Comparing Approaches to Structured Concurrency

Adam Hearn

Software Engineer @ Amazon

Follow @adamhearn_

James Ward

Developer Advocate @ AWS

Follow @_JamesWard

What is Structured Concurrency?

Hierarchical Concurrency Which Generally Supports:

- Cancellation e.g. Races (loser cancellation)
- Resource management
- Efficient thread utilization (i.e. reactive, non-blocking)
- Explicit timeouts
- Semantic Errors

Easy Racer

github.com/jamesward/easyracer

> Ten Structured Concurrency "obstacle courses"

Scala 3 + ZIO	Kotlin + Coroutines	OCaml + Lwt + Cohttp
Scala 3 + Ox	Kotlin + Splitties	OCaml + Eio + Cohttp
Scala 3 + Kyo	Kotlin + Arrow	Python (Various)
Scala + Cats Effects 3	Rust + Tokio	C#
Java + Loom	Go	Elm

Approaches to Structured Concurrency

- Effect Oriented
  - Scala ZIO
    - Monadic Effect - God Monad
  - Scala Kyo
    - Algebraic Effects / single monad
- Direct Style (Imperative / Monad free!)
  - Scala Ox
    - Built on Loom, JDK21+ only
  - Rust (Future based syntax)
- Scoped Driven
  - Java Loom

Scenario 1

Race 2 concurrent requests

Scenario 1 - Scala ZIO

```
def scenario1(scenarioUrl: Int => String) =
  defer:
    val url = scenarioUrl(1)
    val req = Client.request(Request.get(url))
    val winner = req.race(req).run
    winner.body.asString.run
```

Scenario 1 - Scala Ox

```
def scenario1(scenarioUrl: Int => Uri): String =
  val url = scenarioUrl(1)
  def req = scenarioRequest(url).send(backend).body
  race(req, req)
```

Scenario 1 - Scala Kyo

```
  val req = Requests.run(Requests[String](_.get(url)))
  Fibers.race(req, req)

def scenario2(scenarioUrl: Int => Uri): String < Fibers =
```

Scenario 1 - Java Loom

```
public String scenario1() throws ExecutionException, InterruptedException {
    var req = HttpRequest.newBuilder(url.resolve("/1")).build();
    try (var scope = new StructuredTaskScope.ShutdownOnSuccess<HttpResponse<String>>()) {
        scope.fork(() -> client.send(req, HttpResponse.BodyHandlers.ofString()));
        scope.fork(() -> client.send(req, HttpResponse.BodyHandlers.ofString()));
        scope.join();
        return scope.result().body();
    }
}
```

Scenario 1 - Rust Tokio

```rust
pub async fn scenario_1(port: u16) -> String {

    async fn req(port: u16) -> Result<String, reqwest::Error> {
        reqwest::get(url(port, "1")).await?.text().await
    }

    tokio::select! {
        Ok(result) = req(port) => result,
        Ok(result) = req(port) => result,
        else => panic!("all failed")
    }
}
```

Scenario 2

Race 2 concurrent requests, where one produces a connection error

Scenario 2 - Scala ZIO

```
def scenario2(scenarioUrl: Int => String) =
  defer:
    val url = scenarioUrl(2)
    val req = Client.request(Request.get(url))
    val winner = req.race(req).run
    winner.body.asString.run
```

Scenario 2 - Scala Ox

```
def scenario2(scenarioUrl: Int => Uri): String =
  val url = scenarioUrl(2)
  def req = scenarioRequest(url).send(backend)
  race(req, req).body
```

Scenario 2 - Scala Kyo

```
  val req = Requests.run(Requests[String](_.get(url)))
  Fibers.race(req, req)

def scenario3(scenarioUrl: Int => Uri): String < Fibers =
```

Scenario 2 - Java Loom

```
public String scenario2() throws ExecutionException, InterruptedException {
    var req = HttpRequest.newBuilder(url.resolve("/2")).build();
    try (var scope = new StructuredTaskScope.ShutdownOnSuccess<HttpResponse<String>>()) {
        scope.fork(() -> client.send(req, HttpResponse.BodyHandlers.ofString()));
        scope.fork(() -> client.send(req, HttpResponse.BodyHandlers.ofString()));
        scope.join();
        return scope.result().body();
    }
}
```

Scenario 2 - Rust Tokio

```rust
pub async fn scenario_2(port: u16) -> String {

    async fn req(port: u16) -> Result<String, reqwest::Error> {
        reqwest::get(url(port, "2")).await?.text().await
    }

    tokio::select! {
        Ok(result) = req(port) => result,
        Ok(result) = req(port) => result,
        else => panic!("all failed")
    }
}
```

Scenario 3

Race 10,000 concurrent requests

Scenario 3 - Scala ZIO

```
def scenario3(scenarioUrl: Int => String) =
  defer:
    val url = scenarioUrl(3)
    val reqs = Seq.fill(10000)(Client.request(Request.get(url)))
    val winner = ZIO.raceAll(reqs.head, reqs.tail).run
    winner.body.asString.run
```

Scenario 3 - Scala Ox

```
def scenario3(scenarioUrl: Int => Uri): String =
  val url = scenarioUrl(3)
  val reqs = Seq.fill(10000): () =>
    scenarioRequest(url).send(backend)
  race(reqs).body
```

Scenario 3 - Scala Kyo

```
  val reqs = Seq.fill(10000):
    Requests.run(Requests[String](_.get(url)))
  Fibers.race(reqs)

def scenario4(scenarioUrl: Int => Uri): String < Fibers =
```

Scenario 3 - Java Loom

```
public String scenario3() throws ExecutionException, InterruptedException {
    var req = HttpRequest.newBuilder(url.resolve("/3")).build();
    try (var scope = new StructuredTaskScope.ShutdownOnSuccess<HttpResponse<String>>()) {
        IntStream.rangeClosed(1, 10_000)
                .forEach(i ->
                        scope.fork(() ->
                                client.send(req, HttpResponse.BodyHandlers.ofString())
                        )
                );

        scope.join();
        return scope.result().body();
    }
}
```

Scenario 3 - Rust Tokio

```rust
pub async fn scenario_3(port: u16) -> String {
    let client = reqwest::Client::new();

    let (tx, mut rx) = mpsc::channel(1);

    async fn req(port: u16, client: reqwest::Client) -> Result<String, reqwest::Error> {
        client.get(url(port, "3")).send().await?.text().await
    }

    for _ in 0..10_000 {
        let cloned_client = client.clone();
        let tx = tx.clone();
        tokio::spawn(async move {
            let result = req(port, cloned_client).await;
            tx.send(result).await
        });
    }

    rx.recv().await.unwrap().unwrap()
}
```

Scenario 4

Race 2 concurrent requests but 1 of them should have a 1 second timeout

Scenario 4 - Scala ZIO

```
def scenario4(scenarioUrl: Int => String) =
  defer:
    val url = scenarioUrl(4)
    val req = Client.request(Request.get(url))
    val winner = req.timeoutFail(TimeoutException())(1.seconds).race(req).run
    winner.body.asString.run
```

Scenario 4 - Scala Ox

```
def scenario4(scenarioUrl: Int => Uri): String =
  val url = scenarioUrl(4)
  def req = scenarioRequest(url).send(backend).body
  race(timeout(1.second)(req), req)
```

Scenario 4 - Scala Kyo

```
  val req = Requests.run(Requests[String](_.get(url)))
  Fibers.race(Fibers.timeout(1.seconds)(req), req)

def scenario5(scenarioUrl: Int => Uri): String < Fibers =
```

Scenario 4 - Java Loom

```
public String scenario4() throws ExecutionException, InterruptedException {
    var req = HttpRequest.newBuilder(url.resolve("/4")).build();
    try (var outer = new StructuredTaskScope.ShutdownOnSuccess<String>()) {
        outer.fork(() -> {
            try (var inner = new StructuredTaskScope.ShutdownOnSuccess<String>()) {
                inner.fork(() -> client.send(req, HttpResponse.BodyHandlers.ofString()).body());
                inner.joinUntil(Instant.now().plusSeconds(1));
                return inner.result();
            }
        });

        outer.fork(() -> client.send(req, HttpResponse.BodyHandlers.ofString()).body());

        outer.join();

        return outer.result();
    }
}
```

Scenario 4 - Rust Tokio

```rust
pub async fn scenario_4(port: u16) -> String {
    async fn req(port: u16) -> Result<String, reqwest::Error> {
        reqwest::get(url(port, "4")).await?.text().await
    }

    enum TimeoutOrRequestError {
        Timeout(Elapsed),
        RequestError,
    }

    async fn req_with_timeout(port: u16) -> Result<String, TimeoutOrRequestError> {
        let timeouted_result = timeout(Duration::from_secs(1), req(port));
        match timeouted_result.await {
            Ok(result) => result.map_err(|_| TimeoutOrRequestError::RequestError),
            Err(elapsed) => Err(TimeoutOrRequestError::Timeout(elapsed)),
        }
    }

    tokio::select! {
        Ok(result) = req(port) => result,
        Ok(result) = req_with_timeout(port) => result,
        else => panic!("all failed"),
    }
}
```

Scenario 5

Race 2 concurrent requests where a non-200 response is a loser

Scenario 5 - Scala ZIO

```
def scenario5(scenarioUrl: Int => String) =
  defer:
    val url = scenarioUrl(5)
    val req = Client.request(Request.get(url)).filterOrFail(_.status.isSuccess)(Error())
    val winner = req.race(req).run
    winner.body.asString.run
```

Scenario 5 - Scala Ox

```
def scenario5(scenarioUrl: Int => Uri): String =
  val url = scenarioUrl(5)
  def req = basicRequest.get(url).response(asString.getRight).send(backend).body
  race(req, req)
```

Scenario 5 - Scala Kyo

```
  val req = Requests.run(Requests[String](_.get(url)))
  Fibers.race(req, req)

def scenario6(scenarioUrl: Int => Uri): String < Fibers =
```

Scenario 5 - Java Loom

```
public String scenario5() throws ExecutionException, InterruptedException {
    class Req {
        final HttpRequest req = HttpRequest.newBuilder(url.resolve("/5")).build();

        HttpResponse<String> make() throws Exception {
            var resp = client.send(req, HttpResponse.BodyHandlers.ofString());
            if (resp.statusCode() == 200) {
                return resp;
            } else {
                throw new Exception("invalid response");
            }
        }
    }

    try (var scope = new StructuredTaskScope.ShutdownOnSuccess<HttpResponse<String>>()) {
        scope.fork(() -> new Req().make());
        scope.fork(() -> new Req().make());
        scope.join();
        return scope.result().body();
    }
}
```

Scenario 5 - Rust Tokio

```rust
pub async fn scenario_5(port: u16) -> String {

    async fn req(port: u16) -> Result<String, reqwest::Error> {
        let response = reqwest::get(url(port, "5")).await?;
        response.error_for_status()?.text().await
    }

    tokio::select! {
        Ok(result) = req(port) => result,
        Ok(result) = req(port) => result,
        else => panic!("all failed")
    }
}
```

Scenario 6

Race 3 concurrent requests where a non-200 response is a loser

Scenario 6 - Scala ZIO

```
def scenario6(scenarioUrl: Int => String) =
  defer:
    val url = scenarioUrl(6)
    val req = Client.request(Request.get(url)).filterOrFail(_.status.isSuccess)(Error())
    val winner = ZIO.raceAll(req, Seq(req, req)).run
    winner.body.asString.run
```

Scenario 6 - Scala Ox

```
def scenario6(scenarioUrl: Int => Uri): String =
  val url = scenarioUrl(6)
  def req = basicRequest.get(url).response(asString.getRight).send(backend).body
  race(req, req, req)
```

Scenario 6 - Scala Kyo

```
  val req = Requests.run(Requests[String](_.get(url)))
  Fibers.race(Seq(req, req, req))

def scenario7(scenarioUrl: Int => Uri): String < Fibers =
```

Scenario 6 - Java Loom

```
public String scenario6() throws ExecutionException, InterruptedException {
    class Req {
        final HttpRequest req = HttpRequest.newBuilder(url.resolve("/6")).build();

        HttpResponse<String> make() throws Exception {
            var resp = client.send(req, HttpResponse.BodyHandlers.ofString());
            if (resp.statusCode() == 200) {
                return resp;
            } else {
                throw new Exception("invalid response");
            }
        }
    }

    try (var scope = new StructuredTaskScope.ShutdownOnSuccess<HttpResponse<String>>()) {
        scope.fork(() -> new Req().make());
        scope.fork(() -> new Req().make());
        scope.fork(() -> new Req().make());
        scope.join();
        return scope.result().body();
    }
}
```

Scenario 6 - Rust Tokio

```rust
pub async fn scenario_6(port: u16) -> String {

    async fn req(port: u16) -> Result<String, reqwest::Error> {
        let response = reqwest::get(url(port, "6")).await?;
        response.error_for_status()?.text().await
    }

    tokio::select! {
        Ok(result) = req(port) => result,
        Ok(result) = req(port) => result,
        Ok(result) = req(port) => result,
        else => panic!("all failed")
    }
}
```

Scenario 7

Start a request, wait at least 3 seconds then start a second request (hedging)

Scenario 7 - Scala ZIO

```
def scenario7(scenarioUrl: Int => String) =
  defer:
    val url = scenarioUrl(7)
    val req = Client.request(Request.get(url))
    val winner = req.race(req.delay(4.seconds)).run
    winner.body.asString.run
```

Scenario 7 - Scala Ox

```
def scenario7(scenarioUrl: Int => Uri): String =
  val url = scenarioUrl(7)
  def req = scenarioRequest(url).send(backend).body
  def delayedReq =
    Thread.sleep(4000)
    req
  race(req, delayedReq)
```

Scenario 7 - Scala Kyo

```
  val req = Requests.run(Requests[String](_.get(url)))
  val delayedReq = Fibers.delay(4.seconds)(req)
  Fibers.race(req, delayedReq)

def scenario8(scenarioUrl: Int => Uri): String < Fibers =
```

Scenario 7 - Java Loom

```
public String scenario7() throws ExecutionException, InterruptedException {
    var req = HttpRequest.newBuilder(url.resolve("/7")).build();
    try (var scope = new StructuredTaskScope.ShutdownOnSuccess<HttpResponse<String>>()) {
        scope.fork(() -> client.send(req, HttpResponse.BodyHandlers.ofString()));
        scope.fork(() -> {
            Thread.sleep(3000);
            return client.send(req, HttpResponse.BodyHandlers.ofString());
        });
        scope.join();
        return scope.result().body();
    }
}
```

Scenario 7 - Rust Tokio

```rust
pub async fn scenario_7(port: u16) -> String {

    async fn req(port: u16) -> Result<String, reqwest::Error> {
        reqwest::get(url(port, "7")).await?.text().await
    }

    async fn hedge_req(port: u16) -> Result<String, reqwest::Error> {
        sleep(Duration::from_secs(3)).await;
        req(port).await
    }

    tokio::select! {
        Ok(result) = req(port) => result,
        Ok(result) = hedge_req(port) => result,
        else => panic!("all failed")
    }
}
```

Scenario 8

Race 2 concurrent requests that "use" a resource which is obtained and released through other requests. The "use" request can return a non-20x request, in which case it is not a winner.

Scenario 8 - Scala ZIO

```
def scenario8(scenarioUrl: Int => String) =
  def req(url: String) =
    defer:
      val resp = Client.request(Request.get(url)).filterOrFail(_.status.isSuccess)(Error()).run
      resp.body.asString.run

  val open = req(scenarioUrl(8) + "?open")
  def use(id: String) = req(scenarioUrl(8) + s"?use=$id")
  def close(id: String) = req(scenarioUrl(8) + s"?close=$id")

  val reqRes = ZIO.acquireReleaseWith(open)(close(_).orDie)(use)

  reqRes.race(reqRes)
```

Scenario 8 - Scala Ox

```
def scenario8(scenarioUrl: Int => Uri): String =
  def req(url: Uri) = basicRequest.get(url).response(asString.getRight).send(backend).body

  def open = req(uri"${scenarioUrl(8)}?open")
  def use(id: String) = req(uri"${scenarioUrl(8)}?use=$id")
  def close(id: String) = req(uri"${scenarioUrl(8)}?close=$id")

  def reqRes =
    val id = open
    try use(id)
    finally close(id)

  race(reqRes, reqRes)
```

Scenario 8 - Scala Kyo

```
  case class MyResource(id: String):
    def close: Unit < IOs =
      Fibers.run(req(uri"${scenarioUrl(8)}?close=$id")).unit

  val myResource = defer:
    val id = await:
      req(uri"${scenarioUrl(8)}?open")
    await:
      Resources.acquireRelease(MyResource(id))(_.close)

  val reqRes = Resources.run:
    defer:
      val resource = await(myResource)
      await:
        req(uri"${scenarioUrl(8)}?use=${resource.id}")

  Fibers.race(reqRes, reqRes)

def scenario9(scenarioUrl: Int => Uri): String < Fibers =
```

Scenario 8 - Java Loom

```
public String scenario8() throws InterruptedException, ExecutionException {
    class Req implements AutoCloseable {
        final HttpRequest openReq =
                HttpRequest.newBuilder(url.resolve("/8?open")).build();
        final Function<String, HttpRequest> useReq = (id) ->
                HttpRequest.newBuilder(url.resolve("/8?use=" + id)).build();
        final Function<String, HttpRequest> closeReq = (id) ->
                HttpRequest.newBuilder(url.resolve("/8?close=" + id)).build();

        final String id;

        public Req() throws IOException, InterruptedException {
            id = client.send(openReq, HttpResponse.BodyHandlers.ofString()).body();
        }

        HttpResponse<String> make() throws Exception {
            var resp = client.send(useReq.apply(id), HttpResponse.BodyHandlers.ofString());
            if (resp.statusCode() == 200) {
                return resp;
            } else {
                throw new Exception("invalid response");
            }
        }

        @Override
        public void close() throws IOException, InterruptedException {
            client.send(closeReq.apply(id), HttpResponse.BodyHandlers.ofString()).body();
        }
    }

    try (var scope = new StructuredTaskScope.ShutdownOnSuccess<HttpResponse<String>>()) {
        scope.fork(() -> {
            try (var req = new Req()) {
                return req.make();
            }
        });
        scope.fork(() -> {
            try (var req = new Req()) {
                return req.make();
            }
        });

        scope.join();

        return scope.result().body();
    }
}
```

Scenario 8 - Rust Tokio

```rust
pub async fn scenario_8(port: u16) -> String {

    async fn req_open(port: u16) -> Result<String, reqwest::Error> {
        let response = reqwest::get(url(port, "8?open")).await?;
        response.error_for_status()?.text().await
    }

    async fn req_use(port: u16, id: String) -> Result<String, reqwest::Error> {
        let response = reqwest::get(url(port, format!("8?use={}", id).as_str())).await?;
        response.error_for_status()?.text().await
    }

    async fn req_close(port: u16, id: String) -> Result<String, reqwest::Error> {
        let response = reqwest::get(url(port, format!("8?close={}", id).as_str())).await?;
        response.error_for_status()?.text().await
    }

    async fn req(port: u16) -> Result<String, reqwest::Error> {
        let id = req_open(port).await?;
        let resp = req_use(port, id.clone()).await;
        let _ = req_close(port, id).await;
        resp
    }

    tokio::select! {
        Ok(result) = req(port) => result,
        Ok(result) = req(port) => result,
        else => panic!("all failed")
    }
}
```

Scenario 9

Make 10 concurrent requests where 5 return a 200 response with a letter

Scenario 9 - Scala ZIO

```
def scenario9(scenarioUrl: Int => String) =
  val req =
    defer:
      val url = scenarioUrl(9)
      val resp = Client.request(Request.get(url)).filterOrFail(_.status.isSuccess)(Error()).run
      val body = resp.body.asString.run
      val now = Clock.nanoTime.run
      now -> body

  defer(Use.withParallelEval):
    val responses = Queue.unbounded[(Long, String)].run
    for _ <- 1 to 10 do
      req.option.run match
        case Some(resp) => responses.offer(resp).run
        case None => ()

    responses.takeAll.run.to(SortedMap).values.mkString
```

Scenario 9 - Scala Ox

```
def scenario9(scenarioUrl: Int => Uri): String =
  def req =
    val body = basicRequest.get(scenarioUrl(9)).response(asString.getRight).send(backend).body
    val now = System.nanoTime
    now -> body

  unsupervised:
    val forks = Seq.fill(10)(forkUnsupervised(req))
    forks.map(_.joinEither()).collect:
      case Right(v) => v
    .sortBy(_._1).map(_._2).mkString
```

Scenario 9 - Scala Kyo

```
  val req = IOs.attempt:
    defer:
      val body = await(Requests.run(Requests[String](_.get(url))))
      val now = await(Clocks.now)
      now -> body

  val reqs = Seq.fill(10)(req)

  defer:
    val successes = await(Fibers.parallel(reqs)).collect:
      case scala.util.Success(value) => value

    successes.sortBy { (completedTime, _) => completedTime }.map { (_, body) => body }.mkString

def scenario10(scenarioUrl: Int => Uri): String < Fibers =
```

Scenario 9 - Java Loom

```
public String scenario9() throws InterruptedException {
    record TimedResponse(Instant instant, HttpResponse<String> response) {
    }

    final HttpRequest req = HttpRequest.newBuilder(url.resolve("/9")).build();
    try (var scope = new StructuredTaskScope<TimedResponse>()) {
        var futures = IntStream.rangeClosed(1, 10)
                .mapToObj(i ->
                        scope.fork(() -> {
                            var resp = client.send(req, HttpResponse.BodyHandlers.ofString());
                            return new TimedResponse(Instant.now(), resp);
                        })
                ).toList();

        scope.join();

        return futures.stream()
                .map(StructuredTaskScope.Subtask::get)
                .filter(r -> r.response.statusCode() == 200)
                .sorted(Comparator.comparing(TimedResponse::instant)).collect(
                    StringBuilder::new,
                    (acc, timedResponse) -> acc.append(timedResponse.response.body()),
                    StringBuilder::append
                ).toString();
    }
}
```

Scenario 9 - Rust Tokio

```rust
pub async fn scenario_9(port: u16) -> String {

    async fn req(port: u16) -> Result<(String, Instant), reqwest::Error> {
        let response = reqwest::get(url(port, "9")).await?;
        let text = response.error_for_status()?.text().await?;
        let now = Instant::now();
        Ok((text, now))
    }

    let responses_tuple = tokio::join!(
        req(port),
        req(port),
        req(port),
        req(port),
        req(port),
        req(port),
        req(port),
        req(port),
        req(port),
        req(port),
    );

    let responses = vec![
        responses_tuple.0,
        responses_tuple.1,
        responses_tuple.2,
        responses_tuple.3,
        responses_tuple.4,
        responses_tuple.5,
        responses_tuple.6,
        responses_tuple.7,
        responses_tuple.8,
        responses_tuple.9,
    ];

    let mut ok_responses: Vec<&(String, Instant)> = responses.iter().filter_map(|response| response.as_ref().ok()).collect();

    ok_responses.sort_by(|a, b| a.1.cmp(&b.1));

    ok_responses.iter().fold("".to_string(), |acc, response| acc + &response.0)
}
```

Scenario 10

This scenario validates that a computationally heavy task can be run in parallel to another task, and then cancelled.

Part 1) Make a request and while the connection is open, perform something computationally heavy (e.g. repeated SHA calculation), then cancel the task when the connection closes
Part 2) In parallel to **Part 1**, every 1 second, make a request with the current process load (0 to 1)

The request in **Part 2** will respond with a 20x response if it looks like **Part 1** was done correctly (in which case you can stop sending load values), otherwise it will respond with a 30x response if you should continue sending values, or with a 40x response if something has gone wrong.

Scenario 10 - Scala ZIO

```
def scenario10(scenarioUrl: Int => String) =

  def reporter(id: String): ZIO[Client & Scope, Throwable, String] =
    val osBean = ManagementFactory.getPlatformMXBean(classOf[OperatingSystemMXBean])
    val load = osBean.getProcessCpuLoad * osBean.getAvailableProcessors
    defer:
      val resp = Client.request(Request.get(scenarioUrl(10) + s"?$id=$load")).run
      if resp.status.isRedirection then
        reporter(id).delay(1.second).run
      else if resp.status.isSuccess then
        resp.body.asString.run
      else
        val body = resp.body.asString.run
        ZIO.fail(Error(body)).run

  defer:
    val id = Random.nextString(8).run
    val messageDigest = MessageDigest.getInstance("SHA-512")
    val seed = Random.nextBytes(512).run

    val blocking = ZIO.attemptBlockingInterrupt:
      var result = seed.toArray
      while (!Thread.interrupted())
        result = messageDigest.digest(result)

    val blocker =
      Client.request(Request.get(scenarioUrl(10) + s"?$id")).race(blocking) *> ZIO.never

    blocker.fork.run
    reporter(id).run
```

Scenario 10 - Scala Ox

```
def scenario10(scenarioUrl: Int => Uri): String =
  val id = Random.nextString(8)

  def req(url: Uri) =
    basicRequest.get(url).response(asStringAlways).send(backend)

  val messageDigest = MessageDigest.getInstance("SHA-512")

  def blocking(): Unit =
    var result = Random.nextBytes(512)
    while (!Thread.interrupted())
      result = messageDigest.digest(result)

  def blocker =
    val url = scenarioUrl(10).addQuerySegment(QuerySegment.Plain(id))
    race(req(url), blocking())

  @tailrec
  def reporter: String =
    val osBean = ManagementFactory.getPlatformMXBean(classOf[OperatingSystemMXBean])
    val load = osBean.getProcessCpuLoad * osBean.getAvailableProcessors
    val resp = req(scenarioUrl(10).addQuerySegment(QuerySegment.KeyValue(id, load.toString)))
    if resp.code.isRedirect then
      Thread.sleep(1000)
      reporter
    else if resp.code.isSuccess then
      resp.body
    else
      throw Error(resp.body)

  val (_, result) = par(blocker, reporter)
  result
```

Scenario 10 - Scala Kyo

```
def req(uriModifier: Uri => Uri): (String, ResponseMetadata) < Fibers =
  val uri = uriModifier(scenarioUrl(10))
  Requests.run:
    Requests[(String, ResponseMetadata)]:
      _.get(uri).response:
        asString.mapWithMetadata: (stringEither, responseMetadata) =>
          Right(stringEither.fold(identity, identity) -> responseMetadata)

val messageDigest = MessageDigest.getInstance("SHA-512")

// recursive digesting
def blocking(bytesEffect: Seq[Byte] < IOs): Seq[Byte] < IOs =
  IOs:
    bytesEffect.map: bytes =>
      blocking(messageDigest.digest(bytes.toArray).toSeq)

// runs blocking code while the request is open
def blocker(id: String): String < Fibers =
  Fibers.race(
    req(_.addQuerySegment(QuerySegment.Plain(id))).map { (body, _) => body },
    blocking(Randoms.nextBytes(512)).map(_ => ""),
  )

// sends CPU usage every second until the server says to stop
def reporter(id: String): String < Fibers =
  val osBean = ManagementFactory.getPlatformMXBean(classOf[OperatingSystemMXBean])
  val load = osBean.getProcessCpuLoad * osBean.getAvailableProcessors

  defer:
    val (maybeResponseBody, responseMetadata) =
      await(req(_.addQuerySegment(QuerySegment.KeyValue(id, load.toString))))

    if responseMetadata.isRedirect then
      await(Fibers.delay(1.seconds)(reporter(id)))
    else if responseMetadata.isSuccess then
      maybeResponseBody
    else
      await(Fibers.get(IOs.fail(Error(maybeResponseBody))))

defer:
  val id = await(Randoms.nextStringAlphanumeric(8))
  val (_, result) = await(Fibers.parallel(blocker(id), reporter(id)))
  result
```

Scenario 10 - Java Loom

```
    public String scenario10() throws InterruptedException {
        var id = UUID.randomUUID().toString();

        Supplier<String> blocker = () -> {
            try (var scope = new StructuredTaskScope.ShutdownOnSuccess<HttpResponse<String>>()) {
                var req = HttpRequest.newBuilder(url.resolve(STR."/10?\{id}")).build();
                var messageDigest = MessageDigest.getInstance("SHA-512");

                scope.fork(() -> client.send(req, HttpResponse.BodyHandlers.ofString()));
                scope.fork(() -> {
                    var result = new byte[512];
                    new Random().nextBytes(result);
                    while (!Thread.interrupted())
                        result = messageDigest.digest(result);
                    return null;
                });
                scope.join();
                return scope.result().body();
            } catch (ExecutionException | InterruptedException | NoSuchAlgorithmException e) {
                throw new RuntimeException(e);
            }
        };

        class Recursive<I> {
            public I func;
        }

        Recursive<Supplier<String>> recursive = new Recursive<>();
        recursive.func = () -> {
            var osBean = ManagementFactory.getPlatformMXBean(OperatingSystemMXBean.class);
            var load = osBean.getProcessCpuLoad() * osBean.getAvailableProcessors();
            var req = HttpRequest.newBuilder(url.resolve(STR."/10?\{id}=\{load}")).build();
            try {
                var resp = client.send(req, HttpResponse.BodyHandlers.ofString());
                if ((resp.statusCode() >= 200) && (resp.statusCode() < 300)) {
                    return resp.body();
                }
                else if ((resp.statusCode() >= 300) && (resp.statusCode() < 400)) {
                    Thread.sleep(1000);
                    return recursive.func.get();
                }
                else {
                    throw new RuntimeException(resp.body());
                }
            } catch (IOException | InterruptedException e) {
                throw new RuntimeException(e);
            }
        };

        try (var scope = new StructuredTaskScope<String>()) {
            scope.fork(blocker::get);
            var task = scope.fork(recursive.func::get);
            scope.join();
            return task.get();
        }
    }

    List<String> results() throws ExecutionException, InterruptedException {
        return List.of(scenario1(), scenario2(), scenario3(), scenario4(), scenario5(), scenario6(), scenario7(), scenario8(), scenario9());
        //return List.of(scenario10());
    }
}
```

Scenario 10 - Rust Tokio

```rust
pub async fn scenario_10(port: u16) -> String {
    async fn req(port: u16, params: String) -> Result<Response, reqwest::Error> {
        reqwest::get(url(port, format!("10?{}", params).as_str())).await
    }

    async fn blocking(cancellation_token: CancellationToken) -> Result<(), JoinError> {
        tokio::spawn(async move {
            while !cancellation_token.is_cancelled() {
                let mut hasher = Sha512::new();
                let mut bytes = [0u8; 512];
                thread_rng().fill_bytes(&mut bytes);
                hasher.update(bytes);
                hasher.finalize();
            }
        }).await
    }

    // might be a better way than the CancellationToken
    async fn blocker(port: u16, id: String) {
        let token = CancellationToken::new();
        let cloned_token = token.clone();

        tokio::select! {
            Ok(_) = req(port, id) => (),
            Ok(_) = blocking(cloned_token) => (),
            else => panic!("all failed")
        }

        token.cancel()
    }

    let random_string: String = thread_rng()
        .sample_iter(&Alphanumeric)
        .take(8)
        .map(char::from)
        .collect();

    fn current_load(previous_stime: i64) -> (i64, f64) {
        let stime = stat_self().unwrap().utime;
        let cpu_usage = (stime - previous_stime) as f64 / ticks_per_second() as f64;
        return (stime, cpu_usage);
    }

    #[async_recursion]
    async fn reporter(port: u16, id: String, previous_stime: i64) -> String {
        let (stime, load) = current_load(previous_stime);
        let resp = req(port, format!("{}={}", id, load)).await.unwrap();
        let status = resp.status();
        if status.is_success() {
            return resp.text().await.unwrap()
        }
        else if status.is_redirection() {
            sleep(Duration::from_secs(1)).await;
            return reporter(port, id, stime).await;
        }
        else {
            panic!("{}", resp.text().await.unwrap());
        }
    }

    tokio::spawn(
        blocker(port, random_string.clone())
    );

    return reporter(port, random_string.clone(), 0).await;
}
```

Schedulers

- Abstractions
- Limiting costly context switching & thread overhead
- Expressing effects in terms of values / operations, not concerned with how the scheduler does its thing
- Upgrades can change the impl

Composability

Performance