io_uring

Efficient Input & Output
on Modern Linux Kernels

Input & Output: sequential

```
import socket

def handle(conn, addr):
  print('handling', addr)
  while True:
    data = conn.recv(1024)
    if not data: break
    conn.sendall(data)
  conn.close()
  print('closed', addr)
```

```
def serve(host, port):
  s = socket.socket(socket.AF_INET,
        socket.SOCK_STREAM)
  s.bind((host, port))
  print('listening on', port)
  s.listen()

  while True:
    conn, addr = s.accept()
    handle(conn, addr)


# main
serve("localhost", 4223)
```

Input & Output: parallel

```
import socket
from threading import Thread

def handle(conn, addr):
  print('handling', addr)
  while True:
    data = conn.recv(1024)
    if not data: break
    conn.sendall(data)
  conn.close()
  print('closed', addr)
```

```
def serve(host, port):
  s = socket.socket(socket.AF_INET,
        socket.SOCK_STREAM)
  s.bind((host, port))
  print('listening on', port)
  s.listen()

  while True:
    conn, addr = s.accept()
    Thread(
      target=
        lambda: handle(conn, addr)
     ).start()

# main
serve("localhost", 4223)
```

Input & Output: multiplexing

```
NAME
    select – synchronous I/O multiplexing

SYNOPSIS
    #include <sys/select.h>

    int select(int nfds, fd_set *readfds,
           fd_set *writefds, fd_set *errorfds,
           struct timeval *timeout);
```

Input & Output: history http://www.kegel.com/c10k.html

- 1983: The Berkely Socket API, later POSIX
  - send(), recv(), connect(), accept()
  - select(): O(highest file descriptor)
- 1987: poll() on POSIX
  - O(number of file descriptors)
- 2000: /dev/poll on Solaris
  - avoids walking to full list of fds each time
  - O(active file descriptors)
- 2000: kqueue() on BSD
  - O(active file descriptors)
- 2003: epoll() on Linux
  - O(active file descriptors)
- 2020: io_uring on Linux

Asynchronous I/O

So far, we have seen synchronous I/O.

Threads are blocked while I/O is done.

Multiplexing allows “non-blocking I/O” which awaits blocking if there is no I/O to be done.

Threads are still blocked when performing I/O.

Many languages have support for asynchronous I/O:

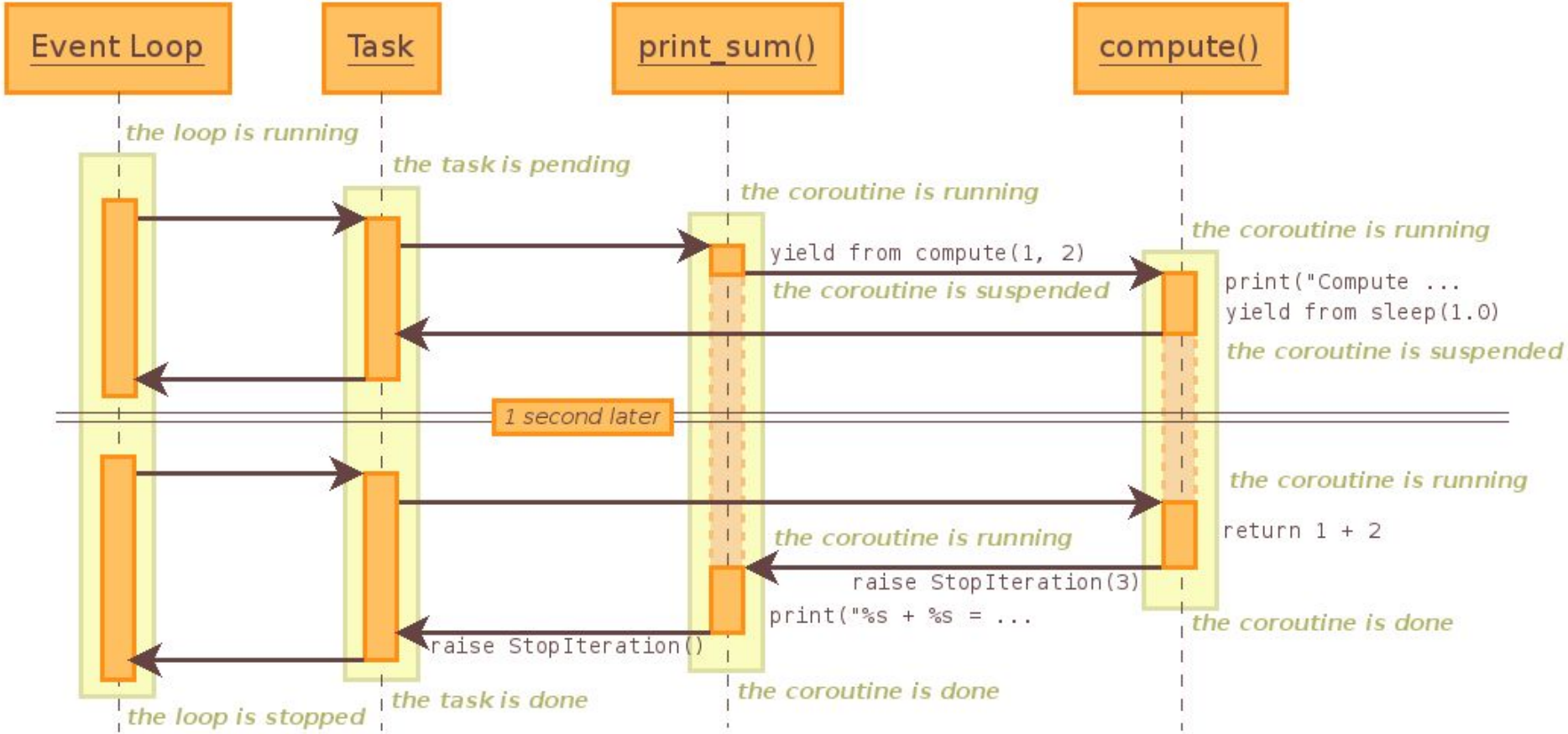
I/O is performed the background by the event loop.

- async/await
  - Python, C#, Rust, Javascript 2017
- callback-based
  - NodeJS

Input & Output: asyncio

```
import asyncio

async def handle(reader, writer):
  addr = writer
    .get_extra_info('peername')
  print('handling', addr)

  while True:
    data = await reader.read(1024)
    if not data: break
    writer.write(data)

  writer.close()
  print('connection closed', addr)
```

```
async def serve(host, port):
  server = await asyncio
    .start_server(handle,
       host, port)
  print('listening on', port)
  async with server:
    await server.serve_forever()

asyncio.run(serve("localhost",
4223))
```


Event Loop
Task
print_sum()
compute()
the loop is running
the task is pending
the coroutine is running
the coroutine is running
yield from compute(1, 2)
the coroutine is suspended
print("Compute ...
yield from sleep(1.0)
the coroutine is suspended
1 second later
the coroutine is running
return 1 + 2
the coroutine is running
raise StopIteration(3)
print("%s + %s = ...
the coroutine is done
raise StopIteration()
the coroutine is done
the task is done
the loop is stopped

io_uring

io_uring is a kernel-based asynchronous I/O mechanism.

Added to Linux kernel 5.1 in 2019. New features added in every release since.

You can submit I/O operations to the kernel without having to wait.

Two queues:

Submission Queue (SQ):

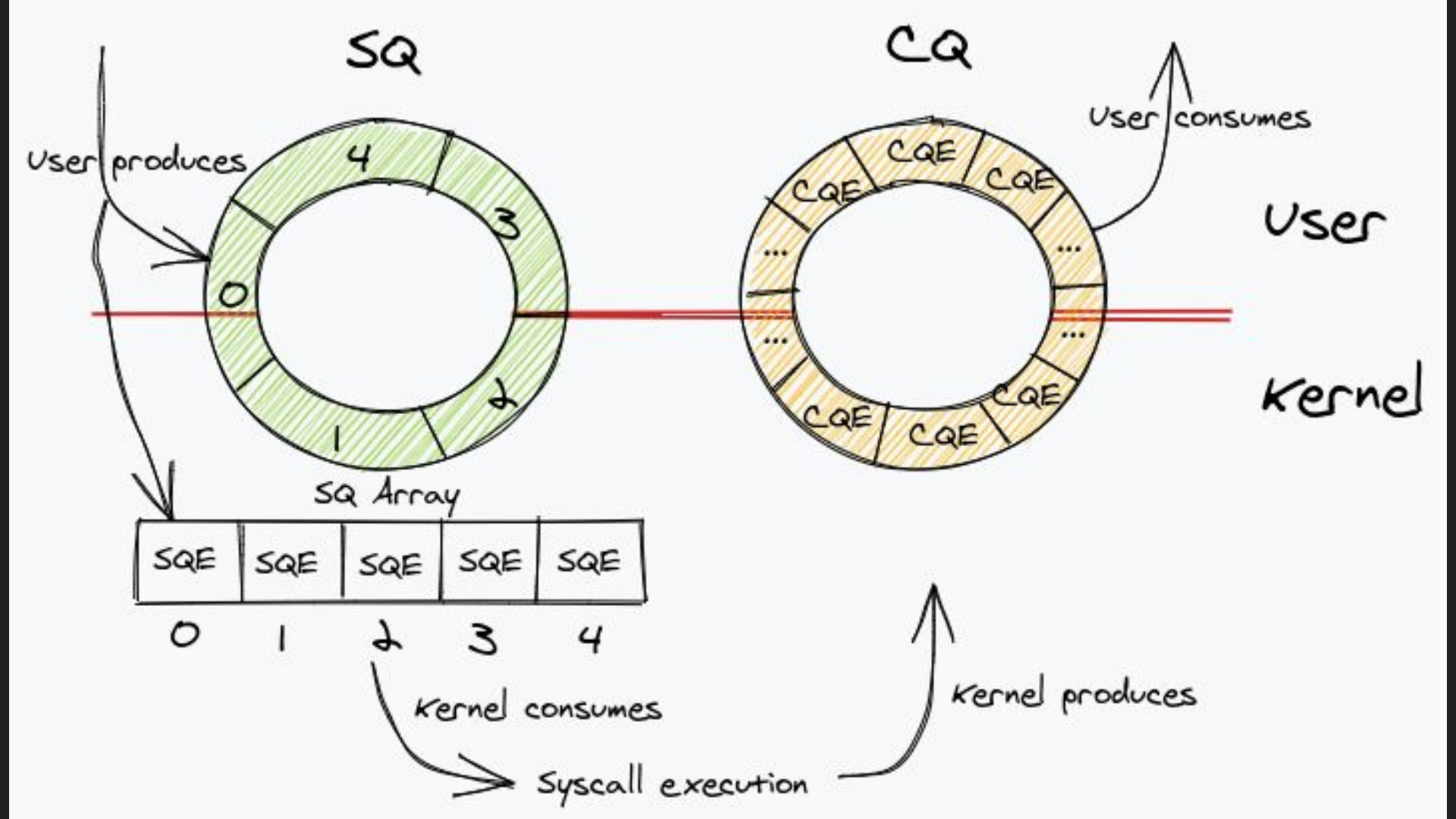
written by the user program, read by the kernel

Completion Queue (CQ):

written by the kernel, read by the user program

io_uring

io_uring moves the event loop into the kernel


SQ
CQ
User produces
User consumes
User
Kernel
0
1
2
3
4
CQE
SQ Array
SQE
SQE
SQE
SQE
SQE
0
1
2
3
4
Kernel consumes
Syscall execution
Kernel produces

io_uring: buf=conn.read(1024)

Submission Queue Entry (SQE):

```
op: IORING_OP_READ
fd: conn
addr: buff
len: 1024
user_data: 0xC0FFEE
```

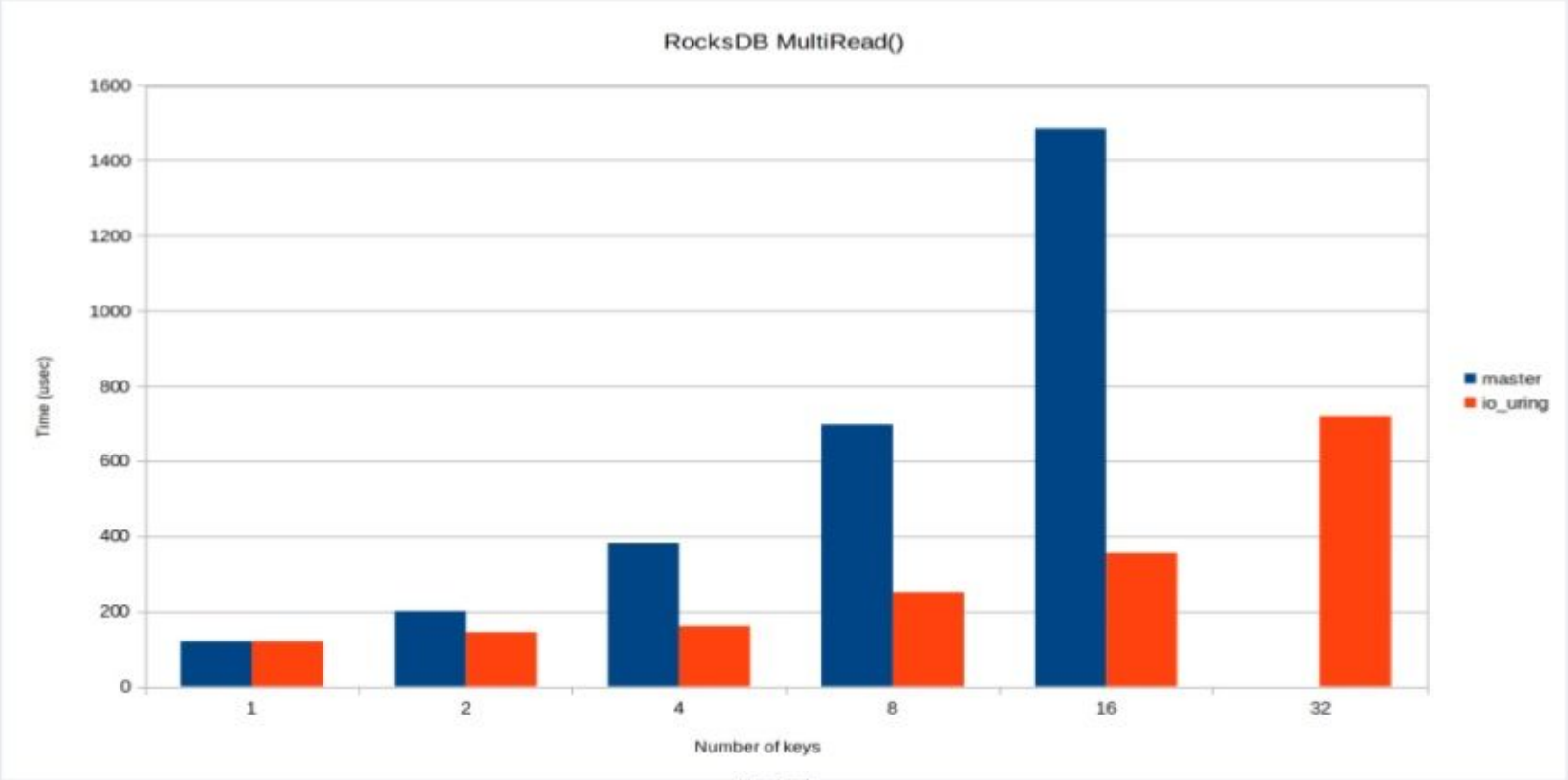
Completion Queue Entry (CQE):

```
res: 10
user_data: 0xC0FFEE
```

Additional features

- I/O Submission without syscalls
  - “polling” mode
- Support for asynchronous file I/O
  - not supported by select/epoll/kqueue
  - very useful for for DBs
- Chaining I/O operations
  - Sending “mini-programs” to the kernel, e.g.
    - 1. read 10 bytes from conn
    - 2. write 10 bytes to conn
    - 3. close conn
  - kernel will notify you and the end of a sequence

RocksDB MultiRead() test



io_uring: how to get it

- libuv (used by many open-source projects)
  - NodeJS
  - Julia
  - CMake
- Postgres (database)
- Ceph (network storage)
- RocksDB (key-value store)
- Language libraires
  - C/C++: liburing or low-level system call
  - Rust: ringbahn, rio
  - Others: bindings to liburing