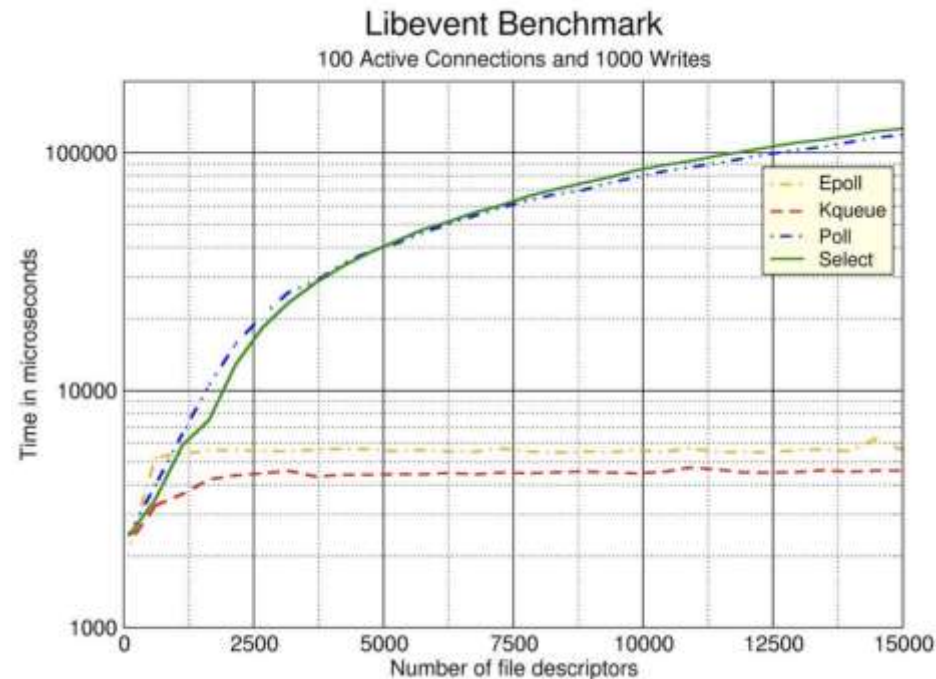


# **COPS: A coroutine-based priority scheduling framework perceived by the operating system**

Fangliang Zhao, Donghai Liao, Jingbang Wu  
Huimei Lu, Yong Xiang

# Concurrency matters

- “Good” concurrency model achieves **high throughput, low latency, and efficient resource utilization**.
- Fine-tailored model can improve performance by an order of magnitude or more.



# What are the problems with existing multi-threading concurrency model?

- **Nondeterministic, resulting in uncertain access order of shared resources.**
  - Synchronization
  - Atomics
  - ...

# What are the problems with existing multi-threading concurrency model?

- **Incompatible with asynchronous I/O mechanism**
  - Additional mechanisms(producer-consumer model, zero-copy, etc.)
  - Complicated asynchronous runtime
  - Extra syscall interfaces
  - Hard to code(manually implement event-loop or callback hell)
  - ...

# I/O Multiplexing

- **Select, Poll, Epoll**

- Need complicated data structure to maintain global I/O states.

- Fd\_set
    - Fd\_queue
    - Red-black tree + ready queue

- Expose extra syscall.

- select
    - poll
    - epoll\_create, epoll\_ctl, epoll\_wait,

- **IOCP**

- Callback hell.

# Userland task

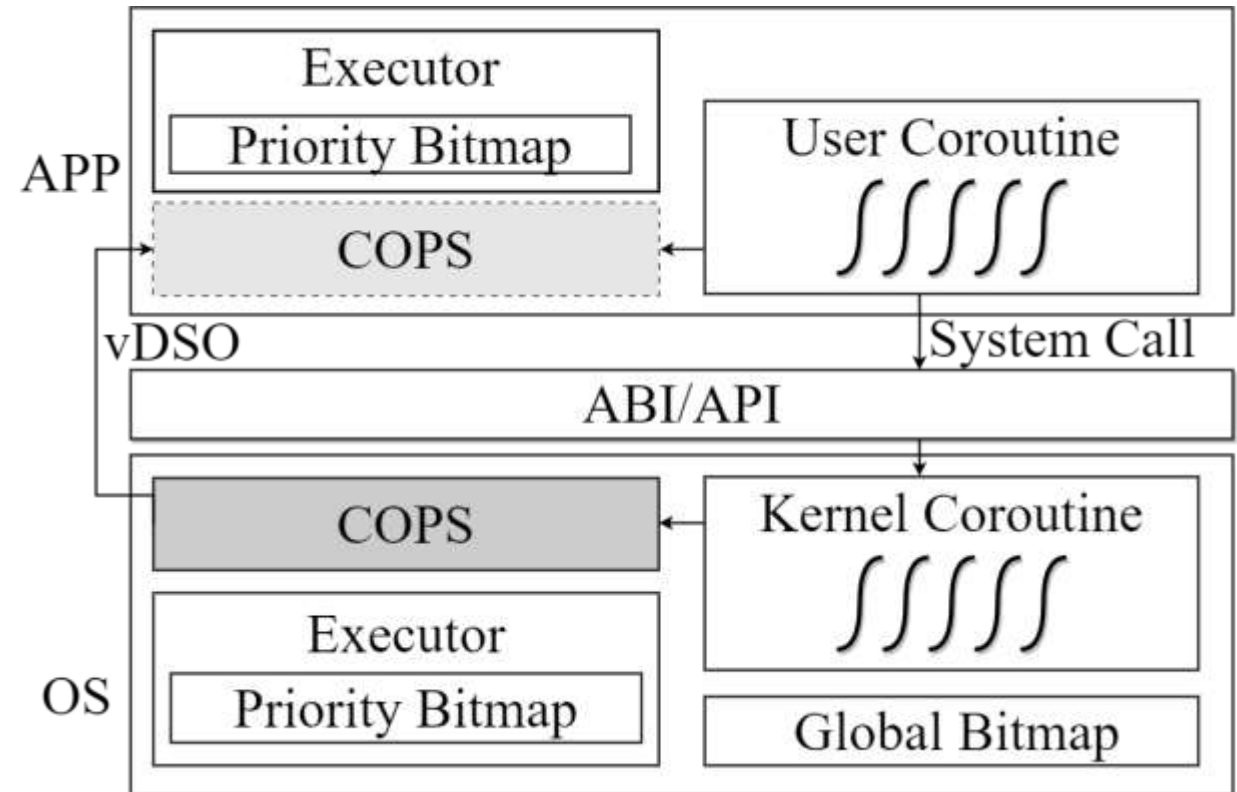
- **POSIX AIO**
  - Without userland scheduling, os lacks awareness.
  - User thread management.
  - Frequent context switching across privilege levels.

# Solution: COPS

- **Benefits from coroutines**
  - Low cost of task switching.
  - Low resource usage.
  - Language facilities make program easy.
- **Treats coroutines as first-class citizens within OS**
  - Introduces coroutines from user space into kernel.
  - Employs coroutine as the fundamental task unit to replace thread(decrease kernel complexity).
  - Offers a unified priority-based scheduling framework across kernel and user space(make kernel aware of userland task).
  - Supports asynchronous syscall(without extra syscall).

# COPS Overview

- Separate executor for task management.
- Shared scheduling framework via vDSO.
- Priority bitmap for cooperative scheduling.
- Obtain asynchronous I/O service through syscall.

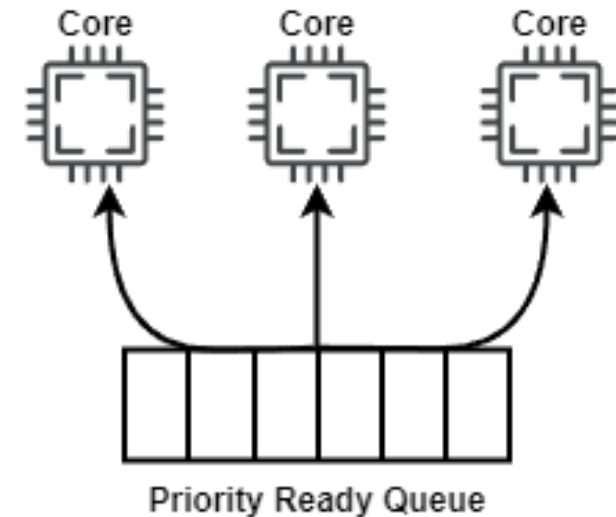
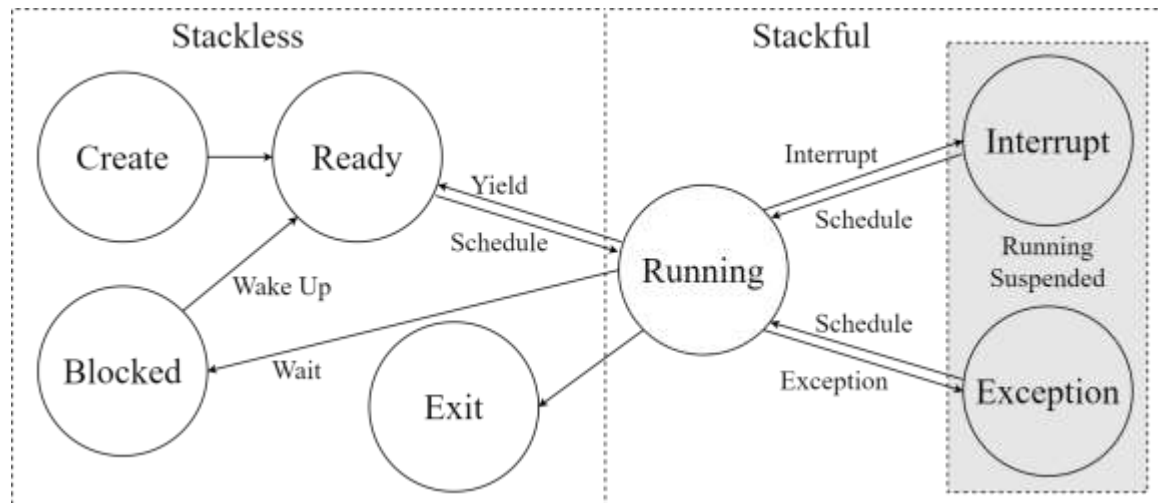




# Coroutine Management

- **Coroutine Control Block**
- **Executor:**
  - Priority ready queues
  - blocking set
- **Coroutine state**

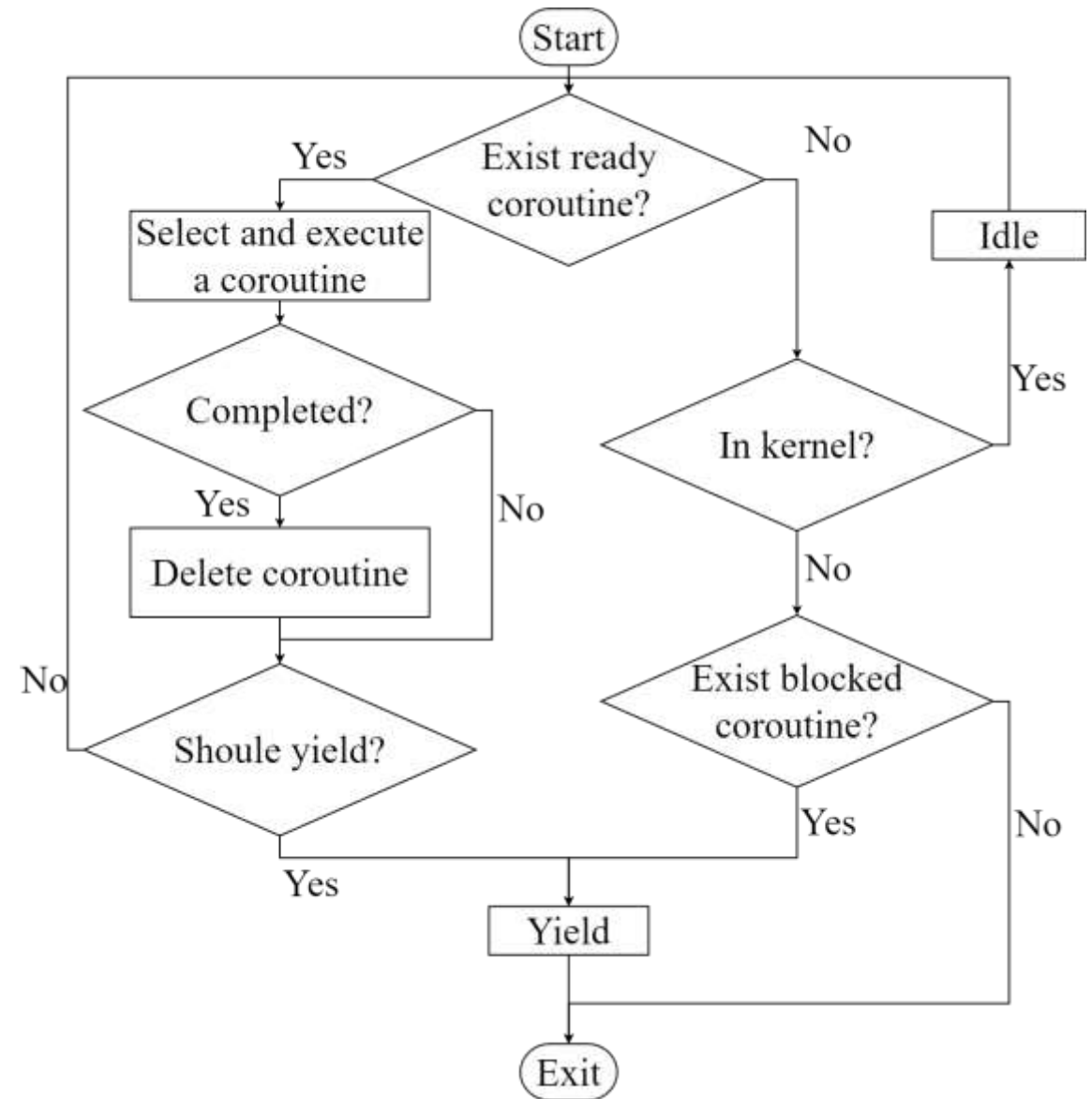
```
pub struct Coroutine{  
    pub cid: CoroutineId,  
    pub kind: CoroutineKind,  
    pub priority: usize,  
    pub future: Pin<Box<dyn Future<Output=()> + 'static + Send + Sync>>,  
    pub waker: Arc<Waker>,  
}
```



# COPS API & Logic

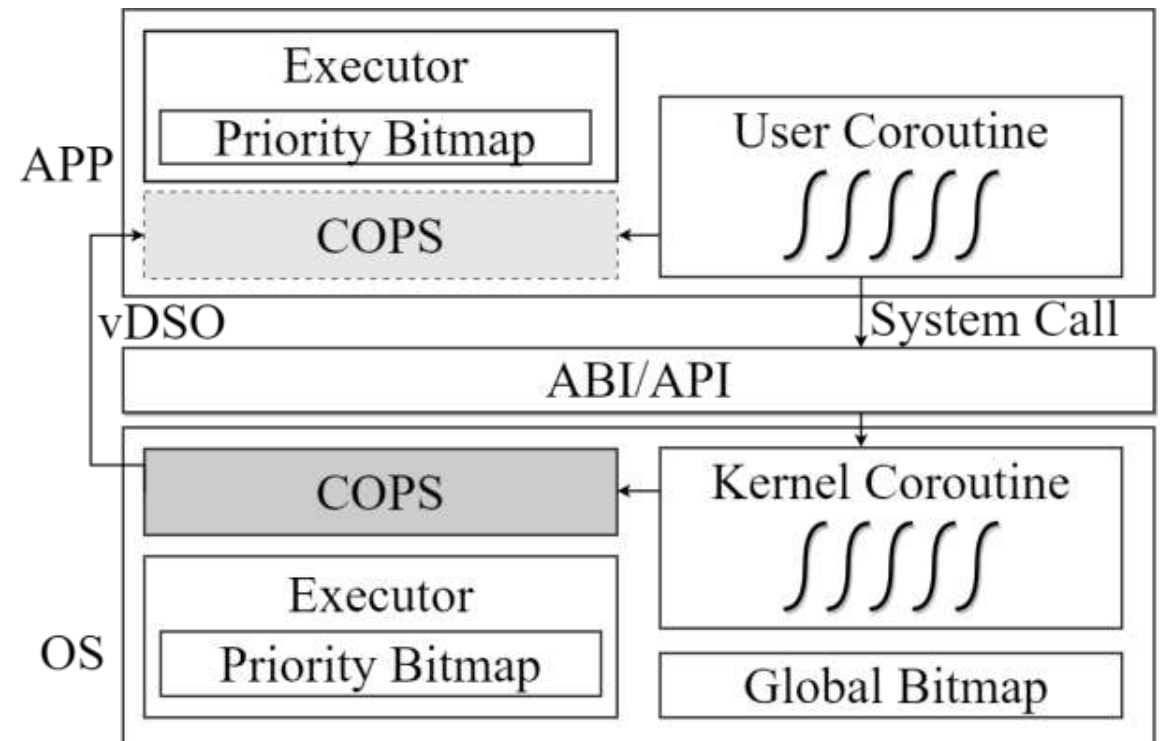
- **Just define task inner logic without concerning about task execution order.**

Function	Return value
spawn(future, priority)	cid
current()	cid
wake_up(cid)	
set_priority(cid, priority)	
alloc_cpu(cpu num)	



# COPS's Global Cooperative Scheduling

- Separative priority bitmap in executor for local cooperative scheduling.
- Global priority bitmap for global cooperative scheduling between kernel and user processes.
  1. Timer interrupt.
  2. Kernel scan local bitmaps.
  3. Kernel update global bitmap.
  4. Select the highest coroutine in all kernel and user processes.



# Usage Patterns of COPS

- Concurrent Programming
- Asynchronous Programming

---

**Algorithm 2** Asynchronous Programming

---

```
1: function MAIN
2:   spawn(||async_read, 0);
3: end function
4: function ASYNC_READ
5:   buf  $\leftarrow$  [0; buf_len];
6:   current_cid  $\leftarrow$  current();
7:   read!(buf.ptr, buf.len, current_cid);
8: end function
```

---

---

**Algorithm 1** Concurrent Programming

---

```
1: MSG_QUEUE;
2: function MAIN
3:   consumer_cid  $\leftarrow$  spawn(||consumer, 1);
4:   spawn(||producer(consumer_cid), 0);
5: end function
6: function CONSUMER
7:   loop
8:     while MSG_QUEUE is empty do
9:       blocked;
10:    end while
11:    ...  $\triangleright$  consume data from MSG_QUEUE
12:  end loop
13: end function
14: function PRODUCER(cid)
15:   loop
16:     while MSG_QUEUE is full do
17:       wake_up(cid);  $\triangleright$  wake up the consumer
18:       yield;
19:     end while
20:     ...  $\triangleright$  produce data into MSG_QUEUE
21:   end loop
22: end function
```

---

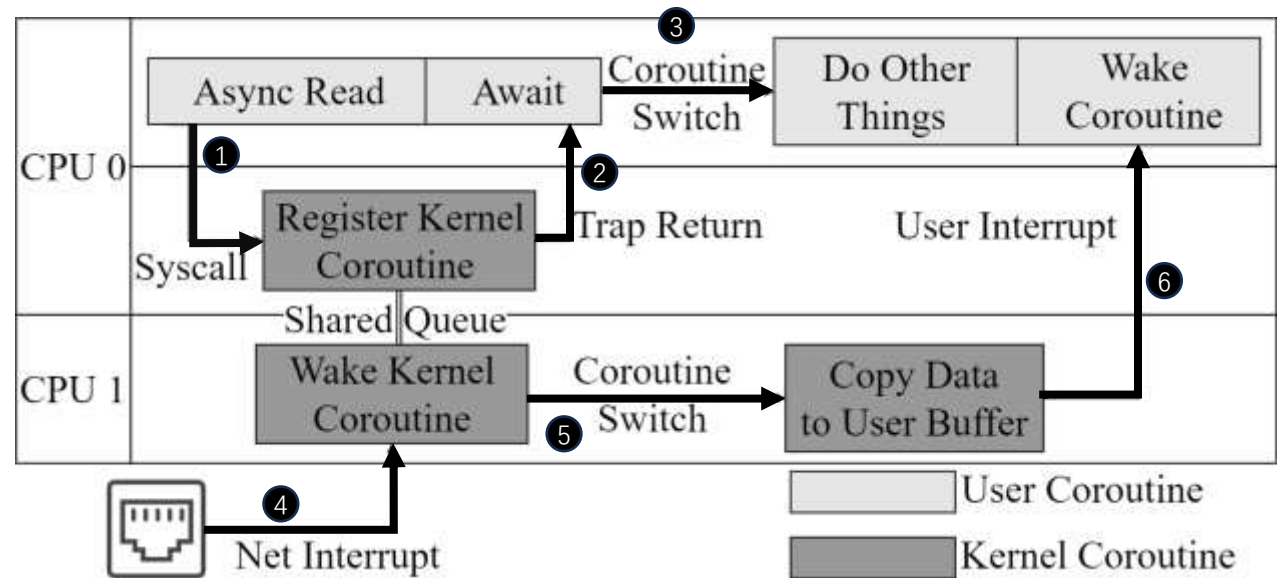
# Asynchronous Syscall

## CPU0

1. App Issue asynchronous I/O request
2. Back to user space
3. Switch to do other things

## CPU1

4. Receive net interrupt
5. Execute kernel coroutine
6. Wake up blocked task on CPU0



# Evaluation Questions

1. Can COPS outperform than multi-threading model?
  1. Throughput
  2. Latency
  3. Memory Usage
2. Are high priority requests handled first?

# Evaluation Setup

- Testbed

Network Stack	smoltcp	
Operating System	rCore-tutorial	
FPGA	RISC-V soft IP core	rocket-chip with N extension, 4 Core, 100MHz
	Ethernet IP core	Xilinx AXI 1G/2.5G Ethernet Subsystem (1Gbps)
	Zynq UltraScale+ XCZU15EG MPSoC	

- Web Server built with three component(using Multi-threading / Coroutine)
  - Request receiver
  - Request handler
  - Response Sender

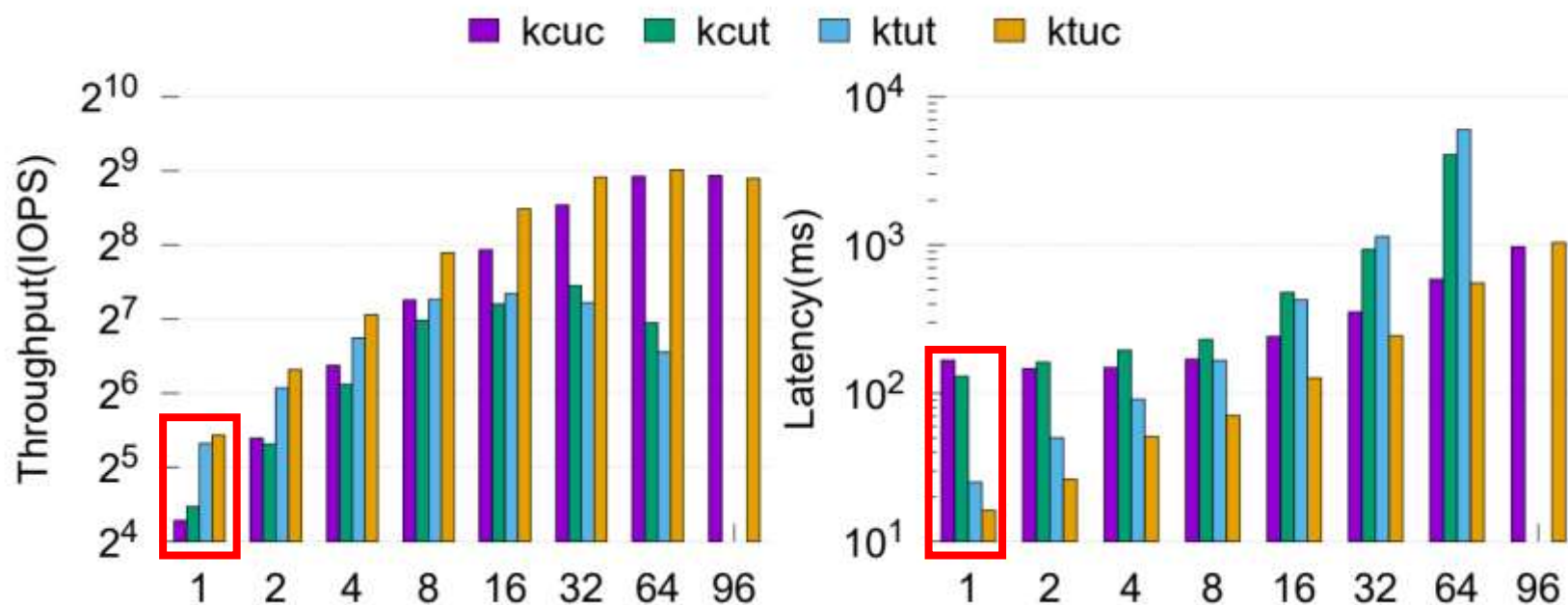
# Evaluation: Thread vs. Coroutine

- Server: Kernel(**K**), User Process(**U**), Thread(**T**), Coroutine(**C**)
  - KCUC
  - KCUT
  - KTUT
  - KTUC: similar to select, poll, epoll
- Client
  - Package: 15\*15 matrix payload
  - Interval: 100ms
  - 5 seconds



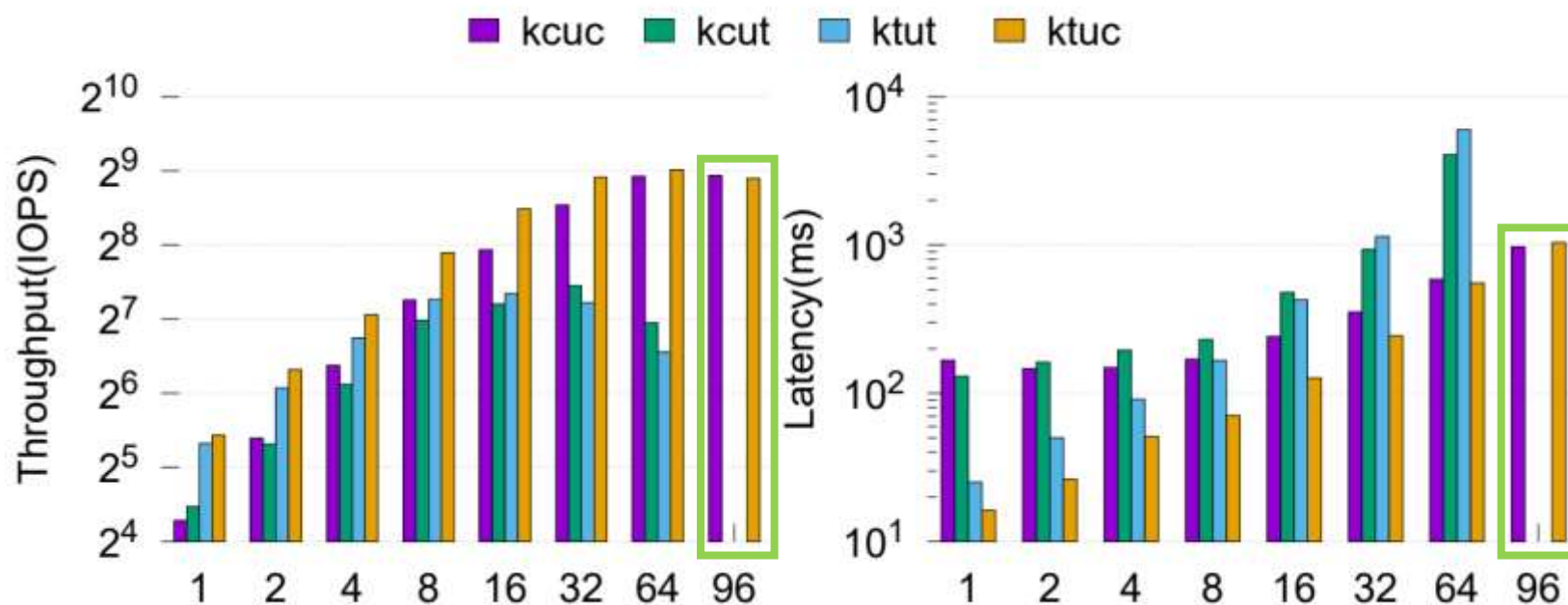
# Evaluation: Coroutine vs. Thread

- Runtime overhead
  - Ready Queue across multiply cores is being lock-protected.
  - Extra core for light workload(as same core allocation as thread model).



# Evaluation: Coroutine vs. Thread

- Suitable for heavy workload(KCUC vs. KTUC)
  - Higher throughput
  - Lower latency



# Evaluation: Coroutine vs. Thread

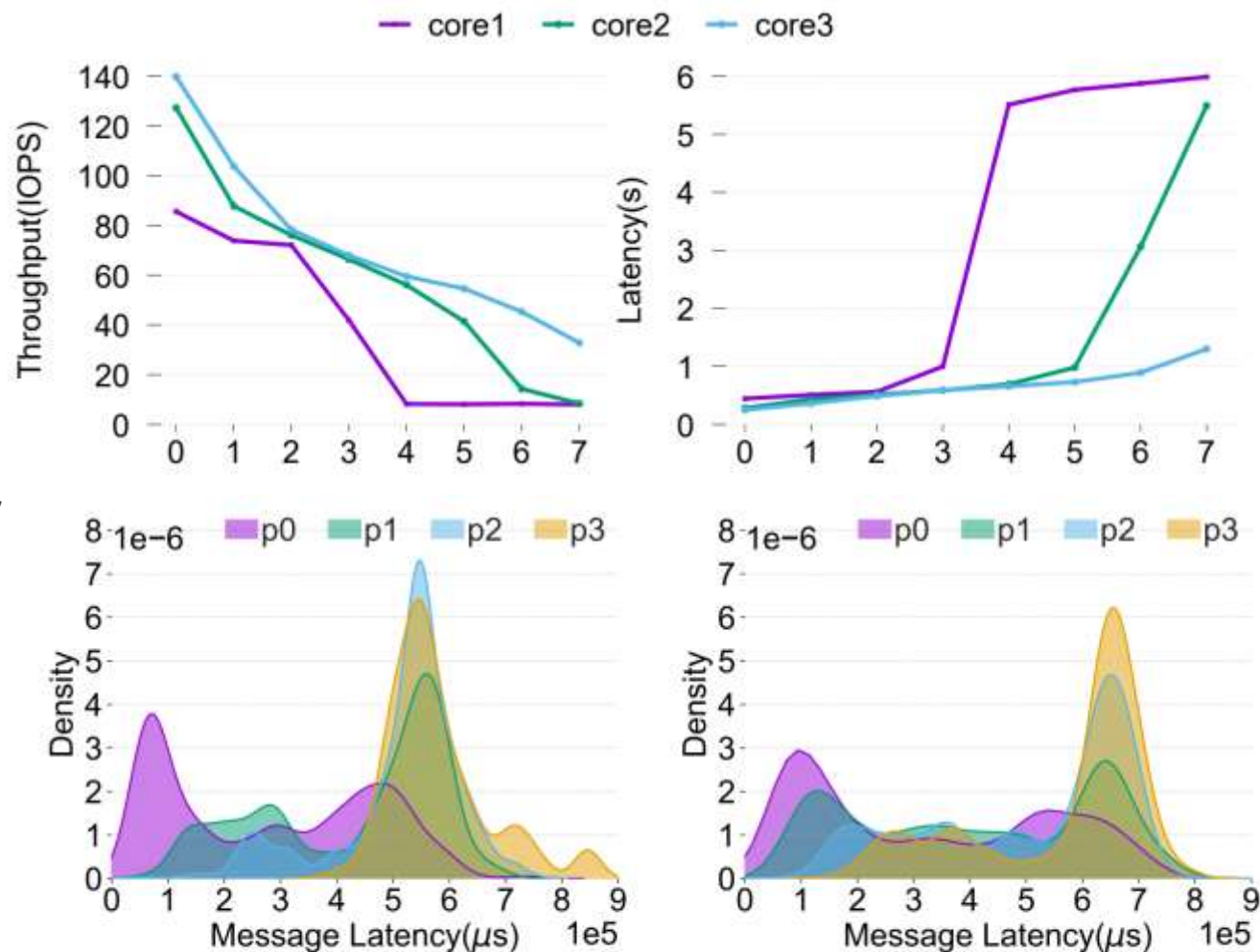
- Less memory usage(establish more connections at the same config)
  - Thread: each component need a kernel stack(0x4000 bytes) and a user stack(0x4000 bytes).
  - Coroutine:
    - receiver(120 bytes)
    - handler(64 bytes)
    - sender(80 bytes)
    - kernel syscall coroutine(176 bytes)

Memory Usage(bytes)	
KTUT	$0x4000 * 2 * 3$
KCUC	$0x4000 * 2 + 176 + 120 + 80 + 64$

Configuration	Size(bytes)	KCUC	KTUT
Kernel heap	0x80_0000	385	186
Kernel frame	0x1A0_0000		
User heap	0x20_0000		

# Evaluation: Priority Orientation

- Setup
  - 64 connections across 8 priority
  - Interval: 50ms
  - 5 seconds
- Higher priority ->
  - Higher throughput + lower latency
- More resources ->
  - Lower priority connection is being improved.



(a) Core-2

(b) Core-4

# Conclusion

COPS improve concurrency by:

- Treating coroutines as the first-class citizens within OS to benefit from coroutines.
- Employing priority scheduling to provide cooperation and efficient resource utilization.