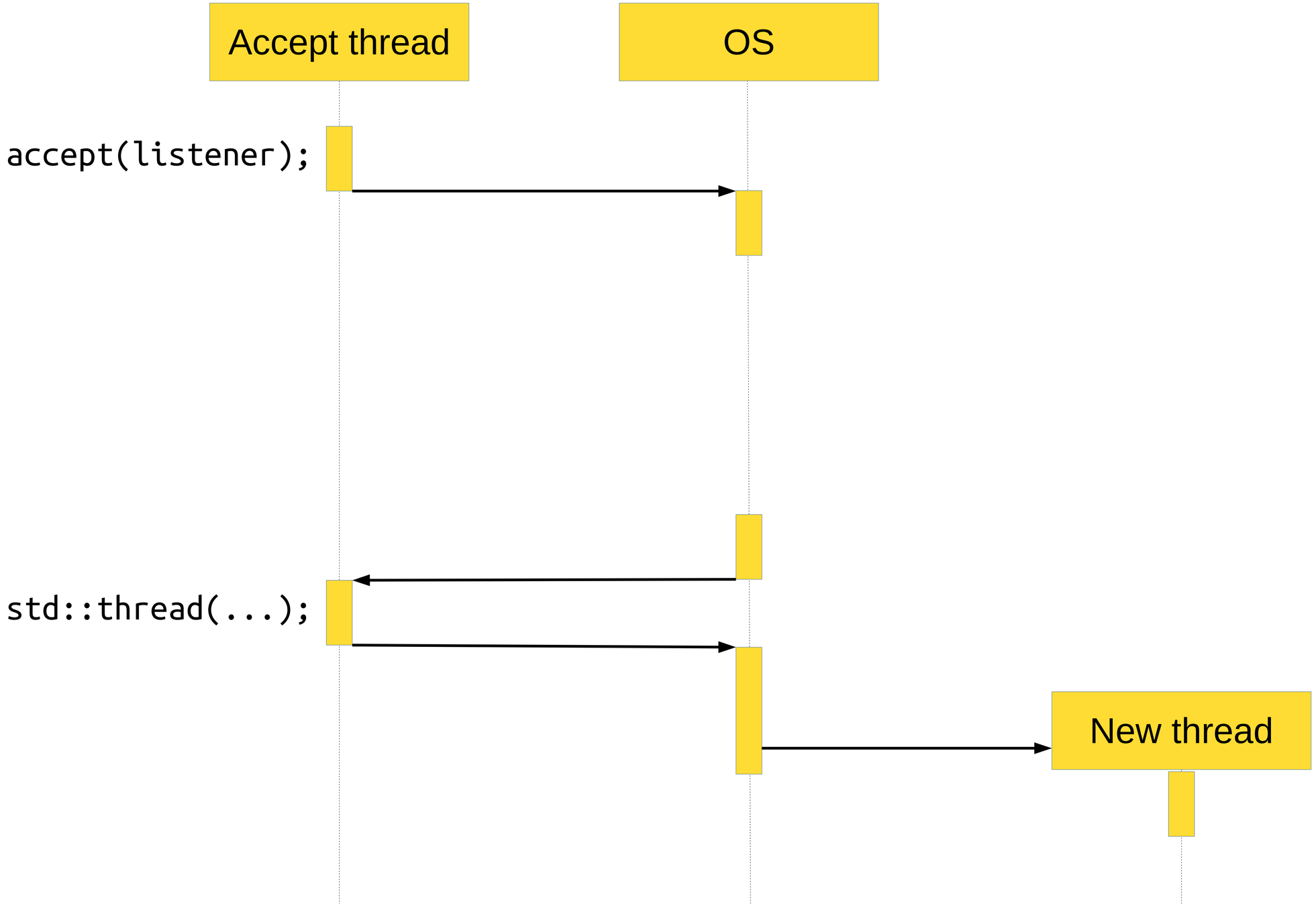
Синхронный сервер

```
void naive_accept() {
  for (;;) {
    auto new_socket = accept(listener);

    std::thread thrd([socket = std::move(new_socket)] {
      auto data = socket.receive();
      process(data);
      socket.send(data);
    });

    thrd.detach();
  }
}
```


Accept thread
OS

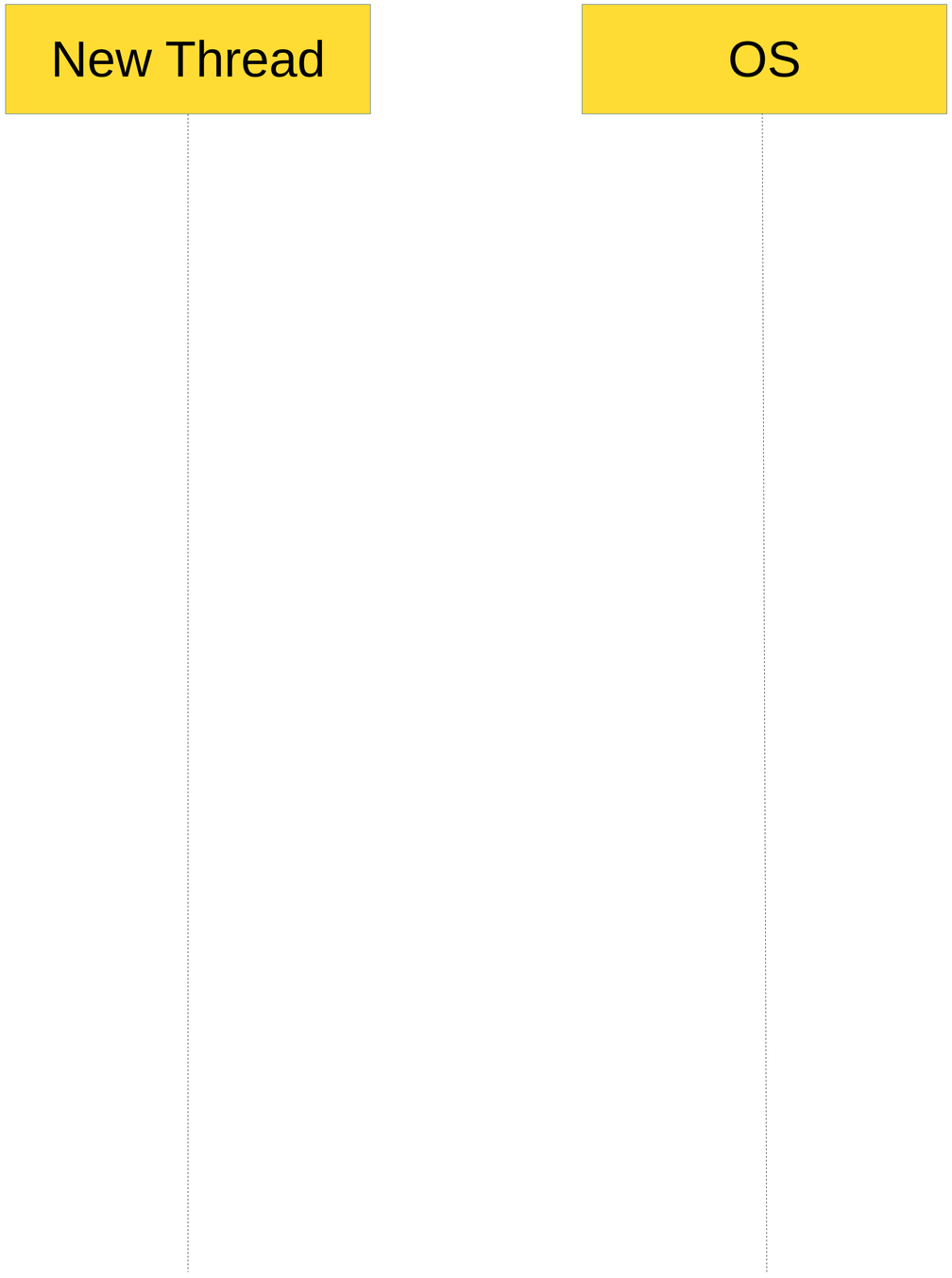

Accept thread
OS
accept(listener);

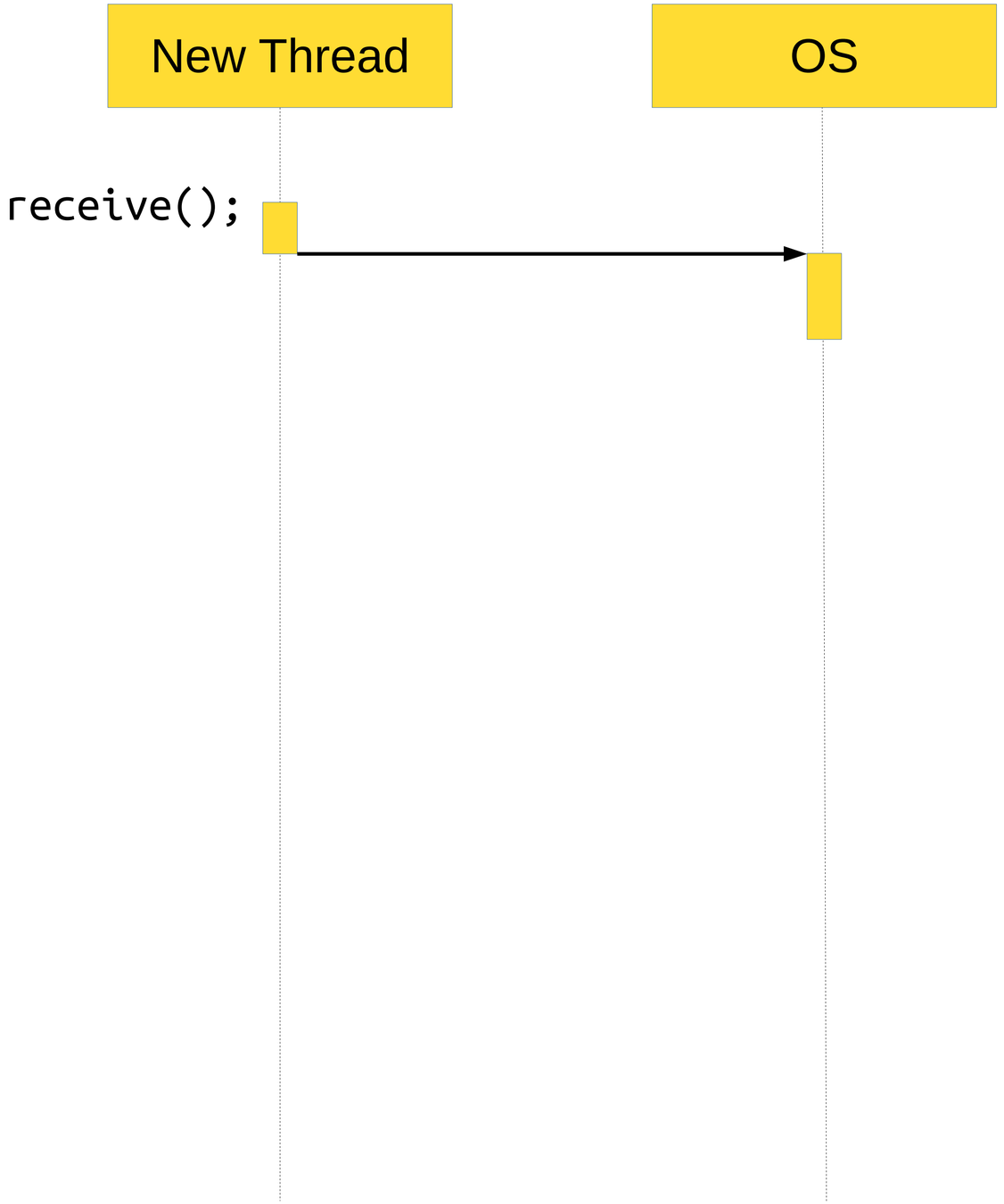

Accept thread
OS
accept(listener);
std::thread(...);

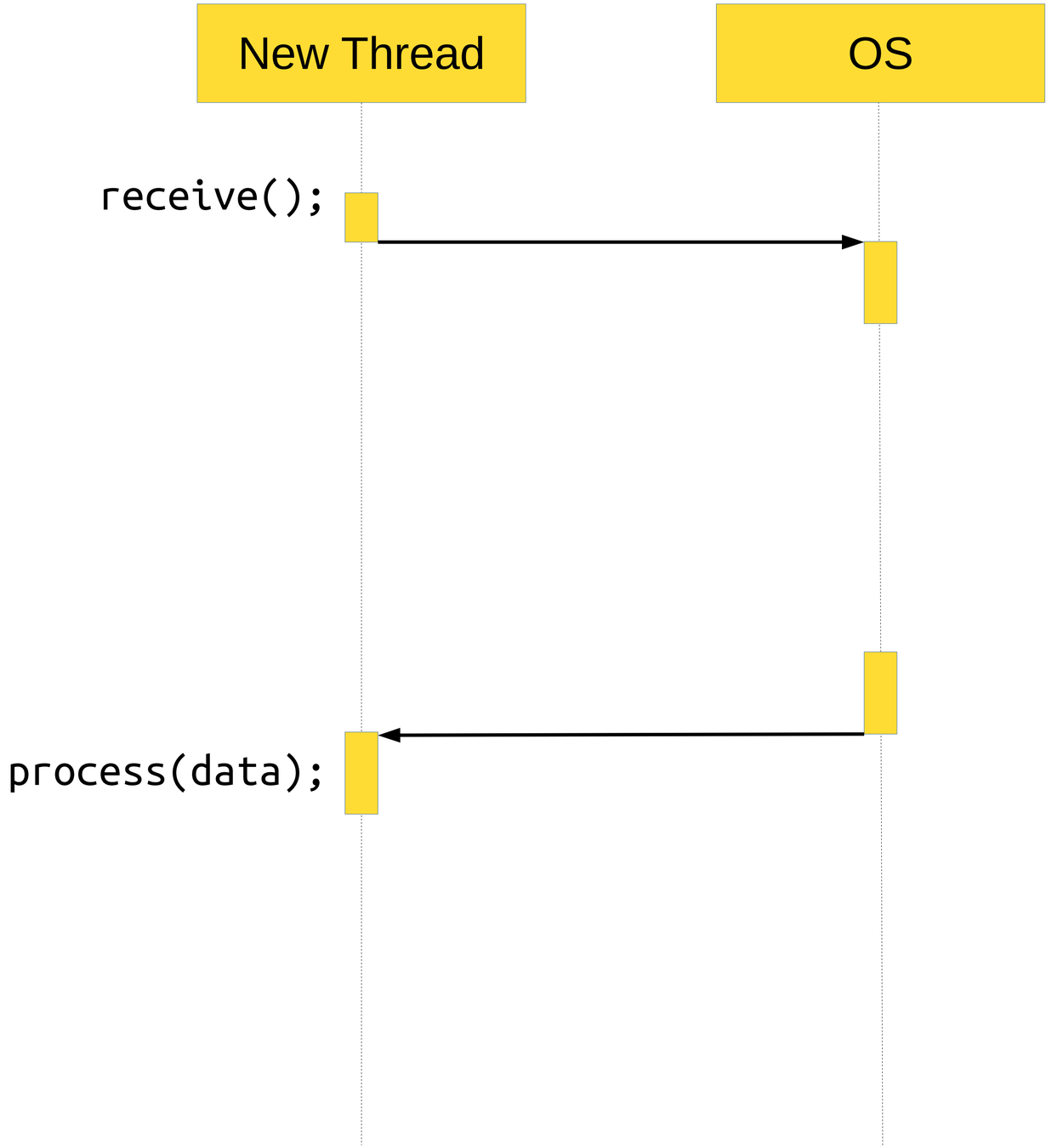

Accept thread
OS
accept(listener);
std::thread(...);
New thread

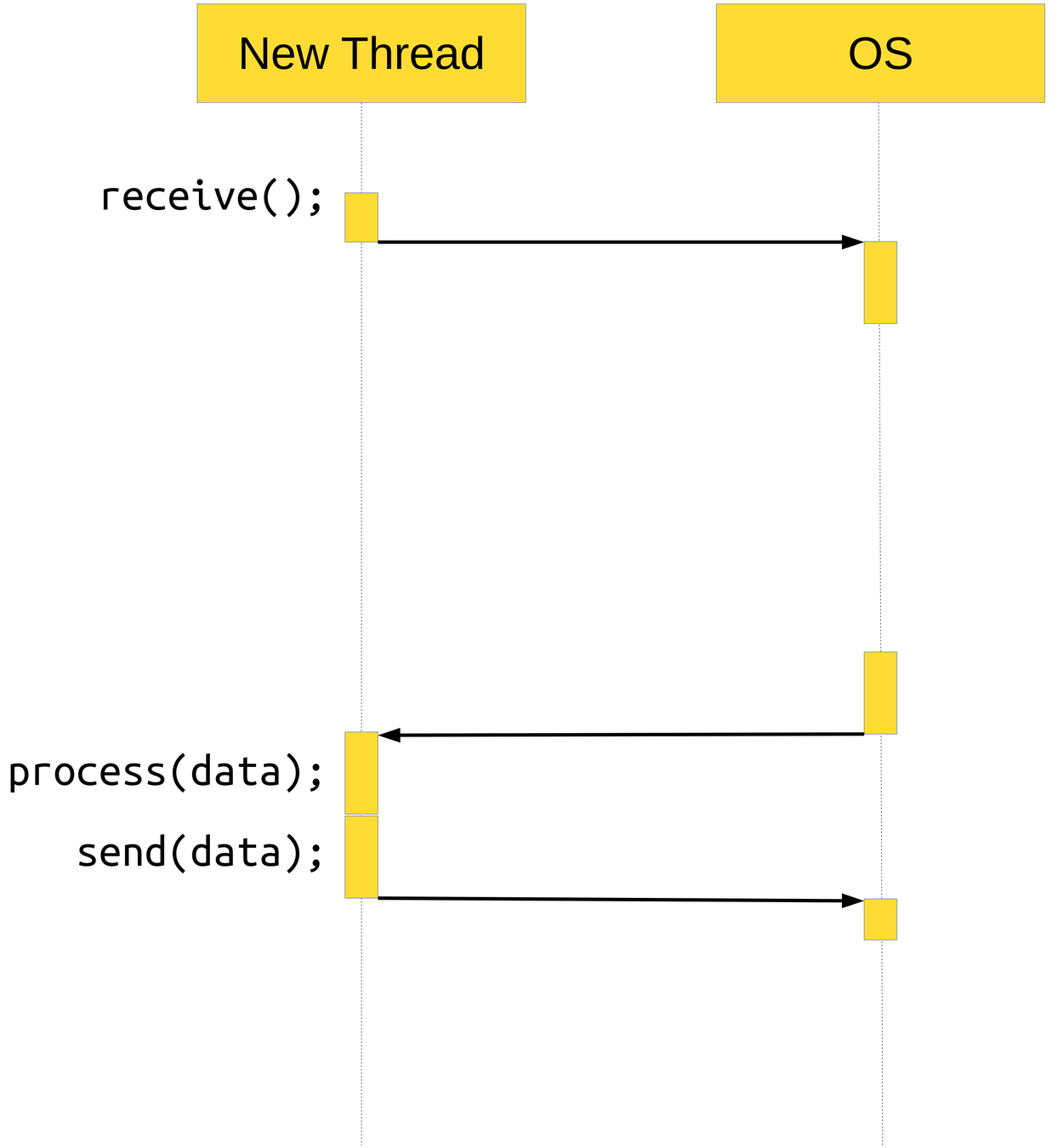

Accept thread
OS
accept(listener);
std::thread(...);
New thread
accept(listener);

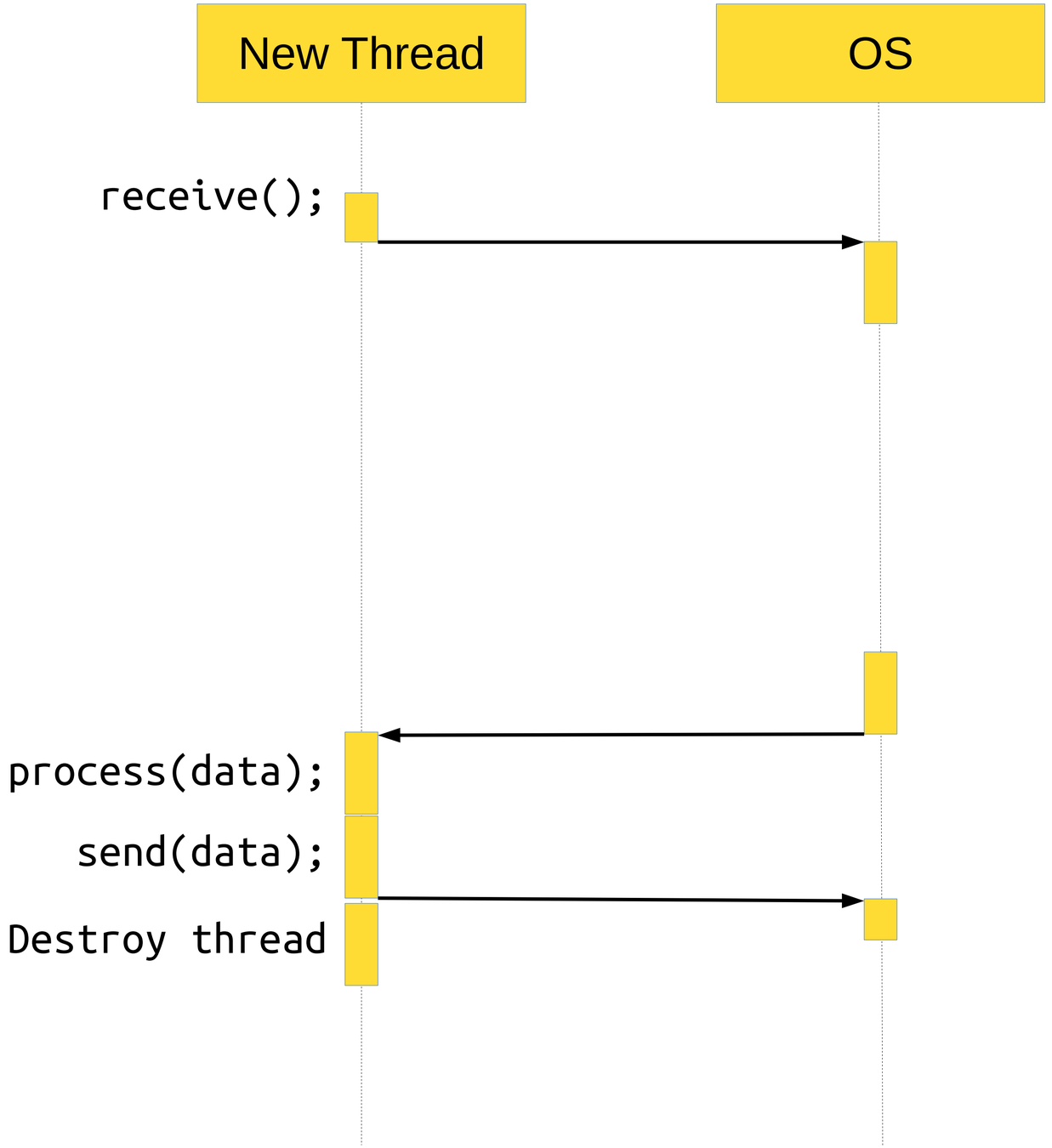
Что там делает новый поток?


New Thread
OS


New Thread
OS
receive();


New Thread
OS
receive();
process(data);


New Thread
OS
receive();
process(data);
send(data);


New Thread
OS
receive();
process(data);
send(data);
Destroy thread

Плюсы/минусы наивного подхода

Плюсы/минусы наивного подхода

Плюсы:

Плюсы/минусы наивного подхода

Плюсы:

- Всё просто и читаемо

Плюсы/минусы наивного подхода

Плюсы:

- Всё просто и читаемо

Минусы:

Плюсы/минусы наивного подхода

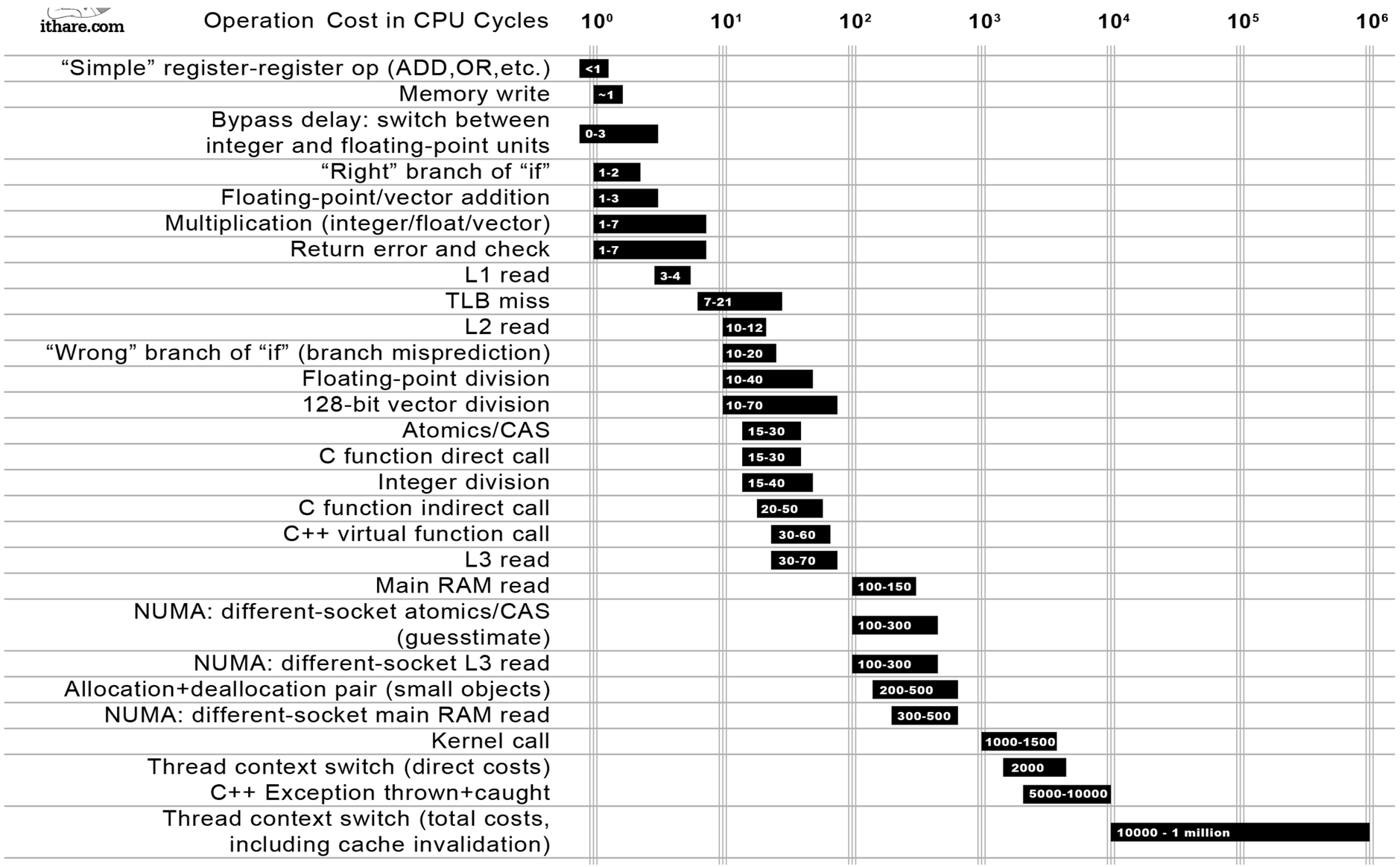
Плюсы:

- Всё просто и читаемо

Минусы:

- Не эффективно, потому что...

«Цена» операции



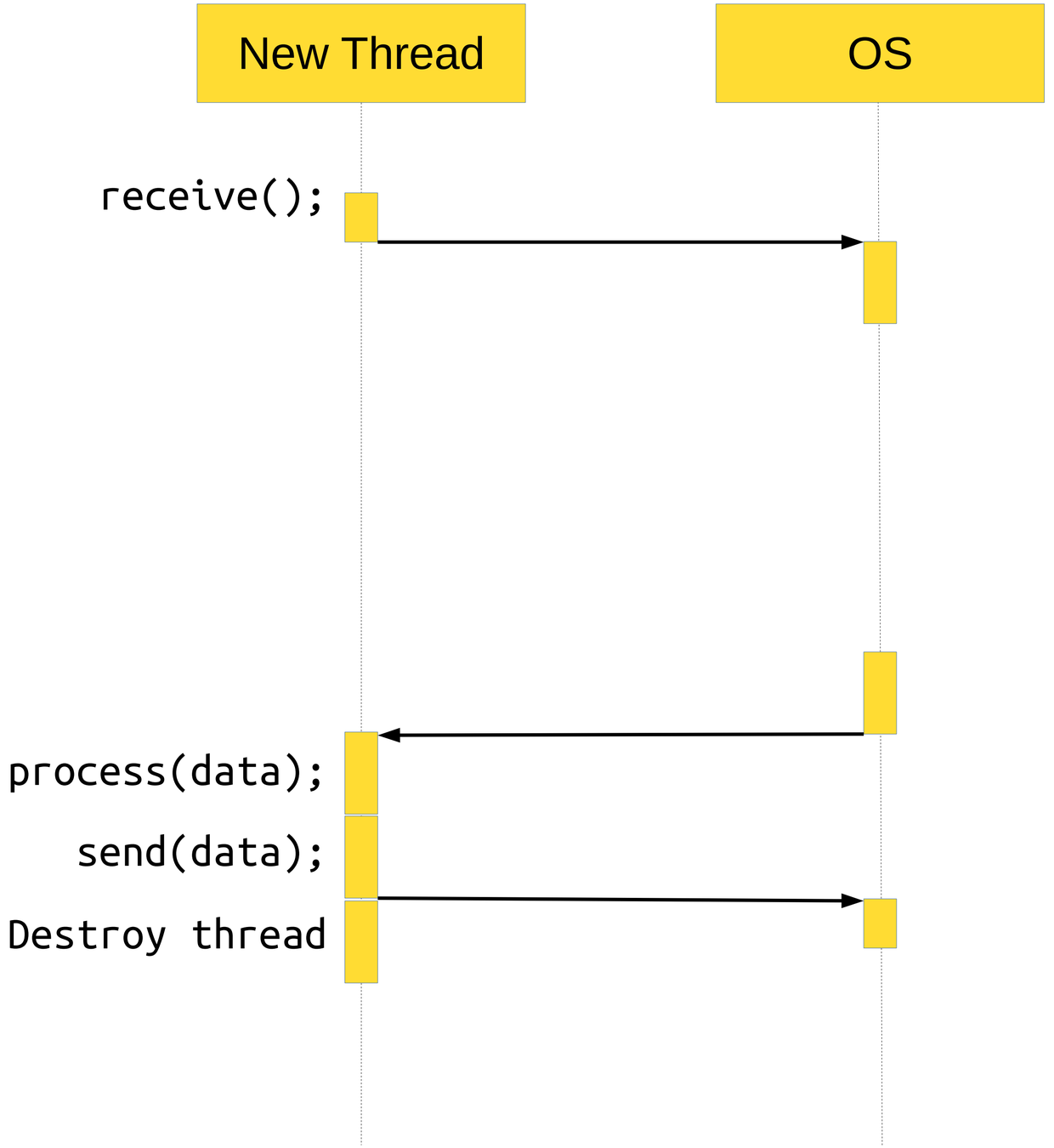
Kernel call / Context Switch

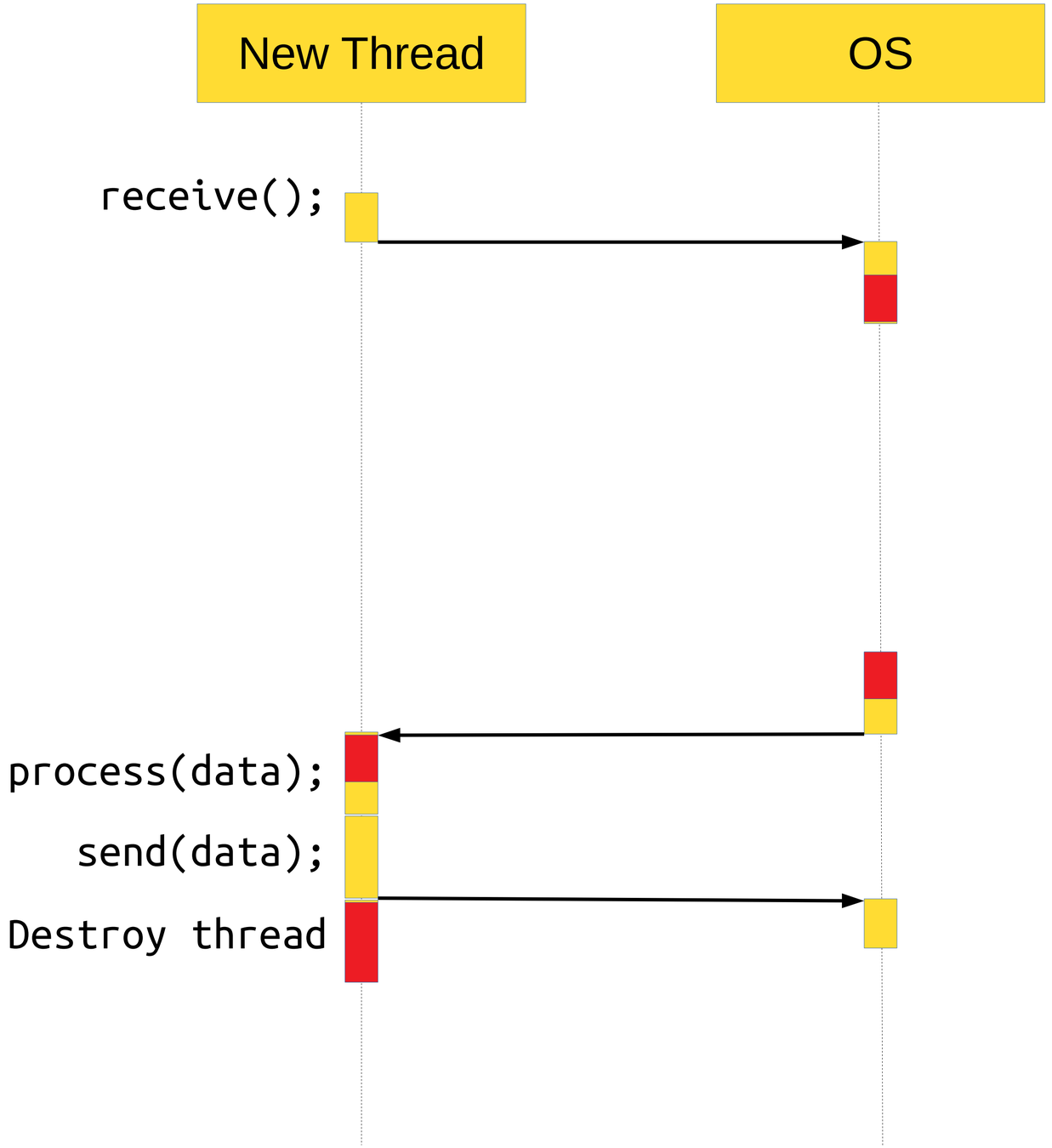

Accept thread
OS
accept(listener);
std::thread(...);
New thread
accept(listener);


Accept thread
OS
accept(listener);
std::thread(...);
New thread
accept(listener);


Accept thread
OS
accept(listener);
std::thread(...);
New thread
accept(listener);


Accept thread
OS
accept(listener);
std::thread(...);
New thread
accept(listener);


New Thread
OS
receive();
process(data);
send(data);
Destroy thread


New Thread
OS
receive();
process(data);
send(data);
Destroy thread

Пишем асинхронный сервер

Было/станет

Было/станет

Было:

Было/станет

Было:

- Отдаём управление ОС при системном вызове

Было/станет

Было:

- Отдаём управление ОС при системном вызове
- Ждём пока событие случится

Было/станет

Было:

- Отдаём управление ОС при системном вызове
- Ждём пока событие случится
- ОС будит поток

Было/станет

Было:

- Отдаём управление ОС при системном вызове
- Ждём пока событие случится
- ОС будит поток

Станет:

Было/станет

Было:

- Отдаём управление ОС при системном вызове
- Ждём пока событие случится
- ОС будит поток

Станет:

- «Забираем» **случившиеся** события

Было/станет

Было:

- Отдаём управление ОС при системном вызове
- Ждём пока событие случится
- ОС будит поток

Станет:

- «Забираем» **случившиеся** события
- Выполняем коллбеки, связанные с этим событиями

Асинхронный сервер

```
void async_accept() {
  accept(listener, [](socket_t socket) {
    async_accept();
    socket.receive(
        [socket](std::vector<unsigned char> data) {
          process(data);
          socket.send(data, kNoCallback);
        });
  });
}
```

Асинхронный сервер

```
void async_accept() {
  accept(listener, [](socket_t socket) {
    async_accept();
    socket.receive(
        [socket](std::vector<unsigned char> data) {
          process(data);
          socket.send(data, kNoCallback);
        });
  });
}
```

Асинхронный сервер

```
void async_accept() {
  accept(listener, [](socket_t socket) {
    async_accept();
    socket.receive(
        [socket](std::vector<unsigned char> data) {
          process(data);
          socket.send(data, kNoCallback);
        });
  });
}
```

Асинхронный сервер

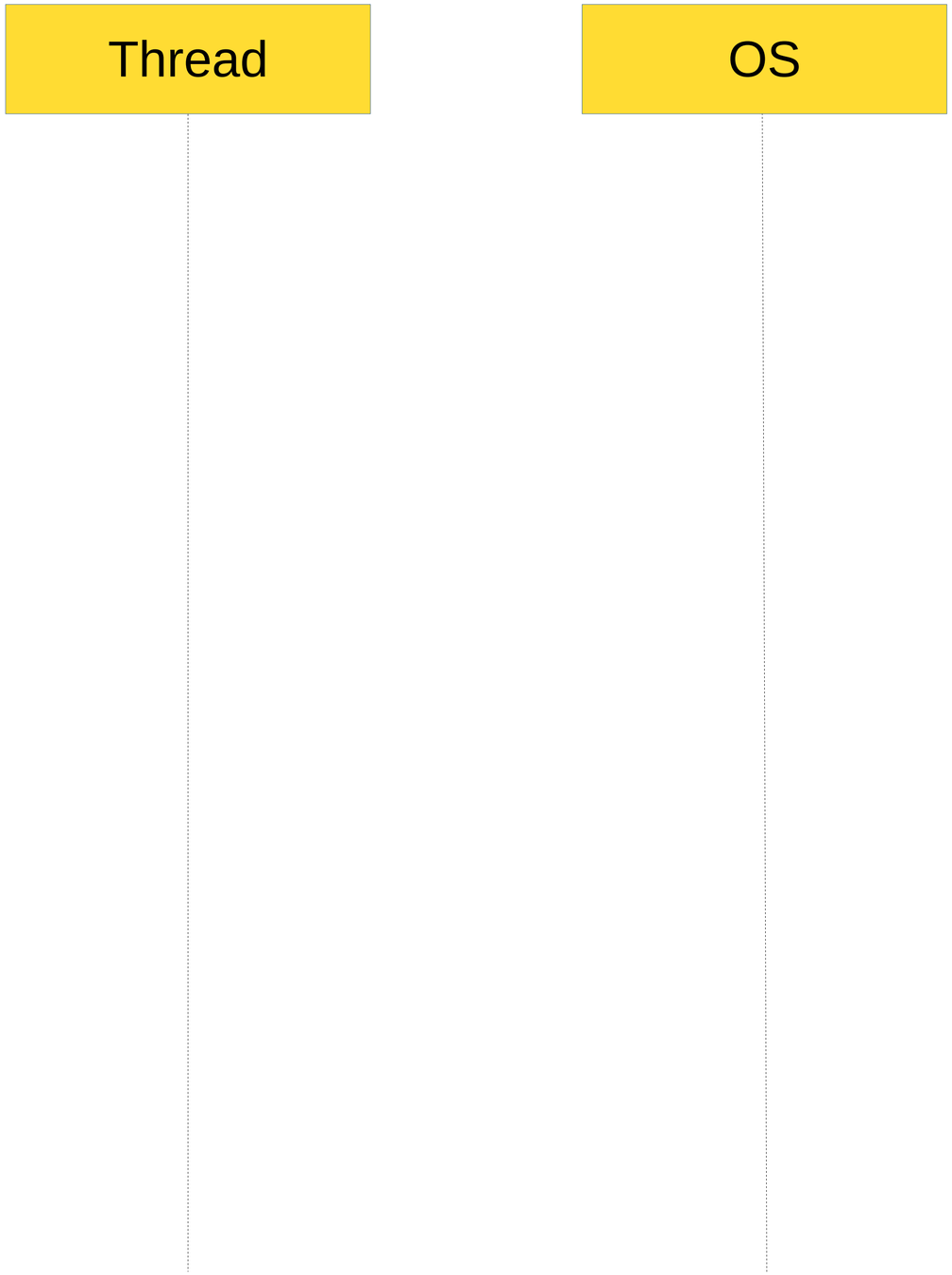
```
void async_accept() {
  accept(listener, [](socket_t socket) {
    async_accept();
    socket.receive(
        [socket](std::vector<unsigned char> data) {
          process(data);
          socket.send(data, kNoCallback);
        });
  });
}
```

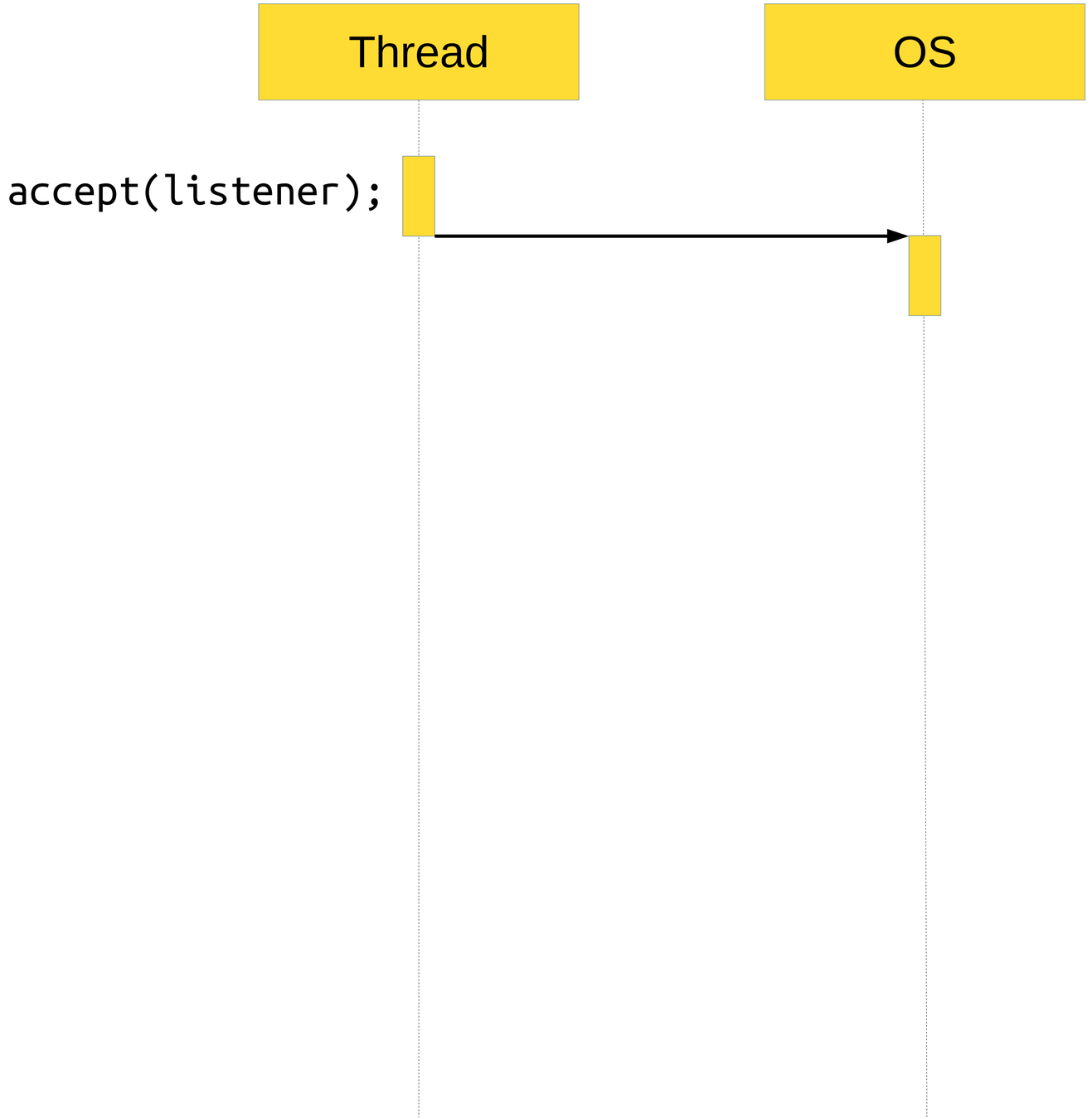
Асинхронный сервер

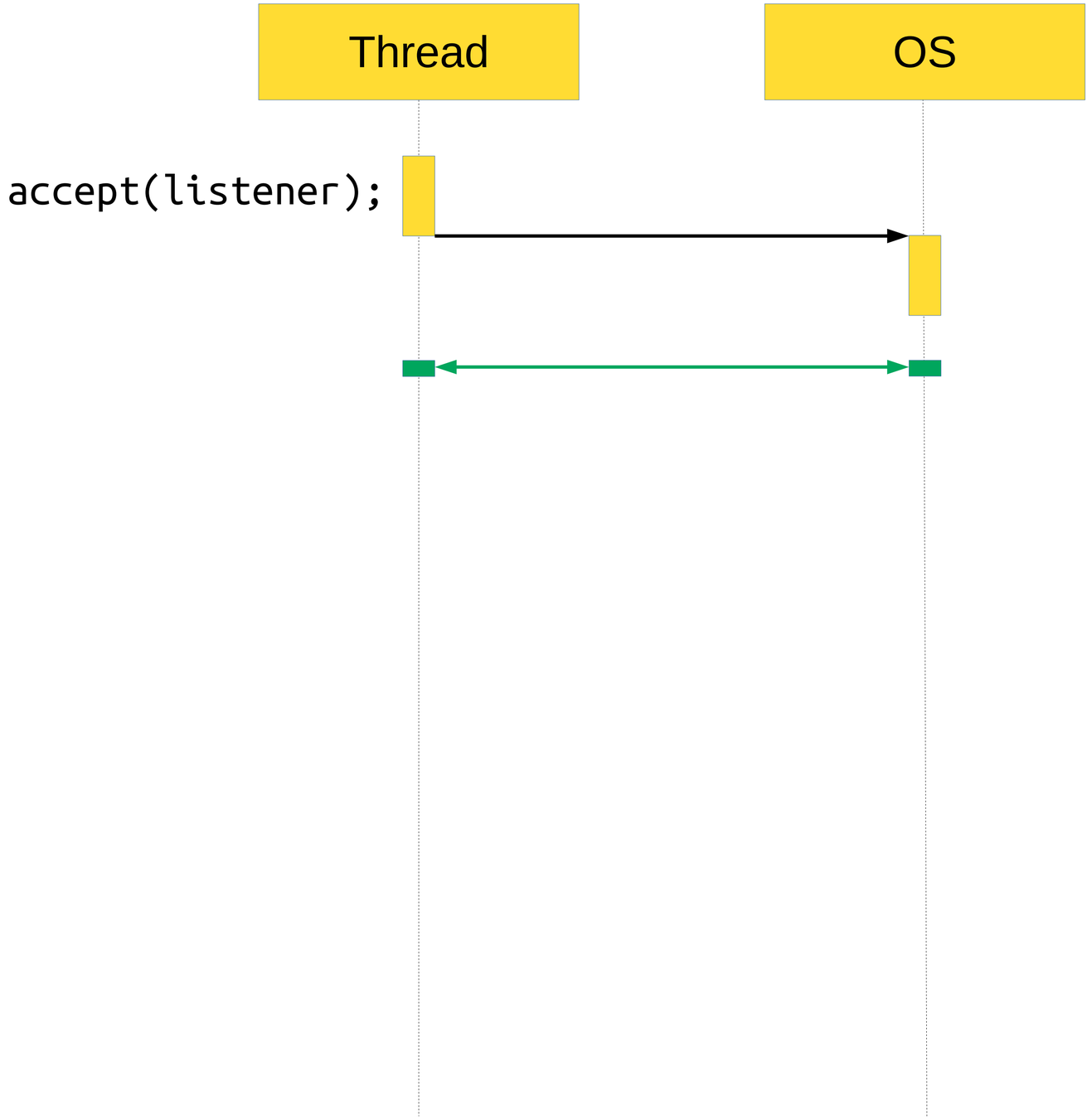
```
void async_accept() {
  accept(listener, [](socket_t socket) {
    async_accept();
    socket.receive(
        [socket](std::vector<unsigned char> data) {
          process(data);
          socket.send(data, kNoCallback);
        });
  });
}
```

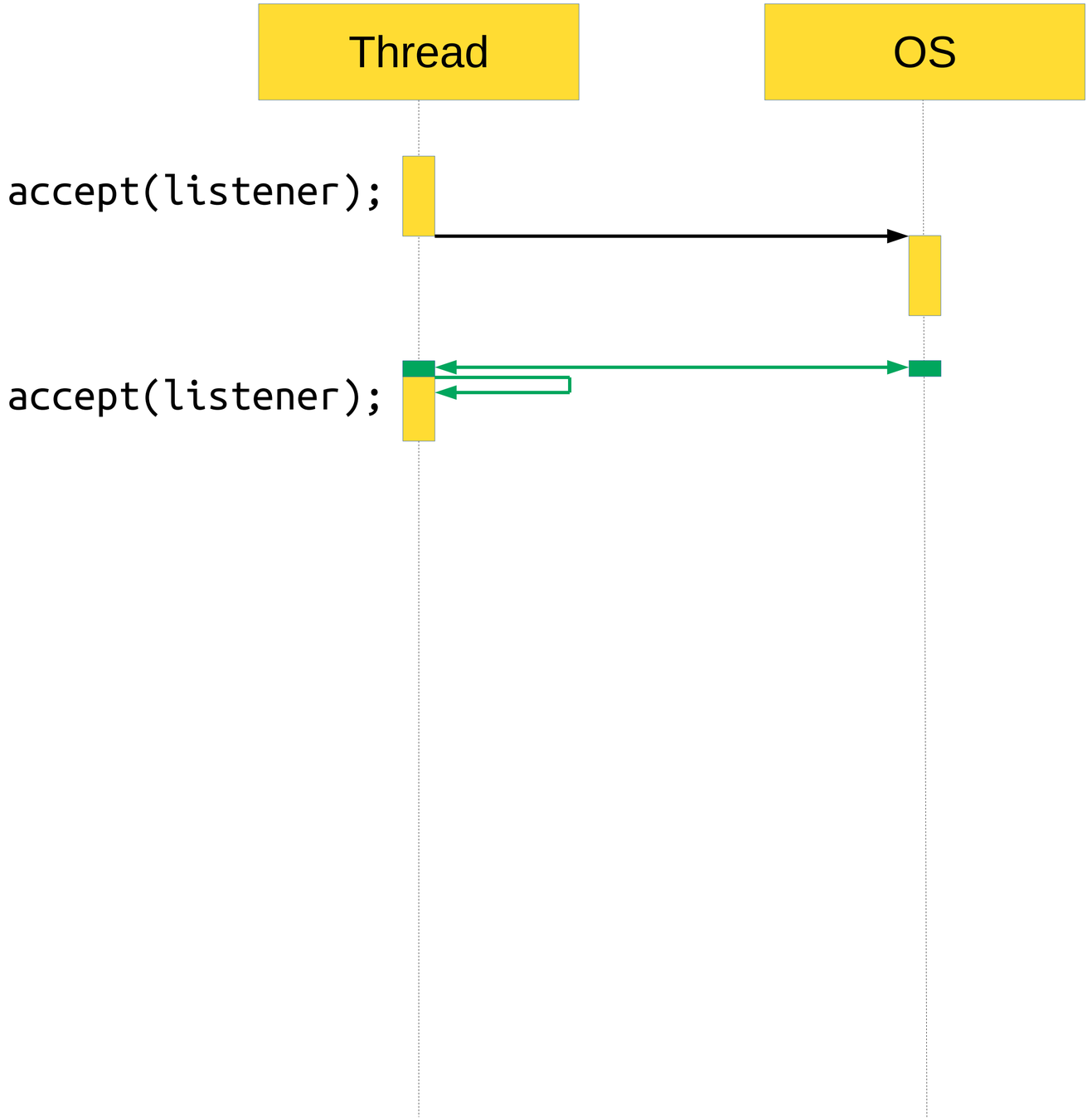
Асинхронный сервер

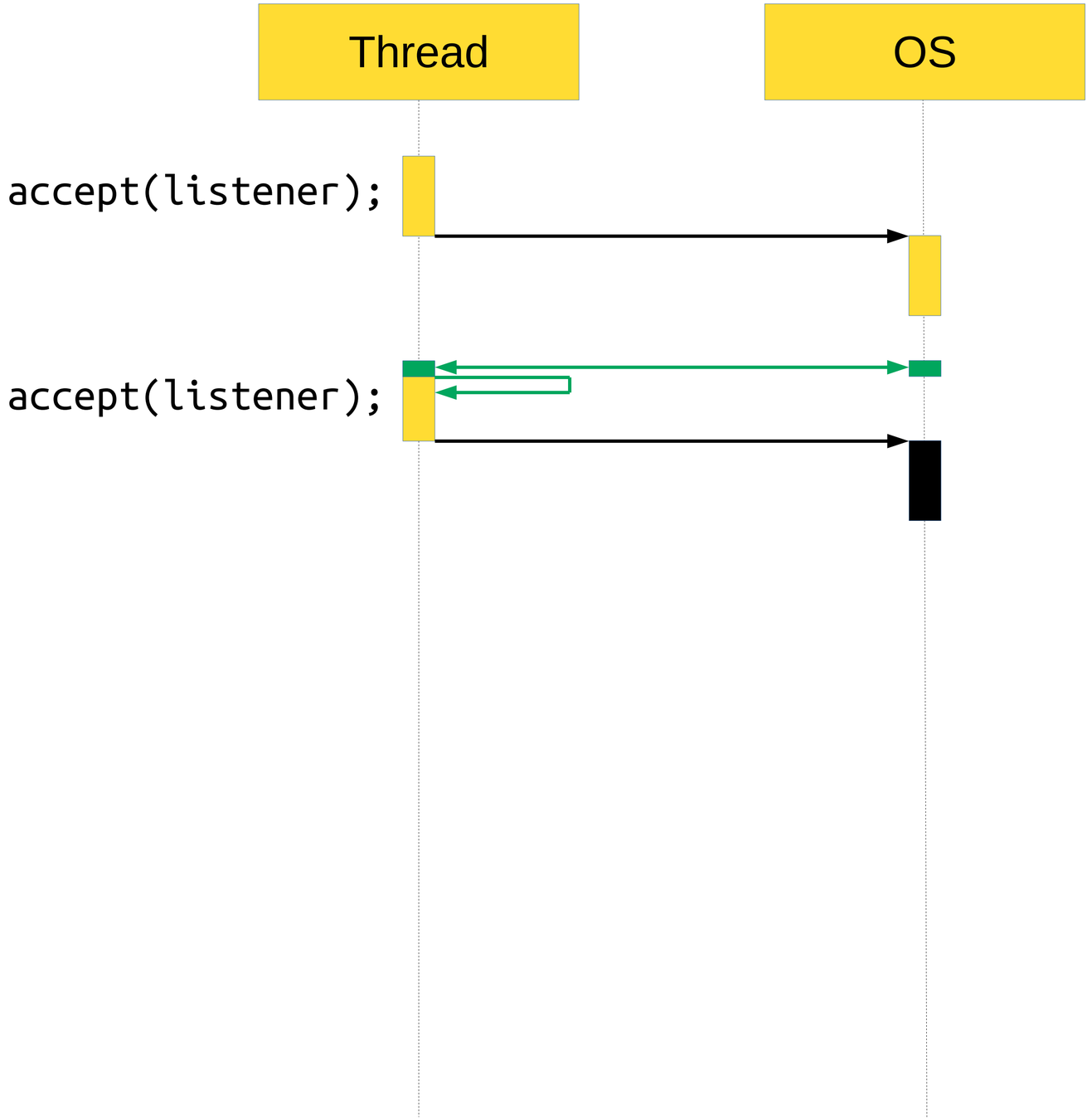
```
void async_accept() {
  accept(listener, [](socket_t socket) {
    async_accept();
    socket.receive(
        [socket](std::vector<unsigned char> data) {
          process(data);
          socket.send(data, kNoCallback);
        });
  });
}
```

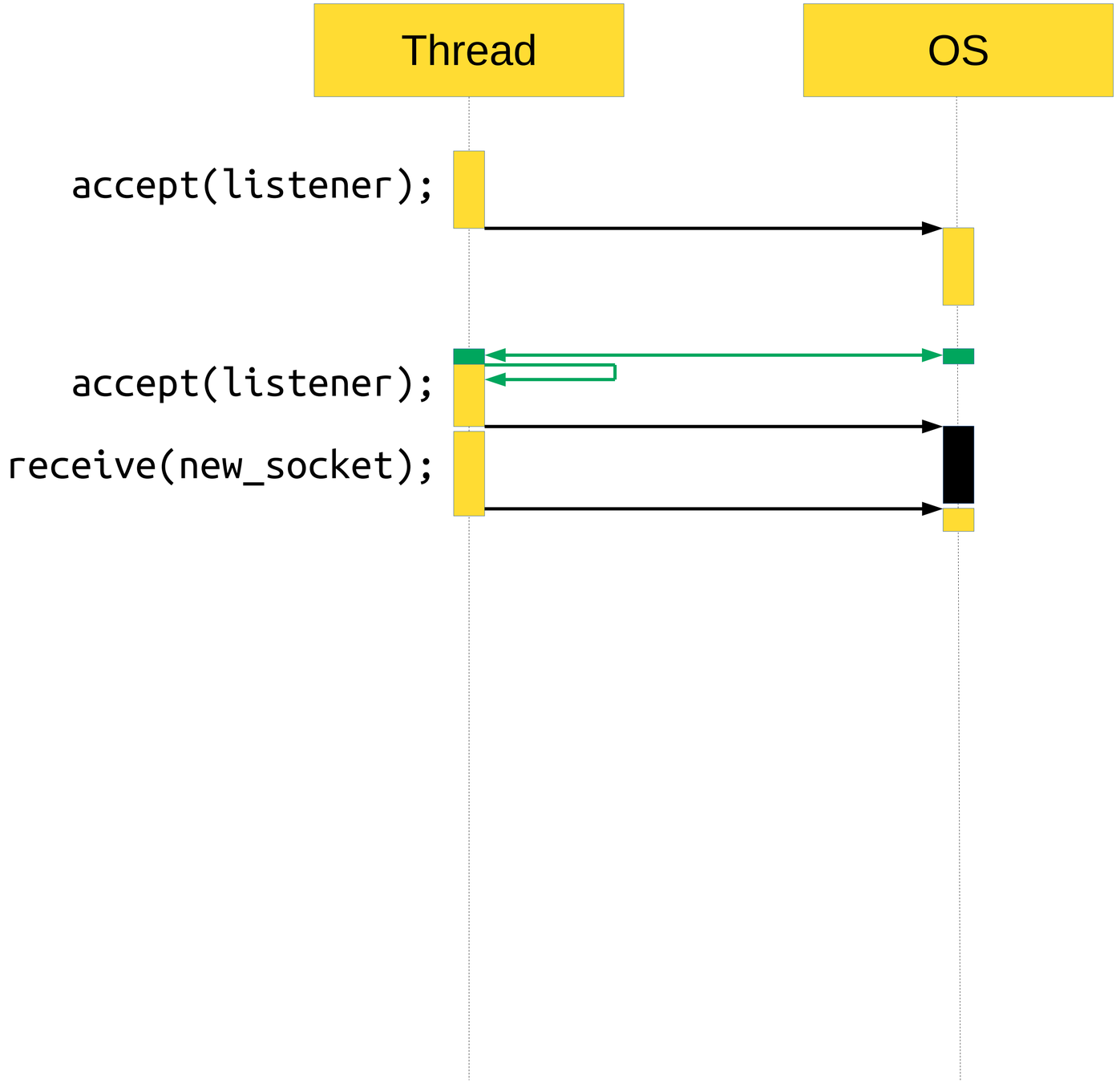

Thread
OS

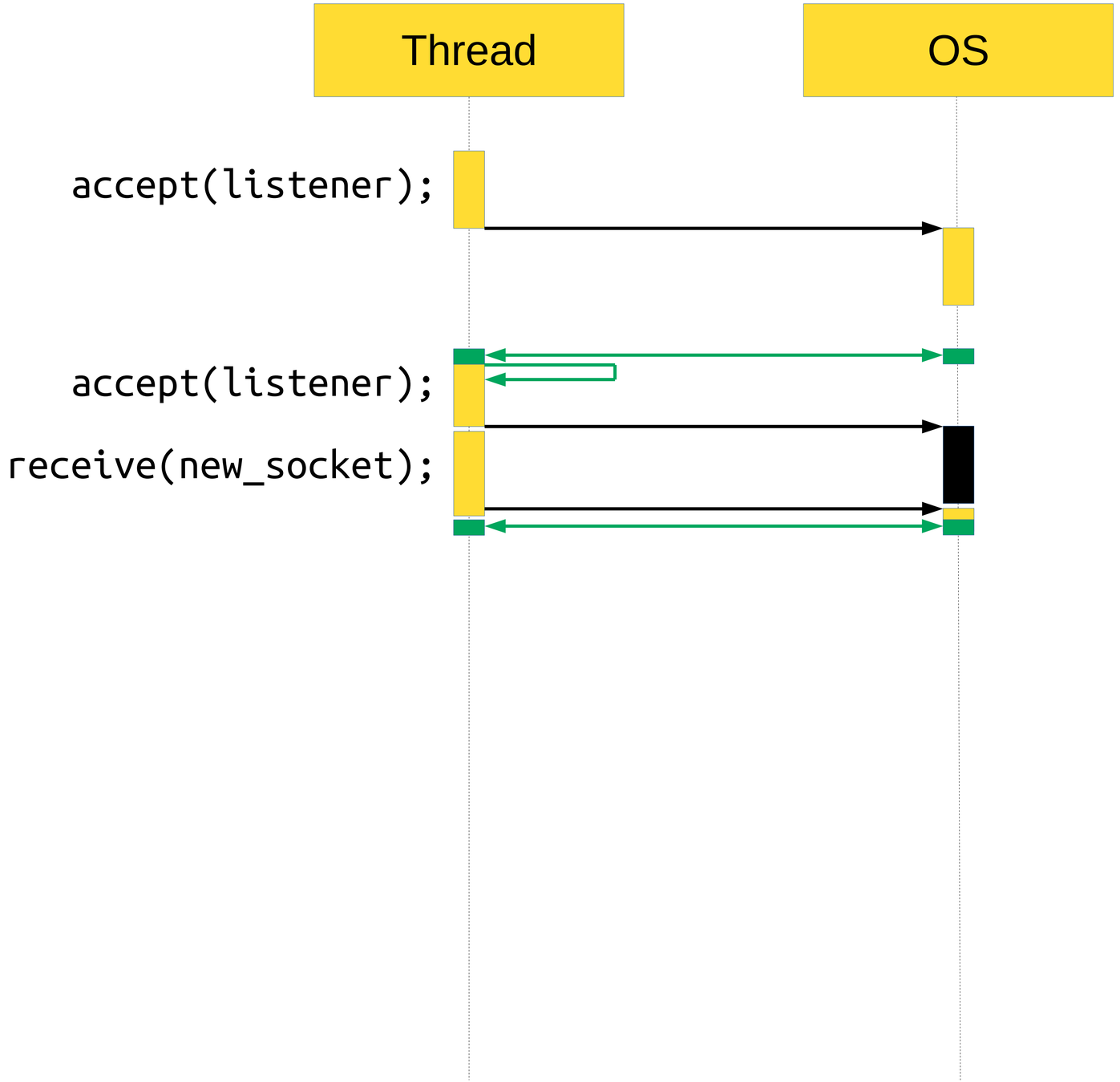

Thread
OS
accept(listener);

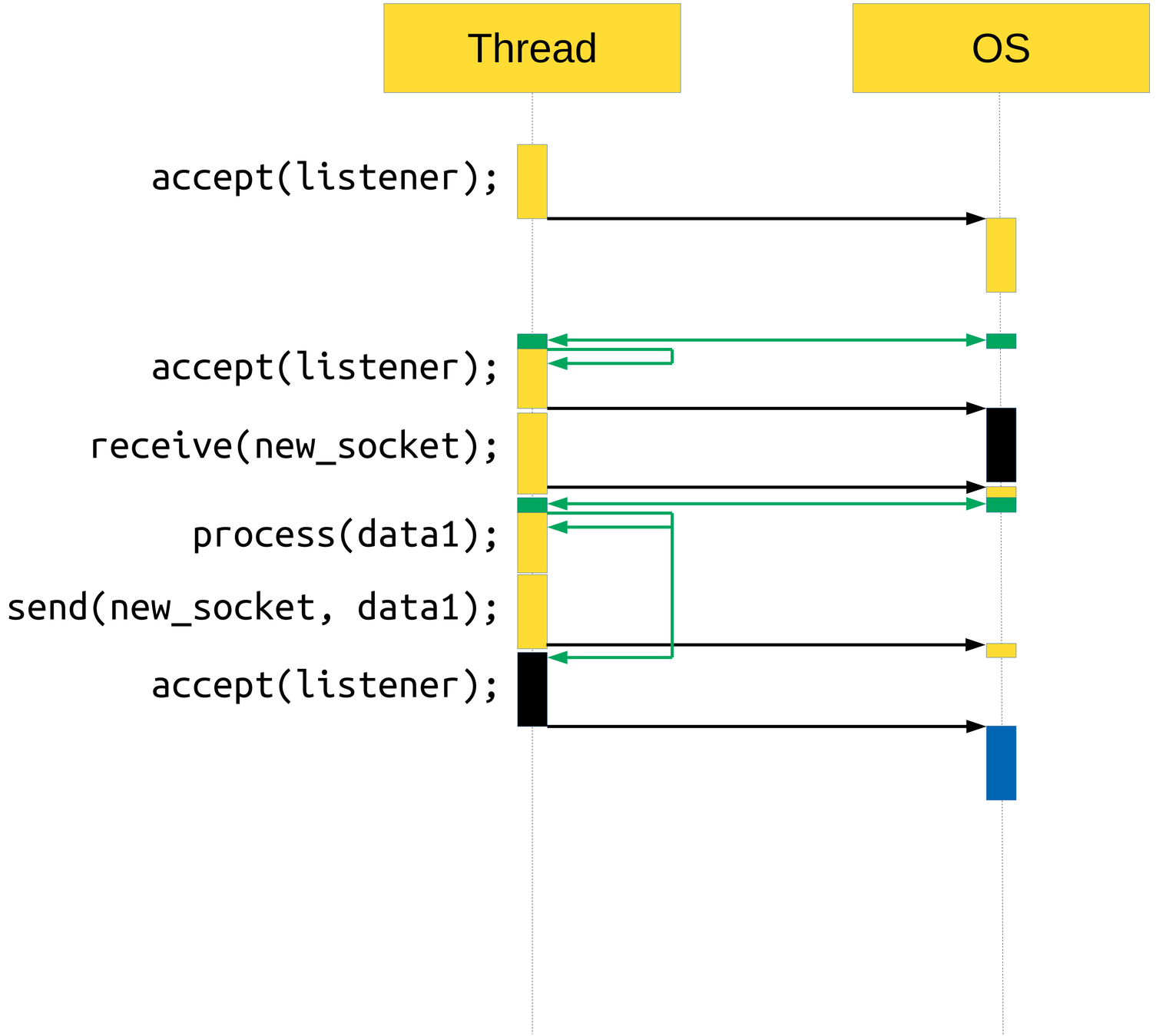

Thread
OS
accept(listener);

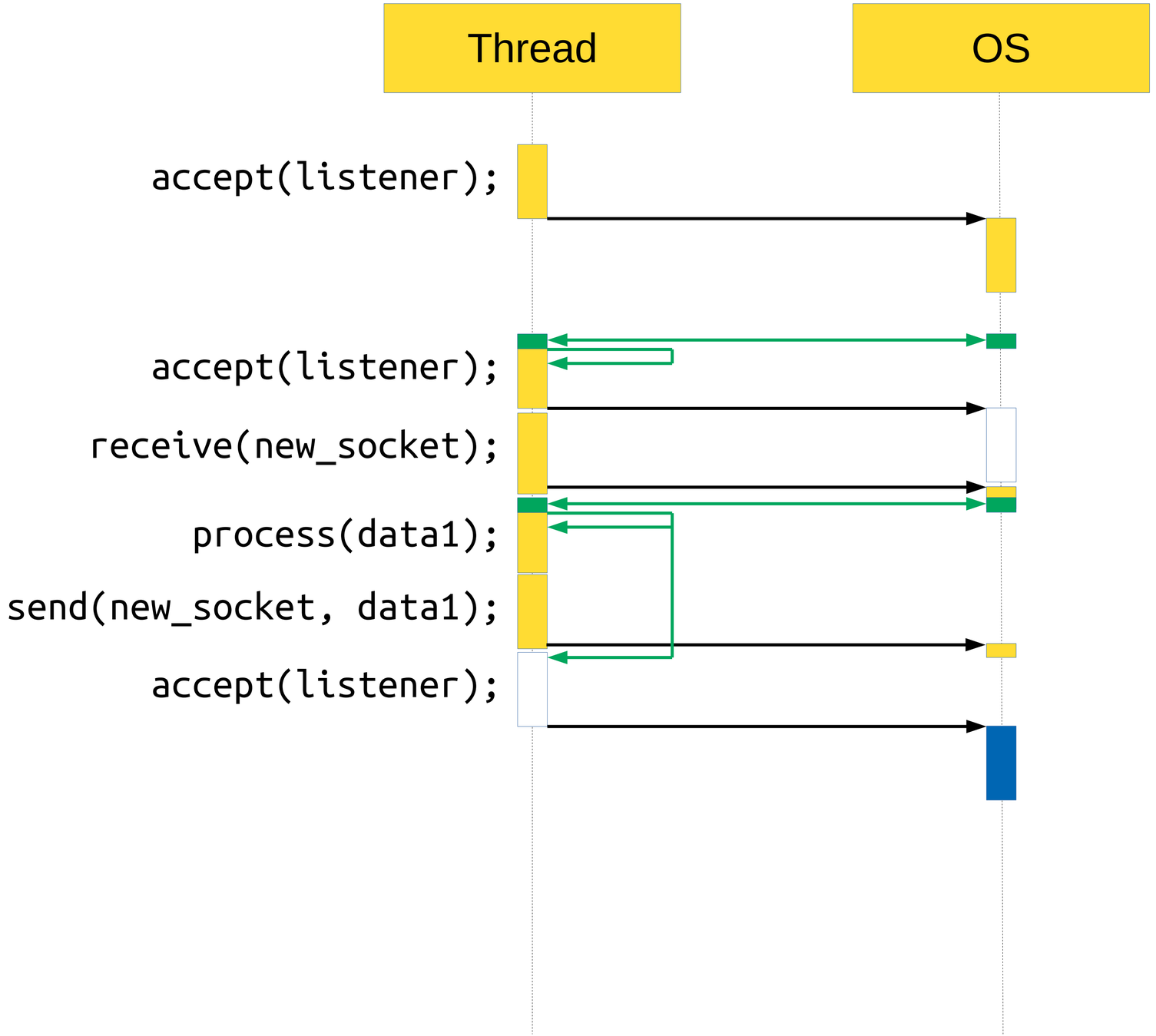

Thread
OS
accept(listener);
accept(listener);

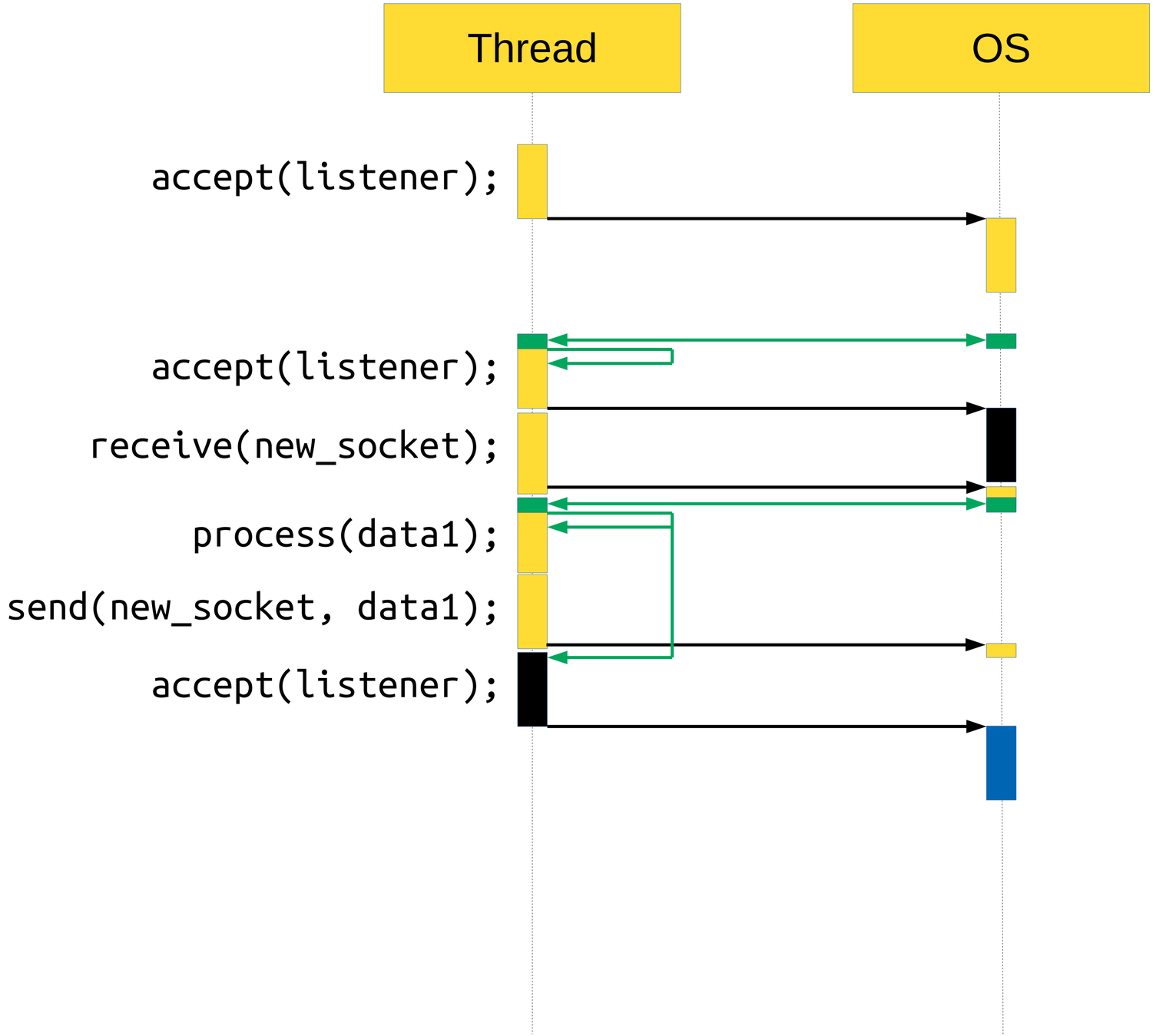

Thread
OS
accept(listener);
accept(listener);

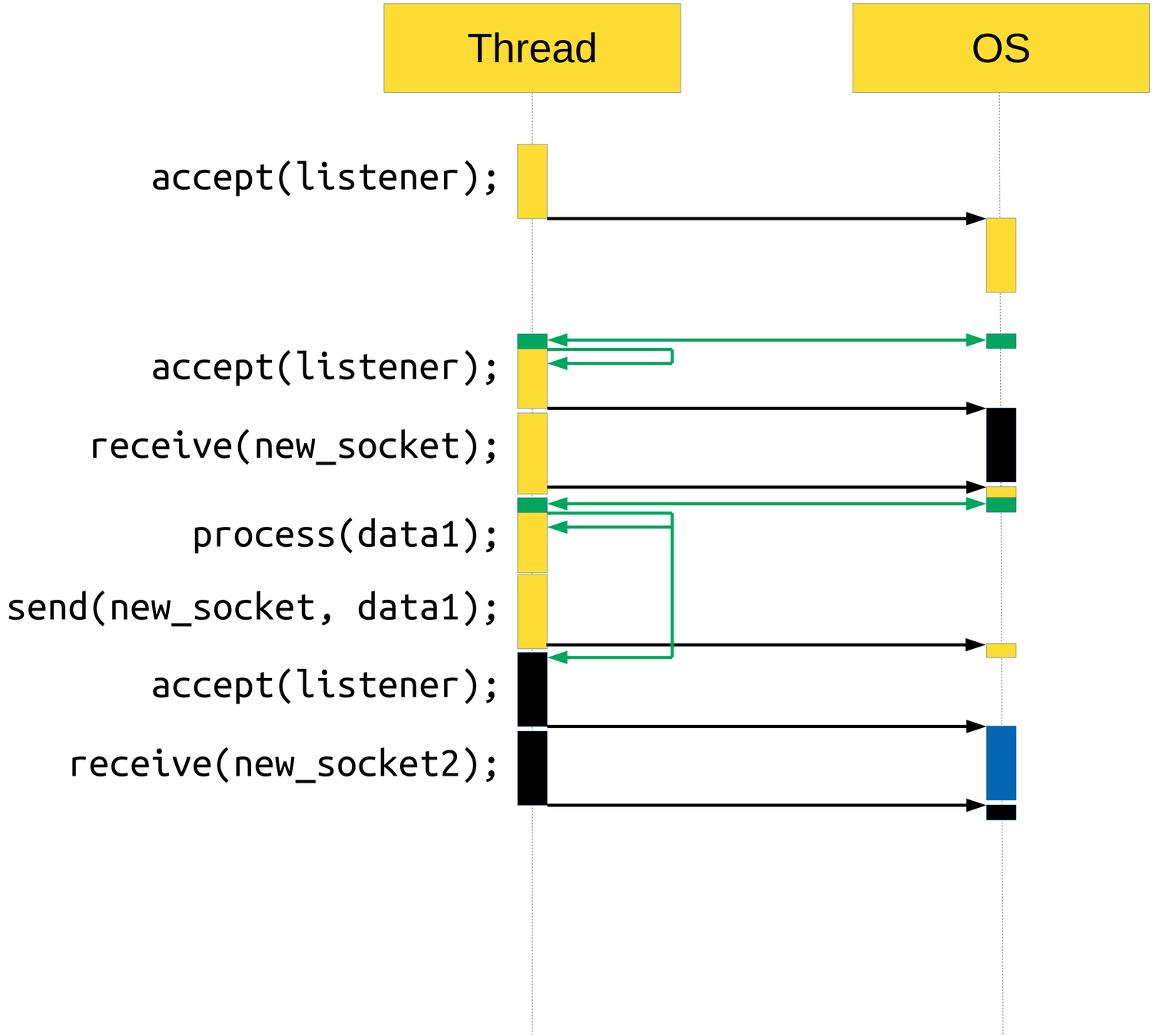

Thread
OS
accept(listener);
accept(listener);
receive(new_socket);


Thread
OS
accept(listener);
accept(listener);
receive(new_socket);


Thread
OS
accept(listener);
accept(listener);
receive(new_socket);
process(data1);
send(new_socket, data1);
accept(listener);


Thread
OS
accept(listener);
accept(listener);
receive(new_socket);
process(data1);
send(new_socket, data1);
accept(listener);


Thread
OS
accept(listener);
accept(listener);
receive(new_socket);
process(data1);
send(new_socket, data1);
accept(listener);


Thread
OS
accept(listener);
accept(listener);
receive(new_socket);
process(data1);
send(new_socket, data1);
accept(listener);
receive(new_socket2);

Есть ньюанс…

Запуск пользовательских задач

```
void async_accept() {
  accept(listener, [](socket_t socket) {
    async_accept();
    socket.receive(
        [socket](std::vector<unsigned char> data) {
          auto task = Async(process1, data);
          process(data);
          task.wait();
          socket.send(data, kNoCallback);
        });
  });
}
```

Запуск пользовательских задач

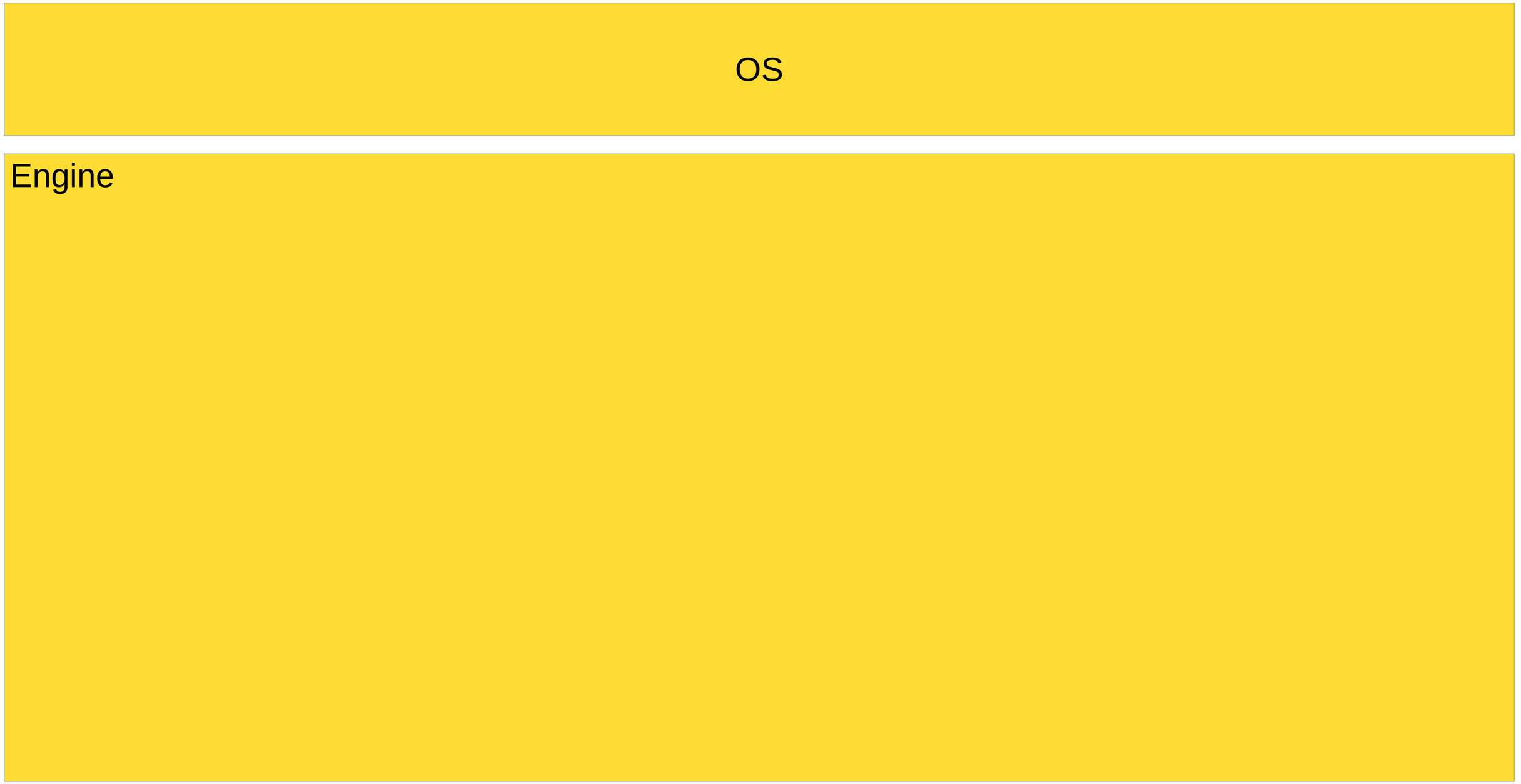
```
void async_accept() {
  accept(listener, [](socket_t socket) {
    async_accept();
    socket.receive(
        [socket](std::vector<unsigned char> data) {
          auto task = Async(process1, data);
          process(data);
          task.wait();
          socket.send(data, kNoCallback);
        });
  });
}
```

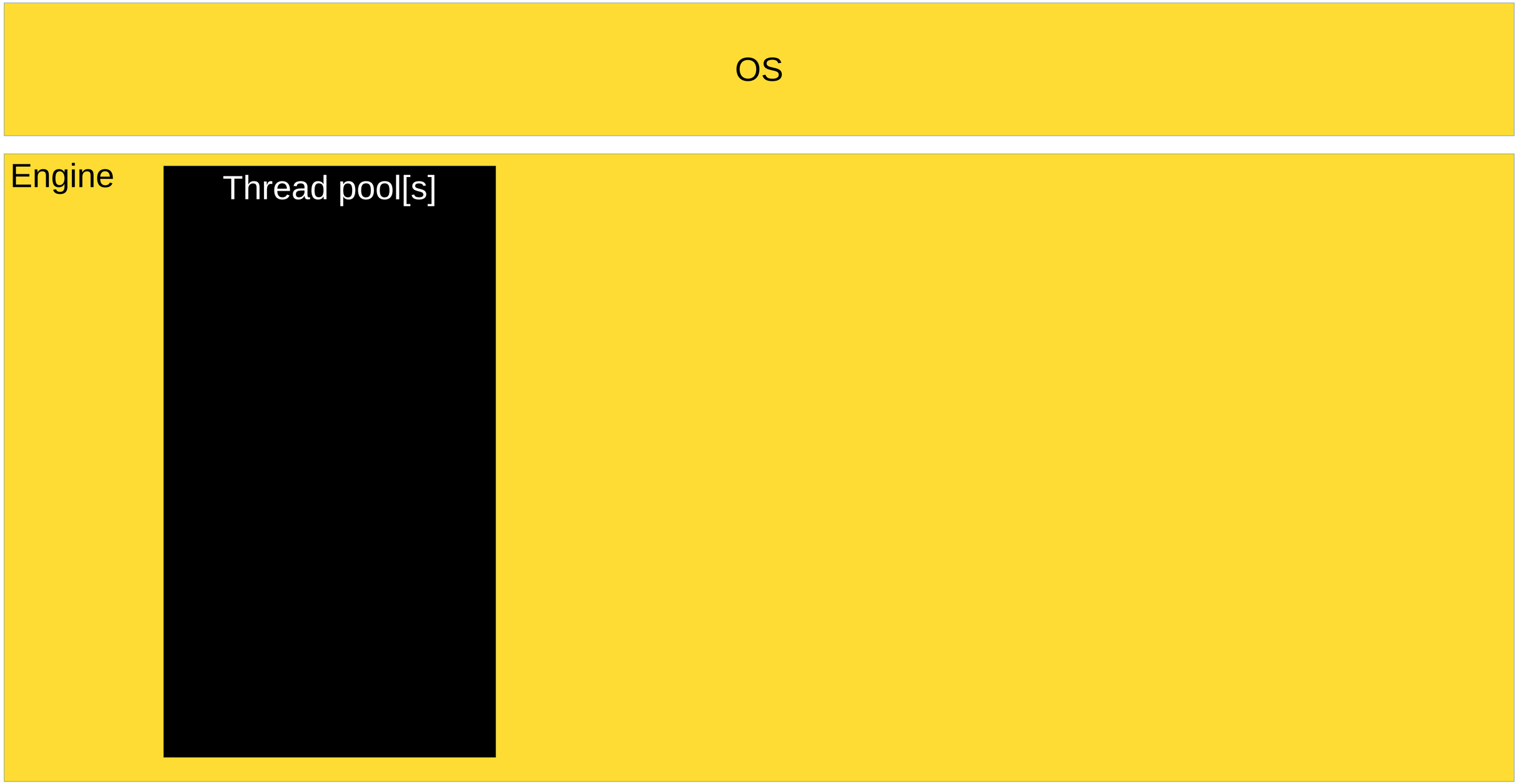
Запуск пользовательских задач

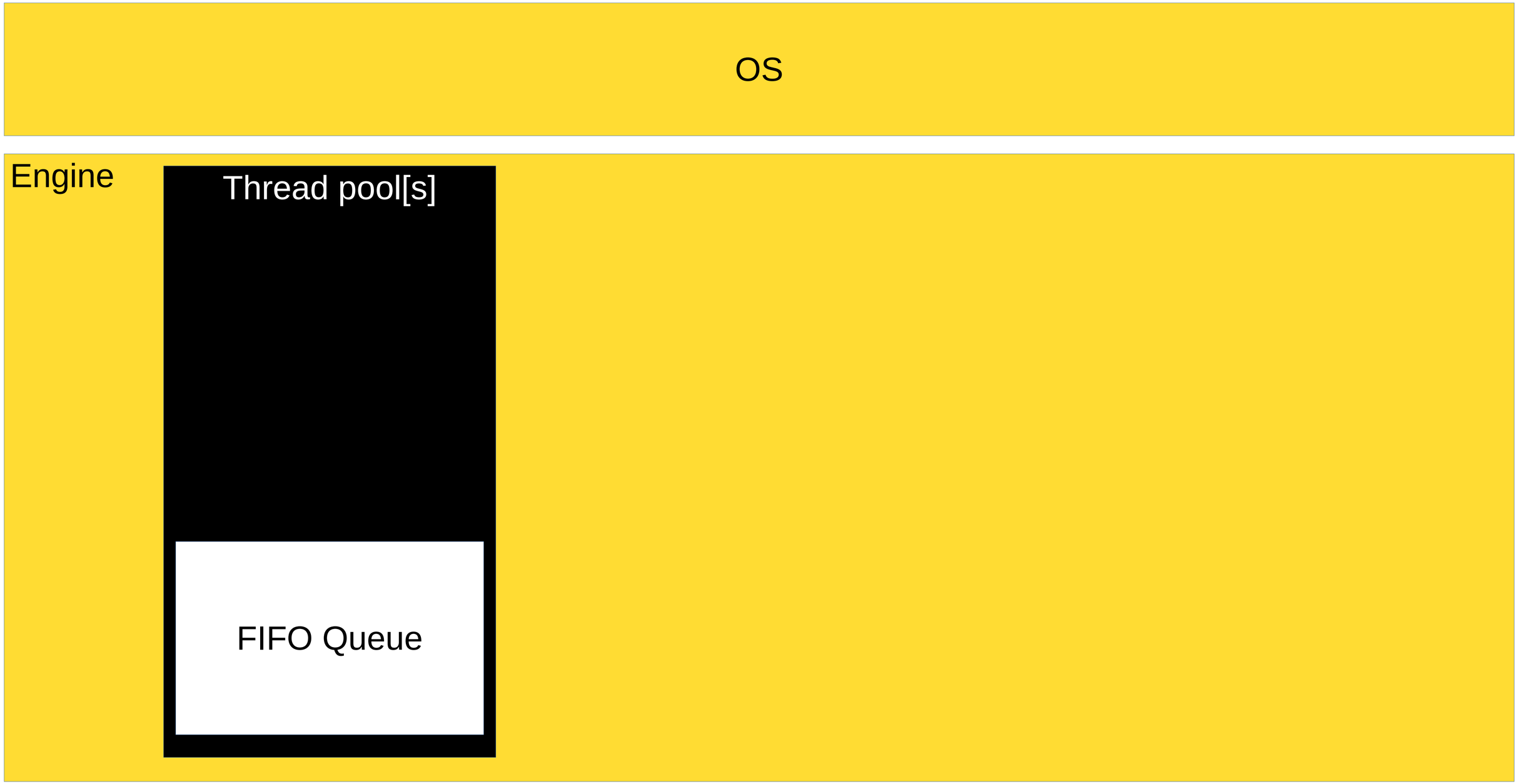
```
void async_accept() {
  accept(listener, [](socket_t socket) {
    async_accept();
    socket.receive(
        [socket](std::vector<unsigned char> data) {
          auto task = Async(process1, data);
          process(data);
          task.wait();
          socket.send(data, kNoCallback);
        });
  });
}
```

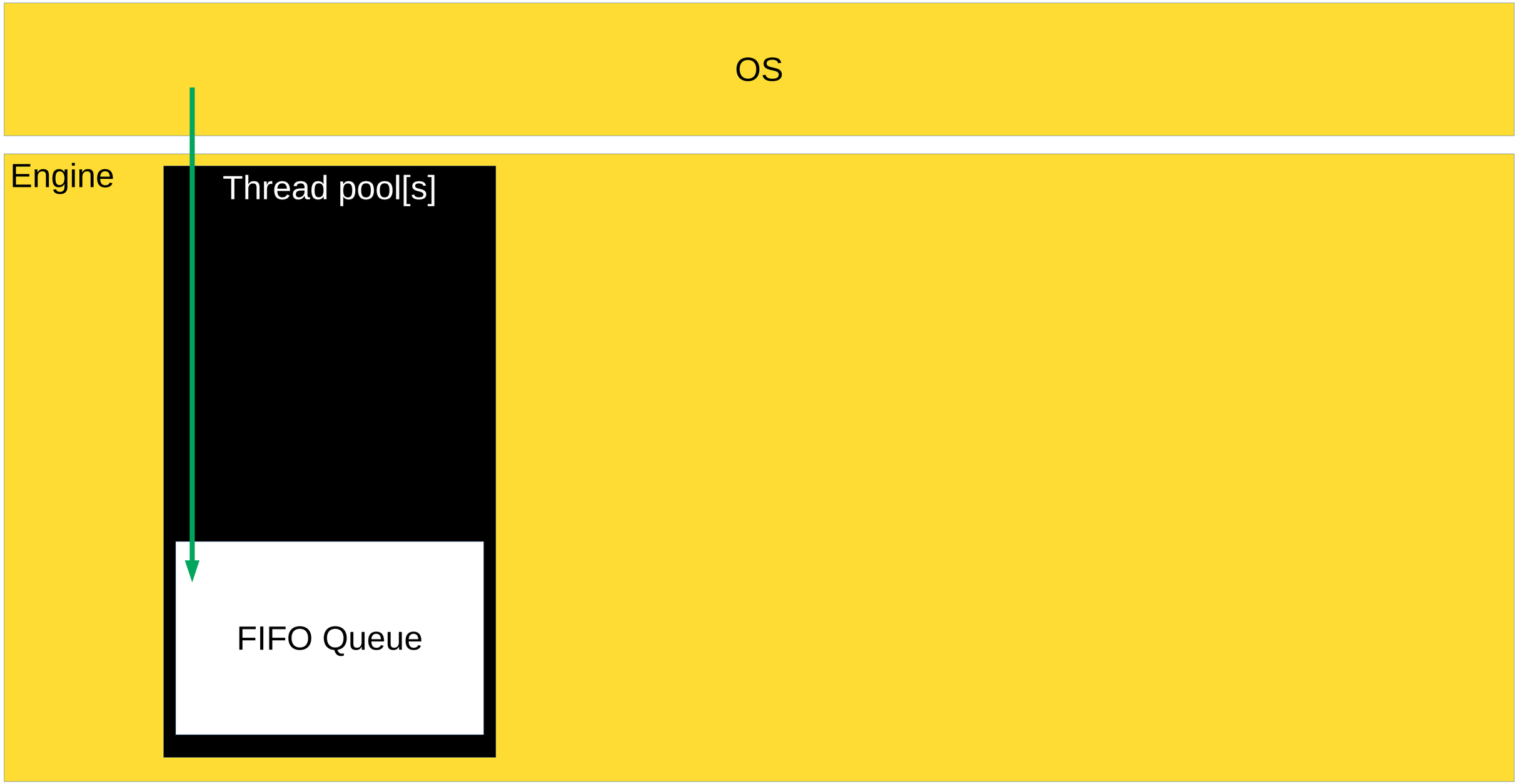
Нужны очереди

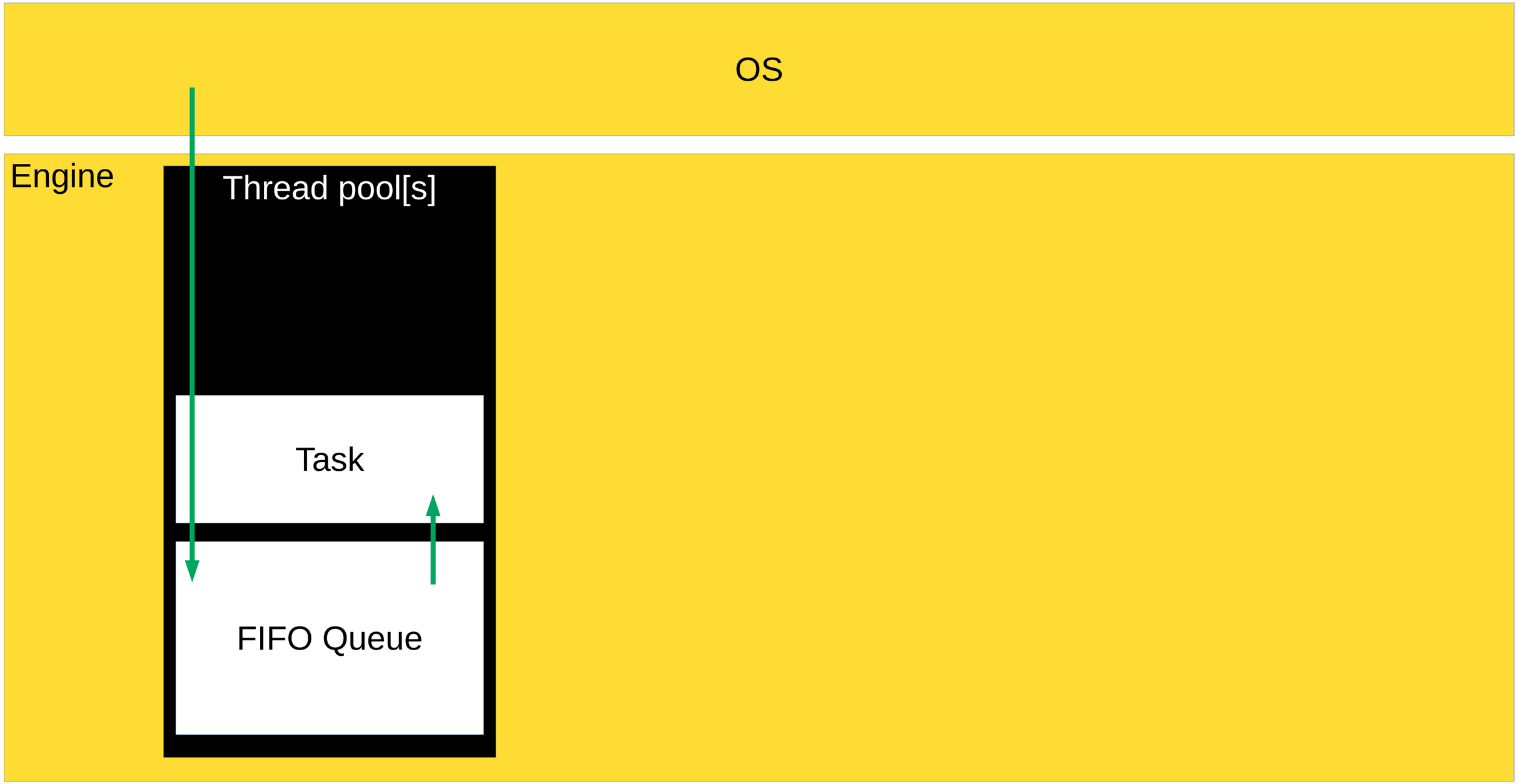

OS

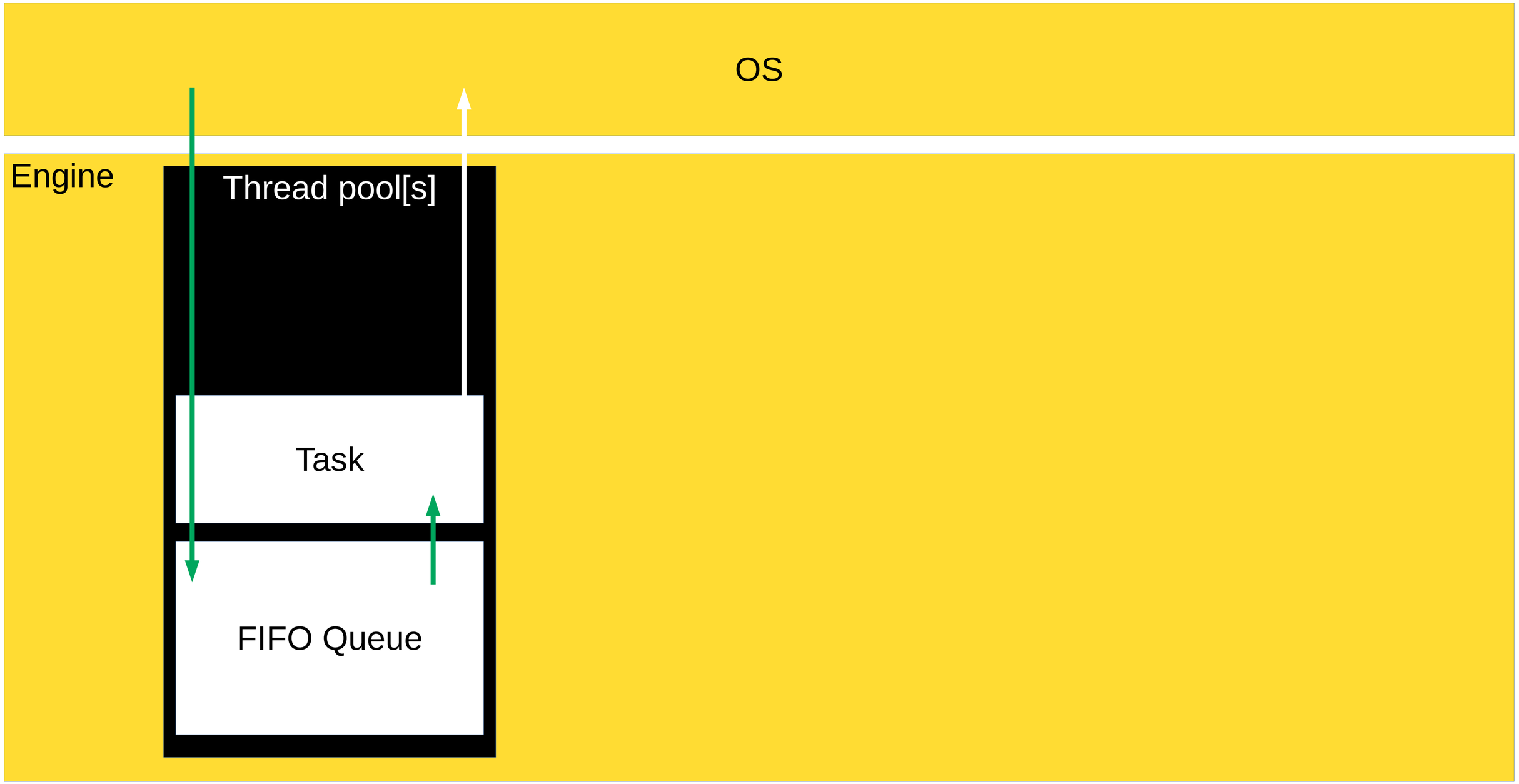

OS
Engine

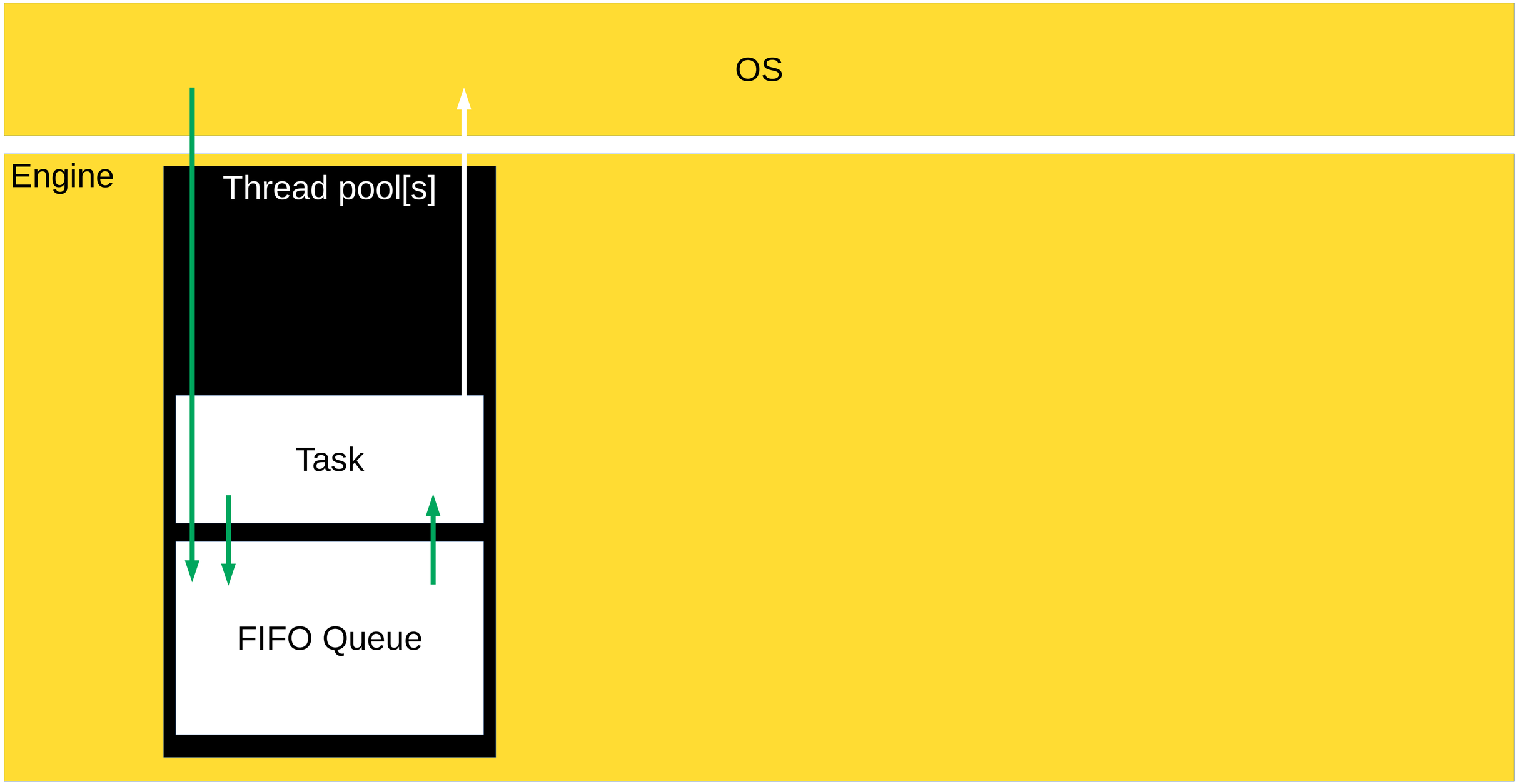

OS
Engine
Thread pool[s]

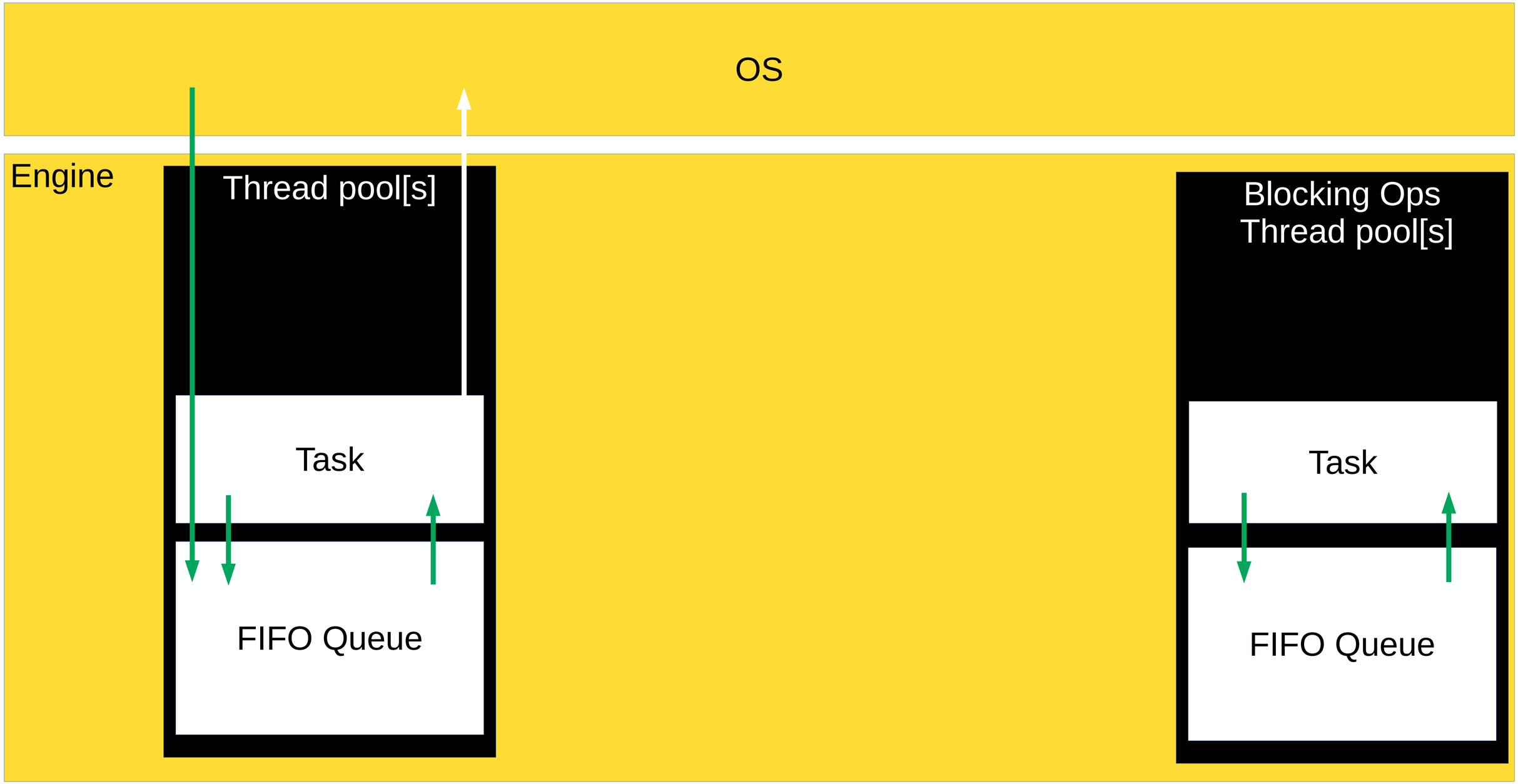

OS
Engine
Thread pool[s]
FIFO Queue

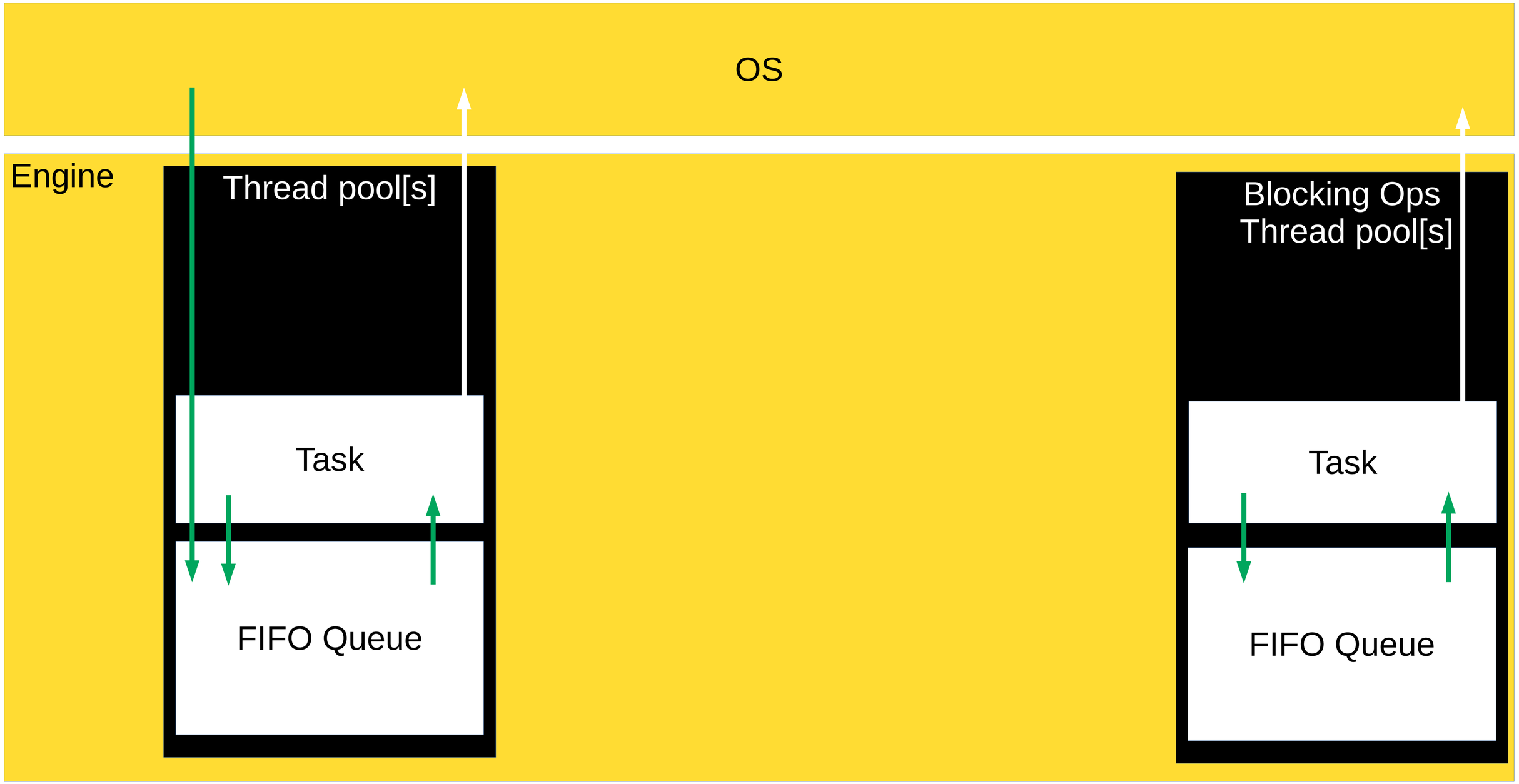

OS
Engine
Thread pool[s]
FIFO Queue

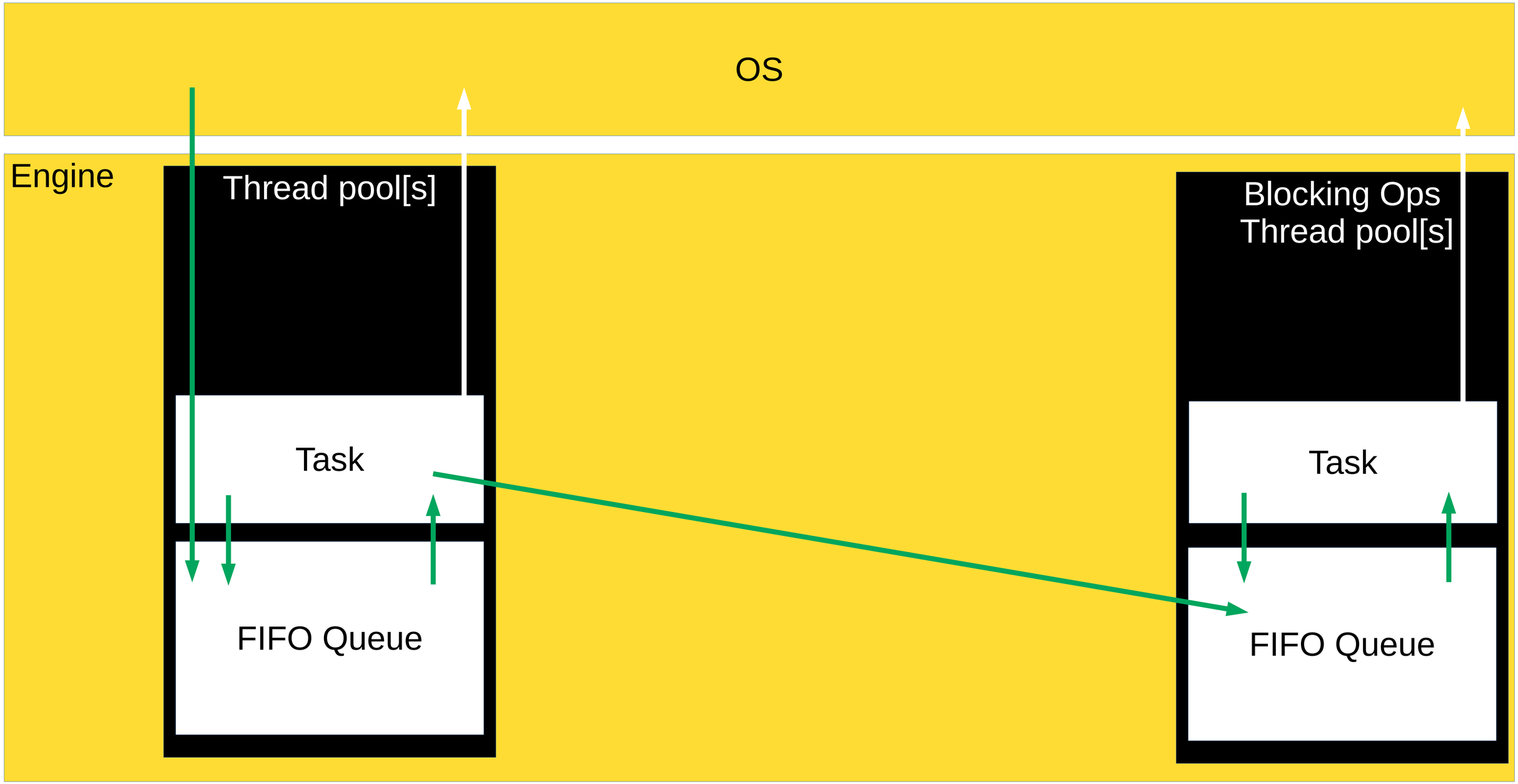

OS
Engine
Thread pool[s]
Task
FIFO Queue

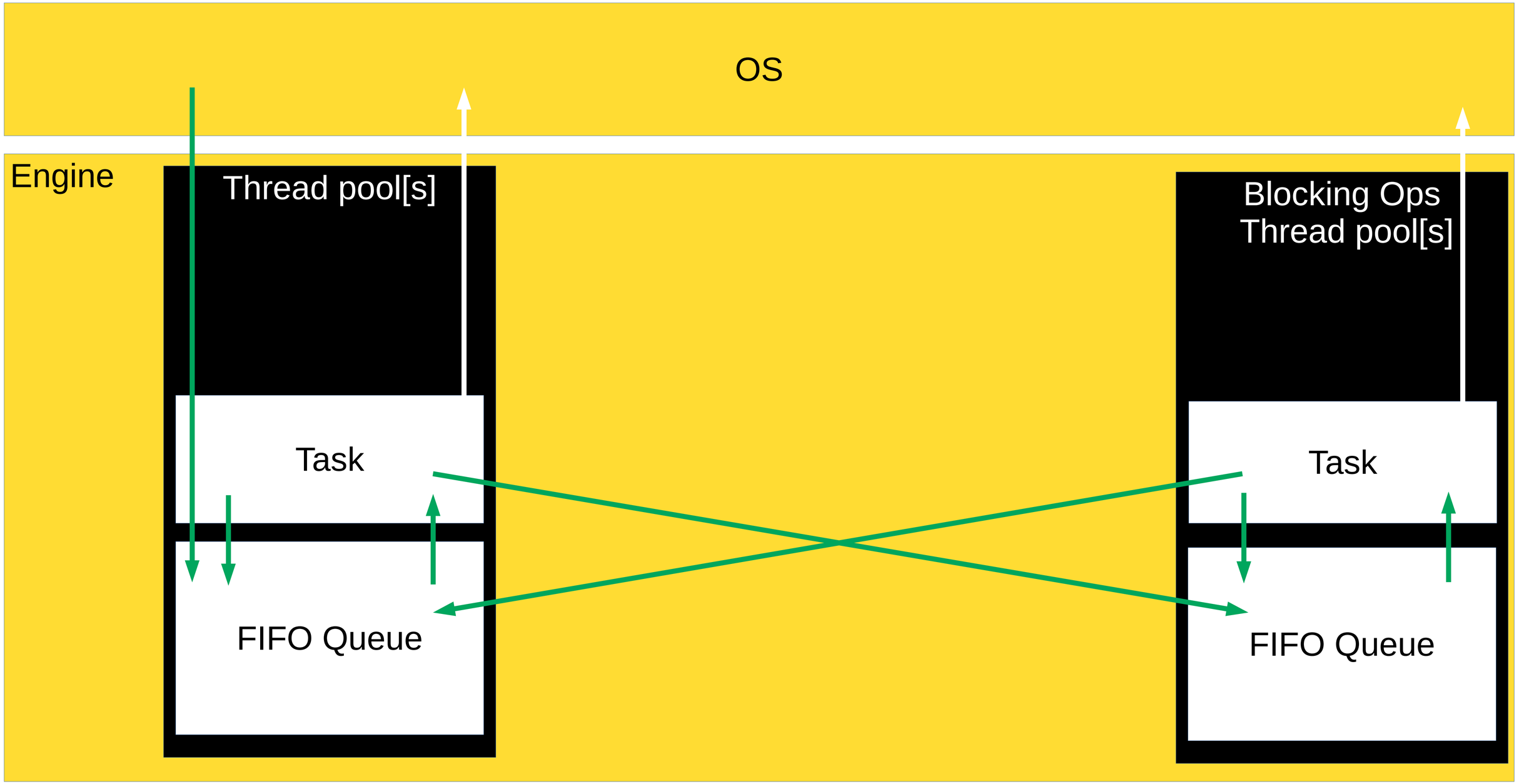

OS
Engine
Thread pool[s]
Task
FIFO Queue


OS
Engine
Thread pool[s]
Task
FIFO Queue

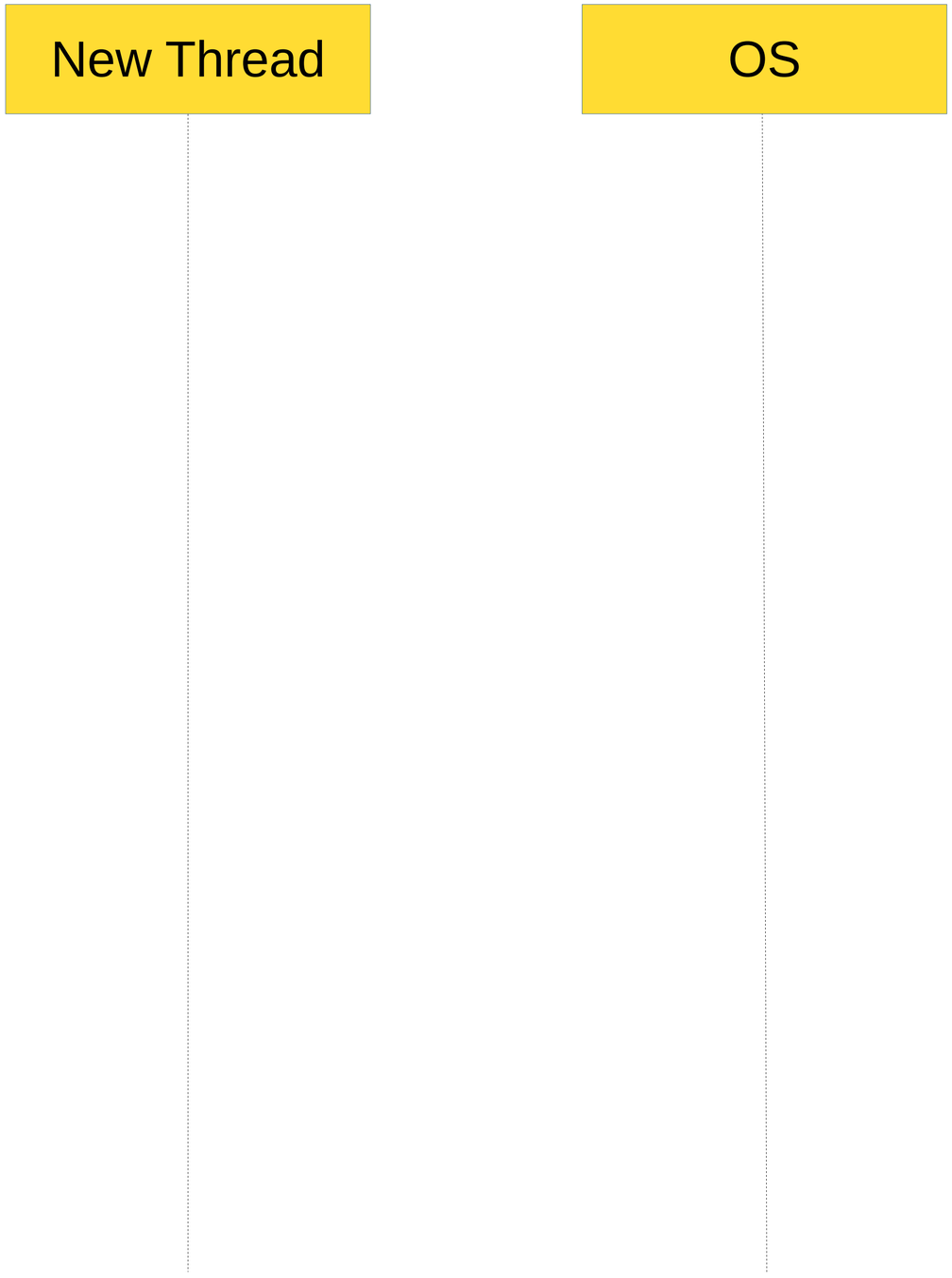

OS
Engine
Thread pool[s]
Task
FIFO Queue
Blocking Ops
Thread pool[s]
Task
FIFO Queue

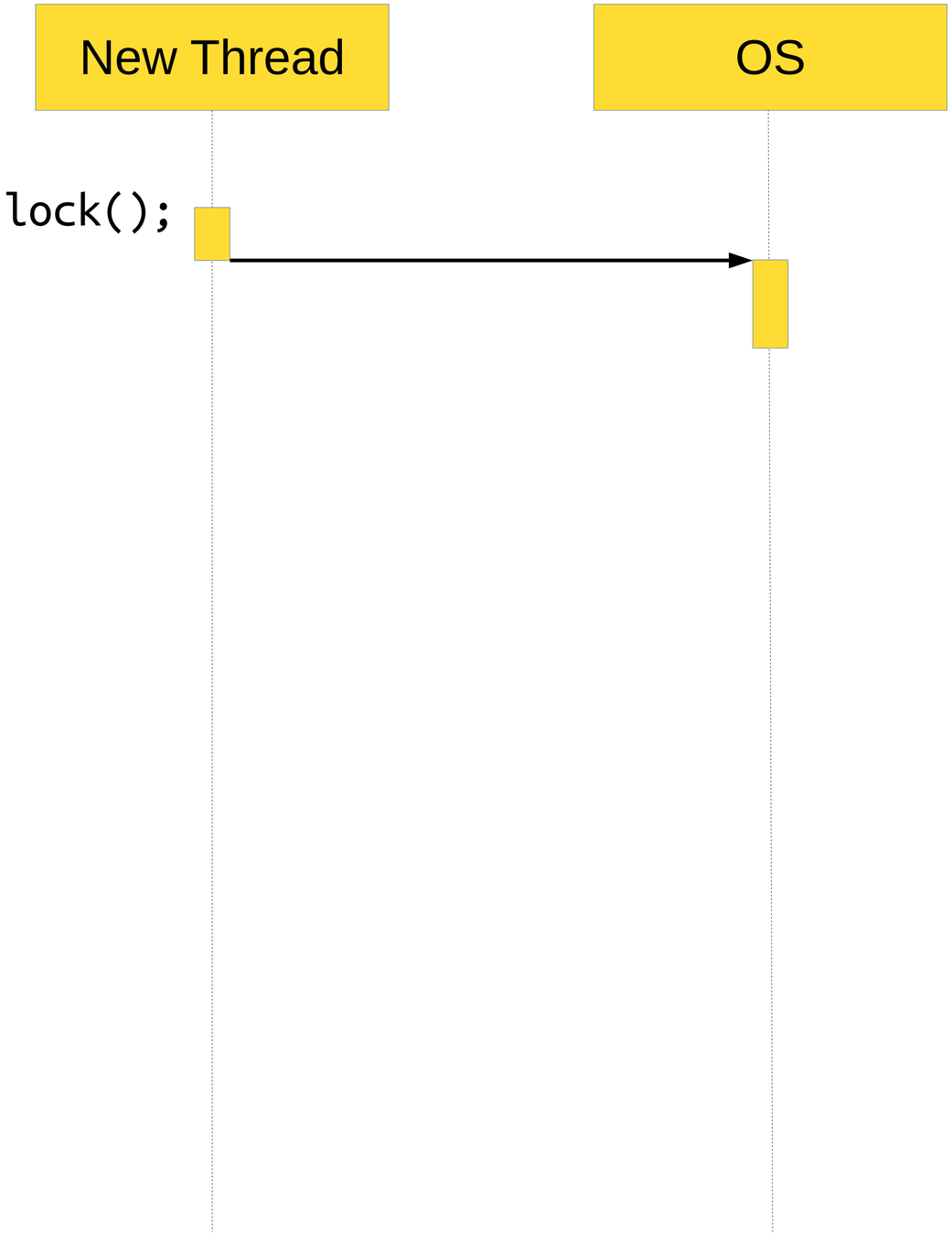

OS
Engine
Thread pool[s]
Task
FIFO Queue
Blocking Ops
Thread pool[s]
Task
FIFO Queue

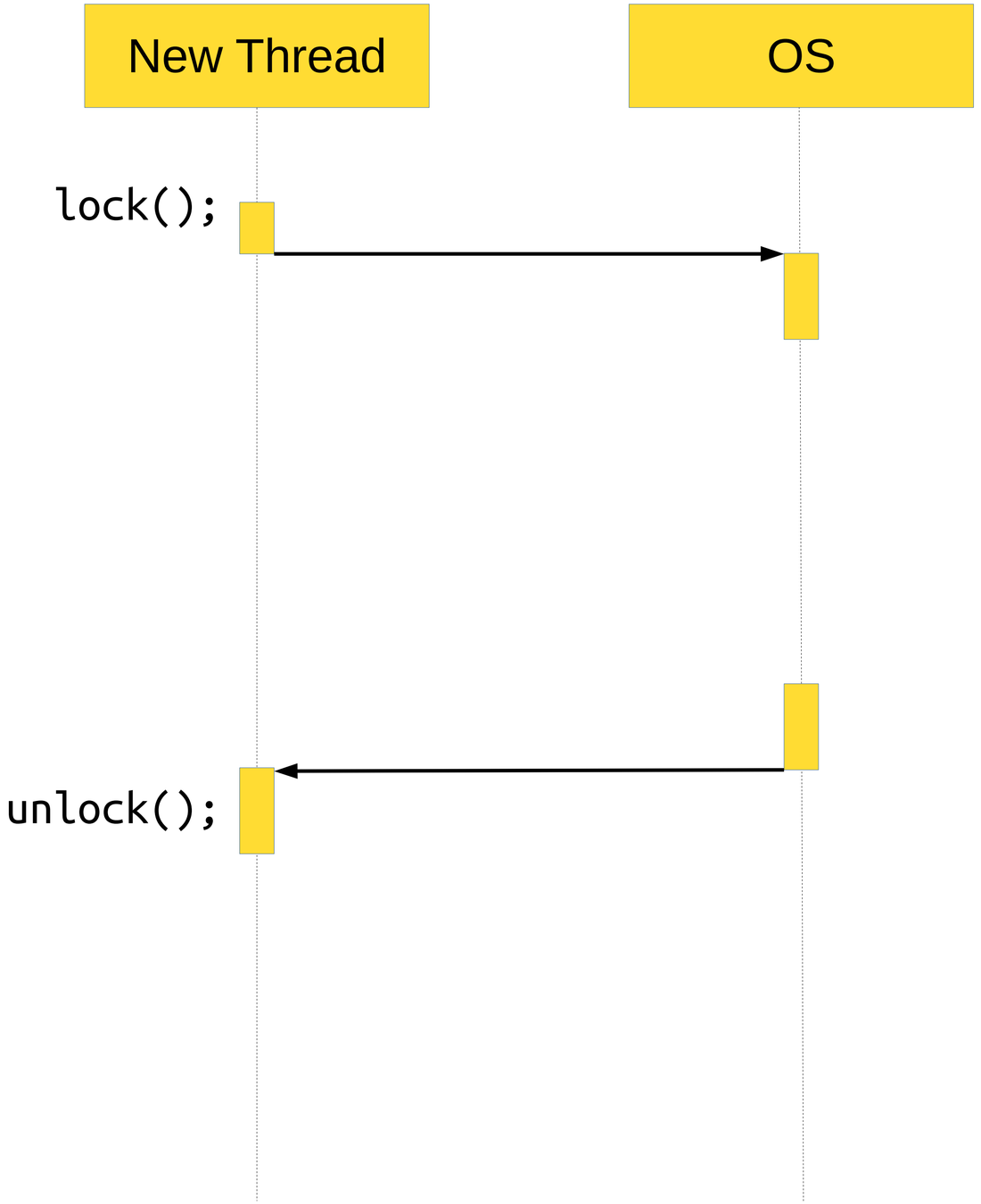

OS
Engine
Thread pool[s]
Task
FIFO Queue
Blocking Ops
Thread pool[s]
Task
FIFO Queue


OS
Engine
Thread pool[s]
Task
FIFO Queue
Blocking Ops
Thread pool[s]
Task
FIFO Queue

А что с синхронизацией?


New Thread
OS


New Thread
OS
lock();


New Thread
OS
lock();
unlock();

Мьютекс

```
void async_accept_lock() {
  accept(listener, [](socket_t socket) {
    async_accept_lock();
    socket.receive(
        [socket](std::vector<unsigned char> data) mutable {
          mutex.lock([data = std::move(data), socket = std::move(socket)]() {
            process2(shared_resource, data);
            socket.send(data, kNoCallback);
          });
        });
  });
}
```

Мьютекс

```
void async_accept_lock() {
  accept(listener, [](socket_t socket) {
    async_accept_lock();
    socket.receive(
        [socket](std::vector<unsigned char> data) mutable {
          mutex.lock([data = std::move(data), socket = std::move(socket)]() {
            process2(shared_resource, data);
            socket.send(data, kNoCallback);
          });
        });
  });
}
```

Внутри мьютекса

```
template <class Functor>
void lock(Functor f) {
  auto lock = this->try_lock();
  if (lock) {
    f();
  } else {
    wait_for_unlock(std::move(f));
  }
}
```

Внутри мьютекса

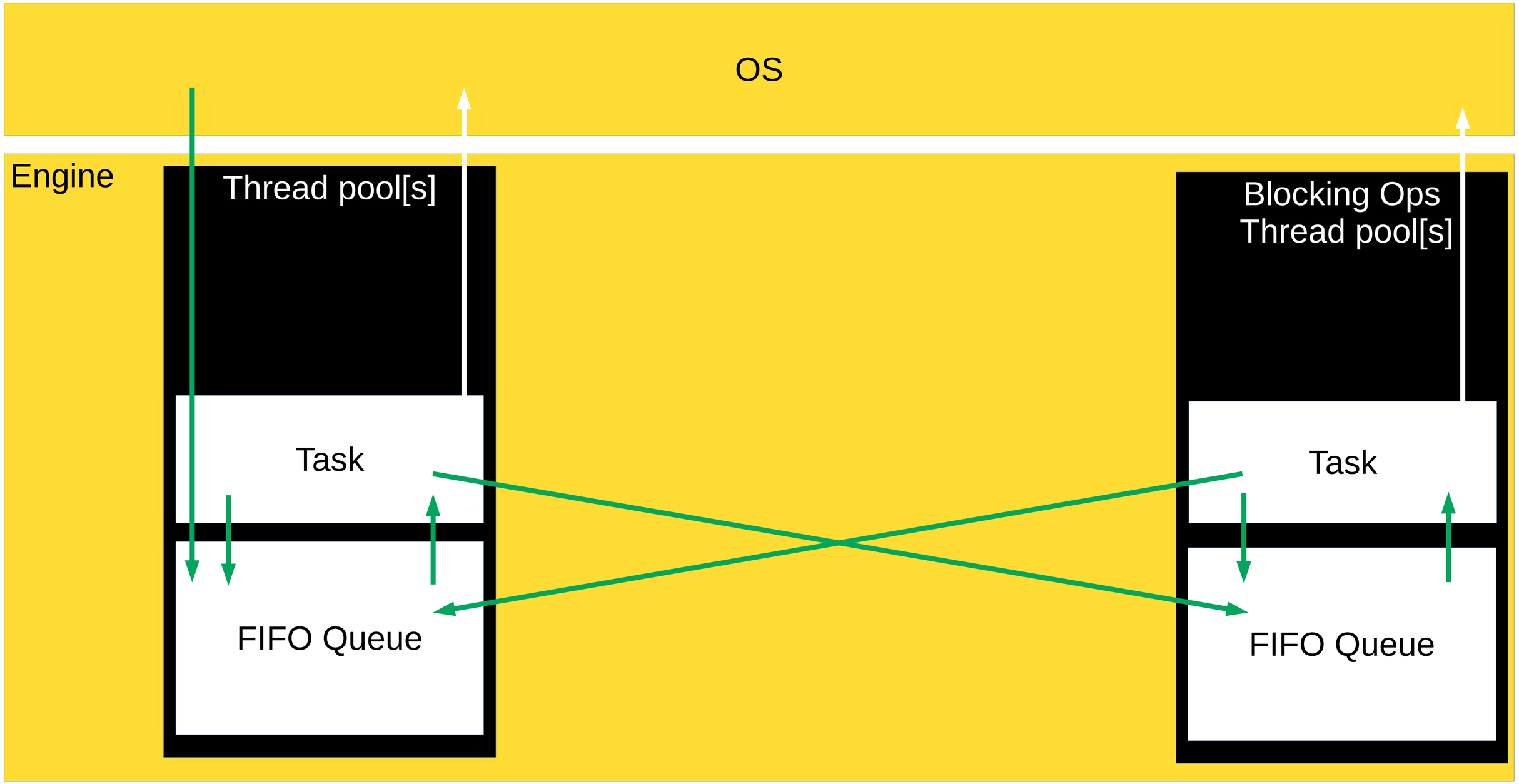
```
template <class Functor>
void lock(Functor f) {
  auto lock = this->try_lock();
  if (lock) {
    f();
  } else {
    wait_for_unlock(std::move(f));
  }
}
```

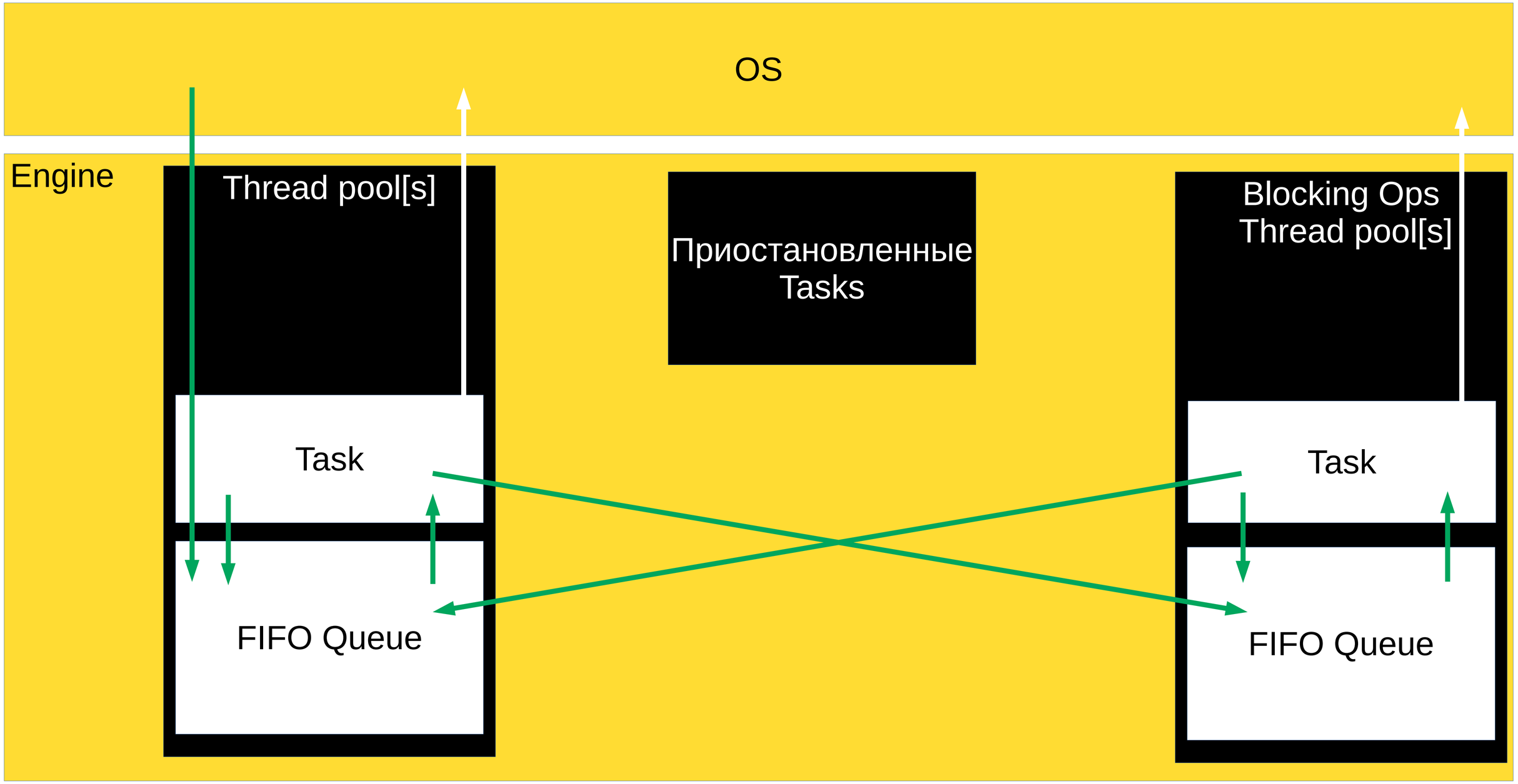
Внутри мьютекса

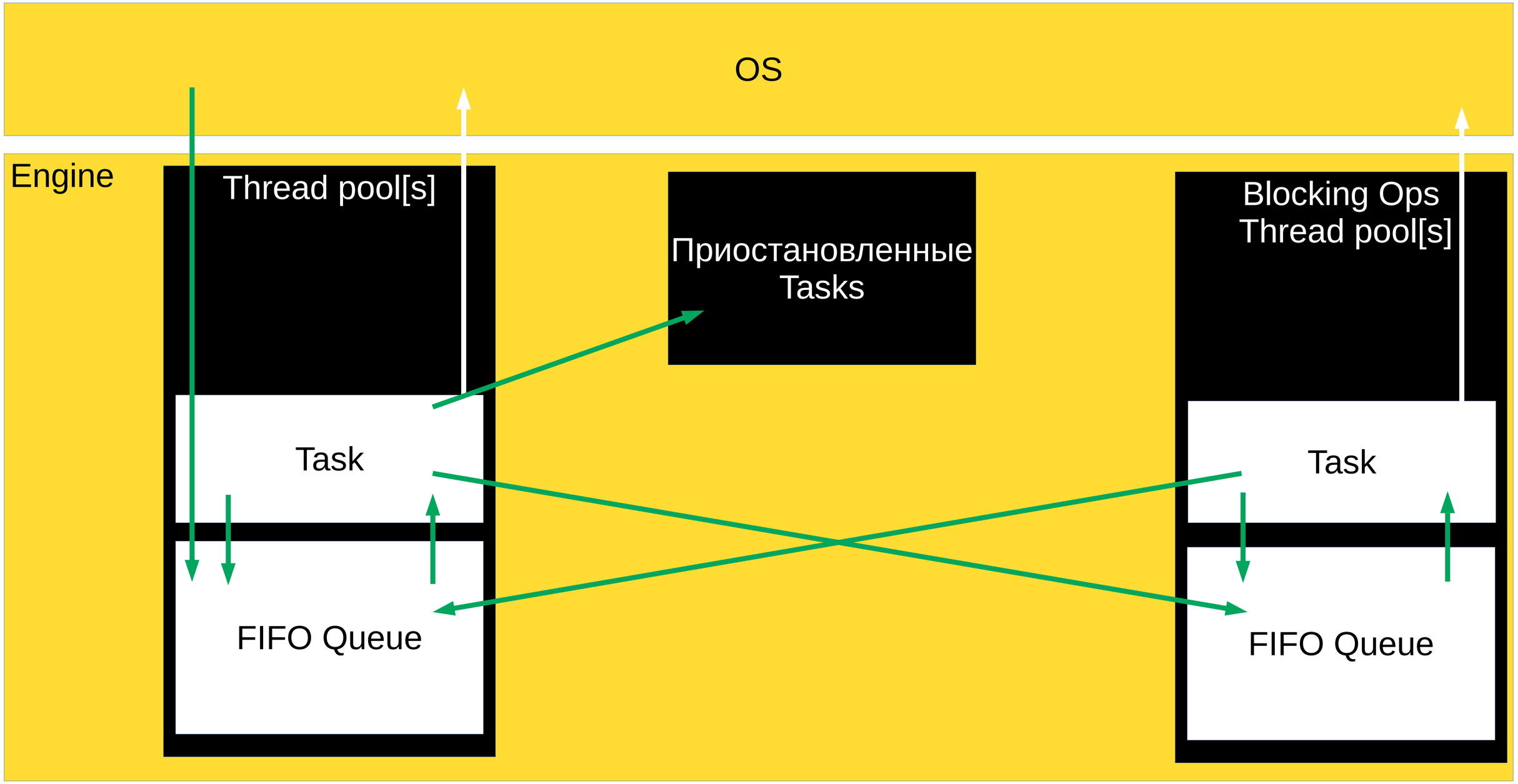
```
template <class Functor>
void lock(Functor f) {
  auto lock = this->try_lock();
  if (lock) {
    f();
  } else {
    wait_for_unlock(std::move(f));
  }
}
```

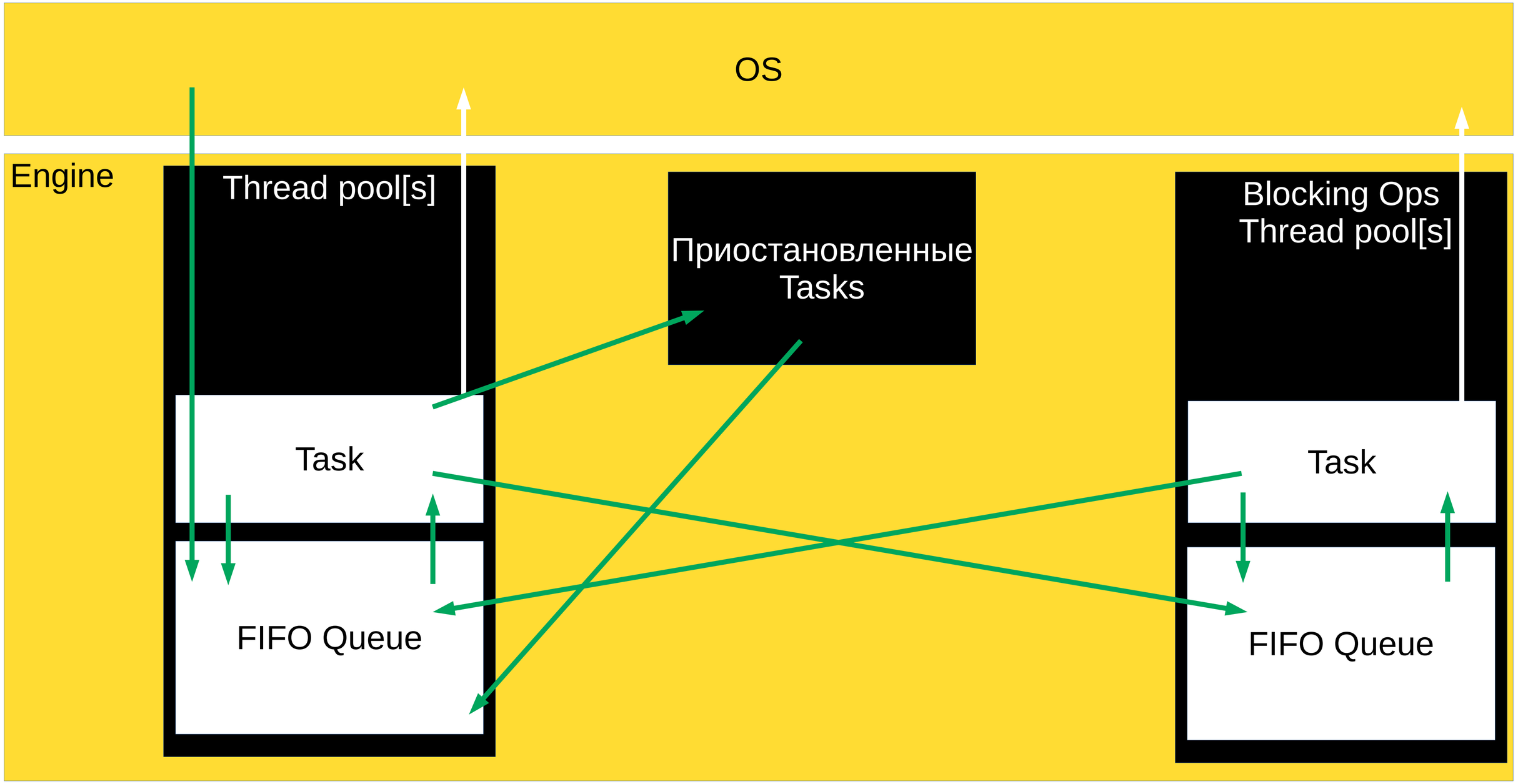
Внутри мьютекса

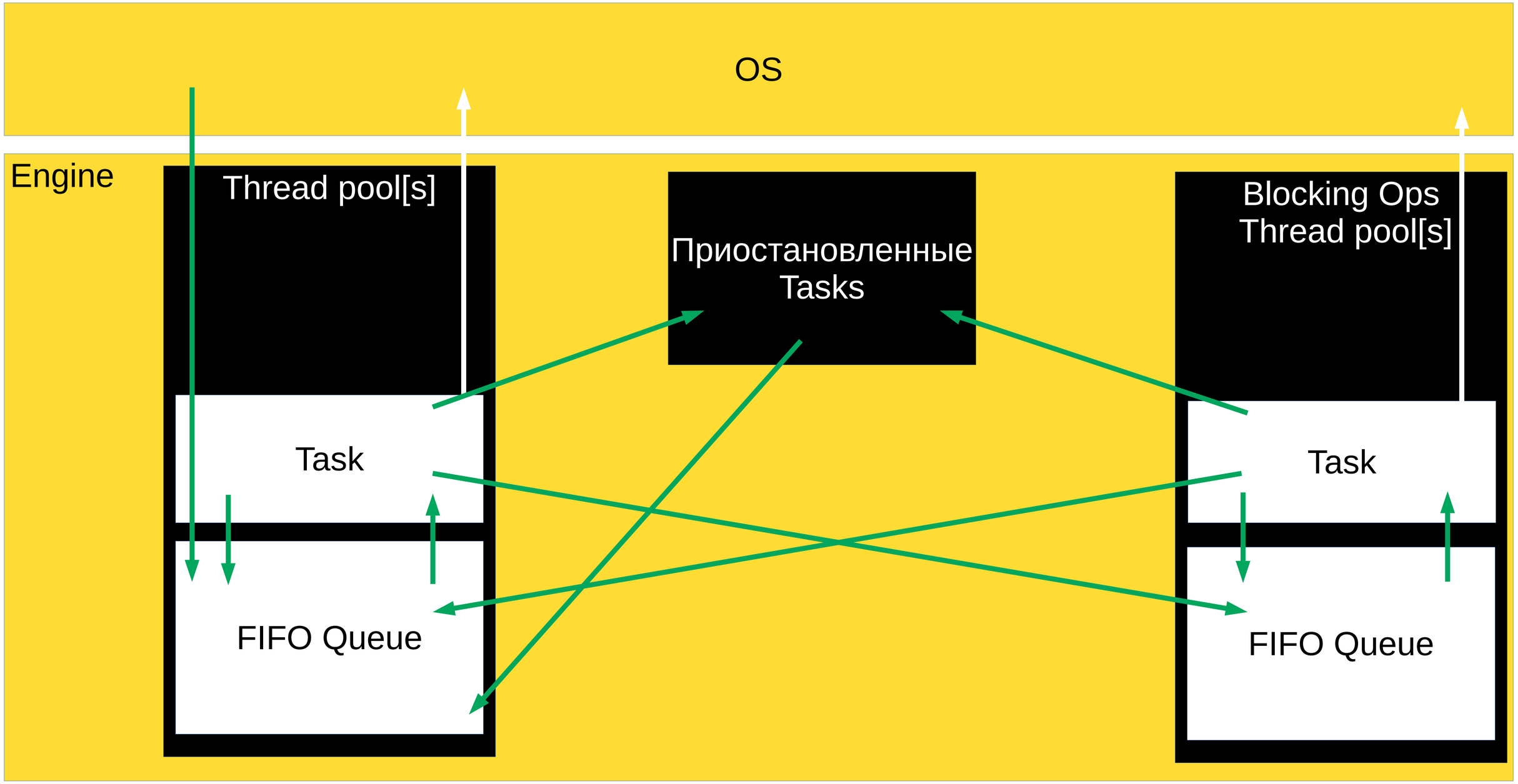
```
template <class Functor>
void lock(Functor f) {
  auto lock = this->try_lock();
  if (lock) {
    f();
  } else {
    wait_for_unlock(std::move(f));
  }
}
```

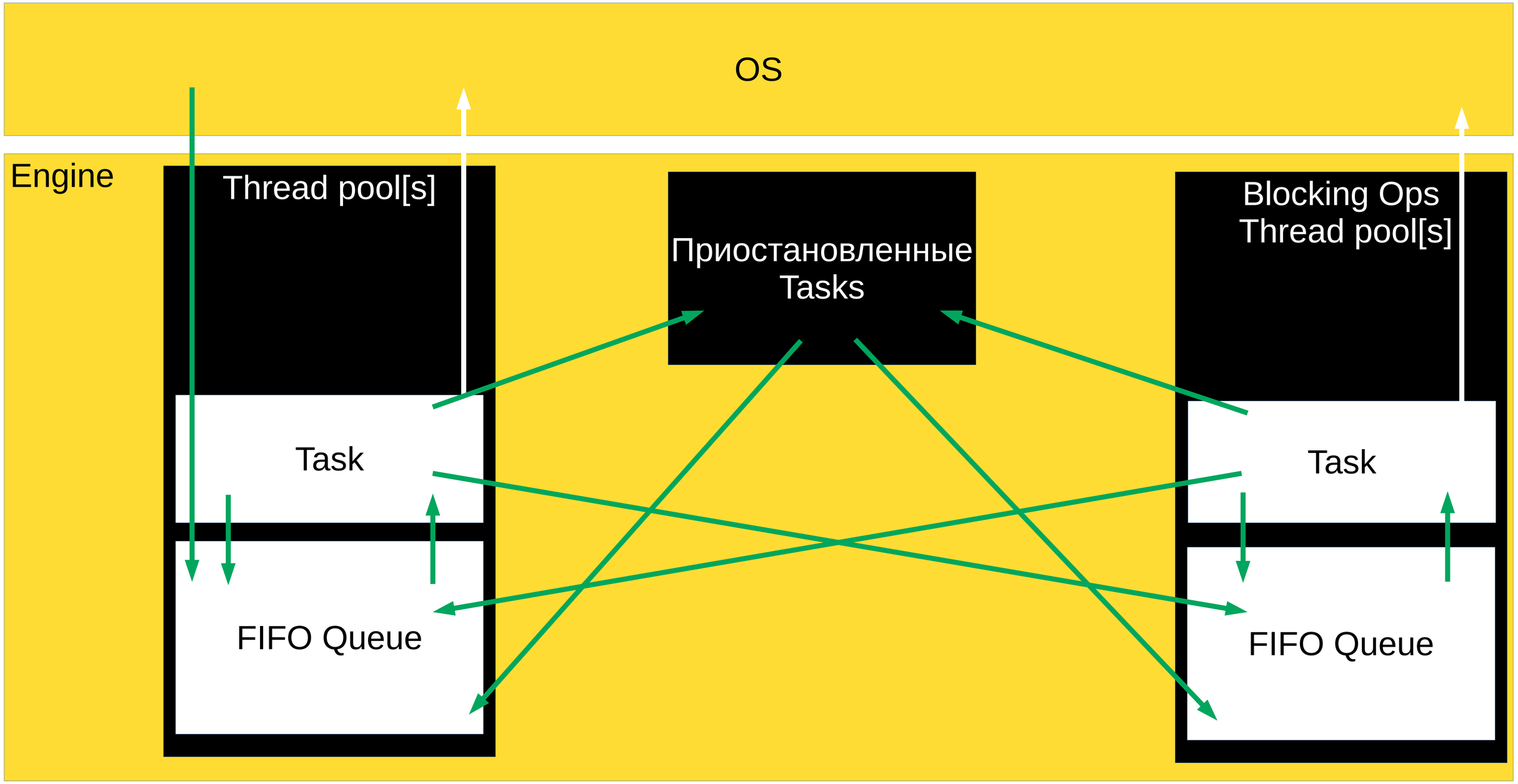

OS
Engine
Thread pool[s]
Task
FIFO Queue
Blocking Ops
Thread pool[s]
Task
FIFO Queue


OS
Engine
Thread pool[s]
Task
FIFO Queue
Приостановленные
Tasks
Blocking Ops
Thread pool[s]
Task
FIFO Queue


OS
Engine
Thread pool[s]
Приостановленные Tasks
Blocking Ops Thread pool[s]
Task
Task
FIFO Queue
FIFO Queue


OS
Engine
Thread pool[s]
Task
FIFO Queue
Приостановленные
Tasks
Blocking Ops
Thread pool[s]
Task
FIFO Queue


OS
Engine
Thread pool[s]
Task
FIFO Queue
Приостановленные
Tasks
Blocking Ops
Thread pool[s]
Task
FIFO Queue


OS
Engine
Thread pool[s]
Приостановленные Tasks
Blocking Ops Thread pool[s]
Task
FIFO Queue
Task
FIFO Queue

Плюсы/минусы асинхронности

Плюсы/минусы асинхронности

Плюсы:

Плюсы/минусы асинхронности

Плюсы:

- Всё очень эффективно

Плюсы/минусы асинхронности

Плюсы:

- Всё очень эффективно

Минусы:

Ну давай, прочти меня!

```
void async_accept() {
  accept(listener, [](socket_t socket) {
    async_accept();
    auto something = Async(process1, {42});
    auto& socket_ref = *socket; socket_ref.receive(
        [socket = std::move(socket), something = std::move(something)]
(std::vector<unsigned char> data) mutable {
          auto task = Async(process1, data);
          process(data);
          task.wait();
          auto& socket_ref = *socket; socket_ref.send(data, [data, socket =
std::move(socket), something = std::move(something)]() mutable {
              mutex.lock([data = std::move(data), socket = std::move(socket), something =
std::move(something)]() mutable {
                process2(shared_resource, data);
                socket->send(data, kNoCallback);
              });
          });
    });
}
```

Плюсы/минусы асинхронности

Плюсы:

- Всё очень эффективно

Минусы:

- Нечитаемо...

Корутины спешат на помощь

Асинхронный сервер с корутинами

```
coro_future coro_accept_stackles() {
  for (;;) {
    auto new_socket = co_await accept(listener);

    auto task = Async([socket = std::move(new_socket)]() -> coro_future {
      auto data = co_await socket.receive();
      process(data);
      co_await socket.send(data);
      co_return;
    });

    task.Detach();
  }
}
```

Асинхронный сервер с корутинами

```
coro_future coro_accept_stackles() {
  for (;;) {
    auto new_socket = co_await accept(listener);

    auto task = Async([socket = std::move(new_socket)]() -> coro_future {
      auto data = co_await socket.receive();
      process(data);
      co_await socket.send(data);
      co_return;
    });

    task.Detach();
  }
}
```

Асинхронный сервер с корутинами

```
coro_future coro_accept_stackles() {
  for (;;) {
    auto new_socket = co_await accept(listener);

    auto task = Async([socket = std::move(new_socket)]() -> coro_future {
      auto data = co_await socket.receive();
      process(data);
      co_await socket.send(data);
      co_return;
    });

    task.Detach();
  }
}
```

Асинхронный сервер с корутинами

```
coro_future coro_accept_stackles() {
  for (;;) {
    auto new_socket = co_await accept(listener);

    auto task = Async([socket = std::move(new_socket)]() -> coro_future {
      auto data = co_await socket.receive();
      process(data);
      co_await socket.send(data);
      co_return;
    });

    task.Detach();
  }
}
```

Асинхронный сервер с корутинами

```
coro_future coro_accept_stackles() {
  for (;;) {
    auto new_socket = co_await accept(listener);

    auto task = Async([socket = std::move(new_socket)]() -> coro_future {
      auto data = co_await socket.receive();
      process(data);
      co_await socket.send(data);
      co_return;
    });

    task.Detach();
  }
}
```

Асинхронный сервер с корутинами

```
coro_future coro_accept_stackles() {
  for (;;) {
    auto new_socket = co_await accept(listener);

    auto task = Async([socket = std::move(new_socket)]() -> coro_future {
      auto data = co_await socket.receive();
      process(data);
      co_await socket.send(data);
      co_return;
    });

    task.Detach();
  }
}
```

Асинхронный сервер с корутинами

```
coro_future coro_accept_stackles() {
  for (;;) {
    auto new_socket = co_await accept(listener);

    auto task = Async([socket = std::move(new_socket)]() -> coro_future {
      auto data = co_await socket.receive();
      process(data);
      co_await socket.send(data);
      co_return;
    });

    task.Detach();
  }
}
```

Асинхронный сервер с корутинами

```
coro_future coro_accept_stackles() {
  for (;;) {
    auto new_socket = co_await accept(listener);

    auto task = Async([socket = std::move(new_socket)]() -> coro_future {
      auto data = co_await socket.receive();
      process(data);
      co_await socket.send(data);
      co_return;
    });

    task.Detach();
  }
}
```

Асинхронный сервер с корутинами

```
coro_future coro_accept_stackles() {
  for (;;) {
    auto new_socket = co_await accept(listener);

    auto task = Async([socket = std::move(new_socket)]() -> coro_future {
      auto data = co_await socket.receive();
      process(data);
      co_await socket.send(data);
      co_return;
    });

    task.Detach();
  }
}
```

Асинхронный сервер с корутинами

```
coro_future coro_accept_stackles() {
  for (;;) {
    auto new_socket = co_await accept(listener);

    auto task = Async([socket = std::move(new_socket)]() -> coro_future {
      auto data = co_await socket.receive();
      process(data);
      co_await socket.send(data);
      co_return;
    });

    task.Detach();
  }
}
```

Асинхронный сервер с корутинами

```
coro_future coro_accept_stackles() {
  for (;;) {
    auto new_socket = co_await accept(listener);

    auto task = Async([socket = std::move(new_socket)]() -> coro_future {
      auto data = co_await socket.receive();
      process(data);
      co_await socket.send(data);
      co_return;
    });

    task.Detach();
  }
}
```

Асинхронный сервер с корутинами vs синхронный

```
coro_future coro_accept_stackles() {
  for (;;) {
    auto new_socket = co_await accept(listener);

    auto task = Async(/*...*/ {
      auto data = co_await socket.receive();
      process(data);
      co_await socket.send(data);
      co_return;
    });

    task.Detach();
  }
}
```

```
void naive_accept() {
  for (;;) {
    auto new_socket = accept(listener);

    std::thread thrd(/*...*/ {
      auto data = socket.receive();
      process(data);
      socket.send(data);

    });

    thrd.detach();
  }
}
```

Асинхронный сервер с корутинами vs синхронный

```
coro_future coro_accept_stackles() {
  for (;;) {
    auto new_socket = co_await accept(listener);

    auto task = Async(/*...*/ {
      auto data = co_await socket.receive();
      process(data);
      co_await socket.send(data);
      co_return;
    });

    task.Detach();
  }
}
```

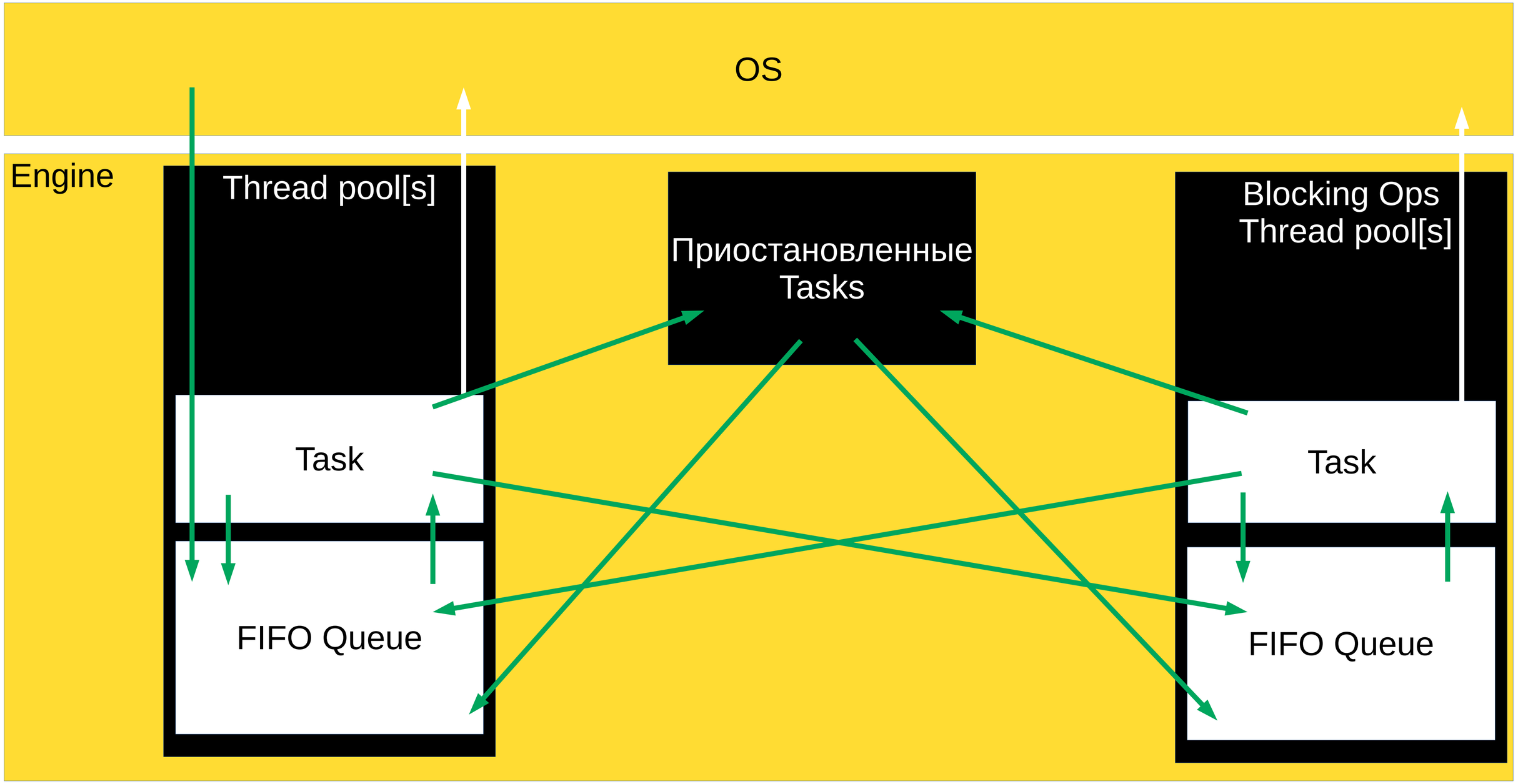
```
void naive_accept() {
  for (;;) {
    auto new_socket = accept(listener);

    std::thread thrd(/*...*/ {
      auto data = socket.receive();
      process(data);
      socket.send(data);

    });

    thrd.detach();
  }
}
```

Асинхронный сервер с корутинами vs синхронный

```
void coro_accept_stackfull() {
  for (;;) {
    auto new_socket = accept(listener);

    auto task = Async(/*...*/ {
      auto data = socket.receive();
      process(data);
      socket.send(data);
    });

    task.Detach();
  }
}
```

```
void naive_accept() {
  for (;;) {
    auto new_socket = accept(listener);

    std::thread thrd(/*...*/ {
      auto data = socket.receive();
      process(data);
      socket.send(data);
    });

    thrd.detach();
  }
}
```

Устройство корутинового движка


OS
Engine
Thread pool[s]
Task
FIFO Queue
Приостановленные
Tasks
Blocking Ops
Thread pool[s]
Task
FIFO Queue

Корутины ≈ колбеки

```
coro_future coro_accept_stackles() {
  for (;;) {
    auto new_socket = co_await accept(listener);

    auto task = Async([socket = std::move(new_socket)]() -> coro_future {
      auto data = co_await socket.receive();
      process(data);
      co_await socket.send(data);
      co_return;
    });

    task.Detach();
  }
}
```

Корутины ≈ колбеки

```
coro_future coro_accept_stackles() {
  for (;;) {
    auto new_socket = co_await accept(listener);

    auto task = Async([socket = std::move(new_socket)]() -> coro_future {
      auto data = co_await socket.receive();
      process(data);
      co_await socket.send(data);
      co_return;
    });

    task.Detach();
  }
}
```

Корутины ≈ колбеки

```
coro_future coro_accept_stackles() {
  for (;;) {
    auto new_socket = co_await accept(listener);

    auto task = Async([socket = std::move(new_socket)]() -> coro_future {
      auto data = co_await socket.receive();
      process(data);
      co_await socket.send(data);
      co_return;
    });

    task.Detach();
  }
}
```

Корутины ≈ колбеки

```
coro_future coro_accept_stackles() {
  for (;;) {
    auto new_socket = co_await accept(listener);

    auto task = Async([socket = std::move(new_socket)]() -> coro_future {
      auto data = co_await socket.receive();
      process(data);
      co_await socket.send(data);
      co_return;
    });

    task.Detach();
  }
}
```

Корутины ≈ колбеки

```
coro_future coro_accept_stackles() {
  for (;;) {
    auto new_socket = co_await accept(listener);

    auto task = Async([socket = std::move(new_socket)]() -> coro_future {
      auto data = co_await socket.receive();
      process(data);
      co_await socket.send(data);
      co_return;
    });

    task.Detach();
  }
}
```

Корутины ≈ колбеки

```
coro_future coro_accept_stackles() {
  for (;;) {
    auto new_socket = co_await accept(listener);

    auto task = Async([socket = std::move(new_socket)]() -> coro_future {
      auto data = co_await socket.receive();
      process(data);
      co_await socket.send(data);
      co_return;
    });

    task.Detach();
  }
}
```

Корутины ≈ колбеки

```
coro_future coro_accept_stackles() {
  for (;;) {
    auto new_socket = co_await accept(listener);

    auto task = Async([socket = std::move(new_socket)]() -> coro_future {
      auto data = co_await socket.receive();
      process(data);
      co_await socket.send(data);
      co_return;
    });

    task.Detach();
  }
}
```

Плюсы/минусы асинхронности с корутинами

Плюсы/минусы асинхронности с корутинами

Плюсы:

Плюсы/минусы асинхронности с корутинами

Плюсы:

- Всё очень эффективно

Плюсы/минусы асинхронности с корутинами

Плюсы:

- Всё очень эффективно
- Просто и читаемо

Плюсы/минусы асинхронности с корутинами

Плюсы:

- Всё очень эффективно
- Просто и читаемо

Минусы:

- Под капотом жесть!..

Плюсы/минусы асинхронности с корутинами

Плюсы:

- Всё очень эффективно
- Просто и читаемо

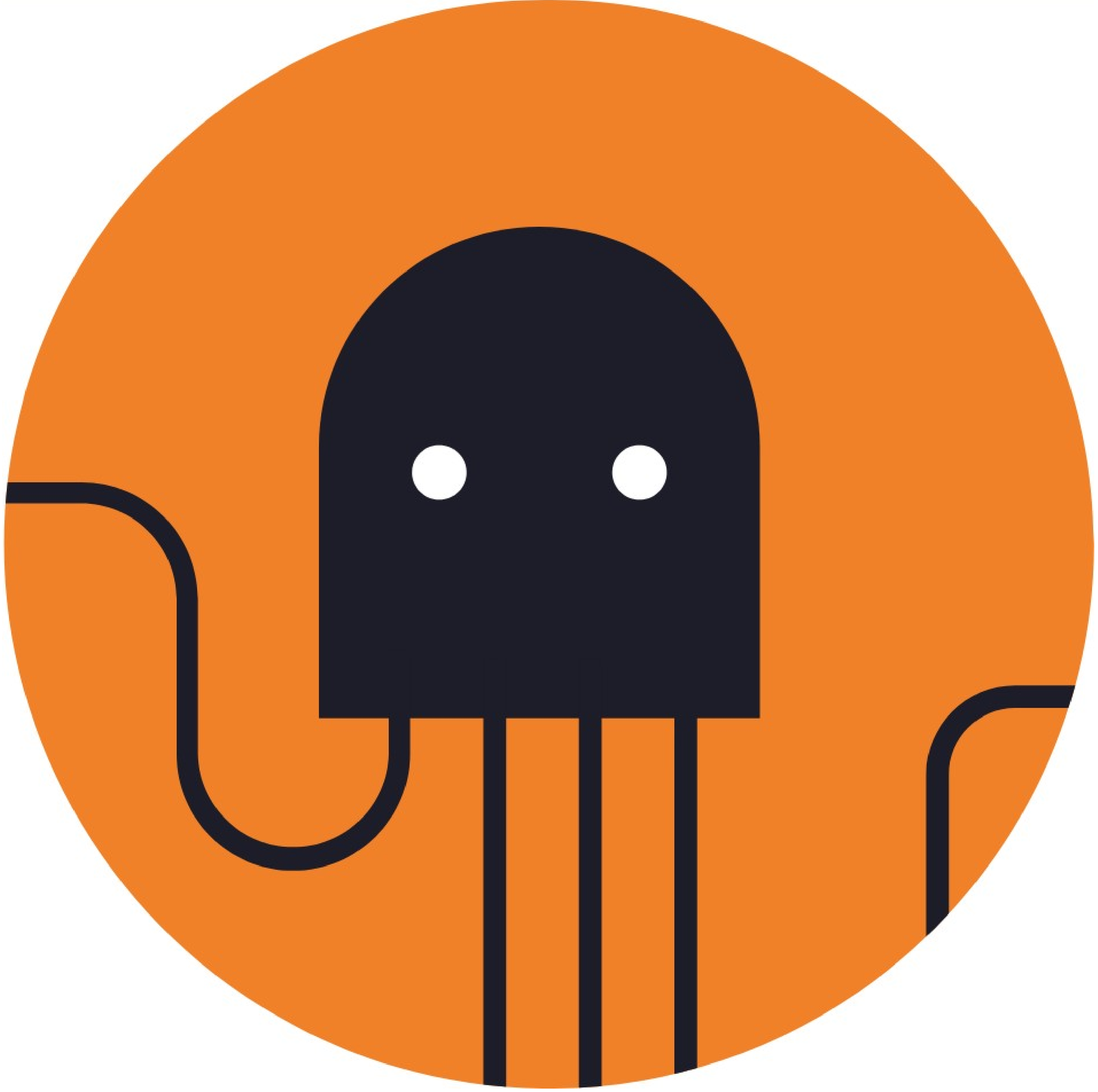
Минусы:

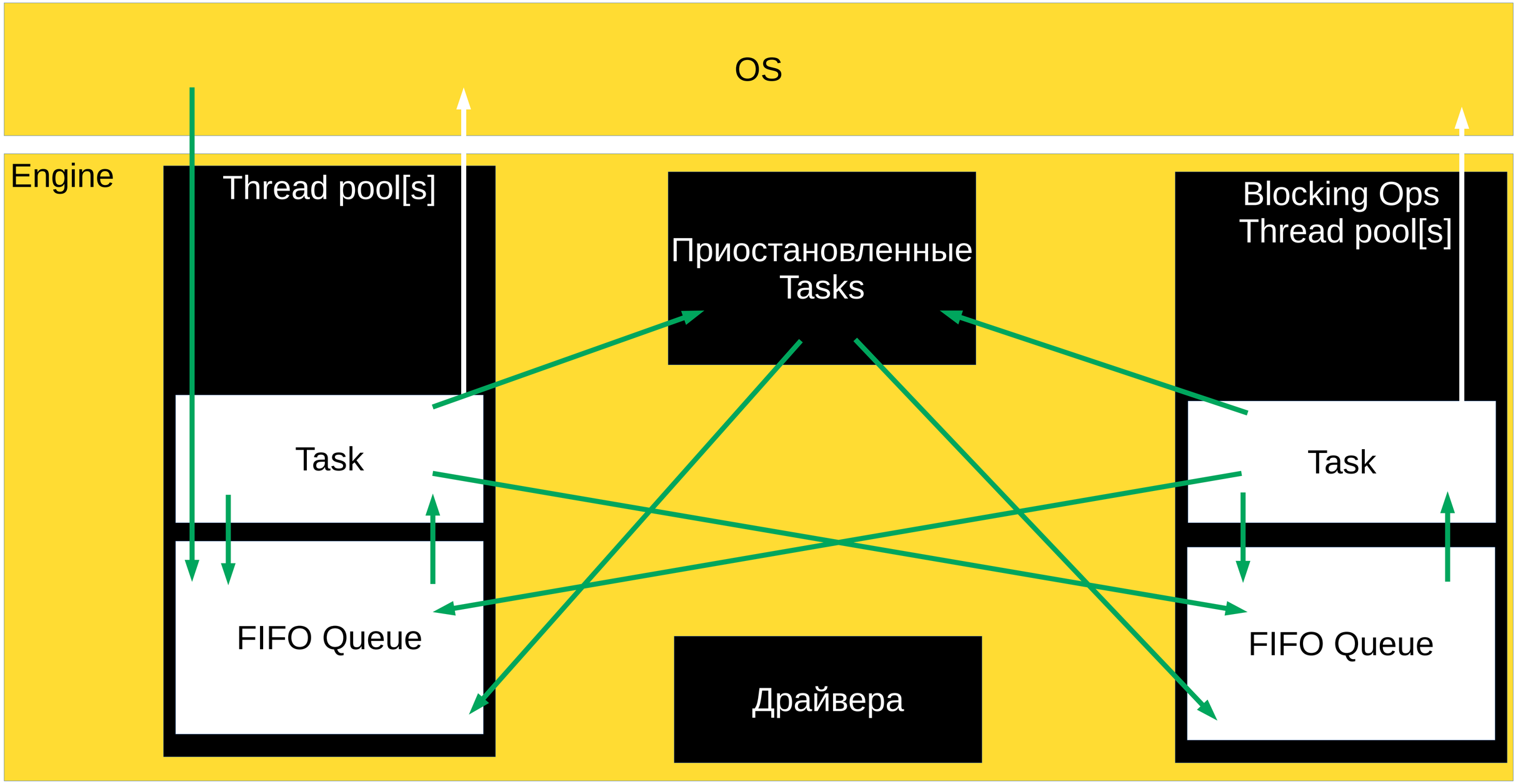
- Под капотом жесть!.. Впрочем, такая же жесть при любой асинхронности

Плюсы/минусы асинхронности с корутинами

Плюсы:

- Всё очень эффективно
- Просто и читаемо
- Под капотом жесть!..




OS
Engine
Thread pool[s]
Task
FIFO Queue
Приостановленные
Tasks
Драйвера
Blocking Ops
Thread pool[s]
Task
FIFO Queue

C++ хардкорище!

Мьютекс в корутиновом движке

```
struct Mutex {
  void lock();
  void unlock();

 private:
  std::atomic<Coroutine*> owner_{nullptr};

};
```

Мьютекс в корутиновом движке

```
struct Mutex {
  void lock();
  void unlock();

 private:
  std::atomic<Coroutine*> owner_{nullptr};

};
```

Мьютекс в корутиновом движке

```
struct Mutex {
  void lock();
  void unlock();

 private:
  std::atomic<Coroutine*> owner_{nullptr};

};
```

Мьютекс в корутиновом движке

```
void Mutex::lock() {
  Coroutine* current = GetCurrentCoro();
  Coroutine* expected = nullptr;

  if (owner_.compare_exchange_strong(expected, current)) return;
  Coroutine* expected = nullptr;

  impl::MutexWaitStrategy wait_manager(lock_waiters_, current);
  while (!owner_.compare_exchange_strong(expected, current)) {
    assert(expected != current && "Mutex is locked twice from the same task");
    current->Sleep(wait_manager);
    expected = nullptr;
  }
}
```

Мьютекс в корутиновом движке

```
void Mutex::lock() {
  Coroutine* current = GetCurrentCoro();
  Coroutine* expected = nullptr;

  if (owner_.compare_exchange_strong(expected, current)) return;
  Coroutine* expected = nullptr;

  impl::MutexWaitStrategy wait_manager(lock_waiters_, current);
  while (!owner_.compare_exchange_strong(expected, current)) {
    assert(expected != current && "Mutex is locked twice from the same task");
    current->Sleep(wait_manager);
    expected = nullptr;
  }
}
```

Мьютекс в корутиновом движке

```
void Mutex::lock() {
  Coroutine* current = GetCurrentCoro();
  Coroutine* expected = nullptr;

  if (owner_.compare_exchange_strong(expected, current)) return;
  Coroutine* expected = nullptr;

  impl::MutexWaitStrategy wait_manager(lock_waiters_, current);
  while (!owner_.compare_exchange_strong(expected, current)) {
    assert(expected != current && "Mutex is locked twice from the same task");
    current->Sleep(wait_manager);
    expected = nullptr;
  }
}
```

Мьютекс в корутиновом движке

```
void Mutex::lock() {
  Coroutine* current = GetCurrentCoro();
  Coroutine* expected = nullptr;

  if (owner_.compare_exchange_strong(expected, current)) return;
  Coroutine* expected = nullptr;

  impl::MutexWaitStrategy wait_manager(lock_waiters_, current);
  while (!owner_.compare_exchange_strong(expected, current)) {
    assert(expected != current && "Mutex is locked twice from the same task");
    current->Sleep(wait_manager);
    expected = nullptr;
  }
}
```

Мьютекс в корутиновом движке

```
void Mutex::lock() {
  Coroutine* current = GetCurrentCoro();
  Coroutine* expected = nullptr;

  if (owner_.compare_exchange_strong(expected, current)) return;
  Coroutine* expected = nullptr;

  impl::MutexWaitStrategy wait_manager(lock_waiters_, current);
  while (!owner_.compare_exchange_strong(expected, current)) {
    assert(expected != current && "Mutex is locked twice from the same task");
    current->Sleep(wait_manager);
    expected = nullptr;
  }
}
```

Мьютекс в корутиновом движке

```
void Mutex::lock() {
  Coroutine* current = GetCurrentCoro();
  Coroutine* expected = nullptr;

  if (owner_.compare_exchange_strong(expected, current)) return;
  Coroutine* expected = nullptr;

  impl::MutexWaitStrategy wait_manager(lock_waiters_, current);
  while (!owner_.compare_exchange_strong(expected, current)) {
    assert(expected != current && "Mutex is locked twice from the same task");
    current->Sleep(wait_manager);
    expected = nullptr;
  }
}
```

Мьютекс в корутиновом движке

```
void Mutex::lock() {
  Coroutine* current = GetCurrentCoro();
  Coroutine* expected = nullptr;

  if (owner_.compare_exchange_strong(expected, current)) return;
  Coroutine* expected = nullptr;

  impl::MutexWaitStrategy wait_manager(lock_waiters_, current);
  while (!owner_.compare_exchange_strong(expected, current)) {
    assert(expected != current && "Mutex is locked twice from the same task");
    current->Sleep(wait_manager);
    expected = nullptr;
  }
}
```

Мьютекс в корутиновом движке

```
void Mutex::lock() {
  Coroutine* current = GetCurrentCoro();
  Coroutine* expected = nullptr;

  if (owner_.compare_exchange_strong(expected, current)) return;
  Coroutine* expected = nullptr;

  impl::MutexWaitStrategy wait_manager(lock_waiters_, current);
  while (!owner_.compare_exchange_strong(expected, current)) {
    assert(expected != current && "Mutex is locked twice from the same task");
    current->Sleep(wait_manager);
    expected = nullptr;
  }
}
```

Мьютекс в корутиновом движке

```
void Mutex::lock() {
  Coroutine* current = GetCurrentCoro();
  Coroutine* expected = nullptr;

  if (owner_.compare_exchange_strong(expected, current)) return;
  Coroutine* expected = nullptr;

  impl::MutexWaitStrategy wait_manager(lock_waiters_, current);
  while (!owner_.compare_exchange_strong(expected, current)) {
    assert(expected != current && "Mutex is locked twice from the same task");
    current->Sleep(wait_manager);
    expected = nullptr;
  }
}
```

Мьютекс в корутиновом движке

```
void Mutex::lock() {
  Coroutine* current = GetCurrentCoro();
  Coroutine* expected = nullptr;

  if (owner_.compare_exchange_strong(expected, current)) return;
  Coroutine* expected = nullptr;

  impl::MutexWaitStrategy wait_manager(lock_waiters_, current);
  while (!owner_.compare_exchange_strong(expected, current)) {
    assert(expected != current && "Mutex is locked twice from the same task");
    current->Sleep(wait_manager);
    expected = nullptr;
  }
}
```

Мьютекс в корутиновом движке

```
void Mutex::lock() {
  Coroutine* current = GetCurrentCoro();
  Coroutine* expected = nullptr;

  if (owner_.compare_exchange_strong(expected, current)) return;
  Coroutine* expected = nullptr;

  impl::MutexWaitStrategy wait_manager(lock_waiters_, current);
  while (!owner_.compare_exchange_strong(expected, current)) {
    assert(expected != current && "Mutex is locked twice from the same task");
    current->Sleep(wait_manager);
    expected = nullptr;
  }
}
```

Мьютекс в корутиновом движке

```
void Mutex::lock() {
  Coroutine* current = GetCurrentCoro();
  Coroutine* expected = nullptr;

  if (owner_.compare_exchange_strong(expected, current)) return;
  Coroutine* expected = nullptr;

  impl::MutexWaitStrategy wait_manager(lock_waiters_, current);
  while (!owner_.compare_exchange_strong(expected, current)) {
    assert(expected != current && "Mutex is locked twice from the same task");
    current->Sleep(wait_manager);
    expected = nullptr;
  }
}
```

Мьютекс в корутиновом движке

```
void Mutex::lock() {
  Coroutine* current = GetCurrentCoro();
  Coroutine* expected = nullptr;

  if (owner_.compare_exchange_strong(expected, current)) return;
  Coroutine* expected = nullptr;

  impl::MutexWaitStrategy wait_manager(lock_waiters_, current);
  while (!owner_.compare_exchange_strong(expected, current)) {
    assert(expected != current && "Mutex is locked twice from the same task");
    current->Sleep(wait_manager);
    expected = nullptr;
  }
}
```

Мьютекс в корутиновом движке

```
void Mutex::lock() {
  Coroutine* current = GetCurrentCoro();
  Coroutine* expected = nullptr;

  if (owner_.compare_exchange_strong(expected, current)) return;
  Coroutine* expected = nullptr;

  impl::MutexWaitStrategy wait_manager(lock_waiters_, current);
  while (!owner_.compare_exchange_strong(expected, current)) {
    assert(expected != current && "Mutex is locked twice from the same task");
    current->Sleep(wait_manager);
    expected = nullptr;
  }
}
```

Мьютекс в корутиновом движке

```
void Mutex::lock() {
  Coroutine* current = GetCurrentCoro();
  Coroutine* expected = nullptr;

  if (owner_.compare_exchange_strong(expected, current)) return;
  Coroutine* expected = nullptr;

  impl::MutexWaitStrategy wait_manager(lock_waiters_, current);
  while (!owner_.compare_exchange_strong(expected, current)) {
    assert(expected != current && "Mutex is locked twice from the same task");
    current->Sleep(wait_manager);
    expected = nullptr;
  }
}
```

Мьютекс в корутиновом движке

```
class Coroutine {
 public:
  // ...
  void Sleep(WaitStrategy& strategy);
  // ...
};
```

Мьютекс в корутиновом движке

```
class Coroutine {
 public:
  // ...
  void Sleep(WaitStrategy& strategy);
  // ...
};
```

Мьютекс в корутиновом движке

```
class Coroutine {
 public:
  // ...
  void Sleep(WaitStrategy& strategy);
  // ...
};
```

Мьютекс в корутиновом движке

```
class Coroutine {
 public:
  // ...
  void Sleep(WaitStrategy& strategy);
  // ...
};

class WaitStrategy {
 public:
  virtual void AfterAsleep() = 0;
  virtual void BeforeAwake() = 0;
 protected:
  ~WaitStrategy() = default;
};
```

Мьютекс в корутиновом движке

```
class Coroutine {
 public:
  // ...
  void Sleep(WaitStrategy& strategy);
  // ...
};

class WaitStrategy {
 public:
  virtual void AfterAsleep() = 0;
  virtual void BeforeAwake() = 0;
 protected:
  ~WaitStrategy() = default;
};
```

Мьютекс в корутиновом движке

```
class Coroutine {
 public:
  // ...
  void Sleep(WaitStrategy& strategy);
  // ...
};

class WaitStrategy {
 public:
  virtual void AfterAsleep() = 0;
  virtual void BeforeAwake() = 0;
 protected:
  ~WaitStrategy() = default;
};
```

Мьютекс в корутиновом движке

```
class Coroutine {
 public:
  // ...
  void Sleep(WaitStrategy& strategy);
  // ...
};

class WaitStrategy {
 public:
  virtual void AfterAsleep() = 0;
  virtual void BeforeAwake() = 0;
 protected:
  ~WaitStrategy() = default;
};
```

Мьютекс в корутиновом движке

```
class Coroutine {
 public:
  // ...
  void Sleep(WaitStrategy& strategy);
  // ...
};

class WaitStrategy {
 public:
  virtual void AfterAsleep() = 0;
  virtual void BeforeAwake() = 0;
 protected:
  ~WaitStrategy() = default;
};
```

Мьютекс в корутиновом движке

```
struct Mutex {
  void lock();
  void unlock();

 private:
  std::atomic<Coroutine*> owner_{nullptr};

};
```

Мьютекс в корутиновом движке

```
struct Mutex {
  void lock();
  void unlock();

 private:
  std::atomic<Coroutine*> owner_{nullptr};
  WaitList lock_waiters_;
};
```

Мьютекс в корутиновом движке

```
void Mutex::lock() {
  Coroutine* current = GetCurrentCoro();
  Coroutine* expected = nullptr;

  if (owner_.compare_exchange_strong(expected, current)) return;
  Coroutine* expected = nullptr;

  impl::MutexWaitStrategy wait_manager(lock_waiters_, current);
  while (!owner_.compare_exchange_strong(expected, current)) {
    assert(expected != current && "Mutex is locked twice from the same task");
    current->Sleep(wait_manager);
    expected = nullptr;
  }
}
```

Мьютекс в корутиновом движке

```
void Mutex::lock() {
  Coroutine* current = GetCurrentCoro();
  Coroutine* expected = nullptr;

  if (owner_.compare_exchange_strong(expected, current)) return;
  Coroutine* expected = nullptr;

  impl::MutexWaitStrategy wait_manager(lock_waiters_, current);
  while (!owner_.compare_exchange_strong(expected, current)) {
    assert(expected != current && "Mutex is locked twice from the same task");
    current->Sleep(wait_manager);
    expected = nullptr;
  }
}
```

Мьютекс в корутиновом движке

```
void Mutex::lock() {
  Coroutine* current = GetCurrentCoro();
  Coroutine* expected = nullptr;

  if (owner_.compare_exchange_strong(expected, current)) return;
  Coroutine* expected = nullptr;

  impl::ListNode<Coroutine*> node{current};
  while (!owner_.compare_exchange_strong(expected, current)) {
    assert(expected != current && "Mutex is locked twice from the same task");
    current->Sleep(wait_manager);
    expected = nullptr;
  }
}
```

Мьютекс в корутиновом движке

```
void Mutex::lock() {
  Coroutine* current = GetCurrentCoro();
  Coroutine* expected = nullptr;

  if (owner_.compare_exchange_strong(expected, current)) return;
  Coroutine* expected = nullptr;

  impl::MutexWaitStrategy wait_manager(lock_waiters_, current);
  while (!owner_.compare_exchange_strong(expected, current)) {
    assert(expected != current && "Mutex is locked twice from the same task");
    current->Sleep(wait_manager);
    expected = nullptr;
  }
}
```

Мьютекс в корутиновом движке

```
class MutexWaitStrategy final : public WaitStrategy {
 public:
  MutexWaitStrategy(WaitList& waiters, Coroutine* current)
      : WaitStrategy(), waiters_(waiters), current_(current), lock_(waiters) {}

  void AfterAsleep() override {
    waiters_.Append(lock_, current_);
    lock_.unlock();
  }

  void BeforeAwake() override {
    lock_.lock();
    waiters_.Remove(lock_, current_);
  }

 private:
  WaitList& waiters_;
  Coroutine* const current_;
  WaitList::Lock lock_;
};
```

Мьютекс в корутиновом движке

```
class MutexWaitStrategy final : public WaitStrategy {
 public:
  MutexWaitStrategy(WaitList& waiters, Coroutine* current)
      : WaitStrategy(), waiters_(waiters), current_(current), lock_(waiters) {}

  void AfterAsleep() override {
    waiters_.Append(lock_, current_);
    lock_.unlock();
  }

  void BeforeAwake() override {
    lock_.lock();
    waiters_.Remove(lock_, current_);
  }

 private:
  WaitList& waiters_;
  Coroutine* const current_;
  WaitList::Lock lock_;
};
```

Мьютекс в корутиновом движке

```
void Mutex::lock() {
  Coroutine* current = GetCurrentCoro();
  Coroutine* expected = nullptr;

  if (owner_.compare_exchange_strong(expected, current)) return;
  Coroutine* expected = nullptr;

  impl::MutexWaitStrategy wait_manager(lock_waiters_, current);
  while (!owner_.compare_exchange_strong(expected, current)) {
    assert(expected != current && "Mutex is locked twice from the same task");
    current->Sleep(wait_manager);
    expected = nullptr;
  }
}
```

Мьютекс в корутиновом движке

```
void Mutex::lock() {
  Coroutine* current = GetCurrentCoro();
  Coroutine* expected = nullptr;

  if (owner_.compare_exchange_strong(expected, current)) return;
  Coroutine* expected = nullptr;

  impl::MutexWaitStrategy wait_manager(lock_waiters_, current);
  while (!owner_.compare_exchange_strong(expected, current)) {
    assert(expected != current && "Mutex is locked twice from the same task");
    current->Sleep(wait_manager);
    expected = nullptr;
  }
}
```

Мьютекс в корутиновом движке

```
void Mutex::lock() {
  Coroutine* current = GetCurrentCoro();
  Coroutine* expected = nullptr;

  if (owner_.compare_exchange_strong(expected, current)) return;
  Coroutine* expected = nullptr;

  impl::MutexWaitStrategy wait_manager(lock_waiters_, current);
  while (!owner_.compare_exchange_strong(expected, current)) {
    assert(expected != current && "Mutex is locked twice from the same task");
    current->Sleep(wait_manager);
    expected = nullptr;
  }
}
```

Мьютекс в корутиновом движке

```
void Mutex::lock() {
  Coroutine* current = GetCurrentCoro();
  Coroutine* expected = nullptr;

  if (owner_.compare_exchange_strong(expected, current)) return;
  Coroutine* expected = nullptr;

  impl::MutexWaitStrategy wait_manager(lock_waiters_, current);
  while (!owner_.compare_exchange_strong(expected, current)) {
    assert(expected != current && "Mutex is locked twice from the same task");
    current->Sleep(wait_manager);
    expected = nullptr;
  }
}
```

Мьютекс в корутиновом движке

```
void Mutex::lock() {
  Coroutine* current = GetCurrentCoro();
  Coroutine* expected = nullptr;

  if (owner_.compare_exchange_strong(expected, current)) return;
  Coroutine* expected = nullptr;

  impl::MutexWaitStrategy wait_manager(lock_waiters_, current);
  while (!owner_.compare_exchange_strong(expected, current)) {
    assert(expected != current && "Mutex is locked twice from the same task");
    current->Sleep(wait_manager);
    expected = nullptr;
  }
}
```

Мьютекс в корутиновом движке

```
class MutexWaitStrategy final : public WaitStrategy {
 public:
  MutexWaitStrategy(WaitList& waiters, Coroutine* current)
      : WaitStrategy(), waiters_(waiters), current_(current), lock_(waiters) {}

  void AfterAsleep() override {
    waiters_.Append(lock_, current_);
    lock_.unlock();
  }

  void BeforeAwake() override {
    lock_.lock();
    waiters_.Remove(lock_, current_);
  }

 private:
  WaitList& waiters_;
  Coroutine* const current_;
  WaitList::Lock lock_;
};
```

Мьютекс в корутиновом движке

```
void Mutex::unlock() {
  [[maybe_unused]] const auto old_owner = owner_.exchange(nullptr);
  assert(old_owner == GetCurrentCoro());
  WaitList::Lock lock(lock_waiters_);
  lock_waiters_.WakeupOne(lock);
}
```

Мьютекс в корутиновом движке

```
void Mutex::unlock() {
  [[maybe_unused]] const auto old_owner = owner_.exchange(nullptr);
  assert(old_owner == GetCurrentCoro());
  WaitList::Lock lock(lock_waiters_);
  lock_waiters_.WakeupOne(lock);
}
```

Мьютекс в корутиновом движке

```
void Mutex::unlock() {
  [[maybe_unused]] const auto old_owner = owner_.exchange(nullptr);
  assert(old_owner == GetCurrentCoro());
  WaitList::Lock lock(lock_waiters_);
  lock_waiters_.WakeupOne(lock);
}
```

Мьютекс в корутиновом движке

```
void Mutex::unlock() {
  [[maybe_unused]] const auto old_owner = owner_.exchange(nullptr);
  assert(old_owner == GetCurrentCoro());
  WaitList::Lock lock(lock_waiters_);
  lock_waiters_.WakeupOne(lock);
}
```

Мьютекс в корутиновом движке

```
void Mutex::unlock() {
  [[maybe_unused]] const auto old_owner = owner_.exchange(nullptr);
  assert(old_owner == GetCurrentCoro());
  WaitList::Lock lock(lock_waiters_);
  lock_waiters_.WakeupOne(lock);
}
```

Мьютекс в корутиновом движке

```
void Mutex::unlock() {
  [[maybe_unused]] const auto old_owner = owner_.exchange(nullptr);
  assert(old_owner == GetCurrentCoro());
  WaitList::Lock lock(lock_waiters_);
  lock_waiters_.WakeupOne(lock);
}
```

Мьютекс в корутиновом движке

```
void Mutex::unlock() {
  [[maybe_unused]] const auto old_owner = owner_.exchange(nullptr);
  assert(old_owner == GetCurrentCoro());
  WaitList::Lock lock(lock_waiters_);
  lock_waiters_.WakeupOne(lock);
}
```

Мьютекс в корутиновом движке

```
class MutexWaitStrategy final : public WaitStrategy {
 public:
  MutexWaitStrategy(WaitList& waiters, Coroutine* current)
      : WaitStrategy(), waiters_(waiters), current_(current), lock_(waiters) {}

  void AfterAsleep() override {
    waiters_.Append(lock_, current_);
    lock_.unlock();
  }

  void BeforeAwake() override {
    lock_.lock();
    waiters_.Remove(lock_, current_);
  }

 private:
  WaitList& waiters_;
  Coroutine* const current_;
  WaitList::Lock lock_;
};
```

Мьютекс в корутиновом движке

```
void Mutex::lock() {
  Coroutine* current = GetCurrentCoro();
  Coroutine* expected = nullptr;

  if (owner_.compare_exchange_strong(expected, current)) return;
  Coroutine* expected = nullptr;

  impl::MutexWaitStrategy wait_manager(lock_waiters_, current);
  while (!owner_.compare_exchange_strong(expected, current)) {
    assert(expected != current && "Mutex is locked twice from the same task");
    current->Sleep(wait_manager);
    expected = nullptr;
  }
}
```

Мьютекс в корутиновом движке

```
void Mutex::lock() {
  Coroutine* current = GetCurrentCoro();
  Coroutine* expected = nullptr;

  if (owner_.compare_exchange_strong(expected, current)) return;
  Coroutine* expected = nullptr;

  impl::MutexWaitStrategy wait_manager(lock_waiters_, current);
  while (!owner_.compare_exchange_strong(expected, current)) {
    assert(expected != current && "Mutex is locked twice from the same task");
    current->Sleep(wait_manager);
    expected = nullptr;
  }
}
```

Мьютекс в корутиновом движке

```
void Mutex::lock() {
  Coroutine* current = GetCurrentCoro();
  Coroutine* expected = nullptr;

  if (owner_.compare_exchange_strong(expected, current)) return;
  Coroutine* expected = nullptr;

  impl::MutexWaitStrategy wait_manager(lock_waiters_, current);
  while (!owner_.compare_exchange_strong(expected, current)) {
    assert(expected != current && "Mutex is locked twice from the same task");
    current->Sleep(wait_manager);
    expected = nullptr;
  }
}
```

Мьютекс

Мьютекс

- Не блокирует поток

Мьютекс

- Не блокирует поток
- Не переключает контекст

Мьютекс

- Не блокирует поток
- Не переключает контекст
- Не аллоцирует

А если с таймаутами?

Мьютекс в корутиновом движке

```
struct Mutex {
  void lock();
  bool try_lock_untill(Deadline deadline);
  void unlock();

 private:
  std::atomic<Coroutine*> owner_{nullptr};
  WaitList lock_waiters_;
};
```

Мьютекс в корутиновом движке

```
struct Mutex {
  void lock();
  bool try_lock_untill(Deadline deadline);
  void unlock();

 private:
  std::atomic<Coroutine*> owner_{nullptr};
  WaitList lock_waiters_;
};
```

Мьютекс в корутиновом движке

```
struct Mutex {
  void lock();
  bool try_lock_untill(Deadline deadline);
  void unlock();

 private:
  std::atomic<Coroutine*> owner_{nullptr};
  WaitList lock_waiters_;
};
```

А таймаут?

```
class Coroutine {
 public:
  // ...
  void Sleep(WaitStrategy& strategy);

  Epoch GetEpoch();
  void Wakeup(Epoch epoch);
  // ...
};

class WaitStrategy {
 public:
  virtual void AfterAsleep() = 0;
  virtual void BeforeAwake() = 0;
 protected:
  ~WaitStrategy() = default;
};
```

А таймаут?

```
class Coroutine {
 public:
  // ...
  void Sleep(WaitStrategy& strategy);

  Epoch GetEpoch();
  void Wakeup(Epoch epoch);
  // ...
};

class WaitStrategy {
 public:
  virtual void AfterAsleep() = 0;
  virtual void BeforeAwake() = 0;
 protected:
  ~WaitStrategy() = default;
};
```

А таймаут?

```
class Coroutine {
 public:
  // ...
  void Sleep(WaitStrategy& strategy);

  Epoch GetEpoch();
  void Wakeup(Epoch epoch);
  // ...
};

class WaitStrategy {
 public:
  virtual void AfterAsleep() = 0;
  virtual void BeforeAwake() = 0;
 protected:
  ~WaitStrategy() = default;
};
```

А таймаут?

```
class Coroutine {
 public:
  // ...
  void Sleep(WaitStrategy& strategy);

  Epoch GetEpoch();
  void Wakeup(Epoch epoch);
  // ...
};

class WaitStrategy {
 public:
  virtual void AfterAsleep() = 0;
  virtual void BeforeAwake() = 0;
 protected:
  ~WaitStrategy() = default;
};
```

А таймаут?

```
class Coroutine {
 public:
  // ...
  void Sleep(WaitStrategy& strategy);

  Epoch GetEpoch();
  void Wakeup(Epoch epoch);
  // ...
};

class WaitStrategy {
 public:
  virtual void AfterAsleep() = 0;
  virtual void BeforeAwake() = 0;
 protected:
  ~WaitStrategy() = default;
};
```

А таймаут?

```
class Coroutine {
 public:
  // ...
  void Sleep(WaitStrategy& strategy);

  Epoch GetEpoch();
  void Wakeup(Epoch epoch);
  // ...
};

class WaitStrategy {
 public:
  virtual void AfterAsleep() = 0;
  virtual void BeforeAwake() = 0;
 protected:
  ~WaitStrategy() = default;
};
```

А таймаут?

```
class Coroutine {
 public:
  // ...
  void Sleep(WaitStrategy& strategy);

  Epoch GetEpoch();
  void Wakeup(Epoch epoch);
  // ...
};

class WaitStrategy {
 public:
  virtual void AfterAsleep() = 0;
  virtual void BeforeAwake() = 0;
 protected:
  ~WaitStrategy() = default;
};
```

А таймаут?

```
class Coroutine {
 public:
  // ...
  void Sleep(WaitStrategy& strategy);

  Epoch GetEpoch();
  void Wakeup(Epoch epoch);
  // ...
};

class WaitStrategy {
 public:
  virtual void AfterAsleep() = 0;
  virtual void BeforeAwake() = 0;

  const Deadline deadline;
 protected:
  ~WaitStrategy() = default;
};
```

А таймаут?

```
class Coroutine {
 public:
  // ...
  void Sleep(WaitStrategy& strategy);

  Epoch GetEpoch();
  void Wakeup(Epoch epoch);
  // ...
};

class WaitStrategy {
 public:
  virtual void AfterAsleep() = 0;
  virtual void BeforeAwake() = 0;

  const Deadline deadline;
 protected:
  ~WaitStrategy() = default;
};
```

А таймаут?

```
class Coroutine {
 public:
  // ...
  void Sleep(WaitStrategy& strategy);

  Epoch GetEpoch();
  void Wakeup(Epoch epoch);
  // ...
};

class WaitStrategy {
 public:
  virtual void AfterAsleep() = 0;
  virtual void BeforeAwake() = 0;

  const Deadline deadline;
 protected:
  ~WaitStrategy() = default;
};
```

А таймаут?

```
class Coroutine {
 public:
  // ...
  WakeupReason Sleep(WaitStrategy& strategy);

  Epoch GetEpoch();
  void Wakeup(Epoch epoch);
  // ...
};

class WaitStrategy {
 public:
  virtual void AfterAsleep() = 0;
  virtual void BeforeAwake() = 0;

  const Deadline deadline;
 protected:
  ~WaitStrategy() = default;
};
```

А таймаут?

```
class Coroutine {
 public:
  // ...
  WakeupReason Sleep(WaitStrategy& strategy);

  Epoch GetEpoch();
  void Wakeup(Epoch epoch);
  // ...
};

class WaitStrategy {
 public:
  virtual void AfterAsleep() = 0;
  virtual void BeforeAwake() = 0;

  const Deadline deadline;
 protected:
  ~WaitStrategy() = default;
};
```

А таймаут?

```
class Coroutine {
 public:
  // ...
  WakeupReason Sleep(WaitStrategy& strategy);

  Epoch GetEpoch();
  void Wakeup(Epoch epoch);
  // ...
};

class WaitStrategy {
 public:
  virtual void AfterAsleep() = 0;
  virtual void BeforeAwake() = 0;

  const Deadline deadline;
 protected:
  ~WaitStrategy() = default;
};
```

А если с отменами?

А отмены?

```
class Coroutine {
 public:
  // ...
  WakeupReason Sleep(WaitStrategy& strategy);
  bool IsCancelled() const;
  void Cancel();

  Epoch GetEpoch();
  void Wakeup(Epoch epoch);
  // ...
};

class WaitStrategy {
 public:
  virtual void AfterAsleep() = 0;
  virtual void BeforeAwake() = 0;
 protected:
  ~WaitStrategy() = default;
};
```

А отмены?

```
class Coroutine {
 public:
  // ...
  WakeupReason Sleep(WaitStrategy& strategy);
  bool IsCancelled() const;
  void Cancel();

  Epoch GetEpoch();
  void Wakeup(Epoch epoch);
  // ...
};

class WaitStrategy {
 public:
  virtual void AfterAsleep() = 0;
  virtual void BeforeAwake() = 0;
 protected:
  ~WaitStrategy() = default;
};
```

А отмены?

```
class Coroutine {
 public:
  // ...
  WakeupReason Sleep(WaitStrategy& strategy);
  bool IsCancelled() const;
  void Cancel();

  Epoch GetEpoch();
  void Wakeup(Epoch epoch);
  // ...
};

class WaitStrategy {
 public:
  virtual void AfterAsleep() = 0;
  virtual void BeforeAwake() = 0;
 protected:
  ~WaitStrategy() = default;
};
```

Итог

Итого

Итого

- Корутины ≈ callbacks

Итого

- Корутины ≈ callbacks
- Асинхронность позволяет экономить потоки и CPU

Итого

- Корутины ≈ callbacks
- Асинхронность позволяет экономить потоки и CPU
- Под капотом жесть

Итого

- Корутины ≈ callbacks
- Асинхронность позволяет экономить потоки и CPU
- Под капотом жесть, прикольная жесть!

Итого

- Корутины ≈ callbacks
- Асинхронность позволяет экономить потоки и CPU
- Под капотом жесть, прикольная жесть!
- Снаружи — всё просто и понятно

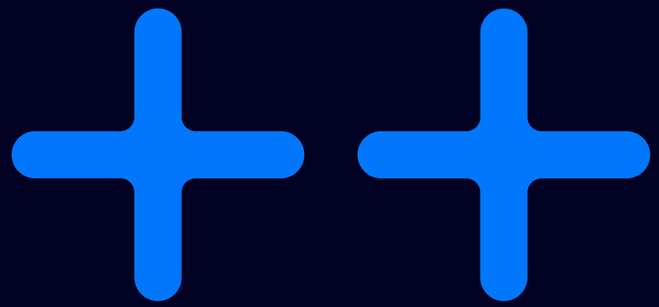
Итого

- Корутины ≈ callbacks
- Асинхронность позволяет экономить потоки и CPU
- Под капотом жесть, прикольная жесть!
- Снаружи — всё просто и понятно
- Во всех движках — очередь готовых к выполнению задач

Спасибо

C++ Zero Cost Conf */

Спасибо за внимание



Полухин Антон
Эксперт разработчик C++

antoshkka@yandex-team.ru
antoshkka@gmail.com

Yandex for developers */>