Continuation-Passing C

Programming with a massive number of lightweight threads

Gabriel Kerneis, Juliusz Chroboczek

Laboratoire PPS

Université Paris Diderot, France

PLACES, Saarbrücken, 2 April 2011

Writing high-concurrency servers

Efficient **concurrency** in C:

scaling your server to thousands of clients...

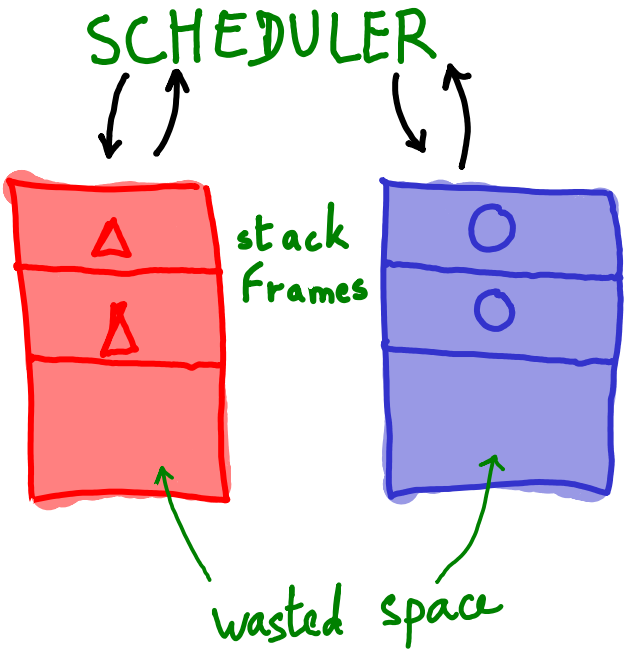
and running it on your

pocket calculator.

Threads and events

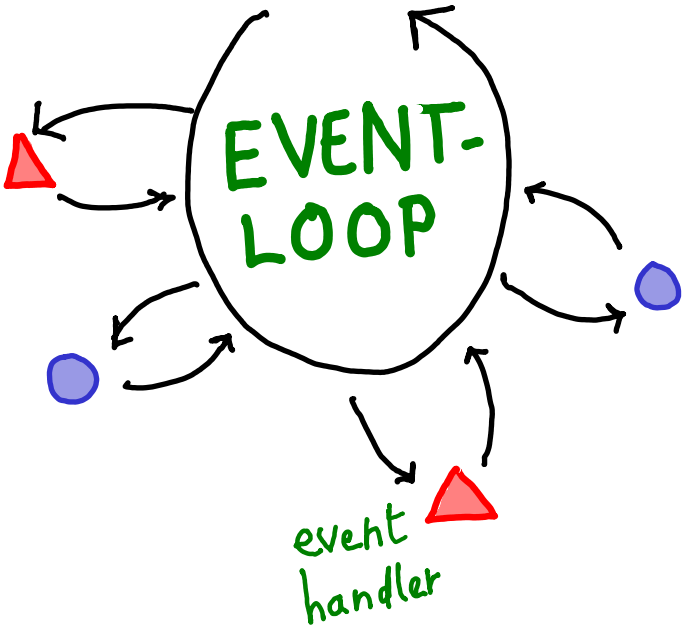
Threads

heavyweight, easy



Events

lightweight, crazy



Lightweight threads

Many user-space **libraries** provide lightweight threads.

A few **compiler-based** frameworks too.
Capriccio, Tamer

Often restricted to **cooperative** threads.

Continuation-Passing C

Continuation-Passing C provides

"**CPC threads**", compiled to

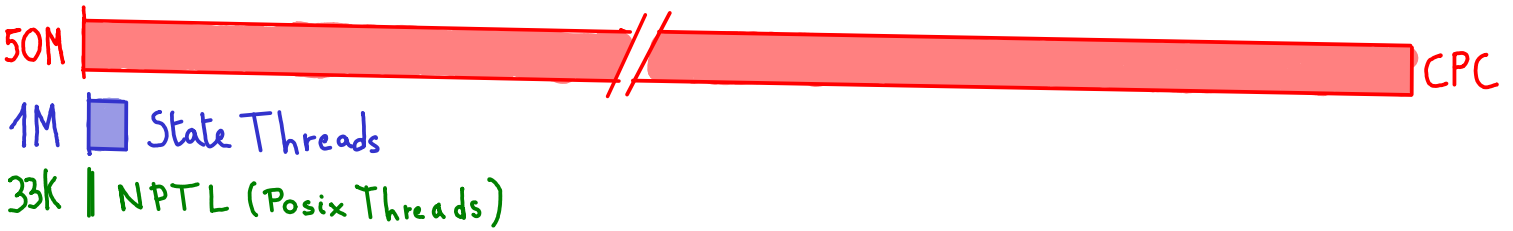
- **event-driven** code, or
- **native** threads

at the **programmer's choice**.

```
cpc_spawn { printf("world\n"); }
printf("Hello... ");
```

Threads for free, everywhere

CPC threads are lightweight.



It has an impact on your
programming style.

Threads are CPC's **unit of modularity**.

Learning CPC programming

Let's try and discover new **idioms**,
writing a non-negligible program.

Hekate is a **BitTorrent** server

written in CPC by **undergrads**,

designed to handle **thousands** of clients.

Outline

Cooperative CPC threads

Detached (native) threads

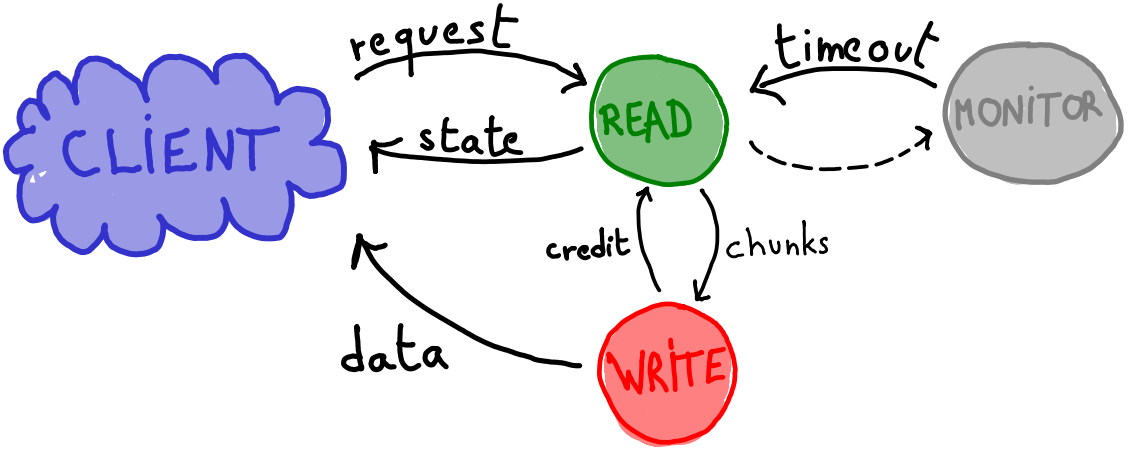
Hybrid programming

Cooperative CPC threads

Really lightweight threads

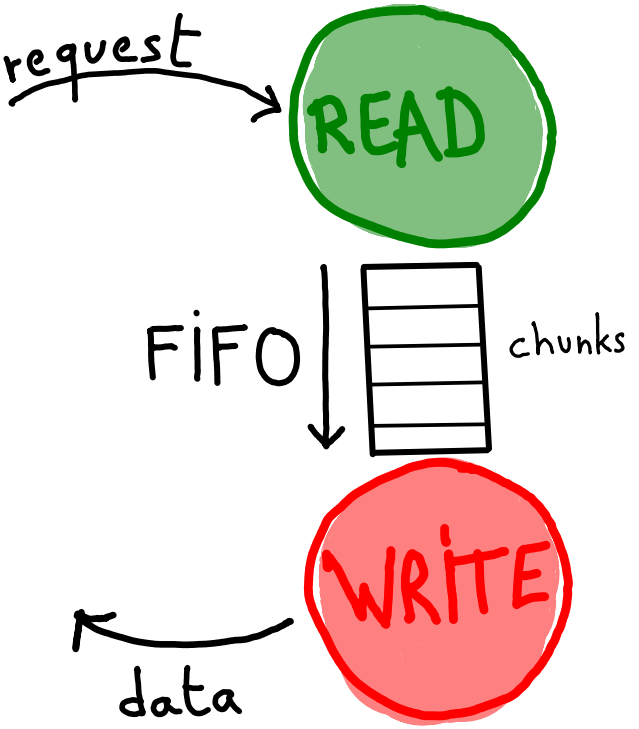
Cooperative CPC threads are compiled to **event-handlers**.

Hekate spawns **three** CPC threads per client.



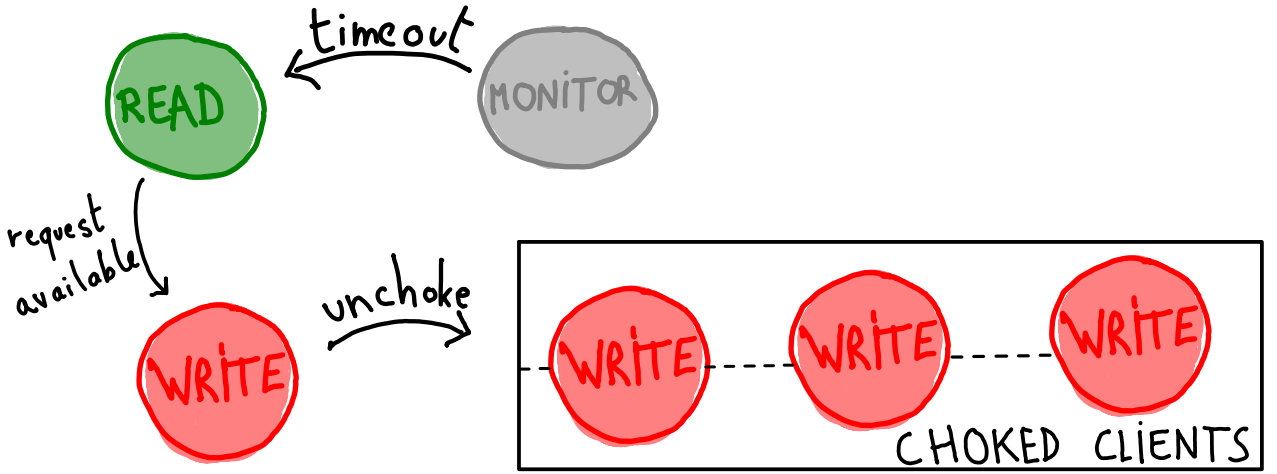
Shared-memory without locks

The list of pending chunks is **shared** and accessed **cooperatively** (no lock).



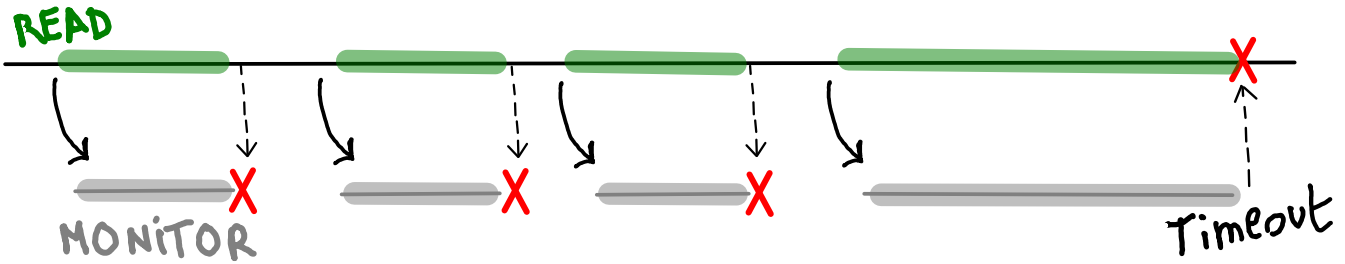
Synchronisation

A single simple synchronisation primitive:
condition variables.



Managing timeouts

A **new** timeout thread is spawned
for **every** read.



A lot of very **short-lived** threads.
Easy. Efficient.

Detached (native) threads

When cooperating is not enough

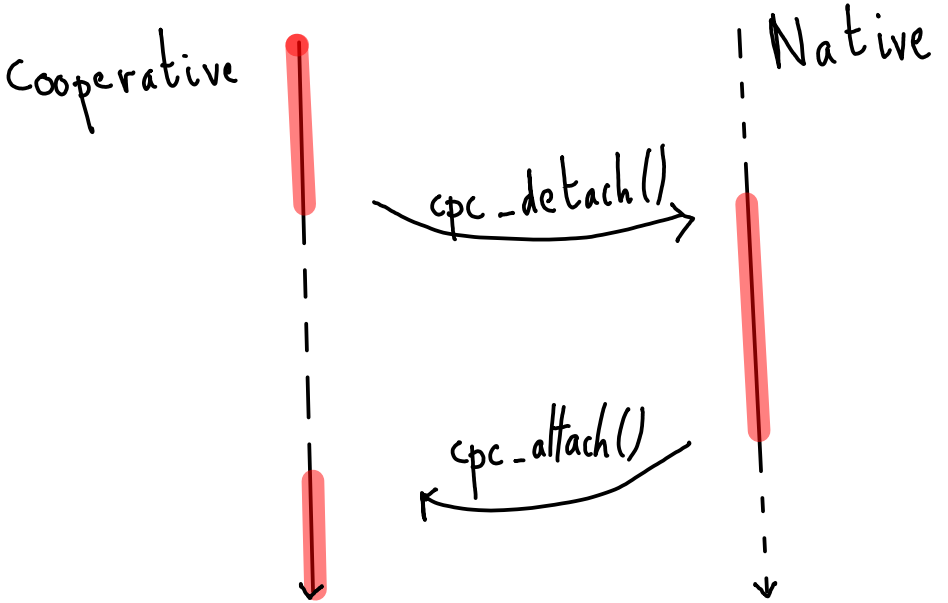
Cooperating threads are fast and easier to use.

But sometimes, you need **native threads**:

- **blocking** OS interfaces,
- **blocking** external libraries,
- **parallelism** (not in Hekate yet).

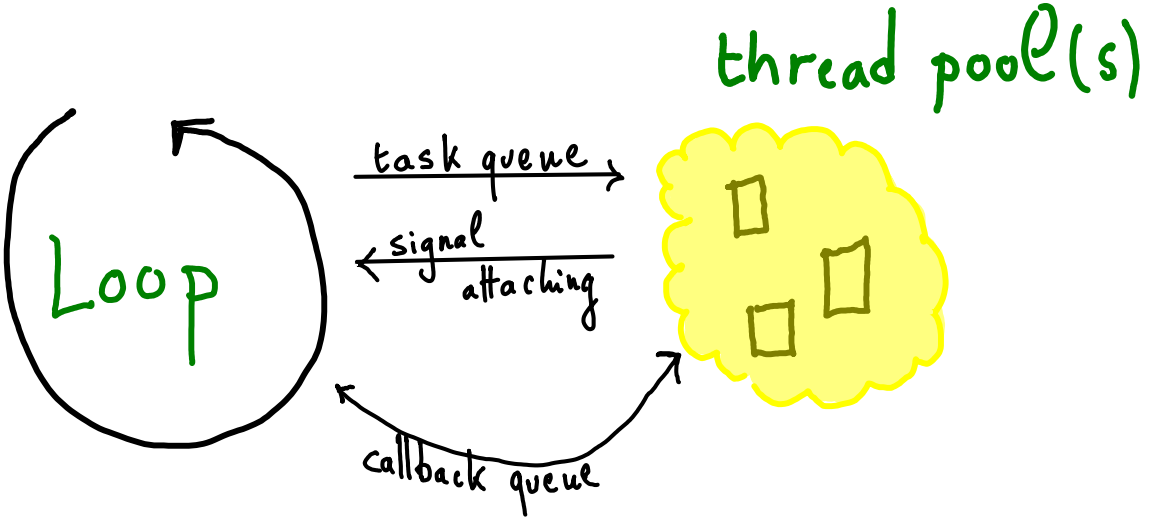
Detached (native) threads

The programmer can **switch** a thread between cooperative and native mode **on-the-fly**.



Detached (native) threads

Lots of **magic** to make this efficient (thread pools, non-blocking queues).



Detached (native) threads in Hekate

Blocking interface: `getaddrinfo` (DNS).

```
cpc_detached {
   rc = getaddrinfo(name, ...);
   return rc;
}
```

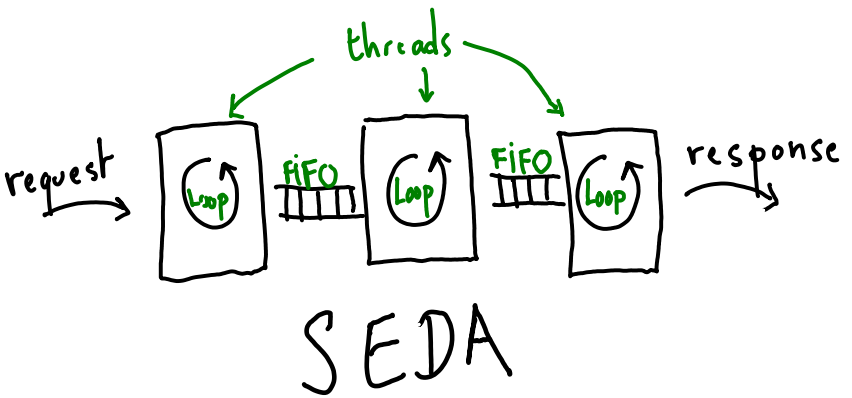
Another example: *libcurl* (HTTP requests).

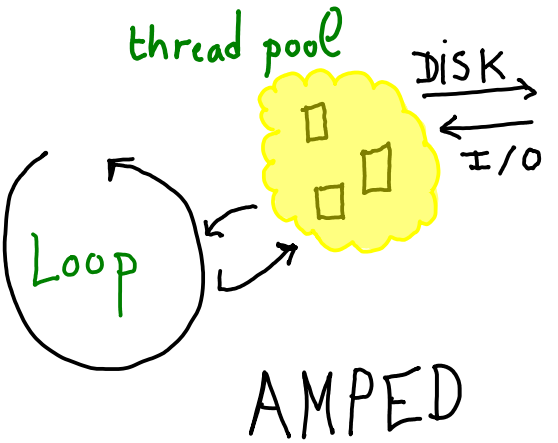
Hybrid programming

Hybrid programming

Combining events for efficient concurrency and threads for blocking parts.

Many “event-driven” programs are actually **hybrid** programs.





Uniform primitives

CPC makes hybrid programming
pleasant and **easy**.

CPC primitives are **well-behaved**
in both attached and detached mode
(except condition variables).

“Write once, run in every mode.”

Blocking disk reads

Disk reads are slow: might **block** if the data is not in cache.

Using a **native** thread: avoids blocking, but heavyweight.

Using a **cooperative** thread: efficient, but you need to prefetch data into the cache.

Blocking disk reads

```
prefetch(data); yield();
if cached(data)
   then send(data);
    else yield();
         if cached(data)
            then send(data);
             else detach();
                  send(data);
                  attach();
         fi
fi
```

Cooperative

Cooperative

Native

Conclusion

Programming with CPC is **pleasant** and **convenient**.

Having many threads yields a different **programming style**.

The resulting code is **efficient**: performance similar to hand-written event-driven code.

Appendix

Blocking disk I/O: the code

```
  prefetch(source, length);                    /* (1) */
  cpc_yield();                                 /* (2) */
  if(!incore(source, length)) {                /* (3) */
    cpc_yield();                               /* (4) */
    if(!incore(source, length)) {              /* (5) */
      cpc_detached {                           /* (6) */
        rc = cpc_write(fd, source, length);
      }
      goto done;
    }
  }
  rc = cpc_write(fd, source, length);          /* (7) */
done:
  ...
```