Package 'mizu'

November 4, 2026

Title Thread-Speed 'Async' and Parallel R over Shared Memory

Version 0.1.0

Description Parallel computation and data exchange between R processes on the same machine: lock-free channels, work-stealing task pools, and a parallel map over 'POSIX' shared memory (Linux, macOS) or 'Win32' file mappings (Windows), with the hot path entirely in user space. Communication is cheap enough that work can be divided as finely as with threads. Vectors and data frames transfer zero-copy, received as 'ALTREP' views over the shared pages, and 'Apache Arrow' tables and record batches travel as a single frame with zero-copy conversion in both directions. Receives and task results integrate with 'later' and 'promises' for non-blocking, event-driven applications such as 'shiny'. Channels, pools, and maps interoperate with 'pymizu', so a peer or worker can equally be a Python process. It complements 'mirai', and its payloads interoperate with 'mori' shared objects.

License MIT + file LICENSE

URL https://shikokuchuo.net/mizu/, https://github.com/shikokuchuo/mizu

BugReports https://github.com/shikokuchuo/mizu/issues

Depends R (>= 4.3)

Suggests arrow, bit64, later, mori, nanoarrow, promises, quarto, rlang, testthat (>= 3.0.0)

VignetteBuilder quarto

Config/build/compilation-database true

Config/Needs/website tidyverse/tidytemplate

Config/roxygen2/markdown TRUE

Config/roxygen2/version 8.1.1

Config/testthat/edition 3

Encoding UTF-8

NeedsCompilation yes

Author Charlie Gao [aut, cre] (ORCID: <https://orcid.org/0000-0002-0750-061X>),
Posit Software, PBC [cph, fnd],
Pierre L'Ecuyer [cph] (RngStreams library)

Maintainer Charlie Gao <charlie.gao@posit.co>

Repository CRAN

Date/Publication 2026-11-04 18:30:02 UTC

Contents

mizu_alive . . . 3
mizu_as_arrow . . . 3
mizu_call . . . 4
mizu_cancel . . . 6
mizu_channel . . . 7
mizu_channel_info . . . 8
mizu_close . . . 9
mizu_collect_all . . . 10
mizu_collect_all_async . . . 11
mizu_collect_any . . . 13
mizu_current_pool . . . 14
mizu_default_pool . . . 15
mizu_error . . . 16
mizu_is_remote_error . . . 18
mizu_is_sentinel . . . 19
mizu_launcher . . . 20
mizu_map . . . 21
mizu_map_prepare . . . 26
mizu_pool . . . 28
mizu_pool_attach . . . 30
mizu_pool_dump . . . 31
mizu_pool_info . . . 32
mizu_pool_stats . . . 32
mizu_pool_stop . . . 33
mizu_pool_trace . . . 34
mizu_prune . . . 35
mizu_py_launcher . . . 36
mizu_recv_async . . . 37
mizu_send . . . 39
mizu_send_batch . . . 40
mizu_spawn_workers . . . 41
mizu_submit . . . 43
mizu_submit_batch . . . 45
print.mizu_channel . . . 46

Index **48**

mizu_alive *Check Whether the Channel Peer Is Alive*

Description

Reports whether the peer process still holds its liveness lock, in about a microsecond, without touching the rings. `mizu_recv()` and `mizu_send()` report peer death automatically as `mizu_peer_gone`. This function asks directly. A peer that closed the channel but still runs counts as alive.

Usage

```
mizu_alive(ch)
```

Arguments

`ch`	a channel handle from `mizu_channel()` (or the `ch` binding inside a peer expression).

Value

`TRUE` while the peer process is alive, `FALSE` after it dies. By then the survivor has deleted the channel's shared memory.

Examples

```
ch <- mizu_channel(quote(mizu_recv(ch)))
mizu_alive(ch)
mizu_close(ch)
```

mizu_as_arrow *Convert a Received Frame to an Arrow Stream*

Description

`mizu_as_arrow()` converts a data frame, typically one received over a channel or collected from a pool, into a struct-typed `nanoarrow_array_stream`, ready for `nanoarrow::convert_array_stream()`, `arrow::as_arrow_table()`, or any other Arrow C Data Interface consumer.

Usage

```
mizu_as_arrow(x)
```

Arguments

`x`	a data frame.

Details

Frames arrive over channels and pools alike. Arrow data sent at top level is staged as one frame region, and Arrow data nested inside a payload (a list element, a task argument, the task expression itself) crosses by reference, so the received object holds a view in place of the table. A collected pool result is that view, and this function converts it back explicitly. One caveat on sends: the conversion warnings (a uint64 past 2^63 - 1 becoming `NA`, an int32 or int64 minimum reading as `NA`) fire at send over a channel, but at a pool publish they fire on the worker and the submitter never sees them.

Columns still backed by their received shared-memory pages export zero-copy: the stream's buffers point directly into the shared region for attribute-free double, integer, `integer64`, and character views (character as `large_string`). This covers any column backed by a shared region (the frame's own columns, columns referencing another region, and received views assembled into a data frame), with one reference held per region. A region whose producer has exited affects only its own columns, which fall back to copying. Everything else (factors, Date, POSIXct and difftime columns, logical and raw vectors, and materialized or never-shared columns) converts per column through `nanoarrow::as_nanoarrow_array()`. The shared region stays mapped until the stream and every array produced from it are released.

Value

A struct-typed `nanoarrow_array_stream`.

Examples

```
ch <- mizu_channel(quote(mizu_send(ch, mizu_recv(ch))))
mizu_send(ch, arrow::arrow_table(data.frame(x = runif(1e5), s = "a")))
df <- mizu_recv(ch, timeout = 30)
stream <- mizu_as_arrow(df)
arrow::as_arrow_table(stream)
mizu_close(ch)
```

`mizu_call` *Call a Function in Another Language*

Description

`mizu_call()` builds a task specification: a qualified name or a source string in the pool workers' language, plus the constant arguments. `mizu_submit_call()` stages it as a neutral task stream that any worker language reads. This is how you drive a pool of workers in another language: spawn the workers with `mizu_py_launcher()` and submit specs to them.

Usage

```
mizu_call(.name = NULL, ..., .source = NULL)

mizu_submit_call(pool, spec, .timeout = Inf)
```

Arguments

.name	[character(1)] a qualified function name: "pkg::fn" (or "pkg:::fn") for R workers, "mod.fn" for Python workers. The qualifier is required because the name resolves in the worker's own namespace, which is the runner module rather than the submitter's global environment.
...	the constant arguments. Unnamed arguments map to the positional argument list and named ones to the named argument dict, matching Python's *args and **kwargs. Arguments must be portable values (the interchange subset documented in mizu_send()). A non-portable argument raises mizu_error_not_portable at submit.
.source	[character(1)] source code in the workers' language. Evaluated in a fresh namespace (parented on the global environment in R, a fresh dict over __builtins__ in Python) with the named arguments bound as names and positional arguments bound as ..1, ..2, ... The result is the trailing expression's value, or NULL when the source ends with a statement.
pool	a pool handle (see mizu_pool()).
spec	a "mizu_call" specification.
.timeout	maximum seconds to wait for submission capacity.

Details

The spec describes a call, not a value. The language is not named at the call site: mizu_submit_call() reads the workers' language from the pool itself (for attached submitters too), and a bare unqualified name errors at submit rather than when the spec is built.

Results and errors cross in the submitter's own formats. A task error arrives as a mizu_error_remote condition (see mizu_is_remote_error()), and a result that is not portable fails the task with a condition naming the value's type. On a same-language pool, mizu_submit_call() keeps the full same-language error format.

Large arguments cross to foreign workers by reference rather than by copy. A fresh value past the zero-copy floor is written once into a shared region, and the worker reads a view over it. An argument that is already a shared view (received from a channel, a pool result, or a mori::share()d vector) crosses as its identifier alone, with no payload bytes. On a pool whose workers are too old to read a reference, such a task fails at submit with an error naming the fix.

Value

mizu_call(): a "mizu_call" specification (a classed list). mizu_submit_call(): a task handle (class "mizu_task"), exactly as mizu_submit() returns.

Examples

```
p <- mizu_pool()
# same-language pools take specs too
t <- mizu_submit_call(p, mizu_call("stats::quantile", runif(100)))
mizu_collect(t)
mizu_pool_stop(p)
```

```
p <- mizu_pool(2, launcher = mizu_py_launcher())
t <- mizu_submit_call(p, mizu_call("numpy.mean", runif(100)))
mizu_collect(t)
mizu_pool_stop(p)
```

`mizu_cancel` *Cancel a Task or a Map*

Description

Cancellation is advisory: it never interrupts running code. The worker skips a task that is still queued. A task already executing runs to completion, and its result is dropped. Collecting a cancelled handle raises `mizu_error_cancelled` (see mizu_error).

Usage

```
mizu_cancel(task)
```

Arguments

`task`	a task handle from `mizu_submit()`, or a map handle from `mizu_map()` / `mizu_map_run()`.

Details

On a map handle from `mizu_map()` / `mizu_map_run()`, the shared cancel word of the map is set: executing runners stop within about one batch of their next transition, and every outstanding runner task is cancelled as above. A blocking `mizu_collect()` on the handle raises `mizu_error_cancelled`, and a pending promise rejects with it.

Value

Invisibly, `TRUE` if this call cancelled the task. `FALSE` if the call was too late: the task completed, was already cancelled, or its pool is gone.

Examples

```
p <- mizu_pool()
t <- mizu_submit(p, runif(1))
mizu_cancel(t)
mizu_pool_stop(p)
```

`mizu_channel` *Create a Channel and Spawn Its Peer*

Description

Creates a shared-memory single-producer single-consumer channel, with one lock-free ring per direction, and spawns a child R process connected to the other end. The returned handle sends on the host-to-peer ring and receives on the peer-to-host ring. Only a short join token (the suffix of the region name) is passed to the child, as a command-line argument.

Usage

```
mizu_channel(
  expr,
  ...,
  capacity = 16384L,
  slot_size = 256L,
  arena_size = 4194304,
  spin = FALSE,
  startup_timeout = 30,
  launcher = mizu_launcher()
)
```

Arguments

`expr`	a quoted expression (for example `quote({ ... })`), evaluated in the peer process with `ch` bound to the peer-side channel handle. A single character string is instead UTF-8 source text in the peer's language, for a non-R peer spawned by a custom `launcher` (an R peer parses and evaluates it as R source).
`...`	These dots are for future extensions and must be empty.
`capacity`	slots per ring. A power of two between 2 and 2^24.
`slot_size`	bytes per slot. A power of two between 64 and 2^20. The inline payload budget is `slot_size - 16`. Payloads that serialize larger spill to the arena, and past it to a fresh region.
`arena_size`	spill-arena bytes per direction. A multiple of 128, or 0 to disable (every spill then creates a region). This bounds the in-flight spill. An undersized arena falls back to creating a region per payload, and never errors.
`spin`	`TRUE` makes the channel spin-wait: consumers never park and producers skip the wake check on publish. Use this only when the consumer never yields, because a spin-mode consumer that parks is never woken.
`startup_timeout`	seconds to wait for the peer to attach and signal ready. On expiry, mizu releases the channel and raises `mizu_error_startup` (see mizu_error).

launcher — a function(token) that spawns the peer process. For an R peer, it arranges for a process to call mizu:::peer_main(token). The default mizu_launcher() spawns Rscript and propagates the .libPaths() of the host. Its stdout and stderr arguments direct the peer output, including the error message. A custom launcher must arrange the library paths itself. For a peer in another language (a source-string expr), it spawns a program that attaches with token and speaks the wire protocol, such as python3 -m pymizu.child for a Python peer. mizu_py_launcher() is the ready-made launcher for that case.

Details

expr is a quoted expression, not a closure. It captures nothing, and unlike mizu_submit(), mizu does not quote it for you. The peer evaluates it in a fresh environment whose parent is the child's global environment, with ch (the peer-side channel handle) as the only binding that mizu provides. The expression must load any packages it needs. When the expression returns or errors, the peer signals an orderly close and exits. An error message goes to the child's standard error.

Payloads interoperate with mori. A mori::share()d object anywhere inside a payload is serialized as its short identifier through the mori hooks, and is mapped zero-copy on the other side.

The channel's two liveness lock files, which detect the death of either side, are created in a per-platform directory: /dev/shm on Linux, and the per-user temporary directory on macOS and Windows. The path is recorded in the region, so both sides use the same files. The environment variable MIZU_LIVENESS_DIR, read in the creating process, overrides the default.

Value

A channel handle (class "mizu_channel"). Handles are process-private and do not survive fork().

Examples

```
ch <- mizu_channel(quote(
  repeat {
    x <- mizu_recv(ch, timeout = 30)
    if (inherits(x, "mizu_sentinel")) break
    mizu_send(ch, x)
  }
))
mizu_send(ch, 42L)
mizu_recv(ch, timeout = 5)
mizu_close(ch)
```

---

mizu_channel_info *Inspect a Channel*

Description

A read-only snapshot of the channel region and the handle-local counters: ring geometry and watermarks, the free-list and open caches, and the notify signals sent to an armed async consumer.

Usage

```
mizu_channel_info(ch)
```

Arguments

ch	a channel handle from mizu_channel() (or the ch binding inside a peer expression).

Value

A list with elements name (the platform region name, kept for debugging), side, capacity, slot_size, arena_size, inline_max, spin, ready, closed, peer_pid, tx_sent, tx_published, tx_consumed, rx_consumed, rx_published, fl_entries, fl_hits, open_hits, open_misses, ledger_entries, zc_open_hits, zc_open_misses, and notify_sends.

Examples

```
ch <- mizu_channel(quote(mizu_recv(ch)))
mizu_channel_info(ch)
mizu_close(ch)
```

---

mizu_close *Close a Channel*

Description

Signals close to the peer, then waits up to timeout seconds for the peer to close (or die). The peer confirms only after it has finished reading its messages, which is what makes it safe to release the resources held by sent payloads. On that confirmation, all resources are released and the region name is deleted. The handle is dead afterwards, and closing it again does nothing. On timeout the handle stays usable, and the resources are released at garbage collection, which runs the same check again.

Usage

```
mizu_close(ch, timeout = 5)
```

Arguments

ch	a channel handle from mizu_channel() (or the ch binding inside a peer expression).
timeout	seconds to wait for the close of the peer.

Details

After either side signals close, sends on both sides return the mizu_closed sentinel, and receives drain the remaining messages before they return the same.

Value

Invisibly, `TRUE` on a clean mutual close, `FALSE` on timeout (with a warning).

Examples

```
ch <- mizu_channel(quote(mizu_send(ch, "done")))
mizu_recv(ch, timeout = 5)
mizu_close(ch)
```

`mizu_collect_all` *Collect the Results of Several Tasks, in Order*

Description

`mizu_collect_all()` waits until every task in `tasks` reaches a terminal state and returns all results in input order. It is the batch counterpart of `mizu_collect()`, with one R call for the whole set instead of one per task. The wait sleeps until a publishing worker wakes it directly, with no polling.

Usage

```
mizu_collect_all(tasks, timeout = Inf)
```

Arguments

`tasks`	a non-empty list of task handles from `mizu_submit()` on the same pool handle.
`timeout`	seconds to wait for every task to reach a terminal state before the call returns the `mizu_timeout` sentinel. `Inf` (the default) waits indefinitely; `0` does not wait.

Details

For homogeneous element-wise work, use `mizu_map()`, which batches submission and staging as well as collection. `mizu_collect_all()` is for sets of tasks with different expressions and arguments, which is what the per-task API is for.

Value

A plain list of the task values in the order of `tasks`; the names of `tasks` carry over. On timeout, the `mizu_timeout` sentinel.

Outcomes

As for `mizu_collect()`, attributed to a handle by position as in `mizu_collect_any()`:

outcome	surfaced as	class
all results published	list of values, returned	—
not all terminal within `timeout`	sentinel, returned	`c("mizu_timeout", "mizu_sentinel")`

a task raised an error	re-signalled with an `index` field	the condition classes of the task itself
a task cancelled, or pool stopped	raised with an `index` field	`mizu_error_cancelled`
an executing worker died	raised with an `index` field	`mizu_error_worker_died`

The `index` field of a raised condition is the 1-based position in `tasks` of the first such task. Only the reported handle is consumed. Every other handle, including those ahead of it, stays valid and collectible. A timeout consumes nothing, so every handle stays valid and collectible.

Examples

```
p <- mizu_pool(2L)
ts <- list(
  total = mizu_submit(p, sum(x), x = runif(10)),
  label = mizu_submit(p, "done")
)
mizu_collect_all(ts, timeout = 30)
mizu_pool_stop(p)
```

`mizu_collect_all_async` *Collect Pool Results as Promises*

Description

`mizu_collect_all_async()` and `mizu_collect_any_async()` are the async counterparts of `mizu_collect_all()` and `mizu_collect_any()`. Each returns immediately with one `promises::promise()` for the whole collection, and no per-task promise is allocated. On each wake the pool checks for newly finished tasks, and when the collection completes, the result is assembled once as in the blocking version and the promise resolves.

Usage

```
mizu_collect_all_async(tasks, timeout = Inf)

mizu_collect_any_async(tasks, timeout = Inf)

mizu_collect_async(task, timeout = Inf)

## S3 method for class 'mizu_task'
as.promise(x)

## S3 method for class 'mizu_task'
is.promising(x)

## S3 method for class 'mizu_map'
```

```
as.promise(x)

## S3 method for class 'mizu_map'
is.promising(x)
```

Arguments

tasks	a non-empty list of task handles from mizu_submit() on the same pool handle.
timeout	seconds after which the promise rejects with the mizu_timeout classed error; nothing is consumed. Inf (the default) waits indefinitely.
task	a task handle from mizu_submit().
x	for the as.promise methods: a task handle, or a map handle from mizu_map() / mizu_map_run().

Details

mizu_collect_async() is the single-task form, the async counterpart of mizu_collect(). It returns immediately with a promises::promise() that resolves with the task's value and rejects with the same classed condition a blocking collect would raise. There is one promise per task. While the task is pending the promise is cached, so a repeated call, or as.promise() on the handle (which promises::then() calls), returns the same promise, and timeout applies only to the call that created it. Once settled the task is collected, and a further call errors (one consumer per result, below). A timeout rejects with the mizu_timeout classed error and consumes nothing, so the task can be collected with a blocking call again.

The behavior mirrors the blocking functions. Collect-all resolves in input order with names carried over, and the first erroring handle by position rejects with index (only that handle is consumed, the rest stay collectible). Collect-any resolves list(index, value) for the first terminal task, and a raised outcome rejects with the index field. A timeout rejects with the mizu_timeout classed error and consumes nothing. When a collection finishes, whether resolved, rejected, or timed out, every handle in it is released, and handles whose results were not consumed can be collected with a blocking call again.

A task result belongs to exactly one consumer. A task with a promise attached, or one that is part of a pending collection, refuses mizu_collect(), mizu_collect_any() and mizu_collect_all() with an error naming the pending consumer, and a task enrolled in a collection likewise refuses a promise. mizu_cancel() is still allowed and is the destructive option: the cancel publishes, the result is gone for every consumer, and the pending promise rejects with mizu_error_cancelled. A timeout is the non-destructive counterpart. It only detaches the promise, and the task keeps running with its result still collectible. (A channel message instead belongs to whoever takes it first; see mizu_recv_async().)

Requires the **later** and **promises** packages.

Value

A promises::promise(): mizu_collect_async() resolves the task's value; collect-all resolves a plain list of the task values in the order of tasks (names carried); collect-any resolves list(index, value).

Examples

```
library(promises)
p <- mizu_pool(2L)
t <- mizu_submit(p, sum(runif(10)))
mizu_collect_async(t, timeout = 5) %...>% print()
ts <- list(
  total = mizu_submit(p, sum(x), x = runif(10)),
  label = mizu_submit(p, "done")
)
mizu_collect_all_async(ts) %...>% print()
later::run_now()
mizu_pool_stop(p)
```

`mizu_collect_any` *Collect the First Available Result From Several Tasks*

Description

`mizu_collect_any()` waits on several task handles at once and returns as soon as any of them reaches a terminal state (result published, error raised, cancelled, or its worker died). Among handles already terminal, the earliest in `tasks` is reported. The wait sleeps until a publishing worker wakes it directly, with no polling. For the whole set at once, `mizu_collect_all()` waits until every task is terminal and returns all results in input order.

Usage

```
mizu_collect_any(tasks, timeout = Inf)
```

Arguments

`tasks`	a non-empty list of task handles from `mizu_submit()` on the same pool handle.
`timeout`	seconds to wait for a task to reach a terminal state before the call returns the `mizu_timeout` sentinel. `Inf` (the default) waits indefinitely; `0` does not wait.

Value

For a published result, `list(index = i, value = v)`: the 1-based position of the task in `tasks` and its value. Otherwise the `mizu_timeout` sentinel.

Outcomes

As for `mizu_collect()`, but attributed to a handle by position:

outcome	surfaced as	class
result published	`list(index, value)`, returned	—
nothing terminal within `timeout`	sentinel, returned	`c("mizu_timeout", "mizu_sentinel")`

task raised an error	re-signalled with an `index` field	the condition classes of the task itself
cancelled, or pool stopped	raised with an `index` field	`mizu_error_cancelled`
executing worker died	raised with an `index` field	`mizu_error_worker_died`

The `index` field of a raised condition is the 1-based position of the task in `tasks` (conditions are lists, so the field travels in place). The reported handle is consumed; the remaining handles stay valid and collectible.

Examples

```
p <- mizu_pool(2L)
slow <- mizu_submit(p, { Sys.sleep(0.5); "slow" })
fast <- mizu_submit(p, "fast")
# completion order, not submission order
mizu_collect_any(list(slow, fast), timeout = 30)
mizu_collect(slow, timeout = 30)
mizu_pool_stop(p)
```

`mizu_current_pool` *The Evaluating Worker's Own Pool Handle*

Description

Inside a pool task, `mizu_current_pool()` returns the pool handle of the worker evaluating the task. Pass it to `mizu_submit()` for nested submission: subtasks push onto the worker's own work-stealing deque, and a worker blocked collecting them helps instead of sleeping. Outside a task, it returns `NULL`.

Usage

```
mizu_current_pool()
```

Details

The pool itself provides the handle around each task evaluation, so unlike a variable binding it cannot be shadowed by a task argument or a local assignment.

Value

A pool handle (class `"mizu_pool"`) inside a pool task; otherwise `NULL`.

Examples

```
p <- mizu_pool()
t <- mizu_submit(p, {
  s <- mizu_submit(mizu_current_pool(), x * 2L, x = x)
  mizu_collect(s, timeout = 30)
}, x = 21L)
mizu_collect(t, timeout = 30)
mizu_pool_stop(p)
```

`mizu_default_pool` *The Process-Wide Default Pool*

Description

`mizu_default_pool()` returns the pool registered as this process's default, or `NULL` when none is set. `mizu_set_default_pool()` sets the default (or clears it with `NULL`) and invisibly returns the previous one, so callers can save and restore it. `mizu_with_pool()` evaluates `expr` with `pool` as the default and restores the previous default on exit, including on error. `mizu_local_pool()` sets the default until the calling frame exits (the withr `local_*` pattern).

Usage

```
mizu_default_pool()

mizu_set_default_pool(pool)

mizu_with_pool(pool, expr)

mizu_local_pool(pool, frame = parent.frame())
```

Arguments

`pool`	a pool handle from `mizu_pool()` or `mizu_pool_attach()`, or `NULL` to clear the default.
`expr`	an expression to evaluate with `pool` as the default.
`frame`	the frame at whose exit the previous default is restored; defaults to the caller of `mizu_local_pool()`.

Details

The default keeps the pool alive even after its variable is removed, until the default is cleared or replaced. Handles from `mizu_pool_attach()` are valid defaults; ownership and teardown stay with the pool's creator.

Setting a default checks only the type. A stopped pool is accepted (liveness is transient, so a probe would prove nothing about use time) and fails later with the usual stopped-pool errors. The default is process-global state, and a child process created by `fork()` reads it as unset.

Value

`mizu_default_pool()`: the default pool handle, or `NULL`. `mizu_set_default_pool()` and `mizu_local_pool()`: the previous default (a pool handle or `NULL`), invisibly. `mizu_with_pool()`: the value of `expr`.

Package authors

A function that takes an optional pool resolves it in this order: an explicit `pool` argument, then `mizu_current_pool()` inside a task (the evaluating worker's own pool, for nested submission), then `mizu_default_pool()`, then the caller's own fallback, such as sequential evaluation or an error. `mizu_current_pool()` is set by the pool around each task evaluation and cannot be shadowed. The default pool is user-set state and never overrides it.

Examples

```
p <- mizu_pool()
old <- mizu_set_default_pool(p)

# the package-author resolution idiom
f <- function(x, pool = mizu_default_pool()) {
  if (is.null(pool)) {
    stop("no pool: pass one, or set a default", call. = FALSE)
  }
  mizu_collect(mizu_submit(pool, x + 1L, x = x))
}
f(1L)

# scoped use: the previous default returns on exit
mizu_with_pool(NULL, mizu_default_pool())
mizu_default_pool()

mizu_set_default_pool(old)
mizu_pool_stop(p)
```

`mizu_error` *Error Conditions*

Description

Terminal failures that a caller can act on programmatically are raised as classed conditions. Each inherits `"mizu_error"` (alongside `"error"` and `"condition"`), with a subclass that names the failure. Catch them with `tryCatch(..., mizu_error_worker_died = ...)` or test with `inherits()` instead of matching message text. Messages are not API and can be reworded. The class vectors and fields below are API.

Details

The subclasses, where they are raised, and the structured fields they carry as condition elements:

- `mizu_error_submit_timeout` — `mizu_submit()` on `.timeout` expiry with the injection ring of the submitter still full.
- `mizu_error_slots_exhausted` — `mizu_submit()` and `mizu_map()` when every result slot in the subrange of the submitter is already outstanding. Collect or cancel before resubmitting.
- `mizu_error_stopped` — `mizu_submit()`, `mizu_map()` and `mizu_pool_attach()` against a pool that was stopped or whose owner process died.
- `mizu_error_cancelled` — `mizu_collect()` on a task that was cancelled or whose pool was stopped.
- `mizu_error_worker_died` — `mizu_collect()` on a task whose executing worker died. Fields `slot` (worker registry slot, 0-based as in `mizu_pool_dump()`) and `pid`: the claimant record of the result slot, read at collect time. This is informational, can race with slot reuse as `mizu_pool_dump()` does, and is `NA` where no claim was recorded. `mizu_map()` signals this class with the lost elements as an additional `lost` field: a two-column matrix of inclusive `lo, hi` ranges, runner-granular and conservative (see the Errors section of `mizu_map()`). pymizu's `WorkerDiedError.lost` carries the same set as a list of 0-based half-open `[lo, hi)` spans.
- `mizu_error_task_lost` — `mizu_collect()` on a task whose payload region vanished before any worker could read it (its submitter died or released the pool with the task still queued). Unlike `mizu_error_worker_died`, no worker death is involved, and the task never started executing anywhere.
- `mizu_error_startup` — `mizu_channel()`, `mizu_pool()` and `mizu_spawn_workers()` when a child process fails to attach within `startup_timeout`.
- `mizu_error_shm` — shared-memory region create or open failure anywhere in the API. Field `bytes`: the requested size of a region that was not created, `NA` when a region was not opened.
- `mizu_error_python_payload` — `mizu_recv()` or `mizu_recv_batch()` on a channel message written in a language-private stream (a pymizu codec or pickle payload) that this reader cannot interpret. The portable interchange subset crosses; the declined message is consumed, so the channel keeps flowing.
- `mizu_error_not_portable` — `mizu_send()` on a channel whose peer is a foreign-language process, of a value outside the portable interchange subset (a named atomic vector, an environment, a closure, an S4 object, an ordered factor, a data frame subclass, and the like). Fields `path` (where in the value the walk declined), `reason`, and `remedy` (a one-line rewrite where one exists, else `character(0)`).
- `mizu_error_remote` — never raised by mizu itself: the value a channel receive returns for a remote error stream, most commonly a peer process whose expression errored (the peer shims send one before exit, on every channel). Fields `remote_type` (the most-specific class of the original error), `message`, `detail` (the call or traceback text), and `index` (the 1-based element index) when the remote error carries one. Test with `mizu_is_remote_error()`; signal it with `stop(x)`.

Errors of misuse (unnamed task arguments, out-of-range slots, operations on a closed handle) stay plain errors: the classed hierarchy covers the outcomes that a running system produces, not programming mistakes.

Raised `mizu_error` conditions are distinct from sentinels (class `mizu_sentinel`, returned by `mizu_send()`, `mizu_recv()` and `mizu_collect()`). A sentinel is an ordinary return value that marks a terminal state, not a signalled condition. See `mizu_is_sentinel()`.

Which of the two you get depends on the shape of the call. The functions that move payloads and wait with a bound (`mizu_send()`, `mizu_recv()`, `mizu_collect()`, `mizu_map()`) return sentinels for transport states: nothing yet (`mizu_timeout`), no room now (`mizu_full`), and stream over (`mizu_closed`, `mizu_peer_gone`). Their caller is a loop, and these are its normal outcomes. Conditions are raised where a request has failed for good. Constructors and `mizu_submit()`, whose return value is a handle the next line uses, raise on every failure. `mizu_collect()` raises when the value can never arrive: the task's own error re-signalled, `mizu_error_cancelled`, or `mizu_error_worker_died`. A sentinel invites the next iteration of the loop, and a condition means stop and deal with it.

Task error transport

A task's own error, re-signalled by `mizu_collect()` (and `mizu_collect_any()` / `mizu_collect_all()`), is a transport condition built on the worker at publish time. The caught condition itself never crosses, so a condition that cannot be serialized can never kill the worker. The transport condition carries the original classes, the raw `message` field (custom `conditionMessage()` methods are bypassed; the message is truncated past its share of the result slot's inline budget), `call`, and every named field the compact payload codec can carry within that budget, in the priority order message, `call`, fields, then `dropped_fields`. Fields the codec cannot carry (environments, closures, S4 objects, ALTREP vectors, external pointers) and fields past the remaining budget are dropped and named in a `dropped_fields` field, absent when nothing was dropped. Only the named elements of a condition are considered: an unnamed element cannot be named in `dropped_fields` and is dropped silently.

`mizu_is_remote_error` *Remote Errors*

Description

A channel receive returns a remote error as a value, not a raised condition: an error in the peer is data until your code decides otherwise. The value is a `mizu_error_remote` condition (see mizu_error). It is an ordinary condition object, so it prints, and `conditionMessage()` leads with `remote_type: message`.

Usage

```
mizu_is_remote_error(x)
```

Arguments

`x`	any R object.

Details

mizu_is_remote_error() tests the class. Signal it with stop(x), and handlers dispatch on its classes (mizu_error_remote, then mizu_error, error, condition).

Value

TRUE or FALSE.

Examples

```
ch <- mizu_channel(quote(stop("boom")))
x <- mizu_recv(ch, timeout = 5)
mizu_is_remote_error(x)
conditionMessage(x)
mizu_close(ch)
```

---

mizu_is_sentinel *Test for a mizu Sentinel*

Description

Tests identity against the four sentinel values (mizu_full, mizu_timeout, mizu_closed, mizu_peer_gone) that mizu's functions return to signal terminal states. inherits(x, "mizu_sentinel") tests the class alone, which any payload can carry, and even a genuine sentinel forwarded over a channel arrives as an ordinary copy. mizu_is_sentinel() returns TRUE only for the sentinel objects that mizu's own calls returned in this process, so code that relays untrusted values can tell its terminal states from look-alike payloads.

Usage

```
mizu_is_sentinel(x)
```

Arguments

x	any R object.

Details

Sentinels are ordinary values, not R conditions: nothing is signalled, and condition handlers never see them.

Value

TRUE or FALSE.

Examples

```
mizu_is_sentinel(42)
# class alone does not make a sentinel:
mizu_is_sentinel(structure("x", class = c("mizu_timeout", "mizu_sentinel")))
```

`mizu_launcher` *Default Child Process Launcher*

Description

Returns the launcher that `mizu_channel()`, `mizu_pool()` and `mizu_spawn_workers()` use unless given a custom one. It spawns a detached child R process through a static `Rscript` runner, with the entry expression and the `.libPaths()` of the host hex-encoded in argv.

Usage

```
mizu_launcher(stdout = "", stderr = "")
```

Arguments

`stdout`, `stderr`	forwarded to `system2()`. The default `""` sends the child output (for a channel peer, including its error epilogue) to the console of the host, `FALSE` discards it, and a file name collects it in that file. On Windows, the child holds a redirection file exclusively for its lifetime, so other processes cannot read it while the child runs, and a second spawn with the same file fails to launch (in a pool, as a startup timeout). A custom launcher that sets the file name from `slot` gives one log file per worker (see examples).

Value

A `function(token, slot)`. `mizu_pool()` and `mizu_spawn_workers()` call it with both arguments to spawn the worker for `slot`. `mizu_channel()` calls it with `token` alone to spawn the peer.

Containers

`system2(wait = FALSE)` detaches background children, so workers and peers are adopted by PID 1 of the process namespace and reaped by that init when they exit. In a container whose PID 1 does not reap (a plain `docker run` without `--init`), exited children accumulate as zombie PID-table entries. This is harmless to mizu itself, which detects death through the liveness lock rather than the PID, but PID-probe supervision misreads zombies as alive, and `mizu_prune()` cannot reclaim a dead process's regions while its PID stays taken.

Examples

```
# One log file per worker on Windows.
launcher <- function(token, slot) {
  f <- if (missing(slot)) "peer.log" else sprintf("worker-%d.log", slot)
  mizu_launcher(stderr = f)(token, slot)
}
```

`mizu_map` *Parallel Map Over a Pool*

Description

Maps `.f` over the elements of `.x` on a pool and returns a map handle immediately, just as `mizu_submit()` returns a task. `mizu_collect()` on the handle waits for completion and returns the results in input order: a list by default, or an atomic vector (or matrix) in the style of `vapply()` when `.template` is given. `mizu_cancel()` stops it. `as.promise()` (or `then()` directly) collects it through the event loop instead; see `mizu_collect_all_async()`.

Usage

```
mizu_map(
  .pool,
  .x,
  .f,
  ...,
  .template = NULL,
  .n_chunks = NULL,
  .seed = NULL,
  .timeout = Inf,
  .stream = FALSE,
  .collect = "copy"
)
```

Arguments

`.pool`	a pool handle from `mizu_pool()` or `mizu_pool_attach()`; inside a task, the evaluating worker's own handle from `mizu_current_pool()`.
`.x`	a vector (atomic or list) to map over. Anything else is coerced with `as.list()`, as `lapply()` does.
`.f`	a function (or, as `match.fun()` accepts, its name) applied as `.f(.x[[i]], ...)`. Serialized once with its enclosing environment. Keep that environment small, as with any cross-process map. A `mizu_call()` specification maps over a pool of any worker language; see the Cross-language maps section.
`...`	further constant arguments to `.f`, staged once. As in `mizu_submit()`, the formals ahead of `...` are dot-prefixed, so an argument name passed here never collides with them.

`.template` NULL for a list result, or a vapply()-style FUN.VALUE: an atomic vector template that each result must match.

`.n_chunks` the morsel count of the map (its scheduling granularity), or NULL for the default. See the Granularity section.

`.seed` NULL (default: no RNG guarantees, no cost), or a numeric vector of length 1 or 2 that derives reproducible per-element RNG streams: seed, or c(seed, offset) to shift every element's stream by offset positions. See the Reproducible RNG section.

`.timeout` seconds after which submission gives up, cancels the tasks already in, and returns the mizu_timeout sentinel (class c("mizu_timeout", "mizu_sentinel")). Inf (the default) waits indefinitely. Submission blocks only while an injection ring is full; the collection deadline is mizu_collect()'s timeout.

`.stream` TRUE streams slices of .x to workers under a sliding window instead of staging .x into shared memory wholesale, bounding shared-memory use by the window times the slice size. See the Streaming maps section. A streaming map handle collects with mizu_collect() only (it cannot be promise-armed).

`.collect` "copy" (the default) gathers template results into an owning R vector at mizu_collect(). "view" instead returns an ALTREP view over the map's shared output area, with no gather copy, for pipelines that reduce it immediately. Requires an atomic .template. The view is copy-on-write: the first write materializes a private copy. It keeps the map region alive until released, and re-sending it through a channel or pool crosses as a full copy, not by reference. The choice is stored on the handle.

Details

Unlike a loop of per-element mizu_submit() calls, one mizu_map() call serializes .f, the constant arguments in ..., and .x once. It submits one *runner* task per live worker, and each worker sets up that map context at most once. The remaining per-element cost is one R function call, as in lapply(). Runners schedule themselves: they claim contiguous batches of elements from a shared cursor in the map region and size each batch adaptively toward a fixed time target. A trivial .f runs in large batches with near-zero scheduling overhead, while an expensive or skewed .f settles on fine claims that keep the workers balanced. An atomic, non-ALTREP .x with no attributes beyond names travels as bare bytes: the workers slice their batches directly from shared memory without deserializing .x, and no worker holds more than a batch of it at a time.

Runners are ordinary pool tasks, stolen and balanced like any other work. Worker death is detected and reported (see Errors), and every pool invariant applies unchanged. Between batches, a runner also answers a pool-wide doorbell. When another submitter's task arrives while every worker is busy inside a map, one runner picks it up at its next batch boundary, so pickup latency is bounded and independent of length(.x). A map runner picked up this way is not executed on the spot. The helper moves it onto its own deque, where the next free worker steals it and joins that map.

Value

A map handle (class "mizu_map"). mizu_collect() on it returns a list of the results of .f in the order of .x, with names(.x) reapplied. With .template, an atomic vector of type typeof(.template),

or an `m * length(.x)` matrix when `length(.template) > 1`. This is an owning vector, or a copy-on-write view over the shared output area with `.collect = "view"`. On `.timeout` expiry mid-submission, `mizu_map()` itself returns the `mizu_timeout` sentinel, and `mizu_collect()` returns it on its own `timeout` expiry. A length-0 `.x` returns a handle whose collect yields the empty result, as `lapply()` does, and creates no region or tasks.

Granularity

Elements are claimed in *morsels*: contiguous ranges of `max(1, min(n %/% (workers * 256), 256))` elements. The morsel is the granularity floor for cancellation, help, and loss reporting. Morsels go to runners in adaptively sized batches of consecutive morsels. `.n_chunks` overrides the morsel count (giving `min(length(.x), .n_chunks)` morsels). There is no per-morsel shared state, so `.n_chunks = length(x)` costs no extra memory and suits extremely skewed element times. A map submitted while no worker is live queues a single runner in the injection ring and runs when a worker joins. If no worker has ever joined the pool, `mizu_map()` errors instead, as `mizu_submit_call()` does. If the submitter's result slots are all occupied by outstanding tasks, `mizu_map()` errors immediately, before it stages anything.

Templates

`.template` works like `vapply()`'s `FUN.VALUE`: every result must match its type and length exactly, or coerce upward (logical -> integer -> double -> complex, checked on the workers per element). A classed `integer64` template (bit64's layout) is also supported. Results must then be `integer64` of the template's length, since an int64 result never widens to double, and the gathered vector or view keeps the class. Results are written directly into a shared output area and gathered in one copy, so no result is serialized. With `.collect = "view"`, the gather copy is skipped as well, and the result is a copy-on-write view over the shared output area itself, for pipelines that reduce it immediately. A template of length `m > 1` gathers an `m * length(.x)` matrix with the names of the template as row names, as `vapply()` does. Character templates are assembled through the generic result path instead, so their type checks surface at assembly rather than per element on the workers. Large results of a consistent shape belong on the template path. A runner's generic results accumulate on the worker and publish once, so `.template` both caps worker memory and moves the values across processes without serialization. For readers of `mizu_pool_stats()`: a map with large *generic* results publishes them through the ordinary result path, and can add a few result-slot `spills` per call even when the pool's `slot_size` is right for its usual traffic.

Errors, timeout, and cleanup

An error raised by `.f` is signalled again in the caller as the original condition, with a `mizu_map_index` field naming the failing element (the lowest index among the elements that ran). Failure is fail-fast: the erroring runner sets the map's shared cancel word before publishing its error, so every other runner stops within about one batch instead of finishing the remaining elements. If a worker dies mid-map, the map raises `mizu_error_worker_died` (see mizu_error) with the lost elements in a `lost` field, a two-column matrix of inclusive `lo, hi` ranges. Loss is reported conservatively, per runner. A dead runner's results publish only at the end, so everything it completed is reported as lost along with what it was executing, and a lost element is never reported as completed. On `.timeout` expiry, during submit or collect, the outstanding work is cancelled and the `mizu_timeout` sentinel is returned, not raised. Executing runners observe cancellation within about one batch (one element where `.f` is expensive), independent of `length(.x)`. The slot of a published but uncollected result

is released only when the dropped handle's finalizer runs at the next garbage collection, and the map's staging region is deleted then too. A later map absorbs these temporarily occupied slots by using fewer runners.

Reproducible RNG

By default, nothing is guaranteed about random draws inside `.f`. The workers seed lazily and independently, and the fast path pays nothing for the option. `.seed` gives every element a reproducible L'Ecuyer-CMRG stream: element `i` runs under the stream `i` jumps of 2^127 steps from the base state that `set.seed(.seed, "L'Ecuyer-CMRG")` installs. The caller's own `.Random.seed` is not touched, and the RNG state of each worker is saved and restored around its batches. Because the streams are per-element, the results are identical for any `.n_chunks` value, batch sizing, worker count, or steal order. With `c(seed, offset)`, element `i` runs under the stream of element `i + offset` of an unoffset run, so a map split across runs or processes reproduces the streams of one uninterrupted run.

Nested maps

`mizu_map(mizu_current_pool(), ...)` inside a task expression uses the handle of the worker running it. Runner submissions push onto the worker's own deque. The blocked collect executes its own runners while idle peers steal the rest, so nested parallelism has little overhead. The first nested map on a worker claims a submitter slot, so with the default `max_submitters = 8` (one held by the controller) at most 7 workers can nest at the same time. Raise `max_submitters` for wider nested fan-outs.

Cross-language maps

`.f` may be a `mizu_call()` specification instead of a function. This is how you map over a pool whose workers are not R, such as one spawned with `mizu_py_launcher()`. A spec `.f` always stages a shared map region: the descriptor crosses in the interchange format, and each runner task carries a region reference that any worker language reads. The map element fills the spec's first positional argument (name kind) or binds as `x` (source kind), and the spec's own constant arguments travel with it, so `...` must be empty with a spec `.f`. Constants and elements must be portable values (the interchange subset documented in `mizu_send()`), and a non-portable one raises `mizu_error_not_portable` at stage time. `.template` and `.collect` work unchanged, with the results gathered (or view-wrapped) by the submitter's own language, and a per-element error crosses with its element index. `.seed` crosses as the language-neutral `(seed, offset)` pair and each worker language derives its own per-element streams. Batching and steal-order invariance holds within a worker language, but the draws are not identical across languages. A spec map runs on a same-language pool too, subject to the same portability rules.

Streaming maps

With `.stream = TRUE`, the map does not stage the whole of `.x` into shared memory. It streams slices of `.x` to workers as they take work, and the return value is unchanged. Fixed slices of `.x` ride ordinary chunk tasks under a sliding submit/collect window of at most `min(.n_chunks, 2 * live workers, free result slots)` outstanding tasks, so shared-memory use is bounded by the window times the slice size instead of the size of `.x`. With the default chunk count that is roughly `(2 * workers) / .n_chunks` of the serialized `.x`. The chunk count defaults to `min(length(.x), 32 * live workers)`, and `.n_chunks`

overrides it (`.n_chunks = length(.x)` is the mirai-style extreme of one element per task). Everything else (result order, `.template` and `.collect`, `.seed` invariance, the error behavior) is the same as for the non-streaming map. Fail-fast latency grows from about one adaptive morsel batch to about one chunk, because the outstanding work at any time is bounded by the window, so the bound moves with `.n_chunks`. Fixed chunks have two costs: the adaptive batch sizing of the morsel scheduler is lost (raise `.n_chunks` to mitigate skew), and slices cross serialized, so each chunk pays an ordinary submit's serialization instead of the non-streaming map's serialization-free slicing. A streaming map with R workers shares one description of the map with every worker, and each worker reads that description at most once. A `mizu_call()` spec as `.f` streams too, over workers of any language. Each chunk in the window then stages a small shared region of its own (the slice plus the task that runs on it, one runner task per chunk), removed as soon as the chunk completes. Shared-memory use stays bounded by the window plus a small per-worker cache (up to 8 recent slices per worker) rather than by the size of `.x`. Each chunk also writes its results separately, so they cannot combine into one view: `.collect = "view"` errors on a streaming spec map, as does preparing one with `mizu_map_prepare()`.

Very large x

The serialized runner wrapper needs a little over 200 bytes of entry inline budget, so pools created with `slot_size = 256L` (224-byte budget) fit it. The exception is `.seed`: its 6-word RNG state pushes the wrapper to about 270 bytes. Seeded maps on such pools work but spill a region per runner, so keep the default `slot_size` on pools meant for seeded maps. For a very large `.x`, two paths avoid materializing it on every worker. If the shared-memory use is acceptable, share it first: a zero-copy view received from a channel or a pool result, or a `mori::share()`d vector, reduces to its 30-byte identifier inside the staged descriptor, and workers read elements directly off the shared pages with OS demand paging, so no worker copies any part of `.x` (`mizu_map()` itself never calls mori). If `.x` must not be staged wholesale at all, `.stream = TRUE` keeps only a window of slices in shared memory; see the Streaming maps section.

As in `lapply()`, `.x` is indexed with `[[` on the workers after an `as.list()` coercion of anything that is not a plain vector, so a data.frame maps over its columns and a factor over its elements.

Examples

```
p <- mizu_pool()
m <- mizu_map(p, 1:10, function(i) i * 2L)
mizu_collect(m)

m <- mizu_map(p, rnorm(1e5), abs, .template = numeric(1))
v <- mizu_collect(m)

m <- mizu_map(p, 1:4, function(i) rnorm(2), .seed = 42L)
mizu_collect(m)
mizu_pool_stop(p)
```

`mizu_map_prepare` *Prepared Maps: Stage Once, Run Many*

Description

`mizu_map_prepare()` stages a map (`.f`, the constant arguments in `...`, and `.x`), serialized once into a shared map region, without running it, and returns a prepared-map handle. Each `mizu_map_run()` then costs only task submission and collection, with no serialization and no region create. Because the region (and its name) stays alive across runs, workers that ran a previous run reuse their cached map context instead of re-attaching. Repeated stochastic simulation is the main use: `mizu_map_run(pm, .seed = i)` varies the RNG streams per run at no extra cost, because the seed state rides the runner payloads rather than the region.

Usage

```
mizu_map_prepare(
  .pool,
  .x,
  .f,
  ...,
  .template = NULL,
  .n_chunks = NULL,
  .stream = FALSE
)

mizu_map_run(pm, .x = NULL, .seed = NULL, .timeout = Inf, .collect = "copy")
```

Arguments

`.pool`	a pool handle from `mizu_pool()` or `mizu_pool_attach()`; inside a task, the evaluating worker's own handle from `mizu_current_pool()`.
`.x`	a vector (atomic or list) to map over. Anything else is coerced with `as.list()`, as `lapply()` does.
`.f`	a function (or, as `match.fun()` accepts, its name) applied as `.f(.x[[i]], ...)`. Serialized once with its enclosing environment. Keep that environment small, as with any cross-process map. A `mizu_call()` specification maps over a pool of any worker language; see the Cross-language maps section.
`...`	further constant arguments to `.f`, staged once. As in `mizu_submit()`, the formals ahead of `...` are dot-prefixed, so an argument name passed here never collides with them.
`.template`	`NULL` for a list result, or a `vapply()`-style `FUN.VALUE`: an atomic vector template that each result must match.
`.n_chunks`	the morsel count of the map (its scheduling granularity), or `NULL` for the default. See the Granularity section.

.stream	TRUE streams slices of .x to workers under a sliding window instead of staging .x into shared memory wholesale, bounding shared-memory use by the window times the slice size. See the Streaming maps section. A streaming map handle collects with mizu_collect() only (it cannot be promise-armed).
pm	a prepared-map handle from mizu_map_prepare().
.seed	NULL (default: no RNG guarantees, no cost), or a numeric vector of length 1 or 2 that derives reproducible per-element RNG streams: seed, or c(seed, offset) to shift every element's stream by offset positions. See the Reproducible RNG section.
.timeout	seconds after which submission gives up, cancels the tasks already in, and returns the mizu_timeout sentinel (class c("mizu_timeout", "mizu_sentinel")). Inf (the default) waits indefinitely. Submission blocks only while an injection ring is full; the collection deadline is mizu_collect()'s timeout.
.collect	"copy" (the default) gathers template results into an owning R vector at mizu_collect(). "view" instead returns an ALTREP view over the map's shared output area, with no gather copy, for pipelines that reduce it immediately. Requires an atomic .template. The view is copy-on-write: the first write materializes a private copy. It keeps the map region alive until released, and re-sending it through a channel or pool crosses as a full copy, not by reference. The choice is stored on the handle.

Details

Between runs, the region's shared scheduling state resets in O(1). The cursor and cancel word clear, and the run generation embedded in every claim word advances, so a leftover task from a previous run can never interfere with the new run. After an unclean run (a .timeout expiry, an error in .f, a worker death) the handle is marked stale, and the next mizu_map_run() restages into a fresh region automatically (the old one is deleted at garbage collection, once any leftover tasks are done with it). A run collected with .collect = "view" restages likewise: the returned view keeps its region alive, so the next run stages fresh rather than reusing pages a held view still reads. A map small enough to ride entirely inline keeps its staged blob on the handle instead, and runs resubmit it, still skipping the serialization.

The prepared handle keeps the staged .x (for automatic restaging) and the map region alive for its lifetime. Both release at garbage collection when the handle is dropped. The number of chunks is fixed at prepare time. The runner count adapts to the live workers at each run.

Value

mizu_map_prepare(): a prepared-map handle. mizu_map_run(): what mizu_map() returns for the staged map, a map handle whose mizu_collect() returns a list, a templated atomic vector, or the mizu_timeout sentinel.

Replacing x between runs

mizu_map_run(pm, .x = x2) runs over a replacement .x. When both the staged and the replacement .x are bare-byte eligible (atomic, non-ALTREP, no attributes beyond names) with identical type and length, the swap is in place: the new bytes are copied over the .x section of the region. This runs a loop over data of the same shape (optimizer steps, simulation sweeps) at the cost of one

copy, skipping the region create and the re-attach of every worker. Any other change of `.x` (a different shape or type, a list, a map staged inline) restages automatically on the next run. A streaming prepared map keeps `.x` in the submitting process, so a replacement `.x` of any shape re-slices on the next run. Only a length change under `.template` restages, because the output area is sized for the staged length.

Examples

```
p <- mizu_pool()
pm <- mizu_map_prepare(p, 1:1000, function(i, draws) {
  mean(rnorm(draws)) * i
}, draws = 100L)
runs <- lapply(1:50, function(s) {
  m <- mizu_map_run(pm, .seed = s)
  mizu_collect(m)
})
mizu_pool_stop(p)
```

`mizu_pool` *Create a Task Pool and Spawn Its Workers*

Description

Creates a shared-memory task pool and spawns its worker processes. Each submitter feeds the workers through its own queue (an injection ring), and results are published through result slots. A submission is one shared-memory write plus at most one wake-up, with no dispatcher process in between. Each worker owns a work-stealing deque. Idle workers steal from busy ones and drain the injection rings. A fairness tick bounds the latency of external submissions on a saturated pool.

Usage

```
mizu_pool(
  n_workers = 1L,
  ...,
  max_workers = n_workers,
  max_submitters = 8L,
  injection_cap = 1024L,
  per_worker_cap = 1024L,
  result_slots = 4096L,
  slot_size = 512L,
  startup_timeout = 30,
  launcher = mizu_launcher()
)
```

Arguments

n_workers	number of worker processes to spawn, at most max_workers.
...	These dots are for future extensions and must be empty.
max_workers	worker registry capacity (at most 64).
max_submitters	submitter registry capacity (at most 64). Each submitter owns its own injection ring and an equal share of result_slots.
injection_cap	entries per submitter injection ring. A power of two.
per_worker_cap	entries per worker work-stealing deque. A power of two.
result_slots	total result slots, partitioned equally across the submitter slots and rounded up to a multiple of max_submitters. This bounds the outstanding (uncollected) tasks of each submitter.
slot_size	bytes per queue entry and result slot. A power of two between 128 and 2^20. A payload (task or result) that serializes past the inline budget travels in a fresh region per payload. This costs an order of magnitude more latency, reported per submitter as mizu_pool_stats()$submitters$spills. The default 512L keeps typical expression-plus-arguments tasks inline. Pools that move only scalar payloads can drop to 256L.
startup_timeout	seconds to wait for all workers to join. On expiry, mizu destroys the pool and raises mizu_error_startup (see mizu_error).
launcher	a function(token, slot) that arranges for an R process to call mizu:::worker_main(token, slot). The default mizu_launcher() spawns Rscript and propagates the .libPaths() of the host. Its stdout and stderr arguments direct the worker output. A custom launcher must arrange the library paths itself.

Details

The pool exists only as long as the process that created it. Use the returned handle directly with mizu_submit() and mizu_collect(). Other processes join as submitters through mizu_pool_attach(). Dropping the handle (or exiting R) shuts the pool down as mizu_pool_stop() does, but without waiting for the workers.

Payloads interoperate with mori. A mori::share()d object anywhere in a task's arguments or result crosses as a short identifier, and is mapped zero-copy on the other side.

The pool's liveness lock files, which detect the death of a participant, are created in a per-platform directory: /dev/shm on Linux, and the per-user temporary directory on macOS and Windows. The path is recorded in the region, so every participant uses the same files. The environment variable MIZU_LIVENESS_DIR, read in the creating process, overrides the default.

Value

A pool handle (class "mizu_pool") holding submitter slot 0. Handles are process-private and do not survive fork().

Examples

```
p <- mizu_pool()
t <- mizu_submit(p, x + y, x = 1, y = 2)
mizu_collect(t)
mizu_pool_stop(p)
```

`mizu_pool_attach` *Attach to a Pool as a Submitter*

Description

Joins an existing pool from another process, claiming a free submitter slot with its own injection ring and result slots. You pass the join token of the pool to the joining process yourself: it is `mizu_pool_info(p)$token` on the creator.

Usage

```
mizu_pool_attach(token)
```

Arguments

`token`	the join token of the pool. The full region name (`mizu_pool_info(p)$name`) is also accepted.

Value

A pool handle (class `"mizu_pool"`) holding a submitter slot.

Examples

```
p <- mizu_pool()
token <- mizu_pool_info(p)[["token"]]
# pass `token` to the joining process yourself. There, run:
# pa <- mizu_pool_attach(token)
# mizu_collect(mizu_submit(pa, runif(3)))
mizu_pool_stop(p)
```

`mizu_pool_dump` *Dump the Distributed State of a Pool*

Description

A read-only debugging snapshot of the entire pool region, one level deeper than `mizu_pool_info()`. It shows per-slot registry detail and every occupied result slot. State is distributed across processes and execution is non-deterministic, so start here when a pool hangs. The scan takes no locks and can race with in-flight transitions: each field is a single consistent read, but the rows need not be consistent with each other.

Usage

```
mizu_pool_dump(pool)
```

Arguments

`pool`	a pool handle from `mizu_pool()` or `mizu_pool_attach()`; inside a task, the evaluating worker's own handle from `mizu_current_pool()`.

Value

A list with elements `token` and `name` (as in `mizu_pool_info()`), `shutdown`, `workers` (data frame: slot, status, pid, park_state, parked, deque `top` and `bottom`, and the in-flight result slot), `submitters` (data frame: slot, status, pid, result-slot subrange, queued injection entries, ready and full-waiter mask bits), `tasks` (data frame of occupied result slots: slot, status, sequence, executing worker, parked waiter), `help`, and `local`. `help` is the help-wanted doorbell: `TRUE` while injection entries are queued with every worker busy, awaiting pickup by a map runner at its next batch transition or a worker helping out of a nested `mizu_collect()` (the `rehome` event of `mizu_pool_trace()`). `local` is the process-private state of this handle: the occupancy of the producer free list (`fl_entries`, `fl_bytes`), its reuse count (`fl_hits`), the `open_hits` and `open_misses` of the consumer mapping cache, `collect_parks` (how often a collect on this handle parked waiting for a result), and `ledger_entries` (lent-out regions still pinned by consumer views).

Examples

```
p <- mizu_pool()
t <- mizu_submit(p, runif(1))
mizu_pool_dump(p)
mizu_collect(t)
mizu_pool_stop(p)
```

mizu_pool_info *Inspect a Pool*

Description

A read-only snapshot of the pool region: registry states, parked-worker count, queued injection entries, and result-slot occupancy.

Usage

```
mizu_pool_info(pool)
```

Arguments

pool	a pool handle from mizu_pool() or mizu_pool_attach(); inside a task, the evaluating worker's own handle from mizu_current_pool().

Value

A list with elements token (the join identity: pass it to mizu_pool_attach()), name (the platform region name, kept for debugging), role, max_workers, max_submitters, injection_cap, result_slots, slot_size, workers (per-slot states), parked, submitters (per-slot states), injection (entries queued and unclaimed), tasks (result slots by state: pending / running / ok / err / cancel / died / lost), deque (per-worker deque depths), and shutdown.

Examples

```
p <- mizu_pool()
mizu_pool_info(p)
mizu_pool_stop(p)
```

mizu_pool_stats *Cumulative Pool Counters*

Description

Per-worker and per-submitter counters, accumulated since each participant joined. They complement the point-in-time snapshots of mizu_pool_info() and mizu_pool_dump(). Keeping them costs the pool nothing. The submitter counts are derived from the injection ring's own positions, so submission writes nothing extra, and the spill counter moves only on the spill path, where creating the region already costs far more than the counter update. The worker counters are kept in the worker process and mirrored into the region only when a worker parks, leaves, or passes its fairness tick, so under continuous load a worker's row can lag by up to 61 claims. The row is exact whenever that worker is parked or retired, or the pool is idle.

Usage

```
mizu_pool_stats(pool)
```

Arguments

pool	a pool handle from mizu_pool() or mizu_pool_attach(); inside a task, the evaluating worker's own handle from mizu_current_pool().

Value

A list of two data frames. workers: one row per worker slot, with status, pid, tasks (task evaluations run, help-mode and nested inline execution included), steals (entries claimed from the deques of peers), injections (entries claimed from injection rings), parks (kernel parks in the worker loop), helps (claims executed while blocked in a nested collect), and the current deque depth. submitters: one row per submitter slot, with status, pid, injected (entries ever published to its injection ring), claimed (entries the workers took from it), spills (payloads past the inline budget that traveled in their own region: task payloads at submit and result payloads at publish, both attributed to the submitter of the task; a nonzero value means slot_size is undersized for the traffic), spill_reuse (the subset of spills that recycled a retired region from the free list of the producer instead of creating one), and queued (injected - claimed). Steady-state spill traffic approaches spills, so spills - spill_reuse is the region-churn rate. Counters reset when a new joiner reuses a slot.

Examples

```
p <- mizu_pool()
t <- mizu_submit(p, runif(5))
mizu_collect(t)
mizu_pool_stats(p)
mizu_pool_stop(p)
```

---

mizu_pool_stop *Stop a Pool*

Description

Broadcasts shutdown, wakes every sleeping participant, and cancels all pending tasks (blocked collectors raise mizu_error_cancelled). Then waits up to timeout seconds for the workers to exit cleanly, and deletes the shared region and the liveness files. The handle is dead afterwards, and stopping it again does nothing. Only the creating process can stop a pool.

Usage

```
mizu_pool_stop(pool, timeout = 5)
```

Arguments

pool	a pool handle from mizu_pool() or mizu_pool_attach(); inside a task, the evaluating worker's own handle from mizu_current_pool().
timeout	seconds to wait for the clean exit of the workers.

Details

A stop also rejects every promise still pending on the pool (task promises, mizu_collect_all_async() and mizu_collect_any_async() collections, and map promises) with mizu_error_stopped.

Value

Invisibly, TRUE if all workers exited within the timeout. FALSE otherwise, with a warning (the workers still exit on their own).

Examples

```
p <- mizu_pool()
t <- mizu_submit(p, 1 + 1)
mizu_collect(t)
mizu_pool_stop(p)
```

---

mizu_pool_trace *Trace Task Lifecycle Events*

Description

Registers a hook on a pool handle. The hook is called as fn(event, id) at each task lifecycle event that this process observes:

Usage

```
mizu_pool_trace(pool, fn = NULL)
```

Arguments

pool	a pool handle from mizu_pool() or mizu_pool_attach(); inside a task, the evaluating worker's own handle from mizu_current_pool().
fn	a function(event, id), or NULL to remove a registered hook.

Details

- "submit" when a task is committed.
- On worker handles: "start" before the evaluation of a task, and "done" or "error" when its result is published.
- "drop" when a claimed task is discarded (cancelled before or during execution, or its payload's shared region died with its submitter).
- "rehome" when a worker helping with the doorbell claims a map runner. The helper moves it onto its own deque, where idle peers can steal it, instead of executing it nested.

id identifies the task as "<submitter slot>:<counter>". This id is stable across processes, so logs from both sides of a pool can be correlated.

Registration is per-handle and per-process. A submitter that traces its own handle sees only "submit". Execution events happen on the workers. To trace a worker, install the hook from a task on the worker's own handle: mizu_submit(p, mizu_pool_trace(mizu_current_pool(), fn)). The disabled hook costs one pointer check per event site, and the channel functions have no event sites. An error raised by the hook propagates as an internal failure at its site. On a worker, it takes the worker down. This differs from a task's own error, which is published as the error result of that task.

Value

Invisibly, NULL.

Examples

```
p <- mizu_pool()
mizu_pool_trace(p, function(event, id) cat(event, id, "\n"))
t <- mizu_submit(p, 1 + 1)
mizu_collect(t)
mizu_pool_trace(p)
mizu_pool_stop(p)
```

---

mizu_prune *Remove Orphaned Shared Memory Regions*

Description

Removes the mizu shared memory regions that dead processes leave behind. This is needed only in exceptional circumstances. Cleanup after a crash is automatic as long as any participant of the pool or channel survives it, because the survivor detects the death through the liveness lock and reclaims the dead process's regions itself. Orphans arise only when every attached process dies at once (for example, the whole process group is killed), so no survivor remains to run that cleanup. They persist until pruned or until the machine reboots.

Usage

```
mizu_prune()
```

Details

Regions of running processes are never touched, and mori regions are invisible to mizu (and mizu regions to mori), because the two packages use disjoint namespaces. Do not run this while another mizu session on the machine is recovering from a crash: payloads sent by a now-dead peer stay deliverable to its survivor until drained, and this function would reap them.

A crashed process cannot clean up after itself, and a new process that happens to reuse its PID cannot reap its orphans either (it reads its own PID as alive). Run `mizu_prune()` while the PID is free, before reuse.

Value

Invisibly, a character vector of the region names removed, or `NULL` if none were. On platforms whose shared memory namespace cannot be enumerated (Windows, where orphans cannot exist), always `NULL`.

Examples

```
mizu_prune()
```

---

mizu_py_launcher *Python Launcher*

Description

Returns a launcher that spawns Python processes running pymizu, the Python binding of the same shared-memory core: `python -m pymizu.child` for the channel peer of `mizu_channel()`, and `python -m pymizu.worker` for the pool workers of `mizu_pool()` and `mizu_spawn_workers()`. It is the counterpart of pymizu's `r_launcher()`, which spawns R processes from a Python host.

Usage

```
mizu_py_launcher(python = NULL, stdout = "", stderr = "")
```

Arguments

`python`	path to the Python interpreter. The default looks up `python3` on the `PATH`.
`stdout, stderr`	forwarded to `system2()` for the spawned process, as in `mizu_launcher()`.

Details

The first worker to join records the workers' language in the pool, so the launcher carries no language attribute. A pool of Python workers takes `mizu_call()` specifications through `mizu_submit_call()`, and a plain `mizu_submit()` fails immediately with an error naming `mizu_submit_call()`. A launcher that spawns the wrong language fails at join, not at the first task.

The interpreter is probed for pymizu when the launcher is created, so a missing interpreter or package errors here, before any channel or pool is created. The payload rules for a non-R channel peer apply: only vectors and strings cross (see `mizu_send()`).

Value

A `function(token, slot)`. `mizu_channel()` calls it with `token` alone to spawn the peer. `mizu_pool()` and `mizu_spawn_workers()` call it with both arguments to spawn the worker for `slot`.

Examples

```
ch <- mizu_channel(
  r"(
import pymizu
while True:
    x = ch.recv(30)
    if pymizu.is_sentinel(x):
        break
    ch.send(x * 2)
)",
  launcher = mizu_py_launcher()
)
mizu_send(ch, c(1.5, 2.5, 3.5))
mizu_recv(ch, timeout = 5)
mizu_close(ch)

p <- mizu_pool(2, launcher = mizu_py_launcher())
t <- mizu_submit_call(p, mizu_call("numpy.mean", runif(100)))
mizu_collect(t)
mizu_pool_stop(p)
```

---

mizu_recv_async *Receive Channel Messages as a Promise or a Stream*

Description

`mizu_recv_async()` returns a `promises::promise()` immediately, without blocking. The promise resolves with the next received payload, the same R object a blocking `mizu_recv()` would have returned, zero-copy ALTREP views included. Chain it with `promises::then()` or the promise pipe `%...>%`.

Usage

```
mizu_recv_async(ch, timeout = Inf)

mizu_recv_handler(ch, fn, batch = FALSE)

## S3 method for class 'mizu_channel'
as.promise(x)

## S3 method for class 'mizu_channel'
is.promising(x)
```

Arguments

ch	a channel handle from mizu_channel() (or the ch binding inside a peer expression). Arming a spin channel errors: its publish path skips the wake checkpoint by design.
timeout	seconds after which the promise rejects with the mizu_timeout classed error. Inf (the default) waits indefinitely.
fn	a function(x) called with each received message (or, with batch = TRUE, each burst as one list), and finally with a terminal condition. NULL removes the handler.
batch	deliver each drained burst to fn as one list in a single call, instead of one call per message.
x	for the as.promise method: a channel handle, treated as mizu_recv_async(x).

Details

The promise rejects on mizu_closed, mizu_peer_gone, and timeout expiry with classed *error* conditions that carry the same class names as the sentinels, so inherits(x, "mizu_closed") works on either form. Close and peer death wake the event loop directly, so the rejection is prompt rather than waiting for the timeout.

Several outstanding calls on one channel resolve in FIFO call order, one message each. No message is lost: messages already in the ring resolve immediately (still asynchronously, as promises require), and only messages yet to arrive wait on the event.

mizu_recv_handler() is the stream form. It installs fn on the channel, and the event loop delivers each burst of messages to it without allocating a promise per message. With batch = TRUE, each drained burst is delivered as one list in a single fn(list) call, which is the high-throughput form. With batch = FALSE, fn(value) is called once per message. Close and peer death are delivered to fn as the classed error conditions (mizu_closed, mizu_peer_gone), after which the handler is removed. A declined foreign payload (see mizu_send()) is delivered as its mizu_error_python_payload condition and the handler continues. An error raised by fn removes the handler and propagates at the event loop. mizu_recv_handler(ch, NULL) removes the handler.

A channel message belongs to whoever takes it first. A channel can have only one of the two consumer forms at a time: installing a handler while promises are pending, or requesting a promise while a handler is installed, is an error. A blocking mizu_recv() mixed with either consumer form consumes in arrival order, so the blocking call may take a message that a promise or the handler was waiting on. A close or peer death that it observes is observed by the consumer too, because both are channel state, re-read on every receive and never consumed. (A task result instead belongs to exactly one consumer; see mizu_collect_all_async().)

Every main-thread drain is bounded per event-loop turn and re-scheduled, so a saturated ring starves neither Shiny nor later::run_now().

Requires the **later** package, and **promises** for mizu_recv_async().

Value

mizu_recv_async(): a promises::promise() resolving with the next message. mizu_recv_handler(): the channel handle, invisibly.

Examples

```
library(promises)
ch <- mizu_channel(quote(mizu_send(ch, mizu_recv(ch))))
mizu_recv_async(ch) %...>% print()
mizu_send(ch, 42L)
later::run_now()
mizu_close(ch)
```

`mizu_send` *Send and Receive over a Channel*

Description

`mizu_send()` publishes a message to the peer. The message is visible as soon as the call returns, with no separate flush step. `mizu_recv()` returns the next message, and waits up to `timeout` seconds for it.

Usage

```
mizu_send(ch, x)

mizu_recv(ch, timeout = Inf)
```

Arguments

`ch`	a channel handle from `mizu_channel()` (or the `ch` binding inside a peer expression).
`x`	the payload: any R object.
`timeout`	seconds to wait for a message before returning the `mizu_timeout` sentinel. `Inf` (the default) waits indefinitely; `0` does not wait.

Details

Sends never block for ring space. Receives report every terminal state as a class-tagged sentinel, not an error. Test for them with `inherits(x, "mizu_sentinel")`, or with the specific classes:

- `mizu_full` — the ring is full (send). Back off until the peer drains, or drop the message.
- `mizu_timeout` — no message arrived within `timeout` (recv).
- `mizu_closed` — the other side closed the channel. A receive drains all published messages before it reports this.
- `mizu_peer_gone` — the peer died without closing. Death is detected through the liveness lock that the kernel releases, as soon as the OS reports it. A receive drains first here too, since the published messages of a dead peer are complete and valid. Sticky once returned.

NULL is a legal payload. Sentinels are ordinary values identified by their class, never signalled conditions. mizu_is_sentinel() checks identity where payloads are untrusted.

NULL crosses as an immediate, with no serialization and no allocation on receive. A length-1 character vector that fits the inline budget crosses with a single byte copy, encoding mark preserved. Attribute-free non-ALTREP atomic vectors, and integer64 vectors whose only attribute is the class, take a serialization-free path with a byte-identical round trip: inline within the budget, and past it as bare bytes in the arena or a spill region. Other plain values (attributed vectors, strings, lists, calls) cross as a compact binary stream written and read without R's serializer. Anything else is R-serialized. Mori-shared objects reduce to their identifiers through the mori hooks.

A NumPy or Arrow int64 array from a Python peer arrives as an integer64 vector in bit64's layout, constructed without bit64. With bit64 loaded the result is fully functional, and without it the bits round-trip but print as raw doubles. The missing-value sentinel is shared in both directions: a genuine INT64_MIN reads as NA_integer64_ in R, and an NA_integer64_ arrives as INT64_MIN in Python.

From a non-R peer, only vectors and strings are legal payloads. Anything else (a pymizu codec stream or a pickle) is declined, and the receive raises a classed mizu_error_python_payload error (see mizu_error). The declined message is consumed, so the channel keeps flowing.

Large payloads can arrive as zero-copy views over shared memory. A view is a small R object that pins memory the garbage collector cannot see. To return that memory promptly, the receive functions (and worker task boundaries) run a minor garbage collection once 8 MiB of released view memory has accumulated. Re-receiving the same region does not re-trigger it. This is an occasional pause of about 0.2 ms, the largest latency outlier in the system. Set options(mizu.gc_boundary) (in bytes; 0 disables it) before loading the package to tune the threshold.

Value

mizu_send() returns TRUE (invisibly) on success, or a sentinel otherwise. mizu_recv() returns the received payload or a sentinel.

Examples

```
ch <- mizu_channel(quote(mizu_send(ch, mizu_recv(ch))))
mizu_send(ch, list(1, "a"))
mizu_recv(ch, timeout = 5)
mizu_close(ch)
```

---

mizu_send_batch *Batched Send and Receive*

Description

At high message rates the cost of each R call matters. mizu_send_batch() moves a list of payloads in a single .Call and publishes them to the peer in one batched tail store. mizu_recv_batch() drains up to n messages in a single park cycle and a single batched head publication.

Usage

```
mizu_send_batch(ch, xs)

mizu_recv_batch(ch, n = 256L, timeout = Inf)
```

Arguments

ch	a channel handle from mizu_channel() (or the ch binding inside a peer expression).
xs	a list of payloads.
n	maximum number of messages to return.
timeout	seconds to wait for a message before returning the mizu_timeout sentinel. Inf (the default) waits indefinitely; 0 does not wait.

Value

mizu_send_batch() returns the number of messages accepted. This is less than length(xs) when the ring filled or the channel closed midway. Send the next element with mizu_send() to learn which. mizu_recv_batch() waits for the first message like mizu_recv() and returns its sentinels on timeout, close, or peer death. It then returns a list of 1 to n already-published messages without waiting further. A batch that reaches a message it cannot read (a foreign Python payload) returns what it read before it, and the failure surfaces on the next receive, which raises as mizu_recv() does. The batch itself raises when the first message is the one declined.

Examples

```
ch <- mizu_channel(quote(mizu_send_batch(ch, mizu_recv_batch(ch, 3L))))
mizu_send_batch(ch, list(1, 2, 3))
mizu_recv_batch(ch, 3L, timeout = 5)
mizu_close(ch)
```

---

mizu_spawn_workers *Grow or Shrink the Worker Set of a Pool*

Description

mizu_spawn_workers() spawns additional workers into free registry slots, up to the max_workers of the pool, and waits for them to join. mizu_retire_worker() asks one worker to exit cleanly. The request is non-blocking and never interrupts a task. The worker observes it between tasks and releases its slot. The remaining workers take over any work still queued on its deque. A retired worker's process can linger briefly to keep alive the results it produced that are not yet collected.

Usage

```
mizu_spawn_workers(
  pool,
  n = 1L,
  ...,
  startup_timeout = 30,
  launcher = mizu_launcher()
)

mizu_retire_worker(pool, slot)
```

Arguments

pool	a pool handle from mizu_pool() or mizu_pool_attach(); inside a task, the evaluating worker's own handle from mizu_current_pool().
n	number of workers to spawn.
...	These dots are for future extensions and must be empty.
startup_timeout	seconds to wait for all workers to join. On expiry, mizu destroys the pool and raises mizu_error_startup (see mizu_error).
launcher	a function(token, slot) that arranges for an R process to call mizu:::worker_main(token, slot). The default mizu_launcher() spawns Rscript and propagates the .libPaths() of the host. Its stdout and stderr arguments direct the worker output. A custom launcher must arrange the library paths itself.
slot	the slot index of the worker (0-based, as reported by mizu_pool_dump()).

Details

Slots free up when workers retire, exit at shutdown, or die and are reaped, so a pool can cycle workers within its registry capacity for its whole lifetime. Only the creating process can resize a pool.

Value

mizu_spawn_workers() invisibly returns the slot indices spawned into. mizu_retire_worker() invisibly returns NULL.

Examples

```
p <- mizu_pool(n_workers = 1L, max_workers = 2L)
mizu_retire_worker(p, 0L)
mizu_spawn_workers(p)
mizu_pool_stop(p)
```

`mizu_submit` *Submit a Task and Collect Its Result*

Description

`mizu_submit()` captures `.expr` unevaluated, serializes it with its named arguments into the injection ring of the submitter, and returns a task handle immediately. Payloads past the inline budget travel in a fresh region. A worker evaluates the expression in a fresh environment that contains the arguments as bindings. `mizu_collect()` blocks until the result is published, then returns the value of the task. If the task raised an error, `mizu_collect()` signals that condition again in the collecting process.

Usage

```
mizu_submit(.pool, .expr, ..., .timeout = Inf)

mizu_collect(task, timeout = Inf)
```

Arguments

`.pool`	a pool handle from `mizu_pool()` or `mizu_pool_attach()`; inside a task, the evaluating worker's own handle from `mizu_current_pool()`.
`.expr`	an expression, captured unevaluated. This differs from `mizu_channel()`, which requires its expression pre-quoted. The expression sees only the arguments in `...` and the global environment of the worker. The expression itself must load any packages it needs.
`...`	named values bound in the evaluation environment. The values are serialized. `mori::share()`d objects reduce to identifiers and map zero-copy on the worker. The formals ahead of `...` are dot-prefixed, so a name you pass through `...` can never collide with them.
`.timeout`	seconds to wait for injection-ring space before the call raises `mizu_error_submit_timeout`. Submission blocks only when the ring is full (back-pressure) and returns immediately otherwise. `Inf` (the default) waits indefinitely; `0` does not wait.
`task`	a task handle from `mizu_submit()`, or a map handle from `mizu_map()` / `mizu_map_run()`. A map handle collects the whole map: the gathered list (or templated vector), an element error re-signalled with its `mizu_map_index`, or a worker death as `mizu_error_worker_died` with the lost element ranges, following the rules in `mizu_map()`'s Errors section. A task or map handle with a pending promise (or enrolled in a pending `mizu_collect_all_async()` / `mizu_collect_any_async()` collection) refuses a blocking collect: one consumer per result.
`timeout`	seconds to wait for the result before the call returns the `mizu_timeout` sentinel. `Inf` (the default) waits indefinitely; `0` does not wait.

Details

Submission blocks only when the submitter's own injection ring is full, so one waiting submitter never slows another. On `.timeout` expiry, submission raises `mizu_error_submit_timeout` instead of stalling. If no result arrives within `timeout`, collection returns the `mizu_timeout` sentinel (class `c("mizu_timeout", "mizu_sentinel")`). Collecting a task whose handle was cancelled (or whose pool was stopped) raises `mizu_error_cancelled`. Collecting a task whose executing worker died raises `mizu_error_worker_died`, carrying the slot and pid of the worker (see mizu_error). Worker death is detected as soon as the operating system reports it. Each worker holds a lock that the kernel releases when its process exits, so there are no heartbeats and no polling. Only the tasks the dead worker had claimed fail, and the surviving workers take over the work still queued on its deque.

Value

`mizu_submit()` returns a task handle (class `"mizu_task"`). `mizu_collect()` returns the value of the task, or the `mizu_timeout` sentinel.

Outcomes

A timeout on collect is a normal outcome and is returned. Every exceptional outcome is raised as a classed condition (see mizu_error):

outcome	surfaced as	class
result published	the value, returned	—
no result within `timeout`	sentinel, returned	`c("mizu_timeout", "mizu_sentinel")`
ring full past `.timeout`	raised by `mizu_submit()`	`mizu_error_submit_timeout`
result slots exhausted	raised by `mizu_submit()`	`mizu_error_slots_exhausted`
pool stopped, or owner died	raised by `mizu_submit()`	`mizu_error_stopped`
task raised an error	re-signalled on collect	the condition classes of the task itself
cancelled, or pool stopped	raised on collect	`mizu_error_cancelled`
executing worker died	raised on collect	`mizu_error_worker_died`
payload region vanished before execution	raised on collect	`mizu_error_task_lost`

A re-signalled task error is a transport condition: it carries the original classes, `message`, `call`, and every named field the payload codec can carry within the result slot's inline budget, with anything it cannot carry dropped and named in a `dropped_fields` field. See the Task error transport section of mizu_error.

Inside a task, `mizu_current_pool()` returns the handle of the worker running it, so a task can submit nested subtasks. `mizu_submit(mizu_current_pool(), ...)` inside a task pushes onto the worker's own work-stealing deque, with no ring and no wait. If the deque is full, the subtask runs inline instead. A worker blocked in `mizu_collect()` on a nested handle helps instead of sleeping: it executes work from its own deque (and steals from peers) until the awaited result is published. Nested fan-outs therefore cost very little and cannot deadlock the pool. A nested submission claims a submitter slot for the worker on first use.

A handle can be collected exactly once: the result slot is released to the pool as the value is returned. If an uncollected handle goes to the garbage collector, a still-queued task is cancelled and a

published result is discarded. To wait on several handles at once, `mizu_collect_any()` reports the first terminal task and `mizu_collect_all()` returns every result in input order.

Examples

```
p <- mizu_pool()
t <- mizu_submit(p, sum(x), x = runif(10))
mizu_collect(t, timeout = 30)
mizu_pool_stop(p)
```

---

mizu_submit_batch *Submit a Batch of Tasks*

Description

`mizu_submit_batch()` submits one task per element of `.exprs` in a single `.Call`: one crossing into shared memory and one wake-up sweep per batch instead of per task. At high task rates the cost of each R call matters, so a burst submitted this way reaches the workers sooner than the same burst looped through `mizu_submit()`. Pair with `mizu_collect_all()` to batch the collection side too.

Usage

```
mizu_submit_batch(.pool, .exprs, ..., .timeout = Inf)
```

Arguments

`.pool`	a pool handle from `mizu_pool()` or `mizu_pool_attach()`; inside a task, the evaluating worker's own handle from `mizu_current_pool()`.
`.exprs`	a list of expressions, one per task. Quote them yourself: elements of a list cannot be captured unevaluated.
`...`	named values bound in the evaluation environment. The values are serialized. `mori::share()`d objects reduce to identifiers and map zero-copy on the worker. The formals ahead of `...` are dot-prefixed, so a name you pass through `...` can never collide with them.
`.timeout`	seconds to wait for injection-ring space before the call raises `mizu_error_submit_timeout`. Submission blocks only when the ring is full (back-pressure) and returns immediately otherwise. `Inf` (the default) waits indefinitely; `0` does not wait.

Details

Each task's wire payload is the same `list(.expr, args)` as `mizu_submit()`'s, with the `...` arguments shared by every task in the batch. Unlike `mizu_submit()`, expressions are not captured: the elements of `.exprs` are pre-quoted (or plain values, which evaluate to themselves).

Submission behaves per task as in `mizu_submit()`, with one difference: if the injection ring fills past `.timeout` mid-batch, the call returns the handles accepted so far instead of raising `mizu_error_submit_timeout`. Fatal outcomes (pool stopped, result slots exhausted) still raise, and tasks already submitted stay valid and collectible.

Value

A list of task handles (class "mizu_task"), one per accepted task. The list is shorter than .exprs when the ring filled past .timeout mid-batch.

Examples

```
p <- mizu_pool()
ts <- mizu_submit_batch(p, list(quote(1 + 1), quote(2 + 2)))
mizu_collect_all(ts, timeout = 30)
mizu_pool_stop(p)
```

---

print.mizu_channel *Print Methods for mizu Objects*

Description

One-line summaries. A channel prints its region name, side, and conversation state: open, closed once either side signalled close, or peer gone as reported by mizu_alive(), probed at print. A pool prints its region name, the role of this handle, live workers out of registry capacity, and pending (uncompleted) tasks. A task handle prints its state, probed without consuming the result. The state is pending, running, ok, err, cancel, died, or lost in the result-slot vocabulary of mizu_pool_info(), collected once the result is taken, or dropped when its pool is gone. A mizu_call() task specification prints its qualified name, or the first line of its source string truncated, and its argument counts. Sentinels print as their class. The handle methods never error and never touch the rings, and a handle whose resources are released (a closed channel, a stopped pool) prints as closed, so auto-printing is always safe.

Usage

```
## S3 method for class 'mizu_channel'
print(x, ...)

## S3 method for class 'mizu_pool'
print(x, ...)

## S3 method for class 'mizu_task'
print(x, ...)

## S3 method for class 'mizu_call'
print(x, ...)

## S3 method for class 'mizu_map'
print(x, ...)

## S3 method for class 'mizu_map_prepared'
print(x, ...)
```

```
## S3 method for class 'mizu_sentinel'
print(x, ...)

## S3 method for class 'mizu_error_remote'
print(x, ...)
```

Arguments

x	the object.
...	ignored.

Value

x, invisibly.

Examples

```
p <- mizu_pool()
p
t <- mizu_submit(p, 1 + 1)
t
mizu_collect(t)
mizu_pool_stop(p)
p

mizu_call("stats::quantile", runif(10))
```

Index

as.promise.mizu_channel
(mizu_recv_async), 37
as.promise.mizu_map
(mizu_collect_all_async), 11
as.promise.mizu_task
(mizu_collect_all_async), 11

inherits(), *16*
is.promising.mizu_channel
(mizu_recv_async), 37
is.promising.mizu_map
(mizu_collect_all_async), 11
is.promising.mizu_task
(mizu_collect_all_async), 11

lapply(), *21–23*, *25, 26*
later::run_now(), *38*

match.fun(), *21*, *26*
mizu_alive, 3
mizu_alive(), *46*
mizu_as_arrow, 3
mizu_call, 4
mizu_call(), *21*, *24–26*, *36*, *46*
mizu_cancel, 6
mizu_cancel(), *12*, *21*
mizu_channel, 7
mizu_channel(), *3*, *9*, *17*, *20*, *36–39*, *41*, *43*
mizu_channel_info, 8
mizu_close, 9
mizu_collect (mizu_submit), 43
mizu_collect(), *6*, *10*, *12, 13*, *17, 18*, *21–23*, *27*, *29*, *31*
mizu_collect_all, 10
mizu_collect_all(), *11–13*, *18*, *45*
mizu_collect_all_async, 11
mizu_collect_all_async(), *21*, *34*, *38*, *43*
mizu_collect_any, 13
mizu_collect_any(), *10–12*, *18*, *45*
mizu_collect_any_async
(mizu_collect_all_async), 11
mizu_collect_any_async(), *34*, *43*
mizu_collect_async
(mizu_collect_all_async), 11
mizu_current_pool, 14
mizu_current_pool(), *16*, *21*, *26*, *31–34*, *42–45*
mizu_default_pool, 15
mizu_error, *6, 7*, 16, *18*, *23*, *29*, *40*, *42*, *44*
mizu_error_cancelled (mizu_error), 16
mizu_error_not_portable (mizu_error), 16
mizu_error_python_payload (mizu_error), 16
mizu_error_remote (mizu_error), 16
mizu_error_shm (mizu_error), 16
mizu_error_slots_exhausted
(mizu_error), 16
mizu_error_startup (mizu_error), 16
mizu_error_stopped (mizu_error), 16
mizu_error_submit_timeout (mizu_error), 16
mizu_error_task_lost (mizu_error), 16
mizu_error_worker_died (mizu_error), 16
mizu_is_remote_error, 18
mizu_is_remote_error(), *5*, *17*
mizu_is_sentinel, 19
mizu_is_sentinel(), *18*, *40*
mizu_launcher, 20
mizu_launcher(), *8*, *29*, *36*, *42*
mizu_local_pool (mizu_default_pool), 15
mizu_map, 21
mizu_map(), *6*, *10*, *12*, *17, 18*, *27*, *43*
mizu_map_prepare, 26
mizu_map_prepare(), *25*, *27*
mizu_map_run (mizu_map_prepare), 26
mizu_map_run(), *6*, *12*, *43*
mizu_pool, 28
mizu_pool(), *5*, *15*, *17*, *20, 21*, *26*, *31–34*, *36,*

37, *42, 43*, *45*
mizu_pool_attach, 30
mizu_pool_attach(), *15*, *17*, *21*, *26*, *29*, *31–34*, *42, 43*, *45*
mizu_pool_dump, 31
mizu_pool_dump(), *17*, *32*, *42*
mizu_pool_info, 32
mizu_pool_info(), *31, 32*, *46*
mizu_pool_stats, 32
mizu_pool_stats(), *23*, *29*
mizu_pool_stop, 33
mizu_pool_stop(), *29*
mizu_pool_trace, 34
mizu_pool_trace(), *31*
mizu_prune, 35
mizu_prune(), *20*
mizu_py_launcher, 36
mizu_py_launcher(), *4*, *8*, *24*
mizu_recv *(*mizu_send*)*, 39
mizu_recv(), *3*, *17, 18*, *37, 38*, *41*
mizu_recv_async, 37
mizu_recv_async(), *12*
mizu_recv_batch *(*mizu_send_batch*)*, 40
mizu_recv_batch(), *17*
mizu_recv_handler *(*mizu_recv_async*)*, 37
mizu_retire_worker *(*mizu_spawn_workers*)*, 41
mizu_send, 39
mizu_send(), *3*, *5*, *17, 18*, *24*, *36*, *38*, *41*
mizu_send_batch, 40
mizu_set_default_pool *(*mizu_default_pool*)*, 15
mizu_spawn_workers, 41
mizu_spawn_workers(), *17*, *20*, *36, 37*
mizu_submit, 43
mizu_submit(), *5*, *8*, *10*, *12–14*, *17, 18*, *21, 22*, *26*, *29*, *36*, *45*
mizu_submit_batch, 45
mizu_submit_call *(*mizu_call*)*, 4
mizu_submit_call(), *4*, *36*
mizu_with_pool *(*mizu_default_pool*)*, 15

print.mizu_call *(*print.mizu_channel*)*, 46
print.mizu_channel, 46
print.mizu_error_remote *(*print.mizu_channel*)*, 46
print.mizu_map *(*print.mizu_channel*)*, 46
print.mizu_map_prepared *(*print.mizu_channel*)*, 46
print.mizu_pool *(*print.mizu_channel*)*, 46
print.mizu_sentinel *(*print.mizu_channel*)*, 46
print.mizu_task *(*print.mizu_channel*)*, 46
promises::promise(), *11, 12*, *37, 38*
promises::then(), *12*, *37*

system2(), *20*, *36*

vapply(), *21–23*, *26*