Brave New Asynchronous World

Towards Asynchronous System Calls For Concurrency Platforms

March 13, 2021

Florian Schmaus Florian Fischer

Friedrich-Alexander-Universität Erlangen-Nürnberg

Lehrstuhl für Verteilte Systeme
und Betriebssysteme

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

Motivation

```
void server() {
  fd = accept();

  read(fd, input);

  output = process(input);

  write(fd, output);

  close(fd);
}
```

Sequential server using blocking operations

Motivation

```
void server() {
  fd = accept();

  read(fd, input);

  output = process(input);

  write(fd, output);

  close(fd);
}
```

Sequential server using blocking operations

What if we want to serve many clients?

Motivation

```
void server() {
  fd = accept();

  read(fd, input);

  output = process(input);

  write(fd, output);

  close(fd);
}
```

Sequential server using blocking operations

```
eventLoop loop;
void on_accept(fd) {
    loop.register(fd, READ);
}
void on_read(fd, input) {
    output = process(input);
    loop.register(fd, WRITE, output);
}
void on_write(fd, output) { close(fd); }
void server() {
    loop.register(listener, ACCEPT);
    loop.run();
}
```

Event-driven server

Motivation

- ☺ Handles multiple clients with just one CPU core
- ☹ Scattered logic and control flow

  von Behren et al., HOTOS'03
- How to run parallel on many cores? Threadpool?
- ☹ What if `process()` blocks?

```
eventLoop loop;
void on_accept(fd) {
    loop.register(fd, READ);
}
void on_read(fd, input) {
    output = process(input);
    loop.register(fd, WRITE, output);
}
void on_write(fd, output) { close(fd); }
void server() {
    loop.register(listener, ACCEPT);
    loop.run();
}
```

Event-driven server

Motivation

Can’t we have everything?

- Comprehensible source code
- Non-blocking IO operations
- Scalability for future many-core architectures

Motivation

Can’t we have everything?

- Comprehensible source code
- Non-blocking IO operations
- Scalability for future many-core architectures

Yes, we can!

Pseudo-blocking System Calls for Concurrency Platforms

Outline

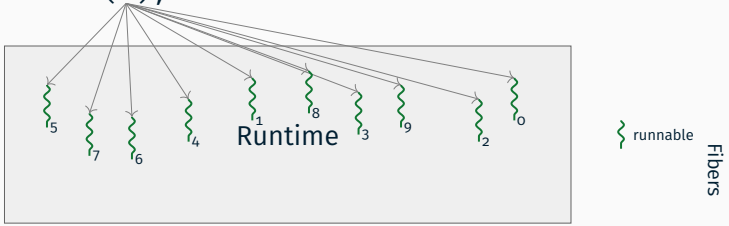
1. Motivation
2. Concurrency Platforms
3. Blocking Anomaly
4. Short History of Operating-System Request Techniques
5. Concurrency Platforms and Asynchronous System Calls
6. Conclusion

Concurrency Platform

```
cilk_for (int i = 0; i < 10; ++i)
    f(i);
```

Concurrency Platform

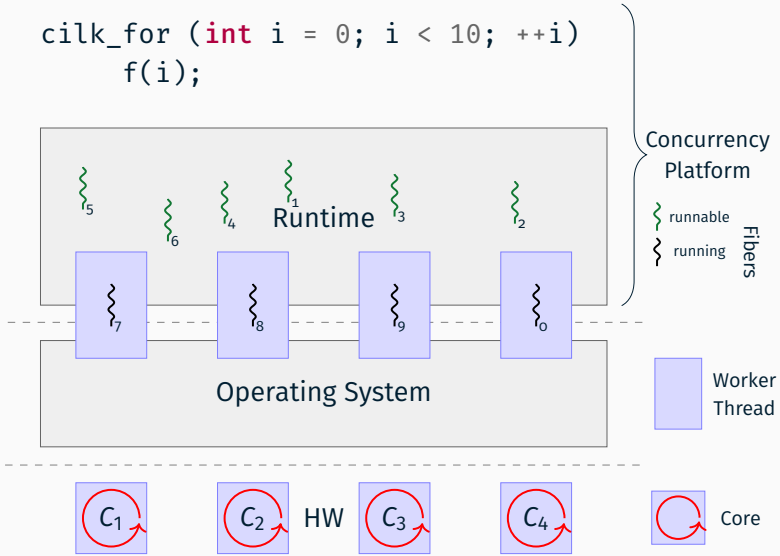
```
cilk_for (int i = 0; i < 10; ++i)
    f(i);
```



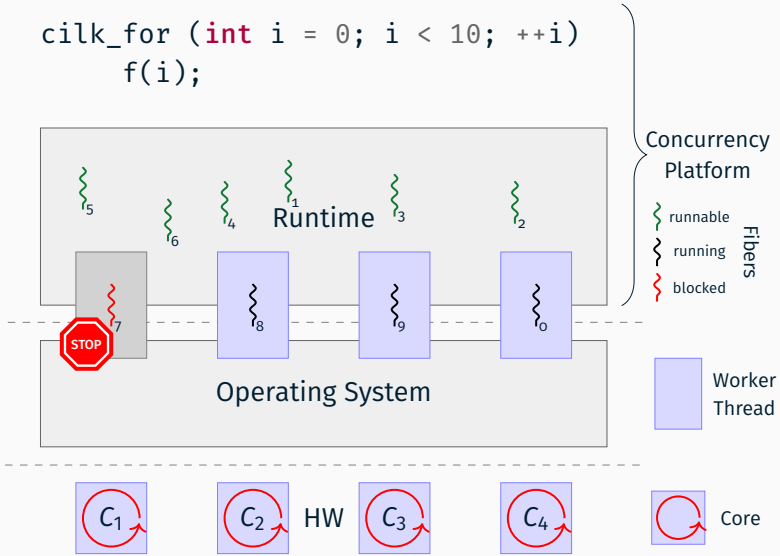
Concurrency Platform

```
cilk_for (int i = 0; i < 10; ++i)
    f(i);
```

Concurrency Platform



Concurrency Platform



Concurrency And Blocking Syscalls

Blocking Anomaly[1]

- Blocking Fiber also blocks Worker
- $\Rightarrow$ Resources may be underutilized
- Blocking is transparent for the runtime system
- $\Rightarrow$ Requires cooperation with Operating System
- Not specific to Concurrency Platforms, also an issue with thread pools etc.

[1] Yangmin Seo et al. 1999. Supporting preemptive multithreading in the ARX real-time operating system. In *Proceedings of the IEEE Region 10 Conference*. TENCON 99. Volume 1. (September 1999), 443–446.

Approaches

- Detection (Linux Delay Accounting)

Approaches

- Detection (Linux Delay Accounting)
- Mitigation (Scheduler Activations[2])

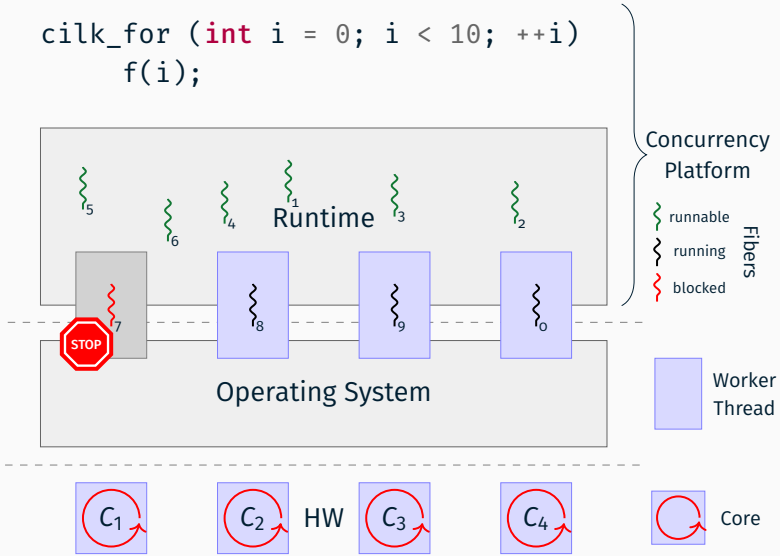
[2] Thomas E Anderson et al. 1992. Scheduler activations: effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems (TOCS)*, 10, 1, 53–79.

Approaches

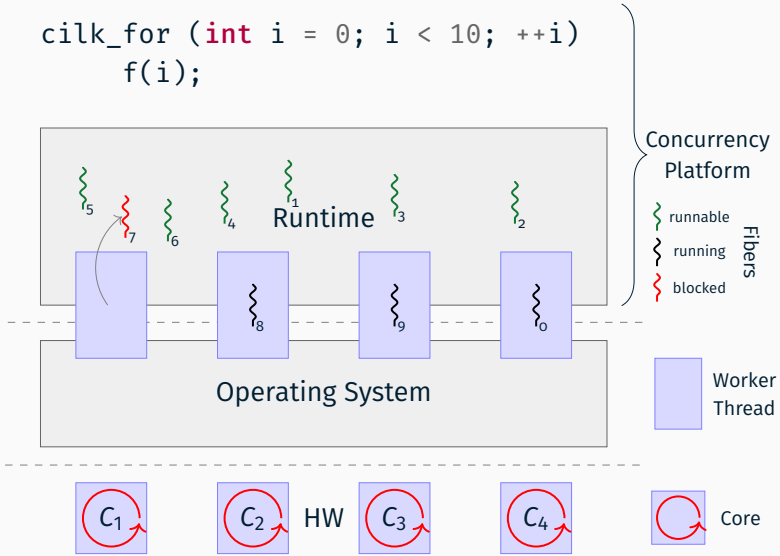
- Detection (Linux Delay Accounting)
- Mitigation (Scheduler Activations[2])
- Avoidance (Non-Blocking or Asynchronous System Calls)

[2] Thomas E Anderson et al. 1992. Scheduler activations: effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems (TOCS)*, 10, 1, 53–79.

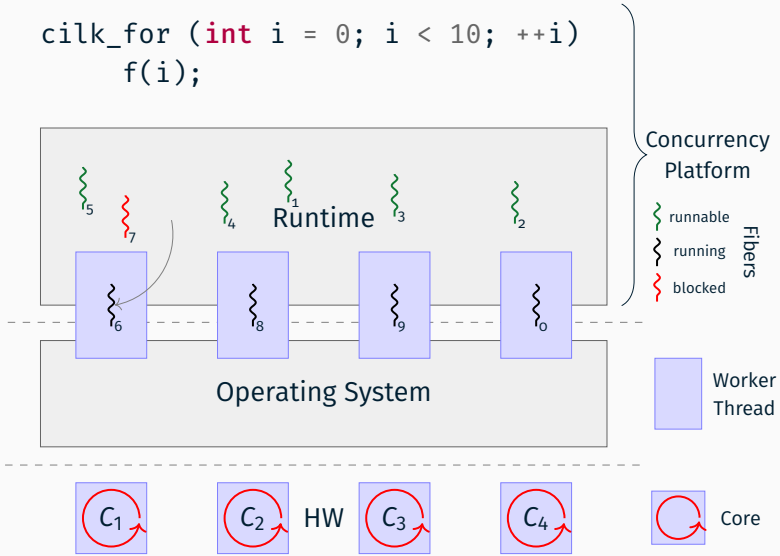
Concurrency Platform



Concurrency Platform



Concurrency Platform



Pseudo-synchronous interface

```
ssize_t read(int fd, void* buf, size_t len) {
  ReadFuture future(fd, buf, len);
  return future.get(); // Fiber may block here.
}
```

- Transform asynchronous APIs to synchronous ones
- Base future on concurrency-platform native synchronization primitive

From here on:
IO system calls as representative of arbitrary blocking system calls.

Pseudo-synchronous interface

```
ssize_t read(int fd, void* buf, size_t len) {
  ReadFuture future(fd, buf, len);
  return future.get(); // Fiber may block here.
}
```

- Transform asynchronous APIs to synchronous ones
- Base future on concurrency-platform native synchronization primitive

From here on:
IO system calls as representative of arbitrary blocking system calls.

But who does fullfil the future's promise?

System (Call) Request Techniques

Kind	Technique	Examples	$\#\frac{\text{traps}}{\text{sys requests}}$
Synchronous	Blocking	e.g. `read()`/`write()`	1
Synchronous	Non-Blocking	SOCK_NONBLOCK w select/epoll	≥ 1
Asynchronous	Callback	Linux/POSIX AIO	1

System (Call) Request Techniques

Kind	Technique	Examples	$\#\frac{\text{traps}}{\text{sys requests}}$
Synchronous	Blocking	e.g. `read()`/`write()`	1
Synchronous	Non-Blocking	SOCK_NONBLOCK w select/epoll	≥ 1
Asynchronous	Callback	Linux/POSIX AIO	1

- “Traditional” Syscall Technique
- Causes Blocking Anomaly

System (Call) Request Techniques

Kind	Technique	Examples	$\#\frac{\text{traps}}{\text{sys requests}}$
Synchronous	Blocking	e.g. `read()`/`write()`	1
Synchronous	Non-Blocking	SOCK_NONBLOCK w select/epoll	≥ 1
Asynchronous	Callback	Linux/POSIX AIO	1

- C10k Problem (Kegel 1999)
- Reactor / Proactor design pattern
- ☹ Hard to implement, maintain and debug

System (Call) Request Techniques

Kind	Technique	Examples	$\#\frac{\text{traps}}{\text{sys requests}}$
Synchronous	Blocking	e.g. `read()`/`write()`	1
Synchronous	Non-Blocking	SOCK_NONBLOCK w select/epoll	≥ 1
Asynchronous	Callback	Linux/POSIX AIO	1

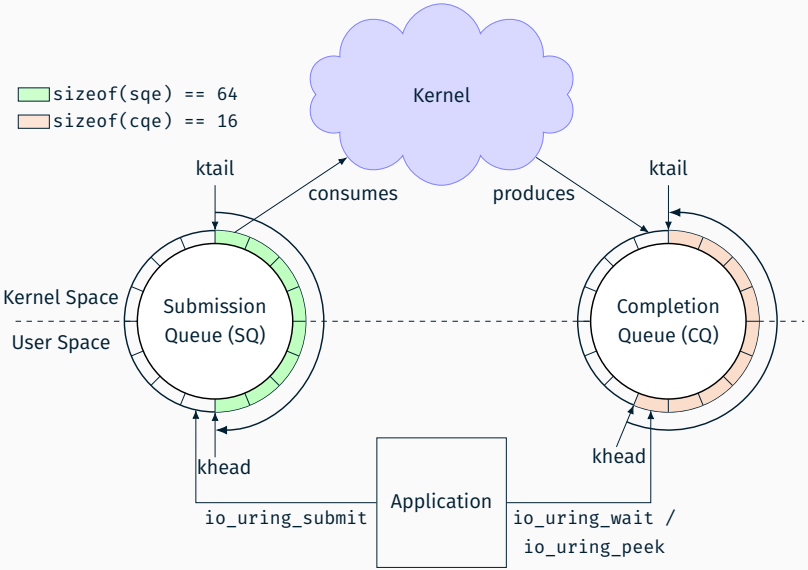
- Invoke callback (signal, new thread) upon system call completion
- ☹ Signal delivery impacts Instruction Per Cycle (IPC)
- ☹ New thread has a high ramp-up latency and competes with existing threads
- "AIO is a horrible ad-hoc design, ..." (Torvalds, `https://lwn.net/Articles/671657/`)

System (Call) Request Techniques

Kind	Technique	Examples	$\#\frac{\text{traps}}{\text{sys requests}}$
Synchronous	Blocking	e.g. `read()`/`write()`	1
Synchronous	Non-Blocking	SOCK_NONBLOCK w select/epoll	≥ 1
Asynchronous	Callback	Linux/POSIX AIO	1
Asynchronous	Queue-based	`io_uring`	$[0, 1]$

- Soares et al. “FlexSC: Flexible System Call Scheduling with Exception-Less System Calls” (OSDI’10)
- `io_uring` introduced in Linux 5.1 (2019)
- Candidate to become **the** generic asynchronous system-call interface

io_uring Architecture

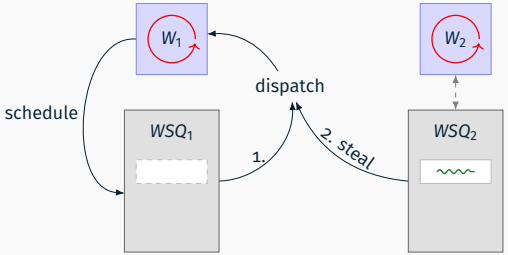


How can Concurrency Platforms exploit this asynchronous system call interface?

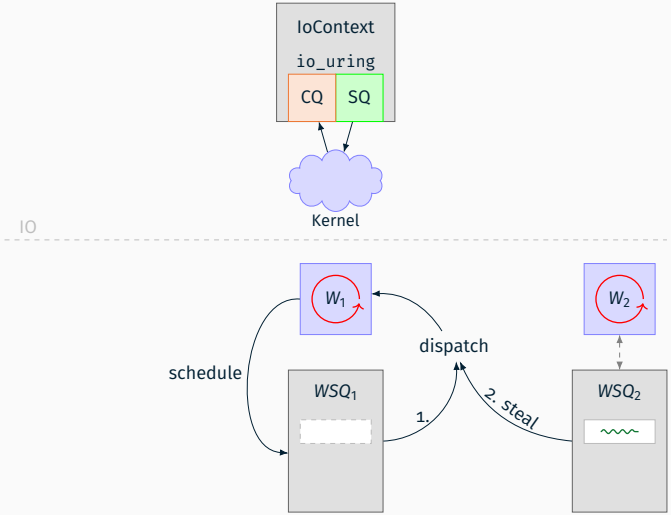
How can Concurrency Platforms exploit this asynchronous system call interface?

Presenting a case study based on our research Concurrency Platform **“EMPER”**

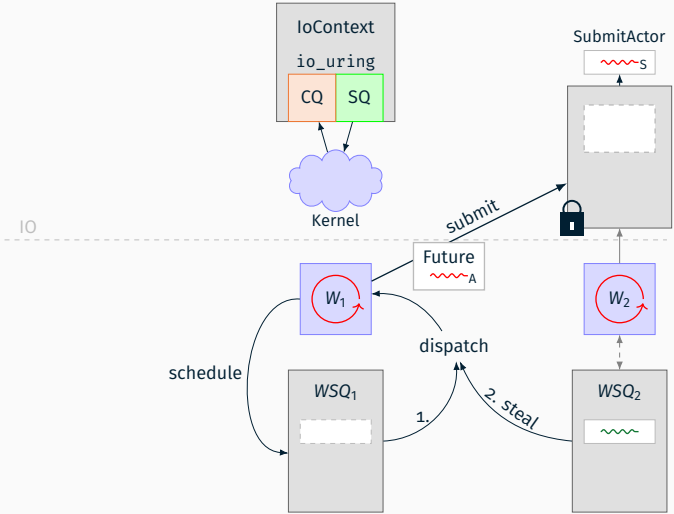
First Architecture of pseudo-synchronous System Calls



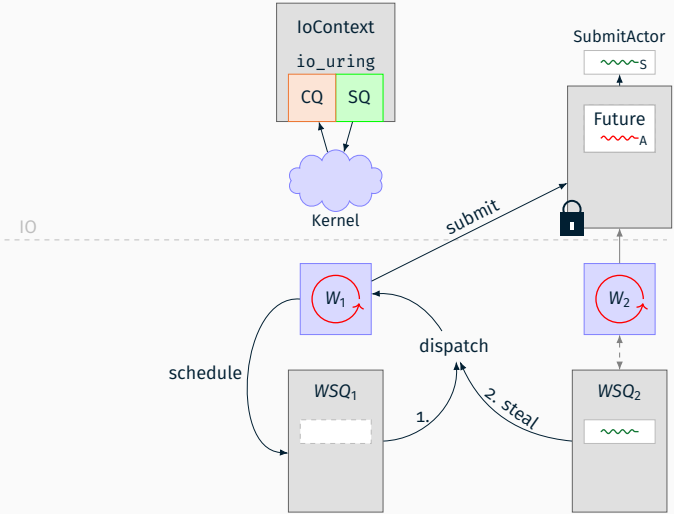
First Architecture of pseudo-synchronous System Calls



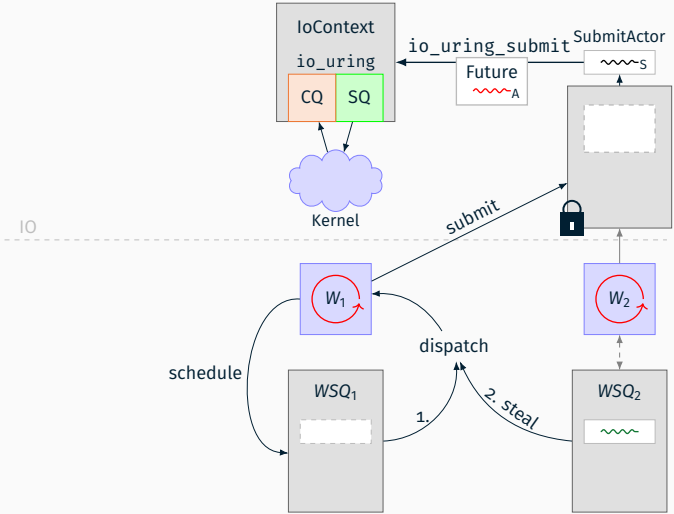
First Architecture of pseudo-synchronous System Calls



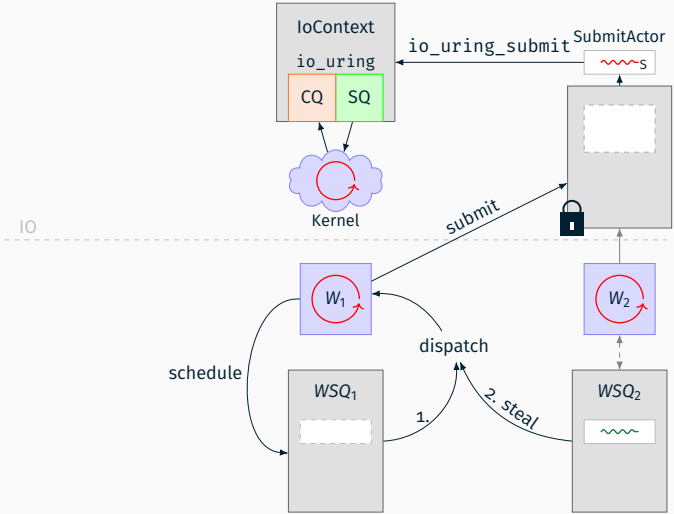
First Architecture of pseudo-synchronous System Calls



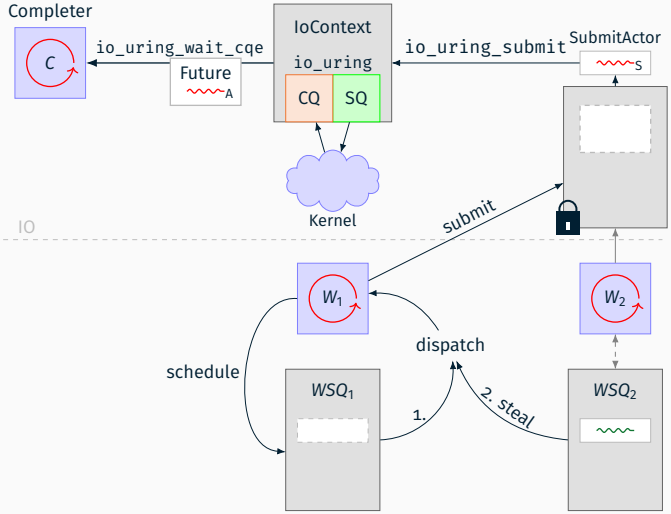
First Architecture of pseudo-synchronous System Calls



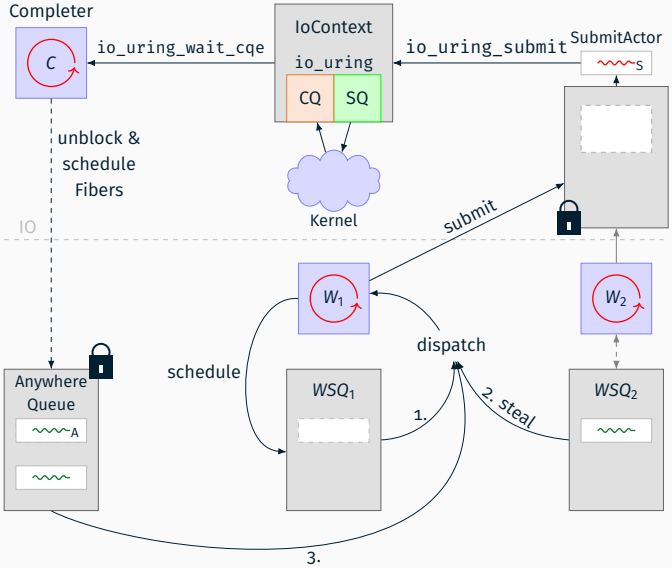
First Architecture of pseudo-synchronous System Calls



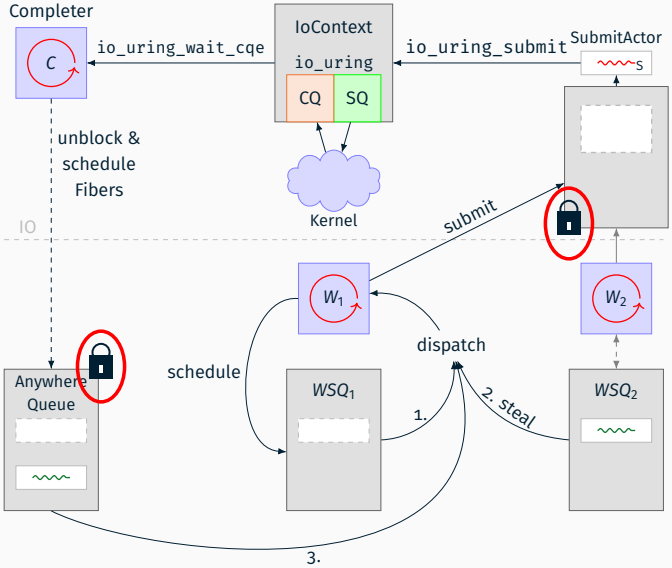
First Architecture of pseudo-synchronous System Calls



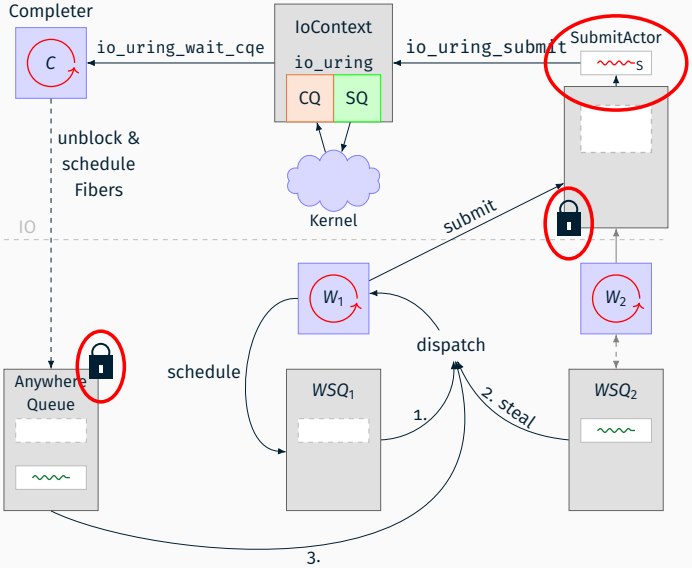
First Architecture of pseudo-synchronous System Calls



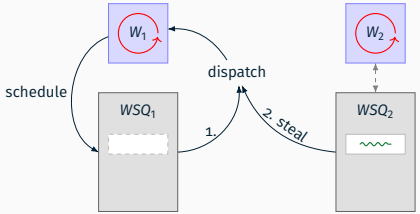
First Architecture of pseudo-synchronous System Calls



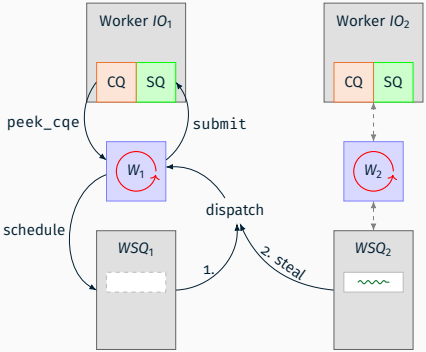
First Architecture of pseudo-synchronous System Calls



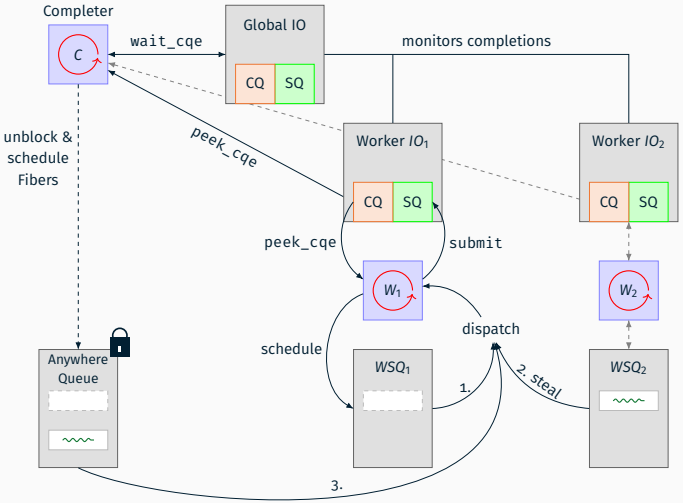
Final Architecture



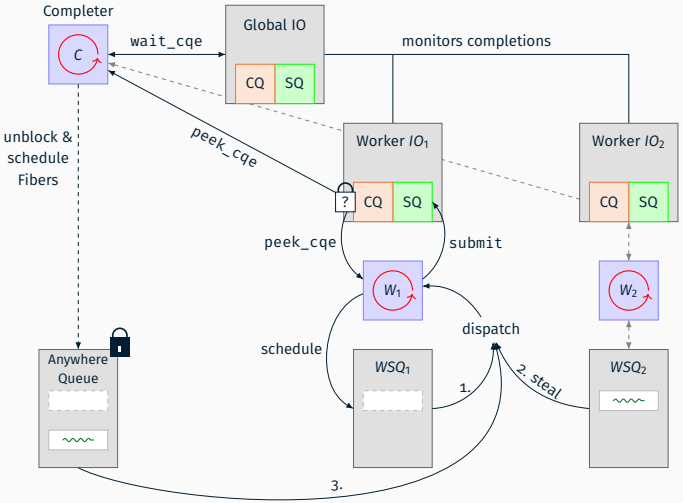
Final Architecture



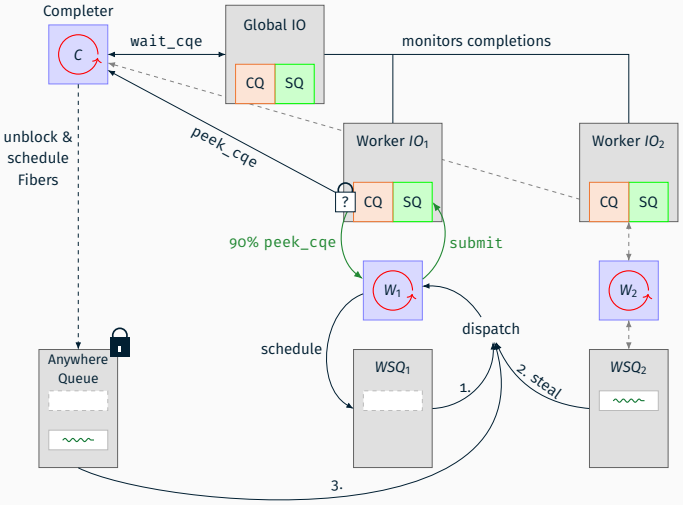
Final Architecture



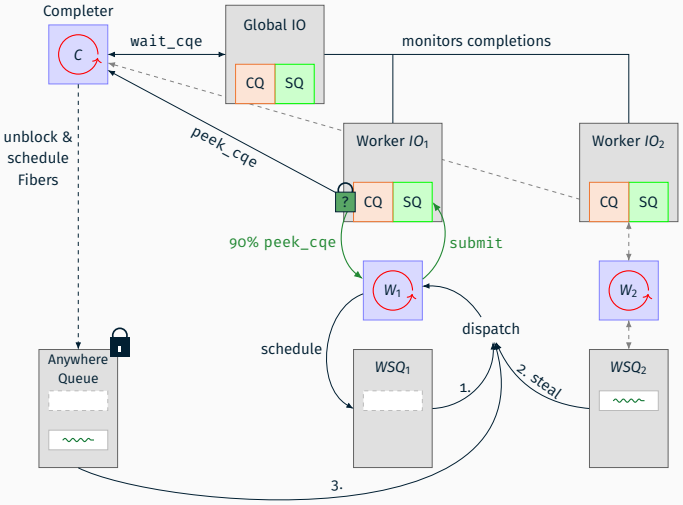
Final Architecture



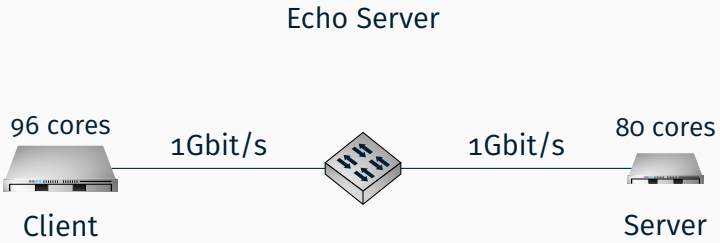
Final Architecture



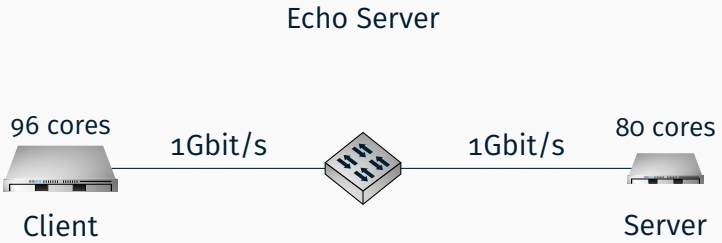
Final Architecture



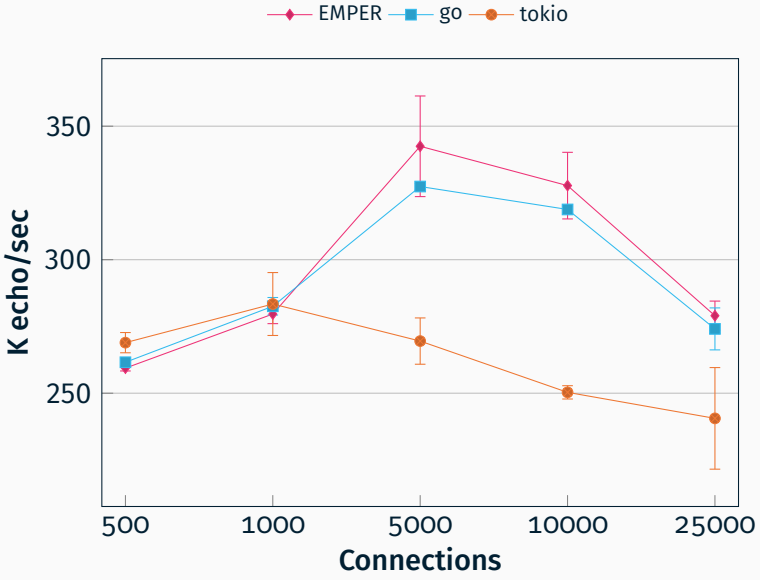
Evaluation



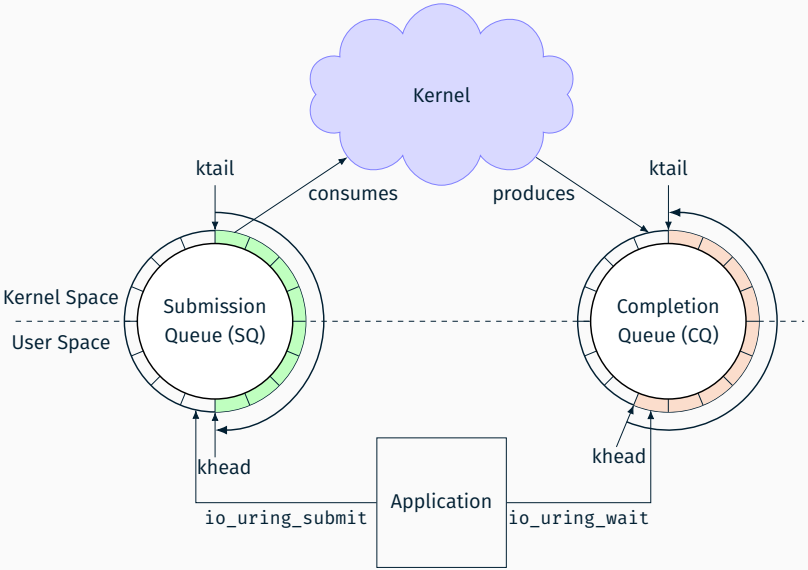
Evaluation



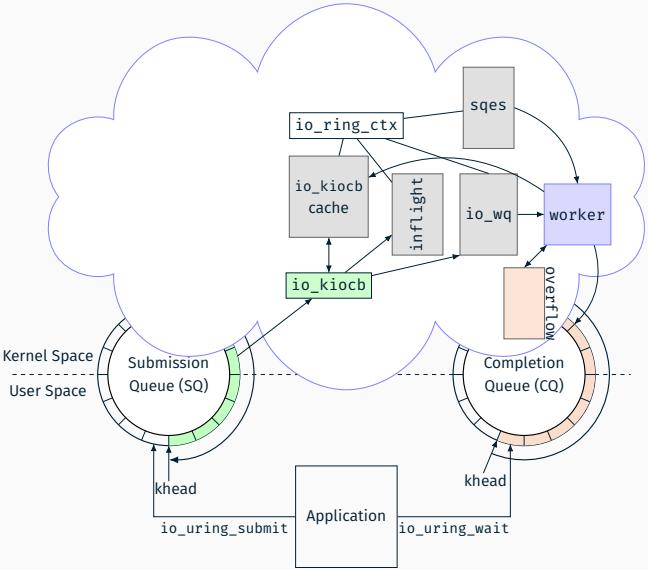
- 500 – 25.000 connections
- Linux 5.11-rc6


EMPER
go
tokio
K echo/sec
350
300
250
500
1000
5000
10000
25000
Connections

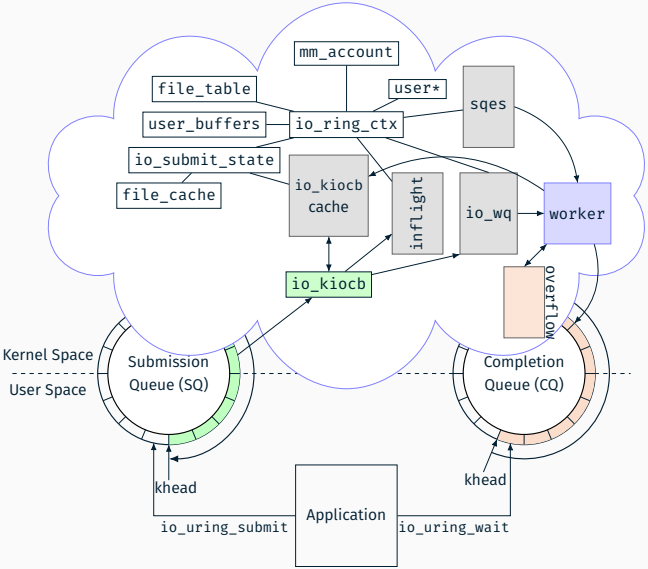
But what actually happens in the kernel?



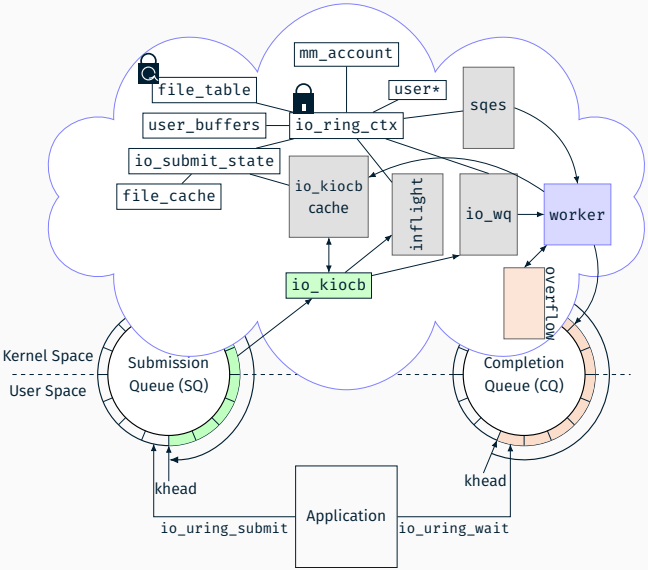
But what actually happens in the kernel?



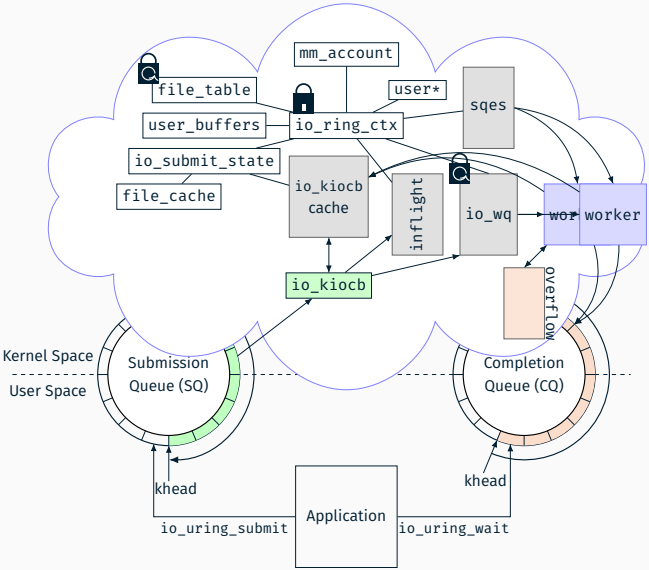
But what actually happens in the kernel?



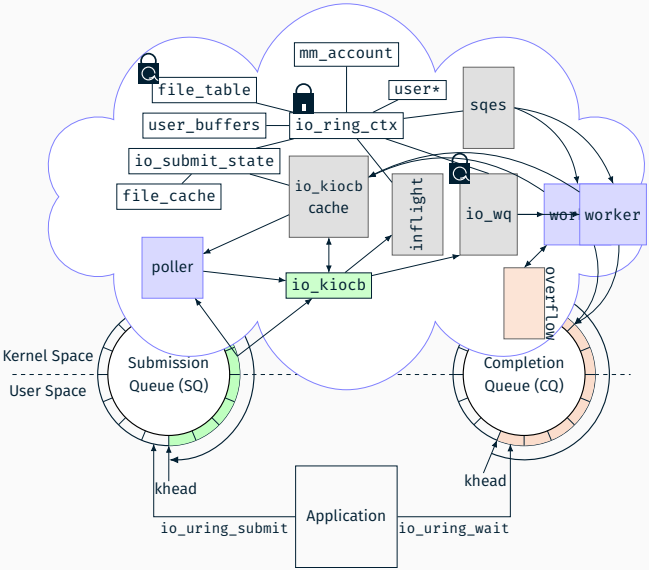
But what actually happens in the kernel?



But what actually happens in the kernel?



But what actually happens in the kernel?



Future work

Are we building a concurrency platform in the kernel?
If yes, why not entangle both sides?

Benefits of a Concurrency Platform aware kernel

- Architectural awareness (e.g. NUMA)
- Kernel knows best how to fully use available hardware resources
- Block user-space Fibers on any trap (e.g. major page fault)

Conclusion

Fruitfully fusion of Concurrency Platforms and queue-based asynchronous system-call interfaces

- Reduce off-CPU time of application
- Again, locality is key to efficiency
- One ring (interface) to rule them all
- Many more `io_uring` techniques to exploit: `SQPOLL`, automatic buffer selection, link SQEs, ...
- Return of the M:N threading model
- Asynchronous system-call techniques are a great opportunity for operating-system research

Free Software

- https://gitlab.cs.fau.de/i4/manycore/emper
- https://gitlab.cs.fau.de/i4/manycore/emper-io-eval
- https://github.com/axboe/liburing

Free Software

- https://gitlab.cs.fau.de/i4/manycore/emper
- https://gitlab.cs.fau.de/i4/manycore/emper-io-eval
- https://github.com/axboe/liburing

Thank you for your attention!

EMPER Echoserver

```
auto main(int argc, char* argv[]) -> int {
  Runtime runtime;
  auto* listener = emper::io::tcp_listener(host, port, [](int socket) {
    char buf[1024];
    for (;;) {
      ssize_t bytes_recv = emper::io::recvAndWait(socket, buf,
                                                  sizeof(buf), 0);
      if (bytes_recv <= 0) {
      finish:
        emper::io::closeAndForget(socket); return;
      }

      ssize_t bytes_send = emper::io::sendAndWait(socket, buf,
        bytes_recv, MSG_NOSIGNAL, true);
      if (bytes_recv != bytes_send) { goto finish; }
    }
  });
  runtime.scheduleFromAnywhere(*listener);
  runtime.waitUntilFinished();
}
```