10 Lightweight Concurrency

CS 6868: Concurrent Programming

KC Sivaramakrishnan

Spring 2026, IIT Madras

Go-like Concurrency Library

- Let’s implement a Go language-like concurrency library
  - goroutines — Lightweight threads aka ***fibers***
  - channels — communication between goroutines
  - select — choose between multiple communication events
- Start with a uniprocessor implementation
  - Make it multicore-capable next
  - See `golike_unicore/`

Go-like concurrency library

Unicore

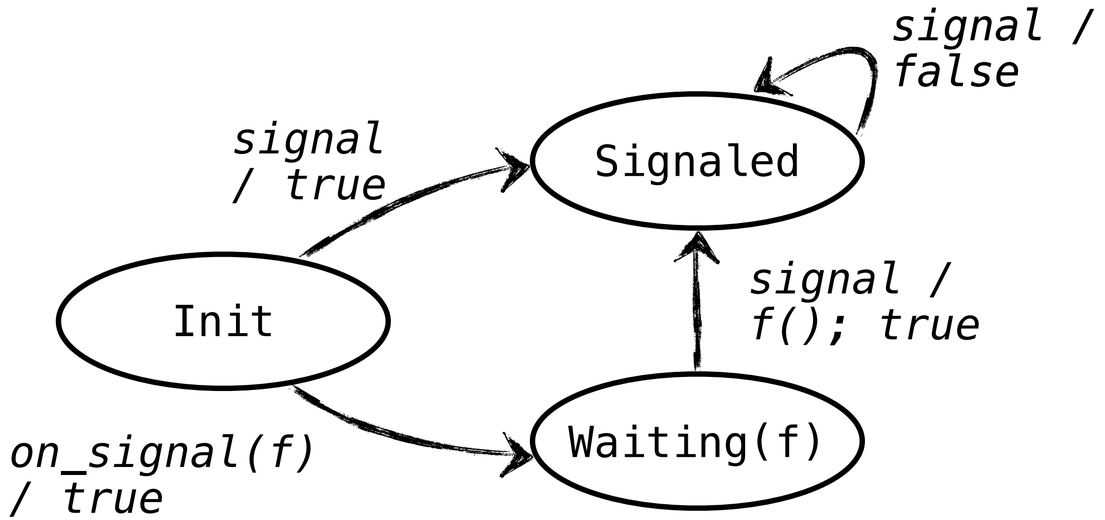
IVar

- Write-once, read-many synchronisation variable
  - Readers block until the value is written
- Coordinate with the scheduler for suspending and resuming tasks
- Use ***Trigger*** to implement IVar

```
let () =
  Printf.printf "\n=== IVar: multiple readers ===\n";
  Sched.run (fun () ->
    let iv = Ivar.create () in
    for i = 1 to 3 do
      Sched.fork (fun () ->
        let v = Ivar.read iv in
        Printf.printf "  Reader %d got: %d\n" i v
      )
    done;
    Printf.printf "  Filling with 7\n";
    Ivar.fill iv 7
  )
(* Output:
   === IVar: multiple readers ===
     Filling with 7
     Reader 3 got: 7
     Reader 2 got: 7
     Reader 1 got: 7
*)
```

Trigger

- A one-shot wakeup with callback attached to it
- Help (lightweight) threads sleep and wake them up on signal
- See
  - trigger.mli
  - trigger.ml



IVar Implementation using Trigger

- Use Trigger as the mechanism to
  - Block when reading an empty Ivar
    - **`perform Trigger.Await`**
  - Broadcast when writing to an empty Ivar with waiting readers
    - Do **`Trigger.signal`**
- See
  - ivar.mli
  - ivar.ml

Handling Trigger.Await

- Who handles `Trigger.Await`?
  - Scheduler
- On await, run the next thread from the scheduler
- On signal, enqueue the thread to the scheduler queue
  - Do this by attaching an ***on_signal callback*** when handling Trigger.Await
- See
  - sched.ml

Scheduler

```
Sched.run (fun () ->
  let iv = Ivar.create () in
  ignore (Ivar.read iv))
```

- What happens when you run this program?
  - Runs to completion
  - Blocks forever

Channels

- Channels are the primary means of communication between goroutines
  - Channels can be buffered (with some capacity c) or unbuffered (capacity 0)
  - Operations are send and receive
  - FIFO ordering of the messages
- Sending (receiving) on a not-full (not-empty) channel immediately succeeds
- Sending (receiving) on a full (empty) channel blocks until state transitions to not-full (not-empty)
- See
  - chan.ml, chan_test.ml

Separation of concerns

- Ivar and Channel are called synchronisation structures
- Synchronisation structures are ***agnostic*** to the scheduler
  - Thanks to `Effect.Await`
- Scheduler and synchronisation structures can be freely changed/combined

Async/await Promises

- We can implement async/await using Ivars and fork.

```
type 'a t = 'a Ivar.t

let async f =
  let p = Ivar.create () in
  Sched.fork (fun () -> Ivar.fill p (f ()));
  p

let await p = Ivar.read p
```

```
Sched.run (fun () ->
   let rec fib n =
     if n <= 1 then n
     else
       let a = Promise.async (fun () -> fib (n - 1)) in
       let b = fib (n - 2) in
       Promise.await a + b
   in
   for i = 0 to 10 do
     Printf.printf "  fib(%d) = %d\n" i (fib i)
   done
 )
(* fib(0) = 0
   fib(1) = 1
   fib(2) = 1
   fib(3) = 2
   fib(4) = 3
   ...   *)
```

No speedup here (yet)!

Go-like concurrency library

Multicore

Plan

- Extend the unicore implementation to multicore
- **Trigger** is made lock-free
- **Scheduler**
  - A pool of domains
  - A shared lock-protected FIFO scheduler queue
  - *Worker* domains
    - Wait until a thread is available
    - Quit when the total thread count goes to 0
- **Ivar** — lock-free and lock-based
- **Async/await promise** remains unchanged!

Parallel Fibonacci (Nested Parallelism)

```
Sched.run ~num_domains (fun () ->
   let rec fib n =
     if n <= 1 then n
     else
       let a = Promise.async (fun () -> fib (n - 1)) in
       let b = fib (n - 2) in
       Promise.await a + b
   in
   for i = 0 to 10 do
     Printf.printf "  fib(%d) = %d\n" i (fib i)
   done
 )
```

- Do we expect speedup with this program?
- Unfortunately, no!
  - Too much parallelisation overhead in spawning the threads

Parallel Fibonacci

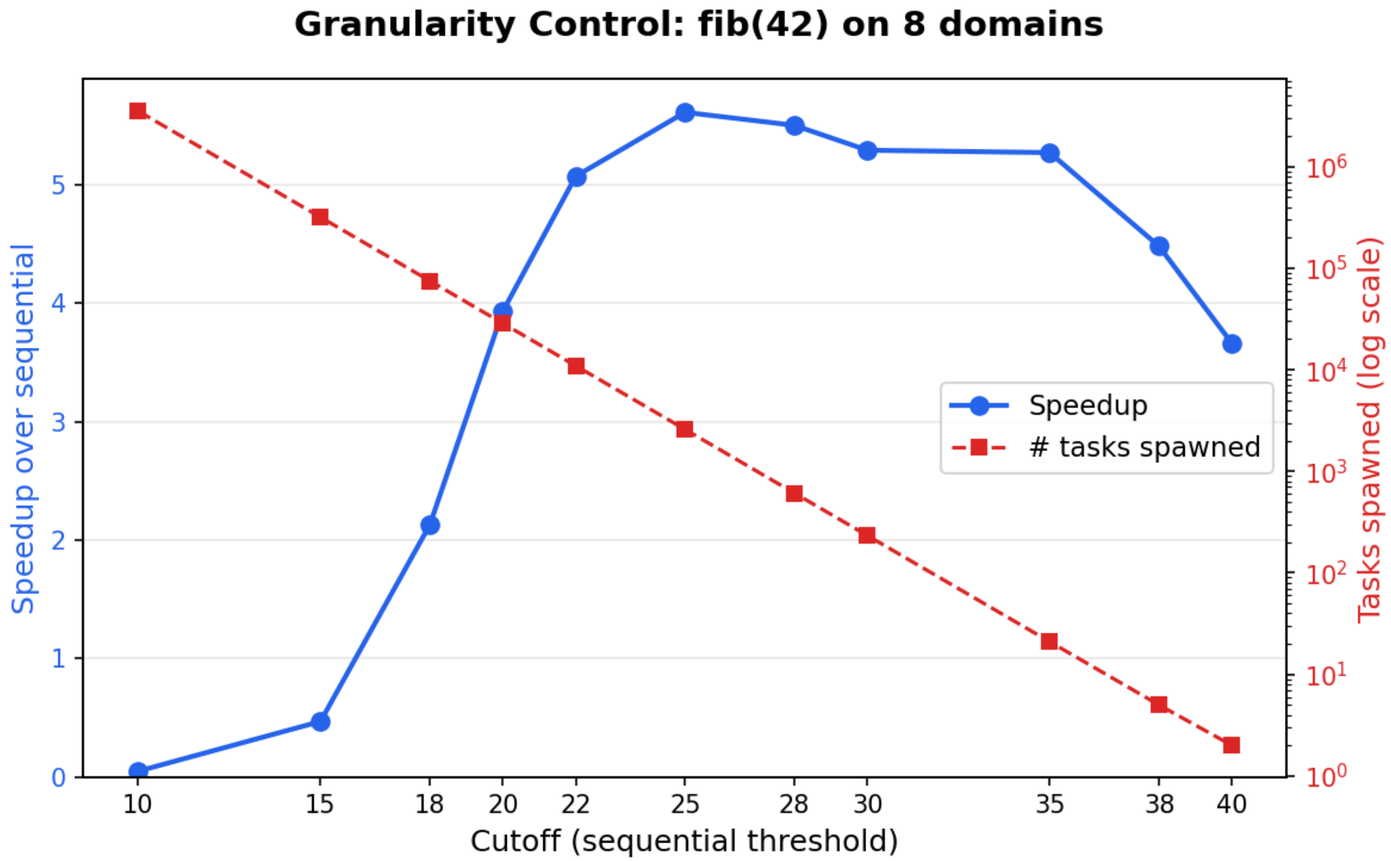
```
let rec fib_seq n =
  if n < 2 then n else fib_seq (n - 1) + fib_seq (n - 2)

let rec fib_par cutoff n =
  if n < cutoff then fib_seq n
  else
    let p = Promise.async (fun () -> fib_par cutoff (n - 1)) in
    let r2 = fib_par cutoff (n - 2) in
    Promise.await p + r2
```

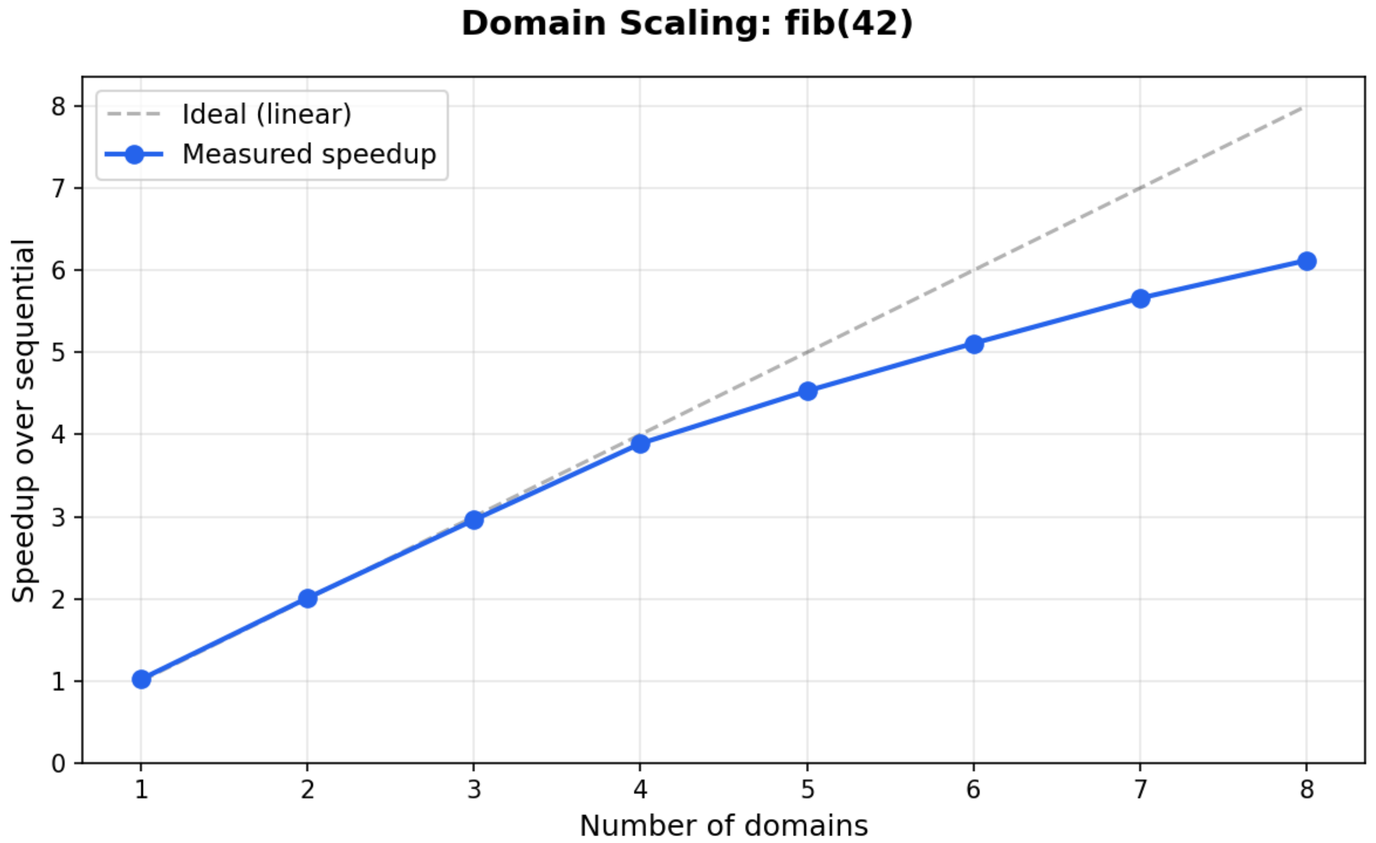
- Introduce a ***sequential cutoff*** for the problem size
  - Run sequentially below the cutoff

Determining cutoff

- Best cutoff depends on the execution environment (machine, architecture, problem)
- Empirical analysis is necessary



Parallel Fibonacci Speedup



- Very respectable!

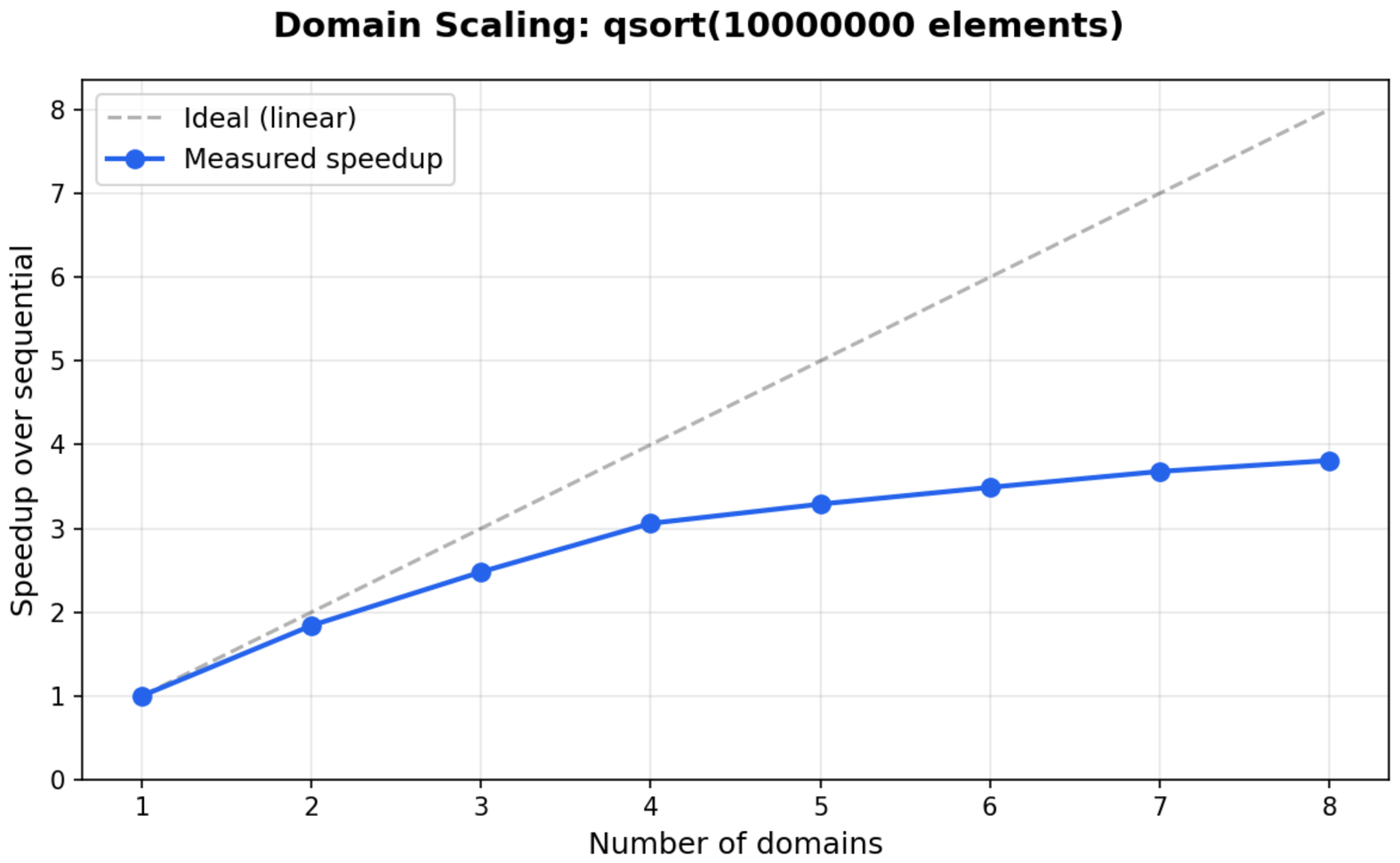
Parallel Quicksort

```
let rec qsort_seq arr lo hi =
  if hi - lo < 16 then insertion_sort arr lo hi
  else begin
    let p = partition arr lo hi in
    qsort_seq arr lo (p - 1);
    qsort_seq arr (p + 1) hi
  end

let rec qsort_par cutoff arr lo hi =
  if hi - lo < cutoff then qsort_seq arr lo hi
  else begin
    let p = partition arr lo hi in
    let promise = Promise.async (fun () -> qsort_par cutoff arr lo (p - 1)) in
    qsort_par cutoff arr (p + 1) hi;
    Promise.await promise
  end
```

- Cutoff is the length of the subarray

Parallel Quicksort



Multicore-safe Channels

- Just put a lock on it!
- See
  - chan.ml
- Unfortunately, the primes sieve example ***blocks*** if we implement them as in unicore
  - Scheduler terminate only when all the fibers have completed
  - In primes sieve unicore, fibers live for ever

Primes Sieve Multicore

```
let generate ch =
  let i = ref 2 in
  while true do
    Chan.send ch !i;
    incr i
  done
in
let filter in_ch out_ch prime =
  while true do
    let n = Chan.recv in_ch in
    if n mod prime <> 0 then
      Chan.send out_ch n
  done
```

Unicore

```
let generate ch =
  for i = 2 to limit do
    Chan.send ch (Some i)
  done;
  Chan.send ch None
in
let filter in_ch out_ch prime =
  let running = ref true in
  while !running do
    match Chan.recv in_ch with
    | None ->
        Chan.send out_ch None;
        running := false
    | Some n ->
        if n mod prime <> 0 then
          Chan.send out_ch (Some n)
  done
```

Multicore

Selective Communication

Limitation with the current model

- Our model so far has
  - Fibers — units of concurrency
  - Channels and Ivars — communication and synchronisation structures
  - Promises — built on top of Ivars
- **Restriction:** One fiber can only perform one communication action at a time
- ***How to wait on multiple things simultaneously***
  - Proceed when one of them is complete

Example — Message Relay

```
(* A chat relay: forward messages from Alice or Bob to the display *)
let relay alice_ch bob_ch display_ch =
  while true do
    let msg = Chan.recv alice_ch in   (* STUCK HERE if Alice is silent *)
    Chan.send display_ch msg;
    let msg = Chan.recv bob_ch in     (* Never reached until Alice speaks *)
    Chan.send display_ch msg
  done
```

```
let relay alice_ch bob_ch display_ch =
  Sched.fork (fun () ->
    while true do Chan.send display_ch (Chan.recv alice_ch) done);
  while true do Chan.send display_ch (Chan.recv bob_ch) done
```

Spawning a fiber per user doesn’t scale and adds complexity

Example — graceful shutdown

```
let generate ch =
  for i = 2 to limit do
    Chan.send ch (Some i)
  done;
  Chan.send ch None
in
let filter in_ch out_ch prime =
  let running = ref true in
  while !running do
    match Chan.recv in_ch with
    | None ->
        Chan.send out_ch None;
        running := false
    | Some n ->
        if n mod prime <> 0 then
          Chan.send out_ch (Some n)
  done
```

- Had to encode shutdown as an optional type in the value pipeline
- Not great since we are mixing data (values for primes) with control (termination)

Example — Load balancing

```
(* Load balancer: send a job to whichever worker is free *)
let dispatch job worker1_ch worker2_ch worker3_ch =
  Chan.send worker1_ch job   (* blocks if worker1 is busy *)
  (* never tries worker2 or worker3! *)
```

What do other systems do?

- UNIX
  - IO multiplexing using select/epoll/poll
  - Works on file descriptor and not channels
- Go select statement

```
select {
  case msg := <-alice:
      handle(msg)
  case msg := <-bob:
      handle(msg)
  case <-timeout:
      log("timed out")
}
```

Selective Communication

- Our implementation is inspired by Concurrent ML, a concurrency library for standard ML (Reppy 1988)
- Events
  - Describe potential synchronisation
  - First-class value

```
type 'a event

val recv_evt  : 'a Chan.t -> 'a event          (* receiving gives 'a *)
val send_evt  : 'a Chan.t -> 'a -> unit event (* sending gives unit *)
val read_evt  : 'a Ivar.t -> 'a event         (* reading gives 'a *)
val wrap     : 'a event -> ('a -> 'b) -> 'b event
val select   : 'a event list -> 'a           (* synchronize *)
```

- If multiple events are ***enabled,*** select chooses non-deterministically (fairness)

Message Relay using Select

```
let relay alice_ch bob_ch display_ch =
  while true do
    let msg = Select.select [
      Chan.recv_evt alice_ch;
      Chan.recv_evt bob_ch;
    ] in
    Chan.send display_ch msg
  done
```

- Choose between receiving between `alice_ch` and `bob_ch`

Graceful shutdown

```
let or_cancel cancel evt = Select.select [
  evt;
  Ivar.read_evt cancel |> Select.wrap (fun () -> raise Cancelled);
]
```

- **or_cancel** is a reusable ***combinator***
- **evt** is the event, **cancel** is an Ivar which is filled to indicate cancellation
- Chooses between the event **evt** and reading on the **cancel** IVar
- Raises the exception **Cancelled** if cancelled

Fibonacci with graceful shutdown

```
Sched.run (fun () ->
  let cancel = Ivar.create () in
  let c = Chan.make 0 in

  let fibonacci out_ch =
    try
      let rec loop x y =
        or_cancel cancel (Chan.send_evt out_ch x);
        loop y (x + y)
      in
      loop 0 1
    with Cancelled -> ()
  in

  Sched.fork (fun () -> fibonacci c);
  for i = 1 to n_fibs do
    let v = Chan.recv c in
    Printf.printf "  %d\n" v
  done;
  Ivar.fill cancel ())
```

Primes with graceful shutdown

```
let cancel = Ivar.create () in

let generate out_ch =
  try
    let i = ref 2 in
    while true do
      or_cancel cancel (Chan.send_evt out_ch !i);
      incr i
    done
  with Cancelled -> ()
in

let filter in_ch out_ch prime =
  try
    while true do
      let n = or_cancel cancel (Chan.recv_evt in_ch) in
      if n mod prime <> 0 then
        or_cancel cancel (Chan.send_evt out_ch n)
    done
  with Cancelled -> ()
```

Implementing Select

- Need a way to be able to inspect channels and Ivars
  - Make offers for performing communication
  - Exactly one of the many options must be satisfied
    - Use Trigger mechanism for this
- Describe an Event type

```
(** An event that, when synchronised on, produces a value of type ['b]. *)
type 'b event = Evt : {
  try_complete : unit -> 'a option;
  (** Non-blocking attempt (called under lock). *)
  offer : 'a option ref -> Trigger.t -> unit;
  (** Enqueue a waiter (called under lock). *)
  mutex : Mutex.t;
  (** The sync object's mutex. *)
  wrap : 'a -> 'b;
  (** Post-processing applied to the raw result. *)
} -> 'b event
```

Channel implementation

- Update to support selection
- Must handle the case when the trigger could have already been signalled
- Must support the ability to
  - poll for operations (try_complete)
  - have the ability to block with the given trigger and result slot (offer)
- See
  - golike_multicore_select/chan.ml

Select Protocol

- Proceeds in Phases
- Mirrors the implementation of select in Go language
- Same between unicore and multicore versions except the absence of locking in unicore version

Select Protocol

- **Phase 1** — Get all the locks associated with the events
  - Sort the locks and deduplicate to canonical order to avoid deadlocks
- **Phase 2** — Scan for immediately ready case
  - Non-deterministically scan for fairness
  - If found, release locks, return value
- **Phase 3** — Create a shared trigger and offer for all cases
- **Phase 4** — Unlock all the locks
- **Phase 5** — Await on Trigger
- **Phase 6** — Relock all the locks, find the winner, release the locks, perform the wrapped computation

golike_multicore_select/select.ml

Takeaway

- Effect handlers are powerful enough to implement Go-like concurrency
  - Goroutines — lightweight threads/fibers
  - Channels
  - Selective communication
  - Multicore-capable scheduler
- Further reading
  - Communicating Sequential Processes (Tony Hoare) — formal foundations
  - Concurrent ML — powerful practical implementation
  - Actor Model — no shared state

Asynchronous IO

Fibers vs The Outside World

- All communication only between fibers
- Real programs communicate to the outside world
  - Timers
  - Network IO
  - File IO
- How do we talk to the outside world?

System Calls

- User-space programs cannot touch the hardware directly
  - Hardware — network cards, disks, clocks
- Asks OS kernel to perform the operation through a **system call**
- Default system calls are ***blocking***
- Call `read(fd, buf, n)`
  - Kernel checks whether data is available on fd
  - If yes —> copy data to buf, return immediately
  - If no —> ***the calling OS thread is suspended***, until data arrives
- OS thread cannot do anything when blocked

What does this mean for our scheduler?

- Our scheduler runs fibers cooperatively on an OS thread (domain)
- On a blocking system call, the entire domain blocks
  - Every other fiber on that domain is also blocked!

Blocking IO starves fibers

```
let rd, _wr = Unix.pipe () in   (* nobody writes to _wr *)
Sched.run ~num_domains:1 (fun () ->
    (* Fiber 1: blocking read — freezes the OS thread *)
    Sched.fork (fun () ->
        Printf.printf "Fiber 1: about to do blocking read (will hang)...\n%!";
        let buf = Bytes.create 1024 in
        let n = Unix.read rd buf 0 1024 in  (* blocks OS thread! *)
        Printf.printf "Fiber 1: got %d bytes (you won't see this)\n%!" n);
    (* Fiber 2: cooperative ticker — yields between prints *)
    let rec ticker i =
      if i >= 5 then
        Printf.printf "Fiber 2: done (you won't see this either)\n%!"
      else begin
        Printf.printf "Fiber 2: tick %d\n%!" i;
        Sched.yield ();
        ticker (i + 1)
      end
    in
    ticker 0)
```

tests/golike_multicore_select/blocking_io_demo.ml

Non-blocking IO

- POSIX provides a flag `O_NONBLOCK` that changes the contract
  - `read(fd,buf,n)` returns immediately with `EAGAIN` or `EWOULDBLOCK` if data is not available
  - Program must decide when to retry

```
(* BAD: spin-wait — wastes CPU, poor latency *)
let rec busy_recv fd buf =
  try Unix.recv fd buf 0 1024 []
  with Unix.Unix_error (EAGAIN, _, _) ->
    Sched.yield ();       (* let other fibers run, but we'll be right back *)
    busy_recv fd buf      (* and waste a full scheduling round each time *)
```

- We need the OS to tell us ***when*** a file descriptor is ready

IO Multiplexing

- Every major OS provides some form of IO multiplexing

Mechanism	OS	Notes
`select`	POSIX (everywhere)	Oldest; simple API; O(n) scan; `FD_SETSIZE` limit (typically 1024)
`poll`	POSIX	Removes `FD_SETSIZE` limit; still O(n) per call
`epoll`	Linux	O(1) readiness notification; edge- or level-triggered
`kqueue`	macOS, BSDs	Similar to epoll; unified file/socket/signal/timer/process events
GCD / dispatch sources	macOS	Integrates with Grand Central Dispatch
IOCP	Windows	Completion-based (not readiness-based); different paradigm
`io_uring`	Linux (5.1+)	Async submission/completion rings; can avoid syscalls entirely

Unix.select

- Our focus will be on **select** system call
- Available in OCaml as **Unix.select**

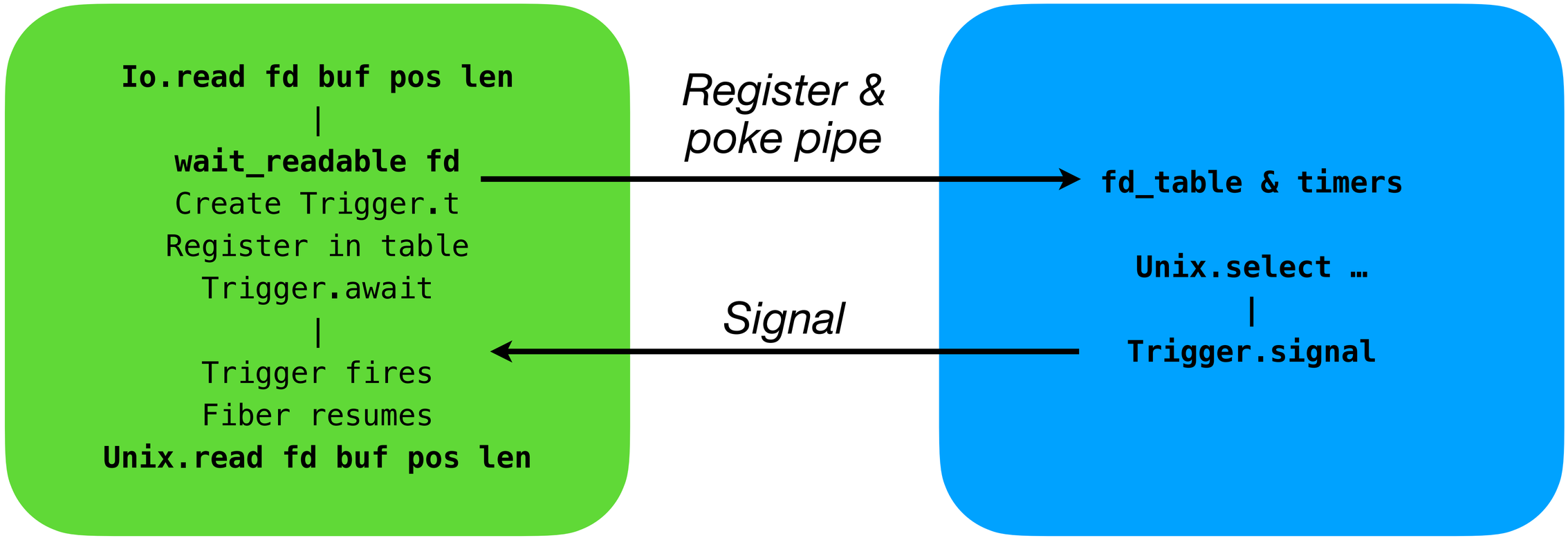
```
val Unix.select : file_descr list -> file_descr list -> file_descr list
                  -> float -> file_descr list * file_descr list * file_descr list
```

- `Unix.select read_fds write_fds _ timeout`
  - ***Blocks*** the calling thread until one of the fds is ***ready*** or the timeout expires
  - Does not perform any IO
  - Negative timeout => timeout doesn't expire
- ***How to integrate this into our scheduler?***

Key insight

- Run Unix.select on a dedicated OS thread (IO thread)
  - This thread does not run any fibers
- In order to perform an IO operation
  - Submit the fd to the IO thread along with a trigger, and then await on the trigger
  - IO thread blocks on Unix.select
  - Signals the trigger when the fd is ready
  - IO operation is then completed by the original thread
- *No change needed to the rest of the scheduler code, the sync libraries, etc!*

Architecture



Fiber domains (run scheduler)

IO System Thread

IO Module Interface

```
val sleep : float -> unit
(** [sleep d] suspends the current fiber for [d] seconds. *)

val read : file_descr -> bytes -> int -> int -> int
(** [read fd buf pos len] waits until [fd] is readable, then reads. *)

val write : file_descr -> bytes -> int -> int -> int
(** [write fd buf pos len] waits until [fd] is writable, then writes. *)

val recv : file_descr -> bytes -> int -> int -> msg_flag list -> int
(** [recv fd buf pos len flags] waits until [fd] is readable, then receives. *)

val send : file_descr -> bytes -> int -> int -> msg_flag list -> int
(** [send fd buf pos len flags] waits until [fd] is writable, then sends. *)

val accept : file_descr -> file_descr * sockaddr
(** [accept fd] waits until [fd] is readable, then accepts a connection.
    The returned socket is set to non-blocking mode. *)

val connect : file_descr -> sockaddr -> unit
(** [connect fd addr] initiates a connection on a non-blocking socket and
    waits until the handshake completes. *)
```

- Same as blocking interface, except fiber-safe
  - Does not block domain!

IO Module

- Much of it is usual socket programming, which we shall skip
  - Focus on concurrency specific parts

```
(** Per-fd read/write trigger queues. *)
type fd_waiters = {
  rd : Trigger.t Queue.t;  (** fibers waiting for read-readiness  *)
  wr : Trigger.t Queue.t;  (** fibers waiting for write-readiness *)
}

type state = {
  mutex : Mutex.t;           (** protects all mutable fields below       *)
  cond : Condition.t;        (** IO thread waits here when idle           *)
  fd_table : (file_descr, fd_waiters) Hashtbl.t; (** fd → waiter queues *)
  timers : (float * Trigger.t) list ref; (** pending sleep deadlines     *)
  started : bool Atomic.t;   (** gates lazy IO thread creation           *)
  wakeup_r : file_descr;     (** self-pipe read end                      *)
  wakeup_w : file_descr;     (** self-pipe write end                     *)
}
```

IO Thread — startup

```
(** Lazily start the IO thread (double-checked lock on [st.started]). *)
let ensure_started () =
  if not (Atomic.get st.started) then begin
    Mutex.lock st.mutex;
    if not (Atomic.get st.started) then begin
      ignore (Thread.create io_loop st.wakeup_r : Thread.t);
      Atomic.set st.started true
    end;
    Mutex.unlock st.mutex
  end
```

- One IO thread per process
  - Shared between multiple domains
- Could potentially have one per domain also

Waiting until fd is readable

```
(** [wait_readable fd] suspends the current fiber until [fd] is readable. *)
let wait_readable fd =
  ensure_started ();        (* lazy initialisation of IO module *)
  let trigger = Trigger.create () in
  Mutex.lock st.mutex;
  let w = get_fd_waiters fd in
  Queue.push trigger w.rd;
  Condition.signal st.cond; (* wake IO thread if it is idle *)
  poke_wakeup ();           (* wake IO thread if it currently doing select *)
  Mutex.unlock st.mutex;
  Trigger.await trigger
```

Public API

```
let recv fd buf pos len flags =
  let rec loop () =
    match Unix.recv fd buf pos len flags with
    | n -> n
    | exception Unix.Unix_error ((Unix.EAGAIN | Unix.EWOULDBLOCK), _, _) ->
        wait_readable fd; loop ()
  in
  loop ()
```

- **Assume/Expect** fd to be a non-blocking fd
- Attempt to do the operation, if it “would block”
  - Block this fiber until fd is ready for read, and try again
- **Fast path:** If the fd is ready to read, no suspension occurs

Timeouts

```
let register_timer delay trigger =
  ensure_started ();
  let deadline = Unix.gettimeofday () +. delay in
  Mutex.lock st.mutex;
  st.timers := (deadline, trigger) :: !(st.timers);
  Condition.signal st.cond;
  poke_wakeup ();
  Mutex.unlock st.mutex

let sleep delay =
  if delay <= 0. then ()
  else begin
    let trigger = Trigger.create () in
    register_timer delay trigger;
    Trigger.await trigger
  end
```

IO Thread — IO Loop

```
let rec io_loop wakeup_fd =
  Mutex.lock st.mutex;
  while Hashtbl.length st.fd_table = 0 && !(st.timers) = [] do
    Condition.wait st.cond st.mutex
  done;
  fire_due_timers_locked (Unix.gettimeofday ()); (* signal all timers who deadline has passed *)
  let now = Unix.gettimeofday () in
  let timeout = timeout_locked now in (* time until the earliest pending timer or -1 *)
  let readable, writable =
    ready_fds_locked () (** Collect file descriptors that have waiting readers/writers. *)
  in
  Mutex.unlock st.mutex;

  let readable, writable, _ =
   Unix.select (wakeup_fd :: readable) writable [] timeout
  in

  Mutex.lock st.mutex;
  process_ready_locked wakeup_fd readable writable; (* Signal triggers for these ready fds *)
  Mutex.unlock st.mutex;
  io_loop wakeup_fd
```

Blocking IO starves fibers

```
let rd, _wr = Unix.pipe () in   (* nobody writes to _wr *)
Sched.run ~num_domains:1 (fun () ->
    (* Fiber 1: blocking read — freezes the OS thread *)
    Sched.fork (fun () ->
        Printf.printf "Fiber 1: about to do blocking read (will hang)...\n%!";
        let buf = Bytes.create 1024 in
        let n = Unix.read rd buf 0 1024 in  (* blocks OS thread! *)
        Printf.printf "Fiber 1: got %d bytes (you won't see this)\n%!" n);
    (* Fiber 2: cooperative ticker — yields between prints *)
    let rec ticker i =
      if i >= 5 then
        Printf.printf "Fiber 2: done (you won't see this either)\n%!"
      else begin
        Printf.printf "Fiber 2: tick %d\n%!" i;
        Sched.yield ();
        ticker (i + 1)
      end
    in
    ticker 0)
```

tests/golike_multicore_select/blocking_io_demo.ml

Async IO does not starve fibers

```
let rd, _wr = Unix.pipe () in   (* nobody writes to _wr *)
 Unix.set_nonblock rd;
 Sched.run ~num_domains:1 (fun () ->
    (* Fiber 1: non-blocking read — suspends without freezing the OS thread *)
    Sched.fork (fun () ->
        Printf.printf "Fiber 1: about to do non-blocking read (will suspend, not hang)...\n%!";
        let buf = Bytes.create 1024 in
        let n = Io.read rd buf 0 1024 in
        Printf.printf "Fiber 1: got %d bytes\n%!" n);
    (* Fiber 2: cooperative ticker — yields between prints *)
    let rec ticker i =
      if i >= 5 then
        Printf.printf "Fiber 2: done — read never blocked us!\n%!"
      else begin
        Printf.printf "Fiber 2: tick %d\n%!" i;
        Sched.yield ();
        ticker (i + 1)
      end
    in
    ticker 0)
```

tests/golike_multicore_select/async_io_demo.ml

Async IO — can also sleep fine

```
let rd, _wr = Unix.pipe () in   (* nobody writes to _wr *)
 Unix.set_nonblock rd;
 Sched.run ~num_domains:1 (fun () ->
    (* Fiber 1: non-blocking read — suspends without freezing the OS thread *)
    Sched.fork (fun () ->
        Printf.printf "Fiber 1: about to do non-blocking read (will suspend, not hang)...\n%!";
        let buf = Bytes.create 1024 in
        let n = Io.read rd buf 0 1024 in
        Printf.printf "Fiber 1: got %d bytes\n%!" n);
    (* Fiber 2: cooperative ticker — sleeps 1s between prints *)
    let rec ticker i =
      if i >= 5 then
        Printf.printf "Fiber 2: done — read never blocked us!\n%!"
      else begin
        Printf.printf "Fiber 2: tick %d\n%!" i;
        Io.sleep 1.0;
        ticker (i + 1)
      end
    in
    ticker 0)
```

tests/golike_multicore_select/async_io_sleep_demo.ml

Composable Timeout

- Unix.select had a option to do time waiting

```
val Unix.select : file_descr list -> file_descr list -> file_descr list
                -> float -> file_descr list * file_descr list * file_descr list
```

- Select.select in our library does not have such an ability.
  - Wouldn’t it be nice to have such a facility?

```
let timeout_evt delay : unit Select.event = Select.Evt {
  try_complete = (fun () -> if delay <= 0. then Some () else None);
  offer = (fun slot trigger ->
    let proxy = Trigger.create () in
    ignore (Trigger.on_signal proxy (fun () ->
      slot := Some ();
      if not (Trigger.signal trigger) then
        slot := None) : bool);
    register_timer delay proxy);
  mutex = timeout_mutex;
  wrap = Fun.id;
}
```

Composable Timeout

```
let sender ch n =
  for i = 0 to n - 1 do
    Io.sleep 1.0;
    Chan.send ch i;
  done;
  Printf.printf "[sender] done\n%!"

let receiver ch n =
  let received = ref 0 in
  while !received < n do
    (match
      Select.select
        [ Chan.recv_evt ch |> Select.wrap (fun v  -> `Msg v)
        ; timeout_evt 0.5  |> Select.wrap (fun () -> `Timeout)
        ]
    with
    | `Msg v  -> Printf.printf "[receiver] msg %d\n%!" v; incr received
    | `Timeout -> Printf.printf "[receiver] timeout\n%!");
  done;
  Printf.printf "[receiver] done\n%!"
```

dune exec ./select_recv_timeout_test.exe

Echo server

- Server echoes the messages sent by clients over TCP
- Server fiber calls **lo.accept** in a loop
  - New session fiber for each accepted connection
- Clients use **lo.connect**, **lo.send** and **lo.recv** to communicate with the server

Server accept loop

```
let run_server port =
  let fd = Unix.socket ~cloexec:true Unix.PF_INET Unix.SOCK_STREAM 0 in
  Unix.set_nonblock fd;
  Unix.setsockopt fd Unix.SO_REUSEADDR true;
  Unix.bind fd (Unix.ADDR_INET (Unix.inet_addr_loopback, port));
  Unix.listen fd 5;
  Printf.printf "echo server listening on port %d\n%!" port;
  Sched.run ~num_domains:4 (fun () ->
      let rec accept_loop () =
        let cfd, _addr = Io.accept fd in
        Sched.fork (fun () -> echo_handler cfd);
        accept_loop ()
      in
      accept_loop ())
```

Server echo loop

```
let echo_handler cfd =
  let buf = Bytes.create 1024 in
  let rec loop () =
    let n = Io.recv cfd buf 0 (Bytes.length buf) [] in
    if n > 0 then begin
      let rec send_all off rem =
        if rem > 0 then
          let w = Io.send cfd buf off rem [] in
          send_all (off + w) (rem - w)
      in
      send_all 0 n;
      loop ()
    end
  in
  (try loop () with Unix.Unix_error _ -> ());
  Unix.close cfd
```

Client

```
let run_client port =
  Sched.run ~num_domains:1 (fun () ->
      let fd = Unix.socket ~cloexec:true Unix.PF_INET Unix.SOCK_STREAM 0 in
      Unix.set_nonblock fd;
      Io.connect fd (Unix.ADDR_INET (Unix.inet_addr_loopback, port));
      let buf = Bytes.create 1024 in
      let rec loop () =
        Printf.printf "> %!";
        let line = input_line stdin in
        let payload = Bytes.of_string line in
        ignore (Io.send fd payload 0 (Bytes.length payload) [] : int);
        let n = Io.recv fd buf 0 (Bytes.length buf) [] in
        if n > 0 then begin
          Printf.printf "%s\n%!" (Bytes.sub_string buf 0 n);
          loop ()
        end
      in
      (try loop () with End_of_file -> ());
      Unix.close fd)
```

Async IO — discussion

- How many OS threads do we use?
  - 1 for the IO thread and N domains in the domain pool
  - Could have 1 IO thread per domain if necessary
- 1M clients —> 1M fibers
  - Still N domains + 1 IO thread!
- Comparison to Go's netpoller
  - Same idea, much simpler
  - Go integrates the poller into the runtime; ours is a library module
  - ***Modular*** — did not modify the scheduler, trigger, channel, ivar, promise

Summary

- **Effect handlers permit concurrent programming as modular libraries**
  - Build concurrency runtime from scratch, one layer at a time
- Fiber — cooperative concurrency via effect handlers
- Parallelism — through OS threads + concurrent data structures
  - Parallel algorithms through nested parallelism
- Synchronisation structures
  - channels, ivars, promises
  - Selective communication
- Async IO as a separate module
- Synchronisation structures and asynchronous IO independent of the scheduler